Numpy

Notations:

[- -]: variable to input(compulsory).

{- -}: variable to input (not necessary).

#: annotation

@: example

ndarray object:

Attributes:

ndim:

#[int] the dimension of the array.

shape:

#[tuple] the dimensions. This is a tuple indicating the size of each dimension.

size:

#[int] the total number of elements in the array.

dtype:

#[string] the type of the elements.

itemsize:

#[int]the size in bytes of each element of the array.

flat:

#extract all elements in the array to form a 1-dimensional list following the axis order.

#often used in iterations

real:

#the real part of the array.

imag:

#the imaginary part of the array.

T:

#view of the transposed array.

@a.T

Methods:

1.sum(axis={-num-})

#return the sum of the elements.

#if the axis is specified, return the sum on the axis.

2.min(axis={-num-})

#return the minimum of the elements

#if the axis is specified, return the minimum along the axis.

3.max(axis={-num-})

#return the maximum of the elements.

4.argmin(axis={-num-})

#return the index of the minimum of the elements.

#by default, the index is that of the flattened array.

#if the axis is specified, the index is that along the axis.

5.argmax(axis={-num-})

#return the index of the minimum of the elements.

6.reshape([-dim1-],[-dim2-],...)

@a=b.reshape(3,4)

7.copy()

#return the deep copy of the array

@b=a.copy()

8.transpose({-num1-}, {-num2-}, ...)

#return the transposed array

#i in the j-th place in the parameters means that the array’s i-th axis becomes the transposed array’s j-th axis.

@a=b.transpose() #suppose b is a 2-dim array

@a=b.transpose(2, 3, 1) #suppose b is a 3-dim array

9.flatten()

#Return a copy of the array collapsed into one dimension.

@

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

b=a.flatten()

print(b)

>>array([1, 2 ,3 , 4])

10.swapaxes([-axis1-], [-axis2-])

#return a view of the array with axis1 and axis2 interchanged.

11.mean(axis={-num-})

#return the mean of the array elements.

#if the axis is specified, return the mean along the axis.

12.var(axis={-num-})

#return the variance of the array elements.

13.std(axis={-num-})

#return the standard deviation of the array elements.

Functions:

1.sum([-array-], axis={-num-})

#return the sum of the elements

#if the axis is specified, return the sum on the axis

@b=np.sum(a)

@b=np.sum(a, axis=0)

2.min([-array-], axis={-num-})

#return the minimum of the elements

3.max([-array-], axis={-num-})

#return the maximum of the elements

4.argmin([-array-], axis={-num-})

5.argmax([-array-], axis={-num-})

Operations:

1. Slice:

One-dimensional arrays can be sliced like lists.

@a[2:5]

Multidimensional arrays can be sliced with indices separated by commas.

@a[2:5, 1]

When fewer indices are provided than the number of axes, the missing indices are considered complete slices

2. Index:

The expression within brackets in b[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes; the dots (...) represent as many colons as needed to produce a complete indexing tuple.

@

#consider an array a with 5 axes:

a[2] is equivalent to a[2, :, :, :, :]

a[..., 3] is equivalent to a[:, :, :, :, 3]

a[1, ..., 3] is equivalent to a[1, :, :, :, 3]

Array creation:

1.np.array([-content list-],dtype={-typename-})

@a=np.array([1,2,3,4])

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

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

@a=np.array([1,2,3,4], dtype=’complex’)

2.np.zeros(([-dim1-],[-dim2-],...), dtype={-typename-})

#creates an array full of zeros

@a=np.zeros((2,3,4), dtype=int)

3.np.ones(([-dim1],[dim2],...), dtype={-typename-})

#creates an array full of ones

@a=np.ones((3,3), dtype=int16)

4.np.empty(([-dim1],[dim2],...), dtype={-typename-})

#creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64.

@a=np.empty((2,3,2), dtype=int32)

5.np.eye([-num-])

#return an identity matrix of size num

@

a=np.eye(3)

print(a)

>>

[[1,0,0],

[0,1,0],

[0,0,1]]

6.np.arange({-num1-},[-num2-],{-step-})

#the default value of num1 is 0.

#creates an array whose range is num1 to num2. num1 is the initial content but num2 can’t be reached. “step” denotes the difference between adjacent elements. The default step is 1.

#Note: “arange”, not “arrange”!

@a=np.arange(1,5)

>>array([1,2,3,4])

@a=np.arange(5,20,4)

>>array([5,9,13,17])

7.np.linspace([-num1-], [-num2-], {-num-})

#the default value of num is 50.

#creates an array whose range is num1 to num2. num1 is the initial content and num2 is the end content. Numbers in the array are evenly distributed. “num” denotes the quantity of elements in the array.

@print(np.linspace(0, 2, 5))

>>array([0.0, 0.5, 1.0, 1.5, 2.0])

Array printing:

print([-array name-])

@

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

print(a)

>>

[[1,2,3],

[2,3,4]]

Arithmetic operations:

1.[-array1 name-]+[-array2 name-]

#calculate the elementwise sum of two arrays.

2.[-array1 name-]-[-array2 name-]

#calculate the elementwise difference of two arrays.

3.[-array1 name-]\*[-array2 name-]

#calculate the elementwise product of two arrays.

4.[-array1 name-]@[-array2 name-]

#calculate the matrix product of two arrays.

5.[-array1 name-]+=[-array2 name-]/[-number-]

#update array1 elementwise.

@

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

a+=3

print(a)

>>

[[4,5,6],

[5,6,7]]

6.[-array1 name-]\*=[-array2 name-]/[-number-]

#update array1 elementwise.

7.[-array1 name-]\*\*[number-]

#calculate power elementwise.

8.np.ufunc(-[array name]-)

#ufunc is a notation which represent any function among these:

exp, log, log10, log2, sin, cos, tan, arcsin, arccos, arctan, floor, ceil, ...

#apply the function to every single element and output the new array.

Functions:

1.np.add([-array1-], [-array2-])

#a=np.add(b, c)

2.np.subtract([-array1-], [-array2-])

3.np.multiply([-array1-], [-array2-])

#elementwise product

4.np.kron([-array1-], [-array2-])

#compute the Kronecker product

5.np.linspace([-start number-], [-end number-], num={-num-}, dtype={-str-})

#return a numpy.array.

numpy.linalg:

1.np.linalg.matrix\_rank([-matrix-])

#return the rank of the matrix.

2.np.linalg.trace([-matrix-])

#return the sum along the diagonal of the array.

3.np.linalg.inv([-matrix-])

#return the inverse of the matrix.

4.np.linalg.det([-matrix-])

#return the determinant of the matrix.

5.np.linalg.eig([-matrix-])

#return a tuple. The first entry is a numpy array consisting of eigenvalues. The second entry is a numpy array consisting of corresponding normalized eigenvectors.

6.np.linalg.eigh([-matrix-])

#same as np.linalg.eig, but the input is a conjugate symmetric matrix.

7.np.linalg.solve([-A-], [-B-])

#solve a linear matrix equation AX=B, giving the exact solution. A should be of full rank.

8.np.linalg.lstsq([-A-], [-B-])

#return the least-squares solution to a linear matrix equation AX=B. X minimizes the Euclidean 2-norm of B-AX.

9.np.linalg.qr([-matrix-])

#factor the matrix as QR, where Q is orthonormal and R is upper-triangular.

#return a tuple where the first entry is Q and the second entry is R.

10.np.linalg.svd([-matrix-])

#factor the matrix as USVH, where U and V are unitary and S is diagonal.

#return a tuple where the first entry is Q, the second entry is S and the third entry is VH.

11.np.linalg.cholesky([-matrix-])

#factor the matrix as LLH, where L is lower-triangular, and return L.

#the input matrix must be Hermitian (symmetric if real-valued) and positive-definite.

numpy.random:

#use keyword argument dtype to specify the data type. e.g. dtype=’float32’.

1.np.random.rand({-\*dims-})

#generate random floats from [0.0, 1.0) in a given shape.

@

print(np.random.rand())

print(np.random.rand(2,2))

>>

0.08661911019377366

[[0.89481829 0.25576485]

[0.93256078 0.77398281]]

2.np.random.random(size={-tuple-})

#generate random floats from [0.0, 1.0) in a given shape.

#basically identical with np.random.rand(), except that the arguments in np.random.random() is a tuple rather than separated numbers.

3.np.random.random\_sample(size={-tuple-})

#alias for np.random.random

4.np.random.randint(low={-num-}, high=[-num-], size={-tuple-})

#generate random integers from [low, high).

#the default value of low argument is 0.

@print(np.random.randint(4, size=(2,3)))

>>[[3 0 3]

[3 2 3]]

5.np.random.[random\_integers](https://numpy.org/doc/stable/reference/random/generated/numpy.random.random_integers.html" \l "numpy.random.random_integers" \o "numpy.random.random_integers)(low={-num-}, high=[-num-], size={-tuple-})

#similar to [randint](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html" \l "numpy.random.randint" \o "numpy.random.randint), only for the closed interval [low, high], and 1 is the lowest value if high is omitted.

6.np.random.randn({-\*dims-})

#generate random floats from standard normal distribution in a given shape.

@print(np.random.randn(2,3)

>>[[ 1.561041 0.24618852 0.2857795 ]

[ 0.45012798 -0.97070411 0.4513093 ]]

7.np.random.chioce([-a-], size={-tuple-}, p={-array-})

#generates a random sample from a given 1-D array.

#a: if an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if it were np.arange(a)

#size: the shape of generated data.

#p: the probabilities associated with each entry in a. If not given, the sample assumes a uniform distribution over all entries in a.

@print(np.random.choice([1,2,3,4],size=(2,3)))

>>[[3 4 1]

[2 4 2]]

@print(np.random.choice([1,2,3,4],size=(2,3),p=[0.1,0.2,0.7,0.0]))

>>[[3 1 3]

[3 2 1]]

8.np.random.shuffle([-array-])

#return the shuffled array

7.np.random.binomial()

8.np.random.hypergeometric()

9.np.random.laplace

10.np.random.lognormal

11.np.random.multinomial

12.np.random.multivariate\_normal

13.np.random.normal

14.np.random.standard\_normal