Numpy

NumPy stands for Numerical Python.

NumPy is a Python library used for working with arrays.

add()

Returns element-wise string concatenation for two arrays of str or Unicode

```
0]: np.add(1.0, 4.0)
0]: 5.0

1]: x1 = np.arange(9.0).reshape((3, 3))
    x2 = np.arange(3.0)
    np.add(x1, x2)
1]: array([[ 0.,  2.,  4.],
           [ 3.,  5.,  7.],
           [ 6.,  8., 10.]])
```

multiply()

Returns the string with multiple concatenation, element-wise

```
: a =[1,2,3]
  b =[2,3,4]

: c=np.multiply(a,b)

: c

: array([ 2,  6, 12])
```

split()

Returns a list of the words in the string, using separator delimiter

```
: arr = np.array([1, 2, 3, 4, 5, 6])

  newarr = np.array_split(arr, 3)

  print(newarr)

  [array([1, 2]), array([3, 4]), array([5, 6])]
```

join()

Returns a string which is the concatenation of the strings in the sequence

```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

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

encode()

Calls str.encode element-wise

```
: x = np.array(['aAaAaA', '  aA  ', 'abBABba'])
  print("x:", x)

  x: ['aAaAaA' '  aA  ' 'abBABba']
```

```
: e = np.char.encode(x, encoding='cp037')
  e

: array([b'\x81\xc1\x81\xc1\x81\xc1', b'@@\x81\xc1@@',
         b'\x81\x82\xc2\xc1\xc2\x82\x81'], dtype='|S7')
```

```
: d = np.char.decode(e, encoding='cp037')
  d

: array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
```

Functions for Rounding

numpy.around()

returns the value rounded to the desired precision.

```
23]: np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value

23]: array([0., 2., 2., 4., 4.])
```

numpy.floor()

This function returns the largest integer not greater than the input parameter

Note- In Python, flooring always is rounded away from 0.

```
[21]: a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
      np.floor(a)

[21]: array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

numpy.ceil()

The ceil() function returns the ceiling of an input value, i.e. the ceil of the **scalar x** is the smallest **integer i**, such that **i >= x**

```
a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
np.ceil(a)

array([-1., -1., -0.,  1.,  2.,  2.,  2.])
```

numpy.round()

Round an array to the given number of decimals.

```
19]: import numpy as np

20]: np.round(56294995342131.5, 3)
```

numpy.rint()

Round elements of the array to the nearest integer.

```
24]: a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
     np.rint(a)

24]: array([-2., -2., -0.,  0.,  2.,  2.,  2.])
```

numpy.fix()

Round to nearest integer towards zero.

```
26]: np.fix([2.1, 2.9, -2.1, -2.9])

26]: array([ 2.,  2., -2., -2.])
```

numpy.trunc()

Return the truncated value of the input, element-wise.

```
[25]: a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
      np.trunc(a)
[25]: array([-1., -1., -0.,  0.,  1.,  1.,  2.])
```

Arithmetical Functions

numpy.reciprocal()

This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0 because of the way in which Python handles integer division.

```
: np.reciprocal([1, 2., 3.33])
: array([1.        , 0.5       , 0.3003003])
```

numpy.power()

This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array

```
5]: x1 = np.arange(6)
    x1
5]: array([0, 1, 2, 3, 4, 5])
6]: np.power(x1, 3)
6]: array([  0,   1,   8,  27,  64, 125], dtype=int32)
```

numpy.mod()

This function returns the remainder of division of the corresponding elements in the input array. The function **numpy.remainder()** also produces the same result.

```
np.remainder([4, 7], [2, 3])
```

```
array([0, 1], dtype=int32)
```

numpy.real()

returns the real part of the complex data type argument.

```
]: a = np.array([1+2j, 3+4j, 5+6j])
   a.real
]: array([1., 3., 5.])
```

numpy.conj()

returns the complex conjugate, which is obtained by changing the sign of the imaginary part.

```
: np.conjugate(1+2j)
: (1-2j)
```

numpy.angle()

returns the angle of the complex argument. The function has degree parameter. If true, the angle in the degree is returned, otherwise the angle is in radians

```
: np.angle([1.0, 1.0j, 1+1j])               # in radians
: array([0.        , 1.57079633, 0.78539816])
```

```
: np.angle(1+1j, deg=True)                  # in degrees
: 45.0
```

numpy.imag()

returns the imaginary part of the complex data type argument.

```
: a = np.array([1+2j, 3+4j, 5+6j])
  a.imag
: array([2., 4., 6.])
```

numpy.positive()

Numerical positive, element-wise.

```
]: x1 = np.array(([1., -1.]))
   np.positive(x1)
```

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

numpy.true_divide()

Returns a true division of the inputs, element-wise.

```
]: x = np.arange(5)
   np.true_divide(x, 4)
```

```
]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

numpy.floor_divide()

Return the largest integer smaller or equal to the division of the inputs.

```
]: np.floor_divide(7,3)
```

```
]: 2
```

```
]: np.floor_divide([1., 2., 3., 4.], 2.5)
```

```
]: array([0., 0., 1., 1.])
```

numpy.float_power()

First array elements raised to powers from second array, element-wise.

```
3]: np.float_power(x1, 3)
```

```
3]: array([  0.,   1.,   8.,  27.,  64., 125.])
```

numpy.fmod()

Return the element-wise remainder of division.

```
: np.fmod([-3, -2, -1, 1, 2, 3], 2)
```

```
: array([-1,  0, -1,  1,  0,  1], dtype=int32)
```

numpy.modf()

Return the fractional and integral parts of an array, element-wise.

```
np.modf([0, 3.5])
```

```
(array([0. , 0.5]), array([0., 3.]))
```

```
np.modf(-0.5)
```

```
(-0.5, -0.0)
```

numpy.divmod()

```
np.divmod(np.arange(5), 3)
```

```
(array([0, 0, 0, 1, 1], dtype=int32), array([0, 1, 2, 0, 1], dtype=int32))
```

Statistical Function

numpy.amin()

These functions return the minimum from the elements in the given array along the specified axis.

```
a = np.arange(4).reshape((2,2))
a
```

```
array([[0, 1],
       [2, 3]])
```

```
np.amin(a)
```

```
0
```

```
np.amin(a, axis=0)
```

```
array([0, 1])
```

numpy.ptp()

The **numpy.ptp()** function returns the range (maximum-minimum) of values along an axis

```
x = np.array([[4, 9, 2, 10],
              [6, 9, 7, 12]])
```

```
np.ptp(x, axis=1)
```

```
array([8, 6])
```

numpy.percentile()

Percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. The function **numpy.percentile()** takes the following arguments

]:
```
a = np.array([[10, 7, 4], [3, 2, 1]])
a
```

]:
```
array([[10,  7,  4],
       [ 3,  2,  1]])
```

]:
```
np.percentile(a, 50)
```

]:
```
3.5
```

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half. The **numpy.median()** function is used as shown in the following program

```
a = np.array([[10, 7, 4], [3, 2, 1]])
a
```

```
array([[10,  7,  4],
       [ 3,  2,  1]])
```

```
np.median(a, axis=0)
```

```
array([6.5, 4.5, 2.5])
```

numpy.mean()

The **numpy.mean()** function returns the arithmetic mean of elements in the array.

:
```
a = np.array([[1, 2], [3, 4]])
np.mean(a)
```

:
```
2.5
```

numpy.average()

The **numpy.average()** function computes the weighted average of elements in an array according to their respective weight given in another array.

:
```
data = np.arange(1, 5)
data
```

:
```
array([1, 2, 3, 4])
```

:
```
np.average(data)
```

:
```
2.5
```

:
```
np.average(np.arange(1, 11), weights=np.arange(10, 0, -1))
```

:
```
4.0
```

Sort, Search and Counting Function

numpy.sort()

The sort() function returns a sorted copy of the input array.

```
a = np.array([[1,4],[3,1]])                    #sort along last axis
np.sort(a)
```

```
array([[1, 4],
       [1, 3]])
```

```
np.sort(a, axis=0)                             #sort along first axis
```

```
array([[1, 1],
       [3, 4]])
```

```
np.sort(a, axis=None)                          #sort flattened array
```

```
array([1, 1, 3, 4])
```

numpy.argsort()

The **numpy.argsort()** function performs an indirect sort on input array, along the given axis and using a specified kind of sort to return the array of indices of data.

```
ind = np.argsort(x, axis=0)                    # sorts along first axis (down)
ind
```

```
array([[0, 0, 0, 0],
       [1, 1, 1, 1]], dtype=int32)
```

```
np.take_along_axis(x, ind, axis=0)             # same as np.sort(x, axis=0)
```

```
array([[ 4,  9,  2, 10],
       [ 6,  9,  7, 12]])
```

numpy.lexsort()

The function returns an array of indices, using which the sorted data can be obtained. Note, that the last key happens to be the primary key of sort.

```
a = [1,5,1,4,3,4,4]             # First column
b = [9,4,0,4,0,2,1]             # Second column
ind = np.lexsort((b,a))         # Sort by a, then by b
ind
```

```
array([2, 0, 4, 6, 5, 3, 1], dtype=int32)
```

numpy.nonzero()

The **numpy.nonzero()** function returns the indices of non-zero elements in the input array

```
x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
x
```

```
array([[3, 0, 0],
       [0, 4, 0],
       [5, 6, 0]])
```

```
np.nonzero(x)
```

```
(array([0, 1, 2, 2], dtype=int32), array([0, 1, 0, 1], dtype=int32))
```

numpy.where()

The **where()** function returns the indices of elements in an input array where the given condition is satisfied.

```
a = np.random.randn(2, 3)
print(a)
```

```
[[ 1.03923819 -0.38814259 -0.5789615 ]
 [-0.03429005 -1.85615805 -1.80616705]]
```

```
b = np.where(a > 0, a, 0)
b
```

```
array([[1.03923819, 0.        , 0.        ],
       [0.        , 0.        , 0.        ]])
```

numpy.extract()

The extract() function returns the elements satisfying any condition

```
arr = np.arange(12).reshape((3, 4))
arr
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
condition = np.mod(arr, 3)==0
condition
```

```
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]])
```

```
np.extract(condition, arr)
```

```
array([0, 3, 6, 9])
```

Sums, products, differences

numpy.prod()

Return the product of array elements over a given axis.

```
]: x = np.array([536870910, 536870910, 536870910, 536870910])
   np.prod(x)
]: 16
```

numpy.sum()

Sum of array elements over a given axis

```
: np.sum([[0, 1], [0, 5]])
: 6
```

```
: np.sum([[0, 1], [0, 5]], axis=0)
: array([0, 6])
```

numpy.nanprod()

Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.

```
30]: np.nanprod([1])
30]: 1
```

```
34]: a = np.array([[1, 2], [3, np.nan]])
     np.nanprod(a)
34]: 6.0
```

numpy.nansum()

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero

```
]: np.nansum([1, np.nan])
]: 1.0
```

```
]: a = np.array([[1, 1], [1, np.nan]])
   np.nansum(a)
]: 3.0
```

numpy.cumprod()

Return the cumulative product of elements along a given axis

```
7]: a = np.array([1,2,3])
    np.cumprod(a) # intermediate results 1, 1*2
                  # total product 1*2*3 = 6
```

```
7]: array([1, 2, 6], dtype=int32)
```

```
8]: a = np.array([[1, 2, 3], [4, 5, 6]])
    np.cumprod(a, dtype=float) # specify type of output
```

```
8]: array([  1.,   2.,   6.,  24., 120., 720.])
```

numpy.cumsum()

Return the cumulative sum of the elements along a given axis.

```
9]: a = np.array([[1,2,3], [4,5,6]])
    a
```

```
9]: array([[1, 2, 3],
           [4, 5, 6]])
```

```
1]: np.cumsum(a)
    np.cumsum(a, dtype=float)     # specifies type of output value(s)
```

```
1]: array([ 1.,  3.,  6., 10., 15., 21.])
```

numpy.nancumprod()

Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one.

```
2]: np.nancumprod([1, np.nan])
```

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

```
3]: a = np.array([[1, 2], [3, np.nan]])
    np.nancumprod(a)
```

```
3]: array([1., 2., 6., 6.])
```

numpy.diff()

Calculate the n-th discrete difference along the given axis.

```
16]: x = np.array([1, 2, 4, 7, 0])
     np.diff(x)
16]: array([ 1,  2,  3, -7])
```

numpy.ediff1d()

The differences between consecutive elements of an array.

```
7]: x = np.array([1, 2, 4, 7, 0])
    np.ediff1d(x)
7]: array([ 1,  2,  3, -7])
```

```
3]: y = [[1, 2, 4], [1, 6, 24]]
    np.ediff1d(y)
3]: array([ 1,  2, -3,  5, 18])
```

numpy.gradient()

Return the gradient of an N-dimensional array.

```
9]: f = np.array([1, 2, 4, 7, 11, 16], dtype=float)
    np.gradient(f)
9]: array([1. , 1.5, 2.5, 3.5, 4.5, 5. ])
```

```
0]: np.gradient(f, 2)
0]: array([0.5 , 0.75, 1.25, 1.75, 2.25, 2.5 ])
```

numpy.cross()

Return the cross product of two (arrays of) vectors.

```
: x = [1, 2, 3]
  y = [4, 5, 6]
  np.cross(x, y)
: array([-3,  6, -3])
```

numpy.trapz()

Integrate along the given axis using the composite trapezoidal rule.

```
]: np.trapz([1,2,3], x=[4,6,8])
]: 8.0
```

Trigonometric functions:

numpy.sin()

Trigonometric sine, element-wise.

```
: np.sin(np.pi/2.)
: 1.0
```

numpy.cos()

Cosine element-wise.

```
]: np.cos(np.array([0, np.pi/2, np.pi]))
]: array([ 1.000000e+00,  6.123234e-17, -1.000000e+00])
```

numpy.tan()

Compute tangent element-wise.

```
: from math import pi
  np.tan(np.array([-pi,pi/2,pi]))
: array([ 1.22464680e-16,  1.63312394e+16, -1.22464680e-16])
```

numpy.arcsin()

Inverse sine, element-wise.

```
: np.arcsin(1)     # pi/2
: 1.5707963267948966
```

numpy.arccos()

Trigonometric inverse cosine, element-wise.

```
np.arccos([1, -1])
```

```
array([0.        , 3.14159265])
```

numpy.arctan()

Trigonometric inverse tangent, element-wise.

```
np.arctan([0, 1])
```

```
array([0.        , 0.78539816])
```

numpy.hypot()

Given the "legs" of a right triangle, return its hypotenuse.

```
np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
```

```
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])
```

numpy.degrees()

Convert angles from radians to degrees.

```
rad = np.arange(12.)*np.pi/6
np.degrees(rad)
```

```
array([  0.,  30.,  60.,  90., 120., 150., 180., 210., 240., 270., 300.,
       330.])
```

numpy.radians()

Convert angles from degrees to radians.

```
deg = np.arange(12.) * 30.
np.radians(deg)
```

```
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265, 3.66519143, 4.1887902 , 4.71238898,
       5.23598776, 5.75958653])
```

numpy.unwrap()

Unwrap by changing deltas between values to 2*pi complement.

```
phase = np.linspace(0, np.pi, num=5)
phase[3:] += np.pi
phase
```

```
array([0.        , 0.78539816, 1.57079633, 5.49778714, 6.28318531])
```

```
np.unwrap(phase)
```

```
array([ 0.        ,  0.78539816,  1.57079633, -0.78539816,  0.        ])
```

Note:" Some **of the methods below are not necessary for our course, but it's always good to have knowledge**".]