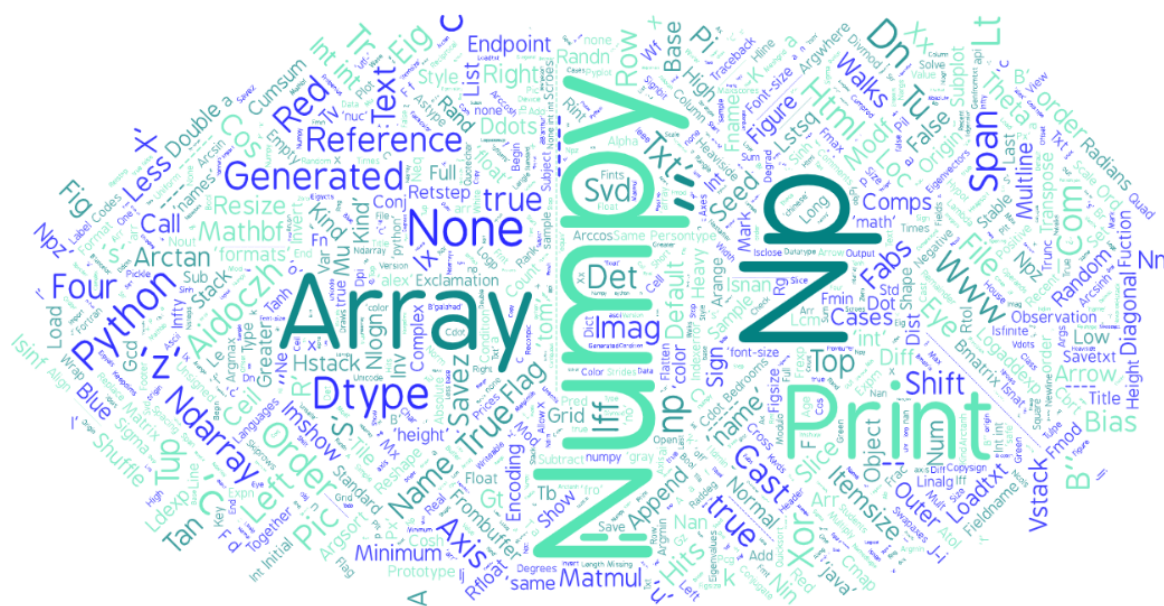


Numpy



一、NumPy基础

1、数据类型

NumPy常用的数据类型如下

类型	类型代码	对应C的类型	说明
<code>bool</code> 或 <code>bool_</code>	N/A	<code>bool</code>	布尔值True或False, 1bytes
<code>int8/uint8</code>	<code>i1/u1</code>	<code>char/unsigned char</code>	有符号位和无符号位的8位整型 (1bytes)
<code>int16/uint16</code>	<code>i2/u2</code>	<code>short/unsigned short</code>	有符号位和无符号位的16位整型 (2bytes)
<code>int32/uint32</code>	<code>i4/u4</code>	<code>int/unsigned int</code>	有符号位和无符号位的32位整型 (4bytes)
<code>int64/uint64</code>	<code>i8/u8</code>	<code>long/unsigned long</code>	有符号位和无符号位的64位整型 (8bytes) 至少有32位整型 (也就是int32/uint32也可以对应long类型)
<code>float16</code>	<code>f2</code>	N/A	半精度浮点数: 符号位、5位指数和10位尾数
<code>float32</code>	<code>f4</code> 或 <code>f</code>	<code>float</code>	单精度浮点数: 通常包括符号位、8位指数和23位尾数
<code>float64</code>	<code>f8</code> 或 <code>d</code>	<code>double</code>	双精度浮点数: 通常包括符号位、11位指数和52位尾数
<code>float96</code> 或 <code>float128</code>		<code>long double</code>	平台定义的扩展精度浮点数
<code>complex64</code>		<code>float complex</code>	复数, 由两个单精度浮点数表示 (实部和虚部)
<code>complex128</code>		<code>double complex</code>	复数, 由两个双精度浮点数表示 (实部和虚部)
<code>object</code>	<code>O</code>		任意python对象
<code>bytes_</code>	<code>S</code>		固定长度的ASCII字符串类型, 例如: 要创建一个长度为10的字符串, 应使用'S10'
<code>unicode_</code>	<code>U</code>		固定长度的Unicode类型 (字节数由平台决定) 例如: 'U10'

2、副本和视图

【引言】: NumPy数组是一个由两部分组成的数据结构: 包含实际数据元素的连续数据缓冲区和包含有关数据缓冲区信息的元数据。元数据包括数据类型、步幅和其他重要信息, 有助于轻松操作ndarray。

对于索引操作来说, 基本索引总是返回视图, 高级索引总是返回副本

A 视图

通过更改某些元数据 (如 步幅 和 数据类型) 而不更改数据缓冲区, 可以以不同的方式访问数组。这创建了一种查看数据的新方式, 这些新数组被称为视图。数据缓冲区保持不变, 因此对视图所做的任何更改都会反映在原始副本中。可以通过 `ndarray.view` 方法强制创建视图。

B 副本

当通过复制数据缓冲区和元数据创建一个新数组时，这称为复制。对副本所做的更改不会反映在原始数组上，制作副本较慢且消耗内存，但有时是必要的。可以通过使用 `ndarray.copy` 强制创建副本。

如何判断数组是视图还是副本

`base` 属性使得判断一个数组是视图还是副本变得容易，视图的`base`属性返回原始数组，而副本的`base`属性返回 `None`

3、在ndarray上的索引

`ndarray` 可以使用标准的Python `x[obj]` 语法进行索引，其中`x`是数组，`obj` 是选择，根据`obj`的不同，有不同类型的索引可用：基本索引、高级索引和字段访问。

注意在Python中，`x[(exp1, exp2, ..., expN)]` 等价于 `x[exp1, exp2, ..., expN]`，后者只是前者的语法糖

(1) 基本索引

单元元素索引

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

# x 如下
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

print(x[1, 2])           #6
print(x[1, 2] == x[1][2]) #True
print(x[0])             #[0 1 2 3]
```

请注意：

！如果使用少于维度的索引数来索引多维数组，会得到一个子数组，也就是说，每个指定的索引选择对应于所选其余维度的数组，在上面的例子中，选择0意味着长度为4的剩余维度未被制定，并且返回的是该维度大小的数组，必须注意的是，返回的数组是一个[视图]，它不是副本。

！`x[0, 2] == x[0][2]`，第二种情况效率更低，因为在第一个索引之后会创建一个新的临时数组，随后由下一个进行索引

切片和步幅

NumPy的切片扩展了Python的基本切片概念到N维，当`obj`是一个 `slice` 对象（通过括号内的 `start:stop:step` 符号构造）、一个整数或一个由切片对象和整数组成的元组时，会发生切片。

最简单的用N个整数索引的情况会返回一个数组标量，表示相应的项，与Python一样，所有索引都是从零开始的：对于第 i 个索引，其有效范围是 $0 \leq n_i < d_i$ ，其中 d_i 是数组形状的第 i 个元素。负索引被解释为从数组末尾开始计数（如：如果 $n_i < 0$ ，意味着 $n_i + d_i$ ）

通过基本切片生成的所有数组始终是原始数组的 视图

🔑 NumPy切片创建的是一个视图，而不是像字符串、元组和列表这样的内置Python序列中的副本。所以从一个大的数组中提取一小部分时必须小心，因为提取的小部分包含对大的原始数组的引用，其内存不会在所有从它派生的数组被垃圾回收之前释放，在这种情况下，内存占用会高，所以建议显式使用 `copy()`。

序列切片的常规规则适用于按维度进行的基本切片（包括使用步长索引），一些需要牢记的有用概念包括：

- 基本的切片语法是 `i:j:k`，其中 `i` 是起始索引，`j` 是结束索引，`k` 是步长 ($k \neq 0$)，这会在相应的维度中选择 m 个元素，其索引值为 $i, i+k, \dots, i+(m-1)k$ ，其中 $m = q + r (r \neq 0)$ ，`q`和`r`是通过将 $j-i$ 除以 k 得到的商和余数： $j-i = qk + r$ ，因此 $i + (m-1)k < j$ ，例如：

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> x[1:6:2]
array([2, 4, 6])
```

- 负的 `i` 和 `j` 被解释为 $n+i$ 和 $n+j$ ，其中 n 是相应维度中的元素数量，负的 `k` 使步长向小的索引方向进行。例如：

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> x[-3:6]
array([6])
>>> x[6:2:-1]
array([7, 6, 5, 4])
```

- 假设 `n` 是要切片的维度中的元素数量，那么，如果 `i` 为给出，对于 $k>0$ ，默认值为0，对于 $k<0$ ，默认值为 $n-1$ 。如果 `j` 未给出，对于 $k>0$ ，默认值为 `n`，对于 $k<0$ ，默认值为 $-n-1$ 。如果 `k` 未给出，默认值为1。注意 `:` 与 `:` 相同，表示选择此轴上的所有索引，例如

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> x[5:]
array([6, 7, 8])
>>> x[3:-1]
array([8, 7, 6, 5])
>>> x[5::-1]
array([6, 5, 4, 3, 2, 1])
```

- 如果某一个维度的数量少于 `N`，则假设后续维度为 `:`，例如

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

- 你可以使用切片来设置数组中的值，但（与列表不同）你永远不能扩展数组，要在 `x[obj]=value` 中设置的值的大小必须（可广播为）与 `x[obj]` 的形状相同。
- 一个切片元组可以总是被构造为 `obj` 并在 `x[obj]` 符号中使用。切片对象可以在构造中代替 `[start:stop:step]` 符号使用。例如，`x[1:10:5, ::-1]` 可以实现为 `obj = (slice(1, 10, 5), slice(None, None, -1)); x[obj]`。

(2) 高级索引

当选择对象 *obj* 是一个非元组序列对象、一个 `ndarray`，或至少包含一个序列对象或数据类型为整数或布尔的 `ndarray` 的元组时，会触发高级索引，高级索引有两种类型：整数和布尔。

高级索引总是返回数据的副本（与返回视图的基本切片形成对比）

！高级索引意味着 `x[(1, 2, 3),]` 与 `x[(1, 2, 3)]` 本质上不同，后者等同于 `x[1, 2, 3]`，触发基本索引，而前者触发高级索引

整数数组索引

整数数组索引允许根据数组的 *N* 维索引选择任意项，每个整数数组表示该维度中的若干索引。

索引数组中允许使用负值，其工作方式与单个索引或切片相同。如果索引值超出范围，则会抛出 `IndexError`。

```
x = np.arange(10, 0, -1)
print(x)
index_a = np.array([6, 6, 2, 1])
index_b = np.array([5, 6, -1, 1])
print(x[index_a])    #4 4 8 9
print(x[index_b])    #5 4 1 9
```

#运行结果如下

```
[10  9  8  7  6  5  4  3  2  1]
[4 4 8 9]
[5 4 1 9]
```

```
x = np.arange(9).reshape(3, -1)
print(x)
slice1 = np.array([0, -1])
print(x[slice1])          #表示取第0行和最后一行
print(x[:, slice1])       #表示取第0列和最后一列
```

#运行结果如下

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[0 1 2]
 [6 7 8]]
[[0 2]
 [3 5]
 [6 8]]
```

```
x = np.arange(9).reshape(3, -1)
>>> x[3][4]
IndexError: index 3 is out of bounds for axis 0 with size 3    #索引3超出轴0的范围
```

高级索引总是广播

A 从最简单的高级索引开始：

```
a = np.arange(35).reshape(5, 7)
print(a)
x = np.array([0, 2, 4])
```

```
y = np.array([0, 1, 2])
print(a[x, y])
```

#运行结果如下

```
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
[ 0 15 30]
```

在这种情况下，如果索引数组具有相同的形状，并且对于索引数组的每个维度都有一个索引数组，则结果数组有着与索引数组相同的形状。在这个例子中，第一个索引值对于两个索引数组都是0，因此结果数组的第一个值是 `a[0, 0]`，下一个值是 `a[2, 1]`，最后一个是 `a[4, 2]`。

如果索引数组没有相同的形状，则会尝试将它们广播到相同的形状，标量总是可以广播，如果它们不能被广播到相同的形状，则会引发异常[`IndexError`]

```
a = np.arange(35).reshape(5, 7)
print(a)
x = np.array([0, 2, 4])
y = np.array([0, 3])
print(a[x, 1])
print(a[x, y])
```

#报错信息提示无法广播到相同的形状

#运行结果如下

```
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
[ 1 15 29] #a[0, 1] a[2, 1] a[4, 1]
```

```
-----
IndexError                                Traceback (most recent call last)
Cell In[5], line 6
      4 y = np.array([0, 3])
      5 print(a[x, 1])
----> 6 print(a[x, y])
```

IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (3,) (2,)

B 跳到下一个复杂级别：可以使用索引数组部分索引一个数组，

例如：如果我们只使用一个索引数组

```
a = np.arange(35).reshape(5, 7)
print(a)
x = np.array([0, 2, 4])
y = np.array([0, 3])
print(a[x])
```

#其结果如下

```
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
[[ 0  1  2  3  4  5  6]
 [14 15 16 17 18 19 20]
 [28 29 30 31 32 33 34]]
```

下面看几个示例:

```
a = np.arange(35).reshape(5, 7)
print(a)
x = np.array([0, 2])
y = np.array([0, 3])
print("=====a[0][0] a[2][3]=====")
print(a[x, y])
print(f"=====输出的结果为=====\n\n\t[a[0][0],a[0][3]],\n\t[a[2][0],a[2][3]]\n\n=====")
#下面两行等效
print(a[x][:,y])
print(a[np.ix_(x, y)])

#结果如下
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
=====a[0][0] a[2][3]=====
[ 0 17]
=====输出的结果为=====
[
    [a[0][0],a[0][3]],
    [a[2][0],a[2][3]]
]
=====
[[ 0  3]
 [14 17]]
[[ 0  3]
 [14 17]]
```

布尔数组索引

当`obj`是一个布尔类型的数组对象时, 会发生这种高级索引, 例如可能由比较运算符返回的对象。单个布尔值索引数组实际上与 `x[obj.nonzero()]` 相同。

常见的用例是过滤所需元素值。

例如, 你可能希望从数组中选择所有不是 `numpy.nan` 的元素, 或者希望将一个常数加到所有的负元素中:

```
x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
>>> x[~np.isnan(x)]          #筛选所有不是np.nan的元素
array([1., 2., 3.])

a = np.random.randint(-10, 5, size=10)
print("=====原数组=====")
print(a)
a[a < 0] += 10
print("=====原数组的所有负数加10=====")
print(a)
```

#运行结果如下

```
=====原数组=====
```

```
[ -5  -5  -2 -10  -7  -4  -6   4   2  -7]
=====原数组的所有负数加10=====
[5  5  8  0  3  6  4  4  2  3]
```

通常，如果一个索引包含一个布尔数组，结果将与在其相同位置插入 `obj.nonzero()` 并使用其整数数组（0，1组成）索引机制相同

```
x = np.arange(25).reshape(5, 5)
print("=====原数组=====")
print(x)
b = x > 15
print("=====原数组中大于20的布尔数组=====")
print(b)
print("=====原数组中大于20的布尔数组，取第三列=====")
print(b[:, 2])
print("=====当布尔数组的维度小于被索引数组时=====")
print(x[:, b[:, 2]]) #这个其实会对列做布尔索引
#以下三行等价
print(x[b[:, 2], :]) #这个其实是只对行做布尔索引
print(x[b[:, 2], 2, :])
print(x[b[:, 2], 2, ...])
print(x[[False, False, False, True, True]])

#运行结果如下
=====原数组=====
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
=====原数组中大于20的布尔数组=====
[[False False False False False]
 [False False False False False]
 [False False False False False]
 [False True True True True]
 [ True True True True True]]
=====原数组中大于20的布尔数组，取第三列=====
[False False False True True]
=====当布尔数组的维度小于被索引数组时=====
[[ 3  4]
 [ 8  9]
 [13 14]
 [18 19]
 [23 24]]
[[15 16 17 18 19]
 [20 21 22 23 24]]
[[15 16 17 18 19]
 [20 21 22 23 24]]
[[15 16 17 18 19]
 [20 21 22 23 24]]
[[15 16 17 18 19]
 [20 21 22 23 24]]
```

使用布尔索引选择所有行加起来为偶数的行.同时,应使用高级整数索引选择第0列和第2列，使用 `np.ix_()`

```
x = np.arange(12).reshape((4, 3))
print(x)
row = (x.sum(-1) % 2 == 0)
col = [0, 2]
```



```
print(x[np.ix_(row, col)])
```

#运行结果如下

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[ 3  5]
 [ 9 11]]
```

使用形状为 (2, 3) 的二维布尔数组,其中包含四个 True 元素,从形状为 (2, 3, 5) 的三维数组中选择行,结果是一个形状为 (4, 5) 的二维结果:

```
x = np.arange(30).reshape(2, 3, 5)
>>> x
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],
       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> b = np.array([[True, True, False], [False, True, True]])
>>> x[b]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

(3) 结合高级和基本索引

当索引中至少有一个切片 :, 省略号 ... 或 newaxis 时 (或者数组的维度 > 高级索引的维度), 行为可能更复杂。这类似于连接每个高级索引元素的索引结果。

最简单的情况, 只有一个单一高级索引与一个切片结合。例如:

```
>>> x = np.arange(35).reshape(5, 7)
>>> x[np.array([0, 2, 4]), 1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

实际上, 切片和索引数组操作是独立的, 切片操作提取索引为1和2的列, 接着是索引数组操作, 提取索引为0、2、4的行, 这等效于:

```
>>> x[:,1:3][np.array([0, 2, 4]), :]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

当然, 一个高级索引可以替换一个切片, 其结果数组是相同的, 但使用高级索引会生成一个副本, 并且可能具有不同的内存布局。因此, 建议使用切片

#建议第一种方式

```
>>> x[np.array([0, 2, 4]), 1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])

>>> x[np.array([0, 2, 4]), np.array([1, 2])]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

切片可以与广播布尔索引结合使用:

```
x = np.arange(35).reshape(5, 7)
print(x)
b = x > 20
print(x[b[:, 2], 1:3])
```

#运行结果如下

```
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
[[22 23]
 [29 30]]
```

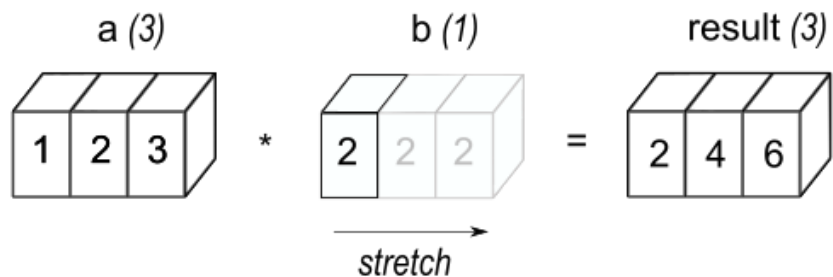
4、广播 (broadcasting)

广播描述了NumPy如何在算术运算期间处理不同形状的数组。在某些约束条件下，较小的数组会广播到较大的数组上，以便它们具有兼容的形状。广播提供了一种向量化数组操作的方法，使得循环发生在C语言中而不是Python中

最简单的广播：数组和标量值作运算

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2.,  4.,  6.])
```

标量 b 被拉伸成为一个与 a 形状相同的数组



通用广播规则

当操作两个数组时，NumPy会逐元素比较它们的形状。它从最右边的维度开始，并向左进行，两个维度在以下情况下是兼容的：

- 它们是相等的
- 其中之一是1

如果不满足以上条件，会抛出异常[ValueError: operands could not be broadcast together][]

例如：

A 如果你有一个256x256x3的RGB值数组，并且你想按不同的值缩放图像中的每种颜色，你可以将图形乘以一个包含3个值的一维数组，其广播如下：

```
Image (3d array): 256x256x3
scale (1d array):      3
Result (3d array): 256x256x3
```

B 当比较的两个维度之一为1时，使用另一个维度，换句话说，大小为1的维度会被拉伸或复制以匹配另一个维度

```
A (4d array): 8 x 1 x 6 x 1
B (3d array): 7 x 1 x 5
Result (4d array): 8 x 7 x 6 x 5
```

广播数组

1 可以被广播的例子

```
A (2d array): 5 x 4
B (1d array): 1
Result (2d array): 5 x 4

A (2d array): 5 x 4
B (1d array): 4
Result (2d array): 5 x 4

A (3d array): 15 x 3 x 5
B (3d array): 15 x 1 x 5
Result (3d array): 15 x 3 x 5

A (3d array): 15 x 3 x 5
B (2d array): 3 x 5
Result (3d array): 15 x 3 x 5

A (3d array): 15 x 3 x 5
B (2d array): 3 x 1
Result (3d array): 15 x 3 x 5
```

2 不可被广播的例子

```
A (1d array): 3
B (1d array): 4

A (2d array): 2 x 1
B (3d array): 8 x 4 x 3
```

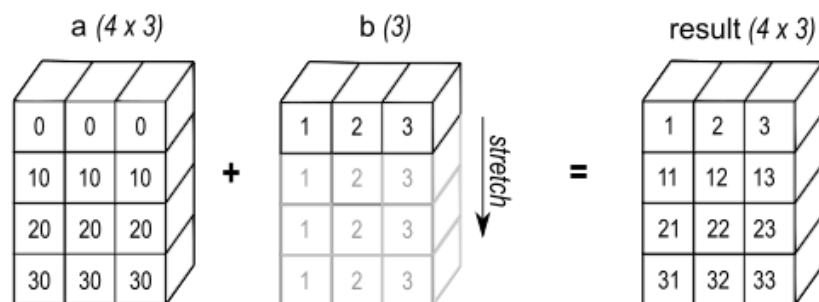
3 一维数组和二维数组运算的广播

```
a = np.array([[ 0.0,  0.0,  0.0],
...          [10.0, 10.0, 10.0],
...          [20.0, 20.0, 20.0],
...          [30.0, 30.0, 30.0]])
#可以广播
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b                                #a 4x3
array([[ 1.,  2.,  3.],                  #b 3
       [11., 12., 13.],                  # 4x3
       [21., 22., 23.],
       [31., 32., 33.]])

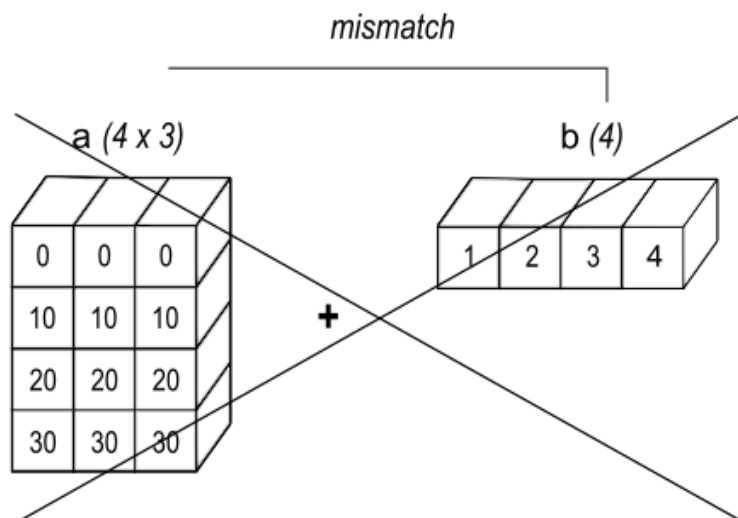
#不可广播
>>> b = np.array([1.0, 2.0, 3.0, 4.0])
>>> a + b
Traceback (most recent call last):
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

如图所示：

一个一维数组和一个二维数组运算时，如果一维数组的元素数量与二维数组的列数相同，则会导致广播



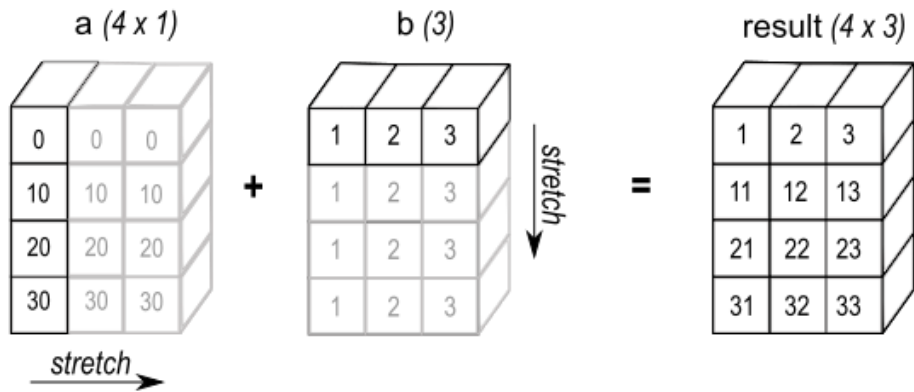
反之，不可广播



4 两个一维数组的广播，便于获取两个向量的外积（等同于`np.outer(a, b)`）

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b           #则a变成了(4,1)的矩阵
array([[ 1.,  2.,  3.],           # a 4x1
       [11., 12., 13.],           # b 1x3
       [21., 22., 23.],           #   4x3
       [31., 32., 33.]])
```

如图所示：在某些情况下，广播会拉伸两个数组以形成一个比初始数组中任何一个都大的输出数组



#实际例子：向量量化（VQ算法）

VQ算法中的基本操作是找到一组点中最接近给定点的点。这组点在VQ术语中称为 `codes`，给定点称为 `observation`。

下述例子中：

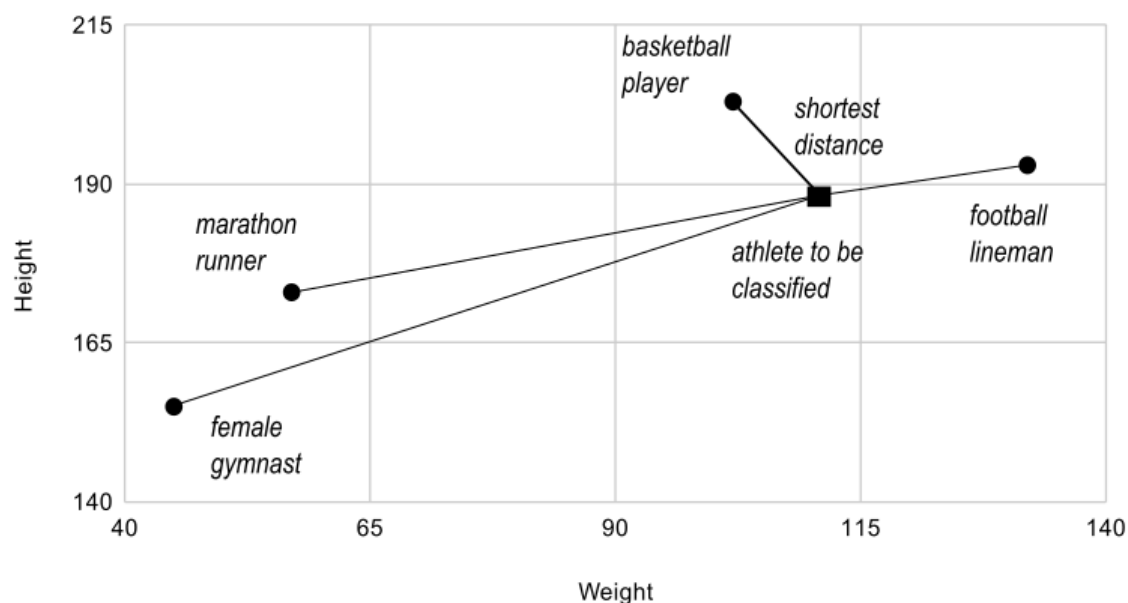
- `observation`：描述了要分类的运动员的体重和身高
- `codes`：代表了不同类别的运动员的体重和身高
- `diff`：差值数组
- `dist`：`observation`中的点到`codes`中的点的距离

```
from numpy import array, argmin, sqrt, sum
>>> observation = array([111.0, 188.0])
>>> codes = array([[102.0, 203.0],
...                [132.0, 193.0],
...                [45.0, 155.0],
...                [57.0, 173.0]])
>>> diff = codes - observation           # 在这里广播
>>> dist = sqrt(sum(diff**2,axis=-1))
>>> argmin(dist)
0
```

#其过程如下

```
Observation    (1d array):      2
Codes          (2d array):  4 x 2
diff           (2d array):  4 x 2
```

VQ算法如图所示



5、数组的算术运算

数组的算术运算符应用于**逐元素的运算**

一个简单的例子

例如，你创建了两个数组，`data`和`ones`，并且让它们做四则运算

```
>>> data = np.array([1, 2])
>>> ones = np.ones(2, dtype=int)
>>> data + ones
array([2, 3])

>>> data - ones
array([0, 1])
>>> data * data
array([1, 4])
>>> data / data
array([1., 1.])
```

运算过程如图所示：可以看到它们是逐元素运算的

$$\begin{array}{c} \text{data} \\ \text{data} + \text{ones} \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} + \begin{array}{|c|} \hline \text{ones} \\ \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline \end{array}$$

data		ones		
1	-	1	=	0
2		1		1

data		data		
1	*	1	=	1
2		2		4

data		data		
1	/	1	=	1
2		2		1

其他运算

A 当然，也可以做平方，三角函数等操作，甚至可以 `a**a`，就相当于 `a` 中的每个元素执行 `a[i]**a[i]`

这种情况下会创建一个新数组并把结果赋值到新数组

```
a = np.array([1, 2, 3, 4])
print(a.dtype)
print(a**a)          # [1**1, 2**2, 3**3, 4**4]
print(a>>2)          # 每个元素右移两位
print(2<<a)           # 2分别左移1,2,3,4位
print(a >> a)         # [1>>1, 2>>2, 3>>3, 4>>4]
print(1j*a)
print(np.sin(a))      #求sin1 sin2 sin3 sin4
print(a<3)           #生成布尔数组
```

#这里重点说一下最后一行的输出结果，我通过上述运算之后，
#原本的 `a` 是不会被改变的，也就是说在进行上述运算时，系统会创建一个新数组来保存结果
#这与底下的例子不同
`print(a)`

#运行结果如下

```
int64
[ 1  4 27 256]
[0 0 0 1]
[ 4  8 16 32]
[0 0 0 0]
[0.+1.j 0.+2.j 0.+3.j 0.+4.j]
[ 8.41470985  9.09297427  1.41120008 -7.56802495]
[ True  True False False]
[1 2 3 4]
```

B 一些操作。例如 `+=` 和 `*=`，会就地修改现有数组，而不是创建一个新数组

```
rg = np.random.default_rng(1)          #创建一个随机数生成器的实例，1是seed
a = np.ones((2, 3), dtype=int)         #a的数据类型是int64
b = rg.random((2, 3))                 #b的数据类型是float64
a *= 3                                  #
print(a)
b += a                                  #数据类型会自动向上转换，也就是int64可以自动转成float64
print(b)
```

#输出结果如下

```
[[3 3 3]
 [3 3 3]]
[[3.51182162 3.9504637 3.14415961]
 [3.94864945 3.31183145 3.42332645]]
```

```
>>> a += b          #浮点数不能自动转换成整数，即float64不能自动转成int64
UFuncTypeError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

6、结构化数组

1、基本操作

结构化数组是数据类型为简单数据类型组合并以一系列命名字段组织的ndarray。结构化数据类型旨在能够模仿C语言中的[结构体]，并共享类似的内存布局。例如：

```
import numpy as np
x = np.array([('Alex', 18, 175.5), ('Tom', 21, 180.6)], dtype=[('name', 'U10'), ('age', 'i4'), ('height', 'f4')])
print("=====原数组=====")
print(x)
print("=====取第二个数据=====")
print(x[1])
print("=====单个字段索引=====")
print(x['age'])
print("=====多个字段索引=====")
print(x[['name', 'age']])
print("=====将年龄改成20之后=====")
x['age'] = 20
print(x)
```

#运行结果如下

```
=====原数组=====
[('Alex', 18, 175.5) ('Tom', 21, 180.6)]
=====取第二个数据=====
('Tom', 21, 180.6)
=====单个字段索引=====
[18 21]
=====多个字段索引=====
[('Alex', 18) ('Tom', 21)]
=====将年龄改成20之后=====
[('Alex', 20, 175.5) ('Tom', 20, 180.6)]
```

这里的 `x` 是一个长度为2的一维数组，其数据类型是包含三个字段的结构

- 一个长度为10或更少的字符串，名为 `name`
- 一个名为 `age` 的32位整数
- 一个名为 `height` 的32位浮点数

通过一个字段名或多个字段名来索引结构化数组总是生成原数组的[视图]

2、结构化数据类型的创建

一个结构化数据类型可以被认为是一个特定长度（结构体的 `itemsize`）的字节序列，这些字节被解释为一组字段的集合。每个字段都有一个名称、一个数据类型和一个在结构体中的字节偏移量。字段的数据类型可以是任何numpy数据类型，包括其他结构化的数据类型，它也可以是一个子数组数据类型，其行为类似于指定形状的ndarray。字节的偏移量是任意的，字段甚至可以重叠，这些偏移量通常由numpy自动确定，但也可以指定。

可以使用函数 `numpy.dtype` 创建结构化数据类型，有四种不同的规范形式，如下

1 一个元组列表，每个字段是一个元组

每个元组的形式是 `(fieldname, datatype, shape)`，其中`shape`是可选的，`fieldname` 是一个字符串（如果使用标题，则为元组），`datatype` 可以是任何可转换为数据类型的对象，`shape` 是一个整数元组，指定子数组形状。

```
>>> np.dtype([( 'x', 'f4'), ('y', np.float32), ('z', 'i4', (2, 2))])
dtype([( 'x', '<f4'), ('y', '<f4'), ('z', '<i4', (2, 2))])
```

如果 `fieldname` 是空字符串 `''`，该字段将被赋予一个默认名称，形式为 `f#`，其中 `#` 是字段的整数索引，从左到右从0开始计数，结构中字段的字节偏移量和总结项大小事自动确定的

```
>>> np.dtype([( 'x', 'f4'), ('', np.float32), ('z', 'i4')])
dtype([( 'x', '<f4'), ('f1', '<f4'), ('z', '<i4')])
```

2 一串以逗号分隔的dtype规范

在这种简写表示法中，任何字符串数据类型规范都可以在字符串中使用，并用逗号分隔。字段的 `itemsize` 和字节偏移量是自动确定的，字段名称被赋予默认名称 `f0`, `f1` 等

```
>>> np.dtype('i4, f4, S3')
dtype([( 'f0', '<i4'), ('f1', '<f4'), ('f2', 'S3')])

>>> np.dtype('3int8, float32, (2, 3)float64')
dtype([( 'f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

3 字段参数数组的字典

这是最灵活的规范形式，因为它允许控制字段的字节偏移量和结构的 `itemsize`。

字典有两个必须的键，`names` 和 `formats`，以及四个可选的键 `offsets`, `itemsize`, `aligned` 和 `titles`（可以像字段名称一样索引数组）。`names` 和 `formats` 的值分别为字段名称列表和dtype的规范列表，长度必须相同。可选的 `offsets` 值应为整数字节偏移量的列表，结构中的每个字段对应一个偏移量。如果未给出 `offsets`，则自动确定偏移量。可选的 `itemsize` 值应为描述dtype总字节大小的整数，必须足够大以包含所有字段。

```
np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4']})
dtype([( 'col1', '<i4'), ('col2', '<f4')])

>>> np.dtype({'names': ['col1', 'col2'],
...           'formats': ['i4', 'f4'],
...           'offsets': [0, 4],
...           'itemsize': 12})
dtype({'names': ['col1', 'col2'], 'formats': ['<i4', '<f4'], 'offsets': [0, 4], 'itemsize': 12})
```

偏移量可以使得字段重叠，尽管这意味着对一个字段的赋值可能会破坏任何重叠字段的数据。作为一种例外，`numpy.object_` 类型的字段不能与其他字段重叠，因为有存在破坏内部对象指针的风险，并且在后续使用间接引用访问内部对象时会访问到无效内存。

可选的 `aligned` 值可以设置为 `True`，以使自动偏移计算使用对齐的偏移，等效于 `numpy.dtype` 的 `align = True`。

可选的 `titles` 值应该是一个与 `names` 长度相同的标题列表。

4 字段名称的字典（这种类型不鼓励使用）

字典的键是字段名，值是元组，指定类型和偏移量

```
>>> np.dtype({'col1': ('i1', 0), 'col2': ('f4', 1)})
dtype([('col1', 'i1'), ('col2', '<f4')])
```

3、操作和显示结构化数据类型

结构化数据类型的字段名称列表可以在 `dtype` 对象的 `names` 中找到，每个单独字段的 `dtype` 也可以通过名称查找。`dtype` 对象有一个类似字典的属性 `fields`，其键是字段名称（以及字段标题），值是包含每个字段的 `dtype` 和字节偏移量的元组。

```
d = np.dtype([('x', 'i8'), ('y', 'f4')])
>>> d.names          # 字段名称列表
('x', 'y')

>>> d['x']           # 某个字段名的 dtype
dtype('int64')

>>> d.fields         # 键值对
mappingproxy({'x': (dtype('int64'), 0), 'y': (dtype('float32'), 8)})
```

4、从其他结构化数组赋值

两个结构化数组之间的赋值就好像是源元素被转换为元组，然后赋值给目标元素。也就是说，源数组的一个字段被赋值给目标数组的第一个字段，第二个字段同样如此，依此类推，不考虑字段名称。具有不同数量字段的结构化数组不能相互赋值，目标结构数组中不属于任何字段的字节不受影响。

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'f4'), ('c', 's3')])
>>> b = np.ones(3, dtype=[('x', 'f4'), ('y', 's3'), ('z', 'o')])
>>> b[:] = a
>>> b
array([(0., b'0.0', b''), (0., b'0.0', b''), (0., b'0.0', b'')],
      dtype=[('x', '<f4'), ('y', 's3'), ('z', 'o')])
```

示例：

```
# # 请编写一个函数，接收两个参数：
# 第一个参数是结构化数组，数组的dtype是定义的person_type
# 第二个参数是课程名（'chinese', 'math', 'english' 其中一个）：
# 返回该课程分数最高的学生姓名和分数，如果有多名并列最高分，全部返回
# # 返回的是一个python的列表，列表内的元素是元组：（学生名字，学生学科的分）
def get_top_students(people, subject):
```

```

top_list = [] #存最高分的学生名和学科分数
maxscores = np.max(peoples[subject]) #求最高分
max_scores_people = peoples[peoples[subject] == maxscores] #索引最高分的人
a_tulpe = max_scores_people['name'], max_scores_people[subject] #求学生名字和分数
top_list.append(a_tulpe)
return top_list

#结构化数组定义，以字典的方式
persontype = np.dtype({
    'names': ['name', 'age', 'chinese', 'math', 'english'],
    'formats': ['S32', 'i', 'i', 'i', 'f']})

#dtype为自定义的结构化数组
peoples = np.array([
    ("ZhangFei", 32, 75, 100, 90),
    ("GuanYu", 24, 85, 96, 88.5),
    ("ZhaoYun", 28, 85, 92, 96.5),
    ("HuangZhong", 29, 65, 85, 100)
], dtype=persontype)

result = get_top_students(peoples, 'math')
print("====数学最高分的学生名字和分数====")
print(result)
result1 = get_top_students(peoples, 'chinese')
print("====语文最高分的学生名字和分数====")
print(result1)

#输出结果如下
====数学最高分的学生名字和分数====
[(array([b'ZhangFei'], dtype='<S32'), array([100], dtype=int32))]
====语文最高分的学生名字和分数====
[(array([b'GuanYu', b'ZhaoYun'], dtype='<S32'), array([85, 85], dtype=int32))]

```

7、通用函数 (ufunc)

`ufunc` 是一种对 `ndarrays` 进行逐元素操作的函数。也就是说，`ufunc` 是一个 [向量化] 的包装器，用于一个接受固定数量特定输入并产生固定数量特定输出的函数。

向量化：使用 NumPy 的程序员消除了 Python 的循环，转而使用数组到数组的操作。

(1) 可选的参数

`out`, `where`, `axes`, `axis`, `keepdims`, `casting`, `order`, `dtype`, `subok`, `signature`

out 参数的优点：

- 节省内存：避免临时数组分配
- 原地计算：直接修改输入数组
- 控制数据类型：强制指定输出的 `dtype`
- 优化链式运算：减少中间数组创建，例如 `sin(cos(a))`
- 广播兼容：支持不同形状的运算

(2) 属性

以下属性都不能被设置

属性值	说明
<code>__doc__</code>	函数文档
<code>__name__</code>	ufunc的名称
<code>ufunc.nin</code>	输入的数量
<code>ufunc.nout</code>	输出数量
<code>ufunc.nargs</code>	参数的数量
<code>ufunc.ntypes</code>	类型的数量
<code>ufunc.types</code>	返回一个按类型分组的输入->输出列表

```
print(f"函数名: {np.add.__name__}")
print(f"输入参数数量: {np.add.nin}")
print(f"输出参数数量: {np.add.nout}")
print(f"参数的总数量: {np.add.nargs}")
```

#输出结果如下

```
函数名: add
输入参数数量: 2
输出参数数量: 1
参数的总数量: 3
```

(3) 可用的ufuncs

1 数学函数

add (x1, x2, /[, out, where, casting, order, ...])	逐元素添加参数.
subtract (x1, x2, /[, out, where, casting, ...])	逐元素减去参数.
multiply (x1, x2, /[, out, where, casting, ...])	逐元素相乘.
matmul (x1, x2, /[, out, casting, order, ...])	两个数组的矩阵乘积.
divide (x1, x2, /[, out, where, casting, ...])	逐元素分割参数.
logaddexp (x1, x2, /[, out, where, casting, ...])	输入的指数和的对数.
logaddexp2 (x1, x2, /[, out, where, casting, ...])	以2为底的输入的指数和的对数.
true_divide (x1, x2, /[, out, where, ...])	逐元素分割参数.
floor_divide (x1, x2, /[, out, where, ...])	返回小于或等于输入除法的最大整数.
negative (x, /[, out, where, casting, order, ...])	逐元素取负数.
positive (x, /[, out, where, casting, order, ...])	逐元素的数值正值.
power (x1, x2, /[, out, where, casting, ...])	第一个数组的元素按元素依次提升到第二个数组的幂.
float_power (x1, x2, /[, out, where, ...])	第一个数组的元素按元素依次提升到第二个数组的幂.
remainder (x1, x2, /[, out, where, casting, ...])	返回逐元素除法的余数.
mod (x1, x2, /[, out, where, casting, order, ...])	返回逐元素除法的余数.
fmod (x1, x2, /[, out, where, casting, ...])	返回逐元素除法的余数.
divmod (x1, x2[, out1, out2], / [[, out, ...])	同时返回元素级的商和余数.
absolute (x, /[, out, where, casting, order, ...])	逐元素计算绝对值.
fabs (x, /[, out, where, casting, order, ...])	计算逐元素的绝对值.
rint (x, /[, out, where, casting, order, ...])	将数组的元素四舍五入到最近的整数.
sign (x, /[, out, where, casting, order, ...])	返回一个数字的元素级符号指示.
heaviside (x1, x2, /[, out, where, casting, ...])	计算 Heaviside 阶跃函数.
conj (x, /[, out, where, casting, order, ...])	返回逐元素的复共轭.
conjugate (x, /[, out, where, casting, ...])	返回逐元素的复共轭.
exp (x, /[, out, where, casting, order, ...])	计算输入数组中所有元素的指数.
exp2 (x, /[, out, where, casting, order, ...])	计算输入数组中所有 p 的 2^{*p} .
log (x, /[, out, where, casting, order, ...])	逐元素的自然对数.
log2 (x, /[, out, where, casting, order, ...])	x 的以 2 为底的对数.
log10 (x, /[, out, where, casting, order, ...])	返回输入数组元素的以10为底的对数.
expm1 (x, /[, out, where, casting, order, ...])	计算数组中所有元素的 $\exp(x) - 1$.
log1p (x, /[, out, where, casting, order, ...])	返回输入数组加一的自然对数,逐元素计算.
sqrt (x, /[, out, where, casting, order, ...])	返回数组的非负平方根,逐元素进行.
square (x, /[, out, where, casting, order, ...])	返回输入的元素平方.
cbrt (x, /[, out, where, casting, order, ...])	返回一个数组的三次方根,逐元素进行.
reciprocal (x, /[, out, where, casting, ...])	返回参数的倒数,逐元素进行.

add (x1, x2, /[, out, where, casting, order, ...])	逐元素添加参数.
gcd (x1, x2, /[, out, where, casting, order, ...])	返回 <code> x1 </code> 和 <code> x2 </code> 的最大公约数
lcm (x1, x2, /[, out, where, casting, order, ...])	返回 <code> x1 </code> 和 <code> x2 </code> 的最小公倍数

2 三角函数

所有三角函数都是使用弧度。例如：弧度 a 转成角度 α , $\alpha = a * 180^\circ / \pi$

sin (x, /[, out, where, casting, order, ...])	三角正弦,逐元素计算.
cos (x, /[, out, where, casting, order, ...])	余弦逐元素计算.
tan (x, /[, out, where, casting, order, ...])	逐元素计算切线.
arcsin (x, /[, out, where, casting, order, ...])	逐元素计算反正弦.
arccos (x, /[, out, where, casting, order, ...])	逐元素计算三角反余弦.
arctan (x, /[, out, where, casting, order, ...])	三角反切,逐元素计算.
arctan2 (x1, x2, /[, out, where, casting, ...])	逐元素计算 $x1/x2$ 的反正切值,并正确选择象限.
hypot (x1, x2, /[, out, where, casting, ...])	给定一个直角三角形的"边",返回其斜边.
sinh (x, /[, out, where, casting, order, ...])	双曲正弦,逐元素计算.
cosh (x, /[, out, where, casting, order, ...])	双曲余弦,逐元素计算.
tanh (x, /[, out, where, casting, order, ...])	逐元素计算双曲正切.
arcsinh (x, /[, out, where, casting, order, ...])	逐元素计算反双曲正弦.
arccosh (x, /[, out, where, casting, order, ...])	逐元素的反双曲余弦.
arctanh (x, /[, out, where, casting, order, ...])	逐元素计算反双曲正切.
degrees (x, /[, out, where, casting, order, ...])	将角度从弧度转换为度.
radians (x, /[, out, where, casting, order, ...])	将角度从度转换为弧度.
deg2rad (x, /[, out, where, casting, order, ...])	将角度从度转换为弧度.
rad2deg (x, /[, out, where, casting, order, ...])	将角度从弧度转换为度.

3 位操作函数

这些函数都需要整数参数

bitwise_and (x1, x2, /[, out, where, ...])	计算两个数组元素按位的与操作.
bitwise_or (x1, x2, /[, out, where, casting, ...])	计算两个数组按元素逐位的或运算.
bitwise_xor (x1, x2, /[, out, where, ...])	计算两个数组元素按位异或的结果.
invert (x, /[, out, where, casting, order, ...])	按元素计算位反转,或按位非.
left_shift (x1, x2, /[, out, where, casting, ...])	将整数的位向左移动.
right_shift (x1, x2, /[, out, where, ...])	将整数的位向右移动.

4 比较函数

<code>greater(x1, x2, /[, out, where, casting, ...])</code>	返回逐元素比较 ($x1 > x2$) 的真值.
<code>greater_equal(x1, x2, /[, out, where, ...])</code>	返回 ($x1 \geq x2$) 的元素级真值.
<code>less(x1, x2, /[, out, where, casting, ...])</code>	返回 ($x1 < x2$) 的元素级真值.
<code>less_equal(x1, x2, /[, out, where, casting, ...])</code>	返回 ($x1 \leq x2$) 的元素级真值.
<code>not_equal(x1, x2, /[, out, where, casting, ...])</code>	逐元素返回 ($x1 \neq x2$).
<code>equal(x1, x2, /[, out, where, casting, ...])</code>	逐元素返回 ($x1 == x2$).
<code>maximum(x1, x2, /[, out, where, casting, ...])</code>	数组元素的逐元素最大值.
<code>minimum(x1, x2, /[, out, where, casting, ...])</code>	数组元素的逐元素最小值.
<code>fmax(x1, x2, /[, out, where, casting, ...])</code>	数组元素的逐元素最大值.
<code>fmin(x1, x2, /[, out, where, casting, ...])</code>	数组元素的逐元素最小值.

⚠ *NumPy*中的逐元素布尔运算:

- 1、不要使用Python关键字 `and` 和 `or` 来组合逻辑数组表达式, 这些关键字将测试整个数组的真值, 而不是你期望的逐元素运算, 这时你必须改用按位运算符 `&` 和 `|`
- 2、`(a>2) & (a<5)` 是正确的语法, 因为 `a>2 & a<5` 会导致错误, `2 & a` 会先被计算

<code>logical_and(x1, x2, /[, out, where, ...])</code>	计算 $x1$ 和 $x2$ 的逐元素逻辑与值.
<code>logical_or(x1, x2, /[, out, where, casting, ...])</code>	计算 $x1$ 或 $x2$ 的元素级真值.
<code>logical_xor(x1, x2, /[, out, where, ...])</code>	计算 $x1$ 和 $x2$ 的按元素异或值.
<code>logical_not(x, /[, out, where, casting, ...])</code>	计算 NOT x 的元素级真值

5 浮动函数

请记住,所有这些函数都是逐元素对数组进行操作,返回一个数组输出.描述仅详细说明了一个操作.

isfinite (x, /[, out, where, casting, order, ...])	测试逐元素是否为有限（不是无穷大且不是非数字）。
isinf (x, /[, out, where, casting, order, ...])	测试逐元素是否为正无穷或负无穷。
isnan (x, /[, out, where, casting, order, ...])	测试逐元素是否为 NaN 并返回结果作为布尔数组。
isnat (x, /[, out, where, casting, order, ...])	逐元素测试是否为 NaT（不是时间）,并以布尔数组形式返回结果。
fabs (x, /[, out, where, casting, order, ...])	计算逐元素的绝对值。
signbit (x, /[, out, where, casting, order, ...])	返回元素级 True,其中符号位已设置（小于零）。
copysign (x1, x2, /[, out, where, casting, ...])	逐元素地将 x1 的符号改为 x2 的符号。
nextafter (x1, x2, /[, out, where, casting, ...])	返回 x1 向 x2 方向的下一个浮点值,逐元素进行。
spacing (x, /[, out, where, casting, order, ...])	返回 x 与其最近的相邻数之间的距离。
modf (x[, out1, out2], / [[, out, where, ...])	返回数组的分数部分和整数部分,逐元素进行。
ldexp (x1, x2, /[, out, where, casting, ...])	返回 $x1 * 2^{x2}$,逐元素计算。
frexp (x[, out1, out2], / [[, out, where, ...])	将 x 的元素分解为尾数和二的指数。
fmod (x1, x2, /[, out, where, casting, ...])	返回逐元素除法的余数。
floor (x, /[, out, where, casting, order, ...])	返回输入的向下取整结果,逐元素进行。
ceil (x, /[, out, where, casting, order, ...])	返回输入的上限,逐元素进行。
trunc (x, /[, out, where, casting, order, ...])	返回输入的截断值,逐元素进行。

(4) 例子

35. How to compute $((A+B)*(-A/2))$ in place (without copy)? (★★☆)

可以将计算结果存到out参数中，这样不会开辟新的内存空间

```
A = np.ones(15)*2
B = np.ones(15)*3
np.add(A, B, out=B)
np.divide(A, 2, out=A)
np.negative(A, out=A)
np.multiply(A, B, out=A)
```

```
>>> print(A)
```

```
[-5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5. -5.]
```


二、线性代数(*linera algebra*)

向量和矩阵的范数

(1) 向量的范数

L^p -范数定义

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p} \quad (p \geq 1)$$

L^1 -范数 (曼哈顿范数)

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

L^2 -范数 (欧几里得范数)

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

L^∞ -范数

$$\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |x_i|$$

(2) 矩阵的范数

L^1 范数 (列和范数)

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$$

L^∞ 范数 (行和范数)

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$$

L^2 范数 (谱范数) 最大奇异值

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^T A)}$$

矩阵的核范数 (奇异值矩阵的迹)

$$\|A\|_* = \text{tr} \left(\sqrt{A^T A} \right) = \sum_{i=1}^r \sigma_i(A)$$

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}$$

矩阵乘法的理解

1 A的行向量和B的列向量作内积

$$AB = \begin{bmatrix} \text{---} a_1^T \text{---} \\ \text{---} a_2^T \text{---} \\ \vdots \\ \text{---} a_m^T \text{---} \end{bmatrix} \begin{bmatrix} | & | & \cdots & | \\ b_1 & b_2 & \cdots & b_n \\ | & | & \cdots & | \end{bmatrix} = \begin{bmatrix} a_1^T b_1 & a_1^T b_2 & \cdots & a_1^T b_n \\ a_2^T b_1 & a_2^T b_2 & \cdots & a_2^T b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m^T b_1 & a_m^T b_2 & \cdots & a_m^T b_n \end{bmatrix}$$

2 A的列向量和B的行向量作外积

$$AB = \begin{bmatrix} | & | & \cdots & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & \cdots & | \end{bmatrix} \begin{bmatrix} \text{---} b_1^T \text{---} \\ \text{---} b_2^T \text{---} \\ \vdots \\ \text{---} b_n^T \text{---} \end{bmatrix} = \sum_{i=1}^n a_i b_i^T$$

3 列向量的组合

$$AB = A \begin{bmatrix} | & | & \cdots & | \\ b_1 & b_2 & \cdots & b_n \\ | & | & \cdots & | \end{bmatrix} = \begin{bmatrix} | & | & \cdots & | \\ Ab_1 & Ab_2 & \cdots & Ab_n \\ | & | & \cdots & | \end{bmatrix}$$

4 行向量的组合

$$AB = \begin{bmatrix} \text{---} a_1^T \text{---} \\ \text{---} a_2^T \text{---} \\ \vdots \\ \text{---} a_m^T \text{---} \end{bmatrix} B = \begin{bmatrix} \text{---} a_1^T B \text{---} \\ \text{---} a_2^T B \text{---} \\ \vdots \\ \text{---} a_m^T B \text{---} \end{bmatrix}$$

矩阵的分块

行分块：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

列分块：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

代码实现：

```
A = np.arange(15).reshape(5, 3)
print(A)
#行分块
for row in range(A.shape[0]):
    print(A[row])
```

```
#列分块
```

```
for col in range(A.shape[1]):  
    print(A[:, col])
```

```
[[ 0  1  2]  
 [ 3  4  5]  
 [ 6  7  8]  
 [ 9 10 11]  
 [12 13 14]]  
[0 1 2]  
[3 4 5]  
[6 7 8]  
[ 9 10 11]  
[12 13 14]  
[ 0  3  6  9 12]  
[ 1  4  7 10 13]  
[ 2  5  8 11 14]
```

#正交矩阵

正交矩阵对应欧几里得空间中的**旋转或反射**变换

1 定义

矩阵 $Q \in \mathbb{R}^{n \times n}$ 是正交矩阵，当且仅当：

$$QQ^T = Q^TQ = I_n$$

2 性质

①向量经过正交变换后，其长度保持不变

对任意的向量 $\mathbf{v} \in \mathbb{R}^n$ ，均有

$$\|Q\mathbf{v}\|_2 = \|\mathbf{v}\|_2$$

②两个向量经过相同的正交变换后，两向量夹角保持不变

对任意的向量 $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ ，均有

$$\text{两向量的夹角 } \theta = \langle \mathbf{u}, \mathbf{v} \rangle = \langle Q\mathbf{u}, Q\mathbf{v} \rangle$$

③行列式为 ± 1

$\det(Q) = 1$: 旋转矩阵，方向不变

$\det(Q) = -1$: 反射矩阵，方向反转

④对于正交矩阵的列（行）向量

$$\mathbf{q}_i^T \mathbf{q}_j = \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

⑤其他性质

$$Q^{-1} = Q^T$$

特征值满足 $|\lambda| = 1$

3 简单的例子：2维正交矩阵

矩阵旋转；从空间上看，二维空间的坐标轴旋转了 θ 角度

$$Q = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}, \quad \det(Q) = 1$$

矩阵反射；从空间上看，相当于翻转坐标轴，即原来第一象限的向量被反射到了第四象限

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \det(Q) = -1$$

特征值和特征向量

对于 $n \times n$ 的方阵 A ，若存在非零向量 \mathbf{v} 和标量 λ 满足：

$$A\mathbf{v} = \lambda\mathbf{v}$$

则：

- λ 称为矩阵 A 的特征值
- \mathbf{v} 称为 特征值 λ 所对应的 特征向量

奇异值分解 (SVD)

定义

对于任意矩阵 $A \in \mathbb{R}^{m \times n}$ ，其SVD分解为：

$$A = U\Sigma V^T$$

其中：

- $U \in \mathbb{R}^{m \times m}$ 是左奇异向量矩阵（正交矩阵， $U^T U = I$ ）
- $\Sigma \in \mathbb{R}^{m \times n}$ 是奇异值矩阵（对角矩阵，非负降序排列）
- $V \in \mathbb{R}^{n \times n}$ 是右奇异向量矩阵（正交矩阵， $V^T V = I$ ）

A 正交矩阵U和V

$U \in \mathbb{R}^{m \times m}$ 是左奇异向量矩阵，它的每一列是 AA^T 的特征向量，其第 i 列是特征值 λ_i 对应的特征向量

$V \in \mathbb{R}^{n \times n}$ 是右奇异向量矩阵，它的每一列是 $A^T A$ 的特征向量，其第 i 列是特征值 λ_i 对应的特征向量

也就是 V^T 的每一行是 $A^T A$ 的特征向量

B Σ 形如：

$$m \geq n \quad \Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \quad \text{其中 } \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

$$m \leq n \quad \Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_m \end{bmatrix} \quad \text{其中 } \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m \geq 0$$

σ_i 是对称矩阵 AA^T 特征值的非负平方根，也是 $A^T A$ 特征值的非负平方根，即 $\sigma_i = \sqrt{\lambda_i} \ (1 \leq i \leq \min(m, n))$

#一些性质

1 AA^T 和 $A^T A$ 的**非零**特征值相等

2 AA^T 和 $A^T A$ 的特征值等于矩阵 A 的奇异值的平方

证明第 1 点:

1、假设 AA^T 的特征值为 λ , 特征值 λ 对应的特征向量为 \mathbf{v} , 则根据定义有

$$AA^T \mathbf{v} = \lambda \mathbf{v} (\lambda \neq 0)$$

$$\text{两边左乘 } A^T: \quad A^T AA^T \mathbf{v} = \lambda A^T \mathbf{v}$$

$$\text{可以看成:} \quad A^T A(A^T \mathbf{v}) = \lambda(A^T \mathbf{v})$$

即: $A^T A$ 的特征值也是 λ , 对应的特征向量为 $A^T \mathbf{v}$

2、假设 $A^T A$ 的特征值为 μ , 特征值 μ 对应的特征向量为 \mathbf{x} , 则根据定义有

$$A^T A \mathbf{x} = \mu \mathbf{x} (\mu \neq 0)$$

$$\text{两边左乘 } A: \quad AA^T A \mathbf{x} = \mu A \mathbf{x}$$

$$\text{可以看成:} \quad AA^T(A \mathbf{x}) = \mu(A \mathbf{x})$$

即: AA^T 的特征值也是 μ , 对应的特征向量为 $A \mathbf{x}$

证明第 2 点:

1、已知矩阵 $A \in \mathbb{R}^{m \times n}$ 可以进行奇异值分解, 则根据定义有

$$A^T A = (U \Sigma V^T)^T U \Sigma V^T = V \Sigma^T U^T U \Sigma V^T$$

$$\text{因为 } U^T U = I, \Sigma^T \Sigma = \Sigma^2 = \Lambda_{N \times N}$$

$$\text{所以 } A^T A = V \Sigma^2 V^T = V \Lambda V^T$$

即: $A^T A$ 可相似对角化, 它的特征值是 $\Lambda_{N \times N}$ 每一个对角线元素, 也是 Σ 中对角元素 σ_i 的平方, 所对应的特征向量为 V 的每一列

2、已知矩阵 $A \in \mathbb{R}^{m \times n}$ 可以进行奇异值分解, 则根据定义有

$$A A^T = U \Sigma V^T (U \Sigma V^T)^T = U \Sigma V^T V \Sigma^T U^T$$

$$\text{因为 } V^T V = I, \Sigma \Sigma^T = \Sigma^2 = \Lambda_{M \times M}$$

$$\text{所以 } A A^T = U \Sigma^2 U^T = U \Lambda U^T$$

即: $A A^T$ 可相似对角化, 它的特征值是 $\Lambda_{M \times M}$ 每一个对角线元素, 也是 Σ 中对角元素 σ_i 的平方, 所对应的特征向量为 U 的每一列

#应用: 图像压缩


原理: 低秩近似 $A_{m \times n} (r(A) \leq \min(m, n)) \rightarrow A_{m \times n} (r(A) = k, k \text{ 个奇异值})$

核心: 选取矩阵 A 的 k 个奇异值, 重新构建矩阵 A , 其中 $k \leq \min(m, n)$

重构矩阵的方法: $A_{m \times n} = U_{m \times k} \Sigma_{k \times k} V_{k \times n}^T$

! 两个重要结论:

- 奇异值为0 \iff 矩阵 A 中存在线性相关的行或列
- 矩阵的秩 = 非零奇异值的数量 (np.linalg.matrix_rank的底层计算方法)

 下面来举一个具体例子, 有几个函数需要提前说明:

plt.imshow(ndarray, cmap)

- 作用: 将2d数组 (矩阵) 或图像数组 可视化图像
- 输入: 接受 np.ndarray 或类似的数组结构 (形状可以是 (M, N) 灰度图或 (M, N, 3) 的彩色图)
- cmap: 颜色映射
- 不直接显示图像: 它只是将数据加载到当前的 Axes 对象中, 等待后续渲染

plt.show()

将所有已定义的绘图命令 (如 imshow()、plot() 等) 渲染到屏幕上

plt.figure(figsize, dpi, facecolor, edgecolor)

创建一个新的图形窗口 (Figure 对象), 作为所有绘图元素的顶级容器。一个 Figure 可以包含多个子图 (Axes)

参数:

① `figsize: array_like`

图形的尺寸 (宽度, 高度), 以英寸为单位, 如 (2, 3)

② `dpi: float, 可选`

分辨率 (每英寸像素数), 默认100dpi

③ `facecolor: ColorType, 可选`

图形背景色

④ `edgecolor: ColorType, 可选`

图形边框颜色

返回值:

➡ 返回Figure对象

`plt.subplot(nrows, ncols, index)`

在 Figure 中创建或选择一个子图 (Axes 对象), 用于绘制具体内容

通过网格定位子图, 将图形划分为 `nrows` 行 × `ncols` 列, `index` 指定子图位置 (从1开始)

例如: 在 Figure(4, 6) 下 `subplot(2, 2, 1)` 表示将宽度为4高度为6的图像分割成2x2的网格, 并且在索引1的位置生成 Axes 对象, 之后通过 `plt.show()` 显示图像

代码实现:

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
# todo: 导入图片
#这时pic是一个 ImageFile
#<PIL.JpegImagePlugin.JpegImageFile image mode=L size=230x286 at 0x2744DC63610>
pic = Image.open('Einstein_tongue.jpg')

# todo: 画图: 通过plt.imshow, plt.show查看图片
plt.imshow(pic)                                #传入图像文件之后, 把图像的数据加载到当前Axes对象中
plt.imshow(pic, cmap='gray_r')                 #把传入的图像文件转成灰度图, 加载到当前的Axes对象中
plt.show()                                     #渲染之前加载过的所有文件

# 将图片转成numpy数组, 浮点数类型
pic_ndarray = np.array(pic)

# todo: 对图片转成的numpy数组 做SVD (奇异值分解)
U, S, V = np.linalg.svd(pic_ndarray)

#下面开始重构图片, 原理: 低秩近似
-----
# 定义变量k, 表示想选前k个奇异值
# 根据k值, 以及奇异值分解的结果, 去重建图片
k = 25                                          #这里选25个奇异值, 原数组分解后有230个奇异值
comps = np.arange(k)
reconPic = U[:,comps] @ np.diag(S[comps]) @ V[comps]
```

```

#           mxk           kxk           kxn

# 画图：展示原始图片 和 重建图片
plt.figure(figsize=(5,10))
plt.subplot(1,2,1)
plt.imshow(pic,cmap='gray_r')
plt.title('Original')
plt.axis('off')

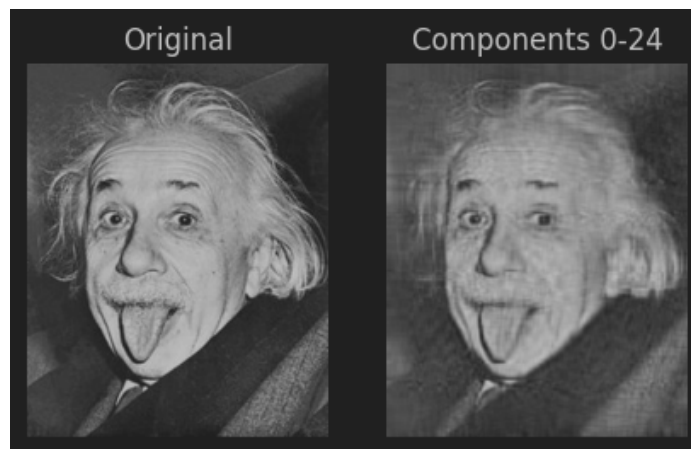
#生成Figure对象
#在Figure对象下生成子图（Axes对象），在1x2网格的索引1的位置上
#将灰度图加载到当前的Axes对象下

plt.subplot(1,2,2)
plt.imshow(reconPic,cmap='gray_r')
plt.title(f'Components {comps[0]}-{comps[-1]}')
plt.axis('off')
#在Figure对象下生成子图（Axes对象），在1x2网格的索引2的位置上
#注意，参数传的是ndarray，生成灰度图到当前的Axes对象下

#渲染所有
plt.show()

```

运行结果如下(只显示部分):



投影矩阵

定义

给定一个 $m \times n$ 的矩阵 A ，假设其列向量线性无关，即 $r(A) = n$ ，则投影矩阵 P 将任意向量 $b \in \mathbb{R}^m$ 投影到 A 的列空间 $Col(A)$ 上，得到投影向量 $p = Pb$

投影矩阵的表达式为：

$$P = A(A^T A)^{-1} A^T$$

其中：

- $A^T A$ 是可逆的，因为 A 列满秩
- P 是一个 $m \times m$ 的对称矩阵

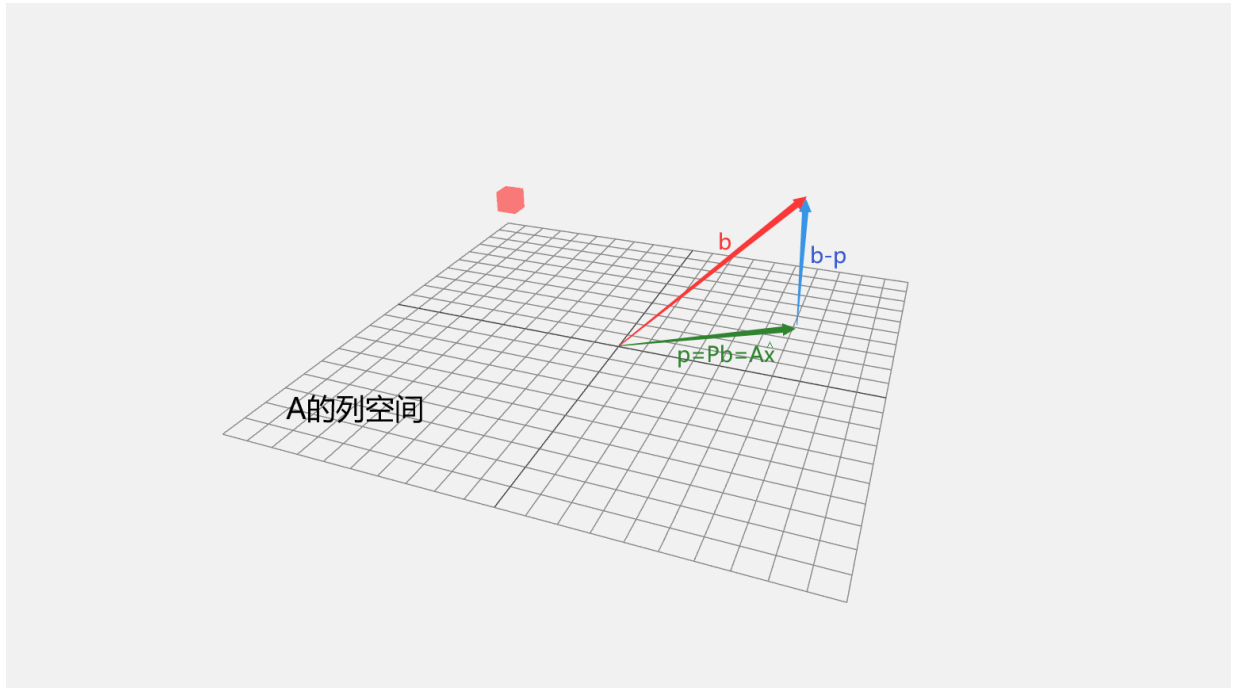
性质

- 幂等性: $P^2 = P$
- 对称性: $P^T = P$
- $r(P) = r(A) = n$

投影矩阵公式推导

给定一个 $m \times n (m > n)$ 的矩阵 $A (r(A) = n)$ 和向量 $\mathbf{b} \in \mathbb{R}^m$, 我们希望求解 $A\mathbf{x} = \mathbf{b}$, 但由于 \mathbf{b} 可能不在 A 的列空间内, 即方程无解, 这时我们就要寻找一个近似解 (最优解) $\hat{\mathbf{x}}$, 使得残差 $\|\mathbf{b} - A\hat{\mathbf{x}}\|^2$ 最小, 即向量 \mathbf{b} 与其自身投影到 A 的列空间的向量的距离最近

上述问题通过几何关系理解如下:



如图所示, 可以得到以下公式:

$$A^T(\mathbf{b} - \mathbf{p}) = 0$$

$$A^T(\mathbf{b} - A\hat{\mathbf{x}}) = 0$$

$$A^T\mathbf{b} = A^TA\hat{\mathbf{x}}$$

$$\text{因为 } |A^TA| = |A^T||A| = |A|^2 \neq 0 \ (r(A) = n)$$

$$\text{所以 } \hat{\mathbf{x}} = (A^TA)^{-1}A^T\mathbf{b}$$

所以, 我们找到了最优解 $\hat{\mathbf{x}}$, 使得残差 $\|\mathbf{b} - A\hat{\mathbf{x}}\|^2$ 最小

上式同时左乘矩阵 A , 得到

$$A\hat{\mathbf{x}} = A(A^TA)^{-1}A^T\mathbf{b} = P\mathbf{b}$$

$$\text{投影矩阵 } P = A(A^TA)^{-1}A^T$$

即: 存在投影矩阵 P , 在 $A\mathbf{x} = \mathbf{b}$ 无解的情况下, 使得向量 \mathbf{b} 到 A 的列空间的距离最短

应用: 最小二乘法

问题背景:

给定 $A \in \mathbb{R}^{m \times n} (m \geq n)$, $\mathbf{b} \in \mathbb{R}^m$, 求解 $\mathbf{x} \in \mathbb{R}^n$, 使得:

$$A\mathbf{x} \approx \mathbf{b}$$

当 $\mathbf{b} \notin \text{Col}(A)$, 即方程组无解时, 最小二乘法寻找 $\hat{\mathbf{x}}$ 最小化残差:

$$\min_x \|Ax - \mathbf{b}\|^2$$

示例代码如下：

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

house_prices_data = np.loadtxt('house.txt', delimiter=',', skiprows=1, encoding='utf-8')

# todo: 划分出和房价相关的数据 和 房价本身的数据，划分成两个numpy数组，存到两个变量里
Y = house_prices_data[:, -1]          #最后一列是因变量      mx1
X = house_prices_data[:, :-1]         #从第一列到倒数第二列是自变量  mx2
print(X)

# todo: 给房价特征数据 加一列，这一列全是1
#横向拼接，对应常数c1
col1 = np.ones((X.shape[0], 1))
X_add_bias = np.hstack((col1, X))      #mx3
print(X_add_bias)

# todo: 用最小二乘法拟合数据，可以基于投影矩阵，也可以用np.linalg.lstsq
#求出ci对应的矩阵
c, _, _, _ = np.linalg.lstsq(X_add_bias, Y)      #3x1
#上一行等价于 c = np.linalg.inv(X.T @ X) @ X.T @ Y
print(c)

#通过最小二乘法来预测Y
Y_pred = X_add_bias @ c

#拟合的结果
print("拟合结果: y = {:.4f} + {:.4f} * size + {:.4f} * bedrooms".format(c[0], c[1], c[2]))

new_data = np.array([75, 2])
newPrice = c[0] + c[1]*new_data[0] + c[2]*new_data[1]
print(f"size = {new_data[0]}, bedrooms={new_data[1]}的房价为: {newPrice}")

# 创建网格点用于绘制平面
x1_grid, x2_grid = np.meshgrid(np.linspace(40, 150, 10),
                                np.linspace(0, 10, 10))
y_grid = c[0] + c[1] * x1_grid + c[2] * x2_grid

# 绘制
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], Y_pred, color='red', label='Original_data')
ax.plot_surface(x1_grid, x2_grid, y_grid, alpha=0.5, color='blue', label='fitting plane')
ax.set_xlabel('size')
ax.set_ylabel('bedrooms')
ax.set_zlabel('Y_pred')
ax.legend()
plt.title('Result')
plt.show()

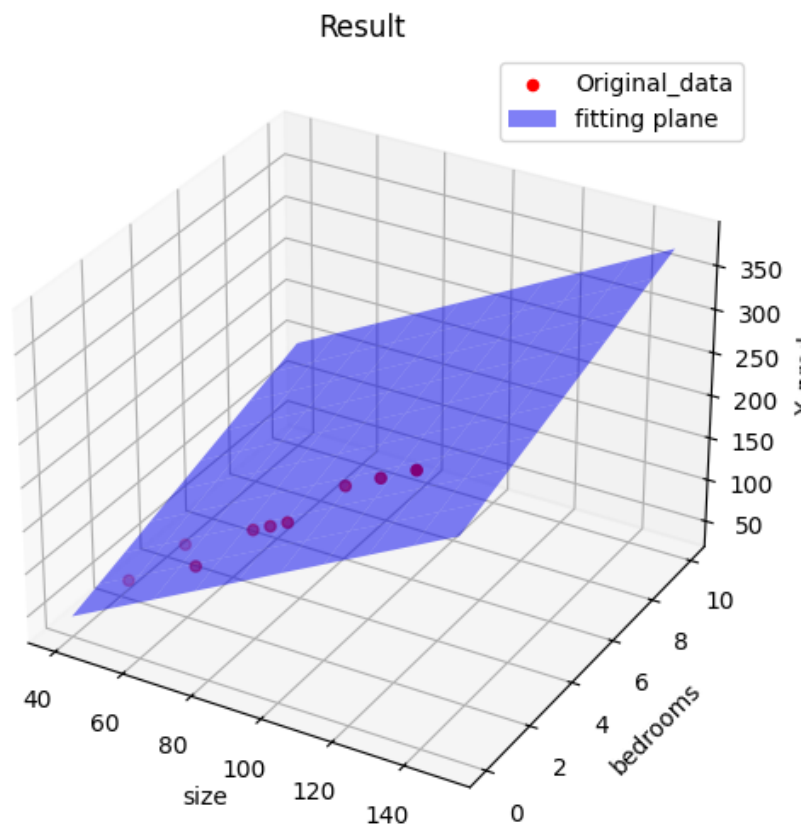
#运行结果如下
[[ 50.  1.]
 [ 60.  2.]
 [ 80.  2.]
 [100.  3.]
 [120.  3.]
 [ 90.  2.]
```

```

[ 70.  1.]
[ 85.  2.]
[110.  3.]
[[ 1.  50.  1.]
 [ 1.  60.  2.]
 [ 1.  80.  2.]
 [ 1. 100.  3.]
 [ 1. 120.  3.]
 [ 1.  90.  2.]
 [ 1.  70.  1.]
 [ 1.  85.  2.]
 [ 1. 110.  3.]]
[-40.20373514  2.08715337 10.27164686]
拟合结果:  $y = -40.2037 + 2.0872 * \text{size} + 10.2716 * \text{bedrooms}$ 
size = 75,bedrooms=2的房价为: 136.8760611205432

```

拟合结果: $y = -40.2037 + 2.0872 * \text{size} + 10.2716 * \text{bedrooms}$



numpy.dot

numpy.dot(a, b, out=None)

两个数组的点积（内积）

- 如果a和b都是一维数组，则结果是向量的内积，等价于 `a @ b`
- 如果a和b都是二维数组，这是矩阵乘法，但矩阵乘法首选是 `matmul` 或 `a @ b`
- 如果a或b是标量，则使用 `a * b` 是首选
- 如果a是一个N维数组，而b是一个一维数组，它是a的最后一个轴和b的乘积和
- 如果a是一个N维数组，而b是一个M维数组 ($M \geq 2$)，它是a的最后一个轴和b的倒数第二个轴的和积：

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

参数:

①a:array_like

第一个参数

②b: array_like

第二个参数

③out: ndarray, 可选

输出参数

返回值:

➡ output: ndarray

返回a和b的点积

$$x \cdot y = x^T y = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

numpy.outer

numpy.outer(a, b)

numpy.linalg.outer(x1, x2)

计算两个向量的外积，它把a/x1当成了列向量，b/x2当成了行向量，所以其结果是个矩阵

$$xy^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}$$

numpy.linalg.norm

numpy.linalg.norm(x, ord=None, axis=None, keepdims=False)

矩阵或向量的范数

参数:

①x: array_like

输入数组，如果axis是None，必须是一维或二维，除非ord=None。如果axis和ord都是None，将返回x.ravel的2-范数

②ord: {int, float, inf, -inf, 'fro', 'nuc'}, 可选

范数的顺序，见下面的表格

③axis: {None, int, 两个整数的元组}, 可选

如果axis是一个整数，它指定沿x的哪个轴计算向量范数。如果axis是一个2元组，它指定保存二维矩阵的轴，并计算这些矩阵的矩阵范数。如果axis为None，则返回向量范数或矩阵范数，默认值为None

④keepdims: bool，可选

如果为True，则保持维度

返回值：

➡ n: float或ndarray

矩阵或向量的范数

ord	矩阵的范数	向量的范数
None	Frobenius 范数	向量的模长
'fro'	Frobenius 范数	—
'nuc'	核范数（奇异值矩阵的迹 $\text{tr}\left(\sqrt{A^T A}\right)$ ）	—
inf	$\max(\text{sum}(\text{abs}(x), \text{axis}=1))$ 行和范数	$\max(\text{abs}(x))$
-inf	$\min(\text{sum}(\text{abs}(x), \text{axis}=1))$	$\min(\text{abs}(x))$
0	-	$\text{sum}(x \neq 0)$
1	$\max(\text{sum}(\text{abs}(x), \text{axis}=0))$ 列和范数	通过定义计算
-1	$\min(\text{sum}(\text{abs}(x), \text{axis}=0))$	通过定义计算
2	谱范数（最大奇异值）	通过定义计算
-2	最小的奇异值	通过定义计算

numpy.cross

numpy.cross(a, b)

返回两个向量的叉乘，其结果是一个垂直于a和b的向量

numpy.diag

numpy.diag(v, k=0)

提取对角线或构造对角线数组

参数：

①v: array_like

如果v是二维数组，则返回其第k条对角线的副本。如果v是一维数组，则返回v位于第k条对角线上的二维数组

②k: int，可选

默认值为0，指的是主对角线，对于主对角线上方的对角线，使用 $k>0$ ，下方的对角线，使用 $k<0$

返回值:

➡ out: ndarray

提取或者构造的对角线数组

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

```
a = np.arange(9).reshape((3, 3))
print("=====原数组=====")
print(a)
print("=====#提取主对角线元素=====")
print(np.diag(a))
print("=====#提取主对角线以上的第一个对角线元素=====")
print(np.diag(a, 1))
print("=====#提取主对角线以下的第一个对角线元素=====")
print(np.diag(a, -1))
print("=====创建对角矩阵=====")
b = np.diag([1, 5, 6])
print(b)
```

#运行结果如下

```
=====原数组=====
[[0 1 2]
 [3 4 5]
 [6 7 8]]
=====#提取主对角线元素=====
[0 4 8]
=====#提取主对角线以上的第一个对角线元素=====
[1 5]
=====#提取主对角线以下的第一个对角线元素=====
[3 7]
=====创建对角矩阵=====
[[1 0 0]
 [0 5 0]
 [0 0 6]]
```

numpy.fill_diagonal

numpy.fill_diagonal(a, val, wrap=False)

使用val就地填充给定任意维度的数组a的主对角线，对角线是值 $a[i, \dots, i]$ 的列表

参数:

①a: 数组，至少是二维

要就地填充对角线的数组

②val: 标量或array_like

对角线的值。

如果val是标量，则矩阵a的对角线都是val。

如果val是数组，则展平的val填充a的对角线的每个 `a[i,...,i]`；

- 必要时重复，也就是说当 $A_{m \times n} (n > \text{length}(\text{val}))$ 时，会重复 val 数组中的元素以填充对角线
- val数组的元素不一定会填充，也就是说当 $A_{m \times n} (n < \text{length}(\text{val}))$ 时，只会填充A的主对角线上的前 n 个元素

③wrap: bool

此参数仅影响高矩阵（即： $A_{m \times n} (m \geq n)$ ），如果为True,则数据在 n 行后继续沿着主对角线方向填充数据，即填充的第一个位置为 `a[0, 0]`，则第二个位置为 `a[n+1, n+1]`，从此沿主对角线方向继续填充

返回值：

➡ 返回None：就地填充数组

```
a = np.zeros((3, 3), int)
np.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])

a = np.zeros((6, 8), int)
val_arr = np.array([1,2,3,4])
np.fill_diagonal(a, val_arr)
>>> print(a)                                     #重复val_arr的元素填充其对角线
[[1 0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0]
 [0 0 3 0 0 0 0 0]
 [0 0 0 4 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 2 0 0]]

a = np.zeros((5, 3), int)
val_arr = np.array([1,2,3,4])
np.fill_diagonal(a, val_arr)
>>> print(a)                                     #可以看的val_arr中的数据并不会全部填充，只填充n行
[[1 0 0]
 [0 2 0]
 [0 0 3]
 [0 0 0]
 [0 0 0]]

a = np.zeros((12, 3), int)
np.fill_diagonal(a, 4, wrap=True)
>>> print(a)
[[4 0 0]
 [0 4 0]
 [0 0 4]
 [0 0 0]
 [4 0 0]
 [0 4 0]
```

```
[0 0 4]
[0 0 0]
[4 0 0]
[0 4 0]
[0 0 4]
[0 0 0]]
```

numpy.eye

numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', device=None, like=None)

返回一个对角线上为1（默认主对角线），其他位置为0的二维数组

参数：

①N: int

输出中的行数

②M: int, 可选

输出中的列数，如果为None,默认为N

③K: int, 可选

对角线索引：0（默认值），指的是主对角线，正值指的是上对角线，负值指的是下对角线。

④dtype: 数据类型, 可选

返回数组的数据类型

⑤order: {'C','F'}, 可选

输出在内存中应存储为行优先还是列优先顺序

⑥device: str, 可选

要放置创建数组的设备，默认值None，仅用于数组API互操作性，因此如果传递，必须为 "cpu"

⑦like: array_like, 可选

引用对象以允许创建不是NumPy数字的数组，生成的数组结果类似传入的数组

返回值：

➡ I: 形状为 (N, M) 的ndarray

一个数组，其中所有元素都等于0，除了第K条对角线，其值等于1

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

```
print(np.eye(5,k=1))
```

```
print(np.eye(5,k=-1))
```

```
print(np.eye(6, dtype=np.int32))
```

#生成5x5的矩阵，其第一个上对角线的元素是1

#生成5x5的矩阵，其第一个下对角线的元素是1

#生成6x6的单位矩阵

#运行结果如下

```
[[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
[[0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

numpy.linalg.trace

numpy.linalg.trace(x, offset=0, dtype=None)

返回矩阵（或矩阵堆栈） x 沿指定轴的和。

如果矩阵是三维或者更高维度的，**numpy.linalg.trace** 与 **np.trace** 不同，前者的迹是根据最后两个轴计算的，后者默认是使用前两个轴计算的

参数：

①x: (...M,N) array_like

具有形状(..., M, N)的输入数组，其最内层的两个维度形成MxN矩阵

②offset: int, 可选

偏移量指定相对于主对角线的位置，其中：

- offset = 0, 主对角线
- offset > 0, 主对角线上方的对角线，offset=n 代表主对角线上方的第n条对角线
- offset < 0, 主对角线下方的对角线，offset=n 代表主对角线下方的第n条对角线

③dtype: dtype, 可选

返回数组的类型

返回值：

→ out: ndarray

一个包含迹的数组

```
>>> np.linalg.trace(np.eye(3))          #默认主对角线
3.0
```

#可以看出两个函数返回值不同

#np.linalg.trace是根据最后两个轴计算的，而np.trace是根据前两个轴计算的

```
>>> a = np.arange(8).reshape((2, 2, 2))
```

```

>>> a
[[[0 1]
  [2 3]]

  [[4 5]
  [6 7]]]
>>> np.linalg.trace(a)
array([3, 11])           #0+3=3 4+7=11

>>>> np.trace(a)
array([6, 8])           #0+6=6 1+7=8

#使用offset参数
a = np.arange(9).reshape((3, 3)); a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
np.linalg.trace(a, offset=1)   # 1+5=6
6
np.linalg.trace(a, offset=2)   # 2
2
np.linalg.trace(a, offset=-1)  # 3+7=10
10
np.linalg.trace(a, offset=-2)  # 6
6

```

numpy.matmul

numpy.linalg.matmul(x1, x2)

numpy.matmul

矩阵乘法，等同于 `x1 @ x2`，注意x1和x2的顺序不能颠倒

numpy.linalg.matrix_rank

numpy.linalg.matrix_rank(A)

返回矩阵A的秩

numpy.linalg.solve

numpy.linalg.solve(a, b)

计算线性方程组 $Ax=b$ (也是 $ax=b$) 的解，仅有唯一解时才返回结果，无解或无穷多解时函数引发[`LinAlgError: Singular matrix`]]

numpy.column_stack

numpy.column_stack(tup)

将一维数组作为列堆叠成二维数组，就像使用 `hstack` 一样

numpy.linalg.det

numpy.linalg.det(a)

计算矩阵A的行列式

numpy.linalg.inv

numpy.linalg.inv(a)

计算矩阵的逆

$$\begin{aligned} A_{nn} \text{可逆的充分必要条件为 } |A| &\neq 0 \\ \iff r(A) &= n (\text{满秩}) \\ \iff A \text{中有 } n \text{个线性无关的行(列)向量} \\ \iff Ax = b \text{有唯一解, } Ax = 0 &\text{仅有 } 0 \text{解} \\ \iff A \text{的特征值均不为 } 0 \end{aligned}$$

$$\text{二阶矩阵求逆: } A^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$n \text{阶矩阵求逆: } A^{-1} = \frac{1}{|A|} A^*$$

numpy.linalg.eig

numpy.linalg.eig(a)

计算一个方阵a的特征值和特征向量

➡ 返回一个命名元组 (NamedTuple) ,其属性是 eigenvalues 和 eigenvectors

A eigenvalues: 特征值数组

B eigenvectors: 单位化的特征向量, 并且第 i 列 eigenvectors[:,i] 是对应于特征值 eigenvalues[i] 的特征向量

```
eigvals, eigvcts = np.linalg.eig(np.diag((1, 2, 3)))
>>> eigvals
array([1., 2., 3.])
>>> eigvcts
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

#这里可以使用访问属性名(.)的方式的原因在于,

#np.linalg.eig返回的是一个NamedTuple对象, 这种命名元组不仅可以通过下标访问, 还可以通过属性名的方式访问

#np.linalg.eig其内部实现是继承了Python中的class typing.NamedTuple

```
>>> np.linalg.eig(np.diag((1, 2, 3))).eigenvalues
array([1., 2., 3.])
>>> np.linalg.eig(np.diag((1, 2, 3))).eigenvectors
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

numpy.linalg.svd

numpy.linalg.svd(a)

矩阵A的奇异值分解, $A = U\Sigma V^T$

➡ 返回值: ① U ② Σ ③ V^T

numpy.linalg.lstsq

numpy.linalg.lstsq(a, b)

返回线性方程组 $Ax = b$ 的最小二乘解。

参数:

① a: (M, N) array_like

系数矩阵

② b: {(M,), (M, K)}, array_like

纵坐标或因变量的值, 如果 b 是二维的, 则对 b 的每一列 K 计算最小二乘解

返回值:

① x: {(N,), (N,K)} ndarray

最小二乘解, 如果b是二维的, 解在x的K列中

② residuals: ndarray

残差平方和 $\|Ax - b\|^2$

③ rank: int

矩阵(A) 的秩

④ s: ndarray

A的奇异值

#常用函数

ndarray创建

numpy.arange

`numpy.arange([start=0, stop, [step=1,]dtype=None, device=None, like=None)`

返回ndarray， ndarray里面是给定区间内均匀间隔的值

arange 可以调用不同数量的位置参数：

- `arange(stop)`：值在半开区间 `[0, stop)` 内生成
- `arange(start, stop)`：值在半开区间 `[start, stop)` 内生成
- `arange(start, stop, step)`：值在半开区间 `[start, stop)` 内生成，值之间的间距由 `step` 给出

对于整数参数，该函数大致等同于Python内置的 `range`，但返回的是一个ndarray而不是range实例。

参数：

①start：整数或实数，可选

默认开始值为0，区间开始，包括此值

②stop：整数或实数

区间结束，区间不包括这个值，除非在某些情况下，`step` 不是整数并且浮点数舍入影响out的长度

③step：整数或实数，可选

值之间的间距，对于任何输出out，这是两个相邻值之间的距离，`out[i+1]-out[i]`，默认步长为1。如果step被指定为位置参数，`start` 也必须给出

④dtype: dtype 可选

输出数组的类型，如果未指定 `dtype`，则从其他输入参数推断数据类型

⑤device: str可选

创建的数组放置的设备，默认为 `None`，仅用于数组API的互操作性，因此如果传递，必须为 `"cpu"`

| [Add in 2.0.0 version](#)

⑥like: array_like, 可选

引用对象以允许创建不是numpy数组的数组。

| [Add in 1.20.0 version](#)

返回值：

➡ arange: ndarray

均匀间隔值的数组。

对于浮点数参数，结果的长度是 `ceil((stop - start)/step)`。由于浮点数溢出，此规则可能导致out的最后一个元素大于stop。

[注意：] `numpy.arange` 生成 `numpy.int32` 或 `numpy.int64` 数字，这可能导致对大整数值的结果不正确，而Python中内置的 `range` 生成的数不会溢出

```
import numpy as np
power = 40
modulo = 10000
a_array = [ (n ** power) % modulo for n in range(8)]
b_array = np.array([(n ** power) % modulo for n in np.arange(8)])
>>> print(a_array)
>>> print(b_array)

[0, 1, 7776, 8801, 6176, 625, 6576, 4001]      #正确结果
[ 0   1 7776 7185   0 5969 4816 3361]        #错误结果, 因为numpy里面int数据最大为, int64
```

下面是具体的例子:

```
import numpy as np
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

numpy.linspace

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0, device=None)

在指定区间内返回均匀间隔的数字

返回num个均匀间隔的样本, 计算在区间[start, stop]上, 区间的端点可以选择性排除

参数

①start: array_like

序列的起始值

②stop: array_like

序列的结束值, 除非endpoint设置为False,在这种情况下, 序列由 `num + 1` 个均匀间隔的样本中除最后一个外的所有样本组成, 因此stop被排除在外。请注意, 当endpoint为False时, 步长会改变。

③num: int

要生成的样本数量, 默认值是50, 必须是非负数

④endpoint: bool, 可选

默认为True。如果为False, `stop` 不包括在内

⑤retstep: bool, 可选

如果为真, 返回 `(samples, step)`, 其中step是样本之间的间距。

⑥dtype: dtype, 可选

输出数组的类型, 如果未指定 `dtype`, 则数据类型从start和stop推断, 推断的dtype永远不会是整数; 即使参数会产生整数数组, 也会选择float

| Add in 1.9.0 version

⑦axis: int, 可选

结果中存储样本的轴，仅在`start`或`stop`是类数组时相关，默认情况下是0，样本将沿一个新的轴插入到开头，使用-1以在末尾获取一个轴

| Add in 1.16.0 version

⑧`device`: str, 可选

要放置创建数组的设备，默认值为None，仅用于数组API互操作性，因此如果传递，必须为"cpu"

| Add in 2.0.0 version

返回值:

→ `samples`: ndarray

在闭区间 `[start, stop]` 或半开区间 `[start, stop)` 中有`num`个等距的样本（取决于`endpoint`是True还是False）

→ `step`: 浮点数, 可选

样本之间的间距大小，仅在`retstep`为True时返回。

例子:

```
print(np.linspace(0, 1, 5))
print(np.linspace(0, 1, num=5, endpoint=False))
print(np.linspace(0, 1, num=5, retstep=True))

#运行结果如下
[0.  0.25 0.5  0.75 1.  ]          #返回浮点数组，即使start和stop为整数
[0.  0.2 0.4 0.6 0.8]            #endpoint=False,表示不能取stop，所以取不到1
(array([0.  , 0.25, 0.5 , 0.75, 1.  ]), np.float64(0.25))    #retstep=True,返回(samples, step)
```

39、创建一个大小为10的向量，其值范围为0到1，不包括0和1

```
z = np.linspace(0,1,11,endpoint=False)[1:]    #endpoint=False，则取不到1，从第二个开始，则取不到0
print(z)

array([0.09090909, 0.18181818, 0.27272727, 0.36363636, 0.45454545,
       0.54545455, 0.63636364, 0.72727273, 0.81818182, 0.90909091])
```

numpy.array

`numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0, like=None)`

创建一个数组

参数:

①`object`: array_like

数组、任何暴露数组接口的对象、其`__array__`方法返回数组的对象，或任何嵌套的序列。如果对象是标量，则返回包含该对象的0维数组。

②`dtype`: 数据类型, 可选

数组所需的数据类型，如果未指定，numpy将尝试使用一个默认的`dtype`来表示这些值（必要时应用提升规则）

③copy: bool, 可选

如果为 `True`, 则数组数据将被复制。如果为 `None`, 只有在 `__array__` 返回一个副本、如果obj是一个嵌套序列, 或者如果需要复制以满足其他要求 (`dtype`、`order` 等) 时才会进行复制。请注意, 任何数据的复制都是浅拷贝, 对于对象的dtype的数组, 新数组将指向相同的对象。对于 `False`, 如果无法避免复制, 则会引发 `ValueError`。默认值为 `True`

④order: {'K', 'A', 'C', 'F'}, 可选

指定数组的内存布局, 如果对象不是数组, 新创建的数组将按C顺序 (行优先), 除非指定了F, 在这种情况下它将按Fortran顺序 (列优先), 如果对象是数组, 则以下规则适用:

order	no copy	copy=True
K	不可更改	F&C顺序保留, 否则保持最相似的顺序
A	不可更改	如果输入是F而不是C, 则按F顺序排列, 否则按C顺序排列
C	C顺序	C顺序
F	F顺序	F顺序

当 `copy=None` 且由于其他原因创建了一个副本时, 结果与 `copy=True` 相同。默认为 'K'。

当顺序为 'A' 且 object 是一个既不是C也不是F顺序的数组, 并且由于dtype的变化而强制复制时, 结果的顺序不一定如预期的C, 这可能是一个错误。

⑤subok: bool, 可选

如果为真, 则子类将被传递, 否则返回的数组将被强制为基类数组 (默认)

⑥ndmin: int, 可选

指定结果数组应具有的最小维度数。

⑦like: array_like, 可选

引用对象以允许创建不是NumPy数组的数组, 如果作为 like 传递的类数组对象支持 `__array_fuction__` 协议, 结果将由它定义。在这种情况下, 它确保创建一个与通过此参数传递的对象兼容的数组对象。

| Add in 1.20.0 version

返回值:

👉 out: ndarray

满足指定要求的数组对象。

例子:

```
print(np.array([1, 2, 3]))
print("====向上转换====")
print(np.array([1, 2, 3.0]))
print("====生成二维数组====")
print(np.array([[1, 2], [3, 4]]))
print("====生成复数数组====")
print(np.array([1+0j, 1+1j, 2+1j], dtype=complex))
print("====最小维度指定2维====")
print(np.array([1, 2, 3], ndmin=2))
```

#结果如下

```
[1 2 3]
```

====向上转换====

```
[1. 2. 3.]
```



```
=====生成二维数组=====
[[1 2]
 [3 4]]
=====生成复数数组=====
[1.+0.j 1.+1.j 2.+1.j]
=====最小维度指定2维=====
[[1 2 3]]
```

numpy.zeros

numpy.zeros(shape, dtype=float, order='C', like=None)

返回一个给定形状和类型的新数组，填充为零

参数：

①shape: 整数或整数的元组

新数组的形状，例如 (2, 3) 或 2

②dtype: 数据类型，可选

数组所需的数据类型，例如 `numpy.int8`。默认是 `numpy.float64`

③order: {'C', 'F'}, 可选，默认'C'

是否将多维数据在内存中按行优先（C风格）或列优先（Fortran风格）顺序存储

④like: array_like, 可选

引用对象以允许创建不是NumPy数组的数组，如果作为 like 传递的类数组对象支持 `__array_fuction__` 协议，结果将由它定义。在这种情况下，它确保创建一个与通过此参数传递的对象兼容的数组对象。

| Add in 1.20.0 version

返回值：

➡ out: ndarray

具有给定形状、dtype和顺序的零数组

numpy.empty

numpy.empty(shape, dtype=float, order='C', device=None, like=None)

返回一个具有给定形状和类型的新数组，而不初始化条目

参数：

①shape: int 或 int的元组

空数组的形状，例如 (2, 3) 或 2

②dtype: 数据类型，可选

数组期望的输出数据类型，例如 `numpy.int8`。默认是 `numpy.float64`。

③order: {'C', 'F'}, 可选，默认'C'

是否将多维数组数据在内存中按行优先（C风格）或列优先（Fortran风格）顺序存储

④device: str, 可选

要放置创建数组的设备，默认值为 `None`。仅用于数组API的互操作性，因此如果传递，必须为 `"cpu"`

| Add in 2.0.0 version

④like: array_like, 可选

引用对象以允许创建不是NumPy数组的数组，如果作为 `like` 传递的类数组对象支持 `__array_fuction__` 协议，结果将由它定义。在这种情况下，它确保创建一个与通过此参数传递的对象兼容的数组对象。

| Add in 1.20.0 version

返回值:

→ out: ndarray

给定形状、数据类型和顺序的未初始化（任意）数据的数组。对象数组将被初始化为`None`。

numpy.full

`numpy.full(shape, fill_value, dtype=None, order='C', device=None, like=None)`

返回一个具有给定形状和类型的新数组，并用`fill_value`填充

参数:

①shape: 整数或整数序列

新数组的形状，例如 `(2, 3)` 或 `2`

②fill_value: 标量或类数组

填充值

③dtype: 数据类型，可选

数字所需的数据类型，默认值`None`表示 `np.array(fill_value).dtype`

④order: {'C', 'F'}, 可选

是否将多维数组数据在内存中按行优先（C风格）或列优先（Fortran风格）顺序存储

⑤device: str, 可选

要放置创建数组的设备，默认值为 `None`。仅用于数组API的互操作性，因此如果传递，必须为 `"cpu"`

| Add in 2.0.0 version

⑥like: array_like, 可选

引用对象以允许创建不是NumPy数组的数组，如果作为 `like` 传递的类数组对象支持 `__array_fuction__` 协议，结果将由它定义。在这种情况下，它确保创建一个与通过此参数传递的对象兼容的数组对象。

| Add in 1.20.0 version

返回值:

→ ndarray

具有给定形状、dtype和顺序的fill_value数组

```
import numpy as np
zero_a = np.zeros((2, 3))
print("=====形状为 (2,3) 的全零数组, dtype=float =====")
print(zero_a)
print("=====形状为 (3,2) 的全一数组, dtype=float =====")
b = np.ones((3,2))
print(b)
c = np.full((2, 3), 5)
print("=====形状为 (2,3) 的全零数组, 填充值为5 =====")
print(c)
```

#输出结果如下

```
=====形状为 (2,3) 的全零数组, dtype=float =====
[[0. 0. 0.]
 [0. 0. 0.]]
=====形状为 (3,2) 的全一数组, dtype=float =====
[[1. 1.]
 [1. 1.]
 [1. 1.]]
=====形状为 (2,3) 的全零数组, 填充值为5 =====
[[5 5 5]
 [5 5 5]]
```

numpy.ones_like

numpy.ones_like(a, dtype=None, order='K', subok=True, shape=None, device=None)

返回一个与给定数组具有相同形状和类型的全一数组

参数:

①a: array_like

a的形状和数据类型定义了返回数组的这些相同属性

②dtype: 数据类型, 可选

覆盖结果的数据类型

| Add in 1.6.0 version

③order: {'C', 'F', 'A', 'K'}, 可选

覆盖结果的内存布局。C: 行优先, F: 列优先; A表示如果a是列连续的则为F, 否则为C; K表示尽可能匹配a的布局

| Add in 1.6.0 version

④subok: bool 可选

如果为True, 则新创建的数组将使用a的子类类型, 否则它将是一个基类数组。默认为True。

⑤shape: int或int序列, 可选

覆盖结果的形状, 如果order='K'并且维度数量不变, 将尝试保持顺序, 否则, 隐含order='C'

⑥device: str, 可选

要放置创建数组的设备。默认 `None` ,仅用于数组AI互操作性, 因此如果传递, 必须为 `"cpu"`。

| Add in 2.0.0 version

返回值:

➡ out: ndarray

与a具有相同形状和类型的全一数组

numpy.zeros_like

`numpy.zeros_like(a, dtype=None, order='K', subok=True, shape=None, device=None)`

返回一个与给定数组具有相同形状和类型的全零数组

参数:

①a: array_like

a的形状和数据类型定义了返回数组的这些相同属性

②dtype: 数据类型, 可选

覆盖结果的数据类型

| Add in 1.6.0 version

③order: {'C', 'F', 'A', 'K'}, 可选

覆盖结果的内存布局。C: 行优先, F: 列优先; A表示如果a是列连续的则为F, 否则为C; K表示尽可能匹配a的布局

| Add in 1.6.0 version

④subok: bool 可选

如果为True, 则新创建的数组将使用a的子类类型, 否则它将是一个基类数组。默认为True。

⑤shape: int或int序列, 可选

覆盖结果的形状, 如果order='K'并且维度数量不变, 将尝试保持顺序, 否则, 隐含order='C'

⑥device: str, 可选

要放置创建数组的设备。默认 `None` ,仅用于数组AI互操作性, 因此如果传递, 必须为 `"cpu"`。

| Add in 2.0.0 version

返回值:

➡ out: ndarray

与a具有相同形状和类型的全零数组

numpy.full_like

`numpy.full_like(a, fill_value, dtype=None, order='K', subok=True, shape=None, device=None)`

返回一个与给定数组具有相同形状和类型的完整数组

参数:

①a: array_like

a的形状和数据类型定义了返回数组的这些相同属性

②fill_value: array_like

填充值

③dtype: 数据类型, 可选

覆盖结果的数据类型

| Add in 1.6.0 version

④order: {'C', 'F', 'A', 'K'}, 可选

覆盖结果的内存布局。C: 行优先, F: 列优先; A表示如果a是列连续的则为F, 否则为C; K表示尽可能匹配a的布局

| Add in 1.6.0 version

⑤subok: bool 可选

如果为True, 则新创建的数组将使用a的子类类型, 否则它将是一个基类数组。默认为True。

⑥shape: int或int序列, 可选

覆盖结果的形状, 如果order='K'并且维度数量不变, 将尝试保持顺序, 否则, 隐含order='C'

⑦device: str, 可选

要放置创建数组的设备。默认None, 仅用于数组AI互操作性, 因此如果传递, 必须为"cpu"。

| Add in 2.0.0 version

返回值:

➡ out: ndarray

与a具有相同形状和类型的fill_value数组

numpy.empty_like

numpy.empty_like(prototype, dtype=None, order='K', subok=True, shape=None, device=None)

返回一个与给定数组具有相同形状和类型的新数组

参数:

①prototype: array_like

prototype的形状和数据类型定义了返回数组的这些相同属性。

②dtype: 数据类型, 可选

覆盖结果的数据类型

| Add in 1.6.0 version

③order: {'C', 'F', 'A', 'K'}, 可选

覆盖结果的内存布局。C: 行优先, F: 列优先; A表示如果a是列连续的则为F, 否则为C; K表示尽可能匹配a的布局

| Add in 1.6.0 version

④subok: bool 可选

如果为True, 则新创建的数组将使用a的子类类型, 否则它将是一个基类数组。默认为True。

⑤shape: int或int序列, 可选

覆盖结果的形状, 如果order='K'并且维度数量不变, 将尝试保持顺序, 否则, 隐含order='C'

⑥device: str, 可选

要放置创建数组的设备。默认None, 仅用于数组AI互操作性, 因此如果传递, 必须为"cpu"。

| Add in 2.0.0 version

返回值:

➡ out: ndarray

一个未初始化(任意)数据的数组, 形状和类型与prototype相同

```
print("=====设置一个原始数组origin_arr=====")
origin_arr = np.array([[5, 6, 7],
                       [1, 2, 3]])

print(origin_arr)
zeros_arr = np.zeros_like(origin_arr)
ones_arr = np.ones_like(origin_arr)
full_value_arr = np.full_like(origin_arr, 2.1)           #即使我设置填充值为2.1, dtype也按原数组元素类型赋值
print("=====形状类似origin_arr的数组, 元素的数据类型与原数组的一样 =====")
print(zeros_arr)
print(ones_arr)
print(full_value_arr)

#运行结果如下
=====设置一个原始数组origin_arr=====
[[5 6 7]
 [1 2 3]]
=====形状类似origin_arr的数组, 并且元素的数据类型与原数组的一样 =====
[[0 0 0]
 [0 0 0]]
[[1 1 1]
 [1 1 1]]
[[2 2 2]
 [2 2 2]]
```

ndarray操作

numpy.astype

numpy.astype(x, dtype, copy=True, device=None)

ndarray.astype(dtype)

将数组复制到指定的数据类型

参数:

①x: ndarray

输入要转换的NumPy数组, `array_like` 在这里明确不支持。

②dtype: dtype

结果的数据类型

③copy: bool, 可选

如果为 `True`, 则必须返回新分配的数组。如果 `False`, 并且指定的dtype与输入数组的数据类型匹配, 则必须返回输入数组; 否则, 必须返回新分配的数组。默认为 `True`。

④device: str, 可选

默认为 `None`, 如果传递, 必须为 `"cpu"`。

返回值:

➡ out: ndarray

具有指定数据类型的数组

```
a = np.array([1, 2, 3])
a_float = np.astype(a, np.float32)      #等效于下一行
b_float = a.astype(np.float32)
print(a_float)
print(b_float)
```

#执行结果如下, 两个函数运行结果一样说明两个函数等效

```
[1.  2.  3.]
[1.  2.  3.]
```

ndarray.reshape

ndarray.reshape(shape, order='C', copy=None) 与下面的函数等效

numpy.reshape(a, shape, order='C', copy=None)

返回一个包含相同数据但具有新形状的数组

参数:

①a: array_like(ndarray.reshape不用传)

要重塑的数组

②shape: 整数或整数元组

新形状应与原始形状兼容，如果是整数，则结果将是该长度的一维数组，一个形状维度可以是 `-1`，在这种情况下，值是从数组的长度和剩余维度推断出来的。

③order: {'C','F','A'}可选

C: 行优先, F: 列优先, A: 如果 `a` 在内存中列连续，则使用列优先，否则使用行优先。

④copy: 布尔值, 可选

如果 `True`，则数组数据将被复制。如果 `None`，只要在 `order` 需要时才会进行复制。对于 `False`，如果无法避免复制，则会引发 `ValueError`。默认值是 `None`。

返回值:

➡ reshaped_array: ndarray

这将是新的视图对象

```
a = np.arange(10).reshape((2, -1))          #-1表示，可以从剩余维度中推断出来值 例如 10 / 2 = 5
a_1d = np.reshape(a, 10)
print("=====使用ndarray.reshape=====")
print(a)
print("=====使用numpy.reshape =====")
print(a_1d)
b = a_1d.reshape((5, -1), order='F')
print("=====指定参数order=F =====")
print(b)

#运行结果如下
=====使用ndarray.reshape=====
[[0 1 2 3 4]
 [5 6 7 8 9]]
=====使用numpy.reshape =====
[0 1 2 3 4 5 6 7 8 9]
=====指定参数order=F =====
[[0 5]
 [1 6]
 [2 7]
 [3 8]
 [4 9]]
```

numpy.transpose

`numpy.transpose(a, axes=None)`

等效于 `ndarray.transpose(*axes)`

返回一个转置轴的数组

参数:

① **a**: array_like

输入数组

② **axes**: 整数的元组或列表, 可选

如果指定, 它必须是一个包含[0, 1, ..., N-1]排列的元组或列表, 其中N是a的轴数。负索引也可以用于指定轴, 返回数组的第 *i* 个轴将对应于输入的第 `axes[i]` 个轴。如果未指定, 默认为 `range(a.ndim)[::-1]`, 这将反转轴的顺序。

返回值:

➡ **p**: ndarray

a的轴被置换, 只要可能, 就会返回一个视图。

numpy.swapaxes

numpy.swapaxes(a, axis1, axis2)

ndarray.swapaxes

交换数组的两个轴

参数:

① **a**: array_like

输入数组

② **axis1**: int

第一个轴

③ **axis2**: int

第二个轴

返回值:

➡ **a_swapped**: ndarray

返回a的一个视图

```
x = np.array([[0,1],[2,3]],[[4,5],[6,7]])
>>> x
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])

>>> np.swapaxes(x,0,2)
array([[0, 4],
       [2, 6],
       [1, 5],
       [3, 7]])
```

numpy.append

numpy.append(arr, values, axis=None)

将值追加到数组的末尾

参数:

①arr: array_like

值被追加到此数组的一个副本中

②values: array_like

如果指定了axis, 则values必须是正确的形状; 如果未指定, 在添加values之前会把数组展平

③axis: int, 可选

指定在哪个轴添加values, 如果未给出, 则会将arr展平。

返回值:

➡ append: ndarray

在axis上附加了values的arr副本, 注意 append 不是就地发生的, 而是分配并填充了一个新数组。如果 axis=None, 则输出一个展平的数组

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, ..., 7, 8, 9])

>>> np.append([1, 2, 3], [4, 5, 6], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

>>> np.append([1, 2, 3], [4, 5, 6], [[7], [8]], axis=1)
array([[1, 2, 3, 7],
       [4, 5, 6, 8]])
```

numpy.hstack

numpy.hstack(*tup*, dtype=None, casting='same_kind')

横向拼接，按*tup*的顺序水平堆叠数组（按列），这相当于在第二个轴上进行数组的拼接

参数：

①*tup*: `ndarrays`序列

数组必须在除第二个轴之外的所有轴上具有相同的形状，除了一维数组可以是任意长度

②*dtype*: `str`或*dtype*，可选

如果提供，目标数组将具有此数据类型

③*casting*: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'},可选

控制可能发生的数据类型转换，默认 `same_kind`

返回值：

➡ *stacked*: `ndarray`

通过水平堆叠形成的数组

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

>>> a = np.array([[1],[2],[3]])      #3x1
>>> b = np.array([[4],[5],[6]])      #3x1
>>> np.hstack((a,b))                  #变成3x2
array([[1, 4],
       [2, 5],
       [3, 6]])
```

numpy.vstack

numpy.vstack(*tup*, dtype=None, casting='same_kind')

纵向拼接，按*tup*的顺序垂直堆叠数组（按行），这相当于沿第一个轴进行连接

参数：

①*tup*: `ndarrays`序列

数组必须在除第一个轴之外的所有轴上具有相同的形状，一维数组必须具有相同的长度

②*dtype*: `str`或*dtype*，可选

如果提供，目标数组将具有此数据类型

③*casting*: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'},可选

控制可能发生的数据类型转换，默认 `same_kind`

返回值:

→ **stacked: ndarray**

通过垂直堆叠形成的数组，至少是二维的

```
>>> a = np.array([1, 2, 3])           #1x3如果是一维数组必须是相同的长度
>>> b = np.array([4, 5, 6])           #1x3
>>> np.vstack((a,b))                  #变成2x3
array([[1, 2, 3],
       [4, 5, 6]])

>>> a = np.array([[1], [2], [3]])     #3x1
>>> b = np.array([[4], [5], [6]])     #3x1
>>> np.vstack((a,b))                  #变成6x1
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

numpy.column_stack

numpy.column_stack(*tup*)

将一系列的一维数组作为列向量堆叠成二维数组

参数:

tup: 一维或二维数组的序列

如果是一维数组的序列，它们必须具有相同的维度。如果是二维数组的序列，等同于 `np.vstack`

返回值:

→ **stacked: 二维数组**

通过横向拼接形成的数组

```
a = np.array((1,2,3))
b = np.array((2,3,4))
>>> np.column_stack((a,b))            #这里column_stack是把a和b转成列向量，之后横向拼接a和b产生的结果
array([[1, 2],
       [2, 3],
       [3, 4]])
```

numpy.tile

numpy.tile(A, reps)

通过重复 A 由 reps 给定的次数来构造一个数组，也就是整个结构重复

如果reps的长度为 d，则结果的维度将是 $\max(d, A.ndim)$

如果 $A.ndim < d$, A 会被提升为 d 维，通过在前面添加新的轴，因此，一个形状为 (3,) 的数组会被提升为 (1, 3)，用于 2 维复制，或者形状为 (1, 1, 3)，用于三维复制。

如果 $A.ndim > d$, reps 会被提升到 A.ndim，通过在其前面添加 1 来实现。因此对于形状为 (2, 3, 4, 5) 的 A，一个 reps 为 (2, 2) 的值会被视为 (1, 1, 2, 2)

参数：

①A: array_like

输入数组

②reps: array_like

沿每个轴重复A的次数

返回值：

➡ c: ndarray

平铺整个数组

```
a = np.array([0, 1, 2])
>>> np.tile(a, 2)           #重复a两次
array([0, 1, 2, 0, 1, 2])

#reps=(2, 2)的长度为2 > a.ndim, 所以数组会在前面添加新的轴变成与元组长度相同的二维数组(1, 2)，再重复相应的次数
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])

#reps=(2, 1, 2)的长度为3 > a.ndim,所以数组会在前面添加新的轴变成与元组长度相同的三维数组(1, 1, 3)，再重复相应的次数
>>> np.tile(a, (2, 1, 2))
array([[[0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2]])

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

#reps=2 的长度为1 < b.ndim, 所以reps会通过前面加1来提升到b的维度，即reps会变成(1, 2),即在列方向上重复两次
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])

#reps=(2, 1)的长度 = b.ndim,对应每个轴的重复次数，即第0轴重复两次，第1轴重复一次
np.tile(b, (2, 1))
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

```
# 问题：生成一个8x8数组，其中：第 1 行全为1，第 2 行全为2，第 3 行全为3，第 4 行全为4...第8行全为8
arr = np.arange(1, 9).reshape((8, 1))
>>> np.tile(arr, [1, 8])
[[1 1 1 1 1 1 1 1]
 [2 2 2 2 2 2 2 2]
 [3 3 3 3 3 3 3 3]
 [4 4 4 4 4 4 4 4]
 [5 5 5 5 5 5 5 5]
 [6 6 6 6 6 6 6 6]
 [7 7 7 7 7 7 7 7]
 [8 8 8 8 8 8 8 8]]
```

numpy.repeat

numpy.repeat(a, repeats, axis=None)

ndarray.repeat(repeats, axis=None)

重复数组中的每个元素在其自身之后，每个元素重复

参数：

①a: array_like

输入数组

②repeats: 整数或整数数组

每个元素的重复次数，repeats 会被广播以适应给定轴的形状。

③axis: int, 可选

要沿其重复值的轴，默认情况下，使用展平的输入数组，并返回一个展平的输出数组

返回值：

➡ repeated_array: ndarray

输出一个与a形状相同的数组，除了沿给定轴之外

```
arr = np.array([[1, 2],[3, 4]])
b = arr.repeat(3, axis=1)           #每个元素先沿着第二个轴上重复三次 2x6
>>> print(b.repeat(2, axis=0))      #每个元素再沿着第一个轴重复两次    4x6

[[1 1 1 2 2 2]
 [1 1 1 2 2 2]
 [3 3 3 4 4 4]
 [3 3 3 4 4 4]]
```

ndarray.flatten

ndarray.flatten(*order='C'*)

返回一个副本，扁平化为一维

参数：

order: {'C','F','A','K'}

C：行优先展平；F：列优先展平；A：如果a在内存中列连续，则列优先展平，否则按行优先顺序展平；K：表示按元素在内存中出现的顺序展平a；

返回值：

➡ **y:** ndarray

输入数组的一个副本，扁平化为一维

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()           #默认行优先展平
array([1, 2, 3, 4])
>>> a.flatten('F')       #指定为列优先展平
array([1, 3, 2, 4])
```

numpy.ravel

numpy.ravel(*a, order='C'*)

ndarray.ravel(*forder*)

返回一个连续的扁平化数组

返回一个包含输入元素的一维数组，仅在必要时进行复制，返回数组与输入数组类型相同

参数：

① **a:** array_like

输入数组。a 中的元素按照order指定的顺序读取，并打包成一个一维数组

② **order:** {'C','F','A','K'}, 可选

读取a的顺序。C：行优先；F：列优先；C和F不考虑原数组的内存布局。A：如果a在内存中列连续，则按列优先读取，否则行优先顺序读取；K：按元素在内存中出现的顺序读取，除非在步幅为负时反转数据。

返回值：

➡ **y:** array_like

扁平化的一维数组

`np.ravel(x)` 等价于 `x.reshape(-1)`

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

>>> np.ravel(x)           #展平，默认行优先
array([1, 2, 3, 4, 5, 6])

>>> x.reshape(-1)         #等价于np.ravel(x)
array([1, 2, 3, 4, 5, 6])

>>> np.ravel(x, order='F') #列优先展平
array([1, 4, 2, 5, 3, 6])
```

当 `order='A'`，它将保留数组的C或F顺序

```
x = np.array([[1, 2, 3], [4, 5, 6]], order='F')
>>> x.ravel(order='A')
array([1, 4, 2, 5, 3, 6])
```

当 `order='K'` 时，它将保留既不是C也不是F的顺序，但是不会反转轴

```
x = np.arange(12).reshape(2, 3, 2)
print("====原数组====")
print(x)
y = x.swapaxes(1, 2)
print("====交换轴之后====")
print(y)
print("====可以看到交换轴之后，虽然数组变了，但其内存地址是不变的，所以展平之后还是底层数组的内存顺序====")
y.ravel(order='K')

#输出结果如下
====原数组====
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]

====交换轴之后====
[[[ 0  2  4]
  [ 1  3  5]]

 [[ 6  8 10]
  [ 7  9 11]]]

====可以看到交换轴之后，虽然数组变了，但其内存地址是不变的，所以展平之后还是底层数组的内存顺序====
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

numpy.reshape

`numpy.reshape(a, shape=None, order='C', copy=None)`

`ndarray.reshape`

改变数组的形状

参数:

①a: array_like

要重塑的数组

②shape: int或int的元组

新形状应与原始形状兼容, 如果一个形状是-1, 在这种情况下, 其值可以从数组的长度和剩余的维度推断出来

③order: {'C','F','A'}

使用此索引顺序读取 a 的元素, 并使用此索引顺序将元素放入重塑后的数组。

C: 行优先; F: 列优先; A: 如果 a 在内存中列优先顺序存储, 则列优先读取/写入元素, 否则行优先

④copy: bool, 可选

如果 True, 数组数据将被复制, 如果 None, 只有在order需要时才会进行复制。

返回值:

➡ reshaped_array: ndarray

返回一个新的视图对象

```
a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')           #列优先读取之后, 列优先放入(6,)中
array([1, 4, 2, 5, 3, 6])
>>> np.reshape(a, (3,-1))                 # 推断维度
array([[1, 2],
       [3, 4],
       [5, 6]])
```

numpy.resize

numpy.resize(a, new_shape)

ndarray.resize(new_shape)

返回一个具有指定形状的新数组

如果新数组比原数组大, 那么新数组会用a的重复副本填充, **注意, 这种行为与a.resize(new_shape)不同, 后者用零填充而不是a的重复副本**

参数:

①a: array_like

要调整大小的数组

②new_shape: int或int的元组

调整大小后的数组形状

返回值:

→ reshaped_array: ndarray

如果新形状的大小比原数组大，会以行优先的顺序填充重复数据/或者填充0

```
a = np.array([[0,1],[2,3]])
print(np.resize(a,(1,4)))
print("=====新数组的大小 > 原数组的大小=====")
print("=====NumPy层级下调用，用原数组的重复副本=====")
print(np.resize(a,(4,3)))
print("=====ndarray层级下调用，用0填充=====")
a.resize((4, 3))
print(a) #可以看到NumPy调用和ndarray调用结果是不一样的
```

#输出结果如下

```
[[0 1 2 3]] #正常调整大小
=====新数组的大小 > 原数组的大小=====
=====NumPy层级下调用，用原数组的重复副本=====
[[0 1 2]
 [3 0 1]
 [2 3 0]
 [1 2 3]]
=====ndarray层级下调用，用0填充=====
[[0 1 2]
 [3 0 0]
 [0 0 0]
 [0 0 0]]
```

numpy.meshgrid

numpy.meshgrid(*xi)

从坐标向量返回坐标矩阵的元组

参数:

x1, x2, ..., xn: array_like

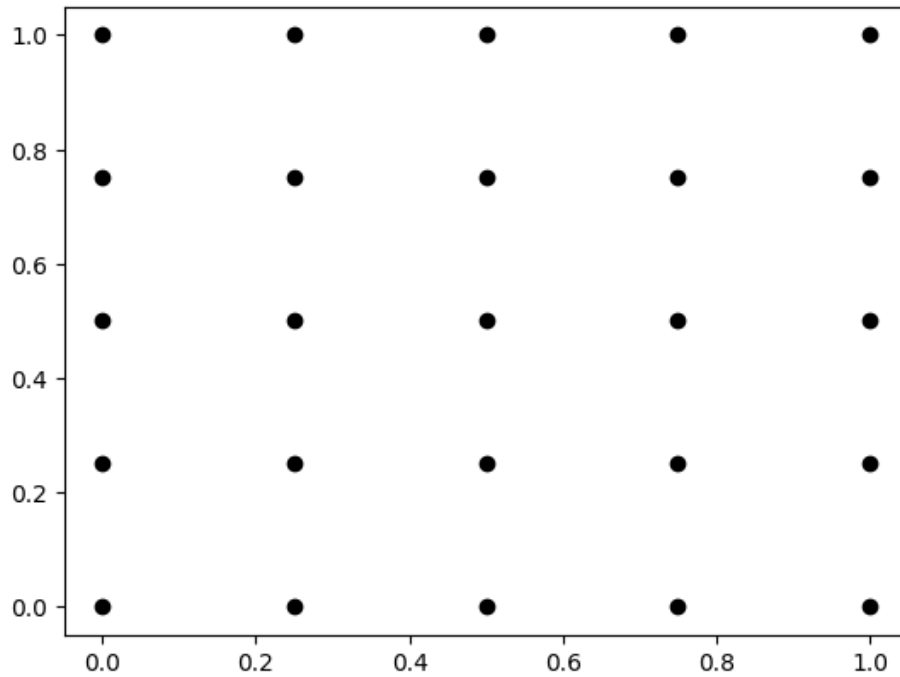
表示网格坐标的一维数组

```
46. Create a structured array with `x` and `y` coordinates covering the [0,1]x[0,1] area (★★☆)
Z = np.zeros((5,5), [('x',float),('y',float)])
Z['x'], Z['y'] = np.meshgrid(np.linspace(0,1,5),
                             np.linspace(0,1,5))

>>> print(Z)
[[ (0. , 0. ) (0.25, 0. ) (0.5 , 0. ) (0.75, 0. ) (1. , 0. )]
 [ (0. , 0.25) (0.25, 0.25) (0.5 , 0.25) (0.75, 0.25) (1. , 0.25)]
 [ (0. , 0.5 ) (0.25, 0.5 ) (0.5 , 0.5 ) (0.75, 0.5 ) (1. , 0.5 )]
 [ (0. , 0.75) (0.25, 0.75) (0.5 , 0.75) (0.75, 0.75) (1. , 0.75)]
 [ (0. , 1. ) (0.25, 1. ) (0.5 , 1. ) (0.75, 1. ) (1. , 1. )]]
```

`meshgrid` 的结果是一个坐标网格

```
plt.plot(Z['x'], Z['y'], marker='o', color='k', linestyle='none')
plt.show()
```



ndarray属性

ndarray.T

转置数组的视图，与 `self.transpose()` 相同

ndarray.shape

ndarray.shape

返回数组的形状元组

ndarray.ndim

ndarray.ndim

返回数组的维度

例子：

```
a33 = np.random.randint(10, size=(3, 3))
print(f"矩阵的形状为: {a33.shape}, 表示{a33.shape[0]}行, {a33.shape[1]}列")
print(f"矩阵的维数是: {a33.ndim}")
```

#运行结果如下

矩阵的形状为: (3, 3), 表示3行, 3列

矩阵的维数是: 2

ndarray.flags

ndarray.flags

关于数组内存布局的信息。

①C_CONTIGUOUS(C): 仅有行优先时为True

②F_CONTIGUOUS(F): 仅有列优先时为True

③writeable: 如果为False, 则ndarray变为只读

```
A = np.random.randint(0,10,5)
A.flags.writeable = False
>>> A[0] = 1
```

```
-----
ValueError                                Traceback (most recent call last)
```

```
Cell In[2], line 3
```

```
1 A = np.random.randint(0,10,5)
```

```
2 A.flags.writeable = False
```

```
----> 3 A[0] = 1
```

```
ValueError: assignment destination is read-only
```

ndarray.strides

在遍历数组时, 每个维度中跨越的字节数

数组a中元素(i[0], i[1], ..., i[n])的字节偏移量为: `offset = sum(np.array(i) * a.strides)`

ndarray.size

数组中的元素数量, 等于 `np.prod(a.shape)`, 即数组各维度的乘积。注意, `a.size` 返回一个标准的任意精度的Python整数, 这可能导致两次调用产生的值不同。

ndarray.itemsize

一个数组元素的字节长度

#如何得到数组的【内存大小】, 内存大小: 也就是数组占多少个字节

```
Z = np.zeros((10,10))
```

```
print(f"{(Z.size * Z.itemsize)} bytes")
```

ndarray.nbytes

数组元素消耗的总字节数

ndarray.base

如果内存来自其他对象，则为基本对象

示例：

拥有其内存的数组的基础是None。切片创建一个视图，其内存与原数组共享

```
a = np.array([1, 2, 3, 4, 5])
>>> print(a.base is None)
True

b = a[:3]
>>> print(b.base is a)
True
```

ndarray.dtype

数组元素的数据类型

#传统生成器（RandomState）

random.randint

random.randint(low, high=None, size=None, dtype=int)

返回 [low,high) 半开区间内的随机整数。

返回指定dtype的离散均匀分布中的随机整数，在半开区间 [low, high) 内，如果high为None，则结果来自 [0, low)

参数：

①low: int或类似数组的ints

要从分布中抽取的最低（有符号）整数（除非 high=None，在这种情况下，取不到low）

②high: int或int的可迭代对象，可选

如果提供，则要高于从分布中抽取的最大有符号整数，如果是类数组对象，则必须包含整数值

③size: 整数或整数的元组，可选

输出形状，如果给定的形状是，例如 (m, n, k)，那么会抽取 $m * n * k$ 个样本。默认是None，在这种情况下会返回一个单一值。

④dtype: dtype可选

期望的结果数据类型，字节顺序必须是本地的，默认是长整型

返回值：

➡ out: int或ints的ndarray

返回size形状的随机整数数组，来自适当的分布，或者如果没有提供size，则为单个这样的随机整数

```
print(np.random.randint(5, size=10))      #生成一个包含0到5的整数的数组，长度为10
print(np.random.randint(5, size=(2, 3)))  #生成一个0-4的整数数组，形状为 (2,3)
print(np.random.randint(1, [3, 5, 10]))   #有三个不同的上限
print(np.random.randint([1, 5, 7], 10))   #有三个不同的下限
```

#运行结果如下

```
[1 3 0 0 4 3 3 3 4 1]
[[4 3 3]
 [0 2 4]]
[2 2 9]
[6 7 8]
```

random.randn

random.randn(d0, d1, ..., dn)

返回来自标准正态分布（均值为0，方差为1）的样本（或多个样本）

参数：

d0, d1, ..., dn: int 可选

返回数组的维度必须是非负的，如果没有给出参数，则返回一个单独的Python浮点数

返回值：

➡ Z: ndarray或float

一个 (d0, d1, ..., dn) 形状的从标准正态分布中抽取的浮点样本数组，或者如果没有提供参数，则为单个这样的浮点数

! 对于从均值为 mu 和标准差为 sigma 的正态分布中随机抽样。请使用：

```
sigma * np.random.randn(...) + mu
```

```
print(np.random.randn())           #生成一个浮点数
print(np.random.randn(2, 3))       #生成一个形状为（2,3）的数组，元素服从标准正态
print(3 + 5*np.random.randn(4, 2)) #生成一个形状为（4,2）的数组，元素服从均值为3，方差为5的正态
分布

#运行结果如下
1.5828847571765243
[[-0.08526865  0.12551746 -0.78860294]
 [ 0.24404405  1.94560315  0.9965965 ]]
[[-3.03597307  4.98792739]
 [ 9.34283388  7.9309673 ]
 [-3.08805505 10.43988641]
 [10.48311086 -5.91902459]]
```

random.rand

random.rand(d0, d1, ..., dn)

给定形状中的随机值。创建一个给定形状的数组，并用从 [0, 1) 上的均匀分布中随机抽取的样本填充它。

参数：

d0, d1, ..., dn: int 可选

返回数组的维度必须是非负的，如果没有给出参数，则返回一个单独的Python浮点数

返回值：

out: ndarray, 形状

随机值

```
>>> print(np.random.rand(2, 3))           #生成一个形状为（2,3）的数组，元素服从[0,1)上的均匀分布

[[0.83609836 0.7780242  0.56917799]
 [0.60061554 0.45927977 0.23115657]]
```

random.uniform

random.uniform(low=0.0, high=1.0, size=None)

从 [low, high) 的均匀分布中抽取样本，size是int或int的元组

均匀分布的概率密度函数

$$f(x) = \begin{cases} \frac{1}{high-low} & low \leq x < high \\ 0 & else \end{cases}$$

random.seed

random.seed(seed=None)

重新播种单例 RandomState 示例。

我们看似没有种子的时候默认生成的随机数，一般都是以系统时钟等为种子计算出来的。

相同的种子可以生成相同的随机数，我们之所以指定种子，多半是要进行测试操作，也就是说我们生成随机序列，需要反复生成相同的随机序列来观察结果。

```
np.random.seed(123)
print(np.random.rand())
np.random.seed(123)
print(np.random.rand())
```

```
#运行结果如下，可以看到设置相同的种子，两次产生的随机数是相同的
0.6964691855978616
0.6964691855978616
```

random.choice

random.choice(a, size=None, replace=True, p=None)

从一个给定的一维数组中生成一个随机样本。

参数：

①a：一维类数组或整数

如果是一个ndarray，则从其元素中生成一个随机样本，则生成的随机样本就好像它是 `np.arange(a)`

②size：整数或整数的元组，可选

输出形状，如果给定的形状是，例如 `(m, n, k)`，那么会抽取 `m * n * k` 个样本，默认是None，在这种情况下会返回一个单一值。

③replace：布尔值，可选

样本是有放回还是无放回，默认为True,表示可以多次选择值 a

④：一维类数组，可选

与 a 中每个条目相关的概率，如果未给出，样本假设 a 中所有条目的均匀分布

返回值：

➡ samples：单个项目或ndarray

生成的随机样本

引发：

ValueError

如果a是一个小于零的整数，如果a或p不是一维的，如果a是一个大小为0的类数组对象，如果p不是一个概率向量，如果a和p的长度不同，或者如果replace=False且样本大小大于总体大小。


```

print("====从np.range(5)生成大小为6的均匀样本====")
print(np.random.choice(5, 6))
print(np.random.choice([0, 1, 2, 3, 4], 6))
print("====从np.range(5)生成大小为3的无放回的均匀样本====")
print(np.random.choice(5, 3, replace=False))
print("====从np.range(5)生成大小为3的非均匀样本, 指定概率p====")
print(np.random.choice(5, 3, p=[0.2, 0.1, 0.2, 0.3, 0.2]))
print("====指定概率p数组的大小必须和a的大小一样, 且p中的元素总和为1, 否则会触发ValueError====")
languages = ['C++', 'C#', 'Java', 'Python', 'SQL']
print(np.random.choice(languages, 10, p=[0.3, 0.2, 0.1, 0.3, 0.1]))

```

#执行结果如下

```

====从np.range(5)生成大小为6的均匀样本====
[3 1 3 4 4 1]
[4 4 3 4 3 0]
====从np.range(5)生成大小为3的无放回的均匀样本====
[1 2 3]
====从np.range(5)生成大小为3的非均匀样本, 指定概率p====
[3 0 2]
====指定概率p数组的大小必须和a的大小一样, 且p中的元素总和为1, 否则会触发ValueError====
['C++' 'C#' 'Java' 'Python' 'C++' 'C++' 'C#' 'Python' 'Python' 'Java']

```

```

>>> np.random.choice(languages, 10, p=[0.7, 0.3])
"ValueError: 'a' and 'p' must have same size"      #p的大小和a不同
>>> np.random.choice(languages, 10, p=[0.7, 0.1, 0.1, 0.1, 0.1])
"ValueError: probabilities do not sum to 1"        #p中的元素和必须为1

```

random.shuffle

random.shuffle(x)

通过打乱其内容来就地修改序列。此函数仅沿多维数组的第一个轴打乱数组，子数组的顺序被改变，但它们的内容保持不变。

返回值为None

参数:

x: ndarray或MutableSequence

要被洗牌的数组、列表或可变序列。

返回值:

→ None

```

a = np.arange(10)
print(f"原数组: {a}")
np.random.shuffle(a)
print(f"混洗后: {a}")

#输出结果如下
原数组: [0 1 2 3 4 5 6 7 8 9]
混洗后: [0 7 1 4 9 6 2 5 3 8]

print(np.random.shuffle(np.arange(10))) --> 运行结果为: None, 因为返回值为None

```

random.random

random.random(size=None)

返回在半开区间 $[0.0, 1.0)$ 内的随机浮点数。

#随机生成器

random.default_rng(生成器的构造函数)

random.default_rng(seed=None)

使用默认的BitGenerator(PCG64)构造一个新的生成器。

seed: {None, int, array_like[ints], SeedSequence, BitGenerator, Generator}, 可选

➡ Generator: 返回初始化的生成器对象

default_rng 是随机数类 Generator 推荐的构造函数。

以下是使用 default_rng 和 Generator 类构造随机数生成器的几种方法:

```

rng = np.random.default_rng()
print(rng)
rfloat = rng.random()           #使用生成器生成一个[0.0, 1.0)的浮点数
print(rfloat, type(rfloat))
fints_arr = rng.integers(0, 10, size=(5, 3))   #使用生成器生成一个元素在[0, 10)的随机整数数组, 形状为(5, 3)
print(fints_arr)

#运行结果如下
Generator(PCG64)
0.5198093898963828 <class 'float'>
[[3 1 3]
 [9 1 6]
 [3 7 6]
 [4 0 6]
 [5 8 9]]

```

Generator.standard_normal

Generator.standard_normal(size=None, dtype=np.float64, out=None)

从标准正态分布中抽取样本

参数:

①size: 整数或整数的元组, 可选

输出形状, 如果给定的形状是, 例如 `(m, n, k)`, 那么会抽取 `m * n * k` 个样本。默认为 `None`, 这种情况下会返回一个值

②dtype: dtype, 可选

期望的结果数据类型, 仅支持float64和float32字节, 默认值是np.float64

③out: ndarray, 可选

要在其中放置结果的替代输出数组。如果size不为None,它必须与提供的size具有相同的形状, 并且必须与输出值的类型匹配

返回值:

→ out: 浮点数或ndarray

一个形状为 size 的浮点数组, 如果未指定 size, 则为单个样本

Generator.normal

Generator.normal(loc=0.0, scale=1.0, size=None)

从正态 (高斯) 分布中随机抽取样本

参数:

①loc: 浮点数或浮点数的类数组对象

分布的均值 ("中心")

②scale: 浮点数或浮点数的类数组对象

标准差, 非负数

③size: 整数或整数的元组, 可选

输出形状, 如果给定的形状是, 例如 `(m, n, k)`, 那么会抽取 `m * n * k` 个样本。如果size为None, 当 loc 和 scale 都是标量时, 返回一个单一值。否则会抽取 `np.broadcast(loc, scale).size` 个样本

返回值:

→ out: ndarray或标量

从参数化的正态分布中抽取样本

```

print("====生成服从标准正态的20个样本的数组====")
print(rng.standard_normal(20))
print("====生成服从标准正态的二维数组，形状为（5，4）====")
print(rng.standard_normal((5, 4)))
print("====生成服从均值为3，方差为2的10个样本的数组====")
print((3 + 2*rng.standard_normal(10)))
print("====生成服从均值为1，标准差为0.1(方差为0.01)的10个样本的数组====")
a = rng.normal(1, 0.1, 10)
print(a)
print("====因为样本数为10，所以基于此样本计算出来的均值和方差会偏离设定值====")
print(f"上述样本的均值为: {a.mean()},方差为: {a.var()}")

```

#运行结果如下

```

====生成服从标准正态的20个样本的数组====
[ 1.25623578  0.95826065  0.33941347  1.28201476 -0.70215456  1.42018741
  0.9206137   0.57545776  0.48576265 -0.42042053  0.02094672 -1.38091806
 -0.03635924  0.36284079 -1.1728381  -0.99189395 -0.31979236  2.72144821
 1.8450494  -0.3676662 ]
====生成服从标准正态的二维数组，形状为（5，4）====
[[-0.24619268  0.97550327  0.4313814  0.0758458 ]
 [ 1.64830561  0.6614323  -1.51891793 -0.11233501]
 [ 0.31150257 -3.99031636  0.76583078 -2.65038156]
 [ 1.45536708  0.07493618  0.7562625  -1.28802602]
 [ 0.33810112 -1.35608493 -0.79666706  0.88358043]]
====生成服从均值为3，方差为2的10个样本的数组====
[ 3.28374416  0.91203725  3.93013278  3.17452366  5.16975494  2.8208682
 -0.03253342  6.18694624  3.65331257  5.28209665]
====生成服从均值为1，标准差为0.1(方差为0.01)的10个样本的数组====
[0.87773764 1.0619596  0.96989239 1.09867176 1.04804133 0.91816361
 0.9745139  0.90243528 1.0976393  1.12862031]
====因为样本数为10，所以基于此样本计算出来的均值和方差会偏离设定值====
上述样本的均值为: 1.0077675126196848,方差为: 0.007407651619766019

```

排序、搜索和计数

numpy.nonzero

numpy.nonzero(a)

ndarray.nonzero()

返回非零元素的索引

返回一个数组元组，每个维度一个数组，包含该维度中非零元素的索引。a 中的值总是行优先的顺序进行测试和返回

示例：

```

import numpy as np
x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
>>> x
array([[3, 0, 0],
       [0, 4, 0],
       [5, 6, 0]])
>>> np.nonzero(x)
#这里就返回一个元组，元组的第一个元素是行索引，第二个元素是列索引
#即非零元素下标为x[0,0] x[1,1] x[2,0] x[2,1]
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))

```

```
>>> x[np.nonzero(x)]
array([3, 4, 5, 6])
>>> np.transpose(np.nonzero(x))  #转置一下就是非零元素对应的下标
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]])
```

numpy.where

numpy.where(condition, [x, y, I])

根据 *condition* 从 *x* 或 *y* 中返回元素。

参数:

① **condition**: array_like, bool

如果为真, 则生成 *x*, 否则生成 *y*

② **x, y**: array_like

可供选择的值, *x*, *y* 和 *condition* 需要能够广播到某种形状

返回值:

➡ **out**: ndarray

返回一个数组, 如果 *condition* 为 True, 则数组包含来自 *x* 的元素, 否则元素来自 *y*

```
a = np.arange(10)
>>> print(a)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(np.where(a < 5, a, 10*a))
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

#也可以用于多维数组

```
a = np.where([
    [True, False],
    [True, True]
],
             [[1, 2],
              [3, 4]],
             [[9, 8],
              [7, 6]])
>>> print(a)
[[1 8]
 [3 4]]
```

numpy.argwhere

numpy.argwhere(*a*)

查找数组中非零元素的索引，按元素分组

参数：

a: array_like

输入数据，也可以是`condition`

返回值：

➡ **index_array:** (N, a.ndim) ndarray

非零元素的索引，索引按元素分组，该数组的形状将是 (N, a.ndim)，其中 N 是非零项的数量

```
>>> x = np.arange(6).reshape(2,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])

>>> print(np.argwhere(x))      #这里输出的是非0元素的下标，形状为(5,2)
[[0 1]
 [0 2]
 [1 0]
 [1 1]
 [1 2]]

>>> np.argwhere(x>1)          #这里输出的是x中大于1的元素下标，形状为(4,2)
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

numpy.argmax/argmin

numpy.argmax(*a*, axis=None, out=None, keepdims=<no value>)

返回沿某个轴的最大值/最小值的索引

在最大值/最小值多次出现的情况下，返回第一次出现最值的索引

参数：

① **a:** array_like

输出数组

② **axis:** int, 可选

默认情况下，索引是针对展平的数组，否则沿指定的轴进行。

③ **out:** 数组, 可选

如果提供，结果将被插入到这个数组中，它应该具有适当的形状和数据类型

④ **keepdims:** bool, 可选

如果设置为True，被减少的轴将作为尺寸为1的维度保留在结果中，使用此选项，结果将正确地与数组广播

返回值:

→ index_array: ints的ndarray

数组的索引数组。它与 `a.shape` 具有相同的形状，但沿axis的维度被移除，如果`keepdims`设置True，则axis的大小将为1，结果数组将具有与 `a.shape` 相同的形状。

numpy.sort/argsort

numpy.sort(argsort(a, axis=-1, kind=None, order=None, stable=None))

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

返回一个数组的排序副本/索引

参数:

①a: array_like

要排序的数组

②axis: int或None

要排序的轴，如果为None，则在排序之前将数组展平。默认值为-1，即沿最后一个轴排序

③kind: {'quicksort', 'mergesort', 'heapsort', 'stable'}, 可选

排序算法，默认是 quicksort，但是快排和堆排序是不稳定的算法，归并排序是稳定的算法。

所谓排序算法的稳定性，指的是：如果两个元素值相同，则排序后它们的相对位置是否保持不变，保持不变即稳定，改变即不稳定。

kind	速度	平均情况	最坏情况	空间复杂度	stable
quicksort	1	$O(n\log n)$	$O(n^2)$	$O(\log n)$ (递归栈)	✗
heapsort	3	$O(n\log n)$	$O(n\log n)$	$O(1)$	✗
mergesort	2	$O(n\log n)$	$O(n\log n)$	$O(n)$	✓
timsort (归并排序+插入排序+优化)	2	$O(n\log n)$	$O(n\log n)$	$O(n)$	✓

④order: str或str列表, 可选

当a是一个定义了字段的数组时，此参数指定首先比较哪些字段，其次比较哪些字段。

⑤stable: bool, 可选

排序稳定性，如果为 True，返回的数组将保持 a 数组中值的顺序。如果为 False 或 None，则不保证顺序

返回值:

➡ sorted_array: ndarray

与a的类型 and 形状相同的数组/索引数组

```
a = np.array([[1, 3, 2],[5, 4, 1]])
print("=====默认排序算法为快排=====")
print("=====沿最后一个轴排序=====")
print(np.sort(a))
print("=====元素对应的索引=====")
print(np.argsort(a))
print("=====沿第一个轴排序=====")
print(np.sort(a, axis=0))
print("=====元素对应的索引=====")
print(np.argsort(a, axis=0))
print("=====如果axis=None, 摊开排序=====")
print(np.sort(a, axis=None))
print("=====元素对应的索引=====")
print(np.argsort(a, axis=None))
```

#输出结果如下

```
=====默认排序算法为快排=====
=====沿最后一个轴排序=====
[[1 2 3]
 [1 4 5]]
=====元素对应的索引=====
[[0 2 1]
 [2 1 0]]
=====沿第一个轴排序=====
[[1 3 1]
 [5 4 2]]
=====元素对应的索引=====
[[0 0 1]
 [1 1 0]]
=====如果axis=None, 摊开排序=====
[1 1 2 3 4 5]
=====元素对应的索引=====
[0 5 2 1 4 3]
```

使用 **order** 关键字来指定排序结构化数组时要使用的字段

```
my_dtype = [('name', 'S10'), ('height', float), ('age', int)] #结构化数组, 使用(字段名, 类型)的元组进行定义
values = [('Arthur', 1.8, 41),
          ('Lancelot', 1.9, 38),
          ('Galahad', 1.7, 38)]
a = np.array(values, dtype=my_dtype)
print(np.sort(a, order='height')) #按身高进行排序
print(np.sort(a, order=['age', 'height'])) #如果年龄相同, 则按身高进行排序

#输出结果如下
[(b'Galahad', 1.7, 38) (b'Arthur', 1.8, 41) (b'Lancelot', 1.9, 38)]
[(b'Galahad', 1.7, 38) (b'Lancelot', 1.9, 38) (b'Arthur', 1.8, 41)]
```

数学函数

numpy.max/min

`numpy.max/min(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)`

`ndarray.max/min()`

返回数组中的最大值/最小值或沿轴的最大值/最小值

参数:

①a: array_like

输入数据

②axis: None或int或int的元组, 可选

默认情况下, 将输入的数据展平之后, 再计算最大值/最小值, 即使它是多维数组

如果这是一个整数的元组, 则最大值是在多个轴上选择的。

③out: ndarray, 可选

要在其中放置结果的替代输出数组, 必须与预期输出的形状和缓冲区长度相同。

④keepdims: bool, 可选

如果设置为True, 被减少的轴将作为尺寸为1的维度保留在结果中。使用此选项, 结果将正确地与输入数组广播。

⑤initial: 标量, 可选

输出元素的最小值, 必须存在以允许在空切片上进行计算。

⑥where: 类数组的布尔值, 可选

用于比较的最大/最小元素

返回值:

➡ max: ndarray或标量

最大值/最小值a, 如果axis为None, 返回是一个标量值。如果axis是int, 结果是一个维度为 `a.ndim - 1` 的数组。如果axis是元组, 结果是一个维度为 `a.ndim - len(axis)` 的数组

```
a = np.arange(54).reshape(6,3,3)
print("=====原数组=====")
print(a)
print("=====求数组中的最大值, 保持原来的维度=====")
print(a.max(keepdims=True))
print("=====沿着轴1去求最大值=====")
print(a.max(axis=1))          #其结果为6x3的二维数组
print("=====沿着轴2去求最大值=====")
print(a.max(axis=2))          #其结果为6x3的二维数组
print("=====沿着轴1和轴2去求最大值=====")
print(a.max(axis=(1, 2)))      #其结果为3-len((1, 2))的一维数组
print("=====求数组中的最小值, 保持原来的维度=====")
print(a.min(keepdims=True))
print("=====沿着轴1去求最小值=====")
print(a.min(axis=1))          #其结果为6x3的二维数组
print("=====沿着轴2去求最小值=====")
print(a.min(axis=2))
```

#输出结果如下

=====原数组=====

```
[[[ 0  1  2]
   [ 3  4  5]
   [ 6  7  8]]
```

```
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

```
[[18 19 20]
 [21 22 23]
 [24 25 26]]
```

```
[[27 28 29]
 [30 31 32]
 [33 34 35]]
```

```
[[36 37 38]
 [39 40 41]
 [42 43 44]]
```

```
[[45 46 47]
 [48 49 50]
 [51 52 53]]]
```

=====求数组中的最大值，保持原来的维度=====

```
[[[53]]]
```

=====沿着轴1去求最大值=====

```
[[ 6  7  8]
 [15 16 17]
 [24 25 26]
 [33 34 35]
 [42 43 44]
 [51 52 53]]
```

=====沿着轴2去求最大值=====

```
[[ 2  5  8]
 [11 14 17]
 [20 23 26]
 [29 32 35]
 [38 41 44]
 [47 50 53]]
```

=====沿着轴1和轴2去求最大值=====

```
[ 8 17 26 35 44 53]
```

=====求数组中的最小值，保持原来的维度=====

```
[[[0]]]
```

=====沿着轴1去求最小值=====

```
[[ 0  1  2]
 [ 9 10 11]
 [18 19 20]
 [27 28 29]
 [36 37 38]
 [45 46 47]]
```

=====沿着轴2去求最小值=====

```
[[ 0  3  6]
 [ 9 12 15]
 [18 21 24]
 [27 30 33]
 [36 39 42]
 [45 48 51]]
```

numpy.modf

numpy.modf(x)

逐元素返回数组的分数部分和整数部分。如果给定的数字是负数，则分数部分和整数部分都是负的。

x: array_like, 输入数组

→ y1: ndarray

x的小数部分，如果x是标量，则这也是一个标量

→ y2: ndarray

x的整数部分，如果x是标量，则这也是一个标量

```
>>> np.modf([0, 3.5])
(array([ 0. ,  0.5]), array([ 0.,  3.]))
>>> np.modf(-0.5)
(-0.5, -0)
```

numpy.maximum/minimum

numpy.maximum(x1, x2)

返回一个新数组，数组的元素是逐个比较x1和x2中的元素得到的最大值/最小值，如果被比较元素之一是nan，则返回nan。如果两个元素都是nan，则返回第一个。后者的区分对复数nan很重要，它们定义为实部或虚部中至少有一个是nan，效果是nan被传播

参数：

x1,x2: array_like

存储要比较的元素的数组。如果 x1.shape != x2.shape，它们必须能够广播到一个共同的形状，这将成为输出形状。

返回值：

→ y: ndarray或标量

逐元素计算 x1和x2的最大值/最小值，如果x1和x2都是标量，则结果为标量。

```
>>> np.maximum([2, 3, 4], [1, 5, 2])           #比较两个数组的最大值
array([2, 5, 4])

#一个二阶单位矩阵和一个数组作比较，一维数组广播成二维数组
>>> np.maximum(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])

#只要有一个nan，就返回nan，当两个元素都是nan，返回第一个nan，最大值/最小值的机制一样
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([nan, nan, nan])
>>> np.minimum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([nan, nan, nan])
>>> np.maximum(np.inf, 1)
```

```
inf          #无穷大
>>> np.minimum(-np.inf, 1)
-inf         #负无穷
```

numpy.sign

numpy.sign(x)

对于array_like的输入，逐个对元素进行sign函数运算，最后输出相应的ndarray

x: array_like

输入值

→ y: ndarray

如果x是标量，则返回值也是个标量

1 对于一般的实数来说，其返回：

$$y = f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

2 对于复数来说，NumPy2.0之后，其返回：

$$y = f(x) = x.real/\sqrt{real^2 + imag^2} + x.imag/\sqrt{real^2 + imag^2} j$$

3 对于复杂的输入，sign 函数返回 $x/abs(x)$ ，这个定义是最常见并且最有用的一种。另外，对于 nan 的输入，输出也是 nan

```
#输入数组为一般数组
x = [2, -5.1, 0, np.nan]
>>> np.sign(x)
array([ 1., -1.,  0., nan])

#输入数组是复数数组
x = [3+4j, 5j]
>>> np.sign(x)
array([0.6+0.8j, 0. +1.j ])
```

numpy.sum

numpy.sum(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)

ndarray.sum

沿着给定轴求数组元素的总和

参数：

①a: array_like

要相加的元素

②axis: None 或 int或int的元组，可选

执行求和的一个轴或多个轴，默认情况下 `axis=None`，将对输入数组所有元素进行求和。如果 `axis` 为负数，则从最后一个轴计数到第一个轴

③ `dtype`: `dtype`, 可选

返回数组的类型以及用于累加元素的累加器的类型，默认情况下使用 `a` 的 `dtype`

④ `out`: `ndarray`, 可选

在其中放置结果的替代输出数组，它必须具有与预期输出相同的形状，如果需要，输出值的类型将被转换。

⑤ `keepdims`: 布尔值, 可选

如果设置为 `True`，则输出结果保持原数组维度不变

⑥ `initial`: 标量, 可选

求和的初始值

⑦ `where`: 布尔值的类数组对象, 可选

要在总和中包含的元素

返回值:

➡ `sum_along_axis`: `ndarray`

一个与 `a` 形状相同的数组，但指定的轴被移除，如果 `a` 是一个标量，或者如果 `axis=None`，则返回一个标量。如果指定了输出数组，则返回对 `out` 的引用

numpy.cumsum/cumprod

`numpy.cumsum(a, axis=None, dtype=None, out=None)`

`numpy.cumprod(a, axis=None, dtype=None, out=None)`

返回沿给定轴的元素累加和/累积乘积

参数:

① `a`: `array_like`

输入数组

② `axis`: `int`, 可选

计算累加和/累积乘积的轴，默认 `None` 是计算将数组展平的累加和/累积乘积

③ `dtype`: `dtype`, 可选

返回的数组类型以及元素求和的累加器类型，如果未指定，则默认为 `a` 的 `dtype`

④ `out`: `ndarray`, 可选

在其中放置结果的替代输出数组，它必须具有与预期输出相同的形状和缓冲区长度，如果有必要，类型将被强制转换

返回值:

➡ `cumsum_along_axis/cumprod`: `ndarray`

返回一个包含结果的新数组，如果指定了 `out`，则返回对 `out` 的引用。结果与 `a` 具有相同的大小，如果 `axis` 不是 `None` 或者 `a` 是一个一维数组，则结果与 `a` 有相同的形状。

! 例子：随机漫步

```
#有一点说明的是，代码实现是漫步达到2的某一行显示，这只是一个微小的例子
#只要调整数组大小，例如（5000， 1000）的数组，即可完成步数到30的条件
import numpy as np
nwalks = 3          #行走了多少次
nsteps = 5          #走了多少步
rng = np.random.default_rng()      #构造一个生成器
draws = rng.integers(0, 2, size=(nwalks, nsteps))  # 0 or 1
#
# todo: 把draws里大于0的改成1，否则的改成 -1
#保证每次往前或者往后走
steps = np.where(draws > 0, 1, -1)
print(steps)
# todo: 对每一个随机漫步过程（每一行），记录到每一步的累加和
#一行相当于走一次，在这一次里面计算累加和，就相当于相对于开始走的步数
#例如某一行行为[ 1  2  1  0  1]，则表示这个人
#
#           从0开始 -> 前进1步(1) -> 前进1步(2) -> 后退1步(1) -> 后退1步(0) -> 前进1步(1)
-> 结束
#其走路的步数为steps的长度，通过cumsum函数可以实现逐步计数，模拟这个过程
walks = np.cumsum(steps, axis=1)
print(walks)

print(walks.max()) # todo: 计算漫步的最大值
print(walks.min()) # todo: 计算漫步的最小值
#
# # 对于每一次漫步，计算达到30或-30的最小穿越时间
# # 但不一定5000个漫步过程都达到了30/-30
# todo: 检查每个漫步过程是否达到了 30/-30,一定是检查每一行，因为一行代表漫步一次
# 如果有一个达到了30步，就可以判定为True
hits30 = np.any(np.abs(walks) >= 2, axis=1)
print(hits30) # 提示: hits30.shape: (5000,)
# todo: 统计有多少次达到了30/-30的漫步过程的个数
# hits30是一个布尔数组，True是达到了30步，对应1，False对应0，所以对hits30直接计数即可
count30 = np.sum(hits30)
#
# # 获取达到30/-30的漫步过程，达到30/-30的最小步数（从0开始计数）
# 提示: 要用到hits30变量 和 np.argmax: 可以用多行代码
print(walks[hits30])
#numpy.argmax 有多个最大值时，返回第一个索引
crossing_times = np.argmax(np.abs(walks[hits30]) >= 2, axis=1) #达到了30、-30
print(f"最小步数为: {crossing_times}")
print(np.mean(hits30)) # todo: 计算到30/-30的的最小步数的平均值

#运行结果如下
[[ 1 -1  1 -1  1]
 [ 1  1 -1  1 -1]
 [ 1  1 -1 -1 -1]]
[[ 1  0  1  0  1]
 [ 1  2  1  2  1]
 [ 1  2  1  0 -1]]
2
-1
[False True True]
[[ 1  2  1  2  1]
 [ 1  2  1  0 -1]]
[1 1]
0.6666666666666666
```

np.ceil

np.ceil(x)

对x向上取整

np.floor

np.floor(x)

对x向下取整

```
# 如何对浮点数组进行四舍五入
z = np.random.uniform(-10, 10, 10)
print(Z)
np.where(z > 0, np.ceil(z), np.floor(z))

#运行结果如下
[ 9.70266411 -2.95467394 -4.08128861 -6.30774297 -0.50828123  2.77329432
  5.65194891  7.71743217  8.99669939 -9.89336276]
array([-6.,  1.,  1., -1.,  3., -1., -10.,  3., -9.,  2.])    #四舍五入后
```

NumPy常量

numpy.nan

IEEE 754 浮点数表示的非数字 (NaN)

```
print(0 * np.nan)      #0
print(np.nan == np.nan) #True
print(np.inf > np.nan) #True
print(np.nan - np.nan) #nan
print(np.nan in {np.nan}) #True
print(0.3 == 3 * 0.1)  #False
```

numpy.inf

IEEE 754 浮点数表示法的正无穷大

numpy.e

自然对数的底

```
e = 2.71828182845904523536028747135266249775724709369995...
```

numpy.pi

```
pi = 3.1415926535897932384626433...
```

numpy.newaxis

None的一个方便别名，适用于数组索引

```
print(np.newaxis is None)
a = np.arange(3)
print("=====原数组a的形状为(3,)=====")
print(a)           #[1, 2, 3]
print("=====形状变为(3,1)=====")
#下面两句等价
print(a[:, np.newaxis])
print(a[:, None])
print("=====形状变为(3,1,1)=====")
print(a[:, np.newaxis, np.newaxis])
print("=====形状为(3,1)，a广播=====")
print(a[:, np.newaxis] * a)
print("=====a[np.newaxis, :]等同于a[np.newaxis] 还等价于a[None]=====")
print(a[np.newaxis, :])
print(a[np.newaxis])
print(a[None])
```

#输出结果如下

```
True
=====原数组a的形状为(3,)=====
[0 1 2]
=====形状变为(3,1)=====
[[0]
 [1]
 [2]]
[[0]
 [1]
 [2]]
=====形状变为(3,1,1)=====
[[[0]]

 [[1]]

 [[2]]]
=====形状为(3,1)，a广播=====
[[0 0 0]
 [0 1 2]
 [0 2 4]]
=====a[np.newaxis, :]等同于a[np.newaxis] 还等价于a[None]=====
[[0 1 2]]
[[0 1 2]]
[[0 1 2]]
```

逻辑函数

numpy.all

numpy.all(a, axis=None, out=None, keepdims=<no value>, where=<no value>)

判断沿给定轴的 **所有** 数组元素是否为True

对于非数字 `nan`、正无穷和负无穷，它会被评估为True，因为这些不等于零

参数：

①a: array_like

可以转换为数组的输入数组或对象

②axis: None或int或int的元组，可选

默认 `axis=None`，这是对输入数组的所有维度执行逻辑与。`axis` 可以是负数，在这种情况下，它从最后一个轴计数到第一个轴。如果这是一个整数元组，则会在多个轴上执行缩减。

③out: ndarray，可选

在其中放置结果的备用输出数组

④keepdims: bool，可选

如果设置为 `True`，则保持原维度

⑤where: 类数组的布尔值，可选

检查所有True值时要包含的元素

返回值：

➡ all: ndarray, 布尔类型

除非指定了 `out`，否则将返回一个新的布尔值或数组，在这种情况下，将返回对out的引用

```
>>> np.all([[True,False],[True,True]])
False

>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False])

>>> np.all([1.0, np.nan])
True
```

numpy.any

numpy.any(a, axis=None, out=None, keepdims=<no value>, where=<no value>)

判断沿给定轴的 **任何一个** 数组元素是否为True。如果 `axis=None`，则返回单个布尔值

对于非数字 `nan`、正无穷和负无穷，它会被评估为True，因为这些不等于零

参数：

①a: array_like

可以转换为数组的输入数组或对象

②axis: None或int或int的元组, 可选

默认 `axis=None`, 这是对输入数组的所有维度执行逻辑与。`axis` 可以是负数, 在这种情况下, 它从最后一个轴计数到第一个轴。如果这是一个整数元组, 则会在多个轴上执行缩减。

③out: ndarray, 可选

在其中放置结果的备用输出数组

④keepdims: bool, 可选

如果设置为 `True`, 则保持原维度

⑤where: 类数组的布尔值, 可选

检查任何一个True值时要包含的元素

返回值:

➡ any: 布尔值或ndarray

除非指定了 `out`, 否则将返回一个新的布尔值或 `ndarray`, 在这种情况下, 将返回对 `out` 的引用

```
>>> np.any([[True, False], [True, True]])
True

>>> np.any([[True, False, True ],
            [False, False, False]], axis=0)
array([ True, False,  True])

>>> np.any([-1, 0, 5])
True

>>> np.any([[np.nan], [np.inf]], axis=1, keepdims=True)
array([[ True],
       [ True]])

>>> a = np.array([[1, 0, 0],
                  [0, 0, 1],
                  [0, 0, 0]])
>>> np.any(a, axis=0)           #沿着第一个轴看
array([ True, False,  True])
>>> np.any(a, axis=1)          #沿着第二个轴看
array([ True,  True, False])
```

#实例 (all和any)

```
# 已知一个二维数组 a 表示一批学生的作业成绩, 每一行代表一个学生, 每一列代表一次作业成绩。
import numpy as np
a = np.array([
    [80, 90, 85],
    [60, 75, 70],
    [100, 100, 100],
    [55, 40, 65]
])
print("=====1. 哪些学生所有作业都及格(即 ≥ 60)? =====")
print(a[np.all(a >= 60, axis=1)])
print("=====2. 哪些学生至少有一门作业不及格? =====")
```

```
print(a[np.any(a < 60, axis=1)])
print("=====3. 所有学生中，是否存在有人全都得了满分（100 分）？=====")
print(np.any(np.all(a == 100, axis=1)))

#输出结果如下
=====1. 哪些学生所有作业都及格（即 ≥ 60）？=====
[[ 80  90  85]
 [ 60  75  70]
 [100 100 100]]
=====2. 哪些学生至少有一门作业不及格？=====
[[55 40 65]]
=====3. 所有学生中，是否存在有人全都得了满分（100 分）？=====
True
```

numpy.isclose

numpy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)

返回一个布尔数组，如果两个数组逐元素比较在容差范围内，则对应的输出数组的元素为True，否则为False

对于有限值，isclose 使用以下方程来测试两个浮点数是否等价：

$$|a - b| \leq atol + rtol * |b|$$

参数：

①a,b: array_like

要比较的输入数组

②rtol: array_like

相对容差

③atol: array_like

绝对容差

④equal_nan: bool

是否将NaN 视为相等。如果为True，数组a中的NaN将被视为与数组b中的NaN相等

返回值：

→ y: array_like

返回一个布尔数组，表示a和b在给定的容差范围内是否相等。如果a和b都是标量，则返回单个布尔值

统计

numpy.mean

numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, where=<no value>)

计算沿指定轴的算术平均值

返回数组元素的平均值，默认情况下，平均值是基于展平的数组计算的，否则基于指定的轴计算。对于整数输出，使用

np.float64 作为中间值和返回值

参数:

①a: array_like

输入的数字数组

②axis: None 或 int或int的元组, 可选

计算均值沿着的一个轴或多个轴, 默认计算展平数组的均值。如果axis是一个整数的元组, 则会在多个轴上执行平均值。

③dtype: 数据类型, 可选

用于计算平均值的类型, 对于整数输入, 默认值为 `np.float64`; 对于浮点输入, 它与输入的dtype相同

④out: ndarray, 可选

在其中放置结果的备用输出结果, 默认为None

⑤keepdims: bool, 可选

如果设置为True, 则保留原有维度

⑥where: 类数组的布尔值, 可选

要在均值中包含的元素

返回值:

➡ m: ndarray, 见上面的dtype参数

如果out=None, 返回一个包含平均值的新数组, 否则返回对输出数组的引用

```
a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)           #沿第一个轴计算平均值
array([2., 3.])
>>> np.mean(a, axis=1)           #沿第二个轴计算平均值
array([1.5, 3.5])

a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
>>> np.mean(a)
12.0
>>> np.mean(a, where=[[True], [False], [False]])
9.0                               #(5+9+13) / 3 = 9   where表示只选择第一行算其平均值
```

numpy.std/var

`numpy.std(a, axis=None, dtype=None, ...)`

`numpy.var(a, axis=None, dtype=None, ...)`

计算沿指定轴的数组的标准差/方差

返回数组元素的标准差/方差, 默认情况下, 标准差是基于展平数组计算的, 否则在指定的轴上计算

参数:

①a: array_like

输入数组

②axis: None或int或int的元组, 可选

计算标准差/方差沿着的一个轴或多个轴, 默认计算展平数组的标准差/方差。如果axis是一个整数的元组, 则会沿多个轴计算标准差/方差。

③dtype: dtype, 可选

用于计算标准差/方差的类型, 对于整数类型的数组, 默认值是 `np.float64`, 对于浮点类型的数组, 它与数组类型相同

返回值:

→ standard_deviation/variance: ndarray, 参见上面的dtype参数

如果out是None, 返回一个包含标准差/方差的新数组, 否则返回对输出数组的引用

NumPy文件

🔑 [引言]格式化

A NPY / NPZ格式

一个简单的格式, 用于将NumPy数组连同它们的所有信息一起保存到磁盘上。

`.npy` 格式是NumPy中用于持久化 [单个] 任意NumPy数组到磁盘的标准二进制文件格式。该格式存储了重建数组所需的所有形状和数据类型信息, 即使在具有不同架构的另一台机器上也能正确重建。

`.npz` 格式是持久化 [多个] NumPy数组到磁盘的标准格式, 一个 `.npz` 文件是包含多个 `.npy` 文件的zip文件, 每个文件对应一个数组。

B 格式化/fmt参数

fmt参数: %[flag]width[.precision]specifier

1 flag:

- : 左对齐
- +: 强制在结果前加上 + 或 -
- 0: 用零而不是空格左填充数字 (见width)

2 width:

要打印的最小字符数, 如果值包含更多字符, 则不会截断

3 precision: 精度

- 对于整数说明符 (例如 `d, i, o, x`), 表示最小的数字位数
- 对于 `e`, `E` 和 `f` 说明符, 表示小数点后要打印的位数
- 对于 `g` 和 `G`, 表示最大有效数字位数
- 对于 `s`, 表示字符的最大数量

4 specifier: 说明符

- `c`: 字符
- `d` 或 `i`: 有符号十进制整数
- `e` 或 `E`: 使用 `e` 或 `E` 的科学计数法
- `f`: 十进制浮点数

`g` 或 `G`: 使用 `e`, `E` 或 `f` 中较短的一个

`o`: 有符号八进制

`s`: 字符串

`u`: 无符号十进制整数

`x`, `X`: 无符号十六进制整数

numpy.save

numpy.save(file, arr, allow_pickle=True)

将数组保存到NumPy的 `.npy` 格式的二进制文件中

参数:

①file: 文件对象, 字符串或路径

要保存数据的文件或文件名, 如果file是一个文件对象, 则文件名保持不变。如果file是一个字符串或路径, 而且文件名还没有 `.npy` 扩展名, 则会附加该扩展名。

②arr: array_like

要保存的数组数据

③allow_pickle: bool, 可选

默认为True, 允许使用Python pickles保存对象数组

numpy.savez/savez_compressed

numpy.savez(file, *args, **kwargs)

将多个数组保存到一个 未压缩/已压缩 的 `.npz` 格式文件中

在输出文件中将数组作为关键字参数提供, 以便将它们存储在相应的名称下: `savez/savez_compressed(fn, x=x, y=y)`

如果数组被指定为位置参数, 即 `savez_compressed(fn, x, y)`, 它们的名称将是 `arr_0`, `arr_1` 等

参数:

①file: 文件对象, 字符串或路径

数据将被保存的文件名 (字符串) 或打开的文件 (类文件对象)。如果文件是字符串或路径, 而且文件名还没有扩展名, 将附加 `.npz` 扩展名

②args: 位置参数, 可选

要保存到文件的数组, 请使用关键字参数 (kwargs) 为数组分配名称。使用位置参数传入的数组将被命名为 `arr_0`, `arr_1` 依此类推

③kwargs: 关键字参数, 可选

要保存到文件的数组。每个数组将与其对应的键名一起保存到输出文件中

numpy.savetxt

```
numpy.savetxt(fname, X, fmt='%%.18e, delimiter=',', newline='\n', header="", footer="", comments='# ', encoding=None)
```

参数:

①fname: 文件名, 文件句柄或路径

如果文件名以 `.gz` 结尾, 文件将自动以压缩的gzip格式保存。 `loadtxt` 可以透明的理解 gzipped 文件

②X: 一维或二维 的array_like

要保存到文本文件的数据

③fmt: str或str序列, 可选

数据的格式: 单一格式、一系列格式或者一个多格式字符串

④delimiter: str, 可选

分隔列的字符串或字符, 默认以空格分隔

⑤newline: str, 可选

分隔行的字符串或字符

⑥header: str, 可选

将写入文件开头的字符串

⑦footer: str, 可选

将在文件末尾写入的字符串

⑧comments: str, 可选

将添加到 header 和 footer 字符串前面的字符串, 以将它们标记为注释, 默认值'# '。

⑨encoding: {None, str}可选

字符编码集

numpy.load

```
numpy.load(file, encoding='ASCII')
```

从 `.npy`, `.npz` 文件中加载数组

参数:

①file: 类文件对象、字符串或路径

要读取的文件。类文件对象必须支持seek和read方法, 并且始终以二进制模式打开。

②encoding: str, 可选

默认值 `ASCII`, 只能使用 `latin1`, `ASCII` 和 `bytes`

返回值:

👉 result: 数组、元组、字典等等

返回文件中存储的数据

- 如果文件是 `.npy` 文件，则返回一个数组
- 如果文件是一个 `.npz` 文件，那么返回一个类似字典的对象，包含 {文件名: 数组} 键值对，每个文件在存档中对应一个
- 如果文件是一个 `.npz` 文件，返回的值支持上下文管理器协议，类似于 `open` 函数：

```
with load('foo.npz') as data:  
    a = data['a']
```

numpy.loadtxt

`numpy.loadtxt(fname, comments='#', delimiter=None, skiprows=0, usecols=None, encoding=None, quotechar=None)`

从文本文件中加载数据

参数：

① **fname**：文件对象，字符串，字符串列表，路径或生成器

要读取的文件、文件名、列表或生成器。如果文件扩展名是 `.gz` 或 `.bz2`，则先解压缩文件

② **comments**：str或str序列或None，可选

用于表示注释开始的字符或字符列表

③ **delimiter**：str，可选

用于分隔值的字符，只支持单字符分隔符，`\n` 不能用作分隔符，默认是空白

④ **skiprows**：int，可选

跳过头skiprows行，包括注释，默认跳过第一行，即表头

⑤ **usecols**：整数或序列，可选

要读取哪些列，其中0是第一列，默认值为None，表明将读取所有列

⑥ **encoding**：str，可选

用于解码输入文件的编码。

⑦ **quotechar**：Unicode字符或None，可选

用于表示引用项开始和结束的字符，如果找到了连续的quotechar，则第一个被视为转义字符。

默认为None，如果要写，一般是 `quotechar='\"'`

返回值：

👉 out: ndarray

从文本文件中读取的数据

numpy.genfromtxt

numpy.genfromtxt(fname, delimiter=None, ...)

从文本文件中加载数据，适合有缺失值/带列名的文本

它将把文件中的缺失值读成 `nan`

```
# 题目1
-----

# 有如下二维数组：
import numpy as np

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

# 请将其保存为array_simple.txt文本文件，并读取回来，要求数据分隔符为逗号。请写出保存和读取的完整代码。
np.savetxt('array_simple.txt', arr, delimiter=',')
arr1=np.loadtxt('array_simple.txt', delimiter=',')
>>> print(arr1)
[[1. 2. 3.]
 [4. 5. 6.]]

# 题目2
-----

# 现有文件people.txt，内容如下（每行为姓名、年龄、身高）：
#Tom 181 75.0
#Jerry 22 180.0
#Ben 21 170.0

# 请用NumPy读取该文件为结构化数组（字段名为name、age、height），并输出所有年龄大于20的人的姓名。
persontype = np.dtype({
    'names':['name','age','height'],
    'formats':['<U10','<i','<f']
})
#此处使用字典的方式创建结构化数组

arr = np.loadtxt('people.txt', dtype=persontype) #loadtxt的dtype也可以是自己定义的结构化数据类型
>>> print(f"年龄大于20的人的姓名为: {arr[arr['age'] > 20]['name']}")
年龄大于20的人的姓名为: ['Jerry' 'Ben']

# 题目3
-----

# 假设有array_missing.txt文件，其内容如下：
# 1.0, 2.0,,
# 3.0,, 5.0, 6.0
# , 4.0, 5.0,

#有缺失值的文件使用genfromtxt读取，会把缺失值填充为nan
arr = np.genfromtxt('array_missing.txt', delimiter=',')
print(arr)
print("=====计算忽略nan的每列的均值=====")
print(np.nanmean(arr, axis=0))

#输出结果如下
[[ 1.  2. nan nan]
 [ 3. nan  5.  6.]
 [nan  4.  5. nan]]
=====计算忽略nan的每列的均值=====
[2.  3.  5.  6.]
```

题目4

```
-----  
# 请使用NumPy生成一个3行4列的随机浮点数组，分别保存为.txt（文本）、.npy（NumPy专有格式）和.npz（压缩格式，保存为自定义的键）。  
# 然后分别加载这三种格式的数据，并判断内容是否完全一致。  
arr1 = np.random.randn(3, 4)  
arr2 = np.random.randn(4, 3)  
#数据保存  
#保存数组 arr1 到 sample1.txt  
np.savetxt('sample1.txt', arr1, delimiter=',')  
#保存数组 arr1 到 sample2.npy 中，如果不写后缀名默认 .npy  
np.save('sample2', arr1)  
#保存数组 arr1和arr2到arr.npz中,使用关键字参数自定义两个数组的键  
np.savez('arr.npz', arr34=arr1, arr43=arr2)  
  
#数据读取  
out_arr1 = np.loadtxt('sample1.txt', delimiter=',', encoding='utf-8')  
print("====这是从sample1.txt文件中读取的数据====")  
print(out_arr1)  
  
out_arr2 = np.load('sample2.npy')  
print("====这是从sample2.npy文件中读取的数据====")  
print(out_arr2)  
  
print("====这是从arr.npz文件中读取的数据====")  
out_dict = np.load('arr.npz')  
print("====这是第一个数组====")  
print(out_dict['arr34'])  
print("====这是第二个数组====")  
print(out_dict['arr43'])  
  
#输出结果如下  
====这是从sample1.txt文件中读取的数据====  
[[ 0.6128446  0.0704319  0.14396492 -0.89066807]  
 [ 1.19444392 -0.80224298  0.22585326 -1.27494367]  
 [ 0.06029224  1.90356456  0.32605585  0.08107159]]  
====这是从sample2.npy文件中读取的数据====  
[[ 0.6128446  0.0704319  0.14396492 -0.89066807]  
 [ 1.19444392 -0.80224298  0.22585326 -1.27494367]  
 [ 0.06029224  1.90356456  0.32605585  0.08107159]]  
====这是从arr.npz文件中读取的数据====  
====这是第一个数组====  
[[ 0.6128446  0.0704319  0.14396492 -0.89066807]  
 [ 1.19444392 -0.80224298  0.22585326 -1.27494367]  
 [ 0.06029224  1.90356456  0.32605585  0.08107159]]  
====这是第二个数组====  
[[ 1.35970121  0.66332351  0.95250609]  
 [-2.29802752 -0.50898919  0.30596031]  
 [ 0.31023254  0.42633846 -0.05764889]  
 [-0.55459136  0.13286064 -0.66911576]]
```

numpy.frombuffer

numpy.frombuffer(buffer, dtype=float, count=-1, offset=0, like=None)

将缓冲区转换为二维数组

参数：

①buffer: buffer_like

一个缓冲区接口的对象

②dtype: 数据类型, 可选

返回数组的数据类型, 默认float

③count: int, 可选

要读取的项目数量, -1 表示缓冲区的所有数据

④offset: int, 可选

从该偏移量 (以字节为单位) 开始读取缓冲区, 默认值0

⑤like: array_like, 可选

引用对象, 以允许创建不是NumPy数组的数组

返回值:

→ out: ndarray

示例

```
import numpy as np
>>> s = b'hello world'
>>> np.frombuffer(s, dtype='s1', count=5, offset=6)      #表示从第六个字节开始, 只读五个字节
array([b'w', b'o', b'r', b'l', b'd'], dtype='|s1')
>>> np.frombuffer(b'\x01\x02', dtype=np.uint8)          #默认读取所有数据, 指定dtype为无符号的八位整数
array([1, 2], dtype=uint8)
>>> np.frombuffer(b'\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)  #读取三个字节, 指定dtype为无符号八位整数
array([1, 2, 3], dtype=uint8)
```

一般 frombuffer 用来操纵字节文件 (多半和其余语言进行交互, 这种文件可以是音频、视频等任何格式的文件), 此处的字节文件是一个二进制的字节流文件。

```
import wave
import numpy as np

with wave.open('ringing.wav') as wf:
    data = wf.readframes(wf.getnframes())      #将打开的音频文件转换为二进制数据
    data1 = np.frombuffer(data, dtype='float32')  #将二进制文件转换为ndarray, 指定读取后的ndarray的dtype=float32
>>> print(data1)
[0. 0. 0. ... 0. 0. 0.]                      #读取文件之后的ndarray
```

#通用函数

numpy.ufunc.outer

numpy.ufunc.outer(A, B, **kwargs)

若 $a \in A, b \in B$, 则将所有的ufunc的 op 应用到 (a, b) 对中。

如果A和B是一维的, 这个函数等价于:

```

r = np.empty(len(A), len(B))
for i in range(len(A)):
    for j in range(len(B)):
        r[i, j] = op(A[i], B[j])          #op = ufunc

```

如果A和B是多维，则输出数组的维度的形状为 (A.shape, B.shape)，就像：

$$C[[i_0, i_1, \dots, i_{M-1}], [j_0, j_1, \dots, j_{N-1}]] = op(A[i_0, i_1, \dots, i_{M-1}], B[j_0, j_1, \dots, j_{N-1}])$$

其中 $M = A.ndim, N = B.ndim$

A和B为一维数组

```

>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])

```

例：47. Given two arrays, X and Y, construct the Cauchy matrix C ($C_{ij} = 1/(x_i - y_j)$) (★★☆)

```

#使用for循环实现
X = np.random.randn(5)
Y = np.random.randn(5)
C = np.zeros((5, 5))
for i in range(len(X)):
    for j in range(len(Y)):
        C[i, j] = 1 / (X[i] - Y[j])
>>> print(C)
[[ 0.58617142  4.01110834  0.61182571  0.58654482  0.39822889]
 [ 3.53545049 -0.85191315  4.73224541  3.54907798  0.91913247]
 [ 1.1605019  -1.68072361  1.26556179  1.16196642  0.59994149]
 [ 6.16370008 -0.77253619 11.02450742  6.20523902  1.03372713]
 [ 0.47017315  1.49209624  0.48653684  0.47041336  0.34106306]]

#使用ufunc.outer
C = np.zeros((5, 5))
C = 1.0 / np.subtract.outer(X, Y)
>>> print(C)
[[ 0.58617142  4.01110834  0.61182571  0.58654482  0.39822889]
 [ 3.53545049 -0.85191315  4.73224541  3.54907798  0.91913247]
 [ 1.1605019  -1.68072361  1.26556179  1.16196642  0.59994149]
 [ 6.16370008 -0.77253619 11.02450742  6.20523902  1.03372713]
 [ 0.47017315  1.49209624  0.48653684  0.47041336  0.34106306]]

```

A和B是多维数组

```

A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1, 2, 3, 4]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
(2, 3, 1, 4)
array([[[[ 1,  2,  3,  4]],
        [[ 2,  4,  6,  8]],
        [[ 3,  6,  9, 12]]],
       [[[ 4,  8, 12, 16]]],

```

```
[[ 5, 10, 15, 20]],  
[[ 6, 12, 18, 24]]])
```

numpy.ufunc.reduce

numpy.ufunc.reduce(array, axis=0, dtype=None, out=None, keepdims=False, initial=<no value>, where=True)

沿某个轴应用ufunc，将array的维度减少一维

对于一维数组，reduce产生的结果等同于：

```
for i in range(len(A)):  
    r = op(r, A[i])          #op = ufunc  
return r
```

例如：add.reduce()等同于sum()

参数：

①array: array_like

输入数组

②axis: None 或 int 或int的元组，可选

默认axis=0，对输入数组的第一个维度进行归约操作；如果为None，对所有轴进行操作。

③dtype: 数据类型，可选

指定输出的数据类型，如果为None，则根据原array推断

④out: ndarray, None或元组，可选

如果提供，则函数返回的数据保存在out参数中

⑤keepdims: bool，可选

如果为True，则保持原数组的维度

⑥initial: 标量，可选

初始值，用于ufunc的op操作，例如：若add.reduce(array)返回的结果为sum，则如果指定了initial，则结果变为sum+initial

⑦where: bool或array_like，可选

用于表示哪个或哪些元素要被计算

返回值：

➡ r: ndarray

缩减后的数组

A 对于一维数组应用 reduce

```
>>> np.multiply.reduce([2,3,5])  
30          #2x3x5 = 30  
>>> np.multiply.reduce([2,3,5], initial=4)
```

```

120      #4x2x3x5 = 120

41.How to sum a small array faster than np.sum? (★★☆)
>>> np.add.reduce([2,3,5])
10      #2+3+5 = 10

>>> np.add.reduce([2,3,5], initial=5)
15      #5+2+3+5 = 10

>>> a = np.array([10., np.nan, 10])
>>> np.add.reduce(a, where=~np.isnan(a))      #这里非nan的没有计算
20.0

```

B 多维数组应用 reduce

```

X = np.arange(8).reshape((2,2,2))
>>> X
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> np.add.reduce(X, 0)
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X)      # 默认归约轴为第一个轴
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X, 1)
array([[ 2,  4],
       [10, 12]])
>>> np.add.reduce(X, 2)
array([[ 1,  5],
       [ 9, 13]])

```