Useful function for numpy:

<https://numpy.org/doc/stable/reference/index.html>

Array creation:

* Unlike “tensor” in “pytorch”, numpy array allows string values to be stored, although some functionalities might not be used;

Np.array([input array])

Np.zeros((row, col))

Np.eye(N=Rows of matrix, M=N(if given, number of columns), k=diagonal shift\_>0 refers to upper diagonal)

Np.full((row, col), which value to fill in)

From file

Np.fromfile(file\_source, count=how many elements to take, sep=use what to separate each element)

By interval

Np.arange(stop), np.arange(start, stop), np.arange(start, stop, interval) -> lst[:stop], lst[start:stop], lst[start:stop:interval]

Np.linspace(start, stop, num=“how many evenly splitted interval needed”)

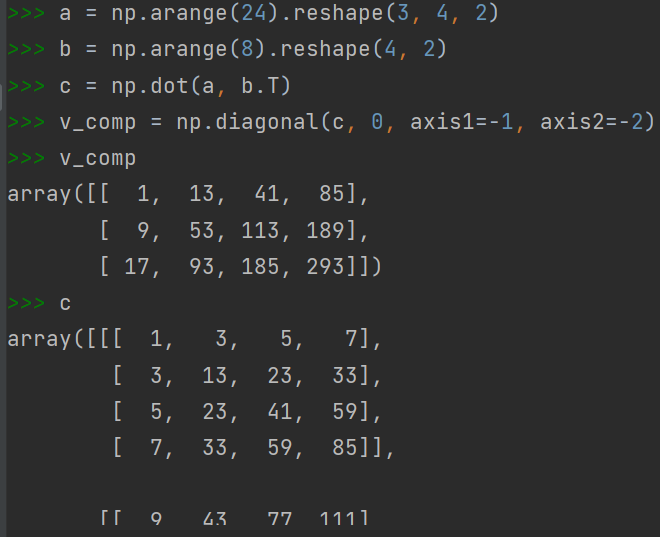
Np.logspace(start, stop, num=“how many evenly splitted LOG scale space is needed”, endpoint=False“if end points not included”) (1, 3, 3)

only base 10 log ensures 3 evenly separated space -> outputs 10, 100, 1000 -> log\_10(10), log\_10(100), …

Np.geomspace(start, stop, num=“how many elements to display, by exponential power”, endpoint=) (1, 32, 6) only exponents of 2 allows 6 many evenly separated space between 1 and 32

Outputs 0, 1, 2, 3, 4, 5 -> 2^0=1, 2^5=32…

Np.diag(1-D-array), Np.diag(2-D-array); construct diagonal matrix using input 1D matrix, extract diagonal elements from 2D input matrix.

Np.diagonal(ndarray, [diag\_offset, axis1, axis2]) -> 2darray: handles higher dimensional array’s diagonal element extraction;

Diag\_offset: the displacement of diagonal line on matrix to extract

Axis1: the first axis for the the matrix contruction to extract diagonals

Axis2: the second axis for the matrix contruction to extract diagonals

Realizing the acquired matrix by axis selection need not to be diagonal;

Useful for extracting from dot product of multiple vectors, see image code right, which extracts perpendicular vector component using diagonal elements

Array indexing

Basic indexing:

X[start:end:interval], X[:, start:end:interval, start:end:interval, …], X[…, start:end:interval, :, :],

X[…, :, :, :], …

First is for 1D indexing, second is for multi-dimensional indexing, with only 2nd and 3rd axes’s range being restricted; third is the indexing of everything except last 3rd axis is restricted; fourth is the index of every elements.

Advanced indexing:

X[[row1, row2, …row\_n], [col1, col2, …col\_n]] -> [(row1, col1), (row2, col2), ….(row\_n, col\_n)].

X[ [[rowi, rowj, rowk],

[rowl, rowm, rown]],

[[coli, colj, colk],

[coll, colm, coln]] ] ->

[ [(rowi, coli), (rowj, colj), (rowk, colk)],

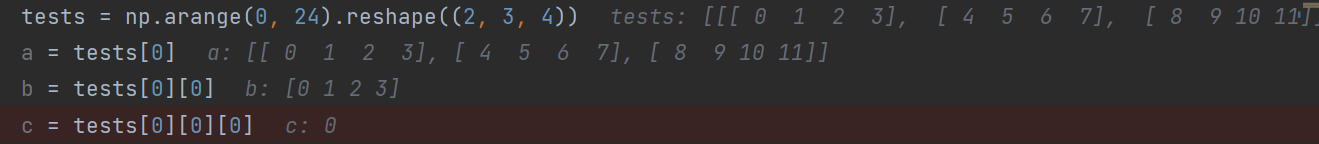
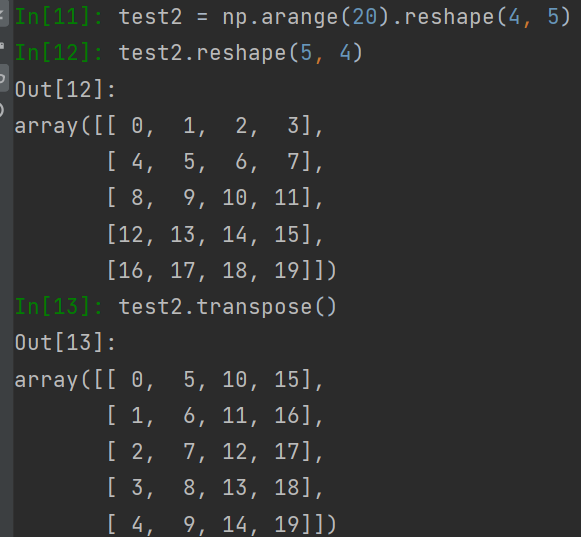
[(rowl, coll), (rowm, colm), (rown, coln)] ]

Note the consistency of dimensionality for row and column indexing (red brackets), and consistency with final output’s dimension (red brackets)

Array operations

Shape and axis

Shape: (a, b, c, d, …) the order of shape indicates for the ndarray, “a” indicates outer-most dimension



Np.reshape(nparray, (new shape)) both arrays should have same amount of elements, or an error would occur

Realizing that reshaping is different from transpose(np.transpose())

Ndarray.flatten(optional[order]) collapse a 2D array into 1D (refer to 28\*28 images convert to 784 pixels)

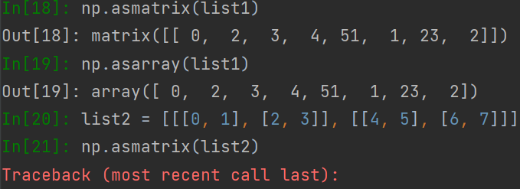
Order: ‘C’-row by row append, ‘F’-col by col append, (‘A’, ‘K’ not used frequently)

Ndarray.T transpose of array, same to linear algebra

Np.swapaxis(ndarray, axis1, axis2) -> a 3D array would have axis (0, 1, 2). Realizing this function works for axis > 2

Np.expand\_dims()

Convert to array

Np.asarray(array)

Np.asmatrix(array)

* Difference with “asarray”: “asmatrix”’s result will strictly converted to 2D array, and any input with “dimension > 2” leads to error.

Np.asscalar(array with size 1)

Np.asfarray(array) -> convert array to np array of float values

Np.diagonal(): see reference in previous “array creation section”

Array combination and splitting

Np.concatenate((ndarray1, ndarray2, …), axis=None)

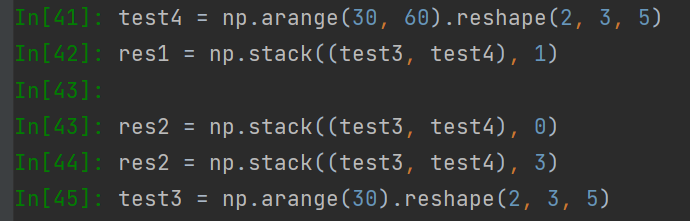
“axis=none” gives flattened combination of all input arrays;

“axis=0” gives concatenation at outer-most dimension [concat, …]; “axis=-1” gives concatenation at inner-most dimension […, concat];

All ndarrays must have same “side length” at ALL non-concatenational axes;

Np.stack(Union[ndarray, (ndarray1, ndarray2)], axis=) -> input 1: rearrange all ndarray[i]’s element along “axis”; input2: concatenate two ndarrays along “axis”; two arrays must have same “side length”

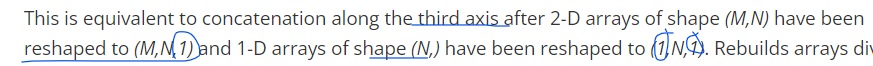
Input1 elaboration: suppose “axis=n”; then is equivalent to “ndarray.swapaxes(0, n)”

****Input2 elaboration: first combine all elements together to form a new dimension, then treat the new array as “input1” and perform same analysis as above. The input axis can be “1 extra” to compensate the newly added dimension.

**Warning: this method always pile up data to form a new dimension;**

**Requiring all inputs to have exact same shape, and “axis” only affects at last two dim;**

**Above warning doesn’t apply to hstack, vstack or dstack**

* Note: for 3D cases, could be convenient to use: np.vstack((nparray1, nparray2)) to replace axis=0, np.hstack((nparray1, nparray2)) to replace axis=1, and np.dstack((nparray1, nparray2)) to combine two arrays, by creating a third dimension. 

Np.split(ndarray, Union[int, array[int]], axis=”0 by default”) -> if “int” is given, will split the array evenly by “int many”, but requires “divisibility (a|b)”; if array[int], then will split the array by “index in array[int]”(index exceeding len(ndarray) result in subsequent empty ndarray).

Np.array\_split(ndarray, Union[int, array[int]], axis=0 for default) -> same as np.split(ndarray, int, axis=) “except” ‘divisibility’ is no longer required.

Adding and modifying array elements

Np.delete(ndarray, obj, axis=None) -> obj: a int or an array of int, as an index. axis: when “None”, delete “flattened” array’s elements (returns “flattened” array)

If index is out of bound at given axis, will raise error!!!

Reference of “MASK: boolean ndarray; (np[mask])”

Np.insert(ndarray, obj, value, axis=None) -> returns a new array, input doesn’t change

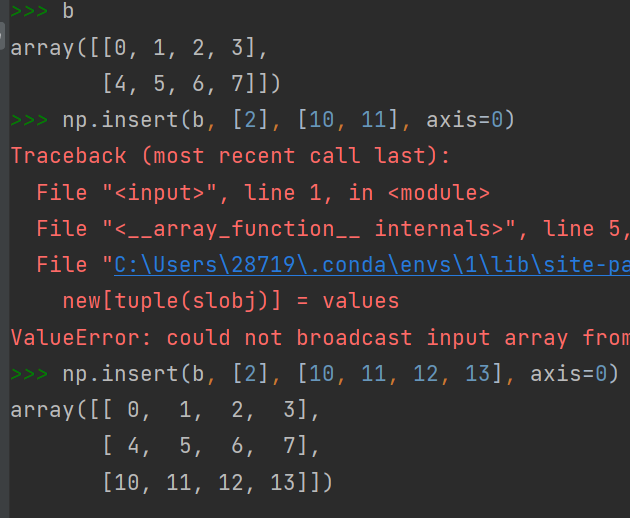
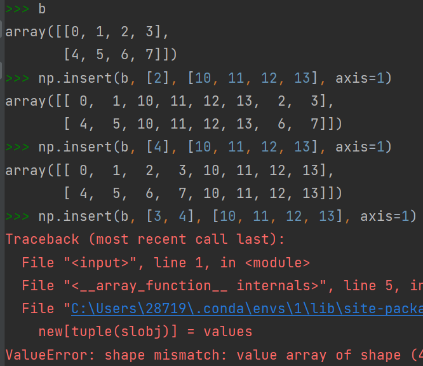
obj: index to insert (new element will have the index after insertion), a scalar or int array. Obj cannot exceed number of elements along an axis (if this number is given, same as append along that axis)

Value: if a scalar, will insert a row/col/one value depending on shape of ndarray; if an array list, will insert according to “axis”(requiring shape-consistency).

Axis=None will first “flatten” ndarray, and return “flattened” one.

(insert considers type-casting: float->int, boolean -> int for ndarray[int])

Realizing: “value” will be inserted into every element along that axis, and shape consistency will be preserved by adding same row/columns/…, require boardcastable!

Also realizing if “obj” has only one element, all “value” will be added at that same position, with 1st element take the “obj” index. However if “obj” is not size “1” and shape is inconsistent with “value”’s shape, will result in error. The dimension of “obj” and “value” should both be “1”.

Np.append(ndarray, value, axis=None) -> must maintain shape-consistency before appending,

Np.unique(ndarray, return\_index=False, return\_inverse=False, return\_counts=False, axis=None)

(return an array of unique elements in SORTED ORDER, and FLATTENED)

* Return\_index: if True will ALSO return each unique element’s first apparence in FLATTENED 1d-array, in the form of an ndarray.
* Return\_inverse: if Ture, will ALSO return index array having same shape as input ndarray, each element is the corresponding index of element in returned unique list. -> unique\_array[index] will give original input “ndarray”
* Return\_counts: if true, will return an array “count” where “unique\_array[i]” appeared in “ndarray” for “count[i]” many times.
* Axis: refer to previous discussions

Np.flip(ndarray, axis=None) (revert elements according to axis)

* Axis: if None ,will revert elements in flattened array (but returned array has same shape as input)

If is an int, just revert by the indicated axis.

If a tuple of int, will revert by all those axes.

Np.flipud(ndarray) -> up and down -> same as np.flip(ndarray, axis=0)

Np.fliplr(ndarray) -> left and right -> same as np.flip(ndarray, axis=1)

Both flipud and fliplr also works on higher dimensional arrays, with “axis” shown as above(only applicable to first two axes…)

Np.reshape(ndarray, (new\_shape[0], new\_shape[1]) , order=’C’ by default)

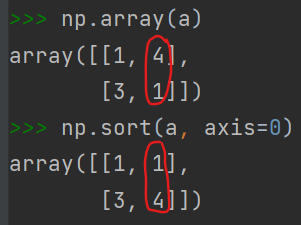
Requiring new shape has the “SAME” number of elements as input;

First flatten ndarray, then re-construct according to given new shape -> not equal to “transpose”

* (int, int) as new shape. First int is num of rows
* Order: C: filling inner-layers first; ([[a]] [[a, b]]); F: filling outer-layers first [[a]] [[a], [b]]

Sorting, searching

Numpy.sort(ndarray, axis=-1, kind=None, order=None):

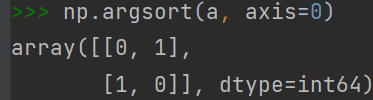
 Return a sorted copyof “ndarray”

Kind: {‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’}

If axis is not last axis: see the example shown right, which treats columns as vectors and sort each column vector.

Sorting along last axis doesn’t consume extra space when sorting;

(might be useful if swapaxis is used before sorting for space saving?)



Numpy.argsort(ndarray, axis=-1, kind=None, order=None):

Return INDICES that would sort “ndarray”.

Ndarray.sort(axis=-1, kind=None, order=None):

IN-PLACE sorting of “ndarray”

Numpy.partition(ndarray, kth, kind=’introselect’):

Return a new array with partition as follows:

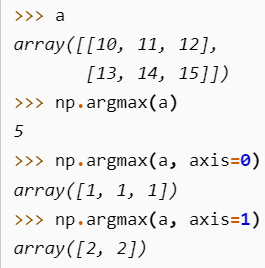
The element at index “kth” of output array is the element in “ndarray[kth]” after sorting

Any elements at “index” < “kth” are all less than “ndarray[kth]”;

Same as “splitting” in merge sort

‘introselect’: the algorithm for performing partition;

Basically it performs similar technique as quicksort/heapsort depending on efficiency of which algorithm is faster.

Numpy.(nan)argmax(ndarray, axis=None, out=None, )

Return the INDICES of maximum values along an “axis” for “ndarray”.

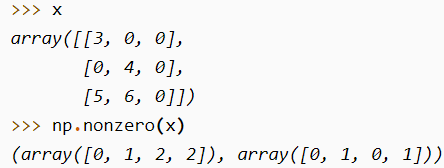
If “axis” is none, will flatten “ndarray” and return the flattened max’s indice

Out: if given, the output of this method will be filled into “out” ndarray.

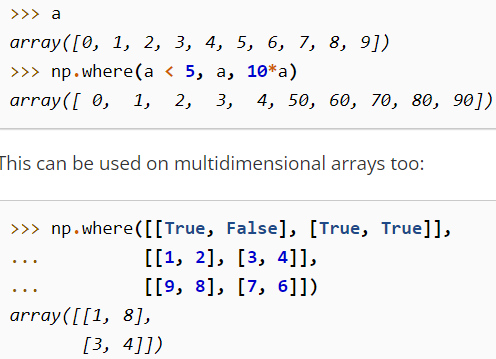
It must have the appropriate shape to contain the result.

“nan” is ignored if “(nan)” is included, “ValueError” is raised if “ndarray” is all-nan;

Numpy.(nan)argmin(ndarray, axis=None, out=None, )

Numpy.nonzero(ndarray):

Return all indices, following the form: array(dim0\_indices, dim1\_indices, ….) where the indexed elements are non-zero.

Numpy.where(condition, x, y):

Returns a new ndarray, where when “condition”’s indices is true, corresponding output indice will be “x”’s corresponding element. Otherwise the output’s corresponding indice will be “y”’s corresopnding element.

Image shown right shows what can condition be:

boolean expression or a True/False array(useful for masking).

np.clip()

Datatype operations

<https://numpy.org/doc/stable/user/basics.types.html> numpy’s data types

Np.can\_cast(source, destination) ->

* Both source and destination can be: type (np.float, float) and attributes or mix of both: -> (np.float, float), (1000, boolean)…

Np.emath package:

Np.sqrt(x), np.log(x), np.log2(x), np.logn(n, x), np.log10(x), np.power(x, p), np.arccos(x), np.arcsin(x), np.arctanh(x)

Functional programming: Apply a function to all elements of an array

Np.apply\_along\_axis(func\_name, axis:int, ndarray, <args>) -> returns a new array

“func” must accept 1D array; once this method is called, it will extract ndarray along axis as 1D array and transfer to “func”.

If “func” returns a higher dimensional array, then the position of 1D array in ndarray will be replaced by the higher dimensional array.

Args: more arguments for func\_name

Similar function in “pandas”

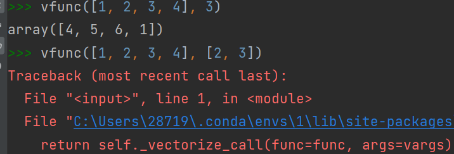
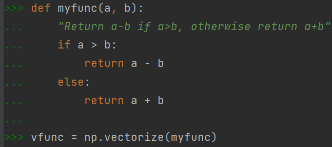
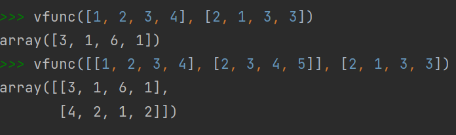
Np.apply\_over\_axes(func\_name, ndarray, axis: array-like) -> returns a new array

“func” must accept 1D array as the ONLY input; will implement operations on axes one by one in “axis” array.

Np.vectorize(function) -> return a function which could be applied on vectors;

“function” can only be applied on SCALARS!!! When need to use the vectorized function, simply call it and give ndarray or scalars as input.

* + - if “function” allows multiple inputs, then each input is either a scalar, or ndarray ALL with BOARDCASTABLE shape

 Realizing the function is iterated using for loop, thus not enhancing performance;

Np.piecewise(ndarray, [boolean-condition1, boolean-condition2, …], [func1, func2,…]) ->

return element has same dimension as ndarray

Boolean-condition must be based on “input-variable”, and usually use inequalities;

(useful for creating activation/piecewise function)

Advanced indexing using indices and masks

Indexing:

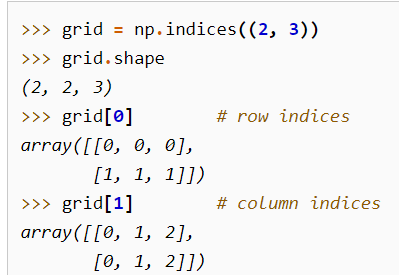
Np.where(condition, array1, array2) -> return a new array

“Condition” is a statement placable after “if”; applied on “array1”

If condition on array1’s current element returns true, take an element from array1;

If condition is false on array1’s current element, take array2’s corresponding element

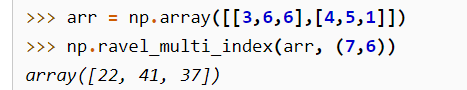
Array1 and array2 must have same shape for 1-1 correspondence.



Np.indices((shape1, shape2, shape3, …shapen)) -> return a ndarray

Indices will return a (n, shape1, shape2, …shapen)’s ndarray, where “n” indicates the axis of ndarray, and for each shape1, ….shapen, each element represents index along shape1’s axis, …shapen’s axis.

Can be used for indexing a real array.

Np.ravel\_multi\_index(ndarray, (shape)) -> return the represented index of flattened (shape) array

this method treats ndarray as an “index array”, as described in “ARRAY-INDEXING” part and returns corresponding flattened (shape) array’s index.

For the right image’s example: 22 = 3x6+4, [3,4]; 41=6x6+5, [6,5], …

One potential constraint is, the indexing array’s index cannot exceed (shape)’s axis length; e.g, [8, 6] fails because 8>7, where 7 is first axis’s length.

Np.unravel\_index(1Darray, (shape)) -> return a ndarray of indices on (shape) array.

This method does exactly the opposite of np.ravel\_multi\_index

Np.mask\_indices(int, mask\_function, k) -> returns indices to access values NOT being masked;

“int” means the generated boolean array will have shape (int, int)

Mask\_function: func(input\_array, k) returns a boolean array.

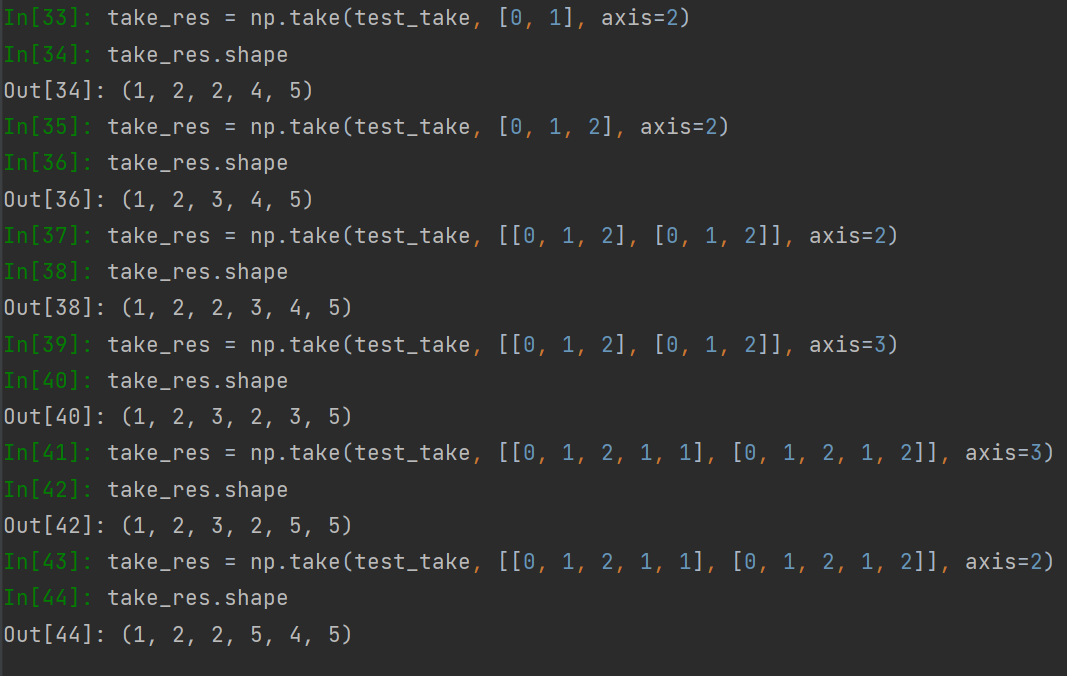
K: argument for mask\_function.

Used for generating masked-indexes for array creation.

Better only apply numpy functions as “mask\_function”; require further detail on which functions can be used, as “tril, triu” which doesn’t generate boolean also works

Np.diag\_indicies(int, optional[ndim]) -> return ndim many arrays with shape (1, int)

Used for indexing the diagonal elements of an array having (ndim, int, int) shape.



Extracting elements according to advanced indexing

Np.take(ndarray, indices, axis=None) -> return a new ndarray being indexed by given “indices”

Equivalent to ndarray[:, :, …:, indices, :, :…] where “indices” locates at “axis” position

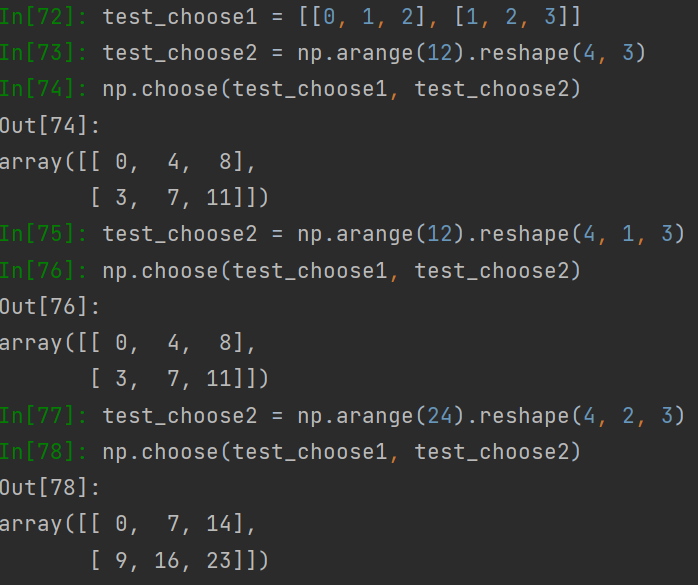
When axis=None, indices will be flattened before using

Once a indices (with int element) is given, each element of that indice would represent a

ndarray[:, :, …:, indice\_element] ndarray. This array will fill that element’s position in

indices array.

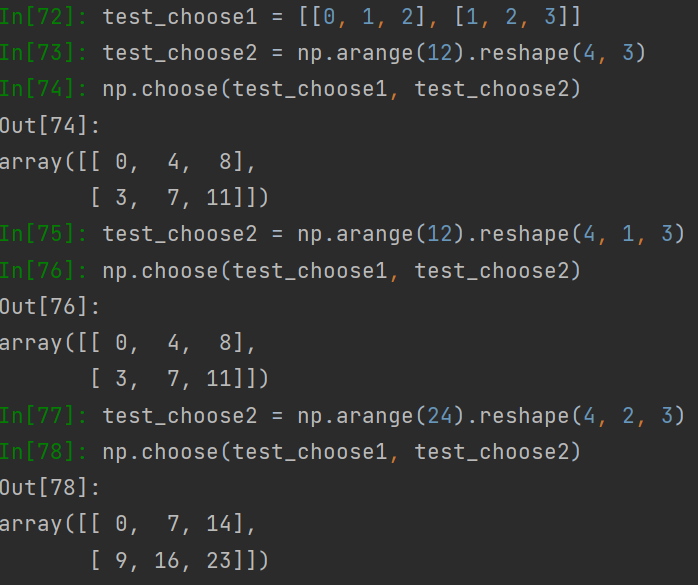
only works on given axis; indexed elements (possibly arrays, as in demo right) will be re-arranged according to indices to form new arrays, and become elements of super-dimensions(the change of axis and 2D indices, on shape; [2, 3] is the shape of indices, indicating indexed elements are being newly generated).

np.choose(array1, array2) -> return a newly created array

“array1” needs to have same length as outer-dimension of “array2”.

Output will have same shape as “array1”

For any array at LAST dimension of “array1”: the value cannot exceed how many choice arrays are there in “array2”;

when “array2” has 24 elements, “array1[0]” by default takes element from “array2[,0,]” , and “array[1]” by default takes elements from “array2[,1,]”;

one heuristic: the dimension of “array1” must be boardcastable with LAST few dimensions as “array2”, where “array1”’s elements must not exceed “array2.shape[0]”.

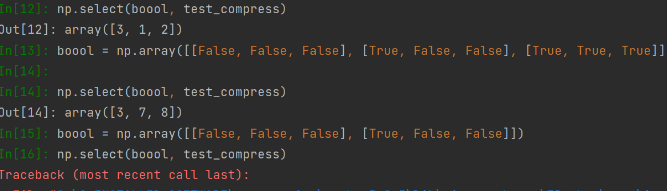
Np.select(condlist, choicelist, default=0) -> return a new array extracted according to condlist

Condlist is an boolean array of conditions, each applied on elements of choicelist.

Choicelist is a list of ndarrays. The ordering of choicelist’s elements affects ordering of

evaluation.

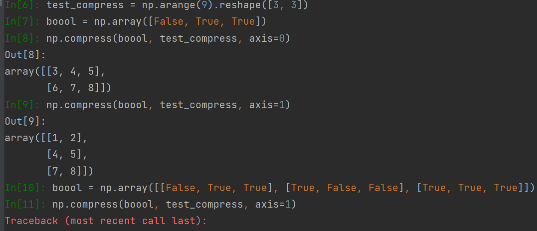
Default is the value to fill if no elements at given index from any choice-ndarray satisfies

condlist.

In practise, if 1st ndarray in choicelist doesn’t satisfy any condition in condlist, will move

on to 2nd, …and so on. However if one “True” is met, will take the true -> handles multiple “true conditions”

Requirements on shape boardcasting is similar to that of “np.choose”; require “condlist.shape” boardcastable with “choicelist”’s last few dimensions

Np.compress(condition, ndarray, axis=None) -> return a new array

Condition: MUST BE 1D!!! a boolean “1darray”; if condition has less length than ndarray.shape[axis], will only output “len(condition)” dimensional ndarray along “axis”.

Axis: the axis where condition would be applied to extract elements out;

Axis=None: will flatten ndarray first

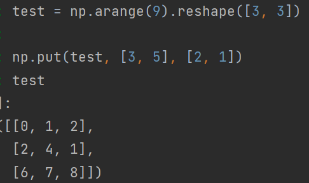
Adding elements according to advanced indexing

Np.place(ndarray, mask, vals) -> return None

Replaces ndarray’s elements according to mask-conditions;

Mask must be a boolean expression applied on “ndarray”, SAME size as “ndarray”

Vals will be FLATTENED, and replaced consecutively and cyclic -> 1st occurrence of “True” on mask: replace with val[0], …. And a cycle if all elements in “vals” is done replacing in ndarray for a cycle

np.put(ndarray, indices, value-to-replace) -> return None

**warning: indices are for flattend array only!!!**

replaces ndarray’s element into “value-to-replace” based on indices

indices can be indexing array or boolean array

value-to-replace will be done consecutively and cyclic, and flattened

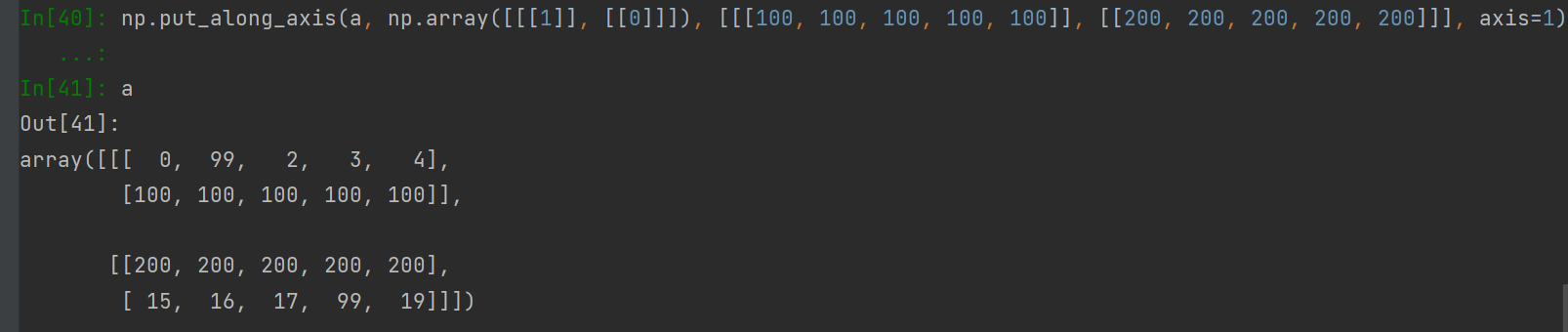
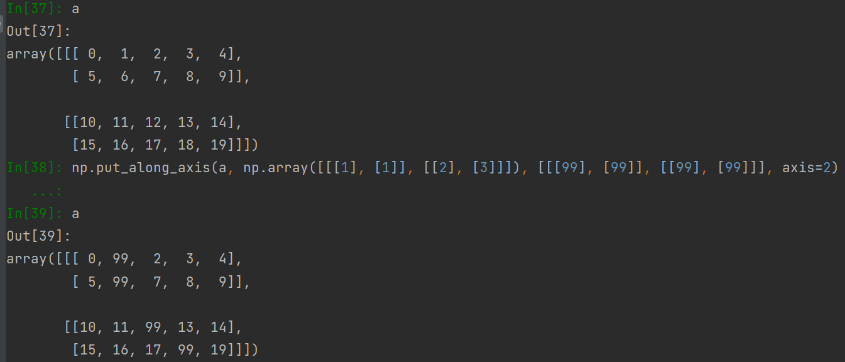
Np.put\_along\_axis(ndarray, indices, values, axis) -> return None

Indices: the “indices” of sub-ndarray along arrays, must have same depth as “ndarray”.

Must also be a type “ndarray”.

Axis: indicates interpretation of indices; e.g: indices in below example are applied to the last axis & second-last respectively

-> each indice’s value must less than length of that dimension;

 Value: can be a scalar or ndarray value, requires shape to be boardcastable.

**Indices must have the same dimension as ndarray, and value must have same dimension as the “ndarray” about to be replaced, depending on axis, and how many to be replaced.**

Np.putmask(ndarray, mask, values) -> return None

Mask: boolean array, must have same shape as ndarray

A value will be replaced only when element-mask is “True”

Values: can be a scalar or array-like(will be flattened); values will be replaced consecutively and cyclic.

Linear algebra: np.linalg from numpy import linalg as lin

Multiplications

np.dot(a, b) -> return dot product of two arrays as a new array

“a” and “b” must satisfy shape restriction for matrix/vector multiplication.

Only multiply last two axes

Using “@” (a @ b) is preferred for “Matrix multiplication”

(supports scalar multiplication as well)

Lin.multi\_dot(arrays) -> return matrix-multiplication of multiple vectors in “arrays”

Arrays: a sequence of vectors/matrix for multiplication

All arrays must be 2D except 1st and last, treated as row-vec and col-vec respectively

np.vdot(a, b) -> dot product of two vectors

Doesn’t work on matrix-vector multiplication

If “a, b” have same shape but not 1D: will flatten both “a, b” to vectors!!!

np.inner(a, b) -> inner product of two arrays (a1\*b1+a2\*b2+…)

np.outer(a, b) -> outer product of two arrays (two vectors -> one matrix)

Useful for creating COVARIANCE MATRIX, where two inputs are variance of each X^{->}.

Output shape: [0]: a.shape; [1]: b.shape

For higher dimensional “a, b”: will flatten to 1D vector before operations;

np.matmul(a, b) -> matrix product of two arrays

Require a, b satisfy shape constraints, and will perform matrix multiplication.

Equivalent to “a @ b”

Lin.matrix\_power(a, n) -> return the result of matrix multiplication of “n” many “a”.

“a” is required to be a square matrix

If “n” is 0 will return identity

Np.multiply(a, b) -> elementwise multiplication of two matrix/vector/…

Requires “a, b” to have same shape

Special\_values

Lin.eig(a) -> return (array1, array2)

Suppose “a” has shape (n, n)

Array1 is the array of eigenvalues, with dimension (…, n)

Array 2 is the array of eigenvectors with dimension (n, n)

Array2[:i] corresponds to array1[i]

Lin.norm(a) -> return norm of a matrix / vector

Lin.det(a) -> return square matrix “a”’s determinant

Lin.matrix\_rank(M) -> return rank of matrix M

Lin.trace(a) -> return the sum along diagonal of array

Linear solver:

Lin.solve(a, b) -> solve equation ax=b and return “x”,

“a, b” are variable matrix and vector, respectively

Lin.lstsq(a, b) -> find a best fit line for y=mx+c;

Given a set of points{(**x**, y)}, generate “a” as follows: a = [**x** 1] (last col contains only 1)

And solves equation “y = ap” where p is: [[m], [c]] by letting “a” = x, “b” = y.

Returns “p” containing “m” and “c” for best fit line

Lin.inv(a) -> compute inverse of matrix “a”

Must be a square matrix for last two dimensions!!!

Matrix operations: np.matlib

Np.asmatrix(ndarray): interpret input as a matrix;

Realizing the shape has to be 2D (or an error would occur)

**From numpy import matlib**

matlib.zeros(shape): returns a new matrix with given shape;

shape: a sequence with length AT MOST 2!!!

Matlib.eye(n, M=None, k):

n: number of rows in output matrix;

M: number of columns in output matrix;

K: index at diagonals (refer to “diag” in some pytorch methods);

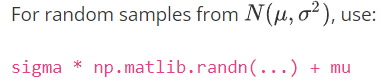
Matlib.rand(shape\_tuple)

Shape\_tuple: a size 2 tuple;

Matlib.randn(shape\_tuple): generated matrix have values sampled from a normal distribution

with mean 0 and std 1;

random normal distribution can be acquired by:

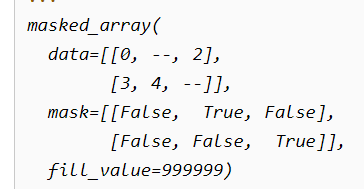


Useful for reparameterization trick?

Matlib.empty(shape), matlib.ones(shape), matlib.identity(n: square matrix size),

Masked array package import numpy.ma as ma

Most operations which can be performed on a normal array can also be performed here, ignoring masked value whenever possible.



Example on the left shows properties of a masked array OBJECT.

Elements with “--” in “data” is where mask is applied (“true” in mask).

“fill\_value” indicates what value to give when convert back to original array.

Creation and examination

Ma.MaskedArray(data, mask=boolean\_ndarray) -> returns a masked array with all properties

“data” and “boolean\_ndarray” must have same shape!!!

Ma.masked\_all(shape) -> return a masked array with shape “shape” where all elements are masked.

Shape is a tuple of integers.

ma\_ndarray.all() -> returns true if all elements in “ma\_ndarray” are masked.

.any() -> if any element in “ma.ndarray” is masked.

ma.count\_masked(ma\_ndarray, axis=None) -> count how many elements are masked.

If axis=None, the count flattened version and return a scalar.

If not, will return a “ndarray” where each element indicates how many “ma\_ndarray”’s elements are masked along that axis.

mask out values

Ma.mask\_cols/rows/rowcols(ma\_ndarray) -> return a modified ma\_ndarray as follows

**ma\_ndarray is 2D**

If there is a mask at a column/row/location, then all entries of that

column/row/row&col will be applied with a mask

ma.masked\_invalid(ndarray) -> mask a location of “ndarray” whenever “NAN or INFs” occurs

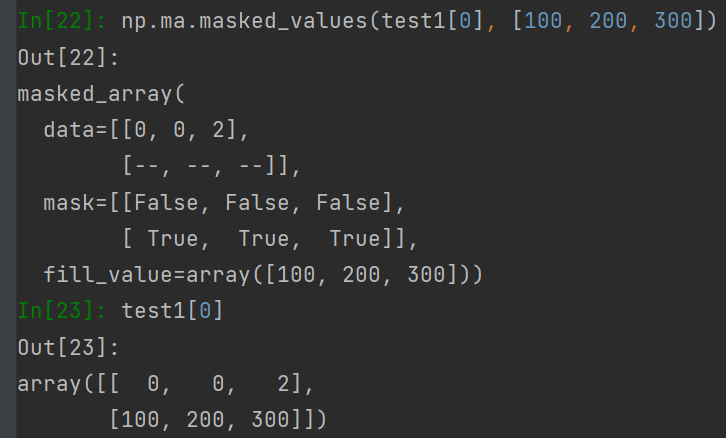
ma.masked\_equal/greater/greater\_equal/less/less\_equal/not\_equal(ndarray, value) -> return a ma\_ndarray according to follows:

A mask is applied to a value in “ndarray” whenever the value is “equal/greater/greater\_

equal/less/less\_equal/not\_equal” the “value”.

Ma.masked\_value(ndarray, value, copy=True):

mask entries from “ndarray” where the entry is equal to “value”.

Value: can be either a scalar, or an “ndarray” with shape BOARDCASTABLE with “last few dimensions” -> if matches, the ARRAY will be masked, instead of values occurred;

Copy: whether to return a new copy of masked array(same as inplace)

Ma.masked\_inside/outside(ndarray, v1, v2) -> return a ma\_ndarray as:

If the current element in “ndarray” is inside/outside interval “[v1, v2]”.

Ma.masked\_where(condition, ndarray) -> return a ma\_ndarray as:

If the current element in “ndarray” meets with all conditions in “condition”.

“condition”: a 1D array, with expressions producing true/false. Applied on “ndarray”.

Other similar functions to ndarray operation: all functions applied on ma\_ndarray, with masks combined, masked\_elements ignored in math calculations.

Concatenate, stack, vstack, hstack, dstack, append;

Mathematical operations:

**All operations require adding “np.” at front;**

Np.sin(ndarray), cos(ndarray), tan(…), arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh

Degree(x: radian)<->radian(x: degree)

Floor(numeric), ceil(numeric),

Sum(ndarray, axis, keepdims, ), prod(ndarray, axis, keepdims)

Nansum(ndarray, axis, keepdims), nanprod(ndarray, axis, keepdims)

Nan values are treated by “0” in “sum” and “1” in “product”

Cumprod(ndarray, axis), cumsum(ndarray, axis):

Each entry of returned array is the sum of all previous elements along a given axis

Nancumsum(ndarray, axis), nancumprod(ndarray, axis)

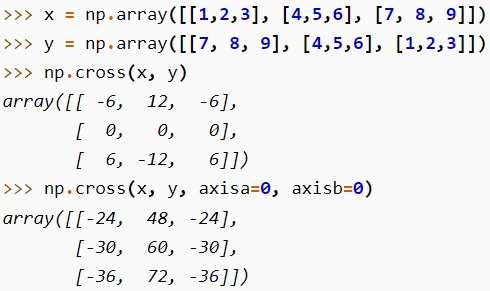
Diff(ndarray, n=1, axis=-1, [prepend, append]):

For each indices “i” (1D as an example), calculate the value “ndarray[i] – ndarray[i-n]”, along given “axis”;

Prepend/append: values to append to “ndarray” before performing operations;

Realizing if “n=1”, there are at most “n-1” many calculations available;

Thus require prepend/append to fill in the value.

Np.cross(ndarraya, ndarrayb, axisa=-1, axisb=-1, axisc=-1, axis=None)

Axisa: determines which dimension of ndarraya should be treated as vectors;

Axisc: determines which dimension of the output ndarray represents cross product result.

Axis: if defined, will overwrite axisa, axisb and axisc (replacing all 3 values with “axis”)

Image on left gives better understanding of axis;

Exp, log, log10, log2, exp2;

Arithmetic operations:

Add(ndarray1, ndarray2): adding elements ELEMENTWISE;

Requiring ndarray1 and ndarray2’s shape to be boardcastable.

Reciprocal(ndarray): returns reciprocal of elements in “ndarray”, elementwise. 1/x

Multiply(ndarray1, ndarray2): perform multiplication operations elementwise;

Ndarray1 and ndarray2’s shape must be boardcastable;

Divide(ndarray1, ndarray2)

Power(base\_ndarray, power\_ndarray):

Raise elements from “base\_ndarray” by corresponsing power from “power\_ndarray”;

Require shape-boardcastble

Nanmax(ndarray, axis, keepdim), nanmin(ndarray, axis, keepdim):

Take max or min while IGNORING “nan”

Sqrt, absolute, sign,

Nan\_to\_num(ndarray, copy=True, nan=0.0, posinf=None, neginf=None):

Replace all “NAN” value in “ndarray” to “nan” in parameter,

Positive infinity to “posinf”, negative infinity to “neginf”;

All three parameters should be a finite number.

**From numpy.polynomial import Polynomial**

Polynomial(coef=[a, b, c, …. z], domain=None, window=None):

Create a 1D polynomial “Y = z x26 + y x25 + … + c x2 + b x + a”;

Window: the window to use; given [domain[0], domain[1]], the region is shifted and scaled to region [window[0], window[1]].

Cov, cumsum, mean, median, power, prod(product), std, sum, var, argmax, argmin, max, min, sort, diag,

dot, identity, inner, outer, trace, transpose.

Random:

Np.random.seed(int);

Np.random.Generator.integers(low, high=None, size=None, endpoint=False)

Generate integers following a uniform distribution;

if “high” is not given, digits will be generated from zero to “low”;

size: tuple of output ndarray shape; will generate enough integers to fill the ndarray;

if not given, will generate a single integer;

endpoint: if True, “high” will be possible to generate;

np.random.Generator.random(size=None, )

generate float points in the interval [0.0, 1.0), following a uniform distribution;

can use shift-and-scale to get any desired interval;

Np.random.Generator.shuffle(ndarray, axis=0)

Np.random.Generator.permutation(x, axis=0)

**Distribution generators:**

Np.random.Generator.binomial(n, p, size=None)

n: # of trials, the key parameter for binomial distribution;

p: the probability, another parameter for binomial distribution;

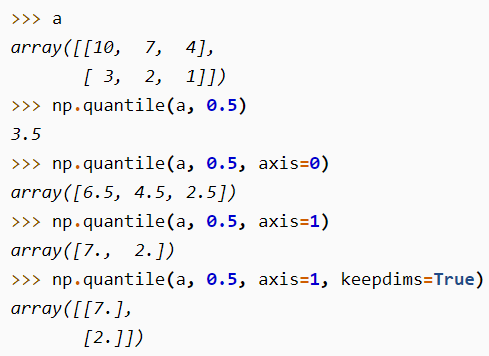
other supported distributions: (for specific usage see API reference)

np.random.Generator.chisquare/exponential/gamma/geometric/logistic/multinomial/normal/poisson/uniform

Statistics:

Percentile & quantile

Numpy.(nan)percentile(ndarray, q-array, axis=None, out=None, overwrite\_input=False, method=’linear’, keepdims=False, )

Percentile: returns the observations/samples/values… that, other observations /values/… below the returned one, altogether can be ovbserved for a probability of “p%”

q-array: the ndarray containing percentiles; (integers between 0 and 100)

if “axis” is not given, will flatten “ndarray” and perform operations

out: alternative output array for filling the result.

The output, regardless of given one or returned one, should match with the dimension of “q-array”.

Overwrite\_input: allowing “ndarray” to be modified during calculation, for saving space(but “ndarray” would no longer be used; )

Nan: if “nan” is prefix, will ignore “nan” values in array.

Numpy.(nan)quantile(ndarray, q-array, axis=None, out=None, overwrite\_input=False, method=’linear’, keepdims=False,)

Quantile: squivalent to percentile, but with “q-array” values between 0 and 1;

Statistical evaluations:

Numpy.median(ndarray, axis=None out=None, overwrite\_input=False, keepdims=False)

Numpy.average(ndarray, axis=None, weights=None, )

Weights: for calculating weighted average. Can be 1D array or with same shape as “ndarray”. Realizing the 1d shape must match with “ndarray AND axis”.

Numpy.mean(ndarray, axis=None, out=None, keepdims=bool[optional], where=bool[optional])

Calculate equally-weighted average of “ndarray”

Numpy.std(ndarray, axis=None, out=None, ddof=0, keepdims=bool[optional], where=bool[optional])

Ddof: when calculating standard deviation, the denominator has the form “N-ddof”, where “N” is the sample amount. One example of using “ddof” is 1, “N-1”;

Numpy.cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None,)

M: 1D or 2D array, when “rowvar” is true, will treat each row as a single variable, and each element in the row as observation. (if False, then opposite.)

Bias: if “True”, the denominator of calculating std is “N”, instead of “N-1” when bias=F

Fweights: frequency weights 1D vector; (currently not commonly used)

Aweights: 1D observation vector weights; (currently not commonly used)

Numpy.histogram(ndarray, bins=10, range=None, )

“ndarray” will be flattened for computing histogram

Bins: the number of bins for the histogram graph;

If input is a sequence instead of a scalar:

[0, 1, 4, 5]: the second bar will have width “3”;

Range: if not provided, histogram will plot for data from “ndarray.min()” to “ndarray.max()”; if provided: (float, float)