Module 2: Introduction to Numpy and Pandas

The following tutorial contains examples of using the numpy and pandas library modules. The notebook can be downloaded from http://www.cse.msu.edu/~ptan/dmbook/tutorials/tutorial2/tutorial2.ipynb (http://www.cse.msu.edu/~ptan/dmbook/tutorials/tutorial2/tutorial2.ipynb). Read the step-by-step instructions below carefully. To execute the code, click on the cell and press the SHIFT-ENTER keys simultaneously.

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]:

```
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 [2]:

```
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 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
```

```
[ 0.90281327  0.75839203  0.66665486  0.83655218  0.08552842]
[ 0.23166823  0.46371994  0.52017539 -0.73508468 -0.14592176]
[-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.        ]
[  1.00000000e-03   1.00000000e-02   1.00000000e-01   1.00000000e+00
   1.00000000e+01   1.00000000e+02   1.00000000e+03]
```

In [3]:

```
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 [4]:

```
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 [5]:

```
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 [6]:

```
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 [7]:

```
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 [8]:

```
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 [9]:

```
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 [10]:

```
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 [11]:

```
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]
[-1.16419469 -0.37527889  0.40155879 -0.30558396 -0.48555089]
[-3.16419469 -1.37527889  0.40155879  0.69441604  1.51444911]
[-0.83580531 -0.62472111 -0.40155879  1.30558396  2.48555089]
[ 2.32838938  0.37527889  0.         -0.30558396 -0.97110177]
[ 1.71792572  2.66468491  0.         -3.27242304 -4.11903275]
[-1.16419469 -0.37527889  0.40155879  1.          2.        ]
```

In [12]:

```
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 = 0.34
Std deviation = 2.43277619193
Sum = 1.7
```

2.1.5 Numpy linear algebra

Numpy provides many functions to support linear algebra operations.

In [13]:

```
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
```

```
[[ 1.69507998  0.30902078  0.43156108]
 [-1.42763319  1.30035289 -0.27448127]]
[[ 1.69507998 -1.42763319]
 [ 0.30902078  1.30035289]
 [ 0.43156108 -0.27448127]]
[-0.37019872  0.70765478  0.70404352]
[-0.10499861  1.25546216]
[[ 3.15503496 -2.1365718 ]
 [-2.1365718   3.80439413]]
[[ 4.91143266 -1.332612    1.12338913]
 [-1.332612    1.78641149 -0.22356118]
 [ 1.12338913 -0.22356118  0.26158494]]
```

In [14]:

```
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)
```

```
[[-0.16909965  0.19093152  1.57951874]
 [-0.25412173 -0.2681879   0.584444  ]
 [ 1.41642911 -0.30983459 -0.69628628]
 [-0.51880255  0.77167516 -2.39867059]
 [-2.28891959  0.58906726 -0.02703302]]
[[ 0.42226645  1.02055275  0.17383389]
 [ 1.02055275  3.58154345  0.60519842]
 [ 0.17383389  0.60519842  0.21246448]]
19.2754740085
[ 0.25077991  8.15950504  9.41994804]
[[-0.27885001 -0.93385987  0.22393841]
 [-0.94613752  0.22720547 -0.23065442]
 [-0.16451888  0.27619452  0.94691611]]
```

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 [15]:

```
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 [16]:

```
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    0.541925
1    0.878919
2   -0.447277
3   -0.741945
4    0.115769
5   -2.019612
dtype: float64
Values= [ 0.54192501  0.87891863 -0.44727743 -0.74194521  0.11576869 -2.0196119 ]
Index= RangeIndex(start=0, stop=6, step=1)
```

In [17]:

```
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='objec
t')
```

In [18]:

```
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
```

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

In [19]:

```
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 [20]:

```
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 [21]:

```
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 [22]:

```
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 [23]:

```
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 [24]:

```
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[24]:

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

In [25]:

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

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

In [26]:

```
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[26]:

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

Creating DataFrame from a list of tuples.

In [27]:

```
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[27]:

	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 [28]:

```
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[28]:

	x1	x2	x3
0	-1.079832	-0.050927	-0.299053
1	0.018703	-1.187343	0.409459
2	1.059970	-0.857066	1.138166
3	0.614850	-0.075424	0.246003
4	0.467886	0.312978	-0.213227

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

In [29]:

```
# accessing an entire column will return a Series object

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

```
0   -0.050927
1   -1.187343
2   -0.857066
3   -0.075424
4    0.312978
Name: x2, dtype: float64
<class 'pandas.core.series.Series'>
```

In [30]:

```
# 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    1.059970
x2   -0.857066
x3    1.138166
Name: 2, dtype: float64
<class 'pandas.core.series.Series'>

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

In [31]:

```
# 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])
```

```
Accord
Taurus
carData2.iloc[1:3,1:3]=
     make   model
2   Honda  Accord
3  Toyota   Camry
```

In [32]:

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

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

In [33]:

```
# selection and filtering

print('carData2[carData2.MSRP > 25000]')
print(carData2[carData2.MSRP > 25000])
```

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

2.2.3 Arithmetic Operations

In [34]:

```
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 -1.079832 -0.050927 -0.299053
1  0.018703 -1.187343  0.409459
2  1.059970 -0.857066  1.138166
3  0.614850 -0.075424  0.246003
4  0.467886  0.312978 -0.213227
Data transpose operation:
           0         1         2         3         4
x1 -1.079832  0.018703  1.059970  0.614850  0.467886
x2 -0.050927 -1.187343 -0.857066 -0.075424  0.312978
x3 -0.299053  0.409459  1.138166  0.246003 -0.213227
Addition:
         x1        x2        x3
0  2.920168  3.949073  3.700947
1  4.018703  2.812657  4.409459
2  5.059970  3.142934  5.138166
3  4.614850  3.924576  4.246003
4  4.467886  4.312978  3.786773
Multiplication:
          x1         x2         x3
0 -10.798325  -0.509272  -2.990528
1   0.187029 -11.873425   4.094586
2  10.599699  -8.570661  11.381656
3   6.148502  -0.754238   2.460027
4   4.678856   3.129776  -2.132268
```

In [35]:

```
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 -1.079832 -0.050927 -0.299053
1  0.018703 -1.187343  0.409459
2  1.059970 -0.857066  1.138166
3  0.614850 -0.075424  0.246003
4  0.467886  0.312978 -0.213227

data2 =
         x1        x2        x3
0  0.588217 -2.021532 -0.619930
1  0.971340  1.946699  0.186294
2  0.435159  0.632938  0.870996
3  0.564977  0.341676 -0.102520
4  1.742437  1.604896 -1.580463

data + data2 =
         x1        x2        x3
0 -0.491615 -2.072459 -0.918983
1  0.990043  0.759356  0.595753
2  1.495128 -0.224128  2.009161
3  1.179827  0.266253  0.143483
4  2.210322  1.917874 -1.793690

data * data2 =
         x1        x2        x3
0 -0.635176  0.102951  0.185392
1  0.018167 -2.311398  0.076280
2  0.461255 -0.542470  0.991338
3  0.347376 -0.025771 -0.025220
4  0.815261  0.502297  0.336997
```

In [36]:

```
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  1.079832  0.050927  0.299053
1  0.018703  1.187343  0.409459
2  1.059970  0.857066  1.138166
3  0.614850  0.075424  0.246003
4  0.467886  0.312978  0.213227

Maximum value per column:
x1    1.059970
x2    0.312978
x3    1.138166
dtype: float64

Minimum value per row:
0   -1.079832
1   -1.187343
2   -0.857066
3   -0.075424
4   -0.213227
dtype: float64

Sum of values per column:
x1    1.081576
x2   -1.857782
x3    1.281347
dtype: float64

Average value per row:
0   -0.476604
1   -0.253060
2    0.447023
3    0.261810
4    0.189212
dtype: float64

Calculate max - min per column
x1    2.139802
x2    1.500320
x3    1.437218
dtype: float64

Calculate max - min per row
0    1.028905
1    1.596801
2    1.995232
3    0.690274
4    0.681112
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.

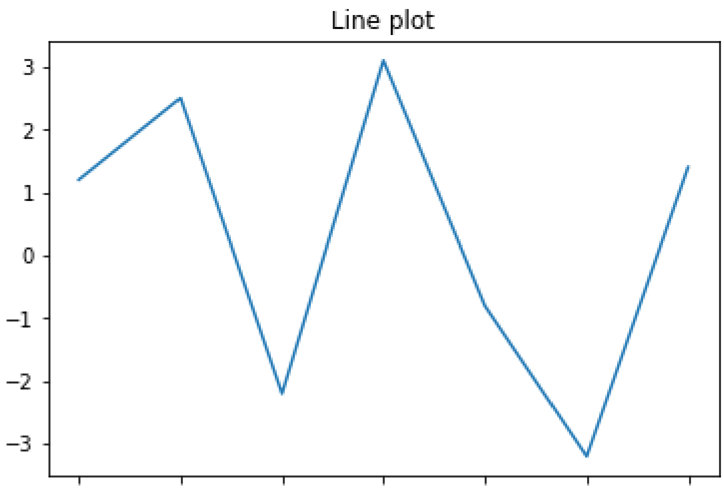
In [37]:

```
%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[37]:

<matplotlib.axes._subplots.AxesSubplot at 0x2871e6f0978>

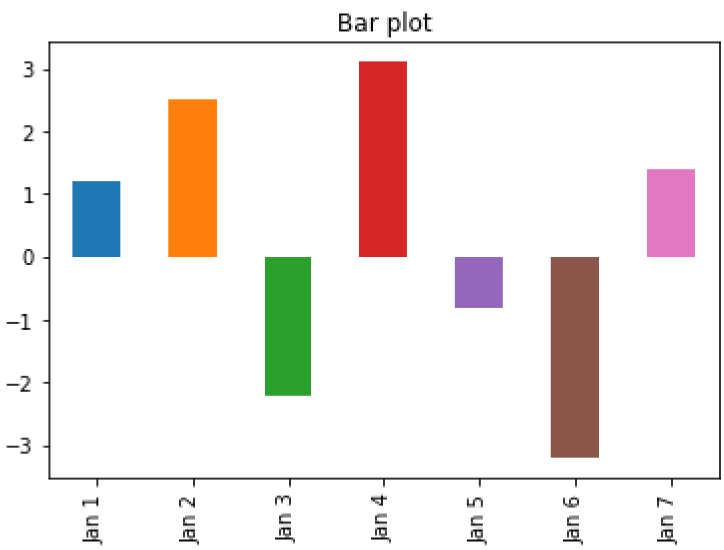


In [38]:

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

Out[38]:

<matplotlib.axes._subplots.AxesSubplot at 0x2871e796908>

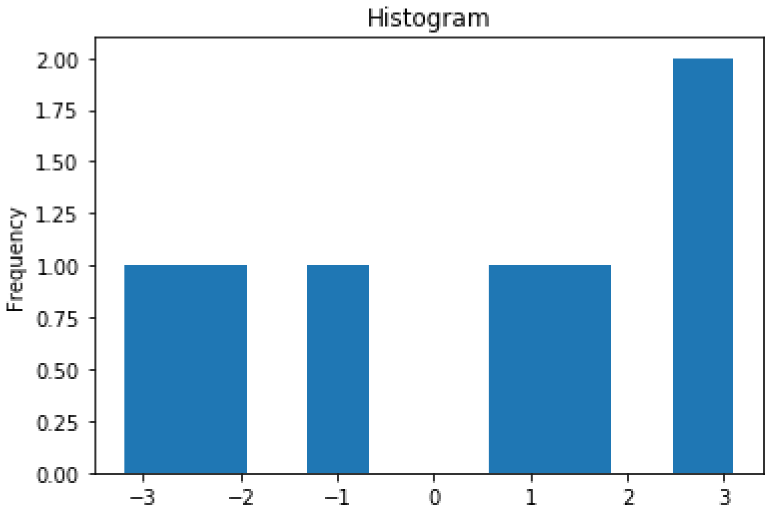


In [39]:

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

Out[39]:

<matplotlib.axes._subplots.AxesSubplot at 0x2871e720208>



In [40]:

```
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[40]:

<matplotlib.axes._subplots.AxesSubplot at 0x2871e894780>