Pandas important Concepts

* machine can only understand numbers and not the strings
* Here the species column is string so we need to convert them to numerical.

so for this we can assume

Iris-setosa = 0

Iris-versicolor = 1

Iris-Virginica = 2

Here we will learn advanced functions like:

1) map
2) apply
3) apply map

map(): It is a series function to do mapping.

* cannot be used on DataFrames.
* This is different from map in Python.
* we need to access the series and map on it.

```
datac["species"].unique()
↳ array(['Iris-setosa', 'Iris-versicolor', 'Iris-Virginica'],
        dtype=object)
```

* Now if we want give the species their respective numbers we use map() function.

```
datac["species"].map({'Iris-setosa':0, 'Iris-Versicolor':1,
                      'Iris-Virginica':2})
```

```
0    0
1    0
2    0
⋮
Name: species, Length: 150, dtype=int64
```

This is a temporary change so if we want to use it we can save it and use it.

```
data["species"] = datac["species"].map({'Iris-setosa':0,
                  'Iris-Versicolor':1,'Iris-Virginica':2})
```

```
datac["species"].unique()
↳ array([0, 1, 2], dtype=int64)
```

apply(): It is both series and DataFrame function.

How apply() works on series

* apply() takes any function as parameter
  - user defined
  - inbuilt
  - anonymous

  } input parameter

apply() applies this function elementwise to the series.

* The function used inside apply() will not have '()' as apply calls the function for it.

```
datac["species"].apply(type)
↳ 0    <class 'int'>
  1    <class 'int'>
  ⋮          ⋮
Name: species, length: 150, dtype: object
```

convert Sepallengthcm to complex.

```
datac["Sepallengthcm"].apply(complex)
↳ 0    5.1+0.0j
  1    4.9+0.0j
  ⋮
Name: Sepallengthcm, length: 150, dtype: complex128
```

* we can create our own function and use it

ex:-
```
def sq(x):
    return x**2
```

* return is mandatory as otherwise we cannot use result.

```
datac["Sepallengthcm"].apply(sq)
↳ 0    26.01
  1    24.01
  ⋮
Name: Sepallengthcm, length: 150, dtype = float64
```

apply() function for DataFrame

* apply() takes any function as parameter
  - user defined
  - inbuilt
  - anonymous

  } input parameter

* apply() applies the function axis wise
  - that is one row or one column at a time.
  - this is not element wise.

* So we need to use the functions that we can apply to the complete row or column.

ex:- datac[["sepallengthcm"]].apply(int)

↳ X error as int function needs to be applied element wise not to entire row or column.

* double square brackets indicate DataFrame.

datac[["sepallengthcm"]].apply(sum)

↳ sepallengthcm 876.5
dtype: float64

* default for sum is columnwise (axis=0)

Sepallengthcm has data only in single column so even if we do sum rowwise we get original values.

datac[["sepallengthcm"]].apply(sum, axis=1)

↳ 0 5.1
⋮
length: 150, dtype: float64

```
datac[["sepal length cm","sepal width cm"]].apply(sum)
```

```
"sepal length cm"   876.5
"sepal width cm"    458.1
dtype: float64
```

* adding "sepal length cm" with "sepal width cm" when axis=1 does not make sense logically.

applymap(): It is a DataFrame function.

* This is not a series function.
* when we pass any function to DataFrame it is applied element wise.

```
datac[["sepal length cm","sepal width cm"]].applymap(type)
```

	sepal length cm	sepal width cm
0	<class 'float'>	<class 'float'>
1	⋮	
	⋮	

150 rows x 2 columns

```
datac[["sepal length cm","sepal width cm"]].applymap(sq)
```

	sepal length cm	sepal width cm
0	26.01	12.25
1	⋮	
	⋮	

150 rows x 2 columns

String methods in pandas

* Pandas has a module called 'str' using which we can apply string methods.
* These string methods can only be applied on Series.
* All these are applied elementwise.

ex:- Some of the supported methods:

1) str.upper()
2) str.lower()
3) str.capitalize()
4) str.title()
5) str.isalpha()
6) str.isalnum()
7) str.isdigit()
8) str.startswith()
9) str.endswith()
10) str.swapcase()
11) str.replace()
12) str.index()
13) str.find()
14) str.strip()
15) str.rstrip()
16) str.lstrip()
17) str.partition()

18) str.contains( ) - checks if given string is present or not.

19) str.maketrans( ) - This cannot be directly applied to data we need to first get translate table assign it to variable and use this in translate method.

20) str.translate( )

ex:- datac["species"].str.upper( )

```
0    IRIS-SETOSA
1           :
:           :
Name: species, Length: 150, dtype: object
```

datac["species"].str.isalpha( )

```
0    False
1    False
:      :
Name: species, Length: 150, dtype: object
```

datac["species"].str.startswith("I")

```
0    True
1    True
:      :
Name: species, Length: 150, dtype: object
```

* we can use these methods in conditions

```
datac.loc[datac["species"].str.endswith("a")==False]
```

↳

	Sepallength cm	Sepalwidth cm	Petallength cm	petalwidth cm	Species
50	7.0	3.2	4.7	1.4	Iris-Versicolor

⋮

```
datac["species"].str.replace("i","I")
```

↳

```
0     IrIs-setosa
⋮
Name: species, Length: 150, dtype: object
```

```
datac["species"].str.partition("-")
```

↳

	0	1	2
0	Iris	-	setosa
⋮			

150 rows x 3 columns

*
```
a = str.maketrans("Ir","123")
datac["species"].str.translate(a)
```

↳

```
0     123s-setosa
⋮
Name: species, Length: 150, dtype: object
```

datac["species"].str.contains("virginica")

↳

0	False
⋮	⋮
145	True
⋮	

Name: species, Length: 150, dtype: object

Assignment:

```
import pandas as pd
pd.DataFrame({"name":["Subhadra Bhupathiraju", "pranavi
                Kondraguntla", "Raghav Kondraguntla"]})
```

↳

	name
0	Subhadra Bhupathiraju
1	Pranavi Kondraguntla
2	Raghav Kondraguntla

```
data = pd.DataFrame({"name":["Subhadra Bhupathiraju",
        "pranavi Kondraguntla", Raghav Kondraguntla"]})
```

```
type(data)
```

↳ pandas.core.frame.DataFrame

```
data["name"].str.split()
```

↳

0	[Subhadra, Bhupathiraju]
1	[Pranavi, Kondraguntla]
2	[Raghav, Kondraguntla]

write code such that first column has first name and second column has last name.

hint: Can use apply(lambda(split))

element wise so we use python split

```
data["First"] = data["name"].apply(lambda x:x.split()[0])
data["Second"] = data["name"].apply(lambda x:x.split()[1])
```

data

	name	First	second
0	Subhadra Bhupathiraju	Subhadra	Bhupathiraju
1	Pranavi Kondraguntla	Pranavi	Kondraguntla
2	Raghav Kondraguntla	Raghav	Kondraguntl

Advanced concepts in python pandas

Groupby :-

let us assume we are given a DataFrame consisting of Height, weight and Gender.

height	weight	Gender
H1	W1	m
H2	W2	f
H3	W3	m
H4	W4	f
H5	W5	f
H6	W6	m

3 feature Variables

$= D_6$ (data)

when we are doing analysis on this data. If we want to calculate things like.

- Avg height of men
- Avg height of women
- Avg weight of men
- min height of men

for this we can use data.loc[condition] but we have advanced concept called Groupby.

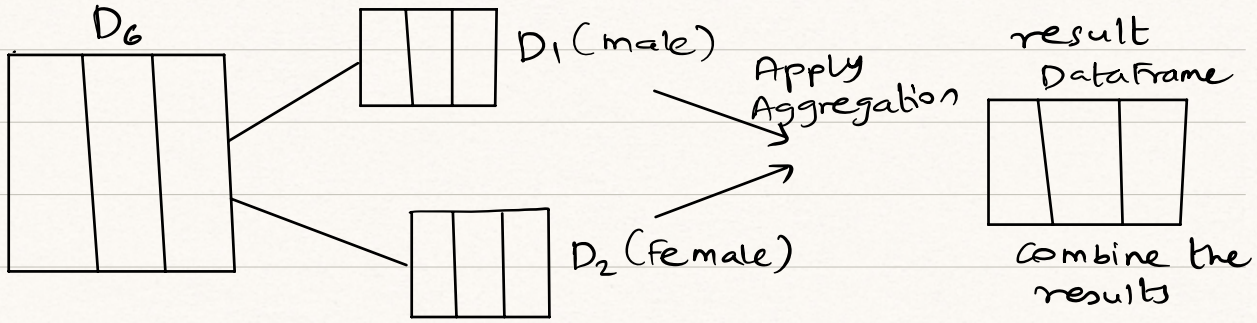
Split-apply-Combine : This refers to chain of three steps:

1) split a table (DataFrame) in to groups.
2) Apply operations to each of the smaller tables.
3) Combine the results.

Group by :-

we can form groups based on conditions.

we can divide the given data in to 2 groups based on a condition (here we can use Gender based on the problem statement)



* How many small DataFrames depends on the unique value of that feature variable.

D1 (DataFrame)

H1	W1	m
H3	W3	m
H6	W6	m

D2 (DataFrame)

H2	W2	f
H4	W4	f
H5	W5	f

* we can apply any Aggregations on this like:
  sum, mean, median, ...
* The result is combined to give one DataFrame.
* Groupby is mostly done as categorical data as we will get a subset of DataFrames.

The concept is Groupby.

```
import pandas as pd
pd.read_csv(r"C:\.....\Book4.csv")
```

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90

```
data = pd.read_csv(r"C:\....\Book4.csv")
```

groupby() function takes the condition as input

This is a DataFrame function.

```
data.groupby("Gender")
```

↳ Groupby object

```
g = data.groupby("Gender")
```

* There are various methods that we can apply on this Groupby object. we will learn a few.

ngroups gives how many groups are present

```
g.ngroups
```

↳ 2

groups gives the groups that are there and their row index.

g.groups

↳ {'f': [2, 3, 4, 5], 'm': [0, 1, 6]}

g actually internally behaves as dictionary

get_group() we can get that particular group.

g.get_group("m")

↳

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
6	P7	m	157	90

g.get_group("f")

↳

	Name	Gender	Height	weight
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43

* Since Groupby object acts as dictionary we can use for loop on it.

```
For keys, values in g:
    Print (keys)
```

↳

f

m

```
for keys in g:
    print(keys)
```

↳ ('f',

	Name	Gender	Height	weight
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43)

('m'

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
6	P7	m	157	90)

```
for keys,values in g:
    print(keys)
    print(values)
    print("----"*25)
```

↳ f

	Name	Gender	Height	weight
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43

---

m

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67

6	P7	m	157	90

If we want to see only female data

```
for keys, values in g:
    print(keys)
```

↳ f

	Name	Gender	Height	weight
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43

first data from the sub-dataset

```
g.first()
```

↳

Gender	Name	Height	weight
f	P3	125	67
m	P1	123	45

g.last()

↘

Gender	Name	Height	weight
f	P6	133	43
m	P7	157	90

g.min()

↘

Gender	Name	Height	weight
f	P3	12	43
m	P1	123	45

g["Height"].min()

↘

Gender
f 12
m 123
Name: Height, dtype: int64

we can count Non-null values in Variables

g.count()

↘

Gender	Name	Height	weight
f	4	4	4
m	3	3	3

max height and weight in male and female
these can be of different people.

```
g[["Height","Weight"]].max()
```

↳

Gender	Height	weight
f	145	89
m	157	90

Average height and weight in male and female
* Groupby is very very powerful in Pandas.

```
g[["Height","Weight"]].mean()
```

↳

Gender	Height	weight
f	103.750...	66.50...
m	145.333...	67.33...

describe() will give details for female and male based on height & weight as these are numerical values.

```
g.describe()
```

↳

	Height							weight						
Gender	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
f	4.0	103.0...	61.71...		12.0	96.75										
m	3.0	145.33...	19.34...		123.0	139.50										

data.describe()

	Height	weight
count	7.00...	7.00...
mean	121.5714	66.857
min	—	—
25%	—	—
50%	—	—
75%	—	—
max	—	—

describe by default only gives info for numerical columns.

if we want to see describe for categorical columns we can use include_object.

data.describe(include=object)

	Name	Gender
count	7	7
unique	7	2
top	P1	f
freq	1	4

top is which has highest frequency. If equal frequency then topmost value is used.

* we can also apply aggregation directly.

```
g.agg(["mean", "median", "count"])
```

	Height			weight		
	mean	median	count	mean	median	count
Gender						
f	103.75...	129.0	4	66.50...	67.0	4
m	145.33...	156.0	3	67.33...	67.0	3

* If we only want to apply aggregations on one particular column.

```
g["Height"].agg(["mean", "median", "count", "min", "max"])
```

	mean	median	count	min	max
Gender					
f	103.75...	129.0...	4	12	145
m	145.33...	156.0...	3	123	157

* If we want to apply aggregation column wise differently for different columns.

aggregation column wise

```
g.agg({"Height": ["mean", "median"], "weight": ["min", "max"]})
```

	Height		weight	
	mean	median	min	max
Gender				
f	103.750...	129.0	43	89
m	145.33...	156.0	45	90

Concatenation : Concatenation means combining 2 or more DataFrames in to a Single DataFrame

- row wise (axis=1)
- column wise (axis=0) (default)

* Default concatenation is columnwise and uses column labels.
* when we are doing row wise we will use row indexes, it will check for each row index and Concatenate, missing data is marked as NaN.
* During concatenation index values are also concatenated so we use ignore_index= True.

```
import pandas as pd
data = pd.read_csv(r"C:\.....\Book4.csv")
```

data

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90

```
datac = data.copy()
```

```
pd.concat([data, datac])
```

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90

```
pd.concat([data, datac], ignore_index=True)
```

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89

4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90
7	P1	m	123	45
8	P2	m	156	67
9	P3	f	125	67
10	P4	f	145	89
11	P5	f	12	67
12	P6	f	133	43
13	P7	m	157	90

```
pd.concat([data, datac], axis=1)
```

↳

	Name	Gender	Height	weight	Gender	Height	weight
0	P1	m	123	45	m	123	45
1	P2	m	156	67	m	156	67
2	P3	f	125	67	f	125	67
3	P4	f	145	89	f	145	89
4	P5	f	12	67	f	12	67
5	P6	f	133	43	f	133	43
6	P7	m	157	90	m	157	90

```
data.drop(["weight"], inplace=True, axis=1)
```

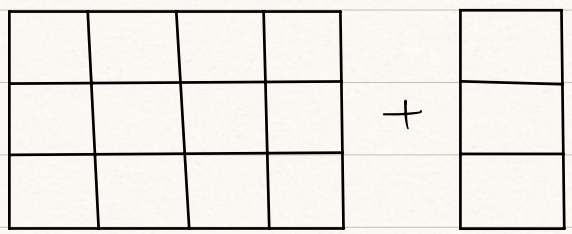
pd.concat([data, datac], ignore_index=True)

↳

	Name	Gender	Height	weight
0	P1	m	123	NAN
1	P2	m	156	NAN
2	P3	f	125	NAN
3	P4	f	145	NAN
4	P5	f	12	NAN
5	P6	f	133	NAN
6	P7	m	157	NAN
7	P1	m	123	45
8	P2	m	156	67
9	P3	f	125	67
10	P4	f	145	89
11	P5	f	12	67
12	P6	f	133	43
13	P7	m	157	90

if we want data weight to data

Since we are combining DataFrame with series we do combine row wise ... axis = 1



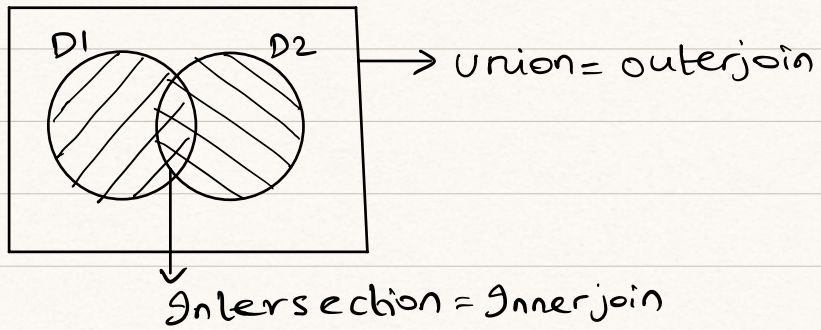
Pd.concat([data, datac["weight"]], axis=1)

	Name	Gender	Height	weight
0	P1	m	123	45
1	P2	m	156	67
2	P3	f	125	67
3	P4	f	145	89
4	P5	f	12	67
5	P6	f	133	43
6	P7	m	157	90

Merge: we can only merge 2 DataFrames.

* Merge and concatenation are different

merging is similar to set theory



* inner join: intersection (common elements in both DataFrames)

* outer join: Union (all the elements in 2 DataFrames)

* left join: left DataFrame values and intersection

* Right join: Intersection and right DataFrame values.

```
import pandas as pd

data1 = pd.DataFrame({"state": ["tl", "kt", "ap", "up", "tn"],
                      "temp": [34, 53, 23, 45, 29],
                      "humid": [34, 54, 34, 56, 57]})

data2 = pd.DataFrame({"state": ["tl", "kt", "ap", "up", "kr"],
                      "temp": [42, 56, 41, 55, 38],
                      "humid": [24, 44, 54, 66, 67]})
```

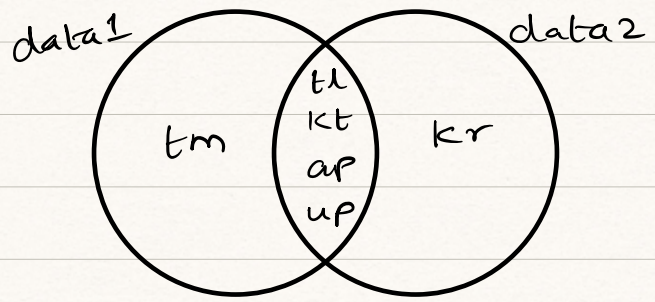
data1

	state	temp	humd
0	tl	34	34
1	kt	53	54
2	ap	23	34
3	up	45	56
4	tm	29	57

data

	state	temp	humd
0	tl	42	24
1	kt	56	44
2	ap	41	54
3	up	55	66
4	kr	38	67

* For merge we have to mention based on which column label we have to merge.
* default is inner join.
* If we are merging on state.



innerjoin: tl, kt, ap, up

outerjoin: tl, kt, ap, up, tm, kr

pd.merge(data1, data2, on="state") # default inner join

↳

	state	temp_x	humd_x	temp_x	humd_x
0	tl	34	34	42	24
1	kt	53	54	56	44
2	ap	23	34	41	54
3	up	45	56	55	66
4	tm	29	57	38	67

* 'how' means how to join, we have to mention this if we want other joins. 'outer', 'left', 'right'.

pd.merge(data1, data2, on="state", how="outer")

↳

	state	temp_x	humd_x	temp_x	humd_x
0	tl	34.0	34.0	42.0	24.0
1	kt	53.0	54.0	56.0	44.0
2	ap	23.0	34.0	41.0	54.0
3	up	45.0	56.0	55.0	66.0
4	tm	29.0	57.0	NaN	NaN
5	kr	NaN	NaN	38.0	67.0

Pd.merge(data1, data2, on="state", how="left")

↳

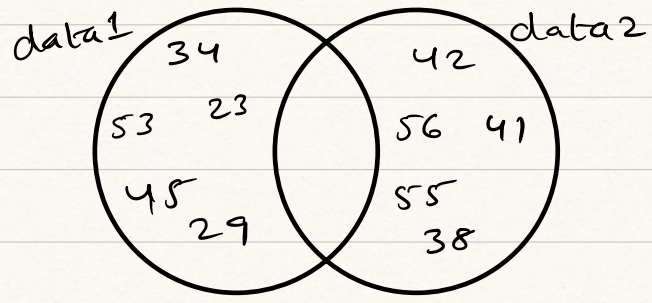
	state	temp_x	humd_x	temp_x	humd_x
0	tl	34	34	42.0	24.0
1	kt	53	54	56.0	44.0
2	ap	23	34	41.0	54.0
3	up	45	56	55.0	66.0
4	tm	29	57	NaN	NaN

Pd.merge(data1, data2, on="state", how='right')

↳

	state	temp_x	humd_x	temp_x	humd_x
0	tl	34.0	34.0	42	24
1	kt	53.0	54.0	56	44
2	ap	23.0	34.0	41	54
3	up	45.0	56.0	55	66
4	kr	NaN	NaN	38	67

* Merge on temp



pd.merge(data1, data2, on="temp")

↳

state_x	temp	humd_x	state_y	humd_y

#empty

pd.merge(data1, data2, on="temp", how="outer")

	state_x	temp	humd_x	state_y	humd_y
0	tl	34	34.0	NaN	NaN
1	kt	53	54.0	NaN	NaN
2	ap	23	34.0	NaN	NaN
3	up	45	56.0	NaN	NaN
4	tm	29	67.0	NaN	NaN
5	NaN	42	NaN	tl	24.0
6	NaN	56	NaN	kt	44.0
7	NaN	41	NaN	ap	54.0
8	NaN	55	NaN	up	66.0
9	NaN	38	NaN	kr	67.0

* If we use indicator=True it shows where the values are present i.e left-only, both, right-only

pd.merge(data1, data2, on="state", how="outer", indicator=True)

	state	temp_x	humd_x	temp_x	humd_x	_merge
0	tl	34.0	34.0	42.0	24.0	both
1	kt	53.0	54.0	56.0	44.0	both
2	ap	23.0	34.0	41.0	54.0	both
3	up	45.0	56.0	55.0	66.0	both
4	tm	29.0	57.0	NaN	NaN	left_only
5	kr	NaN	NaN	38.0	67.0	right_only

Pivot & pivot table

* Pivot and pivot table are different

Pivot

column1	column2	values
C_1	A_1	x_1
C_2	A_2	x_2
C_1	A_3	x_3

transform my column values in to row index and column index

	A_1	A_2	A_3
C_1	x_1	NaN	x_3
C_2	NaN	x_2	NaN

Pivot we will learn in more detail in statistics

```
import pandas as pd
import numpy as np
pd.read_csv(r"C:\....\weather.csv")
```

↘

	date	city	temperature	humidity
0	5/1/2017	newyork	65	56
1	5/2/2017	newyork	66	58
2	5/3/2017	newyork	68	60
3	5/1/2017	mumbai	75	80
4	5/2/2017	mumbai	78	83

5	5/3/2017	mumbai	82	85
6	5/1/2017	beijing	80	26
7	5/2/2017	beijing	77	30
8	5/3/2017	beijing	79	35

```
df = pd.read_csv(r"c:\....\weather.csv")
```

* Here if we want to find what is temperature on a particular date we can use PIVOT.
* Date will become row index
* City will become column index.
* remaining columns will behave as values.
* Pivot() is a DataFrame function
* makes sense to use categorical data in columns also index better to use categorical data.

sub index of date is created.

```
df.pivot(index='city', columns='date')
```

	temperature			humidity		
date	5/1/2017	5/2/2017	5/3/2017	5/1/2017	5/2/2017	5/3/2017
city						
beijing	80	77	79	26	30	35
mumbai	75	78	82	80	83	85
new york	65	66	68	56	58	60

if we want to see only humidity

```
df.pivot(index='city', columns='date', values='humidity')
```

	humidity		
date	5/1/2017	5/2/2017	5/3/2017
city			
beijing	26	30	35
mumbai	80	83	85
new york	56	58	60

(or) we can change the view we want to see

```
df.pivot(index='date', columns='city')
```

	temperature			humidity		
City	beijing	mumbai	newyork	beijing	mumbai	newyork
Date						
5/1/2017	80	75	65	26	80	56
5/2/2017	77	78	66	30	83	58
5/3/2017	79	82	68	35	85	60

* **Pivot table**: It is a powerful tool to calculate, summarize and analyze data that let's you see comparisions, patterns, and trends in your data.

* **pivot_table()** function creates a spread-sheet style pivot table as DataFrame.

* Pivot-table is mainly used to summarize the data.

* Instead of actual values if we want aggregate values.

margins = True will give row wise and column wise result, by default will calculate mean.

```
df.pivot_table(index='date', columns='city',
                                    margins=True)
```

	humidity				temperature			
city	beijing	mumbai	newyork	All	beijing	mumbai	newyork	All
Date								
5/1/2017	26.0...	80.0...	56	54.0	80.0...	75.0...	65.0...	73.33...
5/2/2017	30.0...	83.0...	58	57.0	77.0...	78.0...	66.0...	73.66...
5/3/2017	38.0...	85.0...	60	60.0	79.0...	82.0...	68.0...	76.33...
All	30.33...	82.66...	58	57.0	78.66...	78.33...	66.33...	74.44...

we can tell it to apply different aggregations

we can use name for margins

```
df.pivot_table(index='date', columns='city', margins=True,
                    margins_name=sum, aggfunc=np.sum)
```

	humidity				temperature			
City / Date	beijing	mumbai	newyork	sum	beijing	mumbai	newyork	sum
5/1/2017	26	80	56	162	80	75	65	220
5/2/2017	30	83	58	171	77	78	66	221
5/3/2017	35	85	60	180	79	82	68	229
Sum	91	248	174	513	236	235	199	670

we can apply different aggregations to different numerical data.

Jupyter notebook giving errors when using one aggregation as std

```
df.pivot_table(index='date', columns='city', margins=True,
               margins_name=sum, aggfunc={"humidity":
               np.mean, "temperature": np.median})
```

	humidity				temperature			
City / Date	beijing	mumbai	newyork	All	beijing	mumbai	newyork	All
5/1/2017	26.0...	80.0...	56	54.0	80	75	65	75.0
5/2/2017	30.0...	83.0...	58	57.0	77	78	66	77.0
5/3/2017	35.0...	85.0...	60	60.0	79	82	68	79.0
All	30.33...	82.66...	58	57.0	79	79	66	77.0

Crosstab() function is used to compute a simple cross tabulation of two (or more) factors.

* if we are given data like

Gender	Color
m	B
m	B
f	W
f	B
m	W

Cross tab applies 'and' operator

	B	W
m	2	1
f	1	1

↳ these are not actual values but counts.

```
import pandas as pd
import numpy as np
pd.read_excel(r"C:\....\survey.xls")
```

↘

	Name	Nationality	sex	Age	Handedness
0	Kathy	USA	Female	23	Right
1	Linda	USA	Female	18	Right
2	Peter	USA	Male	19	Right
3	John	USA	Male	22	Left
4	Fatima	Bangladesh	Female	31	Left
5	Kadir	Bangladesh	Male	25	Left
6	Dhaval	India	Male	35	Left
7	Sudhir	India	Male	31	Left
8	Parvir	India	Male	37	Right
9	Yan	china	Female	52	Right
10	Juan	china	Female	58	Left
11	Liang	china	Female	43	Left

```
df = pd.read_excel(r"C:\....\survey.xls")
```

* Crosstab is visualization concept when we are having category vs category
* x-axis and y-axis should have Categorical data

* result will be 'and' operator so we will get count

pd.crosstab(df.Nationality, df.Handedness)

↳

Handedness / Nationality	Left	Right
Bangladesh	2	0
China	2	1
India	2	1
USA	1	3

from this we can summarize that

- In Bangladesh there are 2 left handed people and 0 right handed people.
- In china there are 2 left handed people and 1- right handed person.
- In India there are 2 left handed people and 1 right handed person.
- In USA there are 1 left handed person and 3 right handed people.

```
pd.crosstab(df.Sex, df.Handedness)
```

↘

Handedness / Sex	left	Right
Female	2	3
male	5	2

from this we can summarize that
- Females are more right handed
- male are more left handed

we can add margins = True, we will see rowwise and column wise margins

```
pd.crosstab(df.Sex, df.Handedness, margins=True)
```

↘

Handedness / Sex	left	Right	All
Female	2	3	5
male	5	2	7
All	7	5	12

we can have multiple index columns and rows.

```
pd.crosstab(df.sex, [df.Handedness, df.Nationality],
                                    margins = True)
```

Handedness	Left				Right			All
Nationality	Bangladesh	China	India	USA	China	India	USA	
sex								
Female	1	1	0	0	1	0	2	5
male	1	1	2	1	0	1	1	7
All	2	2	2	1	1	1	3	12

(or)

```
pd.crosstab([df.sex, df.Handedness], df.nationality,
                margins = True, margins_name = "total")
```

	Nationality	Bangladesh	China	India	USA	total
Sex	Handedness					
Female	Left	1	1	0	0	2
	Right	0	1	0	2	3
male	left	1	1	2	1	5
	Right	0	0	1	1	2
total		2	3	3	4	12