Pandas - 3

Content

- **Apply()**
- **Grouping**
  - `groupby()`
- **Group based Aggregates**
- **Group based Filtering**
- **Group based Apply**

Importing Data

Let's first import our data and prepare it as we did in the last lecture

```
import pandas as pd
import numpy as np

!gdown 1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
!gdown 1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm

movies = pd.read_csv('movies.csv', index_col=0)
directors = pd.read_csv('directors.csv',index_col=0)
```

```
Downloading...
From: https://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
To: /content/movies.csv
100% 112k/112k [00:00<00:00, 77.0MB/s]
Downloading...
From: https://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
To: /content/directors.csv
100% 65.4k/65.4k [00:00<00:00, 73.5MB/s]
```

IMDB Movie Business Use-case (Continued...)

In the previous lecture (Pandas-2), we concluded that:

- Movie dataset contains info about movies, release, popularity, ratings and the director ID
- Director dataset contains detailed info about the director

In this lecture we begin to perform some operations on the data

Merging the director and movie data

Now, how can we know the details about the Director of a particular movie?

We will have to merge these datasets

So on which column we should merge the dfs ?

We will use the **ID columns** (representing unique director) in both the datasets

If you observe,

=> `director_id` of movies are taken from `id` of directors dataframe

Thus we can merge our dataframes based on these two columns as **keys**

Before that, lets first check number of unique director values in our `movies` data

How do we get the number of unique directors in `movies` ?

```
movies['director_id'].nunique()
```

```
199
```

Recall,

we had learnt about nunique earlier

Similarly for unique diretors in `directors` df

```
directors['id'].nunique()
```

```
    2349
```

Summary:

- Movies Dataset: 1465 rows, but only 199 unique directors
- Directors Dataset: 2349 unique directors (= no of rows)

What can we infer from this?

=> Directors in `movies` is a subset of directors in `directors`

Now, how can we check if all `director_id` values are present in `id`?

```
movies['director_id'].isin(directors['id'])
```

```
    0       True
    1       True
    2       True
    3       True
    5       True
            ...
    4736    True
    4743    True
    4748    True
    4749    True
    4768    True
    Name: director_id, Length: 1465, dtype: bool
```

The `isin()` method checks if the Dataframe column contains the specified value(s).

How is `isin` different from Python `in`?

- `in` works for **one element** at a time
- `isin` does this for **all the values** in the column

If you notice,

- This is like a boolean "mask"
- It returns a df similar to the original df
- For rows with values of `director_id` present in `id` it returns True, else False

How can we check if there is any False here?

```
np.all(movies['director_id'].isin(directors['id']))
```

```
    True
```

Lets finally merge our dataframes

Do we need to keep **all the rows for movies**?

YES

Do we need to keep **all the rows of directors**?

NO

- only the ones for which we have a corresponding row in movies

So which `join` type do you think we should apply here ?

We can use LEFT JOIN

```
data = movies.merge(directors, how='left', left_on='director_id',right_on='id')
data
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	director_id	year	month	day	direct
0	43597	237000000	150	2787965087	Avatar	7.2	11800	4762	2009	Dec	Thursday	James
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	4763	2007	May	Saturday	Gore
2	43599	245000000	107	880674609	Spectre	6.3	4466	4764	2015	Oct	Monday	Sar
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	4765	2012	Jul	Monday	Cl
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	4767	2007	May	Tuesday	S
...	...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	4809	1978	May	Monday	Martin
1461	48370	27000	19	3151130	Clerks	7.4	755	5369	1994	Sep	Tuesday	Ke
1462	48375	0	7	0	Rampage	6.0	131	5148	2009	Aug	Friday	
1463	48376	0	3	0	Slacker	6.4	77	5535	1990	Jul	Friday	Richarc
1464	48395	220000	14	2040920	El Mariachi	6.6	238	5097	1992	Sep	Friday	F

1465 rows × 14 columns

Notice, two stranger id columns `id_x` and `id_y`.

What do you think these newly created cols are?

Since the columns with name `id` is present in both the df

- `id_x` represents **id values from movie df**
- `id_y` represents **id values from directors df**

Do you think any column is redundant here and can be dropped?

- `id_y` is redundant as it is same as `director_id`
- But we dont require `director_id` further

So we can simply drop these features

```
data.drop(['director_id','id_y'],axis=1,inplace=True)
data.head()
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gender
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	James Cameron	Male
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbinski	Male
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sam Mendes	Male
					The Dark						Christopher	

Apply

- Apply a function along an axis of the DataFrame or Series

Task: we want to convert our `Gender` column data to numerical format

Basically,

- 0 for Male
- 1 for Female

How can we encode the column?

Let's first write a function to do it for a single value

```
def encode(data):
  if data == "Male":
    return 0
  else:
    return 1
```

Now how can we apply this function to the whole column?

```
data['gender'] = data['gender'].apply(encode)
data
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	da
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursda
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturda
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monda
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monda
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesda
...	...	...	...	...	...	...	...	...	...	.
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monda
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesda
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Frida
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Frida
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Frida

1465 rows × 12 columns

> Notice how this is similar to using `vectorization` in Numpy

We thus can use `apply` to use a function throughout a column

Applying a function using `apply` on multiple columns

finding sum of revenue and budget per movie?

```
data[['revenue', 'budget']].apply(np.sum)
```

```
revenue    209866997305
budget      70353617179
dtype: int64
```

We can pass **multiple cols by packing them** within `[]`

But there's a mistake here. We wanted our results per movie (per row)

But, we are getting the sum of the columns

Applying function with `apply` on rows using the `axis`

```
data[['revenue', 'budget']].apply(np.sum, axis=1)
```

```
0       3024965087
1       1261000000
```

```
2       1125674609
3       1334939099
4       1148871626
            ...
1460        321952
1461       3178130
1462             0
1463             0
1464       2260920
Length: 1465, dtype: int64
```

Every row of `revenue` was added to same row of `budget`

What does this `axis` mean in apply ?

- **axis = 0** => it will apply to **each column**
- **axis = 1** => **each row**
- By default axis = 0

=> `apply()` can be applied on any dataframe along any particular axis

Similarly, how can I find profit per movie (revenue-budget)?

```
def prof(x): # We define a function to calculate profit
  return x['revenue']-x['budget']
data['profit'] = data[['revenue', 'budget']].apply(prof, axis = 1)
data
```

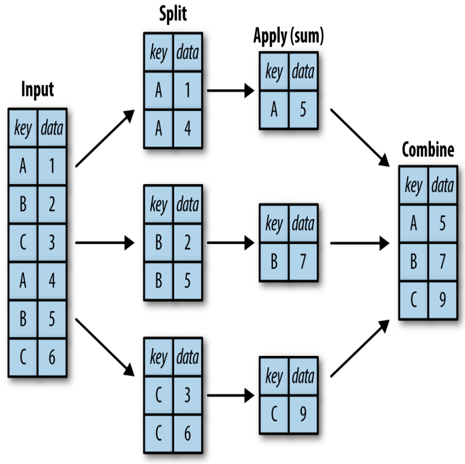
	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gend
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	James Cameron	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbinski	
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sam Mendes	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Christopher Nolan	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sam Raimi	
...	...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday	Martin Scorsese	
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday	Kevin Smith	
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday	Uwe Boll	
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday	Richard Linklater	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday	Robert Rodriguez	

1465 rows × 13 columns

Grouping

What is Grouping ?

Simply it could be understood through the terms - Split, apply, combine



1. **Split**: **Breaking up and grouping** a DataFrame depending on the value of the specified key.

Group based Aggregates

- We use different aggregate functions like mean, sum, min, max, count etc. on columns while grouping.

Grouping data director-wise

```
data.groupby('director_name')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f71cad3ead0>
```

Notice,

- It's a **DataFrameGroupBy type object**
- **NOT a DataFrame** type object

Number of groups our data is divided into?

```
data.groupby('director_name').ngroups
```

```
199
```

Based on this grouping, we can find which keys belong to which group?

```
data.groupby('director_name').groups
```

```
{'Adam McKay': [176, 323, 366, 505, 839, 916], 'Adam Shankman': [265, 300, 350, 404, 458, 843, 999, 1231], 'Alejandro
González Iñárritu': [106, 749, 1015, 1034, 1077, 1405], 'Alex Proyas': [95, 159, 514, 671, 873], 'Alexander Payne':
[793, 1006, 1101, 1211, 1281], 'Andrew Adamson': [11, 43, 328, 501, 947], 'Andrew Niccol': [533, 603, 701, 722, 1439],
'Andrzej Bartkowiak': [349, 549, 754, 911, 924], 'Andy Fickman': [517, 681, 909, 926, 973, 1023], 'Andy Tennant': [314,
320, 464, 593, 676, 885], 'Ang Lee': [99, 134, 748, 840, 1089, 1110, 1132, 1184], 'Anne Fletcher': [610, 650, 736, 789,
1206], 'Antoine Fuqua': [310, 338, 424, 467, 576, 808, 818, 1105], 'Atom Egoyan': [946, 1128, 1164, 1194, 1347, 1416],
'Barry Levinson': [313, 319, 471, 594, 878, 898, 1013, 1037, 1082, 1143, 1185, 1345, 1378], 'Barry Sonnenfeld': [13,
48, 90, 205, 591, 778, 783], 'Ben Stiller': [209, 212, 547, 562, 850], 'Bill Condon': [102, 307, 902, 1233, 1381],
'Bobby Farrelly': [352, 356, 481, 498, 624, 630, 654, 806, 928, 972, 1111], 'Brad Anderson': [1163, 1197, 1350, 1419,
1430], 'Brett Ratner': [24, 39, 188, 207, 238, 292, 405, 456, 920], 'Brian De Palma': [228, 255, 318, 439, 747, 905,
919, 1088, 1232, 1261, 1317, 1354], 'Brian Helgeland': [512, 607, 623, 742, 933], 'Brian Levant': [418, 449, 568, 761,
860, 1003], 'Brian Robbins': [416, 441, 669, 962, 988, 1115], 'Bryan Singer': [6, 32, 33, 44, 122, 216, 297, 1326],
'Cameron Crowe': [335, 434, 488, 503, 513, 698], 'Catherine Hardwicke': [602, 695, 724, 937, 1406, 1412], 'Chris
Columbus': [117, 167, 204, 218, 229, 509, 656, 897, 996, 1086, 1129], 'Chris Weitz': [17, 500, 794, 869, 1202, 1267],
'Christopher Nolan': [3, 45, 58, 59, 74, 565, 641, 1341], 'Chuck Russell': [177, 410, 657, 1069, 1097, 1339], 'Clint
Eastwood': [369, 426, 447, 482, 490, 520, 530, 535, 645, 727, 731, 786, 787, 899, 974, 986, 1167, 1190, 1313], 'Curtis
Hanson': [494, 579, 606, 711, 733, 1057, 1310], 'Danny Boyle': [527, 668, 1083, 1085, 1126, 1168, 1287, 1385], 'Darren
Aronofsky': [113, 751, 1187, 1328, 1363, 1458], 'Darren Lynn Bousman': [1241, 1243, 1283, 1338, 1440], 'David Ayer':
[50, 273, 741, 1024, 1146, 1407], 'David Cronenberg': [541, 767, 994, 1055, 1254, 1268, 1334], 'David Fincher': [62,
213, 253, 383, 398, 478, 522, 555, 618, 785], 'David Gordon Green': [543, 862, 884, 927, 1376, 1418, 1432, 1459],
'David Koepp': [443, 644, 735, 1041, 1209], 'David Lynch': [583, 1161, 1264, 1340, 1456], 'David O. Russell': [422,
556, 609, 896, 982, 989, 1229, 1304], 'David R. Ellis': [582, 634, 756, 888, 934], 'David Zucker': [569, 619, 965,
1052, 1175], 'Dennis Dugan': [217, 260, 267, 293, 303, 718, 780, 977, 1247], 'Donald Petrie': [427, 507, 570, 649, 858,
894, 1106, 1331], 'Doug Liman': [52, 148, 251, 399, 544, 1318, 1451], 'Edward Zwick': [92, 182, 346, 566, 791, 819,
825], 'F. Gary Gray': [308, 402, 491, 523, 697, 833, 1272, 1380], 'Francis Ford Coppola': [487, 559, 622, 646, 772,
1076, 1155, 1253, 1312], 'Francis Lawrence': [63, 72, 109, 120, 679], 'Frank Coraci': [157, 249, 275, 451, 577, 599,
963], 'Frank Oz': [193, 355, 473, 580, 712, 813, 987], 'Garry Marshall': [329, 496, 528, 571, 784, 893, 1029, 1169],
'Gary Fleder': [518, 667, 689, 867, 981, 1165], 'Gary Winick': [258, 797, 798, 804, 1454], 'Gavin O'Connor': [820, 841,
939, 953, 1444], 'George A. Romero': [250, 1066, 1096, 1278, 1367, 1396], 'George Clooney': [343, 450, 831, 966, 1302],
'George Miller': [78, 103, 233, 287, 1250, 1403, 1450], 'Gore Verbinski': [1, 8, 9, 107, 119, 633, 1040], 'Guillermo
del Toro': [35, 252, 419, 486, 1118], 'Gus Van Sant': [595, 1018, 1027, 1159, 1240, 1311, 1398], 'Guy Ritchie': [124,
215, 312, 1093, 1225, 1269, 1420], 'Harold Ramis': [425, 431, 558, 586, 788, 1137, 1166, 1325], 'Ivan Reitman': [274,
643, 816, 883, 910, 935, 1134, 1242], 'James Cameron': [0, 19, 170, 173, 344, 1100, 1320], 'James Ivory': [1125, 1152,
1180, 1291, 1293, 1390, 1397], 'James Mangold': [140, 141, 557, 560, 829, 845, 958, 1145], 'James Wan': [30, 617, 1002,
1047, 1337, 1417, 1424], 'Jan de Bont': [155, 224, 231, 270, 781], 'Jason Friedberg': [812, 1010, 1012, 1014, 1036],
'Jason Reitman': [792, 1092, 1213, 1295, 1299], 'Jaume Collet-Serra': [516, 540, 640, 725, 1011, 1189], 'Jay Roach':
[195, 359, 389, 397, 461, 703, 859, 1072], 'Jean-Pierre Jeunet': [423, 485, 605, 664, 765], 'Joe Dante': [284, 525,
638, 1226, 1298, 1428], 'Joe Wright': [85, 432, 553, 803, 814, 855], 'Joel Coen': [428, 670, 691, 707, 721, 889, 906,
980, 1157, 1238, 1305], 'Joel Schumacher': [128, 184, 348, 484, 572, 614, 652, 764, 876, 886, 1108, 1230, 1280], 'John
Carpenter': [537, 663, 686, 861, 938, 1028, 1080, 1102, 1329, 1371], 'John Glen': [601, 642, 801, 847, 864], 'John
Landis': [524, 868, 1276, 1384, 1435], 'John Madden': [457, 882, 1020, 1249, 1257], 'John McTiernan': [127, 214, 244,
351, 534, 563, 648, 782, 838, 1074], 'John Singleton': [294, 489, 732, 796, 1120, 1173, 1316], 'John Whitesell': [499,
632, 763, 1119, 1148], 'John Woo': [131, 142, 264, 371, 420, 675, 1182], 'Jon Favreau': [46, 54, 55, 382, 759, 1346],
'Jon M. Chu': [100, 225, 810, 1099, 1186], 'Jon Turteltaub': [64, 180, 372, 480, 760, 846, 1171], 'Jonathan Demme':
[277, 493, 1000, 1123, 1215], 'Jonathan Liebesman': [81, 143, 339, 1117, 1301], 'Judd Apatow': [321, 710, 717, 865,
881], 'Justin Lin': [38, 123, 246, 1437, 1447], 'Kenneth Branagh': [80, 197, 421, 879, 1094, 1277, 1288], 'Kenny
Ortega': [412, 852, 1228, 1315, 1365], 'Kevin Reynolds': [53, 502, 639, 1019, 1059], ...}
```

Now what if we want to extract data of a particular group from this list?

```
data.groupby('director_name').get_group('Alexander Payne')
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gende
793	45163	30000000	19	105834556	About Schmidt	6.7	362	2002	Dec	Friday	Alexander Payne	
1006	45699	20000000	40	177243185	The Descendants	6.7	934	2011	Sep	Friday	Alexander Payne	
1101	46004	16000000	23	109502303	Sideways	6.9	478	2004	Oct	Friday	Alexander Payne	
1211	46446	12000000	29	17654912	Nebraska	7.4	636	2013	Sep	Saturday	Alexander Payne	
1281	46813	0	13	0	Election	6.7	270	1999	Apr	Friday	Alexander Payne	

extending this to finding an aggregate metric of the data

How can we find the count of movies by each director?

```
data.groupby('director_name')['title'].count()
```

```
director_name
Adam McKay                     6
Adam Shankman                  8
Alejandro González Iñárritu    6
Alex Proyas                    5
Alexander Payne                5
                              ..
Wes Craven                    10
Wolfgang Petersen              7
Woody Allen                   18
Zack Snyder                    7
Zhang Yimou                    6
Name: title, Length: 199, dtype: int64
```

Finding multiple aggregations of any feature

Finding the very first year and the latest year a director released a movie i.e basically the `min` and `max` of `year` column, grouped by director

```
data.groupby(['director_name'])["year"].aggregate(['min', 'max'])
# note: can also use .agg instead of .aggregate (both are same)
```

	min	max
director_name		
Adam McKay	2004	2015
Adam Shankman	2001	2012
Alejandro González Iñárritu	2000	2015
Alex Proyas	1994	2016
Alexander Payne	1999	2013
...	...	...
Wes Craven	1984	2011
Wolfgang Petersen	1981	2006
Woody Allen	1977	2013
Zack Snyder	2004	2016
Zhang Yimou	2002	2014

199 rows × 2 columns

Group based Filtering

- Group based filtering allows us to filter rows from each group by using conditional statements on each group rather than the whole dataframe.

finding the details of the movies by high budget directors

Lets assume,

- high budget director -> any director with **atleast one movie with budget >100M**

1. We can get the highest budget movie data of every director

```
data_dir_budget = data.groupby("director_name")["budget"].max().reset_index()
data_dir_budget.head()
```

	director_name	budget
0	Adam McKay	100000000
1	Adam Shankman	80000000
2	Alejandro González Iñárritu	135000000
3	Alex Proyas	140000000
4	Alexander Payne	30000000

2. we can **filter** out the director names with **max budget >100M**

```
names = data_dir_budget.loc[data_dir_budget["budget"] >= 100, "director_name"]
```

3. Finally, we can filter out the details of the movies by these directors

```
data.loc[data['director_name'].isin(names)]
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gend
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	James Cameron	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbinski	
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sam Mendes	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Christopher Nolan	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sam Raimi	
...	...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday	Martin Scorsese	
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday	Kevin Smith	
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday	Uwe Boll	
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday	Richard Linklater	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday	Robert Rodriguez	

1465 rows × 13 columns

Filtering groups in a single go using Lambda Function

```
data.groupby('director_name').filter(lambda x: x["budget"].max() >= 100)
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gend
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	James Cameron	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbinski	
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sam Mendes	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Christopher Nolan	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sam Raimi	
...	...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday	Martin Scorsese	
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday	Kevin Smith	
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday	Uwe Boll	
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday	Richard Linklater	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday	Robert Rodriguez	

1465 rows × 13 columns

Notice what's happening here?

- We first group data by director and then use `groupby().filter` function
- **Groups are filtered if they do not satisfy the boolean criterion** specified by function
- This is called **Group Based Filtering**

NOTE

We are filtering the **groups** here and **not the rows**

==> The result is **not a groupby object** but **regular pandas DataFrame** with the **filtered groups eliminated**

Group based Apply

- applying a function on grouped objects

Filtering risky movies?

Let's assume, we call a movi risky if,

- its budget is higher than the average revenue of its director

We can subtract the average `revenue` of a director from `budget` col, for each director

```
def func(x):
  # a boolean returning function for whether the movie is risky or not
  x["risky"] = x["budget"] - x["revenue"].mean() >= 0
  return x

data_risky = data.groupby("director_name", group_keys=False).apply(func)

# setting group_keys=True, keeps the group key in the returned dataset (will be default in future version of pandas)
# keep it False if want the normal behaviour

data_risky
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	gend
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009	Dec	Thursday	James Cameron	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Saturday	Gore Verbinski	
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015	Oct	Monday	Sam Mendes	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012	Jul	Monday	Christopher Nolan	
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007	May	Tuesday	Sam Raimi	
...	...	...	...	...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	1978	May	Monday	Martin Scorsese	
1461	48370	27000	19	3151130	Clerks	7.4	755	1994	Sep	Tuesday	Kevin Smith	
1462	48375	0	7	0	Rampage	6.0	131	2009	Aug	Friday	Uwe Boll	
1463	48376	0	3	0	Slacker	6.4	77	1990	Jul	Friday	Richard Linklater	
1464	48395	220000	14	2040920	El Mariachi	6.6	238	1992	Sep	Friday	Robert Rodriguez	

1465 rows × 14 columns

What did we do here?

- Defined a custom function
- Grouped data acc to `director_name`
- Subtracted mean of `budget` from `revenue`
- Used apply with the custom function on the grouped data

Lets see if there are any risky movies

```
data_risky.loc[data_risky["risky"]]
```

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	day	director_name	ge
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	2008	Oct	Thursday	Marc Forster	
12	43614	380000000	135	1045713802	Pirates of the Caribbean: On Stranger Tides	6.4	4948	2011	May	Saturday	Rob Marshall	
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	2010	May	Wednesday	Ridley Scott	
20	43624	209000000	64	303025485	Battleship	5.5	2114	2012	Apr	Wednesday	Peter Berg	
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	2006	May	Wednesday	Brett Ratner	