Aggregating Pandas DataFrames

21/07/2024

Aggregating Pandas DataFrames

This chapter will get us comfortable with performing analyses using DataFrame objects. Consequently, these topics are more advanced compared to the prior content. The following topics will be covered in this chapter:

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

Performing Database-style Operations on DataFrames

DataFrame objects are analogous to tables in a database: each has a name we refer to it by, is composed of rows, and contains columns of specific data types.

Consequently, pandas allows us to carry out database-style operations on them. Traditionally, databases support a minimum of four operations, called CRUD: Create, Read, Update, and Delete.

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

Pandas provides the query() method so that we can easily write complicated filters instead of using a Boolean mask.

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

	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%";
```

Previously, 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
```

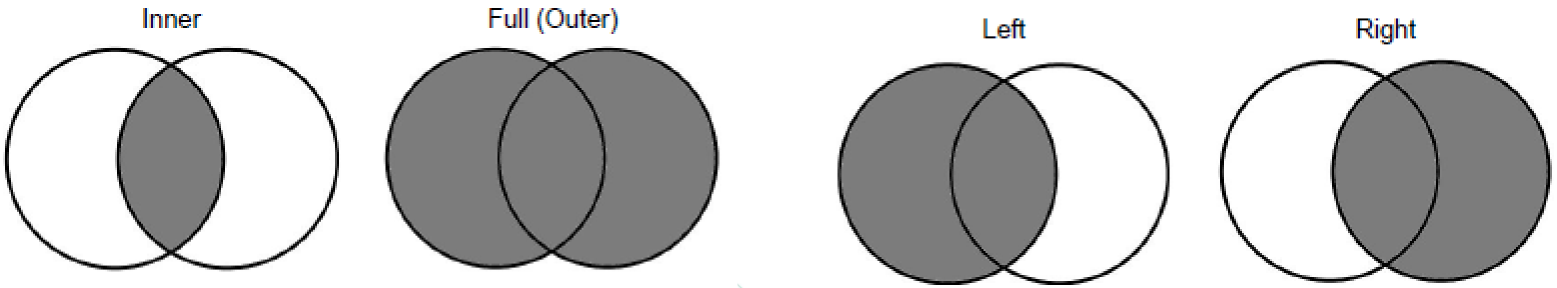
Merging DataFrames

When we discussed stacking dataframes one on top of the other with the pd.concat() function and the append() method, we were performing the equivalent of the SQL UNION ALL statement (or just UNION, if we also removed the duplicates).

Merging dataframes deals with how to line them up by row.

When referring to databases, merging is traditionally called a join. There are four types of joins: full (outer), left, right, and inner.

These join types let us know how the result will be affected by values that are only present on one side of the join.



Merging DataFrames

Inner Join: Only includes rows with keys present in both DataFrames.

python Copy code

```python
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value': [4, 5, 6]})
result = pd.merge(df1, df2, on='key')
# result will be:
#   key  value_x  value_y
# 0   B        2        4
# 1   C        3        5
```

Merging DataFrames

Left Join: Includes all rows from the left DataFrame, with matching rows from the right DataFrame.

```python
result = pd.merge(df1, df2, on='key', how='left')
# result will be:
#   key  value_x  value_y
# 0   A        1      NaN
# 1   B        2      4.0
# 2   C        3      5.0
```

Merging DataFrames

Right Join: Includes all rows from the right DataFrame, with matching rows from the left DataFrame.

python
Copy code

```python
result = pd.merge(df1, df2, on='key', how='right')
# result will be:
#   key  value_x  value_y
# 0   B      2.0        4
# 1   C      3.0        5
# 2   D      NaN        6
```

Merging DataFrames

Outer Join: Includes rows with keys from both DataFrames.

python Copy code

```python
result = pd.merge(df1, df2, on='key', how='outer')
# result will be:
#   key  value_x  value_y
# 0   A      1.0      NaN
# 1   B      2.0      4.0
# 2   C      3.0      5.0
# 3   D      NaN      6.0
```

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()
```

	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

Joins require us to specify how to match the data up. The only data the weather dataframe has in common with the station_info dataframe is the station ID.

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

The difference in the number of unique stations across the dataframes tells us they don't contain all the same stations.

Depending on the type of join we pick, we may lose some data. Therefore, it's important to look at the row count before and after the join.

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

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.

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

Merging DataFrames – Inner Join

We'll begin with the inner join, which will result in the least amount of rows (unless the two dataframes have all the same values for the column being joined on, in which case all the joins will be equivalent).

The inner join will return the columns from both dataframes where they have a match on the specified key column.

Since we will be joining on the weather.station column and the station_info.id column, we will only get weather data for stations that are in station_info.

We will use the merge() method to perform the join (which is an inner join by default) by providing the left and right dataframes, along with specifying which columns to join on.

Merging DataFrames – Inner Join

Since the station ID column is named differently across dataframes, we must specify the names with left_on and right_on. 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)
```

	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 – Inner Join

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)
```

	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	TETERBORO 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 – Left/Right Join

Remember that we had 279 unique stations in the station_info dataframe, but only 110 unique stations for the weather data. When we performed the inner join, we lost all the stations that didn't have weather observations associated with them.

If we don't want to lose rows on a particular side of the join, we can perform a left or right join instead.

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 – Left/Right Join

Wherever the other dataframe contains no data, we will get null values.

	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 – Left/Right Join

Since we placed the station_info dataframe on the left for the left join and on the right for the right join, the results here are equivalent.

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

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 – Outer Join

The final type of join is a full outer join, which will keep all the values, regardless of whether or not they exist in both dataframes.

For instance, say we queried for stations with US1NY in their station ID because we believed that stations measuring NYC weather would have to be labeled as such.

This means that an inner join would result in losing observations from the stations in Connecticut and New Jersey, while a left/right join would result in either lost station information or lost weather data.

The outer join will preserve all the data.

Merging DataFrames – Outer Join

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 – Outer Join

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>;
```

Remember, we had data from two distinct stations: one had a valid station ID and the other was ?. The ? station was the only one recording the water equivalent of snow (WESF). Now that we know about joining dataframes, we can join the data from the valid station ID to the data from the ? station that we are missing by date.

Merging DataFrames

First, we will need to read in the CSV file, setting the date column as the index. 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()
```

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

This time, the column we want to join on (the date) is actually the index, so we will pass in left_index to indicate that the column to use from the left dataframe is the index, and then right_index to indicate the same for the right dataframe.

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

For all the columns that the dataframes had in common, but weren't part of the join, we have two versions now. 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:

date	PRCP_x	SNOW_x	TMAX	TMIN	TOBS	inclement_weather_x	PRCP_y	SNOW_y	WESF	inclement_weather_y
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()
```

date	PRCP	SNOW	TMAX	TMIN	TOBS	inclement_weather	PRCP_?	SNOW_?	WESF	inclement_weather_?
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

When we are joining on the index, an easier way to do this is to use the join() method instead of merge().

It also defaults to an inner join, but this behavior can be changed with the how parameter.

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

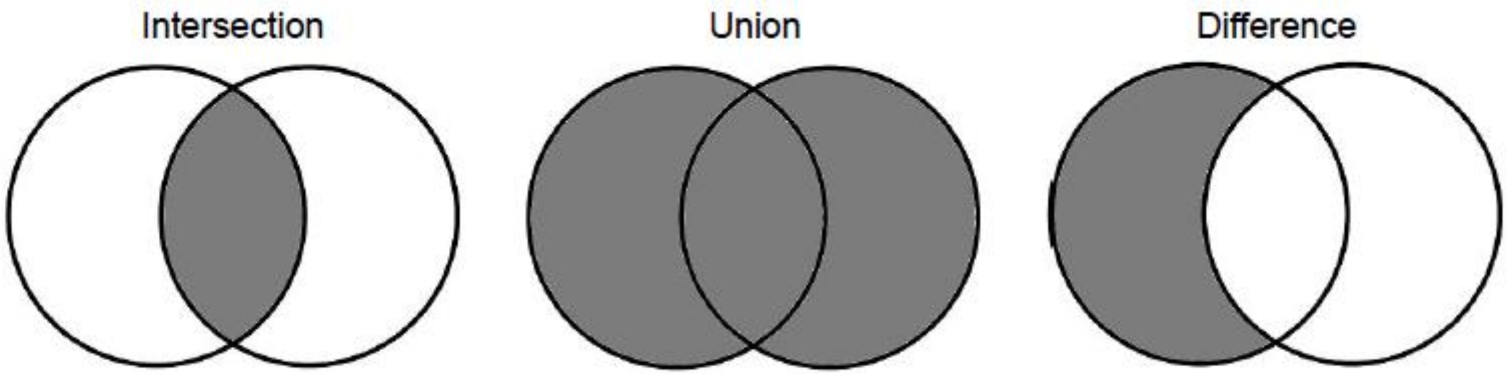
The join() method will always use the index of the left dataframe to join, but it can use a column in the right dataframe if its name is passed to the on parameter.

Merging DataFrames

One important thing to keep in mind is that joins can be rather resource-intensive so it is often beneficial to figure out what will happen to the rows before going through with it.

If we don't already know what type of join we want, this can help give us an idea. We can use set operations on the index we plan to join on to figure this out.

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:



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)
```

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)
```

With the inner join, we only got station information for the stations with weather observations. This doesn't tell us what we lost, though; for this, we need to find the set difference, which will subtract the sets and give us the values of the first index that aren't in the second.

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)
```

Note that this output also tells us how left and right joins will turn out. To avoid losing rows, we want to put the station_info dataframe on the same side as the join (on the left for a left join and on the right for a right join).

Merging DataFrames

> **Tip**
>
> We can use the `symmetric_difference()` method on the indices of the dataframes involved in the join to see what will be lost from both sides: `index_1.symmetric_difference(index_2)`. The result will be the values that are only in one of the indices. An example is in the notebook.

Merging DataFrames

Lastly, we can use the union to view all the values we will keep if we run a full outer join.

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

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:

Using DataFrame Operations to Enrich Data

Now that we've discussed how to query and merge DataFrame objects, let's learn how to perform complex operations on them to create and modify columns and rows.

```
>>> 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
... )
```

We will begin by reviewing operations that summarize entire rows and columns before moving on to binning, applying functions across rows and columns, and window calculations, which summarize data along a certain number of observations at a time (such as moving averages).

Arithmetic and Statistics

To start off, let's create a column with the Z-score for the volume traded in Facebook stock and use it to find the days where the Z-score is greater than three in absolute value.

```
>>> 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')
```

Five days in 2018 had Z-scores for volume traded greater than three in absolute value.

	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

Two other very useful methods are rank() and pct_change(), which let us rank the values of a column (and store them in a new column) and calculate the percentage change between periods, respectively.

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

The largest percentage change in volume traded was on January 12, 2018, coinciding with a Facebook scandal.
This was when Facebook announced changes to the news feed, prioritizing content from friends over brands.
Given that nearly 89% of Facebook's revenue came from advertising in 2017, this announcement caused panic, leading to a significant increase in traded volume and a drop in stock price.

date	open	high	low	close	volume	volume_pct_change	pct_change_rank
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']
```

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):

date	open	high	low	close	volume
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 use aggregated Boolean operations to inspect the dataframe. For example, the any() method shows that Facebook stock never had a daily low price greater than $215 in 2018.

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

To check if all rows meet a criterion, use the all() method. It shows that Facebook had at least one day where the opening, high, low, and closing prices were $215 or less.

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

Binning

Sometimes, it's more convenient to work with categories rather than the specific values. A common example is working with ages.

Binning or discretizing (going from continuous to discrete); we take our data and place the observations into bins (or buckets) matching the range they fall into.

By doing so, we can drastically reduce the number of distinct values our data can take on and make it easier to analyze.

> **Important note**
>
> While binning our data can make certain parts of the analysis easier, keep in mind that it will reduce the information in that field since the granularity is reduced.

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
```

Clearly, we will need to create some ranges for the volume traded in order to look at the days of high trading volume, but how do we decide which range is a good range?

Binning

One way is to use the pd.cut() function for binning based on value. 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; 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

It looks like an overwhelming majority of the trading days were in the low-volume bin; keep in mind that this is all relative because we evenly divided the range between the minimum and maximum trading volumes. Let's look at the data for the three days of high volume:

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

	open	high	low	close	volume
date					~40 million additional shares
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

Facebook stock price July 26, 2018 reveals that Facebook had announced their earnings and disappointing user growth after market close on July 25th. Which was followed by lots of after-hours selling.

When the market opened the next morning, the stock had dropped from $217.50 at close on the 25th to $174.89 at market open on the 26th. Let's pull out this data:

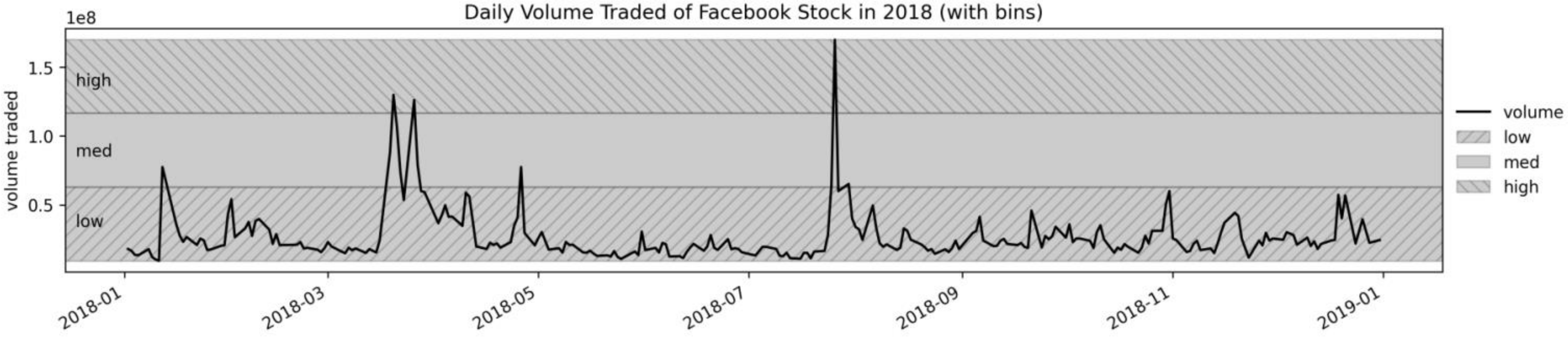
```
>>> fb['2018-07-25':'2018-07-26']
```

Not only was there a huge drop in stock price, the volume traded also skyrocketed, increasing by more than 100 million. All of this resulted in a loss of about $120 billion in Facebook's market capitalization:

	open	high	low	close	volume
date					
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 some of the dates within the medium trading volume group, we can see that many are part of the three trading events.



This forces us to reexamine how we created the bins in the first place. Perhaps equal-width bins wasn't the answer?

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.

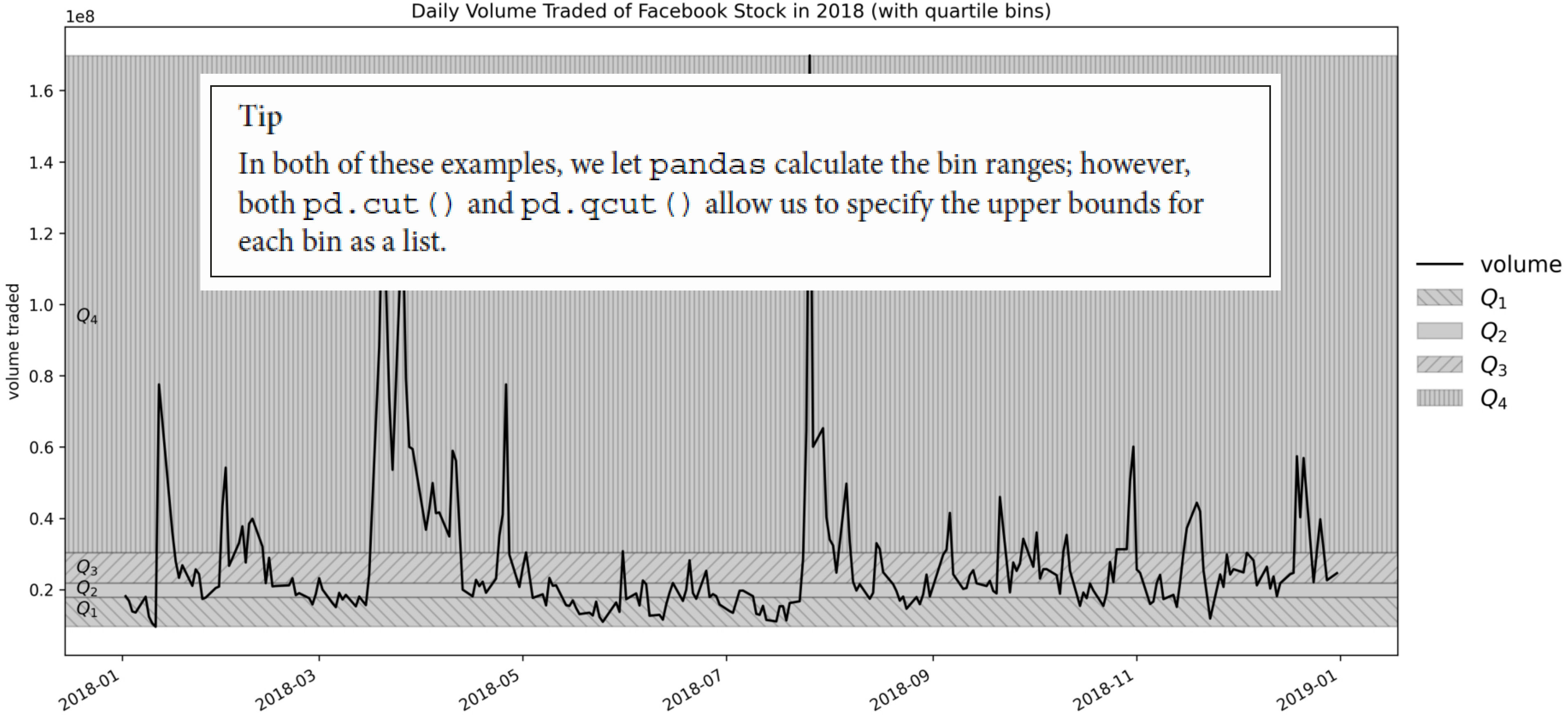
Binning

If we want each bin to have an equal number of observations, we can split the bins based on evenly-spaced quantiles using the pd.qcut() function. 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:



Tip

In both of these examples, we let `pandas` calculate the bin ranges; however, both `pd.cut()` and `pd.qcut()` allow us to specify the upper bounds for each bin as a list.

Applying Functions

So far, most of the actions we have taken on our data have been column-specific. When we want to run the same code on all the columns in our dataframe, we can use the apply() method for more succinct code. Note that this will not be done in-place.

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

Let's calculate the Z-scores of the TMIN (minimum temperature), TMAX (maximum temperature), and PRCP (precipitation) observations in Central Park in October 2018.

```
>>> 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
```

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

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	$\boxed{3.936167}$

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
```

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
```

Window Calculations

All the functions and methods we have used so far have involved the full row or column; however, sometimes, we are more interested in performing window calculations, which use a section of the data.

Pandas makes it possible to perform calculations over a window or range of rows/ columns.

In this section, we will discuss a few ways of constructing these windows. Depending on the type of window, we get a different look at our data.

Window Calculations - Rolling Windows

When our index is of type DatetimeIndex, we can specify the window in day parts (such as 2H for two hours or 3D for three days); otherwise, we can specify the number of periods as an integer. Say we are interested in the amount of rain that has fallen in a rolling 3-day window;

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

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

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

Window Calculations - Rolling Windows

Tip

If we want to use dates for the rolling calculation, but don't have dates in the index, we can pass the name of our date column to the `on` parameter in the call to `rolling()`. Conversely, if we want to use an integer index of row numbers, we can simply pass in an integer as the window; for example, `rolling(3)` for a 3-row window.

Window Calculations - Rolling Windows

To change the aggregation, all we have to do is call a different method on the result of rolling(); for example, mean() for the average and max() for the maximum. The rolling calculation can also be applied to all the columns at once:

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

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 - Rolling Windows

To apply different aggregations across columns, we can use the agg() method instead; it allows us to specify the aggregations to perform per column as a predefined or custom function.

```
>>> 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_
...     lsuffix='_rol
... ).sort_index(axis
```

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

date	AWND	AWND_rolling	PRCP	PRCP_rolling	TMAX	TMAX_rolling	TMIN	TMIN_rolling
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

Expanding calculations will give us the cumulative value of our aggregation function. We use expanding() method to perform a calculation with an expanding window;

```
>>> 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
```

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 datatype	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
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 - Expanding Windows

As we did with rolling(), we can provide column-specific aggregations with the agg() method.

```
>>> 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)
```

date	AWND	AWND_expanding	PRCP	PRCP_expanding	TMAX	TMAX_expanding	TMIN	TMIN_expanding
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

Both rolling and expanding windows equally weight all the observations in the window when performing calculations, Pandas provides the ewm() method for exponentially weighted moving calculations.

```
>>> 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
```

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.

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
```

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']\
...                 .apply(lambda x: (x - x.mean())/x.std()))
```

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())\
...      .pipe(get_info)
```

Pipes can also make it easier to write reusable code.

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

وزارة الاتصــــــالات
وتكنولوجيا المعلومات

رواد مصر الرقمية

Q&A

Questions and answers

Thanks!