2.1 Numpy

- Numpy,which stands for numberical Python,is a Python library package to support numerical computations.
- The basic data structure in numpy is *multi-dimensional array* object called *nadrray* .
- Numpy provides a suite of functions that can efficiently manipulate elements of the ndarray.

2.1.1 Creating ndarray

An ndarray can be created from a list or tuple object.

Function

- array():create the ndarray
- random.rand():random n numbers from a uniform distribution between [0,1],parameter is the quality.
- random.randn():random n numbers from a normal distribution,parameter is the quality.
- arange():similar to range, but returns ndarray instead of list
- reshape() : reshape to a [m x n] matrix
- linspace(): split interval [1st. parameter, 2nd. parameter] into 3rd. parameter equally separated values
- logspace():create ndarray with values from 10^1st. parameter to 10^2nd. parameter. the 3rd. parameter is the quantity.
- zeros(): a matrix of 0, parameter the (n,m) means its shape.
- ones():a matrix of 1,parameter the (n,m)means its shape.
- eye(): a [n x n] identity matrix.

Attribute

- ndim : the dimensionality of the nadrray
- shape : the shape of the nadrray, (1st.,2nd.): 1st. - rows and 2nd. - columns
- size : the size of the nadrray
- dtype : the data type of the nadrray

```python
import numpy as np

# a 1 - dimensional array (vector)
oneDim = np.array([1.0,2,3,4,5])
print(oneDim)
print("#Dimensions = ",oneDim.ndim) # print the dimensionality
print("Dimension = ", oneDim.shape) # print the shape
```

```python
print("Size = ",oneDim.size) # print the size
print("Array type = ", oneDim.dtype) # print the data type

# a two-dimensional array (matrix)
twoDim = np.array([[1,2],[3,4],[5,6],[7,8]])
print(twoDim)
print("#Dimensions = ", twoDim.ndim)
print("Dimension = ", twoDim.shape)
print("Size = ", twoDim.size)
print("Array type = ", oneDim.dtype)

# create ndarray from tuple
arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)])
print(arrFromTuple)
print("#Dimensions = ",arrFromTuple.ndim)
print("Dimension = ",arrFromTuple.shape)
print("Size = ",arrFromTuple.size)
```

```
[1. 2. 3. 4. 5.]
#Dimensions =  1
Dimension =  (5,)
Size =  5
Array type =  float64
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
#Dimensions =  2
Dimension =  (4, 2)
Size =  8
Array type =  float64
[['1' 'a' '3.0']
 ['2' 'b' '3.5']]
#Dimensions =  2
Dimension =  (2, 3)
Size =  6
```

```python
print(np.random.rand(5)) # random numbers from a uniform distribution between [0,1]
print(np.random.randn(5)) # random numbers from a normal distribution
print(np.arange(-10,10,2)) # similar to range, but returns ndarray instead of list
print(np.arange(12).reshape(3,4)) # reshape to a [3x4] matrix
print(np.linspace(0,1,10)) # split interval [0,1] into 10 equally separated values
print(np.logspace(-3,3,7)) # create ndarray with values from 10^-3 to 10^3
```

```
[0.06390371 0.65628429 0.01983321 0.04899113 0.65451714]
[ 0.4043999  -2.28160158 -1.21103859 -1.49661679  0.80819921]
[-10  -8  -6  -4  -2   0   2   4   6   8]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02 1.e+03]
```

```
print(np.zeros((2,3))) # a matrix of zeros
print(np.ones((3,2))) # a matrix of ones
print(np.eye(3)) # a 3 x 3 identity matrix
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2.1.2 Element-wise Operations

You can apply standard operators such as addition and multiplication on each element of the ndarray.

```
x = np.array([1,2,3,4,5])

print(x + 1) # addition
print(x - 1) # subtraction
print(x * 2) # multiplication
print(x // 2) # integer division
print(x ** 2) # square
print(x % 2) # modulo
print(1 / x) # division
```

```
[2 3 4 5 6]
[0 1 2 3 4]
[ 2  4  6  8 10]
[0 1 1 2 2]
[ 1  4  9 16 25]
[1 0 1 0 1]
[1.         0.5        0.33333333 0.25       0.2       ]
```

```python
x = np.array([2,4,6,8,10])
y = np.array([1,2,3,4,5])

print(x + y)
print(x - y)
print(x * y)
print(x / y)
print(x // y)
print(x ** y)
```

```
[ 3  6  9 12 15]
[1 2 3 4 5]
[ 2  8 18 32 50]
[2. 2. 2. 2. 2.]
[2 2 2 2 2]
[     2     16    216   4096 100000]
```

2.1.3 Indexing and Slicing

There are various ways to select certain elements with an ndarray.

- copy() : makes a copy of the subarray, when it changed, original array will not change.
- Note: without copy() function,the slice list changed,original array will change. Example in y = x[3:5] and z = x[3:5].copy()

```python
x = np.arange(-5,5)
print(x)

y = x[3:5] # y is a slice, i.e., pointer to a subarray in x
print(y)

y[:] = 1000 # modifying the value of y will change x
print(y)
print(x)

z = x[3:5].copy() # makes a copy of the subarray
print(z)

z[:] = 500 # modifying the value of z will not affect x
print(z)
print(x)
```

```
[-5 -4 -3 -2 -1  0  1  2  3  4]
[-2 -1]
[1000 1000]
[  -5   -4   -3 1000 1000    0    1    2    3    4]
[1000 1000]
[500 500]
[  -5   -4   -3 1000 1000    0    1    2    3    4]
```

```python
my2dlist = [[1,2,3,4],[5,6,7,8],[9,10,11,12]] # a 2-dim list
print(my2dlist)
print(my2dlist[2]) # access the third sublist
print(my2dlist[:][2]) # can't access third element of each sublist
# print(my2dlist[:,2]) # this will cause syntax error

my2darr = np.array(my2dlist)
print(my2darr)
print(my2darr[2][:]) # access the third row
print(my2darr[2,:]) # access the third row
print(my2darr[:][2]) # access the third row (similar to 2d list)
print(my2darr[:,2]) # access the third column
print(my2darr[:2,2:]) # access the first two rows & last two columns
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[9, 10, 11, 12]
[9, 10, 11, 12]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 9 10 11 12]
[ 9 10 11 12]
[ 9 10 11 12]
[ 3  7 11]
[[3 4]
 [7 8]]
```

Boolean Indexing

```python
my2darr = np.arange(1,13,1).reshape(3,4)
print(my2darr)

divBy3 = my2darr[my2darr % 3 == 0] # print the element can be divided 3 in my2darr
print(divBy3, type(divBy3))

divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0] # print the element can be divided 3 in 2nd. my2darr row
print(divBy3LastRow)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 3  6  9 12] <class 'numpy.ndarray'>
[[ 9 12]]
```

```
my2darr = np.arange(1,13,1).reshape(4,3)
print(my2darr)

indices = [2,1,0,3] # selected row indices
print(my2darr[indices,:])

rowIndex = [0,0,1,2,3] # row index into my2darr
columnIndex = [0,2,0,1,2] # column index into my2darr
print(my2darr[rowIndex,columnIndex])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 7  8  9]
 [ 4  5  6]
 [ 1  2  3]
 [10 11 12]]
[ 1  3  4  8 12]
```

2.1.4 Numpy Arthmetic and Statistical Functions

There are many built-in mathematical functions available for manipulating elements of nd-array.

Function

Single ndarray parameter

- abs() : convert to absolute values
- sqrt(): apply square root to each element
- sing(): get the sign of each element
- exe() : apply exponentiation
- sort(): sort array
- min() : minmum element in ndarray
- max() : maxmum element in ndarray
- mean(): get the mean of ndarray
- std() : standard deviation
- sum() : get the sum of ndarray

Two ndarray parameters

- add() : element - wise addition x + y
- subtract() : element - wise subtraction x - y
- multiply() : element - wise multiplication x * y
- divide() : element - wise division x / y
- maximum() : element - wise maximum max(x,y)

```
y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4]) # generate a random vector
print(y)

print(np.abs(y)) # convert to absolute values
print(np.sqrt(abs(y))) # apply square root to each element
print(np.sign(y)) # get the sign of each element
print(np.exp(y)) # apply exponentiation
print(np.sort(y)) # sort array
```

```
[-1.4  0.4 -3.2  2.5  3.4]
[1.4 0.4 3.2 2.5 3.4]
[1.18321596 0.63245553 1.78885438 1.58113883 1.84390889]
[-1.  1. -1.  1.  1.]
[ 0.24659696  1.4918247   0.0407622  12.18249396 29.96410005]
[-3.2 -1.4  0.4  2.5  3.4]
```

```
x = np.arange(-2,3)
y = np.random.randn(5)
print(x)
print(y)

print(np.add(x,y)) # element-wise addition x + y
print(np.subtract(x,y)) # element-wise subtraction x - y
print(np.multiply(x,y)) # element-wise multiplication x * y
print(np.divide(x,y)) # element-wise division x / y
print(np.maximum(x,y)) # element-wise maximum max(x,y)
```

```
[-2 -1  0  1  2]
[-0.62238064  0.06803966 -0.49192638  2.94373694  0.97320746]
[-2.62238064 -0.93196034 -0.49192638  3.94373694  2.97320746]
[-1.37761936 -1.06803966  0.49192638 -1.94373694  1.02679254]
[ 1.24476127 -0.06803966 -0.          2.94373694  1.94641491]
[  3.21346758 -14.69731118  -0.           0.33970427   2.05506029]
[-0.62238064  0.06803966  0.          2.94373694  2.        ]
```

```
y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4]) # generate a random vector
print(y)

print("Min =", np.min(y)) # min
print("Max =", np.max(y)) # max
print("Average =", np.mean(y)) # mean/average
print("Std deviation =", np.std(y)) # standard deviation
print("Sum =", np.sum(y)) # sum
```

```
[-3.2 -1.4  0.4  2.5  3.4]
Min = -3.2
Max = 3.4
Average = 0.34000000000000014
Std deviation = 2.432776191925595
Sum = 1.7000000000000006
```

2.1.5 Numpy Linear Algebra

Numpy provides many functions to support linear algebra operations.

Function

- dot() : matrix-vector multiplication
- linalg.inv() : inverse of a square matrix
- linalg.det() : determinant of a square matrix
- linalg.eig() : eigenvalue and eigenvector of a square matrix

Attribute

- T : matrix transpose operation X^T

```
X = np.random.randn(2,3) # create a 2 x 3 random matrix
print(X)
print(X.T) # matrix transpose operation X^T

y = np.random.randn(3) # random vector
print(y)
print(X.dot(y)) # matrix-vector multiplication X * y
print(X.dot(X.T)) # matrix-matrix multiplication X * X^T
print(X.T.dot(X)) # matrix-matrix multiplication X^T * X
```

```
[[ 0.30466078 -1.32182636  0.23663167]
 [-0.56404753  0.7886946   0.21350702]]
[[ 0.30466078 -0.56404753]
 [-1.32182636  0.7886946 ]
 [ 0.23663167  0.21350702]]
[-0.2024012  -0.74751189  0.24756148]
[ 0.98499809 -0.42253858]
[[ 1.89603765 -1.16383795]
 [-1.16383795  0.98577404]]
[[ 0.41096781 -0.84756989 -0.04833572]
 [-0.84756989  2.36926409 -0.14439414]
 [-0.04833572 -0.14439414  0.10157979]]
```

```
X = np.random.randn(5,3)
print(X)

```

```
C = X.T.dot(X) # C = X^T * X is a square matrix

invC = np.linalg.inv(C) # inverse of a square matrix
print(invC)

detC = np.linalg.det(C) # determinant of a square matrix
print(detC)

S,U = np.linalg.eig(C) # eigenvalue S and eigenvector U of a square matrix
print(S)
print(U)
```

```
[[-0.40091206  1.88245093  0.96576246]
 [ 0.47607027  0.77646112 -0.3880549 ]
 [-0.41763    -1.60759296  0.09800108]
 [ 0.45919206  0.31563008 -0.48303597]
 [ 0.44783751  0.4741899   0.88965723]]
[[ 1.39248398 -0.23494643  0.46752532]
 [-0.23494643  0.21196706 -0.21140471]
 [ 0.46752532 -0.21140471  0.73109281]]
8.825194646750928
[0.5890574  1.97625887 7.5809363 ]
[[-0.86352887  0.49872157  0.07479766]
 [ 0.20229439  0.20670059  0.95726268]
 [-0.46194682 -0.84175511  0.2793805 ]]
```

2.2 Pandas

- Pandas provide two convenient data structures for storing and manipulating data -- Series and DataFrame.
- Series is similar to a one-dimensional array
- DataFrame is more similar to representing a matrix or a spreadsheet table.

2.2.1 Series

- A Series object consists of a one-dimensional array of values, whose elements can be referenced using anindex array.
- A Series object can be created from a list, a numpy array, or a Python dictionary. You can applymost of the numpy functions on the Series object

Define

- Series():
  - list or numpy array :
    - 1st. parameter list : values of the Series
    - 2st. parameter index list : indexs of the Series, default index begins with 0
  - python dictionary :
    - key : index of the Series
    - values : values of the Series

Attribute

- values : display values of the Series
- index : display indices of the Series
- shape : display the shape of the Series
- size : display the size of the Series

(1) Create Series

```
from pandas import Series

# creating a series from a list
s = Series([3.1, 2.4, -1.7, 0.2, -2.9, 4.5])
print(s)

print('Values=', s.values) # display values of the Series
print('Index=', s.index) # display indices of the Series
```

```
0    3.1
1    2.4
2   -1.7
3    0.2
4   -2.9
5    4.5
dtype: float64
Values= [ 3.1  2.4 -1.7  0.2 -2.9  4.5]
Index= RangeIndex(start=0, stop=6, step=1)
```

```
import numpy as np

# creating a series from a numpy ndarray
s2 = Series(np.random.randn(6))

print(s2)
print('Values=', s2.values) # display values of the Series
print('Index=', s2.index) # display indices of the Series
```

```
0   -1.274029
1   -0.489672
2   -0.720381
3   -0.031721
4   -0.980086
5    0.654080
dtype: float64
Values= [-1.27402889 -0.48967244 -0.7203815  -0.0317211  -0.9800861   0.65408028]
Index= RangeIndex(start=0, stop=6, step=1)
```

```
# creating a series from list
s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
 index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])

print(s3)
print('Values=', s3.values) # display values of the Series
print('Index=', s3.index) # display indices of the Series
```

```
Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
dtype: float64
Values= [ 1.2  2.5 -2.2  3.1 -0.8 -3.2]
Index= Index(['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'],
dtype='object')
```

```
# creating a series from dictionary object
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}

s4 = Series(capitals)
print(s4)
print('Values=', s4.values) # display values of the Series
print('Index=', s4.index) # display indices of the Series
```

```
MI       Lansing
CA    Sacramento
TX        Austin
MN       St Paul
dtype: object
Values= ['Lansing' 'Sacramento' 'Austin' 'St Paul']
Index= Index(['MI', 'CA', 'TX', 'MN'], dtype='object')
```

(2) Accessing elements & Slice

```
s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
 index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
print(s3)

# Accessing elements of a Series

print('\ns3[2]=', s3[2]) # display third element of the Series
print('s3[\'Jan 3\']=', s3['Jan 3']) # indexing element of a Series

print('\ns3[1:3]=') # display a slice of the Series
print(s3[1:3])

print('\ns3.iloc([1:3])=') # display a slice of the Series
```

```
print(s3.iloc[1:3])
```

```
Jan 1     1.2
Jan 2     2.5
Jan 3    -2.2
Jan 4     3.1
Jan 5    -0.8
Jan 6    -3.2
dtype: float64

s3[2]= -2.2
s3['Jan 3']= -2.2

s3[1:3]=
Jan 2     2.5
Jan 3    -2.2
dtype: float64

s3.iloc([1:3])=
Jan 2     2.5
Jan 3    -2.2
dtype: float64
```

```
print('shape =', s3.shape) # get the dimension of the Series
print('size =', s3.size) # get the # of elements of the Series
```

```
shape = (6,)
size = 6
```

(3) Operation

```
print(s3[s3 > 0]) # applying filter to select elements of the Series
```

```
Jan 1     1.2
Jan 2     2.5
Jan 4     3.1
dtype: float64
```

```
print(s3 + 4) # applying scalar operation on a numeric Series
print(s3 / 4)
```

```
Jan 1     5.2
Jan 2     6.5
Jan 3     1.8
Jan 4     7.1
Jan 5     3.2
Jan 6     0.8
dtype: float64
```

```
Jan 1     0.300
Jan 2     0.625
Jan 3    -0.550
Jan 4     0.775
Jan 5    -0.200
Jan 6    -0.800
dtype: float64
```

```python
print(np.log(s3 + 4)) # applying numpy math functions to a numeric Series
```

```
Jan 1     1.648659
Jan 2     1.871802
Jan 3     0.587787
Jan 4     1.960095
Jan 5     1.163151
Jan 6    -0.223144
dtype: float64
```

2.2.2 DataFrame

- A DataFrame object is a tabular, spreadsheet-like data structure containing a collection of columns, each of which can be of different types (numeric, string, boolean, etc).
- Unlike Series, a DataFrame has distinct row and column indices.
- There are many ways to create a DataFrame object (e.g., from a dictionary, list of tuples,or even numpy's ndarrays).

Define

- DataFrame():
    - dictionary : key - column name, value - column value
    - tuple list or numpy's ndarray : 1st. parameter list - value, 2nd. parameter columns : column names
    - index parameter : change the row index

Attribute

- index : print the row indices
- columns : print the column indices
- shape : print the shape of DataFrame
- size : print the size of DataFrame

```python
from pandas import DataFrame

# creating DataFrame from dictionary
cars = {'make': ['Ford', 'Honda', 'Toyota', 'Tesla'],
        'model': ['Taurus', 'Accord', 'Camry', 'Model S'],
        'MSRP': [27595, 23570, 23495, 68000]}

carData = DataFrame(cars)
carData # display the table
```

```css
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	make	model	MSRP
0	Ford	Taurus	27595
1	Honda	Accord	23570
2	Toyota	Camry	23495
3	Tesla	Model S	68000

```python
print(carData.index) # print the row indices
print(carData.columns) # print the column indices
```

```
RangeIndex(start=0, stop=4, step=1)
Index(['make', 'model', 'MSRP'], dtype='object')
```

```python
carData2 = DataFrame(cars, index = [1,2,3,4]) # change the row index
carData2['year'] = 2018 # add column with same value
carData2['dealership'] = ['Courtesy Ford','Capital Honda','Spartan Toyota','N/A']
carData2 # display table
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	23570	2018	Capital Honda
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	N/A

```
# Create from tuple list
tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
 (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]
columnNames = ['year','temp','precip']
weatherData = DataFrame(tuplelist, columns=columnNames)
weatherData
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	year	temp	precip
0	2011	45.1	32.4
1	2012	42.4	34.5
2	2013	47.2	39.2
3	2014	44.2	31.4
4	2015	39.9	29.8
5	2016	41.5	36.7

```
import numpy as np

npdata = np.random.randn(5,3) # create a 5 by 3 random matrix
columnNames = ['x1','x2','x3']
data = DataFrame(npdata, columns=columnNames)
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	x1	x2	x3
0	0.093553	1.201448	-0.198183
1	0.973588	-0.773777	0.211157
2	1.793499	-0.119079	-0.140741
3	0.979697	-0.148298	0.258180
4	0.372209	0.212727	-1.224305

(1) Accessing elements

Index : Column Name

- DataFrame accesse an entire column.
- Return a Series object.

iloc[] :

- Single Parameter : Row Index
  - iloc[row_index]: return the row datas of DataFrame, Series Object
- Two Parameters : Row Index , Column Index
  - iloc[row_index,column_index] : return the data of row_index row and column_index column
  - iloc[row_index,column_name] : return the data of row_index row and name is column_name

```
# accessing an entire column will return a Series object
print(data['x2'])
print(type(data['x2']))
```

```
0    1.201448
1   -0.773777
2   -0.119079
3   -0.148298
4    0.212727
Name: x2, dtype: float64
<class 'pandas.core.series.Series'>
```

```python
# accessing an entire row will return a Series object
print('Row 3 of data table:')
print(data.iloc[2]) # returns the 3rd row of DataFrame
print(type(data.iloc[2]))
print('\nRow 3 of car data table:')
print(carData2.iloc[2]) # row contains objects of different typ
```

```
Row 3 of data table:
x1    1.793499
x2   -0.119079
x3   -0.140741
Name: 2, dtype: float64
<class 'pandas.core.series.Series'>

Row 3 of car data table:
make                    Toyota
model                    Camry
MSRP                     23495
year                      2018
dealership      Spartan Toyota
Name: 3, dtype: object
```

```python
# accessing a specific element of the DataFrame
print(carData2.iloc[1,2]) # retrieving second row, third column
print(carData2.loc[1,'model']) # retrieving second row, column named 'model'

# accessing a slice of the DataFrame
print('carData2.iloc[1:3,1:3]=')
print(carData2.iloc[1:3,1:3])
```

```
23570
Taurus
carData2.iloc[1:3,1:3]=
    model   MSRP
2  Accord  23570
3   Camry  23495
```

```python
print('carData2.shape =', carData2.shape)
print('carData2.size =', carData2.size)
```

```
carData2.shape = (4, 5)
carData2.size = 20
```

```python
# selection and filtering
print('carData2[carData2.MSRP > 25000]')
print(carData2[carData2.MSRP > 25000])
```

```
carData2[carData2.MSRP > 25000]
    make    model   MSRP  year     dealership
1   Ford   Taurus  27595  2018  Courtesy Ford
4  Tesla  Model S  68000  2018            N/A
```

2.2.3 Arithmetic Operations

- T : transpose operation
- + : addition operation
- * : multiplication operation
- axis parameter for following function :
  - axis = 0 default value : operate for each element
  - axis = 1 : operate for each row
- abs() : get the absolute value for each element
- max() : get the maximum value for each element
- min() : get minimum value for each element
- sum() : get sum of values for each element
- mean() : get average value for each column
- apply() :

```
print(data)

print('Data transpose operation:')
print(data.T) # transpose operation

print('Addition:')
print(data + 4) # addition operation

print('Multiplication:')
print(data * 10) # multiplication operation
```

```
         x1        x2        x3
0  0.093553  1.201448 -0.198183
1  0.973588 -0.773777  0.211157
2  1.793499 -0.119079 -0.140741
3  0.979697 -0.148298  0.258180
4  0.372209  0.212727 -1.224305
Data transpose operation:
           0         1         2         3         4
x1  0.093553  0.973588  1.793499  0.979697  0.372209
x2  1.201448 -0.773777 -0.119079 -0.148298  0.212727
x3 -0.198183  0.211157 -0.140741  0.258180 -1.224305
Addition:
         x1        x2        x3
0  4.093553  5.201448  3.801817
1  4.973588  3.226223  4.211157
2  5.793499  3.880921  3.859259
3  4.979697  3.851702  4.258180
4  4.372209  4.212727  2.775695
Multiplication:
          x1         x2         x3
0   0.935534  12.014484  -1.981827
1   9.735883  -7.737774   2.111565
2  17.934992  -1.190787  -1.407409
3   9.796967  -1.482978   2.581804
4   3.722095   2.127269 -12.243045
```

```python
print('data =')
print(data)

columnNames = ['x1','x2','x3']
data2 = DataFrame(np.random.randn(5,3), columns=columnNames)

print('\ndata2 =')
print(data2)

print('\ndata + data2 = ')
print(data.add(data2))

print('\ndata * data2 = ')
print(data.mul(data2))
```

```
data =
         x1        x2        x3
0  0.093553  1.201448 -0.198183
1  0.973588 -0.773777  0.211157
2  1.793499 -0.119079 -0.140741
3  0.979697 -0.148298  0.258180
4  0.372209  0.212727 -1.224305

data2 =
         x1        x2        x3
0 -0.327995 -1.489451 -0.449003
1 -0.673032  1.416312  1.021364
2 -0.001530  1.666680 -0.011742
3  1.673819  0.381367  0.023262
4  0.282997  0.474601 -0.837890

data + data2 =
         x1        x2        x3
0 -0.234441 -0.288003 -0.647186
1  0.300557  0.642535  1.232521
2  1.791970  1.547602 -0.152483
3  2.653516  0.233069  0.281442
4  0.655206  0.687328 -2.062195

data * data2 =
         x1        x2        x3
0 -0.030685 -1.789499  0.088985
1 -0.655256 -1.095910  0.215668
2 -0.002743 -0.198466  0.001653
3  1.639835 -0.056556  0.006006
4  0.105334  0.100960  1.025833
```

```python
print(data.abs()) # get the absolute value for each element

print('\nMaximum value per column:')
print(data.max()) # get maximum value for each column

print('\nMinimum value per row:')
print(data.min(axis=1)) # get minimum value for each row
```

```python

print('\nSum of values per column:')
print(data.sum()) # get sum of values for each column

print('\nAverage value per row:')
print(data.mean(axis=1)) # get average value for each row

print('\nCalculate max - min per column')
f = lambda x: x.max() - x.min()
print(data.apply(f))

print('\nCalculate max - min per row')
f = lambda x: x.max() - x.min()
print(data.apply(f, axis=1))
```

```
         x1        x2        x3
0  0.093553  1.201448  0.198183
1  0.973588  0.773777  0.211157
2  1.793499  0.119079  0.140741
3  0.979697  0.148298  0.258180
4  0.372209  0.212727  1.224305

Maximum value per column:
x1    1.793499
x2    1.201448
x3    0.258180
dtype: float64

Minimum value per row:
0   -0.198183
1   -0.773777
2   -0.140741
3   -0.148298
4   -1.224305
dtype: float64

Sum of values per column:
x1    4.212547
x2    0.373021
x3   -1.093891
dtype: float64

Average value per row:
0    0.365606
1    0.136989
2    0.511227
3    0.363193
4   -0.213123
dtype: float64

Calculate max - min per column
x1    1.699946
x2    1.975226
x3    1.482485
dtype: float64
```

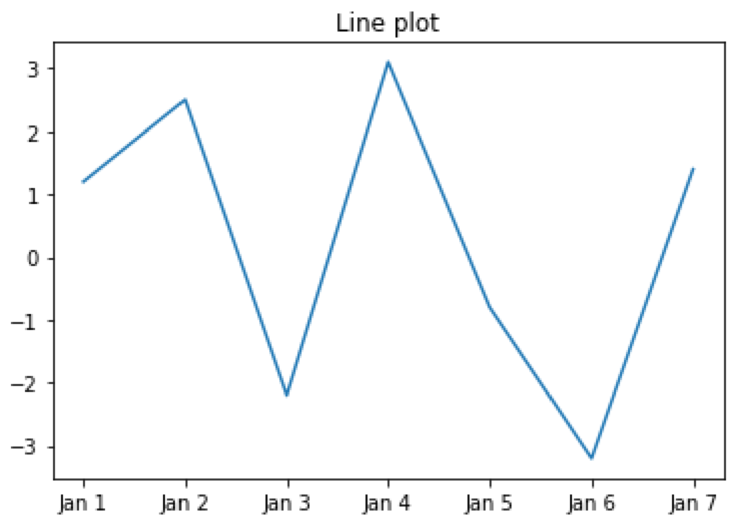
```

Calculate max - min per row
0     1.399631
1     1.747366
2     1.934240
3     1.127995
4     1.596514
dtype: float64
```

2.2.4 Plotting Series and DataFrame

There are built-in functions you can use to plot the data stored in a Series or a DataFrame.

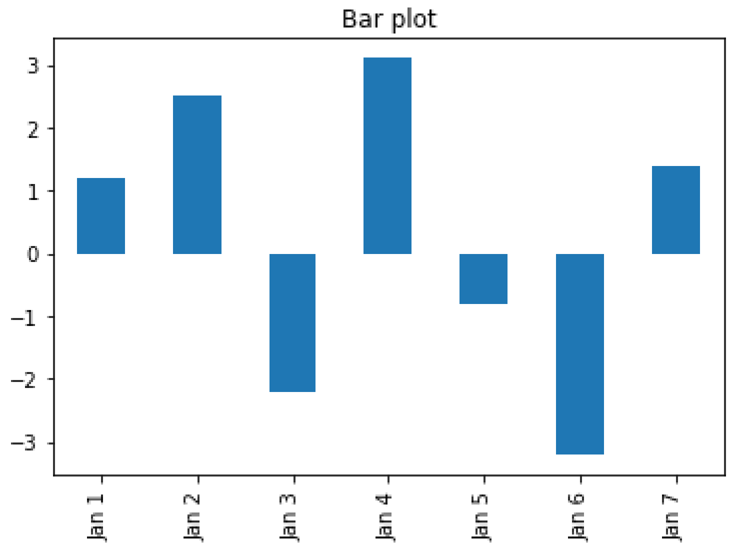
```
%matplotlib inline

s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2,1.4],
 index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6','Jan 7'])
s3.plot(kind='line', title='Line plot')
```

```
<AxesSubplot:title={'center':'Line plot'}>
```



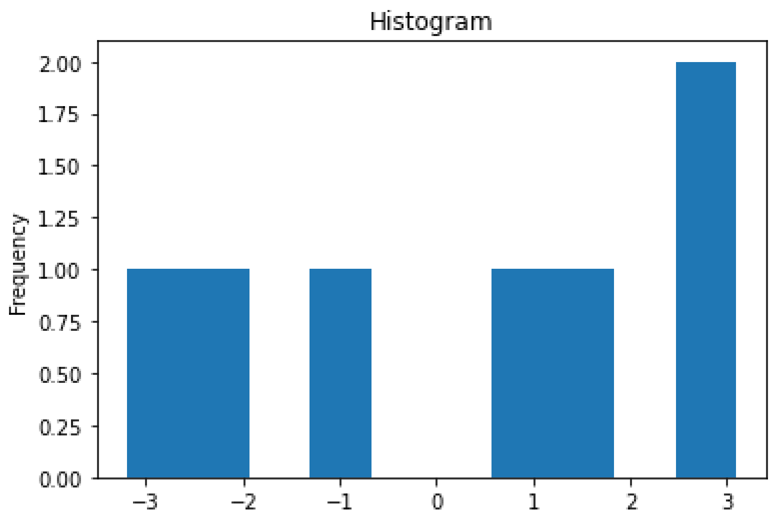

```
s3.plot(kind='bar', title='Bar plot')
```

```
<AxesSubplot:title={'center':'Bar plot'}>
```



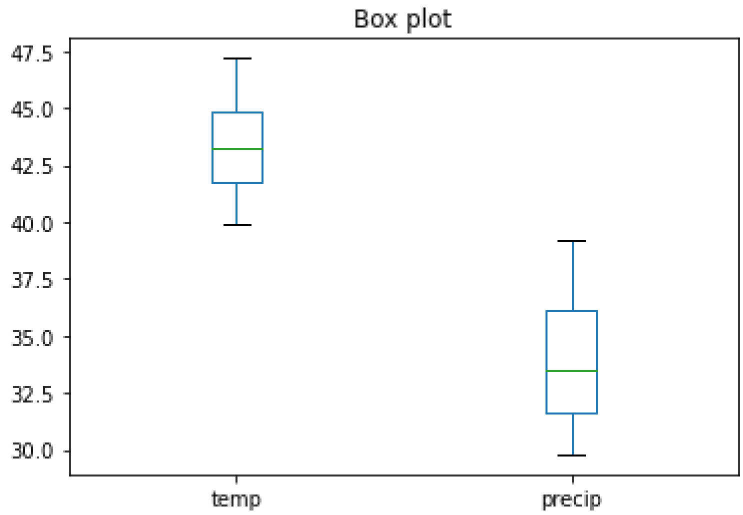

```
s3.plot(kind='hist', title = 'Histogram')
```

```
<AxesSubplot:title={'center':'Histogram'}, ylabel='Frequency'>
```




```
tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
 (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]

columnNames = ['year','temp','precip']
weatherData = DataFrame(tuplelist, columns=columnNames)
weatherData[['temp','precip']].plot(kind='box', title='Box plot')
```

```
<AxesSubplot:title={'center':'Box plot'}>
```


Box plot
47.5
45.0
42.5
40.0
37.5
35.0
32.5
30.0
temp
precip