NumPy & SciPy in Python

Computer Science & Engineering Department

The Chinese University of Hong Kong

Scientific Computing

- With its popular acceptance by scientific community, Python is being used extensively in scientific computing
- open-source add-on modules to Python that provide common mathematical and numerical routines in pre-compiled, fast functions.

- Numpy : provides basic routines for manipulating large arrays and matrices of numeric data

- Mimic Matlab operations

Numpy & Matplotlib

- Matrix/array manipulations are needed frequently in scientific computing
- Plotting of charts are also critical in monitoring the machine learning or other applications

- Python use Numpy as building blocks for many AI applications
- Matplotlib is an essential tool to plot a chart easily

Numpy Basic

- provides basic routines for manipulating large arrays and matrices of numeric data

- a table of elements (usually numbers), all of the *same type*, indexed by a tuple of positive integers.

- dimensions are called *axes*.
- number of axes is *rank*.

Numpy Basic

- E.g., coordinates of a point in 3D [1, 2, 1] : an array of rank 1, (it has one axis)
- That axis has a length of 3.
- eg. [[1.0, 1.0, 2.0], [0.0, 2.0, 1.0]] rank of 2 (or 2 dimensions)
- first dimension (or axis) : length = 2
- Second dimension (or axis) : length = 3

Numpy Basic

```
1  import numpy as np
2  a  = np.array([1, 4, 5, 8], float)
3  print(a)
4  print (type(a))
5
6
7
```

```
[1. 4. 5. 8.]
<class 'numpy.ndarray'>
```

Numpy Basic

- We can index array just like we have done to list in Python
  a = `[1. 4. 5. 8.]`

```
1  print(a[:2])
2  print(a[3])
3  a[3] = 100.0
4  print(a)
5
6
7
```

```
[1. 4.]
8.0
[  1.   4.   5. 100.]
```

Numpy Basic

- Let's try higher dimensional array

```
1  a = np.array([[1,2,3], [4,5,6]], float)  # define 2x3 array
2  print ('a =', a)
3  a[0,0] = 15.0   # we can re-assign elements in the array
4  a[1][0] = 12.0
5  print('a =',a)
6
7
```

```
a = [[1. 2. 3.]
     [4. 5. 6.]]
a = [[15.  2.  3.]
     [12.  5.  6.]]
```

Numpy Basic

- we can even using slicing

```
1  a = np.array([[1,2,3], [4,5,6]], float)  # define 2x3 array
2  print('a[1,:]=', a[1,:])
3  print('a[:,2]=', a[:,2])
4  print('a[-1:,-2:] =', a[-1:, -2:])
5
6
7
```

```
a[1,:]= [4. 5. 6.]
a[:,2]= [3. 6.]
a[-1:,-2:] = [[5. 6.]]
```

Methods on Array

- To find out the "dimension" of an array

```
1  a = np.array([[1,2,3], [4,5,6]], float)  # define 2x3 array
2
3  print(a.shape)
4  print (a.dtype)
5
6  print('length of the first axis =', len(a))
7  print('length of the second axis =', len(a[0]))  # length of the 2nd axis
```

```
(2,3)
float64
length of the first axis = 2
length of the second axis = 3
```

Methods on Array

- test whether an element is in the array

```
1  a = np.array([[1,2,3], [4,5,6]], float)  # define 2x3 array
2  print ("Is 2 in a? ", 2 in a)  # test  for membership
3  print ("Is 9 in a? ", 9 in a)
4  a = np.array(range(10), float)  # generate float array with a single item
5  print('float a =', a)
6  a = np.array([range(3), range(3)], int) # generate integer array with 2 items
7  print('integer a = ', a)
```

```
Is 2 in a?  True
Is 9 in a?  False
float a = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
integer a =  [[0 1 2]
 [0 1 2]]
```

Methods on Array

- use reshape() method to re-arrange an array

```
1  a = np.array(range(10), float)
2  print('Before reshape(), a =', a)
3  a = a.reshape(2,5)
4  print('After 1st reshape(), a =', a)
5  a = a.reshape(5,2)
6  print('After 2nd reshape(), a =', a)
7
```

```
Before reshape(), a = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
After 1st reshape(), a = [[0. 1. 2. 3. 4.]
                          [5. 6. 7. 8. 9.]]
After 2nd reshape(), a = [[0. 1.]
 [2. 3.]  [4. 5.] [6. 7.] [8. 9.]]
```

Reference to Array

- Array assignment is just a reference copy

```
1  a = np.array(range(5), float)
2  b = a
3  print ('a=', a, 'b=', b)
4
5  a[0] = 10.0   # reassign an item in a
6  print ('a=', a, 'b=', b)
7
```

```
a= [0. 1. 2. 3. 4.] b= [0. 1. 2. 3. 4.]
a= [10.  1.  2.  3.  4.] b= [10.  1.  2.  3.  4.]
```

Reference to Array

- If we want to have another array, use copy method

```
1  a = np.array(range(5), float)
2  b = a
3  c = a.copy()
4  print ('a=', a, 'b=', b, 'c=', c)
5
6  a[0] = 10.0   # reassign an item in a
7  print ('a=', a, 'b=', b, 'c=',c)
```

```
a= [0. 1. 2. 3. 4.] b= [0. 1. 2. 3. 4.] c= [0. 1. 2. 3. 4.]
a= [10.  1.  2.  3.  4.] b= [10.  1.  2.  3.  4.] c= [0. 1. 2. 3. 4.]
```

Array methods

- Define an array and fill it with some entries

```
1  a = np.array(range(5), float)
2  print('a =', a)
3  a.fill(0)   # use fill method to initialize the array
4  print('a =', a)
5  a.fill(100.0)
6  print('a =', a)
7
```

```
a = [0. 1. 2. 3. 4.]
a = [0. 0. 0. 0. 0.]
a = [100. 100. 100. 100. 100.]
```

Array methods

- Concatenate

```
1  a = np.array([1,2], float)        # float type
2  b = np.array([3,4,5], float)      # float type
3  c = np.array([7,8,9,10], int)     # integer type
4  d = np.concatenate((a,b,c))
5  print('a =',a)
6  print('b =',b)
7  print('c =',c)
8  print('d =',d)
```

```
a = [1. 2.]
b = [3. 4. 5.]
c = [ 7  8  9 10]
d = [ 1.  2.  3.  4.  5.  7.  8.  9. 10.]
```

Array methods

- arange() similar to range() function except it returns an array.

```
1  a = np.array(range(5), float)
2  print('a = ', a)
3  print('type of a: ', type(a))
4
5  b = np.arange(5, dtype=float)
6  print('b = ', b)
7  print('type of b: ', type(b))
8
```

```
a =  [0. 1. 2. 3. 4.]
type of a:  <class 'numpy.ndarray'>
b =  [0. 1. 2. 3. 4.]
type of b:  <class 'numpy.ndarray'>
```

Array methods

- Zeros and ones set value 0 , 1 accordingly

```
1  a = np.zeros(7,dtype=int)
2  print('a=',a)
3
4  b = np.zeros((2,3),dtype=int)
5  print('b=',b)
6
7  c = np.ones((3,2),dtype=float)
8  print('c=',c)
```

```
a= [0 0 0 0 0 0 0]
b= [[0 0 0]
 [0 0 0]]
c= [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

Matrix/Vector methods

- eye : returns matrices with ones along the k-th diagonal

```
1  a = np.identity(3, dtype=float)  # create a 3x3 identity matrix
2  print('a=',a)
3
4  a = np.eye(3, k=1, dtype=float)  # create a 3x3 matrix with 1st diagonal being 1
5  print('a=',a)
6
7
8
```

```
a= [[1. 0. 0.]
    [0. 1. 0.]
    [0. 0. 1.]]
a= [[0. 1. 0.]
    [0. 0. 1.]
    [0. 0. 0.]]
```

Matrix/Vector Operations

- for higher dimension arrays, it remains to be an element-wise operation

```
1  c = np.array([[1,2,3,4,5],[1,1,1,1,1]],int)
2  d = np.array([[5,4,3,2,1],[2,2,2,2,2]],float)
3  print('c+d =', c+d)
4  print('c-d =', c-d)
5  print('c*d =', c*d)
6  print('c/d =', c/d)
7  print('c%d =', c%d)
8  print('c**d =',c**d)
```

```
c+d = [[6. 6. 6. 6. 6.]
       [3. 3. 3. 3. 3.]]
c-d = [[-4. -2.  0.  2.  4.]
       [-1. -1. -1. -1. -1.]]
c*d = [[5. 8. 9. 8. 5.]
       [2. 2. 2. 2. 2.]]
```

```
c/d = [[0.2 0.5 1.  2.  5. ]
       [0.5 0.5 0.5 0.5 0.5]]
c%d = [[1. 2. 0. 0. 0.]
       [1. 1. 1. 1. 1.]]
c**d = [[ 1. 16. 27. 16.  5.]
        [ 1.  1.  1.  1.  1.]]
```

Matrix/Vector methods

- Arrays that do not match in number of dimensions will be broadcasted

```
1  a = np.array([[1,2], [3,4], [5,6]], int)    # 3 x 2 array
2  b = np.array([-1,3], float)                 # 1 x 2
3  print('a =', a)
4  print('b =', b, end='\n\n')
5  print('a+b =', a+b, end='\n\n')             # 3 x 2
6  print('a*b =', a*b, end='\n\n')
7
8
```

```
a = [[1 2]
     [3 4]
     [5 6]]
b = [-1.  3.]

a+b = [[0. 5.]
       [2. 7.]
       [4. 9.]]
```

```
a*b = [[-1.  6.]
       [-3. 12.]
       [-5. 18.]]
```

Matrix/Vector methods

- Array iterations

```
1  a = np.array([[1,2],[3,4],[5,6]], dtype=int)
2
3  for num in a:
4      print('number = ', num)
5
6
7
8
```

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

Supplementary

More Array/Matrix/Vector Operations, Polynomial & SciPy

Basic Array Operations

```
1  a = np.array([2,4,3], dtype=float)
2  b = np.array([[1,2],[3,4]], dtype=float)
3  print('a =', a)
4  print('b =', b, end='\n\n')
5  print('element sum of a = ', a.sum())  # sum all elements
6  print('element sum of b = ', b.sum())  # sum all elements
7  print('element prduct of a = ', a.prod())  # multiply all elements
8  print('element product  of b = ', b.prod())  # multiply all elements
```

```
a = [2. 4. 3.]
b = [[1. 2.]
 [3. 4.]]
element sum of a =  9.0
element sum of b =  10.0
element prduct of a =  24.0
element product  of b =  24.0
```

Basic Array Operations

- Another way is to use NumPy method with array as argument

```
1  a = np.array([2,4,3], dtype=float)
2  b = np.array([[1,2],[3,4]], dtype=float)
3
4  print('element sum of a = ', np.sum(a))  # sum all elements
5  print('element sum of b = ', np.sum(b))  # sum all elements
6  print('element prduct of a = ', np.prod(a))  # multiply all elements
7  print('element product  of b = ', np.prod(b)) # multiply all elements
8
```

```
element sum of a =  9.0
element sum of b =  10.0
element prduct of a =  24.0
element product  of b =  24.0
```

Basic Array Operations

- sort all entries in an array

```
1  a = np.array([6, 2, 5, -1, 0],float)
2  print('a =', a)
3
4  print('sorted form of a =', sorted(a))
5  print('clipped form of a =', a.clip(0,4.1)) # specify lower/upper bound
6
7
8
```

```
a = [ 6.  2.  5. -1.  0.]
sorted form of a = [-1.0, 0.0, 2.0, 5.0, 6.0]
clipped form of a = [4.1 2.  4.1 0.  0. ]
```

Basic Array Operations

- finding unique entries in an array

```
1  a = np.array([1, 1, 2, 2, 3, 4, 4, 5, 5, 5],float)
2  print('a =', a)
3  print('unqiue entries of a :', np.unique(a))
4
5
6
7
8
```

```
a = [1. 1. 2. 2. 3. 4. 4. 5. 5. 5.]
unqiue entries of a : [1. 2. 3. 4. 5.]
```

Basic Array Operations

- finding diagonal entries in an array

```
1  a = np.array([[1,2,3],[4,5,6],[7,8,9]], int)
2  print('a =', a)
3  print('diagonal entries of a are :', a.diagonal())
4
5
6
7
8
```

```
a = [[1 2 3]
     [4 5 6]
     [7 8 9]]
diagonal entries of a are : [1 5 9]
```

Array comparison & testing

```
1  a = np.array([1, 3, 0], float)
2  b = np.array([0, 3, 2], float)
3  print('a=',a, '; b=', b)
4
5  print('Is a > b: ', a>b)   # a>b returns an array of boolean
6  print('Is a == b:', a==b)
7  print('Is a <= b:', a<=b)
8
```

```
a= [1. 3. 0.] ; b= [0. 3. 2.]
Is a > b:  [ True False False]
Is a == b: [False  True False]
Is a <= b: [False  True  True]
```

Array comparison & testing

- use of logical_and, logical_or and logical_not in array

```
1  a = np.array([1,3,0], float)
2  print('a =', a)
3  b = np.logical_and(a>0, a<3)
4  print('Are entries in a > 0 AND a < 3: ', b)
5  c = np.logical_not(b)
6  print('Use of logical_not in a: ', c)
7  print('Use of logical_or: ', np.logical_or(b,c))
8
```

```
a = [1. 3. 0.]
Are entries in a > 0 AND a < 3:  [ True False False]
Use of logical_not in a:  [False  True  True]
Use of logical_or:  [ True  True  True]
```

Array comparison & testing

- where forms a new array from two arrays of equivalent size using a Boolean filter to choose between elements of the two

  np.where (boolarray, truearray, falsearray)

```
1  a = np.array([1, 3, 0], float)
2  b = np.where(a > 0, a+1, a)
3  c = np.where(a > 0, 5.0, -1.0)
4  print('a =', a)
5  print('b =', b)
6  print('c =', c)
7
8
```

```
a = [1. 3. 0.]
b = [2. 4. 0.]
c = [ 5.  5. -1.]
```

Array comparison & testing

- test whether or not values are NaN ("not a number") or finite

```
1  a = np.array([2, np.NaN, np.Inf], float)
2  print('a =', a)
3  print('Entry is not a number :', np.isnan(a))
4  print('Entry is finite :', np.isfinite(a))
5
6
7
8
```

```
a = [ 2. nan inf]
Entry is not a number : [False  True False]
Entry is finite : [ True False False]
```

Statistics in an array

- find the mean and variance of a series

```
1  a = np.array([2, 1, 3, 10.0, 5.3, 18.2, 16.3],dtype=float)
2  mean = a.sum()/len(a)
3  print('mean of a =', mean)
4  print('mean of a =', a.mean(), end='\n')
5  print('variance of a =', a.var())
6  print('my standard deviation of a =', np.sqrt(a.var()))
7  print('standard deviation of a =', a.std())
8
```

```
mean of a = 7.971428571428571
mean of a = 7.971428571428571
variance of a = 42.03061224489795
my standard deviation of a = 6.483102054178844
standard deviation of a = 6.483102054178844
```

Statistics in an array

- find the min or max or argmin or argmax in a series

```
1  a = np.array([2, 1, 3, 10.0, 5.3, 18.2, 16.3],dtype=float)
2
3  print('a =', a)
4  print('minimum element in a =', a.min())
5  print('minimum occurs in index:', a.argmin())
6  print('maximum element in a =', a.max())
7  print('maximum occurs in index:', a.argmax())
8
```

```
a = [ 2.   1.   3.  10.   5.3 18.2 16.3]
minimum element in a = 1.0
minimum occurs in index: 1
maximum element in a = 18.2
maximum occurs in index: 5
```

Statistics in an array

- can even control which axis to take the statistics

```
1  a = np.array([[0,2],[3,-1],[3,5]], dtype=float)
2  print('a =', a)
3  print('mean in axis 0 = ', a.mean(axis=0))
4  print('mean in axis 1 = ', a.mean(axis=1))
5  print('min  in axis 0 = ', a.min(axis=0))
6  print('min  in axis 1 = ', a.min(axis=1))
7  print('max  in axis 0 = ', a.max(axis=0))
8  print('max  in axis 1 = ', a.max(axis=1))
```

```
a = [[ 0.  2.]
 [ 3. -1.]
 [ 3.  5.]]
mean in axis 0 =  [2. 2.]
mean in axis 1 =  [1. 1. 4.]
```

```
min  in axis 0 =  [ 0. -1.]
min  in axis 1 =  [ 0. -1.  3.]
max  in axis 0 =  [3. 5.]
max  in axis 1 =  [2. 3. 5.]
```

Array item selection & Manipulation

```
1  a = np.array([[6,4], [5,9]], float)  # define a 2x2 array
2  print('a=', a)
3
4  b = (a >= 6)
5  print('b = ', b, '. Type of b:', type(b))
6
7  c = a[b]
8  print('c = ', c, '. Type of c:', type(c))
```

```
a= [[6. 4.]
    [5. 9.]]
b =  [[ True False]
      [False  True]] . Type of b: <class 'numpy.ndarray'>
c =  [6. 9.] . Type of c: <class 'numpy.ndarray'>
```

Array item selection & Manipulation

```
1  a = np.array([[6,4], [5,9]], float)  # define a 2x2 array
2  print('a=', a)
3
4  b = a[np.logical_and(a > 5, a <9)]
5  print('b = ', b, '. Type of b:', type(b))
6
7
8
```

```
a= [[6. 4.]
 [5. 9.]]
b =  [6.] . Type of b: <class 'numpy.ndarray'>
```

Array item selection & Manipulation

```
1  a = np.array([2, 4, 6, 8], float)
2  print('a=', a, '; b=', b)
3  b = np.array([0, 0, 1, 3, 2, 1], int)  # it has to be an integer array
4  c = a[b]   # create array c
5  print('c = ', c, '. Type of c:', type(c))
6
7
8
```

```
a= [2. 4. 6. 8.] ; b= [6.]
c =  [2. 2. 4. 8. 6. 4.] . Type of c: <class 'numpy.ndarray'>
```

Array item selection & Manipulation

- For multidimensional arrays, we have to use multiple one-dimensional integer array to the selection bracket, one for each axis

```
1  a = np.array([[1,4], [9,16]], float)
2  b = np.array([0, 0, 1, 1, 0], int)
3  c = np.array([0, 1, 1, 1, 1], int)
4  d = a[b,c]
5
6  print('a =', a)
7  print('d =', d, '; type of d is:', type(d))
8
```

```
a = [[ 1.  4.]
 [ 9. 16.]]
d = [ 1.  4. 16. 16.  4.] ; type of d is: <class 'numpy.ndarray'>
```

Vector & matrix mathematics

- Math functions also apply to array

```
1  a = np.array([1,4,9,16,25],float)
2
3  print ('Square root of a =', np.sqrt(a)) # use square root
4  print ('Sign of a =', np.sign(a))        # use sign
5  print ('exp of a =', np.exp(a))          # use exponential
6  print ('log of a =', np.log(a))          # use log
7  print ('log10 of a =', np.log10(a))      # use log10
8
```

```
Square root of a = [1. 2. 3. 4. 5.]
Sign of a = [1. 1. 1. 1. 1.]
exp of a = [2.71828183e+00 5.45981500e+01 8.10308393e+03 8.88611052e+06
7.20048993e+10]
log of a = [0.         1.38629436 2.19722458 2.77258872 3.21887582]
log10 of a = [0.          0.60205999 0.95424251 1.20411998 1.39794001]
```

Vector & matrix mathematics

- can do dot products and matrix multiplication,...etc

```
1  a = np.array([[1,2], [3,4]], float)
2  b = np.array([2,3], float)
3  c = np.array([[1,1], [4,0]], float)
4  print('a=', a, end='\n\n')
5  print('b=', b, end='\n\n')
6  print('c=', c, end='\n\n')
7  print('b*a =', np.dot(b,a))
8  print('a*b =', np.dot(a,b))
9  print('a*c =', np.dot(a,c))
```

```
a= [[1. 2.]
    [3. 4.]]
b= [2. 3.]
c= [[1. 1.]
    [4. 0.]]
```

```
b*a = [11. 16.]
a*b = [ 8. 18.]
a*c = [[ 9.  1.]
 [19.  3.]]
c*a = [[4. 6.]
 [4. 8.]]
```

Vector & matrix mathematics

- inner, outer and cross prodcuts of matrices and vectors

```
1  a = np.array([1,4,0], float)
2  b = np.array([2,2,1], float)
3  print('a=', a, '; b=', b)
4  print('np.outer(a,b)=', np.outer(a,b))
5  print('np.inner(a,b)=', np.inner(a,b))
6  print('np.cross(a,b)=', np.cross(a,b))
7
8
9
```

```
a= [1. 4. 0.] ; b= [2. 2. 1.]
np.outer(a,b)= [[2. 2. 1.]
 [8. 8. 4.]
 [0. 0. 0.]]
```

```
np.inner(a,b)= 10.0
np.cross(a,b)= [ 4. -1. -6.]
```

Vector & matrix mathematics

- Determinants, inverse and eigenvalues

```
1  a = np.array([[4, 2, 0], [9, 3, 7],[1, 2, 1]], float)
2  print ('a =', a)
3  print('determinant of a is: ', np.linalg.det(a)) # get determinant
4  vals, vecs = np.linalg.eig(a)   # get eigenvalues/engenvectors
5  print('eigenalues: ', vals)
6  print('eigenvectors: ', vecs)
7  b = np.linalg.inv(a)     # compute inverse of a
8  print('inverse of a = ', b)
9  print("let's check, a * b = ", np.dot(a,b))
```

```
a = [[4. 2. 0.]
 [9. 3. 7.]
 [1. 2. 1.]]
determinant of a is:  -48.00000000000003
eigenalues:  [ 8.85591316  1.9391628  -2.79507597]
eigenvectors:  [[-0.3663565  -0.54736745  0.25928158]
 [-0.88949768  0.5640176  -0.88091903]
 [-0.27308752  0.61828231  0.39592263]]
```

```
inverse of a =  [[ 0.22916667  0.04166667 -0.29166667]
 [ 0.04166667 -0.08333333  0.58333333]
 [-0.3125      0.125       0.125     ]]
let's check, a * b =  [[1.00000000e+00 5.55111512e-17
0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 2.22044605e-16]
 [0.00000000e+00 1.38777878e-17 1.00000000e+00]]
```

Vector & matrix mathematics

- Singular Value Decomposition (SVD)

```
1  a = np.array([[1,3,4], [5,2,3]], float)
2  U, s, Vh = np.linalg.svd(a)   # get SVD of a
3  print ('U is:', U)
4  print('s is:', s)
5  print('Vh is:', Vh)
6
7
8
9
```

```
U is: [[-0.6113829  -0.79133492]
 [-0.79133492  0.6113829 ]]
s is: [7.46791327 2.86884495]
```

```
Vh is: [[-0.61169129 -0.45753324 -
0.64536587]
 [ 0.78971838 -0.40129005 -0.46401635]
 [-0.046676   -0.79349205  0.60678804]]
```

Polynomial mathematics

- Evaluate polynomial at a given point
- Let's say we want to evaluate x^3-2x^2+2 at $x=4$.

```
1  np.polyval([1,-2,0, 2], 4)
2
3
4
5
6
```

```
34
```

Polynomial mathematics

- for the polynomial: $x^4-11x^3+9x^2+11x-10$.
- If we know the roots are (-1, 1, 1, 10), poly method can find coefficient.

```
1  print('Polynomial coefficients are: ', np.poly([-1, 1, 1, 10]))
2
3
4
5
6
```

```
Polynomial coefficients are:  [  1. -11.   9.  11. -10.]
```

Polynomial mathematics

- Given a set of coefficients in a polynomial, we can find the roots
- x^3+4x^2-2x+3

```
1  np.roots([1, 4, -2, 3])   # get roots
2
3
4
5
6
```

```
array([-4.5797401 +0.j         ,  0.28987005+0.75566815j,
       0.28987005-0.75566815j])
```

Polynomial mathematics

- For polynomial: $x^4+x^3+x^2+x+1$
- derivative

```
1  np.polyder([1/4, 1/3, 1/2, 1, 0])  # perform derivative of polynomial (specify coefficients)
2
3
4
5
6
```

```
array([1., 1., 1., 1.])
```

Polynomial mathematics

- For polynomial: x^3+x^2+x+1
- If we integrate it, we should have $x^4/4+x^3/3+x^2/2+x+C$, where C is a constant

```
1  np.polyint([1,1,1,1])  # perform integration on a polynomial (specify coefficients)
2
3
4
5
6
```

```
array([0.25      , 0.33333333, 0.5       , 1.        , 0.        ])
```

Polynomial mathematics

- can use polyfit() method, which fits a polynomial of specified order to a set of data using the least-square method.

```
1  x = [1, 2, 3, 4, 5, 6, 7, 8]
2  y = [0, 2, 1, 3, 7, 10, 11, 19]
3
4  # use polyfit to find the least square fit using polynomail of degree 2
5  np.polyfit(x,y,2)     # it returns all coefficients
6
```

```
array([ 0.375     , -0.88690476,  1.05357143])
```

Polynomial mathematics

- Note that there are other polynomial methods
- polyadd
- polysub
- polymul
- polydiv
- Polyval

- Read the documentation of Numpy

SciPy

- Greatly extends the functionality of NumPy
- We can use "import scipy" to import the module
- SciPy has **many packages**
- To explore, do **help(scipy)**

SciPy (Linear Algebra scipy.linalg)

- #Matrix determinant

```
1  import numpy as np
2  from scipy import linalg
3  arr = np.array([[1,2], [3,4]], float)
4  print('arr =\n', arr)
5  print("arr's determinant is: ", linalg.det(arr))
6
```

```
arr =
 [[1. 2.]
 [3. 4.]]
arr's determinant is:  -2.0
```

SciPy (Linear Algebra scipy.linalg)

- Matrix inverse

```
1  arr_inv = linalg.inv(arr)
2  print("arr's inverse is:\n", arr_inv)
3
4  # Let's do a test
5  print("\narr * arr_inv is:\n", np.dot(arr, arr_inv))
6
```

```
arr's inverse is:
 [[-2.   1. ]
 [ 1.5 -0.5]]

arr * arr_inv is:
 [[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

SciPy (Linear Algebra scipy.linalg)

- Finding the roots of a scalar function

```
1  from scipy import optimize   # import optimization package
2  def f(x):
3      return x**2 + 10*np.sin(x)
4
5  root = optimize.fsolve(f,1)   # our initial guess
6  print("Guess from 1: ", root)
7
8  root = optimize.fsolve(f, -2.5) # another quess
9  print("guess from -2.5:", root)
```

```
Guess from 1:  [0.]
guess from -2.5: [-2.47948183]
```