Join Using Pandas - 2

Mutating versus filtering joins

Mutating joins:

- Combines data from two tables based on matching observations in both tables

Filtering joins:

- Filter observations from table based on whether or not they match an observation in another table

Left, Right, Inner and Outer Joins are types of mutating joins, where we combined the data from two tables, to return a new table.

Filtering Joins

- Filtering Joins filter observations from table based on whether or not they match an observation in another table.

Semi Join

- A type of filtering join in which we combine the tables based on the joining column and the resultant table contains the intersection of the two tables.
- And returns only 1 match, even if there is a one to many relationship.

What is a semi join?

Left Table

A	B	C
A2	B2	C2
A3	B3	C3
A4	B4	C4

Right Table

C	D
C1	D1
C2	D2
C4	D4
C5	D5

Result Table

A	B	C
A2	B2	C2
A4	B4	C4

Semi joins

- Returns the intersection, similar to an inner join
- Returns only columns from the left table and ***not*** the right
- No duplicates

Example datasets

```
  gid  name
0 1    Rock
1 2    Jazz
2 3    Metal
3 4    Alternative ...
4 5    Rock And Roll
```

```
  tid  name             aid  mtid  gid  composer         u_price
0 1    For Those Ab...  1    1     1    Angus Young,...  0.99
1 2    Balls to the...  2    2     1    nan              0.99
2 3    Fast As a Shark  3    2     1    F. Baltes, S...  0.99
3 4    Restless and...  3    2     1    F. Baltes, R...  0.99
4 5    Princess of ...  3    2     1    Deaffy & R.A...  0.99
```

Step 1 - semi join

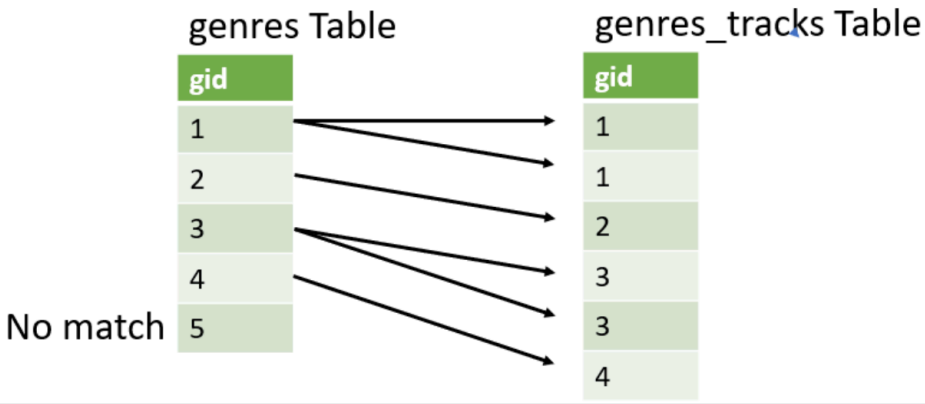
```
genres_tracks = genres.merge(top_tracks, on='gid')
print(genres_tracks.head())
```

```
  gid  name_x  tid   name_y           aid  mtid  composer         u_price
0 1    Rock    2260  Don't Stop M...  185  1     Mercury, Fre...  0.99
1 1    Rock    2933  Mysterious Ways  232  1     U2               0.99
2 1    Rock    2618  Speed Of Light   212  1     Billy Duffy/...  0.99
3 1    Rock    2998  When Love Co...  237  1     Bono/Clayton...  0.99
4 1    Rock    685   Who'll Stop ...  54   1     J. C. Fogerty    0.99
```

Perform an inner join on the ‘gid’ column.

Step 2 - semi join

```
genres['gid'].isin(genres_tracks['gid'])
```



In Step 2, we filter the rows, and return True only those gid rows that have their value in the genre_tracks gid column

Step 3 - semi join

```
genres_tracks = genres.merge(top_tracks, on='gid')
top_genres = genres[genres['gid'].isin(genres_tracks['gid'])]
print(top_genres.head())
```

```
  gid  name
0 1    Rock
1 2    Jazz
2 3    Metal
3 4    Alternative & Punk
4 6    Blues
```

In Step 3, we use the filter we created in step 2, to apply on the inner joined table, to get our semi join table.

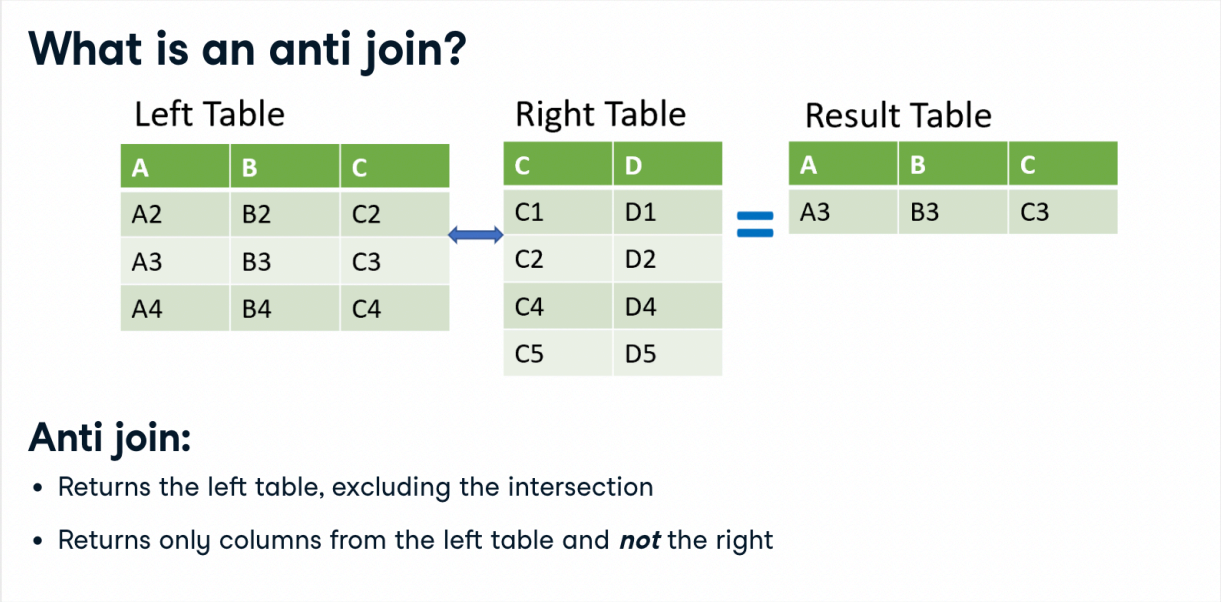
Merge the left and right tables on key column using an inner join.

Search if the key column in the left table is in the merged tables using the `.isin()` method creating a Boolean `Series`.

Subset the rows of the left table.

———————————————

Anti Join
This join operation, return only those rows in the left table, that do not have a matching value in the right table.



Step 1. Perform left join operation and set the indicator parameter to True. Setting the parameter to True, will result in the rows that didn't have a matching gid column in the left table to have left_only as the value in the _merge column.

Step 1 - anti join

```
genres_tracks = genres.merge(top_tracks, on='gid', how='left', indicator=True)
print(genres_tracks.head())
```

```
  gid  name_x           tid     name_y           aid    mtid  composer         u_price  _merge
0 1    Rock             2260.0  Don't Stop M...  185.0  1.0   Mercury, Fre...  0.99     both
1 1    Rock             2933.0  Mysterious Ways  232.0  1.0   U2               0.99     both
2 1    Rock             2618.0  Speed Of Light   212.0  1.0   Billy Duffy/...  0.99     both
3 1    Rock             2998.0  When Love Co...  237.0  1.0   Bono/Clayton...  0.99     both
4 5    Rock And Roll    NaN     NaN              NaN    NaN   NaN              NaN      left_only
```

Step 2 - anti join

```
gid_list = genres_tracks.loc[genres_tracks['_merge'] == 'left_only', 'gid']
print(gid_list.head())
```

```
23     5
34     9
36    11
37    12
38    13
Name: gid, dtype: int64
```

Step2: We will make use of the loc function to get the gids that are only in the in the tracks(left) table.

Step 3 - anti join

```
genres_tracks = genres.merge(top_tracks, on='gid', how='left', indicator=True)
gid_list = genres_tracks.loc[genres_tracks['_merge'] == 'left_only','gid']
non_top_genres = genres[genres['gid'].isin(gid_list)]
print(non_top_genres.head())
```

```
  gid  name
0 5    Rock And Roll
1 9    Pop
2 11   Bossa Nova
3 12   Easy Listening
4 13   Heavy Metal
```

We use the gid list obtained in Step 2, to get the rows that have gid values

Example:
You performed an anti join by first merging the tables with a left join, selecting the ID of those employees who did not support a top customer, and then subsetting the original employee's table.

```
# Merge employees and top_cust
empl_cust = employees.merge(top_cust, on='srid',
                            how='left', indicator=True)

# Select the srid column where _merge is left_only
srid_list = empl_cust.loc[empl_cust['_merge'] == 'left_only', 'srid']

# Get employees not working with top customers
print(employees[employees['srid'].isin(srid_list)])
```

Concatenate DataFrames together vertically

Concatenate two tables vertically

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

A	B	C
A4	B4	C4
A5	B5	C5
A6	B6	C6

- pandas `.concat()` method can concatenate both vertical and horizontal.
  - `axis=0`, vertical

- 3 different tables
- Same column names
- Table variable names:
  - `inv_jan` *(top)*
  - `inv_feb` *(middle)*
  - `inv_mar` *(bottom)*

```
   iid  cid  invoice_date  total
0  1    2    2009-01-01    1.98
1  2    4    2009-01-02    3.96
2  3    8    2009-01-03    5.94
```

```
   iid  cid  invoice_date  total
0  7    38   2009-02-01    1.98
1  8    40   2009-02-01    1.98
2  9    42   2009-02-02    3.96
```

```
   iid  cid  invoice_date  total
0  14   17   2009-03-04    1.98
1  15   19   2009-03-04    1.98
2  16   21   2009-03-05    3.96
```

We can concatenate the tables using the concat method of pandas, to this we will pass in the dataframes we need to concatenate and the result returned will contain the concatenated data frame
Note: The default value for the axis parameter of the concat method is 0, axis = 0 indicates vertical concatenation.

```
pd.concat([inv_jan, inv_feb, inv_mar])
```

	iid	cid	invoice_date	total
0	1	2	2009-01-01	1.98
1	2	4	2009-01-02	3.96
2	3	8	2009-01-03	5.94
0	7	38	2009-02-01	1.98
1	8	40	2009-02-01	1.98
2	9	42	2009-02-02	3.96
0	14	17	2009-03-04	1.98
1	15	19	2009-03-04	1.98
2	16	21	2009-03-05	3.96

As we can see in the above result, we have indexes of the old data frames,

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=True)
```

	iid	cid	invoice_date	total
0	1	2	2009-01-01	1.98
1	2	4	2009-01-02	3.96
2	3	8	2009-01-03	5.94
3	7	38	2009-02-01	1.98
4	8	40	2009-02-01	1.98
5	9	42	2009-02-02	3.96
6	14	17	2009-03-04	1.98
7	15	19	2009-03-04	1.98
8	16	21	2009-03-05	3.96

We can add a key to make the table multiindexed. To do this we have to make sure that the ignore_index parameter is set to False.

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=False,
          keys=['jan','feb','mar'])
```

```
        iid  cid  invoice_date  total
jan 0   1    2    2009-01-01    1.98
    1   2    4    2009-01-02    3.96
    2   3    8    2009-01-03    5.94
feb 0   7    38   2009-02-01    1.98
    1   8    40   2009-02-01    1.98
    2   9    42   2009-02-02    3.96
mar 0   14   17   2009-03-04    1.98
    1   15   19   2009-03-04    1.98
    2   16   21   2009-03-05    3.96
```

Concatenating tables vertically when we have different columns

Table: `inv_jan`

```
   iid  cid  invoice_date  total
0  1    2    2009-01-01    1.98
1  2    4    2009-01-02    3.96
2  3    8    2009-01-03    5.94
```

Table: `inv_feb`

```
   iid  cid  invoice_date  total  bill_ctry
0  7    38   2009-02-01    1.98   Germany
1  8    40   2009-02-01    1.98   France
2  9    42   2009-02-02    3.96   France
```

When we set the sort = True parameter, in the result we can find that the columns are sorted alphabetically by the column name.

```
pd.concat([inv_jan, inv_feb],
          sort=True)
```

	bill_ctry	cid	iid	invoice_date	total
0	NaN	2	1	2009-01-01	1.98
1	NaN	4	2	2009-01-02	3.96
2	NaN	8	3	2009-01-03	5.94
0	Germany	38	7	2009-02-01	1.98
1	France	40	8	2009-02-01	1.98
2	France	42	9	2009-02-02	3.96

If we want to keep only the common columns we can make use of the join parameter, to this if we pass the inner parameter we will be performing inner join operation on the tables and only the common columns will be returned.
Note: By default the value of join in the concat method is outer.

```
pd.concat([inv_jan, inv_feb],
          join='inner')
```

iid	cid	invoice_date	total
1	2	2009-01-01	1.98
2	4	2009-01-02	3.96
3	8	2009-01-03	5.94
7	38	2009-02-01	1.98
8	40	2009-02-01	1.98
9	42	2009-02-02	3.96

Append method - Dataframe method

- This is a simplified version of the concat method, where we are not allowed to change the join type, and we can only set the ignore_index and sort parameters.

`.append()`

- Simplified version of the `.concat()` method
- Supports: `ignore_index` , and `sort`
- Does Not Support: `keys` and `join`
  - Always `join = outer`

```
inv_jan.append([inv_feb, inv_mar],
               ignore_index=True,
               sort=True)
```

	bill_ctry	cid	iid	invoice_date	total
0	NaN	2	1	2009-01-01	1.98
1	NaN	4	2	2009-01-02	3.96
2	NaN	8	3	2009-01-03	5.94
3	Germany	38	7	2009-02-01	1.98
4	France	40	8	2009-02-01	1.98
5	France	42	9	2009-02-02	3.96
6	NaN	17	14	2009-03-04	1.98
7	NaN	19	15	2009-03-04	1.98
8	NaN	21	16	2009-03-05	3.96

We can see that outer join operation has been performed and some column in the first and last rows have NaN values.

Verifying integrity of our data.

- **When merging two columns, there may be some scenarios wherein we do not want one to many relationship, but there exist a row with the same column value, this may create one to many relationship**
- **Similarly when we are concatenating tables vertically, there may be rows that occur in both the dataframes, this will create duplicates in the data.**
- **Luckily pandas provides methods to verify the integrity of the data.**

Let's check our data

Possible merging issue:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

C	D
C1	D1
C1	D2
C1	D3
C2	D4

- Unintentional one-to-many relationship
- Unintentional many-to-many relationship

Possible concatenating issue:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

A	B	C
A3 (duplicate)	B3 (duplicate)	C3 (duplicate)
A4	B4	C4
A5	B5	C5

- Duplicate records possibly unintentionally introduced

Validating merges

`.merge(validate=None)` :

- Checks if merge is of specified type
- `'one_to_one'`
- `'one_to_many'`
- `'many_to_one'`
- `'many_to_many'`

We can specify the type of relationship we require in the validate parameter of the merge method. If we specify it as one_to_one and if the relationship is one to many, then this method will be throwing an error.

Merge dataset for example

Table Name: `tracks`

```
   tid  name             aid  mtid  gid  u_price
0  2    Balls to the...  2    2     1    0.99
1  3    Fast As a Shark  3    2     1    0.99
2  4    Restless and...  3    2     1    0.99
```

Table Name: `specs`

```
   tid  milliseconds  bytes
0  2    342562        5510424
1  3    230619        3990994
2  2    252051        4331779
```

We are expecting the above two tables to produce a one to one, but we can see that there are multiple entries for the same tid in the specs column, when we set the validate to be one to one, we get the below error.

Merge validate: one_to_one

```
tracks.merge(specs, on='tid',
             validate='one_to_one')
```

```
Traceback (most recent call last):
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

Merge validate: one_to_many

```
albums.merge(tracks, on='aid',
             validate='one_to_many')
```

```
  aid  title            artid  tid  name             mtid  gid  u_price
0 2    Balls to the...  2      2    Balls to the...  2     1    0.99
1 3    Restless and...  2      3    Fast As a Shark  2     1    0.99
2 3    Restless and...  2      4    Restless and...  2     1    0.99
```

Verifying concat method.
The concat method has a verify_integrity parameter, which when set to True will check if there are any duplicate rows in the concatenated dataframe. This does not check for duplicate values in the columns.

Verifying concatenations

`.concat(verify_integrity=False)` :

- Check whether the new concatenated index contains duplicates
- Default value is `False`

Dataset for .concat() example

Table Name: `inv_feb`

iid	cid	invoice_date	total
7	38	2009-02-01	1.98
8	40	2009-02-01	1.98
9	42	2009-02-02	3.96

Table Name: `inv_mar`

iid	cid	invoice_date	total
9	17	2009-03-04	1.98
15	19	2009-03-04	1.98
16	21	2009-03-05	3.96

Verifying concatenation: example

```
pd.concat([inv_feb, inv_mar],
          verify_integrity=True)
```

```
Traceback (most recent call last):
ValueError: Indexes have overlapping
values: Int64Index([9], dtype='int64',
name='iid')
```

```
pd.concat([inv_feb, inv_mar],
          verify_integrity=False)
```

iid	cid	invoice_date	total
7	38	2009-02-01	1.98
8	40	2009-02-01	1.98
9	42	2009-02-02	3.96
9	17	2009-03-04	1.98
15	19	2009-03-04	1.98
16	21	2009-03-05	3.96

Why verify integrity and what to do

Why:

- Real world data is often ***NOT*** clean

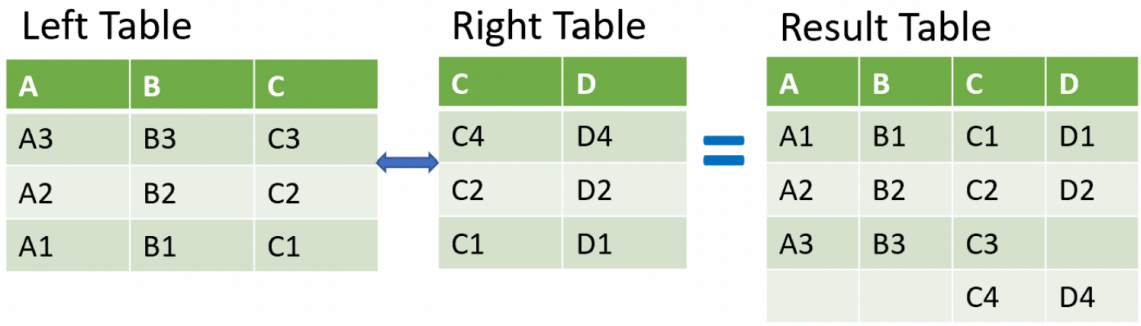
What to do:

- Fix incorrect data
- Drop duplicate rows

---

merge_ordered()

merge_ordered()



The results of the merge_ordered are similar to the merge with the outer join, but with the exception that the results here are sorted.
The ordered results make this an useful method for time series data.

Method comparison

`.merge()` method:

- Column(s) to join on
  - `on` , `left_on` , and `right_on`
- Type of join
  - `how` *(left, right, inner, outer)* {{@}}
  - **default** inner
- Overlapping column names
  - `suffixes`
- Calling the method
  - `df1.merge(df2)`

`merge_ordered()` method:

- Column(s) to join on
  - `on` , `left_on` , and `right_on`
- Type of join
  - `how` *(left, right, inner, outer)*
  - **default** outer
- Overlapping column names
  - `suffixes`
- Calling the function
  - `pd.merge_ordered(df1, df2)`

Merging stock data

```
import pandas as pd
pd.merge_ordered(appl, mcd, on='date', suffixes=('_aapl','_mcd'))
```

```
  date        close_aapl  close_mcd
0 2007-01-01  NaN         44.349998
1 2007-02-01  12.087143   43.689999
2 2007-03-01  13.272857   45.049999
3 2007-04-01  14.257143   48.279999
4 2007-05-01  17.312857   50.549999
5 2007-06-01  17.434286   NaN
```

As we can see there are many NaN values in the above table, we can fill this up using a method called **Forward Fill**

Forward fill

Forward fill example

```
pd.merge_ordered(appl, mcd, on='date',
                 suffixes=('_aapl','_mcd'),
                 fill_method='ffill')
```

	date	close_aapl	close_mcd
0	2007-01-01	NaN	44.349998
1	2007-02-01	12.087143	43.689999
2	2007-03-01	13.272857	45.049999
3	2007-04-01	14.257143	48.279999
4	2007-05-01	17.312857	50.549999
5	2007-06-01	17.434286	50.549999

```
pd.merge_ordered(appl, mcd, on='date',
                 suffixes=('_aapl','_mcd'))
```

	date	close_AAPL	close_mcd
0	2007-01-01	NaN	44.349998
1	2007-02-01	12.087143	43.689999
2	2007-03-01	13.272857	45.049999
3	2007-04-01	14.257143	48.279999
4	2007-05-01	17.312857	50.549999
5	2007-06-01	17.434286	NaN

We can perform forward filling by specifying in the fill_method parameter the ‘ffill’ value.

When to use merge_ordered()?

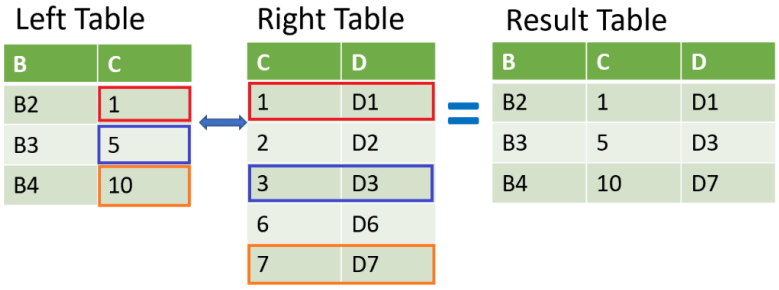
- Ordered data / time series
- Filling in missing values

———————————

merge_asof()

- **merge_asof is similar to the left, as here we are keeping all the rows and columns of the left table, and if there doesn’t exist a matching value for the column in the right table, we will substitute it for the closest value.**

Using merge_asof()



- Similar to a `merge_ordered()` left join
  - Similar features as `merge_ordered()`
- Match on the nearest key column and not exact matches.
  - Merged "on" columns must be sorted.

In the above example we can see that, we did not have a value of 5, in the right C column, so instead the value of the right 3 column has been used.

merge_asof() example

```
pd.merge_asof(visa, ibm, on='date_time',
              suffixes=('_visa','_ibm'))
```

```
   date_time            close_visa  close_ibm
0 2017-11-17 16:00:00   110.32      149.11
1 2017-11-17 17:00:00   110.24      149.83
2 2017-11-17 18:00:00   110.065     149.59
3 2017-11-17 19:00:00   110.04      149.505
4 2017-11-17 20:00:00   110.0       149.42
5 2017-11-17 21:00:00   109.9966    149.26
6 2017-11-17 22:00:00   109.82      148.97
```

Table Name: `ibm`

```
   date_time            close
0  2017-11-17 15:35:12  149.3
1  2017-11-17 15:40:34  149.13
2  2017-11-17 15:45:50  148.98
3  2017-11-17 15:50:20  148.99
4  2017-11-17 15:55:10  149.11
5  2017-11-17 16:00:03  149.25
6  2017-11-17 16:05:06  149.5175
7  2017-11-17 16:10:12  149.57
8  2017-11-17 16:15:30  149.59
9  2017-11-17 16:20:32  149.82
10 2017-11-17 16:25:47  149.96
```

default value for direction is backward

merge_asof() example with direction

```
pd.merge_asof(visa, ibm, on=['date_time'],
              suffixes=('_visa','_ibm'),
              direction='forward')
```

	date_time	close_visa	close_ibm
0	2017-11-17 16:00:00	110.32	149.25
1	2017-11-17 17:00:00	110.24	149.6184
2	2017-11-17 18:00:00	110.065	149.59
3	2017-11-17 19:00:00	110.04	149.505
4	2017-11-17 20:00:00	110.0	149.42
5	2017-11-17 21:00:00	109.9966	149.26
6	2017-11-17 22:00:00	109.82	148.97

Table Name: `ibm`

	date_time	close
0	2017-11-17 15:35:12	149.3
1	2017-11-17 15:40:34	149.13
2	2017-11-17 15:45:50	148.98
3	2017-11-17 15:50:20	148.99
4	2017-11-17 15:55:10	149.11
5	2017-11-17 16:00:03	149.25
6	2017-11-17 16:05:06	149.5175
7	2017-11-17 16:10:12	149.57
8	2017-11-17 16:15:30	149.59
9	2017-11-17 16:20:32	149.82
10	2017-11-17 16:25:47	149.96

When to use merge_asof()

- Data sampled from a process
- Developing a training set (no data leakage)

Selecting data with .query()

- **to the query method we will pass in a string which is similar to the what is passed in the sql where clause**

The .query() method

```
.query('SOME SELECTION STATEMENT')
```

- Accepts an input string
  - Input string used to determine what rows are returned
  - Input string similar to statement after **WHERE** clause in **SQL** statement
    - **Prior knowledge of SQL is not necessary**

Querying on a single condition

This table is `stocks`

	date	disney	nike
0	2019-07-01	143.009995	86.029999
1	2019-08-01	137.259995	84.5
2	2019-09-01	130.320007	93.919998
3	2019-10-01	129.919998	89.550003
4	2019-11-01	151.580002	93.489998
5	2019-12-01	144.630005	101.309998
6	2020-01-01	138.309998	96.300003
7	2020-02-01	117.650002	89.379997
8	2020-03-01	96.599998	82.739998
9	2020-04-01	99.580002	84.629997

```
stocks.query('nike >= 90')
```

	date	disney	nike
2	2019-09-01	130.320007	93.919998
4	2019-11-01	151.580002	93.489998
5	2019-12-01	144.630005	101.309998
6	2020-01-01	138.309998	96.300003

Querying on a multiple conditions, "and", "or"

This table is `stocks`

	date	disney	nike
0	2019-07-01	143.009995	86.029999
1	2019-08-01	137.259995	84.5
2	2019-09-01	130.320007	93.919998
3	2019-10-01	129.919998	89.550003
4	2019-11-01	151.580002	93.489998
5	2019-12-01	144.630005	101.309998
6	2020-01-01	138.309998	96.300003
7	2020-02-01	117.650002	89.379997
8	2020-03-01	96.599998	82.739998
9	2020-04-01	99.580002	84.629997

```
stocks.query('nike > 90 and disney < 140')
```

	date	disney	nike
2	2019-09-01	130.320007	93.919998
6	2020-01-01	138.309998	96.300003

```
stocks.query('nike > 96 or disney < 98')
```

	date	disney	nike
5	2019-12-01	144.630005	101.309998
6	2020-01-01	138.309998	96.300003
28	020-03-01	96.599998	82.739998

Using .query() to select text

```
stocks_long.query('stock=="disney" or (stock=="nike" and close < 90)')
```

```
   date        stock   close
0 2019-07-01   disney  143.009995
1 2019-08-01   disney  137.259995
2 2019-09-01   disney  130.320007
3 2019-10-01   disney  129.919998
4 2019-11-01   disney  151.580002
5 2019-07-01   nike    86.029999
6 2019-08-01   nike    84.5
8 2019-10-01   nike    89.550003
```

- social_fin.query('company == "facebook"')
- social_fin.query('value > 50000000')

Reshaping data with .melt()

- Can unpivot a table

Wide versus long data

Wide Format

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

Long Format

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

What does the .melt() method do?

- The melt method will allow us to unpivot our dataset

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Example of .melt()

```
social_fin_tall = social_fin.melt(id_vars=['financial','company'])
print(social_fin_tall.head(10))
```

```
  financial      company   variable  value
0 total_revenue  twitter   2019      3459329
1 gross_profit   twitter   2019      2322288
2 net_income     twitter   2019      1465659
3 total_revenue  facebook  2019      70697000
4 gross_profit   facebook  2019      57927000
5 net_income     facebook  2019      18485000
6 total_revenue  twitter   2018      3042359
7 gross_profit   twitter   2018      2077362
8 net_income     twitter   2018      1205596
9 total_revenue  facebook  2018      55838000
```

Here id_vars denote the columns which we do not want to change, these can be thought of as the primary key columns

Melting with value_vars

```
social_fin_tall = social_fin.melt(id_vars=['financial','company'],
                                  value_vars=['2018','2017'])
print(social_fin_tall.head(9))
```

```
  financial      company   variable  value
0 total_revenue  twitter   2018      3042359
1 gross_profit   twitter   2018      2077362
2 net_income     twitter   2018      1205596
3 total_revenue  facebook  2018      55838000
4 gross_profit   facebook  2018      46483000
5 net_income     facebook  2018      22112000
6 total_revenue  twitter   2017      2443299
7 gross_profit   twitter   2017      1582057
8 net_income     twitter   2017      -108063
```

value_vars allows us to specify which columns to unpivot, only the included columns are returned in the result.

Melting with column names

```
social_fin_tall = social_fin.melt(id_vars=['financial','company'],
                                  value_vars=['2018','2017'],
                                  var_name=['year'], value_name='dollars')
print(social_fin_tall.head(8))
```

```
  financial      company   year  dollars
0 total_revenue  twitter   2018  3042359
1 gross_profit   twitter   2018  2077362
2 net_income     twitter   2018  1205596
3 total_revenue  facebook  2018  55838000
4 gross_profit   facebook  2018  46483000
5 net_income     facebook  2018  22112000
6 total_revenue  twitter   2017  2443299
7 gross_profit   twitter   2017  1582057
```