Introduction to Computation (CS2201)

Lecture 4

Kripabandhu Ghosh

CDS, IISER Kolkata

NumPy

NumPy

Nomenclature

NumPy stands for **Num**erical **Py**thon

Features

- Python library used for working with **arrays** (type: ndarray)
- Scientific calculations
- Functionalities for linear algebra, matrices etc.
- Ease of application
- Efficiency (e.g. over Lists)
- NumPy arrays used popularly in data science due to speed and resources

But we have **List** !!!

NumPy

Nomenclature

NumPy stands for **Num**erical **Py**thon

Features

- Python library used for working with **arrays** (type: ndarray)
- Scientific calculations
- Functionalities for linear algebra, matrices etc.
- Ease of application
- Efficiency (e.g. over Lists)
- NumPy arrays used popularly in data science due to speed and resources

But we have **List** !!!

NumPy arrays are faster than Lists

List Vs. NumPy array

List	**NumPy array**
Slow	Fast (50x faster)

List Vs. NumPy array

List	**NumPy array**
Slow	Fast (50x faster)
Stores items of different types	Stores items of the same type

List Vs. NumPy array

List	**NumPy array**
Slow	Fast (50x faster)
Stores items of different types	Stores items of the same type
Takes more memory	Takes less memory

Using NumPy: import

Code

```
import numpy
a = numpy.array([1, 2, 3, 4, 5]) #array of numpy
print(numpy.sum(a))
```

Output

15

Interpretation

Accessing **array** and **sum** functions in numpy package

Using NumPy: import with alias

Code

```
import numpy as np
a = np.array([1, 2, 3, 4, 5]) #array of numpy
print(np.sum(a))
```

Output

15

Interpretation

Accessing **array** and **sum** functions in numpy package using the alias **np**

Using NumPy: import everything

Code

```
from numpy import *
a = array([1, 2, 3, 4, 5]) #array of numpy
print(sum(a))
```

Output

15

Advantages

- Accessing **array** and **sum** functions in numpy package without **np/numpy**
- Importing all the functions and classes of the package numpy into the namespace (system with unique name for every object)

Using NumPy: import everything (contd.)

Code

```
from numpy import *
def sum(a, b):
   print (a+b)
a = array([1, 2, 3, 4, 5]) #array of numpy
print(sum(a))
```

Output

TypeError: sum() takes exactly 2 arguments (1 given)

Using NumPy: import everything (contd.)

Code

```
from numpy import *
def sum(a, b):
   print (a+b)
a = array([1, 2, 3, 4, 5]) #array of numpy
print(sum(a))
```

Output

TypeError: sum() takes exactly 2 arguments (1 given)

User-defined function **sum** clashing with **sum** of numpy !!!

Disadvantages

- Namespace is polluted with all (possibly unnecessary) functions and classes of a package
- Chance of clashing with the used defined functions and classes
- User need to be aware of all possible functions and classes of the package before defining one of own

NumPy array: creation

Create from List

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

Output

```
[1 2 3 4 5]
<type 'numpy.ndarray'>
```

Create from Tuple

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
print(type(arr))
```

Output

```
[1 2 3 4 5]
<type 'numpy.ndarray'>
```

We can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray

NumPy array: dimension

0-dimensional array (scalars)

arr0d = np.array(50)

1-dimensional array (array of 0-dimensional arrays (scalars))

arr1d = np.array([1, 2, 3, 4, 5])

2-dimensional array (array of 1-dimensional arrays)

arr2d = np.array([[1, 2], [4, 5]])

3-dimensional array (array of 2-dimensional arrays)

arr3d = np.array([[[1, 2], [4, 5]], [[6, 7], [8, 9]]])

- **ndarray.ndim**: Number of array dimensions
- **ndarray.shape**: Tuple of array dimensions

NumPy array: dimension (contd.)

Example

```
arr0d = np.array(50)
arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2], [4, 5]])
arr3d = np.array([[[1, 2], [4, 5]], [[6, 7], [8, 9]]])
print("The dimensions:")
print(arr0d.ndim)
print(arr1d.ndim)
print(arr2d.ndim)
print(arr3d.ndim)
print("The shapes:")
print(arr0d.shape)
print(arr1d.shape)
print(arr2d.shape)
print(arr3d.shape)
```

Output

```
The dimensions:
0
1
2
3
The shapes:
()
(5,)
(2, 2)
(2, 2, 2)
```

NumPy array: accessing the elements

1-D array

```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d[0])
```

Output

```
1
```

Using the index number i ($i = 0, 1, \ldots,$ num of elements - 1)

NumPy array: accessing the elements (contd.)

2-D array

```
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d[0, 1]) #prints element (0, 1)
print(arr2d[0]) #prints 0th row
```

Output

```
2
[1 2 3]
```

Using the comma separated integers i, j

- i: dimension (starting from 0)
- j: index (starting from 0)

NumPy array: accessing the elements (contd.)

3-D array

```
import numpy as np
arr3d = np.array(
[
[
[1, 2, 3 ], [4, 5, 6]
],

[
[6, 7, 4], [8, 9, 10]
]

]
)

print(arr3d[1, 1, 1]) #print element (1,1,1)
print(arr3d[1, 1]) #print (1,1)the array
print(arr3d[1]) #print 1st dimension
```

Output

```
9
[8 9 10]
[[ 6 7 4]
[ 8 9 10]]
```

NumPy array: accessing the elements (contd.)

Using the comma separated integers i, j, k

- i: first dimension (starting from 0)
- j: second dimension (starting from 0)
- k: index (starting from 0)

NumPy array: accessing the elements (contd.)

3-D array

```
import numpy as np
arr3d = np.array(
[
[
[1, 2, 3 ], [4, 5, 6]
],

[
[6, 7, 4], [8, 9, 10]
]

]
)

print(arr3d[1, 1, 1]) #print element (1,1,1)
print(arr3d[1, 1]) #print (1,1)the array
print(arr3d[1]) #print 1st dimension
```

Output

```
9
[8 9 10]
[[ 6 7 4]
[ 8 9 10]]
```

NumPy array: accessing the elements (contd.)

Negative indexing

```
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d[1, -1]) #element (1, 2)
```

Output

```
6
```

NumPy array: accessing the elements (contd.)

Slicing: 1-D

```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d[1:4]) #start to end index - 1
print(arr1d[3:]) #start to all
print(arr1d[0:3:2]) #start to end -1 with 2-length steps
```

Output

```
[2 3 4]
[4 5]
[1 3]
```

NumPy array: accessing the elements (contd.)

Slicing: 2-D

```
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d[1, 0:2]) #dimension, start:end-1
print(arr2d[0:2, 1:3]) #dimension_start:dimension_end-1, index_start:index_end-1
```

Output

```
[4 5]
[[2 3]
[5 6]]
```

NumPy array: accessing the elements (contd.)

Slicing: 3-D

```
import numpy as np
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[6, 7, 4], [8, 9, 10]]])
print(arr3d[0:2, 0:2, 2:3]) #dimension1_start:dimension1_end-1, dimension2_start:dimension2_end-1, index_start:index_end-1
```

Output

```
[[[ 3]
[ 6]]

[[ 4]
[10]]]
```

NumPy array: accessing the elements (contd.)

Get specific rows and columns

```
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d[0,:]) #0th row
print(arr2d[:,1]) #1st column
```

Output

```
[1 2 3]
[2 5]
```

- arr2d[r,:] : gets the r^{th} row (r starts from 0)
- arr2d[:,c] : gets the c^{th} column (c starts from 0)

NumPy array: operations

Applying a function on a list

```
def f(x):
    return x**2
x_list = [1.0, 2.5, 4.0, 6.0]
y_list = []
for e in x_list:
    y_list.append(f(e))
print(y_list)
```

Output

```
[1.0, 6.25, 16.0, 36.0]
```

NumPy array: operations (contd.)

Applying a function on a NumPy array

```
def f(x):
    return x**2

import numpy as np
x = np.array([1.0, 2.5, 4.0, 6.0])
y = f(x)
print(y)
```

Output

```
[ 1. 6.25 16. 36. ]
```

NumPy array: operations (contd.)

Elementwise operations

```
import numpy as np
x = np.array([1.0, 2.5, 4.0, 6.0])
y = x+2
print(y)
y = x*2
print(y)
print(x+y) #Note: you get a concatenated result for a list
y = x**2
print(y)
```

Output

```
[3.  4.5 6.  8. ]
[ 2.  5.  8.  12.]
[ 3.  7.5 12.  18. ]
[ 1.  6.25 16.  36. ]
```

NumPy array: operations (contd.)

matrix multiplications

```
import numpy as np
arr1 = np.array([[1, 2], [4, 5]])
arr2 = np.array([[3, 3], [1,1]])
print("Elementwise:")
print(np.multiply(arr1, arr2))
print("Matrix product:")
print(np.matmul(arr1, arr2))
```

Output

```
Elementwise:
[[[3 6]]
[4 5]]
Matrix product:
[[[ 5 5 ]]
[17 17]]
```

User-defined functions applied on NumPy arrays

Simple functions

```
def f(x):
    return 3*x - 5

import numpy as np
a = np.array([1, 2, 3, 4])
b = f(a)
print(b)
```

Output

```
[-2 1 4 7]
```

User-defined functions applied on NumPy arrays

Complex functions

```
def fact(x):
    if x == 0 or x == 1:
        return 1
    else:
        f = 1
        for i in range(2, x+1):
            f = f * i
    return f

import numpy as np
a = np.array([1, 2, 3, 4])
b = fact(a)
print(b)
```

Output

```
Traceback (most recent call last):
File "test3.py", line 63, in <module>
y = fact(a)
File "test3.py", line 44, in fact
if x == 0 or x == 1:
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

User-defined functions applied on NumPy arrays: universal functions

Complex functions

```
def fact(x):
    if x == 0 or x == 1:
        return 1
    else:
        f = 1
        for i in range(2, x+1):
            f = f * i
    return f

import numpy as np
a = np.array([1, 2, 3, 4])
calcFact = np.frompyfunc(fact, 1, 1)
y = calcFact(a)
```

Output

```
[1 2 6 24]
```

numpy.frompyfunc

Format

numpy.frompyfunc(func, nin, nout, ...)

Utility

Takes an arbitrary Python function and returns a NumPy ufunc (A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion)

Parameters

- **func** : Python function
- **nin**: (int) The number of input arguments of func
- **nout**: (int) The number of objects returned by func

Returns

ufunc (Returns a NumPy universal function (ufunc) object)

NumPy array: universal functions

sum: 1-D

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(np.sum(a)) # add all the elements
```

Output

```
15
```

NumPy array: universal functions (contd.)

sum: 2-D

```
import numpy as np
a = np.array([[1, 2, 3], [1, 1, 1], [1, 2, 0]])
print(np.sum(a))
print(np.sum(a, axis = 0)) #columnwise sum
print(np.sum(a, axis = 1)) #rowwise sum
print(np.sum(a, axis = 1, keepdims=True)) #Input dimension maintained
print(a.ndim)
print(np.sum(a, axis = 1).ndim)
print(np.sum(a, axis = 1, keepdims=True).ndim)
```

Output

```
12
[3 5 4]
[6 3 3]
[[6]
[3]
[3]]
2
1
2
```

Similar function

prod

NumPy array: universal functions (contd.)

add

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2, 3])
newarr = np.add(arr1, arr2) # between two arguments
newarr1 = np.sum([arr1, arr2], axis = 0)
print(newarr)
print(newarr1)
```

Output

```
[2 4 6]
[2 4 6]
```

Other binary functions (like add)

subtract, multiply, power, remainder

NumPy array: universal functions (contd.)

Set operations

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 4, 5])
newarr = np.union1d(arr1, arr2)
print(newarr)
newarr = np.intersect1d(arr1, arr2, assume_unique=True) #To speed up
print(newarr)
newarr = np.setdiff1d(arr1, arr2, assume_unique=True)
print(newarr)
```

Output

```
[1 2 3 4 5]
[1]
[2 3]
```

NumPy array: Solving a system of equations

$$x + y = 2$$
$$x + 3y = 6$$

System of linear equations

$$AX = B$$

- $$A = \begin{bmatrix} 1 & 1 \\ 1 & 3 \end{bmatrix}$$
- $$X = \begin{bmatrix} x \\ y \end{bmatrix}$$
- $$B = \begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

NumPy array: Solving a system of equations (contd.)

Code

```
import numpy as np
A = np.array([[1, 1], [1, 3]])
B = np.array([2, 6])
print(np.linalg.inv(A).dot(B))
print(np.linalg.solve(A, B))
```

Output

```
[0. 2.]
[0. 2.]
```

NumPy array: more on creation

Using linspace

- **Format**: numpy.linspace(start, stop, num, endpoint, retstep, ...)
- **Characteristics**: Returns *num* evenly spaced samples, calculated over the interval [start, stop]
  - **start**: The starting value of the sequence
  - **stop**: The end value of the sequence, unless endpoint is set to False
  - **num**: Number of samples to generate. Default is 50. Must be non-negative.
  - **endpoint**: If True, stop is the last sample. Otherwise, it is not included. Default is True.
  - **retstep**: If True, return (samples, step), where step is the spacing between samples.

Code

```
import numpy as np
a = np.linspace(0, 5, 5) #Evenly spaced 5 points in the interval [0, 5]
print(a)
a, h = np.linspace(0, 5, 5, retstep=True) #Evenly spaced 5 points in the interval [0, 5], returning the step-size
print(a)
print(h)
```

Output

```
[0. 1.25 2.5 3.75 5. ]
[0. 1.25 2.5 3.75 5. ]
1.25
```

NumPy array: more on creation (contd.)

Using arange

- **Format**: numpy.arange(start, stop, step ....)
- **Characteristics**: Return evenly spaced values within a given interval.
  - **start**: Start of interval. The interval includes this value. The default start value is 0.
  - **stop**: End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.
  - **step**: Spacing between values. The default step size is 1. If step is specified as a position argument, start must also be given.

Code

```
import numpy as np
a = np.arange(0, 1, 0.1)
print(a)
```

Output

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

NumPy array: more on creation (contd.)

Using zeros

- **Format**: numpy.zeros(shape, dtype, ...)
- **Characteristics**: Return a new array of given shape and type, filled with zeros.
    - **shape**: Shape of the new array, e.g., (2, 3) or 2.
    - **dtype**: The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64

Code

```
import numpy as np
a = np.zeros(5, int)
b = np.zeros((3,5), int)
print(a)
print(b)
```

Output

```
[0 0 0 0 0]
[[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]]
```

NumPy array: more on creation (contd.)

Using ones

- **Format**: numpy.ones(shape, dtype, ...)
- **Characteristics**: Return a new array of given shape and type, filled with ones.
  - **shape**: Shape of the new array, e.g., (2, 3) or 2.
  - **dtype**: The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64

Code

```
import numpy as np
a = np.ones(5, int)
b = np.ones((3,5), int)
print(a)
print(b)
```

Output

```
[1 1 1 1 1]
[[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 1]]
```

NumPy array: more on creation (contd.)

Using zeros_like and ones_like

- **Formats**
  - numpy.zeros_like(a, dtype...)
  - numpy.ones_like(a, dtype, ...)
- **Characteristics**: Return an array of zeros/ones with the same shape and type as a given array.
  - **a**: The shape and data-type of *a* define these same attributes of the returned array.
  - **dtype**: Overrides the data type of the result.

Code

```
import numpy as np
a = np.arange(5)
print(a)
b = np.zeros_like(a, dtype=float)
c = np.ones_like(a, dtype=float)
print(b)
print(c)
```

Output

```
[0 1 2 3 4]
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
```

NumPy array: Creating an Alias

- Makes another array with the same content and same id (doesn't make the direct copy of the original array)
- It represents the reference to the original array.
- Changes made on this reference are also reflected in the original array.

Code

```
import numpy as np
a = np.array([1, 2, 3, 4])
print("After aliasing:")
b = a
print("a:" + str(a))
print("b:" + str(b))
a[0] = 55
b[1] = 25
print("After modification:")
print("a:" + str(a))
print("b:" + str(b))
print("Identifier of a:" + str(id(a))) #Universal identifier (similar to pointer)
print("Identifier of b:" + str(id(b)))
```

Output

```
After aliasing:
a:[1 2 3 4]
b:[1 2 3 4]
After modification:
a:[55 25 3 4]
b:[55 25 3 4]
Identifier of a:140129140201472
Identifier of b:140129140201472
```

NumPy array: Creating a Copy

- It returns the deep copy of the original array which doesn't share any memory with the original array.
- The modification made to the deep copy of the original array doesn't reflect in the original array.

Code

```
import numpy as np
a = np.array([1, 2, 3, 4])
print("After copying:")
b = a.copy()
print("a:" + str(a))
print("b:" + str(b))
a[0] = 5
b[1] = 99
print("After modification:")
print("a:" + str(a))
print("b:" + str(b))
print("Identifier of a:" + str(id(a))) #Universal identifier (similar to pointer)
print("Identifier of b:" + str(id(b)))
print("Base of a: " + str(a.base))
print("Base of b: " + str(b.base))
```

NumPy array: Creating a Copy (contd.)

Output

```
After copying:
a:[1 2 3 4]
b:[1 2 3 4]
After modification:
a:[5 2 3 4]
b:[ 1 99 3 4]
Identifier of a:140496274626560
Identifier of b:140496274626800
Base of a: None
Base of b: None
```

NumPy array: Creating a View

- It returns another array with the same content as the original one but at different memory locations.
- The modification made to the view of the original array reflects in the original array.

Code

```
import numpy as np
a = np.array([1, 2, 3, 4])
print("After view:")
b = a.view()
print("a:" + str(a))
print("b:" + str(b))
a[0] = 1
b[1] = -99
print("After modification:")
print("a:" + str(a))
print("b:" + str(b))
print("Identifier of a:" + str(id(a))) #Universal identifier (similar to pointer)
print("Identifier of b:" + str(id(b)))
print("Base of a: " + str(a.base))
print("Base of b: " + str(b.base))
```

NumPy array: Creating a View (contd.)

Output

After view:
a:[1 2 3 4]
b:[1 2 3 4]
After modification:
a:[1 -99 3 4]
b:[1 -99 3 4]
Identifier of a:139788357048320
Identifier of b:139788357048560
Base of a: None
Base of b: [1 -99 3 4]

NumPy array: Alias vs. Copy vs. View

Feature	Alias	Copy	View
Memory location	Same	Different	Different
Modification dependency	Exists	Does not exist	Exists

NumPy array: Reshape

- Changing the shape (number of elements in each dimension) of an array
- Add or remove dimensions or change number of elements in each dimension (assuming compatibility)

1-D to 2-D

```
a = np.array([1, 2, 3, 4, 5, 6])
b = a.reshape(2, 3) # Convert to a 2 x 3 matrix
print(a)
print(b)
```

Output

```
[1 2 3 4 5 6]
[[1 2 3]
[4 5 6]]
```

NumPy array: Reshape (contd.)

1-D to 3-D

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
b = a.reshape(2, 2, 2) # Convert to a 2 x 2 x 2 matrix
print(a)
print(b)
```

Output

```
[1 2 3 4 5 6 7 8]
[[[1 2]
[3 4]]

[[5 6]
[7 8]]]
```

NumPy array: Reshape (Unknown dimension)

On assigning -1, the one unknown dimension is automatically inferred

Code

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
b = a.reshape(2, 2, -1) # The third dimension is unknown
print(a)
print(b)
```

Output

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

NumPy array: Reshape (Flattening)

Convert a higher dimension array to 1-D

Code

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = a.reshape(-1) #Convert to 1-D; b = a.flatten() also works
print(a)
print(b)
```

Output

```
[[1 2 3]
[4 5 6]]
[1 2 3 4 5 6]
```

NumPy array: Reshape produces view

Code

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
b = a.reshape(2, 4)
print("Before change:")
print(a)
print(b)
print("Address of a: " + str(id(a)))
print("Address of b: " + str(id(b)))
a[0] = 5
print("After change:")
print(a)
print(b)
print("Base of b: " + str(b.base))
```

Output

```
Before change: [1 2 3 4 5 6 7 8]
[[1 2 3 4]
[5 6 7 8]]
Address of a: 139800965244848
Address of b: 139800965245088
After change:
[5 2 3 4 5 6 7 8]
[[5 2 3 4]
[5 6 7 8]]
Base of b: [5 2 3 4 5 6 7 8]
```

Matplotlib

Matplotlib

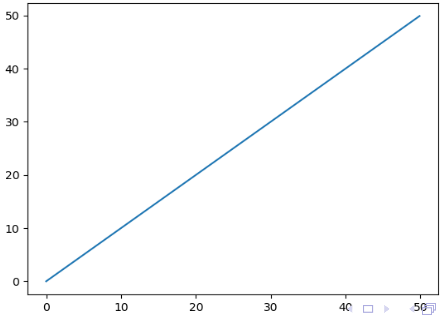
Introduction

- Graph plotting library used for data visualization
- Uses NumPy
- Matplotlib along with NumPy can be considered as the open source equivalent of MATLAB

Matplotlib: the first graph

Code

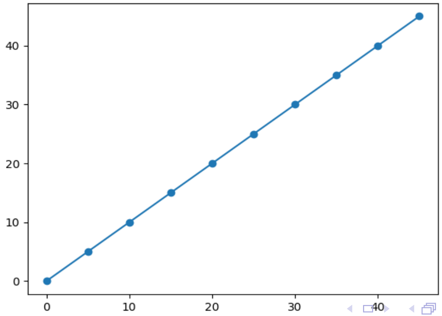
```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 50, 0.1)
y = x
plt.plot(x, y)
plt.savefig('plot.png')
```



Matplotlib: adding markers

Code

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 50, 5)
y = x
plt.plot(x, y, marker = 'o')
plt.savefig('plot-marker.png')
```



Matplotlib: marker table[1]

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

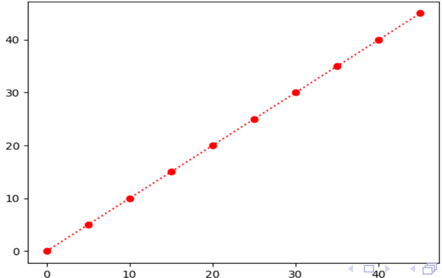
[1] https://www.w3schools.com/python/matplotlib_markers.asp

Matplotlib: adding markers (fmt)

marker line color

Code

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 50, 5)
y = x
plt.plot(x, y, 'o:r')
plt.savefig('plot-marker-fmt.png')
```



Matplotlib: line and color table[2]

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

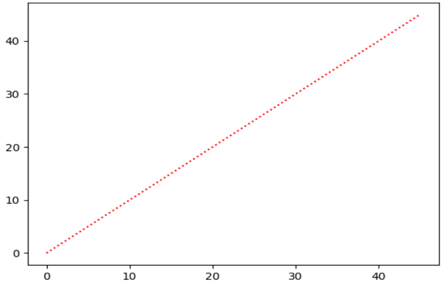
Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

[2] https://www.w3schools.com/python/matplotlib_markers.asp

Matplotlib: adding linestyle

Code

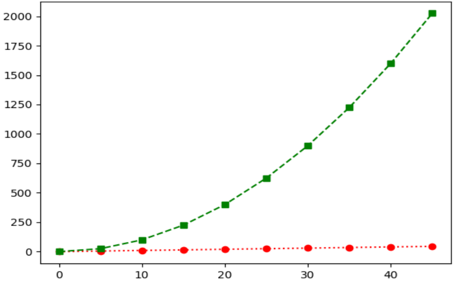
```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 50, 5)
y = x
plt.plot(x, y, linestyle = 'dotted', color = 'r')
plt.savefig('plot-linestyle.png')
```



Matplotlib: multiple lines

Code

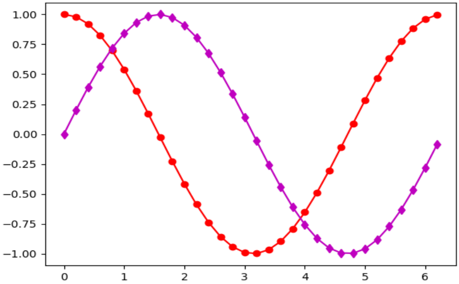
```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 50, 5)
y = x
w = x**2
plt.plot(x, y, linestyle = 'dotted', color = 'r', marker = 'o')
plt.plot(x, w, linestyle = 'dashed', color = 'g', marker = 's')
plt.savefig('plot-multiple-lines.png')
```



Matplotlib: multiple lines (contd.)

Code

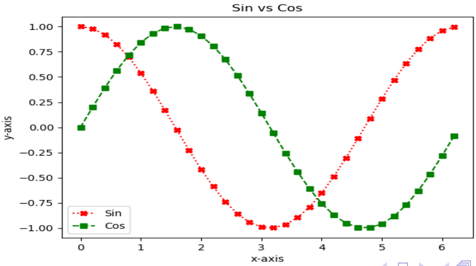
```
import matplotlib.pyplot as plt
import numpy as np
x_new = np.arange(0, 2*np.pi, 0.2)
w = np.cos(x_new)
y_new = np.sin(x_new)
plt.plot(x_new, w, 'o-r', x_new, y_new, 'd-m')
plt.savefig('plot-multiple-lines-sincos.png')
```



Matplotlib: title, labels, legend

Code

```
import matplotlib.pyplot as plt
import numpy as np
x_new = np.arange(0, 2*np.pi, 0.2)
w = np.cos(x_new)
y_new = np.sin(x_new)
plt.plot(x_new, w, 'X:r', label = 'Sin')
plt.plot(x_new, y_new, 's--g', label = 'Cos')
plt.title('Sin vs Cos')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
plt.savefig('plot-title-labels-legend.png')
```

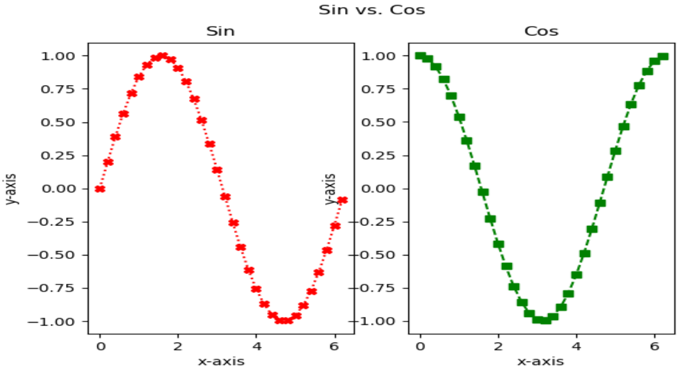


Matplotlib: subplots

Code

```
import matplotlib.pyplot as plt
import numpy as np

x_new = np.arange(0, 2*np.pi, 0.2)
w = np.cos(x_new)
y_new = np.sin(x_new)

plt.subplot(1, 2, 1) #the figure has 1 row, 2 columns, and this plot is the first plot
plt.title('Sin')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.plot(x_new, y_new, 'X:r', label = 'Sin')

plt.subplot(1, 2, 2) #the figure has 1 row, 2 columns, and this plot is the second plot
plt.title('Cos')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.plot(x_new, w, 's--g', label = 'Cos')

plt.suptitle('Sin vs. Cos')

plt.savefig('plot-subplot.png')
```

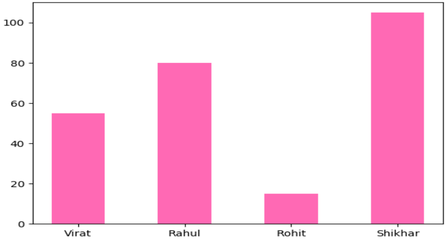
Matplotlib: subplots (contd.)



Matplotlib: bar

Code

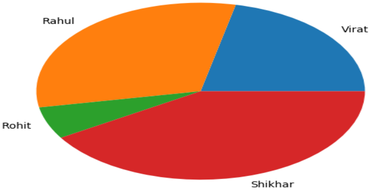
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["Virat", "Rahul", "Rohit", "Shikhar"])
y = np.array([55, 80, 15, 105])
plt.bar(x, y, color = "hotpink", width = 0.5)
plt.savefig('plot-bar.png')
```



Matplotlib: pie

Code

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["Virat", "Rahul", "Rohit", "Shikhar"])
y = np.array([55, 80, 15, 105])
plt.pie(y, labels = x)
pplt.savefig('plot-pie.png')
```



THANK YOU !!!