

Information

- Title : 2. Numpy and Pandas 수행한 결과를 캡처한 파일
- ID : 12164484
- Name : 송해소
- Time : 2021.03.11

2.1 Numpy

- Numpy, which stands for numerical Python, is a Python library package to support numerical computations.
- The basic data structure in numpy is *multi-dimensional array* object called *ndarray*.
- 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 $10^{1st. \text{ parameter}}$ to $10^{2nd. \text{ parameter}}$. 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 ndarray
- `shape` : the shape of the ndarray, (1st., 2nd.): 1st. - rows and 2nd. - columns
- `size` : the size of the ndarray
- `dtype` : the data type of the ndarray

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

```

8 print("Size = ",oneDim.size) # print the size
9 print("Array type = ", oneDim.dtype) # print the data type
10
11 # a two-dimensional array (matrix)
12 twoDim = np.array([[1,2],[3,4],[5,6],[7,8]])
13 print(twoDim)
14 print("#Dimensions = ", twoDim.ndim)
15 print("Dimension = ", twoDim.shape)
16 print("Size = ", twoDim.size)
17 print("Array type = ", oneDim.dtype)
18
19 # create ndarray from tuple
20 arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)])
21 print(arrFromTuple)
22 print("#Dimensions = ",arrFromTuple.ndim)
23 print("Dimension = ",arrFromTuple.shape)
24 print("Size = ",arrFromTuple.size)

```

```

1 [1. 2. 3. 4. 5.]
2 #Dimensions = 1
3 Dimension = (5,)
4 Size = 5
5 Array type = float64
6 [[1 2]
7  [3 4]
8  [5 6]
9  [7 8]]
10 #Dimensions = 2
11 Dimension = (4, 2)
12 Size = 8
13 Array type = float64
14 [['1' 'a' '3.0']
15  ['2' 'b' '3.5']]
16 #Dimensions = 2
17 Dimension = (2, 3)
18 Size = 6

```

```

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

```

```

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

```

```

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

```

```

1 [[0. 0. 0.]
2  [0. 0. 0.]]
3 [[1. 1.]
4  [1. 1.]
5  [1. 1.]]
6 [[1. 0. 0.]
7  [0. 1. 0.]
8  [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.

```

1 x = np.array([1,2,3,4,5])
2
3 print(x + 1) # addition
4 print(x - 1) # subtraction
5 print(x * 2) # multiplication
6 print(x // 2) # integer division
7 print(x ** 2) # square
8 print(x % 2) # modulo
9 print(1 / x) # division

```

```

1 [2 3 4 5 6]
2 [0 1 2 3 4]
3 [ 2  4  6  8 10]
4 [0 1 1 2 2]
5 [ 1  4  9 16 25]
6 [1 0 1 0 1]
7 [1.          0.5          0.33333333 0.25          0.2          ]

```

```

1 x = np.array([2,4,6,8,10])
2 y = np.array([1,2,3,4,5])
3
4 print(x + y)
5 print(x - y)
6 print(x * y)
7 print(x / y)
8 print(x // y)
9 print(x ** y)

```

```

1 [ 3  6  9 12 15]
2 [1 2 3 4 5]
3 [ 2  8 18 32 50]
4 [2. 2. 2. 2. 2.]
5 [2 2 2 2 2]
6 [      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()`

```

1 x = np.arange(-5,5)
2 print(x)
3
4 y = x[3:5] # y is a slice, i.e., pointer to a subarray in x
5 print(y)
6
7 y[:] = 1000 # modifying the value of y will change x
8 print(y)
9 print(x)
10
11 z = x[3:5].copy() # makes a copy of the subarray
12 print(z)
13
14 z[:] = 500 # modifying the value of z will not affect x
15 print(z)
16 print(x)

```

```

1 [-5 -4 -3 -2 -1  0  1  2  3  4]
2 [-2 -1]
3 [1000 1000]
4 [ -5  -4  -3 1000 1000  0  1  2  3  4]
5 [1000 1000]
6 [500 500]
7 [ -5  -4  -3 1000 1000  0  1  2  3  4]

```

```

1 my2dlist = [[1,2,3,4],[5,6,7,8],[9,10,11,12]] # a 2-dim list
2 print(my2dlist)
3 print(my2dlist[2]) # access the third sublist
4 print(my2dlist[:,2]) # can't access third element of each sublist
5 # print(my2dlist[:,2]) # this will cause syntax error
6
7 my2darr = np.array(my2dlist)
8 print(my2darr)
9 print(my2darr[2,:]) # access the third row
10 print(my2darr[:,2]) # access the third row
11 print(my2darr[:,2]) # access the third row (similar to 2d list)
12 print(my2darr[:,2]) # access the third column
13 print(my2darr[:2,2:]) # access the first two rows & last two columns

```

```

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

```

Boolean Indexing

```

1 my2darr = np.arange(1,13,1).reshape(3,4)
2 print(my2darr)
3
4 divBy3 = my2darr[my2darr % 3 == 0] # print the element can be divided 3 in my2darr
5 print(divBy3, type(divBy3))
6
7 divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0] # print the element can be
  divided 3 in 2nd. my2darr row
8 print(divBy3LastRow)

```

```

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

```

```

1 my2darr = np.arange(1,13,1).reshape(4,3)
2 print(my2darr)
3
4 indices = [2,1,0,3] # selected row indices
5 print(my2darr[indices,:])
6
7 rowIndex = [0,0,1,2,3] # row index into my2darr
8 columnIndex = [0,2,0,1,2] # column index into my2darr
9 print(my2darr[rowIndex,columnIndex])

```

```

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

```

2.1.4 Numpy Arithmetic 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
- `sign()`: get the sign of each element
- `exp()` : apply exponentiation
- `sort()`: sort array
- `min()` : minimum element in ndarray
- `max()` : maximum 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 $\max(x,y)$

```

1 y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4]) # generate a random vector
2 print(y)
3
4 print(np.abs(y)) # convert to absolute values
5 print(np.sqrt(abs(y))) # apply square root to each element
6 print(np.sign(y)) # get the sign of each element
7 print(np.exp(y)) # apply exponentiation
8 print(np.sort(y)) # sort array

```

```

1 [-1.4  0.4 -3.2  2.5  3.4]
2 [1.4  0.4  3.2  2.5  3.4]
3 [1.18321596 0.63245553 1.78885438 1.58113883 1.84390889]
4 [-1.  1. -1.  1.  1.]
5 [ 0.24659696  1.4918247  0.0407622  12.18249396 29.96410005]
6 [-3.2 -1.4  0.4  2.5  3.4]

```

```

1 x = np.arange(-2,3)
2 y = np.random.randn(5)
3 print(x)
4 print(y)
5
6 print(np.add(x,y)) # element-wise addition x + y
7 print(np.subtract(x,y)) # element-wise subtraction x - y
8 print(np.multiply(x,y)) # element-wise multiplication x * y
9 print(np.divide(x,y)) # element-wise division x / y
10 print(np.maximum(x,y)) # element-wise maximum max(x,y)

```

```

1 [-2 -1  0  1  2]
2 [-0.62238064  0.06803966 -0.49192638  2.94373694  0.97320746]
3 [-2.62238064 -0.93196034 -0.49192638  3.94373694  2.97320746]
4 [-1.37761936 -1.06803966  0.49192638 -1.94373694  1.02679254]
5 [ 1.24476127 -0.06803966 -0.          2.94373694  1.94641491]
6 [ 3.21346758 -14.69731118 -0.          0.33970427  2.05506029]
7 [-0.62238064  0.06803966  0.          2.94373694  2.          ]

```

```

1 y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4]) # generate a random vector
2 print(y)
3
4 print("Min =", np.min(y)) # min
5 print("Max =", np.max(y)) # max
6 print("Average =", np.mean(y)) # mean/average
7 print("Std deviation =", np.std(y)) # standard deviation
8 print("Sum =", np.sum(y)) # sum

```

```

1 [-3.2 -1.4  0.4  2.5  3.4]
2 Min = -3.2
3 Max = 3.4
4 Average = 0.34000000000000014
5 Std deviation = 2.432776191925595
6 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

```

1 X = np.random.randn(2,3) # create a 2 x 3 random matrix
2 print(X)
3 print(X.T) # matrix transpose operation X^T
4
5 y = np.random.randn(3) # random vector
6 print(y)
7 print(X.dot(y)) # matrix-vector multiplication X * y
8 print(X.dot(X.T)) # matrix-matrix multiplication X * X^T
9 print(X.T.dot(X)) # matrix-matrix multiplication X^T * X

```

```

1 [[ 0.30466078 -1.32182636  0.23663167]
2  [-0.56404753  0.7886946   0.21350702]]
3 [[ 0.30466078 -0.56404753]
4  [-1.32182636  0.7886946 ]
5  [ 0.23663167  0.21350702]]
6 [-0.2024012  -0.74751189  0.24756148]
7 [ 0.98499809 -0.42253858]
8 [[ 1.89603765 -1.16383795]
9  [-1.16383795  0.98577404]]
10 [[ 0.41096781 -0.84756989 -0.04833572]
11  [-0.84756989  2.36926409 -0.14439414]
12  [-0.04833572 -0.14439414  0.10157979]]

```

```

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

```



```

4 C = X.T.dot(X) # C = X^T * X is a square matrix
5
6 invC = np.linalg.inv(C) # inverse of a square matrix
7 print(invC)
8
9 detC = np.linalg.det(C) # determinant of a square matrix
10 print(detC)
11
12 S,U = np.linalg.eig(C) # eigenvalue S and eigenvector U of a square matrix
13 print(S)
14 print(U)

```

```

1 [[-0.40091206  1.88245093  0.96576246]
2  [ 0.47607027  0.77646112 -0.3880549 ]
3  [-0.41763    -1.60759296  0.09800108]
4  [ 0.45919206  0.31563008 -0.48303597]
5  [ 0.44783751  0.4741899   0.88965723]]
6 [[ 1.39248398 -0.23494643  0.46752532]
7  [-0.23494643  0.21196706 -0.21140471]
8  [ 0.46752532 -0.21140471  0.73109281]]
9 8.825194646750928
10 [0.5890574  1.97625887 7.5809363 ]
11 [[-0.86352887  0.49872157  0.07479766]
12  [ 0.20229439  0.20670059  0.95726268]
13  [-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 an index array.
- A Series object can be created from a list, a numpy array, or a Python dictionary. You can apply most 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 : indexes 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

```
1 from pandas import Series
2
3 # creating a series from a list
4 s = Series([3.1, 2.4, -1.7, 0.2, -2.9, 4.5])
5 print(s)
6
7 print('Values=', s.values) # display values of the Series
8 print('Index=', s.index) # display indices of the Series
```

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

```
1 import numpy as np
2
3 # creating a series from a numpy ndarray
4 s2 = Series(np.random.randn(6))
5
6 print(s2)
7 print('Values=', s2.values) # display values of the Series
8 print('Index=', s2.index) # display indices of the Series
```

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

```

1 # creating a series from list
2 s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
3             index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
4
5 print(s3)
6 print('Values=', s3.values) # display values of the Series
7 print('Index=', s3.index) # display indices of the Series

```

```

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

```

```

1 # creating a series from dictionary object
2 capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}
3
4 s4 = Series(capitals)
5 print(s4)
6 print('Values=', s4.values) # display values of the Series
7 print('Index=', s4.index) # display indices of the Series

```

```

1 MI      Lansing
2 CA      Sacramento
3 TX       Austin
4 MN      St Paul
5 dtype: object
6 Values= ['Lansing' 'Sacramento' 'Austin' 'St Paul']
7 Index= Index(['MI', 'CA', 'TX', 'MN'], dtype='object')

```

(2) Accessing elements & Slice

```

1 s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
2             index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
3 print(s3)
4
5 # Accessing elements of a Series
6
7 print('\ns3[2]=', s3[2]) # display third element of the Series
8 print('s3[\'Jan 3\']='', s3['Jan 3']) # indexing element of a Series
9
10 print('\ns3[1:3]=') # display a slice of the Series
11 print(s3[1:3])
12
13 print('\ns3.iloc([1:3])=') # display a slice of the Series

```

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

```
1 Jan 1    1.2
2 Jan 2    2.5
3 Jan 3   -2.2
4 Jan 4    3.1
5 Jan 5   -0.8
6 Jan 6   -3.2
7 dtype: float64
8
9 s3[2]= -2.2
10 s3['Jan 3']= -2.2
11
12 s3[1:3]=
13 Jan 2    2.5
14 Jan 3   -2.2
15 dtype: float64
16
17 s3.iloc([1:3])=
18 Jan 2    2.5
19 Jan 3   -2.2
20 dtype: float64
```

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

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

(3) Operation

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

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

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

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

```
8 Jan 1    0.300
9 Jan 2    0.625
10 Jan 3   -0.550
11 Jan 4    0.775
12 Jan 5   -0.200
13 Jan 6   -0.800
14 dtype: float64
```

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

```
1 Jan 1    1.648659
2 Jan 2    1.871802
3 Jan 3    0.587787
4 Jan 4    1.960095
5 Jan 5    1.163151
6 Jan 6   -0.223144
7 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

```

1 from pandas import DataFrame
2
3 # creating DataFrame from dictionary
4 cars = {'make': ['Ford', 'Honda', 'Toyota', 'Tesla'],
5         'model': ['Taurus', 'Accord', 'Camry', 'Model S'],
6         'MSRP': [27595, 23570, 23495, 68000]}
7
8 carData = DataFrame(cars)
9 carData # display the table

```

```

1 .dataframe tbody tr th {
2     vertical-align: top;
3 }
4
5 .dataframe thead th {
6     text-align: right;
7 }

```

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

```

1 print(carData.index) # print the row indices
2 print(carData.columns) # print the column indices

```

```

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

```

```

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

```

```

1 .dataframe tbody tr th {
2     vertical-align: top;
3 }
4
5 .dataframe thead th {
6     text-align: right;
7 }

```

	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

```

1 # Create from tuple list
2 tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
3             (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]
4 columnNames = ['year','temp','precip']
5 weatherData = DataFrame(tuplelist, columns=columnNames)
6 weatherData

```

```

1 .dataframe tbody tr th {
2     vertical-align: top;
3 }
4
5 .dataframe thead th {
6     text-align: right;
7 }

```

	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

```

1 import numpy as np
2
3 npdata = np.random.randn(5,3) # create a 5 by 3 random matrix
4 columnNames = ['x1','x2','x3']
5 data = DataFrame(npdata, columns=columnNames)
6 data

```

```

1 .dataframe tbody tr th {
2     vertical-align: top;
3 }
4
5 .dataframe thead th {
6     text-align: right;
7 }

```

	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 accesses 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

```

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

```

```

1 0    1.201448
2 1   -0.773777
3 2   -0.119079
4 3   -0.148298
5 4    0.212727
6 Name: x2, dtype: float64
7 <class 'pandas.core.series.Series'>

```



```

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

```

```

1 Row 3 of data table:
2 x1    1.793499
3 x2   -0.119079
4 x3   -0.140741
5 Name: 2, dtype: float64
6 <class 'pandas.core.series.Series'>
7
8 Row 3 of car data table:
9 make           Toyota
10 model          Camry
11 MSRP           23495
12 year           2018
13 dealership    Spartan Toyota
14 Name: 3, dtype: object

```

```

1 # accessing a specific element of the DataFrame
2 print(carData2.iloc[1,2]) # retrieving second row, third column
3 print(carData2.loc[1,'model']) # retrieving second row, column named 'model'
4
5 # accessing a slice of the DataFrame
6 print('carData2.iloc[1:3,1:3]=')
7 print(carData2.iloc[1:3,1:3])

```

```

1 23570
2 Taurus
3 carData2.iloc[1:3,1:3]=
4      model  MSRP
5 2  Accord  23570
6 3   Camry  23495

```

```

1 print('carData2.shape =', carData2.shape)
2 print('carData2.size =', carData2.size)

```

```

1 carData2.shape = (4, 5)
2 carData2.size = 20

```

```

1 # selection and filtering
2 print('carData2[carData2.MSRP > 25000]')
3 print(carData2[carData2.MSRP > 25000])

```

```

1 carData2[carData2.MSRP > 25000]
2      make  model  MSRP  year  dealership
3 1   Ford  Taurus  27595  2018  Courtesy Ford
4 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() :

```
1 print(data)
2
3 print('Data transpose operation:')
4 print(data.T) # transpose operation
5
6 print('Addition:')
7 print(data + 4) # addition operation
8
9 print('Multiplication:')
10 print(data * 10) # multiplication operation
```

```
1          x1          x2          x3
2 0  0.093553  1.201448 -0.198183
3 1  0.973588 -0.773777  0.211157
4 2  1.793499 -0.119079 -0.140741
5 3  0.979697 -0.148298  0.258180
6 4  0.372209  0.212727 -1.224305
7 Data transpose operation:
8          0          1          2          3          4
9 x1  0.093553  0.973588  1.793499  0.979697  0.372209
10 x2  1.201448 -0.773777 -0.119079 -0.148298  0.212727
11 x3 -0.198183  0.211157 -0.140741  0.258180 -1.224305
12 Addition:
13          x1          x2          x3
14 0  4.093553  5.201448  3.801817
15 1  4.973588  3.226223  4.211157
16 2  5.793499  3.880921  3.859259
17 3  4.979697  3.851702  4.258180
18 4  4.372209  4.212727  2.775695
19 Multiplication:
20          x1          x2          x3
21 0  0.935534  12.014484 -1.981827
22 1  9.735883  -7.737774  2.111565
23 2  17.934992 -1.190787 -1.407409
24 3  9.796967  -1.482978  2.581804
25 4  3.722095  2.127269 -12.243045
```

```

1 print('data =')
2 print(data)
3
4 columnNames = ['x1', 'x2', 'x3']
5 data2 = DataFrame(np.random.randn(5,3), columns=columnNames)
6
7 print('\ndata2 =')
8 print(data2)
9
10 print('\ndata + data2 = ')
11 print(data.add(data2))
12
13 print('\ndata * data2 = ')
14 print(data.mul(data2))

```

```

1 data =
2      x1      x2      x3
3 0  0.093553  1.201448 -0.198183
4 1  0.973588 -0.773777  0.211157
5 2  1.793499 -0.119079 -0.140741
6 3  0.979697 -0.148298  0.258180
7 4  0.372209  0.212727 -1.224305
8
9 data2 =
10     x1      x2      x3
11 0 -0.327995 -1.489451 -0.449003
12 1 -0.673032  1.416312  1.021364
13 2 -0.001530  1.666680 -0.011742
14 3  1.673819  0.381367  0.023262
15 4  0.282997  0.474601 -0.837890
16
17 data + data2 =
18     x1      x2      x3
19 0 -0.234441 -0.288003 -0.647186
20 1  0.300557  0.642535  1.232521
21 2  1.791970  1.547602 -0.152483
22 3  2.653516  0.233069  0.281442
23 4  0.655206  0.687328 -2.062195
24
25 data * data2 =
26     x1      x2      x3
27 0 -0.030685 -1.789499  0.088985
28 1 -0.655256 -1.095910  0.215668
29 2 -0.002743 -0.198466  0.001653
30 3  1.639835 -0.056556  0.006006
31 4  0.105334  0.100960  1.025833

```

```

1 print(data.abs()) # get the absolute value for each element
2
3 print('\nMaximum value per column:')
4 print(data.max()) # get maximum value for each column
5
6 print('\nMinimum value per row:')
7 print(data.min(axis=1)) # get minimum value for each row

```

```

8
9 print('\nSum of values per column:')
10 print(data.sum()) # get sum of values for each column
11
12 print('\nAverage value per row:')
13 print(data.mean(axis=1)) # get average value for each row
14
15 print('\nCalculate max - min per column')
16 f = lambda x: x.max() - x.min()
17 print(data.apply(f))
18
19 print('\nCalculate max - min per row')
20 f = lambda x: x.max() - x.min()
21 print(data.apply(f, axis=1))

```

```

1      x1      x2      x3
2 0  0.093553  1.201448  0.198183
3 1  0.973588  0.773777  0.211157
4 2  1.793499  0.119079  0.140741
5 3  0.979697  0.148298  0.258180
6 4  0.372209  0.212727  1.224305
7
8 Maximum value per column:
9 x1      1.793499
10 x2      1.201448
11 x3      0.258180
12 dtype: float64
13
14 Minimum value per row:
15 0   -0.198183
16 1   -0.773777
17 2   -0.140741
18 3   -0.148298
19 4   -1.224305
20 dtype: float64
21
22 Sum of values per column:
23 x1      4.212547
24 x2      0.373021
25 x3     -1.093891
26 dtype: float64
27
28 Average value per row:
29 0      0.365606
30 1      0.136989
31 2      0.511227
32 3      0.363193
33 4     -0.213123
34 dtype: float64
35
36 Calculate max - min per column
37 x1      1.699946
38 x2      1.975226
39 x3      1.482485
40 dtype: float64

```

```

41
42 Calculate max - min per row
43 0    1.399631
44 1    1.747366
45 2    1.934240
46 3    1.127995
47 4    1.596514
48 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.

```

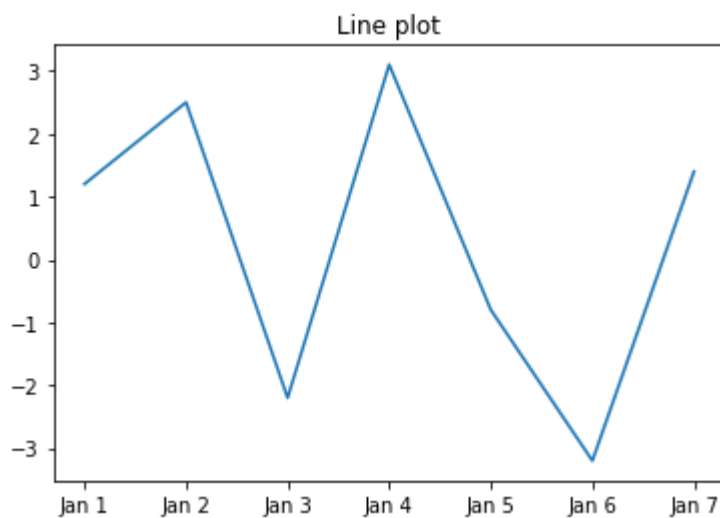
1 %matplotlib inline
2
3 s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2,1.4],
4 index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6','Jan 7'])
5 s3.plot(kind='line', title='Line plot')

```

```

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

```



```

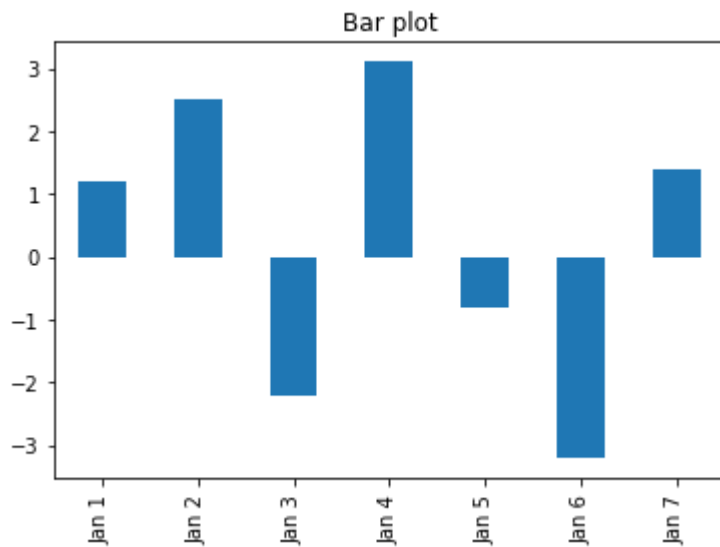
1 s3.plot(kind='bar', title='Bar plot')

```

```

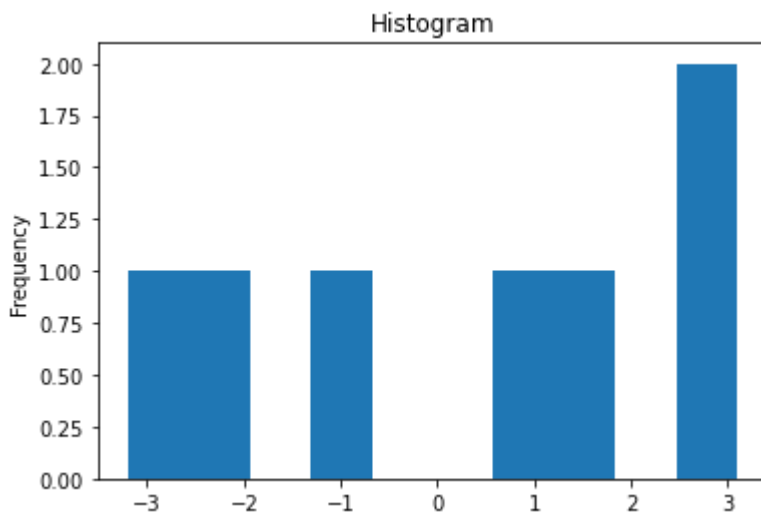
1 <AxesSubplot:title={'center':'Bar plot'}>

```



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

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



```
1 tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
2 (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]
3
4 columnNames = ['year','temp','precip']
5 weatherData = DataFrame(tuplelist, columns=columnNames)
6 weatherData[['temp','precip']].plot(kind='box', title='Box plot')
```

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

