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. 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 a tuple object as shown in the examples below. It is possible to create a 1-dimensional or multi-dimensional array from the list objects as well as tuples.

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, '\n')

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, '\n')

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 also built-in functions available in numpy to create the ndarrays.

In [2]:

```
print('Array of random numbers from a uniform distribution')
print(np.random.rand(5))      # random numbers from a uniform distribution between [0,1

print('\nArray of random numbers from a normal distribution')
print(np.random.randn(5))     # random numbers from a normal distribution

print('\nArray of integers between -10 and 10, with step size of 2')
print(np.arange(-10,10,2))    # similar to range, but returns ndarray instead of list

print('\n2-dimensional array of integers from 0 to 11')
print(np.arange(12).reshape(3,4))  # reshape to a matrix

print('\nArray of values between 0 and 1, split into 10 equally spaced values')
print(np.linspace(0,1,10))    # split interval [0,1] into 10 equally separated values

print('\nArray of values from 10^-3 to 10^3')
print(np.logspace(-3,3,7))    # create ndarray with values from 10^-3 to 10^3
```

```
Array of random numbers from a uniform distribution
[0.78208188 0.22571995 0.68350307 0.44186644 0.97712786]

Array of random numbers from a normal distribution
[-0.66811668 -0.15633771 -0.26815877 -1.50843193 -0.18443227]

Array of integers between -10 and 10, with step size of 2
[-10  -8  -6  -4  -2   0   2   4   6   8]

2-dimensional array of integers from 0 to 11
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Array of values between 0 and 1, split into 10 equally spaced values
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]

Array of values from 10^-3 to 10^3
[1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02 1.e+03]
```

In [3]:

```
print('A 2 x 3 matrix of zeros')
print(np.zeros((2,3)))        # a matrix of zeros

print('\nA 3 x 2 matrix of ones')
print(np.ones((3,2)))         # a matrix of ones

print('\nA 3 x 3 identity matrix')
print(np.eye(3))              # a 3 x 3 identity matrix
```

```
A 2 x 3 matrix of zeros
[[0. 0. 0.]
 [0. 0. 0.]]

A 3 x 2 matrix of ones
[[1. 1.]
 [1. 1.]
 [1. 1.]]

A 3 x 3 identity matrix
[[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]:

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

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

```
x = [1 2 3 4 5]
x + 1 = [2 3 4 5 6]
x - 1 = [0 1 2 3 4]
x * 2 = [ 2  4  6  8 10]
x // 2 = [0 1 1 2 2]
x ** 2 = [ 1  4  9 16 25]
x % 2 = [1 0 1 0 1]
1 / x = [1.         0.5        0.33333333 0.25       0.2       ]
```

In [5]:

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

print('x =', x)
print('y =', y)
print('x + y =', x + y)      # element-wise addition
print('x - y =', x - y)      # element-wise subtraction
print('x * y =', x * y)      # element-wise multiplication
print('x / y =', x / y)      # element-wise division
print('x // y =', x // y)    # element-wise integer division
print('x ** y =', x ** y)    # element-wise exponentiation
```

```
x = [ 2  4  6  8 10]
y = [1 2 3 4 5]
x + y = [ 3  6  9 12 15]
x - y = [1 2 3 4 5]
x * y = [ 2  8 18 32 50]
x / y = [2. 2. 2. 2. 2.]
x // y = [2 2 2 2 2]
x ** y = [     2     16    216   4096 100000]
```

2.1.3 Indexing and Slicing

There are various ways to select a subset of elements within a numpy array. Assigning a numpy array (or a subset of its elements) to another variable will simply pass a reference to the array instead of copying its values. To make a copy of an ndarray, you need to explicitly call the .copy() function.

In [6]:

```
x = np.arange(-5,5)
print('Before: x =', x)

y = x[3:5]     # y is a slice, i.e., pointer to a subarray in x
print('        y =', y)
y[:] = 1000    # modifying the value of y will change x
print('After : y =', y)
print('        x =', x, '\n')

z = x[3:5].copy()   # makes a copy of the subarray
print('Before: x =', x)
print('        z =', z)
z[:] = 500          # modifying the value of z will not affect x
print('After : z =', z)
print('        x =', x)
```

```
Before: x = [-5 -4 -3 -2 -1  0  1  2  3  4]
        y = [-2 -1]
After : y = [1000 1000]
        x = [  -5   -4   -3 1000 1000    0    1    2    3    4]

Before: x = [  -5   -4   -3 1000 1000    0    1    2    3    4]
        z = [1000 1000]
After : z = [500 500]
        x = [  -5   -4   -3 1000 1000    0    1    2    3    4]
```

There are many ways to access elements of an ndarray. The following example illustrates the difference between indexing elements of a list and elements of ndarray.

In [7]:

```
my2dlist = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]  # a 2-dim list
print('my2dlist =', my2dlist)
print('my2dlist[2] =', my2dlist[2])            # access the third sublist
print('my2dlist[:][2] =', my2dlist[:][2])      # can't access third element of each sub
# print('my2dlist[:,2] =', my2dlist[:,2])      # invalid way to access sublist, will ca

my2darr = np.array(my2dlist)
print('\nmy2darr =\n', my2darr)

print('my2darr[2][:] =', my2darr[2][:])      # access the third row
print('my2darr[2,:] =', my2darr[2,:])        # access the third row
print('my2darr[:][2] =', my2darr[:][2])      # access the third row (similar to 2d list
print('my2darr[:,2] =', my2darr[:,2])        # access the third column
print('my2darr[:2,2:] =\n', my2darr[:2,2:])     # access the first two rows & last two
```

```
my2dlist = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
my2dlist[2] = [9, 10, 11, 12]
my2dlist[:][2] = [9, 10, 11, 12]

my2darr =
 [[ 1  2  3  4]
```

```
 [ 5  6  7  8]
 [ 9 10 11 12]]
my2darr[2][:] = [ 9 10 11 12]
my2darr[2,:] = [ 9 10 11 12]
my2darr[:][2] = [ 9 10 11 12]
my2darr[:,2] = [ 3  7 11]
my2darr[:2,2:] =
 [[3 4]
 [7 8]]
```

Numpy arrays also support boolean indexing.

In [8]:

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

divBy3 = my2darr[my2darr % 3 == 0]
print('\nmy2darr[my2darr % 3 == 0] =', divBy3)          # returns all the elements di

divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0]
print('my2darr[2:, my2darr[2,:] % 3 == 0] =', divBy3LastRow)    # returns elements in t
```

```
my2darr =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

my2darr[my2darr % 3 == 0] = [ 3  6  9 12]
my2darr[2:, my2darr[2,:] % 3 == 0] = [[ 9 12]]
```

More indexing examples.

In [9]:

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

indices = [2,1,0,3]    # selected row indices
print('indices =', indices, '\n')
print('my2darr[indices,:] =\n', my2darr[indices,:])  # this will shuffle the rows of my

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

```
my2darr =
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
indices = [2, 1, 0, 3]

my2darr[indices,:] =
 [[ 7  8  9]
 [ 4  5  6]
 [ 1  2  3]
 [10 11 12]]

rowIndex = [0, 0, 1, 2, 3]
columnIndex = [0, 2, 0, 1, 2]

my2darr[rowIndex,columnIndex] = [ 1  3  4  8 12]
```

2.1.4 Numpy Arithmetic and Statistical Functions

Numpy provides many built-in mathematical functions available for manipulating elements of an ndarray.

In [10]:

```
y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4])
print('y =', y, '\n')

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

```
y = [-1.4  0.4 -3.2  2.5  3.4]

np.abs(y) = [1.4 0.4 3.2 2.5 3.4]
np.sqrt(abs(y)) = [1.18321596 0.63245553 1.78885438 1.58113883 1.84390889]
np.sign(y) = [-1.  1. -1.  1.  1.]
np.exp(y) = [ 0.24659696  1.4918247   0.0407622  12.18249396 29.96410005]
np.sort(y) = [-3.2 -1.4  0.4  2.5  3.4]
```

In [11]:

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

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

```
x = [-2 -1  0  1  2]
y = [-0.35232503 -0.25480014 -0.97984994 -0.60264099  0.61219323]

np.add(x,y) = [-2.35232503 -1.25480014 -0.97984994  0.39735901  2.61219323]
np.subtract(x,y) = [-1.64767497 -0.74519986  0.97984994  1.60264099  1.38780677]
np.multiply(x,y) = [ 0.70465006  0.25480014 -0.         -0.60264099  1.22438646]
np.divide(x,y) = [ 5.67657658  3.92464457 -0.         -1.65936272  3.26694235]
np.maximum(x,y) = [-0.35232503 -0.25480014  0.          1.          2.        ]
```

In [12]:

```
y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4])
print('y =', y, '\n')

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
```

```
y = [-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 [13]:

```
X = np.random.randn(2,3)                          # create a 2 x 3 random matrix
print('X =\n', X, '\n')
print('Transpose of X, X.T =\n', X.T, '\n')       # matrix transpose operation X^T

y = np.random.randn(3) # random vector
print('y =', y, '\n')

print('Matrix-vector multiplication')
print('X.dot(y) =\n', X.dot(y), '\n')             # matrix-vector multiplication  X * y

print('Matrix-matrix product')
print('X.dot(X.T) =', X.dot(X.T))         # matrix-matrix multiplication  X * X^T
print('\nX.T.dot(X) =\n', X.T.dot(X))      # matrix-matrix multiplication  X^T * X
```

```
X =
 [[-0.29018445  0.08793992 -0.45876154]
 [ 1.00046606 -0.43095091  1.20741224]]

Transpose of X, X.T =
 [[-0.29018445  1.00046606]
 [ 0.08793992 -0.43095091]
 [-0.45876154  1.20741224]]

y = [-1.01976031  0.73652894 -0.59432456]

Matrix-vector multiplication
X.dot(y) =
 [ 0.63334213 -2.05523813]

Matrix-matrix product
X.dot(X.T) = [[ 0.30240259 -0.88213178]
 [-0.88213178  2.64449533]]

X.T.dot(X) =
 [[ 1.08513934 -0.45667055  1.34110042]
 [-0.45667055  0.19345212 -0.56067885]
 [ 1.34110042 -0.56067885  1.66830646]]
```

In [14]:

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

C = X.T.dot(X)                # C = X^T * X is a square matrix
print('C = X.T.dot(X) =\n', C, '\n')

invC = np.linalg.inv(C)       # inverse of a square matrix
print('Inverse of C = np.linalg.inv(C)\n', invC, '\n')

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

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

```
X =
```

```
[[-0.33842574 -0.55622619 -0.61480117]
 [ 0.69915151  1.93348959 -0.52185243]
 [-0.37837744 -2.37828233  1.37406252]
 [ 0.90302793 -0.3630706  -0.25711704]
 [-1.02111109 -0.74045332  1.27246204]]

C = X.T.dot(X) =
 [[ 2.60464161  2.868154   -2.2082126 ]
 [ 2.868154   10.38408781 -4.78378344]
 [-2.2082126  -4.78378344  4.22362706]]

Inverse of C = np.linalg.inv(C)
 [[ 0.7027402  -0.05194686  0.30857299]
 [-0.05194686  0.20521502  0.20527251]
 [ 0.30857299  0.20527251  0.63058929]]

Determinant of C = np.linalg.det(C) = 29.84592354503518
Eigenvalues of C =
 [14.10732853  1.00968151  2.09534644]
Eigenvectors of C =
 [[-0.29739575 -0.71375981 -0.63411568]
 [-0.8316417  -0.13257114  0.53925595]
 [ 0.46896467 -0.68772947  0.55416633]]
```

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 a tabular representation akin to 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]:

```python
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('Series, s =\n', s, '\n')

print('s.values =', s.values)     # display values of the Series
print('s.index =', s.index)       # display indices of the Series
print('s.dtype =', s.dtype)       # display the element type of the Series
```

```
Series, s =
 0    3.1
1    2.4
2   -1.7
3    0.2
4   -2.9
5    4.5
dtype: float64

s.values = [ 3.1  2.4 -1.7  0.2 -2.9  4.5]
s.index = RangeIndex(start=0, stop=6, step=1)
s.dtype = float64
```

In [16]:
```python
import numpy as np

s2 = Series(np.random.randn(6))   # creating a series from a numpy ndarray
print('Series s2 =\n', s2, '\n')
print('s2.values =', s2.values)   # display values of the Series
print('s2.index =', s2.index)     # display indices of the Series
print('s2.dtype =', s2.dtype)     # display the element type of the Series
```

```
Series s2 =
 0    2.323357
1    2.531711
2   -0.035593
3   -0.773004
4    0.107579
5    1.234697
dtype: float64

s2.values = [ 2.32335721  2.5317115  -0.03559251 -0.77300359  0.10757853  1.23469683]
s2.index = RangeIndex(start=0, stop=6, step=1)
s2.dtype = float64
```

In [17]:
```python
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('Series s3 =\n', s3, '\n')
print('s3.values =', s3.values)   # display values of the Series
print('s3.index =', s3.index)     # display indices of the Series
print('s3.dtype =', s3.dtype)     # display the element type of the Series
```

```
Series s3 =
 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.values = [ 1.2  2.5 -2.2  3.1 -0.8 -3.2]
s3.index = Index(['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'], dtype='object')
s3.dtype = float64
```

In [18]:
```python
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}

s4 = Series(capitals)   # creating a series from dictionary object
print('Series s4 =\n', s4, '\n')
print('s4.values =', s4.values)   # display values of the Series
print('s4.index=', s4.index)      # display indices of the Series
print('s4.dtype =', s4.dtype)     # display the element type of the Series
```

```
Series s4 =
 MI       Lansing
CA    Sacramento
TX        Austin
MN       St Paul
dtype: object

s4.values = ['Lansing' 'Sacramento' 'Austin' 'St Paul']
s4.index= Index(['MI', 'CA', 'TX', 'MN'], dtype='object')
s4.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 =\n', s3, '\n')

# Accessing elements of a Series

print('s3[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])
```

```
s3 =
 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
```

There are various functions available to find the number of elements in a Series. Result of the function depends on whether null elements are included.

In [20]:

```
s3['Jan 7'] = np.nan
print('Series s3 =\n', s3, '\n')

print('Shape of s3 =', s3.shape)   # get the dimension of the Series
print('Size of s3 =', s3.size)     # get the number of elements of the Series
print('Count of s3 =', s3.count()) # get the number of non-null elements of the Series
```

```
Series s3 =
 Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
Jan 7    NaN
dtype: float64

Shape of s3 = (7,)
Size of s3 = 7
Count of s3 = 6
```

A boolean filter can be used to select elements of a Series

In [21]:
```
print(s3[s3 > 0])   # applying filter to select non-negative elements of the Series
```

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

Scalar operations can be performed on elements of a numeric Series

In [22]:
```
print('s3 + 4 =\n', s3 + 4, '\n')
print('s3 / 4 =\n', s3 / 4)
```

```
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
Jan 7    NaN
dtype: float64

s3 / 4 =
 Jan 1    0.300
Jan 2    0.625
Jan 3   -0.550
Jan 4    0.775
Jan 5   -0.200
Jan 6   -0.800
Jan 7      NaN
dtype: float64
```

Numpy functions can be applied to pandas Series.

In [23]:
```
print('np.log(s3 + 4) =\n', np.log(s3 + 4), '\n')    # applying log function to a nu
print('np.exp(s3 - 4) =\n', np.exp(s3 - 4), '\n')    # applying exponent function to a
```

```
np.log(s3 + 4) =
 Jan 1    1.648659
Jan 2    1.871802
Jan 3    0.587787
Jan 4    1.960095
Jan 5    1.163151
Jan 6   -0.223144
Jan 7         NaN
dtype: float64

np.exp(s3 - 4) =
 Jan 1    0.060810
Jan 2    0.223130
Jan 3    0.002029
Jan 4    0.406570
Jan 5    0.008230
Jan 6    0.000747
Jan 7         NaN
dtype: float64
```

The value_counts() function can be used for tabulating the counts of each discrete value in the Series.

In [24]:

```
colors = Series(['red', 'blue', 'blue', 'yellow', 'red', 'green', 'blue', np.nan])
print('colors =\n', colors, '\n')

print('colors.value_counts() =\n', colors.value_counts())
```

```
colors =
 0       red
1       blue
2       blue
3     yellow
4        red
5      green
6       blue
7        NaN
dtype: object

colors.value_counts() =
 blue      3
red       2
green     1
yellow    1
dtype: int64
```

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 [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 =', carData.index)         # print the row indices
print('carData.columns =', carData.columns)     # print the column indices
```

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

Inserting columns to an existing dataframe

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	-0.283443	-1.119286	0.484762
1	-0.415707	-0.278748	0.759381
2	-0.173857	-0.265252	-2.032578
3	1.452699	-1.646584	-0.073516
4	-1.011746	0.130857	-1.601343

There are many ways to access elements of a DataFrame object.

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

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

```
0   -1.119286
1   -0.278748
2   -0.265252
3   -1.646584
4    0.130857
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   -0.173857
x2   -0.265252
x3   -2.032578
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 =\n', carData2)

print('\ncarData2.iloc[1,2] =', carData2.iloc[1,2])                # retrieving second
print('carData2.loc[1,\'model\'] =', carData2.loc[1,'model'])    # retrieving second ro

# accessing a slice of the DataFrame

print('\ncarData2.iloc[1:3,1:3]=')
print(carData2.iloc[1:3,1:3])
```

```
carData2 =
      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

carData2.iloc[1,2] = 23570
carData2.loc[1,'model'] = Taurus

carData2.iloc[1:3,1:3]=
    model   MSRP
```

```
2  Accord  23570
3   Camry  23495
```

In [33]:

```
print('carData2 =\n', carData2, '\n')

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

```
carData2 =
      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

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

In [34]:

```
# selection and filtering

print('carData2 =\n', carData2, '\n')

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

```
carData2 =
      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

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 [35]:

```
print(data)

print('\nData transpose operation: data.T')
print(data.T)    # transpose operation

print('\nAddition: data + 4')
print(data + 4)    # addition operation

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

```
         x1        x2        x3
0 -0.283443 -1.119286  0.484762
1 -0.415707 -0.278748  0.759381
2 -0.173857 -0.265252 -2.032578
3  1.452699 -1.646584 -0.073516
4 -1.011746  0.130857 -1.601343

Data transpose operation: data.T
           0         1         2         3         4
x1 -0.283443 -0.415707 -0.173857  1.452699 -1.011746
```

```
x2 -1.119286 -0.278748 -0.265252 -1.646584  0.130857
x3  0.484762  0.759381 -2.032578 -0.073516 -1.601343

Addition: data + 4
         x1        x2        x3
0  3.716557  2.880714  4.484762
1  3.584293  3.721252  4.759381
2  3.826143  3.734748  1.967422
3  5.452699  2.353416  3.926484
4  2.988254  4.130857  2.398657

Multiplication: data * 10
          x1         x2         x3
0  -2.834429 -11.192858   4.847616
1  -4.157067  -2.787482   7.593806
2  -1.738572  -2.652520 -20.325780
3  14.526994 -16.465841  -0.735165
4 -10.117462   1.308571 -16.013427
```

In [36]:

```
print('data =\n', 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 -0.283443 -1.119286  0.484762
1 -0.415707 -0.278748  0.759381
2 -0.173857 -0.265252 -2.032578
3  1.452699 -1.646584 -0.073516
4 -1.011746  0.130857 -1.601343

data2 =
         x1        x2        x3
0 -0.345800 -2.505047 -0.156163
1 -0.099844 -0.821167 -1.373162
2  0.644743  0.202566  2.189374
3  2.760083 -0.404531  0.702017
4 -1.826406 -0.599395  0.149487

data + data2 =
         x1        x2        x3
0 -0.629243 -3.624333  0.328599
1 -0.515551 -1.099915 -0.613781
2  0.470885 -0.062686  0.156796
3  4.212782 -2.051115  0.628501
4 -2.838152 -0.468538 -1.451856

data * data2 =
         x1        x2        x3
0  0.098015  2.803864 -0.075702
1  0.041506  0.228899 -1.042753
2 -0.112093 -0.053731 -4.450074
3  4.009571  0.666094 -0.051610
4  1.847860 -0.078435 -0.239379
```

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  0.283443  1.119286  0.484762
1  0.415707  0.278748  0.759381
2  0.173857  0.265252  2.032578
3  1.452699  1.646584  0.073516
4  1.011746  0.130857  1.601343

Maximum value per column:
x1    1.452699
x2    0.130857
x3    0.759381
dtype: float64

Minimum value per row:
0   -1.119286
1   -0.415707
2   -2.032578
3   -1.646584
4   -1.601343
dtype: float64

Sum of values per column:
x1   -0.432054
x2   -3.179013
x3   -2.463295
dtype: float64

Average value per row:
0   -0.305989
1    0.021642
2   -0.823896
3   -0.089134
4   -0.827411
dtype: float64

Calculate max - min per column
x1    2.464446
x2    1.777441
x3    2.791959
dtype: float64
```

```
Calculate max - min per row
0    1.604047
1    1.175087
2    1.858721
3    3.099283
4    1.732200
dtype: float64
```

The value_counts() function can also be applied to a pandas DataFrame

In [38]:
```
objects = {'shape': ['circle', 'square', 'square', 'square', 'circle', 'rectangle'],
           'color': ['red', 'red', 'red', 'blue', 'blue', 'blue']}

shapeData = DataFrame(objects)
print('shapeData =\n', shapeData, '\n')

print('shapeData.value_counts() =\n', shapeData.value_counts().sort_values())
```

```
shapeData =
        shape color
0     circle   red
1     square   red
2     square   red
3     square  blue
4     circle  blue
5  rectangle  blue

shapeData.value_counts() =
 shape      color
circle     blue     1
           red      1
rectangle  blue     1
square     blue     1
           red      2
dtype: int64
```

2.2.4 Plotting Series and DataFrame

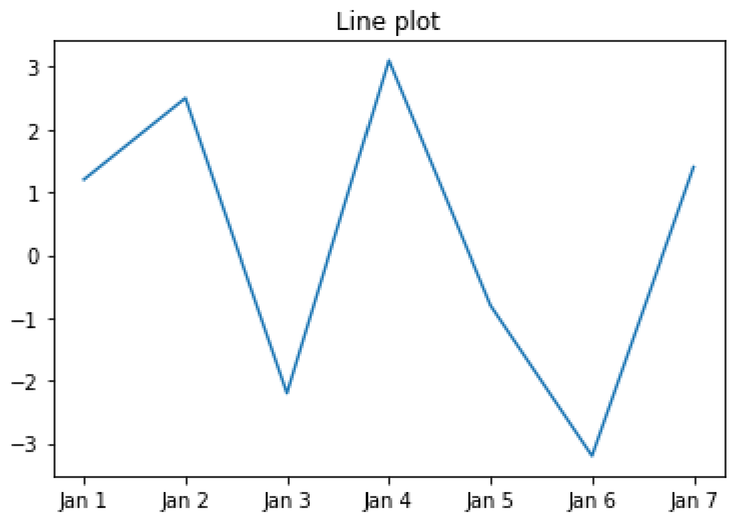
There are many built-in functions available to plot the data stored in a Series or a DataFrame.

(a) Line plot

In [39]:
```
%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')
```

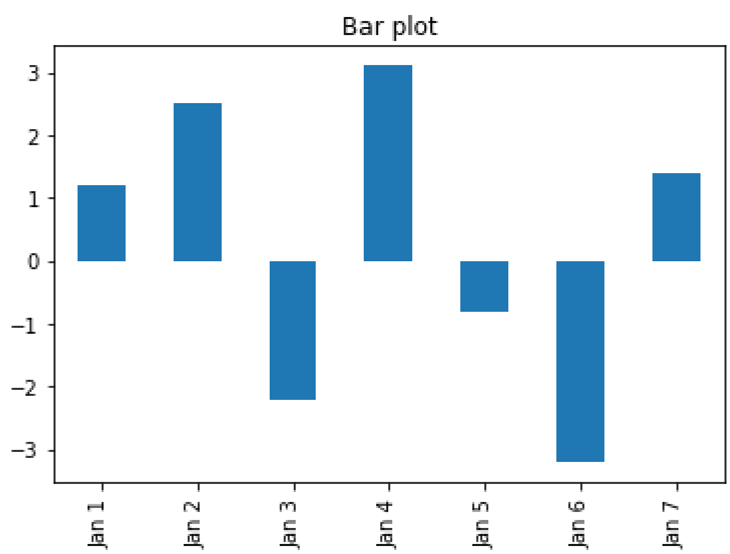
Out[39]: <AxesSubplot:title={'center':'Line plot'}>



(b) Bar plot

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

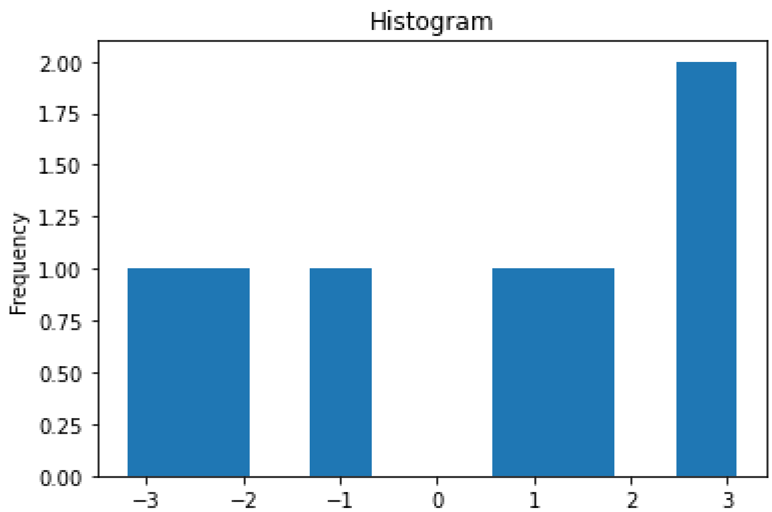
Out[40]: <AxesSubplot:title={'center':'Bar plot'}>



(c) Histogram

In [41]:
```
s3.plot(kind='hist', title = 'Histogram')
```

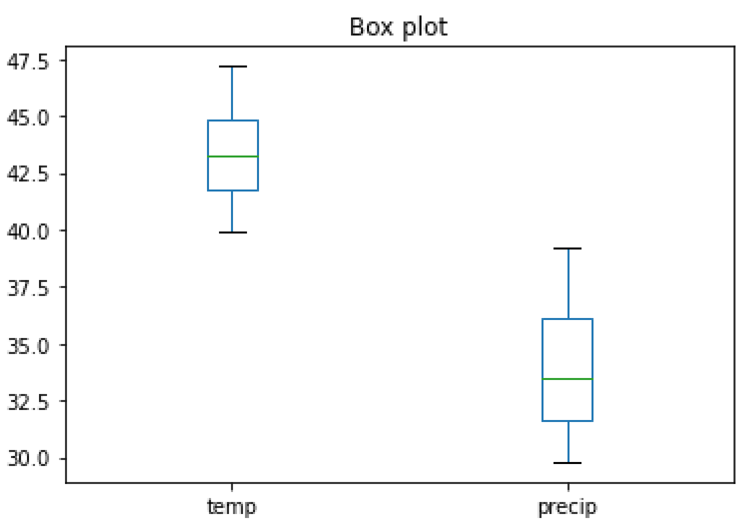
Out[41]: <AxesSubplot:title={'center':'Histogram'}, ylabel='Frequency'>



(d) Box plot

In [42]:
```
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[42]:
```
<AxesSubplot:title={'center':'Box plot'}>
```



(e) Scatter plot

In [43]:
```
print('weatherData =\n', weatherData)

weatherData.plot(kind='scatter', x='temp', y='precip')
```

```
weatherData =
    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
<AxesSubplot:xlabel='temp', ylabel='precip'>
```

Out[43]: