NumPy Notes

NumPy Illustrated: The Visual Guide to NumPy

1. Vectors, the 1D Arrays

Functions used to initialize Numpy array

function	parameter
`np.array(pylist)`	Use Python list to initialize a Numpy array
`np.zeros()`	`shape, dtype=float, order='C'`
`np.ones()`	`shape, dtype=None, order='C'`
`np.empty()`	`shape, dtype=float, order='C'`
`np.full()`	`shape, fill_value, dtype=None, order='C'`
`np.zeros_like()`	`a, dtype=None, order='K', subok=True, shape=None`
`np.ones_like()`	`a, dtype=None, order='K', subok=True, shape=None`
`np.empty_like()`	`prototype, dtype=None, order='K', subok=True, shape=None`
`np.full_like()`	`a, fill_value, dtype=None, order='K', subok=True, shape=None`

Initialize Numpy array with monotonic sequence

function	parameter
`np.arange()`	`[start,] stop[, step,], dtype=None`
`np.linspace()`	`start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0`

create random array

- Old-style creation method (deprecated)

function	parameter
`np.random.randint`	`low, high=None, size=None, dtype=int`
`np.random.rand`	`d0, d1, ..., dn`
`np.random.uniform`	`low=0.0, high=1.0, size=None`

- new creation method

First create the object : `rng = np.random.default_rng()`

function	parameter
`rng.integers`	`low, high=None, size=None, dtype=np.int64, endpoint=False`
`rng.random`	`size=None, dtype=np.float64, out=None`
`rng.uniform`	`low=0.0, high=1.0, size=None`

2. Vector indexing

basic indexing operations

Can specify single index, index range, reverse index, and array index

- Define 1D array (one-dimensional array)

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

Operation and effect

index operation	result	Effect
`a[1]`	`2`	back to view
`a[2:4]`	`array([3, 4])`	back to view
`a[-2:]`	`array([4, 5])`	back to view
`a[::2]`	`array([1, 3, 5])`	back to view
`a[[1,3,4]]`	`array([2, 4, 5])`	fancy indexing, return new array

- Define 2D array (two-dimensional array)

```
a = np.array([[3, 4, 5, 6], [2, 7, 0, -1], [1, 5, 3, 18], [2, 6, 1989, 3]])
>>> a
array([[   3,    4,    5,    6],
       [   2,    7,    0,   -1],
       [   1,    5,    3,   18],
       [   2,    6, 1989,    3]])
```

Operation and effect

index operation	result	Effect
a[1]	array([2,7,0,-1])	return 1 line
a[2:4]	array([[1,5,3,18], [2,6,1989,3]])	return 2 to 4 line
a[-2:]	array([1,5,3,18])	Return to last row
a[::2]	array([[3,4,5,6], [1,5,3,18]])	Return the first 0 and 2 row
a[[0,1,3]]	array([[3,4,5,6],[2,7,0,-1], [2,6,1989,3]])	fancy indexing, return new array, row 0 , 1 and 3 row
a[:,[1,3]]	array([[4,6], [7,-1], [5,18], [6,3]]])	fancy indexing, returns a new array, with columns 1 and 3 columns

Python list vs Numpy list

Python List	Numpy List
a = [1, 2, 3]	a = np.array([1, 2, 3])
b = a (no copy)	b = a (no copy)
c = a[:] (copy)	c = a[:] (no copy)
d = a.copy() (copy)	d = a.copy() (copy)

Boolean index

Define the array: a = np.array([1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1])

Logical comparison (returns a Boolean array)

```
>>> a > 5
array([False, False, False, False, False,  True,  True,  True, False,
       False, False, False, False])
```

`any` and `all` functions

```
>>> np.any(a > 5)
True

>>> np.all(a > 5)
False
```

Utilize Boolean array index

```
>>> a[a > 5]
array([6, 7, 6])

>>> a[(a >= 3) & (a <= 5)]
array([3, 4, 5, 5, 4, 3])
```

`np.where` and `np.clip` functions

function	parameter	effect
`np.where()`	`condition, [x, y]` `condition` : array_like, bool	If `condition` is `true` , yield `x` , otherwise yield `y` . If `x` or `y` is not specified, return the value in the original array
`np.clip()`	`a, a_min, a_max, out=None, **kwargs`	Specify the range of values `[a_min, a_max]` , less than `a_min` the assignment `a_min` , greater than `a_max` the assignment `a_max`

3. Vector operations

Basic operations: addition, subtraction, multiplication and division between vectors

```
# 定义两个数组
>>> a = np.array([4, 8])
>>> b = np.array([2, 5])
# 加
>>> a + b
```

```
array([ 6, 13])
# 减
>>> a - b
array([2, 3])
# 乘
>>> a * b
array([ 8, 40])
# 除
>>> a / b
array([2. , 1.6])
# 整除
>>> a // b
array([2, 1], dtype=int32)
# 乘方
>>> a ** b
array([   16, 32768], dtype=int32)
```

Basic operations: addition, subtraction, multiplication and division between vector and scalar

```
# 定义数组
>>> c = np.array([1, 2])
# 加
>>> c + 3
array([4, 5])
# 减
>>> c - 3
array([-2, -1])
# 乘
>>> c * 3
array([3, 6])
# 除
>>> c / 3
array([0.33333333, 0.66666667])
# 整除
>>> c // 2
array([0, 1], dtype=int32)
# 乘方
>>> c ** 2
array([1, 4], dtype=int32)
```

truncated approximation function

`np.floor` round down (round to negative infinity, -$\infty$)

```
>>> np.floor([1.1, 1.5, 1.9, 2.5])
array([1., 1., 1., 2.])
```

`np.ceil` 向上取整（round to negative infinity, +$\infty$)

```
>>> np.ceil([1.1, 1.5, 1.9, 2.5])
array([2., 2., 2., 3.])
```

`np.round` Truncate to the nearest integer (around to nearest integer)

```
>>> np.round([1.1, 1.5, 1.9, 2.5])
array([1., 2., 2., 2.])
```

some math functions

```
# 开方
>>> np.sqrt([4, 9])
array([2., 3.])
# 以e为底的幂乘
>>> np.exp([1, 2])
array([2.71828183, 7.3890561 ])
# 以e为底的对数 (the logrithm of np.e to base e)
>>> np.log([np.e, np.e**2])
array([1., 2.])
# 点积
>>> np.dot([1,2], [3,4])
11
# 点积的另一种写法
>>> np.array([1,2]) @ np.array([3,4])
11
# 叉积
>>> np.cross([2, 0, 0], [0, 3, 0])
array([0, 0, 6])
# 正弦
>>> np.sin([np.pi, np.pi/2])
array([1.2246468e-16, 1.0000000e+00])
# 反正弦
>>> np.arcsin([0, 1])
array([0.        , 1.57079633])
# 平方和的开方
>>> np.hypot([3,5], [4,12])
array([ 5., 13.])
```

some trigonometric functions

Trigonometric functions	inverse trigonometric functions	hyperbolic function	inverse hyperbolic function
`sin`	`arcsin`	`sinh`	`arcsinh`
`cos`	`arccos`	`cosh`	`arccosh`
`tan`	`arctan`	`tanh`	`arctanh`

Hyperbolic sine function: $\sinh{x}$

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

Hyperbolic cosine function: $\cosh{x}$ $$ \cosh{x} = \frac{e^x + e^{-x}}{2} $$

Hyperbolic tangent function: $\tanh{x}$ $$ \tanh{x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} $$

Basic Statistical Functions

```
# 最大值
>>> np.max([1, 2, 3])
3
# 最大值
>>> np.array([1, 2, 3]).max()
3
# 最大值的索引
>>> np.array([1, 2, 3]).argmax()
2
# 最小值
>>> np.array([1, 2, 3]).min()
1
# 最小值的索引
>>> np.array([1, 2, 3]).argmin()
0
# 求和
>>> np.array([1, 2, 3]).sum()
6
# 求平均
>>> np.array([1, 2, 3]).mean()
2.0
# 标准差
>>> np.array([1, 2, 3]).var()
0.6666666666666666
# 方差
```

```
>>> np.array([1, 2, 3]).std()
0.816496580927726
```

sort function

Python List	Numpy Arrays	Effect
`a.sort()`	`a.sort()`	sort in place
`sorted(a)`	`np.sort(a)`	return a new sorted array
`a.sort(key=f)`	-	sort with key
`a.sort(reversed=False)`	-	ascending/descending order

4. Searching for an element in a vector

Python list search - index method

Python `list` has `index` methods, Numpy doesn't

`a.index(x [, i [, j]])`

Here `x` is the element to be found, `i` and `j` are the upper and lower limits of the specified interval, if not found, it will `raise exception`

```
>>> a = [12, 0, -1, 78, 99]
>>> a.index(78)
3
>>> a.index(78, 4)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a.index(78, 4)
ValueError: 78 is not in list
```

Numpy list search

There are three ways to find an element in Numpy

`np.where`

```
>>> a = [12, 0, -1, 78, 99]
>>> np.where(a == 78)[0][0]
3
```

`next` + `np.ndenumerate` (This needs `Numba` to be accelerated, otherwise it is the `np.where` same as above, slower)

```
>>> a = [12, 0, -1, 78, 99]
>>> next(i[0] for i, v in np.ndenumerate(a) if v == 78)
3
```

`np.searchsorted`

```
>>> a = [12, 0, -1, 78, 99]
>>> b = np.sort(a)
>>> b
array([-1,  0, 12, 78, 99])
>>> np.searchsorted(b, 78)
3
```

5. Comparing floats

`np.allclose(a, b)` for floating-point comparisons within tolerance

but, **there is no silver bullet!**

expression	result
`0.1 + 0.2 == 0.3`	`False`
`np.allclose(0.1 + 0.2, 0.3)`	`True`
`math.isclose(0.1 + 0.2, 0.3)`	`True`

expression	result
`1e-9 == 2e-9`	`False`
`np.allclose(1e-9, 2e-9)`	`True`
`math.isclose(1e-9, 2e-9)`	`False`

expression	result
`0.1 + 0.2 - 0.3 == 0`	`True`
`np.allclose(0.1 + 0.2 - 0.3, 0)`	`True`
`math.isclose(0.1 + 0.2 - 0.3, 0)`	`False`

Notice

- `np.allclose` All comparison numbers are assumed to have a scale of 1.

  For example, if you are on the nanosecond level, you need to divide the default `atol` parameter by 1e9: `np.allclose(1e-9,2e-9, atol=1e-17)==False` .
- `math.isclose` Make no assumptions about the numbers being compared, instead requiring the user to provide a reasonable `abs_tol` value (`np.allclose` the default `atol` is `1e-8`)
- See the link below for some questions
  - floating-point guide
  - NumPy issue on GitHub.

6. Matrices, the 2D Array

basic concept

- Now in Numpy, **matrix** and **2D Array** refer to the same concept and can be used interchangeably
- In Numpy, the original class `matrix` is no longer used (deprecated)

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

- Its `.shape` attributes return a tuple with two elements, the first is **the number of rows** and the second is the number of **columns**
- `len(a)` **Returns the number of rows of** the 2D array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.dtype
dtype('int32')
>>> a.shape
(2, 3)
>>> len(a)
```

```
2
>>> a.shape[0]
2
```

Common functions

The previous `zeros` , `ones` , `full` , `empty` and `eye` all can be used to generate a 2D array

It should be noted that the tuple specifying the 2D array must be enclosed in parentheses `()` to indicate the first parameter, because the second parameter is reserved `dtype` for

```
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.full((3, 2), 7)
array([[7, 7],
       [7, 7],
       [7, 7]])
>>> np.empty((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.eye(3, 3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

and `random` function

```
# x服从[0, 10)上的均匀分布（整数）
>>> np.random.randint(0, 10, [3, 2])
array([[7, 1],
       [6, 6],
       [2, 1]])
# x服从[0, 1)上的均匀分布
>>> np.random.rand(3, 2)
array([[0.1518058 , 0.47967987],
       [0.0242219 , 0.46161326],
       [0.64206284, 0.02072145]])
# x服从[0, 1)上的均匀分布（浮点数）
>>> np.random.uniform(1, 10, [3, 2])
array([[4.60676287, 4.64315581],
       [9.56576352, 2.2958745 ],
       [2.18304639, 5.9622002 ]])
```

`np.random.randn` , x obeys the standard normal distribution, $ N(\mu, \sigma^2), \mu = 0, \sigma = 1 $

```
# x服从标准正态分布
>>> np.random.randn(3, 2)
array([[-0.95367737, -0.35999719],
       [-0.27186541,  1.10111502],
       [-0.36303053,  0.5372727 ]])
```

`np.random.normal` , x obeys normal distribution, $ N(\mu, \sigma^2), \mu = 10, \sigma = 2 $

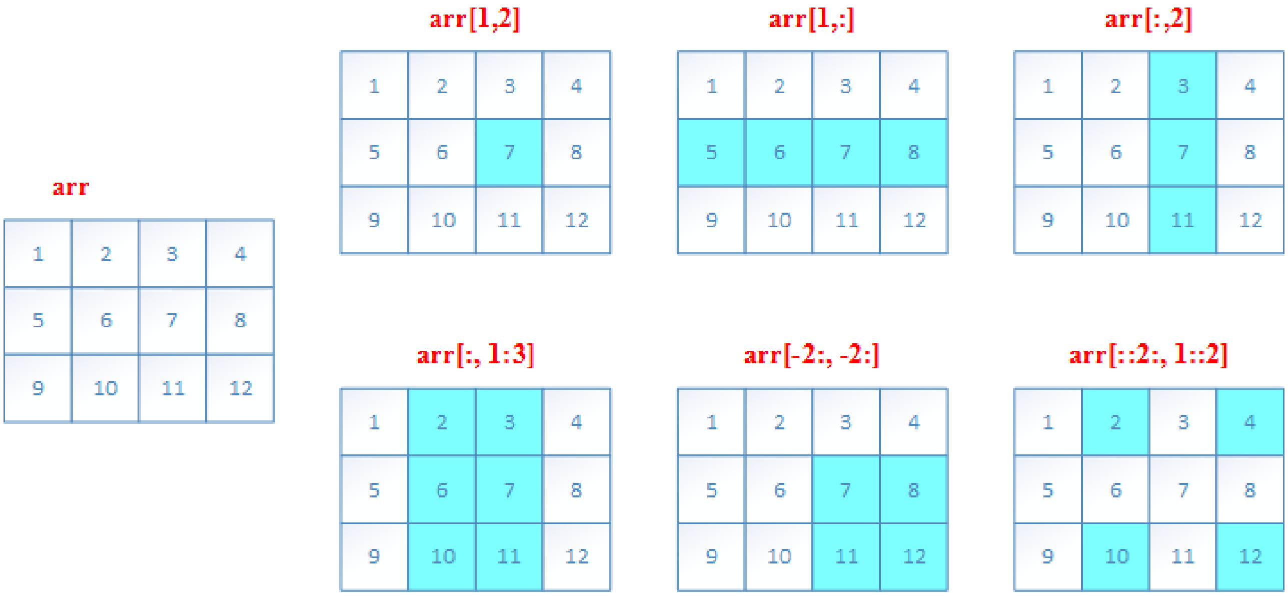
```
# x服从正态分布
>>> np.random.normal(10, 2, [3, 2])
array([[10.06207497,  9.45178632],
       [ 9.02901148, 10.92862084],
       [12.53682855, 10.20647998]])
```

Index of a two-dimensional array (slice)

An example similar to a slice of a one-dimensional array is as follows

```
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> a[1,2]
7
>>> a[1,:]
array([5, 6, 7, 8])
>>> a[:,2]
array([ 3,  7, 11])
>>> a[:,1:3]
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
>>> a[-2:,-2:]
array([[ 7,  8],
       [11, 12]])
```

```
>>> a[::2, 1::2]
array([[ 2,  4],
       [10, 12]])
```



7. The axis argument

axis argument

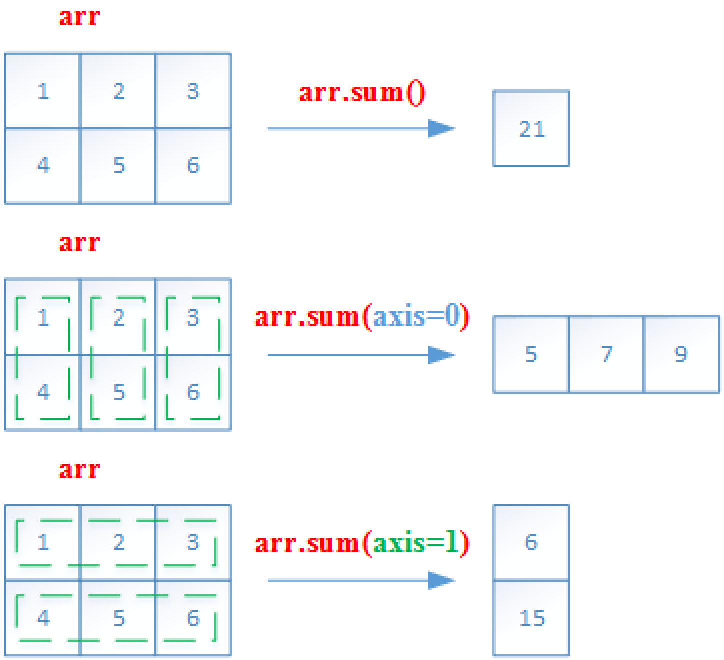
- Numpy introduces `axis` parameters to achieve cross-row or cross-column operations
- `axis` The value of the parameter is actually the number of dimensions. (first dimension `axis=0` , second dimension `axis=1` , and so on)
- In a two-dimensional array, it `axis=0` represents the column direction and `axis=1` represents the row direction

##axis	direction
`axis = 0`	**column** direction (i.e. all rows)
`axis = 1`	**row** direction (i.e. all columns)

As `sum` an example

```
>>> a = np.array([[1,2,3], [4, 5,6]])
>>> a.sum()
21
>>> a.sum(axis=0)
array([5, 7, 9])
```

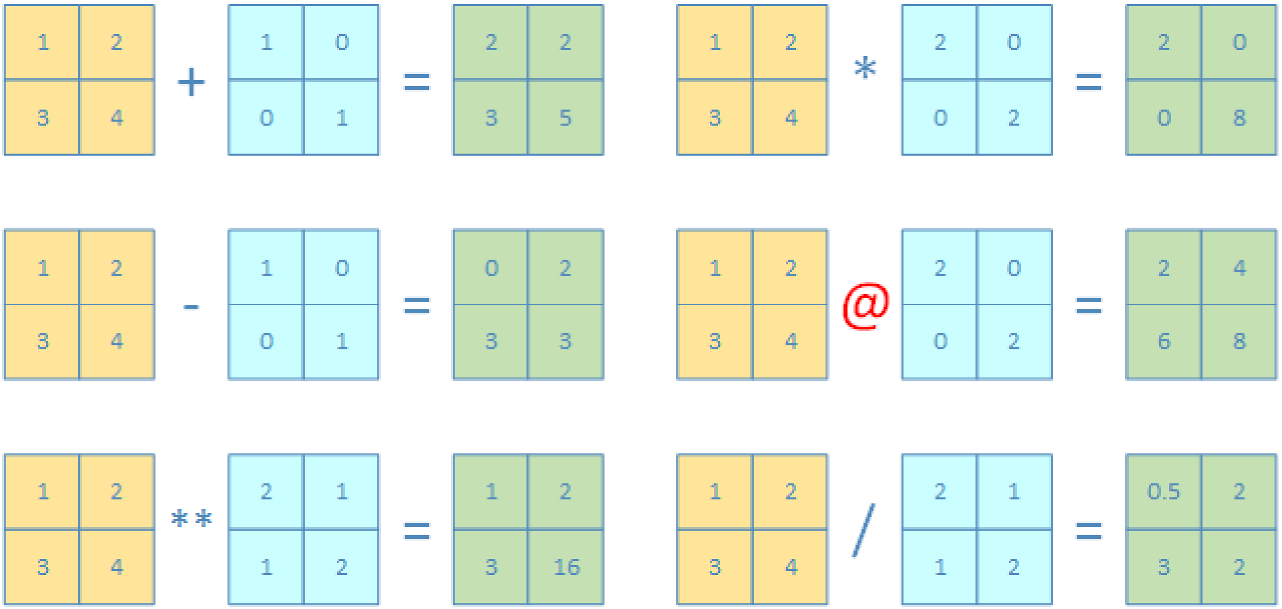
```
>>> a.sum(axis=1)
array([ 6, 15])
```



Arithmetic operations on matrices

- 2D Array supports operations between elements of the array: `+` , `-` , `*` , `/` , `//` , `**`
- 2D Array also supports outer product (outer product)

```
>>> np.array([[1,2],[3,4]]) + np.array([[1,0],[0,1]])
array([[2, 2],
       [3, 5]])
>>> np.array([[1,2],[3,4]]) - np.array([[1,0],[0,1]])
array([[0, 2],
       [3, 3]])
>>> np.array([[1,2],[3,4]]) ** np.array([[2,1],[1,2]])
array([[ 1,  2],
       [ 3, 16]], dtype=int32)
>>> np.array([[1,2],[3,4]]) * np.array([[2,0],[0,2]])
array([[2, 0],
       [0, 8]])
>>> np.array([[1,2],[3,4]]) @ np.array([[2,0],[0,2]])
array([[2, 4],
       [6, 8]])
>>> np.array([[1,2],[3,4]]) / np.array([[2,1],[1,2]])
array([[0.5, 2. ],
       [3. , 2. ]])
```



$$\begin{bmatrix}1&2\\3&4\end{bmatrix}+\begin{bmatrix}1&0\\0&1\end{bmatrix}=\begin{bmatrix}2&2\\3&5\end{bmatrix}\qquad\begin{bmatrix}1&2\\3&4\end{bmatrix}*\begin{bmatrix}2&0\\0&2\end{bmatrix}=\begin{bmatrix}2&0\\0&8\end{bmatrix}$$

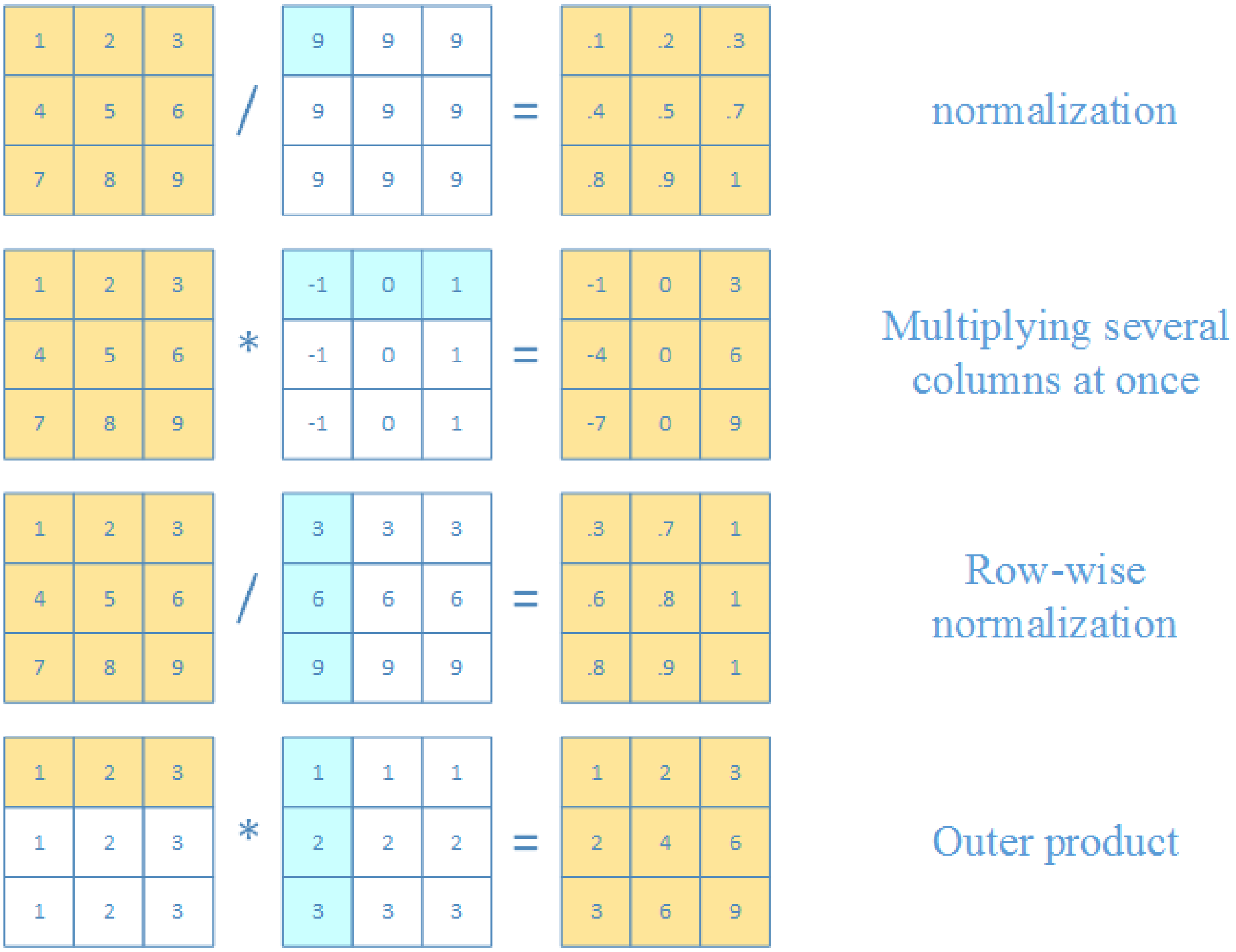
$$\begin{bmatrix}1&2\\3&4\end{bmatrix}-\begin{bmatrix}1&0\\0&1\end{bmatrix}=\begin{bmatrix}0&2\\3&3\end{bmatrix}\qquad\begin{bmatrix}1&2\\3&4\end{bmatrix}@\begin{bmatrix}2&0\\0&2\end{bmatrix}=\begin{bmatrix}2&4\\6&8\end{bmatrix}$$

$$\begin{bmatrix}1&2\\3&4\end{bmatrix}**\begin{bmatrix}2&1\\1&2\end{bmatrix}=\begin{bmatrix}1&2\\3&16\end{bmatrix}\qquad\begin{bmatrix}1&2\\3&4\end{bmatrix}/\begin{bmatrix}2&1\\1&2\end{bmatrix}=\begin{bmatrix}0.5&2\\3&2\end{bmatrix}$$

Numpy can implement the following mixed operations through the scalar broadcasting mechanism (broadcasting from scalar)

- between vectors and matrices
- between two vectors

```
>>> a = np.array([[1,2,3], [4, 5,6], [7,8,9]])
>>> a / 9
array([[0.11111111, 0.22222222, 0.33333333],
       [0.44444444, 0.55555556, 0.66666667],
       [0.77777778, 0.88888889, 1.        ]])
>>> a * np.array([-1, 0, 1])
array([[-1,  0,  3],
       [-4,  0,  6],
       [-7,  0,  9]])
>>> a / np.array([[3],[6],[9]])
array([[0.33333333, 0.66666667, 1.        ],
       [0.66666667, 0.83333333, 1.        ],
       [0.77777778, 0.88888889, 1.        ]])
>>> np.array([1, 2, 3]) * np.array([[1], [2], [3]])
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```



$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} / \begin{bmatrix}9&9&9\\9&9&9\\9&9&9\end{bmatrix} = \begin{bmatrix}.1&.2&.3\\.4&.5&.7\\.8&.9&1\end{bmatrix}$$

normalization

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} * \begin{bmatrix}-1&0&1\\-1&0&1\\-1&0&1\end{bmatrix} = \begin{bmatrix}-1&0&3\\-4&0&6\\-7&0&9\end{bmatrix}$$

Multiplying several columns at once

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} / \begin{bmatrix}3&3&3\\6&6&6\\9&9&9\end{bmatrix} = \begin{bmatrix}.3&.7&1\\.6&.8&1\\.8&.9&1\end{bmatrix}$$

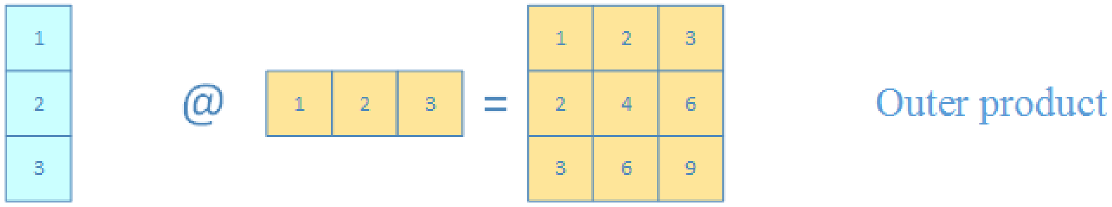
Row-wise normalization

$$\begin{bmatrix}1&2&3\\1&2&3\\1&2&3\end{bmatrix} * \begin{bmatrix}1&1&1\\2&2&2\\3&3&3\end{bmatrix} = \begin{bmatrix}1&2&3\\2&4&6\\3&6&9\end{bmatrix}$$

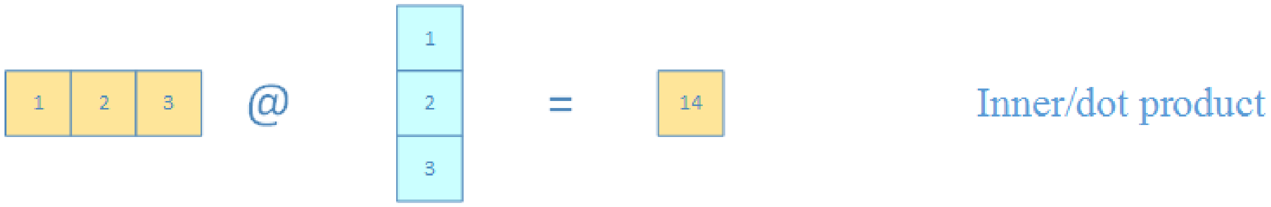
Outer product

Dot and cross products (inner and outer)

```
>>> np.array([[1],[2],[3]]) @ np.array([[1,2,3]])
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])

>>> np.array([1,2,3]) @ np.array([[1],[2],[3]])
array([14])
```



$$\begin{bmatrix}1\\2\\3\end{bmatrix} @ \begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\2 & 4 & 6\\3 & 6 & 9\end{bmatrix}$$



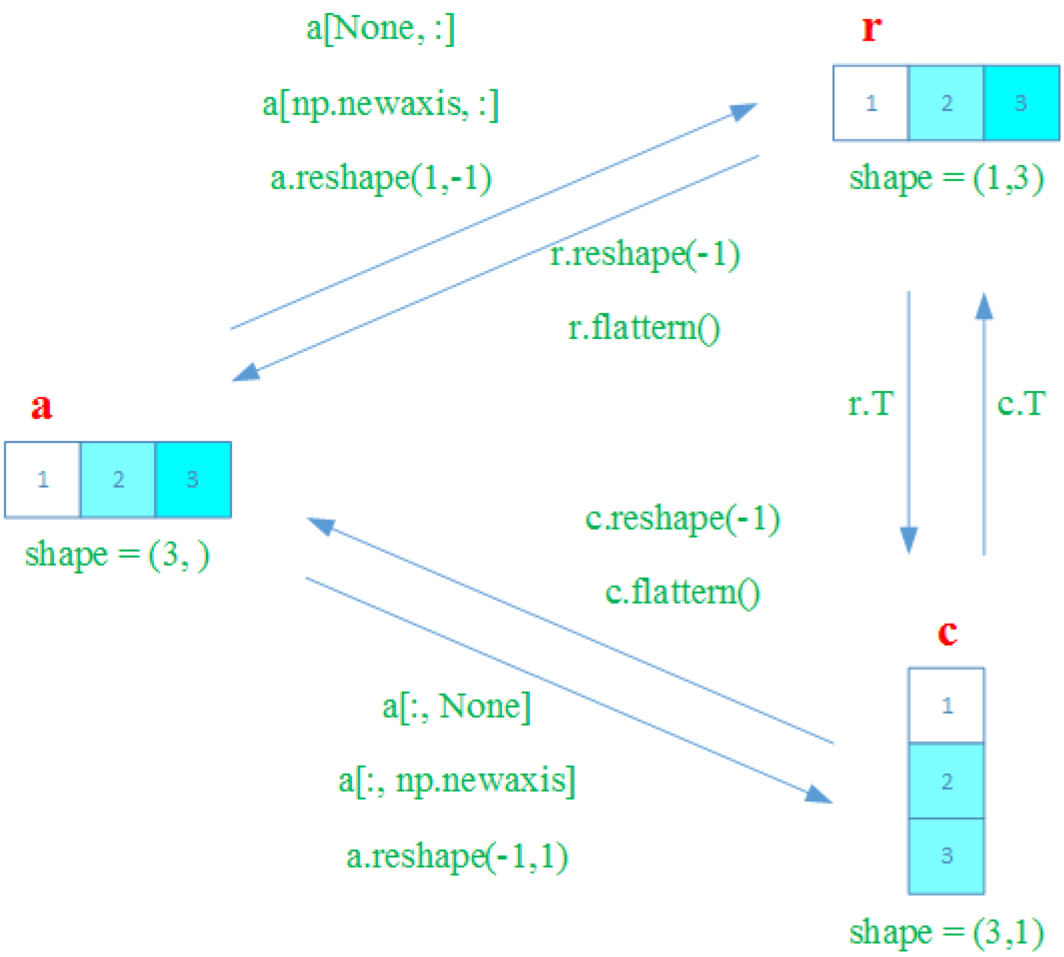
$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} @ \begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}14\end{bmatrix}$$

8. Row vectors & column vectors

- There are three types of vectors in Numpy

vector type	vector types
1D array	1D arrays
2D row vector	2D row vectors
2-dimensional column vector	2D column vectors

The conversion between the three types of vectors is as follows



- `shape` All array sizes (attributes) are vectors except for 1D arrays
  - for example `a.shape == [1, 1, 1, 5, 1, 1]`
  - `shape` The size (property) of a one-dimensional array is `(N, )` of the form
- In Numpy, one-dimensional arrays are row vectors by default.
- Note: In Numpy, a row vector is different from a matrix with only one row vector

  The `shape` attributes of **a row vector are** `(n, )`

  Note that the transpose of a row vector is still a row vector.

```
>>> c = np.array([1, 2, 3])
>>> c.shape
(3,)
>>> c.T
array([1, 2, 3])
```

The `shape` properties of **a matrix with only one row vector are** `(1, n)`

The transpose of a matrix with only one row vector is a matrix with only one column vector.

```
>>> b = np.array([[1,2,3]])
>>> b.shape
(1, 3)
>>> b.T
array([[1],
       [2],
       [3]])
```

- You can `reshape` convert a **row vector** to a matrix of column vectors or a matrix of row vectors by

  Assume that there is an existing row vector `a` as follows

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

  Convert to a matrix of column vectors. `reshape` The first parameter here `-1` indicates the automatic inference decision on this dimension (the first dimension)

  ```
  >>> b = a.reshape(-1, 1)
  >>> b
  array([[1],
         [2],
         [3],
         [4],
         [5],
         [6]])
  >>> b.shape
  (6, 1)
  ```

  Convert to a matrix of row vectors. `reshape` The second parameter here `-1` indicates the automatic inference decision on this dimension (the second dimension)

  ```
  >>> c = a.reshape(1, -1)
  >>> c
  array([[1, 2, 3, 4, 5, 6]])
  >>> c.shape
  (1, 6)
  ```

- Similarly, the `np.newaxis` above row vector can also be transformed using

  Similarly, assume that there is an existing row vector `a` as follows

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

  Convert to a matrix of column vectors.

  here `a[:, None]` is `None` equivalent to `np.newaxis`

  ```
  >>> b = a[:, None] # 这里可以用np.newaxis替换None
  >>> b
  array([[1],
         [2],
         [3],
         [4],
         [5],
         [6]])
  >>> b.shape
  (6, 1)
  ```

  Convert to a matrix of row vectors.

  ```
  >>> c = a[None, :]
  >>> c
  array([[1, 2, 3, 4, 5, 6]])
  >>> c.shape
  (1, 6)
  ```

9. Matrix manipulations

main function

function	effect
`np.hstack`	horizontal stitching
`np.vstatck`	vertical splicing
`np.column_statck`	For 2D and 1D array horizontal splicing

function	effect
`np.hsplit`	Transverse cut (cut along the y axis)
`np.vsplit`	Longitudinal cut (cut along the x axis)
`np.tile`	2D array repeats as a whole
`ndarray.repeat`	element repetition
`np.delete`	delete row or column
`np.insert`	insert row or column
`np.append`	Can realize the function of `np.hstack` and `np.vstack`
`np.pad`	Add the specified number of rows or columns of elements around the matrix

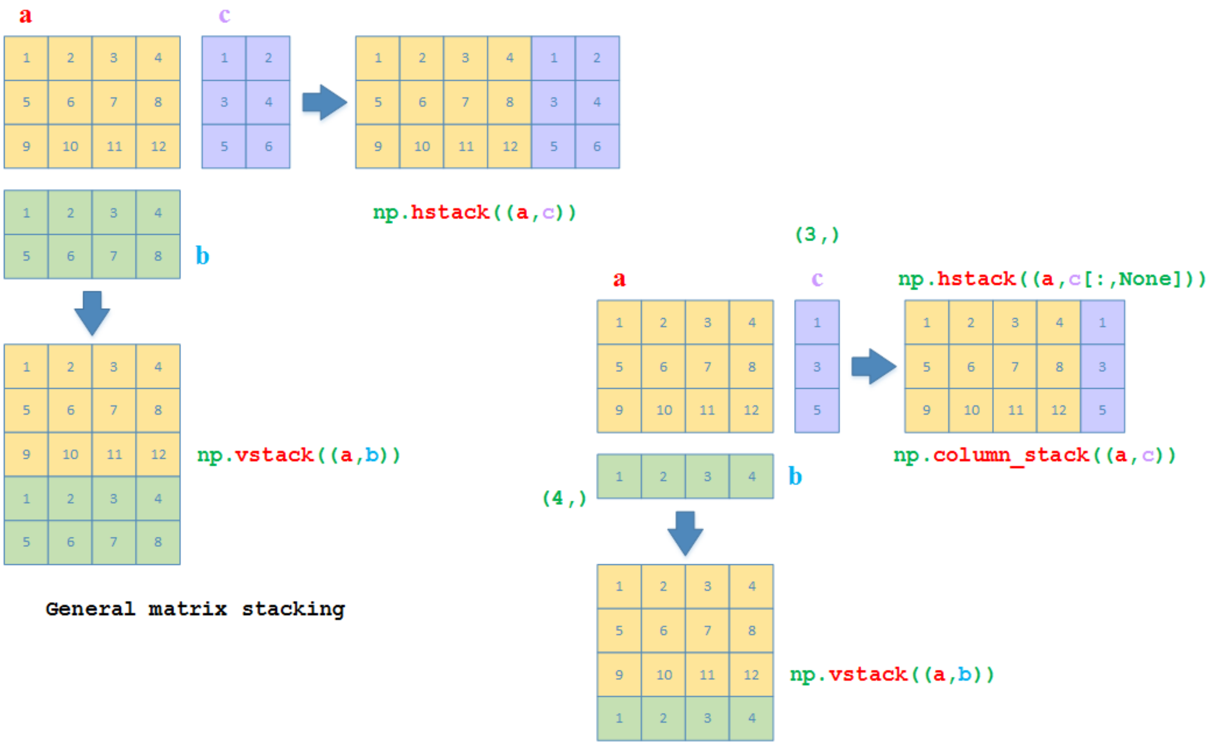
Join and Split Functions

splicing function	partition function
`np.hstack`	`np.hsplit`
`np.vstatck`	`np.vsplit`
`np.column_statck`	

Concatenation function in Numpy

- `np.hstack` : horizontal splicing
- `np.vstatck` : vertical splicing
- `np.column_stack` : used for horizontal splicing of 2D array and 1D array

```
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> b = np.array([[1,2,3,4], [5,6,7,8]])
>>> c = np.array([[1,2], [3,4], [5,6]])
# 纵向拼接
>>> np.vstack((a, b))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [ 1,  2,  3,  4],
       [ 5,  6,  7,  8]])
# 横向拼接
>>> np.hstack((a, c))
array([[ 1,  2,  3,  4,  1,  2],
       [ 5,  6,  7,  8,  3,  4],
       [ 9, 10, 11, 12,  5,  6]])
# b和c现重新改为1D array
>>> b = np.array([1,2,3,4])
# 2D array和1D array可以直接做纵向拼接
>>> np.vstack((a, b))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [ 1,  2,  3,  4]])
>>> c = np.array([1,3,5])
# 2D array和1D array直接做横向拼接会抛异常
>>> np.hstack((a, c))
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    np.hstack((a, c))
  File "<__array_function__ internals>", line 6, in hstack
  File "C:\Python36\lib\site-packages\numpy\core\shape_base.py", line 346, in hs
    return _nx.concatenate(arrs, 1)
  File "<__array_function__ internals>", line 6, in concatenate
ValueError: all the input arrays must have same number of dimensions, but the ar
# 2D array和转换为行向量的1D array再直接做横向拼接
>>> np.hstack((a, c[:,None]))
array([[ 1,  2,  3,  4,  1],
       [ 5,  6,  7,  8,  3],
       [ 9, 10, 11, 12,  5]])
# 或者直接利用np.column_stack来直接拼接2D array和1D array
>>> np.column_stack((a, c))
array([[ 1,  2,  3,  4,  1],
       [ 5,  6,  7,  8,  3],
       [ 9, 10, 11, 12,  5]])
```

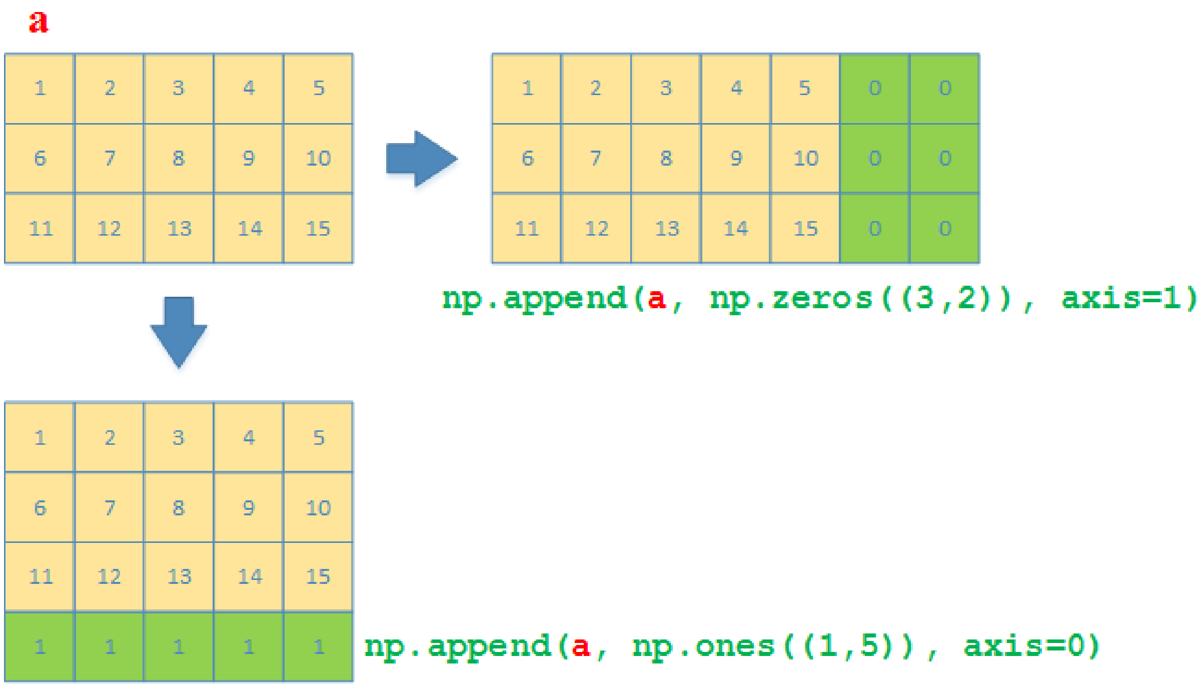


General matrix stacking

Stacking with 1D arrays

- `np.append` The function can achieve the effect of `np.vstack` and `np.hstack`

```
>>> a = np.array([[1,2,3,4,5], [6,7,8,9,10], [11,12,13,14,15]])
# 沿着y方向（列的方向），在矩阵最后添加一行，值都是1
>>> np.append(a, np.ones((1,5)), axis=0)
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10.],
       [11., 12., 13., 14., 15.],
       [ 1.,  1.,  1.,  1.,  1.]])
# 沿着x方向（行的方向），在矩阵最后添加一个3x2的矩阵，值都是0
>>> np.append(a, np.zeros((3,2)), axis=1)
array([[ 1.,  2.,  3.,  4.,  5.,  0.,  0.],
       [ 6.,  7.,  8.,  9., 10.,  0.,  0.],
       [11., 12., 13., 14., 15.,  0.,  0.]])
```



np.append(a, np.zeros((3,2)), axis=1)

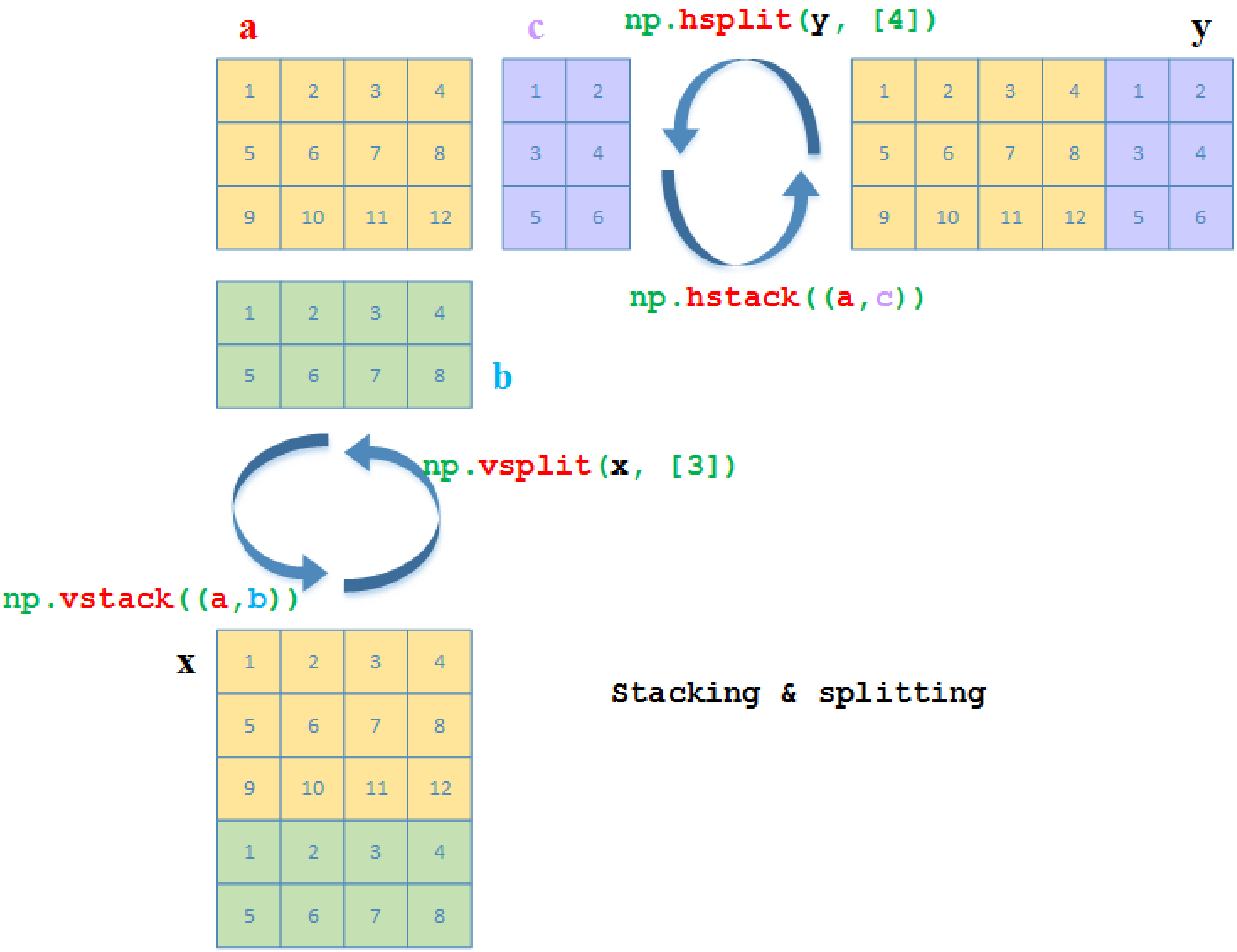
np.append(a, np.ones((1,5)), axis=0)

Split function in Numpy

And `np.vstack` and `np.hstack` correspondingly, there is a partition function

- `np.vsplit` : Split along the `x` axis (axis=0) (vertical split)
    - The second parameter is the index (array) of the **row** , indicating the number of rows from which to start splitting
    - Return a list, a certain element is a split array
- `np.hsplit` : Split along the `y` axis (axis=1) (horizontal split)
    - The second parameter is the index (array) of the **column** , which indicates which column to start splitting from
    - Return a list, a certain element is a split array

```
# 纵向分割
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> b = np.array([[1,2,3,4], [5,6,7,8]])
>>> x = np.vstack((a, b))
>>> np.vsplit(x, [3]) # 第二个参数是行的索引，表示从第几行起开始分割
[array([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]),
 array([[1, 2, 3, 4],
        [5, 6, 7, 8]])]
# 横向分割
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> c = np.array([[1,2], [3,4], [5,6]])
>>> y = np.hstack((a, c))
>>> np.hsplit(y, [4])  # 第二个参数是列的索引，表示从第几列起开始分割
[array([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]),
 array([[1, 2],
        [3, 4],
        [5, 6]])]
```

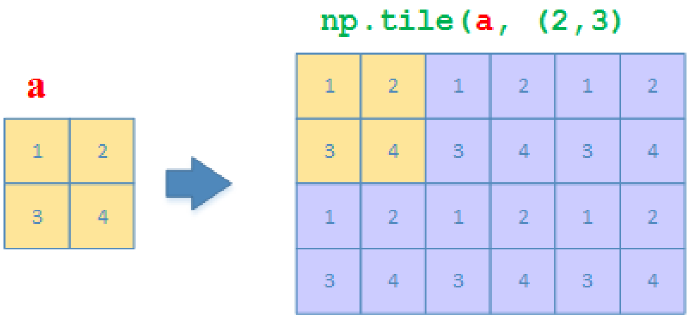


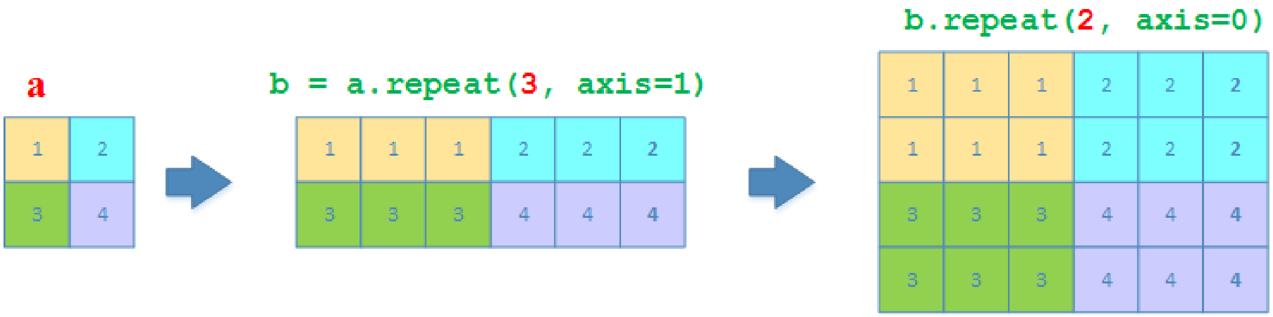
matrix copy

- `np.tile` ： like copy-pasting
  - It is equivalent to treat the entire matrix as a whole, and then copy it in units of the whole
- `np.repeat` ： like collated printing
  - It is equivalent to repeating each element in the matrix in the order of `axist=0` (column direction) or (row direction) `axis=1`

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

```
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
>>> a.repeat(3, axis=1)
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
>>> a.repeat(3, axis=1).repeat(2, axis=0)
array([[1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4],
       [3, 3, 3, 4, 4, 4]])
```





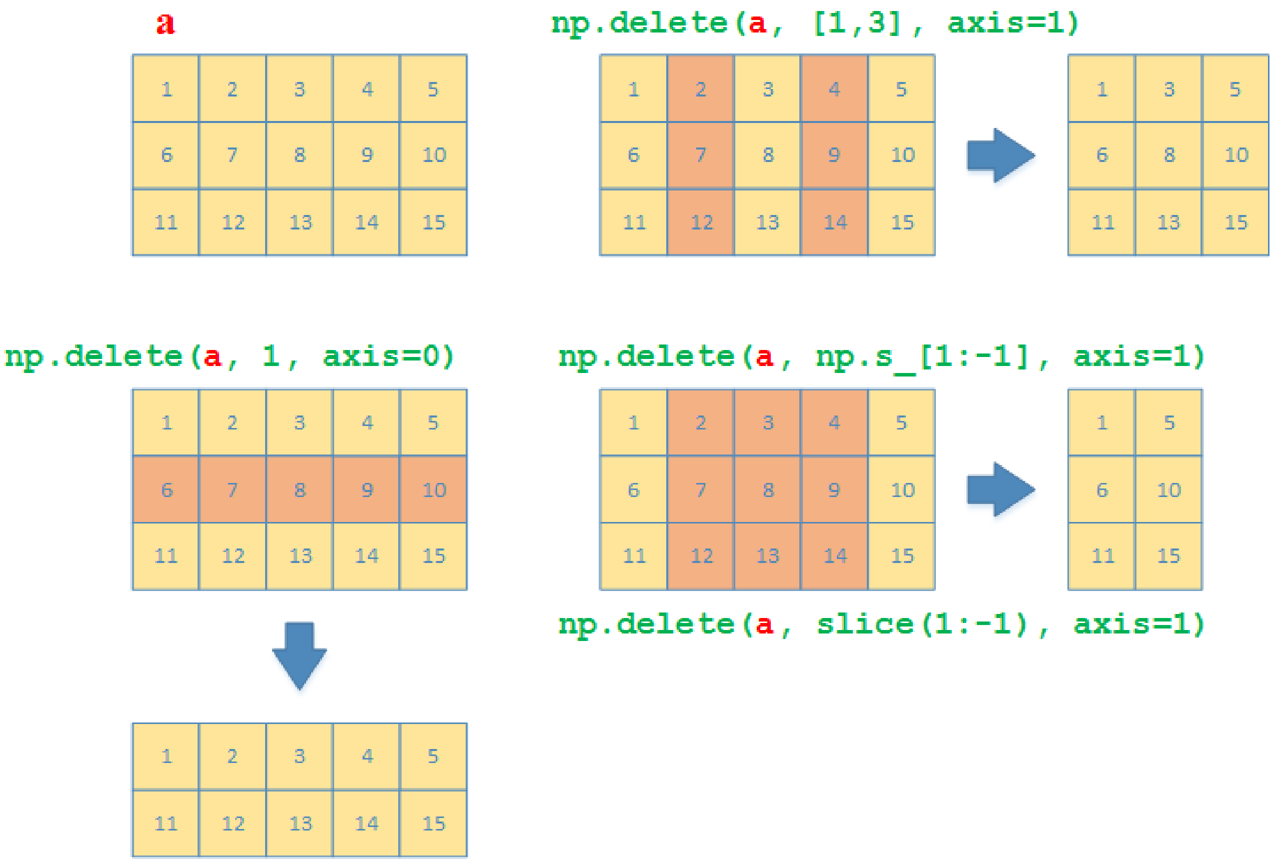
Deletion and insertion of matrix rows and columns

delete and insert functions

delete	insert
`np.delete`	`np.insert`

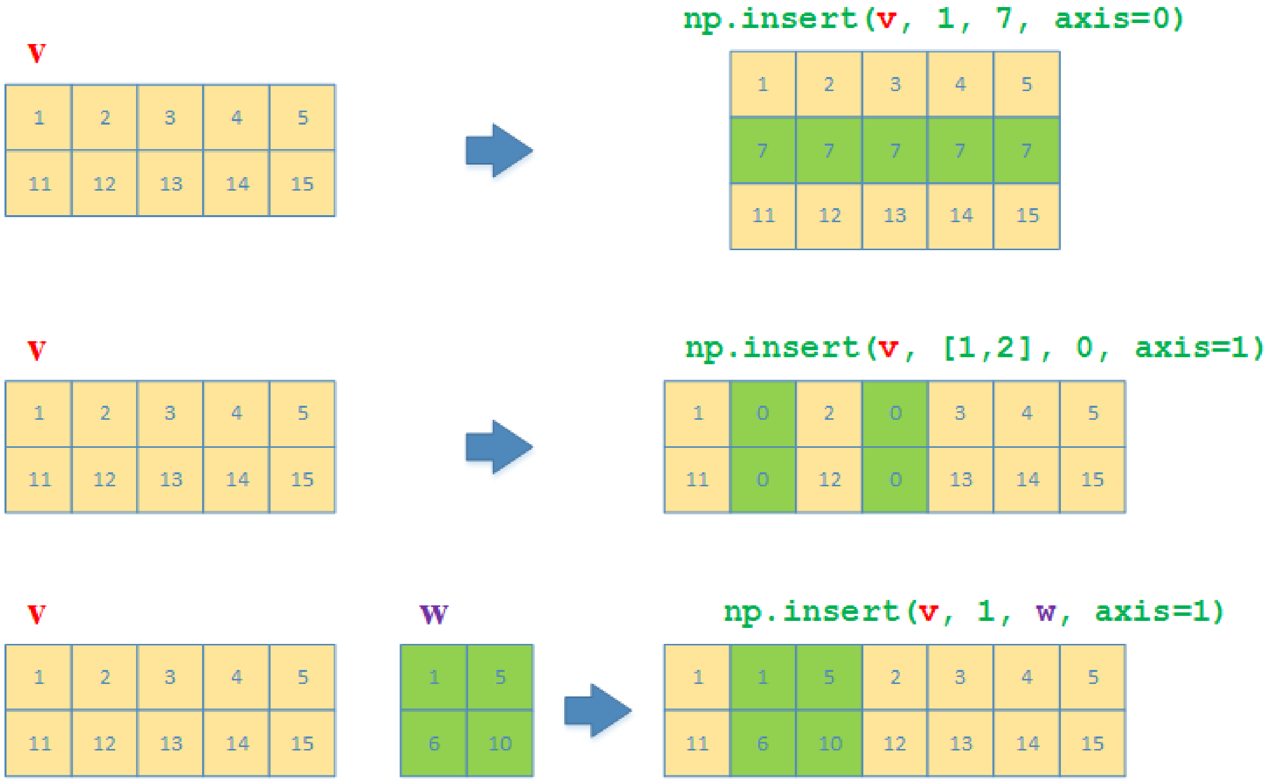
`np.delete` **It is used to specify the row** or **column** to be deleted , and multiple consecutive or discontinuous rows and columns can also be specified at the same time.

```
>>> a = np.array([[1,2,3,4,5], [6,7,8,9,10], [11,12,13,14,15]])
# 删除索引为1的行
>>> np.delete(a, 1, axis=0)
array([[ 1,  2,  3,  4,  5],
       [11, 12, 13, 14, 15]])
# 删除索引为1和3的列
>>> np.delete(a, [1,3], axis=1)
array([[ 1,  3,  5],
       [ 6,  8, 10],
       [11, 13, 15]])
# 删除第一（含）到倒数第一（不含）的列
>>> np.delete(a, np.s_[1:-1], axis=1)
array([[ 1,  5],
       [ 6, 10],
       [11, 15]])
# 删除第一（含）到倒数第一（不含）的列
>>> np.delete(a, slice(1,-1), axis=1)
array([[ 1,  5],
       [ 6, 10],
       [11, 15]])
```


a
np.delete(a, [1,3], axis=1)
np.delete(a, 1, axis=0)
np.delete(a, np.s_[1:-1], axis=1)
np.delete(a, slice(1:-1), axis=1)

`np.delete` Corresponding to, it is used to specify the **row** or **column** `np.insert` to be inserted , and can also specify multiple continuous or discontinuous rows and columns at the same time

```
>>> v = np.array([[1,2,3,4,5], [6,7,8,9,10]])
# 沿着y方向（列），在第一行前面插入一行，元素都是7
>>> np.insert(v, 1, 7, axis=0)
array([[ 1,  2,  3,  4,  5],
       [ 7,  7,  7,  7,  7],
       [ 6,  7,  8,  9, 10]])
# 沿着x方向（行），在第一、二列前面分别各插入一列，元素都是0
>>> np.insert(v, [1, 2], 0, axis=1)
array([[ 1,  0,  2,  0,  3,  4,  5],
       [ 6,  0,  7,  0,  8,  9, 10]])
>>> w = np.array([[1,5], [6,10]])
# 沿着x方向（行），在第一列前面将矩阵w插入
>>> np.insert(v, [1], w, axis=1)
array([[ 1,  1,  5,  2,  3,  4,  5],
       [ 6,  6, 10,  7,  8,  9, 10]])
```



function pad

In 2D arrays (actually more than 2D), it `np.pad` can be used to add values to the surrounding boundaries of the 2D array (matrix)

Typical usage such as

```
np.pad(arr, ((M, N), (S, T)), constant_values=1)
```

The meaning of the boundary parameters here is as follows

`M` `0` : Add a row before `M` row th of the two-dimensional array

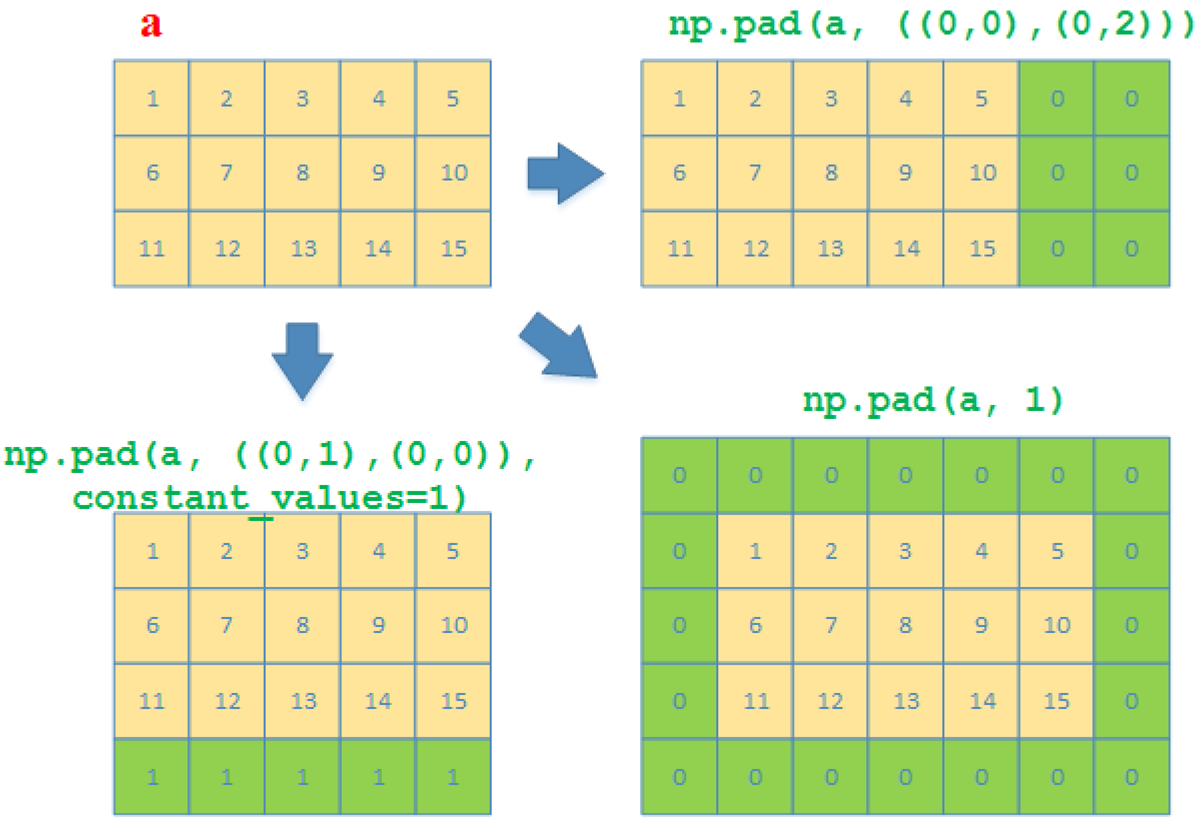
`N` `N` : Add rows before the last row of the two-dimensional array

`S` `0` : add a column before `S` the first column of the two-dimensional array

`T` `T` : add columns before the last column of the two-dimensional array

example:

```
a = np.array([[1,2,3,4,5], [6,7,8,9,10], [11,12,13,14,15]])
# 在最后一行后面再添加一行
>>> np.pad(a, ((0,1),(0,0)), constant_values=1)
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [ 1,  1,  1,  1,  1]])
# 在最后一列后面再添加两列
>>> np.pad(a, ((0,0),(0,2)))
array([[ 1,  2,  3,  4,  5,  0,  0],
       [ 6,  7,  8,  9, 10,  0,  0],
       [11, 12, 13, 14, 15,  0,  0]])
# 在四周个添加一行/列
>>> np.pad(a, 1)
array([[ 0,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4,  5,  0],
       [ 0,  6,  7,  8,  9, 10,  0],
       [ 0, 11, 12, 13, 14, 15,  0],
       [ 0,  0,  0,  0,  0,  0,  0]])
```



10. Meshgrids

Suppose you want to generate a meshgrid as follows $$ A_{ij} = j - i $$

There are four ways to generate meshgrids (here it is assumed that the size of the generated matrix is `3x2`)

The C way

```
>>> A = np.empty((2,3))
>>> for i in range(2):
        for j in range(3):
            A[i,j] = j - i
>>> A
array([[ 0.,  1.,  2.],
       [-1.,  0.,  1.]])
```

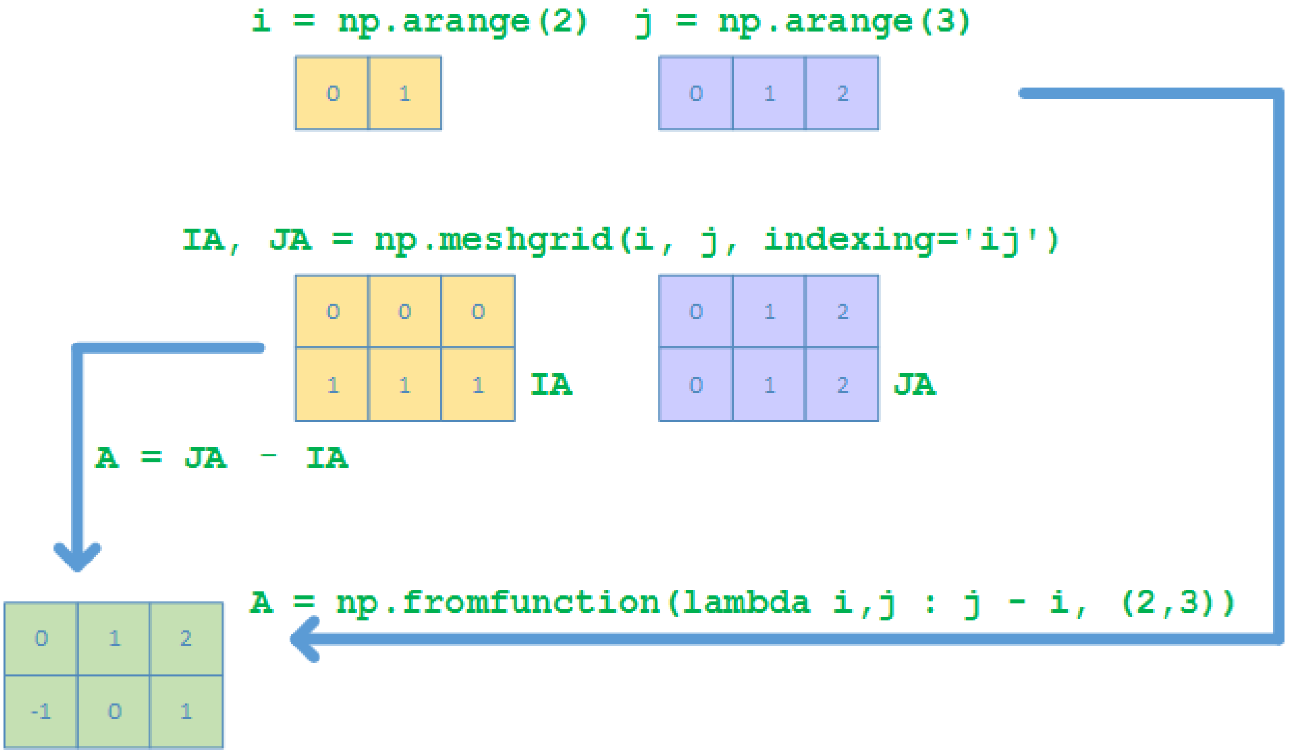
The Python way

```
>>> c = [[(j-i) for j in range(3)] for i in range(2)]
>>> A = np.array(c)
>>> A
```

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

The Matlab way

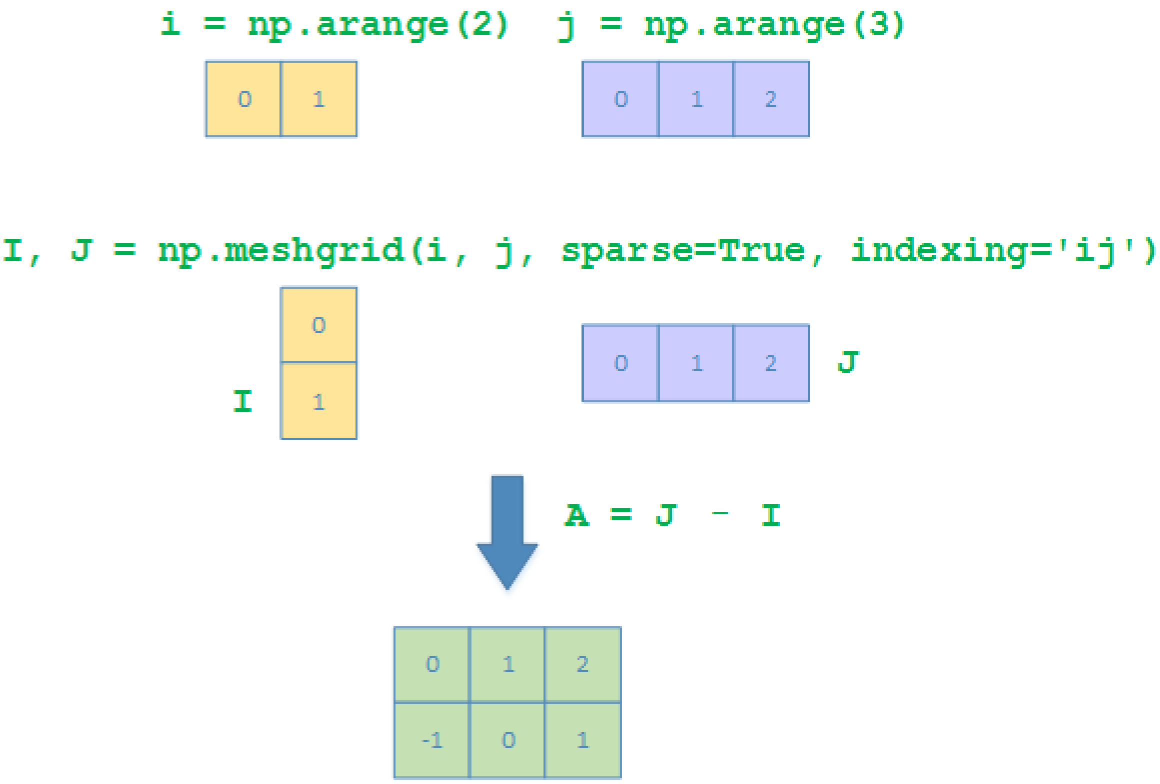
```
>>> i, j = np.arange(2), np.arange(3)
>>> ia, ja = np.meshgrid(i, j, indexing='ij')
>>> ia, ja
(array([[0, 0, 0],
       [1, 1, 1]]),
 array([[0, 1, 2],
       [0, 1, 2]]))
>>> A = ja - ia
>>> A
array([[ 0,  1,  2],
       [-1,  0,  1]])
# 或者
>>> A = np.fromfunction(lambda i,j : j - i, (2,3))
>>> A
array([[ 0.,  1.,  2.],
       [-1.,  0.,  1.]])
```



i = np.arange(2)
j = np.arange(3)
0
1
0
1
2
IA, JA = np.meshgrid(i, j, indexing='ij')
0
0
0
1
1
1
IA
0
1
2
0
1
2
JA
A = JA - IA
A = np.fromfunction(lambda i,j : j - i, (2,3))
0
1
2
-1
0
1

The Numpy way

```
>>> i, j = np.arange(2), np.arange(3)
>>> IA, JA = np.meshgrid(i, j, sparse=True, indexing='ij')
>>> IA
array([[0],
       [1]])
>>> JA
array([[0, 1, 2]])
>>> A = JA - IA
>>> A
array([[ 0,  1,  2],
       [-1,  0,  1]])
```



11. Matrix Statistics

Common Statistical Functions

function	effect
`np.min`	minimum value
`np.max`	maximum value
`np.argmin`	minimum index

function	effect
`np.argmax`	index of maximum
`np.any`	at least one non-zero
`np.all`	All non-zero
`np.sum`	to sum
`np.std`	standard deviation
`np.var`	variance
`np.mean`	average/expectation
`np.median`	average/expectation
`np.percentile`	percentage value

- Each function can `axis` calculate the statistical value of all elements without parameters
- can be added `axis=0` to calculate the statistical value along the column direction
- can be added `axis=1` to calculate the statistical value along the row direction

np.min

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.min(a)
1
>>> np.min(a, axis=0)
array([4, 3, 1])
>>> np.min(a, axis=1)
array([4, 1])
```

np.max

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.max(a)
9
>>> np.max(a, axis=0)
array([9, 8, 5])
```

```
>>> np.max(a, axis=1)
array([8, 9])
```

np.argmin

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.argmin(a)
5
# 可以使用unravel_index来获取转换为二维的索引
>>> np.unravel_index(np.argmin(a), a.shape)
(1, 2)
>>> np.argmin(a, axis=0)
array([0, 1, 1], dtype=int64)
>>> np.argmin(a, axis=1)
array([0, 2], dtype=int64)
```

np.argmax

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.argmax(a)
3
# 可以使用unravel_index来获取转换为二维的索引
>>> np.unravel_index(np.argmax(a), a.shape)
(1, 0)
>>> np.argmax(a, axis=0)
array([1, 0, 0], dtype=int64)
>>> np.argmax(a, axis=1)
array([1, 0], dtype=int64)
```

np.any

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.any(a)
True
>>> np.any(a, axis=0)
array([ True,  True,  True])
>>> np.any(a, axis=1)
array([ True,  True])
```

np.all

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.all(a)
True
>>> np.all(a, axis=0)
array([ True,  True,  True])
>>> np.all(a, axis=1)
array([ True,  True])
```

np.sum

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.sum(a)
30
>>> np.sum(a, axis=0)
array([13, 11,  6])
>>> np.sum(a, axis=1)
array([17, 13])
```

np.std

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.std(a)
2.7688746209726918
>>> np.std(a, axis=0)
array([2.5, 2.5, 2. ])
>>> np.std(a, axis=1)
array([1.69967317, 3.39934634])
```

np.var

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.var(a)
7.666666666666667
>>> np.var(a, axis=0)
array([6.25, 6.25, 4.  ])
>>> np.var(a, axis=1)
array([ 2.88888889, 11.55555556])
```

np.mean,np.median

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.mean(a)
5.0
>>> np.mean(a, axis=0)
array([6.5, 5.5, 3. ])
>>> np.mean(a, axis=1)
array([5.66666667, 4.33333333])
>>> np.median(a)
4.5
>>> np.median(a, axis=0)
array([6.5, 5.5, 3. ])
>>> np.median(a, axis=1)
array([5., 3.])
```

`np.percentile`

```
>>> a = np.array([[4,8,5], [9,3,1]])
>>> np.percentile(a, 65)
5.75
>>> np.percentile(a, 65, axis=0)
array([7.25, 6.25, 3.6 ])
>>> np.percentile(a, 65, axis=1)
array([5.9, 4.8])
```

12. Matrix Sorting

main function

function	effect	Remark
`np.argsort`	Sort by a column (row), return the index	
`ndarray.argsort`	Sort by a column (row), return the index	The calling object is `ndarray`
`np.lexsort`	always sort by column	
`np.flipud`	Flip the 2D array up and down by row (up/down)	

function	effect	Remark
`np.fliplr`	Flip the 2D array left and right by column (left/right)	

`np.argsort`

Sort according to the specified column (row), return the index array

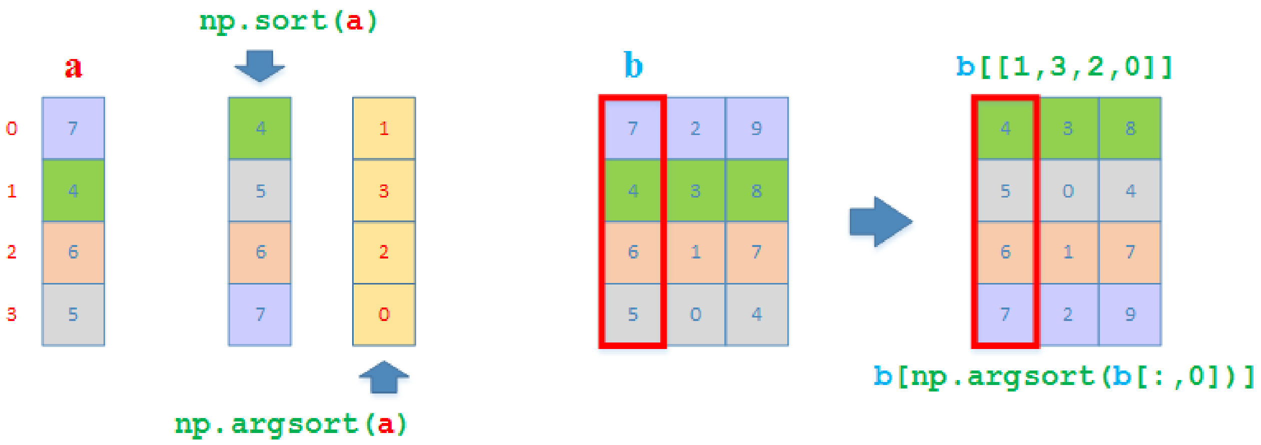
In fact `ndarray` , there are methods on an object that `argsort` can be called directly

- Sort a 1D array

```
>>> a = np.array([7, 4, 6, 5])
>>> a.argsort()
array([1, 3, 2, 0], dtype=int64)
>>> np.argsort(a)
array([1, 3, 2, 0], dtype=int64)
```

- Sort the entire 2D array by the first column

```
>>> b = np.array([[7,2,9], [4,3,8], [6,1,7], [5,0,4]])
>>> b[np.argsort(b[:,0])]
array([[4, 3, 8],
       [5, 0, 4],
       [6, 1, 7],
       [7, 2, 9]])
# 也可以直接在新的ndarray上调用argsort，然后借用索引数组取得排序后的2D array
>>> b[b[:,0].argsort()]
array([[4, 3, 8],
       [5, 0, 4],
       [6, 1, 7],
       [7, 2, 9]])
# 效果相当于按索引返回新2D array
>>> b[[1,3,2,0]]
array([[4, 3, 8],
       [5, 0, 4],
       [6, 1, 7],
       [7, 2, 9]])
```



np.flipud , np.fliplr

- `np.flipud` : Used to flip the 2D array **up and down**
- `np.fliplr` : Used to flip the 2D array **left and right**

```
>>> a = np.array([[3, 4, 5, 6], [2, 7, 0, -1], [1, 5, 3, 18], [2, 6, 1989, 3]])
>>> a
array([[   3,    4,    5,    6],
       [   2,    7,    0,   -1],
       [   1,    5,    3,   18],
       [   2,    6, 1989,    3]])
>>> np.flipud(a)
array([[   2,    6, 1989,    3],
       [   1,    5,    3,   18],
       [   2,    7,    0,   -1],
       [   3,    4,    5,    6]])
>>> np.fliplr(a)
array([[   6,    5,    4,    3],
       [  -1,    0,    7,    2],
       [  18,    3,    5,    1],
       [   3, 1989,    6,    2]])
```

np.lexsort

- `np.lexsort` Always sort each column as a whole (from bottom to top), and the returned index array is the index of the column

```
>>> a = np.array([[3, 4, 5, 6], [2, 7, 0, -1], [1, 5, 3, 18], [2, 6, 1989, 3]])
>>> a
array([[   3,    4,    5,    6],
       [   2,    7,    0,   -1],
       [   1,    5,    3,   18],
       [   2,    6, 1989,    3]])
# 返回列的索引
>>> idx = np.lexsort(a)
>>> idx
array([0, 3, 1, 2], dtype=int64)
# 按照列的索引数组取得新数组，得到排序后的结果
>>> a[:,idx]
array([[   3,    6,    4,    5],
       [   2,   -1,    7,    0],
       [   1,   18,    5,    3],
       [   2,    3,    6, 1989]])   # <--- 按照2 < 3 < 6 < 1989得到的排序顺序
```

- `pandas` There are more readable `sort_values` functions to sort by row/column in

```
>>> a = np.array([[3, 4, 5, 6], [2, 7, 0, -1], [1, 5, 3, 18], [2, 6, 1989, 3]])
>>> a
array([[   3,    4,    5,    6],
       [   2,    7,    0,   -1],
       [   1,    5,    3,   18],
       [   2,    6, 1989,    3]])
# 先按照第0列，再按照第2列进行排序，这里axis参数默认为0（按列）
>>> pd.DataFrame(a).sort_values(by=[0,2]).to_numpy()
array([[   1,    5,    3,   18],
       [   2,    7,    0,   -1],
       [   2,    6, 1989,    3],
       [   3,    4,    5,    6]])
# 先按照第0列，再按照第2列进行排序，这里显示指定axis参数为0（按列）
>>> pd.DataFrame(a).sort_values(by=[0, 2], axis=0).to_numpy()
array([[   1,    5,    3,   18],
       [   2,    7,    0,   -1],
       [   2,    6, 1989,    3],
       [   3,    4,    5,    6]])
# 先按照第1行，再按照第3行进行排序，这里显示指定axis参数为1（按行）
>>> pd.DataFrame(a).sort_values(by=[1, 3], axis=1).to_numpy()
array([[   6,    5,    3,    4],
       [  -1,    0,    2,    7],
       [  18,    3,    1,    5],
       [   3, 1989,    2,    6]])
```

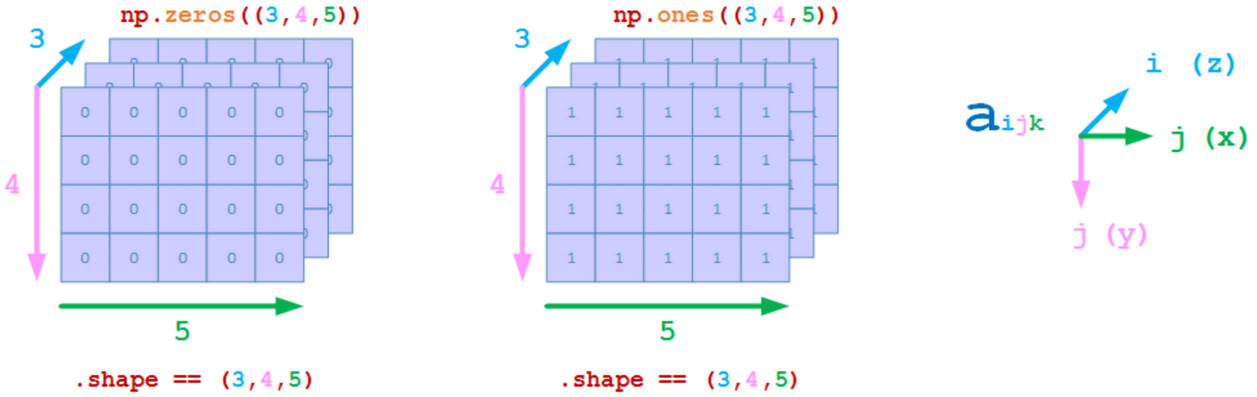
13. 3D and Above Matrix

function used

function	effect
`np.concatenate`	along `axis=0
`np.moveaxis`	The parameter is `(a, srcAxis, destAxis)`
`np.swapaxes`	
`np.einsum`	

Dimension order of 3D array in Numpy

In Numpy, the dimension order of a 3D array is , that is, the reverse order of the `(z, y, x)` normal one. `(x, y, z)`

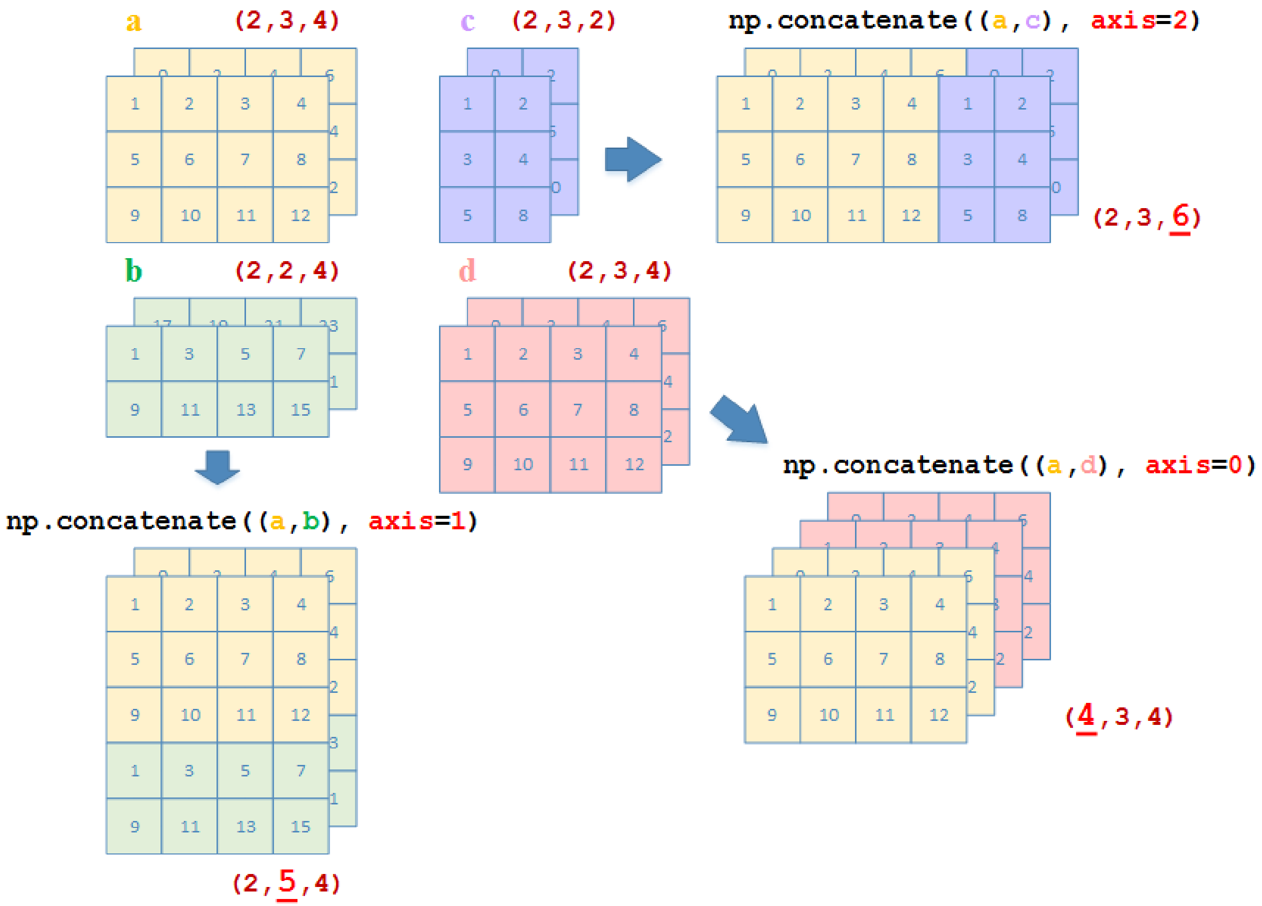


For example, to create a 3D array with `z` direction `3`, `y` direction `4`, `x` direction `5`

```
>>> np.zeros((3,4,5))
array([[[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]])
>>> np.ones((3,4,5))
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]])
# 注意，np.random.ran函数的参数不是一个元组
>>> np.random.rand(3,4,5)
array([[[0.06299431, 0.09575032, 0.2670843 , 0.51132979, 0.29522357],
        [0.99386711, 0.18979029, 0.88268506, 0.8844051 , 0.3979463 ],
        [0.52145756, 0.2953166 , 0.98018492, 0.15031576, 0.35522931],
        [0.01009925, 0.93094333, 0.23642368, 0.00796226, 0.2548972 ]],

       [[0.92253603, 0.5261462 , 0.07146182, 0.06805223, 0.67127133],
        [0.14054481, 0.27912101, 0.12922132, 0.08241845, 0.78589279],
        [0.90330273, 0.32816306, 0.41328599, 0.18650637, 0.58881924],
        [0.39545397, 0.47953933, 0.25072117, 0.2345669 , 0.14524772]],

       [[0.28491979, 0.55260934, 0.19610983, 0.86477694, 0.35238863],
        [0.85913433, 0.56783708, 0.76408683, 0.11193118, 0.36949155],
        [0.36358321, 0.51957791, 0.34842336, 0.79710159, 0.58843289],
        [0.74677968, 0.44109851, 0.35527215, 0.733685  , 0.40455108]]])
```

np.concatenate

The schematic diagram of using the `np.concatenate` connected 3D array is as follows


a (2,3,4)
c (2,3,2)
np.concatenate((a,c), axis=2)
(2,3,6)
b (2,2,4)
d (2,3,4)
np.concatenate((a,d), axis=0)
np.concatenate((a,b), axis=1)
(4,3,4)
(2,5,4)

Define the following 3D arrays, where

- a ： `a.shape = (2, 3, 4)`
- b ： `b.shape = (2, 2, 4)`
- c ： `c.shape = (2, 3, 2)`
- d ： `d.shape = (2, 3, 4)`

```
>>> a = np.array([ [[1,2,3,4], [5,6,7,8], [9,10,11,12]], [[0,2,4,6], [8,10,12,14], [
>>> a
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]],

       [[ 0,  2,  4,  6],
        [ 8, 10, 12, 14],
        [16, 18, 20, 22]]])
>>> b = np.array([ [[1,3,5,7], [9,11,13,15]], [[17,19,21,23], [25,27,29,31]] ])
>>> b
array([[[ 1,  3,  5,  7],
        [ 9, 11, 13, 15]],

       [[17, 19, 21, 23],
        [25, 27, 29, 31]]])
>>> c = np.array([ [[1,2], [3,4], [5,8]], [[0,2],[4,6],[8,10]] ])
>>> d = a - 10
>>> d
array([[[ -9,  -8,  -7,  -6],
        [ -5,  -4,  -3,  -2],
        [ -1,   0,   1,   2]],

       [[-10,  -8,  -6,  -4],
        [ -2,   0,   2,   4],
        [  6,   8,  10,  12]]])
>>> a.shape
(2, 3, 4)
>>> b.shape
(2, 2, 4)
>>> c.shape
(2, 3, 2)
>>> d.shape
(2, 3, 4)
```

Connect along z direction (`axis = 0`)

```
>>> np.concatenate((a,d), axis=0)
array([[[  1,   2,   3,   4],
        [  5,   6,   7,   8],
        [  9,  10,  11,  12]],

       [[  0,   2,   4,   6],
        [  8,  10,  12,  14],
        [ 16,  18,  20,  22]],

       [[ -9,  -8,  -7,  -6],
        [ -5,  -4,  -3,  -2],
        [ -1,   0,   1,   2]],

       [[-10,  -8,  -6,  -4],
        [ -2,   0,   2,   4],
        [  6,   8,  10,  12]]])
```

Connect along y direction (`axis = 1`)

```
>>> np.concatenate((a,b), axis=1)
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12],
        [ 1,  3,  5,  7],
        [ 9, 11, 13, 15]],

       [[ 0,  2,  4,  6],
        [ 8, 10, 12, 14],
        [16, 18, 20, 22],
        [17, 19, 21, 23],
        [25, 27, 29, 31]]])
```

Connect along `x` direction (`axis = 2`)

```
>>> np.concatenate((a,c), axis=2)
array([[[ 1,  2,  3,  4,  1,  2],
        [ 5,  6,  7,  8,  3,  4],
        [ 9, 10, 11, 12,  5,  8]],

       [[ 0,  2,  4,  6,  0,  2],
        [ 8, 10, 12, 14,  4,  6],
        [16, 18, 20, 22,  8, 10]]])
```

`np.hstack` , `np.vstack` , `np.dstack`

It is worth noting that, `np.hstack` , `np.vstack` , `np.dstack` treat the three dimensions of the input 3D array parameters as `(y, x, z)` , not in the usual sense `(z, y, x)`

14. Summary

Related functions

Functions	Effect	Notes
`np.zeros()`		
`np.empty()`		
`np.full()`		
`np.zeros_like()`		
`np.ones_like()`		
`np.empty_like()`		
`np.full_like()`		

Functions	Effect	Notes
`np.arange()`		
`np.linspace()`		
`np.random.randint()`		
`np.random.rand()`		
`np.random.uniform`		
`np.random.default_rng().intergers()`		
`np.random.default_rng().random()`		
`np.random.default_rng().uniform()`		
`np.where()`		
`np.clip()`		
`np.floor()`		
`np.ceil()`		
`np.round()`		
`np.sqrt()`		
`np.exp()`		
`np.log()`		
`np.dot()`	`arr @ arr`	
`np.cross()`		
`np.hypot()`		
`np.sort()`		
`arr.index()`	`arr` is `ndarry`	
`np.allclose()`		