Pandas 5 (GroupBy)

Think back to the cars example in the previous lecture, where we had 4-cylinder, 6-cylinder, and 8-cylinder cars, alongside their weights, their speeds, their horsepower, etc.

In [1]:
```
from pandas import Series, DataFrame
import pandas as pd
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
cars = pd.read_csv('Pandas_4_data/cars.csv').dropna()
cars[:5]
```

Out[2]:

	type	mpg	cyl	disp	hp	wt	speed	wt.1
0	AMC Ambassador Brougham	13.0	8	360.0	175.0	3821	11.0	73
1	AMC Ambassador DPL	15.0	8	390.0	190.0	3850	8.5	70
2	AMC Ambassador SST	17.0	8	304.0	150.0	3672	11.5	72
3	AMC Concord DL 6	20.2	6	232.0	90.0	3265	18.2	79
4	AMC Concord DL	18.1	6	258.0	120.0	3410	15.1	78

If we wanted the average speed of all 4-cylinder, 6-cylinder, and 8-cylinder cars, how would we do it?

In [3]:
```
cylinder_speed_pivot = pd.pivot_table(cars,
                                      index='cyl',
                                      columns=None,
                                      values='speed',
                                      aggfunc='mean')
cylinder_speed_pivot
```

Out[3]:

	speed
cyl	
3	13.250000
4	16.581910
5	18.633333
6	16.254217
8	12.955340

The pivot table is really just a convenient shorthand for a basic operation called a *groupby*. This is such an important concept that we'll go into detail into the basics of:

- groupby,
- group aggregations, and
- group transformations.

This lecture borrows a lot from this other lecture.

Main Idea

Conceptually, the basic steps are as follows:

- **split** the data into several *groups*
  - e.g., split all cars into the "4-cylinder", "6-cylinder", "8-cylinder" groups
- **apply** some function independently to each of the groups
  - e.g., average the speed of all cars in each group
- and then **merge** the results for all groups together
  - e.g., get back one Series with the average speed indexed by number of cylinders.

Hadoop MapReduce is based on the same concept.

GroupBy

Let us just look at the number of cylinders and speeds of all cars.

In [4]:
```
cylinder_speeds = cars[['cyl', 'speed']]
cylinder_speeds[:5]
```

Out[4]:

	cyl	speed
0	8	11.0
1	8	8.5
2	8	11.5
3	6	18.2
4	6	15.1

Step 1: Split

In [5]:
```
grps = cylinder_speeds.groupby('cyl')
print(len(grps))
```

```
5
```

What does 'grps' look like?

In [6]:
```
print(type(grps))
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

This is not a DataFrame, or any object we have seen so far. However, in many respects, it acts as a *dictionary*.

In [7]:
```
# Let us iterate over the keys and values of this groupby object
for key, group in grps:
```

```
    print('group name =', key)
    print(group[:5])
```

```
group name = 3
     cyl  speed
240    3   13.5
248    3   13.5
249    3   13.5
250    3   12.5
group name = 4
     cyl  speed
7      4   20.1
28     4   15.0
29     4   14.0
30     4   15.0
31     4   14.5
group name = 5
     cyl  speed
33     5   15.9
34     5   19.9
253    5   20.1
group name = 6
    cyl  speed
3     6   18.2
4     6   15.1
6     6   17.2
8     6   15.0
9     6   13.0
group name = 8
     cyl  speed
0      8   11.0
1      8    8.5
2      8   11.5
17     8   15.5
18     8   12.5
```

Summary: We can think of the result of groupby() as a dictionary

- whose keys are the groups on which we split the data, and
- whose values are the DataFrame parts corresponding to those groups

Step 2: Apply

How do we find the average speed for each group of cars?

In [8]:

```
# Let us find the mean speed in each group
group_names = []
group_means = []
for name, group in grps:
    mean = group['speed'].mean()
    print('mean for', name, 'is', mean)

    group_names.append(name)
    group_means.append(mean)
```

```
mean for 3 is 13.25
mean for 4 is 16.581909547738693
mean for 5 is 18.633333333333333
```

```
mean for 6 is 16.25421686746988
mean for 8 is 12.955339805825243
```

Step 3: Merge

How do we merge the means for all the groups into one Series?

In [9]:
```
# Merge the means
Series(group_means, index=group_names)
```

Out[9]:
```
3    13.250000
4    16.581910
5    18.633333
6    16.254217
8    12.955340
dtype: float64
```

Groupby:

All of these steps can be compressed into one.

In [10]:
```
cylinder_speeds[:5]
```

Out[10]:

	cyl	speed
0	8	11.0
1	8	8.5
2	8	11.5
3	6	18.2
4	6	15.1

In [11]:
```
cylinder_speeds.groupby('cyl').mean()
```

Out[11]:

	speed
cyl	
3	13.250000
4	16.581910
5	18.633333
6	16.254217
8	12.955340

While we grouped together all cars with the same number of cylinders, perhaps different manufacturers have different technology? This leads to the following question.

Find the mean speed of cars grouped by number of cylinders *and* manufacturer.

How do we do this?

1. Extract the manufacturer from the car's name.

2. Run groupby() on both cylinders and manufacturer

Step 1: Extract the manufacturer

In [12]:
```
cars['type'][:5]
```

Out[12]:
```
0    AMC Ambassador Brougham
1      AMC Ambassador DPL
2      AMC Ambassador SST
3      AMC Concord DL 6
4      AMC Concord DL
Name: type, dtype: object
```

How do we extract the manufacturer from the car name series?

In [13]:
```
def get_manufacturer(s):
    """Extract manufacturer from car name"""
    # Split the string by spaces, and pick the first
    words_in_car_name = s.split()
    manufacturer = words_in_car_name[0]
    return manufacturer

# Test it
get_manufacturer('AMC Ambassador Brougham')
```

Out[13]:
```
'AMC'
```

Let us create a new column in cars for the car manufacturer.

In [14]:
```
cars['manufacturer'] = cars['type'].map(get_manufacturer)
cars[['type', 'manufacturer']][:10]
```

Out[14]:

	type	manufacturer
0	AMC Ambassador Brougham	AMC
1	AMC Ambassador DPL	AMC
2	AMC Ambassador SST	AMC
3	AMC Concord DL 6	AMC
4	AMC Concord DL	AMC
6	AMC Concord	AMC
7	AMC Concord	AMC
8	AMC Gremlin	AMC
9	AMC Gremlin	AMC
10	AMC Gremlin	AMC

Step 2: Run the groupby()

In [15]:
```
mean_group_speed = cars.groupby(['cyl', 'manufacturer'])[['speed']].mean()
mean_group_speed[:8]
```

Out[15]:

		speed
cyl	manufacturer	
3	Maxda	13.500000

		speed
cyl	manufacturer	
4	Mazda	13.166667
	AMC	17.550000
	Audi	15.160000
	BMW	12.650000
	Buick	15.733333
	Chevrolet	18.253333
	Chevy	19.400000

In [16]:
```
# How do you access a hierarchical index? As a _tuple_
mean_group_speed.loc[(3, 'Mazda')]
```

Out[16]:
```
speed    13.166667
Name: (3, Mazda), dtype: float64
```

Let us run through this groupby() in detail.

> cars.groupby(['cyl', 'manufacturer'])[['speed']].mean()

- **cars**: Obvious; run on the cars DataFrame
- **groupby(['cyl', 'manufacturer'])**: Group together all cars with the same number of cylinders *and* manufacturer
- **[['speed']]**: Recall that for each of the groups, we get a DataFrame of cars in that group. By specifying [[column1, column2, ...]], we *select* those columns of the group DataFrames. Here, we select just the speed column.
- **mean()**: Apply the mean() function on the selected columns (here, 'speed') of each group's DataFrame.

The result of this groupby() is a DataFrame with a *hierarchical* index with two levels. The first level of the index is number of cylinders, and the second level is the manufacturer.

Group Aggregations

In addition to mean(), there are several standard functions such as count(), min(), max(), etc. But we aren't restricted to these; we can define our own functions as well.

Find the range of speeds for these car groups.

How will we do this?

1. Define a special **aggregation** function for the range of speeds.

2. Apply this function to the groups found by groupby().

Step 1: Define aggregation function.

What should the aggregation function look like? Consider several examples:

- mean(): It takes a Series of speeds, and returns one number (the mean speed).
- max(): It takes a Series of speeds, and returns one number (the max speed).

So that is the pattern we must follow in designing our own aggregation function.

We need to define a function

- speed_range(): Take a Series of speeds, and return one number (the max - min speed)

In [17]:
```
def speed_range(s):
    """Given a series of speeds (called s), get the range"""
    return s.max() - s.min()
```

Step 2: Apply this function to the groups in groupby()

In [18]:
```
# Apply our aggregator using the agg() function
cars.groupby(['cyl', 'manufacturer'])[['speed']].agg(speed_range)[:10]
```

Out[18]:

		speed
cyl	**manufacturer**	
3	**Maxda**	0.0
	Mazda	1.0
4	**AMC**	5.1
	Audi	2.5
	BMW	0.3
	Buick	1.6
	Chevrolet	6.7
	Chevy	0.0
	Chrysler	0.0
	Datsun	4.9

In the groupby() calls we just saw, we just called one aggregation function on each group's DataFrame

- the mean(), or
- our own speed_range().

What if we want multiple aggregate statistics?

Find number of cars, their average speed, and speed range for the various car groups.

In [19]:
```
cars.groupby(['cyl', 'manufacturer'])[['speed']].agg(['mean', 'count', speed_range])[:10]
```

Out[19]:

		speed		
		mean	count	speed_range
cyl	manufacturer			
3	Maxda	13.500000	1	0.0
	Mazda	13.166667	3	1.0
4	AMC	17.550000	2	5.1
	Audi	15.160000	5	2.5
	BMW	12.650000	2	0.3
	Buick	15.733333	3	1.6
	Chevrolet	18.253333	15	6.7
	Chevy	19.400000	1	0.0
	Chrysler	14.500000	1	0.0
	Datsun	16.885000	20	4.9

Summary so far

```
df.groupby([groupby-columns])[[aggregation-columns]].agg([aggregation-functions])
```

- Split the data according to the list of groupby-columns
- Each split corresponds to a piece of the full DataFrame
- Select just the aggregation-columns from these DataFrame pieces
- Apply each of the aggregation functions to these columns

Final result:

- A DataFrame
- The groupby columns form the index (possibly a hierarchical index)
- A column for each (aggregation-column, aggregation-function)

Let us analyze an example movies dataset.

From the README:

> These files contain 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

In [25]:
```
!head Pandas_5_data/users.dat
```

```
1::F::1::10::48067
2::M::56::16::70072
3::M::25::15::55117
4::M::45::7::02460
5::M::25::20::55455
6::F::50::9::55117
```

```
7::M::35::1::06810
8::M::25::12::11413
9::M::25::17::61614
10::F::35::1::95370
```

How do we read in the data?

- Fields are separated by '::' instead of ',' as in a CSV.
- We can *still* use pd.read_csv()

In [26]:
```
users = pd.read_csv('Pandas_5_data/users.dat',
                    sep='::',      # use this field separator
                    header=None,   # do not use the first line as a header
                    names=['user_id', 'gender', 'age', 'occupation', 'zip'])
users[:5]
```

```
C:\Users\deepay\Miniconda\lib\site-packages\ipykernel_launcher.py:4: ParserWarni
ng: Falling back to the 'python' engine because the 'c' engine does not support
regex separators (separators > 1 char and different from '\s+' are interpreted a
s regex); you can avoid this warning by specifying engine='python'.
  after removing the cwd from sys.path.
```

Out[26]:

	user_id	gender	age	occupation	zip
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

In [27]:
```
# Similarly, we read in the other data
movies = pd.read_csv('Pandas_5_data/movies.dat',
                     sep='::',
                     header=None,
                     names=['movie_id', 'title', 'genres'],
                     engine='python')
ratings = pd.read_csv('Pandas_5_data/ratings.dat',
                      sep='::',
                      header=None,
                      names=['user_id', 'movie_id', 'rating', 'timestamp'],
                      engine='python')

print(movies[:3])
print
print(ratings[:3])
```

```
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
```

Find the top-rated movies with at least 1000 ratings

We need to combine information from the ratings and movies DataFrames.

It will be good to just build a merged DataFrame and work with that.

In [28]:
```
print('The ratings DataFrame has', ratings.columns.values)
print('The movies DataFrame has', movies.columns.values)
print('The users DataFrame has', users.columns.values)
```

```
The ratings DataFrame has ['user_id' 'movie_id' 'rating' 'timestamp']
The movies DataFrame has ['movie_id' 'title' 'genres']
The users DataFrame has ['user_id' 'gender' 'age' 'occupation' 'zip']
```

On what fields will we do the merge?

In [29]:
```
df_movie_ratings = ratings.merge(movies, left_on='movie_id', right_on='movie_id')
df = df_movie_ratings.merge(users, left_on='user_id', right_on='user_id')

df.columns.values
```

Out[29]:
```
array(['user_id', 'movie_id', 'rating', 'timestamp', 'title', 'genres',
       'gender', 'age', 'occupation', 'zip'], dtype=object)
```

What does the DataFrame 'df' contain?

It has, for each rating given by a user to a movie, all details of the user *and* the movie, alongside the details of the rating itself.

Returning to our question:

Find the top-rated movies with at least 1000 ratings

How do we do it?

1. Group the merged DataFrame (df) by movie
2. Find the average rating and the number of ratings for each movie
   - Each movie being each group
3. Mask on movies with >= 1000 ratings, and sort by average rating

Steps 1 and 2 can be done together in one call to groupby()!

Steps 1 and 2: Find the average rating and number of ratings for each movie.

What is the correct groupby() statement?

Call groupby() on what data?

> df.groupby()

Group by what fields?

- movie_id
- also title (we'll want it later...) Each movie_id corresponds to one movie title anyway.

> df.groupby(['movie_id', 'title'])

Steps 1 and 2: Find the average rating and number of ratings for each movie.

Which field of each movie group do we care about?

- rating

> df.groupby(['movie_id', 'title'])[['rating']]

How do we want to aggregate the ratings for each group?

- count (so we can mask out movies with at least 1000 ratings)
- mean (so we can sort these movies later)

> df.groupby(['movie_id', 'title'])[['rating']].agg(['mean', 'count'])

In [30]:
```
mean_df = df.groupby(['movie_id', 'title'])[['rating']].agg(['mean', 'count'])
mean_df[:5]
```

Out[30]:

		rating	
		mean	count
movie_id	title		
1	Toy Story (1995)	4.146846	2077
2	Jumanji (1995)	3.201141	701
3	Grumpier Old Men (1995)	3.016736	478
4	Waiting to Exhale (1995)	2.729412	170
5	Father of the Bride Part II (1995)	3.006757	296

The columns are *hierarchical*:

- First level is just 'rating'
- Second level contains 'mean' and 'count'

We will only to access the mean and count of the rating, so let us just select that.

In [31]:
```
mean_ratings = mean_df['rating']
mean_ratings[:5]
```

Out[31]:

		mean	count
movie_id	title		
1	Toy Story (1995)	4.146846	2077
2	Jumanji (1995)	3.201141	701

		mean	count
movie_id	title		
3	Grumpier Old Men (1995)	3.016736	478
4	Waiting to Exhale (1995)	2.729412	170
5	Father of the Bride Part II (1995)	3.006757	296

An alternative that accomplishes this same thing in one step is the following:

In [32]:
```
mean_ratings = df.groupby(['movie_id', 'title'])['rating'].agg(['mean', 'count'])
mean_ratings[:5]
```

Out[32]:

		mean	count
movie_id	title		
1	Toy Story (1995)	4.146846	2077
2	Jumanji (1995)	3.201141	701
3	Grumpier Old Men (1995)	3.016736	478
4	Waiting to Exhale (1995)	2.729412	170
5	Father of the Bride Part II (1995)	3.006757	296

The difference is between saying

```
df.groupby(['movie_id', 'title'])['rating']
```

versus

```
df.groupby(['movie_id', 'title'])[['rating']]
```

In the former, Pandas understands we only want to work with the 'rating' column, so it doesn't create hierarchical columns.

Step 3: Mask on movies with >= 1000 ratings, and sort by average rating.

In [33]:
```
# movie_ids with at least 100 ratings
at_least_1000 = mean_ratings[mean_ratings['count'] >= 1000]

# sort these by the mean
at_least_1000.sort_values(by='mean', ascending=False)[:10]
```

Out[33]:

		mean	count
movie_id	title		
318	Shawshank Redemption, The (1994)	4.554558	2227
858	Godfather, The (1972)	4.524966	2223
50	Usual Suspects, The (1995)	4.517106	1783
527	Schindler's List (1993)	4.510417	2304
1198	Raiders of the Lost Ark (1981)	4.477725	2514
904	Rear Window (1954)	4.476190	1050
260	Star Wars: Episode IV - A New Hope (1977)	4.453694	2991
750	Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1963)	4.449890	1367

		mean	count
movie_id	title		
912	Casablanca (1942)	4.412822	1669
2762	Sixth Sense, The (1999)	4.406263	2459

Who rates more, men or women?

In [34]:
```
df['gender'].value_counts()
```

Out[34]:
```
M    753769
F    246440
Name: gender, dtype: int64
```

Roughly 75% or the ratings are by men.

We could also have done:

In [35]:
```
df.groupby('gender')['gender'].count()
```

Out[35]:
```
gender
F    246440
M    753769
Name: gender, dtype: int64
```

value_counts() is actually a groupby() function!

Find movies liked by both men and women

This is a bit vague, but let's formalize it this way:

- Find movies that rated by at least 750 men and 250 women,
- get their mean ratings among men and women, and
- sort them by the minimum of these two mean ratings.

How do we do the groupby() now?

- Need to group by movie,
- **and** gender

In [36]:
```
means_by_gender = df.groupby(['movie_id', 'title', 'gender'])['rating'].agg(['mean', 'count'])
means_by_gender[:5]
```

Out[36]:

			mean	count
movie_id	title	gender		
1	Toy Story (1995)	F	4.187817	591
		M	4.130552	1486
2	Jumanji (1995)	F	3.278409	176
		M	3.175238	525
3	Grumpier Old Men (1995)	F	3.073529	136

To find movies rated by 750 men and 250 women each, we need to have

- one row per movie,
- with the 'count' for men and the 'count' for women on the same row,
- so that we can easily create a mask.

How do we get counts for men and women on the same row?

In [37]:
```
means_by_gender_unstacked = means_by_gender.unstack('gender')
means_by_gender_unstacked[:5]
```

Out[37]:

		mean		count	
	gender	F	M	F	M
movie_id	title				
1	Toy Story (1995)	4.187817	4.130552	591.0	1486.0
2	Jumanji (1995)	3.278409	3.175238	176.0	525.0
3	Grumpier Old Men (1995)	3.073529	2.994152	136.0	342.0
4	Waiting to Exhale (1995)	2.976471	2.482353	85.0	85.0
5	Father of the Bride Part II (1995)	3.212963	2.888298	108.0	188.0

Now, we can do the **filtering** of movies with enough ratings by both men and women.

In [38]:
```
# Do the filtering
men_mask = (means_by_gender_unstacked['count']['M'] >= 750)
women_mask = (means_by_gender_unstacked['count']['F'] >= 250)
means_by_gender_filtered = means_by_gender_unstacked[men_mask & women_mask]

means_by_gender_filtered[:5]
```

Out[38]:

		mean		count	
	gender	F	M	F	M
movie_id	title				
1	Toy Story (1995)	4.187817	4.130552	591.0	1486.0
21	Get Shorty (1995)	3.597902	3.630841	286.0	1070.0
32	Twelve Monkeys (1995)	3.845324	3.968370	278.0	1233.0
34	Babe (1995)	3.953368	3.860922	579.0	1172.0
39	Clueless (1995)	3.827004	3.514640	474.0	888.0

Now we need the minimum of men's and women's ratings for each movie.

How do we calculate the minimum?

- We want to select the 'mean' columns, and
- apply the min() function to the two ratings

In [39]:
```
men_women_ratings = means_by_gender_filtered['mean'].copy()
men_women_ratings.T.iloc[0:5, 0:5]
```

Out[39]:

movie_id	1	21	32	34	39
title	Toy Story (1995)	Get Shorty (1995)	Twelve Monkeys (1995)	Babe (1995)	Clueless (1995)
gender					
F	4.187817	3.597902	3.845324	3.953368	3.827004
M	4.130552	3.630841	3.968370	3.860922	3.514640

In [40]:

```
min_ratings = men_women_ratings.T.apply(min)
min_ratings[:5]
```

Out[40]:

```
movie_id  title
1         Toy Story (1995)          4.130552
21        Get Shorty (1995)         3.597902
32        Twelve Monkeys (1995)     3.845324
34        Babe (1995)               3.860922
39        Clueless (1995)           3.514640
dtype: float64
```

In [41]:

```
# Let's add this as an extra column
men_women_ratings['min'] = min_ratings
men_women_ratings[:5]
```

Out[41]:

gender		F	M	min
movie_id	title			
1	Toy Story (1995)	4.187817	4.130552	4.130552
21	Get Shorty (1995)	3.597902	3.630841	3.597902
32	Twelve Monkeys (1995)	3.845324	3.968370	3.845324
34	Babe (1995)	3.953368	3.860922	3.860922
39	Clueless (1995)	3.827004	3.514640	3.514640

Finally, we want to sort by the min rating.

In [42]:

```
# Sort by the min ratings
men_women_ratings.sort_values(by='min', ascending=False)[:10]
```

Out[42]:

gender		F	M	min
movie_id	title			
318	Shawshank Redemption, The (1994)	4.539075	4.560625	4.539075
50	Usual Suspects, The (1995)	4.513317	4.518248	4.513317
527	Schindler's List (1993)	4.562602	4.491415	4.491415
904	Rear Window (1954)	4.484536	4.472991	4.472991
2762	Sixth Sense, The (1999)	4.477410	4.379944	4.379944
908	North by Northwest (1959)	4.364458	4.390641	4.364458
1198	Raiders of the Lost Ark (1981)	4.332168	4.520597	4.332168
923	Citizen Kane (1941)	4.332143	4.407895	4.332143
858	Godfather, The (1972)	4.314700	4.583333	4.314700
1193	One Flew Over the Cuckoo's Nest (1975)	4.310811	4.418423	4.310811

On what movies do men and women disagree the most?

We already have the mean ratings for each movie, grouped by gender.

In [43]:
```
men_women_ratings.T.iloc[0:5, 0:5]
```

Out[43]:

movie_id	1	21	32	34	39
title	Toy Story (1995)	Get Shorty (1995)	Twelve Monkeys (1995)	Babe (1995)	Clueless (1995)
gender					
F	4.187817	3.597902	3.845324	3.953368	3.827004
M	4.130552	3.630841	3.968370	3.860922	3.514640
min	4.130552	3.597902	3.845324	3.860922	3.514640

Step 1: We need to design a function that

- given each column (i.e., values for F, M, and min),
- returns the absolute difference between F and M

Step 2: Then, we need to apply this function to all columns. How?

```
the apply() method
```

Step 1: A function that computes differences between F and M ratings.

The input to the function will be a column, expressed as a Series.

What is in the Series?

In [44]:
```
men_women_ratings.T.index.values
```

Out[44]:
```
array(['F', 'M', 'min'], dtype=object)
```

In [45]:
```
# write a function for the difference between men's and women's ratings
def rating_diff(gender_series):
    """Given an input Series of ratings for a movie,
       return the absolute difference."""

    # The input gender_seris has:
    #   women's rating in index 'F'
    #   men's rating in index 'M'
    #   min rating in index 'min'
    return abs(gender_series['F'] - gender_series['M'])

# Test
first_movie = men_women_ratings.index.values[0]
test_series = men_women_ratings.T[first_movie]  # Select first movie column
print(test_series)
rating_diff(test_series)
```

```
gender
F      4.187817
M      4.130552
min    4.130552
Name: (1, Toy Story (1995)), dtype: float64
```

Out[45]:
```
0.057265441924971405
```

Step 2: Apply this function to the entire DataFrame.

In [46]:
```
rating_differences = men_women_ratings.T.apply(rating_diff)
rating_differences[:5]
```

Out[46]:
```
movie_id  title
1         Toy Story (1995)          0.057265
21        Get Shorty (1995)         0.032939
32        Twelve Monkeys (1995)     0.123046
34        Babe (1995)               0.092446
39        Clueless (1995)           0.312365
dtype: float64
```

In [47]:
```
# Add this as an extra column
men_women_ratings['diff'] = rating_differences
men_women_ratings[:5]
```

Out[47]:

	gender	F	M	min	diff
movie_id	title				
1	Toy Story (1995)	4.187817	4.130552	4.130552	0.057265
21	Get Shorty (1995)	3.597902	3.630841	3.597902	0.032939
32	Twelve Monkeys (1995)	3.845324	3.968370	3.845324	0.123046
34	Babe (1995)	3.953368	3.860922	3.860922	0.092446
39	Clueless (1995)	3.827004	3.514640	3.514640	0.312365

In [48]:
```
# Alternatively, and more simply:
# men_women_ratings['diff'] = abs(men_women_ratings['M'] - men_women_ratings['F'])
```

In [49]:
```
# Sort by diff to get the movies
men_women_ratings.sort_values(by='diff', ascending=False)[:10]
```

Out[49]:

	gender	F	M	min	diff
movie_id	title				
3421	Animal House (1978)	3.628906	4.167192	3.628906	0.538286
2657	Rocky Horror Picture Show, The (1975)	3.673016	3.160131	3.160131	0.512885
2700	South Park: Bigger, Longer and Uncut (1999)	3.422481	3.846686	3.422481	0.424206
2791	Airplane! (1980)	3.656566	4.064419	3.656566	0.407854
1221	Godfather: Part II, The (1974)	4.040936	4.437778	4.040936	0.396842
1200	Aliens (1986)	3.802083	4.186684	3.802083	0.384601
1641	Full Monty, The (1997)	4.113456	3.760976	3.760976	0.352481
3418	Thelma & Louise (1991)	3.916268	3.581582	3.581582	0.334686
589	Terminator 2: Judgment Day (1991)	3.785088	4.115367	3.785088	0.330279
1214	Alien (1979)	3.888252	4.216119	3.888252	0.327867