Tutorial_Graphics

September 28, 2018

---

1 1. GRAPHICS

- Visualize the input data and the results produced by a program, in the form of graphs, piecharts, histograms, and 3D plots.

1.1 2D Graphics

- Python library **matplotlib** provides several methods that facilitate drawing of 2D objects i.e. point, line, circle, rectangle, oval, polygon, and text.
- The library supports graphs, histograms, bar charts, pie charts, scatter plots, error charts, etc.

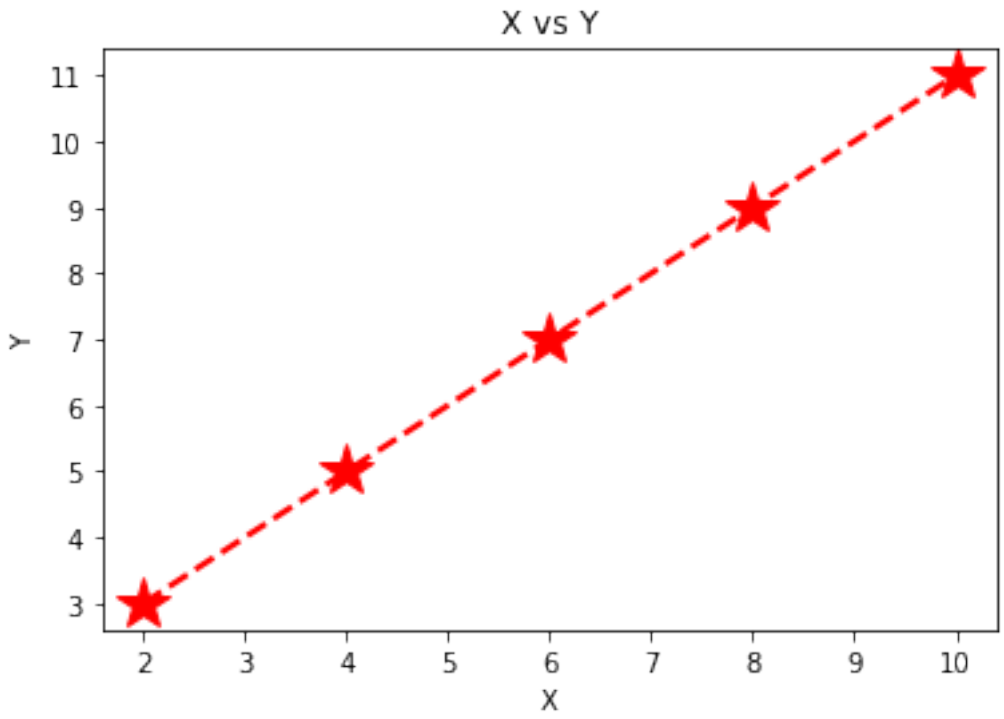
  Commands for installing library matplotlib

```
> pip install matplotlib
```

- **pyplot** module of **matplotlib** contains several functions for plotting figures and modifying or setting various properties such as labels for the figure and layout of the plot area.

1.2 Plotting a line

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        x = [2, 4, 6, 8, 10]
        y = [3, 5, 7, 9, 11]
        plt.plot(x,y, 'r*--',markersize = 20.5,linewidth = 2.2)
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('X vs Y')
        plt.show()
```

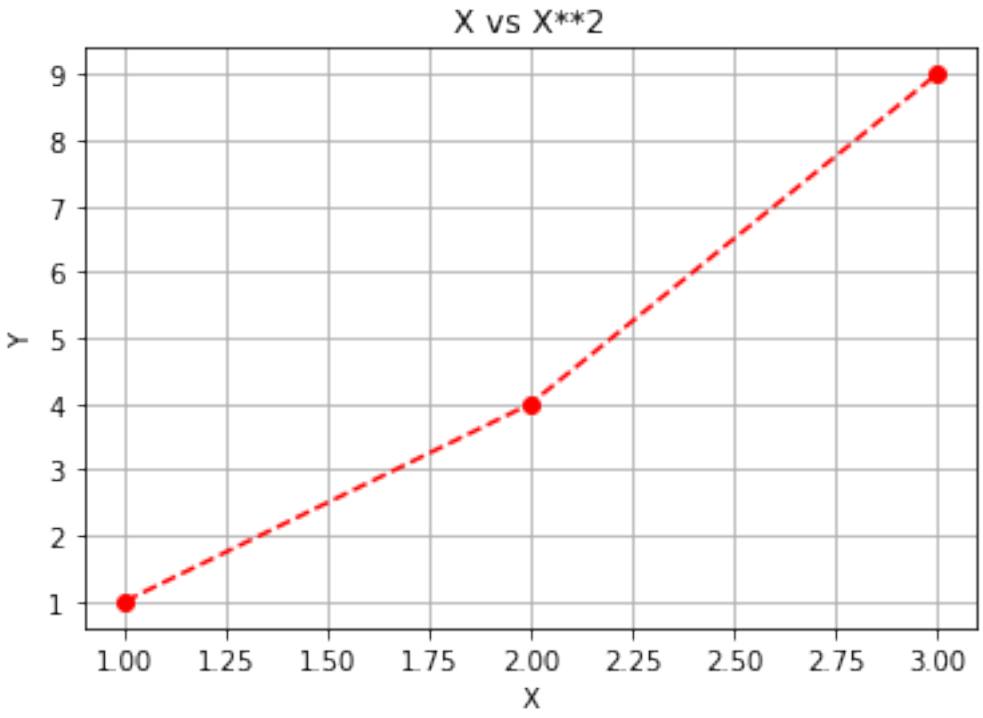


1.2.1 Plotting a function y = x**2

In [3]:
```
import matplotlib.pyplot as plt

def func(x):
    '''
    Objective: To compute x**2
    Input Parameters: x-numeric
    Return Value: numeric
    '''
    return x**2

def plotFunctions(xList, func):
    '''
    Objective: To plot functions f(x) = x**2
    Input Parameters: xList - list
    Return Value: None
    '''
    yList = []
    for x in xList:
        val = func(x)
        yList.append(val)
```

```
        plt.plot(xList, yList, 'ro--')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('X vs X**2')
        plt.grid()
        plt.show()

    def main():
        '''
        Objective: To plot two functions on same graph based on user
        input
        Input Parameter: None
        Return Value: None
        '''
        xList = eval(input('Enter the list of x: '))
        plotFunctions(xList, func)

    if __name__ == '__main__':
        main()
```

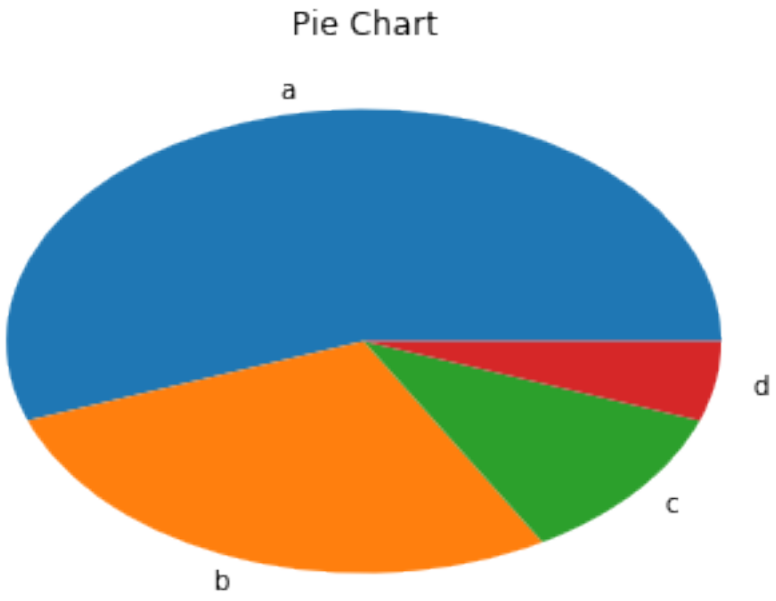
```
Enter the list of x: 1,2,3
```



1.2.2 Pie Chart

```
In [4]: import matplotlib.pyplot as plt
        %matplotlib inline
        def plotPieChart(data, labels):
            '''
            Objective: To plot a pie chart
            Input Parameters: data, labels - list
            Return Value: None
            '''
            plt.pie(data, labels = labels)
            plt.title('Pie Chart')
            plt.show()

        def main():
            '''
            Objective: To plot pie chart based on user input
            Input Parameter: None
            Return Value: None
            '''
            data = eval( input('Enter data to be plotted as pie chart: '))

            labels = eval(input('Enter the labels: '))
            plotPieChart(data, labels)

        main()
```

```
Enter data to be plotted as pie chart: 50,25,10,5
Enter the labels: 'a','b','c','d'
```



2 2. NumPy

- NumPy is the fundamental package for scientific computing with Python.
- NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

2.1 ndarray

- NumPy's array class is called ndarray.
- numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

2.2 An example to show the important attributes of an ndarray object

```
In [5]: import numpy as np
        a = np.arange(15).reshape(3, 5)
        a

Out[5]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])

In [11]: type(a)

Out[11]: numpy.ndarray

In [6]: a.shape

Out[6]: (3, 5)

In [7]: a.ndim

Out[7]: 2

In [8]: a.dtype.name

Out[8]: 'int64'

In [10]: a.size

Out[10]: 15
```

2.3 Array Creation

```
In [12]: #b=np.array(6,7,8)
         b=np.array([6,7,8])
         b

Out[12]: array([6, 7, 8])

In [13]: type(b)

Out[13]: numpy.ndarray
```

Array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
In [14]: b = np.array([(1.5,2,3), (4,5,6)])
         b

Out[14]: array([[1.5, 2. , 3. ],
                [4. , 5. , 6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
In [16]: c = np.array( [ [1+2j,2], [3,4] ], dtype=complex )
         c

Out[16]: array([[1.+2.j, 2.+0.j],
                [3.+0.j, 4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

```
In [ ]: np.zeros((3,4))

In [ ]: np.ones((2,3,4))

In [ ]: np.empty((3,4))
```

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```
In [17]: np.arange( 10, 30, 5 )

Out[17]: array([10, 15, 20, 25])
```

2.4 Printing arrays

One-dimensional arrays are printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
In [ ]: a = np.arange(6)                         # 1d array
        print(a)

In [ ]: b = np.arange(12).reshape(4,3)           # 2d array
        print(b)

In [ ]: c = np.arange(24).reshape(2,3,4)         # 3d array
        print(c)
```

Note: **ndarray.reshape** function returns the array in modified shape.

3 Basic operations

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

```
In [18]: a = np.array( [20,30,40,50] )
         b = np.arange( 4 )

         c = a-b
         c

Out[18]: array([20, 29, 38, 47])

In [19]: b**2

Out[19]: array([0, 1, 4, 9])

In [20]: a<35

Out[20]: array([ True,  True, False, False])
```

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
In [21]: A = np.array( [[1,1],
                        [0,1]] )
         B = np.array( [[2,0],
                        [3,4]] )
         A * B  #element wise product

Out[21]: array([[2, 0],
                [0, 4]])
```

```
In [22]: A @ B   #matrix product

Out[22]: array([[5, 4],
                [3, 4]])
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
In [26]: a = np.ones((2,3), dtype=int)
         b = np.random.random((2,3))
         a *= 3
         a

Out[26]: array([[3, 3, 3],
                [3, 3, 3]])

In [24]: b += a
         b

Out[24]: array([[3.88807806, 3.14164279, 3.03681247],
                [3.42442689, 3.41947876, 3.47766127]])

In [27]: a+=b #b is not automatically converted into integer type

         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-27-90979ab942a1> in <module>()
         ----> 1 a+=b #b is not automatically converted into integer type


         TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with ca
```

linspace function in numpy returns evenly spaced data points over an interval of time.

```
In [36]: b = np.linspace(0,9,4)
         b

Out[36]: array([0., 3., 6., 9.])

In [ ]: a.sum()
        a.min()
        a.max()
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
In [ ]: b = np.arange(12).reshape(3,4)

In [ ]: b.sum(axis=0)                          # sum of each column
        b.min(axis=1)                          # min of each row
        b.cumsum(axis=1)
```

3.1 Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions"(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
In [64]: B = np.arange(3)

         B=np.exp(B)
         #np.sqrt(B)
         print(B)
         C = np.array([2., -1., 4.])
         print(C)
         np.add(B, C)
```

```
[1.          2.71828183 7.3890561 ]
[ 2. -1.  4.]
```

```
Out[64]: array([ 3.        ,  1.71828183, 11.3890561 ])
```

3.2 Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
In [65]: a = np.arange(10)**3
         a
```

```
Out[65]: array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

```
In [ ]: a[2]
```

```
In [ ]: a[2:5]
```

```
In [66]: a[:6:2] = -1000
         a
```

```
Out[66]: array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,
                  729])
```

```
In [67]: a[ : :-1]
```

```
Out[67]: array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1,
                -1000])
```

```
In [ ]: for i in a:
        ...     print(i**(1/3.))       # why this output?
```

```
In [69]: def f(x,y):
         ...     return 10*x+y
```

```
In [70]: f(2,3)

Out[70]: 23
```

numpy.fromfunction construct an array by executing a function over each coordinate.

```
In [71]: b = np.fromfunction(f,(5,4),dtype=int)
         b

Out[71]: array([[ 0,  1,  2,  3],
                [10, 11, 12, 13],
                [20, 21, 22, 23],
                [30, 31, 32, 33],
                [40, 41, 42, 43]])

In [72]: b[2,3]

Out[72]: 23

In [73]: b[0:5, 1]                          # each row in the second column of b

Out[73]: array([ 1, 11, 21, 31, 41])

In [ ]: b[ : ,1]                           # equivalent to the previous example

In [ ]: b[1:3, : ]                         # each column in the second and third row of b

In [ ]: b[-1]                                     # the last row. Equivalent to b[-1,:]

In [75]: for i in b:
             for j in i:
                 print(j,end=" ")
             print("\n")

0 1 2 3

10 11 12 13

20 21 22 23

30 31 32 33

40 41 42 43
```