ÖZYEĞİN UNIVERSITY

EE393 Python for Engineers

Dr. Orhan Gökçöl

orhan.gokcol@ozyegin.edu.tr

online

23.11.2020

2020-2021 Fall Semester

1

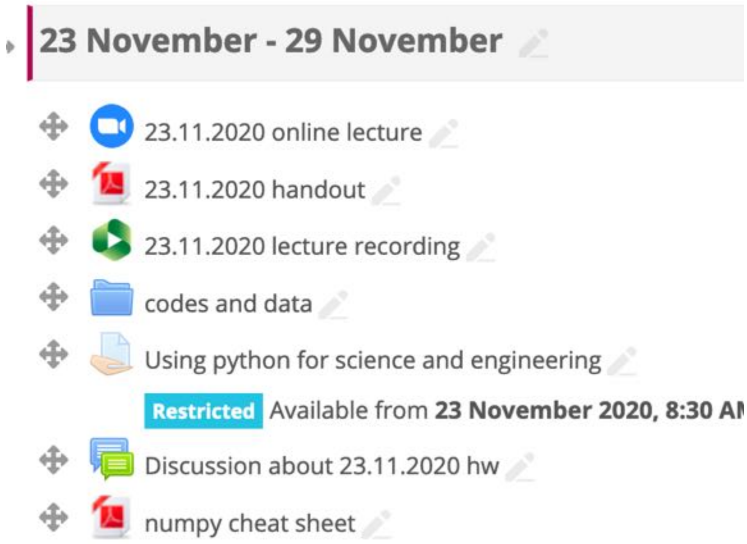
Agenda

- Short review
- NumPy
- SciPy
- An introduction to matplotlib
- Many examples selected from engineering fundamentals
- Homework

(Download to your local)

2

23.11.2020 | LMS resources



Learning objectives for 23.11.2020

- Understands how to use Python in science and engineering
- Applies numpy methods to Python programs
- Applies scipy and numpy methods to engineering problem solving
- Plots (x,y) discrete points using several strategies in Python-matplotlib

MIDTERM EXAM

DECEMBER 21, 2020; 08:40

Online
(Details will be announced later)

FINAL EXAM

JANUARY 15, 2021; 09:00

Online
(Details will be announced later)

Last week

- openpyxl –working with excel from Python
- Objects in Python
- OOP basics

Scientific/Engineering Python Uses

- Extra features required:
  - fast, multidimensional arrays
    - With homogenous elements inside!!!! e.q. all numbers!
  - libraries of reliable, tested **scientific functions**
  - plotting tools
- **NumPy** is at the core of nearly every scientific Python application or module since it provides a fast **N-d array** datatype that can be manipulated in a vectorized form.

Speedtest
Python vs Numpy

100M numbers are multiplied in a) Python lists, b) Numpy Array; and execution times are measured for comparison

Numpy is approx. 20 (+) times faster

```python
#SPEED comparison
# importing required packages
import time

# size of arrays and lists
size = 100000000

# declaring lists
list1 = range(size)
list2 = range(size)
# declaring arrays
array1 = np.arange(size)
array2 = np.arange(size)

# list
initialTime = time.time()
resultantList = [(a * b) for a, b in zip(list1, list2)]

# calculating execution time
print("Time taken by Lists :",
      (time.time() - initialTime),
      "seconds")

# NumPy array
initialTime = time.time()
resultantArray = array1 * array2
# calculating execution time
print("Time taken by NumPy Arrays :",
      (time.time() - initialTime),
      "seconds")
```

```
Time taken by Lists : 8.10220313072204 6 seconds
Time taken by NumPy Arrays : 0.38413882255542 seconds
```

```
Time taken by Lists : 8.33761191368103 seconds
Time taken by NumPy Arrays : 0.4112420082092285 seconds
```

Python 2.7

```
(base) Orhans-MacBook-Pro:week8 orhang$ python2.7 speedtest.py
('Time taken by Lists :', 65.85366821289062, 'seconds')
('Time taken by NumPy Arrays :', 3.162869930267334, 'seconds')
```

Arrays – Numerical Python (Numpy)

- Lists ok for storing small amounts of one-dimensional data

```
>>> a = [1,3,5,7,9]
>>> print(a[2:4])
[5, 7]
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

```
>>> a = [1,3,5,7,9]
>>> b = [3,5,6,7,9]
>>> c = a + b
>>> print (c)
[1, 3, 5, 7, 9, 3, 5, 6, 7, 9]
```

- But, can't use directly with arithmetical operators (+, -, *, /, …)
- Need efficient arrays with arithmetic and better multidimensional tools
- **Numpy**

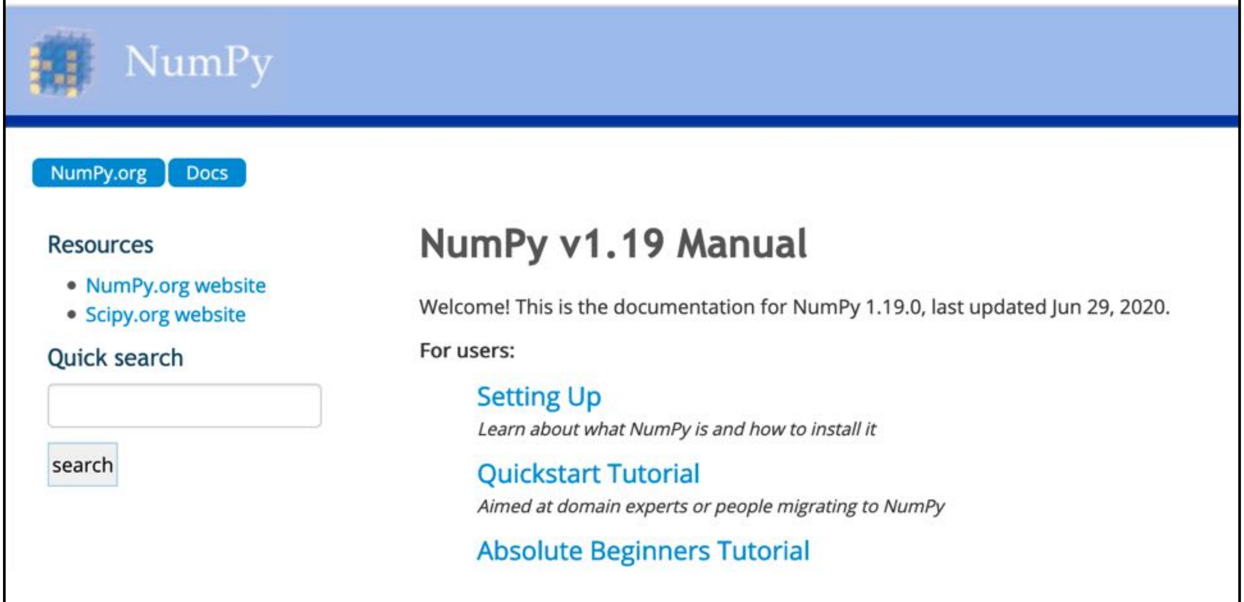
```
>>> import numpy
```

- Similar to lists, but much more capable, except it is fixed in size

What is NumPy?

- NumPy is the fundamental package needed for scientific computing with Python. It contains:
- a powerful N-dimensional array object
- **basic linear algebra functions**
- **basic Fourier transforms**
- **sophisticated random number capabilities**
- tools for integrating Fortran code
- tools for integrating C/C++ code

numpy.org/docs/stable



Numpy methods

- We generally import numpy as follows (as many Python coders do):

import numpy as np

- Main object type for numpy is **nd** array (or, numpy array)
  - Some of the methods are public methods and called as **np.methodname.** For ex. np.array, np.arrange, np.linspace, np.sort etc.
  - Some others are called via an instance such as
    - X = np.array(….)
      X.sum, X.mean etc.

Numpy array

- The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index as we did for Python lists.
- Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called **dtype**).
- Basic numpy array is created using **numpy.array**

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. The elements are all of the same type, referred to as the array **dtype.**

Numpy – Creating vectors (or 1D arrays)

- From lists
  - numpy.array

```
#creating 1D arrays (vectors)
# as vectors from lists
a = np.array([1,3,5,7,9]) #a is a 1-d array
b = np.array([3,5,6,7,9])
c = a + b
print (c)
print (type(c))
print (c.shape)
print (c.dtype)
#Accessing array elements
print ("First: ", a[0])
print ("Last minus 1: ", a[-2])
for x in a:   #ndarrays are iterable
    print (x, end=" ")
```

```
[ 4  8 11 14 18]
<class 'numpy.ndarray'>
(5,)
int64
First:  1
Last minus 1:  7
1 3 5 7 9
```

numpy.ipynb

Data Types

- **bool_** :Boolean (True or False) stored as a byte
- **int_** :Default integer type (same as C long; normally either int64 or int32)
- **intc** : Identical to C int (normally int32 or int64)
- **intp** : Integer used for indexing (same as C ssize_t; normally either int32 or int64)
- **int8** : Byte (-128 to 127)
- **int16** : Integer (-32768 to 32767)
- **int32** : Integer (-2147483648 to 2147483647)
- **int64** : Integer (-9223372036854775808 to 9223372036854775807)
- **uint8** : Unsigned integer (0 to 255)
- **uint16** : Unsigned integer (0 to 65535)

int8, int16, int32, int64 can be replaced by equivalent string i1, i2, i4 etc.

Data Types

- **uint32** : Unsigned integer (0 to 4294967295)
- **uint64** : Unsigned integer (0 to 18446744073709551615)
- **float_** : Shorthand for float64
- **float16** : Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
- **float32** : Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
- **float64** : Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
- **double** : Shorthand for float64
- **longdouble** : Shorthand for float96 or float128 (-1.7E4932 .. -1.7E4932)
- **complex_** : Shorthand for complex128
- **complex64** : Complex number, represented by two 32-bit floats (real and imaginary components)
- **complex128** : Complex number, represented by two 64-bit floats (real and imaginary components)

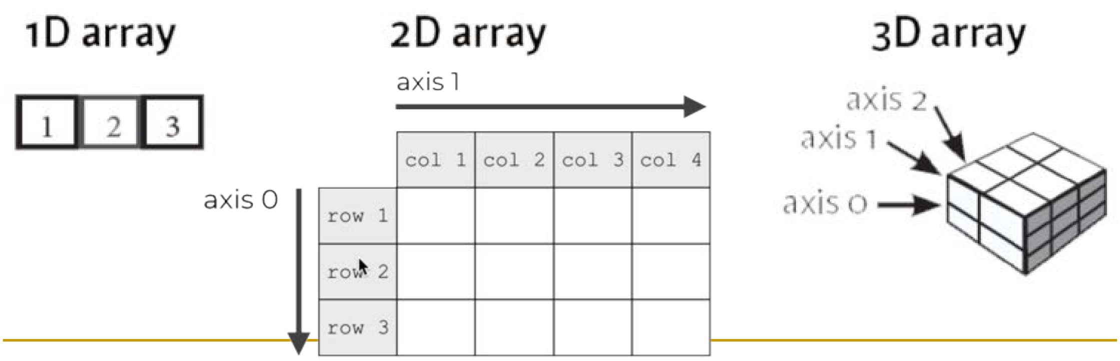
Creating n-dim arrays & shape

```
#CREATING A MATRIX
#more than one dimension!
l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]]  # create a list
a = np.array(l)  # convert a list to an array
print(a)
print (a.shape)
print(a.dtype)  # get type of an array
M = np.array([[1, 2], [3, 4]])
print (M.shape)
print (M.dtype)
```

```
[[1 2 3]
 [3 6 9]
 [2 4 6]]
(3, 3)
int64
(2, 2)
int64
```

Numpy – Creating arrays

- There are a number of ways to initialize new numpy arrays, for example from
  - a Python list or tuples
  - using functions that are dedicated to generating numpy arrays, such as **arange**, **linspace**, etc.
  - reading data from files (csv, excel, txt etc.)



```python
#checking number of dimensions, shape and size
import numpy as np
a = np.array(1+2j)
b = np.array([0,2,4,6,8])
c = np.array([[1,2,3], [5,8,13]])
c2 = np.array([[1,2,3], [5,8,13],[1,1,1]])
d = np.array([[[1, 2, 3,0], [5,8,13,0]],
              [[21,34,55,0], [89,144,233,0]],
              [[21,34,55,0], [89,144,233,0]]])
print("dimension of a: {}, shape: {}".format(a.ndim, a.shape))
print("dimension of b: {}, shape: {}".format(b.ndim, b.shape))
print("dimension of c: {}, shape: {}".format(c.ndim, c.shape))
print("dimension of c2: {}, shape: {}".format(c2.ndim, c2.shape))
print("dimension of d: {}, shape: {}, size: {}".format(d.ndim, d.shape, d.size))
print (d)
```

```
dimension of a: 0, shape: ()
dimension of b: 1, shape: (5,)
dimension of c: 2, shape: (2, 3)
dimension of c2: 2, shape: (3, 3)
dimension of d: 3, shape: (3, 2, 4), size: 24
[[[  1   2   3   0]
  [  5   8  13   0]]

 [[ 21  34  55   0]
  [ 89 144 233   0]]

 [[ 21  34  55   0]
  [ 89 144 233   0]]]
```

ndim, shape, size

Special arrays : empty, zeros, ones

```python
import numpy as np
x = np.empty([3,2], dtype = int)
print (x)
#values are randomly assigned values in the memory
```

```
[[4607182418800017408                   0]
 [4611686018427387904                   0]
 [4613937818241073152                   0]]
```

```python
# array of five zeros. Default dtype is float
import numpy as np
x = np.zeros(5)
print (x)
```

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

```python
# array of five ones. Default d
import numpy as np
x = np.ones(5)
print (x)
y = np.ones([5,3], dtype=int)
print (y)
```

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

```python
#identity matrix
I = np.eye(5)
print (I)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

asarray

```python
#asarray
#This function is similar to numpy.
#It is useful for converting Python sequence into ndarray
import numpy as np
x = [1,2,3]
a = np.asarray(x, dtype = float)
print (a)
```

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

LOOK!

numpy.ipynb

Numpy – Creating matrices

```python
import numpy as np
# create a list
l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]]
a = np.array(l) # convert a list to an array
print(a)
print (a.shape)
print(a.dtype) # get type of an array
print ("row:", a[0])
print ("column:", a[:,1])

M = np.array([[1, 2], [3, 4]])
print (M.shape)
print (M.dtype)

M = np.array([[1, 2], [3, 4]], dtype=complex)
print (M)
```

```
[[1 2 3]
 [3 6 9]
 [2 4 6]]
(3, 3)
int64
row: [1 2 3]
column: [2 6 4]
(2, 2)
int64
[[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

ONLY ONE TYPE!!!!!!!!! FOR ALL ARRAY ELEMENTS!!!!!

Try this to get an error !!!
M[0,0]=12
M[1,0]='hello'

Numpy – Matrices

```python
#USE OF MATRICES
print("Matrix a:\n", a)
print("First row:", a[0]) # this is just like a list of lists
print("element at (1,2) :", a[1, 2])   # arrays can be given comma separated indices
print(a[1, 1:3]) # and slices

print(a[:,1])
a[1, 2] = -3
print("a changed as\n",a)

a[:, 0] = [2, -1, 5]
print ("see how first row is changed")
print(a)
```

```
Matrix a:
 [[ 0  2  3]
 [ 9  6 -3]
 [ 8  4  6]]
First row: [0 2 3]
element at (1,2) : -3
[ 6 -3]
[2 6 4]
a changed as
 [[ 0  2  3]
 [ 9  6 -3]
 [ 8  4  6]]
see how first row is changed
[[ 2  2  3]
 [-1  6 -3]
 [ 5  4  6]]
```

Numpy – array creation and use

```
#array creation and use
x, y = numpy.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
print (x)
# random data
z=numpy.random.rand(5,5)
print (z)
```

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
[[0.17706244 0.17677506 0.76115697 0.43130308 0.95218228]
 [0.61385839 0.91252151 0.30113525 0.47876756 0.48179825]
 [0.96204896 0.64342828 0.6630705  0.19714807 0.32601357]
 [0.65244553 0.14365803 0.97918597 0.6870939  0.95028749]
 [0.51279853 0.26362755 0.90909429 0.32577498 0.52959444]]
```


numpy.arrange and numpy.linspace

- **arrange** returns an **ndarray** object containing evenly spaced values within a given range.
- In **linspace**, instead of step size, the number of evenly spaced values between the interval is specified.

```
#Creating numpy arrays -using creation functions
x = np.arange(0, 10, 1) # arguments: start, stop, step
print (x)
y=np.linspace(0, 10, 25) #arguments: start,stop,no.of points
print (y)
y = y.astype('complex')
print (y)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0.          0.41666667  0.83333333  1.25        1.66666667  2.08333333
  2.5         2.91666667  3.33333333  3.75        4.16666667  4.58333333
  5.          5.41666667  5.83333333  6.25        6.66666667  7.08333333
  7.5         7.91666667  8.33333333  8.75        9.16666667  9.58333333
 10.        ]
[ 0.        +0.j  0.41666667+0.j  0.83333333+0.j  1.25      +0.j
  1.66666667+0.j  2.08333333+0.j  2.5       +0.j  2.91666667+0.j
  3.33333333+0.j  3.75      +0.j  4.16666667+0.j  4.58333333+0.j
  5.        +0.j  5.41666667+0.j  5.83333333+0.j  6.25      +0.j
  6.66666667+0.j  7.08333333+0.j  7.5       +0.j  7.91666667+0.j
  8.33333333+0.j  8.75      +0.j  9.16666667+0.j  9.58333333+0.j
 10.        +0.j]
```

dtype could be used in both functions

np.arrange

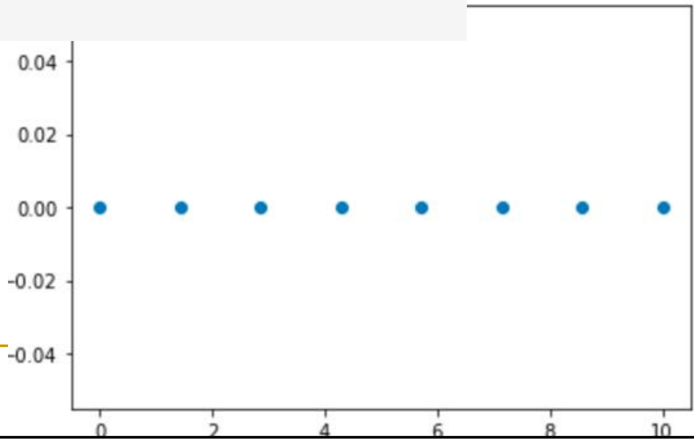
```
#np.arrange different uses
x = np.arange(10)
print ("x =",x)
y = np.arange(5,10)
z = np.arange(5,10,0.5)
print (z)

#reshape to a different dimension
zz = np.arange(1,10).reshape(3,3)
print (zz)
```

```
x = [0 1 2 3 4 5 6 7 8 9]
[5.  5.5 6.  6.5 7.  7.5 8.  8.5 9.  9.5]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

25

```
#linspace
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
N = 8
y = np.zeros(N)
x1 = np.linspace(0, 10, N, endpoint=True)
p1 = plt.plot(x1, y, 'o')
ax.set_xlim([-0.5,10.5])
plt.show()
```

np.linspace



26

Miscellaneous array creations : mgrid and random

```
#array creation and use
#mgrid
#random values in ndarrays
x,y = np.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
print (x)
print (y)
# random data
z=np.random.rand(5,5)
print (z)
```

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
[[0.12385852 0.30939883 0.94205462 0.19855433 0.20146846]
 [0.66198873 0.9794651  0.33698499 0.11377853 0.45960751]
 [0.00726439 0.2761222  0.18953797 0.66304801 0.94782969]
 [0.15115426 0.90193745 0.13085806 0.8968881  0.2364141 ]
 [0.49409317 0.66690493 0.52205047 0.88732595 0.60540875]]
```


Diagonal matrices

```
#Creating arrays
# a diagonal matrix
a=np.diag([1,2,3])
print (a)
b = np.zeros(5)
print(b)
print (b.dtype)
n = 1000
my_int_array = np.zeros(n, dtype=np.int)
print (my_int_array.dtype)
c = np.ones((3,3))
print (c)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
[0. 0. 0. 0. 0.]
float64
int64
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
#useful ndarray functions : sum, max, min, mean
x = np.array([[1,2,3],[4,5,6],[7,8,9]])
print ("x array is\n",x)
print ("ndim:",x.ndim)
print ("Axis-0 sum is {}".format(x.sum(axis=0)))
print ("Axis-1 sum is {}".format(x.sum(axis=1)))
print ("Axis-0 max is {}".format(x.max(axis=0)))
print ("Axis-1 min is {}".format(x.min(axis=1)))
print ("Axis-1 mean is {}".format(x.mean(axis=1)))
print ("Global sum is {}".format(x.sum()))
```

```
x array is
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
ndim: 2
Axis-0 sum is [12 15 18]
Axis-1 sum is [ 6 15 24]
Axis-0 max is [7 8 9]
Axis-1 min is [1 4 7]
Axis-1 mean is [2. 5. 8.]
Global sum is 45
```

Useful ndarray functions working on axis

Numpy – other array methods

```
#other functions
#note that you can specify axis value in each of the functions
arr = np.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
print (arr.sum())
print (arr.mean())
print (arr.std())
print (arr.max())
print (arr.min())
print (div_by_3.all())
print (div_by_3.any())
print (div_by_3.sum())
print (div_by_3.nonzero())
```

```
22.0
3.6666666666666665
2.028683207293725
6.7
1.2
False
True
3
(array([2, 5, 8]),)
```

Numpy – Creating arrays from a file

testdata.txt

```
"Stn", "Data", "Tg", "qTg", "Tn", "qTn", "Tx", "qTx"
001, 19010101,   -49, 00,   -68, 00,   -22, 40
001, 19010102,   -21, 00,   -36, 30,   -13, 30
001, 19010103,   -28, 00,   -79, 30,    -5, 20
001, 19010104,   -64, 00,   -91, 20,   -10, 00
001, 19010105,   -59, 00,   -84, 30,   -18, 00
001, 19010106,   -99, 00,  -115, 30,   -78, 30
001, 19010107,   -91, 00,  -122, 00,   -66, 00
001, 19010108,   -49, 00,   -94, 00,    -6, 00
001, 19010109,    11, 00,   -27, 40,    42, 00
```

```
#file I/O
import os
print(os.system('head testdata.txt'))
data = np.genfromtxt('testdata.txt', delimiter=',', skip_header=1)
print (data.shape)
print (data)
np.savetxt('datasaved.txt', data)
```

```
0
(9, 8)
[[ 1.0000000e+00  1.9010101e+07 -4.9000000e+01  0.0000000e+00
  -6.8000000e+01  0.0000000e+00 -2.2000000e+01  4.0000000e+01]
 [ 1.0000000e+00  1.9010102e+07 -2.1000000e+01  0.0000000e+00
  -3.6000000e+01  3.0000000e+01 -1.3000000e+01  3.0000000e+01]
 [ 1.0000000e+00  1.9010103e+07 -2.8000000e+01  0.0000000e+00
  -7.9000000e+01  3.0000000e+01 -5.0000000e+00  2.0000000e+01]
```


Numpy – file i/o

```
#file I/O
M = np.random.rand(3,3)
print (M)
np.save('saved-matrix.npy', M)     #binary format!
N=np.load('saved-matrix.npy')
print (N)
print (N.ndim, N.shape)
```

```
[[0.51996506 0.80997384 0.73765913]
 [0.26754792 0.30598719 0.44882472]
 [0.26466518 0.87206017 0.99212161]]
[[0.51996506 0.80997384 0.73765913]
 [0.26754792 0.30598719 0.44882472]
 [0.26466518 0.87206017 0.99212161]]
2 (3, 3)
```

Numpy – object copying

Two ndarrays are mutable and may be views to the same memory:

```
#numpy arrays are mutable
#be careful when you create
#another copy of an nd array
x = np.array([1,2,3,4])
y = x
print (x is y)
print (id(x), id(y))
x[0] = 9
print (y)
x[0] = 1
z = x[:]
print (x is z)
print (id(x), id(z))
x[0] = 8
print (z)
```

```
True
140216049723232 140216049723232
[9 2 3 4]
False
140216049723232 140216049762352
[8 2 3 4]
```

```
x = np.array([1,2,3,4])
y = x.copy()
print (x is y)
print (id(x), id(y))
x[0] = 9
print (x)
print (y)
```

```
False
140216049724512 140216049765632
[9 2 3 4]
[1 2 3 4]
```

(Faster)

numpy basic operations

- Operations on ndarrays are elemet-wise

```
#numpy ndarray basic operations
a = np.array([3,4,5])
b = np.ones(3)
print ("a-b:", a - b)

a = np.array([[1,2],[3,4]])
b = np.array([[1,2],[3,4]])
print ("a:\n",a)
print ("b:\n",b)
print ("a*b:\n", a * b)
print ("dot product of a and b:\n",np.dot(a,b))
```

```
a-b: [2. 3. 4.]
a:
 [[1 2]
 [3 4]]
b:
 [[1 2]
 [3 4]]
a*b:
 [[ 1  4]
 [ 9 16]]
dot product of a and b:
 [[ 7 10]
 [15 22]]
```

Special operators: +=, -=, *=, /=, **= , //=, %=

```
#special addition and
#multiplication operators
a = np.zeros((2,2),dtype='float')
a += 5
print (a)
a *= 5
print (a)
a = a + 2*a
print (a)
```

```
[[5. 5.]
 [5. 5.]]
[[25. 25.]
 [25. 25.]]
[[75. 75.]
 [75. 75.]]
```


Concatenation of arrays

```
#Concatenation of arrays
a = np.array([1,2,3])
b = np.array([4,5,6])
c = np.array([7,8,9])
a_h = np.hstack([a,b,c]) #horizontal stack
a_v = np.vstack([a,b,c])
print (a_h)
print (a_v)
```

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

Array sort

```
#array sort
arr = np.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
arr.sort()  # acts on array itself
print(arr)
x = np.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
print ("x is\n",x)
xx=np.sort(x)
print (xx)
print(x)
s = x.argsort() #sorts indice no.s wrt corresponding values
print (s)
print (x[s])
```

```
[1.2 1.8 2.3 4.5 5.5 6.7]
x is
 [4.5 2.3 6.7 1.2 1.8 5.5]
[1.2 1.8 2.3 4.5 5.5 6.7]
[4.5 2.3 6.7 1.2 1.8 5.5]
[3 4 1 0 5 2]
[1.2 1.8 2.3 4.5 5.5 6.7]
```


Miscellaneous uses

```
a = numpy.arange(4.0)
b = a * 23.4
c = b/(a+1)
c += 10
print (c)
arr = numpy.arange(100, 200)
select = [5, 25, 50, 75, -5]
print(arr[select])  # can use integer lists as indices
arr = numpy.arange(10, 20 )
div_by_3 = arr%3 == 0  # comparison produces boolean array
print(div_by_3)
print(arr[div_by_3])  # can use boolean lists as indices
arr = numpy.arange(10, 20) . reshape((2,5))
print (arr)
```

```
[10.   21.7  25.6  27.55]
[105 125 150 175 195]
[False False  True False False  True False False  True False]
[12 15 18]
[[10 11 12 13 14]
 [15 16 17 18 19]]
```

Common numpy functions useful for engineers

```
#common mathematical functions
x = np.arange(1,5)
result = np.sqrt(x) * np.pi
print (result)
print (np.power(2,4)) #much faster than python equivalent
print (x.max() - x.min())

#exponential & log
arr = np.array([10,8,4])
print(np.exp(arr)) #e^x
print (np.log(arr)) #ln(x), base is e
print ("e :", np.e, "pi:", np.pi)
print (np.log10(arr)) #log(x), base is 10
print (np.log2(arr)) #log(x), base is 2

#rounding
print ("\nROUNDING")
arr = np.array([20.8999,67.89899,54.23409])
print(np.around(arr,2)) #round off with 2 decimals
print(np.floor(arr)) #largest integer less than input number
print(np.ceil(arr))  #smallest integer greater than input num

#trigonometric
print ("\nTRIGONOMETRIC FUNCTIONS")
arr = np.array([0, 30, 60, 90, 120, 150, 180])
print(np.sin(arr * np.pi / 180)) #sine function
print(np.cos(arr * np.pi / 180)) #cosine
```

Algebraic
Rounding
Logarithm
Trigonometry
Complex numbers

Numpy – statistics

More functions are available in **scipy**

```
#statistics
a = np.array([1, 4, 3, 8, 9, 2, 3], float)
print ("median:",np.median(a))
b = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
c = np.corrcoef(b)
print ("Correlation:", c)
d = np.corrcoef(a,a)
print ("Correlation:", d)
a = np.array([1,2,3,4,6,7,8,9])
b = np.array([2,4,6,8,10,12,13,15])
c = np.array([-1,-2,-2,-3,-4,-6,-7,-8])
print (np.corrcoef([a,b,c]))
print ("Covariance: ", np.cov(a)) #covariance
print ("Variance: ", np.var(a)) #covariance
print ("Standard deviation: ", np.std(a)) #covariance
```

```
median: 3.0
Correlation: [[1.         0.72870505]
 [0.72870505 1.        ]]
Correlation: [[1. 1.]
 [1. 1.]]
[[ 1.          0.99535001 -0.9805214 ]
 [ 0.99535001  1.         -0.97172394]
 [-0.9805214  -0.97172394  1.        ]]
Covariance:  8.571428571428571
Variance:  7.5
Standard deviation:  2.7386127875258306
```

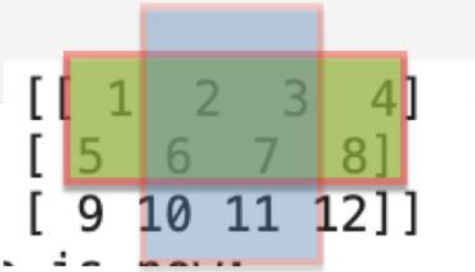
Linear Algebra

Matrix operations

```python
#numpy linear algebra package : linalg
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])
# Rank of a matrix
print("Rank of A:", np.linalg.matrix_rank(A))

# Trace of matrix A
print("\nTrace of A:", np.trace(A))

# Determinant of a matrix
print("\nDeterminant of A:", np.linalg.det(A))

# Inverse of matrix A
print("\nInverse of A:\n", np.linalg.inv(A))

print("\nMatrix A raised to power 3:\n",
           np.linalg.matrix_power(A, 3))

#solving linear eqn systems
b = [1,2,3]
x = np.linalg.solve(A,b)
print ("Solution vector:", x)

#Eigenvalues and eigen vectors
#Creating an array using diag function
a = np.diag((1, 2, 3))
print("Array is :",a)

# calculating an eigen value using eigvals() function
c = np.linalg.eigvals(a)
print("Eigen value is :",c)

# calculating an eigen value using eig() function
c, d = np.linalg.eig(a)
print("Eigen value is :",c)
print("Eigen vector is :",d)
```


Array slicing –sub arrays

```python
# ARRAY INDEXING
# Create the following  2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print ("a is :\n", a)

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print ("b is now:\n",b)
```




```
a is :
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
b is now:
 [[2 3]
 [6 7]]
```

```
#misc ndarray operations
print(a[0])  # this is just like a list of lists
print(a[1][2])  #print a single value ==> 7 will be written
print(a[1, 2])  # arrays can be given comma separated indices
print(a[1, 1:3])  # and slices
print(a[:,1])
a[1, 2] = 7
print(a)
a[:, 0] = [0, 9, 8]
print(a)
```

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


Boolean array indexing

```
#BOOLEAN ARRAY INDEXING
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)  # Find the elements which are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print (bool_idx)
print (a[a > 2])
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
```

Using arrays wisely

- Array operations are implemented in C or Fortran
- Optimised algorithms - i.e. fast!
- Python loops (i.e. for i in a:…) are much slower
- **Prefer array operations over loops, especially when speed important**
- Also produces shorter code, often more readable

Numpy – arrays, matrices

For **two dimensional** arrays NumPy defined a special matrix class in module matrix. Objects are created either with matrix() or mat() or converted from an array with method **asmatrix().**

Note that the statement **m = mat(a)** creates a copy of array 'a'.

Changing values in 'a' will not affect 'm'.

On the other hand, method **m = asmatrix(a)** returns a new reference to the same data. Changing values in 'a' will affect matrix 'm'.

```
#matrices -different definitions
m = numpy.mat([[1,2],[3,4]])
print (m)
a = numpy.array([[1,2],[3,4]])
print (a)
m = numpy.mat(a)
print (m)
#or
a = numpy.array([[1,2],[3,4]])
print (a)
m = numpy.asmatrix(a)
print (m)
```

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

Numpy – matrices

Array and matrix operations may be quite different!

```
#array vs matrix
a = numpy.array([[1,2],[3,4]])
m = numpy.mat(a) # convert 2-d array to matrix
m = numpy.matrix([[1, 2], [3, 4]])
print (a[0]) # result is 1-dimensional
print (m[0])    # result is 2-dimensional
print (a*a)    # element-by-element multiplication
print (m*m)    # (algebraic) matrix multiplication
print (a**3)    # element-wise power
print (m**3)    # matrix multiplication m*m*m
print (m.T) # transpose of the matrix
print (m.H)  # conjugate transpose (differs from .T for complex matrices)
print (m.I) # inverse matrix
```

```
[1 2]
[[1 2]]
[[ 1  4]
 [ 9 16]]
[[ 7 10]
 [15 22]]
[[ 1  8]
 [27 64]]
[[ 37  54]
```


Numpy – matrices

- Operator *, dot(), and multiply():
  - For array, '*' **means element-wise multiplication**, and the dot() function is used for matrix multiplication.
  - For matrix, '*'**means matrix multiplication**, and the multiply() function is used for element-wise multiplication.
- Handling of vectors (rank-1 arrays)
  - For array, the vector shapes 1xN, Nx1, and N are all different things. Operations like A[:,1] return a rank-1 array of shape N, not a rank-2 of shape Nx1. Transpose on a rank-1 array does nothing.
  - For matrix, rank-1 arrays are always upgraded to 1xN or Nx1 matrices (row or column vectors). A[:,1] returns a rank-2 matrix of shape Nx1.
- Handling of higher-rank arrays (rank > 2)
  - array objects can have rank > 2.
  - matrix objects always have exactly rank 2.
- Convenience attributes
  - array has a .T attribute, which returns the transpose of the data.
  - matrix also has .H, .I, and .A attributes, which return the conjugate transpose, inverse, and asarray() of the matrix, respectively.
- Convenience constructor
  - The array constructor takes (nested) Python sequences as initializers. As in array([[1,2,3],[4,5,6]]).
  - The matrix constructor additionally takes a convenient string initializer. As in matrix("[1 2 3; 4 5 6]")

Numpy – matrix mathematics

```
#array math
A = numpy.array([[n+m*10 for n in range(5)] for m in range(5)])
v1 = numpy.arange(0, 5)
print (A)
print (v1)
print (numpy.dot(A,A))
print (numpy.dot(A,v1))
print (numpy.dot(v1,v1))
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[0 1 2 3 4]
[[ 300  310  320  330  340]
 [1300 1360 1420 1480 1540]
 [2300 2410 2520 2630 2740]
 [3300 3460 3620 3780 3940]
 [4300 4510 4720 4930 5140]]
[ 30 130 230 330 430]
30
```

```
#Alternatively, we can cast the array objects to the type matrix.
#This changes the behavior of the standard arithmetic
#operators +, -, * to use matrix algebra.
M = numpy.matrix(A)
v = numpy.matrix(v1).T
print (v)
print (M*v)
print (v.T * v)   # inner product
# standard matrix algebra applies
print (v + M*v)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]]
[[ 30]
 [130]
 [230]
 [330]
 [430]]
[[30]]
[[ 30]
 [131]
 [232]
 [333]
 [434]]
```


scipy

- SciPy is a scientific python open source to perform Mathematical, Scientific and Engineering Computations.
- The SciPy library depends on **NumPy**, which provides convenient and fast N-dimensional array manipulation.
- The SciPy library is built to work with NumPy arrays and provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization.
- Together, they run on all popular operating systems, are quick to install and are free of charge.
- SciPy is easy to use, but powerful enough to depend on by some of the world's leading scientists and engineers.

scipy core sub-packages

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

51

```
import scipy.constants as sc
print ("Pi: {} , elementary charge: {}, Planck: {}".format(sc.pi, sc.e, sc.h))
print ("Speed of light: {} , gravitational constant: {}, Avogadro: {}".format(sc.c, sc.G, sc.Avogadro))
print ("Standard aptodphere in pascal {} , Boltzman constant: {}, km/h to m/s: {}".format(sc.atm, sc.k, sc.kmh))
print ("And many more!!!!!")
```

```
Pi: 3.141592653589793 , elementary charge: 1.602176634e-19, Planck: 6.62607015e-34
Speed of light: 299792458.0 , gravitational constant: 6.6743e-11, Avogadro: 6.02214076e+23
Standard aptodphere in pascal 101325.0 , Boltzman constant: 1.380649e-23, km/h to m/s: 0.2777777777777778
And many more!!!!!
```

$$\int_0^1 e^{-x^2}\, dx = \frac{1}{2}\sqrt{\pi}\ \mathrm{erf}(1) \approx 0.746824$$

```
#Numerical Integration
import scipy.integrate
from numpy import exp

def f(x):
    return exp(-x**2)
i = scipy.integrate.quad(f, 0, 1)
print (i)
```

```
(0.7468241328124271, 8.291413475940725e-15)
```

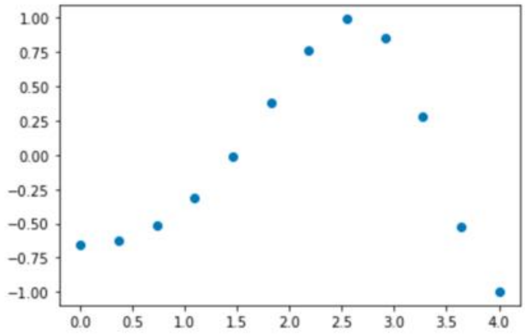
52

scipy interpolation

```python
#Interpolation
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt

x = np.linspace(0, 4, 12)
y = np.cos(x**2/3+4)
print (x,y)
```

```
[0.         0.36363636 0.72727273 1.09090909 1.454545
 2.18181818 2.54545455 2.90909091 3.27272727 3.636363
5364362 -0.61966189 -0.51077021 -0.31047698 -0.007154
  0.76715099  0.99239518  0.85886263  0.27994201 -0.5
```

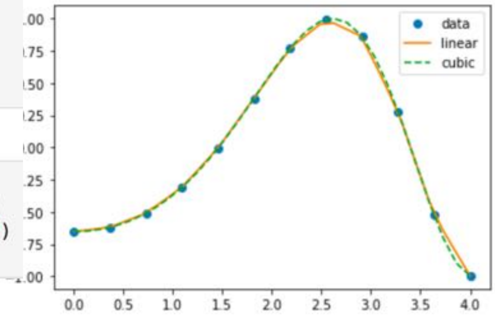
```python
#plot points
plt.plot(x, y,'o')
plt.show()
```



```python
#linear and cubic interpolations
f1 = interpolate.interp1d(x, y, kind = 'linear')
f2 = interpolate.interp1d(x, y, kind = 'cubic')
print (f2(3.7))
```

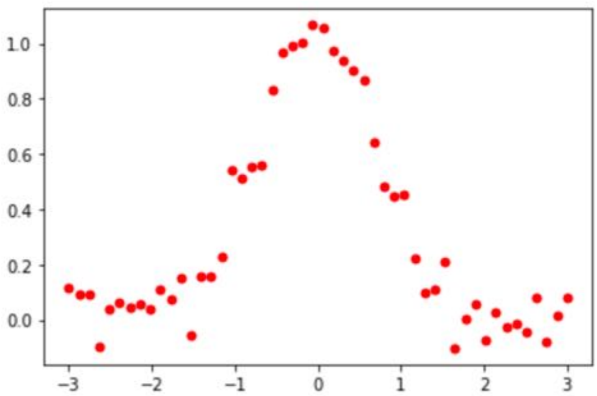
```
-0.6503130110905226
```

```python
xnew = np.linspace(0, 4,30)
plt.plot(x, y, 'o', xnew, f1(xnew), '-', xnew, f2(xnew), '--')
plt.legend(['data', 'linear', 'cubic','nearest'], loc = 'best')
plt.show()
```

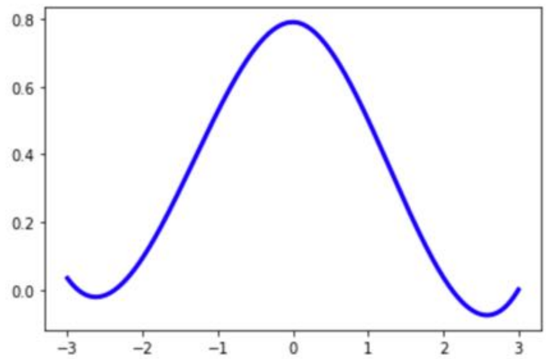


Cubic spline

```python
#scipy cubic spline
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline
x = np.linspace(-3, 3, 50)
y = np.exp(-x**2) + 0.1 * np.random.randn(50)
plt.plot(x, y, 'ro', ms = 5)
plt.show()
```



```python
spl = UnivariateSpline(x, y)
xs = np.linspace(-3, 3, 1000)
#fine smoothing
spl.set_smoothing_factor(1.1)
plt.plot(xs, spl(xs), 'b', lw = 3)
plt.show()
```



linalg

```
#scipy linalg
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy arrays
a = np.array([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
b = np.array([10, 8, 3])

#Passing the values to the solve function
x = linalg.solve(a, b)

#printing the result array
print (x)
```

```
[-9.28  5.16  0.76]
```

$$x + 3y + 5z = 10$$

$$2x + 5y + z = 8$$

$$2x + 3y + 8z = 3$$

Root finding

```
import numpy as np
from scipy.optimize import root
def func(x):
    return x*2 + 2 * np.cos(x)
sol = root(func, 0.3, method="lm")
print ("Solution response :\n", sol)
print ("Solution:", sol["x"])
print ("Error:", sol["fun"])
```

```
Solution response :
     cov_x: array([[0.08925456]])
      fjac: array([[-3.34722404]])
       fun: array([0.])
      ipvt: array([1], dtype=int32)
   message: 'The relative error between twc
0.000000'
      nfev: 12
       qtf: array([2.49797583e-09])
    status: 2
   success: True
         x: array([-0.73908513])
Solution: [-0.73908513]
Error: [0.]
```

- **lm**: Use least squares with Levenberg-Marquardt
- **hybr**: Find the roots of a multivariate function using MINPACK's hybrd and hybrj routines (modified Powell method)
- broyden1
- broyden2
- anderson
- linearmixing
- diagbroyden
- excitingmixing
- kyrlov
- df-sane

$$x^2 + 2\cos(x) = 0$$

Optimization

```python
from scipy.optimize import minimize

def eqn(x):
  return x**2 + x + 2

#use Broyden-Fletcher-Goldfarb-Shanno algorithm
mymin = minimize(eqn, 0, method='BFGS')
print(mymin)
```

```
      fun: 1.75
 hess_inv: array([[0.50000001]])
      jac: array([0.])
  message: 'Optimization terminated successfully.'
     nfev: 8
      nit: 2
     njev: 4
   status: 0
  success: True
        x: array([-0.50000001])
```


(backup slides)

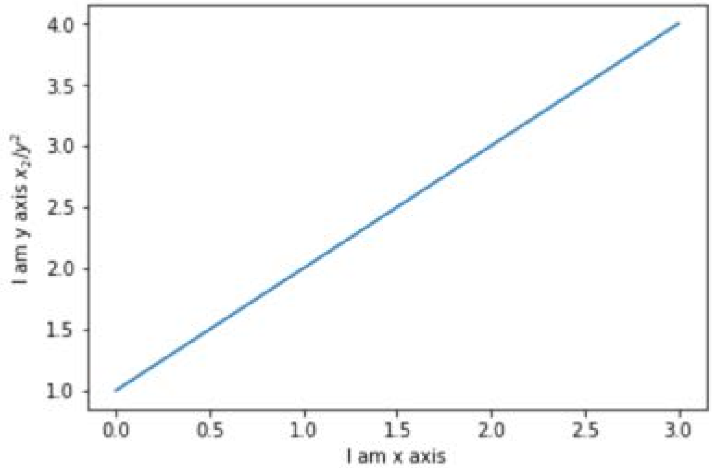
- We'll first investigate an important python library: **matplotlib**

matplotlib

Simple Plot. The most basic plot(), with text labels

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('I am y axis $x_2/{y^2}$')
plt.xlabel('I am x axis')
plt.show()
```

Only y data is provided. x automatically becomes [0,1,2,3]



LaTeX is possible in texts

See that data is given as a list

matplotlib

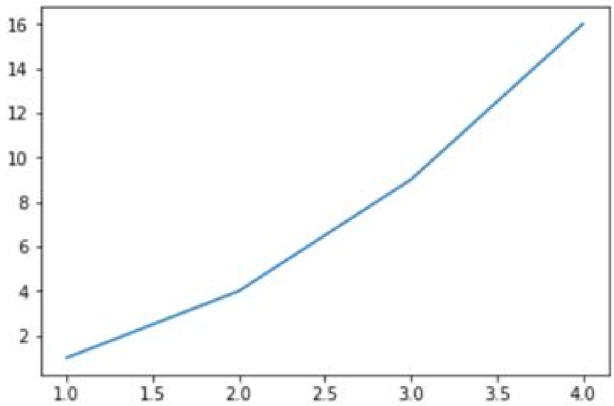
plot() is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

plt.plot([1,2,3,4], [1,4,9,16])

there is an optional third argument which is the format string that indicates the color and line type of the plot.

```
plt.plot([1,2,3,4], [1,4,9,16])
```

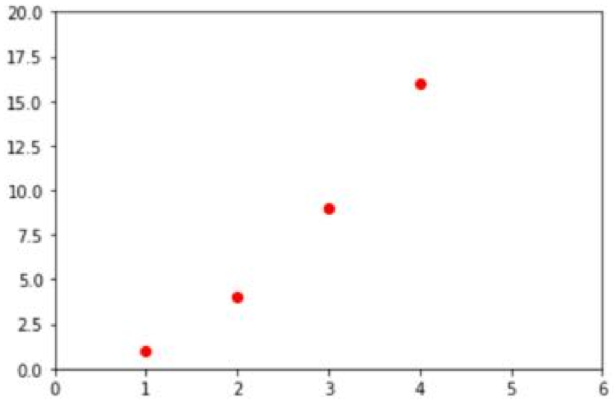
```
[<matplotlib.lines.Line2D at 0x1135c3860>]
```



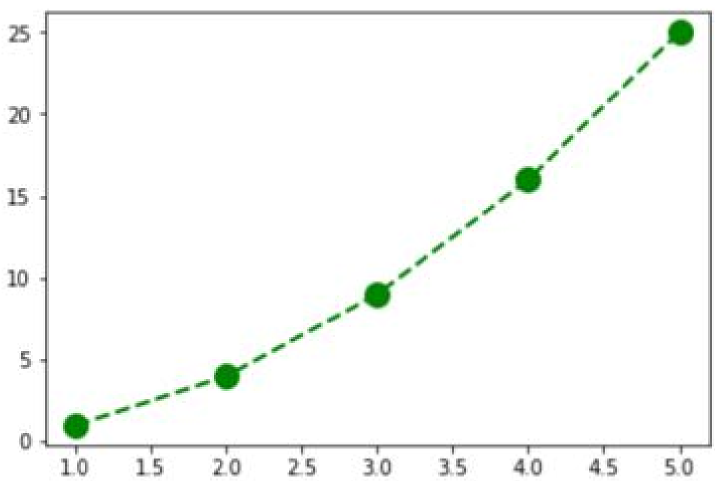
matplotlib –miscellaneous examples

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

r ➔ red
o ➔ circle

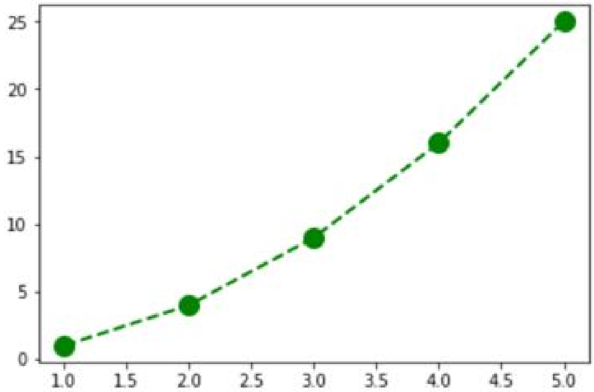



```
x=[1,2,3,4,5]
y=[1,4,9,16,25]
plt.plot(x, y, 'go--', linewidth=2, markersize=12)
plt.show();
```



Try changing values in **plt.plot**

```
x=[1,2,3,4,5]
y=[1,4,9,16,25]
plt.plot(x, y, color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12)
plt.show()
```

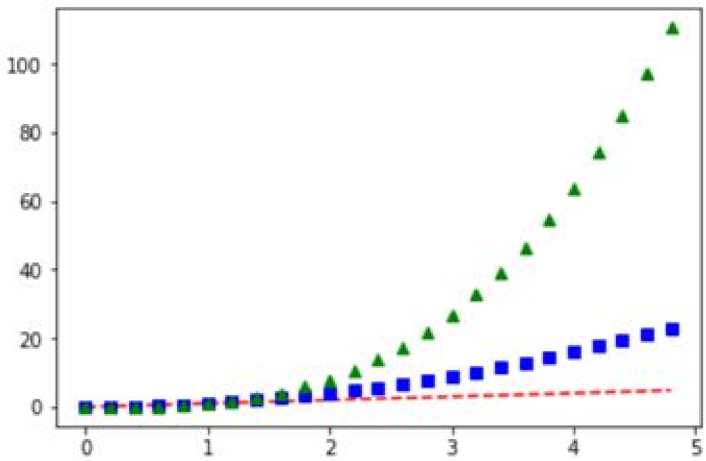


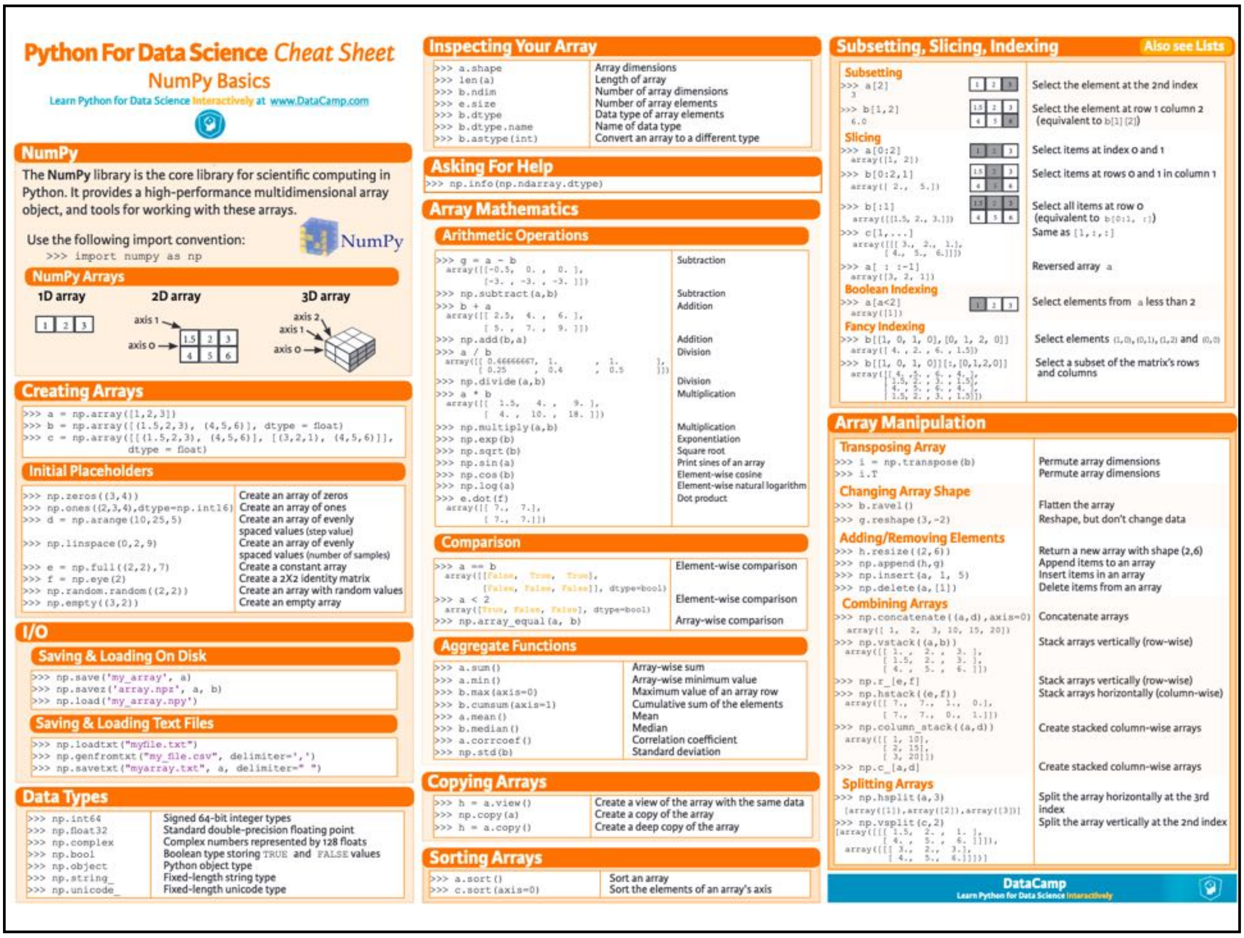
Linestyles: dashed, dotted, solid,- , -- , -. , . , :

Markers: o x X . , v < > ^ 1 2 3 4 8 s p P * h H d D |

Matplotlib –many graphics at once

```
import matplotlib.pyplot as plt
t= [float(x)/10.0 for x in range(0,50,2)]
t2=[x**2 for x in t]
t3=[x**3 for x in t]
plt.plot(t,t,'r--', t,t2,'bs', t,t3,'g^')
plt.show()
```





Python For Data Science Cheat Sheet

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com

NumPy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:
```
>>> import numpy as np
```

NumPy Arrays

1D array, 2D array, 3D array (axis 0, axis 1, axis 2)

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

`>>> np.zeros((3,4))`	Create an array of zeros
`>>> np.ones((2,3,4),dtype=np.int16)`	Create an array of ones
`>>> d = np.arange(10,25,5)`	Create an array of evenly spaced values (step value)
`>>> np.linspace(0,2,9)`	Create an array of evenly spaced values (number of samples)
`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing TRUE and FALSE values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

Inspecting Your Array

`>>> a.shape`	Array dimensions
`>>> len(a)`	Length of array
`>>> b.ndim`	Number of array dimensions
`>>> e.size`	Number of array elements
`>>> b.dtype`	Data type of array elements
`>>> b.dtype.name`	Name of data type
`>>> b.astype(int)`	Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

`>>> g = a - b`	Subtraction
`>>> np.subtract(a,b)`	Subtraction
`>>> b + a`	Addition
`>>> np.add(b,a)`	Addition
`>>> a / b`	Division
`>>> np.divide(a,b)`	Division
`>>> a * b`	Multiplication
`>>> np.multiply(a,b)`	Multiplication
`>>> np.exp(b)`	Exponentiation
`>>> np.sqrt(b)`	Square root
`>>> np.sin(a)`	Print sines of an array
`>>> np.cos(b)`	Element-wise cosine
`>>> np.log(a)`	Element-wise natural logarithm
`>>> e.dot(f)`	Dot product

Comparison

`>>> a == b`	Element-wise comparison
`>>> a < 2`	Element-wise comparison
`>>> np.array_equal(a, b)`	Array-wise comparison

Aggregate Functions

`>>> a.sum()`	Array-wise sum
`>>> a.min()`	Array-wise minimum value
`>>> b.max(axis=0)`	Maximum value of an array row
`>>> b.cumsum(axis=1)`	Cumulative sum of the elements
`>>> a.mean()`	Mean
`>>> b.median()`	Median
`>>> a.corrcoef()`	Correlation coefficient
`>>> np.std(b)`	Standard deviation

Copying Arrays

`>>> h = a.view()`	Create a view of the array with the same data
`>>> np.copy(a)`	Create a copy of the array
`>>> h = a.copy()`	Create a deep copy of the array

Sorting Arrays

`>>> a.sort()`	Sort an array
`>>> c.sort(axis=0)`	Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Also see Lists

Subsetting	
`>>> a[2]`	Select the element at the 2nd index
`>>> b[1,2]`	Select the element at row 1 column 2 (equivalent to b[1][2])
Slicing	
`>>> a[0:2]`	Select items at index 0 and 1
`>>> b[0:2,1]`	Select items at rows 0 and 1 in column 1
`>>> b[:1]`	Select all items at row 0 (equivalent to b[0:1, :])
`>>> c[1,...]`	Same as [1,:,:]
`>>> a[ : :-1]`	Reversed array a
Boolean Indexing	
`>>> a[a<2]`	Select elements from a less than 2
Fancy Indexing	
`>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]`	Select elements (1,0),(0,1),(1,2) and (0,0)
`>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]`	Select a subset of the matrix's rows and columns

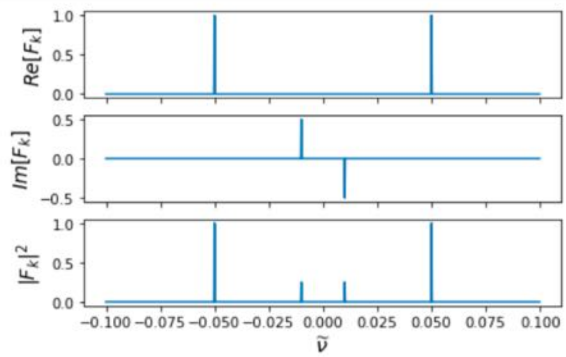
Array Manipulation

Transposing Array	
`>>> i = np.transpose(b)`	Permute array dimensions
`>>> i.T`	Permute array dimensions
Changing Array Shape	
`>>> b.ravel()`	Flatten the array
`>>> g.reshape(3,-2)`	Reshape, but don't change data
Adding/Removing Elements	
`>>> h.resize((2,6))`	Return a new array with shape (2,6)
`>>> np.append(h,g)`	Append items to an array
`>>> np.insert(a, 1, 5)`	Insert items in an array
`>>> np.delete(a,[1])`	Delete items from an array
Combining Arrays	
`>>> np.concatenate((a,d),axis=0)`	Concatenate arrays
`>>> np.vstack((a,b))`	Stack arrays vertically (row-wise)
`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)
`>>> np.hstack((e,f))`	Stack arrays horizontally (column-wise)
`>>> np.column_stack((a,d))`	Create stacked column-wise arrays
`>>> np.c_[a,d]`	Create stacked column-wise arrays
Splitting Arrays	
`>>> np.hsplit(a,3)`	Split the array horizontally at the 3rd index
`>>> np.vsplit(c,2)`	Split the array vertically at the 2nd index

DataCamp
Learn Python for Data Science Interactively

(OPTIONAL) Numpy FFT

```python
#numpy FFT :-)
#optional topic
from numpy import fft
import numpy as np
import matplotlib.pyplot as plt
n = 1000    # Number of data points
dx= 5.0     # Sampling period (in meters)
x = dx*np.arange(0,n)    # x coordinates
w1 = 100.0  # wavelength (meters)
w2 = 20.0   # wavelength (meters)
fx= np.sin(2*np.pi*x/w1) + 2*np.cos(2*np.pi*x/w2) # signal
Fk= fft.fft(fx)/n     # Fourier coefficients (divided by n)
nu= fft.fftfreq(n,dx)  # Natural frequencies
Fk= fft.fftshift(Fk)   # Shift zero freq to center
nu= fft.fftshift(nu)   # Shift zero freq to center
f, ax = plt.subplots(3,1,sharex=True)
ax[0].plot(nu, np.real(Fk))    # Plot Cosine terms
ax[0].set_ylabel(r'$Re[F_k]$', size = 'x-large')
ax[1].plot(nu, np.imag(Fk))          # Plot Sine terms
ax[1].set_ylabel(r'$Im[F_k]$', size = 'x-large')
ax[2].plot(nu, np.absolute(Fk)**2)  # Plot spectral power
ax[2].set_ylabel(r'$\vert F_k\vert ^2$', size = 'x-large')
ax[2].set_xlabel(r'$\widetilde{\nu}$', size = 'x-large')
plt.show()
```



SEE YOU NEXT WEEK!!!

DR. ORHAN GÖKÇÖL
gokcol@gmail.com
orhan.gokcol@ozyegin.edu.tr