

9: Aggregating Pandas DataFrames



Aggregating Pandas DataFrames

The following topics will be covered in this lesson:

- Performing database-style operations on DataFrames
- Using DataFrame operations to enrich data
- Aggregating data
- Working with time series data

lesson materials

- Throughout this lesson, we will use a variety of datasets, which can be found in the data/ directory:

File	Description	Source
<code>dirty_data.csv</code>	Dirty weather data from the <i>Handling duplicate, missing, or invalid data</i> section in Chapter 3, Data Wrangling with Pandas	Adapted from the NCEI API's GHCND dataset
<code>fb_2018.csv</code>	Facebook stock's opening, high, low, and closing price daily, along with volume traded for 2018.	The <code>stock_analysis</code> package (see Chapter 7, Financial Analysis – Bitcoin and the Stock Market).
<code>fb_week_of_may_20_per_minute.csv</code>	Facebook stock's opening, high, low, and closing price per minute, along with volume traded for May 20, 2019 through May 24, 2019.	Nasdaq
<code>melted_stock_data.csv</code>	The contents of <code>fb_week_of_may_20_per_minute.csv</code> melted into a single column for the price and another for the timestamp.	Adapted from Nasdaq
<code>nyc_weather_2018.csv</code>	Long format weather data for New York City across various stations.	The NCEI API's GHCND dataset.
<code>stocks.db</code>	The <code>fb_prices</code> and <code>aapl_prices</code> tables contain the stock prices for Facebook and Apple, respectively, for May 20, 2019 through May 24, 2019. Facebook is at a minute granularity, whereas Apple has timestamps that include (fictitious) seconds.	Adapted from Nasdaq
<code>weather_by_station.csv</code>	Long format weather data for New York City across various stations, along with station information.	The NCEI API's GHCND dataset and the <code>stations</code> endpoint.
<code>weather_stations.csv</code>	Information on all the stations providing weather data for New York City.	The NCEI API's <code>stations</code> endpoint.
<code>weather.db</code>	The <code>weather</code> table contains New York City weather data, while the <code>stations</code> table contains information on the stations.	The NCEI API's GHCND dataset and the <code>stations</code> endpoint.

Performing database-style operations on DataFrames

- For this section, we will be working in the 1-querying_and_merging.ipynb notebook.
- We will begin with our imports and read in the NYC weather data CSV file:

```
>>> import pandas as pd  
>>> weather = pd.read_csv('data/nyc_weather_2018.csv')  
>>> weather.head()
```

Performing database-style operations on DataFrames

- This is long format data—we have several different weather observations per day for various stations covering NYC in 2018:

	date	datatype	station	attributes	value
0	2018-01-01T00:00:00	PRCP	GHCND:US1CTFR0039	, „N,	0.0
1	2018-01-01T00:00:00	PRCP	GHCND:US1NJBG0015	, „N,	0.0
2	2018-01-01T00:00:00	SNOW	GHCND:US1NJBG0015	, „N,	0.0
3	2018-01-01T00:00:00	PRCP	GHCND:US1NJBG0017	, „N,	0.0
4	2018-01-01T00:00:00	SNOW	GHCND:US1NJBG0017	, „N,	0.0

Querying DataFrames

- To illustrate this, let's query the weather data for all the rows where the value of the SNOW column was greater than zero for stations with US1NY in their station ID:

```
>>> snow_data = weather.query(  
...     'datatype == "SNOW" and value > 0'  
...     'and station.str.contains("US1NY")'  
... )  
>>> snow_data.head()
```

Querying DataFrames

- Each row is a snow observation for a given combination of date and station.
- Notice that the values vary quite a bit for January 4th—some stations received more snow than others:

		date	datatype	station	attributes	value
114		2018-01-01T00:00:00	SNOW	GHCND:US1NYWC0019	,,N,	25.0
789		2018-01-04T00:00:00	SNOW	GHCND:US1NYNS0007	,,N,	41.0
794		2018-01-04T00:00:00	SNOW	GHCND:US1NYNS0018	,,N,	10.0
798		2018-01-04T00:00:00	SNOW	GHCND:US1NYNS0024	,,N,	89.0
800		2018-01-04T00:00:00	SNOW	GHCND:US1NYNS0030	,,N,	102.0

Querying DataFrames

- This query is equivalent to the following in SQL.
- Note that SELECT * selects all the columns in the table (our dataframe, in this case):

```
SELECT * FROM weather
WHERE
    datatype == "SNOW" AND value > 0 AND station LIKE
    "%US1NY%";
```

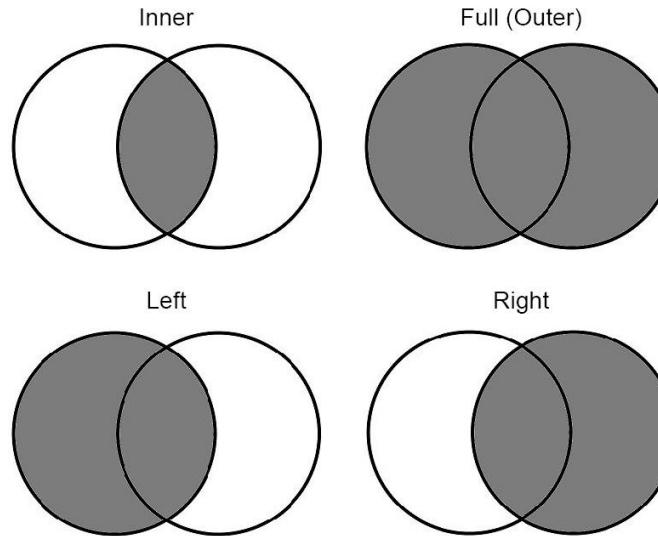
Querying DataFrames

- In Working with Pandas DataFrames, we learned how to use a Boolean mask to get the same result:

```
>>> weather[  
...     (weather.datatype == 'SNOW') & (weather.value > 0)  
...     & weather.station.str.contains('US1NY')  
... ].equals(snow_data)  
True
```

Merging DataFrames

- Here, the darker regions represent the data we are left with after performing the join:



Merging DataFrames

- The NCEI API's stations endpoint gives us all the information we need for the stations.
- This is in the `weather_stations.csv` file, as well as in the stations table in the SQLite database.
- Let's read this data into a dataframe:

```
>>> station_info = pd.read_csv('data/weather_stations.csv')
>>> station_info.head()
```

Merging DataFrames

- The ones in New Jersey are southwest of NYC, while the ones in Connecticut are northeast of NYC:

	id	name	latitude	longitude	elevation
0	GHCND:US1CTFR0022	STAMFORD 2.6 SSW, CT US	41.064100	-73.577000	36.6
1	GHCND:US1CTFR0039	STAMFORD 4.2 S, CT US	41.037788	-73.568176	6.4
2	GHCND:US1NJBG0001	BERGENFIELD 0.3 SW, NJ US	40.921298	-74.001983	20.1
3	GHCND:US1NJBG0002	SADDLE BROOK TWP 0.6 E, NJ US	40.902694	-74.083358	16.8
4	GHCND:US1NJBG0003	TENAFLY 1.3 W, NJ US	40.914670	-73.977500	21.6

Merging DataFrames

- Before we join the data, let's get some information on how many distinct stations we have and how many entries are in each dataframe:

```
>>> station_info.id.describe()
count                279
unique               279
top      GHCND:US1NJBG0029
freq                  1
Name: id, dtype: object
>>> weather.station.describe()
count                78780
unique                 110
top      GHCND:USW00094789
freq                  4270
Name: station, dtype: object
```

Merging DataFrames

- To select the rows, we just grab the value at index 0 (1 for columns):

```
>>> station_info.shape[0], weather.shape[0] # 0=rows, 1=cols  
(279, 78780)
```

Merging DataFrames

- Since we will be checking the row count often, it makes more sense to write a function that will give us the row count for any number of dataframes.
- The `*dfs` argument collects all the input to this function in a tuple, which we can iterate over in a list comprehension to get the row count:

```
>>> def get_row_count(*dfs):
...     return [df.shape[0] for df in dfs]
>>> get_row_count(station_info, weather)
[279, 78780]
```

Merging DataFrames

- The left dataframe is the one we call merge() on, while the right one is the dataframe that gets passed in as an argument:

```
>>> inner_join = weather.merge(  
...     station_info, left_on='station', right_on='id'  
... )  
>>> inner_join.sample(5, random_state=0)
```

Merging DataFrames

- This operation also kept both the station and id columns, which are identical:

		date	datatype	station	attributes	value		id	name	latitude	longitude	elevation
10739		2018-08-07T00:00:00	SNOW	GHCND:US1NJMN0069	,,N,	0.0	GHCND:US1NJMN0069	LONG BRANCH 1.7 SSW, NJ US	40.275368	-74.006027	9.4	
45188		2018-12-21T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	16.7	GHCND:USW00014732	LAGUARDIA AIRPORT, NY US	40.779440	-73.880350	3.4	
59823		2018-01-15T00:00:00	WDF5	GHCND:USW00094741	,,W,	40.0	GHCND:USW00094741	TETERBORO AIRPORT, NJ US	40.850000	-74.061390	2.7	
10852		2018-10-31T00:00:00	PRCP	GHCND:US1NJMN0069	T,,N,	0.0	GHCND:US1NJMN0069	LONG BRANCH 1.7 SSW, NJ US	40.275368	-74.006027	9.4	
46755		2018-05-05T00:00:00	SNOW	GHCND:USW00014734	,,W,	0.0	GHCND:USW00014734	NEWARK LIBERTY INTERNATIONAL AIRPORT, NJ US	40.682500	-74.169400	2.1	

Merging DataFrames

- In order to remove the duplicate information in the station and id columns, we can rename one of them before the join.
- Consequently, we will only have to supply a value for the on parameter because the columns will share the same name:

```
>>> weather.merge(  
...     station_info.rename(dict(id='station'), axis=1),  
...     on='station'  
... ).sample(5, random_state=0)
```

Merging DataFrames

- Since the columns shared the name, we only get one back after joining on them:

		date	datatype	station	attributes	value	name	latitude	longitude	elevation
10739		2018-08-07T00:00:00	SNOW	GHCND:US1NJMN0069	,,N,	0.0	LONG BRANCH 1.7 SSW, NJ US	40.275368	-74.006027	9.4
45188		2018-12-21T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	16.7	LAGUARDIA AIRPORT, NY US	40.779440	-73.880350	3.4
59823		2018-01-15T00:00:00	WDF5	GHCND:USW00094741	,,W,	40.0	TERBOROUGH AIRPORT, NJ US	40.850000	-74.061390	2.7
10852		2018-10-31T00:00:00	PRCP	GHCND:US1NJMN0069	T,,N,	0.0	LONG BRANCH 1.7 SSW, NJ US	40.275368	-74.006027	9.4
46755		2018-05-05T00:00:00	SNOW	GHCND:USW00014734	,,W,	0.0	NEWARK LIBERTY INTERNATIONAL AIRPORT, NJ US	40.682500	-74.169400	2.1

Merging DataFrames

- A left join requires us to list the dataframe with the rows that we want to keep (even if they don't exist in the other dataframe) on the left and the other dataframe on the right; a right join is the inverse:

```
>>> left_join = station_info.merge(  
...     weather, left_on='id', right_on='station', how='left'  
... )  
>>> right_join = weather.merge(  
...     station_info, left_on='station', right_on='id',  
...     how='right'  
... )  
>>> right_join[right_join.datatype.isna()].head() # see nulls
```

Merging DataFrames

- Alternatively, our analysis may involve determining the availability of data per station, so getting null values isn't necessarily an issue:

	date	datatype	station	attributes	value		id	name	latitude	longitude	elevation
0	NaN	NaN	NaN	NaN	NaN	GHCND:US1CTFR0022		STAMFORD 2.6 SSW, CT US	41.064100	-73.577000	36.6
344	NaN	NaN	NaN	NaN	NaN	GHCND:US1NJBG0001		BERGENFIELD 0.3 SW, NJ US	40.921298	-74.001983	20.1
345	NaN	NaN	NaN	NaN	NaN	GHCND:US1NJBG0002		SADDLE BROOK TWP 0.6 E, NJ US	40.902694	-74.083358	16.8
718	NaN	NaN	NaN	NaN	NaN	GHCND:US1NJBG0005		WESTWOOD 0.8 ESE, NJ US	40.983041	-74.015858	15.8
719	NaN	NaN	NaN	NaN	NaN	GHCND:US1NJBG0006		RAMSEY 0.6 E, NJ US	41.058611	-74.134068	112.2

Merging DataFrames

- To prove they are equivalent, we need to put the columns in the same order, reset the index, and sort the data:

```
>>> left_join.sort_index(axis=1)\n...     .sort_values(['date', 'station'], ignore_index=True)\n...     .equals(right_join.sort_index(axis=1).sort_values(\n...             ['date', 'station'], ignore_index=True\n...         ))\nTrue
```

Merging DataFrames

- Note that we have additional rows in the left and right joins because we kept all the stations that didn't have weather observations:

```
>>> get_row_count(inner_join, left_join, right_join)  
[78780, 78949, 78949]
```

Merging DataFrames

- We will also pass in indicator=True to add an additional column to the resulting dataframe, which will indicate which dataframe each row came from:

```
>>> outer_join = weather.merge(  
...     station_info[station_info.id.str.contains('US1NY')],  
...     left_on='station', right_on='id',  
...     how='outer', indicator=True  
... )  
# view effect of outer join  
>>> pd.concat([  
...     outer_join.query(f'_merge == "{kind}"')\  
...         .sample(2, random_state=0)  
...     for kind in outer_join._merge.unique()  
... ]).sort_index()
```

Merging DataFrames

- This join keeps all the data and will often introduce null values, unlike inner joins, which won't:

		date	datatype	station	attributes	value		id	name	latitude	longitude	elevation	_merge
23634		2018-04-12T00:00:00	PRCP	GHCND:US1NYNS0043	,,N,	0.0	GHCND:US1NYNS0043	PLAINVIEW 0.4 ENE, NY US	40.785919	-73.466873	56.7	both	
25742		2018-03-25T00:00:00	PRCP	GHCND:US1NYSF0061	,,N,	0.0	GHCND:US1NYSF0061	CENTERPORT 0.9 SW, NY US	40.891689	-73.383133	53.6	both	
60645		2018-04-16T00:00:00	TMIN	GHCND:USW00094741	,,W,	3.9		NaN	NaN	NaN	NaN	NaN	left_only
70764		2018-03-23T00:00:00	SNWD	GHCND:US1NJHD0002	,,N,	203.0		NaN	NaN	NaN	NaN	NaN	left_only
78790		NaN	NaN	NaN	NaN	NaN	GHCND:US1NYQN0033	HOWARD BEACH 0.4 NNW, NY US	40.662099	-73.841345	2.1	right_only	
78800		NaN	NaN	NaN	NaN	NaN	GHCND:US1NYWC0009	NEW ROCHELLE 1.3 S, NY US	40.904000	-73.777000	21.9	right_only	

Merging DataFrames

- The aforementioned joins are equivalent to SQL statements of the following form, where we simply change <JOIN_TYPE> to (INNER) JOIN, LEFT JOIN, RIGHT JOIN, or FULL OUTER JOIN for the appropriate join:

```
SELECT *
FROM left_table
<JOIN_TYPE> right_table
ON left_table.<col> == right_table.<col>;
```

Merging DataFrames

- We will drop the duplicates and the SNWD column (snow depth), which we found to be uninformative since most of the values were infinite (both in the presence and absence of snow):

```
>>> dirty_data = pd.read_csv(  
...     'data/dirty_data.csv', index_col='date'  
... ).drop_duplicates().drop(columns='SNWD')  
>>> dirty_data.head()
```

Merging DataFrames

- Our starting data looks like this:

		station	PRCP	SNOW	TMAX	TMIN	TOBS	WESF	inclement_weather
	date								
2018-01-01T00:00:00		?	0.0	0.0	5505.0	-40.0	NaN	NaN	NaN
2018-01-02T00:00:00	GHCND:USC00280907		0.0	0.0	-8.3	-16.1	-12.2	NaN	False
2018-01-03T00:00:00	GHCND:USC00280907		0.0	0.0	-4.4	-13.9	-13.3	NaN	False
2018-01-04T00:00:00		?	20.6	229.0	5505.0	-40.0	NaN	19.3	True
2018-01-05T00:00:00		?	0.3	NaN	5505.0	-40.0	NaN	NaN	NaN

Merging DataFrames

- Now, we need to create a dataframe for each station.
- To reduce output, we will drop some additional columns:

```
>>> valid_station = dirty_data.query('station != "?")\n...     .drop(columns=['WESF', 'station'])\n>>> station_with_wesf = dirty_data.query('station == "?")\n...     .drop(columns=['station', 'TOBS', 'TMIN', 'TMAX'])
```

Merging DataFrames

- We will perform a left join to make sure we don't lose any rows from our valid station, and, where possible, augment them with the observations from the ? station:

```
>>> valid_station.merge(  
...     station_with_wesf, how='left',  
...     left_index=True, right_index=True  
... ).query('WESF > 0').head()
```

Merging DataFrames

- The versions coming from the left dataframe have the `_x` suffix appended to the column names, and those coming from the right dataframe have `_y` as the suffix:

	PRCP_x	SNOW_x	TMAX	TMIN	TOBS	inclement_weather_x	PRCP_y	SNOW_y	WESF	inclement_weather_y
date										
2018-01-30T00:00:00	0.0	0.0	6.7	-1.7	-0.6	False	1.5	13.0	1.8	True
2018-03-08T00:00:00	48.8	NaN	1.1	-0.6	1.1	False	28.4	NaN	28.7	NaN
2018-03-13T00:00:00	4.1	51.0	5.6	-3.9	0.0	True	3.0	13.0	3.0	True
2018-03-21T00:00:00	0.0	0.0	2.8	-2.8	0.6	False	6.6	114.0	8.6	True
2018-04-02T00:00:00	9.1	127.0	12.8	-1.1	-1.1	True	14.0	152.0	15.2	True

Merging DataFrames

- We can provide our own suffixes with the `suffixes` parameter.
- Let's use a suffix for the ? station only:

```
>>> valid_station.merge(  
...     station_with_wesf, how='left',  
...     left_index=True, right_index=True,  
...     suffixes=("_", '_?')  
... ).query('WESF > 0').head()
```

Merging DataFrames

- However, the right suffix of `_?` was added to the names of the columns that came from the right dataframe:

	PRCP	SNOW	TMAX	TMIN	TOBS	inclement_weather	PRCP_?	SNOW_?	WESF	inclement_weather_?
date										
2018-01-30T00:00:00	0.0	0.0	6.7	-1.7	-0.6	False	1.5	13.0	1.8	True
2018-03-08T00:00:00	48.8	NaN	1.1	-0.6	1.1	False	28.4	NaN	28.7	NaN
2018-03-13T00:00:00	4.1	51.0	5.6	-3.9	0.0	True	3.0	13.0	3.0	True
2018-03-21T00:00:00	0.0	0.0	2.8	-2.8	0.6	False	6.6	114.0	8.6	True
2018-04-02T00:00:00	9.1	127.0	12.8	-1.1	-1.1	True	14.0	152.0	15.2	True

Merging DataFrames

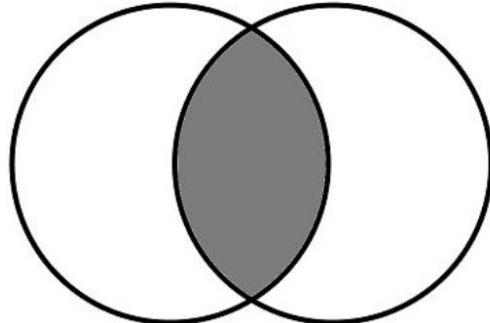
- This yields the same result as the previous example:

```
>>> valid_station.join(  
...     station_with_wesf, how='left', rsuffix='_?')  
... ).query('WESF > 0').head()
```

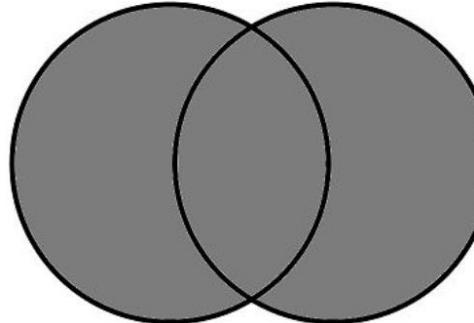
Merging DataFrames

- Remember that the mathematical definition of a set is a collection of distinct objects.
- By definition, the index is a set. Set operations are often explained with Venn diagrams:

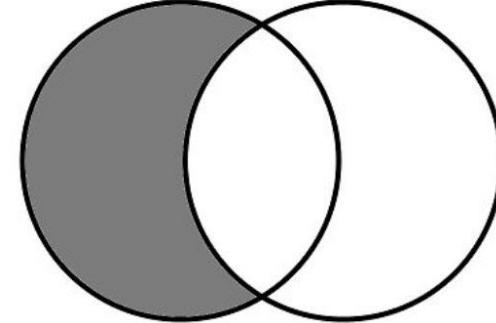
Intersection



Union



Difference



Merging DataFrames

- Let's use the weather and station_info dataframes to illustrate set operations.
- First, we must set the index to the column(s) that will be used for the join operation:

```
>>> weather.set_index('station', inplace=True)  
>>> station_info.set_index('id', inplace=True)
```

Merging DataFrames

- To see what will remain with an inner join, we can take the intersection of the indices, which shows us the overlapping stations:

```
>>> weather.index.intersection(station_info.index)
Index(['GHCND:US1CTFR0039', ..., 'GHCND:USW1NYQN0029'],
      dtype='object', length=110)
```

Merging DataFrames

- With the set difference, we can easily see that, when performing an inner join, we don't lose any rows from the weather data, but we lose 169 stations that don't have weather observations:

```
>>> weather.index.difference(station_info.index)
Index([], dtype='object')
>>> station_info.index.difference(weather.index)
Index(['GHCND:US1CTFR0022', ..., 'GHCND:USW00014786'],
      dtype='object', length=169)
```

Merging DataFrames

- Remember, the weather dataframe contains the stations repeated throughout because they provide daily measurements, so we call the unique() method before taking the union to see the number of stations we will keep:

```
>>> weather.index.unique().union(station_info.index)
Index([ 'GHCND:US1CTFR0022' , ... , 'GHCND:USW00094789' ],
      dtype='object', length=279)
```

Using DataFrame operations to enrich data

- Let's import what we will need and read in the data:

```
>>> import numpy as np  
>>> import pandas as pd  
>>> weather = pd.read_csv(  
...     'data/nyc_weather_2018.csv', parse_dates=['date'])  
... )  
>>> fb = pd.read_csv(  
...     'data/fb_2018.csv', index_col='date', parse_dates=True  
... )
```

Arithmetic and statistics

- Rather than using mathematical operators for subtraction and division, we will use the sub() and div() methods, respectively:

```
>>> fb.assign(  
...     abs_z_score_volume=lambda x: x.volume \  
...             .sub(x.volume.mean()).div(x.volume.std()).abs()  
... ).query('abs_z_score_volume > 3')
```

Arithmetic and statistics

- Five days in 2018 had Z-scores for volume traded greater than three in absolute value.
- These dates in particular will come up often in the rest of this lesson as they mark some trouble points for Facebook's stock price:

	open	high	low	close	volume	abs_z_score_volume
date						
2018-03-19	177.01	177.17	170.06	172.56	88140060	3.145078
2018-03-20	167.47	170.20	161.95	168.15	129851768	5.315169
2018-03-21	164.80	173.40	163.30	169.39	106598834	4.105413
2018-03-26	160.82	161.10	149.02	160.06	126116634	5.120845
2018-07-26	174.89	180.13	173.75	176.26	169803668	7.393705

Arithmetic and statistics

- By combining these, we can see which five days had the largest percentage change of volume traded in Facebook stock from the day prior:

```
>>> fb.assign(  
...     volume_pct_change=fb.volume.pct_change(),  
...     pct_change_rank=lambda x: \  
...         x.volume_pct_change.abs().rank(ascending=False)  
... ).nsmallest(5, 'pct_change_rank')
```

Arithmetic and statistics

	open	high	low	close	volume	volume_pct_change	pct_change_rank
date							
2018-01-12	178.06	181.48	177.40	179.37	77551299	7.087876	1.0
2018-03-19	177.01	177.17	170.06	172.56	88140060	2.611789	2.0
2018-07-26	174.89	180.13	173.75	176.26	169803668	1.628841	3.0
2018-09-21	166.64	167.25	162.81	162.93	45994800	1.428956	4.0
2018-03-26	160.82	161.10	149.02	160.06	126116634	1.352496	5.0

Arithmetic and statistics

- We can use slicing to look at the change around this announcement:

```
>>> fb['2018-01-11':'2018-01-12']
```

Arithmetic and statistics

- Notice how we are able to combine everything we learned in the last few lessons to get interesting insights from our data.
- We were able to sift through a year's worth of stock data and find some days that had large effects on Facebook stock (good or bad):

	open	high	low	close	volume
date					
2018-01-11	188.40	188.40	187.38	187.77	9588587
2018-01-12	178.06	181.48	177.40	179.37	77551299

Arithmetic and statistics

- Lastly, we can inspect the dataframe with aggregated Boolean operations.
- For example, we can see that Facebook stock never had a daily low price greater than \$215 in 2018 with the any() method:

```
>>> (fb > 215).any()  
open           True  
high           True  
low            False  
close          True  
volume         True  
dtype: bool
```

Arithmetic and statistics

- If we want to see if all the rows in a column meet the criteria, we can use the `all()` method.
- This tells us that Facebook has at least one day for the opening, high, low, and closing prices with a value less than or equal to \$215:

```
>>> (fb > 215).all()
open      False
high     False
low      False
close     False
volume    True
dtype: bool
```

Binning

- One interesting thing we could do with the volume traded would be to see which days had high trade volume and look for news about Facebook on those days or large swings in price.
- Unfortunately, it is highly unlikely that the volume will be the same any two days; in fact, we can confirm that, in the data, no two days have the same volume traded:

```
>>> (fb.volume.value_counts() > 1).sum()  
0
```

Binning

- First, we should decide how many bins we want to create—three seems like a good split, since we can label the bins low, medium, and high.
- Next, we need to determine the width of each bin; pandas tries to make this process as painless as possible, so if we want equally-sized bins, all we have to do is specify the number of bins we want (otherwise, we must specify the upper bound for each bin as a list):

```
>>> volume_binned = pd.cut(  
...     fb.volume, bins=3, labels=['low', 'med', 'high']  
... )  
>>> volume_binned.value_counts()  
low    240  
med      8  
high     3  
Name: volume, dtype: int64
```

Binning

- Let's look at the data for the three days of high volume:

```
>>> fb[volume_binned == 'high']\n...     .sort_values('volume', ascending=False)
```

Binning

- Even among the high-volume days, we can see that July 26, 2018 had a much higher trade volume compared to the other two dates in March (nearly 40 million additional shares were traded):

date	open	high	low	close	volume
2018-07-26	174.89	180.13	173.75	176.26	169803668
2018-03-20	167.47	170.20	161.95	168.15	129851768
2018-03-26	160.82	161.10	149.02	160.06	126116634

Binning

- Let's pull out this data:

```
>>> fb['2018-07-25':'2018-07-26']
```

Binning

- Not only was there a huge drop in stock price, but the volume traded also skyrocketed, increasing by more than 100 million.

date	open	high	low	close	volume
2018-07-25	215.715	218.62	214.27	217.50	64592585
2018-07-26	174.890	180.13	173.75	176.26	169803668

Binning

- If we look at the other two days marked as high-volume trading days, we will find a plethora of information as to why.
- Both of these days were marked by scandal for Facebook. :

```
>>> fb['2018-03-16':'2018-03-20']
```

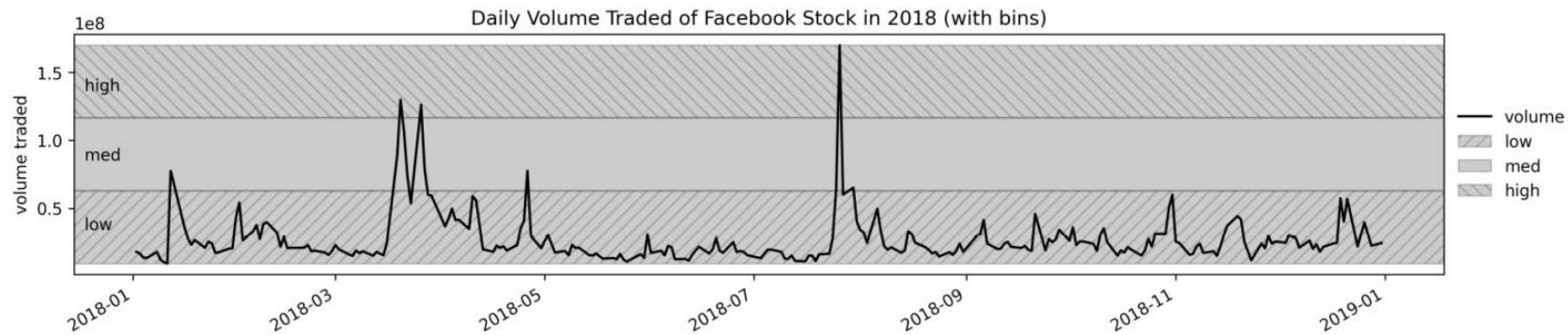
Binning

- Things only got worse once more information was revealed in the following days with regards to the severity of the incident:

	open	high	low	close	volume
date					
2018-03-16	184.49	185.33	183.41	185.09	24403438
2018-03-19	177.01	177.17	170.06	172.56	88140060
2018-03-20	167.47	170.20	161.95	168.15	129851768

Binning

- Most days were pretty close in volume traded; however, a few days caused the bin width to be rather large, which left us with a large imbalance of days per bin:



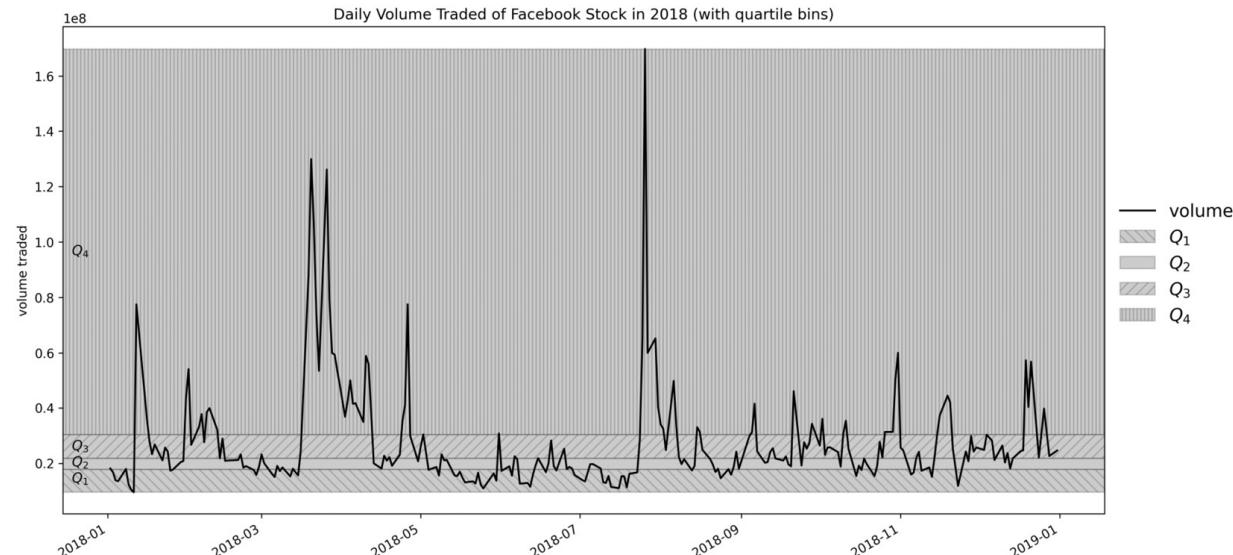
Binning

- We can bin the volumes into quartiles to evenly bucket the observations into bins of varying width, giving us the 63 highest trading volume days in the q4 bin:

```
>>> volume_qbinned = pd.qcut(  
...      fb.volume, q=4, labels=['q1', 'q2', 'q3', 'q4']  
... )  
>>> volume_qbinned.value_counts()  
q1    63  
q2    63  
q4    63  
q3    62  
Name: volume, dtype: int64
```

Binning

- Notice that the bins don't cover the same range of volume traded anymore:



Applying functions

- Before we get started, let's isolate the weather observations from the Central Park station and pivot the data:

```
>>> central_park_weather = weather.query(  
...     'station == "GHCND:USW00094728"'  
... ).pivot(index='date', columns='datatype',  
values='value')
```

Applying functions

- By isolating our calculation to October, we can see if October had any days with very different weather:

```
>>> oct_weather_z_scores = central_park_weather\  
...     .loc['2018-10', ['TMIN', 'TMAX', 'PRCP']]\  
...     .apply(lambda x: x.sub(x.mean()).div(x.std()))  
>>> oct_weather_z_scores.describe().T
```

Applying functions

- TMIN and TMAX don't appear to have any values that differ much from the rest of October, but PRCP does:

	count	mean	std	min	25%	50%	75%	max
datatype								
TMIN	31.0	-1.790682e-16	1.0	-1.339112	-0.751019	-0.474269	1.065152	1.843511
TMAX	31.0	1.951844e-16	1.0	-1.305582	-0.870013	-0.138258	1.011643	1.604016
PRCP	31.0	4.655774e-17	1.0	-0.394438	-0.394438	-0.394438	-0.240253	3.936167

Applying functions

- We can use `query()` to extract the value for this date:

```
>>> oct_weather_z_scores.query('PRCP > 3').PRCP  
date  
2018-10-27    3.936167  
Name: PRCP, dtype: float64
```

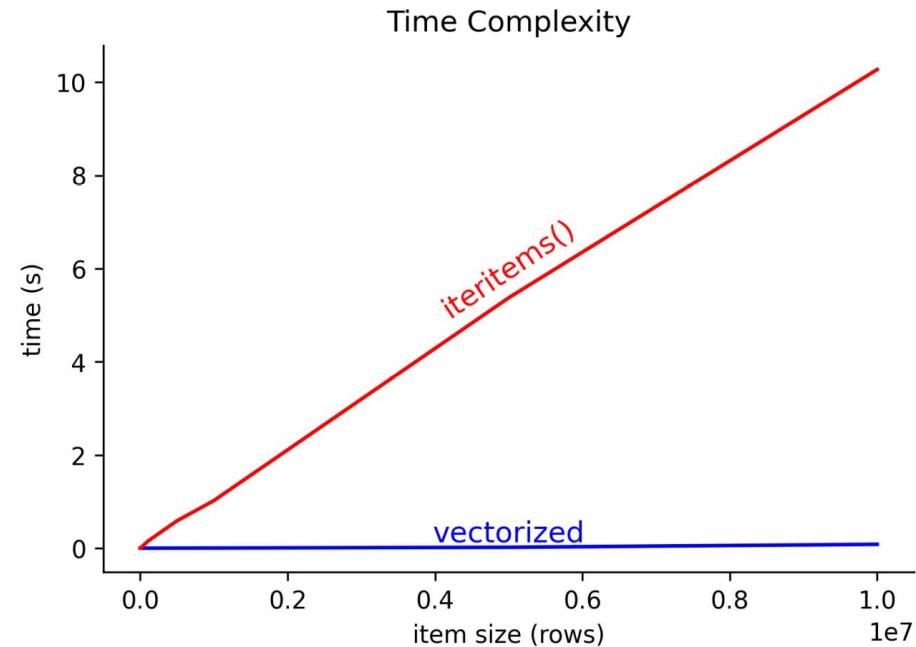
Applying functions

- If we look at the summary statistics for precipitation in October, we can see that this day had much more precipitation than the rest:

```
>>> central_park_weather.loc['2018-10', 'PRCP'].describe()  
count      31.000000  
mean       2.941935  
std        7.458542  
min       0.000000  
25%       0.000000  
50%       0.000000  
75%       1.150000  
max      32.300000  
Name: PRCP, dtype: float64
```

Applying functions

- For instance, look at how the time to complete the simple operation of adding the number 10 to each value in a series of floats grows linearly with the number of rows when using `iteritems()`, but stays near zero, regardless of size, when using a vectorized operation:



Window calculations

Rolling windows

- Fortunately, we can use the `rolling()` method to get this information easily:

```
>>> central_park_weather.loc['2018-10'].assign(  
...     rolling_PRCP=lambda x: x.PRCP.rolling('3D').sum()  
... )[['PRCP', 'rolling_PRCP']].head(7).T
```

Window calculations

- After performing the rolling 3-day sum, each date will show the sum of that day's and the previous two days' precipitation:

date	2018-10-01	2018-10-02	2018-10-03	2018-10-04	2018-10-05	2018-10-06	2018-10-07
datatype							
PRCP	0.0	17.5	0.0	1.0	0.0	0.0	0.0
rolling_PRCP	0.0	17.5	17.5	18.5	1.0	1.0	0.0

Window calculations

- The rolling calculation can also be applied to all the columns at once:

```
>>> central_park_weather.loc['2018-10']\n...     .rolling('3D').mean().head(7).iloc[:,6]
```

Window calculations

- This gives us the 3-day rolling average for all the weather observations from Central Park:

datatype	AWND	PRCP	SNOW	SNWD	TMAX	TMIN
date						
2018-10-01	0.900000	0.000000	0.0	0.0	24.400000	17.200000
2018-10-02	0.900000	8.750000	0.0	0.0	24.700000	17.750000
2018-10-03	0.966667	5.833333	0.0	0.0	24.233333	17.566667
2018-10-04	0.800000	6.166667	0.0	0.0	24.233333	17.200000
2018-10-05	1.033333	0.333333	0.0	0.0	23.133333	16.300000
2018-10-06	0.833333	0.333333	0.0	0.0	22.033333	16.300000
2018-10-07	1.066667	0.000000	0.0	0.0	22.600000	17.400000

Window calculations

- Then, we will join it to the original data so that we can compare the results:

```
>>> central_park_weather\  
...     ['2018-10-01':'2018-10-07'].rolling('3D').agg({  
...     'TMAX': 'max', 'TMIN': 'min',  
...     'AWND': 'mean', 'PRCP': 'sum'  
... }).join( # join with original data for comparison  
...     central_park_weather[['TMAX', 'TMIN', 'AWND', 'PRCP']],  
...     lsuffix='_rolling'  
... ).sort_index(axis=1) # put rolling calcs next to originals
```

Window calculations

- Using `agg()`, we were able to calculate different rolling aggregations for each column:

	AWND	AWND_rolling	PRCP	PRCP_rolling	TMAX	TMAX_rolling	TMIN	TMIN_rolling
date								
2018-10-01	0.9	0.900000	0.0	0.0	24.4	24.4	17.2	17.2
2018-10-02	0.9	0.900000	17.5	17.5	25.0	25.0	18.3	17.2
2018-10-03	1.1	0.966667	0.0	17.5	23.3	25.0	17.2	17.2
2018-10-04	0.4	0.800000	1.0	18.5	24.4	25.0	16.1	16.1
2018-10-05	1.6	1.033333	0.0	1.0	21.7	24.4	15.6	15.6
2018-10-06	0.5	0.833333	0.0	1.0	20.0	24.4	17.2	15.6
2018-10-07	1.1	1.066667	0.0	0.0	26.1	26.1	19.4	15.6

Window calculations

- **Expanding windows**
- With the Central Park weather data, let's use the expanding() method to calculate the month-to-date average precipitation:

```
>>> central_park_weather.loc['2018-06'].assign(  
...     TOTAL_PRCP=lambda x: x.PRCP.cumsum(),  
...     AVG_PRCP=lambda x: x.PRCP.expanding().mean()  
... ).head(10)[['PRCP', 'TOTAL_PRCP', 'AVG_PRCP']].T
```

Window calculations

- Note that while there is no method for the cumulative mean, we are able to use the expanding() method to calculate it.
- The values in the AVG_PRCP column are the values in the TOTAL_PRCP column divided by the number of days processed:

date	2018-06-01	2018-06-02	2018-06-03	2018-06-04	2018-06-05	2018-06-06	2018-06-07	2018-06-08	2018-06-09	2018-06-10
datatype										
PRCP	6.9	2.00	6.4	4.10	0.00	0.000000	0.000000	0.000	0.000000	0.30
TOTAL_PRCP	6.9	8.90	15.3	19.40	19.40	19.400000	19.400000	19.400	19.400000	19.70
AVG_PRCP	6.9	4.45	5.1	4.85	3.88	3.233333	2.771429	2.425	2.155556	1.97

Window calculations

- As we did with rolling(), we can provide column-specific aggregations with the agg() method.
- Let's find the expanding maximum temperature, minimum temperature, average wind speed, and total precipitation.
- Note that we can also pass in NumPy functions to agg():

```
>>> central_park_weather\  
...     ['2018-10-01':'2018-10-07'].expanding().agg({  
...     'TMAX': np.max, 'TMIN': np.min,  
...     'AWND': np.mean, 'PRCP': np.sum  
... }).join(  
...     central_park_weather[['TMAX', 'TMIN', 'AWND', 'PRCP']],  
...     lsuffix='_expanding'  
... ).sort_index(axis=1)
```

Window calculations

- Once again, we joined the window calculations with the original data for comparison:

	AWND	AWND_expanding	PRCP	PRCP_expanding	TMAX	TMAX_expanding	TMIN	TMIN_expanding
date								
2018-10-01	0.9	0.900000	0.0	0.0	24.4	24.4	17.2	17.2
2018-10-02	0.9	0.900000	17.5	17.5	25.0	25.0	18.3	17.2
2018-10-03	1.1	0.966667	0.0	17.5	23.3	25.0	17.2	17.2
2018-10-04	0.4	0.825000	1.0	18.5	24.4	25.0	16.1	16.1
2018-10-05	1.6	0.980000	0.0	18.5	21.7	25.0	15.6	15.6
2018-10-06	0.5	0.900000	0.0	18.5	20.0	25.0	17.2	15.6
2018-10-07	1.1	0.928571	0.0	18.5	26.1	26.1	19.4	15.6

Window calculations

- **Exponentially weighted moving windows**
- Note that we use the span argument to specify the number of periods to use for the EWMA calculation:

```
>>> central_park_weather.assign(  
...     AVG=lambda x: x.TMAX.rolling('30D').mean(),  
...     EWMA=lambda x: x.TMAX.ewm(span=30).mean()  
... ).loc['2018-09-29':'2018-10-08', ['TMAX', 'EWMA', 'AVG']].T
```

Window calculations

- Unlike the rolling average, the EWMA places higher importance on more recent observations, so the jump in temperature on October 7th has a larger effect on the EWMA than the rolling average:

date	2018-09-29	2018-09-30	2018-10-01	2018-10-02	2018-10-03	2018-10-04	2018-10-05	2018-10-06	2018-10-07	2018-10-08
datatype										
TMAX	22.200000	21.100000	24.400000	25.000000	23.300000	24.400000	21.700000	20.000000	26.100000	23.300000
EWMA	24.410887	24.197281	24.210360	24.261304	24.199285	24.212234	24.050154	23.788854	23.937960	23.896802
AVG	24.723333	24.573333	24.533333	24.460000	24.163333	23.866667	23.533333	23.070000	23.143333	23.196667

Pipes

- Pipes facilitate chaining together operations that expect pandas data structures as their first argument.
- By using pipes, we can build up complex workflows without needing to write highly nested and hard-to-read code.
- In general, pipes let us turn something like `f(g(h(data), 20), x=True)` into the following, making it much easier to read:

```
data.pipe(h)\ # first call h(data)
    .pipe(g, 20)\ # call g on the result with positional arg 20
    .pipe(f, x=True) # call f on result with keyword arg x=True
```

Pipes

- Say we wanted to print the dimensions of a subset of the Facebook dataframe with some formatting by calling this function:

```
>>> def get_info(df):
...     return '%d rows, %d cols and max closing Z-score: %d'
...             % (*df.shape, df.close.max())
```

Pipes

- Before we call the function, however, we want to calculate the Z-scores for all the columns.
- One approach is the following:

```
>>> get_info(fb.loc['2018-Q1']\n...     .apply(lambda x: (x - x.mean())/x.std()))
```

Pipes

- Alternatively, we could pipe the dataframe after calculating the Z-scores to this function:

```
>>> fb.loc['2018-Q1'].apply(lambda x: (x - x.mean())/x.std())\n...     .pipe(get_info)
```

Pipes

- We can use pipes to extend that functionality to methods of pandas data structures:

```
>>> fb.pipe(pd.DataFrame.rolling, '20D').mean().equals(  
...     fb.rolling('20D').mean()  
... ) # the pipe is calling pd.DataFrame.rolling(fb, '20D')  
True
```

Pipes

- We will import the function and use ?? from IPython to view the function definition:

```
>>> from window_calc import window_calc
>>> window_calc??
Signature: window_calc(df, func, agg_dict, *args, **kwargs)
Source:
def window_calc(df, func, agg_dict, *args, **kwargs):
    """
        Run a window calculation of your choice on the data.

    Parameters:
        - df: The `DataFrame` object to run the calculation on.
        - func: The window calculation method that takes `df` as the first argument.
        - agg_dict: Information to pass to `agg()`, could be a dictionary mapping the columns to the aggregation function to use, a string name for the function, or the function itself.
        - args: Positional arguments to pass to `func`.
        - kwargs: Keyword arguments to pass to `func`.

    Returns:
        A new `DataFrame` object.

    """
    return df.pipe(func, *args, **kwargs).agg(agg_dict)
File:      ~/.../ch_04/window_calc.py
Type:     function
```

Pipes

- Our window_calc() function takes the dataframe, the function to execute (as long as it takes a dataframe as its first argument), and information on how to aggregate the result, along with any optional parameters, and gives us back a new dataframe with the results of the window calculations.
- Let's use this function to find the expanding median of the Facebook stock data:

```
>>> window_calc(fb, pd.DataFrame.expanding, np.median).head()
```

Pipes

	open	high	low	close	volume
date					
2018-01-02	177.68	181.580	177.5500	181.420	18151903.0
2018-01-03	179.78	183.180	179.4400	183.045	17519233.0
2018-01-04	181.88	184.780	181.3300	184.330	16886563.0
2018-01-05	183.39	185.495	182.7148	184.500	15383729.5
2018-01-08	184.90	186.210	184.0996	184.670	16886563.0

Pipes

- We need this behavior in order to use the `ewm()` method for the EWMA of the closing price of Facebook stock:

```
>>> window_calc(fb, pd.DataFrame.ewm, 'mean', span=3).head()
```

Pipes

- In the previous example, we had to use `**kwargs` because the `span` argument is not the first argument that `ewm()` receives, and we didn't want to pass the ones before it:

	open	high	low	close	volume
date					
2018-01-02	177.680000	181.580000	177.550000	181.420000	1.815190e+07
2018-01-03	180.480000	183.713333	180.070000	183.586667	1.730834e+07
2018-01-04	183.005714	185.140000	182.372629	184.011429	1.534980e+07
2018-01-05	184.384000	186.078667	183.736560	185.525333	1.440299e+07
2018-01-08	185.837419	187.534839	185.075110	186.947097	1.625679e+07

Pipes

- To calculate the rolling 3-day weather aggregations for Central Park, we take advantage of *args since we know that the window is the first argument to rolling():

```
>>> window_calc(  
...     central_park_weather.loc['2018-10'],  
...     pd.DataFrame.rolling,  
...     {'TMAX': 'max', 'TMIN': 'min',  
...      'AWND': 'mean', 'PRCP': 'sum'},  
...     '3D'  
... ).head()
```

Pipes

- We were able to aggregate each of the columns differently since we passed in a dictionary instead of a single value:

	TMAX	TMIN	AWND	PRCP
date				
2018-10-01	24.4	17.2	0.900000	0.0
2018-10-02	25.0	17.2	0.900000	17.5
2018-10-03	25.0	17.2	0.966667	17.5
2018-10-04	25.0	16.1	0.800000	18.5
2018-10-05	24.4	15.6	1.033333	1.0

Aggregating data

- For this section, we will be working in the 3-aggregations.ipynb notebook.
- Let's import pandas and numpy and read in the data we will be working with:

```
>>> import numpy as np
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_2018.csv', index_col='date', parse_dates=True
... ).assign(trading_volume=lambda x: pd.cut(
...     x.volume, bins=3, labels=['low', 'med', 'high']
... ))
>>> weather = pd.read_csv(
...     'data/weather_by_station.csv',
...     index_col='date', parse_dates=True
... )
```

Aggregating data

- Note that the weather data for this section has been merged with some of the station data:

datatype	station	value	station_name	
date				
2018-01-01	PRCP	GHCND:US1CTFR0039	0.0	STAMFORD 4.2 S, CT US
2018-01-01	PRCP	GHCND:US1NJBG0015	0.0	NORTH ARLINGTON 0.7 WNW, NJ US
2018-01-01	SNOW	GHCND:US1NJBG0015	0.0	NORTH ARLINGTON 0.7 WNW, NJ US
2018-01-01	PRCP	GHCND:US1NJBG0017	0.0	GLEN ROCK 0.7 SSE, NJ US
2018-01-01	SNOW	GHCND:US1NJBG0017	0.0	GLEN ROCK 0.7 SSE, NJ US

Aggregating data

- Before we dive into any calculations, let's make sure that our data won't be displayed in scientific notation.
- We will modify how floats are formatted for displaying.
- The format we will apply is .2f, which will provide the float with two digits after the decimal point:

```
>>> pd.set_option('display.float_format', lambda x: '%.2f' % x)
```

Summarizing DataFrames

- Note that we won't get anything back for the trading_volume column, which contains the volume traded bins from pd.cut(); this is because we aren't specifying an aggregation to run on that column:

```
>>> fb.agg({
...     'open': np.mean, 'high': np.max, 'low': np.min,
...     'close': np.mean, 'volume': np.sum
... })
      open          171.45
      high          218.62
      low           123.02
      close          171.51
  volume    6949682394.00
      dtype: float64
```

Summarizing DataFrames

- In this case, since we will be performing the sum on both, we can either use agg('sum') or call sum() directly:

```
>>> weather.query('station == "GHCND:USW00094728"')\
...     .pivot(columns='datatype', values='value')\
...     [['SNOW', 'PRCP']].sum()
datatype
SNOW    1007.00
PRCP    1665.30
dtype: float64
```

Summarizing DataFrames

- The index of this dataframe will tell us which metric is being calculated for which column:

```
>>> fb.agg({  
...     'open': 'mean',  
...     'high': ['min', 'max'],  
...     'low': ['min', 'max'],  
...     'close': 'mean'  
... })
```

Summarizing DataFrames

- This results in a dataframe where the rows indicate the aggregation function being applied to the data columns.
- Note that we get nulls for any combination of aggregation and column that we didn't explicitly ask for:

	open	high	low	close
mean	171.45	NaN	NaN	171.51
min	NaN	129.74	123.02	NaN
max	NaN	218.62	214.27	NaN

Aggregating by group

- To calculate the aggregations per group, we must first call the `groupby()` method on the dataframe and provide the column(s) we want to use to determine distinct groups.
- Let's look at the average of our stock data points for each of the volume traded bins we created with `pd.cut()`; remember, these are three equal-width bins:

```
>>> fb.groupby('trading_volume').mean()
```

Aggregating by group

- The average OHLC prices are smaller for larger trading volumes, which was to be expected given that the three dates in the high-volume traded bin were selloffs:

trading_volume		open	high	low	close	volume
low		171.36	173.46	169.31	171.43	24547207.71
med		175.82	179.42	172.11	175.14	79072559.12
high		167.73	170.48	161.57	168.16	141924023.33

Aggregating by group

- After running `groupby()`, we can also select specific columns for aggregation:

```
>>> fb.groupby('trading_volume')\n...     ['close'].agg(['min', 'max', 'mean'])
```

Aggregating by group

- This gives us the aggregations for the closing price in each volume traded bucket:

trading_volume		min	max	mean
	low	124.06	214.67	171.43
	med	152.22	217.50	175.14
	high	160.06	176.26	168.16

Aggregating by group

- As we did previously, we can provide lists of functions per column; the result, however, will look a little different:

```
>>> fb_agg = fb.groupby('trading_volume').agg({  
...     'open': 'mean', 'high': ['min', 'max'],  
...     'low': ['min', 'max'], 'close': 'mean'  
... })  
>>> fb_agg
```

Aggregating by group

- We now have a hierarchical index in the columns.
- Remember, this means that if we want to select the minimum low price for the medium volume traded bucket, we need to use `fb_agg.loc['med', 'low']['min']`:

		open		high		low	close
		mean	min	max	min	max	mean
trading_volume							
	low	171.36	129.74	216.20	123.02	212.60	171.43
	med	175.82	162.85	218.62	150.75	214.27	175.14
	high	167.73	161.10	180.13	149.02	173.75	168.16

Aggregating by group

- The columns are stored in a MultiIndex object:

```
>>> fb_agg.columns  
MultiIndex([( 'open', 'mean'),  
            ('high', 'min'),  
            ('high', 'max'),  
            ('low', 'min'),  
            ('low', 'max'),  
            ('close', 'mean')],  
           )
```

Aggregating by group

- We can use a list comprehension to remove this hierarchy and instead have our column names in the form of <column>_<agg>.
- At each iteration, we will get a tuple of the levels from the MultiIndex object, which we can combine into a single string to remove the hierarchy:

```
>>> fb_agg.columns = ['_'.join(col_agg)
...                   for col_agg in fb_agg.columns]
>>> fb_agg.head()
```

Aggregating by group

- This replaces the hierarchy in the columns with a single level:

trading_volume	open_mean	high_min	high_max	low_min	low_max	close_mean
low	171.36	129.74	216.20	123.02	212.60	171.43
med	175.82	162.85	218.62	150.75	214.27	175.14
high	167.73	161.10	180.13	149.02	173.75	168.16

Aggregating by group

- Since the result is a single-column DataFrame object, we call `squeeze()` to turn it into a Series object:

```
>>> weather.loc['2018-10'].query('datatype == "PRCP"')\n...     .groupby(level=0).mean().head().squeeze()\n\n      date\n2018-10-01    0.01\n2018-10-02    2.23\n2018-10-03   19.69\n2018-10-04    0.32\n2018-10-05    0.96\n\nName: value, dtype: float64
```

Aggregating by group

- Since this will create a multi-level index, we will also use unstack() to put the inner level (the quarter) along the columns after the aggregation is performed:

```
>>> weather.query('datatype == "PRCP"]').groupby(  
...     ['station_name', pd.Grouper(freq='Q')]  
... ).sum().unstack().sample(5, random_state=1)
```

Aggregating by group

- We could also see which quarter has the most/least precipitation across the stations:

station_name	date	value			
		2018-03-31	2018-06-30	2018-09-30	2018-12-31
WANTAGH 1.1 NNE, NY US		279.90	216.80	472.50	277.20
STATEN ISLAND 1.4 SE, NY US		379.40	295.30	438.80	409.90
SYOSSET 2.0 SSW, NY US		323.50	263.30	355.50	459.90
STAMFORD 4.2 S, CT US		338.00	272.10	424.70	390.00
WAYNE TWP 0.8 SSW, NJ US		246.20	295.30	620.90	422.00

Aggregating by group

- Finally, we will use nlargest() to get the five months with the most precipitation:

```
>>> weather.query('datatype == "PRCP"')\
...      .groupby(level=0).mean()\n...      .groupby(pd.Grouper(freq='M')).sum().value.nlargest()\n\n  date\n  2018-11-30    210.59\n  2018-09-30    193.09\n  2018-08-31    192.45\n  2018-07-31    160.98\n  2018-02-28    158.11\n\n  Name: value, dtype: float64
```

Aggregating by group

- Therefore, we can call it on a Series object and always get a Series object back, regardless of what the aggregation function itself would return:

```
>>> weather.query('datatype == "PRCP"')\
...     .rename(dict(value='prcp'), axis=1)\
...     .groupby(level=0).mean()\n...     .groupby(pd.Grouper(freq='M'))\
...     .transform(np.sum)[ '2018-01-28':'2018-02-03' ]
```

Aggregating by group

- Rather than getting a single sum for January and another for February, notice that we have the same value being repeated for the January entries and a different one for the February ones.
- Note that the value for February is the value we found in the previous result:

prcp	
date	
2018-01-28	69.31
2018-01-29	69.31
2018-01-30	69.31
2018-01-31	69.31
2018-02-01	158.11
2018-02-02	158.11
2018-02-03	158.11

Aggregating by group

- We can make this a column in our dataframe to easily calculate the percentage of the monthly precipitation that occurred each day.
- Then, we can use nlargest() to pull out the largest values:

```
>>> weather.query('datatype == "PRCP"')\
...     .rename(dict(value='prcp'), axis=1)\n...     .groupby(level=0).mean()\n...     .assign(\n...         total_prcp_in_month=lambda x: x.groupby(\n...             pd.Grouper(freq='M')).transform(np.sum),\n...         pct_monthly_prcp=lambda x: \
...             x.prcp.div(x.total_prcp_in_month)\n...     ).nlargest(5, 'pct_monthly_prcp')
```

Aggregating by group

- They were also consecutive days:

	prcp	total_prcp_in_month	pct_monthly_prcp
date			
2018-10-12	34.77	105.63	0.33
2018-01-13	21.66	69.31	0.31
2018-03-02	38.77	137.46	0.28
2018-04-16	39.34	140.57	0.28
2018-04-17	37.30	140.57	0.27

Pivot tables and crosstabs

- Let's create a pivot table of averaged OHLC data for Facebook per volume traded bin:

```
>>> fb.pivot_table(columns='trading_volume')
```

Pivot tables and crosstabs

- Notice that the index for the columns has a name (`trading_volume`):

<code>trading_volume</code>	low	med	high
<code>close</code>	171.43	175.14	168.16
<code>high</code>	173.46	179.42	170.48
<code>low</code>	169.31	172.11	161.57
<code>open</code>	171.36	175.82	167.73
<code>volume</code>	24547207.71	79072559.12	141924023.33

Pivot tables and crosstabs

- We will use the median to aggregate any overlapping combinations (if any):

```
>>> weather.reset_index().pivot_table(  
...     index=['date', 'station', 'station_name'],  
...     columns='datatype',  
...     values='value',  
...     aggfunc='median'  
... ).reset_index().tail()
```

Pivot tables and crosstabs

- After resetting the index, we have our data in wide format.
- One final step would be to rename the index:

datatype	date	station	station_name	AWND	DAPR	MDPR	PGTM	PRCP	SNOW	SNWD	...
28740	2018-12-31	GHCND:USW00054787	FARMINGDALE REPUBLIC AIRPORT, NY US	5.00	NaN	NaN	2052.00	28.70	NaN	NaN	...
28741	2018-12-31	GHCND:USW00094728	NY CITY CENTRAL PARK, NY US	NaN	NaN	NaN	NaN	25.90	0.00	0.00	...
28742	2018-12-31	GHCND:USW00094741	TETERBORO AIRPORT, NJ US	1.70	NaN	NaN	1954.00	29.20	NaN	NaN	...
28743	2018-12-31	GHCND:USW00094745	WESTCHESTER CO AIRPORT, NY US	2.70	NaN	NaN	2212.00	24.40	NaN	NaN	...
28744	2018-12-31	GHCND:USW00094789	JFK INTERNATIONAL AIRPORT, NY US	4.10	NaN	NaN	NaN	31.20	0.00	0.00	...

Pivot tables and crosstabs

- By default, the values in the cells will be the count:

```
>>> pd.crosstab(  
...     index=fb.trading_volume, columns=fb.index.month,  
...     colnames=['month'] # name the columns index  
... )
```

Pivot tables and crosstabs

- This makes it easy to see the months when high volumes of Facebook stock were traded:

month	1	2	3	4	5	6	7	8	9	10	11	12
trading_volume												
low	20	19	15	20	22	21	18	23	19	23	21	19
med	1	0	4	1	0	0	2	0	0	0	0	0
high	0	0	2	0	0	0	1	0	0	0	0	0

Pivot tables and crosstabs

- To illustrate this, let's find the average closing price of each trading volume bucket per month instead of the count in the previous example:

```
>>> pd.crosstab(  
...     index=fb.trading_volume, columns=fb.index.month,  
...     colnames=['month'], values=fb.close, aggfunc=np.mean  
... )
```

Pivot tables and crosstabs

- We now get the average closing price per month, per volume traded bin, with null values when that combination wasn't present in the data:

month	1	2	3	4	5	6	7	8	9	10	11	12
trading_volume												
low	185.24	180.27	177.07	163.29	182.93	195.27	201.92	177.49	164.38	154.19	141.64	137.16
med	179.37	NaN	164.76	174.16	NaN	NaN	194.28	NaN	NaN	NaN	NaN	NaN
high	NaN	NaN	164.11	NaN	NaN	NaN	176.26	NaN	NaN	NaN	NaN	NaN

Pivot tables and crosstabs

- Let's count the number of times each station recorded snow per month and include the subtotals:

```
>>> snow_data = weather.query('datatype == "SNOW"')
>>> pd.crosstab(
...     index=snow_data.station_name,
...     columns=snow_data.index.month,
...     colnames=['month'],
...     values=snow_data.value,
...     aggfunc=lambda x: (x > 0).sum(),
...     margins=True, # show row and column subtotals
...     margins_name='total observations of snow' # subtotals
... )
```

Pivot tables and crosstabs

- Along the bottom row, we have the total snow observations per month, while down the rightmost column, we have the total snow observations in 2018 per station:

month	1	2	3	4	5	6	7	8	9	10	11	12	total observations of snow
station_name													
ALBERTSON 0.2 SSE, NY US	3.00	1.00	3.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	9.00
AMITYVILLE 0.1 WSW, NY US	1.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.00
AMITYVILLE 0.6 NNE, NY US	3.00	1.00	3.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	8.00
ARMONK 0.3 SE, NY US	6.00	4.00	6.00	3.00	0.00	0.00	0.00	0.00	0.00	1.00	3.00	0.00	23.00
BLOOMINGDALE 0.7 SSE, NJ US	2.00	1.00	3.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	8.00
...
WESTFIELD 0.6 NE, NJ US	3.00	0.00	4.00	1.00	0.00	NaN	0.00	0.00	0.00	1.00	NaN	0.00	9.00
WOODBRIDGE TWP 1.1 ESE, NJ US	4.00	1.00	3.00	2.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	11.00
WOODBRIDGE TWP 1.1 NNE, NJ US	2.00	1.00	3.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	7.00
WOODBRIDGE TWP 3.0 NNW, NJ US	NaN	0.00	0.00	NaN	NaN	0.00	NaN	NaN	0.00	0.00	NaN	0.00	0.00
total observations of snow	190.00	97.00	237.00	81.00	0.00	0.00	0.00	0.00	0.00	49.00	13.00	0.00	667.00

Working with time series data

- Let's start off by reading in the Facebook data from the previous sections:

```
>>> import numpy as np  
>>> import pandas as pd  
>>> fb = pd.read_csv(  
...     'data/fb_2018.csv', index_col='date', parse_dates=True  
... ).assign(trading_volume=lambda x: pd.cut(  
...     x.volume, bins=3, labels=['low', 'med', 'high'])  
... ))
```

Time-based selection and filtering

- Let's start with a quick recap of datetime slicing and indexing.
- We can easily isolate data for the year by indexing on it:
`fb.loc['2018']`.
- In the case of our stock data, the full dataframe would be returned because we only have 2018 data; however, we can filter to a month (`fb.loc['2018-10']`) or to a range of dates.
- Note that using `loc[]` is optional with ranges:
`>>> fb['2018-10-11':'2018-10-15']`

Time-based selection and filtering

- We only get three days back because the stock market is closed on the weekends:

	open	high	low	close	volume	trading_volume
date						
2018-10-11	150.13	154.81	149.1600	153.35	35338901	low
2018-10-12	156.73	156.89	151.2998	153.74	25293492	low
2018-10-15	153.32	155.57	152.5500	153.52	15433521	low

Time-based selection and filtering

- Keep in mind that the date range can also be supplied using other frequencies, such as month or the quarter of the year:

```
>>> fb.loc['2018-q1'].equals(fb['2018-01':'2018-03'])  
True
```

Time-based selection and filtering

- When targeting the beginning or end of a date range, pandas has some additional methods for selecting the first or last rows within a specified unit of time.
- We can select the first week of stock prices in 2018 using the `first()` method and an offset of `1W`:

```
>>> fb.first('1W')
```

Time-based selection and filtering

- January 1, 2018 was a holiday, meaning that the market was closed.
- It was also a Monday, so the week here is only four days long:

	open	high	low	close	volume	trading_volume
date						
2018-01-02	177.68	181.58	177.5500	181.42	18151903	low
2018-01-03	181.88	184.78	181.3300	184.67	16886563	low
2018-01-04	184.90	186.21	184.0996	184.33	13880896	low
2018-01-05	185.59	186.90	184.9300	186.85	13574535	low

Time-based selection and filtering

- We can perform a similar operation for the most recent dates as well.
- Selecting the last week in the data is as simple as switching the `first()` method with the `last()` method:

```
>>> fb.last('1W')
```

Time-based selection and filtering

- Since December 31, 2018 was a Monday, the last week only consists of one day:

	open	high	low	close	volume	trading_volume
date						
2018-12-31	134.45	134.64	129.95	131.09	24625308	low

Time-based selection and filtering

- When working with daily stock data, we only have data for the dates the stock market was open.
- Suppose that we reindexed the data to include rows for each day of the year:

```
>>> fb_reindexed = fb.reindex(  
...     pd.date_range('2018-01-01', '2018-12-31', freq='D')  
... )
```

Time-based selection and filtering

- Here, we will also use the `squeeze()` method to turn the 1-row DataFrame object resulting from the call to `first('1D').isna()` into a Series object so that calling `all()` yields a single value:

```
>>> fb_reindexed.first('1D').isna().squeeze().all()
```

```
True
```

Time-based selection and filtering

- For the first quarter of 2018, the first day of trading was January 2nd and the last was March 29th:

```
>>> fb_reindexed.loc['2018-Q1'].first_valid_index()  
Timestamp('2018-01-02 00:00:00', freq='D')  
>>> fb_reindexed.loc['2018-Q1'].last_valid_index()  
Timestamp('2018-03-29 00:00:00', freq='D')
```

Time-based selection and filtering

- Therefore, if we wanted to see how Facebook performed on the last day in each month, we could use `asof()`, and avoid having to first check if the market was open that day:

```
>>> fb_reindexed.asof('2018-03-31')
open           155.15
high          161.42
low           154.14
close          159.79
volume      59434293.00
trading_volume    low
Name: 2018-03-31 00:00:00, dtype: object
```

Time-based selection and filtering

```
>>> stock_data_per_minute = pd.read_csv(  
...     'data/fb_week_of_may_20_per_minute.csv',  
...     index_col='date', parse_dates=True,  
...     date_parser=lambda x: \  
...         pd.to_datetime(x, format='%Y-%m-%d %H-%M')  
... )  
>>> stock_data_per_minute.head()
```

Time-based selection and filtering

- We have the OHLC data per minute, along with the volume traded per minute:

date	open	high	low	close	volume
2019-05-20 09:30:00	181.6200	181.6200	181.6200	181.6200	159049.0
2019-05-20 09:31:00	182.6100	182.6100	182.6100	182.6100	468017.0
2019-05-20 09:32:00	182.7458	182.7458	182.7458	182.7458	97258.0
2019-05-20 09:33:00	182.9500	182.9500	182.9500	182.9500	43961.0
2019-05-20 09:34:00	183.0600	183.0600	183.0600	183.0600	79562.0

Time-based selection and filtering

- The high and low will be the maximum and minimum of their respective columns per day.
- Volume traded will be the daily sum:

```
>>> stock_data_per_minute.groupby(pd.Grouper(freq='1D')).agg({  
...     'open': 'first',  
...     'high': 'max',  
...     'low': 'min',  
...     'close': 'last',  
...     'volume': 'sum'  
... })
```

Time-based selection and filtering

- This rolls the data up to a daily frequency:

	open	high	low	close	volume
date					
2019-05-20	181.62	184.1800	181.6200	182.72	10044838.0
2019-05-21	184.53	185.5800	183.9700	184.82	7198405.0
2019-05-22	184.81	186.5603	184.0120	185.32	8412433.0
2019-05-23	182.50	183.7300	179.7559	180.87	12479171.0
2019-05-24	182.33	183.5227	181.0400	181.06	7686030.0

Time-based selection and filtering

- The next two methods we will discuss help us select data based on the time part of the datetime.
- The `at_time()` method allows us to isolate rows where the time part of the datetime is the time we specify.
- By running `at_time('9:30')`, we can grab all the market open prices (the stock market opens at 9:30 AM):

```
>>> stock_data_per_minute.at_time('9:30')
```

Time-based selection and filtering

- This tells us what the stock data looked like at the opening bell each day:

	open	high	low	close	volume
date					
2019-05-20 09:30:00	181.62	181.62	181.62	181.62	159049.0
2019-05-21 09:30:00	184.53	184.53	184.53	184.53	58171.0
2019-05-22 09:30:00	184.81	184.81	184.81	184.81	41585.0
2019-05-23 09:30:00	182.50	182.50	182.50	182.50	121930.0
2019-05-24 09:30:00	182.33	182.33	182.33	182.33	52681.0

Time-based selection and filtering

- Let's grab all the rows within the last two minutes of trading each day (15:59 - 16:00):

```
>>> stock_data_per_minute.between_time('15:59', '16:00')
```

Time-based selection and filtering

- It looks like the last minute (16:00) has significantly more volume traded each day compared to the previous minute (15:59).
- Perhaps people rush to make trades before close:

	open	high	low	close	volume
date					
2019-05-20 15:59:00	182.915	182.915	182.915	182.915	134569.0
2019-05-20 16:00:00	182.720	182.720	182.720	182.720	1113672.0
2019-05-21 15:59:00	184.840	184.840	184.840	184.840	61606.0
2019-05-21 16:00:00	184.820	184.820	184.820	184.820	801080.0
2019-05-22 15:59:00	185.290	185.290	185.290	185.290	96099.0
2019-05-22 16:00:00	185.320	185.320	185.320	185.320	1220993.0
2019-05-23 15:59:00	180.720	180.720	180.720	180.720	109648.0
2019-05-23 16:00:00	180.870	180.870	180.870	180.870	1329217.0
2019-05-24 15:59:00	181.070	181.070	181.070	181.070	52994.0
2019-05-24 16:00:00	181.060	181.060	181.060	181.060	764906.0

Time-based selection and filtering

- The excluded groups are times that aren't in the time range we want:

```
>>> shares_traded_in_first_30_min = stock_data_per_minute\  
...     .between_time('9:30', '10:00')\  
...     .groupby(pd.Grouper(freq='1D'))\  
...     .filter(lambda x: (x.volume > 0).all())\  
...     .volume.mean()  
>>> shares_traded_in_last_30_min = stock_data_per_minute\  
...     .between_time('15:30', '16:00')\  
...     .groupby(pd.Grouper(freq='1D'))\  
...     .filter(lambda x: (x.volume > 0).all())\  
...     .volume.mean()
```

Time-based selection and filtering

- For the week in question, there were 18,593 more trades on average around the opening time than the closing time:

```
>>> shares_traded_in_first_30_min \  
... - shares_traded_in_last_30_min  
18592.967741935485
```

Shifting for lagged data

- From this new column, we can calculate the price change due to after-hours trading (after the market close one day right up to the market open the following day):

```
>>> fb.assign(  
...     prior_close=lambda x: x.close.shift(),  
...     after_hours_change_in_price=lambda x: \  
...         x.open - x.prior_close,  
...     abs_change=lambda x: \  
...         x.after_hours_change_in_price.abs()  
... ).nlargest(5, 'abs_change')
```

Shifting for lagged data

- This gives us the days that were most affected by after-hours trading:

	open	high	low	close	volume	trading_volume	prior_close	after_hours_change_in_price	abs_change
date									
2018-07-26	174.89	180.13	173.75	176.26	169803668	high	217.50	-42.61	42.61
2018-04-26	173.22	176.27	170.80	174.16	77556934	med	159.69	13.53	13.53
2018-01-12	178.06	181.48	177.40	179.37	77551299	med	187.77	-9.71	9.71
2018-10-31	155.00	156.40	148.96	151.79	60101251	low	146.22	8.78	8.78
2018-03-19	177.01	177.17	170.06	172.56	88140060	med	185.09	-8.08	8.08

Differenced data

- We've already discussed creating lagged data with the `shift()` method.
- However, often, we are interested in how the values change from one time period to the next.
- For this, pandas has the `diff()` method.
- By default, this will calculate the change from time period $t-1$ to time period t :

$$x_{diff} = x_t - x_{t-1}$$

Differenced data

- Note that this is equivalent to subtracting the result of `shift()` from the original data:

```
>>> (fb.drop(columns='trading_volume')
... - fb.drop(columns='trading_volume').shift()
... ).equals(fb.drop(columns='trading_volume').diff())
True
```

Differenced data

- We can use `diff()` to easily calculate the day-over-day change in the Facebook stock data:

```
>>> fb.drop(columns='trading_volume').diff().head()
```

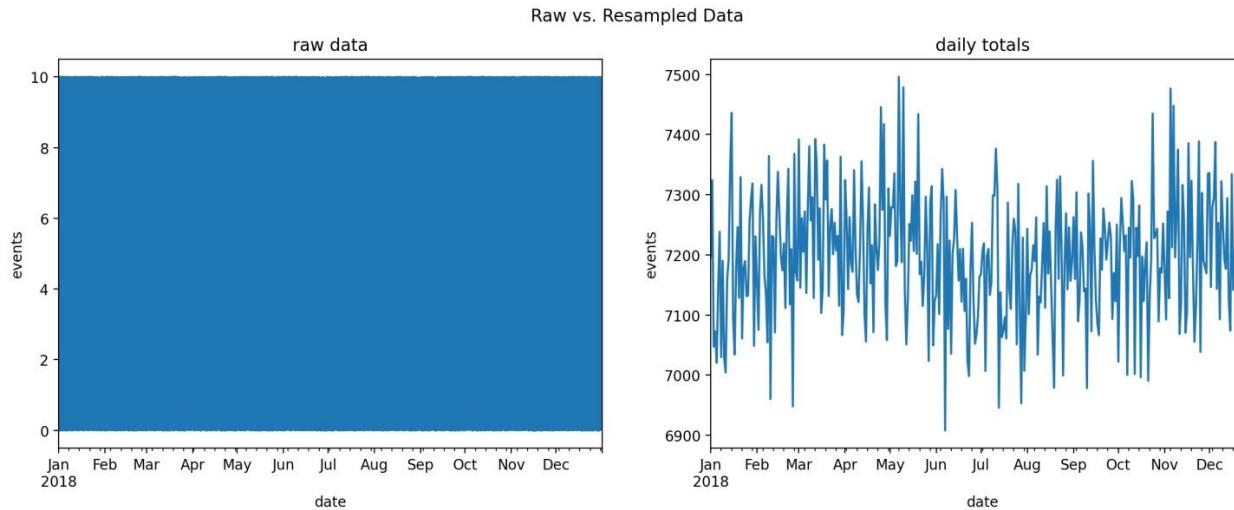
Differenced data

- For the first few trading days of the year, we can see that the stock price increased, and that the volume traded decreased daily:

	open	high	low	close	volume
date					
2018-01-02	NaN	NaN	NaN	NaN	NaN
2018-01-03	4.20	3.20	3.7800	3.25	-1265340.0
2018-01-04	3.02	1.43	2.7696	-0.34	-3005667.0
2018-01-05	0.69	0.69	0.8304	2.52	-306361.0
2018-01-08	1.61	2.00	1.4000	1.43	4420191.0

Resampling

- Therefore, we will need to aggregate the data to a less granular frequency:



Resampling

- For example, we can resample this minute-by-minute data to a daily frequency and specify how to aggregate each column:

```
>>> stock_data_per_minute.resample('1D').agg({  
...     'open': 'first',  
...     'high': 'max',  
...     'low': 'min',  
...     'close': 'last',  
...     'volume': 'sum'  
... })
```

Resampling

- This is equivalent to the result we got back in the Time-based selection and filtering section:

	open	high	low	close	volume
date					
2019-05-20	181.62	184.1800	181.6200	182.72	10044838.0
2019-05-21	184.53	185.5800	183.9700	184.82	7198405.0
2019-05-22	184.81	186.5603	184.0120	185.32	8412433.0
2019-05-23	182.50	183.7300	179.7559	180.87	12479171.0
2019-05-24	182.33	183.5227	181.0400	181.06	7686030.0

Resampling

- We can resample to any frequency supported by pandas (more information can be found in the documentation at http://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html).
- Let's resample the daily Facebook stock data to the quarterly average:

```
>>> fb.resample('Q').mean()
```

Resampling

- This gives us the average quarterly performance of the stock.
The fourth quarter of 2018 was clearly troublesome:

	open	high	low	close	volume
date					
2018-03-31	179.472295	181.794659	177.040428	179.551148	3.292640e+07
2018-06-30	180.373770	182.277689	178.595964	180.704687	2.405532e+07
2018-09-30	180.812130	182.890886	178.955229	181.028492	2.701982e+07
2018-12-31	145.272460	147.620121	142.718943	144.868730	2.697433e+07

Resampling

- To look further into this, we can use the apply() method to look at the difference between how the quarter began and how it ended.
- We will also need the first() and last() methods from the Time-based selection and filtering section:

```
>>> fb.drop(columns='trading_volume').resample('Q').apply(  
...     lambda x: x.last('1D').values - x.first('1D').values  
... )
```

Resampling

- Facebook's stock price declined in all but the second quarter:

	open	high	low	close	volume
date					
2018-03-31	-22.53	-20.1600	-23.410	-21.63	41282390
2018-06-30	39.51	38.3997	39.844	38.93	-20984389
2018-09-30	-25.04	-28.6600	-29.660	-32.90	20304060
2018-12-31	-28.58	-31.2400	-31.310	-31.35	-1782369

Resampling

- Consider the melted minute-by-minute stock data in melted_stock_data.csv:

```
>>> melted_stock_data = pd.read_csv(  
...     'data/melted_stock_data.csv',  
...     index_col='date', parse_dates=True  
... )  
>>> melted_stock_data.head()
```

Resampling

- The OHLC format makes it easy to analyze the stock data, but a single column is trickier:

date	price
2019-05-20 09:30:00	181.6200
2019-05-20 09:31:00	182.6100
2019-05-20 09:32:00	182.7458
2019-05-20 09:33:00	182.9500
2019-05-20 09:34:00	183.0600

Resampling

- The Resampler object we get back after calling resample() has an ohlc() method, which we can use to retrieve the OHLC data we are used to seeing:

```
>>> melted_stock_data.resample('1D').ohlc()['price']
```

Resampling

- Since the column in the original data was called price, we select it after calling ohlc(), which is pivoting our data.
- Otherwise, we will have a hierarchical index in the columns:

	open	high	low	close
date				
2019-05-20	181.62	184.1800	181.6200	182.72
2019-05-21	184.53	185.5800	183.9700	184.82
2019-05-22	184.81	186.5603	184.0120	185.32
2019-05-23	182.50	183.7300	179.7559	180.87
2019-05-24	182.33	183.5227	181.0400	181.06

Resampling

- In the previous examples, we downsampled to reduce the granularity of the data; however, we can also upsample to increase the granularity of the data.
- We can even call `asfreq()` after to not aggregate the result:

```
>>> fb.resample('6H').asfreq().head()
```

Resampling

- Note that when we resample at a granularity that's finer than the data we have, it will introduce NaN values:

	open	high	low	close	volume	trading_volume
date						
2018-01-02 00:00:00	177.68	181.58	177.55	181.42	18151903.0	low
2018-01-02 06:00:00	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-02 12:00:00	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-02 18:00:00	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-03 00:00:00	181.88	184.78	181.33	184.67	16886563.0	low

Merging time series

- Let's read these tables from the database:

```
>>> import sqlite3
>>> with sqlite3.connect('data/stocks.db') as connection:
...     fb_prices = pd.read_sql(
...         'SELECT * FROM fb_prices', connection,
...         index_col='date', parse_dates=['date']
...     )
...     aapl_prices = pd.read_sql(
...         'SELECT * FROM aapl_prices', connection,
...         index_col='date', parse_dates=['date']
...     )
```

Merging time series

- The Facebook data is at the minute granularity; however, we have (fictitious) seconds for the Apple data:

```
>>> fb_prices.index.second.unique()  
Int64Index([0], dtype='int64', name='date')  
>>> aapl_prices.index.second.unique()  
Int64Index([ 0, 52, ..., 37, 28], dtype='int64', name='date')
```

Merging time series

- Since the times are on the index, we pass in `left_index` and `right_index`, just like we did with `merge()`:

```
>>> pd.merge_asof(  
...     fb_prices, aapl_prices,  
...     left_index=True, right_index=True,  
...     # merge with nearest minute  
...     direction='nearest',  
...     tolerance=pd.Timedelta(30, unit='s'))  
... ).head()
```

Merging time series

- We get a null value for 9:31 because the entry for Apple at 9:31 was 9:31:52, which gets placed at 9:32 when using nearest:

	date	FB	AAPL
	2019-05-20 09:30:00	181.6200	183.5200
	2019-05-20 09:31:00	182.6100	NaN
	2019-05-20 09:32:00	182.7458	182.8710
	2019-05-20 09:33:00	182.9500	182.5000
	2019-05-20 09:34:00	183.0600	182.1067

Merging time series

- If we don't want the behavior of a left join, we can use the `pd.merge_ordered()` function instead.
- This will allow us to specify our join type, which will be 'outer' by default.
- We will have to reset our index to be able to join on the datetimes, however:

```
>>> pd.merge_ordered(  
...     fb_prices.reset_index(), aapl_prices.reset_index()  
... ).set_index('date').head()
```

Merging time series

- This strategy will give us null values whenever the times don't match exactly, but it will at least sort them for us:

	FB	AAPL
date		
2019-05-20 09:30:00	181.6200	183.520
2019-05-20 09:31:00	182.6100	NaN
2019-05-20 09:31:52	NaN	182.871
2019-05-20 09:32:00	182.7458	NaN
2019-05-20 09:32:36	NaN	182.500



Summary

- In this lesson, we discussed how to join dataframes, how to determine the data we will lose for each type of join using set operations, and how to query dataframes as we would a database.
- We then went over some more involved transformations on our columns, such as binning and ranking, and how to do so efficiently with the `apply()` method.
- We also learned the importance of vectorized operations in writing efficient pandas code.

"Complete Exercises"

"Complete Lab 9"

DAY 5



10: Visualizing Data with Pandas and Matplotlib



Visualizing Data with Pandas and Matplotlib

In this lesson, we will cover the following topics:

- An introduction to matplotlib
- Plotting with pandas
- The pandas.plotting module

lesson materials

- Throughout this lesson, we will be working through three notebooks.
- These are numbered in the order they will be used—one for each of the main sections of this lesson.

An introduction to matplotlib

- The plotting capabilities in pandas and seaborn are powered by matplotlib: both of these packages provide wrappers around the lower-level functionality in matplotlib.
- Consequently, we have many visualization options at our fingertips with minimal code to write; however, this comes at a price: reduced flexibility in what we can create.

The basics

- Rather than importing the whole matplotlib package, we will only import the pyplot module using the dot (.) notation; this reduces the amount of typing we need to do in order to access what we need, and we don't take up more space in memory with code we won't use.
- Note that pyplot is traditionally aliased as plt:
`import matplotlib.pyplot as plt`

The basics

- Let's create our first plot in the 1-introducing_matplotlib.ipynb notebook, using the Facebook stock prices data from the fb_stock_prices_2018.csv file in the repository for this lesson.
- First, we need to import pyplot and pandas (in this example, we will use plt.show(), so we don't need to run the magic here):

```
>>> import matplotlib.pyplot as plt  
>>> import pandas as pd
```

The basics

- Next, we read in the CSV file and specify the index as the date column, since we know what the data looks like from previous lessons:

```
>>> fb = pd.read_csv(  
...     'data/fb_stock_prices_2018.csv',  
...     index_col='date',  
...     parse_dates=True  
... )
```

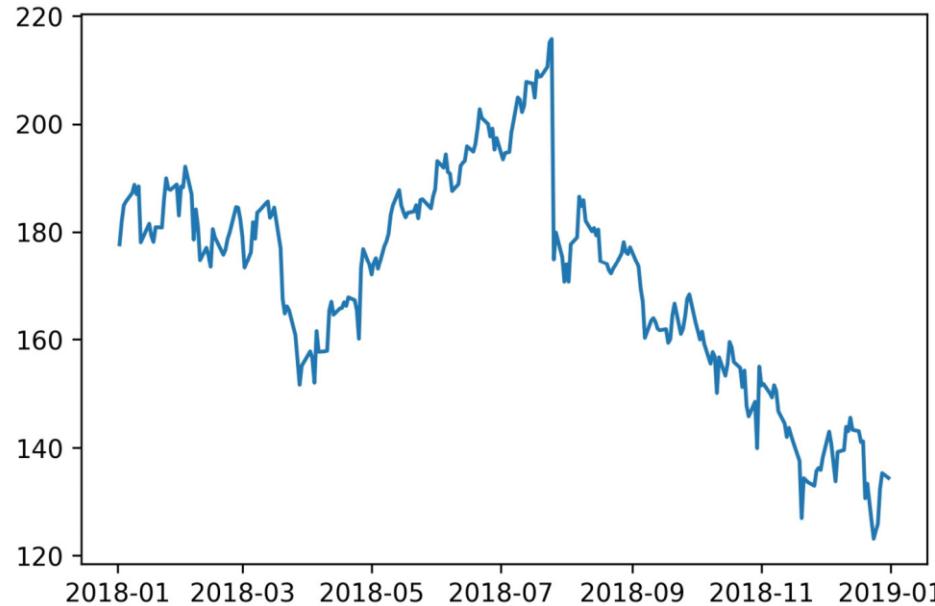
The basics

- To understand how Facebook's stock has evolved over time, we can create a line plot of the daily opening price.
- For this task, we will use the plt.plot() function, providing the data to be used on the x-axis and y-axis, respectively.
- We will then follow up with a call to plt.show() to display it:

```
>>> plt.plot(fb.index, fb.open)  
>>> plt.show()
```

The basics

- The result is the following plot:



The basics

- The following code gives the same output as the preceding block:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
>>> plt.plot(fb.index, fb.open)
```

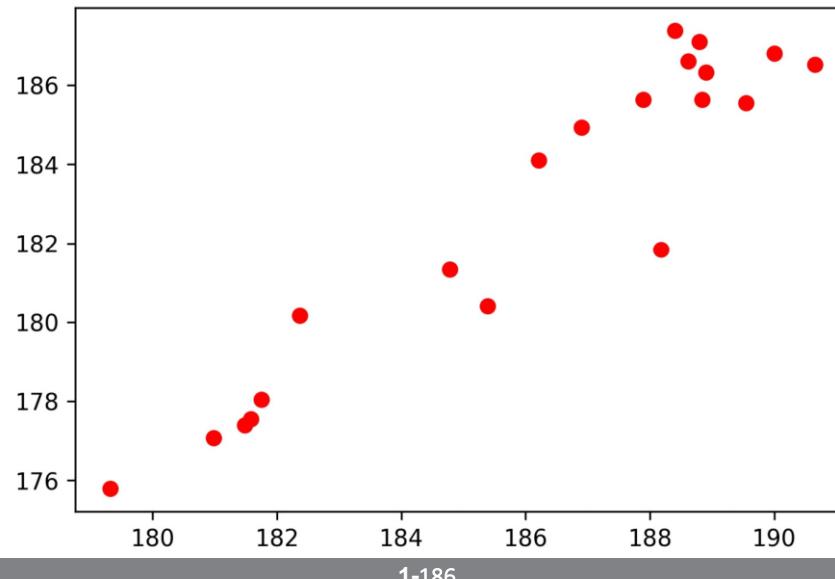
The basics

- Notice that we can pass our dataframe in the data argument and then use the string names for the columns, instead of passing the series as x and y:

```
>>> plt.plot('high', 'low', 'or', data=fb.head(20))
```

The basics

- This is true for the most part, but be careful of the scale that was generated automatically—the x-axis and the y-axis don't line up perfectly:



The basics

- The following table shows some examples of how to formulate a format string for a variety of plot styles; the complete list of options can be found in the Notes section in the documentation at https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html:

Marker	Linestyle	Color	Format String	Result
	-	b	-b	blue solid line
.		k	.k	black points
	--	r	--r	red dashed line
o	-	g	o-g	green solid line with circles
	:	m	:m	magenta dotted line
x	-.	c	x-.c	cyan dot-dashed line with x's

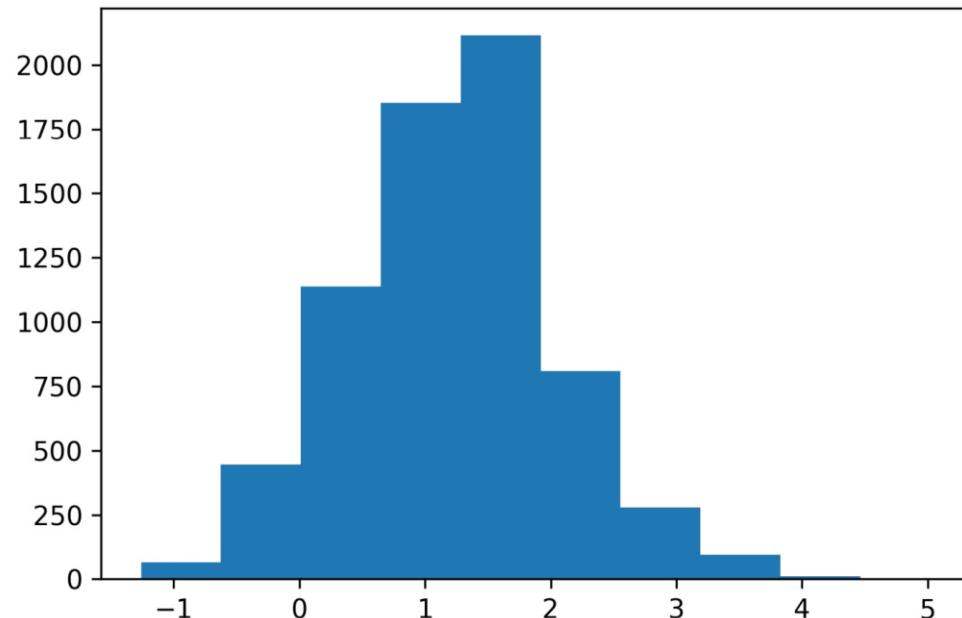
The basics

- To create histograms with matplotlib, we use the `hist()` function instead.
- Let's make a histogram of the earthquake magnitudes in the `earthquakes.csv` file, using those measured with the `ml` magnitude type:

```
>>> quakes = pd.read_csv('data/earthquakes.csv')
>>> plt.hist(quakes.query('magType == "ml").mag)
```

The basics

- The resulting histogram gives us an idea of the range of earthquake magnitudes we can expect using the ml measurement technique:



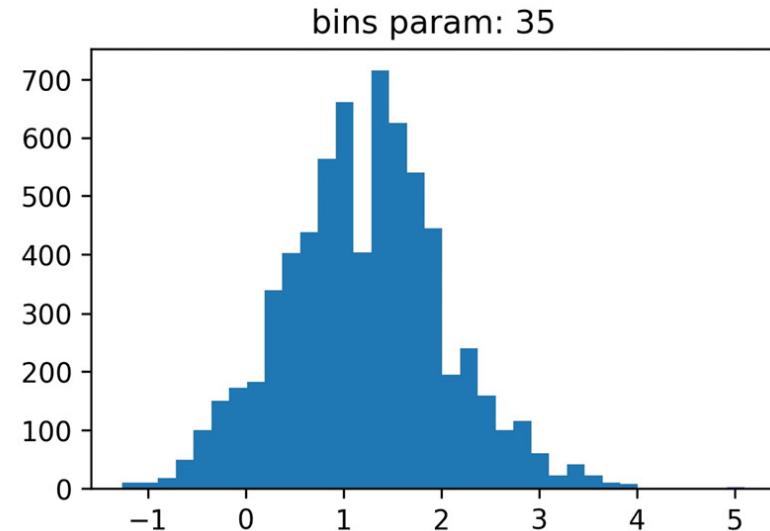
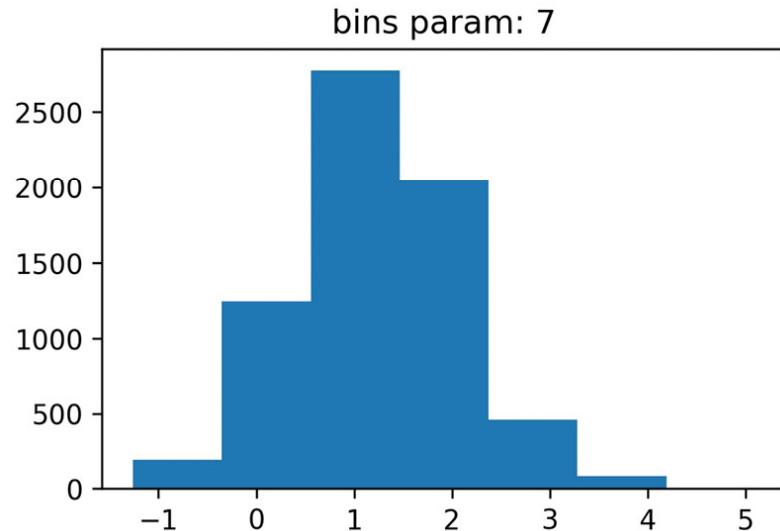
The basics

- For example, if we make two histograms for this data using different numbers of bins, the distributions look different:

```
>>> x = quakes.query('magType == "ml"]').mag
>>> fig, axes = plt.subplots(1, 2, figsize=(10, 3))
>>> for ax, bins in zip(axes, [7, 35]):
...     ax.hist(x, bins=bins)
...     ax.set_title(f'bins param: {bins}')
```

The basics

- Notice how the distribution appears unimodal in the left subplot, but seems bimodal in the right subplot:



The basics

There are a couple of additional things to note from this example, which we will address in the next section on plot components:

- We can make subplots.
- Plotting functions in pyplot can also be used as methods of matplotlib objects, such as Figure and Axes objects.

Plot components

- We use the plt.figure() function to create Figure objects; these will have zero Axes objects until a plot is added:

```
>>> fig = plt.figure()  
<Figure size 432x288 with 0 Axes>
```

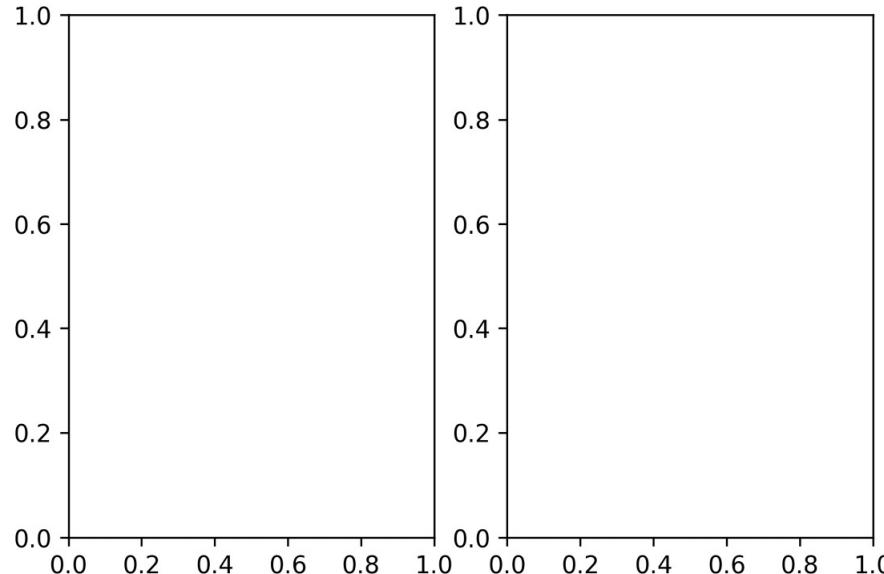
Plot components

- Here, we will specify an arrangement of one row and two columns; this returns a (Figure, Axes) tuple, which we can unpack:

```
>>> fig, axes = plt.subplots(1, 2)
```

Plot components

- When using the %matplotlib inline magic command, we will see the figure that was created:



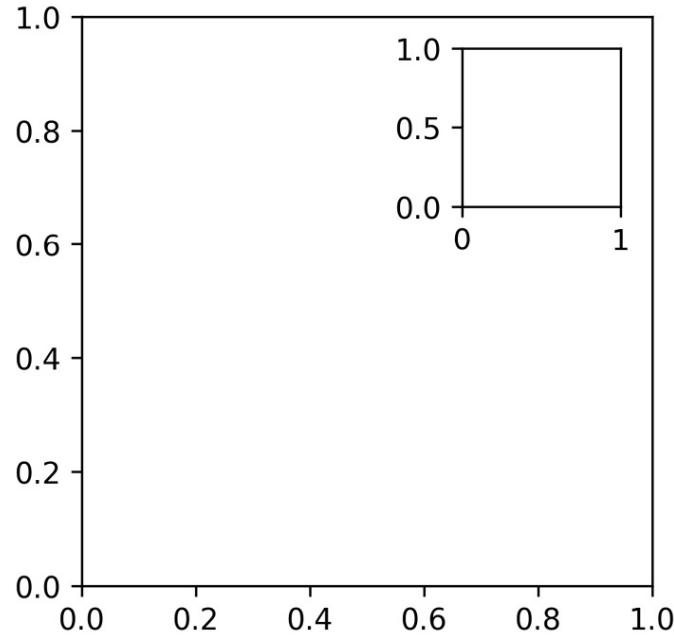
Plot components

- The alternative to using plt.subplots() would be to use the add_axes() method on the Figure object that we get after running plt.figure().
- The add_axes() method takes a list in the form of [left, bottom, width, height] as proportions of the figure dimensions, representing the area in the figure this subplot should occupy:

```
>>> fig = plt.figure(figsize=(3, 3))
>>> outside = fig.add_axes([0.1, 0.1, 0.9, 0.9])
>>> inside = fig.add_axes([0.7, 0.7, 0.25, 0.25])
```

Plot components

- This enables the creation of plots inside of plots:



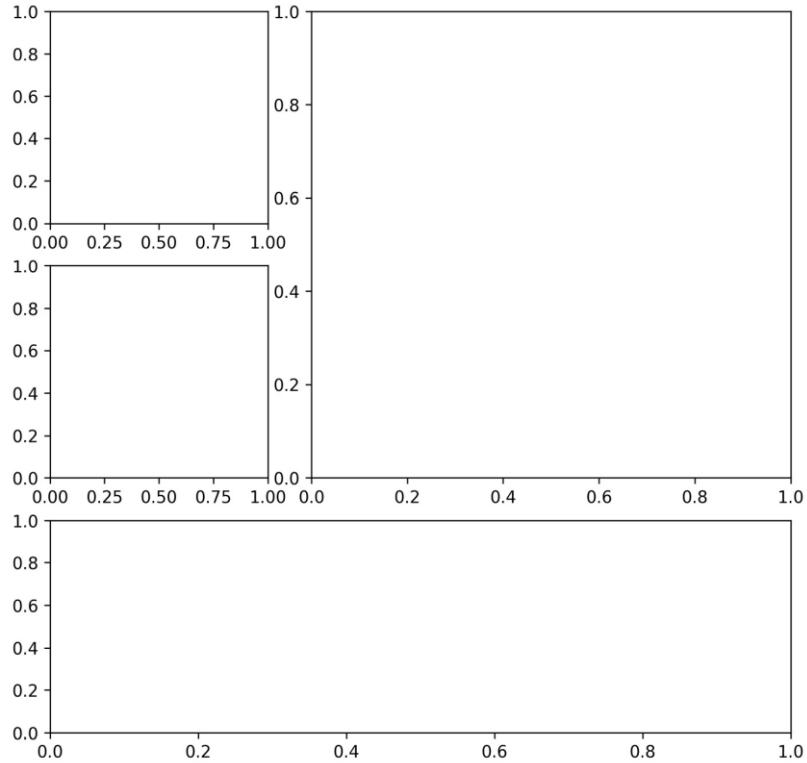
Plot components

- Then, we can run `add_subplot()`, passing in the area(s) from the grid that the given subplot should occupy:

```
>>> fig = plt.figure(figsize=(8, 8))
>>> gs = fig.add_gridspec(3, 3)
>>> top_left = fig.add_subplot(gs[0, 0])
>>> mid_left = fig.add_subplot(gs[1, 0])
>>> top_right = fig.add_subplot(gs[:2, 1:])
>>> bottom = fig.add_subplot(gs[2,:])
```

Plot components

- This results in the following layout:



Plot components

- In the previous section, we discussed how to save visualizations using plt.savefig() but we also can use the savefig() method on Figure objects:

```
>>> fig.savefig('empty.png')
```

Plot components

- If we don't pass in anything, it will close the last Figure object; however, we can pass in a specific Figure object to close only that one or 'all' to close all of the Figure objects we have open:

```
>>> plt.close('all')
```

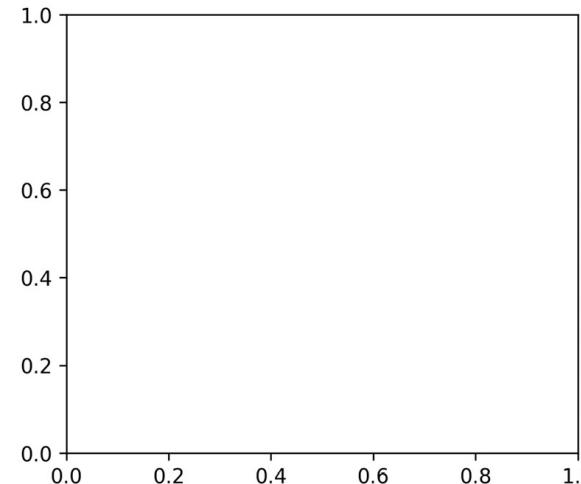
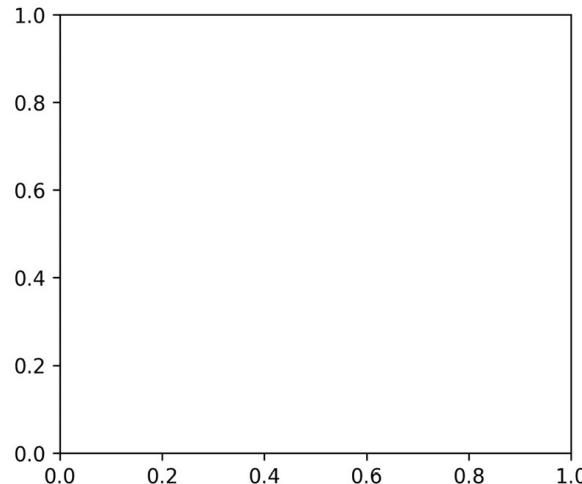
Additional options

- The plot() method we will see with pandas also accepts the figsize parameter, so bear this in mind:

```
>>> fig = plt.figure(figsize=(10, 4))
<Figure size 720x288 with 0 Axes>
>>> fig, axes = plt.subplots(1, 2, figsize=(10, 4))
```

Additional options

- Notice that these subplots are more square-shaped than the subplots in Figure when we didn't specify figsize:



Additional options

- Since there are many options in this dictionary (over 300 at the time of writing), let's randomly select a few of them to get an idea of what is available:

```
>>> import random
>>> import matplotlib as mpl
>>> rcparams_list = list(mpl.rcParams.keys())
>>> random.seed(20) # make this repeatable
>>> random.shuffle(rcparams_list)
>>> sorted(rcparams_list[:20])
['axes.axisbelow',
 'axes.formatter.limits',
 'boxplot.vertical',
 'contour.corner_mask',
 'date.autoformatter.month',
 'legend.labelspacing',
 'lines.dashed_pattern',
 'lines.dotted_pattern',
 'lines.scale_dashes',
 'lines.solid_capstyle',
 'lines.solid_joinstyle',
 'mathtext.tt',
 'patch.linewidth',
 'pdf.fonttype',
 'savefig.jpeg_quality',
 'svg.fonttype',
 'text.latex.preview',
 'toolbar',
 'ytick.labelright',
 'ytick.minor.size']
```

Additional options

- As you can see, there are many options we can tinker with here.
- Let's check what the current default value for figsize is:

```
>>> mpl.rcParams['figure.figsize']
[6.0, 4.0]
```

Additional options

- To change this for our current session, simply set it equal to a new value:

```
>>> mpl.rcParams['figure.figsize'] = (300, 10)
>>> mpl.rcParams['figure.figsize']
[300.0, 10.0]
```

Additional options

- The default value for figsize is actually different than what we had previously; this is because %matplotlib inline sets different values for a few of the plot-related parameters when it is first run:

```
>>> mpl.rcParams()  
>>> mpl.rcParams['figure.figsize'][  
[6.8, 4.8]
```

Additional options

- As we did previously, we can use plt.rcdefaults() to reset the defaults:

```
# change `figsize` default to (20, 20)
>>> plt.rc('figure', figsize=(20, 20))
>>> plt.rcdefaults() # reset the default
```

Plotting with pandas

- Under the hood, pandas is making several calls to matplotlib to produce our plot.
- Some of the most frequently used arguments to the `plot()` method include the following:

Parameter	Purpose	Data Type
<code>kind</code>	Determines the plot type	String
<code>x / y</code>	Column(s) to plot on the x-axis/y-axis	String or list
<code>ax</code>	Draws the plot on the <code>Axes</code> object provided	<code>Axes</code>
<code>subplots</code>	Determines whether to make subplots	Boolean
<code>layout</code>	Specifies how to arrange the subplots	Tuple of <code>(rows, columns)</code>
<code>figsize</code>	Size to make the <code>Figure</code> object	Tuple of <code>(width, height)</code>
<code>title</code>	The title of the plot or subplots	String for the plot title or a list of strings for subplot titles
<code>legend</code>	Determines whether to show the legend	Boolean
<code>label</code>	What to call an item in the legend	String if a single column is being plotted; otherwise, a list of strings
<code>style</code>	<code>matplotlib</code> style strings for each item being plotted	String if a single column is being plotted; otherwise, a list of strings
<code>color</code>	The color to plot the item in	String or red, green, blue tuple if a single column is being plotted; otherwise, a list
<code>colormap</code>	The colormap to use	String or <code>matplotlib</code> colormap object
<code>logx / logy / loglog</code>	Determines whether to use a logarithmic scale for the x-axis, y-axis, or both	Boolean
<code>xticks / yticks</code>	Determines where to draw the ticks on the x-axis/y-axis	List of values
<code>xlim / ylim</code>	The axis limits for the x-axis/y-axis	Tuple of the form <code>(min, max)</code>
<code>rot</code>	The angle to write the tick labels at	Integer
<code>sharex / sharey</code>	Determines whether to have subplots share the x-axis/y-axis	Boolean
<code>fontsize</code>	Controls the size of the tick labels	Integer
<code>grid</code>	Turns on/off the grid lines	Boolean

Plotting with pandas

- Before we begin, we need to handle our imports for this section and read in the data we will be using (Facebook stock prices, earthquakes, and COVID-19 cases):

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
>>> quakes = pd.read_csv('data/earthquakes.csv')
>>> covid = pd.read_csv('data/covid19_cases.csv').assign(
...     date=lambda x: \
...         pd.to_datetime(x.dateRep, format='%d/%m/%Y')
... ).set_index('date').replace(
...     'United_States_of_America', 'USA'
... ).sort_index()['2020-01-18':'2020-09-18']
```

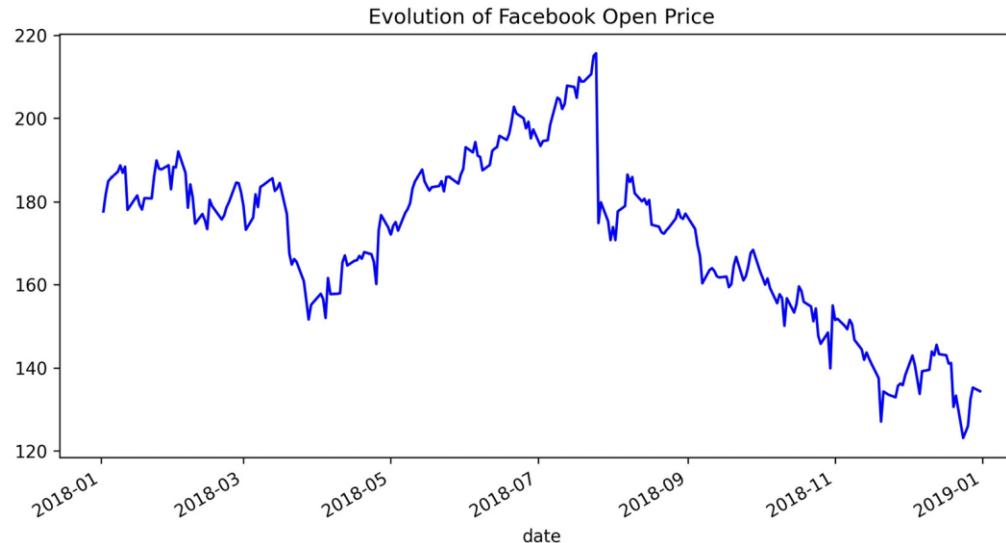
Evolution over time

- Additionally, notice that we can provide a format string to the style argument, just like we did with the matplotlib plots:

```
>>> fb.plot(  
...     kind='line', y='open', figsize=(10, 5), style='-b',  
...     legend=False, title='Evolution of Facebook Open Price'  
... )
```

Evolution over time

- This gives us a plot similar to what we achieved with matplotlib; however, in this single method call, we specified the figure size for this plot only, turned off the legend, and gave it a title:



Evolution over time

- As with matplotlib, we don't have to use the style format strings—instead, we can pass each component separately with its associated keyword.
- For example, the following code gives us the same result as the previous one:

```
fb.plot(  
    kind='line', y='open', figsize=(10, 5),  
    color='blue', linestyle='solid',  
    legend=False, title='Evolution of Facebook Open Price'  
)
```

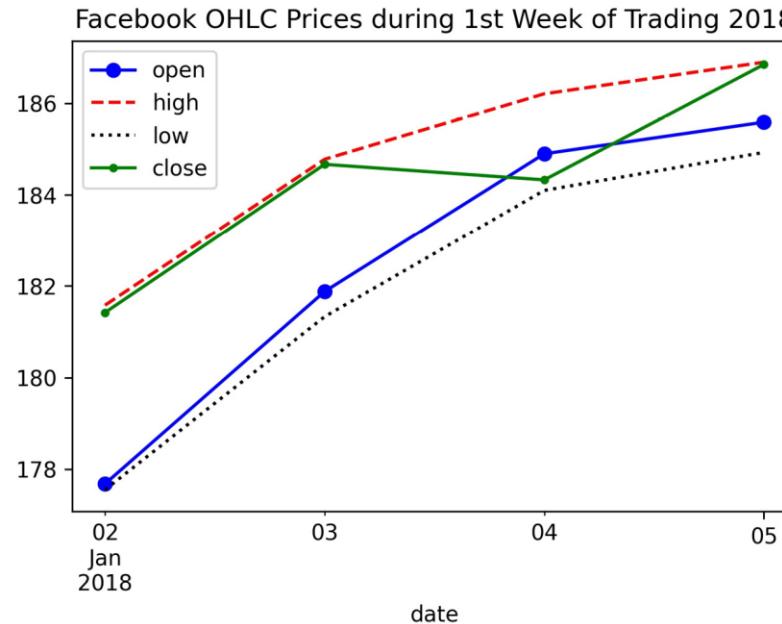
Evolution over time

- We aren't limited to plotting one line at a time with the `plot()` method; we can also pass in a list of columns to plot and style them individually.
- Note that we actually don't need to specify `kind='line'` because that is the default:

```
>>> fb.first('1W').plot(  
...     y=['open', 'high', 'low', 'close'],  
...     style=['o-b', '--r', ':k', '.-g'],  
...     title='Facebook OHLC Prices during '  
...             '1st Week of Trading 2018'  
... ).autoscale() # add space between data and axes
```

Evolution over time

- This results in the following plot, where each line is styled differently:



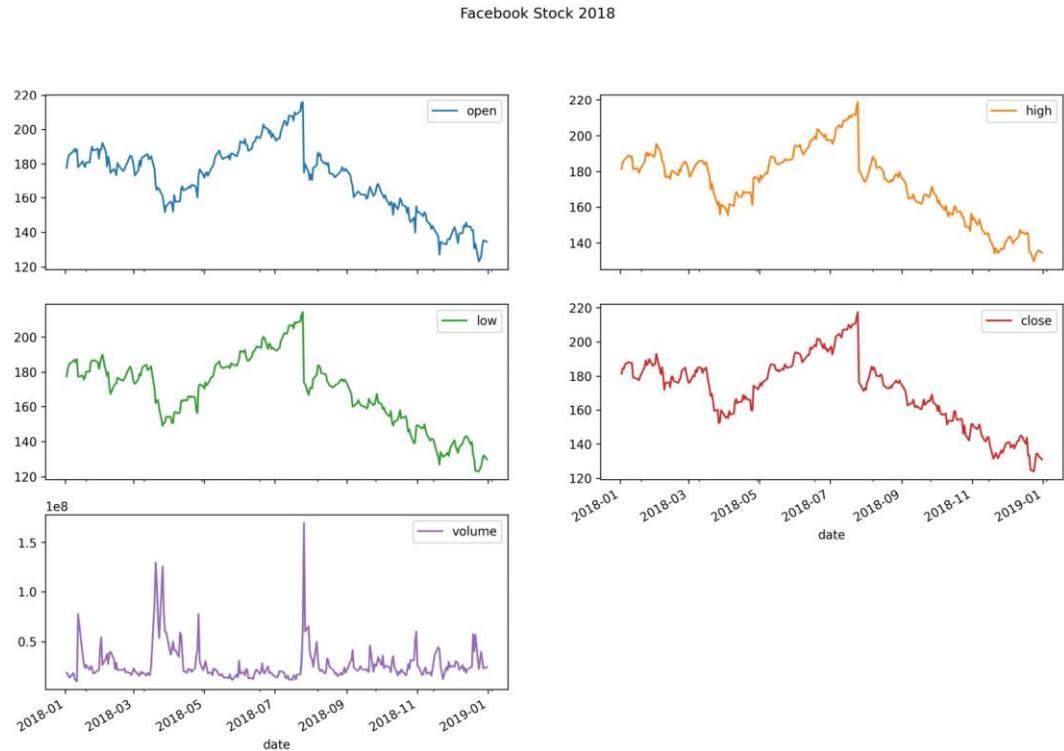
Evolution over time

- Let's visualize all the columns in the Facebook data as line plots:

```
>>> fb.plot(  
...     kind='line', subplots=True, layout=(3, 2),  
...     figsize=(15, 10), title='Facebook Stock 2018'  
... )
```

Evolution over time

- Using the layout argument, we told pandas how to arrange our subplots (three rows and two columns):



Evolution over time

- Since there is a lot of fluctuation in these values, we will plot the 7-day moving average of new cases using the `rolling()` method introduced in lesson 4, Aggregating Pandas DataFrames:

```
>>> new_cases_rolling_average = covid.pivot_table(  
...     index=covid.index,  
...     columns='countriesAndTerritories',  
...     values='cases'  
... ).rolling(7).mean()
```

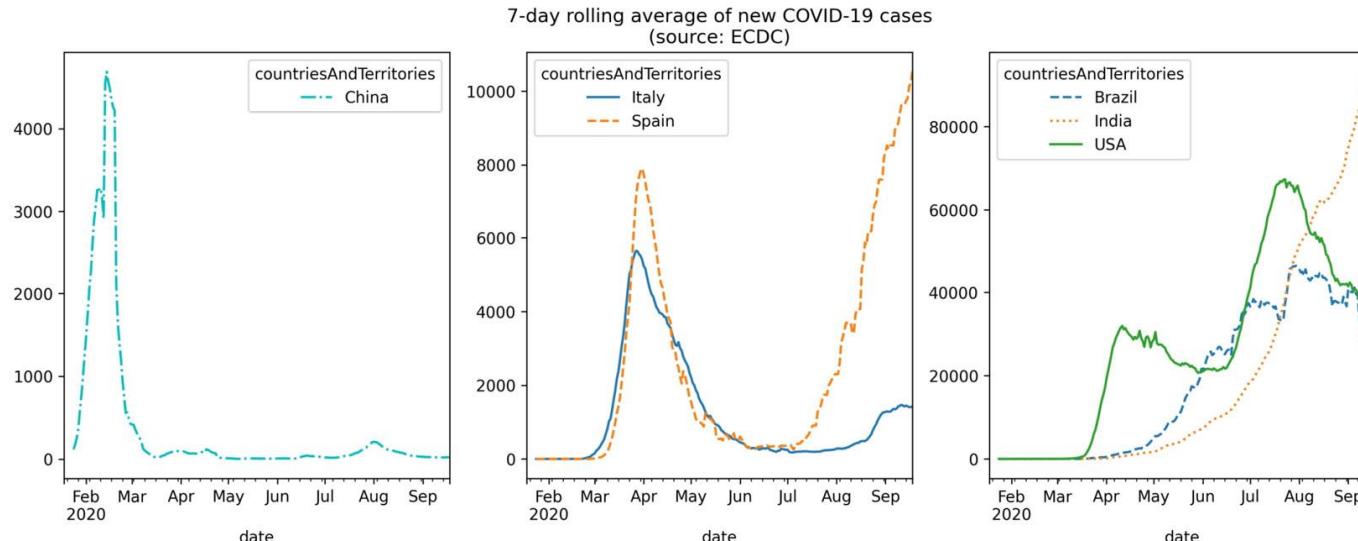
Evolution over time

- We will also use different line styles in order to distinguish between them in black and white:

```
>>> fig, axes = plt.subplots(1, 3, figsize=(15, 5))
>>> new_cases_rolling_average[['China']]\
...     .plot(ax=axes[0], style='-.c')
>>> new_cases_rolling_average[['Italy', 'Spain']].plot(
...     ax=axes[1], style=['-', '--'],
...     title='7-day rolling average of new '
...           'COVID-19 cases\n(source: ECDC)'
... )
>>> new_cases_rolling_average[['Brazil', 'India', 'USA']]\
...     .plot(ax=axes[2], style=[ '--', ':', '-'])
```

Evolution over time

- By directly using matplotlib to generate the Axes objects for each subplot, we gained a lot more flexibility in the resulting layout:



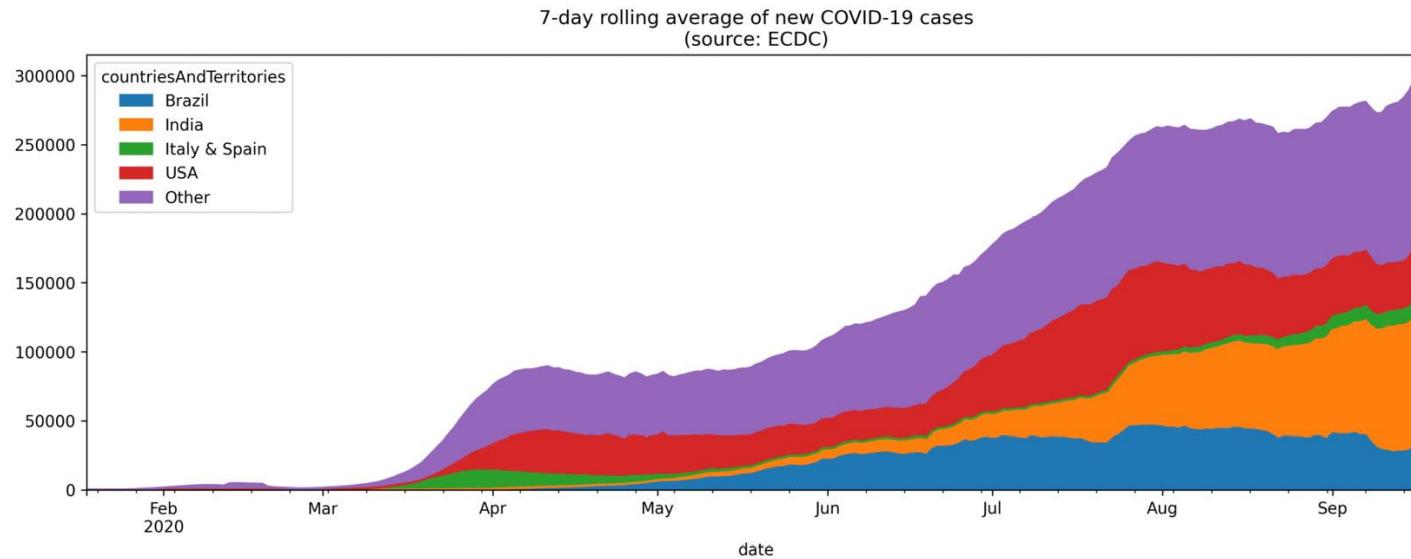
Evolution over time

- In the interest of readability, we will group Italy and Spain together and create another category for countries other than the USA, Brazil, and India:

```
>>> cols = [
...     col for col in new_cases_rolling_average.columns
...     if col not in [
...         'USA', 'Brazil', 'India', 'Italy & Spain'
...     ]
... ]
>>> new_cases_rolling_average.assign(
...     **{'Italy & Spain': lambda x: x.Italy + x.Spain}
... ).sort_index(axis=1).assign(
...     Other=lambda x: x[cols].sum(axis=1)
... ).drop(columns=cols).plot(
...     kind='area', figsize=(15, 5),
...     title='7-day rolling average of new '
...           'COVID-19 cases\n(source: ECDC)'
... )
```

Evolution over time

- This shows us that more than half of the daily new cases are in Brazil, India, Italy, Spain, and the USA combined:



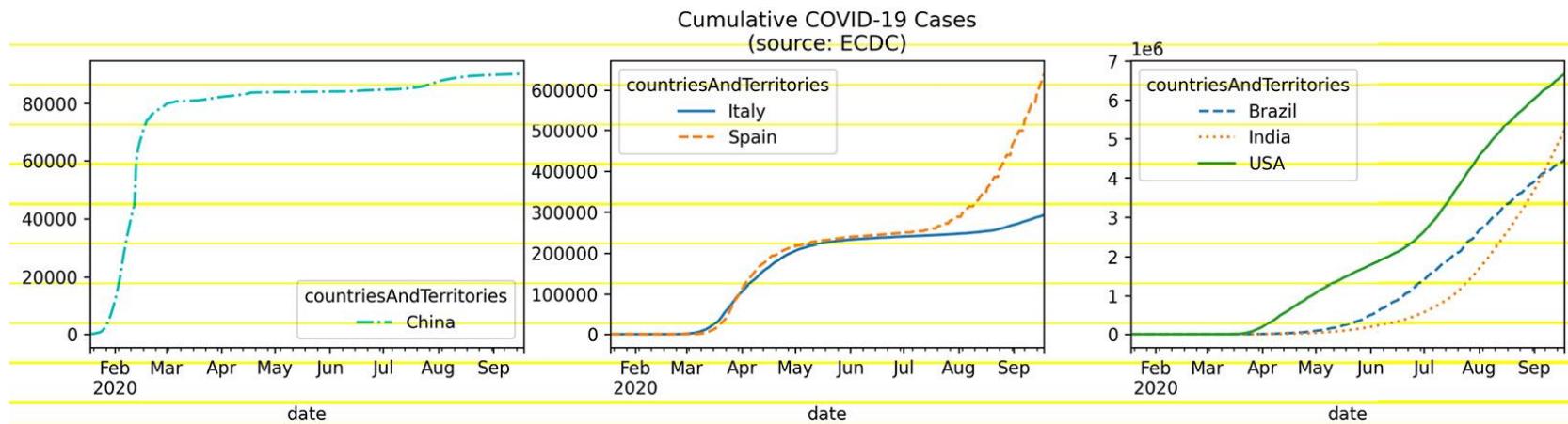
Evolution over time

- To calculate the cumulative sum over time, we group by the location (countriesAndTerritories) and the date, which is our index, so we use pd.Grouper(); this time, we will use groupby() and unstack() to pivot our data into wide format for the plot:

```
>>> fig, axes = plt.subplots(1, 3, figsize=(15, 3))
>>> cumulative_covid_cases = covid.groupby(
...     ['countriesAndTerritories', pd.Grouper(freq='1D')])
... ).cases.sum().unstack(0).apply('cumsum')
>>> cumulative_covid_cases[['China']]\
...     .plot(ax=axes[0], style='-.c')
>>> cumulative_covid_cases[['Italy', 'Spain']].plot(
...     ax=axes[1], style=['-', '--'],
...     title='Cumulative COVID-19 Cases\n(source: ECDC)'
... )
>>> cumulative_covid_cases[['Brazil', 'India', 'USA']]\
...     .plot(ax=axes[2], style='--', ':', '-')
```

Evolution over time

- Viewing the cumulative COVID-19 cases shows that while China and Italy appear to have COVID-19 cases under control, Spain, the USA, Brazil, and India are struggling:



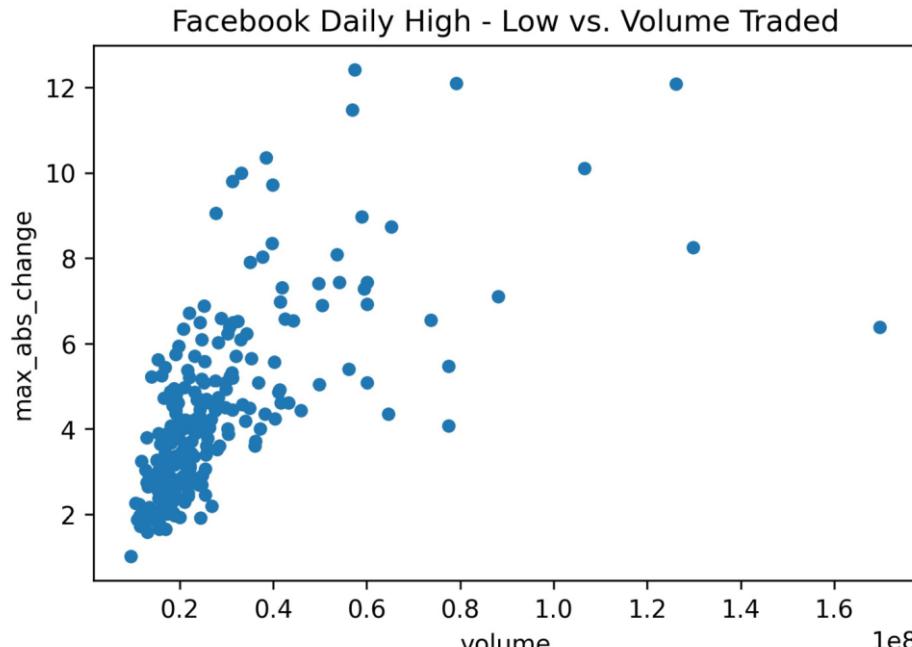
Relationships between variables

- We can use a scatter plot to visualize this relationship:

```
>>> fb.assign(  
...     max_abs_change=fb.high - fb.low  
... ).plot(  
...     kind='scatter', x='volume', y='max_abs_change',  
...     title='Facebook Daily High - Low vs. Volume Traded'  
... )
```

Relationships between variables

- There appears to be a relationship, but it does not seem linear:



Relationships between variables

- Let's try taking the logarithm (log) of the volume.

To do so, we have a couple of options:

- Create a new column that is the log of the volume using `np.log()`.
- Use a logarithmic scale for the x-axis by passing in `logx=True` to the `plot()` method or calling `plt.xscale('log')`.

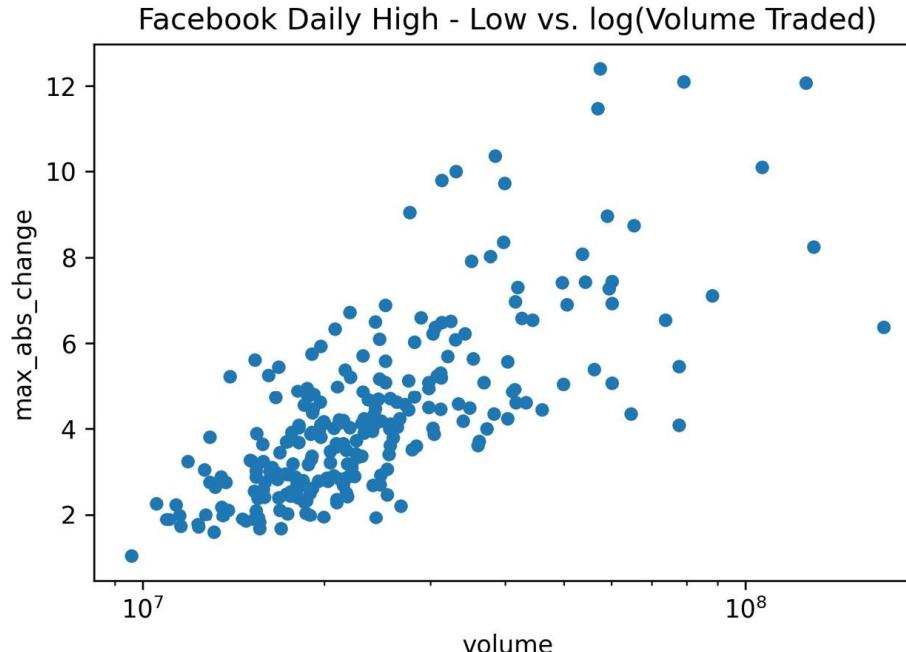
Relationships between variables

- In this case, it makes the most sense to simply change how we display our data, since we aren't going to use the new column:

```
>>> fb.assign(  
...     max_abs_change=fb.high - fb.low  
... ).plot(  
...     kind='scatter', x='volume', y='max_abs_change',  
...     title='Facebook Daily High - '  
...         'Low vs. log(Volume Traded)',  
...     logx=True  
... )
```

Relationships between variables

- After modifying the x-axis scale, we get the following scatter plot:



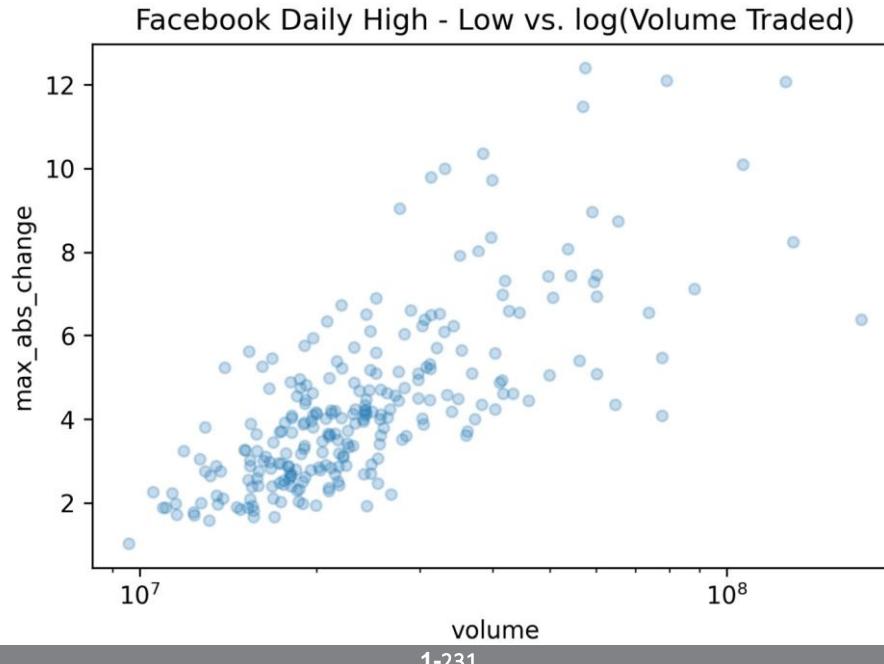
Relationships between variables

- By default, they are opaque (value of 1); however, if we make them more transparent, we should be able to see some of the overlap:

```
>>> fb.assign(  
...     max_abs_change=fb.high - fb.low  
... ).plot(  
...     kind='scatter', x='volume', y='max_abs_change',  
...     title='Facebook Daily High - '  
...         'Low vs. log(Volume Traded)',  
...     logx=True, alpha=0.25  
... )
```

Relationships between variables

- We can now begin to make out the density of points in the lower-left region of the plot, but it's still relatively difficult:



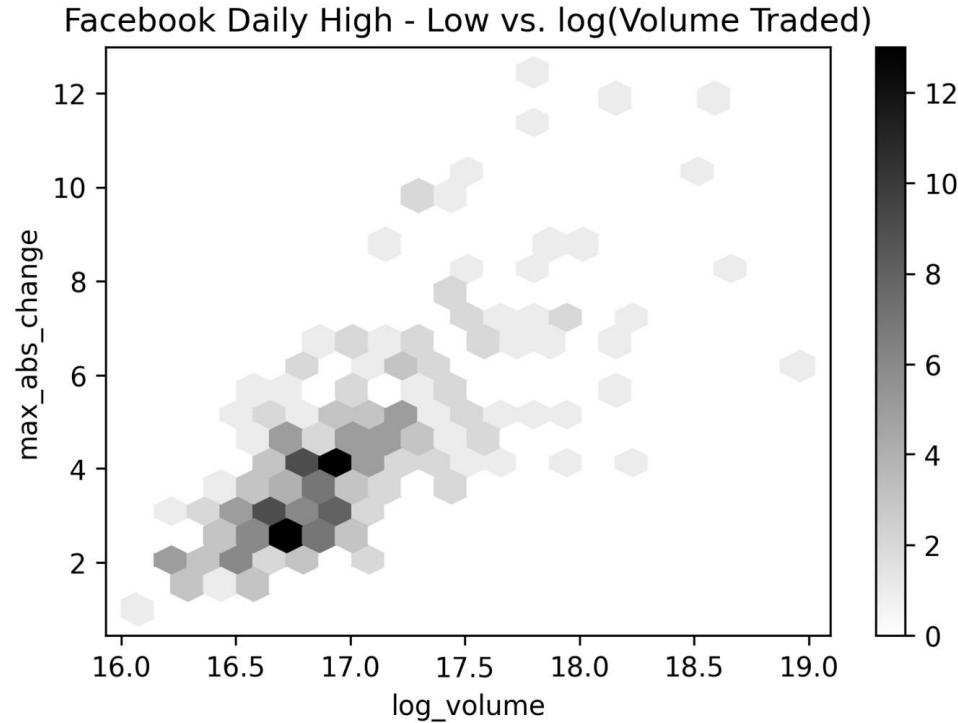
Relationships between variables

- Let's view this data as hexbins:

```
>>> fb.assign(  
...     log_volume=np.log(fb.volume),  
...     max_abs_change=fb.high - fb.low  
... ).plot(  
...     kind='hexbin',  
...     x='log_volume',  
...     y='max_abs_change',  
...     title='Facebook Daily High - '  
...             'Low vs. log(Volume Traded)',  
...     colormap='gray_r',  
...     gridsize=20,  
...     sharex=False # bug fix to keep the x-axis label  
... )
```

Relationships between variables

- A larger value for gridsize will make the bins harder to see, while a smaller one will result in fuller bins that take up more space on the plot—we must strike a balance:



Relationships between variables

- Since there is a lot of code that needs to be run in the same cell, we will discuss the purpose of each section immediately after this code block:

```
>>> fig, ax = plt.subplots(figsize=(20, 10))
# calculate the correlation matrix
>>> fb_corr = fb.assign(
...     log_volume=np.log(fb.volume),
...     max_abs_change=fb.high - fb.low
... ).corr()
# create the heatmap and colorbar
>>> im = ax.matshow(fb_corr, cmap='seismic')
>>> im.set_clim(-1, 1)
>>> fig.colorbar(im)
# label the ticks with the column names
>>> labels = [col.lower() for col in fb_corr.columns]
>>> ax.set_xticks(ax.get_xticks()[1:-1])
>>> ax.set_xticklabels(labels, rotation=45)
>>> ax.set_yticks(ax.get_yticks()[1:-1])
>>> ax.set_yticklabels(labels)
# include the value of the correlation coefficient in the boxes
>>> for (i, j), coef in np.ndenumerate(fb_corr):
...     ax.text(
...         i, j, fr'$\rho$ = {coef:.2f}',
...         ha='center', va='center',
...         color='white', fontsize=14
...     )
```

Relationships between variables

- This can be achieved by selecting the seismic colormap and then setting the limits of the color scale to [-1, 1], since the correlation coefficient has those bounds:

```
im = ax.matshow(fb_corr, cmap='seismic')
im.set_clim(-1, 1) # set the bounds of the color scale
fig.colorbar(im) # add the colorbar to the figure
```

Relationships between variables

- To be able to read the resulting heatmap, we need to label the rows and columns with the names of the variables in our data:

```
labels = [col.lower() for col in fb_corr.columns]
ax.set_xticks(ax.get_xticks()[1:-1]) # to handle matplotlib bug
ax.set_xticklabels(labels, rotation=45)
ax.set_yticks(ax.get_yticks()[1:-1]) # to handle matplotlib bug
ax.set_yticklabels(labels)
```

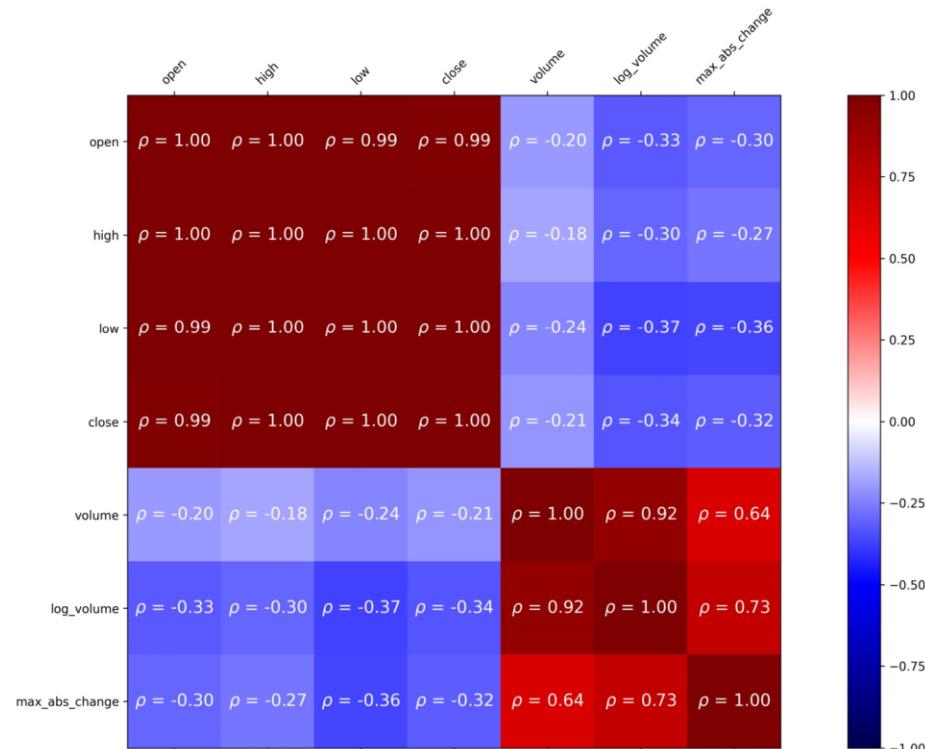
Relationships between variables

- For this plot, we placed white, center-aligned text indicating the value of the Pearson correlation coefficient for each variable combination:

```
# iterate over the matrix
for (i, j), coef in np.ndenumerate(fb_corr):
    ax.text(
        i, j,
        fr'$\rho$ = {coef:.2f}', # raw (r), format (f) string
        ha='center', va='center',
        color='white', fontsize=14
    )
```

Relationships between variables

- This results in an annotated heatmap showing the correlations between the variables in the Facebook dataset:



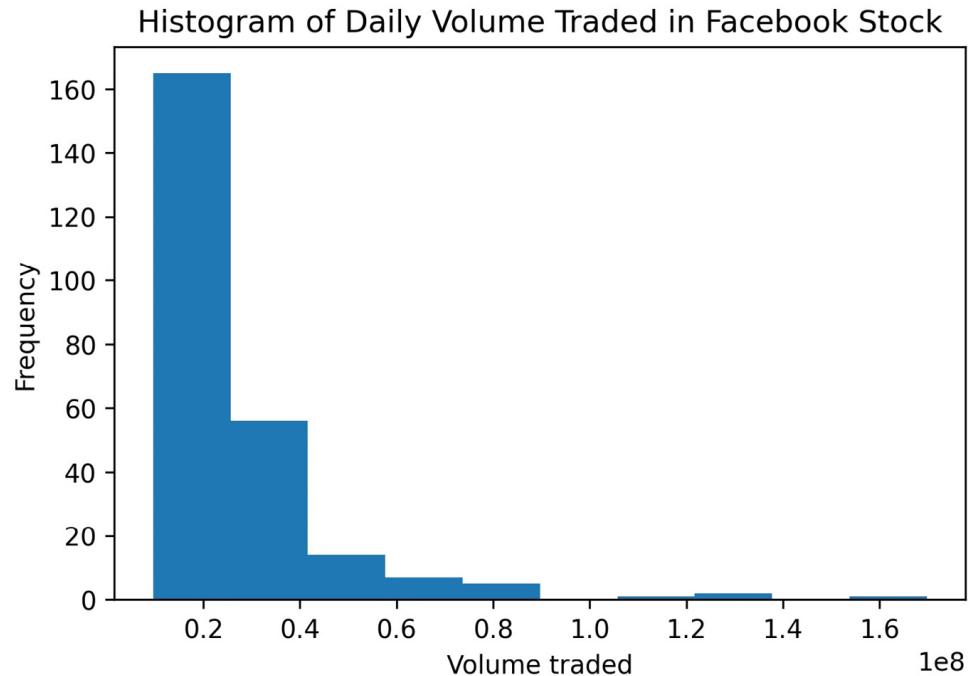
Distributions

- Let's take a look at the histogram of daily volume traded in Facebook stock:

```
>>> fb.volume.plot(  
...     kind='hist',  
...     title='Histogram of Daily Volume Traded '  
...         'in Facebook Stock'  
... )  
>>> plt.xlabel('Volume traded') # label x-axis (see ch 6)
```

Distributions

- Recall that in Aggregating Pandas DataFrames, when we discussed binning and looked at low, medium, and high volume traded, almost all of the data fell in the low bucket, which aligns with what we see in this histogram:



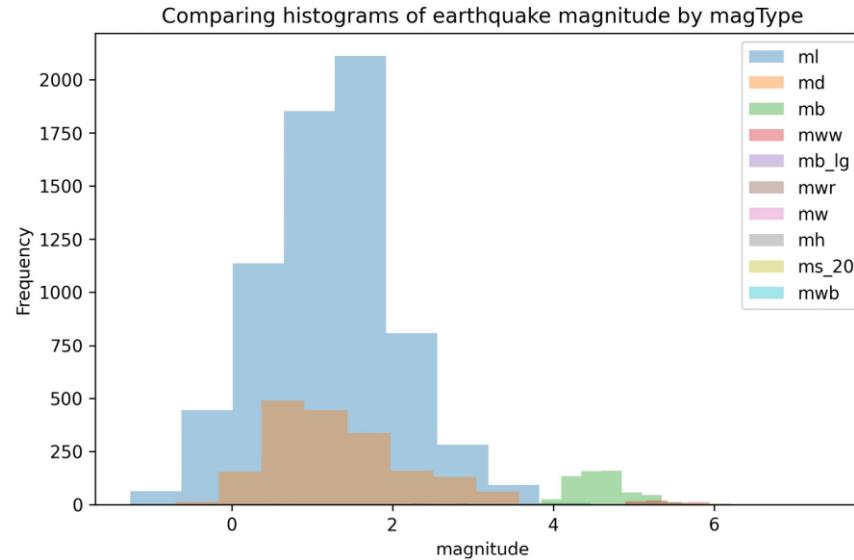
Distributions

- Given that we have many different measurement techniques for earthquakes (the magType column), we may be interested in comparing the different ranges of magnitudes they yield:

```
>>> fig, axes = plt.subplots(figsize=(8, 5))
>>> for magtype in quakes.magType.unique():
...     data = quakes.query(f'magType == "{magtype}"').mag
...     if not data.empty:
...         data.plot(
...             kind='hist',
...             ax=axes,
...             alpha=0.4,
...             label=magtype,
...             legend=True,
...             title='Comparing histograms '
...                   'of earthquake magnitude by magType'
...         )
>>> plt.xlabel('magnitude') # label x-axis (discussed in ch 6)
```

Distributions

- This shows us that ml is the most common magType, followed by md, and that they yield similar ranges of magnitudes; however, mb, which is the third-most common, yields higher magnitudes:



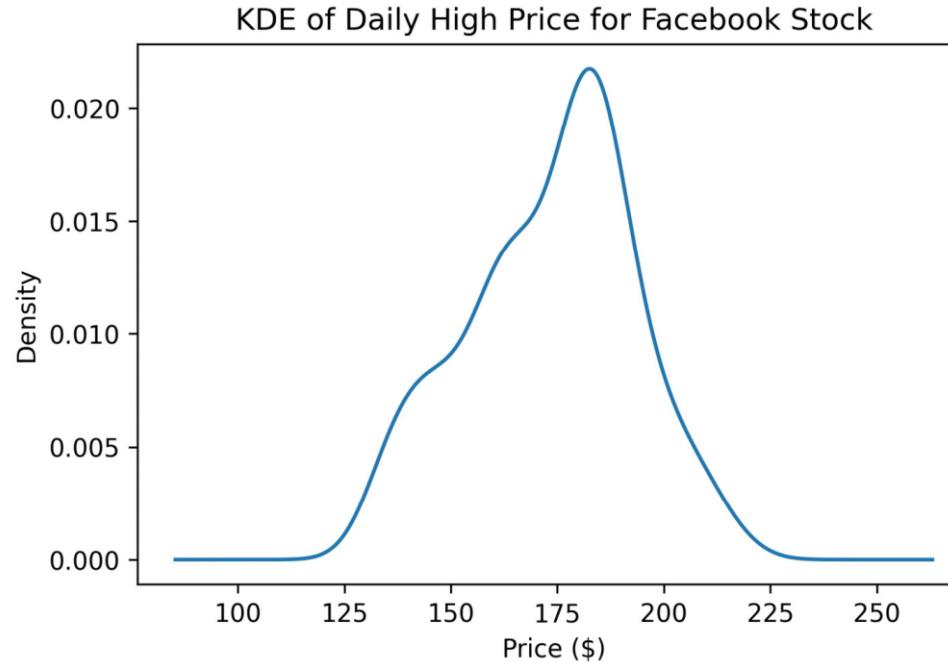
Distributions

- Note that we can pass either kind='kde' or kind='density':

```
>>> fb.high.plot(  
...     kind='kde',  
...     title='KDE of Daily High Price for Facebook Stock'  
... )  
>>> plt.xlabel('Price ($)') # label x-axis
```

Distributions

- The resulting density curve has some left skew:



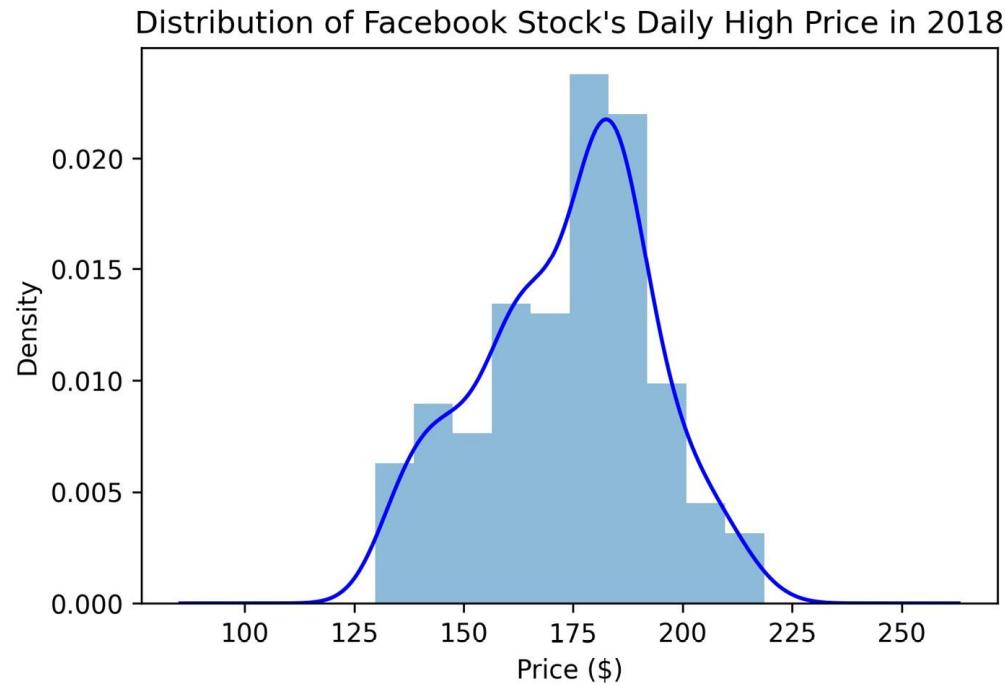
Distributions

- We may also want to visualize the KDE superimposed on top of the histogram.
- Pandas allows us to pass the Axes object we want to plot on, and also returns one after creating the visualization, which makes this a cinch:

```
>>> ax = fb.high.plot(kind='hist', density=True, alpha=0.5)
>>> fb.high.plot(
...     ax=ax, kind='kde', color='blue',
...     title='Distribution of Facebook Stock\'s '
...           'Daily High Price in 2018'
... )
>>> plt.xlabel('Price ($)') # label x-axis (discussed in ch 6)
```

Distributions

- Note that if we remove the color='blue' part of the KDE call, we don't need to change the value of alpha in the histogram call because the KDE and histogram will be different colors; we are plotting them both in blue since they represent the same data:



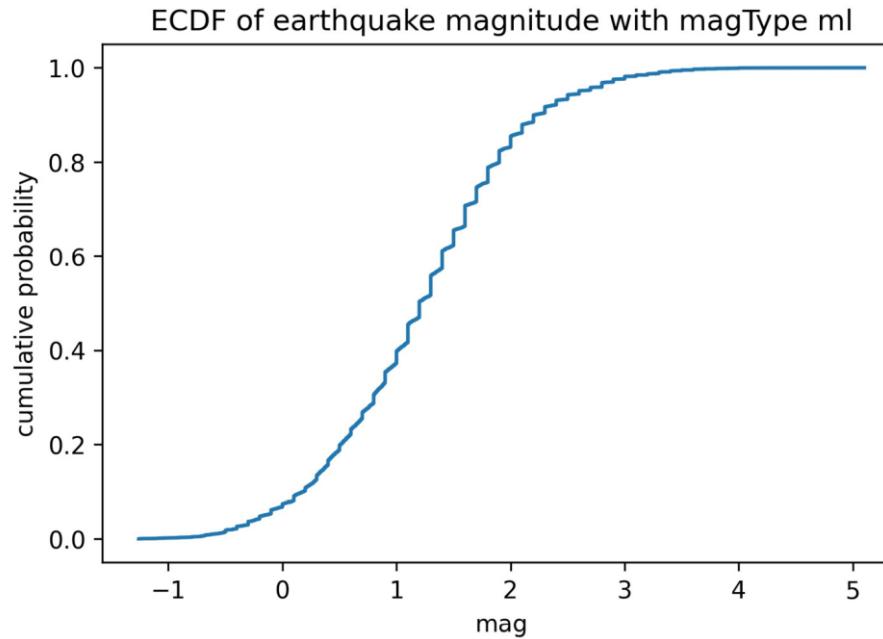
Distributions

- Let's use this to understand the distribution of magnitudes for earthquakes measured with the ml magnitude type:

```
>>> from statsmodels.distributions.empirical_distribution \
...     import ECDF
>>> ecdf = ECDF(quakes.query('magType == "ml"').mag)
>>> plt.plot(ecdf.x, ecdf.y)
# axis labels (we will cover this in chapter 6)
>>> plt.xlabel('mag') # add x-axis label
>>> plt.ylabel('cumulative probability') # add y-axis label
# add title (we will cover this in chapter 6)
>>> plt.title('ECDF of earthquake magnitude with magType ml')
```

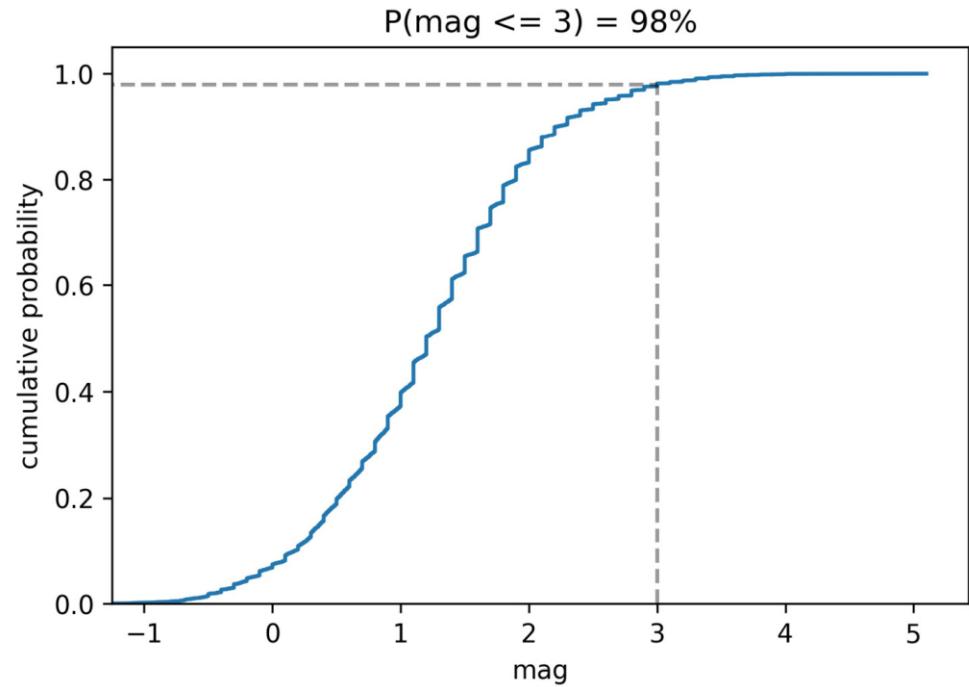
Distributions

- This yields the following ECDF:



Distributions

- Here, we can see that if this distribution is indeed representative of the population, the probability of the ml magnitude of the earthquake being less than or equal to 3 is 98% for earthquakes measured with that measurement technique:



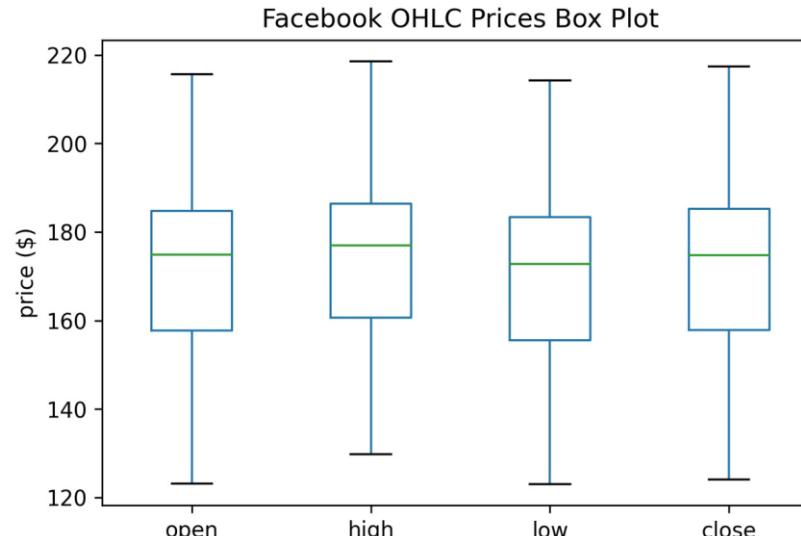
Distributions

- Finally, we can use box plots to visualize potential outliers and the distribution using quartiles.
- As an example, let's visualize the OHLC prices for Facebook stock across the whole dataset:

```
>>> fb.iloc[:, :4].plot(  
...     kind='box',  
...     title='Facebook OHLC Prices Box Plot'  
... )  
>>> plt.ylabel('price ($') # label x-axis
```

Distributions

- We no longer have an idea of the density of points throughout the distribution; with the box plot, we focus on the 5-number summary instead:



Distributions

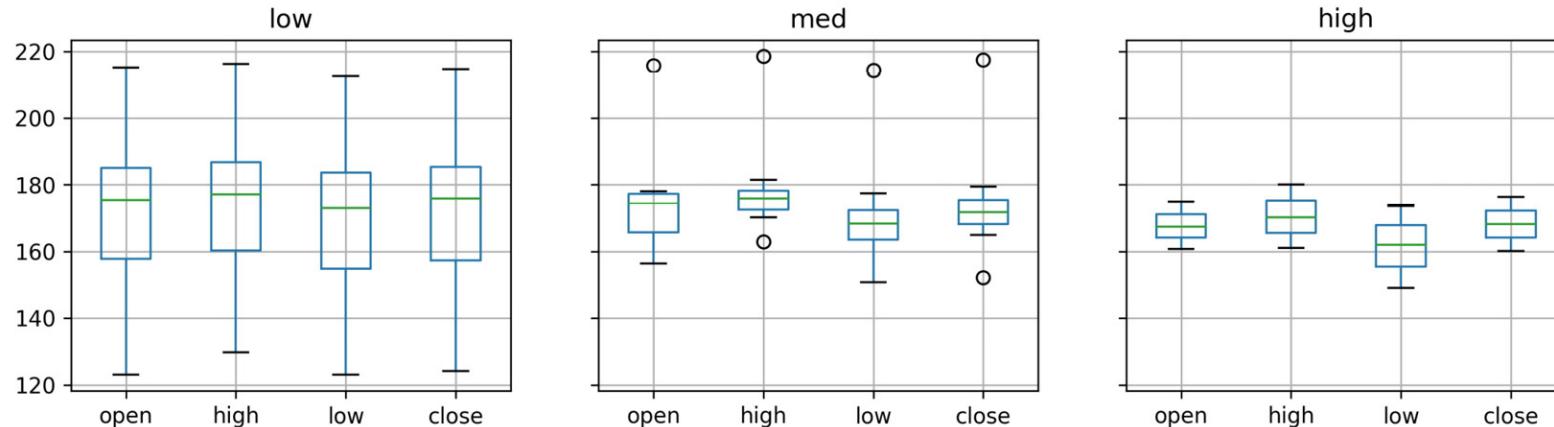
- Let's examine how the box plots change when we calculate them based on the volume traded:

```
>>> fb.assign(  
...     volume_bin=\  
...         pd.cut(fb.volume, 3, labels=['low', 'med', 'high'])  
... ).groupby('volume_bin').boxplot(  
...     column=['open', 'high', 'low', 'close'],  
...     layout=(1, 3), figsize=(12, 3)  
... )  
>>> plt.suptitle(  
...     'Facebook OHLC Box Plots by Volume Traded', y=1.1  
... )
```

Distributions

- Remember that most of the days fell in the low volume traded bucket, so we would expect to see more variation there because of what the stock data looked like over time:

Facebook OHLC Box Plots by Volume Traded



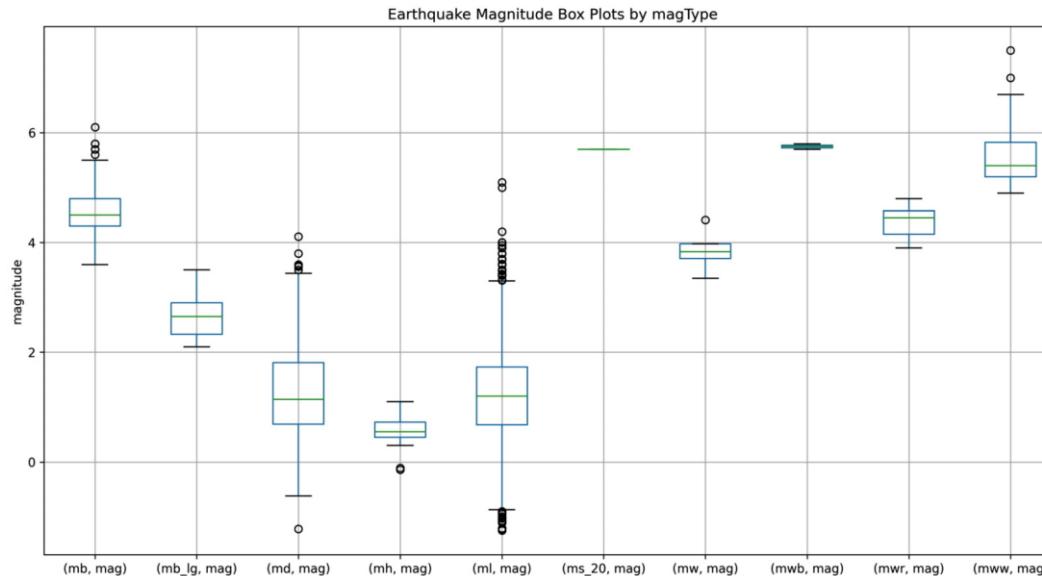
Distributions

- We can also use this technique to see the distribution of earthquake magnitudes based on which magType was used and compare it with the expected ranges on the USGS website (<https://www.usgs.gov/natural-hazards/earthquake-hazards/science/magnitude-types>):

```
>>> quakes[ [ 'mag' , 'magType' ] ]\ 
...      .groupby('magType')\ 
...      .boxplot(figsize=(15, 8), subplots=False)
# formatting (covered in chapter 6)
>>> plt.title('Earthquake Magnitude Box Plots by magType')
>>> plt.ylabel('magnitude')
```

Distributions

- Here, we can see that, together, the techniques cover a wide spectrum of magnitudes, while none of them cover everything:



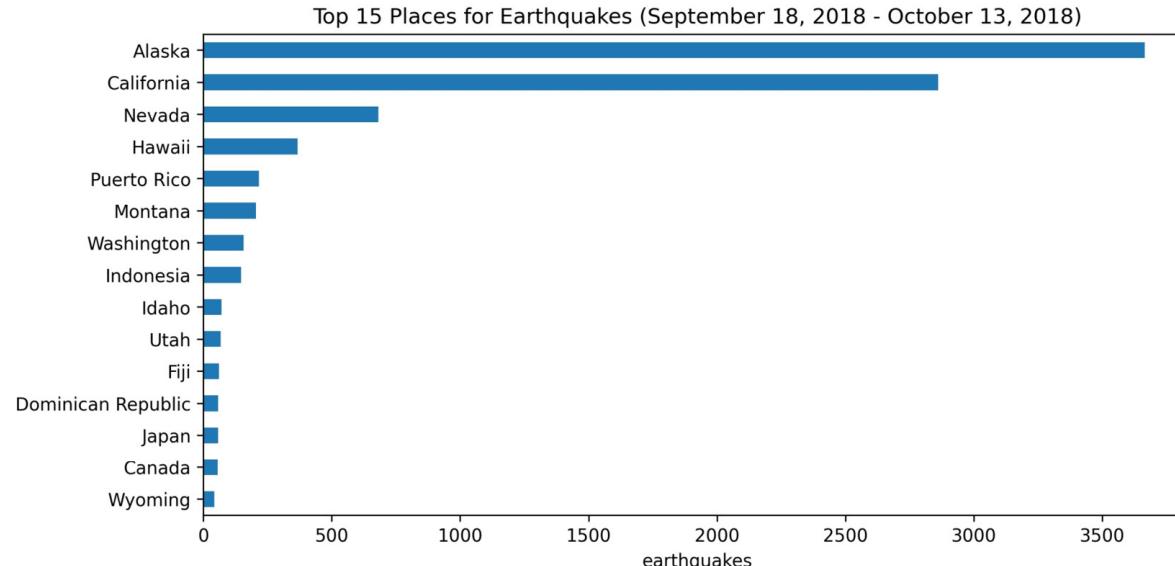
Counts and frequencies

- Note that we could reverse the sort order as an argument to `value_counts()`, but since we would still have to grab the top 15, we are doing both in a single `iloc` call:

```
>>> quakes.parsed_place.value_counts().iloc[14::-1].plot(  
...     kind='barh', figsize=(10, 5),  
...     title='Top 15 Places for Earthquakes '  
...     '(September 18, 2018 - October 13, 2018)'  
... )  
>>> plt.xlabel('earthquakes') # label x-axis
```

Counts and frequencies

- Perhaps it is surprising to see the number of earthquakes over such a short time period, but many of these earthquakes are so small in magnitude that people don't even feel them:



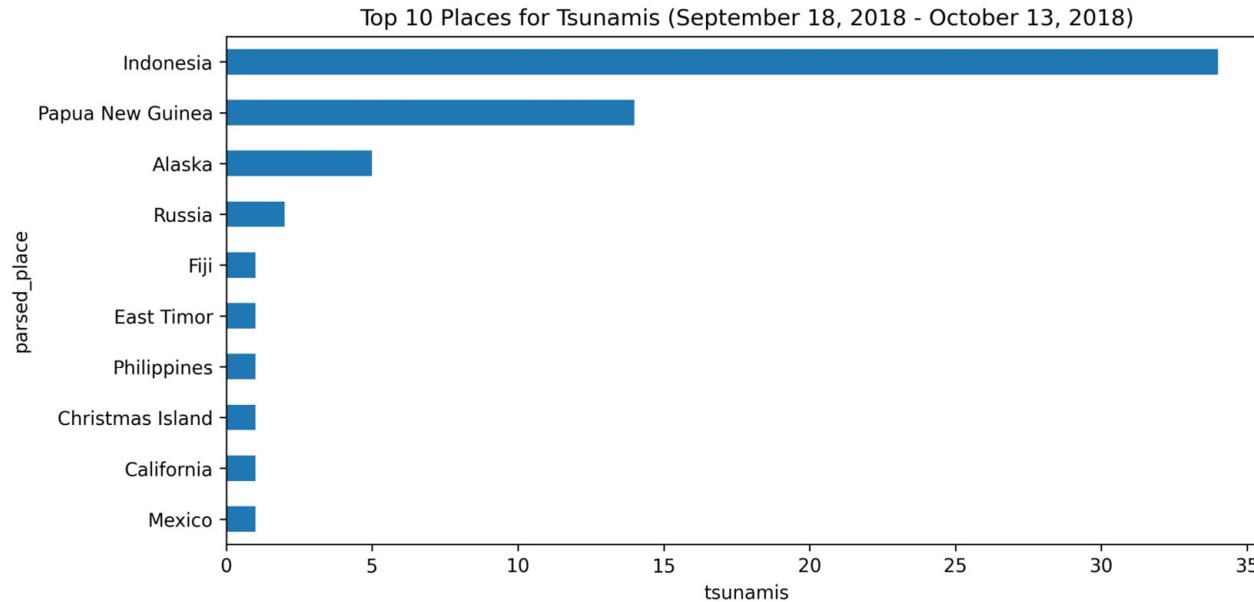
Counts and frequencies

- Let's use groupby() to make a bar plot of the top 10 places that were hit by tsunamis during the time period we have in our data:

```
>>> quakes.groupby(  
...     'parsed_place'  
... ).tsunami.sum().sort_values().iloc[-10:,].plot(  
...     kind='barh', figsize=(10, 5),  
...     title='Top 10 Places for Tsunamis '  
...             '(September 18, 2018 - October 13, 2018)'  
... )  
>>> plt.xlabel('tsunamis') # label x-axis (discussed in ch 6)
```

Counts and frequencies

- Here, we can see that Indonesia had many more tsunamis than the other places during this time period:



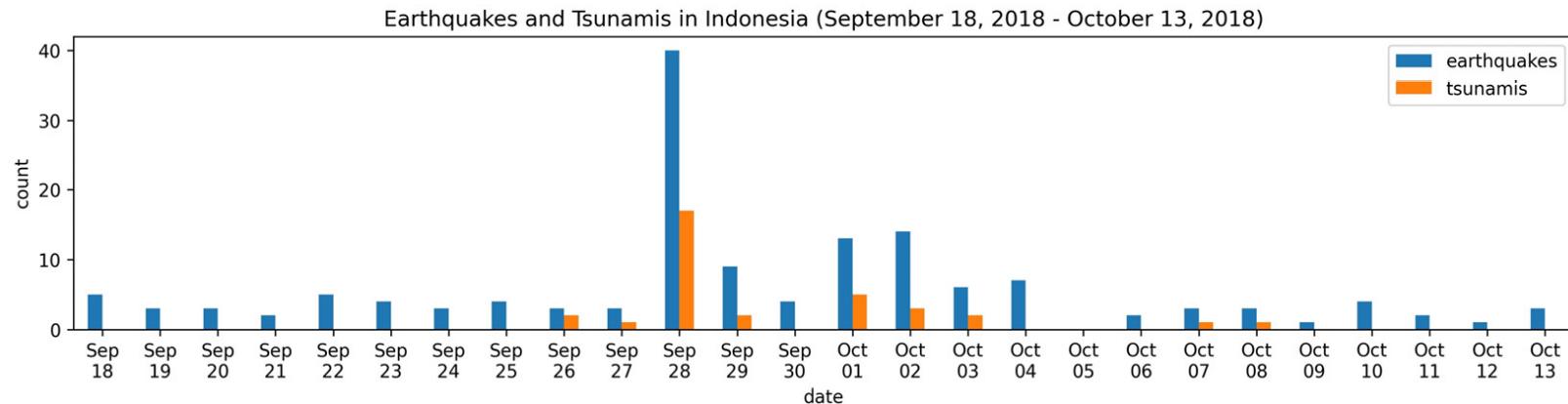
Counts and frequencies

- We can visualize this evolution over time as a line plot or with a vertical bar plot by using kind='bar'.
- Here, we will use bars to avoid interpolating the points:

```
>>> indonesia_quakes = quakes.query(  
...     'parsed_place == "Indonesia"'  
... ).assign(  
...     time=lambda x: pd.to_datetime(x.time, unit='ms'),  
...     earthquake=1  
... ).set_index('time').resample('1D').sum()  
# format the datetimes in the index for the x-axis  
>>> indonesia_quakes.index = \  
...     indonesia_quakes.index.strftime('%b\n%d')  
>>> indonesia_quakes.plot(  
...     y=['earthquake', 'tsunami'], kind='bar', rot=0,  
...     figsize=(15, 3), label=['earthquakes', 'tsunamis'],  
...     title='Earthquakes and Tsunamis in Indonesia '  
...                     '(September 18, 2018 - October 13, 2018)'  
... )  
# label the axes (discussed in chapter 6)  
>>> plt.xlabel('date')  
>>> plt.ylabel('count')
```

Counts and frequencies

- On September 28, 2018, we can see a spike in both earthquakes and tsunamis in Indonesia; on this date a 7.5 magnitude earthquake occurred, causing a devastating tsunami:



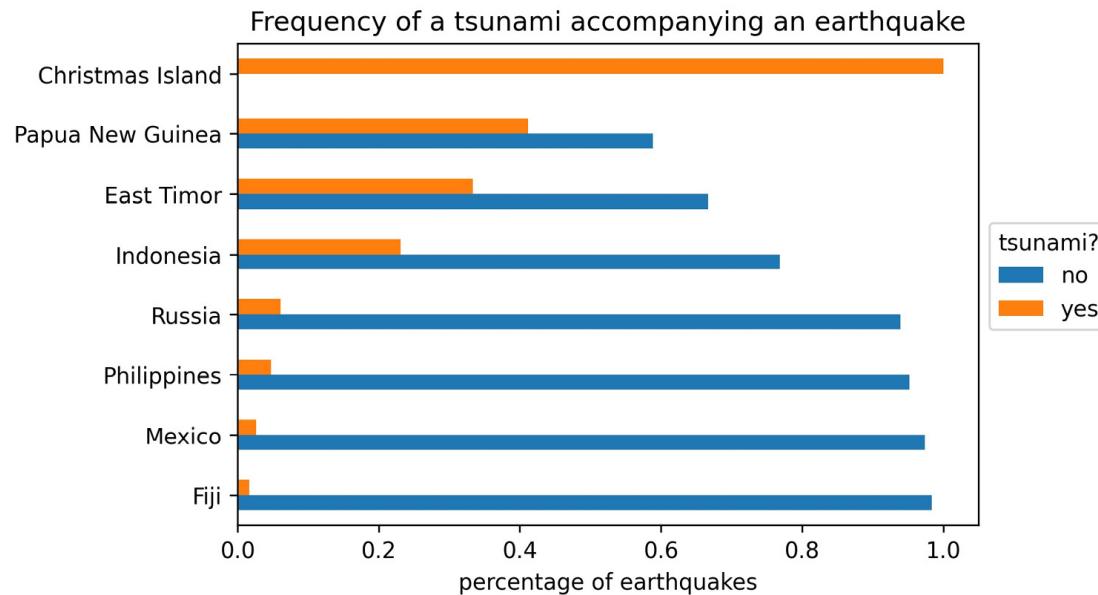
Counts and frequencies

- For illustration purposes, we will look at the seven places with the highest percentage of earthquakes accompanied by a tsunami:

```
>>> quakes.groupby(['parsed_place', 'tsunami']).mag.count()\
...     .unstack().apply(lambda x: x / x.sum(), axis=1)\
...     .rename(columns={0: 'no', 1: 'yes'})\
...     .sort_values('yes', ascending=False)[7::-1]\
...     .plot.barh(
...         title='Frequency of a tsunami accompanying '
...               'an earthquake'
...     )
# move legend to the right of the plot; label axes
>>> plt.legend(title='tsunami?', bbox_to_anchor=(1, 0.65))
>>> plt.xlabel('percentage of earthquakes')
>>> plt.ylabel('')
```

Counts and frequencies

- Papua New Guinea, on the other hand, had tsunamis alongside roughly 40% of its earthquakes:



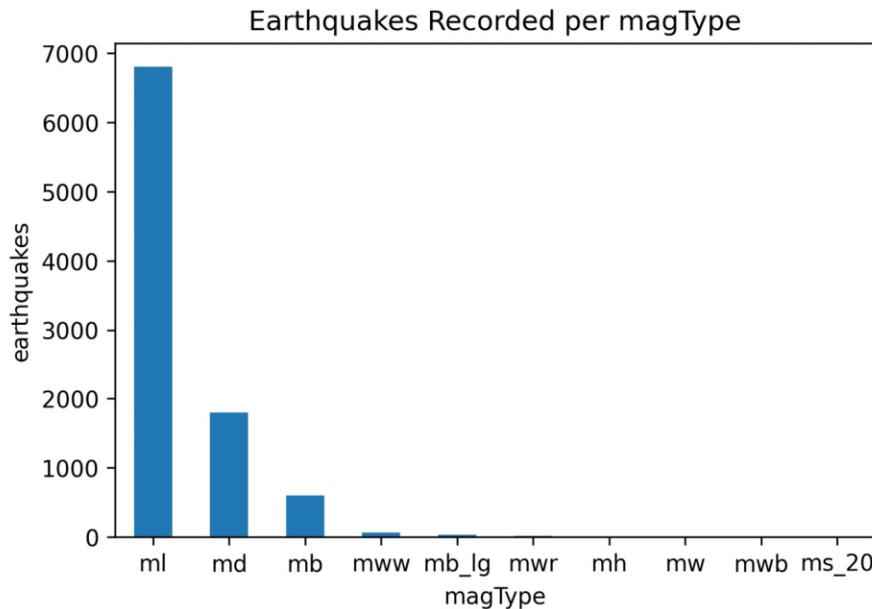
Counts and frequencies

- Now, let's use vertical bars to see which methods of measuring earthquake magnitude are most prevalent by using kind='bar':

```
>>> quakes.magType.value_counts().plot(  
...     kind='bar', rot=0,  
...     title='Earthquakes Recorded per magType'  
... )  
# label the axes (discussed in ch 6)  
>>> plt.xlabel('magType')  
>>> plt.ylabel('earthquakes')
```

Counts and frequencies

- It appears that ml is, by far, the most common method for measuring earthquake magnitudes:



Counts and frequencies

- Say we want to see how many earthquakes of a given magnitude there were and to distinguish them by magType.

This shows us a few things in a single plot:

- Which magnitudes occur most often across magType.
- The relative ranges of magnitude that each magType yields.
- The most common values for magType.

Counts and frequencies

- Next, we will need to create a pivot table with the magnitude in the index and the magnitude type along the columns; we will count the number of earthquakes for the values:

```
>>> pivot = quakes.assign(  
...     mag_bin=lambda x: np.floor(x.mag)  
... ).pivot_table(  
...     index='mag_bin',  
...     columns='magType',  
...     values='mag',  
...     aggfunc='count'  
... )
```

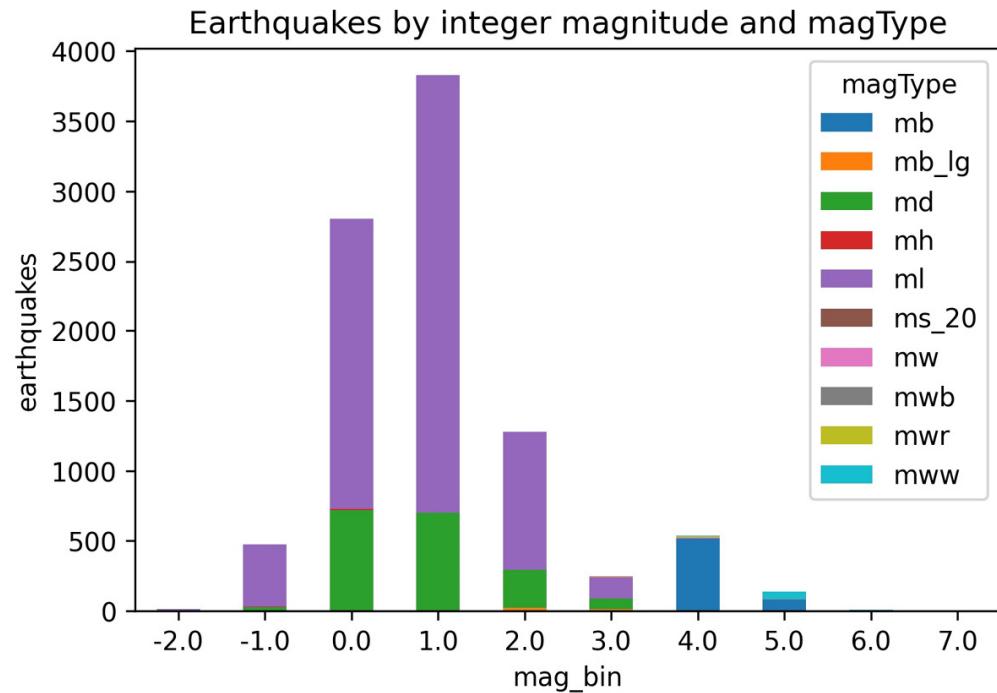
Counts and frequencies

- Once we have the pivot table, we can create a stacked bar plot by passing in stacked=True when plotting:

```
>>> pivot.plot.bar(  
...     stacked=True,  
...     rot=0,  
...     title='Earthquakes by integer magnitude and magType'  
... )  
>>> plt.ylabel('earthquakes') # label axes (discussed in ch 6)
```

Counts and frequencies

- This results in the following plot, which shows that most of the earthquakes are measured with the ml magnitude type and have magnitudes below four:



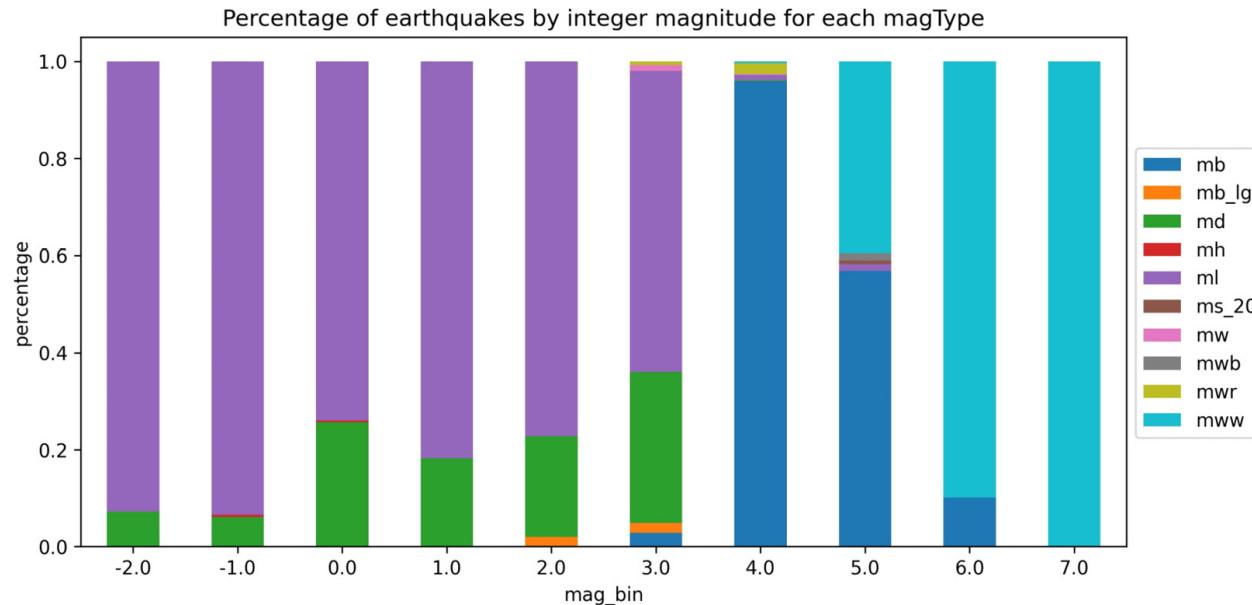
Counts and frequencies

- Rather than showing the count of earthquakes for each combination of magnitude and magType, we will show what percentage of earthquakes of a given magnitude used each magType:

```
>>> normalized_pivot = \
...     pivot.fillna(0).apply(lambda x: x / x.sum(), axis=1)
...
>>> ax = normalized_pivot.plot.bar(
...     stacked=True, rot=0, figsize=(10, 5),
...     title='Percentage of earthquakes by integer magnitude '
...           'for each magType'
... )
>>> ax.legend(bbox_to_anchor=(1, 0.8)) # move legend
>>> plt.ylabel('percentage') # label axes (discussed in ch 6)
```

Counts and frequencies

- Now, we can easily see that mww yields higher magnitudes and that ml appears to be spread across the lower end of the spectrum:



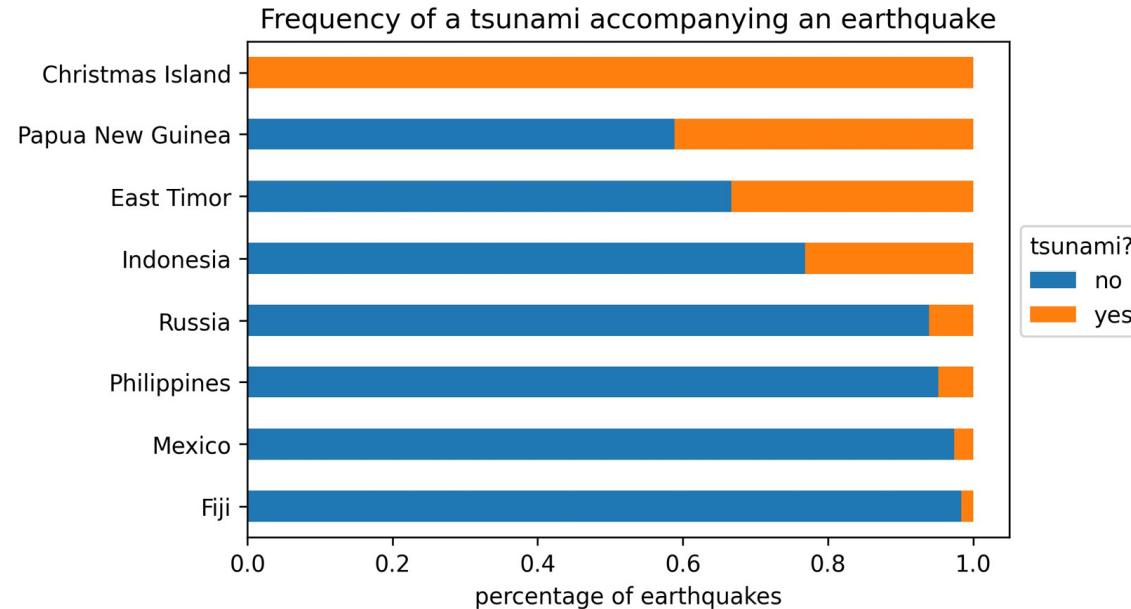
Counts and frequencies

- Let's revisit the frequency of a tsunami accompanying an earthquake plot, but rather than use grouped bars, we will stack them:

```
>>> quakes.groupby(['parsed_place', 'tsunami']).mag.count()\n...     .unstack().apply(lambda x: x / x.sum(), axis=1)\n...     .rename(columns={0: 'no', 1: 'yes'})\n...     .sort_values('yes', ascending=False)[7::-1]\n...     .plot.barh(\n...         title='Frequency of a tsunami accompanying '\n...                 'an earthquake',\n...         stacked=True\n...     )\n# move legend to the right of the plot\n>>> plt.legend(title='tsunami?', bbox_to_anchor=(1, 0.65))\n# label the axes (discussed in chapter 6)\n>>> plt.xlabel('percentage of earthquakes')\n>>> plt.ylabel('')
```

Counts and frequencies

- This stacked bar plot makes it very easy for us to compare the frequencies of tsunamis across different places:



The pandas.plotting module

- As usual, we will begin with our imports and reading in the data; we will only be using the Facebook data here:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
```

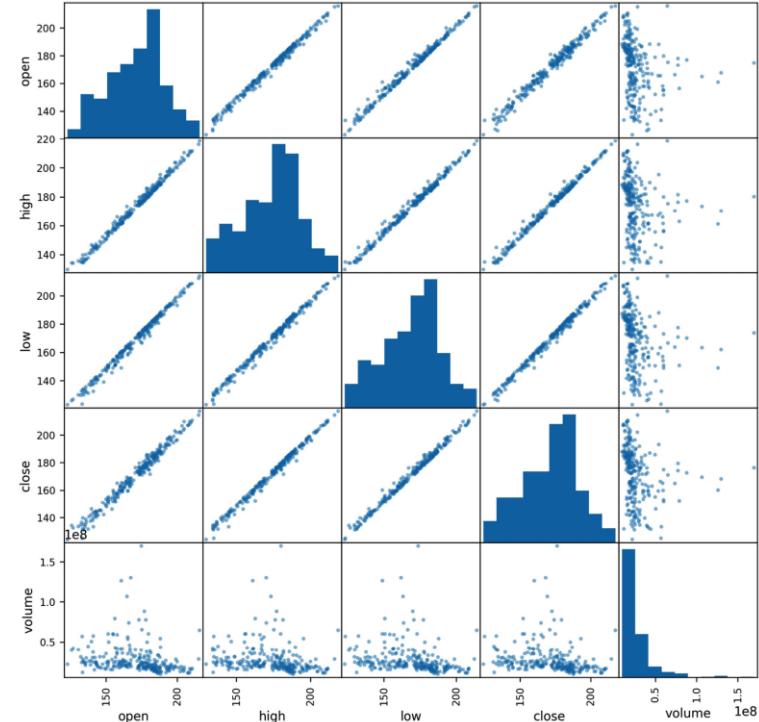
Scatter matrices

- Let's use it to view the scatter plots for each combination of columns in our Facebook stock prices data:

```
>>> from pandas.plotting import scatter_matrix  
>>> scatter_matrix(fb, figsize=(10, 10))
```

Scatter matrices

- We can easily see that we have strong positive correlations between the opening, high, low, and closing prices:



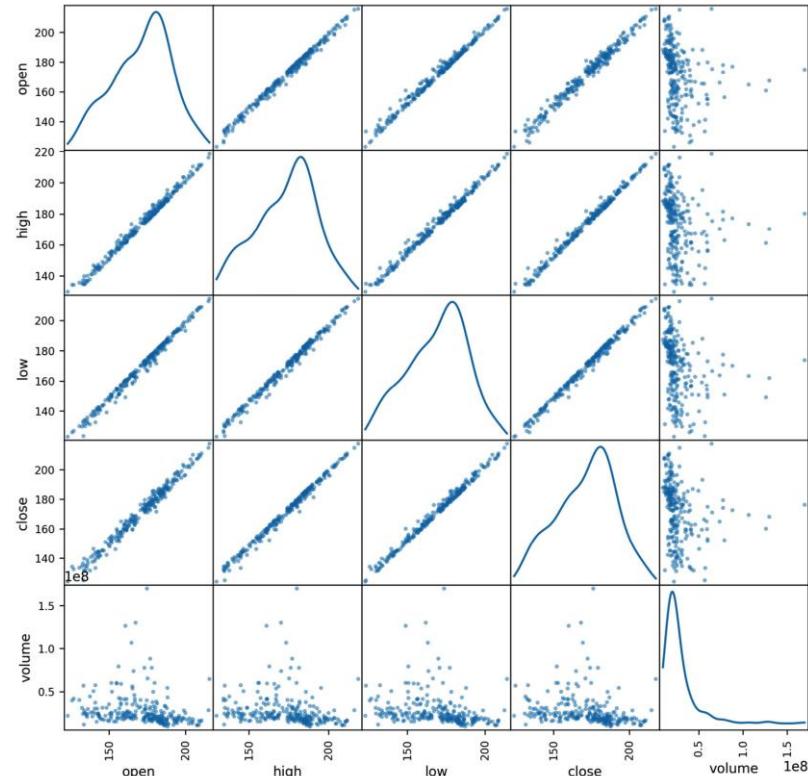
Scatter matrices

- By default, on the diagonal, where the column is paired with itself, we get its histogram.
- Alternatively, we can ask for the KDE by passing in `diagonal='kde'`:

```
>>> scatter_matrix(fb, figsize=(10, 10), diagonal='kde')
```

Scatter matrices

- This results in a scatter matrix with KDEs along the diagonal instead of histograms:



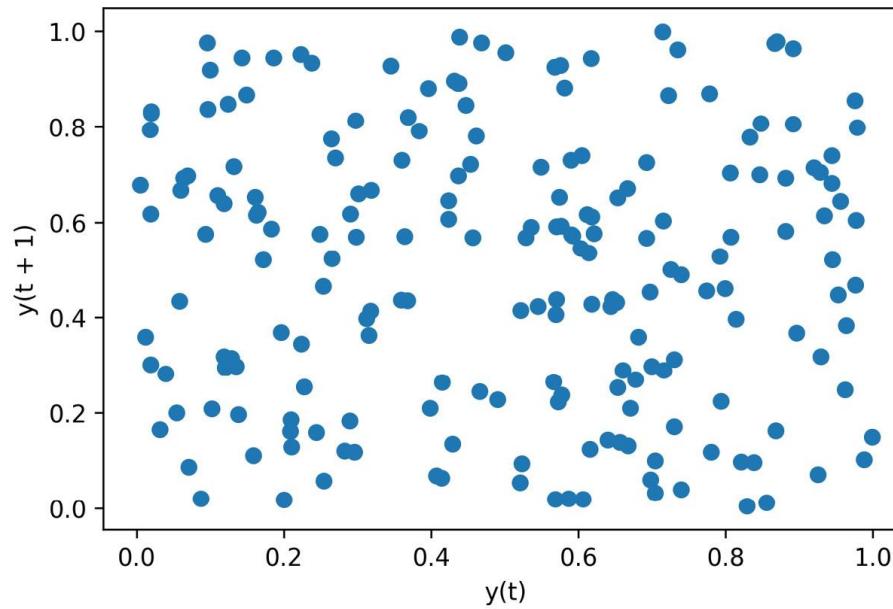
Lag plots

- If our data is random, this plot will have no pattern.
- Let's test this with some random data generated with NumPy:

```
>>> from pandas.plotting import lag_plot  
>>> np.random.seed(0) # make this repeatable  
>>> lag_plot(pd.Series(np.random.random(size=200)))
```

Lag plots

- The random data points don't indicate any pattern, just random noise:



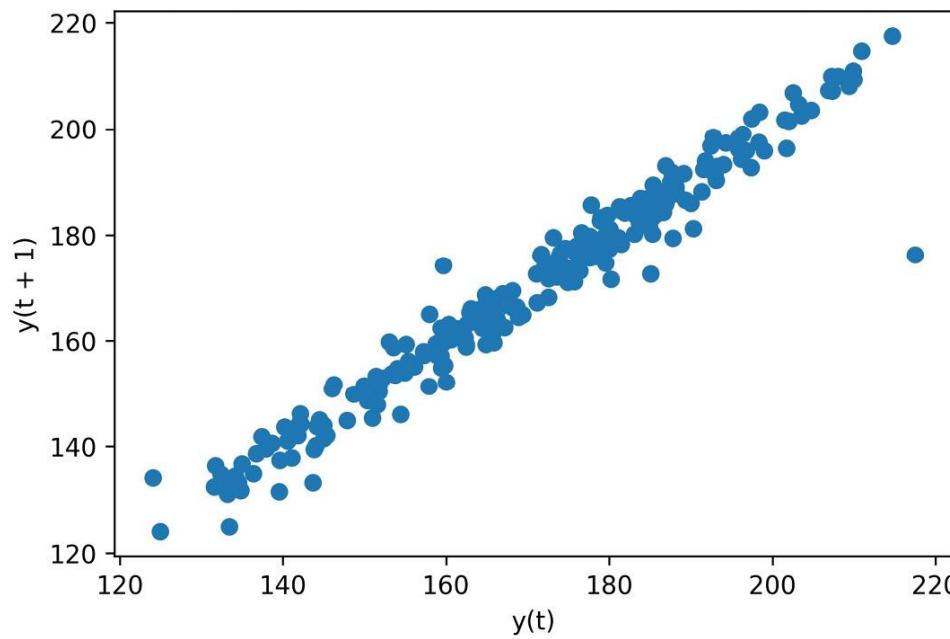
Lag plots

- With our stock data, we know that the prices on a given day are determined by what happened the day before; therefore, we would expect to see a pattern in the lag plot.
- Let's use the closing price of Facebook's stock to test whether our intuition is correct:

```
>>> lag_plot(fb.close)
```

Lag plots

- As expected, this results in a linear pattern:



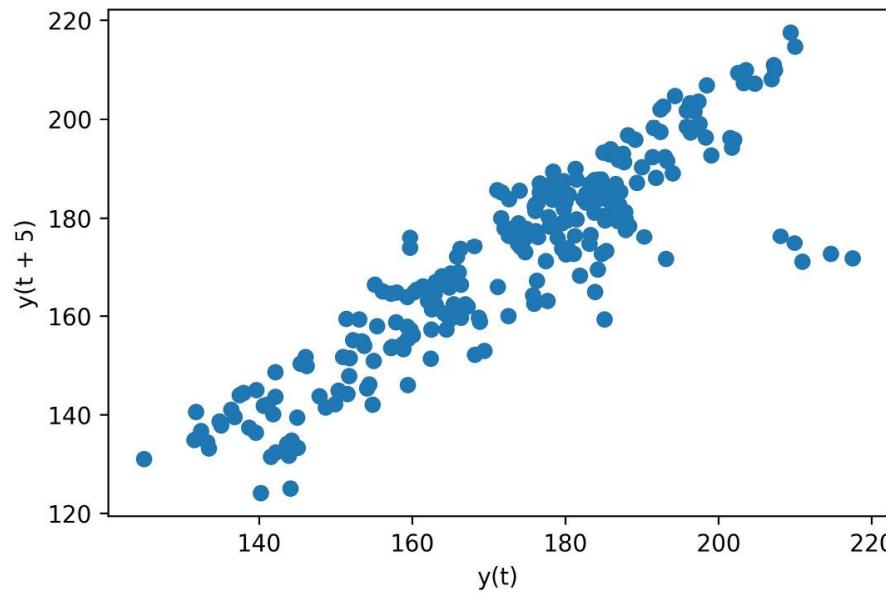
Lag plots

- We can also specify the number of periods to use for the lag. The default lag is one, but we can change this with the lag parameter.
- For example, we can compare each value to the value of the week prior with lag=5 (remember that the stock data only contains data for weekdays since the market is closed on the weekends):

```
>>> lag_plot(fb.close, lag=5)
```

Lag plots

- This still yields a strong correlation, but, compared to Figure, it definitely looks weaker:



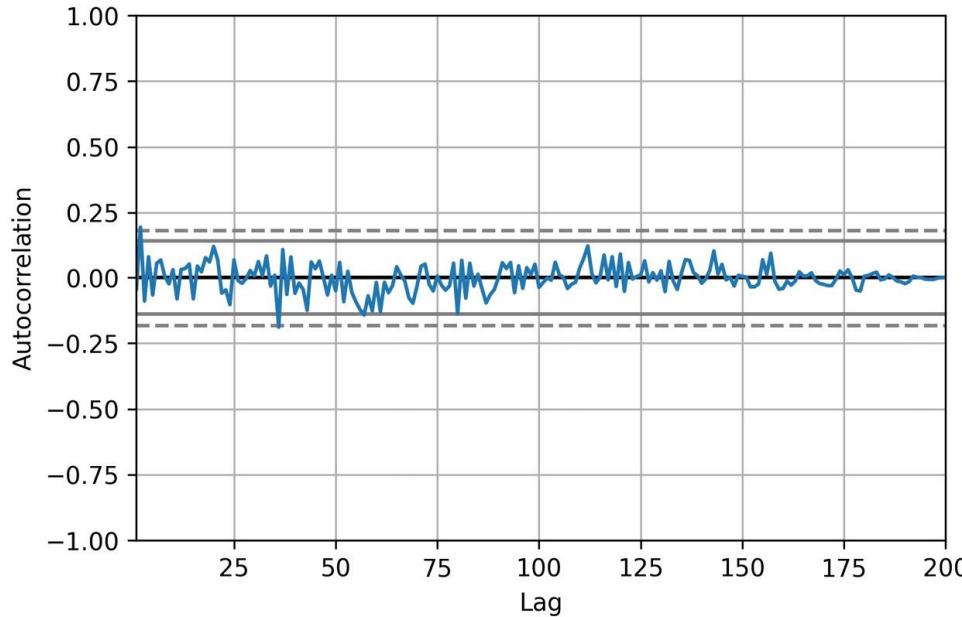
Autocorrelation plots

- As we did when discussing lag plots, let's first examine what this looks like for random data generated with NumPy:

```
>>> from pandas.plotting import autocorrelation_plot  
>>> np.random.seed(0) # make this repeatable  
>>> autocorrelation_plot(pd.Series(np.random.random(size=200)))
```

Autocorrelation plots

- Indeed, the autocorrelation is near zero, and the line is within the confidence bands (99% is dashed; 95% is solid):



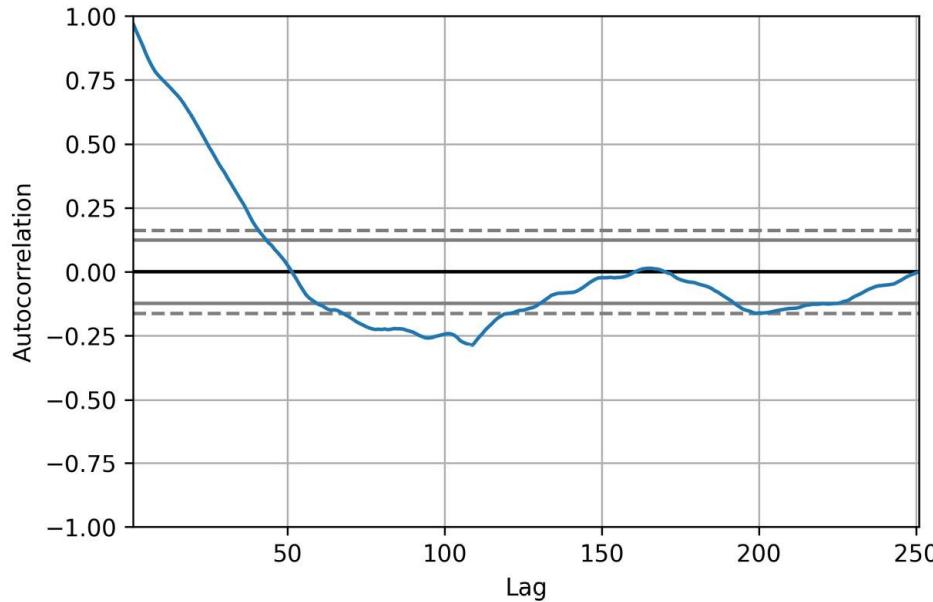
Autocorrelation plots

- Let's explore what the autocorrelation plot looks like for the closing price of Facebook's stock, since the lag plots indicated several periods of autocorrelation:

```
>>> autocorrelation_plot(fb.close)
```

Autocorrelation plots

- Here, we can see that there is autocorrelation for many lag periods before it becomes noise:



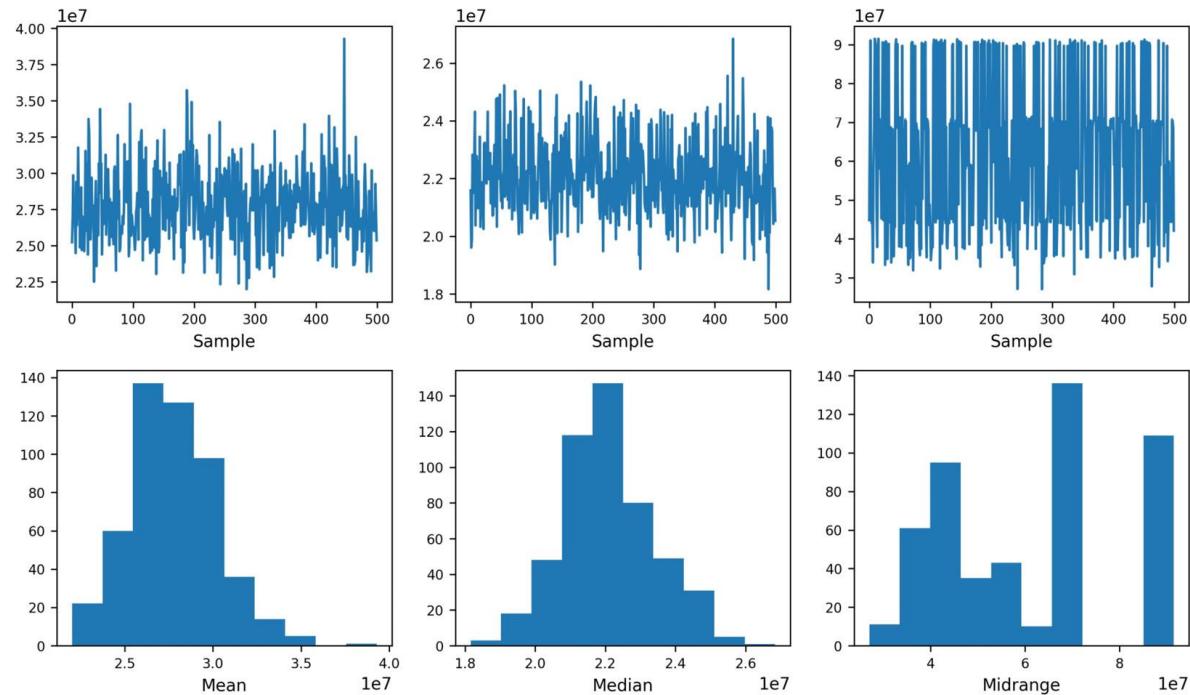
Bootstrap plots

- Let's see what the uncertainty for the summary statistics of the volume traded data looks like:

```
>>> from pandas.plotting import bootstrap_plot  
>>> fig = bootstrap_plot(  
...     fb.volume, fig=plt.figure(figsize=(10, 6))  
... )
```

Bootstrap plots

- This results in the following subplots, which we can use to assess the uncertainty in the mean, median, and midrange (the midpoint of the range):



Summary



- Now that we've completed this lesson, we are well-equipped to quickly create a variety of visualizations in Python using pandas and matplotlib.
- We now understand the basics of how matplotlib works and the main components of a plot.

"Complete Exercises"

"Complete Lab 10"

11: Plotting with Seaborn and Customization Techniques



Plotting with Seaborn and Customization Techniques

In this lesson, we will cover the following topics:

- Utilizing seaborn for more advanced plot types
- Formatting plots with matplotlib
- Customizing visualizations

lesson materials

- We will be working with three datasets once again, all of which can be found in the data/ directory.
- In the fb_stock_prices_2018.csv file, we have Facebook's stock price for all trading days in 2018.
- This data is the OHLC data (opening, high, low, and closing price), along with the volume traded.

Utilizing seaborn for advanced plotting

- For this section, we will be working with the 1-introduction_to_seaborn.ipynb notebook.
- First, we must import seaborn, which is traditionally aliased as sns:

```
>>> import seaborn as sns
```

Utilizing seaborn for advanced plotting

- Let's also import numpy, matplotlib.pyplot, and pandas, and then read in the CSV files for the Facebook stock prices and earthquake data:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
>>> quakes = pd.read_csv('data/earthquakes.csv')
```

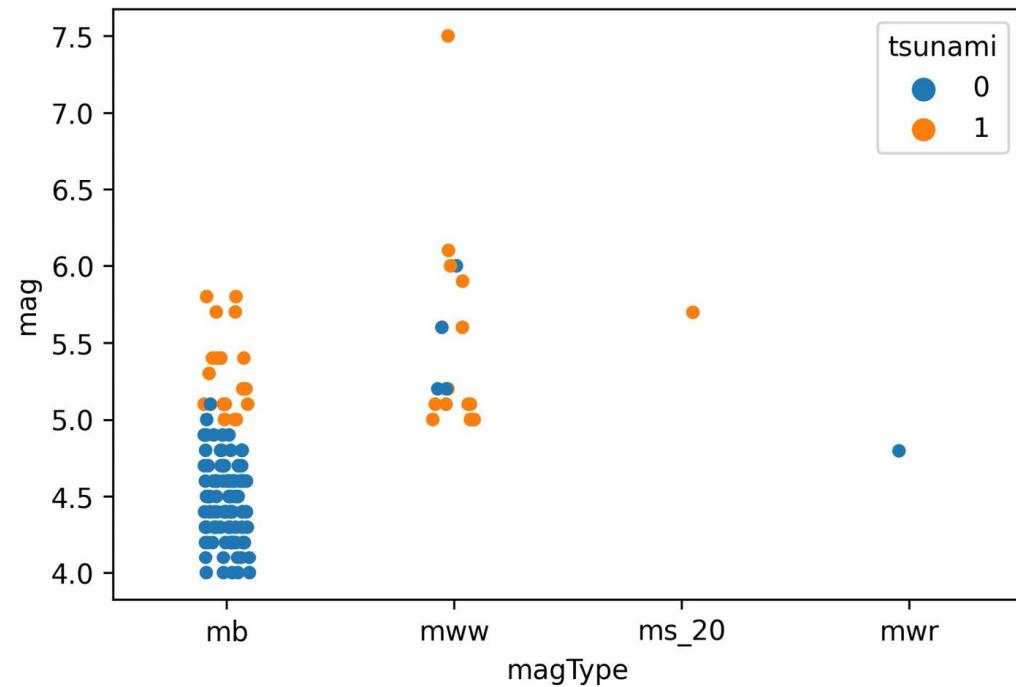
Categorical data

- We pass the subset of earthquakes occurring in Indonesia to the data parameter, and specify that we want to put magType on the x-axis (x), magnitudes on the y-axis (y), and color the points by whether the earthquake was accompanied by a tsunami (hue):

```
>>> sns.stripplot(  
...     x='magType',  
...     y='mag',  
...     hue='tsunami',  
...     data=quakes.query('parsed_place == "Indonesia"')  
... )
```

Categorical data

- Using the resulting plot, we can see that the earthquake in question is the highest orange point in the mww column (don't forget to call plt.show() if not using the Jupyter Notebook provided):



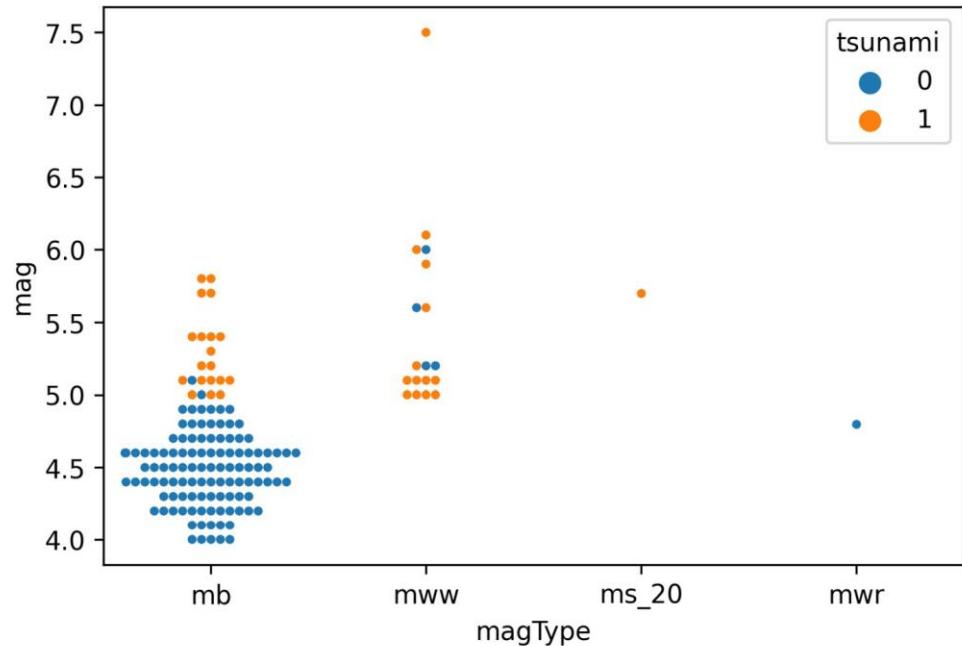
Categorical data

- So we will use that instead:

```
>>> sns.swarmplot(  
...     x='magType',  
...     y='mag',  
...     hue='tsunami',  
...     data=quakes.query('parsed_place == "Indonesia"),  
...     size=3.5 # point size  
... )
```

Categorical data

- The swarm plot (or bee swarm plot) also has the bonus of giving us a glimpse of what the distribution might be.
- We can now see many more earthquakes in the lower section of the mb column:



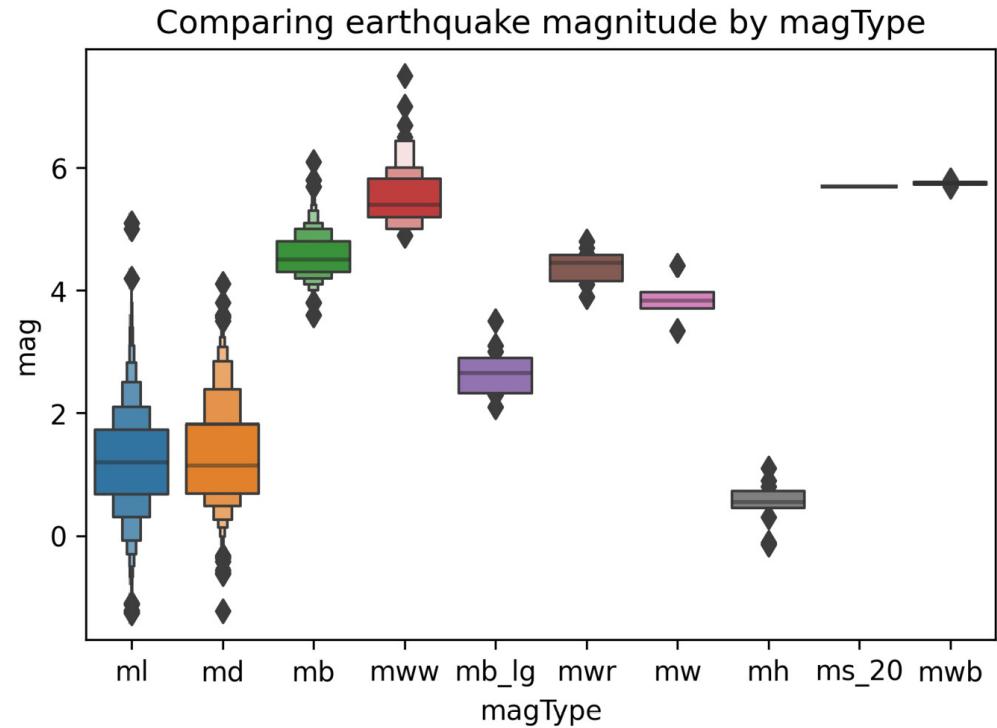
Categorical data

- Let's use the enhanced box plot to compare earthquake magnitudes across different magnitude types, as we did in Visualizing Data with Pandas and Matplotlib:

```
>>> sns.boxenplot(  
...     x='magType', y='mag', data=quakes[['magType', 'mag']]  
... )  
>>> plt.title('Comparing earthquake magnitude by magType')
```

Categorical data

- This results in the following plot:



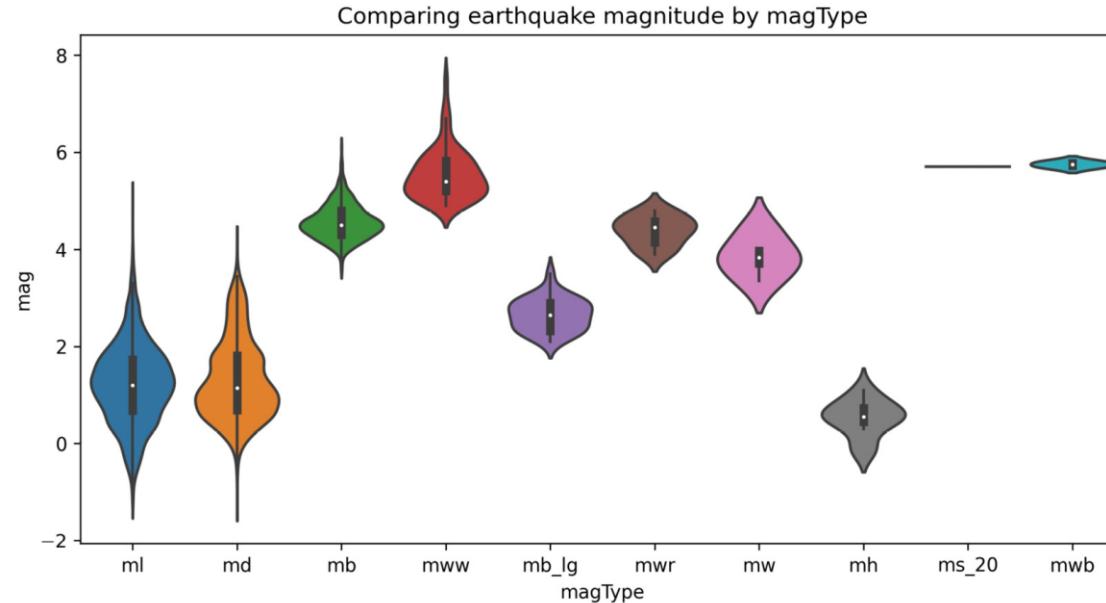
Categorical data

- As we saw, an enhanced box plot is one way to address this—another strategy is to use a violin plot, which combines a kernel density estimate (estimation of the underlying distribution) and a box plot:

```
>>> fig, axes = plt.subplots(figsize=(10, 5))
>>> sns.violinplot(
...     x='magType', y='mag', data=quakes[['magType', 'mag']],
...     ax=axes, scale='width' # all violins have same width
... )
>>> plt.title('Comparing earthquake magnitude by magType')
```

Categorical data

- We can read the KDE from either side of the box plot since it is symmetrical:



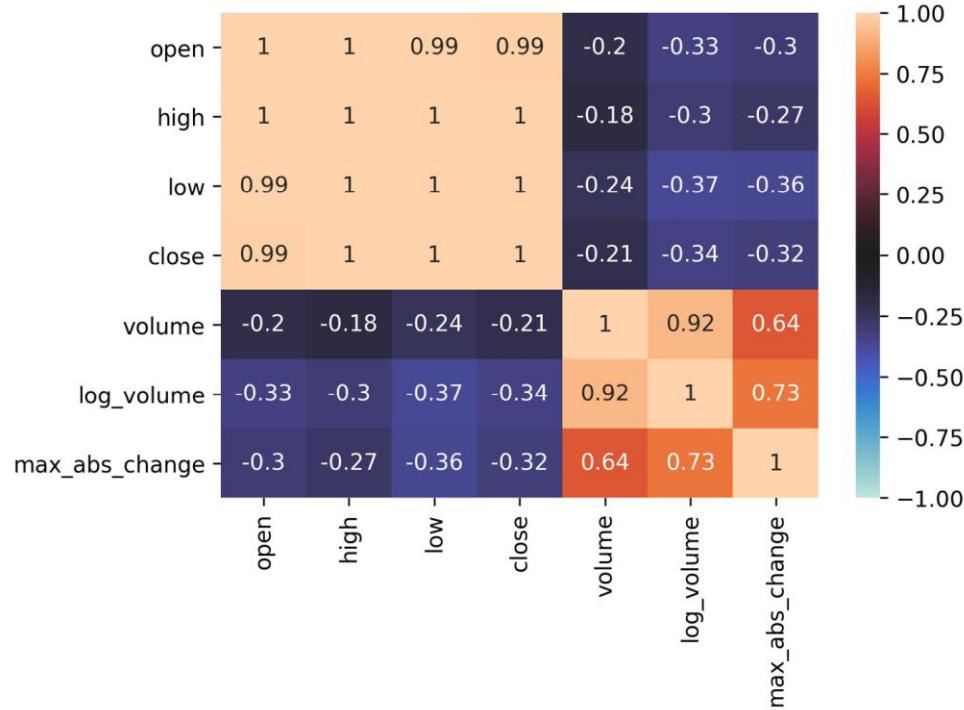
Correlations and heatmaps

- however, this time, we will use seaborn, which gives us the `heatmap()` function for an easier way to produce this visualization:

```
>>> sns.heatmap(  
...     fb.sort_index().assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     ).corr(),  
...     annot=True,  
...     center=0,  
...     vmin=-1,  
...     vmax=1  
... )
```

Correlations and heatmaps

- Notice that we also passed in `annot=True` to write the correlation coefficients in each box—we get the benefit of the numerical data and the visual data all in one plot with a single function call:



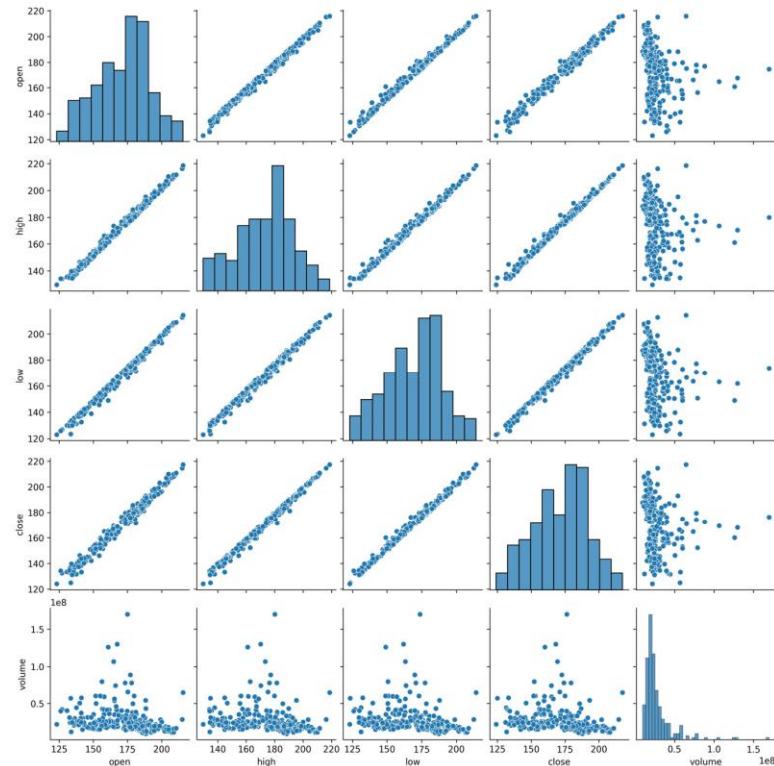
Correlations and heatmaps

- Seaborn also provides us with an alternative to the scatter_matrix() function provided in the pandas.plotting module, called pairplot().
- We can use this to see the correlations between the columns in the Facebook data as scatter plots instead of the heatmap:

```
>>> sns.pairplot(fb)
```

Correlations and heatmaps

- This result makes it easy to understand the near-perfect positive correlation between the OHLC columns shown in the heatmap, while also showing us histograms for each column along the diagonal:



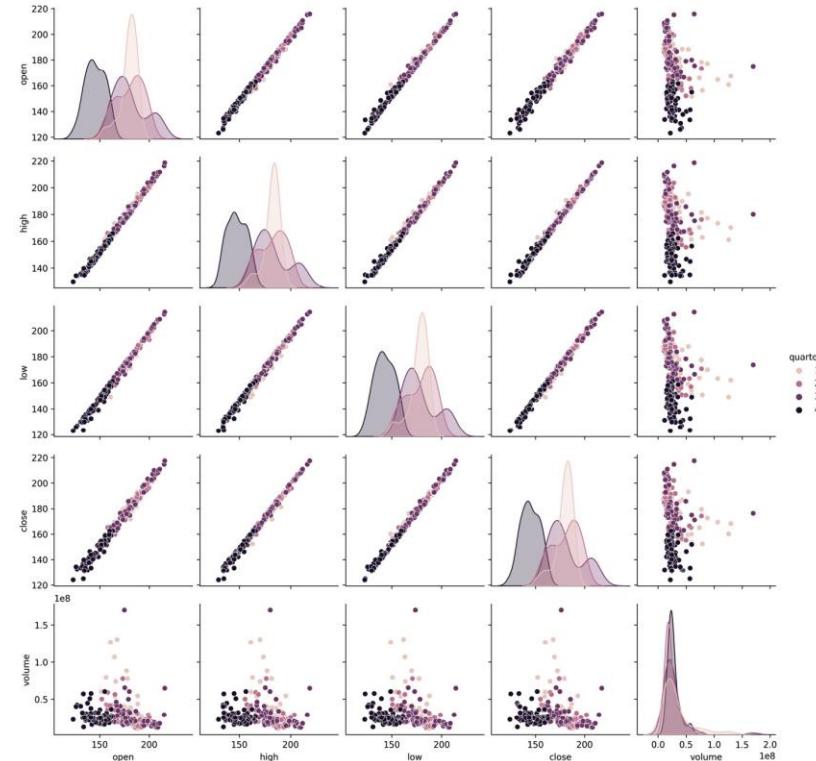
Correlations and heatmaps

- To do so, we just add the quarter column and then provide it to the hue argument:

```
>>> sns.pairplot(  
...     fb.assign(quarter=lambda x: x.index.quarter),  
...     diag_kind='kde', hue='quarter'  
... )
```

Correlations and heatmaps

- We can now see how the distributions of the OHLC columns had lower standard deviations (and, subsequently, lower variances) in the first quarter and how the stock price lost a lot of ground in the fourth quarter (the distribution shifts to the left):



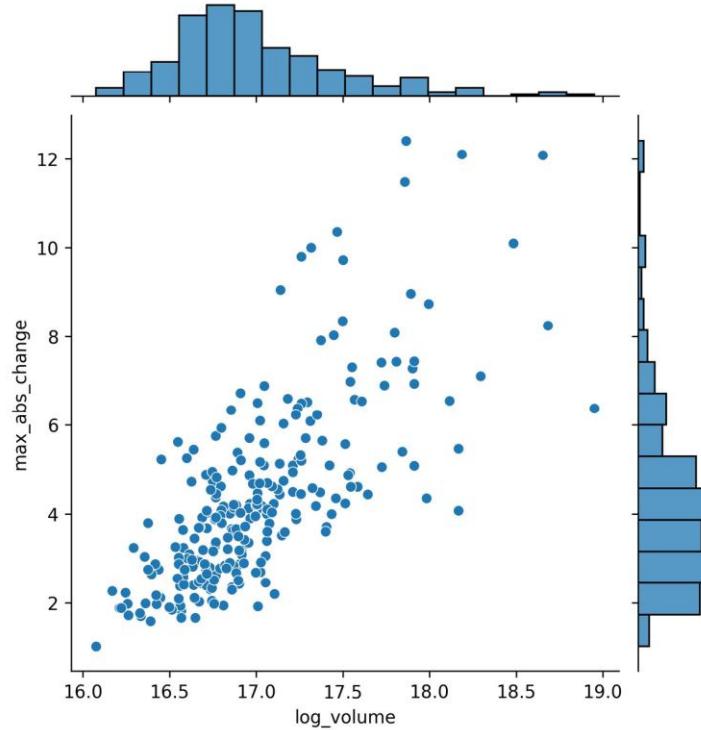
Correlations and heatmaps

- Let's look once again at how the log of volume traded correlates with the difference between the daily high and low prices in Facebook stock, as we did in Visualizing Data with Pandas and Matplotlib:

```
>>> sns.jointplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     )  
... )
```

Correlations and heatmaps

- Using the default value for the kind argument, we get histograms for the distributions and a plain scatter plot in the center:



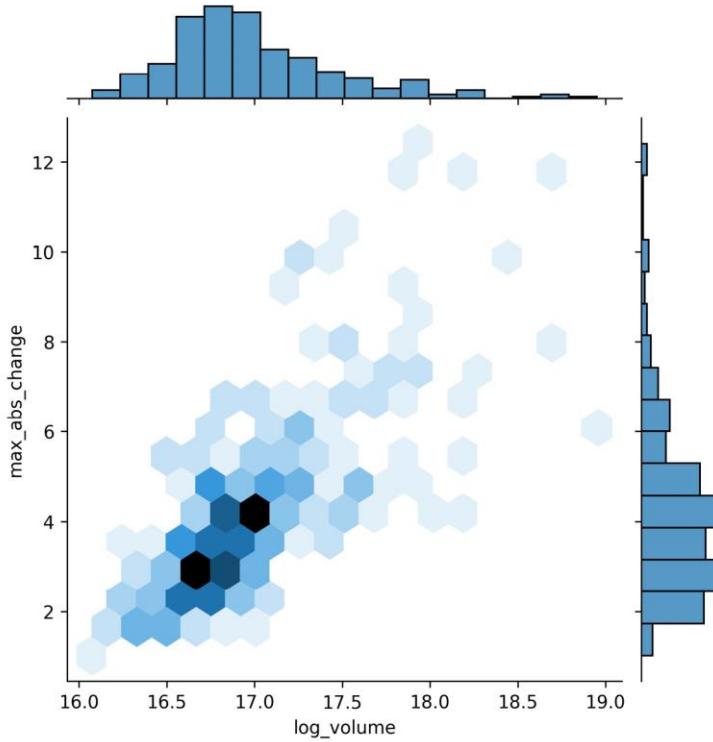
Correlations and heatmaps

- Seaborn gives us plenty of alternatives for the kind argument.
- For example, we can use hexbins because there is a significant overlap when we use the scatter plot:

```
>>> sns.jointplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     kind='hex',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     )  
... )
```

Correlations and heatmaps

- We can now see the large concentration of points in the lower-left corner:



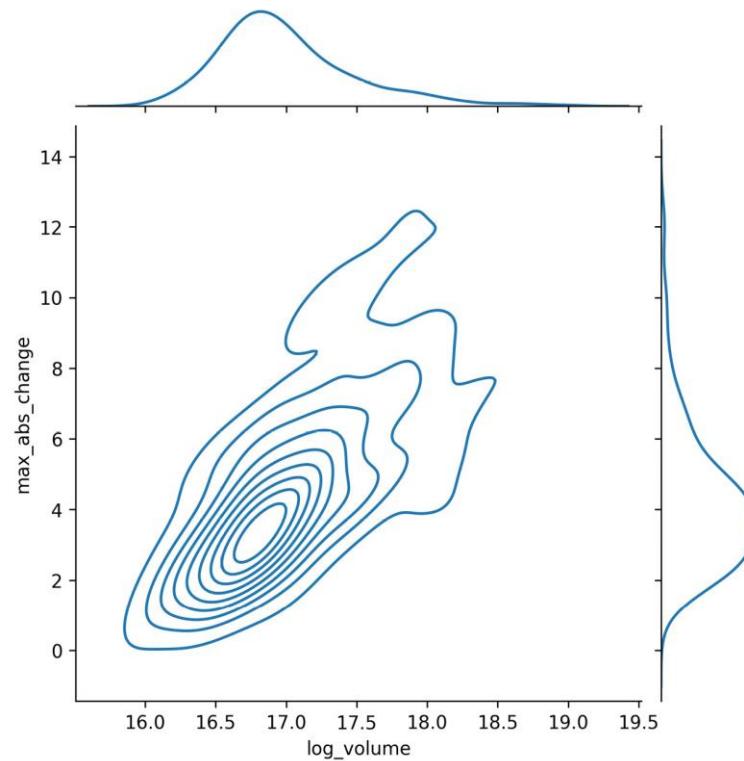
Correlations and heatmaps

- Another way of viewing the concentration of values is to use kind='kde', which gives us a contour plot to represent the joint density estimate along with KDEs for each of the variables:

```
>>> sns.jointplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     kind='kde',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     )  
... )
```

Correlations and heatmaps

- Each curve in the contour plot contains points of a given density:



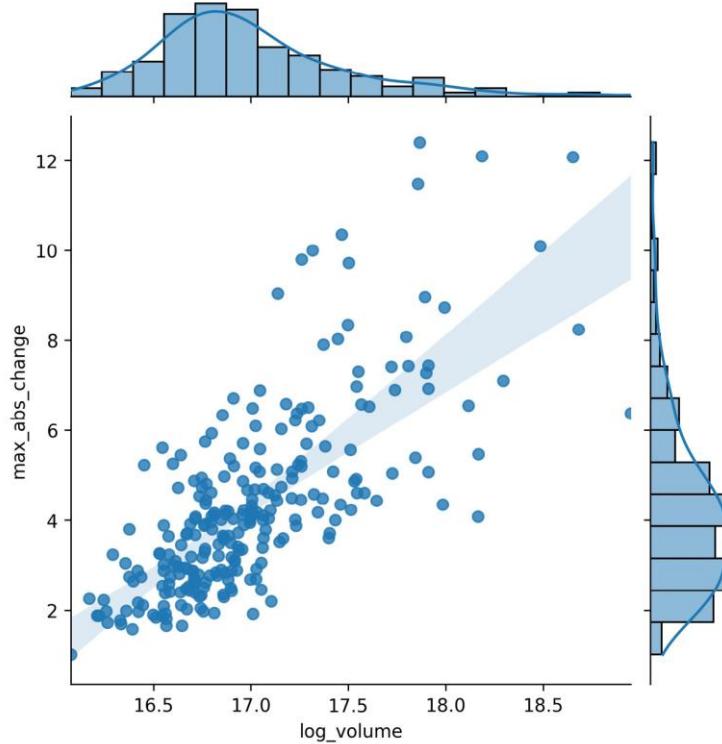
Correlations and heatmaps

- Furthermore, we can plot a regression in the center and get KDEs in addition to histograms along the sides:

```
>>> sns.jointplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     kind='reg',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     )  
... )
```

Correlations and heatmaps

- This results in a linear regression line being drawn through the scatter plot, along with a confidence band surrounding the line in a lighter color:



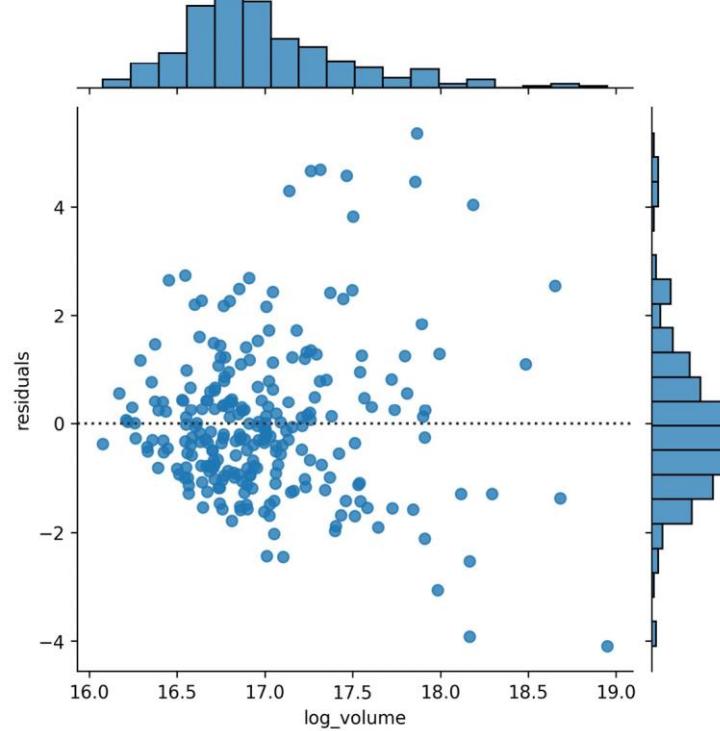
Correlations and heatmaps

- We can look directly at the residuals that would result from the previous regression with kind='resid':

```
>>> sns.jointplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     kind='resid',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low  
...     )  
... )  
# update y-axis label (discussed next section)  
>>> plt.ylabel('residuals')
```

Correlations and heatmaps

- Notice that the residuals appear to be getting further away from zero at higher quantities of volume traded, which could mean this isn't the right way to model this relationship:



Regression plots

- Let's use `assign()` to create these new columns and save them in a new dataframe called `fb_reg_data`:

```
>>> fb_reg_data = fb.assign(  
...     log_volume=np.log(fb.volume),  
...     max_abs_change=fb.high - fb.low  
... ).iloc[:, -2:]
```

Regression plots

- When writing plotting functions, `itertools` can be extremely helpful; it makes it very easy to create efficient iterators for things such as permutations, combinations, and infinite cycles or repeats:

```
>>> import itertools
```

Regression plots

- The iterators we get back when using itertools can only be used once through:

```
>>> iterator = itertools.repeat("I'm an iterator", 1)
>>> for i in iterator:
...     print(f"-->{i}")
-->I'm an iterator
>>> print(
...     'This printed once because the iterator '
...     'has been exhausted'
... )
This printed once because the iterator has been exhausted
>>> for i in iterator:
...     print(f"-->{i}")
```

Regression plots

- A list, on the other hand, is an iterable; we can write something that loops over all the elements in the list, and we will still have a list for later reuse:

```
>>> iterable = list(itertools.repeat("I'm an iterable", 1))
>>> for i in iterable:
...     print(f"-->{i}")
-->I'm an iterable
>>> print('This prints again because it\'s an iterable:')
This prints again because it's an iterable:
>>> for i in iterable:
...     print(f"-->{i}")
-->I'm an iterable
```

Regression plots

- Now that we have some background on `itertools` and `iterators`, let's write the function for our regression and residuals permutation plots:

```
def reg_resid_plots(data):
    """
    Using `seaborn`, plot the regression and residuals plots
    side-by-side for every permutation of 2 columns in data.
    Parameters:
        - data: A `pandas.DataFrame` object
    Returns:
        A matplotlib `Axes` object.
    """
    num_cols = data.shape[1]
    permutation_count = num_cols * (num_cols - 1)
    fig, ax = \
        plt.subplots(permutation_count, 2, figsize=(15, 8))
    for (x, y), axes, color in zip(
        itertools.permutations(data.columns, 2),
        ax,
        itertools.cycle(['royalblue', 'darkorange'])
    ):
        for subplot, func in zip(
            axes, (sns.regplot, sns.residplot)
        ):
            func(x=x, y=y, data=data, ax=subplot, color=color)
            if func == sns.residplot:
                subplot.set_ylabel('residuals')
    return fig.axes
```

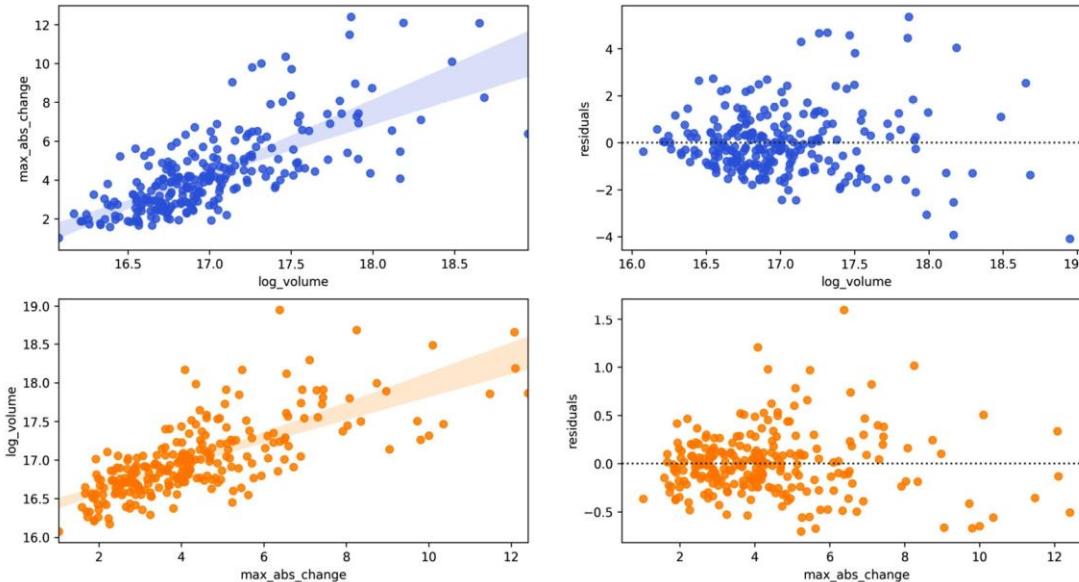
Regression plots

- Calling our function is effortless, with only a single parameter:

```
>>> from viz import reg_resid_plots  
>>> reg_resid_plots(fb_reg_data)
```

Regression plots

- The first row of subsets is what we saw earlier with the joint plots, and the second row is the regression when flipping the x and y variables:



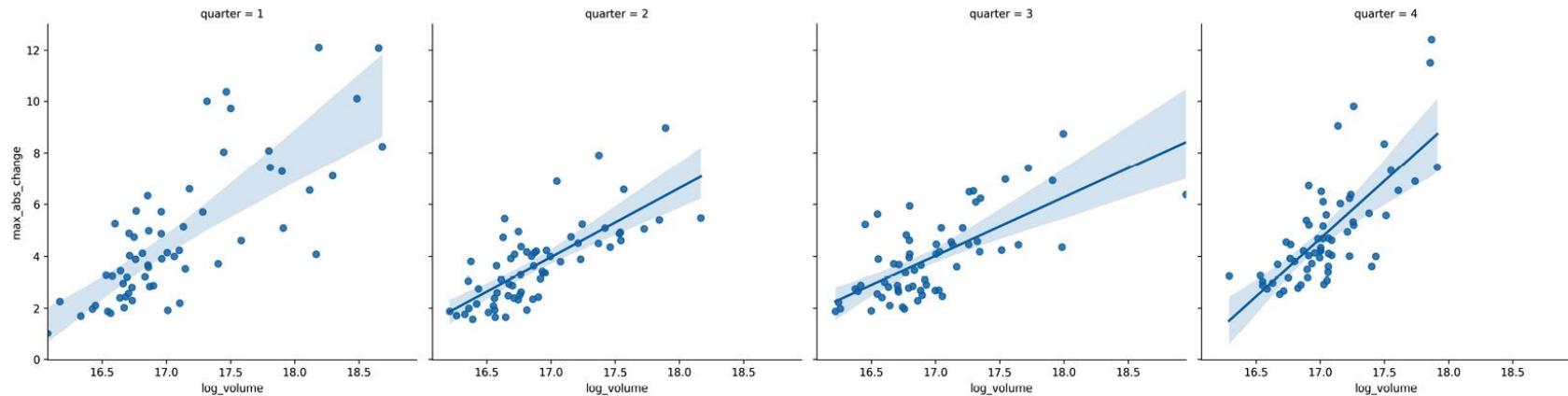
Regression plots

- We saw that Facebook's performance was different across each quarter of the year, so let's calculate a regression per quarter with the Facebook stock data, using the volume traded and the daily difference between the highest and lowest price, to see whether this relationship also changes:

```
>>> sns.lmplot(  
...     x='log_volume',  
...     y='max_abs_change',  
...     col='quarter',  
...     data=fb.assign(  
...         log_volume=np.log(fb.volume),  
...         max_abs_change=fb.high - fb.low,  
...         quarter=lambda x: x.index.quarter  
...     )  
... )
```

Regression plots

- Notice that the regression line in the fourth quarter has a much steeper slope than previous quarters:



Faceting

- First, we create a FacetGrid object with the data we will be using and define how it will be subset with the row and col arguments:

```
>>> g = sns.FacetGrid(  
...     quakes.query(  
...         'parsed_place.isin(''  
...             ["Indonesia", "Papua New Guinea"]) '  
...         'and magType == "mb"'  
...     ),  
...     row='tsunami',  
...     col='parsed_place',  
...     height=4  
... )
```

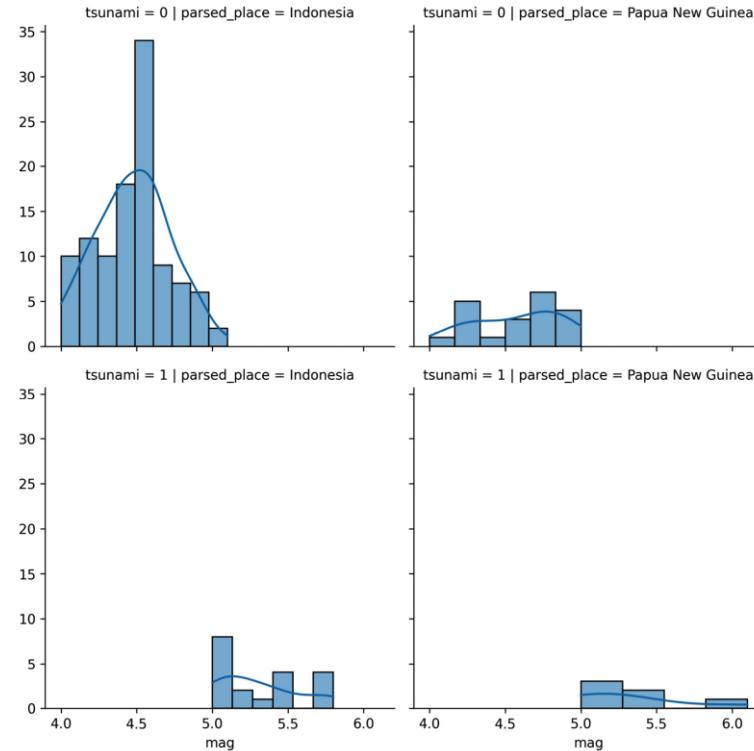
Faceting

- Then, we use the FacetGrid.map() method to run a plotting function on each of the subsets, passing along any necessary arguments.
- We will make histograms with KDEs for the location and tsunami data subsets using the sns.histplot() function:

```
>>> g = g.map(sns.histplot, 'mag', kde=True)
```

Faceting

- For both locations, we can see that tsunamis occurred when the earthquake magnitude was 5.0 or greater:



Formatting plots with matplotlib

- Let's now move to the 2-formatting_plots.ipynb notebook, run the setup code to import the packages we need, and read in the Facebook stock data and COVID-19 daily new cases data:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
>>> covid = pd.read_csv('data/covid19_cases.csv').assign(
...     date=lambda x: \
...         pd.to_datetime(x.dateRep, format='%d/%m/%Y')
... ).set_index('date').replace(
...     'United_States_of_America', 'USA'
... ).sort_index()['2020-01-18':'2020-09-18']
```

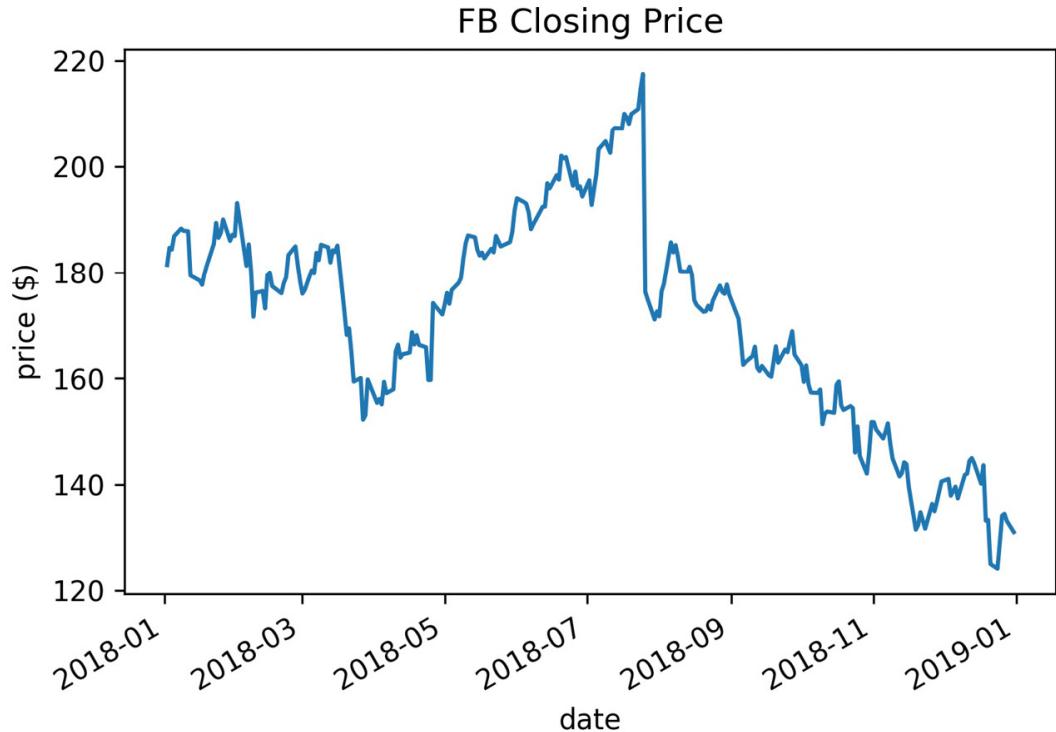
Titles and labels

- Let's plot the Facebook closing price and label everything using matplotlib:

```
>>> fb.close.plot()  
>>> plt.title('FB Closing Price')  
>>> plt.xlabel('date')  
>>> plt.ylabel('price ($)')
```

Titles and labels

- This results in the following plot:



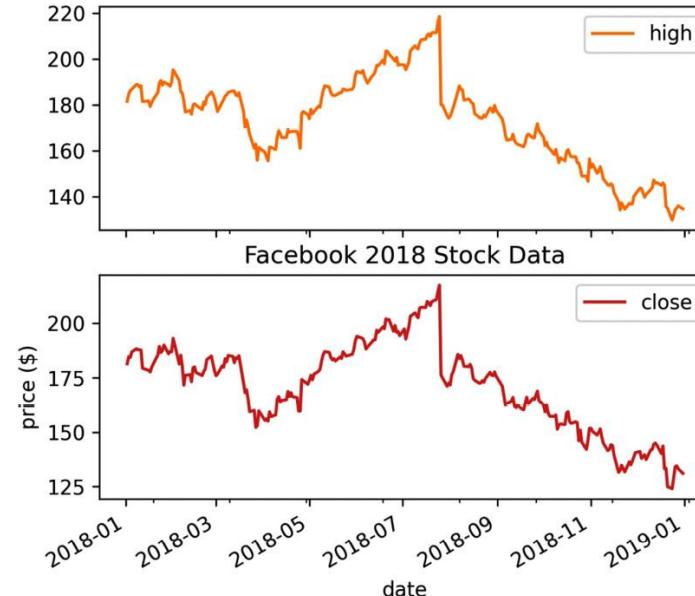
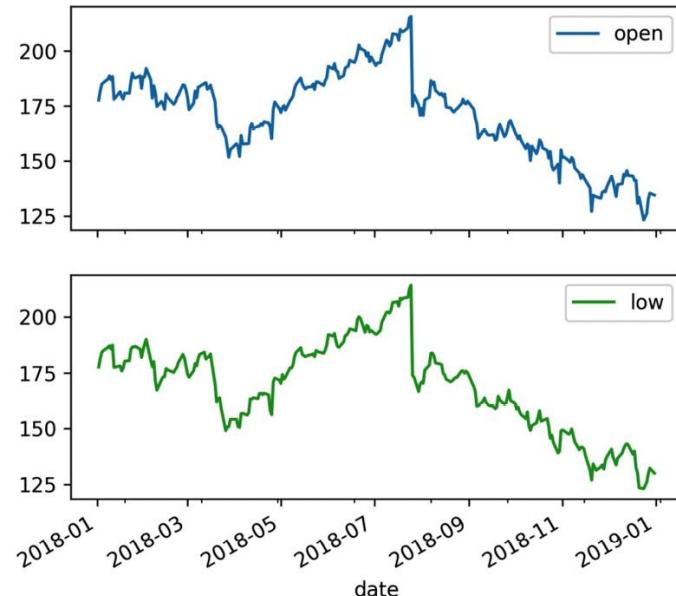
Titles and labels

- To see this firsthand, let's make subplots of Facebook stock's OHLC data and use plt.title() to give the entire plot a title, along with plt.ylabel() to give each subplot's y-axis a label:

```
>>> fb.iloc[:, :4]\  
...     .plot(subplots=True, layout=(2, 2), figsize=(12, 5))  
>>> plt.title('Facebook 2018 Stock Data')  
>>> plt.ylabel('price ($)')
```

Titles and labels

- The same thing happens to the y-axis label:



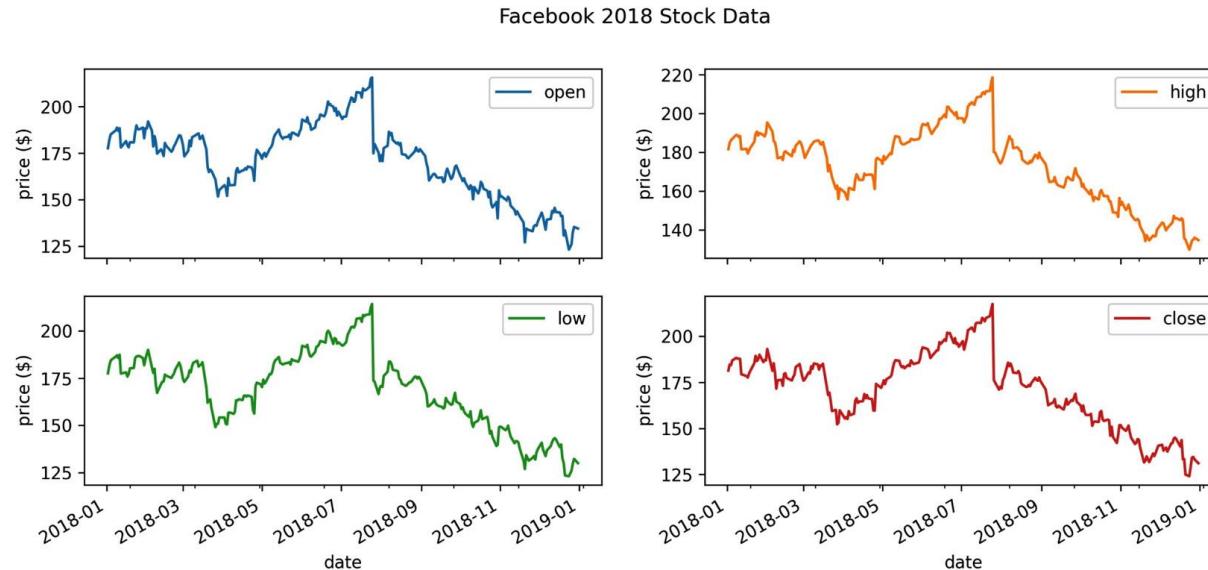
Titles and labels

- Note that the Axes objects are returned in a NumPy array of the same dimensions as the subplot layout, so for easier iteration, we call `flatten()`:

```
>>> axes = fb.iloc[:, :4]\  
...     .plot(subplots=True, layout=(2, 2), figsize=(12, 5))  
>>> plt.suptitle('Facebook 2018 Stock Data')  
>>> for ax in axes.flatten():  
...     ax.set_ylabel('price ($)')
```

Titles and labels

- This results in a title for the plot as a whole and y-axis labels for each of the subplots:



Legends

- Here is a sampling of some commonly used parameters:

Parameter	Purpose
<code>loc</code>	Specify the location of the legend
<code>bbox_to_anchor</code>	Used in conjunction with <code>loc</code> to specify legend location
<code>ncol</code>	Set the number of columns the labels will be broken into, default is 1
<code>framealpha</code>	Control the transparency of the legend's background
<code>title</code>	Give the legend a title

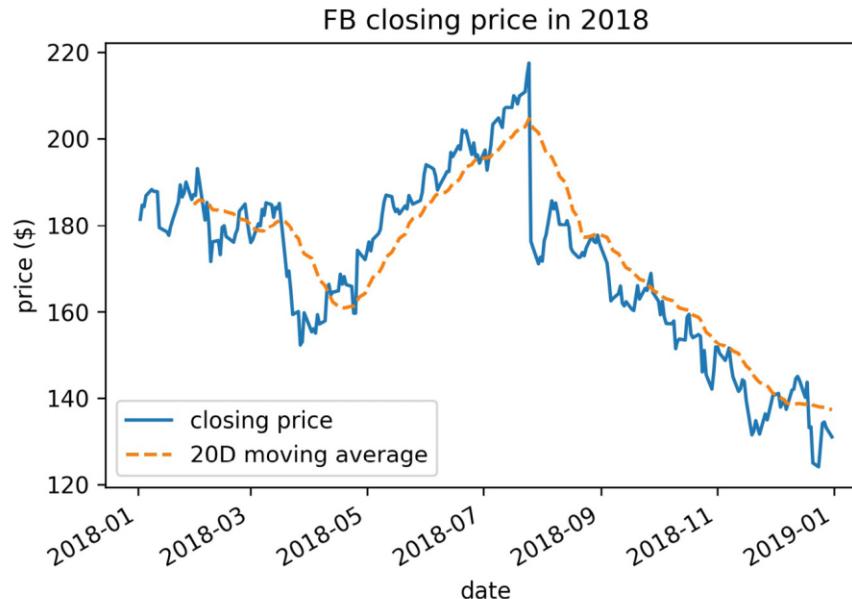
Legends

- Let's plot Facebook stock's closing price and the 20-day moving average, using the label argument to provide a descriptive name for the legend:

```
>>> fb.assign(  
...     ma=lambda x: x.close.rolling(20).mean()  
... ).plot(  
...     y=['close', 'ma'],  
...     title='FB closing price in 2018',  
...     label=['closing price', '20D moving average'],  
...     style=['-', '--']  
... )  
>>> plt.legend(loc='lower left')  
>>> plt.ylabel('price ($)')
```

Legends

- Note that the text in the legend is what we provided in the label argument to plot():



Legends

- The following table contains the possible location strings:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

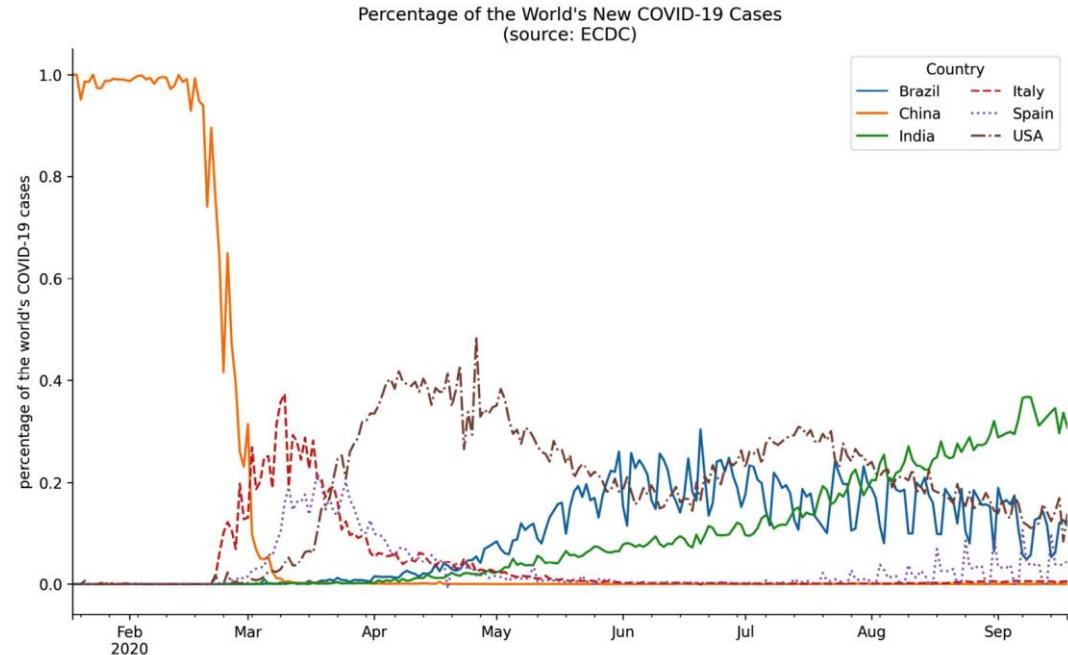
Legends

- In addition, we will remove the top and right spines of the plot to make it look cleaner:

```
>>> new_cases = covid.reset_index().pivot(  
...     index='date',  
...     columns='countriesAndTerritories',  
...     values='cases'  
... ).fillna(0)  
>>> pct_new_cases = new_cases.apply(  
...     lambda x: x / new_cases.apply('sum', axis=1), axis=0  
... )[  
...     ['Italy', 'China', 'Spain', 'USA', 'India', 'Brazil']  
... ].sort_index(axis=1).fillna(0)  
>>> ax = pct_new_cases.plot(  
...     figsize=(12, 7),  
...     style=['-' * 3 + [ '--', ':', '-.'],  
...     title='Percentage of the World\\\'s New COVID-19 Cases'  
...     '\n(source: ECDC)'  
... )  
>>> ax.legend(title='Country', framealpha=0.5, ncol=2)  
>>> ax.set_xlabel('')  
>>> ax.set_ylabel('percentage of the world\\\'s COVID-19 cases')  
>>> for spine in ['top', 'right']:  
...     ax.spines[spine].set_visible(False)
```

Legends

- Our legend is neatly arranged in two columns and contains a title.
- We also increased the transparency of the legend's border:



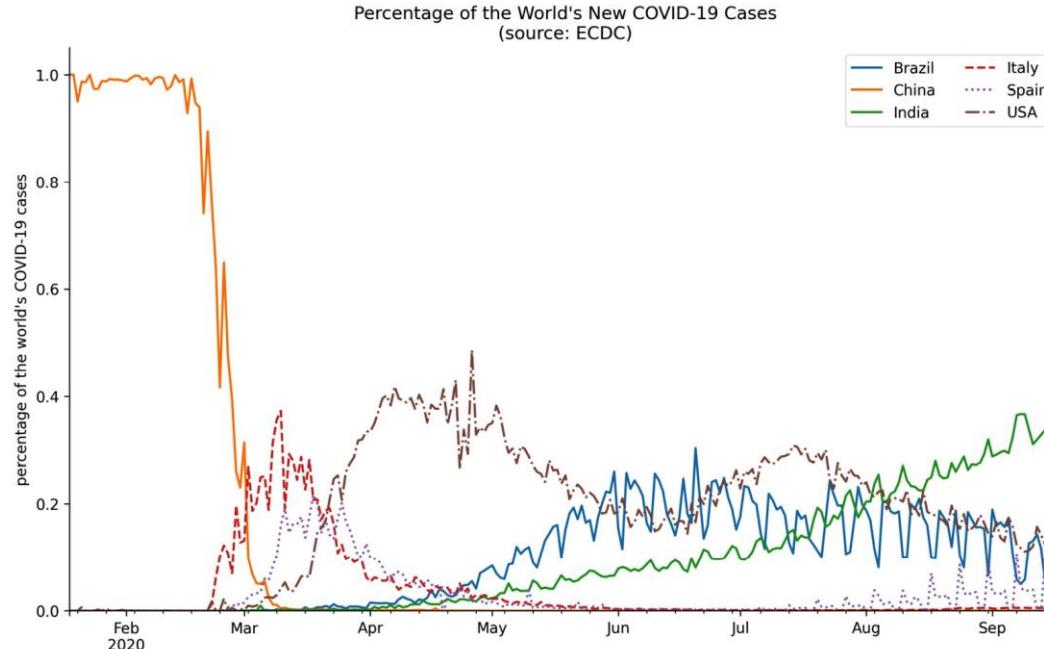
Formatting axes

- Let's modify the previous plot of the percentage of the world's daily new COVID-19 cases per country to start the y-axis at zero:

```
>>> ax = pct_new_cases.plot(  
...     figsize=(12, 7),  
...     style=['-' * 3 + ['--', ':', '-.'],  
...     title='Percentage of the World\\\'s New COVID-19 Cases'  
...             '\n(source: ECDC)'  
... )  
>>> ax.legend(framealpha=0.5, ncol=2)  
>>> ax.set_xlabel('')  
>>> ax.set_ylabel('percentage of the world\\\'s COVID-19 cases')  
>>> ax.set_ylim(0, None)  
>>> for spine in ['top', 'right']:  
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- Notice that the y-axis now begins at zero:



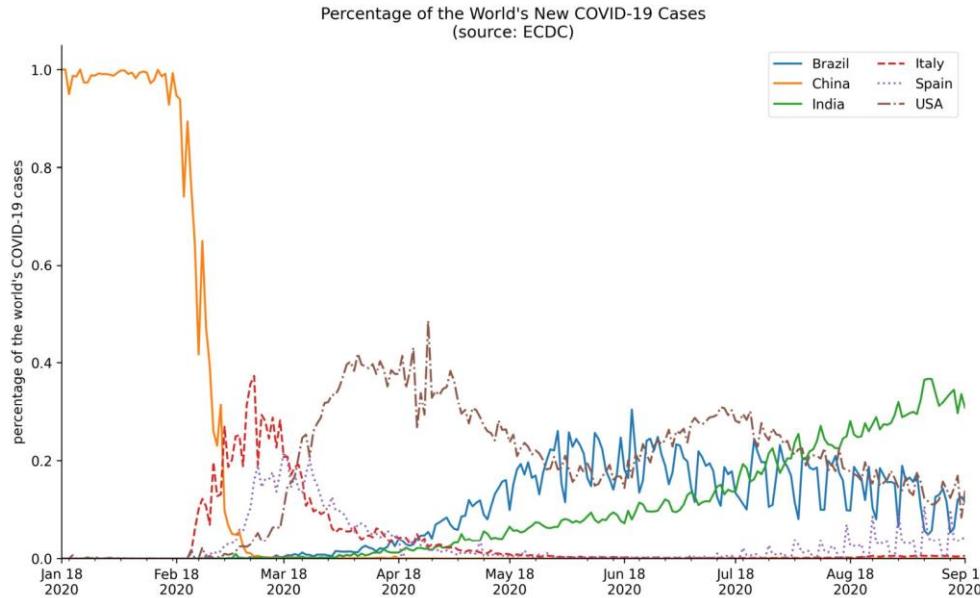
Formatting axes

- For example, since our data starts and ends on the 18th of the month, let's move the tick marks in the previous plot to the 18th of each month and then label the ticks accordingly:

```
>>> ax = pct_new_cases.plot(  
...     figsize=(12, 7),  
...     style=['-' * 3 + [ '--', ':', '-.'],  
...     title='Percentage of the World\'s New COVID-19 Cases'  
...         '\n(source: ECDC)'  
... )  
>>> tick_locs = covid.index[covid.index.day == 18].unique()  
>>> tick_labels = \  
...     [loc.strftime('%b %d\n%Y') for loc in tick_locs]  
>>> plt.xticks(tick_locs, tick_labels)  
>>> ax.legend(framealpha=0.5, ncol=2)  
>>> ax.set_xlabel('')  
>>> ax.set_ylabel('percentage of the world\'s COVID-19 cases')  
>>> ax.set_ylim(0, None)  
>>> for spine in ['top', 'right']:  
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- After moving the tick marks, we have a tick label on the first data point in the plot (January 18, 2020) and on the last (September 18, 2020):



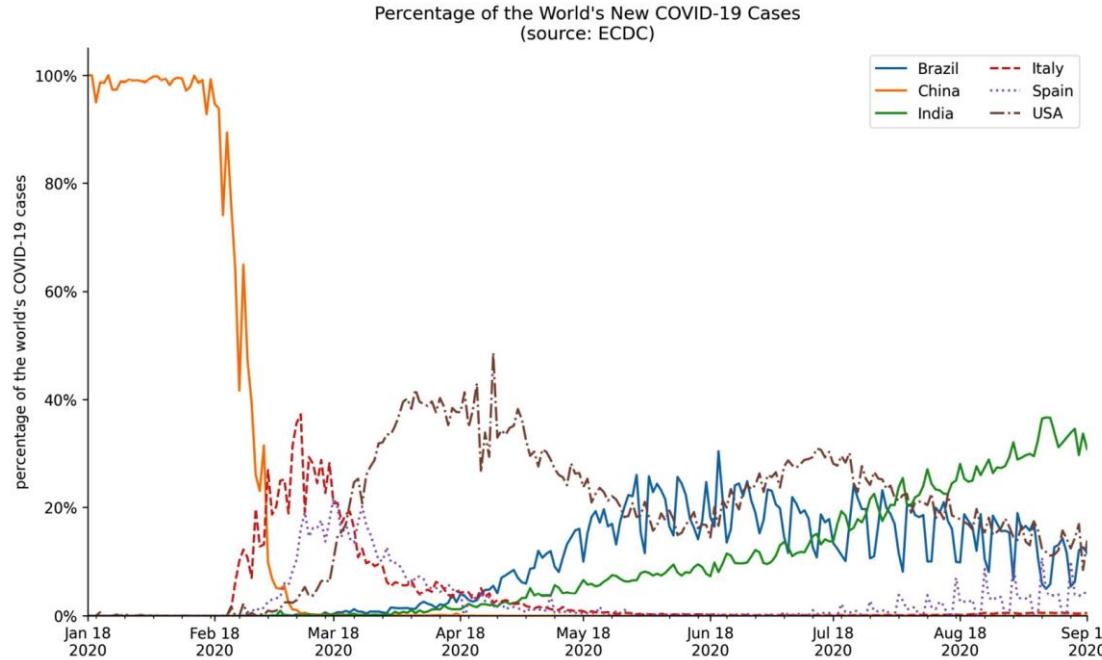
Formatting axes

- We can use the PercentFormatter class from the `matplotlib.ticker` module:

```
>>> from matplotlib.ticker import PercentFormatter
>>> ax = pct_new_cases.plot(
...     figsize=(12, 7),
...     style=['-' * 3 + [ '--', ':', '-.'],
...     title='Percentage of the World\'s New COVID-19 Cases'
...         '\n(source: ECDC)'
... )
>>> tick_locs = covid.index[covid.index.day == 18].unique()
>>> tick_labels = \
...     [loc.strftime('%b %d\n%Y') for loc in tick_locs]
>>> plt.xticks(tick_locs, tick_labels)
>>> ax.legend(framealpha=0.5, ncol=2)
>>> ax.set_xlabel('')
>>> ax.set_ylabel('percentage of the world\'s COVID-19 cases')
>>> ax.set_ylimit(0, None)
>>> ax.yaxis.set_major_formatter(PercentFormatter(xmax=1))
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- This results in percentages along the y-axis:



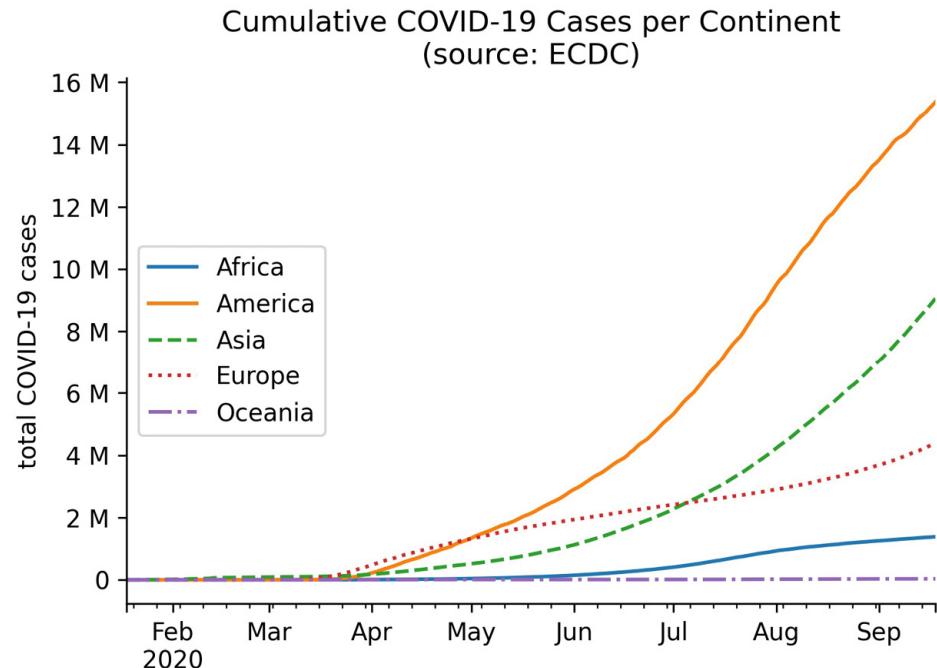
Formatting axes

- Let's use this to plot the cumulative COVID-19 cases per continent in millions:

```
>>> from matplotlib.ticker import EngFormatter
>>> ax = covid.query('continentExp != "Other"').groupby([
...     'continentExp', pd.Grouper(freq='1D')
... ]).cases.sum().unstack(0).apply('cumsum').plot(
...     style=['-', '--', ':', '-.'],
...     title='Cumulative COVID-19 Cases per Continent'
...     '\n(source: ECDC)'
... )
>>> ax.legend(title='', loc='center left')
>>> ax.set(xlabel='', ylabel='total COVID-19 cases')
>>> ax.yaxis.set_major_formatter(EngFormatter())
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- Notice that we didn't need to divide the cumulative case counts by 1 million to get these numbers—the EngFormatter object that we passed to `set_major_formatter()` automatically figured out that it should use millions (M) based on the data:



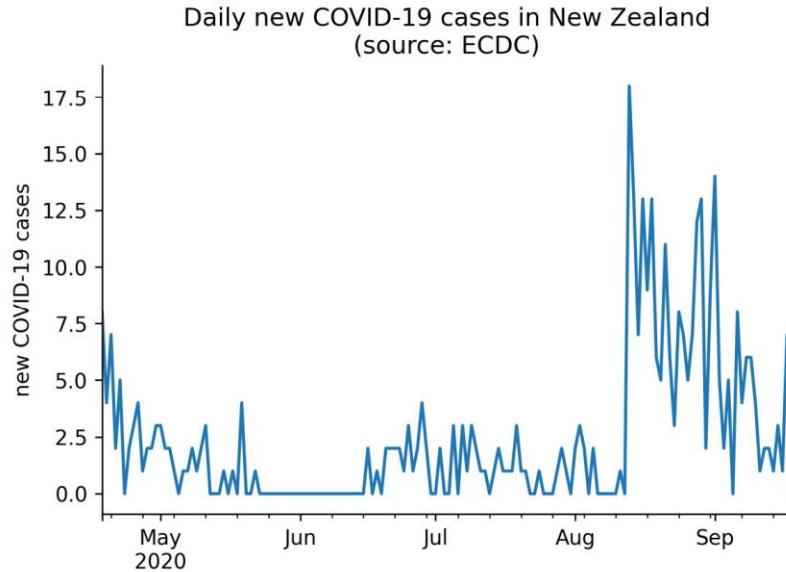
Formatting axes

- To illustrate how we could use this, let's take a look at the daily new COVID-19 cases in New Zealand from April 18, 2020 through September 18, 2020:

```
>>> ax = new_cases.New_Zealand['2020-04-18':'2020-09-18'].plot(  
...     title='Daily new COVID-19 cases in New Zealand'  
...     '\n(source: ECDC)'  
... )  
>>> ax.set(xlabel='', ylabel='new COVID-19 cases')  
>>> for spine in ['top', 'right']:  
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- We know that there is no such thing as half of a case, so it makes more sense to show this data with only integer ticks:



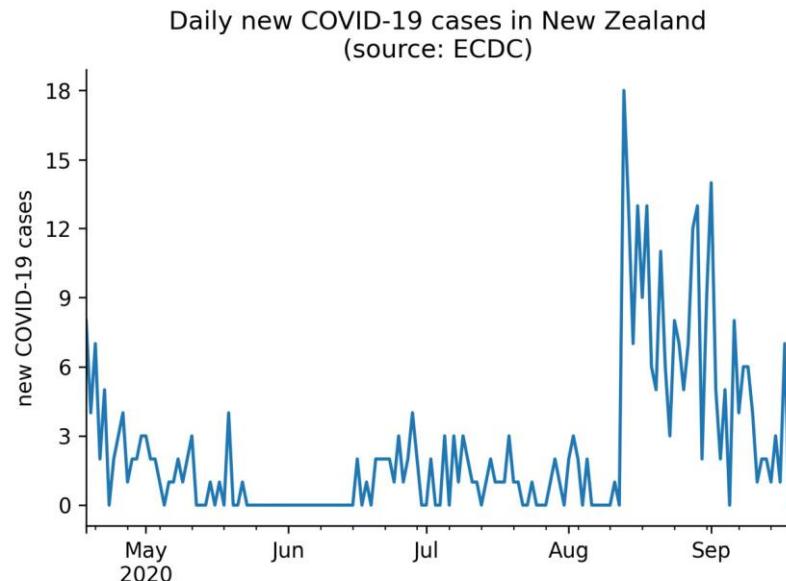
Formatting axes

- we have to call the `set_major_locator()` method instead of `set_major_formatter()`:

```
>>> from matplotlib.ticker import MultipleLocator  
>>> ax = new_cases.New_Zealand['2020-04-18':'2020-09-18'].plot(  
...     title='Daily new COVID-19 cases in New Zealand'  
...     '\n(source: ECDC)'  
... )  
>>> ax.set(xlabel='', ylabel='new COVID-19 cases')  
>>> ax.yaxis.set_major_locator(MultipleLocator(base=3))  
>>> for spine in ['top', 'right']:  
...     ax.spines[spine].set_visible(False)
```

Formatting axes

- Since we passed in base=3, our y-axis now contains integers in increments of three:



Customizing visualizations

- In the 3-customizing_visualizations.ipynb notebook, let's handle our imports and read in the Facebook stock prices and earthquake datasets:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> fb = pd.read_csv(
...     'data/fb_stock_prices_2018.csv',
...     index_col='date',
...     parse_dates=True
... )
>>> quakes = pd.read_csv('data/earthquakes.csv')
```

Adding reference lines

- We simply need to create an instance of the StockAnalyzer class to calculate these metrics:

```
>>> from stock_analysis import StockAnalyzer  
>>> fb_analyzer = StockAnalyzer(fb)  
>>> support, resistance = (  
...     getattr(fb_analyzer, stat)(level=3)  
...     for stat in ['support', 'resistance']  
... )  
>>> support, resistance  
(124.45666666666667, 138.52666666666667)
```

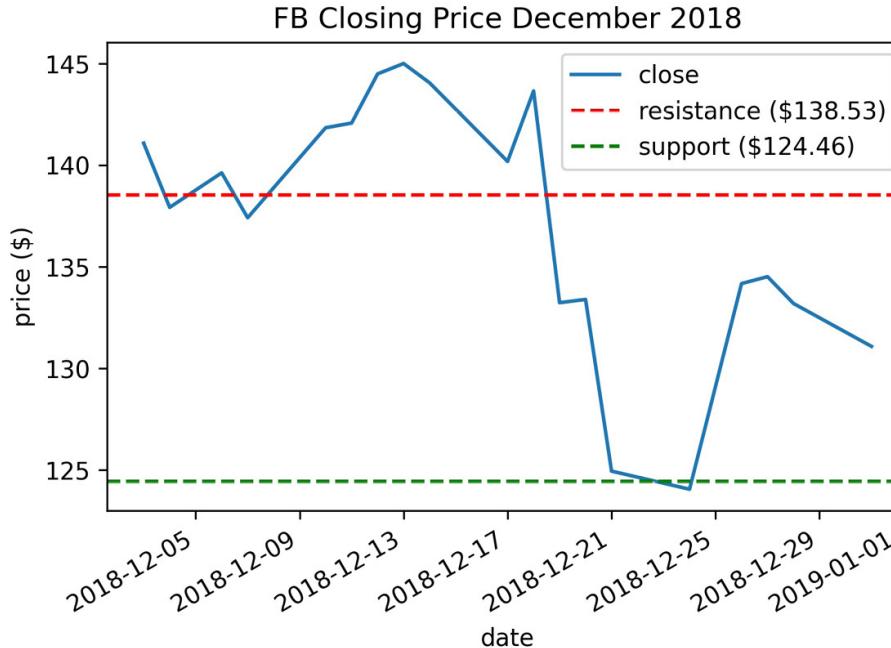
Adding reference lines

- Remember that the text we provide to the label arguments will be populated in the legend:

```
>>> fb.close['2018-12']\n...     .plot(title='FB Closing Price December 2018')\n>>> plt.axhline(\n...     y=resistance, color='r', linestyle='--',\n...     label=f'resistance (${resistance:.2f})'\n... )\n>>> plt.axhline(\n...     y=support, color='g', linestyle='--',\n...     label=f'support (${support:.2f})'\n... )\n>>> plt.ylabel('price ($)')\n>>> plt.legend()
```

Adding reference lines

- This formatting makes for an informative legend in our plot:



Adding reference lines

- The `std_from_mean_kde()` function located in the `viz.py` module in the GitHub repository uses `itertools` to easily make the combinations of the colors and values we need to plot:

```
import itertools
def std_from_mean_kde(data):
    """
    Plot the KDE along with vertical reference lines
    for each standard deviation from the mean.
    Parameters:
        - data: `pandas.Series` with numeric data
    Returns:
        Matplotlib `Axes` object.
    """
    mean_mag, std_mean = data.mean(), data.std()
    ax = data.plot(kind='kde')
    ax.axvline(mean_mag, color='b', alpha=0.2, label='mean')
    colors = ['green', 'orange', 'red']
    multipliers = [1, 2, 3]
    signs = ['-','+']
    linestyles = [':', '-.', '--']
    for sign, (color, multiplier, style) in itertools.product(
        signs, zip(colors, multipliers, linestyles)
    ):
        adjustment = multiplier * std_mean
        if sign == '-':
            value = mean_mag - adjustment
            label = '{} {}{}{}{}'.format(
                r'\mu$', r'\pm$', multiplier, r'\sigma$'
            )
        else:
            value = mean_mag + adjustment
            label = None # label each color only once
        ax.axvline(
            value, color=color, linestyle=style,
            label=label, alpha=0.5
        )
    ax.legend()
    return ax
```

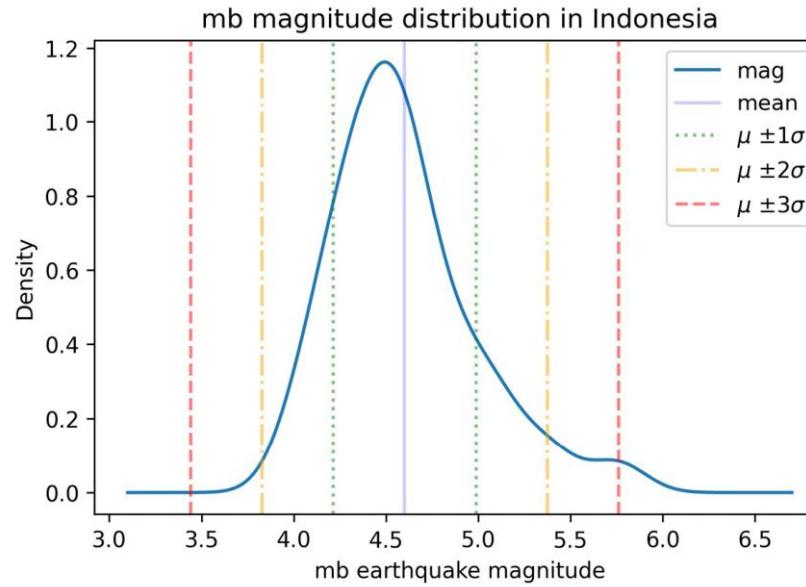
Adding reference lines

- Let's use the std_from_mean_kde() function to see which parts of the estimated distribution of earthquake magnitudes in Indonesia are within one, two, or three standard deviations from the mean:

```
>>> from viz import std_from_mean_kde
>>> ax = std_from_mean_kde(
...     quakes.query(
...         'magType == "mb" and parsed_place == "Indonesia"'
...     ).mag
... )
>>> ax.set_title('mb magnitude distribution in Indonesia')
>>> ax.set_xlabel('mb earthquake magnitude')
```

Adding reference lines

- Notice the KDE is right-skewed—it has a longer tail on the right side, and the mean is to the right of the mode:



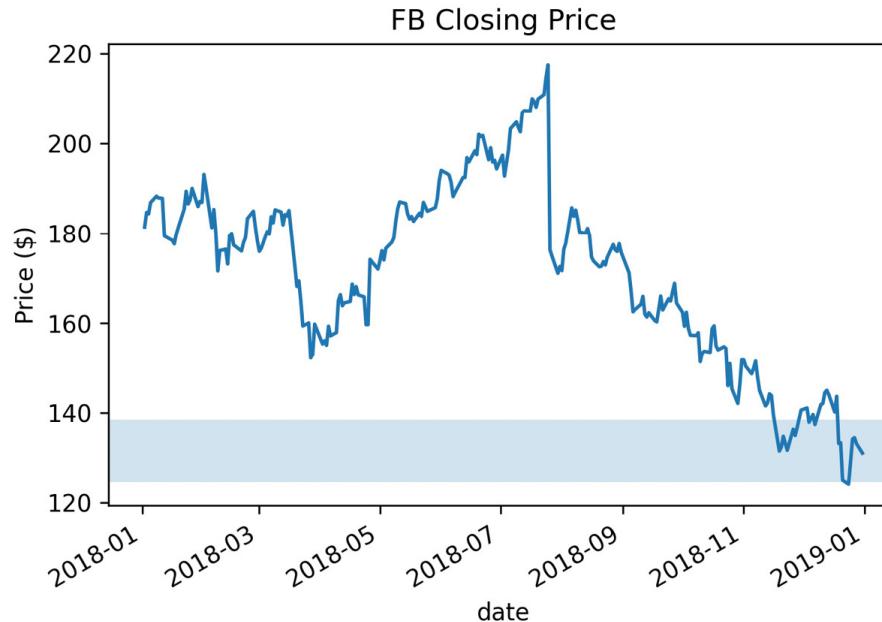
Shading regions

- We can use `axhspan()` to shade the area that falls between the two:

```
>>> ax = fb.close.plot(title='FB Closing Price')
>>> ax.axhspan(support, resistance, alpha=0.2)
>>> plt.ylabel('Price ($)')
```

Shading regions

- For this example, we accepted the default:



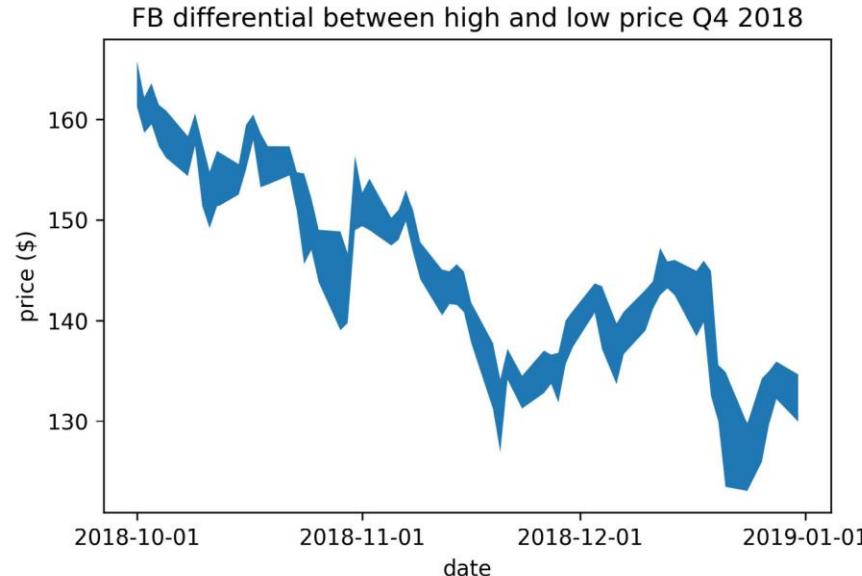
Shading regions

- Let's shade the area between Facebook's high price and low price each day of the fourth quarter using plt.fill_between():

```
>>> fb_q4 = fb.loc['2018-Q4']
>>> plt.fill_between(fb_q4.index, fb_q4.high, fb_q4.low)
>>> plt.xticks([
...     '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'
... ])
>>> plt.xlabel('date')
>>> plt.ylabel('price ($)')
>>> plt.title(
...     'FB differential between high and low price Q4 2018'
... )
```

Shading regions

- This gives us a better idea of the variation in price on a given day; the taller the vertical distance, the higher the fluctuation:



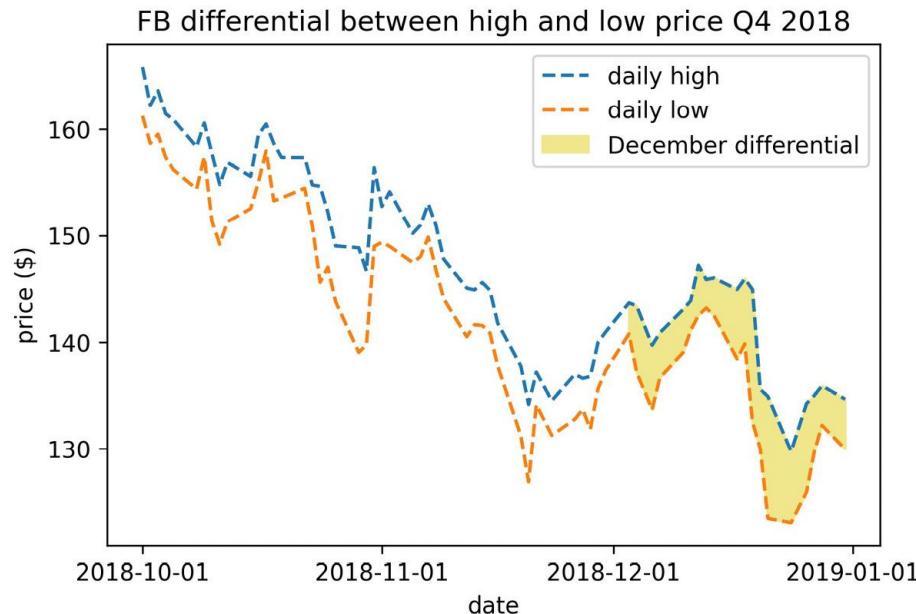
Shading regions

- We will add dashed lines for the high price curve and the low price curve throughout the time period to see what is happening:

```
>>> fb_q4 = fb.loc['2018-Q4']
>>> plt.fill_between(
...     fb_q4.index, fb_q4.high, fb_q4.low,
...     where=fb_q4.index.month == 12,
...     color='khaki', label='December differential'
... )
>>> plt.plot(fb_q4.index, fb_q4.high, '--', label='daily high')
>>> plt.plot(fb_q4.index, fb_q4.low, '--', label='daily low')
>>> plt.xticks([
...     '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'
... ])
>>> plt.xlabel('date')
>>> plt.ylabel('price ($)')
>>> plt.legend()
>>> plt.title(
...     'FB differential between high and low price Q4 2018'
... )
```

Shading regions

- This results in the following plot:



Annotations

- For example, let's use the `plt.annotate()` function to label the support and resistance:

```
>>> ax = fb.close.plot(  
...      title='FB Closing Price 2018',  
...      figsize=(15, 3)  
... )  
>>> ax.set_ylabel('price ($)')  
>>> ax.axhspan(support, resistance, alpha=0.2)  
>>> plt.annotate(  
...      f'support\nn(${support:.2f})',  
...      xy=('2018-12-31', support),  
...      xytext=('2019-01-21', support),  
...      arrowprops={'arrowstyle': '->'}  
... )  
>>> plt.annotate(  
...      f'resistance\nn(${resistance:.2f})',  
...      xy=('2018-12-23', resistance)  
... )  
>>> for spine in ['top', 'right']:  
...      ax.spines[spine].set_visible(False)
```

Annotations

- By doing so, we avoid obscuring the last few days of data with our label:



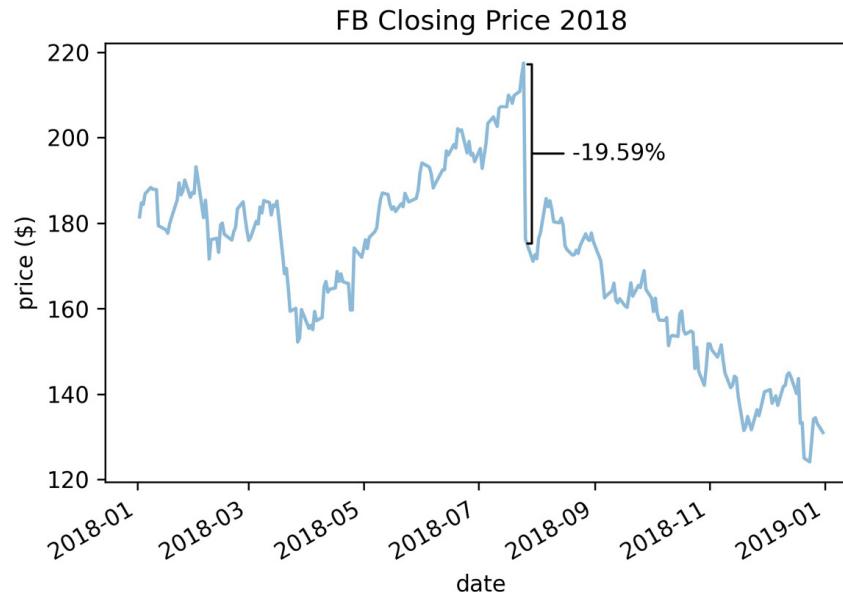
Annotations

- As an example, let's annotate the big decline in the price of Facebook in July with the percentage drop:

```
>>> close_price = fb.loc['2018-07-25', 'close']
>>> open_price = fb.loc['2018-07-26', 'open']
>>> pct_drop = (open_price - close_price) / close_price
>>> fb.close.plot(title='FB Closing Price 2018', alpha=0.5)
>>> plt.annotate(
...     f'{pct_drop:.2%}', va='center',
...     xy=('2018-07-27', (open_price + close_price) / 2),
...     xytext=('2018-08-20', (open_price + close_price) / 2),
...     arrowprops=dict(arrowstyle='-[,widthB=4.0,lengthB=0.2')
... )
>>> plt.ylabel('price ($)')
```

Annotations

- In addition, by specifying `va='center'`, we tell matplotlib to vertically center our annotation in the middle of the arrow:



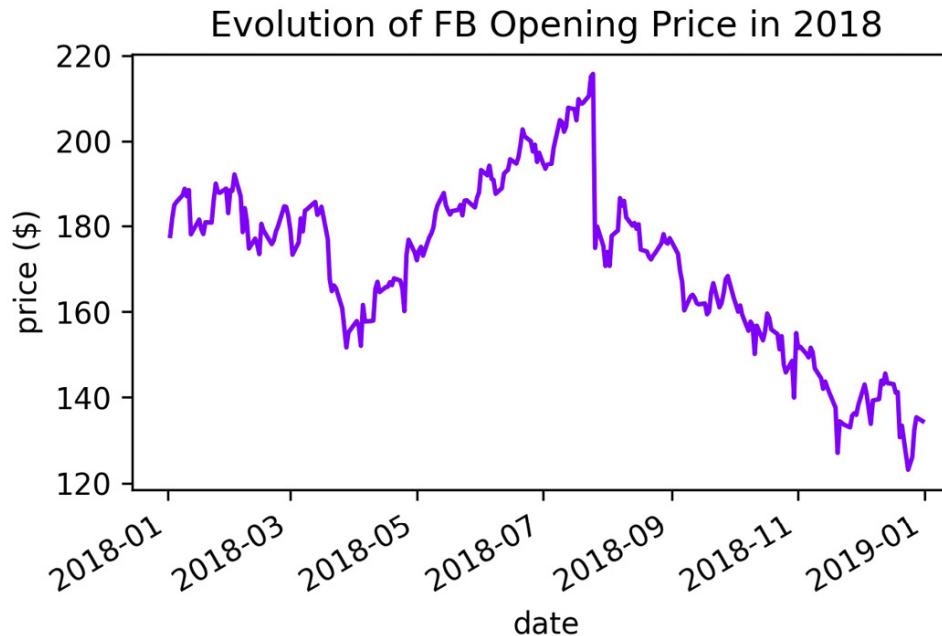
Colors

- Matplotlib accepts hex codes as a string to the color argument.
To illustrate this, let's plot Facebook's opening price in #8000FF:

```
>>> fb.plot(  
...     y='open',  
...     figsize=(5, 3),  
...     color='#8000FF',  
...     legend=False,  
...     title='Evolution of FB Opening Price in 2018'  
... )  
>>> plt.ylabel('price ($)')
```

Colors

- This results in a purple line plot:



Colors

- The following code is equivalent to the preceding example, except we use the RGB tuple instead of the hex code:

```
fb.plot(  
    y='open',  
    figsize=(5, 3),  
    color=(128 / 255, 0, 1),  
    legend=False,  
    title='Evolution of FB Opening Price in 2018'  
)  
plt.ylabel('price ($)')
```

Colormaps

- There are three types of colormaps, each with its own purpose, as shown in the following table:

Class	Purpose
Qualitative	No ordering or relationship between colors; just used to distinguish between groups
Sequential	For information with ordering, such as temperature
Diverging	There is a middle value between two extremes that has meaning; for example, correlation coefficients are bounded in the range $[-1, 1]$, and 0 has meaning (no correlation)

Colormaps

- In Python, we can obtain a list of all the available colormaps by running the following:

```
>>> from matplotlib import cm  
>>> cm.datad.keys()  
dict_keys(['Blues', 'BrBG', 'BuGn', 'BuPu', 'CMRmap', 'GnBu',  
          'Greens', 'Greys', 'OrRd', 'Oranges', 'PRGn',  
          'PiYG', 'PuBu', 'PuBuGn', 'PuOr', 'PuRd', 'Purples',  
          'RdBu', 'RdGy', 'RdPu', 'RdYlBu', 'RdYlGn',  
          'Reds', ..., 'Blues_r', 'BrBG_r', 'BuGn_r', ...])
```

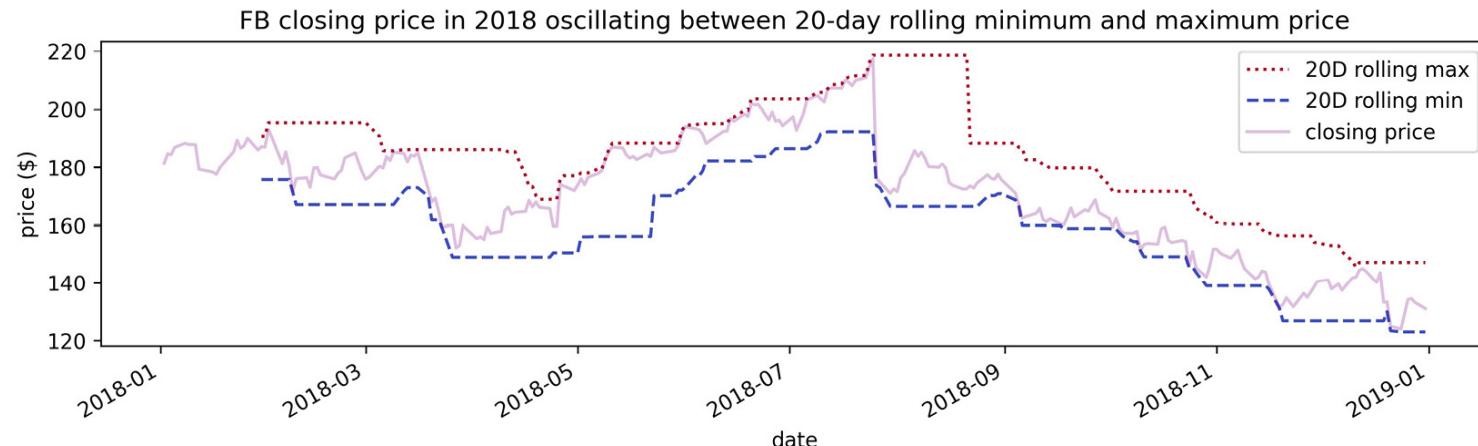
Colormaps

- Let's use the coolwarm_r colormap to show how Facebook stock's closing price oscillates between the 20-day rolling minimum and maximum prices:

```
>>> ax = fb.assign(  
...     rolling_min=lambda x: x.low.rolling(20).min(),  
...     rolling_max=lambda x: x.high.rolling(20).max()  
... ).plot(  
...     y=['rolling_max', 'rolling_min'],  
...     colormap='coolwarm_r',  
...     label=['20D rolling max', '20D rolling min'],  
...     style=[':', '--'],  
...     figsize=(12, 3),  
...     title='FB closing price in 2018 oscillating between '  
...         '20-day rolling minimum and maximum price'  
... )  
>>> ax.plot(  
...     fb.close, 'purple', alpha=0.25, label='closing price'  
... )  
>>> plt.legend()  
>>> plt.ylabel('price ($)')
```

Colormaps

- Notice how easy it was to get red to represent hot performance (rolling maximum) and blue for cold (rolling minimum), by using the reversed colormap, rather than trying to make sure pandas plotted the rolling minimum first:



Colormaps

- For example, we can ask for the midpoint of the ocean colormap to use with the color argument:

```
>>> cm.get_cmap('ocean')(.5)  
(0.0, 0.2529411764705882, 0.5019607843137255, 1.0)
```

Colormaps

- We can import the functions like this if we are running Python from the same directory as the file:

```
>>> import color_utils
```

Colormaps

- First, we need to translate these hex colors to their RGB equivalents

```
import re
def hex_to_rgb_color_list(colors):
    """
    Take color or list of hex code colors and convert them
    to RGB colors in the range [0,1].
    Parameters:
        - colors: Color or list of color strings as hex codes
    Returns:
        The color or list of colors in RGB representation.
    """
    if isinstance(colors, str):
        colors = [colors]
    for i, color in enumerate(
        [color.replace('#', '') for color in colors]
    ):
        hex_length = len(color)
        if hex_length not in [3, 6]:
            raise ValueError(
                'Colors must be of the form #FFFFFF or #FFF'
            )
        regex = '.' * (hex_length // 3)
        colors[i] = [
            int(val * (6 // hex_length), 16) / 255
            for val in re.findall(regex, color)
        ]
    return colors[0] if len(colors) == 1 else colors
```

Colormaps

- This is what the blended_cmap() function does:

```
from matplotlib.colors import ListedColormap
import numpy as np
def blended_cmap(rgb_color_list):
    """
    Create a colormap blending from one color to the other.

    Parameters:
        - rgb_color_list: List of colors represented as
            [R, G, B] values in the range [0, 1], like
            [[0, 0, 0], [1, 1, 1]], for black and white.

    Returns:
        A matplotlib `ListedColormap` object
    """
    if not isinstance(rgb_color_list, list):
        raise ValueError('Colors must be passed as a list.')
    elif len(rgb_color_list) < 2:
        raise ValueError('Must specify at least 2 colors.')
    elif (
        not isinstance(rgb_color_list[0], list)
        or not isinstance(rgb_color_list[1], list)
    ) or (
```

Colormaps

```
) or (
    (len(rgb_color_list[0]) != 3
     or len(rgb_color_list[1]) != 3)
):
    raise ValueError(
        'Each color should be a list of size 3.')
)
N, entries = 256, 4 # red, green, blue, alpha
rgbas = np.ones((N, entries))
segment_count = len(rgb_color_list) - 1
segment_size = N // segment_count
remainder = N % segment_count # need to add this back later
for i in range(entries - 1): # we don't alter alphas
    updates = []
    for seg in range(1, segment_count + 1):
        # handle uneven splits due to remainder
        offset = 0 if not remainder or seg > 1 \
                  else remainder
        updates.append(np.linspace(
            start=rgb_color_list[seg - 1][i],
            stop=rgb_color_list[seg][i],
            num=segment_size + offset
        ))
    rgbas[:, i] = np.concatenate(updates)
return ListedColormap(rgbas)
```

Colormaps

- We can use the `draw_cmap()` function to draw a colorbar, which allows us to visualize our colormap:

```
import matplotlib.pyplot as plt
def draw_cmap(cmap, values=np.array([[0, 1]]), **kwargs):
    """
    Draw a colorbar for visualizing a colormap.

    Parameters:
        - cmap: A matplotlib colormap
        - values: Values to use for the colormap
        - kwargs: Keyword arguments to pass to `plt.colorbar()`

    Returns:
        A matplotlib `Colorbar` object, which you can save
        with: `plt.savefig(<file_name>, bbox_inches='tight')`
    """
    img = plt.imshow(values, cmap=cmap)
    cbar = plt.colorbar(**kwargs)
    img.axes.remove()
    return cbar
```

Colormaps

- We will be using them by importing the module (which we did earlier):

```
>>> my_colors = ['#800080', '#FFA500', '#FFFF00']
>>> rgbs = color_utils.hex_to_rgb_color_list(my_colors)
>>> my_cmap = color_utils.blended_cmap(rgbs)
>>> color_utils.draw_cmap(my_cmap, orientation='horizontal')
```

Colormaps

- This results in the following colorbar showing our colormap:



Colormaps

- We used this technique earlier in the lesson to define our own colors for the regression residuals plots:

```
>>> import itertools  
>>> colors = itertools.cycle(['#ffffff', '#f0f0f0', '#000000'])  
>>> colors  
<itertools.cycle at 0x1fe4f300>  
>>> next(colors)  
'#ffffff'
```



Colormaps

- This just goes to show that we can find many ways to do something in Python; we have to find the implementation that best meets our needs:

```
from my_plotting_module import master_color_list  
def color_generator():  
    yield from master_color_list
```



Colormaps

- Using matplotlib, the alternative would be to instantiate a ListedColormap object with the color list and define a large value for N so that it repeats for long enough (if we don't provide it, it will only go through the colors once):

```
>>> from matplotlib.colors import ListedColormap  
>>> red_black = ListedColormap(['red', 'black'], N=2000)  
>>> [red_black(i) for i in range(3)]  
[(1.0, 0.0, 0.0, 1.0),  
 (0.0, 0.0, 0.0, 1.0),  
 (1.0, 0.0, 0.0, 1.0)]
```

Conditional coloring

- We certainly don't want to keep a list this size in memory, so we create a generator to calculate the color on demand:

```
def color_generator():
    for year in range(1992, 200019): # integers [1992, 200019)
        if year % 100 == 0 and year % 400 != 0:
            # special case (divisible by 100 but not 400)
            color = '#f0f0f0'
        elif year % 4 == 0:
            # leap year (divisible by 4)
            color = '#000000'
        else:
            color = '#ffffff'
        yield color
```

Conditional coloring

- Since we defined `year_colors` as a generator, Python will remember where we are in this function and resume when `next()` is called:

```
>>> year_colors = color_generator()  
>>> year_colors  
<generator object color_generator at 0x7bef148dfed0>  
>>> next(year_colors)  
'#000000'
```



Conditional coloring

- Simpler generators can be written with generator expressions.
- For example, if we don't care about the special case anymore, we can use the following:

```
>>> year_colors = (
...     '#ffffff'
...     if (not year % 100 and year % 400) or year % 4
...     else '#000000' for year in range(1992, 200019)
... )
>>> year_colors
<generator object <genexpr> at 0x7bef14415138>
>>> next(year_colors)
'#000000'
```

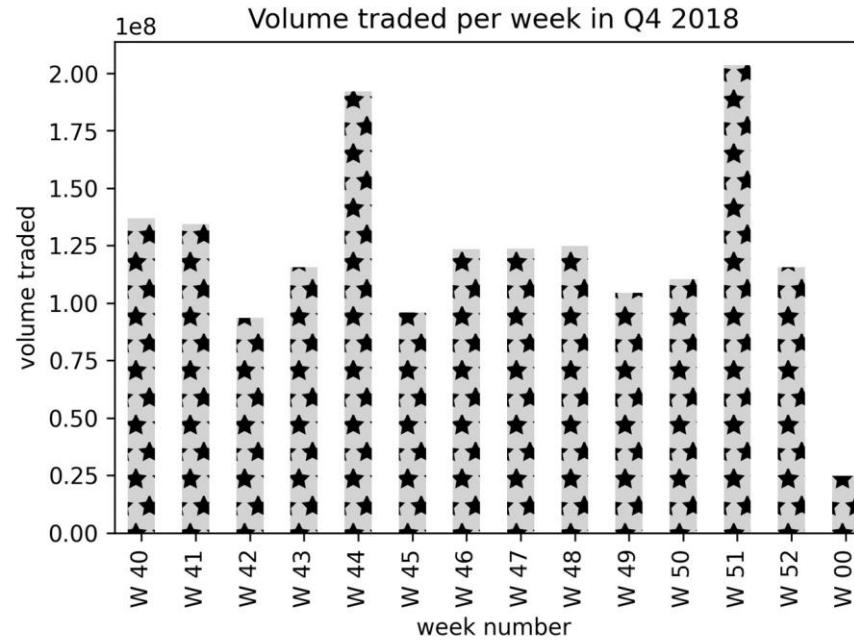
Textures

- Let's create a bar plot of weekly volume traded in Facebook stock during Q4 2018 with textured bars:

```
>>> weekly_volume_traded = fb.loc['2018-Q4']\n...     .groupby(pd.Grouper(freq='W')).volume.sum()\n>>> weekly_volume_traded.index = \
...     weekly_volume_traded.index.strftime('W %W')\n>>> ax = weekly_volume_traded.plot(\n...     kind='bar',\n...     hatch='*',\n...     color='lightgray',\n...     title='Volume traded per week in Q4 2018'\n... )\n>>> ax.set(\n...     xlabel='week number',\n...     ylabel='volume traded'\n... )
```

Textures

- With hatch='*', our bars are filled with stars:



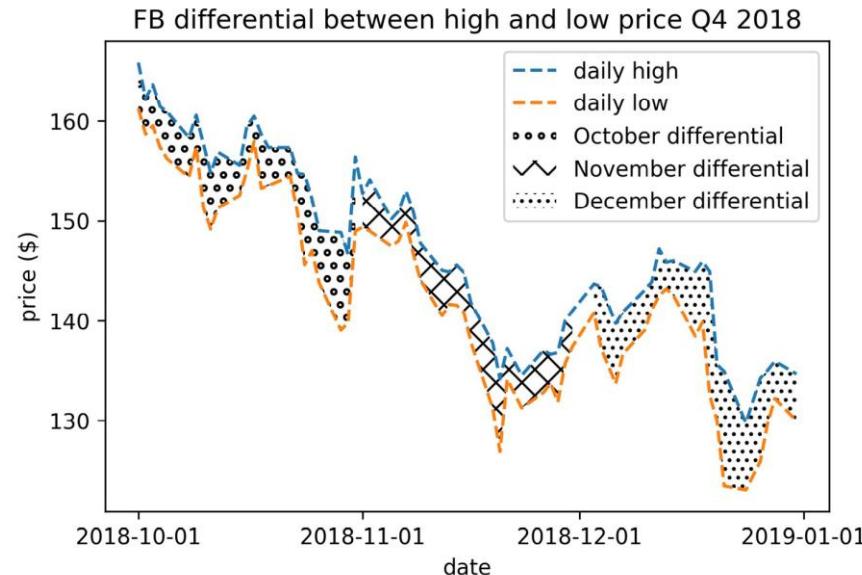
Textures

- This time we will use textures to distinguish between each month, rather than only shading December; we will fill October with rings, November with slashes, and December with small dots:

```
>>> import calendar
>>> fb_q4 = fb.loc['2018-Q4']
>>> for texture, month in zip(
...     ['oo', '/\\\/\\\'', '....'], [10, 11, 12]
... ):
...     plt.fill_between(
...         fb_q4.index, fb_q4.high, fb_q4.low,
...         hatch=texture, facecolor='white',
...         where=fb_q4.index.month == month,
...         label=f'{calendar.month_name[month]} differential'
...     )
>>> plt.plot(fb_q4.index, fb_q4.high, '--', label='daily high')
>>> plt.plot(fb_q4.index, fb_q4.low, '--', label='daily low')
>>> plt.xticks([
...     '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'
... ])
>>> plt.xlabel('date')
>>> plt.ylabel('price ($)')
>>> plt.title(
...     'FB differential between high and low price Q4 2018'
... )
>>> plt.legend()
```

Textures

- To achieve the small dots for December, we used three periods—the more we add, the denser the texture becomes:



Summary

- We learned how to create impressive and customized visualizations using matplotlib, pandas, and seaborn.
- We discussed how we can use seaborn for additional plotting types and cleaner versions of some familiar ones.



"Complete Exercises"

"Complete Lab 11"

12: Financial Analysis – Bitcoin and the Stock Market



Financial Analysis – Bitcoin and the Stock Market

The following topics will be covered in this lesson:

- Building a Python package
- Collecting financial data
- Conducting exploratory data analysis
- Performing technical analysis of financial instruments
- Modeling performance using historical data

lesson materials

- Install from GitHub if we don't plan on editing the source code for our own use.
- Fork and clone the repository and then install it on our machine in order to modify the code.

lesson materials

- we would do the following to install packages from GitHub:

```
(course_env) $ pip3 install \  
git+https://github.com/fenago
```



lesson materials

- Note that this will clone the latest version of the package, which may be different from the version in the text (the version with the 2nd_edition tag):

```
(course_env) $ git clone \
git@github.com:fenango/stock-analysis.git
(course_env) $ pip3 install -r stock-analysis/requirements.txt
(course_env) $ pip3 install -e stock-analysis
```

Building a Python package

- Building packages is considered good coding practice since it allows for writing modular code and reuse.
- Modular code is code that is written in many smaller pieces for more pervasive use, without needing to know the underlying implementation details of everything involved in a task.

Package structure

To turn modules into a package, we follow these steps:

- Create a directory with the name of the package (`stock_analysis` for this lesson).
- Place the modules in the aforementioned directory.
- Add an `__init__.py` file containing any Python code to run upon importing the package (this can be—and often is—empty).
- Make a `setup.py` file at the same level as the package's top-level directory (`stock_analysis` here), which will give pip instructions on how to install the package. See the Further reading section for information on creating this.

Package structure

These subpackages are created just as if we were creating a package, with the exception that they don't need a `setup.py` file:

- Create a directory for the subpackage inside the main package directory (or inside some other subpackage).
- Place the subpackage's modules in this directory.
- Add the `__init__.py` file, with code that should be run when the subpackage is imported (this can be empty).

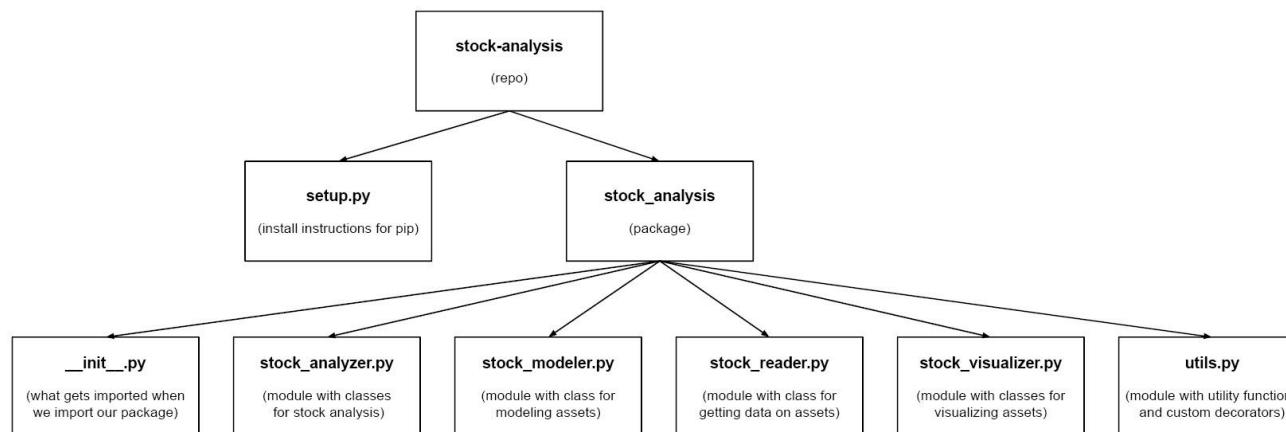
Package structure

- The directory hierarchy for a package with a singlesubpackage would look something like this:

```
repo_folder
|-- <package_name>
|   |-- __init__.py
|   |-- some_module.py
|   `-- <subpackage_name>
|       |-- __init__.py
|       |-- another_module.py
|       `-- last_module.py
`-- setup.py
```

Overview of the stock_analysis package

- This package is located in the stock-analysis repository (<https://github.com/fenango/stock-analysis>), which is arranged like this:



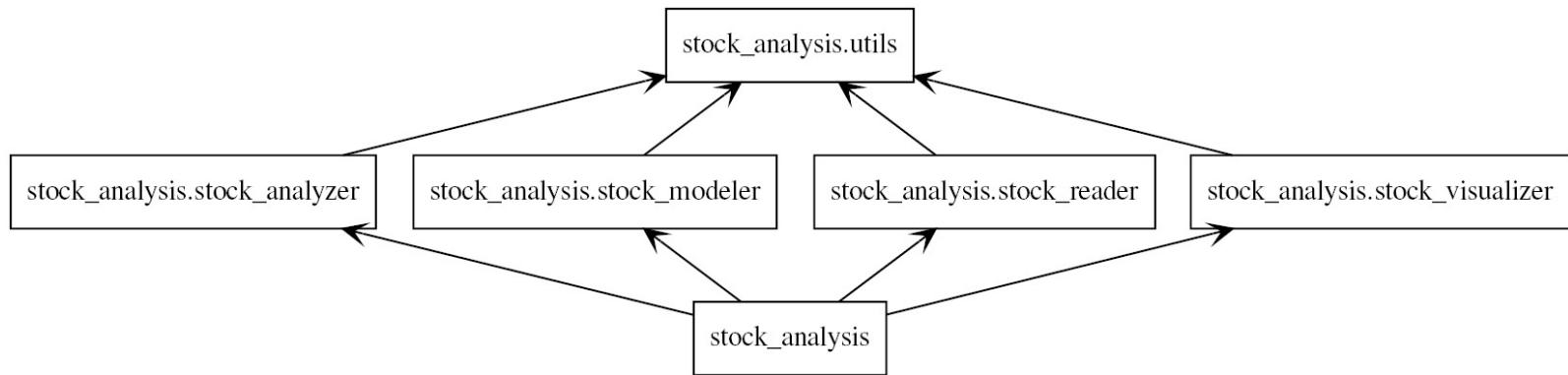
Overview of the stock_analysis package

- We will need a class for each of the following purposes:

Purpose	Class(es)	Module
Collecting the data from various sources	StockReader	stock_reader.py
Visualizing the data	Visualizer , StockVisualizer , AssetGroupVisualizer	stock_visualizer.py
Calculating financial metrics	StockAnalyzer , AssetGroupAnalyzer	stock_analyzer.py
Modeling the data	StockModeler	stock_modeler.py

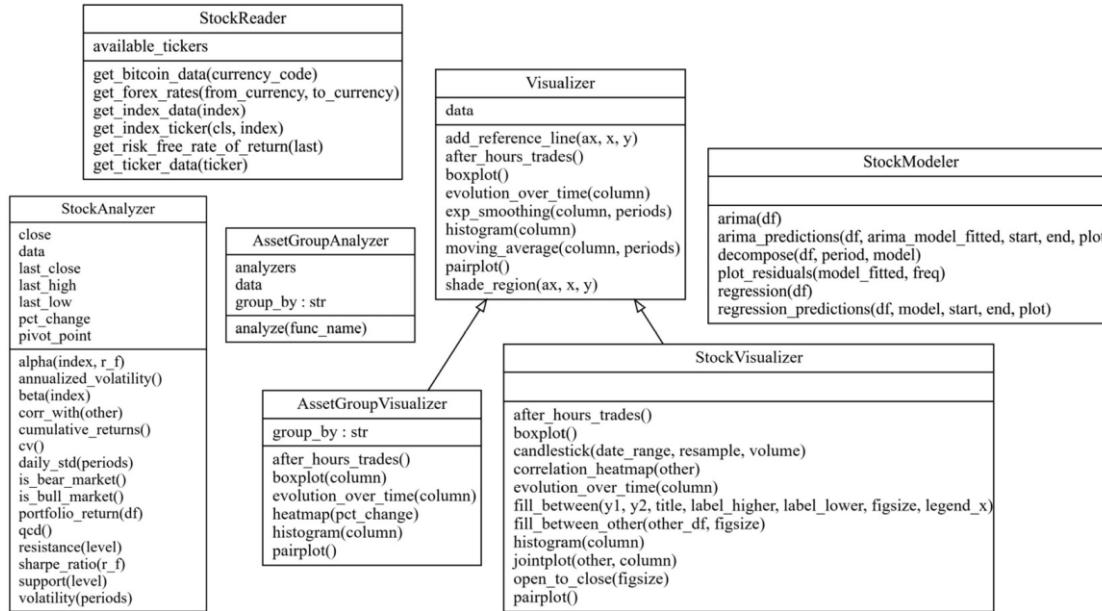
UML diagrams

- UML diagrams show information about which attributes and methods classes have and how classes are related to others.
- We can see in the following diagram that all the modules rely on `utils.py` for utility functions:



UML diagrams

- The UML diagram for the classes in the stock_analysis package looks like this:

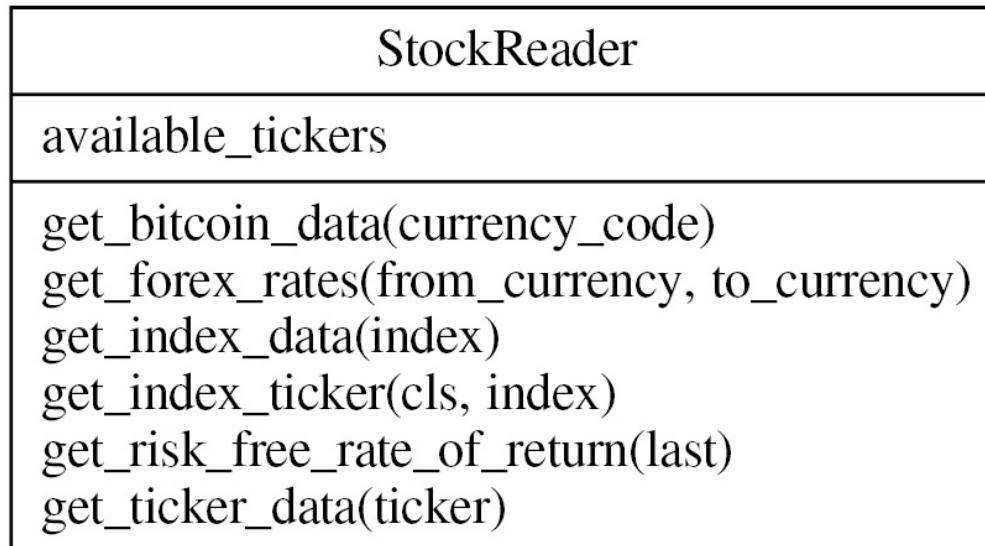


Collecting financial data

- We can use web scraping to extract data from the HTML page itself, which pandas offers with the `pd.read_html()` function—it returns a dataframe for each of the HTML tables it finds on the page.
- For economic and financial data, an alternative is the `pandas_datareader` package, which the `StockReader` class in the `stock_analysis` package uses to collect financial data.

The StockReader class

- The following UML diagram provides a high-level overview of the implementation:



The StockReader class

The UML diagram tells us that the StockReader class provides an attribute for the available tickers (`available_tickers`) and can perform the following actions:

- Pull bitcoin data in the desired currency with the `get_bitcoin_data()` method.
- Pull daily foreign exchange rates data with the `get_forex_rates()` method.
- Pull data for an index on the stock market (such as the S&P 500) with the `get_index_data()` method.

The StockReader class

- This is immediately followed by any imports we will need:

```
"""Gather select stock data."""
import datetime as dt
import re
import pandas as pd
import pandas_datareader.data as web
from .utils import label_sanitizer
```



The StockReader class



- Here, we will only reproduce a few of the tickers available:

```
class StockReader:
```

```
    """Class for reading financial data from websites."""
```

```
    _index_tickers = {'S&P 500': '^GSPC', 'Dow Jones': '^DJI',  
                      'NASDAQ': '^IXIC'}
```

The StockReader class

When building a class, there are many special methods that we can provide to customize the behavior of the class when it's used with language operators:

- Initialize an object (`__init__()`).
- Make an object comparable for sorting (`__eq__()`, `__lt__()`, `__gt__()`, and more).
- Perform arithmetic on the object (`__add__()`, `__sub__()`, `__mul__()`, and so on).

The StockReader class

- The separator, if there is one, gets replaced with an empty string so that we can build our datetimes using the 'YYYYMMDD' format in our method.
- In addition, we raise a ValueError if the caller gives us a start date equal to or after the end date:

```
def __init__(self, start, end=None):
    """
    Create a `StockReader` object for reading across
    a given date range.

    Parameters:
        - start: The first date to include, as a datetime
            object or a string in the format 'YYYYMMDD'.
        - end: The last date to include, as a datetime
            object or string in the format 'YYYYMMDD'.
            Defaults to today if not provided.
    """
    self.start, self.end = map(
        lambda x: x.strftime('%Y%m%d') \
            if isinstance(x, dt.date) \
            else re.sub(r'\D', '', x),
        [start, end or dt.date.today()])
    if self.start >= self.end:
        raise ValueError(`start` must be before `end`)
```

The StockReader class

- To use a decorator, we place it above the function or method definition:

```
@property  
def available_tickers(self):  
    """Indices whose tickers are supported."""  
    return list(self._index_tickers.keys())
```



The StockReader class

- Class methods, by convention, receive `cls` as their first argument while instance methods receive `self`:

```
@classmethod
def get_index_ticker(cls, index):
    """
    Get the ticker of the specified index, if known.

    Parameters:
        - index: The name of the index; check
            `available_tickers` for full list which includes:
            - 'S&P 500' for S&P 500,
            - 'Dow Jones' for Dow Jones Industrial Average,
            - 'NASDAQ' for NASDAQ Composite Index

    Returns:
        The ticker as a string if known, otherwise `None`.
    """
    try:
        index = index.upper()
    except AttributeError:
        raise ValueError(`index` must be a string')
    return cls._index_tickers.get(index, None)
```

The StockReader class

- We will use the `pandas_datareader` package to collect this data from the Federal Reserve Bank of St. Louis

```
def get_risk_free_rate_of_return(self, last=True):  
    """  
    Get risk-free rate of return w/ 10-year US T-bill  
    from FRED (https://fred.stlouisfed.org/series/DGS10)  
    Parameter:  
        - last: If `True`, return the rate on the last  
            date in the date range else, return a `Series`  
            object for the rate each day in the date range.  
    Returns:  
        A single value or a `pandas.Series` object.  
    """  
  
    data = web.DataReader(  
        'DGS10', 'fred', start=self.start, end=self.end  
    )  
    data.index.rename('date', inplace=True)  
    data = data.squeeze()  
    return data.asof(self.end) \  
        if last and isinstance(data, pd.Series) else data
```

The StockReader class

- We will write the following methods in the next section:

```
@label_sanitizer
def get_ticker_data(self, ticker):
    pass
def get_index_data(self, index):
    pass
def get_bitcoin_data(self, currency_code):
    pass
@label_sanitizer
def get_forex_rates(self, from_currency, to_currency,
                    **kwargs):
    pass
```

The StockReader class

- As we did previously, let's begin by looking at the docstring and imports of the utils module:

```
"""Utility functions for stock analysis."""
from functools import wraps
import re
import pandas as pd
```



The StockReader class

- Next, we have the `_sanitize_label()` function, which will clean up a single label.

```
def _sanitize_label(label):
    """
        Clean up a label by removing non-letter, non-space
        characters and putting in all lowercase with underscores
        replacing spaces.
    Parameters:
        - label: The text you want to fix.
    Returns:
        The sanitized label.
    """
    return re.sub(r'[^w\s]', '', label)\n        .lower().replace(' ', '_')
```

The StockReader class

- By using the decorator, the methods will always return a dataframe with the names cleaned, saving us a step:

```
def label_sanitizer(method):
    """
    Decorator around a method that returns a dataframe to
    clean up all labels in said dataframe (column names and
    index name) by using `'_sanitize_label()`'.
    Parameters:
        - method: The method to wrap.
    Returns:
        A decorated method or function.
    """
    @wraps(method) # keep original docstring for help()
    def method_wrapper(self, *args, **kwargs):
        df = method(self, *args, **kwargs)
        # fix the column names
        df.columns = [
            _sanitize_label(col) for col in df.columns
        ]
        # fix the index name
        df.index.rename(
            _sanitize_label(df.index.name), inplace=True
        )
        return df
    return method_wrapper
```

Collecting historical data from Yahoo! Finance

- The foundation of our data collection will be the `get_ticker_data()` method.
- It uses the `pandas_datareader` package to grab the data from Yahoo! Finance:

```
@label_sanitizer
def get_ticker_data(self, ticker):
    """
    Get historical OHLC data for given date range and ticker.
    Parameter:
        - ticker: The stock symbol to lookup as a string.
    Returns: A `pandas.DataFrame` object with the stock data.
    """
    return web.get_data_yahoo(ticker, self.start, self.end)
```

Collecting historical data from Yahoo! Finance

- To collect data for a stock market index, we can use the `get_index_data()` method, which first looks up the index's ticker and then calls the `get_ticker_data()` method we just defined.

```
def get_index_data(self, index):  
    """  
    Get historical OHLC data from Yahoo! Finance  
    for the chosen index for given date range.  
    Parameter:  
        - index: String representing the index you want  
            data for, supported indices include:  
            - 'S&P 500' for S&P 500,  
            - 'Dow Jones' for Dow Jones Industrial Average,  
            - 'NASDAQ' for NASDAQ Composite Index  
    Returns:  
        A `pandas.DataFrame` object with the index data.  
    """  
    if index not in self.available_tickers:  
        raise ValueError(  
            'Index not supported. Available tickers'  
            f"are: {', '.join(self.available_tickers)}"  
        )  
    return self.get_ticker_data(self.get_index_ticker(index))
```

Collecting historical data from Yahoo! Finance

- Yahoo! Finance also provides data for bitcoin; however, we must pick a currency to use.

```
def get_bitcoin_data(self, currency_code):
    """
    Get bitcoin historical OHLC data for given date range.
    Parameter:
        - currency_code: The currency to collect the bitcoin
            data in, e.g. USD or GBP.
    Returns:
        A `pandas.DataFrame` object with the bitcoin data.
    """
    return self\
        .get_ticker_data(f'BTC-{currency_code}')\
        .loc[self.start:self.end] # clip dates
```

Collecting historical data from Yahoo! Finance

- At this point, the StockReader class is ready for use, so let's get started in the financial_analysis.ipynb notebook and import the stock_analysis package that will be used for the rest of this lesson:

```
>>> import stock_analysis
```



Collecting historical data from Yahoo! Finance

- Python runs the stock_analysis/__init__.py file when we import the stock_analysis package:

```
"""Classes for making technical stock analysis easier."""
from .stock_analyzer import StockAnalyzer, AssetGroupAnalyzer
from .stock_modeler import StockModeler
from .stock_reader import StockReader
from .stock_visualizer import \
    StockVisualizer, AssetGroupVisualizer
```

Collecting historical data from Yahoo! Finance

- Note that when we run this code, Python is calling the StockReader.__init__() method:

```
>>> reader = \  
...     stock_analysis.StockReader('2019-01-01', '2020-12-31')
```



Collecting historical data from Yahoo! Finance

- Note that we are using a generator expression and multiple assignment to get dataframes for each FAANG stock:

```
>>> fb, aapl, amzn, nflx, goog = (
...     reader.get_ticker_data(ticker)
...     for ticker in ['FB', 'AAPL', 'AMZN', 'NFLX', 'GOOG']
... )
>>> sp = reader.get_index_data('S&P 500')
>>> bitcoin = reader.get_bitcoin_data('USD')
```

Exploratory data analysis

- To make all of this possible, we have the Visualizer classes in stock_analysis/stock_visualizer.py.

There are three classes in this file:

- Visualizer
- StockVisualizer
- AssetGroupVisualizer



Exploratory data analysis

- Before we discuss the code for these classes, let's go over some additional functions in the stock_analysis/utils.py file, which will help create these asset groups and describe them for EDA purposes.
- For these functions, we need to import pandas:

```
import pandas as pd
```

Exploratory data analysis

- The `group_stocks()` function takes in a dictionary that maps the name of the asset to the dataframe for that asset and outputs a new dataframe with all the data from the input dataframes and a new column, denoting which asset the data belongs to:

```
def group_stocks(mapping):
    """
    Create a new dataframe with many assets and a new column
    indicating the asset that row's data belongs to.

    Parameters:
        - mapping: A key-value mapping of the form
                    {asset_name: asset_df}

    Returns:
        A new `pandas.DataFrame` object
    """
    group_df = pd.DataFrame()
    for stock, stock_data in mapping.items():
        df = stock_data.copy(deep=True)
        df['name'] = stock
        group_df = group_df.append(df, sort=True)
    group_df.index = pd.to_datetime(group_df.index)
    return group_df
```

Exploratory data analysis

- Let's take a look at how this is defined in the stock_analysis/utils.py file:

```
def validate_df(columns, instance_method=True):
    """
    Decorator that raises a `ValueError` if input isn't a
    `DataFrame` or doesn't contain the proper columns. Note
    the `DataFrame` must be the first positional argument
    passed to this method.

    Parameters:
        - columns: A set of required column names.
            For example, {'open', 'high', 'low', 'close'}.
        - instance_method: Whether or not the item being
            decorated is an instance method. Pass `False` to
            decorate static methods and functions.

    Returns:
        A decorated method or function.
    """

```

Exploratory data analysis

```
def method_wrapper(method):
    @wraps(method)
    def validate_wrapper(self, *args, **kwargs):
        # functions and static methods don't pass self so
        # self is the 1st positional argument in that case
        df = (self, *args)[0 if not instance_method else 1]
        if not isinstance(df, pd.DataFrame):
            raise ValueError(
                'Must pass in a pandas `DataFrame`'
            )
        if columns.difference(df.columns):
            raise ValueError(
                'Dataframe must contain the following'
                f' columns: {columns}'
            )
        return method(self, *args, **kwargs)
    return validate_wrapper
return method_wrapper
```

Exploratory data analysis

- Groups made with the `group_stocks()` function can be described in a single output using the `describe_group()` function.

```
@validate_df(columns={'name'}, instance_method=False)
def describe_group(data):
    """
    Run `describe()` on the asset group.

    Parameters:
        - data: Grouped data resulting from `group_stocks()`

    Returns:
        The transpose of the grouped description statistics.
    """
    return data.groupby('name').describe().T
```

Exploratory data analysis

- Let's use the group_stocks() function to make some asset groups for our analysis:

```
>>> from stock_analysis.utils import \
...     group_stocks, describe_group
>>> faang = group_stocks({
...     'Facebook': fb, 'Apple': aapl, 'Amazon': amzn,
...     'Netflix': nflx, 'Google': goog
... })
>>> faang_sp = group_stocks({
...     'Facebook': fb, 'Apple': aapl, 'Amazon': amzn,
...     'Netflix': nflx, 'Google': goog, 'S&P 500': sp
... })
>>> all_assets = group_stocks({
...     'Bitcoin': bitcoin, 'S&P 500': sp, 'Facebook': fb,
...     'Apple': aapl, 'Amazon': amzn, 'Netflix': nflx,
...     'Google': goog
... })
```

Exploratory data analysis

- Using these groups, the output of describe() can be much more informative for comparison purposes compared to running it on each dataframe separately.
- The describe_group() function handles running describe() with groupby().
- This makes it easier to look at the summary for the closing price across assets:

```
>>> describe_group(all_assets).loc['close',]
```

Exploratory data analysis

- At a glance, we can see that we have more data for bitcoin than the rest.

	Amazon	Apple	Bitcoin	Facebook	Google	Netflix	S&P 500
count	505.000000	505.000000	727.000000	505.000000	505.000000	505.000000	505.000000
mean	2235.904988	73.748386	9252.825408	208.146574	1335.188544	387.966593	3065.907599
std	594.306346	27.280933	4034.014685	39.665111	200.793911	78.931238	292.376435
min	1500.280029	35.547501	3399.471680	131.740005	1016.059998	254.589996	2237.399902
25%	1785.660034	50.782501	7218.593750	180.029999	1169.949951	329.089996	2870.719971
50%	1904.280029	66.730003	9137.993164	196.770004	1295.280029	364.369995	3005.469971
75%	2890.300049	91.632500	10570.513184	235.940002	1476.229980	469.959991	3276.020020
max	3531.449951	136.690002	29001.720703	303.910004	1827.989990	556.549988	3756.070068

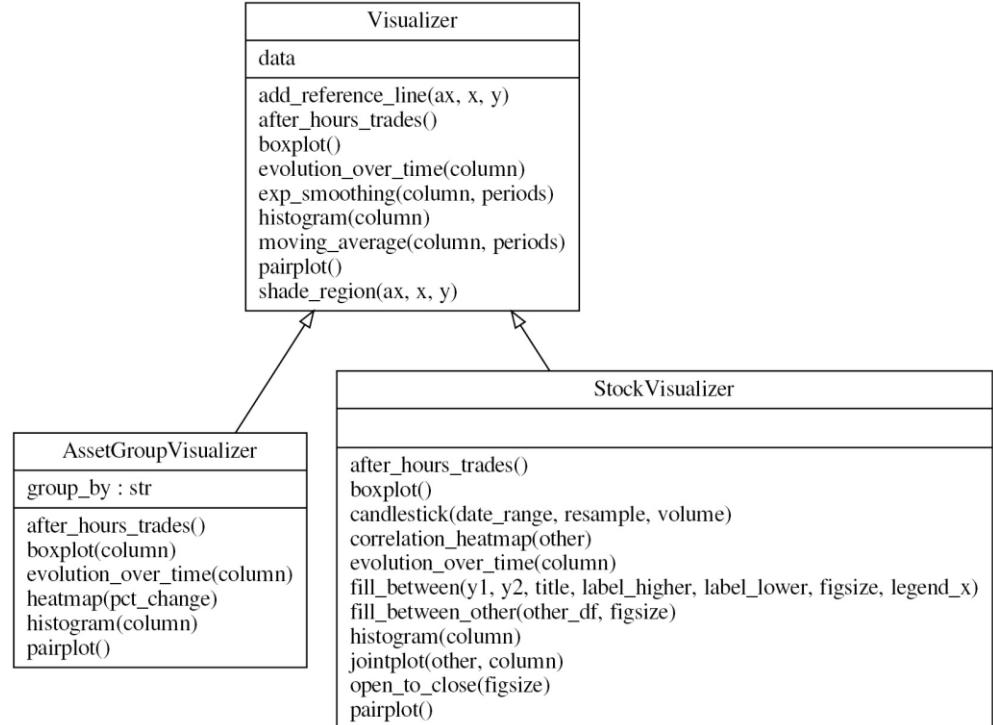
Exploratory data analysis

- If we don't want to look at the assets individually, we can combine them into a portfolio, which we can treat as a single asset.

```
@validate_df(columns=set(), instance_method=False)
def make_portfolio(data, date_level='date'):
    """
    Make a portfolio of assets by grouping by date and
    summing all columns.
    Note: the caller is responsible for making sure the
    dates line up across assets and handling when they don't.
    """
    return data.groupby(level=date_level).sum()
```

The Visualizer class family

- The following UML diagram tells us this is our base class because it has arrows pointing to it.
- These arrows originate from the subclasses (`AssetGroupVisualizer` and `StockVisualizer`):



The Visualizer class family

- We start the module with our docstring and imports.

```
"""Visualize financial instruments."""
```

```
import math
import matplotlib.pyplot as plt
import mplfinance as mpf
import numpy as np
import pandas as pd
import seaborn as sns
from .utils import validate_df
```

The Visualizer class family

- Next, we begin by defining the Visualizer class.
- This class will hold the data it will be used to visualize, so we put this in the `__init__()` method:

```
class Visualizer:
```

```
    """Base visualizer class not intended for direct use."""
```

```
    @validate_df(columns={'open', 'high', 'low', 'close'})
```

```
    def __init__(self, df):
```

```
        """Store the input data as an attribute."""
```

```
        self.data = df
```

The Visualizer class family

```
@staticmethod
def add_reference_line(ax, x=None, y=None, **kwargs):
    """
    Static method for adding reference lines to plots.

    Parameters:
        - ax: `Axes` object to add the reference line to.
        - x, y: The x, y value to draw the line at as a
            single value or numpy array-like structure.
            - For horizontal: pass only `y`
            - For vertical: pass only `x`
            - For AB line: pass both `x` and `y`
        - kwargs: Additional keyword args. to pass down.

    Returns:
        The matplotlib `Axes` object passed in.
    """
    try:
        # numpy array-like structures -> AB line
        if x.shape and y.shape:
            ax.plot(x, y, **kwargs)
```

The Visualizer class family

```
except:  
    # error triggers if x or y isn't array-like  
    try:  
        if not x and not y:  
            raise ValueError(  
                'You must provide an `x` or a `y``'  
            )  
        elif x and not y:  
            ax.axvline(x, **kwargs) # vertical line  
        elif not x and y:  
            ax.axhline(y, **kwargs) # horizontal line  
    except:  
        raise ValueError(  
            'If providing only `x` or `y`, '  
            'it must be a single value'  
        )  
    ax.legend()  
    return ax
```

The Visualizer class family

- The `shade_region()` static method for adding shaded regions to a plot is similar to the `add_reference_line()` static method:

```
@staticmethod
def shade_region(ax, x=tuple(), y=tuple(), **kwargs):
    """
    Static method for shading a region on a plot.

    Parameters:
        - ax: `Axes` object to add the shaded region to.
        - x: Tuple with the `xmin` and `xmax` bounds for
            the rectangle drawn vertically.
        - y: Tuple with the `ymin` and `ymax` bounds for
            the rectangle drawn horizontally.
        - kwargs: Additional keyword args. to pass down.

    Returns:
        The matplotlib `Axes` object passed in.
    """
    if not x and not y:
        raise ValueError(
            'You must provide an x or a y min/max tuple'
        )
    elif x and y:
        raise ValueError('You can only provide x or y.')
    elif x and not y:
        ax.axvspan(*x, **kwargs) # vertical region
    elif not x and y:
        ax.axhspan(*y, **kwargs) # horizontal region
    return ax
```

The Visualizer class family

- This will be utilized in the classes we build using the Visualizer class as our base:

```
@staticmethod
def _iter_handler(items):
    """
        Static method for making a list out of an item if
        it isn't a list or tuple already.

        Parameters:
            - items: The variable to make sure it is a list.

        Returns: The input as a list or tuple.
    """

    if not isinstance(items, (list, tuple)):
        items = [items]
    return items
```

The Visualizer class family

- We want to support window functions for single assets and groups of them

```
def _window_calc(self, column, periods, name, func,
                  named_arg, **kwargs):
    """
    To be implemented by subclasses. Defines how to add
    lines resulting from window calculations.
    """
    raise NotImplementedError('To be implemented by '
                             'subclasses.')
```

The Visualizer class family

- The moving_average() method uses _window_calc() to add moving average lines to the plot:

```
def moving_average(self, column, periods, **kwargs):
    """
    Add line(s) for the moving average of a column.

    Parameters:
        - column: The name of the column to plot.
        - periods: The rule or list of rules for
            resampling, like '20D' for 20-day periods.
        - kwargs: Additional arguments to pass down.

    Returns: A matplotlib `Axes` object.
    """
    return self._window_calc(
        column, periods, name='MA', named_arg='rule',
        func=pd.DataFrame.resample, **kwargs
    )
```

The Visualizer class family

- In a similar fashion, we define the `exp_smoothing()` method, which will use `_window_calc()` to add exponentially smoothed moving average lines to the plot:

```
def exp_smoothing(self, column, periods, **kwargs):  
    """  
        Add line(s) for the exponentially smoothed moving  
        average of a column.  
    Parameters:  
        - column: The name of the column to plot.  
        - periods: The span or list of spans for,  
            smoothing like 20 for 20-day periods.  
        - kwargs: Additional arguments to pass down.  
    Returns:  
        A matplotlib `Axes` object.  
    """  
    return self._window_calc(  
        column, periods, name='EWMA',  
        func=pd.DataFrame.ewm, named_arg='span', **kwargs  
    )
```

The Visualizer class family

- For brevity, the following is a subset of the abstract methods defined in this class:

```
def evolution_over_time(self, column, **kwargs):
    """Creates line plots."""
    raise NotImplementedError('To be implemented by '
                             'subclasses.')
def after_hours_trades(self):
    """Show the effect of after-hours trading."""
    raise NotImplementedError('To be implemented by '
                             'subclasses.')
def pairplot(self, **kwargs):
    """Create pairplots."""
    raise NotImplementedError('To be implemented by '
                             'subclasses.') 
```

Visualizing a stock

- Let's start the StockVisualizer class by inheriting from Visualizer; we will choose not to override the `__init__()` method because the StockVisualizer class will only have a dataframe as an attribute.
- Instead, we will provide implementations for the methods that need to be added (which are unique to this class) or overridden.

Visualizing a stock

- The first method we will override is `evolution_over_time()`, which will create a line plot of a column over time:

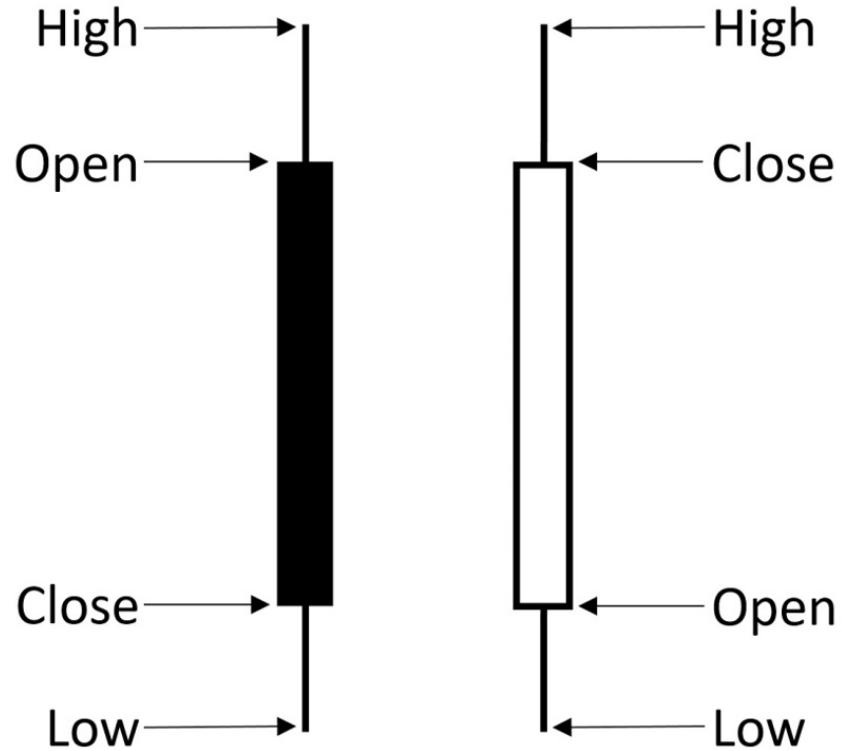
```
class StockVisualizer(Visualizer):
    """Visualizer for a single stock."""
    def evolution_over_time(self, column, **kwargs):
        """
        Visualize the evolution over time of a column.

        Parameters:
        - column: The name of the column to visualize.
        - kwargs: Additional arguments to pass down.

        Returns:
        A matplotlib `Axes` object.
        """
        return self.data.plot.line(y=column, **kwargs)
```

Visualizing a stock

- when the candlestick is white, the asset's closing price was higher than its opening price, as illustrated in the following diagram:



Visualizing a stock

- The candlestick() method also provides options to resample the data, show volume traded, and plot a specific date range:

```
def candlestick(self, date_range=None, resample=None,
                volume=False, **kwargs):
    """
    Create a candlestick plot for the OHLC data.

    Parameters:
        - date_range: String or `slice()` of dates to
                      pass to `loc[]`, if `None` the plot will be
                      for the full range of the data.
        - resample: The offset to use for resampling
                    the data, if desired.
        - volume: Whether to show a bar plot for volume
                  traded under the candlesticks
        - kwargs: Keyword args for `mplfinance.plot()`
    """
```

Visualizing a stock

```
if not date_range:  
    date_range = slice(  
        self.data.index.min(), self.data.index.max()  
    )  
plot_data = self.data.loc[date_range]  
if resample:  
    agg_dict = {  
        'open': 'first', 'close': 'last',  
        'high': 'max', 'low': 'min', 'volume': 'sum'  
    }  
    plot_data = plot_data.resample(resample).agg({  
        col: agg_dict[col] for col in plot_data.columns  
        if col in agg_dict  
    })  
    mpf.plot(  
        plot_data, type='candle', volume=volume, **kwargs  
    )
```

Visualizing a stock

- Now, we add the `after_hours_trades()` method, which helps us visualize the effect after-hours trading had on an individual asset, with bars colored red for losses and green for gains:

```
def after_hours_trades(self):  
    """  
    Visualize the effect of after-hours trading.  
    Returns: A matplotlib `Axes` object.  
    """  
  
    after_hours = self.data.open - self.data.close.shift()  
    monthly_effect = after_hours.resample('1M').sum()  
    fig, axes = plt.subplots(1, 2, figsize=(15, 3))  
    after_hours.plot(  
        ax=axes[0],  
        title='After-hours trading\n'(Open Price - Prior Day\'s Close)  
    ).set_ylabel('price')  
    monthly_effect.index = \  
        monthly_effect.index.strftime('%Y-%b')  
    monthly_effect.plot(  
        ax=axes[1], kind='bar', rot=90,  
        title='After-hours trading monthly effect',  
        color=np.where(monthly_effect >= 0, 'g', 'r')  
    ).axhline(0, color='black', linewidth=1)  
    axes[1].set_ylabel('price')  
    return axes
```

Visualizing a stock

- Next, we will add a static method that will allow us to fill the area between two curves of our choosing.

```
@staticmethod
def fill_between(y1, y2, title, label_higher, label_lower,
                  figsize, legend_x):
    """
    Visualize the difference between assets.

    Parameters:
        - y1, y2: Data to plot, filling y2 - y1.
        - title: The title for the plot.
        - label_higher: Label for when y2 > y1.
        - label_lower: Label for when y2 <= y1.
        - figsize: (width, height) for the plot dimensions.
        - legend_x: Where to place legend below the plot.

    Returns: A matplotlib `Axes` object.
    """
    is_higher = y2 - y1 > 0
    fig = plt.figure(figsize=figsize)
    for exclude_mask, color, label in zip(
        (is_higher, np.invert(is_higher)),
        ('g', 'r'),
        (label_higher, label_lower)
    ):
        plt.fill_between(
            y2.index, y2, y1, figure=fig,
            where=exclude_mask, color=color, label=label
        )
    plt.suptitle(title)
    plt.legend(
        bbox_to_anchor=(legend_x, -0.1),
        framealpha=0, ncol=2
    )
    for spine in ['top', 'right']:
        fig.axes[0].spines[spine].set_visible(False)
    return fig.axes[0]
```

Visualizing a stock

- We will color the area green if the closing price is higher than the opening price and red if the opposite is true:

```
def open_to_close(self, figsize=(10, 4)):  
    """  
        Visualize the daily change in price from open to close.  
        Parameters:  
            - figsize: (width, height) of plot  
        Returns:  
            A matplotlib `Axes` object.  
    """  
  
    ax = self.fill_between(  
        self.data.open, self.data.close,  
        figsize=figsize, legend_x=0.67,  
        title='Daily price change (open to close)',  
        label_higher='price rose', label_lower='price fell'  
    )  
    ax.set_ylabel('price')  
    return ax
```

Visualizing a stock

- We will color the differential green when the visualizer's asset is higher than the other asset and red for when it is lower:

```
def fill_between_other(self, other_df, figsize=(10, 4)):  
    """  
        Visualize difference in closing price between assets.  
        Parameters:  
            - other_df: The other asset's data.  
            - figsize: (width, height) for the plot.  
        Returns:  
            A matplotlib `Axes` object.  
    """  
    ax = self.fill_between(  
        other_df.open, self.data.close, figsize=figsize,  
        legend_x=0.7, label_higher='asset is higher',  
        label_lower='asset is lower',  
        title='Differential between asset price '  
            '(this - other)'  
    )  
    ax.set_ylabel('price')  
    return ax
```

Visualizing a stock

- The time has finally come to override the `_window_calc()` method,

```
def _window_calc(self, column, periods, name, func,
                 named_arg, **kwargs):
    """
    Helper method for plotting a series and adding
    reference lines using a window calculation.
    Parameters:
        - column: The name of the column to plot.
        - periods: The rule/span or list of them to pass
            to the resampling/smoothing function, like '20D'
            for 20-day periods (resampling) or 20 for a
            20-day span (smoothing)
        - name: The name of the window calculation (to
            show in the legend).
        - func: The window calculation function.
        - named_arg: The name of the argument `periods`
            is being passed as.
        - kwargs: Additional arguments to pass down.
    Returns:
        A matplotlib `Axes` object.
```

Visualizing a stock

```
"""
    ax = self.data.plot(y=column, **kwargs)
    for period in self._iter_handler(periods):
        self.data[column].pipe(
            func, **{named_arg: period}
        ).mean().plot(
            ax=ax, linestyle='--',
            label=f"""{period if isinstance(
                period, str
            ) else str(period) + 'D'} {name}"""
        )
    plt.legend()
    return ax
```

Visualizing a stock

- so we will build a wrapper around the jointplot() function from seaborn:

```
def jointplot(self, other, column, **kwargs):
    """
    Generate a seaborn jointplot for given column in
    this asset compared to another asset.
    Parameters:
        - other: The other asset's dataframe.
        - column: Column to use for the comparison.
        - kwargs: Keyword arguments to pass down.
    Returns: A seaborn jointplot
    """
    return sns.jointplot(
        x=self.data[column], y=other[column], **kwargs
    )
```

Visualizing a stock

- In addition, we will use the daily percentage change of each column when calculating the correlations to handle the difference in scale (for instance, between Apple's stock price and Amazon's stock price):

```
def correlation_heatmap(self, other):  
    """  
    Plot the correlations between this asset and another  
    one with a heatmap.  
    Parameters:  
        - other: The other dataframe.  
    Returns: A seaborn heatmap  
    """  
  
    corrs = \  
        self.data.pct_change().corrwith(other.pct_change())  
    corrs = corrs[~pd.isnull(corrs)]  
    size = len(corrs)  
    matrix = np.zeros((size, size), float)  
    for i, corr in zip(range(size), corrs):  
        matrix[i][i] = corr  
    # create mask to only show diagonal  
    mask = np.ones_like(matrix)  
    np.fill_diagonal(mask, 0)  
    return sns.heatmap(  
        matrix, annot=True, center=0, vmin=-1, vmax=1,  
        mask=mask, xticklabels=self.data.columns,  
        yticklabels=self.data.columns  
    )
```

Visualizing a stock

- Now that we understand some of the functionality available in the StockVisualizer class, we can begin our exploratory analysis.
- Let's create a StockVisualizer object to perform some EDA on the Netflix stock data:

```
>>> %matplotlib inline  
>>> import matplotlib.pyplot as plt  
>>> netflix_viz = stock_analysis.StockVisualizer(nflx)
```

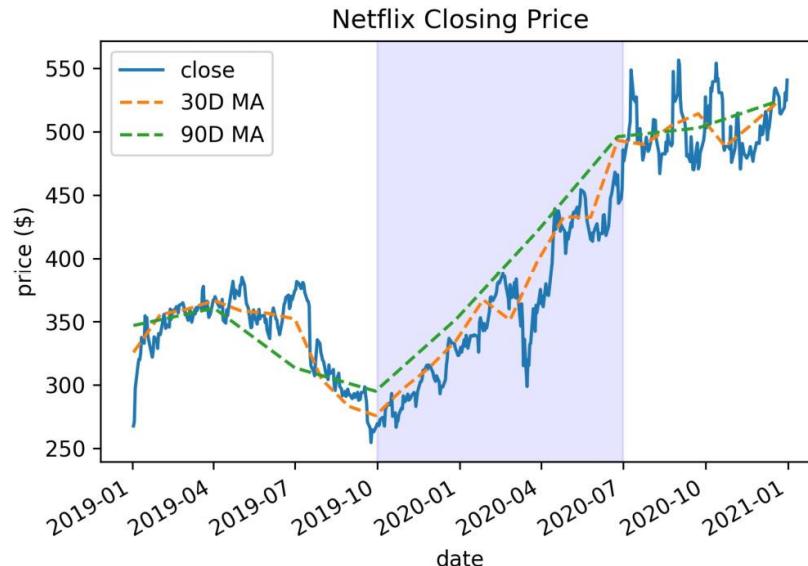
Visualizing a stock

- let's take a look at the closing price over time with some moving averages to study the trend:

```
>>> ax = netflix_viz.moving_average('close', ['30D', '90D'])
>>> netflix_viz.shade_region(
...     ax, x=('2019-10-01', '2020-07-01'),
...     color='blue', alpha=0.1
... )
>>> ax.set(title='Netflix Closing Price', ylabel='price ($)')
```

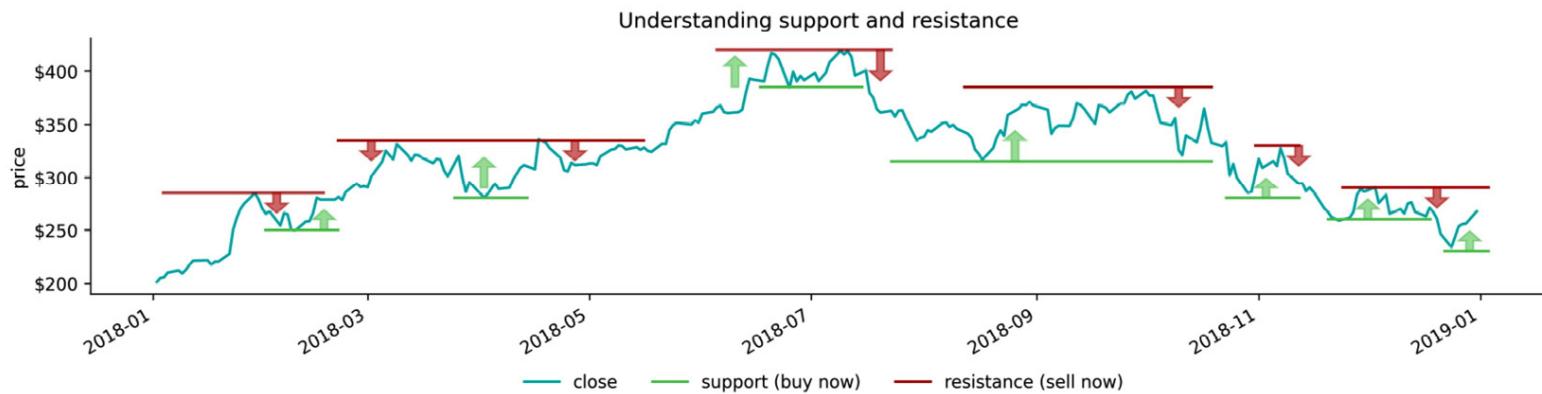
Visualizing a stock

- These moving averages give us a smoothed version of the stock price curve.



Visualizing a stock

- Figure shows an example of how support (green) and resistance (red) act as lower and upper bounds, respectively, for the stock price; once the price hits either of these bounds, it tends to bounce back in the opposite direction due to buyers/sellers of the stock taking action:



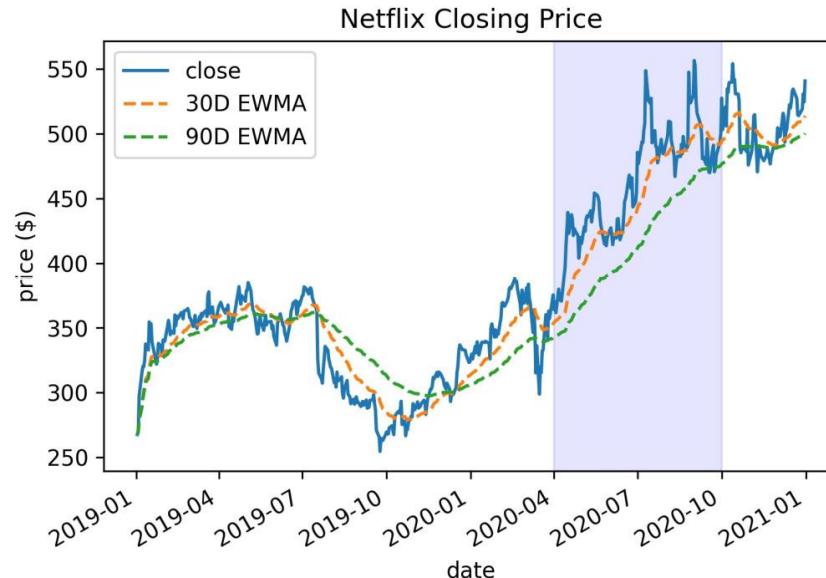
Visualizing a stock

- Let's see how exponential smoothing looks for our data:

```
>>> ax = netflix_viz.exp_smoothing('close', [30, 90])
>>> netflix_viz.shade_region(
...     ax, x=('2020-04-01', '2020-10-01'),
...     color='blue', alpha=0.1
... )
>>> ax.set(title='Netflix Closing Price', ylabel='price ($)')
```

Visualizing a stock

- The 90-day EWMA appears to be acting as the support level in the shaded region:



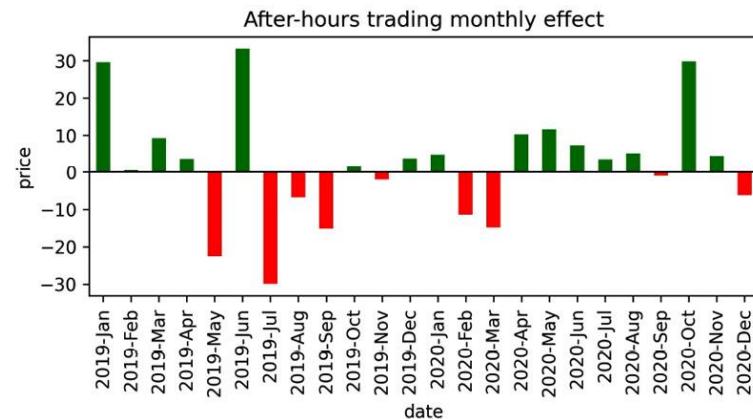
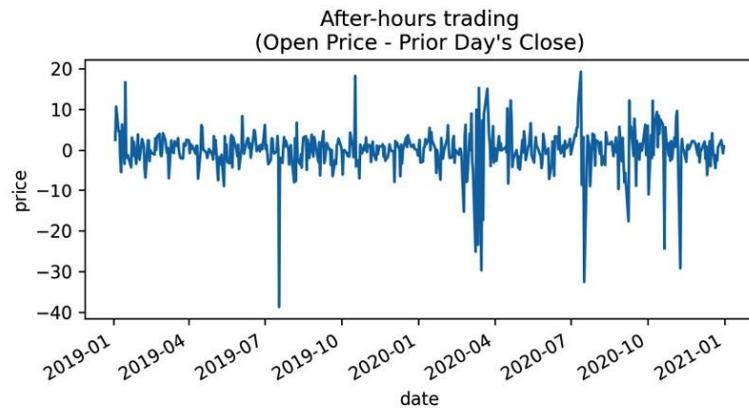
Visualizing a stock

- In the exercises for Visualizing Data with Pandas and Matplotlib, we wrote code for generating a visualization that represented the effect that after-hours trading had on Facebook; the StockVisualizer class also has this functionality.
- Let's use the `after_hours_trades()` method to see how Netflix fared:

```
>>> netflix_viz.after_hours_trades()
```

Visualizing a stock

- Netflix had a rough third quarter in 2019 in terms of after-hours trades:



Visualizing a stock

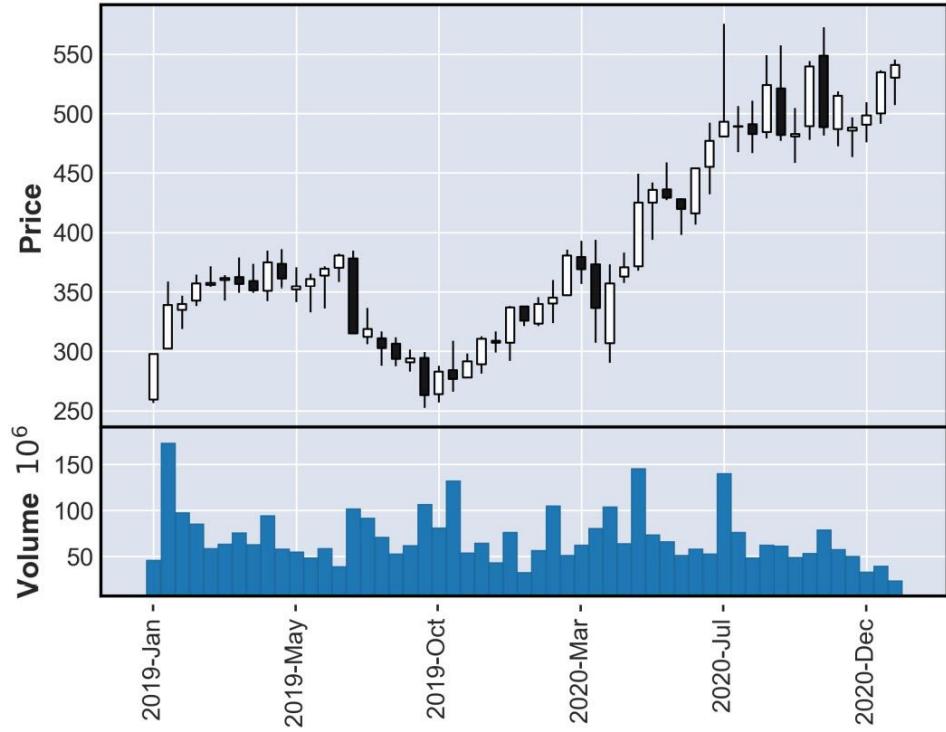
- We can use candlestick plots to study the OHLC data:

```
>>> netflix_viz.candlestick(  
...     resample='2W', volume=True, xrotation=90,  
...     datetime_format='%Y-%b -'  
... )
```



Visualizing a stock

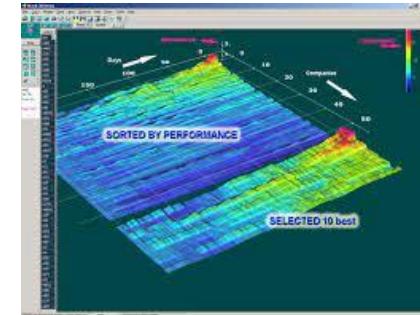
- Remember that when the body of the candlestick is white, it means that the stock gained value.



Visualizing a stock

- Before moving on, we need to reset our plot styles.
- The mplfinance package sets many of the available styling options for its plots, so let's return to the style we are familiar with for now:

```
>>> import matplotlib as mpl  
>>> mpl.rcParams()  
>>> %matplotlib inline
```



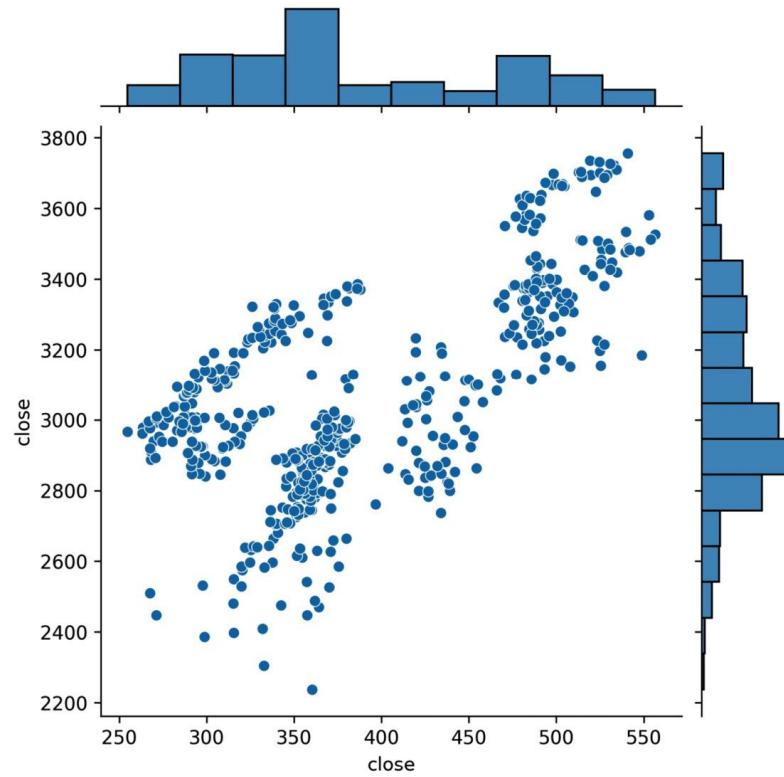
Visualizing a stock

- We have already taken a look at a stock in isolation (Facebook) in prior lessons, so let's take this in a different direction and compare Netflix to others.
- Let's use the jointplot() method to see how Netflix compares to the S&P 500:

```
>>> netflix_viz.jointplot(sp, 'close')
```

Visualizing a stock

- We will calculate beta in the Technical analysis of financial instruments section later in this lesson:



Visualizing a stock

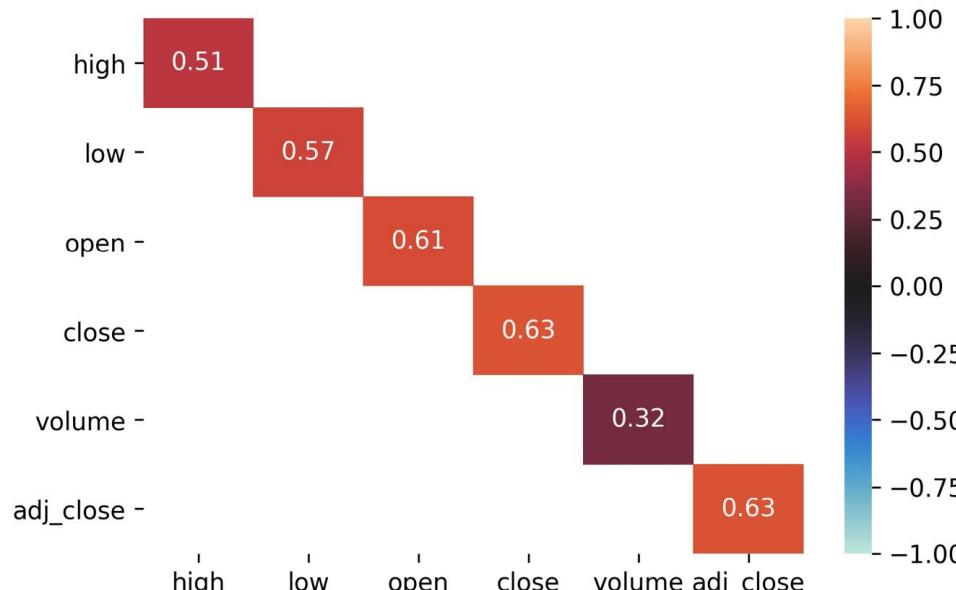
- We can use the correlation_heatmap() method to visualize the correlations between Netflix and Amazon as a heatmap, using the daily percentage change of each of the columns:

```
>>> netflix_viz.correlation_heatmap(amzn)
```



Visualizing a stock

- Netflix and Amazon are weakly positively correlated, but only on the OHLC data:



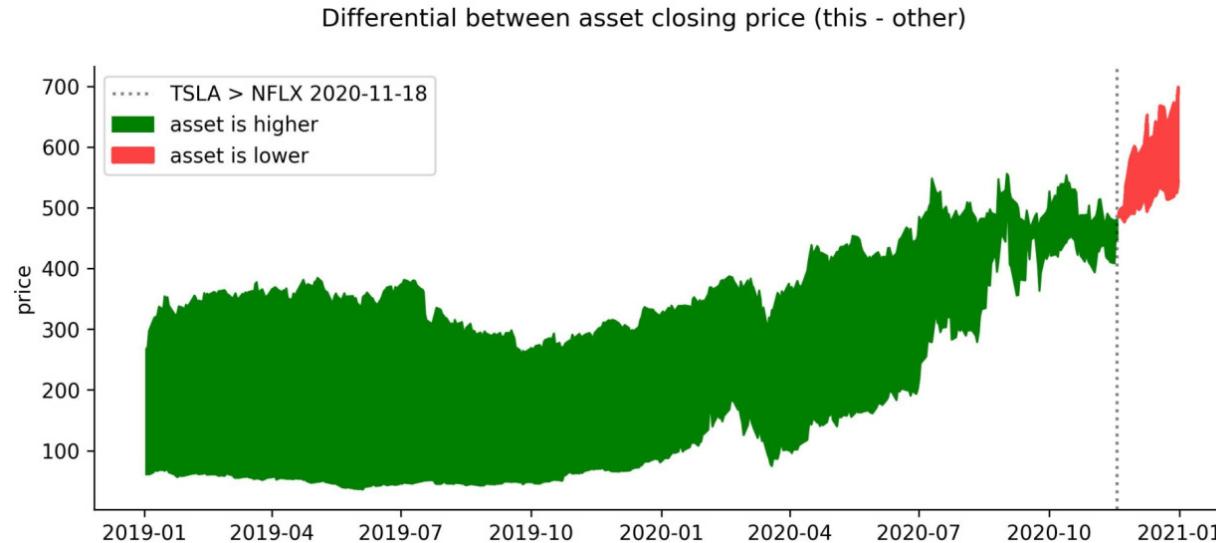
Visualizing a stock

- Lastly, we can use the `fill_between_other()` method to see how another asset grew (or fell) in price compared to Netflix.

```
>>> tsla = reader.get_ticker_data('TSLA')
>>> change_date = (tsla.close > nflx.close).idxmax()
>>> ax = netflix_viz.fill_between_other(tsla)
>>> netflix_viz.add_reference_line(
...     ax, x=change_date, color='k', linestyle=':', alpha=0.5,
...     label=f'TSLA > NFLX {change_date:%Y-%m-%d}'
... )
```

Visualizing a stock

- Notice that the shaded region shrinks in height as it approaches the reference line



Visualizing multiple assets

- we call the `__init__()` method of the superclass as well:

```
class AssetGroupVisualizer(Visualizer):
    """Visualizes groups of assets in a single dataframe."""
    # override for group visuals
    def __init__(self, df, group_by='name'):
        """This object keeps track of the group by column."""
        super().__init__(df)
        self.group_by = group_by
```

Visualizing multiple assets

- Since our data is of a different shape, we will use seaborn this time:

```
def evolution_over_time(self, column, **kwargs):
    """
    Visualize the evolution over time for all assets.
    Parameters:
        - column: The name of the column to visualize.
        - kwargs: Additional arguments to pass down.
    Returns: A matplotlib `Axes` object.
    """
    if 'ax' not in kwargs:
        fig, ax = plt.subplots(1, 1, figsize=(10, 4))
    else:
        ax = kwargs.pop('ax')
    return sns.lineplot(
        x=self.data.index, y=column, hue=self.group_by,
        data=self.data, ax=ax, **kwargs
    )
```

Visualizing multiple assets

- For this, we will add the `_get_layout()` method, which will generate the Figure and Axes objects we need for a given number of subplots

```
def _get_layout(self):
    """
        Helper method for getting an autolayout of subplots.
        Returns: `Figure` and `Axes` objects to plot with.
    """
    subplots_needed = self.data[self.group_by].nunique()
    rows = math.ceil(subplots_needed / 2)
    fig, axes = \
        plt.subplots(rows, 2, figsize=(15, 5 * rows))
    if rows > 1:
        axes = axes.flatten()
    if subplots_needed < len(axes):
        # remove excess axes from autolayout
        for i in range(subplots_needed, len(axes)):
            # can't use comprehension here
            fig.delaxes(axes[i])
    return fig, axes
```

Visualizing multiple assets

```
def _window_calc(self, column, periods, name, func,
                 named_arg, **kwargs):
    """
    Helper method for plotting a series and adding
    reference lines using a window calculation.

    Parameters:
        - column: The name of the column to plot.
        - periods: The rule/span or list of them to pass
            to the resampling/smoothing function, like '20D'
            for 20-day periods (resampling) or 20 for a
            20-day span (smoothing)
        - name: The name of the window calculation (to
            show in the legend).
        - func: The window calculation function.
        - named_arg: The name of the argument `periods`
            is being passed as.
        - kwargs: Additional arguments to pass down.

    Returns:
        A matplotlib `Axes` object.
```

Visualizing multiple assets

```
"""
    fig, axes = self._get_layout()
    for ax, asset_name in zip(
        axes, self.data[self.group_by].unique()
    ):
        subset = self.data.query(
            f'{self.group_by} == "{asset_name}"'
        )
        ax = subset.plot(
            y=column, ax=ax, label=asset_name, **kwargs
        )
        for period in self._iter_handler(periods):
            subset[column].pipe(
                func, **{named_arg: period}
            ).mean().plot(
                ax=ax, linestyle='--',
                label=f"""{period if isinstance(
                    period, str
                ) else str(period) + 'D'} {name}"""
            )
        ax.legend()
    plt.tight_layout()
    return ax
```

Visualizing multiple assets

- We can override `after_hours_trades()` to visualize the effect of after-hours trading on a group of assets using subplots and iterating over the assets in the group:

```
def after_hours_trades(self):  
    """  
    Visualize the effect of after-hours trading.  
    Returns: A matplotlib `Axes` object.  
    """  
  
    num_categories = self.data[self.group_by].nunique()  
    fig, axes = plt.subplots(  
        num_categories, 2, figsize=(15, 3 * num_categories)  
    )  
    for ax, (name, data) in zip(  
        axes, self.data.groupby(self.group_by)  
    ):  
        after_hours = data.open - data.close.shift()  
        monthly_effect = after_hours.resample('1M').sum()  
        after_hours.plot(  
            ax=ax[0],  
            title=f'{name} Open Price - Prior Day\'s Close'  
        ).set_ylabel('price')  
        monthly_effect.index = \  
            monthly_effect.index.strftime('%Y-%b')  
        monthly_effect.plot(  
            ax=ax[1], kind='bar', rot=90,  
            color=np.where(monthly_effect >= 0, 'g', 'r'),  
            title=f'{name} after-hours trading '  
                'monthly effect'  
        ).axhline(0, color='black', linewidth=1)  
        ax[1].set_ylabel('price')  
    plt.tight_layout()  
    return axes
```

Visualizing multiple assets

- With the StockVisualizer class, we were able to generate a joint plot between two assets' closing prices, but here we can override pairplot() to allow us to see the relationships between the closing prices across assets in the group:

```
def pairplot(self, **kwargs):
    """
    Generate a seaborn pairplot for this asset group.
    Parameters:
        - kwargs: Keyword arguments to pass down.
    Returns: A seaborn pairplot
    """
    return sns.pairplot(
        self.data.pivot_table(
            values='close', index=self.data.index,
            columns=self.group_by
        ), diag_kind='kde', **kwargs
    )
```

Visualizing multiple assets

- Finally, we add the `heatmap()` method, which generates a heatmap of the correlations between the closing prices of all the assets in the group:

```
def heatmap(self, pct_change=True, **kwargs):
    """
    Generate a heatmap for correlations between assets.
    Parameters:
        - pct_change: Whether to show the correlations
                      of the daily percent change in price.
        - kwargs: Keyword arguments to pass down.
    Returns: A seaborn heatmap
    """
    pivot = self.data.pivot_table(
        values='close', index=self.data.index,
        columns=self.group_by
    )
    if pct_change:
        pivot = pivot.pct_change()
    return sns.heatmap(
        pivot.corr(), annot=True, center=0,
        vmin=-1, vmax=1, **kwargs
    )
```

Visualizing multiple assets

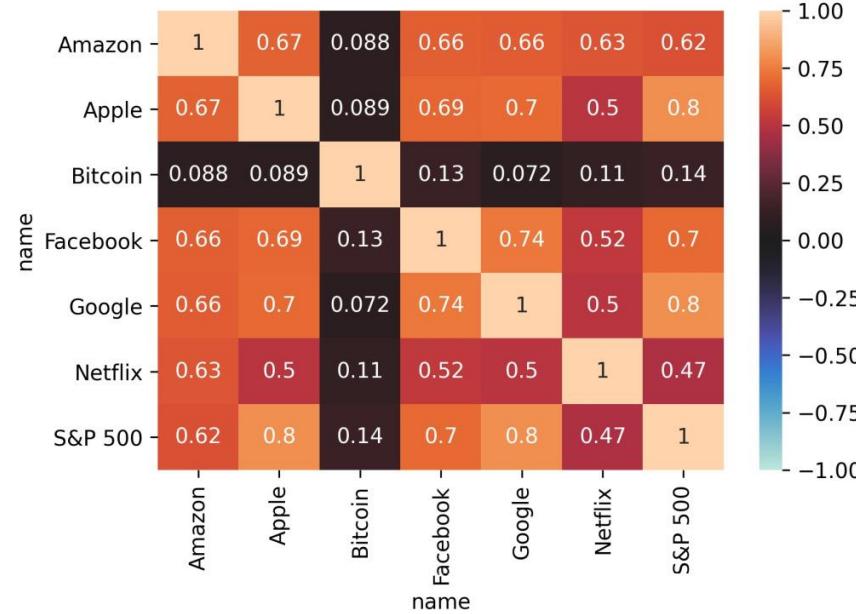
- We can use the heatmap() method to see how the daily percentage change across assets compares.

```
>>> all_assets_viz = \  
...     stock_analysis.AssetGroupVisualizer(all_assets)  
>>> all_assets_viz.heatmap()
```



Visualizing multiple assets

- Apple-S&P 500 and Facebook-Google have the strongest correlations, with bitcoin having no correlation with anything:



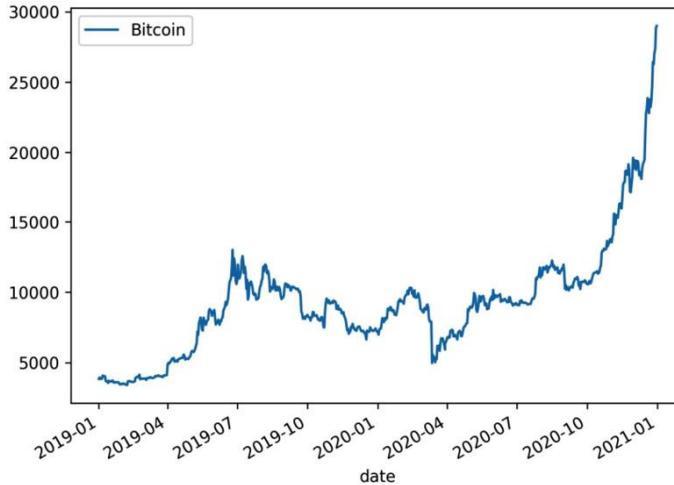
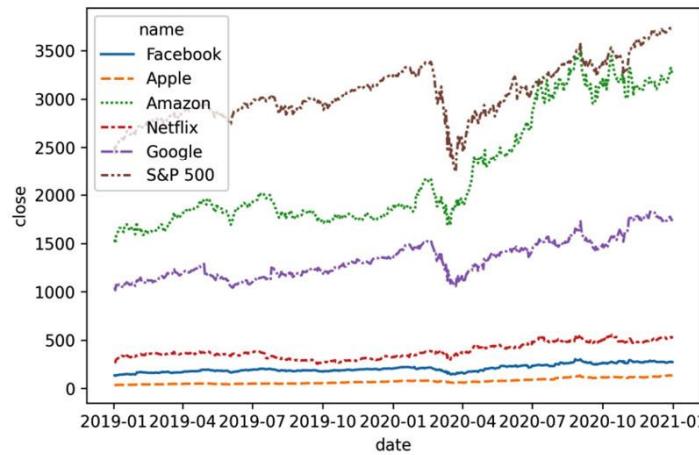
Visualizing multiple assets

- However, let's combine these Visualizers to see how all of our assets evolved over time:

```
>>> faang_sp_viz = \  
...     stock_analysis.AssetGroupVisualizer(faang_sp)  
>>> bitcoin_viz = stock_analysis.StockVisualizer(bitcoin)  
>>> fig, axes = plt.subplots(1, 2, figsize=(15, 5))  
>>> faang_sp_viz.evolution_over_time(  
...     'close', ax=axes[0], style=faang_sp_viz.group_by  
... )  
>>> bitcoin_viz.evolution_over_time(  
...     'close', ax=axes[1], label='Bitcoin'  
... )
```

Visualizing multiple assets

- Note that bitcoin had huge gains to close out 2020 (check out the scale on the y-axis), and Amazon also saw a lot of growth in 2020:



Technical analysis of financial instruments

- As with the other modules, we will start with our docstring and imports:

```
"""Classes for technical analysis of assets."""
```

```
import math  
from .utils import validate_df
```



The StockAnalyzer class

- For analyzing individual assets, we will build the StockAnalyzer class, which calculates metrics for a given asset.
- The following UML diagram shows all the metrics that it provides:

StockAnalyzer
close
data
last_close
last_high
last_low
pct_change
pivot_point
alpha(index, r_f)
annualized_volatility()
beta(index)
corr_with(other)
cumulative_returns()
cv()
daily_std(periods)
is_bear_market()
is_bull_market()
portfolio_return(df)
qcd()
resistance(level)
sharpe_ratio(r_f)
support(level)
volatility(periods)

The StockAnalyzer class

- A StockAnalyzer instance will be initialized with the data for the asset on which we want to perform a technical analysis.
- This means that our `__init__()` method will need to accept the data as a parameter:

```
class StockAnalyzer:  
    """Provides metrics for technical analysis of a stock."""  
    @validate_df(columns={'open', 'high', 'low', 'close'})  
    def __init__(self, df):  
        """Create a `StockAnalyzer` object with OHLC data"""  
        self.data = df
```

The StockAnalyzer class

- This makes our code cleaner and easier to follow:

```
@property  
def close(self):  
    """Get the close column of the data."""  
    return self.data.close
```

The StockAnalyzer class

- A few calculations will also need the percent change of the close column, so we will make a property for easier access to that as well:

```
@property  
def pct_change(self):  
    """Get the percent change of the close column."""  
    return self.close.pct_change()
```

The StockAnalyzer class

- Since we will be calculating support and resistance levels using the pivot point, which is the average of the high, low, and close on the last day in the data, we will make a property for it, as well:

```
@property  
def pivot_point(self):  
    """Calculate the pivot point."""  
    return (self.last_close + self.last_high  
           + self.last_low) / 3
```



The StockAnalyzer class

- Note that we are also using other properties

```
@property
def last_close(self):
    """Get the value of the last close in the data."""
    return self.data.last('1D').close.iat[0]

@property
def last_high(self):
    """Get the value of the last high in the data."""
    return self.data.last('1D').high.iat[0]

@property
def last_low(self):
    """Get the value of the last low in the data."""
    return self.data.last('1D').low.iat[0]
```

The StockAnalyzer class

- Now, we have everything we need to calculate support and resistance.

```
def resistance(self, level=1):
    """Calculate the resistance at the given level."""
    if level == 1:
        res = (2 * self.pivot_point) - self.last_low
    elif level == 2:
        res = self.pivot_point \
              + (self.last_high - self.last_low)
    elif level == 3:
        res = self.last_high \
              + 2 * (self.pivot_point - self.last_low)
    else:
        raise ValueError('Not a valid level.')
    return res
```

The StockAnalyzer class

- The support() method is defined in a similar fashion:

```
def support(self, level=1):
    """Calculate the support at the given level."""
    if level == 1:
        sup = (2 * self.pivot_point) - self.last_high
    elif level == 2:
        sup = self.pivot_point \
              - (self.last_high - self.last_low)
    elif level == 3:
        sup = self.last_low \
              - 2 * (self.last_high - self.pivot_point)
    else:
        raise ValueError('Not a valid level.')
    return sup
```

The StockAnalyzer class

- Next, we will work on creating methods for analyzing asset volatility.

```
@property
def _max_periods(self):
    """Get the number of trading periods in the data."""
    return self.data.shape[0]
```

The StockAnalyzer class

- Now that we have our maximum, we can define the `daily_std()` method, which calculates the daily standard deviation of the daily percentage change:

```
def daily_std(self, periods=252):
    """
    Calculate daily standard deviation of percent change.
    Parameters:
        - periods: The number of periods to use for the
            calculation; default is 252 for the trading days
            in a year. Note if you provide a number greater
            than the number of trading periods in the data,
            `self._max_periods` will be used instead.
    Returns: The standard deviation
    """
    return self.pct_change\
        [min(periods, self._max_periods) * -1: ].std()
```

The StockAnalyzer class

- While `daily_std()` is useful on its own, we can take this a step further and calculate annualized volatility by multiplying the daily standard deviation by the square root of the number of trading periods in the year, which we assume to be 252:

```
def annualized_volatility(self):  
    """Calculate the annualized volatility."""  
    return self.daily_std() * math.sqrt(252)
```

The StockAnalyzer class

- In addition, we can look at rolling volatility by using the `rolling()` method:

```
def volatility(self, periods=252):
    """Calculate the rolling volatility.
    Parameters:
        - periods: The number of periods to use for the
            calculation; default is 252 for the trading
            days in a year. Note if you provide a number
            greater than the number of trading periods in the
            data, `self._max_periods` will be used instead.
    Returns: A `pandas.Series` object.
    """
    periods = min(periods, self._max_periods)
    return self.close.rolling(periods).std()\
        / math.sqrt(periods)
```

The StockAnalyzer class

- We often want to compare assets, so we provide the `corr_with()` method to calculate the correlations between them using daily percentage change:

```
def corr_with(self, other):
    """Calculate the correlations between dataframes.
    Parameters:
        - other: The other dataframe.
    Returns: A `pandas.Series` object
    """
    return \
        self.data.pct_change().corrwith(other.pct_change())
```

The StockAnalyzer class

- Next, we define some metrics for comparing the level of dispersion of assets.

```
def cv(self):  
    """  
        Calculate the coefficient of variation for the asset.  
        The lower this is, the better the risk/return tradeoff.  
    """  
    return self.close.std() / self.close.mean()  
  
def qcd(self):  
    """Calculate the quantile coefficient of dispersion."""  
    q1, q3 = self.close.quantile([0.25, 0.75])  
    return (q3 - q1) / (q3 + q1)
```

The StockAnalyzer class

- We add the beta() method, which allows the user to specify the index to use as a benchmark:

```
def beta(self, index):
    """
        Calculate the beta of the asset.
    Parameters:
        - index: The data for the index to compare to.
    Returns:
        Beta, a float.
    """
    index_change = index.close.pct_change()
    beta = self.pct_change.cov(index_change) \
        / index_change.var()
    return beta
```

The StockAnalyzer class

- Next, we define a method for calculating the cumulative returns of an asset as a series.
- This is defined as the cumulative product of one plus the percent change in closing price:

```
def cumulative_returns(self):  
    """Calculate cumulative returns for plotting."""  
    return (1 + self.pct_change).cumprod()
```

The StockAnalyzer class

- We will define this as a static method since we will need to calculate this for an index, and not just the data stored in self.data:

```
@staticmethod
def portfolio_return(df):
    """
    Calculate return assuming no distribution per share.
    Parameters:
        - df: The asset's dataframe.
    Returns: The return, as a float.
    """
    start, end = df.close[0], df.close[-1]
    return (end - start) / start
```

The StockAnalyzer class

- Calculating alpha requires calculating the portfolio return of the index and the asset, along with beta:

```
def alpha(self, index, r_f):  
    """  
    Calculates the asset's alpha.  
    Parameters:  
        - index: The index to compare to.  
        - r_f: The risk-free rate of return.  
    Returns: Alpha, as a float.  
    """  
    r_f /= 100  
    r_m = self.portfolio_return(index)  
    beta = self.beta(index)  
    r = self.portfolio_return(self.data)  
    alpha = r - r_f - beta * (r_m - r_f)  
    return alpha
```

The StockAnalyzer class

- We also want to add methods that will tell us whether the asset is in a bear market or a bull market, meaning that it had a decline or increase in stock price of 20% or more in the last 2 months, respectively:

```
def is_bear_market(self):
    """
        Determine if a stock is in a bear market, meaning its
        return in the last 2 months is a decline of 20% or more
    """
    return \
        self.portfolio_return(self.data.last('2M')) <= -.2
def is_bull_market(self):
    """
        Determine if a stock is in a bull market, meaning its
        return in the last 2 months is an increase of >= 20%.
    """
    return \
        self.portfolio_return(self.data.last('2M')) >= .2
```

The StockAnalyzer class

- Lastly, we will add a method for calculating the Sharpe ratio, which tells us the return we receive in excess of the risk-free rate of return for the volatility we take on with the investment:

```
def sharpe_ratio(self, r_f):  
    """  
        Calculates the asset's Sharpe ratio.  
        Parameters:  
            - r_f: The risk-free rate of return.  
        Returns:  
            The Sharpe ratio, as a float.  
    """  
    return (  
        self.cumulative_returns().last('1D').iat[0] - r_f  
    ) / self.cumulative_returns().std()
```

The AssetGroupAnalyzer class

- When objects contain instances of other classes, it is referred to as composition.
- This design decision leaves us with the following very simple UML diagram for the AssetGroupAnalyzer class:

AssetGroupAnalyzer

analyzers

data

group_by : str

analyze(func_name)

The AssetGroupAnalyzer class

```
class AssetGroupAnalyzer:  
    """Analyzes many assets in a dataframe."""  
    @validate_df(columns={'open', 'high', 'low', 'close'})  
    def __init__(self, df, group_by='name'):  
        """  
        Create an `AssetGroupAnalyzer` object with a  
        dataframe of OHLC data and column to group by.  
        """  
        self.data = df  
        if group_by not in self.data.columns:  
            raise ValueError(  
                f'`group_by` column "{group_by}" not in df.'  
            )  
        self.group_by = group_by  
        self.analyzers = self._composition_handler()  
    def _composition_handler(self):  
        """  
        Create a dictionary mapping each group to its analyzer,  
        taking advantage of composition instead of inheritance.  
        """  
        return {  
            group: StockAnalyzer(data)  
            for group, data in self.data.groupby(self.group_by)  
        }
```

The AssetGroupAnalyzer class

```
def analyze(self, func_name, **kwargs):
    """
    Run a `StockAnalyzer` method on all assets.

    Parameters:
        - func_name: The name of the method to run.
        - kwargs: Additional arguments to pass down.

    Returns:
        A dictionary mapping each asset to the result
        of the calculation of that function.
    """

    if not hasattr(StockAnalyzer, func_name):
        raise ValueError(
            f'StockAnalyzer has no "{func_name}" method.'
        )
    if not kwargs:
        kwargs = {}
    return {
        group: getattr(analyzer, func_name)(**kwargs)
        for group, analyzer in self.analyzers.items()
    }
```

Comparing assets

- Let's use the AssetGroupAnalyzer class to compare all the assets we have collected data for.
- As with prior sections, we won't use all the methods in the StockAnalyzer class here, so be sure to try them out on your own:

```
>>> all_assets_analyzer = \  
...     stock_analysis.AssetGroupAnalyzer(all_assets)
```

Comparing assets

- Let's use the CV to see which asset's closing price is the most widely dispersed:

```
>>> all_assets_analyzer.analyze('cv')  
{'Amazon': 0.2658012522278963,  
 'Apple': 0.36991905161737615,  
 'Bitcoin': 0.43597652683008137,  
 'Facebook': 0.19056336194852783,  
 'Google': 0.15038618497328074,  
 'Netflix': 0.20344854330432688,  
 'S&P 500': 0.09536374658108937}
```



Comparing assets

- Using annualized volatility, Facebook looks much more volatile compared to when we used the CV (although still not the most volatile):

```
>>> all_assets_analyzer.analyze('annualized_volatility')
{'Amazon': 0.3851099077041784,
 'Apple': 0.4670809643500882,
 'Bitcoin': 0.4635140114227397,
 'Facebook': 0.45943066572169544,
 'Google': 0.3833720603377728,
 'Netflix': 0.4626772090887299,
 'S&P 500': 0.34491195196047003}
```

Comparing assets

- Given that all the assets have gained value toward the end of our dataset, let's check if any of them have entered a bull market, meaning that the asset's return in the last 2 months is a 20% or greater gain:

```
>>> all_assets_analyzer.analyze('is_bull_market')
{'Amazon': False,
 'Apple': True,
 'Bitcoin': True,
 'Facebook': False,
 'Google': False,
 'Netflix': False,
 'S&P 500': False}
```

Comparing assets

- Positive values greater than 1 indicate volatility higher than the index, while negative values less than -1 indicate inverse relationships to the index:

```
>>> all_assets_analyzer.analyze('beta', index=sp)
{'Amazon': 0.7563691182389207,
 'Apple': 1.173273501105916,
 'Bitcoin': 0.3716024282483362,
 'Facebook': 1.024592821854751,
 'Google': 0.98620762504024,
 'Netflix': 0.7408228073823271,
 'S&P 500': 1.0000000000000002}
```

Comparing assets

- Let's compare the alphas for the assets using the S&P 500 as our index:

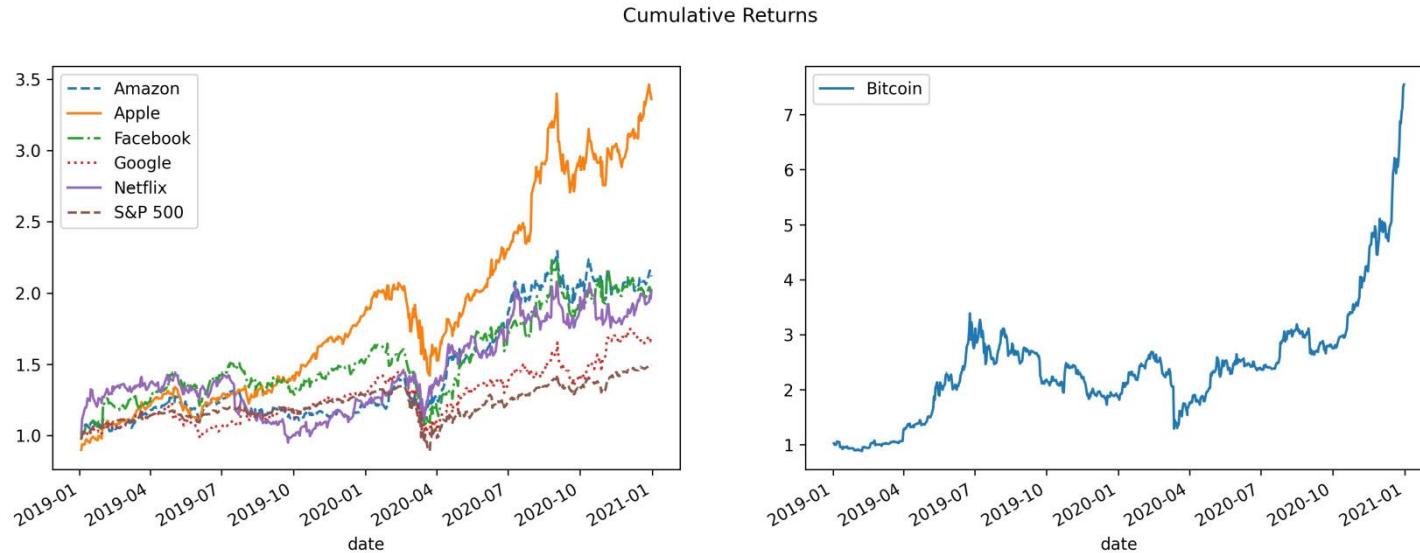
```
>>> r_f = reader.get_risk_free_rate_of_return() # 0.93
>>> all_assets_analyzer.analyze('alpha', index=sp, r_f=r_f)
{'Amazon': 0.7383391908270172,
 'Apple': 1.7801122522388666,
 'Bitcoin': 6.355297988074054,
 'Facebook': 0.5048625273190841,
 'Google': 0.18537197824248092,
 'Netflix': 0.6500392764754642,
 'S&P 500': -1.1102230246251565e-16}
```

Comparing assets

```
>>> from cycler import cycler
>>> bw_viz_cycler = (
...     cycler(color=plt.get_cmap('tab10')(x/10)
...             for x in range(10)))
...     + cycler(linestyle=['dashed', 'solid', 'dashdot',
...                         'dotted', 'solid'] * 2))
>>> fig, axes = plt.subplots(1, 2, figsize=(15, 5))
>>> axes[0].set_prop_cycle(bw_viz_cycler)
>>> cumulative_returns = \
...     all_assets_analyzer.analyze('cumulative_returns')
>>> for name, data in cumulative_returns.items():
...     data.plot(
...         ax=axes[1] if name == 'Bitcoin' else axes[0],
...         label=name, legend=True
...     )
>>> fig.suptitle('Cumulative Returns')
```

Comparing assets

- Despite the struggles in early 2020, all of the assets gained value.



Modeling performance using historical data

- The goal of this section is to give us a taste of how to build some models; as such, the following examples are not meant to be the best possible model, but rather a simple and relatively quick implementation for learning purposes.
- Once again, the stock_analysis package has a class for this section's task: StockModeler.

The StockModeler class

- The StockModeler class will make it easier for us to build and evaluate some simple financial models without needing to interact directly with the statsmodels package.

StockModeler
arima(df) arima_predictions(df, arima_model_fitted, start, end, plot) decompose(df, period, model) plot_residuals(model_fitted, freq) regression(df) regression_predictions(df, model, start, end, plot)

The StockModeler class

- As usual, we start the module with our docstring and imports:

```
"""Simple time series modeling for stocks."""
import matplotlib.pyplot as plt
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.api as sm
from .utils import validate_df
```

The StockModeler class

- Next, we will start the StockModeler class and raise an error if someone tries to instantiate it:

```
class StockModeler:  
    """Static methods for modeling stocks."""  
    def __init__(self):  
        raise NotImplementedError(  
            "This class must be used statically: "  
            "don't instantiate it."  
        )
```

The StockModeler class

- We imported the `seasonal_decompose()` function from `statsmodels`, so we just have to call it on the closing price in our `decompose()` method:

```
@staticmethod
@validate_df(columns={'close'}, instance_method=False)
def decompose(df, period, model='additive'):
    """
    Decompose the closing price of the stock into
    trend, seasonal, and remainder components.

    Parameters:
        - df: The dataframe containing the stock closing
              price as `close` and with a time index.
        - period: The number of periods in the frequency.
        - model: How to compute the decomposition
                 ('additive' or 'multiplicative')

    Returns:
        A `statsmodels` decomposition object.

    """
    return seasonal_decompose(
        df.close, model=model, period=period
    )
```

The StockModeler class

- Notice that we have two decorators for the decompose() method. The topmost decorator is applied on the result of the ones below it.
- In this example, we have the following:

```
staticmethod(  
    validate_df(  
        decompose, columns={'close'}, instance_method=False  
    )  
)
```

The StockModeler class

- In addition, we are going to have our static method provide the option of fitting the model before returning it:

```
@staticmethod
@validate_df(columns={'close'}, instance_method=False)
def arima(df, *, ar, i, ma, fit=True, freq='B'):
    """
        Create an ARIMA object for modeling time series.

    Parameters:
        - df: The dataframe containing the stock closing
              price as `close` and with a time index.
        - ar: The autoregressive order (p).
        - i: The differenced order (q).
        - ma: The moving average order (d).
        - fit: Whether to return the fitted model
        - freq: Frequency of the time series

    Returns:
        A `statsmodels` ARIMA object which you can use
        to fit and predict.
    """
    arima_model = ARIMA(
        df.close.asfreq(freq).fillna(method='ffill'),
        order=(ar, i, ma)
    )
    return arima_model.fit() if fit else arima_model
```

The StockModeler class

- We will also provide the option of getting back the predictions as a Series object or as a plot:

```
@staticmethod
@validate_df(columns={'close'}, instance_method=False)
def arima_predictions(df, arima_model_fitted, start, end,
                      plot=True, **kwargs):
    """
    Get ARIMA predictions as a `Series` object or plot.

    Parameters:
        - df: The dataframe for the stock.
        - arima_model_fitted: The fitted ARIMA model.
        - start: The start date for the predictions.
        - end: The end date for the predictions.
        - plot: Whether to plot the result, default is
            `True` meaning the plot is returned instead of
            the `Series` object containing the predictions.
        - kwargs: Additional arguments to pass down.

    Returns:
        A matplotlib `Axes` object or predictions
        depending on the value of the `plot` argument.
    """
    predictions = \
        arima_model_fitted.predict(start=start, end=end)
    if plot:
        ax = df.close.plot(**kwargs)
        predictions.plot(
            ax=ax, style='r:', label='arima predictions'
        )
        ax.legend()
    return ax if plot else predictions
```

The StockModeler class

- For this, we will once again use statsmodels

```
@staticmethod
@validate_df(columns={'close'}, instance_method=False)
def regression(df):
    """
    Create linear regression of time series with lag=1.
    Parameters:
        - df: The dataframe with the stock data.
    Returns:
        X, Y, and the fitted model
    """
    X = df.close.shift().dropna()
    Y = df.close[1:]
    return X, Y, sm.OLS(Y, X).fit()
```

The StockModeler class

- To handle all this, we will write the `regression_predictions()` method:

```
@staticmethod
@validate_df(columns={'close'}, instance_method=False)
def regression_predictions(df, model, start, end,
                           plot=True, **kwargs):
    """
    Get linear regression predictions as a `pandas.Series` object or plot.

    Parameters:
        - df: The dataframe for the stock.
        - model: The fitted linear regression model.
        - start: The start date for the predictions.
        - end: The end date for the predictions.
        - plot: Whether to plot the result, default is 'True' meaning the plot is returned instead of the `Series` object containing the predictions.
        - kwargs: Additional arguments to pass down.

    Returns:
        A matplotlib `Axes` object or predictions depending on the value of the `plot` argument.
    """

```

The StockModeler class

```
predictions = pd.Series(  
    index=pd.date_range(start, end), name='close'  
)  
last = df.last('1D').close  
for i, date in enumerate(predictions.index):  
    if not i:  
        pred = model.predict(last)  
    else:  
        pred = model.predict(predictions.iloc[i - 1])  
    predictions.loc[date] = pred[0]  
if plot:  
    ax = df.close.plot(**kwargs)  
    predictions.plot(  
        ax=ax, style='r:',  
        label='regression predictions'  
)  
    ax.legend()  
return ax if plot else predictions
```

The StockModeler class

- For this, we will add the `plot_residuals()` method:

```
@staticmethod
def plot_residuals(model_fitted, freq='B'):
    """
    Visualize the residuals from the model.
    Parameters:
        - model_fitted: The fitted model
        - freq: Frequency that the predictions were
            made on. Default is 'B' (business day).
    Returns:
        A matplotlib `Axes` object.
    """
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))
    residuals = pd.Series(
        model_fitted.resid.asfreq(freq), name='residuals'
    )
    residuals.plot(
        style='bo', ax=axes[0], title='Residuals'
    )
    axes[0].set(xlabel='Date', ylabel='Residual')
    residuals.plot(
        kind='kde', ax=axes[1], title='Residuals KDE'
    )
    axes[1].set_xlabel('Residual')
    return axes
```

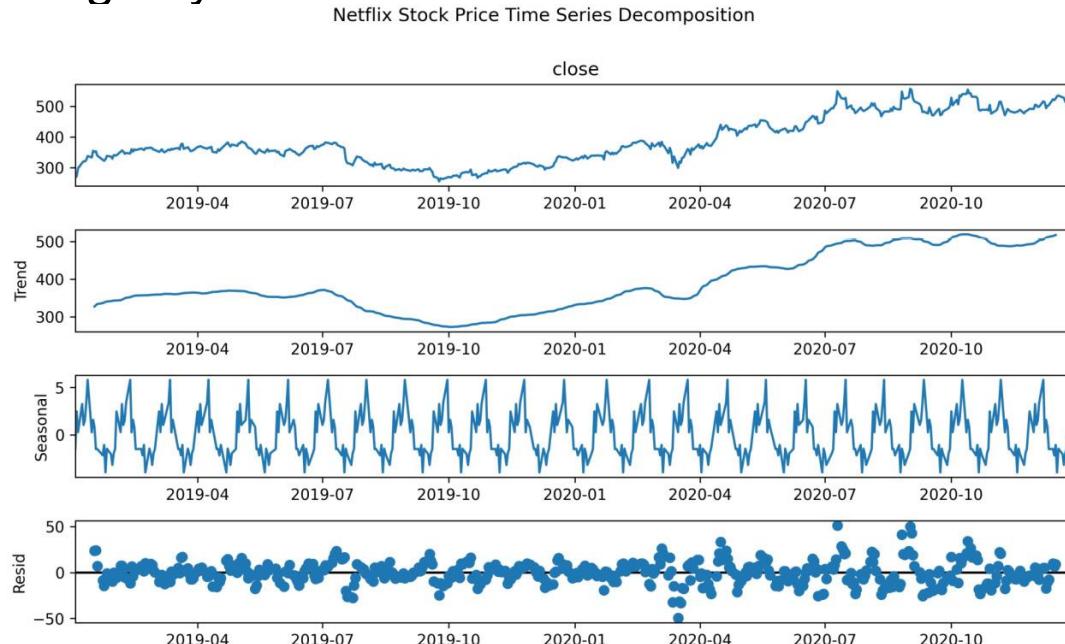
Time series decomposition

- As mentioned in Introduction to Data Analysis, time series can be decomposed into trend, seasonal, and remainder components utilizing a specified frequency.

```
>>> from stock_analysis import StockModeler
>>> decomposition = StockModeler.decompose(nflx, 20)
>>> fig = decomposition.plot()
>>> fig.suptitle(
...     'Netflix Stock Price Time Series Decomposition', y=1
... )
>>> fig.set_figheight(6)
>>> fig.set_figwidth(10)
>>> fig.tight_layout()
```

Time series decomposition

- This returns the decomposition plot for Netflix with a frequency of 20 trading days:



ARIMA

- Here, we will use the %%capture magic to avoid printing any warnings triggered by the ARIMA model fitting, since we are making a simple model to explore functionality:

```
>>> %%capture  
>>> arima_model = StockModeler.arima(nflx, ar=10, i=1, ma=5)
```

ARIMA

- Once the model is fitted, we can obtain information on it with the model's summary() method:

```
>>> print(arima_model.summary())
```



ARIMA

- Be advised that interpreting this summary will require a solid understanding of statistics:

```
SARIMAX Results
=====
Dep. Variable:           close    No. Observations:             522
Model:                 ARIMA(10, 1, 5)   Log Likelihood     -1925.850
Date: Mon, 18 Jan 2021   AIC                  3883.700
Time: 19:02:23          BIC                  3951.792
Sample: 01-02-2019      HQIC                3910.372
                           - 12-31-2020
Covariance Type: opg
=====
            coef    std err      z    P>|z|    [0.025    0.975]
-----
ar.L1    -0.1407    0.254   -0.554    0.580   -0.639    0.358
ar.L2     0.1384    0.178    0.777    0.437   -0.211    0.488
ar.L3    -0.3349    0.165   -2.033    0.042   -0.658   -0.012
ar.L4     0.6575    0.171    3.839    0.000    0.322    0.993
ar.L5     0.5988    0.215    2.787    0.005    0.178    1.020
ar.L6    -0.1005    0.076   -1.315    0.188   -0.250    0.049
ar.L7     0.0555    0.052    1.072    0.284   -0.046    0.157
ar.L8    -0.0522    0.042   -1.256    0.209   -0.134    0.029
ar.L9    -0.0722    0.051   -1.425    0.154   -0.172    0.027
ar.L10   0.1021    0.056    1.813    0.070   -0.008    0.212
ma.L1    -0.0084    0.257   -0.032    0.974   -0.513    0.496
ma.L2    -0.0854    0.196   -0.435    0.663   -0.470    0.299
ma.L3     0.3300    0.184    1.797    0.072   -0.030    0.690
ma.L4    -0.6166    0.174   -3.549    0.000   -0.957   -0.276
ma.L5    -0.5170    0.213   -2.425    0.015   -0.935   -0.099
sigma2   93.0293    3.711   25.071    0.000   85.756   100.302
=====
Ljung-Box (Q):            33.12  Jarque-Bera (JB):        373.34
Prob(Q):                  0.77  Prob(JB):               0.00
Heteroskedasticity (H):   2.46  Skew:                  -0.10
Prob(H) (two-sided):      0.00  Kurtosis:              7.14
=====
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

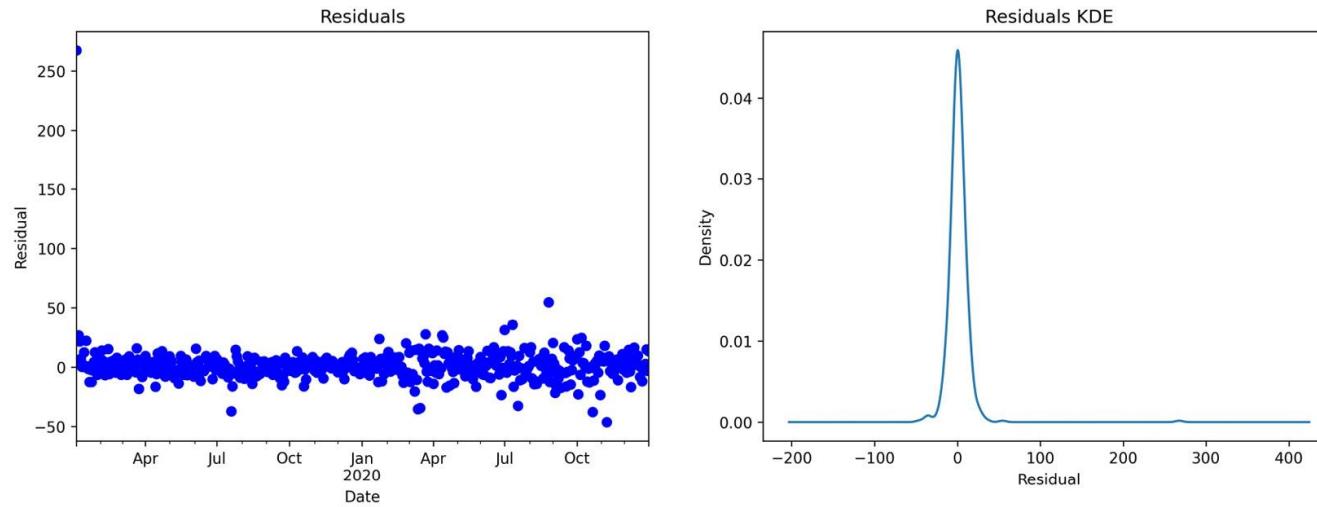
ARIMA

- The StockModeler.plot_residuals() method helps check for this visually:

```
>>> StockModeler.plot_residuals(arima_model)
```

ARIMA

- While the residuals are centered at 0 (right subplot), they are heteroskedastic—note how their variance appears to increase over time (left subplot):



Linear regression with statsmodels

- The StockModeler.regression() method builds a linear regression model for closing price as a function of the prior day's closing price using statsmodels:

```
>>> X, Y, lm = StockModeler.regression(nflx)  
>>> print(lm.summary())
```

Linear regression with statsmodels

- Once again, the `summary()` method gives us statistics on the model's fit:

```
OLS Regression Results
=====
Dep. Variable:          close    R-squared (uncentered):      0.999
Model:                  OLS     Adj. R-squared (uncentered):  0.999
Method:                 Least Squares   F-statistic:           7.470e+05
Date: Mon, 18 Jan 2021   Prob (F-statistic):        0.00
Time: 19:15:40           Log-Likelihood:            -1889.3
No. Observations:      504     AIC:                   3781.
Df Residuals:          503     BIC:                   3785.
Df Model:               1
Covariance Type:       nonrobust
=====
            coef    std err     t      P>|t|      [0.025      0.975]
-----
close      1.0011   0.001    864.291   0.000      0.999      1.003
-----
Omnibus:             50.714   Durbin-Watson:        2.317
Prob(Omnibus):        0.000   Jarque-Bera (JB):    307.035
Skew:                -0.014   Prob(JB):            2.13e-67
Kurtosis:              6.824   Cond. No.             1.00
=====
```

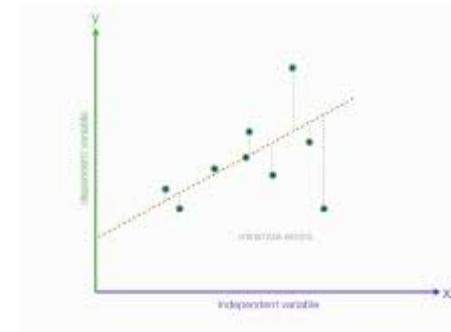
Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Linear regression with statsmodels

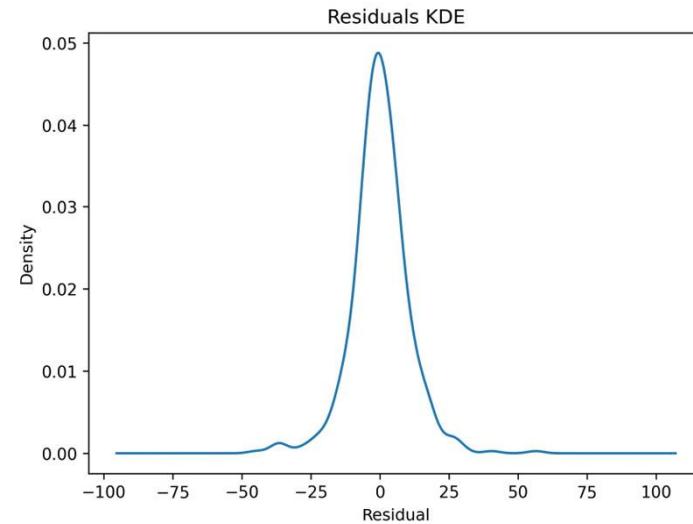
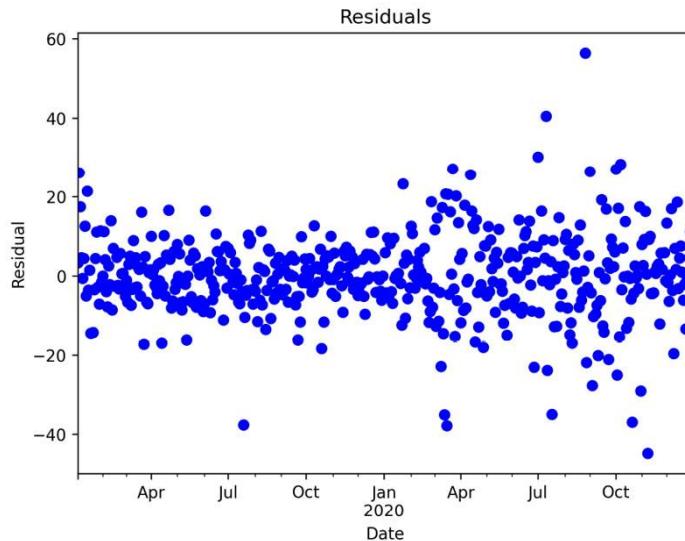
- however, we know that this is simply because stock data is highly autocorrelated, so let's look at the residuals again:

```
>>> StockModeler.plot_residuals(lm)
```



Linear regression with statsmodels

- This model also suffers from heteroskedasticity:

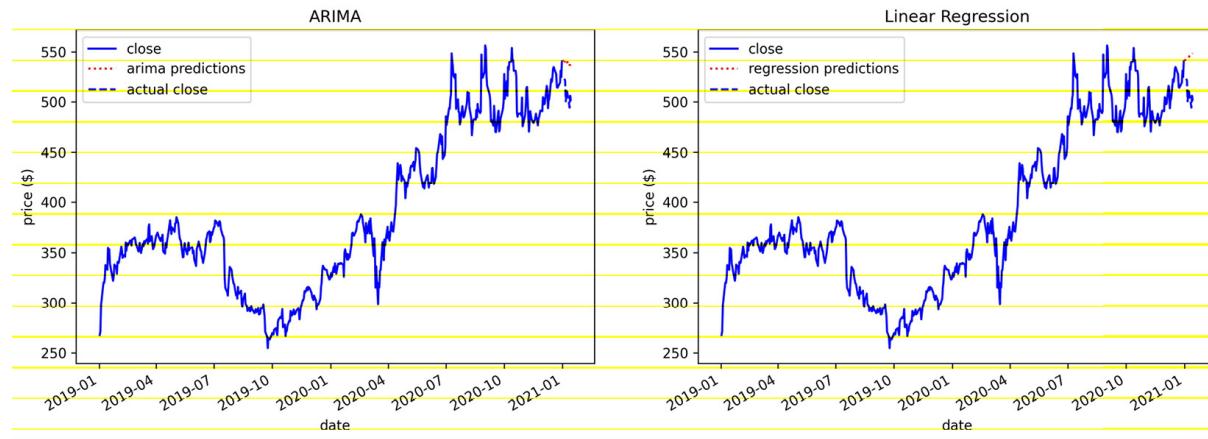


Comparing models

```
>>> import datetime as dt
>>> start = dt.date(2021, 1, 1)
>>> end = dt.date(2021, 1, 14)
>>> jan = stock_analysis.StockReader(start, end) \
...     .get_ticker_data('NFLX')
>>> fig, axes = plt.subplots(1, 2, figsize=(15, 5))
>>> arima_ax = StockModeler.arima_predictions(
...     nflx, arima_model, start=start, end=end,
...     ax=axes[0], title='ARIMA', color='b'
... )
>>> jan.close.plot(
...     ax=arima_ax, style='b--', label='actual close'
... )
>>> arima_ax.legend()
>>> arima_ax.set_ylabel('price ($)')
>>> linear_reg = StockModeler.regression_predictions(
...     nflx, lm, start=start, end=end,
...     ax=axes[1], title='Linear Regression', color='b'
... )
>>> jan.close.plot(
...     ax=linear_reg, style='b--', label='actual close'
... )
>>> linear_reg.legend()
>>> linear_reg.set_ylabel('price ($)')
```

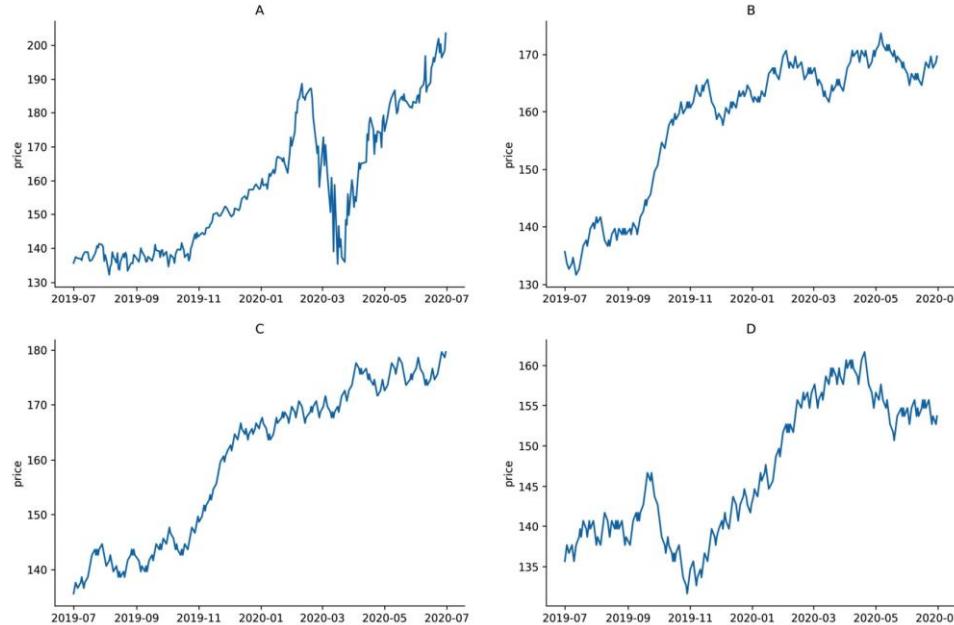
Comparing models

- The ARIMA model's predictions look more in line with the pattern we would expect, but, given the unpredictable nature of the stock market, both models are far off from what actually happened in the first two weeks of January 2021:



Comparing models

- To further illustrate this, take a look at the following set of plots



Summary

- In this lesson, we saw how building Python packages for our analysis applications can make it very easy for others to carry out their own analyses and reproduce ours, as well as for us to create repeatable workflows for future analyses.



"Complete Exercises"

"Complete Lab 12"