Content

Loading our IMDB data

In [ ]:
```
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)
data = movies.merge(directors, how='left', left_on='director_id',right_or
data.drop(['director_id','id_y'],axis=1,inplace=True)
```

```
Downloading...
From: https://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd (http
s://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd)
To: /content/movies.csv
100% 112k/112k [00:00<00:00, 81.1MB/s]
Downloading...
From: https://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm (http
s://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm)
To: /content/directors.csv
100% 65.4k/65.4k [00:00<00:00, 62.7MB/s]
```

Feature Exploration

Lets explore all the features in the merged dataset

In [ ]:
```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1465 entries, 0 to 1464
Data columns (total 12 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id_x           1465 non-null   int64
 1   budget         1465 non-null   int64
 2   popularity     1465 non-null   int64
 3   revenue        1465 non-null   int64
 4   title          1465 non-null   object
 5   vote_average   1465 non-null   float64
 6   vote_count     1465 non-null   int64
 7   year           1465 non-null   int64
 8   month          1465 non-null   object
 9   day            1465 non-null   object
 10  director_name  1465 non-null   object
 11  gender         1341 non-null   object
dtypes: float64(1), int64(6), object(5)
memory usage: 148.8+ KB
```

Looks like only `gender` column has missing values (will come later)

How can we describe these features to know more about their range of values?

In [ ]:
```
1 data.describe()
```

Out[3]:

	id_x	budget	popularity	revenue	vote_average	vote_count	
count	1465.000000	1.465000e+03	1465.000000	1.465000e+03	1465.000000	1465.000000	146
mean	45225.191126	4.802295e+07	30.855973	1.432539e+08	6.368191	1146.396587	200
std	1189.096396	4.935541e+07	34.845214	2.064918e+08	0.818033	1578.077438	
min	43597.000000	0.000000e+00	0.000000	0.000000e+00	3.000000	1.000000	197
25%	44236.000000	1.400000e+07	11.000000	1.738013e+07	5.900000	216.000000	199
50%	45022.000000	3.300000e+07	23.000000	7.578164e+07	6.400000	571.000000	200
75%	45990.000000	6.600000e+07	41.000000	1.792469e+08	6.900000	1387.000000	200
max	48395.000000	3.800000e+08	724.000000	2.787965e+09	8.300000	13752.000000	201

This gives us all **statistical properties** of the columns

If you notice, some columns such as "title", "month" are missing

How are these **missing columns different?**

They are of **object dtype**

Then how can we include object type in `df.describe()` ?

In [ ]:
```
1 data.describe(include=object)
```

Out[4]:

	title	month	day	director_name	gender
count	1465	1465	1465	1465	1341
unique	1465	12	7	199	2
top	Avatar	Dec	Friday	Steven Spielberg	Male
freq	1	193	654	26	1309

If you notice,

- The range of values in the `revenue` and `budget` seem to be very high
- Generally budget and revenue for Hollywood movies is in million dollars

How can we change the values of `revenue` and `budget` into million dollars USD?

In [ ]:
```
data['revenue'] = (data['revenue']/1000000).round(2)
data

```

Out[5]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	mont
0	43597	237000000	150	2787.97	Avatar	7.2	11800	2009	De
1	43598	300000000	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	Ma
2	43599	245000000	107	880.67	Spectre	6.3	4466	2015	Oc
3	43600	250000000	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Ju
4	43602	258000000	115	890.87	Spider-Man 3	5.9	3576	2007	Ma
...	...	...	...	...	...	...	...	...	
1460	48363	0	3	0.32	The Last Waltz	7.9	64	1978	Ma
1461	48370	27000	19	3.15	Clerks	7.4	755	1994	Se
1462	48375	0	7	0.00	Rampage	6.0	131	2009	Au
1463	48376	0	3	0.00	Slacker	6.4	77	1990	Ju
1464	48395	220000	14	2.04	El Mariachi	6.6	238	1992	Se

1465 rows × 12 columns

Similarly, we can do it for 'budget' as well

In [ ]:
```
data['budget']=(data['budget']/1000000).round(2)
data.head()
```

Out[6]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	
0	43597	237.0	150	2787.97	Avatar	7.2	11800	2009	Dec	Thu
1	43598	300.0	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May	Sat
2	43599	245.0	107	880.67	Spectre	6.3	4466	2015	Oct	Mo
3	43600	250.0	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul	Mo
4	43602	258.0	115	890.87	Spider-Man 3	5.9	3576	2007	May	Tue

Fetching queries from dataframe

Lets say we are interested in fetching all **highly rated movies**

- say movies with **ratings > 7**

How can we get movies with ratings > 7?

We can use the concept of `masking`

Lets first create a mask to filter such movies

- In SQL: `SELECT * FROM movies WHERE vote_average > 7`
- In pandas:

In [ ]:
```
data['vote_average'] > 7
```

Out[7]:
```
0        True
1       False
2       False
3        True
4       False
        ...
1460     True
1461     True
1462    False
1463    False
1464    False
Name: vote_average, Length: 1465, dtype: bool
```

But we still don't know the row values ... Only that which row satisfied the condtion

How do we get the row values from this mask?

In [ ]:
```
1 data.loc[data['vote_average'] > 7]
```

Out[8]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
14	43616	250.00	120	956.02	The Hobbit: The Battle of the Five Armies	7.1	4760	2014	Dec
16	43619	250.00	94	958.40	The Hobbit: The Desolation of Smaug	7.6	4524	2013	Dec
19	43622	200.00	100	1845.03	Titanic	7.5	7562	1997	Nov
...	...	...	...	...	...	...	...	...	...
1456	48321	0.01	20	7.00	Eraserhead	7.5	485	1977	Mar
1457	48323	0.00	5	0.00	The Mighty	7.1	51	1998	Oct
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep

301 rows × 12 columns

You can also perform the filtering without even using `loc`

In [ ]:
```
1 data[data['vote_average'] > 7]
```

Out[9]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
14	43616	250.00	120	956.02	The Hobbit: The Battle of the Five Armies	7.1	4760	2014	Dec
16	43619	250.00	94	958.40	The Hobbit: The Desolation of Smaug	7.6	4524	2013	Dec
19	43622	200.00	100	1845.03	Titanic	7.5	7562	1997	Nov
...	...	...	...	...	...	...	...	...	...
1456	48321	0.01	20	7.00	Eraserhead	7.5	485	1977	Mar
1457	48323	0.00	5	0.00	The Mighty	7.1	51	1998	Oct
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep

301 rows × 12 columns

But this is not recommended. Why ?

- It can create a confusion between implicit/explicit indexing used as discussed before
- `loc` is also much faster

Now, how can we return a subset of columns, say, only `title` and `director_name` ?

In [ ]:
```
1 data.loc[data['vote_average'] > 7, ['title','director_name']]
```

Out[10]:

	title	director_name
0	Avatar	James Cameron
3	The Dark Knight Rises	Christopher Nolan
14	The Hobbit: The Battle of the Five Armies	Peter Jackson
16	The Hobbit: The Desolation of Smaug	Peter Jackson
19	Titanic	James Cameron
...	...	...
1456	Eraserhead	David Lynch
1457	The Mighty	Peter Chelsom
1458	Pi	Darren Aronofsky
1460	The Last Waltz	Martin Scorsese
1461	Clerks	Kevin Smith

301 rows × 2 columns

So far we saw only single condition for filtering

What if we want to filter highly rated movies released after 2014?

Notice that two conditions are involved here

1. Movies need to be highly rated i.e.. > 7
2. They should be 2015 and onwards

We can **use AND operator b/w multiple conditions**

In [ ]:
```
1 data.loc[(data['vote_average'] > 7) & (data['year'] >= 2015)].head()
```

Out[11]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	
30	43641	190.0	102	1506.25	Furious 7	7.3	4176	2015	Apr	W
78	43724	150.0	434	378.86	Mad Max: Fury Road	7.2	9427	2015	May	W
106	43773	135.0	100	532.95	The Revenant	7.3	6396	2015	Dec	
162	43867	108.0	167	630.16	The Martian	7.6	7268	2015	Sep	W
312	44128	75.0	48	108.15	The Man from U.N.C.L.E.	7.1	2265	2015	Aug	

Recall how we apply **mutliple conditions in numpy ?**

Use **elementwise operator `&` or `|`**

Note:

- **we cannot use `and` `or` `or`** with dataframe
- **for multiple conditions**, we need to put each **separate condition within parenthesis `()`**

Similarly how can we find movies released on either Friday or Sunday?

In [ ]:
```
data.loc[(data['day'] == 'Friday') | (data['day'] == 'Saturday')].head()
```

Out[12]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
1	43598	300.0	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May
12	43614	380.0	135	1045.71	Pirates of the Caribbean: On Stranger Tides	6.4	4948	2011	May
22	43627	200.0	35	783.77	Spider-Man 2	6.7	4321	2004	Jun
25	43632	150.0	21	836.30	Transformers: Revenge of the Fallen	6.0	3138	2009	Jun
40	43656	200.0	45	769.65	2012	5.6	4903	2009	Oct

Thus we can do complex queries using both `&` and `|` operators

Now let's try to answer few more Questions from this data

How will you find Top 5 most popular movies?

We can simply sort our data based on values of column 'popularity'

In [ ]:
```
1 data.sort_values(['popularity'],ascending=False).head(5)
```

Out[13]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	
58	43692	165.0	724	675.12	Interstellar	8.1	10867	2014	Nov	V
78	43724	150.0	434	378.86	Mad Max: Fury Road	7.2	9427	2015	May	V
119	43796	140.0	271	655.01	Pirates of the Caribbean: The Curse of the Bla...	7.5	6985	2003	Jul	V
120	43797	125.0	206	752.10	The Hunger Games: Mockingjay - Part 1	6.6	5584	2014	Nov	
45	43662	185.0	187	1004.56	The Dark Knight	8.2	12002	2008	Jul	V

On applying this to a string column, it sorts the dataframe ***lexicographically**

In [ ]:
```
1 data.sort_values(['title'],ascending=False).head(5)
```

Out[14]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month	
436	44364	60.0	36	71.07	xXx: State of the Union	4.7	549	2005	Apr	W
330	44165	70.0	46	277.45	xXx	5.8	1424	2002	Aug	
994	45681	15.0	21	2.86	eXistenZ	6.7	475	1999	Apr	W
547	44594	50.0	37	55.97	Zoolander 2	4.7	797	2016	Feb	
850	45313	28.0	38	60.78	Zoolander	6.1	1337	2001	Sep	

Now, how will get list of movies directed by a particular director, say, 'Christopher Nolan'?

In [ ]:
```
data.loc[data['director_name'] == 'Christopher Nolan',['title']]
```

Out[15]:

	title
3	The Dark Knight Rises
45	The Dark Knight
58	Interstellar
59	Inception
74	Batman Begins
565	Insomnia
641	The Prestige
1341	Memento

Note:

- The string indicating "Christopher Nolan" could have been something else as well.
- The better way is to use string methods, we will discuss this later

Apply

Now suppose 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

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

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

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

Out[17]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May
...	...	...	...	...	...	...	...	...	...
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep
1462	48375	0.00	7	0.00	Rampage	6.0	131	2009	Aug
1463	48376	0.00	3	0.00	Slacker	6.4	77	1990	Jul
1464	48395	0.22	14	2.04	El Mariachi	6.6	238	1992	Sep

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

Can we **use apply on multiple columns**?

Say,

How to find sum of revenue and budget per movie?

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

Out[18]:
```
revenue    209867.04
budget      70353.62
dtype: float64
```

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

How can we use apply to work on individual rows?

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

Out[19]:
```
0       3024.97
1       1261.00
2       1125.67
3       1334.94
4       1148.87
         ...
1460       0.32
1461       3.18
1462       0.00
1463       0.00
1464       2.26
Length: 1465, dtype: float64
```

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

What does this `axis` mean in apply ?

- If **axis = 0**, it will apply to **each column**, if **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)?

In [ ]:
```
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
```

Out[20]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May
...	...	...	...	...	...	...	...	...	...
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May
1461	48370	0.03	19	3.15	Clerks	7.4	755	1994	Sep
1462	48375	0.00	7	0.00	Rampage	6.0	131	2009	Aug
1463	48376	0.00	3	0.00	Slacker	6.4	77	1990	Jul
1464	48395	0.22	14	2.04	El Mariachi	6.6	238	1992	Sep

1465 rows × 13 columns

Thus, we can access the columns by their names inside the functions too using apply

Grouping

How can we know the number of movies released by a particular director, say, Christopher Nolan?

In [ ]:
```
data.loc[data['director_name'] == 'Christopher Nolan',['title']].count()
```

Out[21]:
```
title    8
dtype: int64
```

What if we have to do find number of movies of each director?

We have value_counts() for this

In [ ]:
```
1 data["director_name"].value_counts()
```

Out[22]:
```
Steven Spielberg      26
Martin Scorsese       19
Clint Eastwood        19
Woody Allen           18
Ridley Scott          16
                      ..
Tim Hill               5
Jonathan Liebesman     5
Roman Polanski         5
Larry Charles          5
Nicole Holofcener      5
Name: director_name, Length: 199, dtype: int64
```

How does this exactly work?

We can assume pandas must have **grouped the rows internally** to find the count

But what if we need to find some **other metric** besides count?

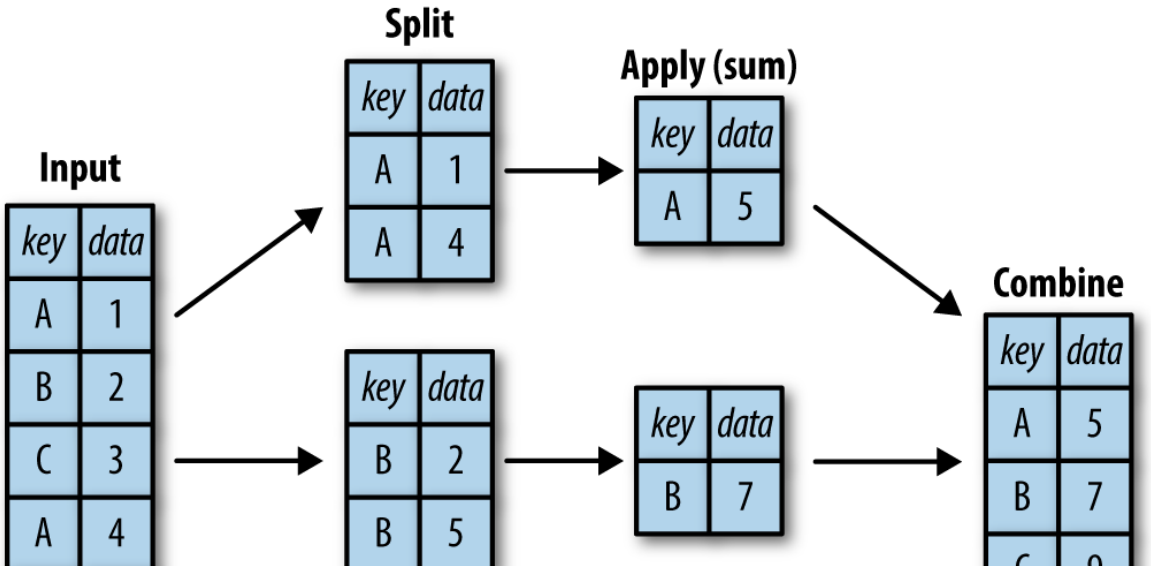
For example, **average popularity** of each director, or **max rating** among all movies by a director?

How can you find the average popularity of each director?

We will have to some group our rows director wise.

What is Grouping ?

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



Group based Aggregates

Now, how can we group our data director-wise?

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

Out[23]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f51da89b950>
```

Notice,

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

What is `groupby('director_name')` doing?

Grouping all rows in which **director_name** value is **same**

But it's returning an object, we would want to get information out of this object.

Let's look at few attributes of the same.

How can we know the number of groups our data is divided into?

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

Out[24]:
```
199
```

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

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

Out[25]:
{'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 Da

```
nte': [284, 525, 638, 1226, 1298, 1428], 'Joe Wright': [85, 432, 553, 803, 8
14, 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, 1
102, 1329, 1371], 'John Glen': [601, 642, 801, 847, 864], 'John Landis': [52
4, 868, 1276, 1384, 1435], 'John Madden': [457, 882, 1020, 1249, 1257], 'Joh
n McTiernan': [127, 214, 244, 351, 534, 563, 648, 782, 838, 1074], 'John Sin
gleton': [294, 489, 732, 796, 1120, 1173, 1316], 'John Whitesell': [499, 63
2, 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 Demm
e': [277, 493, 1000, 1123, 1215], 'Jonathan Liebesman': [81, 143, 339, 1117,
1301], 'Judd Apatow': [321, 710, 717, 865, 881], 'Justin Lin': [38, 123, 24
6, 1437, 1447], 'Kenneth Branagh': [80, 197, 421, 879, 1094, 1277, 1288], 'K
enny 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?

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

Out[26]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
793	45163	30.0	19	105.83	About Schmidt	6.7	362	2002	Dec
1006	45699	20.0	40	177.24	The Descendants	6.7	934	2011	Sep
1101	46004	16.0	23	109.50	Sideways	6.9	478	2004	Oct
1211	46446	12.0	29	17.65	Nebraska	7.4	636	2013	Sep
1281	46813	0.0	13	0.00	Election	6.7	270	1999	Apr

Great! We are able to extract the data from our DataFrameGroupBy object

But can we extend this to finding an aggregate metric of the data?

How can we find average popularity of each director?

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

Out[27]:

director_name	id_x	budget	popularity	revenue	vote_average	vote_count	
Adam McKay	44586.000000	56.916667	30.333333	143.180000	6.466667	1326.500000	20
Adam Shankman	44821.250000	48.375000	23.125000	109.196250	6.037500	623.875000	20
Alejandro González Iñárritu	45660.500000	33.333333	47.000000	146.331667	7.233333	2286.000000	20
Alex Proyas	44477.000000	70.400000	53.200000	154.912000	6.480000	1667.400000	20
Alexander Payne	46025.000000	15.600000	24.800000	82.044000	6.880000	536.000000	20
...	...	...	...	...	...	...	
Wes Craven	45503.300000	23.380000	22.300000	76.478000	5.950000	630.500000	19
Wolfgang Petersen	44511.285714	90.142857	35.857143	230.717143	6.571429	986.714286	19
Woody Allen	46083.777778	11.777778	17.722222	34.495000	6.672222	504.111111	2(
Zack Snyder	44086.857143	122.857143	71.857143	353.742857	6.485714	3501.857143	20
Zhang Yimou	45630.166667	20.833333	12.000000	60.926667	6.616667	254.333333	20

199 rows × 9 columns

This does give us the max value of the data, but for **all the features**

How can we **specify a single feature**, such as **popularity**, in this case?

In [ ]:
```
1 data.groupby('director_name')['popularity'].mean()
```

Out[28]:
```
director_name
Adam McKay                       30.333333
Adam Shankman                    23.125000
Alejandro González Iñárritu      47.000000
Alex Proyas                      53.200000
Alexander Payne                  24.800000
                                   ...
Wes Craven                       22.300000
Wolfgang Petersen                35.857143
Woody Allen                      17.722222
Zack Snyder                      71.857143
Zhang Yimou                      12.000000
Name: popularity, Length: 199, dtype: float64
```

Now say we want to know two aggregations for any feature.

For e.g., the very first year and the latest year a director released a movie

This is basically the `min` and `max` of `year` column, grouped by director

How can we find multiple aggregations of any feature?

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

```

Out[29]:

director_name	min	max
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

How we find details of the movies by high budget directors?

Lets assume,

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

We can get the highest budget movie data of every director

In [ ]:
```
data_dir_budget = data.groupby("director_name")["budget"].max().reset_inc
data_dir_budget.head()
```

Out[30]:

	director_name	budget
0	Adam McKay	100.0
1	Adam Shankman	80.0
2	Alejandro González Iñárritu	135.0
3	Alex Proyas	140.0
4	Alexander Payne	30.0

How can we filter out the director names with max budget >100M?

In [ ]:
```
names = data_dir_budget.loc[data_dir_budget["budget"] >= 100, "director_r
```

Finally, how can we filter out the details of the movies by these directors?

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

Out[32]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May
...	...	...	...	...	...	...	...	...	...
1450	48267	0.40	33	100.00	Mad Max	6.6	1213	1979	Apr
1451	48268	0.20	13	4.51	Swingers	6.8	253	1996	Oct
1452	48274	0.00	5	2.61	Three	6.3	31	2010	Dec
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May

679 rows × 13 columns

Recall `isin()` from last lecture

Can we do filtering of groups in a single go?

YES

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

Out[33]:

	id_x	budget	popularity	revenue	title	vote_average	vote_count	year	month
0	43597	237.00	150	2787.97	Avatar	7.2	11800	2009	Dec
1	43598	300.00	139	961.00	Pirates of the Caribbean: At World's End	6.9	4500	2007	May
2	43599	245.00	107	880.67	Spectre	6.3	4466	2015	Oct
3	43600	250.00	112	1084.94	The Dark Knight Rises	7.6	9106	2012	Jul
4	43602	258.00	115	890.87	Spider-Man 3	5.9	3576	2007	May
...	...	...	...	...	...	...	...	...	...
1450	48267	0.40	33	100.00	Mad Max	6.6	1213	1979	Apr
1451	48268	0.20	13	4.51	Swingers	6.8	253	1996	Oct
1452	48274	0.00	5	2.61	Three	6.3	31	2010	Dec
1458	48335	0.06	27	3.22	Pi	7.1	586	1998	Jul
1460	48363	0.00	3	0.32	The Last Waltz	7.9	64	1978	May

679 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**