In [20]:

```
# If numpy or pandas are missing
!conda install --yes numpy
!conda install --yes pandas
```

Missing Data

Make sure to review the video for a full discussion on the strategies of dealing with missing data.

What Null/NA/nan objects look like:

Source: https://github.com/pandas-dev/pandas/issues/28095

A new pd.NA value (singleton) is introduced to represent scalar missing values. Up to now, pandas used several values to represent missing data: np.nan is used for this for float data, np.nan or None for object-dtype data and pd.NaT for datetime-like data. The goal of pd.NA is to provide a "missing" indicator that can be used consistently across data types. pd.NA is currently used by the nullable integer and boolean data types and the new string data type

In [2]:

```
import numpy as np
import pandas as pd
```

Some ways of missing values and how it shows

In [3]:

```
np.nan
```

Out[3]:

```
nan
```

In [4]:

```
pd.NA
```

Out[4]:

```
<NA>
```

In [5]:

```
pd.NaT
```

Out[5]:

```
NaT
```

Note! Typical comparisons should be avoided with Missing Values

- https://towardsdatascience.com/navigating-the-hell-of-nans-in-python-71b12558895b
- https://stackoverflow.com/questions/20320022/why-in-numpy-nan-nan-is-false-while-nan-in-nan-is-true

This is generally because the logic here is, since we don't know these values, we can't know if they are equal to each other.

In [6]:

```
np.nan == np.nan #It loookmlike it will be true but it false becaus one missing value is not equal to another missing value
```

Out[6]:

```
False
```

In [7]:

```
np.nan is np.nan #To check missing value is present or not , True of yes and false fow no missing value
```

Out[7]:

```
True
```

In [8]:

```
# Here we create variable myvar that is null
myvar = np.nan
```

In [9]:

```
# to check weather the its missng value or not, true means yes it have missing values and false means no it dont have
myvar is np.nan # we use this because np.nan == np.nan will always retun false
```

Out[9]:

```
True
```

Checking and Selecting for Null Values

In [64]:

```
df= pd.read_csv("D:\\Study\\Programming\\python\\Python course from udemy\\[GigaCourse.Com] Udemy - 2022 Python for Machine Learning & Data Science Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\03-Pandas\\movie_scores.csv")
```

In [11]:

```
df
```

Out[11]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
1	NaN	NaN	NaN	NaN	NaN	NaN
2	Hugh	Jackman	51.0	m	NaN	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [12]:

```
df.isnull()  # Here it will return true for null value and false for not null
```

Out[12]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	False	False	False	False	False	False
1	True	True	True	True	True	True
2	False	False	False	False	True	True

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
3	False	False	False	False	False	False
4	False	False	False	False	False	False

In [13]:

```
df.notnull() # Just opposite to .isnull()
```

Out[13]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	True	True	True	True	True	True
1	False	False	False	False	False	False
2	True	True	True	True	False	False
3	True	True	True	True	True	True
4	True	True	True	True	True	True

In [14]:

```
df['pre_movie_score'].notnull()
```

Out[14]:

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

In [15]:

```
# df data where pre_movie_score is not null
df[df['pre_movie_score'].notnull()]
```

Out[15]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [16]:

```
# df data where pre_movie_score is null
df[df['pre_movie_score'].isnull()]
```

Out[16]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
1	NaN	NaN	NaN	NaN	NaN	NaN
2	Hugh	Jackman	51.0	m	NaN	NaN

In [18]:

```
# Apply two conditions at once, where pre_movie_score is null but first name is not null
df[(df['pre_movie_score'].isnull()) & (df['first_name'].notnull())]
```

Out[18]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
2	Hugh	Jackman	51.0	m	NaN	NaN

Drop Data

In [21]:

```
help(df.dropna)   # for Help in in .dropna
```

In [23]:

```
df.dropna()# Here it drop all na values
```

Out[23]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [33]:

```
df.dropna(thresh=4)  # It will delete where non null are values are atleast 4 , imdex val
ues is also count as 1
```

Out[33]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
2	Hugh	Jackman	51.0	m	NaN	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [35]:

```
df.dropna(thresh=5)  # Here Hugh Jackman row also got drop because it have 4 non null valu
es
```

Out[35]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [38]:

```
df
```

Out[38]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
1	NaN	NaN	NaN	NaN	NaN	NaN
2	Hugh	Jackman	51.0	m	NaN	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [39]:

```
df.dropna(axis=1) # on the bases of column all column have atleast 1 null value so it del
ete all column
```

Out[39]:

0
1
2
3
4

In [40]:

```
df.dropna(axis=0) # It will drop all rows where any null values is present
```

Out[40]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [42]:

```
df.dropna(subset=['last_name']) # Here it only remove full row where null value is prese
nt in set column here
```

Out[42]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
2	Hugh	Jackman	51.0	m	NaN	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

Fill Data

In [46]:

```
help(df.fillna) # Help for fillna
```

In [48]:

```
df.fillna('NEW VALUE!') # Here it will replace all  null value with NEW VALUE!
```

Out[48]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
1	NEW VALUE!	NEW VALUE!	NEW VALUE!	NEW VALUE!	NEW VALUE!	NEW VALUE!
2	Hugh	Jackman	51.0	m	NEW VALUE!	NEW VALUE!
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [49]:

```
df['pre_movie_score'].fillna(0) # Replace null value in pre_movie_score with 0
```

Out[49]:

```
0    8.0
1    0.0
2    0.0
3    6.0
4    7.0
Name: pre_movie_score, dtype: float64
```

In [52]:

```
df['pre_movie_score'] = df['pre_movie_score'].fillna(0)
df # Put that in df DataFrame
```

Out[52]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
1	NaN	NaN	NaN	NaN	0.0	NaN
2	Hugh	Jackman	51.0	m	0.0	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [65]:

```
df['pre_movie_score'].mean()  # Here w find out the mean of pre_movie_score
```

Out[65]:

```
7.0
```

In [66]:

```
# Here we replace null values with mean
df['pre_movie_score'] = df['pre_movie_score'].fillna(df['pre_movie_score'].mean())
df
```

Out[66]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.0	m	8.0	10.0
1	NaN	NaN	NaN	NaN	7.0	NaN
2	Hugh	Jackman	51.0	m	7.0	NaN
3	Oprah	Winfrey	66.0	f	6.0	8.0
4	Emma	Stone	31.0	f	7.0	9.0

In [70]:

```
# Fill all null values with mean of the column
df.fillna(df.mean())
```

```
C:\Users\Chromsy\AppData\Local\Temp\ipykernel_12296\2122025003.py:2: FutureWarning: Dropp
ing of nuisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated;
in a future version this will raise TypeError.  Select only valid columns before calling
the reduction.
  df.fillna(df.mean())
```

Out[70]:

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
0	Tom	Hanks	63.00	m	8.0	10.0
1	NaN	NaN	52.75	NaN	7.0	9.0

	first_name	last_name	age	sex	pre_movie_score	post_movie_score
2	Hugh	Jackman	51.00	m	7.0	9.0
3	Oprah	Winfrey	66.00	f	6.0	8.0
4	Emma	Stone	31.00	f	7.0	9.0

Filling with Interpolation

Be careful with this technique, you should try to really understand whether or not this is a valid choice for your data. You should also note there are several methods available, the default is a linear method.

Full Docs on this Method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html

In [71]:

```
airline_tix = {'first':100,'business':np.nan,'economy-plus':50,'economy':30}
ser = pd.Series(airline_tix)
ser
```

Out[71]:

```
first           100.0
business          NaN
economy-plus     50.0
economy          30.0
dtype: float64
```

In [74]:

```
ser.interpolate()  # Here interpolate() it work when data is in series and i will us avera
ge of near values
```

Out[74]:

```
first           100.0
business         75.0
economy-plus     50.0
economy          30.0
dtype: float64
```

Groupby Operations and Multi-level Index

In [81]:

```
dff=pd.read_csv("D:\\Study\\Programming\\python\\Python course from udemy\\[GigaCourse.C
om] Udemy - 2022 Python for Machine Learning & Data Science Masterclass\\01 - Introductio
n to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\03-Pandas\\mpg.csv")
dff
```

Out[81]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
396	28.0	4	120.0	79	2625	18.6	82	1	ford ranger

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
397	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

398 rows × 9 columns

Here we are going to find average of mpg(miles per gallon) has change as per model_year

In [85]:

```
dff['model_year'].unique()
```

Out[85]:

```
array([70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82], dtype=int64)
```

In [87]:

```
dff['model_year'].value_counts()
```

Out[87]:

```
73    40
78    36
76    34
82    31
75    30
70    29
79    29
80    29
81    29
71    28
72    28
77    28
74    27
Name: model_year, dtype: int64
```

In [89]:

```
dff.groupby('model_year')
```

Out[89]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000294AEC245E0>
```

In [91]:

```
dff.groupby('model_year').mean() # we take mean of all data , check list below
```

Out[91]:

	mpg	cylinders	displacement	weight	acceleration	origin
model_year						
70	17.689655	6.758621	281.413793	3372.793103	12.948276	1.310345
71	21.250000	5.571429	209.750000	2995.428571	15.142857	1.428571
72	18.714286	5.821429	218.375000	3237.714286	15.125000	1.535714
73	17.100000	6.375000	256.875000	3419.025000	14.312500	1.375000
74	22.703704	5.259259	171.740741	2877.925926	16.203704	1.666667
75	20.266667	5.600000	205.533333	3176.800000	16.050000	1.466667
76	21.573529	5.647059	197.794118	3078.735294	15.941176	1.470588
77	23.375000	5.464286	191.392857	2997.357143	15.435714	1.571429
78	24.061111	5.361111	177.805556	2861.805556	15.805556	1.611111
79	25.093103	5.827586	206.689655	3055.344828	15.813793	1.275862
80	33.696552	4.137931	115.827586	2436.655172	16.934483	2.206897
81	30.334483	4.620690	135.310345	2522.931034	16.306897	1.965517

82	31.709677	4.193548	128.870968	2453.548387	16.638710	1.645161
	mpg	cylinders	displacement	weight	acceleration	origin

Adding an aggregate method call. To use a grouped object, you need to tell pandas how you want to aggregate the data.

Common Options:

```
mean(): Compute mean of groups
sum(): Compute sum of group values
size(): Compute group sizes
count(): Compute count of group
std(): Standard deviation of groups
var(): Compute variance of groups
sem(): Standard error of the mean of groups
describe(): Generates descriptive statistics
first(): Compute first of group values
last(): Compute last of group values
nth() : Take nth value, or a subset if n is a list
min(): Compute min of group values
max(): Compute max of group values
```

Full List at the Online Documentation: https://pandas.pydata.org/docs/reference/groupby.html

In [94]:

```
dff.groupby('model_year').mean()['mpg'] # mean of mgp as per model_year
```

Out[94]:

```
model_year
70    17.689655
71    21.250000
72    18.714286
73    17.100000
74    22.703704
75    20.266667
76    21.573529
77    23.375000
78    24.061111
79    25.093103
80    33.696552
81    30.334483
82    31.709677
Name: mpg, dtype: float64
```

Groupby Multiple Columns

Let's explore average mpg per year per cylinder count

In [114]:

```
year_cyl = dff.groupby(['model_year','cylinders']).mean()
year_cyl
```

Out[114]:

		mpg	displacement	weight	acceleration	origin
model_year	cylinders					
70	4	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	4	27.461538	101.846154	2056.384615	16.961538	1.923077

model_year	cylinders	mpg	displacement	weight	acceleration	origin
	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
72	3	19.000000	70.000000	2330.000000	13.500000	3.000000
	4	23.428571	111.535714	2382.642857	17.214286	1.928571
	8	13.615385	344.846154	4228.384615	13.000000	1.000000
73	3	18.000000	70.000000	2124.000000	13.500000	3.000000
	4	22.727273	109.272727	2338.090909	17.136364	2.000000
	6	19.000000	212.250000	2917.125000	15.687500	1.250000
	8	13.200000	365.250000	4279.050000	12.250000	1.000000
74	4	27.800000	96.533333	2151.466667	16.400000	2.200000
	6	17.857143	230.428571	3320.000000	16.857143	1.000000
	8	14.200000	315.200000	4438.400000	14.700000	1.000000
75	4	25.250000	114.833333	2489.250000	15.833333	2.166667
	6	17.583333	233.750000	3398.333333	17.708333	1.000000
	8	15.666667	330.500000	4108.833333	13.166667	1.000000
76	4	26.766667	106.333333	2306.600000	16.866667	1.866667
	6	20.000000	221.400000	3349.600000	17.000000	1.300000
	8	14.666667	324.000000	4064.666667	13.222222	1.000000
77	3	21.500000	80.000000	2720.000000	13.500000	3.000000
	4	29.107143	106.500000	2205.071429	16.064286	1.857143
	6	19.500000	220.400000	3383.000000	16.900000	1.400000
	8	16.000000	335.750000	4177.500000	13.662500	1.000000
78	4	29.576471	112.117647	2296.764706	16.282353	2.117647
	5	20.300000	131.000000	2830.000000	15.900000	2.000000
	6	19.066667	213.250000	3314.166667	16.391667	1.166667
	8	19.050000	300.833333	3563.333333	13.266667	1.000000
79	4	31.525000	113.583333	2357.583333	15.991667	1.583333
	5	25.400000	183.000000	3530.000000	20.100000	2.000000
	6	22.950000	205.666667	3025.833333	15.433333	1.000000
	8	18.630000	321.400000	3862.900000	15.400000	1.000000
80	3	23.700000	70.000000	2420.000000	12.500000	3.000000
	4	34.612000	111.000000	2360.080000	17.144000	2.200000
	5	36.400000	121.000000	2950.000000	19.900000	2.000000
	6	25.900000	196.500000	3145.500000	15.050000	2.000000
81	4	32.814286	108.857143	2275.476190	16.466667	2.095238
	6	23.428571	184.000000	3093.571429	15.442857	1.714286
	8	26.600000	350.000000	3725.000000	19.000000	1.000000
82	4	32.071429	118.571429	2402.321429	16.703571	1.714286
	6	28.333333	225.000000	2931.666667	16.033333	1.000000

In [118]:

```
dff.groupby(['model_year','cylinders']).describe().transpose()
```

Out[118]:

model_year	70			71			72
cylinders	4	6	8	4	6	8	3

	model_year	70			71			72	
	cylinders	4	6	8	4	6	8	3	
mpg	count	7.000000	4.000000	18.000000	13.000000	8.000000	7.000000	1.0	14.00000
	mean	25.285714	20.500000	14.111111	27.461538	18.000000	13.428571	19.0	23.42857
	std	1.112697	1.732051	2.609685	3.502746	1.069045	0.786796	NaN	3.05624
	min	24.000000	18.000000	9.000000	22.000000	16.000000	12.000000	19.0	18.00000
	25%	24.500000	20.250000	14.000000	25.000000	17.750000	13.000000	19.0	21.25000
	50%	25.000000	21.000000	14.500000	27.000000	18.000000	14.000000	19.0	23.00000
	75%	26.000000	21.250000	15.000000	30.000000	19.000000	14.000000	19.0	25.75000
	max	27.000000	22.000000	18.000000	35.000000	19.000000	14.000000	19.0	28.00000
displacement	count	7.000000	4.000000	18.000000	13.000000	8.000000	7.000000	1.0	14.00000
	mean	107.000000	199.000000	367.555556	101.846154	243.375000	371.714286	70.0	111.53571
	std	8.660254	0.816497	59.076443	23.053728	11.867573	32.438220	NaN	14.16722
	min	97.000000	198.000000	302.000000	71.000000	225.000000	318.000000	70.0	96.00000
	25%	100.500000	198.750000	309.750000	88.000000	232.000000	350.500000	70.0	97.12500
	50%	107.000000	199.000000	355.000000	97.000000	250.000000	383.000000	70.0	116.50000
	75%	111.500000	199.250000	421.750000	116.000000	250.000000	400.000000	70.0	121.00000
	max	121.000000	200.000000	455.000000	140.000000	258.000000	400.000000	70.0	140.00000
weight	count	7.000000	4.000000	18.000000	13.000000	8.000000	7.000000	1.0	14.00000
	mean	2292.571429	2710.500000	3940.055556	2056.384615	3171.875000	4537.714286	2330.0	2382.64285
	std	263.055037	112.837641	496.964488	217.933300	259.952434	415.550524	NaN	274.99247
	min	1835.000000	2587.000000	3086.000000	1613.000000	2634.000000	4096.000000	2330.0	2100.00000
	25%	2182.000000	2632.750000	3518.750000	1955.000000	3094.750000	4181.500000	2330.0	2198.25000
	50%	2372.000000	2711.000000	3805.500000	2074.000000	3285.000000	4464.000000	2330.0	2283.00000
	75%	2402.500000	2788.750000	4370.500000	2220.000000	3308.750000	4850.500000	2330.0	2481.50000
	max	2672.000000	2833.000000	4732.000000	2408.000000	3439.000000	5140.000000	2330.0	2979.00000
acceleration	count	7.000000	4.000000	18.000000	13.000000	8.000000	7.000000	1.0	14.00000
	mean	16.000000	15.500000	11.194444	16.961538	14.750000	12.214286	13.5	17.21428
	std	2.661453	0.408248	2.668657	2.536907	1.000000	0.755929	NaN	2.42355
	min	12.500000	15.000000	8.000000	14.000000	13.000000	11.500000	13.5	14.50000
	25%	14.500000	15.375000	9.625000	14.500000	14.250000	11.750000	13.5	15.62500
	50%	15.000000	15.500000	10.250000	18.000000	15.250000	12.000000	13.5	16.75000
	75%	17.500000	15.625000	12.000000	19.000000	15.500000	12.500000	13.5	18.00000
	max	20.500000	16.000000	18.500000	20.500000	15.500000	13.500000	13.5	23.50000
origin	count	7.000000	4.000000	18.000000	13.000000	8.000000	7.000000	1.0	14.00000
	mean	2.285714	1.000000	1.000000	1.923077	1.000000	1.000000	3.0	1.92857
	std	0.487950	0.000000	0.000000	0.862316	0.000000	0.000000	NaN	0.82874
	min	2.000000	1.000000	1.000000	1.000000	1.000000	1.000000	3.0	1.00000
	25%	2.000000	1.000000	1.000000	1.000000	1.000000	1.000000	3.0	1.00000
	50%	2.000000	1.000000	1.000000	2.000000	1.000000	1.000000	3.0	2.00000
	75%	2.500000	1.000000	1.000000	3.000000	1.000000	1.000000	3.0	2.75000
	max	3.000000	1.000000	1.000000	3.000000	1.000000	1.000000	3.0	3.00000

40 rows × 43 columns

In [102]:

```
mydff.groupby(['model_year','cylinders']).mean()['mpg'].index # If we want call by index
```

```
number
```

Out[102]:

```
MultiIndex([(70, 4),
            (70, 6),
            (70, 8),
            (71, 4),
            (71, 6),
            (71, 8),
            (72, 3),
            (72, 4),
            (72, 8),
            (73, 3),
            (73, 4),
            (73, 6),
            (73, 8),
            (74, 4),
            (74, 6),
            (74, 8),
            (75, 4),
            (75, 6),
            (75, 8),
            (76, 4),
            (76, 6),
            (76, 8),
            (77, 3),
            (77, 4),
            (77, 6),
            (77, 8),
            (78, 4),
            (78, 5),
            (78, 6),
            (78, 8),
            (79, 4),
            (79, 5),
            (79, 6),
            (79, 8),
            (80, 3),
            (80, 4),
            (80, 5),
            (80, 6),
            (81, 4),
            (81, 6),
            (81, 8),
            (82, 4),
            (82, 6)],
           names=['model_year', 'cylinders'])
```

In [122]:

```
dff.groupby(['model_year','cylinders']).mean()['mpg'].index.names # Name of index or nam
e of columns
# OR year_cyl['mpg'].index.names
```

Out[122]:

```
FrozenList(['model_year', 'cylinders'])
```

In [111]:

```
dff.groupby(['model_year','cylinders']).mean()['mpg'].index.levels # outer index numbers
which is model_year then
                                                                  # inner number which i
s number of cylinders
```

Out[111]:

```
FrozenList([[70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82], [3, 4, 5, 6, 8]])
```

In [125]:

```
year_cyl.loc[70] # Here iloc will give error
```

```
# OR dff.groupby(['model_year','cylinders']).mean().loc[70]
```

Out[125]:

cylinders	mpg	displacement	weight	acceleration	origin
4	25.285714	107.000000	2292.571429	16.000000	2.285714
6	20.500000	199.000000	2710.500000	15.500000	1.000000
8	14.111111	367.555556	3940.055556	11.194444	1.000000

In [128]:

```
year_cyl.loc[[70,82]] # to see more than one value
```

Out[128]:

model_year	cylinders	mpg	displacement	weight	acceleration	origin
70	4	25.285714	107.000000	2292.571429	16.000000	2.285714
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
82	4	32.071429	118.571429	2402.321429	16.703571	1.714286
	6	28.333333	225.000000	2931.666667	16.033333	1.000000

Grab a Single Row

In [131]:

```
year_cyl.loc[70,6] # Here give both outer and inner index number
```

Out[131]:

```
mpg               20.5
displacement     199.0
weight          2710.5
acceleration      15.5
origin             1.0
Name: (70, 6), dtype: float64
```

Grab Based on Cross-section with .xs()

This method takes a `key` argument to select data at a particular level of a MultiIndex.

Parameters

```
key : label or tuple of label
    Label contained in the index, or partially in a MultiIndex.
axis : {0 or 'index', 1 or 'columns'}, default 0
    Axis to retrieve cross-section on.
level : object, defaults to first n levels (n=1 or len(key))
    In case of a key partially contained in a MultiIndex, indicate
    which levels are used. Levels can be referred by label or position.
```

In [141]:

```
year_cyl.xs(key=70,level='model_year') # Here also work as loc but there is a difference we will see later
                                        # It take one value at a time not like loc where w
```

```
e can porvide list of values
```

Out[141]:

	mpg	displacement	weight	acceleration	origin
cylinders					
4	25.285714	107.000000	2292.571429	16.000000	2.285714
6	20.500000	199.000000	2710.500000	15.500000	1.000000
8	14.111111	367.555556	3940.055556	11.194444	1.000000

In [135]:

```
year_cyl.loc[70] # same as xs but now we will see difference
```

Out[135]:

	mpg	displacement	weight	acceleration	origin
cylinders					
4	25.285714	107.000000	2292.571429	16.000000	2.285714
6	20.500000	199.000000	2710.500000	15.500000	1.000000
8	14.111111	367.555556	3940.055556	11.194444	1.000000

In [140]:

```
year_cyl.xs(key=4,level='cylinders') # Now we see differnce b calling level as column nam
e we can call index ,
                                      # here it show only 4 cylinder from all years
```

Out[140]:

	mpg	displacement	weight	acceleration	origin
model_year					
70	25.285714	107.000000	2292.571429	16.000000	2.285714
71	27.461538	101.846154	2056.384615	16.961538	1.923077
72	23.428571	111.535714	2382.642857	17.214286	1.928571
73	22.727273	109.272727	2338.090909	17.136364	2.000000
74	27.800000	96.533333	2151.466667	16.400000	2.200000
75	25.250000	114.833333	2489.250000	15.833333	2.166667
76	26.766667	106.333333	2306.600000	16.866667	1.866667
77	29.107143	106.500000	2205.071429	16.064286	1.857143
78	29.576471	112.117647	2296.764706	16.282353	2.117647
79	31.525000	113.583333	2357.583333	15.991667	1.583333
80	34.612000	111.000000	2360.080000	17.144000	2.200000
81	32.814286	108.857143	2275.476190	16.466667	2.095238
82	32.071429	118.571429	2402.321429	16.703571	1.714286

Careful note!

Keep in mind, its usually much easier to filter out values before running a groupby() call, so you should attempt to filter out any values/categories you don't want to use. For example, its much easier to remove 4 cylinder cars before the groupby() call, very difficult to this sort of thing after a group by.

Question:

Now we want to data from only 6,8 cylinders from each year and we know that xs take only 1 value and loc doesnt short data like this

In [149]:

```
# Here is the the trick we are going to use loc but we will short out data only for 6 and 8 cylinders with isin (is in)
dff[dff['cylinders'].isin([6,8])]
```

Out[149]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
365	20.2	6	200.0	88	3060	17.1	81	1	ford granada gl
366	17.6	6	225.0	85	3465	16.6	81	1	chrysler lebaron salon
386	25.0	6	181.0	110	2945	16.4	82	1	buick century limited
387	38.0	6	262.0	85	3015	17.0	82	1	oldsmobile cutlass ciera (diesel)
389	22.0	6	232.0	112	2835	14.7	82	1	ford granada l

187 rows × 9 columns

In [150]:

```
# Here we get data for 6 and 8
dff[dff['cylinders'].isin([6,8])].groupby(['model_year','cylinders']).mean()
```

Out[150]:

		mpg	displacement	weight	acceleration	origin
model_year	**cylinders**					
70	**6**	20.500000	199.000000	2710.500000	15.500000	1.000000
	8	14.111111	367.555556	3940.055556	11.194444	1.000000
71	**6**	18.000000	243.375000	3171.875000	14.750000	1.000000
	8	13.428571	371.714286	4537.714286	12.214286	1.000000
72	**8**	13.615385	344.846154	4228.384615	13.000000	1.000000
73	**6**	19.000000	212.250000	2917.125000	15.687500	1.250000
	8	13.200000	365.250000	4279.050000	12.250000	1.000000
74	**6**	17.857143	230.428571	3320.000000	16.857143	1.000000
	8	14.200000	315.200000	4438.400000	14.700000	1.000000
75	**6**	17.583333	233.750000	3398.333333	17.708333	1.000000
	8	15.666667	330.500000	4108.833333	13.166667	1.000000
76	**6**	20.000000	221.400000	3349.600000	17.000000	1.300000
	8	14.666667	324.000000	4064.666667	13.222222	1.000000
77	**6**	19.500000	220.400000	3383.000000	16.900000	1.400000
	8	16.000000	335.750000	4177.500000	13.662500	1.000000
78	**6**	19.066667	213.250000	3314.166667	16.391667	1.166667
	8	19.050000	300.833333	3563.333333	13.266667	1.000000

model_year	cylinders	mpg	displacement	weight	acceleration	origin
79	6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	8	18.630000	321.400000	3862.900000	15.400000	1.000000
80	6	25.900000	196.500000	3145.500000	15.050000	2.000000
81	6	23.428571	184.000000	3093.571429	15.442857	1.714286
	8	26.600000	350.000000	3725.000000	19.000000	1.000000
82	6	28.333333	225.000000	2931.666667	16.033333	1.000000

Swap Levels

- **Swapping Levels:** https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html#swapping-levels-with-swaplevel
- **Generalized Method is reorder_levels:** https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html#reordering-levels-with-reorder-levels

In [151]:

```
# Here we can swap inner and outer level
year_cyl.swaplevel()
```

Out[151]:

cylinders	model_year	mpg	displacement	weight	acceleration	origin
4	70	25.285714	107.000000	2292.571429	16.000000	2.285714
6	70	20.500000	199.000000	2710.500000	15.500000	1.000000
8	70	14.111111	367.555556	3940.055556	11.194444	1.000000
4	71	27.461538	101.846154	2056.384615	16.961538	1.923077
6	71	18.000000	243.375000	3171.875000	14.750000	1.000000
8	71	13.428571	371.714286	4537.714286	12.214286	1.000000
3	72	19.000000	70.000000	2330.000000	13.500000	3.000000
4	72	23.428571	111.535714	2382.642857	17.214286	1.928571
8	72	13.615385	344.846154	4228.384615	13.000000	1.000000
3	73	18.000000	70.000000	2124.000000	13.500000	3.000000
4	73	22.727273	109.272727	2338.090909	17.136364	2.000000
6	73	19.000000	212.250000	2917.125000	15.687500	1.250000
8	73	13.200000	365.250000	4279.050000	12.250000	1.000000
4	74	27.800000	96.533333	2151.466667	16.400000	2.200000
6	74	17.857143	230.428571	3320.000000	16.857143	1.000000
8	74	14.200000	315.200000	4438.400000	14.700000	1.000000
4	75	25.250000	114.833333	2489.250000	15.833333	2.166667
6	75	17.583333	233.750000	3398.333333	17.708333	1.000000
8	75	15.666667	330.500000	4108.833333	13.166667	1.000000
4	76	26.766667	106.333333	2306.600000	16.866667	1.866667
6	76	20.000000	221.400000	3349.600000	17.000000	1.300000
8	76	14.666667	324.000000	4064.666667	13.222222	1.000000
3	77	21.500000	80.000000	2720.000000	13.500000	3.000000
4	77	29.107143	106.500000	2205.071429	16.064286	1.857143
6	77	19.500000	220.400000	3383.000000	16.900000	1.400000
8	77	16.000000	335.750000	4177.500000	13.662500	1.000000

cylinders	model_year	mpg	displacement	weight	acceleration	origin
4	78	29.576471	112.117647	2296.764706	16.282353	2.117647
5	78	20.300000	131.000000	2830.000000	15.900000	2.000000
6	78	19.066667	213.250000	3314.166667	16.391667	1.166667
8	78	19.050000	300.833333	3563.333333	13.266667	1.000000
4	79	31.525000	113.583333	2357.583333	15.991667	1.583333
5	79	25.400000	183.000000	3530.000000	20.100000	2.000000
6	79	22.950000	205.666667	3025.833333	15.433333	1.000000
8	79	18.630000	321.400000	3862.900000	15.400000	1.000000
3	80	23.700000	70.000000	2420.000000	12.500000	3.000000
4	80	34.612000	111.000000	2360.080000	17.144000	2.200000
5	80	36.400000	121.000000	2950.000000	19.900000	2.000000
6	80	25.900000	196.500000	3145.500000	15.050000	2.000000
4	81	32.814286	108.857143	2275.476190	16.466667	2.095238
6	81	23.428571	184.000000	3093.571429	15.442857	1.714286
8	81	26.600000	350.000000	3725.000000	19.000000	1.000000
4	82	32.071429	118.571429	2402.321429	16.703571	1.714286
6	82	28.333333	225.000000	2931.666667	16.033333	1.000000

Sorting MultiIndex

- https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html#sorting-a-multiindex

In [153]:

```
# Here we sort on the bases of model_year in decreasing order by setting ascending = fals
e
year_cyl.sort_index(level='model_year', ascending=False)
```

Out[153]:

model_year	cylinders	mpg	displacement	weight	acceleration	origin
82	6	28.333333	225.000000	2931.666667	16.033333	1.000000
	4	32.071429	118.571429	2402.321429	16.703571	1.714286
81	8	26.600000	350.000000	3725.000000	19.000000	1.000000
	6	23.428571	184.000000	3093.571429	15.442857	1.714286
	4	32.814286	108.857143	2275.476190	16.466667	2.095238
80	6	25.900000	196.500000	3145.500000	15.050000	2.000000
	5	36.400000	121.000000	2950.000000	19.900000	2.000000
	4	34.612000	111.000000	2360.080000	17.144000	2.200000
	3	23.700000	70.000000	2420.000000	12.500000	3.000000
79	8	18.630000	321.400000	3862.900000	15.400000	1.000000
	6	22.950000	205.666667	3025.833333	15.433333	1.000000
	5	25.400000	183.000000	3530.000000	20.100000	2.000000
	4	31.525000	113.583333	2357.583333	15.991667	1.583333
78	8	19.050000	300.833333	3563.333333	13.266667	1.000000
	6	19.066667	213.250000	3314.166667	16.391667	1.166667
	5	20.300000	131.000000	2830.000000	15.900000	2.000000

		mpg	displacement	weight	acceleration	origin
model_year	cylinders					
	4	29.576471	112.117647	2296.764706	16.282353	2.117647
77	8	16.000000	335.750000	4177.500000	13.662500	1.000000
	6	19.500000	220.400000	3383.000000	16.900000	1.400000
	4	29.107143	106.500000	2205.071429	16.064286	1.857143
	3	21.500000	80.000000	2720.000000	13.500000	3.000000
76	8	14.666667	324.000000	4064.666667	13.222222	1.000000
	6	20.000000	221.400000	3349.600000	17.000000	1.300000
	4	26.766667	106.333333	2306.600000	16.866667	1.866667
75	8	15.666667	330.500000	4108.833333	13.166667	1.000000
	6	17.583333	233.750000	3398.333333	17.708333	1.000000
	4	25.250000	114.833333	2489.250000	15.833333	2.166667
74	8	14.200000	315.200000	4438.400000	14.700000	1.000000
	6	17.857143	230.428571	3320.000000	16.857143	1.000000
	4	27.800000	96.533333	2151.466667	16.400000	2.200000
73	8	13.200000	365.250000	4279.050000	12.250000	1.000000
	6	19.000000	212.250000	2917.125000	15.687500	1.250000
	4	22.727273	109.272727	2338.090909	17.136364	2.000000
	3	18.000000	70.000000	2124.000000	13.500000	3.000000
72	8	13.615385	344.846154	4228.384615	13.000000	1.000000
	4	23.428571	111.535714	2382.642857	17.214286	1.928571
	3	19.000000	70.000000	2330.000000	13.500000	3.000000
71	8	13.428571	371.714286	4537.714286	12.214286	1.000000
	6	18.000000	243.375000	3171.875000	14.750000	1.000000
	4	27.461538	101.846154	2056.384615	16.961538	1.923077
70	8	14.111111	367.555556	3940.055556	11.194444	1.000000
	6	20.500000	199.000000	2710.500000	15.500000	1.000000
	4	25.285714	107.000000	2292.571429	16.000000	2.285714

Advanced: agg() method

The agg() method allows you to customize what aggregate functions you want per category

In [158]:

```
dff
```

Out[158]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
396	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
397	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

398 rows × 9 columns

In [159]:

```
# These strings need to match up with built-in method names
dff.agg(['median','mean'])
```

```
C:\Users\Chromsy\AppData\Local\Temp\ipykernel_12296\3607702116.py:2: FutureWarning: ['hor
sepower', 'name'] did not aggregate successfully. If any error is raised this will raise
in a future version of pandas. Drop these columns/ops to avoid this warning.
  dff.agg(['median','mean'])
```

Out[159]:

	mpg	cylinders	displacement	weight	acceleration	model_year	origin
median	23.000000	4.000000	148.500000	2803.500000	15.50000	76.00000	1.000000
mean	23.514573	5.454774	193.425879	2970.424623	15.56809	76.01005	1.572864

In [161]:

```
dff.agg(['median','mean'])['mpg']
# If we wanna see only for one column
```

```
C:\Users\Chromsy\AppData\Local\Temp\ipykernel_12296\3128552350.py:1: FutureWarning: ['hor
sepower', 'name'] did not aggregate successfully. If any error is raised this will raise
in a future version of pandas. Drop these columns/ops to avoid this warning.
  dff.agg(['median','mean'])['mpg']
```

Out[161]:

```
median    23.000000
mean      23.514573
Name: mpg, dtype: float64
```

In [163]:

```
dff.agg(['sum','mean'])[['mpg','weight']]
```

```
C:\Users\Chromsy\AppData\Local\Temp\ipykernel_12296\127981824.py:1: FutureWarning: ['hors
epower', 'name'] did not aggregate successfully. If any error is raised this will raise i
n a future version of pandas. Drop these columns/ops to avoid this warning.
  dff.agg(['sum','mean'])[['mpg','weight']]
```

Out[163]:

	mpg	weight
sum	9358.800000	1.182229e+06
mean	23.514573	2.970425e+03

In [164]:

```
dff.agg({'mpg':['median','mean'],'weight':['mean','std']})
```

Out[164]:

	mpg	weight
median	23.000000	NaN
mean	23.514573	2970.424623
std	NaN	846.841774

agg() with groupby()

agg() with groupby()

In [165]:

```
dff.groupby('model_year').agg({'mpg':['median','mean'],'weight':['mean','std']})
```

Out[165]:

	mpg		weight	
	median	mean	mean	std
model_year				
70	16.00	17.689655	3372.793103	852.868663
71	19.00	21.250000	2995.428571	1061.830859
72	18.50	18.714286	3237.714286	974.520960
73	16.00	17.100000	3419.025000	974.809133
74	24.00	22.703704	2877.925926	949.308571
75	19.50	20.266667	3176.800000	765.179781
76	21.00	21.573529	3078.735294	821.371481
77	21.75	23.375000	2997.357143	912.825902
78	20.70	24.061111	2861.805556	626.023907
79	23.90	25.093103	3055.344828	747.881497
80	32.70	33.696552	2436.655172	432.235491
81	31.60	30.334483	2522.931034	533.600501
82	32.00	31.709677	2453.548387	354.276713

In [ ]: