Content

- Introduction to IMDB use case
  - Merging `movies` & `directors` datasets
  - IMDB data exploration (Post-read)
- `apply()`
- `groupby()`
  - Group based Aggregation
  - Group based Filtering
  - Group based Apply

IMDB Movies Data

- Imagine you are working as a Data Scientist for an analytics firm.
- Your task is to analyse some **movie trends** for a client.
- **IMDB** has an online database of information related to movies.

Here we have two CSV files -

- `movies.csv`
- `directors.csv`

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
movies = pd.read_csv('movies.csv')
movies.head()
```

Out[2]:

	Unnamed: 0	id	budget	popularity	revenue	title	vote_average	vote_c
0	0	43597	237000000	150	2787965087	Avatar	7.2	1
1	1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4
2	2	43599	245000000	107	880674609	Spectre	6.3	4
3	3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	
4	5	43602	258000000	115	890871626	Spider-Man 3	5.9	3

So what kind of questions can we ask from this dataset?

- **Top 10 most popular movies**, using `popularity`.
- Find the **highest rated movies**, using `vote_average`.

- We can find number of **movies released per year**.
- Find **highest budget movies in a year** using both `budget` and `year` .

But can we ask more interesting/deeper questions?

- Do you think we can find the **most productive directors**?
- Which **directors produce high budget films**?
- **Highest and lowest rated movies for every month** in a particular year?

Notice that there's a column **Unnamed: 0** which represents nothing but the index of a row.

How to get rid of this `Unnamed: 0` col?

In [3]:

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

Out[3]:

	id	budget	popularity	revenue	title	vote_average	vote_count	direct
0	43597	237000000	150	2787965087	Avatar	7.2	11800	
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	
2	43599	245000000	107	880674609	Spectre	6.3	4466	
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	
5	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	

`index_col=0` explicitly states to treat the first column as the index.

The default value is `index_col=None`

In [4]:

```
movies.shape
```

Out[4]:

```
(1465, 11)
```

The `movies` dataframe contains 1465 rows and 11 columns.

In [5]:

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

Out[5]:

	director_name	id	gender
0	James Cameron	4762	Male
1	Gore Verbinski	4763	Male
2	Sam Mendes	4764	Male
3	Christopher Nolan	4765	Male
4	Andrew Stanton	4766	Male

In [6]:

```
directors.shape
```

Out[6]:

```
(2349, 3)
```

Merging `movies` & `directors` datasets

How can we know the details about the Director of a particular movie?

- We will have to merge these two datasets.

So on which column we should merge?

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

If you observe,

- `director_id` of movies are taken from `id` of directors.
- Thus, we can merge our dataframes based on these two columns as **keys**.

Before that, let's first check the number of unique directors in our `movies` dataset.

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

In [7]:

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

Out[7]:

```
199
```

Recall, we had learnt about `nunique()` earlier.

In [8]:

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

Out[8]:

```
2349
```

Summary:

- `movies` dataset: 1465 rows, but only 199 unique directors
- `directors` dataset: 2349 unique directors (equal to the no. of rows)

What can we infer from this?

- The directors in `movies` data is a subset of directors in `directors` data.

How can we check if all `director_id` values are present in `id`?

In [9]:
```
movies['director_id'].isin(directors['id'])
```

Out[9]:
```
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 a column contains the specified value(s).

How is `isin` different from Python's `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 dataframe similar to the original one.
- For rows with values of `director_id` present in `id`, it returns True, else False.

How can we check if there's any False here?

In [10]:
```
np.all(movies['director_id'].isin(directors['id']))
```

Out[10]:
```
True
```

Let's finally merge the two 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?

- `LEFT` Join

In [11]:
```
data = movies.merge(directors, how='left', left_on='director_id',right_on='
data
```

Out[11]:

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

1465 rows × 14 columns

Notice the two strange id columns - `id_x` and `id_y`.

What do you think these newly created columns are?

Since the columns with name `id` are present in both the dataframes,

- `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 the same as `director_id`
- But we don't require the `director_id` any further.

So we can simply drop these features -

In [12]:

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

Out[12]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year
0	43597	237000000	150	2787965087	Avatar	7.2	11800	2009
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	2007
2	43599	245000000	107	880674609	Spectre	6.3	4466	2015
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	2012
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	2007

Post-read

- IMDB data exploration

From here, we have the opportunity to delve into various aspects of the data, such as:

- Converting the revenue values into Millions of USD.
- Identifying the Top 5 most popular movies.

... and so on.

This task is for you to explore the data on your own.

`apply()`

- It is used apply a function along an axis of the DataFrame/Series.

Say we want to convert the data in `Gender` column into numerical format.

Basically,

- 0 for Male
- 1 for Female

How can we encode the values in the `Gender` column?

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

In [13]:

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

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

In [14]:
```
data['gender'] = data['gender'].apply(encode)
data
```

Out[14]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 12 columns

Notice how this is similar to using `Vectorization` in Numpy.

How to apply a function on multiple columns?

Let's say we want to find the sum of `revenue` and `budget` per movie?

In [15]:
```
data[['revenue', 'budget']].apply(np.sum)
```

Out[15]:
```
revenue    209866997305
budget      70353617179
dtype: int64
```

We can pass multiple columns by packing them within `[]`.

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

But we're getting the sum of the columns.

In [16]:
```
data[['revenue', 'budget']].apply(np.sum, axis=1)
```

Out[16]:

```
0       3024965087
1       1261000000
2       1125674609
3       1334939099
4       1148871626
           ...
1460        321952
1461       3178130
1462             0
1463             0
1464       2260920
Length: 1465, dtype: int64
```

By setting the `axis=1`, 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` → It will apply to **each row**

Note that **by default, axis=0**.

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

In [17]:

```python
# We define a function to calculate profit

def prof(x):
  return x['revenue']-x['budget']
data['profit'] = data[['revenue', 'budget']].apply(prof, axis = 1)
data
```

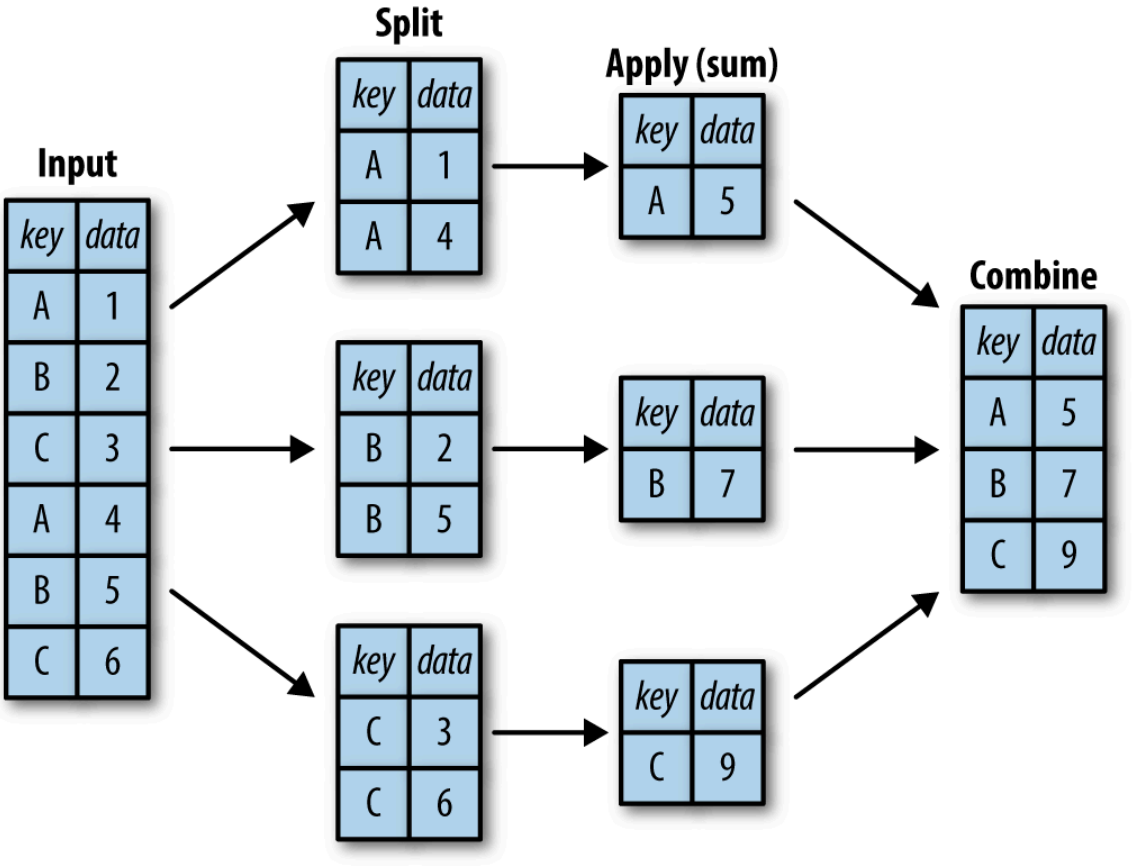
Out[17]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 13 columns

What is Grouping?

In simple terms, we could understood it through - Split, Apply, Combine



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

2. **Apply**: Computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

3. **Combine**: Merging the results of these operations into an output array.

In [18]:
```
data.groupby('director_name')
```

Out[18]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x148f7cb10>
```

Notice,

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

What's the number of groups our data is divided into?

In [19]:
```
data.groupby('director_name').ngroups
```

Out[19]:
```
199
```

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

In [20]:
```
data.groupby('director_name').groups
```

Out[20]:

```
{'Adam McKay': [176, 323, 366, 505, 839, 916], 'Adam Shankman': [265, 300,
350, 404, 458, 843, 999, 1231], 'Alejandro González Iñárritu': [106, 749, 1
015, 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': [3
49, 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, 8
40, 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': [1
3, 48, 90, 205, 591, 778, 783], 'Ben Stiller': [209, 212, 547, 562, 850],
'Bill Condon': [102, 307, 902, 1233, 1381], 'Bobby Farrelly': [352, 356, 48
1, 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, 9
20], 'Brian De Palma': [228, 255, 318, 439, 747, 905, 919, 1088, 1232, 126
1, 1317, 1354], 'Brian Helgeland': [512, 607, 623, 742, 933], 'Brian Levan
t': [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, 7
24, 937, 1406, 1412], 'Chris Columbus': [117, 167, 204, 218, 229, 509, 656,
897, 996, 1086, 1129], 'Chris Weitz': [17, 500, 794, 869, 1202, 1267], 'Chr
istopher Nolan': [3, 45, 58, 59, 74, 565, 641, 1341], 'Chuck Russell': [17
7, 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': [52
7, 668, 1083, 1085, 1126, 1168, 1287, 1385], 'Darren Aronofsky': [113, 751,
1187, 1328, 1363, 1458], 'Darren Lynn Bousman': [1241, 1243, 1283, 1338, 14
40], 'David Ayer': [50, 273, 741, 1024, 1146, 1407], 'David Cronenberg': [5
41, 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, 1
376, 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], 'D
avid 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], 'Edwa
rd Zwick': [92, 182, 346, 566, 791, 819, 825], 'F. Gary Gray': [308, 402, 4
91, 523, 697, 833, 1272, 1380], 'Francis Ford Coppola': [487, 559, 622, 64
6, 772, 1076, 1155, 1253, 1312], 'Francis Lawrence': [63, 72, 109, 120, 67
9], '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 Win
ick': [258, 797, 798, 804, 1454], 'Gavin O'Connor': [820, 841, 939, 953, 14
44], 'George A. Romero': [250, 1066, 1096, 1278, 1367, 1396], 'George Cloon
ey': [343, 450, 831, 966, 1302], 'George Miller': [78, 103, 233, 287, 1250,
1403, 1450], 'Gore Verbinski': [1, 8, 9, 107, 119, 633, 1040], 'Guillermo d
el Toro': [35, 252, 419, 486, 1118], 'Gus Van Sant': [595, 1018, 1027, 115
9, 1240, 1311, 1398], 'Guy Ritchie': [124, 215, 312, 1093, 1225, 1269, 142
0], 'Harold Ramis': [425, 431, 558, 586, 788, 1137, 1166, 1325], 'Ivan Reit
man': [274, 643, 816, 883, 910, 935, 1134, 1242], 'James Cameron': [0, 19,
170, 173, 344, 1100, 1320], 'James Ivory': [1125, 1152, 1180, 1291, 1293, 1
390, 1397], 'James Mangold': [140, 141, 557, 560, 829, 845, 958, 1145], 'Ja
mes Wan': [30, 617, 1002, 1047, 1337, 1417, 1424], 'Jan de Bont': [155, 22
4, 231, 270, 781], 'Jason Friedberg': [812, 1010, 1012, 1014, 1036], 'Jason
Reitman': [792, 1092, 1213, 1295, 1299], 'Jaume Collet-Serra': [516, 540, 6
40, 725, 1011, 1189], 'Jay Roach': [195, 359, 389, 397, 461, 703, 859, 107
2], 'Jean-Pierre Jeunet': [423, 485, 605, 664, 765], 'Joe Dante': [284, 52
5, 638, 1226, 1298, 1428], 'Joe Wright': [85, 432, 553, 803, 814, 855], 'Jo
el 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, 12
76, 1384, 1435], 'John Madden': [457, 882, 1020, 1249, 1257], 'John McTiern
an': [127, 214, 244, 351, 534, 563, 648, 782, 838, 1074], 'John Singleton':
[294, 489, 732, 796, 1120, 1173, 1316], 'John Whitesell': [499, 632, 763, 1
119, 1148], 'John Woo': [131, 142, 264, 371, 420, 675, 1182], 'Jon Favrea
u': [46, 54, 55, 382, 759, 1346], 'Jon M. Chu': [100, 225, 810, 1099, 118
6], 'Jon Turteltaub': [64, 180, 372, 480, 760, 846, 1171], 'Jonathan Demm
e': [277, 493, 1000, 1123, 1215], 'Jonathan Liebesman': [81, 143, 339, 111
7, 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, 6
39, 1019, 1059], ...}
```

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

```
In [21]: data.groupby('director_name').get_group('Alexander Payne')
```

Out[21]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
793	45163	30000000	19	105834556	About Schmidt	6.7	362	20
1006	45699	20000000	40	177243185	The Descendants	6.7	934	20
1101	46004	16000000	23	109502303	Sideways	6.9	478	20
1211	46446	12000000	29	17654912	Nebraska	7.4	636	20
1281	46813	0	13	0	Election	6.7	270	19

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

```
In [23]: data.groupby('director_name')['title'].count()
```

```
Out[23]:
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
```

How to find multiple aggregates of any feature?

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

```
In [24]: data.groupby(['director_name'])["year"].aggregate(['min', 'max'])
```

Out[24]:

	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

Note: We can also use `.agg` instead of `.aggregate` (both are same)

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.

How to find 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 **group** the data by director and use `max` of the budget as aggregator.

In [25]:

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

Out[25]:

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

1. We can **filter** out the director names with **max budget >100M**.

In [26]:

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

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

In [27]:
```
data.loc[data['director_name'].isin(names)]
```

Out[27]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 13 columns

Can we filter groups in a single go using Lambda functions? Yes!

In [28]:
```
data.groupby('director_name').filter(lambda x: x["budget"].max() >= 100)
```

Out[28]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

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 the 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

What if we want to do the transformation of a column using some column's agrregate

Let's say, we want to filter the risky movies whose budget was even higher than the average revenue of the director from his other movies.

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

In [29]:

```
def func(x):
  # returns whether a 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)
data_risky
```

Out[29]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	y
0	43597	237000000	150	2787965087	Avatar	7.2	11800	20
1	43598	300000000	139	961000000	Pirates of the Caribbean: At World's End	6.9	4500	20
2	43599	245000000	107	880674609	Spectre	6.3	4466	20
3	43600	250000000	112	1084939099	The Dark Knight Rises	7.6	9106	20
4	43602	258000000	115	890871626	Spider-Man 3	5.9	3576	20
...	...	...	...	...	...	...	...	
1460	48363	0	3	321952	The Last Waltz	7.9	64	19
1461	48370	27000	19	3151130	Clerks	7.4	755	19
1462	48375	0	7	0	Rampage	6.0	131	20
1463	48376	0	3	0	Slacker	6.4	77	19
1464	48395	220000	14	2040920	El Mariachi	6.6	238	19

1465 rows × 14 columns

Note:

- Setting `group_keys=True`, keeps the group key in the returned dataset.
- This will be default in future versions of Pandas.
- Keep it as False if want the normal behaviour.

What did we do here?

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

Now let's see if there are any risky movies -

```
In [30]: data_risky.loc[data_risky["risky"]]
```

Out[30]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	20
12	43614	380000000	135	1045713802	Pirates of the Caribbean: On Stranger Tides	6.4	4948	20
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	20
20	43624	209000000	64	303025485	Battleship	5.5	2114	20
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	20
...	...	...	...	...	...	...	...	
1347	47224	5000000	7	3263585	The Sweet Hereafter	6.8	103	19
1349	47229	5000000	3	4842699	90 Minutes in Heaven	5.4	40	20
1351	47233	5000000	6	0	Light Sleeper	5.7	15	19
1356	47263	15000000	10	0	Dying of the Light	4.5	118	20
1383	47453	3500000	4	0	In the Name of the King III	3.3	19	20

131 rows × 14 columns

```
In [32]: data_risky[data_risky["risky"]]
```

Out[32]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	ye
7	43608	200000000	107	586090727	Quantum of Solace	6.1	2965	20
12	43614	380000000	135	1045713802	Pirates of the Caribbean: On Stranger Tides	6.4	4948	20
15	43618	200000000	37	310669540	Robin Hood	6.2	1398	20
20	43624	209000000	64	303025485	Battleship	5.5	2114	20
24	43630	210000000	3	459359555	X-Men: The Last Stand	6.3	3525	20
...	...	...	...	...	...	...	...	
1347	47224	5000000	7	3263585	The Sweet Hereafter	6.8	103	19
1349	47229	5000000	3	4842699	90 Minutes in Heaven	5.4	40	20
1351	47233	5000000	6	0	Light Sleeper	5.7	15	19
1356	47263	15000000	10	0	Dying of the Light	4.5	118	20
1383	47453	3500000	4	0	In the Name of the King III	3.3	19	20

131 rows × 14 columns

In [ ]: