**Module 2: Introduction to Numpy and Pandas**

* 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.

* + 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)

### 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

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

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.]]

# **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 =', 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

In [5]:

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

# **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)

### 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 su

# print('my2dlist[:,2] =', my2dlist[:,2]) # invalid way to access sublist, will c

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 lis print('my2darr[:,2] =', my2darr[:,2]) # access the third column print('my2darr[:2,2:] =\n', my2darr[:2,2:]) # access the first two rows & last two

### 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 d

divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0]

print('my2darr[2:, my2darr[2,:] % 3 == 0] =', divBy3LastRow) # returns elements in

### 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 m

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])

# **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

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

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

# **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 \*

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

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)

# **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.

* + 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('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

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

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('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

In [18]:

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

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])

### 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 Serie

### 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

### 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)

### 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 num

print('np.exp(s3 - 4) =\n', np. exp(s3 - 4), '\n') # applying exponent function to

### 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())

* + 1. **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

In [26]:

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

### 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

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

### 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

### 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

### 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']))

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

In [32]:

# accessing a specific element of the DataFrame

print('carData2 =\n', carData2)

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

# accessing a slice of the DataFrame

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

In [33]:

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

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

In [34]:

# selection and filtering

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

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

In [35]:

## **Arithmetic Operations**

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

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))

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))

### 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())

## **Plotting Series and DataFrame**

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

1. 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')

In [40]:

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

In [41]:

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

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')

In [43]:

print('weatherData =\n', weatherData)

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