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

In [5]:
```python
# Create a vector as column
vector_column = np.array([[1], [2], [3]])
vector_column
```

Out[5]:
```
array([[1],
       [2],
       [3]])
```

In [6]:
```python
# Create a vector as row
vector_row = np.array([[1,2,3]])
vector_row
```

Out[6]:
```
array([[1, 2, 3]])
```

In [7]:
```python
from scipy import sparse
```

In [8]:
```python
# Create a Matrix
matrix = np.array([[0,0],
                   [0,1],
                   [3,0]])
# Create compressed sparse row (CSR) matrix
matrix_sparse = sparse.csr_matrix(matrix)
```

In [9]:
```python
matrix
```

Out[9]:
```
array([[0, 0],
       [0, 1],
       [3, 0]])
```

In [10]:
```python
print(matrix_sparse)
```

```
  (1, 1)	1
  (2, 0)	3
```

In [14]:
```python
# Create larger matrix
matrix_large = np.array([[0,0,0,0,0,0,0,0,0], [0,0,1,0,10,0,8,0,7], [0,2,0,0,0,5,0,0,6]])
```

In [16]:
```python
# Create compressed sparse row (CSR) matrix
matrix_large_sparse = sparse.csr_matrix(matrix_large)
# View larger sparse materix
print(matrix_large_sparse)
```

```
  (1, 2)	1
  (1, 4)	10
  (1, 6)	8
  (1, 8)	7
  (2, 1)	2
  (2, 5)	5
  (2, 8)	6
```

In [17]:
```python
# Create 3x3 Matrix
matrix = np.array([[1,2,3], [4,5,6], [7,8,9]])
matrix
```

Out[17]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [18]:
```python
# Find Maximum element in column
np.max(matrix, axis=0)
```

Out[18]:
```
array([7, 8, 9])
```

In [19]:
```python
# Find Maximum element in row
np.max(matrix, axis=1)
```

Out[19]:
```
array([3, 6, 9])
```

In [20]:
```python
# Return Mean
np.mean(matrix)
```

Out[20]:
```
5.0
```

In [21]:
```python
# Returns Variance
np.var(matrix)
```

Out[21]:
```
6.666666666666667
```

In [22]:
```python
#  Returns Standard Deviation
np.std(matrix)
```

Out[22]:
```
2.581988897471611
```

In [ ]:
```python
# RESHAPING ARRAY
```

In [23]:
```python
# Create 4x3 matrix
matrix = np.array([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
matrix
```

Out[23]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [24]:
```python
# Reshape matrix into 2x6 matrix
matrix.reshape(2, 6)
```

Out[24]:
```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

In [25]:
```python
matrix.reshape(1, -1)
#  to get the array in the form of 1D
```

Out[25]:
```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

In [26]:
```python
matrix.size
```

Out[26]:
```
12
```

In [27]:
```python
matrix.reshape(matrix.size)
# it will return a single list of all elements in the matrix
```

Out[27]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [28]:
```python
# Transpose Matrix
matrix.T
```

Out[28]:
```
array([[ 1,  4,  7, 10],
       [ 2,  5,  8, 11],
       [ 3,  6,  9, 12]])
```

In [30]:
```python
# Transpose Vector
np.array([1, 2, 3, 4, 5, 6]).T
```

Out[30]:
```
array([1, 2, 3, 4, 5, 6])
```

In [31]:
```python
# Transpose row vector
np.array([[1, 2, 3, 4, 5, 6]]).T
```

Out[31]:
```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

In [33]:
```python
# Flatten Matrix
matrix.flatten()
# To convert the matrix into 1D array
```

Out[33]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [34]:
```python
# Return Diagonal elements
matrix.diagonal()
```

Out[34]:
```
array([1, 5, 9])
```

In [35]:
```python
c = np.array([1, 2, np.nan, 3, 4])
c
# nan - not a number
```

Out[35]:
```
array([ 1.,  2., nan,  3.,  4.])
```

In [36]:
```python
np.isnan(c)
```

Out[36]:
```
array([False, False,  True, False, False])
```

In [37]:
```python
np.mean(c[~np.isnan(c)])
# ~np.isnan = means is not a nan
```

Out[37]:
```
2.5
```

In [38]:
```python
a = np.array([[1,2,3], [4,5,6], [7,8,9]])
a
```

Out[38]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [39]:
```python
a =a.astype('float')
a
```

Out[39]:
```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

In [44]:
```python
a[0][0] = np.nan
a[1][1] = np.inf # inf - infinity
a
```

Out[44]:
```
array([[nan,  2.,  3.],
       [ 4., inf,  6.],
       [ 7.,  8.,  9.]])
```

In [45]:
```python
np.isinf(a)
# Returns true if inf value
```

Out[45]:
```
array([[False, False, False],
       [False,  True, False],
       [False, False, False]])
```

In [46]:
```python
np.isnan(a)
#  Returns true if nan value
```

Out[46]:
```
array([[ True, False, False],
       [False, False, False],
       [False, False, False]])
```

In [48]:
```python
#  To check both the condition
flag = np.isinf(a) | np.isnan(a)
flag
```

Out[48]:
```
array([[ True, False, False],
       [False,  True, False],
       [False, False, False]])
```

In [56]:
```python

```

Out[56]:
```
array([0., 0.])
```

In [61]:
```python
#DATA CLEANING
a[flag] = 0
a
```

Out[61]:
```
array([[0., 2., 3.],
       [4., 0., 6.],
       [7., 8., 9.]])
```

In [63]:
```python
dict = {'A':100, 'B':200, 'C': 300, 'D':400}
z1 = pd.Series(dict)
z1
```

Out[63]:
```
A    100
B    200
C    300
D    400
dtype: int64
```

In [65]:
```python
z2 = pd.Series([10,20,30,40],'A B D E'.split())
```

In [66]:
```python
'A B C D'.split()
```

Out[66]:
```
['A', 'B', 'C', 'D']
```

In [67]:
```python
z2
```

Out[67]:
```
A    10
B    20
D    30
E    40
dtype: int64
```

In [69]:
```python
z1+z2
```

Out[69]:
```
A    110.0
B    220.0
C      NaN
D    430.0
E      NaN
dtype: float64
```

In [ ]:
```python
# Slicing in a Series
```

In [70]:
```python
z1['A':'C']
```

Out[70]:
```
A    100
B    200
C    300
dtype: int64
```

In [71]:
```python
type(z1['A':'C'])
# We can consider Series as a single column
```

Out[71]:
```
pandas.core.series.Series
```

In [73]:
```python
z1[['A','C']]
```

Out[73]:
```
A    100
C    300
dtype: int64
```

In [74]:
```python
#  using default index values
z1[[0, 1]]
```

Out[74]:
```
A    100
B    200
dtype: int64
```

In [ ]:
```python
# INDEXING
```

In [75]:
```python
# iloc does integer based indexing
z1.iloc[2]
```

Out[75]:
```
300
```

In [76]:
```python
# loc does label based indexing
z1.loc['A']
```

Out[76]:
```
100
```

In [ ]:
```python
#OPERATIONS
```

In [77]:
```python
z1>100
```

Out[77]:
```
A    False
B     True
C     True
D     True
dtype: bool
```

In [78]:
```python
z1[z1>100]
```

Out[78]:
```
B    200
C    300
D    400
dtype: int64
```

In [79]:
```python
b = [True, False, True, False]
z1[b]
```

Out[79]:
```
A    100
C    300
dtype: int64
```

In [ ]:
```python
# BROADCASTING
```

In [80]:
```python
z1+200
```

Out[80]:
```
A    300
B    400
C    500
D    600
dtype: int64
```

In [ ]:
```python
# Ordering on Series
```

In [81]:
```python
z1.sort_values()
```

Out[81]:
```
A    100
B    200
C    300
D    400
dtype: int64
```

In [82]:
```python
z1.sort_values(ascending = False)
```

Out[82]:
```
D    400
C    300
B    200
A    100
dtype: int64
```

In [99]:
```python
z1.sort_values(ascending=False,inplace=True)
```

In [ ]:
```python
# Aggregation on Series
```

In [88]:
```python
z1.mean()
```

Out[88]:
```
250.0
```

In [89]:
```python
z1.sum()
```

Out[89]:
```
1000
```

In [90]:
```python
z1.max()
```

Out[90]:
```
400
```

In [91]:
```python
z1.min()
```

Out[91]:
```
100
```

In [93]:
```python
z1.idxmax()
```

Out[93]:
```
'D'
```

In [100…
```python
# Series with two columns
data_1 = {'Column1': pd.Series([[2,3,4,5], ['a b c d'.split()]]),
          'Column2': pd.Series([[20,30,40,50], ['A B C D'.split()]])}
data_1
```

Out[100]:
```
{'Column1': 0      [2, 3, 4, 5]
 1    [[a, b, c, d]]
 dtype: object,
 'Column2': 0    [20, 30, 40, 50]
 1      [[A, B, C, D]]
 dtype: object}
```

In [ ]:
```python
# DATA FRAME
```

In [126…
```python
rand_mat = np.random.randn(5, 4)
rand_mat
```

Out[126]:
```
array([[-0.14428909,  0.25652554,  0.35639451, -1.11163736],
       [ 0.01521384,  1.36526186,  2.11857241, -0.96544584],
       [-1.38082573, -1.12994592,  0.184108  ,  0.10230005],
       [-0.25162446, -0.66238928,  0.73263871,  0.7107713 ],
       [ 0.41370764,  0.39832704,  0.43151927, -0.44435259]])
```

In [127…
```python
df = pd.DataFrame(rand_mat, index = 'A B C D E'.split(), columns = 'W X Y Z'.split())
df
```

Out[127]:

	W	X	Y	Z
A	-0.144289	0.256526	0.356395	-1.111637
B	0.015214	1.365262	2.118572	-0.965446
C	-1.380826	-1.129946	0.184108	0.102300
D	-0.251624	-0.662389	0.732639	0.710771
E	0.413708	0.398327	0.431519	-0.444353

In [ ]:
```python
# SELECTION & INDEXING
```

In [104…
```python
df['W']
# To fetch the entire column
```

Out[104]:
```
A    0.173537
B    1.470689
C    0.197186
D   -0.873710
E   -0.159152
Name: W, dtype: float64
```

In [105…
```python
type(df['W'])
```

Out[105]:
```
pandas.core.series.Series
```

In [107…
```python
df[['W', 'Z']]
```

Out[107]:

	W	Z
A	0.173537	0.829445
B	1.470689	-1.350043
C	0.197186	0.881326
D	-0.873710	-0.120910
E	-0.159152	-1.036922

In [108…
```python
df.W
# (this way is Not Recommended)
```

Out[108]:
```
A    0.173537
B    1.470689
C    0.197186
D   -0.873710
E   -0.159152
Name: W, dtype: float64
```

In [109…
```python
df.iloc[2]
```

Out[109]:
```
W    0.197186
X   -0.335820
Y   -0.467756
Z    0.881326
Name: C, dtype: float64
```

In [130…
```python
df.loc['F'] = df.loc['A']+df.loc['B']
df
# will add new column F whose value is sum of values of A & B
```

Out[130]:

	W	X	Y	Z
A	-0.144289	0.256526	0.356395	-1.111637
B	0.015214	1.365262	2.118572	-0.965446
C	-1.380826	-1.129946	0.184108	0.102300
D	-0.251624	-0.662389	0.732639	0.710771
E	0.413708	0.398327	0.431519	-0.444353
F	-0.129075	1.621787	2.474967	-2.077083

In [131…
```python
#  Reset to default 0,1,...n index
df.reset_index()
```

Out[131]:

	index	W	X	Y	Z
0	A	-0.144289	0.256526	0.356395	-1.111637
1	B	0.015214	1.365262	2.118572	-0.965446
2	C	-1.380826	-1.129946	0.184108	0.102300
3	D	-0.251624	-0.662389	0.732639	0.710771
4	E	0.413708	0.398327	0.431519	-0.444353
5	F	-0.129075	1.621787	2.474967	-2.077083

In [132…
```python
newind = 'DEL UP UK TN AP KL'.split()
newind
#  NEW INDEX
```

Out[132]:
```
['DEL', 'UP', 'UK', 'TN', 'AP', 'KL']
```

In [133…
```python
df['States'] = newind
df
```

Out[133]:

	W	X	Y	Z	States
A	-0.144289	0.256526	0.356395	-1.111637	DEL
B	0.015214	1.365262	2.118572	-0.965446	UP
C	-1.380826	-1.129946	0.184108	0.102300	UK
D	-0.251624	-0.662389	0.732639	0.710771	TN
E	0.413708	0.398327	0.431519	-0.444353	AP
F	-0.129075	1.621787	2.474967	-2.077083	KL

In [134…
```python
df.set_index('States', inplace=True)
```

In [135…
```python
df
```

Out[135]:

	W	X	Y	Z
States				
DEL	-0.144289	0.256526	0.356395	-1.111637
UP	0.015214	1.365262	2.118572	-0.965446
UK	-1.380826	-1.129946	0.184108	0.102300
TN	-0.251624	-0.662389	0.732639	0.710771
AP	0.413708	0.398327	0.431519	-0.444353
KL	-0.129075	1.621787	2.474967	-2.077083

In [ ]:
```python
# Multi-Index and Index Hierarchy
```

In [136…
```python
# Index levels
outside = ['North', 'North', 'North', 'South', 'South', 'South']
inside = newind
```

In [137…
```python
hier_index = list(zip(outside, inside))
hier_index
```

Out[137]:
```
[('North', 'DEL'),
 ('North', 'UP'),
 ('North', 'UK'),
 ('South', 'TN'),
 ('South', 'AP'),
 ('South', 'KL')]
```

In [139…
```python
hier_index = pd.MultiIndex.from_tuples(hier_index)
hier_index
```

Out[139]:
```
MultiIndex([('North', 'DEL'),
            ('North',  'UP'),
            ('North',  'UK'),
            ('South',  'TN'),
            ('South',  'AP'),
            ('South',  'KL')],
           )
```

In [141…
```python
df = pd.DataFrame(np.random.randn(6,2), index = hier_index, columns = ['A', 'B'])
df
```

Out[141]:

		A	B
North	**DEL**	1.482350	-0.033940
	UP	0.490844	1.459586
	UK	1.306679	-0.269997
South	**TN**	0.200385	-0.011374
	AP	1.851014	-2.195063
	KL	0.389175	-0.394603

In [142…
```python
df.loc['North']
# it will fetch only North Data
```

Out[142]:

	A	B
DEL	1.482350	-0.033940
UP	0.490844	1.459586
UK	1.306679	-0.269997

In [143…
```python
df.loc['North'].loc['DEL']
# to fetch data of north and specifically of DEL
```

Out[143]:
```
A    1.48235
B   -0.03394
Name: DEL, dtype: float64
```

In [144…
```python
# Index has not assigned to any names
df.index.names
```

Out[144]:
```
FrozenList([None, None])
```

In [146…
```python
df.index.names = ['Region', 'States']
df
```

Out[146]:

		A	B
Region	**States**		
North	**DEL**	1.482350	-0.033940
	UP	0.490844	1.459586
	UK	1.306679	-0.269997
South	**TN**	0.200385	-0.011374
	AP	1.851014	-2.195063
	KL	0.389175	-0.394603

In [159…
```python
df.xs('North')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [159], in <cell line: 1>()
----> 1 df.xs('North')

TypeError: 'str' object is not callable
```