In [4]:

```
import numpy as np
from numpy.random import randn as rn
matrix_data = rn(5,4)
matrix_data
```

Out[4]:

```
array([[-0.81402692, -1.76387173,  0.74725681, -1.25133964],
       [ 1.37042734,  0.29245082,  0.51618986, -0.3729342 ],
       [-1.00808916, -0.69457816,  0.4792719 , -0.0079622 ],
       [ 0.97342465,  0.68666135,  2.05929565,  1.2634105 ],
       [-0.16767168, -1.04435569,  0.76022729,  0.38345581]])
```

- Eatch time we try to execute the random function we get the random data
- If you want get the same random data even after executing multiple times we use seed()
- If two people are working on random numbers and both of them need the same data then we use seed(). But we need to mention the same seed(the input)

In [7]:

```
np.random.seed(234242)
matrix_data = rn(5,4)
matrix_data
```

Out[7]:

```
array([[ 1.13718793,  0.34355078, -0.19577051,  0.7806335 ],
       [ 0.12762996,  1.42339834, -1.60679063, -0.4874914 ],
       [-1.20125855, -1.05156737,  0.4346316 , -1.19755624],
       [ 0.08169773, -1.27286313,  1.06221613, -0.04593495],
       [ 0.25564507,  0.23008963, -0.30183767,  0.30722135]])
```

- We can even try to convert the data into a DataFrame
- we can even give our own row names and column names

In [16]:

```
import pandas as pd
matrix_data = rn(5,4)
row_labels = ['A','B','C','D','E']
column_headings = ['W','X','Y','Z']

df = pd.DataFrame(matrix_data,row_labels,column_headings)
df
```

Out[16]:

	W	X	Y	Z
A	0.573827	1.017798	-0.667962	-0.564121
B	-0.224738	1.849096	0.549548	0.483855
C	-0.730735	0.712397	-0.027467	-1.075719
D	-1.666618	-1.546801	0.891393	-1.413843
E	1.797954	-1.692381	0.485253	0.781150

- **Single row or single column is always treated as Series**
- df.loc['E']-->This will take named index value and returns the data ## The main difference between loc and iloc is iloc will take default index where as loc will take named index

iloc -------->default index

loc ---------> named index

In [18]:

```
df.loc['E']
```

Out[18]:

```
W    1.797954
X   -1.692381
Y    0.485253
Z    0.781150
Name: E, dtype: float64
```

In [19]:

```
#Now we see a retriving data in a combination of rows and columns
```

```
df.iloc[[4,1,2],[1]]
#Here first parameter is always treated for rows and second parameter is always for columns
```

Out[19]:

	X
E	-1.692381
B	1.849096
C	0.712397

In [20]:

```
#If we want data from intersection of rows and columns we do it as follows
df.iloc[[1,2],[1,2]]
# We cannot do the samething with loc because it doesnt understand it
```

Out[20]:

	X	Y
B	1.849096	0.549548
C	0.712397	-0.027467

In [21]:

```
# we have to give the indices based on data
df.loc[['B','C'],['W','X']]
```

Out[21]:

	W	X
B	-0.224738	1.849096
C	-0.730735	0.712397

In [22]:

```
#taking data from 4 corners using iloc
df.iloc[[0,4],[0,3]]
```

Out[22]:

	W	Z
A	0.573827	-0.564121
E	1.797954	0.781150

In [27]:

```
print("\n A column is created by assigning it in a relation to existing columns\n",'-'*75,sep='')
df['New'] = df['X']+df['Z']
df['New(Sum of Xand Z)'] = df['X']+df['Z']
print(df)
```

```
 A column is created by assigning it in a relation to existing columns
---------------------------------------------------------------------------
          W         X         Y         Z       New  New(Sum of Xand Z)
A  0.573827  1.017798 -0.667962 -0.564121  0.453676            0.453676
B -0.224738  1.849096  0.549548  0.483855  2.332951            2.332951
C -0.730735  0.712397 -0.027467 -1.075719 -0.363321           -0.363321
D -1.666618 -1.546801  0.891393 -1.413843 -2.960644           -2.960644
E  1.797954 -1.692381  0.485253  0.781150 -0.911231           -0.911231
```

In [29]:

```
#if you want to drop any row or column we do it as shown below
# We need to define the axis when we are dropping the row or column
#axis =0 ----------> row
#axis=1 -----------> column
df.drop('New',axis=1)
```

Out[29]:

	W	X	Y	Z	New(Sum of Xand Z)
A	0.573827	1.017798	-0.667962	-0.564121	0.453676
B	-0.224738	1.849096	0.549548	0.483855	2.332951
C	-0.730735	0.712397	-0.027467	-1.075719	-0.363321
D	-1.666618	-1.546801	0.891393	-1.413843	-2.960644
E	1.797954	-1.692381	0.485253	0.781150	-0.911231

- Even after dropping the column using drop() and axis it doesn't get stored in dataframe.

- Even after dropping the column using drop() and axis it doesn't get stored in dataframe.
- In order to store it back in a same dataframe we need to give one more parameter called **inplace=True** which is **False** by default
- Now data gets dropped from original dataframe

In [31]:

```
from numpy.random import randn as rn
np.random.seed(101)
matrix_data = rn(5,4)
row_labels = [12,23,34,45,56]
column_headings=['W','X','Y','Z']

df = pd.DataFrame(matrix_data,row_labels,column_headings)
df
```

Out[31]:

	W	X	Y	Z
12	2.706850	0.628133	0.907969	0.503826
23	0.651118	-0.319318	-0.848077	0.605965
34	-2.018168	0.740122	0.528813	-0.589001
45	0.188695	-0.758872	-0.933237	0.955057
56	0.190794	1.978757	2.605967	0.683509

In [32]:

```
#if you want to drop the row you need not give axis as it is by default 0
df.drop(56,inplace=True)
df
```

Out[32]:

	W	X	Y	Z
12	2.706850	0.628133	0.907969	0.503826
23	0.651118	-0.319318	-0.848077	0.605965
34	-2.018168	0.740122	0.528813	-0.589001
45	0.188695	-0.758872	-0.933237	0.955057

In [33]:

```
#drop will not understand default index, It will understand only named index
#incase if you dont want to use inplace=True then we can reassign it back to same variable
```

In [36]:

```
df.loc[[12,23,34]]>0
```

Out[36]:

	W	X	Y	Z
12	True	True	True	True
23	True	False	False	True
34	False	True	True	False

In [37]:

```
#Where the condition is false and then we try to print the data again we get NaN in place of False
#THis is conditional filtering data
df[df.loc[[12,23,34]]>0]
```

Out[37]:

	W	X	Y	Z
12	2.706850	0.628133	0.907969	0.503826
23	0.651118	NaN	NaN	0.605965
34	NaN	0.740122	0.528813	NaN
45	NaN	NaN	NaN	NaN

In [38]:

```
import pandas as pd
import numpy as np
matrix_data = np.matrix('22,66,140;42,70,148;30,62,125;35,68,160;25,62,152')
row_labels = ['A','B','C','D','E']
column_headings = ['Age','Height','Weight']
matrix_data
```

Out[38]:

```
matrix([[ 22,  66, 140],
        [ 42,  70, 148],
        [ 30,  62, 125],
        [ 35,  68, 160],
        [ 25,  62, 152]])
```

In [39]:

```python
df = pd.DataFrame(data = matrix_data,index=row_labels,columns = column_headings)
```

In [40]:

```python
df
```

Out[40]:

	Age	Height	Weight
A	22	66	140
B	42	70	148
C	30	62	125
D	35	68	160
E	25	62	152

In [41]:

```python
df['Height']
```

Out[41]:

```
A    66
B    70
C    62
D    68
E    62
Name: Height, dtype: int64
```

In [43]:

```python
df[df['Height']<65]
```

Out[43]:

	Age	Height	Weight
C	30	62	125
E	25	62	152

In [48]:

```python
#Age>30 and Height<65 and Weight>125
```

In [47]:

```python
df[(df['Age']>30) & (df['Height']>65) & (df['Weight']>125)]
```

Out[47]:

	Age	Height	Weight
B	42	70	148
D	35	68	160

In [49]:

```python
df
```

Out[49]:

	Age	Height	Weight
A	22	66	140
B	42	70	148
C	30	62	125
D	35	68	160
E	25	62	152

- If you don't want the custom indexes and want the default indices we do **reset_index()**

In [50]:

```python
df.reset_index()
```

Out[50]:

	index	Age	Height	Weight
0	A	22	66	140

1	B	42	70	148
2	C	30	62	125
3	D	35	68	160
4	E	25	62	152

- After reset_index() we still be able to see old indices.
- Inorder not to see the old indices upon **reset_index()** we use **drop=True** as a parameter

In [51]:

```
df.reset_index(drop=True)
```

Out[51]:

	Age	Height	Weight
0	22	66	140
1	42	70	148
2	30	62	125
3	35	68	160
4	25	62	152

- Inorder to make the changes permanently we use inplace = True

In [55]:

```
#when you want to create a new column for the given dataframe
df['xyz']="Student Teacher Engineer Doctor Nurse".split()
df
```

Out[55]:

	Age	Height	Weight	xyz
A	22	66	140	Student
B	42	70	148	Teacher
C	30	62	125	Engineer
D	35	68	160	Doctor
E	25	62	152	Nurse

In [56]:

```
#If you want to set the new column as index
df.set_index('xyz')
```

Out[56]:

	Age	Height	Weight
xyz			
Student	22	66	140
Teacher	42	70	148
Engineer	30	62	125
Doctor	35	68	160
Nurse	25	62	152

In [65]:

```
#Multi-Indexing
#index levels
outside = ['G1','G1','G1','G2','G2','G2']
inside = [1,2,3,1,2,3]

hier_index= list(zip(outside,inside))
```

In [66]:

```
hier_index
```

Out[66]:

```
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2), ('G2', 3)]
```

In [67]:

```
hier_index = pd.MultiIndex.from_tuples(hier_index)
print("\nIndex hirearchy\n",'-'*25,sep=' ')
print(hier_index)
```

```
Index hirearchy
 -------------------------
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

In [ ]:

```

```

In [69]:

```
print("\nCreating a DataFrame with multi-index\n",'-'*35,sep='')
df1 = pd.DataFrame(data=np.round(rn(6,3)), index = hier_index, columns=['A','B','C'])
print(df1)
```

```
Creating a DataFrame with multi-index
-----------------------------------
        A    B    C
G1 1  0.0  2.0 -2.0
   2 -1.0 -0.0  0.0
   3  0.0  0.0  1.0
G2 1  0.0  1.0  0.0
   2 -0.0 -1.0 -1.0
   3  0.0 -0.0  2.0
```

In [70]:

```
df1.loc['G1']
```

Out[70]:

	A	B	C
1	0.0	2.0	-2.0
2	-1.0	-0.0	0.0
3	0.0	0.0	1.0

In [71]:

```
#inorder to access the data present in 3rd row of G1 then we do it as follows
df1.loc['G1'].loc[3,['B','C']]
```

Out[71]:

```
B    0.0
C    1.0
Name: 3, dtype: float64
```

In [73]:

```
df1.loc['G1'].iloc[2,[1,2]]
```

Out[73]:

```
B    0.0
C    1.0
Name: 3, dtype: float64
```

TASK

In [76]:

```
l1=['A','A','A','B','B','B','C','C','C']
l2 = [1,2,3,1,2,3,1,2,3]
l3 = [1,2,3,4,5,6,7,8,9]
y=list(zip(l1,l2,l3))
g = pd.MultiIndex.from_tuples(y)
data_f = pd.DataFrame(data=np.round(rn(9,3)),index=g,columns=['a','b','c'])
data_f
```

Out[76]:

			a	b	c
A	**1**	**1**	1.0	-0.0	2.0
	2	**2**	-1.0	-1.0	0.0
	3	**3**	-1.0	-1.0	1.0
B	**1**	**4**	1.0	-2.0	1.0

```
     2  5   1.0  -1.0   1.0
     3  6  -1.0  -0.0  -0.0
C    1  7   0.0   0.0   0.0
     2  8  -1.0   2.0   1.0
     3  9  -1.0  -1.0   1.0
```

In [78]:

```
df = pd.DataFrame({'A':[1,2,np.nan],'B':[5,np.nan,np.nan],'C':[1,2,3]})
df['States'] = "CA NV AZ".split()
df.set_index('States',inplace=True)
df
```

Out[78]:

```
          A     B   C
States
    CA  1.0   5.0   1
    NV  2.0   NaN   2
    AZ  NaN   NaN   3
```

In [79]:

```
#dropping the rows that has NaN . Even if we have a single NaN it will drop from the DataFrame
df.dropna()
```

Out[79]:

```
          A    B  C
States
    CA  1.0  5.0  1
```

In [80]:

```
#drop columns with NaN
df.dropna(axis=1)
```

Out[80]:

```
        C
States
    CA  1
    NV  2
    AZ  3
```

In [81]:

```
#we can even set the threashold for the NaN values.
df.dropna(thresh = 2)
# This means we want minimum that many no.of non-NaN values
#This will search default by rows
```

Out[81]:

```
          A     B  C
States
    CA  1.0   5.0  1
    NV  2.0   NaN  2
```

In [ ]: