



Table of Contents

- 1 数组乘法
 - 1.1 点积/内积/数量积/标量积
 - 1.2 叉积/外积/向量积
 - 1.3 张量积/外积
 - 1.4 矩阵乘法
 - 1.5 克罗内克积
 - 1.6 多矩阵乘法
- 2 基础概念
 - 2.1 范数
 - 2.2 行列式、迹
 - 2.3 特征值
- 3 矩阵运算
 - 3.1 矩阵求解
 - 3.2 逆矩阵
 - 3.3 矩阵分解
- 4 Einsum
- 5 Padding
- 6 卷积
- 7 掩码运算
 - 7.1 简介
 - 7.2 创建
 - 7.3 获取
 - 7.4 修改
 - 7.5 索引切片
 - 7.6 代数运算
 - 7.7 使用案例
- 8 小结
- 9 参考

```
In [151... import numpy as np
np.__version__
```

```
Out[151... '1.22.3'
```

文档阅读说明：

-  表示 Tip
-  表示注意事项

数组乘法

注意：不要太关注它们叫什么，看看它们做了什么。

点积/内积/数量积/标量积

点积

`np.dot` :

- 如果 a 和 b 是一维的, 就是内积 `np.inner`
- 如果 a 和 b 是二维的, 是矩阵乘法 `np.matmul` or `a @ b`
- 如果 a 或 b 任意一个是常量 `np.multiply` or `a * b`
- 如果 a 是 N 维, b 是一维 `sum product`
- 如果 a 是 N 维, b 是 M 维 `dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])`

`np.vdot` : 多维输入会被flatten后计算点积。另外在计算复数时与 `np.dot` 有所不同。

内积

`np.inner` : 一维数组向量的普通内积 (无复数共轭), 在更高维度上, 是最后一个轴上的sum product。

- 对一维数组, 就是元素乘积之和 `sum(a * b)`
- 如果有一个是标量, 那就是直接相乘
- 对多维数组, 等于 `np.tensordot(a, b, axes=(-1, -1))`, 对于某个具体的索引, 就是相乘后在最后一个维度上求和 `inner(a, b)[i0,...,ir-2,j0,...,js-2] = sum(a[i0,...,ir-2,:]*b[j0,...,js-2,:])`

数量积 - [维基百科, 自由的百科全书](#)

a和b都是一维:

```
In [912...] a = np.array([1, 2, 4])
            b = np.array([4, 5, 6])

In [913...] np.dot(a, b), 1*4 + 2*5 + 4*6, np.inner(a, b), sum(a * b)

Out[913...] (38, 38, 38, 38)

In [914...] np.vdot(a,b)

Out[914...] 38
```

a和b都是二维:

```
In [915...] rng = np.random.default_rng(42)
            a = rng.integers(0, 10, (3, 4))
            b = rng.integers(0, 10, (3, 4))

In [916...] np.inner(a, b)

Out[916...] array([[119,  71,  63],
                  [126,  52,  98],
                  [112,  86,  96]])
```

```
In [917... np.tensordot(a, b, axes=(-1, -1))
```

```
Out[917... array([[119, 71, 63],  
        [126, 52, 98],  
        [112, 86, 96]])
```

```
In [918... a @ b.T
```

```
Out[918... array([[119, 71, 63],  
        [126, 52, 98],  
        [112, 86, 96]])
```

```
In [919... np.matmul(a, b.T)
```

```
Out[919... array([[119, 71, 63],  
        [126, 52, 98],  
        [112, 86, 96]])
```

```
In [921... np.vdot(a, b), np.dot(a.flatten(), b.flatten())
```

```
Out[921... (267, 267)
```

a或b是常量:

```
In [458... a * 2
```

```
Out[458... array([[ 0, 14, 12, 8],  
        [ 8, 16, 0, 12],  
        [ 4, 0, 10, 18]])
```

```
In [461... np.dot(a, 2)
```

```
Out[461... array([[ 0, 14, 12, 8],  
        [ 8, 16, 0, 12],  
        [ 4, 0, 10, 18]])
```

```
In [459... np.multiply(a, 2)
```

```
Out[459... array([[ 0, 14, 12, 8],  
        [ 8, 16, 0, 12],  
        [ 4, 0, 10, 18]])
```

```
In [460... np.inner(a, 2)
```

```
Out[460... array([[ 0, 14, 12, 8],  
        [ 8, 16, 0, 12],  
        [ 4, 0, 10, 18]])
```

a是多维b是一维:

```
In [462... rng = np.random.default_rng(42)  
a = rng.integers(0, 10, (2, 3, 4))  
b = np.array([1, 2, 3, 4])
```

```
In [463... np.dot(a, b)
```

```
Out[463... array([[48, 44, 53],  
        [70, 47, 50]])
```

```
In [464... # 最后一个维度乘积和
np.inner(a, b)
```

```
Out[464... array([[48, 44, 53],
        [70, 47, 50]])
```

```
In [465... np.tensordot(a, b, axes=(-1, -1))
```

```
Out[465... array([[48, 44, 53],
        [70, 47, 50]])
```

```
In [466... np.sum(a*b, axis=-1)
```

```
Out[466... array([[48, 44, 53],
        [70, 47, 50]])
```

```
In [467... a @ b
```

```
Out[467... array([[48, 44, 53],
        [70, 47, 50]])
```

a是m维b是n维:

```
In [720... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 4, 3))
b = rng.integers(0, 10, (2, 3, 3))
```

```
In [721... # a 和 b 最后一个维度可以不一样，也就是最后一个维度是自由的
dab = np.dot(a, b)
dab.shape
```

```
Out[721... (2, 4, 2, 3)
```

```
In [487... # dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
dab[1,2,1,0], sum(a[1,2,:] * b[1,:,0])
```

```
Out[487... (102, 102)
```

```
In [483... # a 和 b 最后一个维度必须一样
iab = np.inner(a, b)
iab.shape
```

```
Out[483... (2, 4, 2, 3)
```

```
In [489... # inner(a, b)[i0,...,ir-2,j0,...,js-2] = sum(a[i0,...,ir-2,:]*b[j0,...,js-2,:])
iab[1,2,1,0], sum(a[1,2,:] * b[1,0,:])
```

```
Out[489... (72, 72)
```

```
In [442... np.alltrue(iab == np.tensordot(a, b, axes=(-1, -1)))
```

```
Out[442... True
```

```
In [449... np.alltrue(iab == dab), np.any(iab == dab)
```

```
Out[449... (False, False)
```

叉积/外积/向量积

$$\begin{aligned}\mathbf{u} \times \mathbf{v} &= \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix} \mathbf{i} - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix} \mathbf{j} + \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \mathbf{k} \\ &= (u_2 v_3 - u_3 v_2) \mathbf{i} - (u_1 v_3 - u_3 v_1) \mathbf{j} + (u_1 v_2 - u_2 v_1) \mathbf{k}\end{aligned}$$

只支持二维或三维，表示与u和v都垂直的向量。

[叉积 - 维基百科，自由的百科全书](#)

In [546...

```
# 二维
a = [1, 1]
b = [-1, 1]
```

In [551...

```
# 向量积在二维中不起作用，因为返回的向量在二维之外
# 长度就是面积（根号2*根号2）
np.cross(a, b)
```

Out[551...

```
array(2)
```

In [566...

```
# 行变多并不等于维度变多
rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 2))
b = rng.integers(0, 10, (2, 2))
a, b
```

Out[566...

```
(array([[0, 7],
        [6, 4]]),
 array([[4, 8],
        [0, 6]]))
```

In [567...

```
np.cross(a, b)
```

Out[567...

```
array([-28, 36])
```

In [568...

```
np.cross([0,7], [4,8]), np.cross([6, 4], [0, 6])
```

Out[568...

```
(array(-28), array(36))
```

In [589...

```
# 三维
a = [1, 2, 4]
b = [4, 5, 6]
```

In [590...

```
2*6-4*5, -(1*6-4*4), 1*5-2*4
```

Out[590...

```
(-8, 10, -3)
```

In [591...

```
# 与a和b都垂直的向量
np.cross(a, b)
```

Out[591...

```
array([-8, 10, -3])
```

还有几个关于维度的三参数，用于改变数组的定义（有点类似C和F Style）。

```
In [596... # 行变多并不等于维度变多
rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 3))
b = rng.integers(0, 10, (2, 3))
a, b
```

```
Out[596... (array([[0, 7, 6],
        [4, 4, 8]]),
        array([[0, 6, 2],
        [0, 5, 9]]))
```

```
In [600... np.cross(a, b)
```

```
Out[600... array([[ -22,   0,   0],
        [  -4, -36,  20]])
```

```
In [611... np.cross([0,7,6], [0,6,2]), np.cross([4,4,8], [0,5,9])
```

```
Out[611... (array([ -22,   0,   0]), array([  -4, -36,  20]))
```

```
In [627... np.cross(a, b, axisc=0)
```

```
Out[627... array([[ -22,  -4],
        [   0, -36],
        [   0,  20]])
```

```
In [628... a.T, b.T
```

```
Out[628... (array([[0, 4],
        [7, 4],
        [6, 8]]),
        array([[0, 0],
        [6, 5],
        [2, 9]]))
```

```
In [630... np.cross(a.T, b.T)
```

```
Out[630... array([ 0, 11, 38])
```

```
In [631... np.cross(a, b, axisa=0, axisb=0)
```

```
Out[631... array([ 0, 11, 38])
```

```
In [633... np.cross([0,4], [0,0]), np.cross([7,4],[6,5]), np.cross([6,8], [2,9])
```

```
Out[633... (array(0), array(11), array(38))
```

```
In [641... # 这个是实际计算时沿着的维度，和上面的不一样
np.cross(a, b, axis=0)
```

```
Out[641... array([ 0, 11, 38])
```

张量积/外积

- [张量积 - 维基百科，自由的百科全书](#)
- [外积 - 维基百科，自由的百科全书](#)

```
In [522... a = [1, 2, 4]
b = [4, 5, 6]
```

```
In [523... np.outer(a, b)
```

```
Out[523... array([[ 4,  5,  6],
        [ 8, 10, 12],
        [16, 20, 24]])
```

```
In [524... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 3, 4))
b = rng.integers(0, 10, (2, 3, 4))
```

```
In [525... np.alltrue(
    np.outer(a,b) == a.ravel().reshape(24, 1) @ b.ravel().reshape(1, 24)
)
```

```
Out[525... True
```

```
In [526... np.alltrue(np.outer(a, b) == np.tensordot(a.ravel(), b.ravel(), axes=((), ())))
```

```
Out[526... True
```

矩阵乘法

`np.dot` 上面已经提过了，其实还有个 `np.matmul` 和它很类似，但二者是有一些区别的。

- `dot` 不是通函数，而 `matmul` 是通函数，也就意味这有通函数的一些通用参数
- `matmul` 不支持向量和数字相乘
- `matmul` 矩阵（好像元素一样）堆叠在一起广播

关于 `np.matmul`：

- 如果都是二维，就是常规的矩阵乘法
- 如果任意个是多维（>2），则把它当成驻留在最后两个索引中的矩阵堆栈，并相应地广播
- 如果第一个是一维，则通过在其维度前面加上 1 来将其提升为矩阵，矩阵乘法后删除前面附加的 1
- 如果第二个是一维，则在维度上append 1，矩阵乘法后再删除后面附加的 1

二维的情况：

```
In [652... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 3))
b = rng.integers(0, 10, (3, 2))
a, b
```

```
Out[652... (array([[0, 7, 6],
        [4, 4, 8]]),
array([[0, 6],
        [2, 0],
        [5, 9]]))
```

```
In [653... np.matmul(a, b)
```

```
Out[653... array([[44, 54],  
        [48, 96]])
```

其中一个是一维的情况:

```
In [655... rng = np.random.default_rng(42)  
a = rng.integers(0, 10, (2, 3))  
b = rng.integers(0, 10, (3,))  
a, b
```

```
Out[655... (array([[0, 7, 6],  
        [4, 4, 8]]),  
array([0, 6, 2]))
```

```
In [680... np.matmul(a, b), (a @ np.stack((b, [1, 1, 1]), axis=1))[:,0]
```

```
Out[680... (array([54, 40]), array([54, 40]))
```

```
In [690... np.matmul(b, a.T), (np.stack([1, 1, 1], b), axis=0) @ a.T[1,:]
```

```
Out[690... (array([54, 40]), array([54, 40]))
```

任意一个是多维:

```
In [723... rng = np.random.default_rng(42)  
a = rng.integers(0, 10, (2, 3, 4, 5))  
b = rng.integers(0, 10, (2, 3, 5, 9))
```

```
In [712... np.matmul(a, b).shape
```

```
Out[712... (2, 3, 4, 9)
```

```
In [725... (a @ b).shape
```

```
Out[725... (2, 3, 4, 9)
```

```
In [713... np.dot(a, b).shape
```

```
Out[713... (2, 3, 4, 2, 3, 9)
```

```
In [702... np.alltrue(np.dot(a, b) == np.matmul(a, b))
```

```
Out[702... True
```

看个简单点的例子:

```
In [728... rng = np.random.default_rng(42)  
a = rng.integers(0, 10, (2, 3, 2))  
b = rng.integers(0, 10, (2, 2, 2))
```

```
In [767... # 2x3x2  
np.matmul(a, b)
```



```
Out[767... array([[49, 49],
        [70, 70],
        [84, 84]],

        [[48, 24],
        [10,  2],
        [97, 41]]])
```

```
In [739... a[0,:,:] @ b[0,:,:]
```

```
Out[739... array([[49, 49],
        [70, 70],
        [84, 84]])
```

```
In [745... a[1,:,:] @ b[1,:,:]
```

```
Out[745... array([[48, 24],
        [10,  2],
        [97, 41]])
```

```
In [761... # 2x3x2
np.matmul(a[0,:,:], b)
```

```
Out[761... array([[49, 49],
        [70, 70],
        [84, 84]],

        [[56, 28],
        [62, 22],
        [84, 36]]])
```

```
In [763... # 2x3x2
np.matmul(a[1,:,:], b)
```

```
Out[763... array([[42, 42],
        [14, 14],
        [98, 98]],

        [[48, 24],
        [10,  2],
        [97, 41]]])
```

看看dot是怎样表现的:

```
In [756... # 2x3x2x2
np.dot(a,b)
```

```
Out[756... array([[[[49, 49],
          [56, 28]],

        [[70, 70],
          [62, 22]],

        [[84, 84],
          [84, 36]]],

      [[[42, 42],
          [48, 24]],

        [[14, 14],
          [10,  2]],

        [[98, 98],
          [97, 41]]]])
```

```
In [758... # 3x2x2
np.dot(a[0,:,:], b[:,:,:])
```

```
Out[758... array([[[49, 49],
          [56, 28]],

        [[70, 70],
          [62, 22]],

        [[84, 84],
          [84, 36]]])
```

```
In [765... # 3x2x2
np.dot(a[1,:,:], b[:,:,:])
```

```
Out[765... array([[[42, 42],
          [48, 24]],

        [[14, 14],
          [10,  2]],

        [[98, 98],
          [97, 41]]])
```

克罗内克积

维基百科：是两个任意大小的矩阵间的运算，表示为 \otimes 。克罗内克积是外积从向量到矩阵的推广，也是张量积在标准基下的矩阵表示。

[克罗内克积 - 维基百科，自由的百科全书](#)

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}.$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 & 1 \cdot 3 & 2 \cdot 0 & 2 \cdot 3 \\ 1 \cdot 2 & 1 \cdot 1 & 2 \cdot 2 & 2 \cdot 1 \\ 3 \cdot 0 & 3 \cdot 3 & 1 \cdot 0 & 1 \cdot 3 \\ 3 \cdot 2 & 3 \cdot 1 & 1 \cdot 2 & 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 0 & 6 \\ 2 & 1 & 4 & 2 \\ 0 & 9 & 0 & 3 \\ 6 & 3 & 2 & 1 \end{bmatrix}.$$

```
In [932...] np.kron([1,2,3], [1, 10, 100])
```

```
Out[932...] array([ 1, 10, 100,  2, 20, 200,  3, 30, 300])
```

```
In [933...] np.kron([1, 10, 100], [1,2,3])
```

```
Out[933...] array([ 1,  2,  3, 10, 20, 30, 100, 200, 300])
```

```
In [934...] a = np.array([[1,2],[3,1]])  
b = np.array([[0,3],[2,1]])
```

```
In [935...] np.kron(a,b)
```

```
Out[935...] array([[0, 3, 0, 6],  
                [2, 1, 4, 2],  
                [0, 9, 0, 3],  
                [6, 3, 2, 1]])
```

```
In [949...] a = np.ones((2,5,2,5))  
b = np.ones((2,3,4))  
np.kron(a,b).shape
```

```
Out[949...] (2, 10, 6, 20)
```

多矩阵乘法

`linalg.multi_dot` 链式调用 `np.dot`，自动选择最快的顺序。

- 如果第一个数组是一维，则被当做行向量
- 如果最后一个数组是一维，则被当做列向量
- 如果输入的向量超过两个，则其他向量必须是二维

```
In [962...] a = np.ones((2, 4))  
b = np.ones((4, 3))  
c = np.ones((3, 5))
```

```
In [963...] np.linalg.multi_dot((a,b,c)).shape
```

```
Out[963...] (2, 5)
```

不同的顺序性能不同，比如：

`A_{10x100}`, `B_{100x5}`, `C_{5x50}`

`cost((AB)C) = 10*100*5 + 10*5*50 = 5000 + 2500 = 7500` `cost(A(BC)) = 10*100*50 + 100*5*50 = 50000 + 25000 = 75000`

```
In [964... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (10, 100))
b = rng.integers(0, 10, (100, 5))
c = rng.integers(0, 10, (5, 50))
```

```
In [970... %timeit a.dot(b).dot(c).shape
```

8.74 μ s \pm 745 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

```
In [971... %timeit a.dot(b.dot(c)).shape
```

66 μ s \pm 4.96 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

```
In [973... %timeit np.linalg.multi_dot((a, b, c)).shape
```

13.6 μ s \pm 335 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

如果首尾是一维:

```
In [983... a = np.ones((3))
b = np.ones((3, 5))
c = np.ones((5, 8))
d = np.ones(8)
```

```
In [984... a.dot(b).dot(c).shape
```

```
Out[984... (8,)
```

```
In [989... # a=1x3
np.linalg.multi_dot((a, b, c)).shape
```

```
Out[989... (8,)
```

```
In [995... # d=8x1
np.linalg.multi_dot((b, c, d)).shape
```

```
Out[995... (3,)
```

```
In [997... b.dot(c).dot(d).shape
```

```
Out[997... (3,)
```

基础概念

介绍线性代数几个常用的API，不涉及数学知识。

```
In [107... from numpy import linalg as LA
```

范数

共包括:

=====	=====	=====
ord	norm for matrices	norm for vectors
=====	=====	=====

None	Frobenius norm	2-norm
'fro'	Frobenius norm	--
'nuc'	nuclear norm	--
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	--	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	--	sum(abs(x)**ord)**(1./ord)
=====	=====	=====

- [Matrix norm - Wikipedia](#)

```
In [107... a = np.arange(6).reshape(2, 3)
a
```

```
Out[107... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [108... LA.norm(a)
```

```
Out[108... 7.416198487095663
```

```
In [109... # F范数
LA.norm(a, "fro"), np.sqrt(np.sum(a**2))
```

```
Out[109... (7.416198487095663, 7.416198487095663)
```

```
In [109... # 核范数
LA.norm(a, "nuc"), np.sum(LA.svd(a)[1])
```

```
Out[109... (8.348469228349535, 8.348469228349535)
```

```
In [110... # inf
LA.norm(a, np.inf), np.max(np.sum(abs(a), axis=1))
```

```
Out[110... (12.0, 12)
```

```
In [110... # -inf
LA.norm(a, -np.inf), np.min(np.sum(abs(a), axis=1))
```

```
Out[110... (3.0, 3)
```

```
In [111... # 1
LA.norm(a, 1), np.max(np.sum(abs(a), axis=0))
```

```
Out[111... (7.0, 7)
```

```
In [111... # -1
LA.norm(a, -1), np.min(np.sum(abs(a), axis=0))
```

```
Out[111... (3.0, 3)
```

```
In [112... # 2
LA.norm(a, 2), np.max(LA.svd(a)[1])

Out[112... (7.3484692283495345, 7.3484692283495345)

In [113... # -2
LA.norm(a, -2), np.min(LA.svd(a)[1])

Out[113... (0.9999999999999998, 0.9999999999999998)
```

行列式、迹

```
In [119... a = np.array([[1,5],[3,4]])
a

Out[119... array([[1, 5],
          [3, 4]])

In [119... LA.det(a), 1*4-5*3

Out[119... (-11.000000000000002, -11)

In [119... a[0,0] + a[1,1]

Out[119... 5

In [116... # 二维
np.trace(a)

Out[116... 5

In [118... a = np.arange(8).reshape((2,2,2))
a

Out[118... array([[[0, 1],
          [2, 3]],

          [[4, 5],
          [6, 7]]])

In [119... a[0,0] + a[1,1]

Out[119... array([6, 8])

In [119... np.trace(a)

Out[119... array([6, 8])
```

特征值

`eig` 计算一个方阵的特征值和右特征向量，`eigvals` 与之的区别是不返回特征向量。

```
In [106... a = np.diag([1,2,3])
w, v = LA.eig(a)
```

In [106...

```
w
```

Out[106...

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

In [106...

```
v
```

Out[106...

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

In [106...

```
LA.eigvals(a)
```

Out[106...

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

In [106...

```
a @ v == w * v
```

Out[106...

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

In [106...

```
rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 3))
a
```

Out[106...

```
array([[0, 7, 6],
       [4, 4, 8],
       [0, 6, 2]])
```

In [106...

```
w, v = LA.eig(a)
```

In [106...

```
np.allclose(a @ v, w * v)
```

Out[106...

```
True
```

In [106...

```
LA.eigvals(a) == w
```

Out[106...

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

`eigh` 计算埃尔米特矩阵或实对称矩阵的特征值和特征向量，`eigvalsh` 与之的区别是后者不返回特征向量。

- [埃尔米特矩阵 - 维基百科，自由的百科全书](#)
- [對稱矩陣 - 维基百科，自由的百科全书](#)

以实对称矩阵为例。

In [107...

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

In [107...

```
LA.eigh(a)
```

```
Out[107...] (array([-3.07730361,  3.84139016, 10.23591345]),
             array([[ -0.65271955,  0.74655097,  0.12891408],
                    [ 0.55145509,  0.58486128, -0.59483995],
                    [ 0.5194752 ,  0.31717334,  0.79343972]]))
```

```
In [107...] LA.eigvalsh(a)
```

```
Out[107...] array([-3.07730361,  3.84139016, 10.23591345])
```

```
In [105...] LA.eig(a)
```

```
Out[105...] (array([-3.07730361,  3.84139016, 10.23591345]),
             array([[ -0.65271955,  0.74655097,  0.12891408],
                    [ 0.55145509,  0.58486128, -0.59483995],
                    [ 0.5194752 ,  0.31717334,  0.79343972]]))
```

矩阵运算

矩阵求解

`solve` 可直接求解：

```
In [134...] x = np.array([[1, 2], [3, 5]])
             y = np.array([1, 2])
             w = LA.solve(x, y)
```

```
In [125...] np.allclose(x.dot(w), y)
```

```
Out[125...] True
```

`tensorsolve` 更加通用一些：

```
In [125...] LA.tensorsolve(x, y)
```

```
Out[125...] array([-1.,  1.])
```

```
In [125...] rng = np.random.default_rng(42)
             x = rng.integers(0, 10, (6, 4, 2, 3, 4))
             y = rng.integers(0, 10, (6, 4))
```

```
In [125...] LA.tensorsolve(x, y).shape
```

```
Out[125...] (2, 3, 4)
```

```
In [125...] LA.solve(x,y)
```



```

-----
LinAlgError                                Traceback (most recent call last)
<ipython-input-1256-9be96e929103> in <module>
----> 1 LA.solve(x,y)

/usr/local/lib/python3.8/site-packages/numpy/core/overrides.py in solve(*args, **
kwargs)

/usr/local/lib/python3.8/site-packages/numpy/linalg/linalg.py in solve(a, b)
    378     a, _ = _makearray(a)
    379     _assert_stacked_2d(a)
--> 380     _assert_stacked_square(a)
    381     b, wrap = _makearray(b)
    382     t, result_t = _commonType(a, b)

/usr/local/lib/python3.8/site-packages/numpy/linalg/linalg.py in _assert_stacked_
square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array must be squ
are')
    204
    205 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square

```

可以使用最小二乘法近似求解：

```

In [126... from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

```

```

In [126... data = load_iris()

```

```

In [126... x = data["data"]
y = data["target"]

```

```

In [127... x_train, x_test, y_train, y_test = \
train_test_split(x, y, test_size=0.2)

```

```

In [131... w = LA.lstsq(x_train, y_train, rcond=None)[0]

```

```

In [132... # 精准率
np.sum(
    np.abs(x_test.dot(w)).round()==y_test
)/len(y_test)

```

```

Out[132... 1.0

```

逆矩阵

inv 可用于求矩阵逆：

```

In [133... a = np.arange(1, 5).reshape(2, 2)
a

```

```
Out[133... array([[1, 2],
        [3, 4]])
```

```
In [134... inva = LA.inv(a)
```

```
In [134... np.allclose(inva.dot(a), np.eye(2))
```

```
Out[134... True
```

使用奇异值分解计算矩阵伪逆。

```
In [135... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (2, 3))
a
```

```
Out[135... array([[0, 7, 6],
        [4, 4, 8]])
```

```
In [135... inva = LA.pinv(a)
inva
```

```
Out[135... array([[ -0.12751678,  0.14261745],
        [ 0.15436242, -0.08053691],
        [-0.01342282,  0.09395973]])
```

```
In [136... np.allclose(a.dot(inva), np.eye(2))
```

```
Out[136... True
```

```
In [136... np.allclose(a.dot(inva).dot(a), a)
```

```
Out[136... True
```

```
In [137... np.allclose(a.dot(inva.dot(a)), a)
```

```
Out[137... True
```

```
In [136... np.allclose(inva.dot(a.dot(inva)), inva)
```

```
Out[136... True
```

`tensorinv` 适用于高维数组求逆：

```
In [140... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (4, 6, 8, 3))
```

```
In [140... ainv = np.linalg.tensorinv(a, ind=2)
```

```
In [140... np.tensordot(a, ainv).shape
```

```
Out[140... (4, 6, 4, 6)
```

```
In [140... eye = np.eye(4*6)
eye.shape = (4, 6, 4, 6)
```

```
In [141... np.allclose(np.tensordot(a, ainv), eye)
```

Out[141... True

矩阵分解

`cholesky` 分解是把一个对称正定的矩阵表示成一个下三角矩阵L和其转置的乘积的分解。要求所有的特征值必须大于零。

[科列斯基分解 - 维基百科，自由的百科全书](#)

```
In [151... a = np.array([
    [4, 12, -16],
    [12, 37, -43],
    [-16, -43, 98]
])
```

```
In [151... ca = LA.cholesky(a)
ca
```

```
Out[151... array([[ 2.,  0.,  0.],
        [ 6.,  1.,  0.],
        [-8.,  5.,  3.]])
```

```
In [151... np.array_equal(ca.dot(ca.T), a)
```

Out[151... True

`qr` 分解将矩阵分解成一个正交矩阵和一个上三角矩阵的积。

[QR 分解 - 维基百科，自由的百科全书](#)

```
In [141... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (4, 5))
a
```

```
Out[141... array([[0, 7, 6, 4, 4],
        [8, 0, 6, 2, 0],
        [5, 9, 7, 7, 7],
        [7, 5, 1, 8, 4]])
```

```
In [141... # q是正交矩阵，r是上三角矩阵
q, r = LA.qr(a)
```

```
In [142... q.shape, r.shape
```

```
Out[142... ((4, 4), (4, 5))
```

```
In [141... np.allclose(q.dot(r), a)
```

Out[141... True

```
In [142... # 转置=逆
np.allclose(LA.inv(q), q.T)
```

Out[142... True

svd 分解将矩阵分解为一个酉矩阵 U 、一个非负实对角矩阵 Σ 和一个共轭转置矩阵 V^* 的乘积。 Σ 对角线上的元素为奇异值。

$$M = U\Sigma V^*$$

[奇异值分解 - 维基百科，自由的百科全书](#)

In [148...

```
a
```

Out[148...

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

In [148...

```
u, s, vh = LA.svd(a)
```

In [149...

```
u.shape, s.shape, vh.shape
```

Out[149...

```
((4, 4), (4,), (5, 5))
```

In [150...

```
xgm = np.insert(np.diag(s), s.shape[0], 0, axis=1)
xgm.shape
```

Out[150...

```
(4, 5)
```

In [150...

```
np.allclose(u.dot(xgm).dot(vh), a)
```

Out[150...

```
True
```

Einsum

使用 `einsum` 可以让很多常见的数组运算以简洁的方式表示。

下标字符串是一个逗号分隔的下标标签列表，每个标签指的是相应操作的一个维度。

- 标签重复时会被求和 `np.einsum("i,i", a, b)`，等价于 `np.inner(a,b)`。
- 如果只出现一次 `np.einsum("i", a)`，返回自己的view。
- 重复下标标签取对角线 `np.einsum("ii", a)`，等价于 `np.trace(a)`

在隐式模式下，下标很重要，输出的轴会按字母重新排序。比如：

- `np.einsum("ij",a)` 不会影响二维数组，但 `np.einsum("ji",a)` 则返回转置。
- `np.einsum("ij,jk", a, b)` 会返回矩阵乘法，而 `np.einsum("ij,jh", a, b)` 则返回乘法的转置。

在显式模式下，可以通过指定输出下标标签直接控制输出。此时需要 `->` 标识符。

- `np.einsum("i->", a)` 类似于 `np.sum(a, axis=-1)`。
- `np.einsum("ii->i", a)` 类似于 `np.diag(a)`。
- `np.einsum("ij,jh->ih", a, b)` 返回乘法结果，而不是结果的转置。

`einsum` 默认不支持广播，要启用需使用（在左侧添加）省略号。

- `np.einsum("...ii->...i", a)`
- 跟踪第一个和最后一个维度: `np.einsum("i...i", a)`
- 用最左边的轴矩阵乘法: `np.einsum("ij...,jk...->ik...", a, b)`

```
In [182... a = np.arange(3)
b = np.arange(9).reshape(3, 3)
c = np.arange(6).reshape(2, 3)
d = np.arange(6).reshape(3, 2)
e = np.arange(60).reshape(3, 4, 5)
f = np.arange(24).reshape(4, 3, 2)
g = np.arange(30).reshape(3, 5, 2)
```

一个标签:

```
In [126... # 返回自己的view
np.einsum("i", a)
```

```
Out[126... array([0, 1, 2])
```

```
In [127... # 广播
np.einsum("...i", b)
```

```
Out[127... array([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

```
In [128... # 内积
np.einsum("i,i", a,a), np.inner(a, a)
```

```
Out[128... (5, 5)
```

```
In [129... # 迹
np.einsum("ii", b), np.trace(b)
```

```
Out[129... (12, 12)
```

隐式模式:

```
In [130... # 不影响结果
np.einsum("ij", c)
```

```
Out[130... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [131... # 返回转置
np.einsum("ji", c)
```

```
Out[131... array([[0, 3],
        [1, 4],
        [2, 5]])
```

```
In [155... # 外积
np.einsum("i,j", a,a), np.outer(a, a)
```

```
Out[155...] (array([[0, 0, 0],
        [0, 1, 2],
        [0, 2, 4]]),
array([[0, 0, 0],
        [0, 1, 2],
        [0, 2, 4]]))
```

```
In [132...] # 矩阵乘法
np.einsum("ij,jk", c, d)
```

```
Out[132...] array([[10, 13],
        [28, 40]])
```

```
In [133...] # h在i之前, 返回转置
np.einsum("ij,jh", c, d)
```

```
Out[133...] array([[10, 28],
        [13, 40]])
```

```
In [136...] np.einsum("ij,j", c,a), np.dot(c,a)
```

```
Out[136...] (array([ 5, 14]), array([ 5, 14]))
```

显式模式:

```
In [80]: # 求和
np.einsum("i->", a)
```

```
Out[80]: 10
```

```
In [104...] # np.sum(b, axis=1)
np.einsum("ij->i", b), np.sum(b, axis=1)
```

```
Out[104...] (array([ 3, 12, 21]), array([ 3, 12, 21]))
```

```
In [107...] # 求和
np.einsum("ij->i", c), np.sum(c, axis=1)
```

```
Out[107...] (array([ 3, 12]), array([ 3, 12]))
```

```
In [106...] # np.sum(d, axis=0)
np.einsum("ij->j", d), np.sum(d, axis=0)
```

```
Out[106...] (array([6, 9]), array([6, 9]))
```

```
In [294...] # 元素相乘
np.einsum("i,i->i", a, a), a*a
```

```
Out[294...] (array([0, 1, 4]), array([0, 1, 4]))
```

```
In [306...] np.einsum("i,j->", a, a), np.outer(a,a).sum()
```

```
Out[306...] (9, 9)
```

```
In [111...] # 显式, 转置
np.einsum("ij->ji", c)
```

```
Out[111...] array([[0, 3],
                [1, 4],
                [2, 5]])
```

```
In [91]: # 返回diag
np.einsum("ii->i", b)
```

```
Out[91]: array([0, 4, 8])
```

```
In [389...] # 元素相乘
np.einsum("ij,ij->ij", c, c)
```

```
Out[389...] array([[ 0,  1,  4],
                  [ 9, 16, 25]])
```

```
In [282...] # 矩阵乘法, 因为显式指定, 不会转置
np.einsum("ij,jh->ih", c,d)
```

```
Out[282...] array([[10, 13],
                  [28, 40]])
```

```
In [284...] # 求和
np.einsum("ij,jh->i", c, d)
```

```
Out[284...] array([23, 68])
```

```
In [285...] np.einsum("ij,jh->h", c, d)
```

```
Out[285...] array([38, 53])
```

```
In [192...] # 多维
# e=(3,4,5), f=(4,3,2)
np.einsum("ijk,jil->kl", e, f).shape
```

```
Out[192...] (5, 2)
```

```
In [189...] np.einsum("ijk,ikh->ijh", e, g).shape
```

```
Out[189...] (3, 4, 2)
```

```
In [191...] np.array_equal(
    np.einsum("ijk,ikh->ijh", e, g),
    np.matmul(e,g)
)
```

```
Out[191...] True
```

```
In [317...] np.einsum("ij,kl -> ijkl", c, d).shape
```

```
Out[317...] (2, 3, 3, 2)
```

```
In [373...] np.array_equal(
    np.einsum("ij,kl->ijkl", c, d),
    c[:, :, None, None] * d
)
```

```
Out[373...] True
```

```
In [385... np.einsum("ij,jk->ijk", c, d).shape
```

```
Out[385... (2, 3, 2)
```

```
In [384... np.array_equal(np.einsum("ij,jk->ijk", c, d), c[:, :, None]*d)
```

```
Out[384... True
```

广播:

```
In [150... # 矩阵乘法
# c=(2,3), a=(3,)
np.einsum("...j,j", c, a)
```

```
Out[150... array([ 5, 14])
```

```
In [149... # a=(3,), c=(2,3)
np.einsum("j,...j", a, c)
```

```
Out[149... array([ 5, 14])
```

```
In [147... # a=(3,), d=(3, 2)
np.einsum("j,j...", a, d)
```

```
Out[147... array([10, 13])
```

```
In [279... # 延续上面的乘法, 显式
np.einsum("ij,j...->i...", c, d)
```

```
Out[279... array([[10, 13],
        [28, 40]])
```

```
In [280... # 隐式
np.einsum("...j,ji", c, d)
```

```
Out[280... array([[10, 13],
        [28, 40]])
```

Padding

Padding操作, 参数如下:

- 数组
- pad_width: 序列、整数或数组, 每个轴边缘扩展的数量。
- 模式: 默认constant。还包括: edge, linear_ramp, maximum, mean, median, minimum, reflect, symmetric, wrap, empty。
- stat_length: 序列、整数或数组, 模式是 maximum, minimum, mean, median \时, 用来计算每个轴边缘的值, 默认None。
- constant_values: 序列或标量, padding的值, 默认0。
- end_values: 虚列或标量, 模式是 linear_ramp 时使用, 用于结束值, 经形成填充数组的边缘, 默认0。
- reflect_type: 模式是 reflect 和 symmetric 时使用, 默认使用 even 风格, 边缘值周围不改变反射, odd 模式, 数组的拓展部分是通过从边缘值的两倍中减去反射

值来创建的。

首先看pad_width参数:

```
In [798... # tuple
np.pad(
    [1,2,3,4,5],
    (2,3), # 等于((2,3),), ((2,3))
    "constant",
    constant_values=(4, 6)
)
```

```
Out[798... array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])
```

```
In [805... np.pad(
    [1,2,3,4,5],
    2, # 等于(2), (2, )
    "constant",
    constant_values=(4, 6)
)
```

```
Out[805... array([4, 4, 1, 2, 3, 4, 5, 6, 6])
```

```
In [809... # 分别左上, 右下
np.pad(
    [[1,2,3],[4,5,6]],
    (1, 2)
)
```

```
Out[809... array([[0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 0, 0],
        [0, 4, 5, 6, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0]])
```

```
In [813... # 行 (1,2)
# 列 (2,1)
np.pad(
    [[1,2,3],[4,5,6]],
    ((1, 2), (2, 1))
)
```

```
Out[813... array([[0, 0, 0, 0, 0, 0],
        [0, 0, 1, 2, 3, 0],
        [0, 0, 4, 5, 6, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0]])
```

```
In [815... np.pad(
    [[1,2,3],[4,5,6]],
    1
)
```

```
Out[815... array([[0, 0, 0, 0, 0],
        [0, 1, 2, 3, 0],
        [0, 4, 5, 6, 0],
        [0, 0, 0, 0, 0]])
```

接下来看下不同模式，顺带了解不同模式对应的额外参数。简单起见，`pad_width` 我们统一使用整数。

```
In [816... a = np.arange(1, 7).reshape(3, 2)
a
```

```
Out[816... array([[1, 2],
        [3, 4],
        [5, 6]])
```

```
In [817... # edge
np.pad(a, 1, "edge")
```

```
Out[817... array([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 3, 4, 4],
        [5, 5, 6, 6],
        [5, 5, 6, 6]])
```

```
In [821... # linear_ramp
# 需要额外参数: end_values, 默认0
np.pad(a, 1, "linear_ramp")
```

```
Out[821... array([[0, 0, 0, 0],
        [0, 1, 2, 0],
        [0, 3, 4, 0],
        [0, 5, 6, 0],
        [0, 0, 0, 0]])
```

```
In [824... np.pad(a, 1, "linear_ramp", end_values=(1, ))
```

```
Out[824... array([[1, 1, 1, 1],
        [1, 1, 2, 1],
        [1, 3, 4, 1],
        [1, 5, 6, 1],
        [1, 1, 1, 1]])
```

```
In [825... np.pad(a, 1, "linear_ramp", end_values=(1, 2), )
```

```
Out[825... array([[1, 1, 1, 2],
        [1, 1, 2, 2],
        [1, 3, 4, 2],
        [1, 5, 6, 2],
        [1, 2, 2, 2]])
```

```
In [827... # 行 (1,2)
# 列 (3,4)
np.pad(a, 1, "linear_ramp", end_values=((1, 2), (3, 4)))
```

```
Out[827... array([[3, 1, 1, 4],
        [3, 1, 2, 4],
        [3, 3, 4, 4],
        [3, 5, 6, 4],
        [3, 2, 2, 4]])
```

```
In [829... # 和这个等价
np.pad(
    a,
    1,
```

```
"constant",
constant_values=((1, 2), (3,4))
)
```

```
Out[829... array([[3, 1, 1, 4],
        [3, 1, 2, 4],
        [3, 3, 4, 4],
        [3, 5, 6, 4],
        [3, 2, 2, 4]])
```

```
In [832... # maximum, minmum, mean, median
# 需要额外参数stat_length, 默认None, 使用该轴所有值
np.pad(a, 1, "maximum")
```

```
Out[832... array([[6, 5, 6, 6],
        [2, 1, 2, 2],
        [4, 3, 4, 4],
        [6, 5, 6, 6],
        [6, 5, 6, 6]])
```

```
In [837... # 只取2个
np.pad(a, 1, "maximum", stat_length=2)
```

```
Out[837... array([[4, 3, 4, 4],
        [2, 1, 2, 2],
        [4, 3, 4, 4],
        [6, 5, 6, 6],
        [6, 5, 6, 6]])
```

```
In [844... # 分别取, 左上2, 右下1
np.pad(a, 1, "maximum", stat_length=((2, 1), ))
```

```
Out[844... array([[4, 3, 4, 4],
        [2, 1, 2, 2],
        [4, 3, 4, 4],
        [6, 5, 6, 6],
        [6, 5, 6, 6]])
```

```
In [850... # 各自分别指定
# 行 (2,1)
# 列 (1,2)
np.pad(a, 1, "maximum", stat_length=((2, 1), (1, 2)))
```

```
Out[850... array([[3, 3, 4, 4],
        [1, 1, 2, 2],
        [3, 3, 4, 4],
        [5, 5, 6, 6],
        [5, 5, 6, 6]])
```

```
In [858... b = a.astype(np.float16)
```

```
In [860... # mean和median类似
# 行 (2,1)
# 列 (1,2)
np.pad(b, 1, "mean", stat_length=((2, 1), (1, 2)))
```

```
Out[860...] array([[2. , 2. , 3. , 2.5],
        [1. , 1. , 2. , 1.5],
        [3. , 3. , 4. , 3.5],
        [5. , 5. , 6. , 5.5],
        [5. , 5. , 6. , 5.5]], dtype=float16)
```

```
In [870...] a = [1,2,3,4,5]
```

```
In [883...] # reflect, symmetric
# 需要额外参数reflect_type, 默认even
# 首末是对称轴
np.pad(a, 3, "reflect")
```

```
Out[883...] array([4, 3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
In [884...] # 首末是对称轴
np.pad(a, 3, "reflect", reflect_type="odd")
```

```
Out[884...] array([-2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
In [885...] # 边缘是对称轴
np.pad(a, 3, "symmetric")
```

```
Out[885...] array([3, 2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
In [886...] # 边缘是对称轴
np.pad(a, 3, "symmetric", reflect_type="odd")
```

```
Out[886...] array([-1,  0,  1,  1,  2,  3,  4,  5,  5,  6,  7])
```

```
In [889...] # wrap
# 首尾互换
np.pad(a, 2, "wrap")
```

```
Out[889...] array([4, 5, 1, 2, 3, 4, 5, 1, 2])
```

```
In [900...] # empty
# 未定义值扩展
np.pad(a, 1, "empty")
```

```
Out[900...] array([          0,          1,          2,          3,
                   4,          5, 123145302310976])
```

卷积

卷积函数（一维）遵循以下规则：

$$(a * v)[n] = \sum_{m=-\infty}^{\infty} a[m]v[n - m]$$

```
In [782...] np.convolve([1, 2, 3], [0, 1, 0.5], "valid"), 0.5*1+1*2+0*3
```

```
Out[782...] (array([2.5]), 2.5)
```

```

In [783...] np.convolve([1, 2, 3], [0, 1, 0.5], "same")

Out[783...] array([1. , 2.5, 4. ])

In [784...] np.convolve([0, 1, 2, 3, 0], [0, 1, 0.5], "valid")

Out[784...] array([1. , 2.5, 4. ])

In [785...] # 默认 full
np.convolve([1, 2, 3], [0, 1, 0.5])

Out[785...] array([0. , 1. , 2.5, 4. , 1.5])

In [786...] np.convolve([0, 0, 1, 2, 3, 0, 0], [0, 1, 0.5], "valid")

Out[786...] array([0. , 1. , 2.5, 4. , 1.5])

```

掩码运算

使用NumPy的 `ma` 模块，其很多功能和NumPy一样，我们主要介绍与MASK相关的功能。

本模块主要用于处理不完整数据或包含无效数据，或需要人为覆盖掉一部分数据的情况。

一个mask数组是一个ndarray和一个mask的组合。mask既可以是 `nomask`，表示相关联数组对应的值是无效的；要么是一个布尔数组，为False时，关联数组对应位置值有效（未遮掩），为True时无效（被遮掩）。

[Masked arrays — NumPy v1.23.dev0 Manual](#)

```

In [391...] import numpy.ma as ma

```

简介

```

In [395...] x = np.array([1, 2, 3, -1, 5])

In [396...] mx = ma.masked_array(x, mask=[0,0,0,1,0])
mx

Out[396...] masked_array(data=[1, 2, 3, --, 5],
                        mask=[False, False, False,  True, False],
                        fill_value=999999)

In [398...] mx.mean(), (1+2+3+5)/4

Out[398...] (2.75, 2.75)

```

也可以指定mask的值，这里两个阈值`rtol`和`atol`，默认值和 `np.allclose` 一样，表示在该范围内的值都被mask。

```

In [404...] # mask掉1.1
ma.masked_values([1, 1.1, 1.1+1e-8, 2, 3, 4], 1.1)

```

```
Out[404...] masked_array(data=[1.0, --, --, 2.0, 3.0, 4.0],
                    mask=[False, True, True, False, False, False],
                    fill_value=1.1)
```

```
In [413...] # 整数时完全相等才算
ma.masked_values([1, 2, 3, 4], 2, rtol=1, atol=2)
```

```
Out[413...] masked_array(data=[1, --, 3, 4],
                    mask=[False, True, False, False],
                    fill_value=2)
```

```
In [428...] # 小数时按np.isclose
# abs(a-b) < atol + rtol * abs(b)
# 以4为例, 4-2 < 1.5+1*2
ma.masked_values([1., 2., 3., 4.], 2, rtol=1, atol=1.5)
```

```
Out[428...] masked_array(data=[--, --, --, --],
                    mask=[ True, True, True, True],
                    fill_value=2.0,
                    dtype=float64)
```

```
In [430...] (np.isclose(4, 2, rtol=1, atol=1.5),
np.allclose(4,2,rtol=1, atol=1.5))
```

```
Out[430...] (True, True)
```

创建

有多种方法可以构造mask数组：

- 直接调用 `MaskedArray` 类：需要指定数据数组和Mask数组。
- 使用构造器：`array` 和 `masked_array`，后者是 `MaskedArray` 的alias，前者在参数上略有不同。
- 对已有数组通过 `view` 转为mask数组。
- 其他一些内置的函数，比如上面提到的 `masked_values`，还有比如与给定value完全相等的会被mask的 `masked_object`，根据条件mask的 `masked_where` 等。更多可参考下面的文档。

[The numpy.ma module — NumPy v1.23.dev0 Manual](#)

```
In [440...] a = np.arange(6).reshape(2, 3)
mask = [[False, True, False],[False, False, True]]
```

```
In [441...] # 直接调用类
ma.MaskedArray(a, mask=mask)
```

```
Out[441...] masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False, True, False],
          [False, False, True]],
    fill_value=999999)
```

```
In [443...] # 使用构造器
ma.array(a, mask=mask)
```

```
Out[443... masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False, True, False],
          [False, False, True]],
    fill_value=999999)
```

```
In [455... # 类的alias
ma.masked_array(a, mask)
```

```
Out[455... masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False, True, False],
          [False, False, True]],
    fill_value=999999)
```

```
In [456... # view
a.view(ma.MaskedArray)
```

```
Out[456... masked_array(
    data=[[0, 1, 2],
          [3, 4, 5]],
    mask=False,
    fill_value=999999)
```

```
In [471... # masked_object一般用于对象
ma.masked_object(a, 3)
```

```
Out[471... masked_array(
    data=[[0, 1, 2],
          [--, 4, 5]],
    mask=[[False, False, False],
          [ True, False, False]],
    fill_value=3)
```

```
In [466... ma.masked_object(
    np.array(["a", "b", "c"], dtype=object),
    "b"
)
```

```
Out[466... masked_array(data=['a', --, 'c'],
    mask=[False, True, False],
    fill_value='b',
    dtype=object)
```

```
In [476... # 有条件的
ma.masked_where(a>3, a)
```

```
Out[476... masked_array(
    data=[[0, 1, 2],
          [3, --, --]],
    mask=[[False, False, False],
          [False, True, True]],
    fill_value=999999)
```

获取

```
In [477... a
```

```
Out[477... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [478... m = ma.masked_where(a%2==0, a)
m
```

```
Out[478... masked_array(
  data=[[--, 1, --],
        [3, --, 5]],
  mask=[[ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
```

```
In [480... m.data
```

```
Out[480... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [479... m.mask
```

```
Out[479... array([[ True, False,  True],
        [False,  True, False]])
```

```
In [482... a[m.mask]
```

```
Out[482... array([0, 2, 4])
```

```
In [483... a[~m.mask]
```

```
Out[483... array([1, 3, 5])
```

```
In [486... cm = m.compressed()
cm
```

```
Out[486... array([1, 3, 5])
```

```
In [491... cm.data.obj
```

```
Out[491... array([1, 3, 5])
```

修改

mask一个或多个值可以直接指定。

首先是mask操作：

```
In [515... a = np.arange(6).reshape(2, 3)
# 第0行第2列，第1行第1列
a[(0,1),(2,1)] = ma.masked
a
```

```
Out[515... array([[0, 1, 0],
        [3, 0, 5]])
```

```
In [520... a = np.arange(6).reshape(2, 3)
# 第1列
```



```
a[:,1] = ma.masked
a
```

```
Out[520...] array([[0, 0, 2],
        [3, 0, 5]])
```

```
In [525...] a = np.arange(6).reshape(2, 3)
# 第1列以后的
a[:,1:] = ma.masked
a
```

```
Out[525...] array([[0, 0, 0],
        [3, 0, 0]])
```

```
In [527...] a = ma.arange(6).reshape(2, 3)
a.mask = [1,0,1]
a
```

```
Out[527...] masked_array(
  data=[[--, 1, --],
        [--, 4, --]],
  mask=[[ True, False,  True],
        [ True, False,  True]],
  fill_value=999999)
```

```
In [529...] ma.arange(6).reshape(2,3)
```

```
Out[529...] masked_array(
  data=[[0, 1, 2],
        [3, 4, 5]],
  mask=False,
  fill_value=999999)
```

取消Mask，只需要给对应位置一个有效值即可。

```
In [539...] a = np.arange(6).reshape(2, 3)
mask = [[False, True, False],[False, False, True]]
```

```
In [540...] x = ma.array(a, mask=mask)
x
```

```
Out[540...] masked_array(
  data=[[0, --, 2],
        [3, 4, --]],
  mask=[[False,  True, False],
        [False, False,  True]],
  fill_value=999999)
```

```
In [541...] x[0,1] = -1
x
```

```
Out[541...] masked_array(
  data=[[0, -1, 2],
        [3, 4, --]],
  mask=[[False, False, False],
        [False, False,  True]],
  fill_value=999999)
```

如果是hardmask (mask的值不能unmask)，则需要先soft:

```
In [548... x = ma.array(a, mask=mask, hard_mask=True)
x
```

```
Out[548... masked_array(
  data=[[0, --, 2],
        [3, 4, --]],
  mask=[[False, True, False],
        [False, False, True]],
  fill_value=999999)
```

```
In [550... # 看，没啥用
x[0,1] = -1
x
```

```
Out[550... masked_array(
  data=[[0, --, 2],
        [3, 4, --]],
  mask=[[False, True, False],
        [False, False, True]],
  fill_value=999999)
```

```
In [551... # 成功
x.soften_mask()
x[0,1] = -1
x
```

```
Out[551... masked_array(
  data=[[0, -1, 2],
        [3, 4, --]],
  mask=[[False, False, False],
        [False, False, True]],
  fill_value=999999)
```

```
In [553... # 再转成hard
x.harden_mask()
```

```
Out[553... masked_array(
  data=[[0, -1, 2],
        [3, 4, --]],
  mask=[[False, False, False],
        [False, False, True]],
  fill_value=999999)
```

```
In [554... x
```

```
Out[554... masked_array(
  data=[[0, -1, 2],
        [3, 4, --]],
  mask=[[False, False, False],
        [False, False, True]],
  fill_value=999999)
```

```
In [556... # hard后就不能在unmask了
x[1,2] = -2
x
```

```
Out[556... masked_array(
    data=[[0, -1, 2],
          [3, 4, --]],
    mask=[[False, False, False],
          [False, False, True]],
    fill_value=999999)
```

如果想要unmask掉所有的，直接用 `nomask`：

```
In [564... a = np.arange(6).reshape(2, 3)
mask = [[False, True, False],[False, False, True]]
x = ma.array(a, mask=mask)
x
```

```
Out[564... masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False, True, False],
          [False, False, True]],
    fill_value=999999)
```

```
In [567... # 注意，hard后是不可以的
x.mask = ma.nomask
x
```

```
Out[567... masked_array(
    data=[[0, 1, 2],
          [3, 4, 5]],
    mask=[[False, False, False],
          [False, False, False]],
    fill_value=999999)
```

索引切片

因为是 `ndarray` 的子类，所以和 `array` 是类似的，当前，很多其他方面也是通用的。

```
In [570... a = np.arange(6).reshape(2, 3)
mask = [[False, True, False],[False, False, True]]
x = ma.array(a, mask=mask)
x
```

```
Out[570... masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False, True, False],
          [False, False, True]],
    fill_value=999999)
```

```
In [572... x[0]
```

```
Out[572... masked_array(data=[0, --, 2],
    mask=[False, True, False],
    fill_value=999999)
```

```
In [573... x[0,0]
```

```
Out[573... 0
```

```
In [574... x[0,1]
```

```
Out[574... masked
```

```
In [575... x[:, -1]
```

```
Out[575... masked_array(data=[2, --],
                    mask=[False,  True],
                    fill_value=999999)
```

```
In [577... x[:1]
```

```
Out[577... masked_array(data=[[0, --, 2]],
                    mask=[[False,  True, False]],
                    fill_value=999999)
```

代数运算

`ma` 模块有大多数通函数的特定时限，位置被mask或值计算无效时，都会直接变成mask。

`ma` 也支持标准的通函数，输入mask数组，输出对应位置依然是mask的。

```
In [578... ma.log([-1, 0, 1, 2])
```

```
Out[578... masked_array(data=[--, --, 0.0, 0.6931471805599453],
                    mask=[ True,  True, False, False],
                    fill_value=1e+20)
```

```
In [579... x
```

```
Out[579... masked_array(
    data=[[0, --, 2],
          [3, 4, --]],
    mask=[[False,  True, False],
          [False, False,  True]],
    fill_value=999999)
```

```
In [581... np.log(x)
```

```
<ipython-input-581-de666c833898>:1: RuntimeWarning: divide by zero encountered in log
  np.log(x)
```

```
Out[581... masked_array(
    data=[[--, --, 0.6931471805599453],
          [1.0986122886681098, 1.3862943611198906, --]],
    mask=[[ True,  True, False],
          [False, False,  True]],
    fill_value=999999)
```

```
In [582... np.exp(x)
```

```
Out[582... masked_array(
  data=[[1.0, --, 7.38905609893065],
        [20.085536923187668, 54.598150033144236, --]],
  mask=[[False, True, False],
        [False, False, True]],
  fill_value=999999)
```

使用案例

一般用于缺失值或异常值的处理。

```
In [585... # 假设a[0,2]这个值是缺失值
a[0,2] = -9999
x = ma.masked_values(a, -9999)
x
```

```
Out[585... masked_array(
  data=[[0, 1, --],
        [3, 4, 5]],
  mask=[[False, False, True],
        [False, False, False]],
  fill_value=-9999)
```

```
In [588... x.mean(), 13/5
```

```
Out[588... (2.6, 2.6)
```

```
In [589... x - x.mean()
```

```
Out[589... masked_array(
  data=[[-2.6, -1.6, --],
        [0.3999999999999999, 1.4, 2.4]],
  mask=[[False, False, True],
        [False, False, False]],
  fill_value=-9999)
```

```
In [591... x.anom()
```

```
Out[591... masked_array(
  data=[[-2.6, -1.6, --],
        [0.3999999999999999, 1.4, 2.4]],
  mask=[[False, False, True],
        [False, False, False]],
  fill_value=-9999)
```

```
In [594... x.anom(axis=0)
```

```
Out[594... masked_array(
  data=[[-1.5, -1.5, --],
        [1.5, 1.5, 0.0]],
  mask=[[False, False, True],
        [False, False, False]],
  fill_value=-9999)
```

```
In [595... x - x.mean(axis=0)
```

```
Out[595...] masked_array(  
    data=[[-1.5, -1.5, --],  
          [1.5, 1.5, 0.0]],  
    mask=[[False, False,  True],  
          [False, False, False]],  
    fill_value=-9999)
```

填充缺失值:

```
In [601...] x.filled(x.mean())
```

```
Out[601...] array([[0, 1, 2],  
                  [3, 4, 5]])
```

```
In [602...] x
```

```
Out[602...] masked_array(  
    data=[[0, 1, --],  
          [3, 4, 5]],  
    mask=[[False, False,  True],  
          [False, False, False]],  
    fill_value=-9999)
```

两个mask掉的数组也可以计算:

```
In [622...] a = np.arange(6).reshape(2, 3)  
mask1 = [[False, True, False], [False, False, True]]  
mask2 = [[False, False, True], [False, False, True]]  
x1 = ma.array(a, mask=mask1)  
x2 = ma.array(a, mask=mask2)  
x1, x2
```

```
Out[622...] (masked_array(  
    data=[[0, --, 2],  
          [3, 4, --]],  
    mask=[[False,  True, False],  
          [False, False,  True]],  
    fill_value=9999999),  
masked_array(  
    data=[[0, 1, --],  
          [3, 4, --]],  
    mask=[[False, False,  True],  
          [False, False,  True]],  
    fill_value=9999999))
```

```
In [623...] x1+x2
```

```
Out[623...] masked_array(  
    data=[[0, --, --],  
          [6, 8, --]],  
    mask=[[False,  True,  True],  
          [False, False,  True]],  
    fill_value=9999999)
```

```
In [625...] # 0/0无效, 直接变成mask  
np.sqrt(x1/x2)
```

```
Out[625... masked_array(
    data=[[--, --, --],
          [1.0, 1.0, --]],
    mask=[[ True,  True,  True],
          [False, False,  True]],
    fill_value=999999)
```

可以根据条件处理数组：

```
In [630... a = np.arange(6).reshape(2, 3)
a
```

```
Out[630... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [634... # 在给定范围之外的就给mask掉
m = ma.masked_outside(a, 1, 4)
m
```

```
Out[634... masked_array(
    data=[[--, 1, 2],
          [3, 4, --]],
    mask=[[ True, False, False],
          [False, False,  True]],
    fill_value=999999)
```

```
In [636... m.mean(), (1+2+3+4)/4
```

```
Out[636... (2.5, 2.5)
```

```
In [639... # 3到5之间的都给mask掉
m = ma.masked_inside(a, 3, 5)
m
```

```
Out[639... masked_array(
    data=[[0, 1, 2],
          [--, --, --]],
    mask=[[False, False, False],
          [ True,  True,  True]],
    fill_value=999999)
```

```
In [643... m = ma.masked_greater(a, 2)
m
```

```
Out[643... masked_array(
    data=[[0, 1, 2],
          [--, --, --]],
    mask=[[False, False, False],
          [ True,  True,  True]],
    fill_value=999999)
```

小结

参考

- [python - Understanding NumPy's einsum - Stack Overflow](#)
- [Tim Rocktäschel](#)

In []: