Module 2: Introduction to Numpy and Pandas

Assignment 1.2

Python code 2

University of San Diego

ADS 502

Dingyi Duan

2.1 Introduction to Numpy

Numpy, which stands for numerical Python, is a Python library package to support numerical computations. The basic data structure in numpy is a 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.

In [1]:
```
# Creating an array
```

In [2]:
```
import numpy as np

oneDim = np.array([1.0,2,3,4,5])   # a 1-dimensional array (vector)
print(oneDim)
print("#Dimensions =", oneDim.ndim)
print("Dimension =", oneDim.shape)
print("Size =", oneDim.size)
print("Array type =", oneDim.dtype)

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

arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)])  # create ndarray from tuple
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 = int32
[['1' 'a' '3.0']
 ['2' 'b' '3.5']]
#Dimensions = 2
Dimension = (2, 3)
Size = 6
```

There are several built-in functions in numpy that can be used to create ndarrays

In [3]:
```
print(np.random.rand(5))      # random numbers from a uniform distribution betwee
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
print(np.arange(12).reshape(3,4))  # reshape to a matrix
print(np.linspace(0,1,10))    # split interval [0,1] into 10 equally separated va
print(np.logspace(-3,3,7))    # create ndarray with values from 10^-3 to 10^3
```

```
[0.03523041 0.45071115 0.25549321 0.14065045 0.80894721]
[ 0.23522427  1.06151829 -0.25683064 -1.03330967 -1.76698427]
[-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]
```

In [4]:
```
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.

In [42]:
```
# Array operations
```

In [5]:
```
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       ]
```

In [43]:
```
# Operations between arrays require arrays to be the same dimensions
```

In [6]:
```
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.

In [44]:
```
# Manipulation (of elements) of arrays
```

In [7]:
```
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]
```

In [8]:
```
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]]
```

ndarray also supports boolean indexing.

In [9]:
```
my2darr = np.arange(1,13,1).reshape(3,4)
print(my2darr)

divBy3 = my2darr[my2darr % 3 == 0]
print(divBy3, type(divBy3))

divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0]
print(divBy3LastRow)
```

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

More indexing examples.

In [10]:
```
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 Arithmetic and Statistical Functions

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

In [45]:
```
# Mathmetical operations for arrays
```

In [11]:
```
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]
```

In [12]:
```
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]
[-0.79691643  0.53466596  1.24330407 -0.72644778 -1.4685496 ]
[-2.79691643 -0.46533404  1.24330407  0.27355222  0.5314504 ]
[-1.20308357 -1.53466596 -1.24330407  1.72644778  3.4685496 ]
[ 1.59383285 -0.53466596  0.         -0.72644778 -2.93709921]
[ 2.50967345 -1.87032665  0.         -1.37656144 -1.36188794]
[-0.79691643  0.53466596  1.24330407  1.          2.        ]
```

In [13]:
```
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.

In [46]:
```
# Linear algebra for np
```

In [14]:
```
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
```

```
[[0.06043342 0.2278679  1.40367637]
 [0.12336507 1.53691424 0.14158836]]
[[0.06043342 0.12336507]
 [0.2278679  1.53691424]
 [1.40367637 0.14158836]]
[-0.41072136  0.21635507  0.16521881]
[0.25639282 0.30524357]
[[2.02588332 0.55641304]
 [0.55641304 2.39737159]]
[[0.01887114 0.20337238 0.10229602]
 [0.20337238 2.41402916 0.53746196]
 [0.10229602 0.53746196 1.99035461]]
```

In [15]:
```
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)
```

```
[[-1.07533891 -0.33347116 -0.59512284]
 [ 0.65209601 -0.31004385 -0.21264315]
 [-0.82481028  0.21457055  0.55633482]
 [-0.92260511 -0.69747543 -0.00681465]
 [ 0.68415461 -0.11171342  1.10839623]]
[[ 0.33565285 -0.20446278 -0.11187782]
 [-0.20446278  1.52087984 -0.12260865]
 [-0.11187782 -0.12260865  0.5794838 ]]
4.13389624135209
[4.02344909 1.6089282  0.63859336]
[[ 0.90902408  0.38733679  0.15377068]
 [ 0.18219706 -0.03752494 -0.98254573]
 [ 0.37480588 -0.9211743   0.10468269]]
```

2.2 Introduction to Pandas

Pandas provide two convenient data structures for storing and manipulating data--Series and DataFrame. A Series is similar to a one-dimensional array whereas a 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.

In [47]:
```
# pd on series
```

In [16]:
```
from pandas import Series

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

In [17]:
```
import numpy as np

s2 = Series(np.random.randn(6))  # creating a series from a numpy ndarray
print(s2)
print('Values=', s2.values)   # display values of the Series
print('Index=', s2.index)     # display indices of the Series
```

```
0   -2.516147
1    0.508866
2   -1.663254
3   -1.595799
4   -0.205336
5   -1.051907
dtype: float64
Values= [-2.51614662  0.50886619 -1.66325389 -1.59579891 -0.20533585 -1.0519068
4]
Index= RangeIndex(start=0, stop=6, step=1)
```

In [18]:
```
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='obj
ect')
```

In [19]:
```
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}

s4 = Series(capitals)   # creating a series from dictionary object
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')
```

In [20]:
```
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('s3.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
```

In [21]:
```
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
```

In [22]:
```
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
```

In [23]:
```
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
```

In [24]:
```
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).

In [48]:
```
# Dataframe manipulations using pd
```

In [25]:
```
from pandas import DataFrame

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

Out[25]:

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

In [26]:
```
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')
```

In [27]:
```
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
```

Out[27]:

	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

Creating DataFrame from a list of tuples.

In [28]:
```
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
```

Out[28]:

	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

Creating DataFrame from numpy ndarray

In [29]:
```
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
```

Out[29]:

	x1	x2	x3
0	-1.864885	-1.646344	0.030617
1	0.117893	0.102521	1.620343
2	1.893644	-1.017173	0.051040
3	0.038654	1.251772	-1.100382
4	1.530257	0.552807	-0.356653

The elements of a DataFrame can be accessed in many ways.

In [30]:
```
# accessing an entire column will return a Series object

print(data['x2'])
print(type(data['x2']))
```

```
0   -1.646344
1    0.102521
2   -1.017173
3    1.251772
4    0.552807
Name: x2, dtype: float64
<class 'pandas.core.series.Series'>
```

In [31]:
```
# 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 types
```

```
Row 3 of data table:
x1    1.893644
x2   -1.017173
x3    0.051040
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
```

In [32]:
```
# 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
```

In [33]:
```
print('carData2.shape =', carData2.shape)
print('carData2.size =', carData2.size)
```

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

In [34]:
```
# 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

In [49]:
```
# Dataframe mathmetical operations
```

In [35]:
```
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 -1.864885 -1.646344  0.030617
1  0.117893  0.102521  1.620343
2  1.893644 -1.017173  0.051040
3  0.038654  1.251772 -1.100382
4  1.530257  0.552807 -0.356653
Data transpose operation:
           0         1         2         3         4
x1 -1.864885  0.117893  1.893644  0.038654  1.530257
x2 -1.646344  0.102521 -1.017173  1.251772  0.552807
x3  0.030617  1.620343  0.051040 -1.100382 -0.356653
Addition:
         x1        x2        x3
0  2.135115  2.353656  4.030617
1  4.117893  4.102521  5.620343
2  5.893644  2.982827  4.051040
3  4.038654  5.251772  2.899618
4  5.530257  4.552807  3.643347
Multiplication:
          x1         x2         x3
0 -18.648852 -16.463440   0.306172
1   1.178931   1.025211  16.203427
2  18.936436 -10.171729   0.510396
3   0.386540  12.517722 -11.003820
4  15.302568   5.528075  -3.566534
```

In [36]:
```
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 -1.864885 -1.646344  0.030617
1  0.117893  0.102521  1.620343
2  1.893644 -1.017173  0.051040
3  0.038654  1.251772 -1.100382
4  1.530257  0.552807 -0.356653

data2 =
         x1        x2        x3
0 -0.027004 -1.484355 -0.410527
1  1.813162  0.487239  0.237339
2  0.945869 -0.787060  1.576802
3 -0.114726 -1.334311  0.134960
4  0.306308  0.466582 -0.405276

data + data2 =
         x1        x2        x3
0 -1.891889 -3.130699 -0.379910
1  1.931056  0.589760  1.857682
2  2.839513 -1.804233  1.627842
3 -0.076072 -0.082538 -0.965422
4  1.836565  1.019390 -0.761930

data * data2 =
         x1        x2        x3
0  0.050360  2.443758 -0.012569
1  0.213759  0.049952  0.384570
2  1.791139  0.800576  0.080479
3 -0.004435 -1.670253 -0.148508
4  0.468730  0.257930  0.144543
```

In [37]:
```
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

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  1.864885  1.646344  0.030617
1  0.117893  0.102521  1.620343
2  1.893644  1.017173  0.051040
3  0.038654  1.251772  1.100382
4  1.530257  0.552807  0.356653

Maximum value per column:
x1    1.893644
x2    1.251772
x3    1.620343
dtype: float64

Minimum value per row:
0   -1.864885
1    0.102521
2   -1.017173
3   -1.100382
4   -0.356653
dtype: float64

Sum of values per column:
x1    1.715562
x2   -0.756416
x3    0.244964
dtype: float64

Average value per row:
0   -1.160204
1    0.613586
2    0.309170
3    0.063348
4    0.575470
dtype: float64
```

```
Calculate max - min per column
x1    3.758529
x2    2.898116
x3    2.720725
dtype: float64

Calculate max - min per row
0    1.895502
1    1.517822
2    2.910817
3    2.352154
4    1.886910
dtype: float64
```

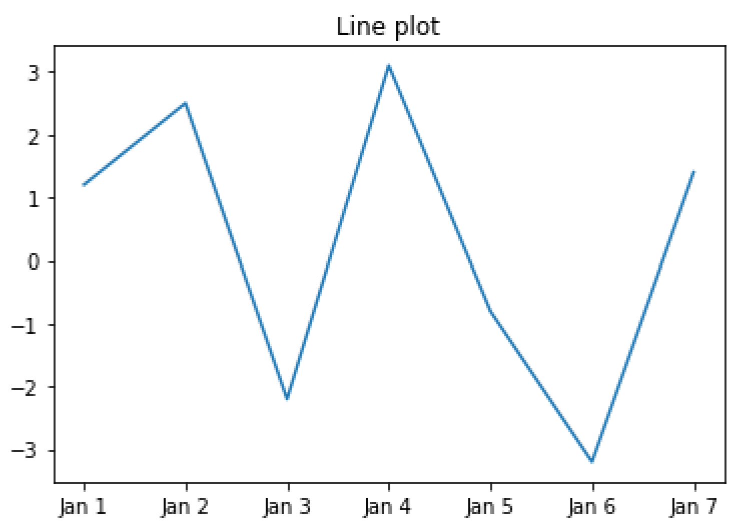
2.2.4 Plotting Series and DataFrame

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

In [50]:
```
# Plotting
```

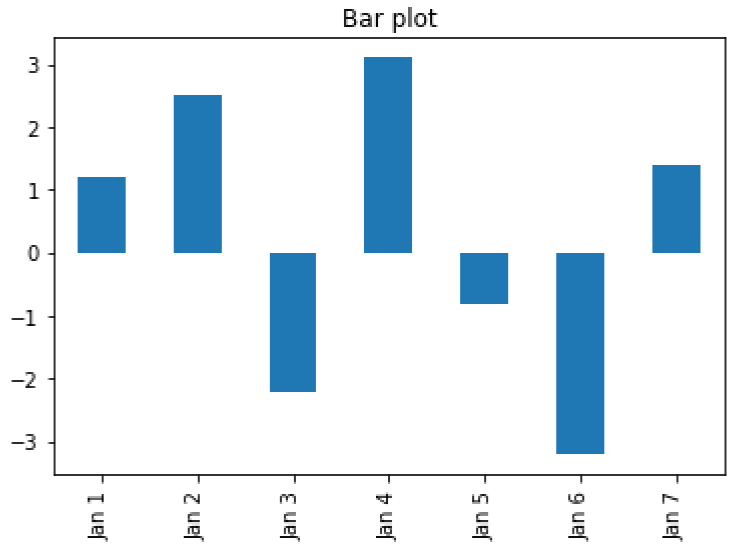
In [38]:
```
%matplotlib inline

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')
```

Out[38]:
```
<AxesSubplot:title={'center':'Line plot'}>
```



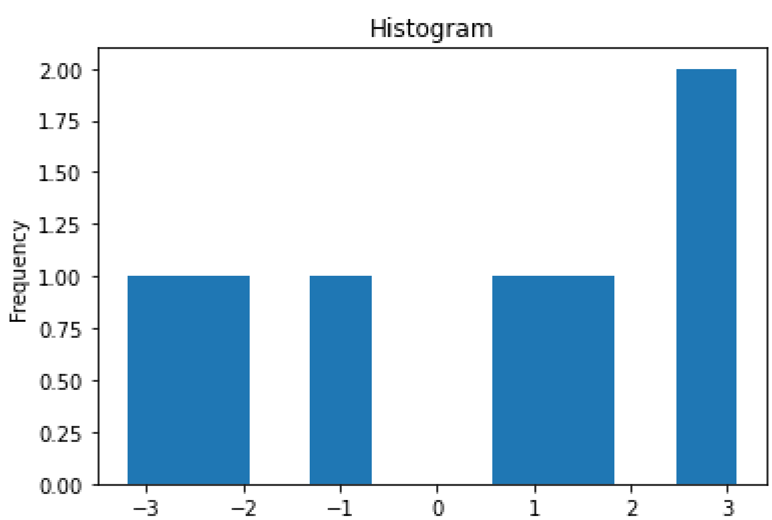
In [39]:
```
s3.plot(kind='bar', title='Bar plot')
```

Out[39]: <AxesSubplot:title={'center':'Bar plot'}>



In [40]:
```
s3.plot(kind='hist', title = 'Histogram')
```

Out[40]: <AxesSubplot:title={'center':'Histogram'}, ylabel='Frequency'>



In [41]:
```
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')
```

Out[41]: <AxesSubplot:title={'center':'Box plot'}>