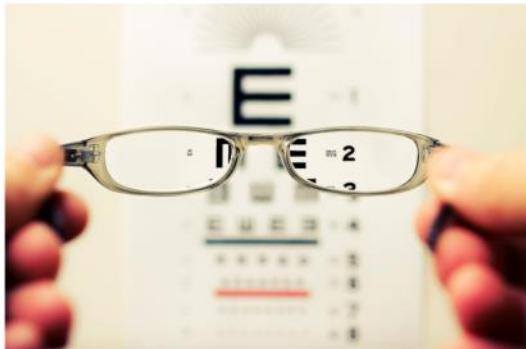


Joining Data with pandas

Wednesday, 11 November 2020 7:25 PM

For clarity



Tables = DataFrames

Merging = Joining

Chicago data portal dataset



The ward data

```
wards = pd.read_csv('Ward_Offices.csv')
print(wards.head())
print(wards.shape)
```

```
   ward  alderman        address      zip
0  1    Proco "Joe" ...  2058 NORTH W...  60647
1  2    Brian Hopkins    1400 NORTH ...  60622
2  3    Pat Dowell      5046 SOUTH S...  60609
3  4    William D. B...  435 EAST 35T...  60616
4  5    Leslie A. Ha...  2325 EAST 71...  60649
(50, 4)
```

Census data

```
census = pd.read_csv('Ward_Census.csv')
print(census.head())
print(census.shape)
```

```
   ward  pop_2000  pop_2010  change        address      zip
0  1    52951     56149    6%  2765 WEST SA...  60647
1  2    54361     55805    3%  WM WASTE MAN...  60622
2  3    40385     53039   31%  17 EAST 38TH...  60653
3  4    51953     54589    5%  31ST ST HARB...  60653
4  5    55302     51455   -7%  JACKSON PARK...  60637
(50, 6)
```

Merging tables

	ward	alderman	address	zip
0	1	Proco "Joe" ...	2058 NORTH W...	60647
1	2	Brian Hopkins	1400 NORTH ...	60622
2	3	Pat Dowell	5046 SOUTH S...	60609
3	4	William D. B...	435 EAST 35T...	60616
4	5	Leslie A. Ha...	2325 EAST 71...	60649

	ward	pop_2000	pop_2010	change	address	zip
0	1	52951	56149	6%	2765 WEST SA...	60647
1	2	54361	55805	3%	WM WASTE MAN...	60622
2	3	40385	53039	31%	17 EAST 38TH...	60653
3	4	51953	54589	5%	31ST ST HARB...	60653
4	5	55302	51455	-7%	JACKSON PARK...	60637

Inner join

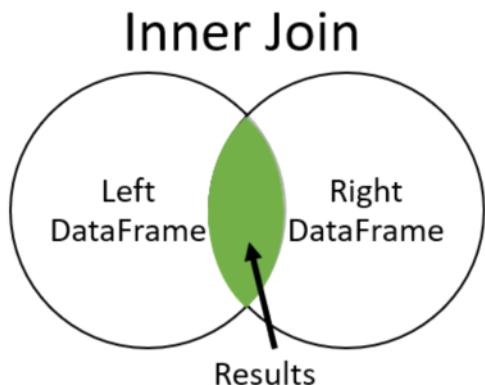
```
wards_census = wards.merge(census, on='ward')
print(wards_census.head(4))
```

	ward	alderman	address_x	zip_x	pop_2000	pop_2010	change	address_y	zip_y
0	1	Proco "Joe" ...	2058 NORTH W...	60647	52951	56149	6%	2765 WEST SA...	60647
1	2	Brian Hopkins	1400 NORTH ...	60622	54361	55805	3%	WM WASTE MAN...	60622
2	3	Pat Dowell	5046 SOUTH S...	60609	40385	53039	31%	17 EAST 38TH...	60653
3	4	William D. B...	435 EAST 35T...	60616	51953	54589	5%	31ST ST HARB...	60653

```
print(wards_census.shape)
```

```
(50, 9)
```

Inner join



An inner join will only return rows that have matching values in both tables.

Suffixes

```
print(wards_census.columns)

Index(['ward', 'alderman', 'address_x', 'zip_x', 'pop_2000', 'pop_2010', 'change',
       'address_y', 'zip_y'],
      dtype='object')
```

To avoid multiple columns with the same name, they are automatically given a suffix by the [merge method](#).

Suffixes

```
wards_census = wards.merge(census, on='ward', suffixes=('_ward', '_cen'))
print(wards_census.head())
print(wards_census.shape)
```

ward	alderman	address_ward	zip_ward	pop_2000	pop_2010	change	address_cen	zi
0 1	Proco "Joe" ...	2058 NORTH W...	60647	52951	56149	6%	2765 WEST SA...	60
1 2	Brian Hopkins	1400 NORTH ...	60622	54361	55805	3%	WM WASTE MAN...	60
2 3	Pat Dowell	5046 SOUTH S...	60609	40385	53039	31%	17 EAST 38TH...	60
3 4	William D. B...	435 EAST 35T...	60616	51953	54589	5%	31ST ST HARB...	60
4 5	Leslie A. Ha...	2325 EAST 71...	60649	55302	51455	-7%	JACKSON PARK...	60
(50, 9)								

One-to-one

A	B	C	C	D
A1	B1	C1	C1	D1
A2	B2	C2	C2	D2
A3	B3	C3	C3	D3

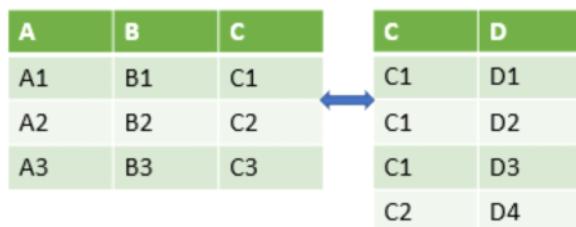
One-To-One = Every row in the left table is related to only one row in the right table

One-to-one example

	ward	alderman	address	zip
0	1	Proco "Joe" ...	2058 NORTH W...	60647
1	2	Brian Hopkins	1400 NORTH ...	60622
2	3	Pat Dowell	5046 SOUTH S...	60609
3	4	William D. B...	435 EAST 35T...	60616
4	5	Leslie A. Ha...	2325 EAST 71...	60649

	ward	pop_2000	pop_2010	change	address	zip
0	1	52951	56149	6%	2765 WEST SA...	60647
1	2	54361	55805	3%	WM WASTE MAN...	60622
2	3	40385	53039	31%	17 EAST 38TH...	60653
3	4	51953	54589	5%	31ST ST HARB...	60653
4	5	55302	51455	-7%	JACKSON PARK...	60637

One-to-many



One-To-Many = Every row in left table is related to one or more rows in the right table

One-to-many example



We will merge the wards table with a table of licensed businesses in each ward.

One-to-many example

```
licenses = pd.read_csv('Business_Licenses.csv')
print(licenses.head())
print(licenses.shape)
```

	account	ward	aid	business	address	zip
0	307071	3	743	REGGIE'S BAR...	2105 S STATE ST	60616
1	10	10	829	HONEYBEERS	13200 S HOUS...	60633
2	10002	14	775	CELINA DELI	5089 S ARCHE...	60632
3	10005	12	nan	KRAFT FOODS ...	2005 W 43RD ST	60609
4	10044	44	638	NEYBOUR'S TA...	3651 N SOUTH...	60613
	(10000, 6)					

One-to-many example

```
ward_licenses = wards.merge(licenses, on='ward', suffixes=('_ward', '_lic'))
print(ward_licenses.head())
```

	ward	alderman	address_ward	zip_ward	account	aid	business	address_lic
0	1	Proco "Joe" ...	2058 NORTH W...	60647	12024	nan	DIGILOG ELEC...	1038 N ASHLA...
1	1	Proco "Joe" ...	2058 NORTH W...	60647	14446	743	EMPTY BOTTLE...	1035 N WESTE...
2	1	Proco "Joe" ...	2058 NORTH W...	60647	14624	775	LITTLE MEL'S...	2205 N CALIF...
3	1	Proco "Joe" ...	2058 NORTH W...	60647	14987	nan	MR. BROWN'S ...	2301 W CHICA...
4	1	Proco "Joe" ...	2058 NORTH W...	60647	15642	814	Beat Kitchen	2000-2100 W ...

Pandas takes care of the one-to-many relationships for us and doesn't require anything special on our end.

We can use the same syntax as we did with one-to-one relationships.

One-to-many example

```
print(wards.shape)
```

```
(50, 4)
```

```
print(ward_licenses.shape)
```

```
(10000, 9)
```

of rows returned will likely be different than the number in the left table.

Merging multiple tables

A	B	C	C	D
A1	B1	C1	C1	D1
A2	B2	C2	C2	D2
A3	B3	C3	C3	D3

A	B	C	C	E	E	F	G
A1	B1	C1	C1	E1	E1	F1	G1
A2	B2	C2	C2	E2	E2	F2	G2
A3	B3	C3	C3	E3	E3	F3	G3

Sometimes we need to merge together more than just two tables to complete our analysis.

Theoretical merge

```
grants_licenses = grants.merge(licenses, on='zip')
print(grants_licenses.loc[grants_licenses['business']=="REGGIE'S BAR & GRILL",
                         ['grant', 'company', 'account', 'ward', 'business']])
```

	grant	company	account	ward	business
0	136443.07	CEDARS MEDIT...	307071	3	REGGIE'S BAR...
1	39943.15	DARRYL & FYL...	307071	3	REGGIE'S BAR...
2	31250.0	JGF MANAGEMENT	307071	3	REGGIE'S BAR...
3	143427.79	HYDE PARK AN...	307071	3	REGGIE'S BAR...
4	69500.0	ZBERRY INC	307071	3	REGGIE'S BAR...

Single merge

```
grants.merge(licenses, on=['address', 'zip'])
```

	address	zip	grant	company	account	ward	aid	business
0	1020 N KOLMA...	60651	68309.8	TRITON INDUS...	7689	37	929	TRITON INDUS...
1	10241 S COMM...	60617	33275.5	SOUTH CHICAG...	246598	10	nan	SOUTH CHICAG...
2	11612 S WEST...	60643	30487.5	BEVERLY RECO...	3705	19	nan	BEVERLY RECO...
3	1600 S KOSTN...	60623	128513.7	CHARTER STEE...	293825	24	nan	LEELO STEEL,...
4	1647 W FULTO...	60612	5634.0	SN PECK BUIL...	85595	27	673	S.N. PECK BU...

requiring that both the address and zip code of a row in the left table match the
address

and zip code of a row in the right table in order for them to be linked to each other in the
merge.

Merging multiple tables

```
grants_licenses_ward = grants.merge(licenses, on=['address','zip']) \
    .merge(wards, on='ward', suffixes=('_bus','_ward'))
grants_licenses_ward.head()
```

First, we merge the grants table with the wards table on the

ward column again, adding suffixes to the repeated column names.

Note that we're using Python's backslash line continuation method to add the second merge on the next line.

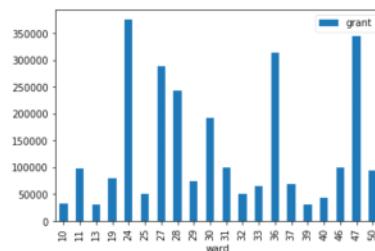
Python will read this as just one line of code.

	address_bus	zip_bus	grant	company	account	ward	aid	business	alderme
0	1020 N KOLMA...	60651	68309.8	TRITON INDUS...	7689	37	929	TRITON INDUS...	Emma M.
1	10241 S COMM...	60617	33275.5	SOUTH CHICAG...	246598	18	nan	SOUTH CHICAG...	Susan S.
2	11612 S WEST...	60643	30487.5	BEVERLY RECO...	3705	19	nan	BEVERLY RECO...	Matthew
3	3502 W 111TH ST	60655	50000.0	FACE TO FACE...	263274	19	704	FACE TO FACE	Matthew
4	1600 S KOSTN...	60623	128513.7	CHARTER STEE...	293825	24	nan	LEELO STEEL...	Michael

Now our output table has information about grants, business, and wards.

Results

```
import matplotlib.pyplot as plt
grant_licenses_ward.groupby('ward').agg('sum').plot(kind='bar', y='grant')
plt.show()
```



We can now sum the grants by ward and plot the results.

Merging even more...

Three tables:

```
df1.merge(df2, on='col') \  
.merge(df3, on='col')
```

Four tables:

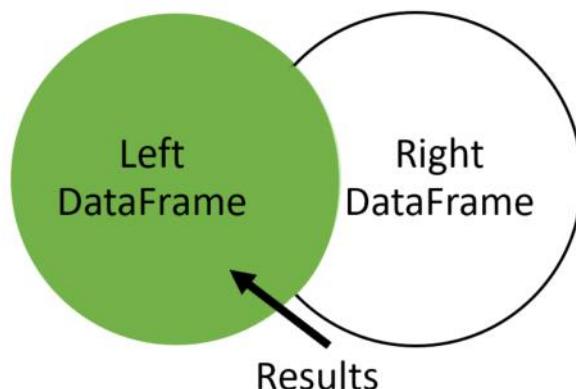
```
df1.merge(df2, on='col') \  
.merge(df3, on='col') \  
.merge(df4, on='col')
```

We stopped at three, but if needed, we could continue to add more.

Merging Tables With Different Join Types

Left join

Left Join



Left 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	D
A2	B2	C2	D2
A3	B3	C3	
A4	B4	C4	D4

We want to use a left join to merge them on key column C.

New dataset



Movies table

```
movies = pd.read_csv('tmdb_movies.csv')
print(movies.head())
print(movies.shape)
```

	id	original_title	popularity	release_date
0	257	Oliver Twist	20.415572	2005-09-23
1	14290	Better Luck ...	3.877036	2002-01-12
2	38365	Grown Ups	38.864027	2010-06-24
3	9672	Infamous	3.6808959999...	2006-11-16
4	12819	Alpha and Omega	12.300789	2010-09-17
	(4803, 4)			

Tagline table

```
taglines = pd.read_csv('tmdb_taglines.csv')
print(taglines.head())
print(taglines.shape)
```

	id	tagline
0	19995	Enter the World of Pandora.
1	285	At the end of the world, the adventure begins.
2	206647	A Plan No One Escapes
3	49026	The Legend Ends
4	49529	Lost in our world, found in another.
	(3955, 2)	

Merge with left join

```
movies_taglines = movies.merge(taglines, on='id', how='left')
print(movies_taglines.head())
```

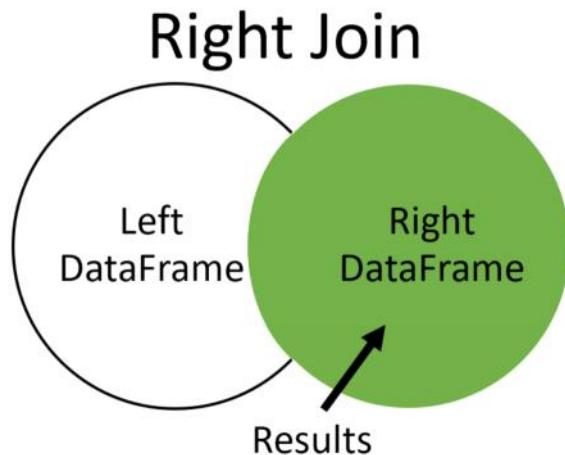
	id	original_title	popularity	release_date	tagline
0	257	Oliver Twist	20.415572	2005-09-23	NaN
1	14290	Better Luck ...	3.877036	2002-01-12	Never undere...
2	38365	Grown Ups	38.864027	2010-06-24	Boys will be...
3	9672	Infamous	3.6808959999...	2006-11-16	There's more...
4	12819	Alpha and Omega	12.300789	2010-09-17	A Pawsome 3D...

Number of rows returned

```
print(movies_taglines.shape)
```

```
(4805, 5)
```

Right join



Right 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	D
		C1	D1
A2	B2	C2	D2
A4	B4	C4	D4
		C5	D5

Looking at data

```
movie_to_genres = pd.read_csv('tmdb_movie_to_genres.csv')
tv_genre = movie_to_genres[movie_to_genres['genre'] == 'TV Movie']
print(tv_genre)
```

```
    movie_id  genre
4998    10947  TV Movie
5994    13187  TV Movie
7443    22488  TV Movie
10061   78814  TV Movie
10790   153397  TV Movie
10835   158150  TV Movie
11096   205321  TV Movie
11282   231617  TV Movie
```

Filtering the data

```
m = movie_to_genres['genre'] == 'TV Movie'
tv_genre = movie_to_genres[m]
print(tv_genre)
```

```
    movie_id  genre
4998    10947  TV Movie
5994    13187  TV Movie
7443    22488  TV Movie
10061   78814  TV Movie
10790   153397  TV Movie
10835   158150  TV Movie
11096   205321  TV Movie
11282   231617  TV Movie
```

Data to merge

	<code>id</code>	<code>title</code>	<code>popularity</code>	<code>release_date</code>
0	257	Oliver Twist	20.415572	2005-09-23
1	14290	Better Luck ...	3.877036	2002-01-12
2	38365	Grown Ups	38.864027	2010-06-24
3	9672	Infamous	3.6808959999...	2006-11-16
4	12819	Alpha and Omega	12.300789	2010-09-17

	<code>movie_id</code>	<code>genre</code>
4998	10947	TV Movie
5994	13187	TV Movie
7443	22488	TV Movie
10061	78814	TV Movie
10790	153397	TV Movie

Merge with right join

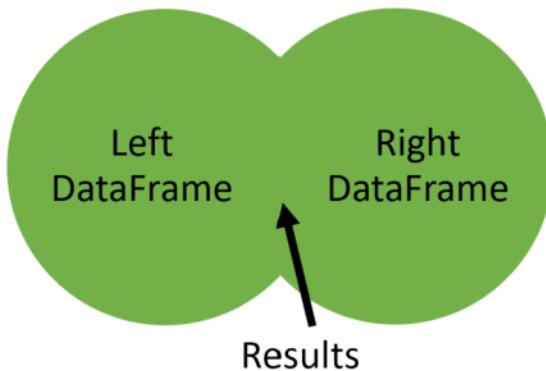
```
tv_movies = movies.merge(tv_genre, how='right',
                        left_on='id', right_on='movie_id')
print(tv_movies.head())
```

We list movies as the left table, so we set `left_on` to `id` and `right_on` to `movie_id`.

	<code>id</code>	<code>title</code>	<code>popularity</code>	<code>release_date</code>	<code>movie_id</code>	<code>genre</code>
0	153397	Restless	0.812776	2012-12-07	153397	TV Movie
1	10947	High School ...	16.536374	2006-01-20	10947	TV Movie
2	231617	Signed, Seal...	1.444476	2013-10-13	231617	TV Movie
3	78814	We Have Your...	0.102003	2011-11-12	78814	TV Movie
4	158150	How to Fall ...	1.923514	2012-07-21	158150	TV Movie

Outer join

Outer Join



Outer join

Left Table			Right Table		Result Table			
A	B	C	C	D	A	B	C	D
A2	B2	C2	C1	D1			C1	D1
A3	B3	C3	C2	D2	A2	B2	C2	D2
A4	B4	C4	C4	D4	A3	B3	C3	
			C5	D5	A4	B4	C4	D4
							C5	D5

Datasets for outer join

```
m = movie_to_genres['genre'] == 'Family'  
family = movie_to_genres[m].head(3)
```

```
movie_id  genre  
0 12      Family  
1 35      Family  
2 105     Family
```

```
m = movie_to_genres['genre'] == 'Comedy'  
comedy = movie_to_genres[m].head(3)
```

```
movie_id  genre  
0 5       Comedy  
1 13      Comedy  
2 35      Comedy
```

Merge with outer join

```
family_comedy = family.merge(comedy, on='movie_id', how='outer',  
                             suffixes=('_fam', '_com'))  
print(family_comedy)
```

```
movie_id  genre_fam  genre_com  
0 12      Family     NaN  
1 35      Family     Comedy  
2 105     Family     NaN  
3 5       NaN        Comedy  
4 13      NaN        Comedy
```

In our original comedy tables ID number 12 does not exist.

Therefore a null is shown.

Similarly, in our last row, movie ID 13 wasn't in the family dataset so it has a null.

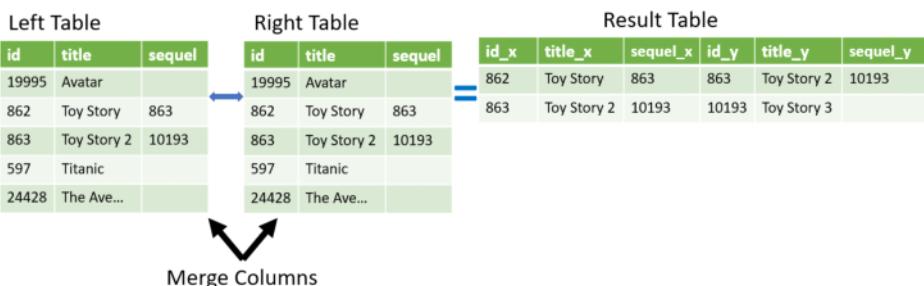
Sequel movie data

```
print(sequel.head())
```

	id	title	sequel
0	19995	Avatar	NaN
1	862	Toy Story	863
2	863	Toy Story 2	10193
3	597	Titanic	NaN
4	24428	The Avengers	NaN



Merging a table to itself



Merging a table to itself

```
original_sequels = sequels.merge(sequels, left_on='sequel', right_on='id',
                                  suffixes=('_org', '_seq'))
print(original_sequels.head())
```

	id_org	title_org	sequel_org	id_seq	title_seq	sequel_seq
0	862	Toy Story	863	863	Toy Story 2	10193
1	863	Toy Story 2	10193	10193	Toy Story 3	NaN
2	675	Harry Potter...	767	767	Harry Potter...	NaN
3	121	The Lord of ...	122	122	The Lord of ...	NaN
4	120	The Lord of ...	121	121	The Lord of ...	122

Continue format results

```
print(original_sequels[['title_org','title_seq']].head())
```

```
    title_org      title_seq
0 Toy Story      Toy Story 2
1 Toy Story 2    Toy Story 3
2 Harry Potter... Harry Potter...
3 The Lord of ... The Lord of ...
4 The Lord of ... The Lord of ...
```

and `title_seq` columns, and we can see that we've successfully linked each movie to its sequel.

Merging a table to itself with left join

```
original_sequels = sequels.merge(sequels, left_on='sequel', right_on='id',
                                 how='left', suffixes=('_org','_seq'))
print(original_sequels.head())
```

```
   id_org  title_org      sequel_org  id_seq  title_seq      sequel_seq
0 19995    Avatar        NaN       NaN      NaN        NaN
1  862    Toy Story     863       863  Toy Story 2  10193
2  863    Toy Story 2  10193      10193  Toy Story 3      NaN
3  597    Titanic        NaN       NaN      NaN        NaN
4 24428  The Avengers  NaN       NaN      NaN        NaN
```

When to merge at table to itself

Common situations:

- Hierarchical relationships
- Sequential relationships
- Graph data

Table with an index

	<code>id</code>	<code>title</code>	<code>popularity</code>	<code>release_date</code>
0	257	Oliver Twist	20.415572	2005-09-23
1	14290	Better Luck ...	3.877036	2002-01-12
2	38365	Grown Ups	38.864027	2010-06-24
3	9672	Infamous	3.680896	2006-11-16
4	12819	Alpha and Omega	12.300789	2010-09-17

	<code>id</code>	<code>title</code>	<code>popularity</code>	<code>release_date</code>
257	Oliver Twist	20.415572	2005-09-23	
14290	Better Luck ...	3.877036	2002-01-12	
38365	Grown Ups	38.864027	2010-06-24	
9672	Infamous	3.680896	2006-11-16	
12819	Alpha and Omega	12.300789	2010-09-17	

Setting an index

```
movies = pd.read_csv('tmdb_movies.csv', index_col=['id'])
print(movies.head())
```

	<code>id</code>	<code>title</code>	<code>popularity</code>	<code>release_date</code>
257	Oliver Twist	20.415572	2005-09-23	
14290	Better Luck ...	3.877036	2002-01-12	
38365	Grown Ups	38.864027	2010-06-24	
9672	Infamous	3.680896	2006-11-16	
12819	Alpha and Omega	12.300789	2010-09-17	

Merging on index

```
movies_taglines = movies.merge(taglines, on='id', how='left')
print(movies_taglines.head())
```

	title	popularity	release_date	tagline
id				
257	Oliver Twist	20.415572	2005-09-23	NaN
14290	Better Luck ...	3.877036	2002-01-12	Never undere...
38365	Grown Ups	38.864027	2010-06-24	Boys will be...
9672	Infamous	3.680896	2006-11-16	There's more...
12819	Alpha and Omega	12.300789	2010-09-17	A Pawsome 3D...

MultilIndex datasets

```
samuel = pd.read_csv('samuel.csv',
                      index_col=['movie_id',
                                 'cast_id'])
print(samuel.head())
```

		name
movie_id	cast_id	
184	3	Samuel L. Jackson
319	13	Samuel L. Jackson
326	2	Samuel L. Jackson
329	138	Samuel L. Jackson
393	21	Samuel L. Jackson

```
casts = pd.read_csv('casts.csv',
                     index_col=['movie_id',
                                'cast_id'])
print(casts.head())
```

		character
movie_id	cast_id	
5	22	Jezebel
	23	Diana
	24	Athena
	25	Elsbeth
	26	Eva

MultilIndex merge

```
samuel_casts = samuel.merge(casts, on=['movie_id','cast_id'])
print(samuel_casts.head())
print(samuel_casts.shape)
```

		name		character
movie_id	cast_id			
184	3	Samuel L. Jackson	Ordell Robbie	
319	13	Samuel L. Jackson	Big Don	
326	2	Samuel L. Jackson	Neville Flynn	
329	138	Samuel L. Jackson	Arnold	
393	21	Samuel L. Jackson	Rufus	
(67, 2)				

Index merge with left_on and right_on

	title	popularity	release_date
id			
257	Oliver Twist	20.415572	2005-09-23
14290	Better Luck ...	3.877036	2002-01-12
38365	Grown Ups	38.864027	2010-06-24
9672	Infamous	3.680896	2006-11-16

	genre
movie_id	
5	Crime
5	Comedy
11	Science Fiction
11	Action

Index merge with left_on and right_on

```
movies_genres = movies.merge(movie_to_genres, left_on='id', left_index=True,
                             right_on='movie_id', right_index=True)
print(movies_genres.head())
```

	id	title	popularity	release_date	genre
5	5	Four Rooms	22.876230	1995-12-09	Crime
5	5	Four Rooms	22.876230	1995-12-09	Comedy
11	11	Star Wars	126.393695	1977-05-25	Science Fiction
11	11	Star Wars	126.393695	1977-05-25	Action
11	11	Star Wars	126.393695	1977-05-25	Adventure

Additionally, since we are merging on indexes, we need to set left_index and right_index to True.

Advanced Merging and Concatenating

Pandas doesn't provide direct support for filtering joins, but we will learn how to replicate them.

Mutating versus filtering joins

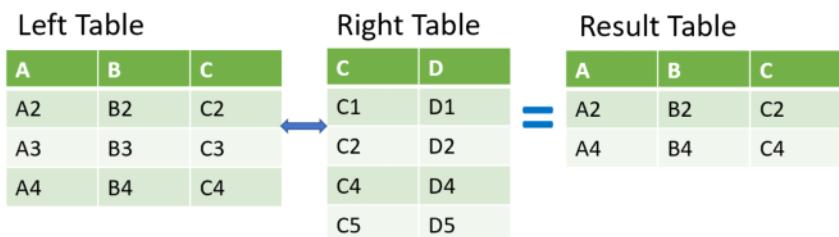
Mutating joins:

- 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

What is a semi-join?



Semi-joins

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

Musical dataset



¹ Photo by Vlad Baacian from Pexels

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

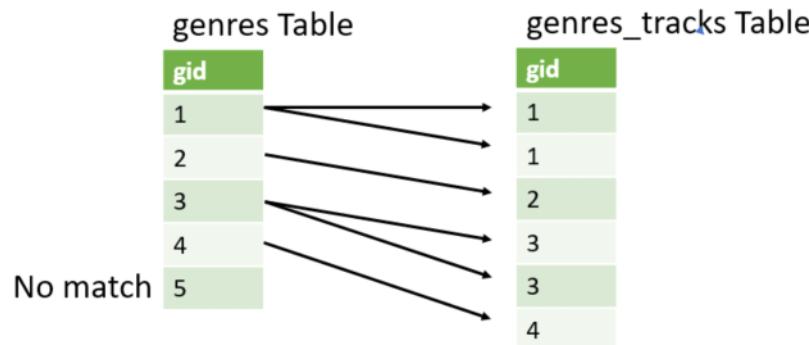
```
genres_tracks = genres.merge(top_tracks, on='gid')
print(genres_tracks.head())
```

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

Since this is an inner join, the returned 'gid' column holds only values where both tables matched.

Step 2 - semi-join

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



Step 2 - semi-join

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

```
0      True
1      True
2      True
3      True
4     False
Name: gid, dtype: bool
```

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
```

What is an anti-join?

Left Table			Right Table		Result Table		
A	B	C	C	D	A	B	C
A2	B2	C2	C1	D1	=	A3	B3
A3	B3	C3	C2	D2	=	C3	
A4	B4	C4	C4	D4			
			C5	D5			

Anti-join:

- Returns the left table, excluding the intersection
- Returns only columns from the left table and *not* the right

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      v_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
```

With indicator set to True, the merge method adds a column called "`_merge`" to the output.

This column tells the source of each row.

For example, the first four rows found a match in both tables, whereas the last can only be found in the left table.

Step 2 - anti-join

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

Next, we use the "loc" accessor and "_merge" column to select the rows that only appeared in the left table and return only the "gid" column from the genres_tracks table.

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

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
```

Our output shows those genres not in the tracks table.

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

So far in this course, we have only discussed how to merge two tables, which mainly grows them horizontally.

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

Basic concatenation

- 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

Basic 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

Ignoring the index

```
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

Setting labels to original tables

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

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

Concatenate tables with different column names

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

Concatenate tables with different column names

```
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

Concatenate tables with different column names

```
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

Using append method

.append()

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

Append these tables

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

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

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

Append is a DataFrame method therefore, we list the "inv_jan" table first then call the method.

We add the other tables as a list, and set the ignore_index and sort arguments similar to the concat method.

Append the tables

```
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

In our output, we see null values for the billing country, except for February.

Additionally, the index is adjusted as expected.

Let's check our data

Possible merging issue:

A	B	C	C	D
A1	B1	C1	C1	D1
A2	B2	C2	C1	D2
A3	B3	C3	C1	D3

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

Both the merge and concat methods have special features that allow us to verify the structure of our data.

Let's check our data

Possible merging issue:

A	B	C	C	D
A1	B1	C1	C1	D1
A2	B2	C2	C1	D2
A3	B3	C3	C1	D3

- 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

The validate and verify integrity

arguments of the merge and concat methods respectively will allow us to verify the data.

Validating merges

```
.merge(validate=None) :
```

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

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

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 concatenations

```
.concat(verify_integrity=False) :
```

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

It has the argument `verify_integrity`, which by default is `False`.

However, if set to `True`, it will check if there are duplicate values in the index and raise an error if there are.

It will only check the index values and not the columns.

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)
```

	cid	invoice_date	total
iid			
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

[Merging Ordered and Time-Series Data](#)

merge_ordered()

Left Table			Right Table		Result Table			
A	B	C	C	D	A	B	C	D
A3	B3	C3	C4	D4	A1	B1	C1	D1
A2	B2	C2	C2	D2	A2	B2	C2	D2
A1	B1	C1	C1	D1	A3	B3	C3	
							C4	D4

The merge_ordered method will allow us to merge the left and right tables shown here.

The results are similar to the standard merge method with an outer join, but here that the results are sorted.

The sorted results make this a useful method for ordered or 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)

However, how you call each of the methods is different.

Stock data

Table Name: appl

date	close
0 2007-02-01	12.087143
1 2007-03-01	13.272857
2 2007-04-01	14.257143
3 2007-05-01	17.312857
4 2007-06-01	17.434286

Table Name: mcd

date	close
0 2007-01-01	44.349998
1 2007-02-01	43.689999
2 2007-03-01	45.049999
3 2007-04-01	48.279999
4 2007-05-01	50.549999

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

Forward fill

Before		After	
A	B	A	B
A1	B1	A1	B1
A2		A2	B1
A3	B3	A3	B3
A4		A4	B3
A5	B5	A5	B5

Fills missing with previous value

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

When to use merge_ordered()?

- Ordered data / time series
- Filling in missing values

Using merge_asof()

Left Table		Right Table		Result Table		
B	C	C	D	B	C	D
B2	1	1	D1	B2	1	D1
B3	5	2	D2	B3	5	D3
B4	10	3	D3	B4	10	D7
		6	D6			
		7	D7			

- 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.

Using merge_asof()

Left Table		Right Table		Result Table		
B	C	C	D	B	C	D
B2	1	1	D1	B2	1	D1
B3	5	2	D2	B3	5	D3
B4	10	3	D3	B4	10	D7
		6	D6			
		7	D7			

Datasets

Table Name: `visa`

	date_time	close
0	2017-11-17 16:00:00	110.32
1	2017-11-17 17:00:00	110.24
2	2017-11-17 18:00:00	110.065
3	2017-11-17 19:00:00	110.04
4	2017-11-17 20:00:00	110.0
5	2017-11-17 21:00:00	109.9966
6	2017-11-17 22:00:00	109.82

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

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

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

This will change the behavior of the method to select the first row in the

right table whose "on" key column is greater than or equal to the left's key column.

The default value for the direction argument is "backward".

When to use merge_asof()

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

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
```

Updated dataset

This table is `stocks_long`

	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
7	2019-09-01	nike	93.919998
8	2019-10-01	nike	89.550003
9	2019-11-01	nike	93.489998

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

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

	first	last	variable	value
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Dataset in wide format

This table is called `social_fin`

	financial	company	2019	2018	2017	2016
0	total_revenue	twitter	3459329	3042359	2443299	2529619
1	gross_profit	twitter	2322288	2077362	1582057	1597379
2	net_income	twitter	1465659	1205596	-108063	-456873
3	total_revenue	facebook	70697000	55838000	40653000	27638000
4	gross_profit	facebook	57927000	46483000	35199000	23849000
5	net_income	facebook	18485000	22112000	15934000	10217000

The first input argument to the method is `id_vars`.

These are columns to be used as identifier variables.

We can also think of them as columns in our original dataset that we do not want to change.

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

Melting with value_vars

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

This time, let's use the argument `value_vars` with the `melt()` method.

This argument will allow us to control which columns are unpivoted.

	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

Our output now only has data for the years 2018 and 2017.

Additionally, the order of the `value_var` was kept.

The output starts with 2018, then moves to 2017.

Finally, notice that the column with the years is now named `variable`, and our values column is named `value`.

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