وزارة الاتصالات وتكنولوجيا المعلومات

رواد مصر الرقمية

Aggregating Pandas DataFrames

27/07/2024

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Aggregating Data

We will focus on summarizing the dataframe through aggregation, which will change the shape of our dataframe (often through row reduction).

We have already seen some NumPy functions commonly used for aggregations, such as np.sum(), np.mean(), np.min(), and np.max(); however, we aren't limited to numeric operations—we can use things such as np.unique() on strings.

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

Summarizing DataFrames

First, we will take a look at summarizing the full dataset before moving on to summarizing by groups and building pivot tables and crosstabs.

When we discussed window calculations, we saw that we could run the agg() method on the result of rolling(), expanding(), or ewm(); however, we can also call it directly on the dataframe in the same fashion.
The only difference is that the aggregations done this way will be performed on all the data, meaning that we will only get a series back that contains the overall result.

```
>>> 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
```

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:

Summarizing DataFrames

We can use aggregations to easily find the total snowfall and precipitation for 2018 in Central Park. 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
```

Additionally, we can provide multiple functions to run on each of the columns we want to aggregate.

```
>>> 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

So far, we have learned how to aggregate over specific windows and over the entire dataframe; however, the real power comes with the ability to aggregate by group membership.

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:

	open	high	low	close	volume
trading_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')\
...     ['close'].agg(['min', 'max', 'mean'])
```

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

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

Aggregating by Group

If we need more fine-tuned control over how each column gets aggregated, we use the agg() method again with a dictionary that maps the columns to their aggregation function.

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

We now have a hierarchical index in the columns.

	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')],
           )
```

We can use a list comprehension to remove this hierarchy and instead have our column names in the form of <column>_<agg>.

```
>>> 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

Say we want to see the average observed precipitation across all the stations per day. We would need to group by the date, but it is in the index. In this case, we have a few options:

- Resampling, which we will cover in the Working with time series data section, later.
- Resetting the index and using the date column that gets created from the index.
- Passing level=0 to groupby() to indicate that the grouping should be performed on the outermost level of the index.
- Using a Grouper object.

Aggregating by Group

Here, we will pass level=0 to groupby(), but note that we can also pass in level='date' because our index is named.

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

Convert from single column DF to Series

We can also group by many categories at once. Let's find the quarterly total recorded precipitation per station.

رواد مصر الرقمية

وزارة الاتصالات
وتكنولوجيا المعلومات

Aggregating by Group

Here, rather than pass in level=0 to groupby(), we need to use a Grouper object to aggregate from daily to quarterly frequency.

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

	value			
date	**2018-03-31**	**2018-06-30**	**2018-09-30**	**2018-12-31**
station_name				
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

There are many possible follow-ups for this result:

1. We could look at which stations receive the most/least precipitation.
2. We could go back to the location and elevation information we had for each station to see if that affects precipitation.
3. We could also see which quarter has the most/least precipitation across the stations.

Tip

The `DataFrameGroupBy` objects returned by the `groupby()` method have a `filter()` method, which allows us to filter groups. We can use this to exclude certain groups from the aggregation. Simply pass a function that returns a Boolean for each group's subset of the dataframe (`True` to include the group and `False` to exclude it). An example is in the notebook.

Aggregating by Group

Let's see which months have the most precipitation. First, we need to group by day and average the precipitation across the stations. Then, we can group by month and sum the resulting precipitation. Finally, we will use nlargest() to get the five months with the most precipitation:

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

Pivot tables and Crosstabs

To wrap up this section, we will discuss some pandas functions that will aggregate our data into some common formats.

The aggregation methods we discussed previously will give us the highest level of customization; however, pandas provides some functions to quickly generate a pivot table and a crosstab in a common format.

In order to generate a pivot table, we must specify what to group on and, optionally, which subset of columns we want to aggregate and/or how to aggregate (average, by default).

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

Since we passed in columns='trading_volume', the distinct values in the trading_volume column were placed along the columns. The columns from the original dataframe then went to the index.

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

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

Pivot tables and Crosstabs

We can use the pd.crosstab() function to create a frequency table. For example, if we want to see how many low-, medium-, and high-volume trading days Facebook stock had each month, we can use a crosstab.

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

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

Tip

We can normalize the output to percentages of the row/column totals by passing in `normalize='rows'`/`normalize='columns'`. An example is in the notebook.

Pivot tables and Crosstabs

To change the aggregation function, we can provide an argument to values and then specify aggfunc.

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
... )
```

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

We can also get row and column subtotals with the margins parameter. 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	0.00	1.00	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	0.00	1.00	3.00	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	0.00	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	NaN	1.00	NaN	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	0.00	1.00	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	0.00	1.00	0.00	7.00
WOODBRIDGE TWP 3.0 NNW, NJ US	NaN	0.00	0.00	NaN	NaN	0.00	NaN	NaN	NaN	0.00	0.00	NaN	0.00
total observations of snow	190.00	97.00	237.00	81.00	0.00	0.00	0.00	0.00	0.00	0.00	49.00	13.00	667.00

Working with Time Series Data

With time series data, we have some additional operations we can use, for anything from selection and filtering to aggregation. 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']
... ))
```

Note that it's important to set the index to our date (or datetime) column, which will allow us to take advantage of the additional functionality we will be discussing.

Time-based Selection and Filtering

Let's start with a quick recap of datetime slicing and indexing. We can filter to a year (fb.loc['2018’]), 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']
```

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

date	open	high	low	close	volume	trading_volume
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
```

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.

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

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 using the last() method:

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

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')
... )
```

The reindexed data would have all nulls for January 1st and any other days the market was closed.

Time-based Selection and Filtering

We can combine the first(), isna(), and all() methods to confirm this. 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
```

We can use the first_valid_index() method to obtain the index of the first non-null entry in our data, which will be the first day of trading in the data.

To obtain the last day of trading, we can use the last_valid_index() method.

```
>>> 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

If we wanted to know what Facebook's stock price looked like as of March 31, 2018, if we try to do so with loc[] (fb_reindexed.loc['2018-03-31']), we will get null values because the stock market wasn't open that day.

If we use the asof() method instead, it will give us the closest non-null data that precedes the date we ask for, which in this case is March 29th.

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

For the next few examples, we will need time information in addition to the date. The datasets we have been working with thus far lack a time component, so we will switch to the Facebook stock data by the minute from May 20, 2019 through May 24, 2019 from Nasdaq.com.

In order to properly parse the datetimes, we need to pass in a lambda function as the date_parser argument since they are not in a standard format (for instance, May 20, 2019 at 9:30 AM is represented as 2019-05-20 09-30); the lambda function will specify how to convert the data in the date field into datetimes:

```
>>> 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

Important note

In order to properly parse datetimes in a non-standard format, we need to specify the format it is in. For a reference on the available codes, consult the Python documentation at `https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior`.

Time-based Selection and Filtering

We can use first() and last() with agg() to bring this data to a daily granularity.

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

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 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')
```

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

date	open	high	low	close	volume
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

We can use the between_time() method to grab all the rows where the time portion of the datetime is between two times (inclusive of the endpoints by default).

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

date	open	high	low	close	volume
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

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.

Shifting for Lagged Data

We can use the shift() method to create lagged data. By default, the shift will be by one period, but this can be any integer (positive or negative). Let's use shift() to create a new column that indicates the previous day's closing price for the daily Facebook stock 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')
```

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Shifting for Lagged Data

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

date	open	high	low	close	volume	trading_volume	prior_close	after_hours_change_in_price	abs_change
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

Tip

To add/subtract time from the datetimes in the index, consider using `Timedelta` objects instead. There is an example of this in the notebook.

Differenced Data

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:

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

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
```

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:

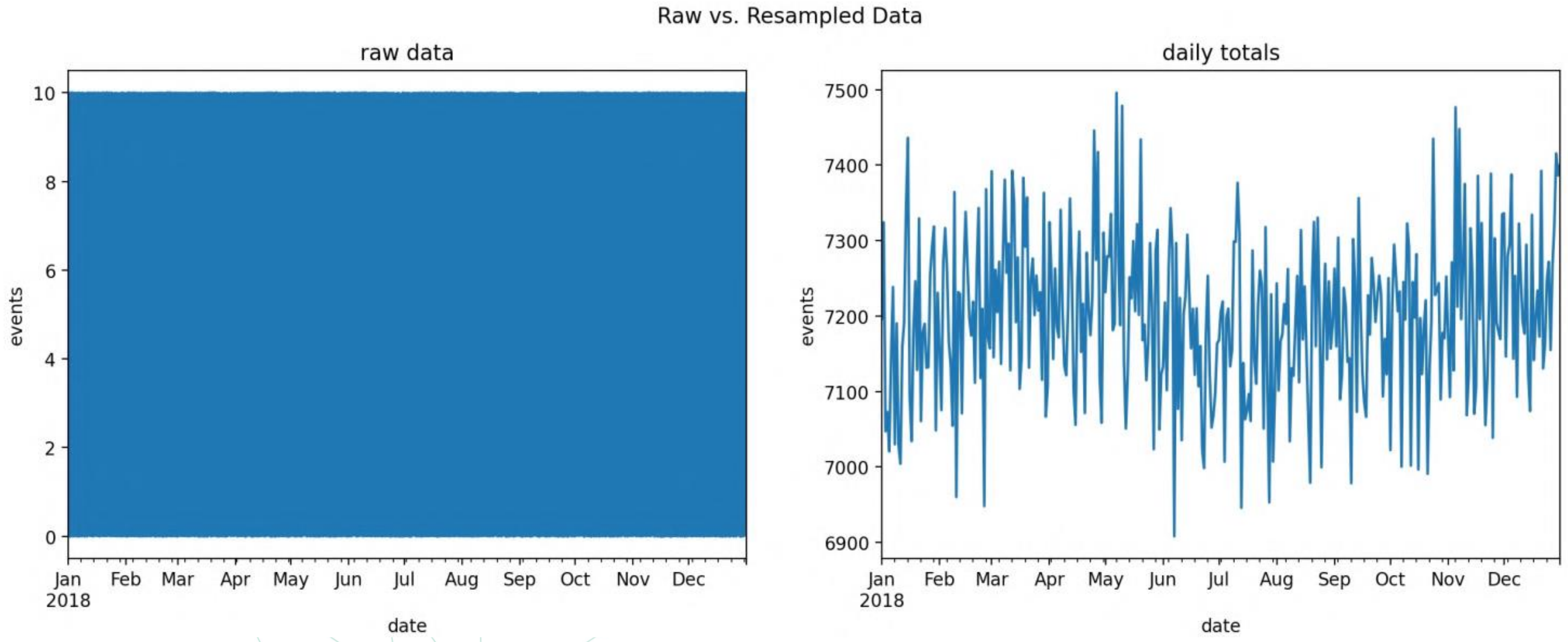
	open	high	low	close	volume
date					
2018-01-02	NaN	NaN	NaN	NaN	NaN
2018-01-03	4.20	3.20	3.7800	3.25	-1265340.0
2018-01-04	3.02	1.43	2.7696	-0.34	-3005667.0
2018-01-05	0.69	0.69	0.8304	2.52	-306361.0
2018-01-08	1.61	2.00	1.4000	1.43	4420191.0

Tip

To specify the number of periods that are used for the difference, simply pass in an integer to `diff()`. Note that this number can be negative. An example of this is in the notebook.

Resampling

Sometimes, the data is at a granularity that isn't conducive to our analysis. Consider the case where we have data per minute for the full year of 2018. The level of granularity and nature of the data may render plotting useless. Therefore, we will need to aggregate the data to a less granular frequency:



Resampling

We can use the resample() method to aggregate our time series data to a different granularity.

To use resample(), all we have to do is say how we want to roll up the data and tack on an optional call to an aggregation method.

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 results:

date	open	high	low	close	volume
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

Let's resample the daily Facebook stock data to the quarterly average:

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

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:

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

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

date	open	high	low	close	volume
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()
```

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

	price
date	
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']
```

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:

date	open	high	low	close
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

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()
```

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

Resampling

The following are a few ways we can handle the NaN values. In the interest of brevity, examples of these are in the notebook:

- Use pad() after resample() to forward fill.
- Call fillna() after resample(), as we saw when we handled missing values.
- Use asfreq() followed by assign() to handle each column individually.

Merging Time Series

Time series often go down to the second or are even more granular, meaning that it can be difficult to merge if the entries don't have the same datetime. Pandas solves this problem with two additional merging functions.

When we want to pair up observations that are close in time, we can use pd.merge_asof() to match on nearby keys rather than on equal keys, like we did with joins.

On the other hand, if we want to match up the equal keys and interleave the keys without matches, we can use pd.merge_ordered().

Merging Time Series

To illustrate how these work, we are going to use the fb_prices and aapl_prices tables in the stocks.db SQLite database.

These contain the prices of Facebook and Apple stock, respectively, along with a timestamp of when the price was recorded.

```
>>> 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')
```

If we use merge() or join(), we will only have values for both Apple and Facebook when the Apple price was at the top of the minute. Instead, to try and line these up, we can perform an as of merge.

Merging Time Series

In order to handle the mismatch, we will specify to merge with the nearest minute (direction='nearest') and require that a match can only occur between times that are within 30 seconds of each other (tolerance).

This will place the Apple data with the minute that it is closest to, so 9:31:52 will go with 9:32 and 9:37:07 will go with 9:37.

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

This is similar to a left join; however, we are more lenient when matching the keys. Note that in the case where multiple entries in the Apple data match the same minute, this function will only keep the closest one.

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:

	FB	AAPL
date		
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()
```

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

> Tip
>
> We can pass `fill_method='ffill'` to `pd.merge_ordered()` to forward-fill the first NaN after a value, but it does not propagate beyond that; alternatively, we can chain a call to `fillna()`. There is an example of this in the notebook.

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Q&A

Questions and answers

Thanks!