



Table of Contents

- 1 广播
- 2 通函数
- 3 基本操作
 - 3.1 Shape
 - 3.2 轴变换
 - 3.3 维度
 - 3.4 构造
 - 3.5 增删元素
 - 3.6 重排元素
- 4 排序搜索
 - 4.1 极值
 - 4.2 搜索
 - 4.3 排序
- 5 集合操作
 - 5.1 包含
 - 5.2 交集
 - 5.3 并集
 - 5.4 差集
 - 5.5 异或集
- 6 函数式编程
- 7 测试
 - 7.1 相等
 - 7.2 相近
 - 7.3 小于
 - 7.4 异常
- 8 小结
- 9 参考

```
In [730... import numpy as np
np.__version__
```

```
Out[730... '1.22.3'
```

文档阅读说明：

-  表示 Tip
-  表示注意事项

广播

广播描述了NumPy在数值计算如何处理不同形状的数组。在一定限制条件下，小数组会广播到大数组以适配其形状。

```
In [464... # 最简单的例子
a = np.array([1., 2., 3.])
a
```

```
Out[464... array([1., 2., 3.])
```

```
In [465... a * 2
```

```
Out[465... array([2., 4., 6.])
```

```
In [466... # 上面的例子等价于
b = np.array([2, 2, 2])
a * b
```

```
Out[466... array([2., 4., 6.])
```

广播的规则如下：

- 从右到左比较
- 兼容时相等或某个维度为1
- 数组可以不同维度

```
In [467... a = np.ones((8, 1, 6, 1))
b = np.ones((7, 1, 5))
```

```
In [468... # b 等价于变成了 (1, 7, 1, 5) 的shape
c = a + b
c.shape
```

```
Out[468... (8, 7, 6, 5)
```

```
In [475... # 不能广播的例子
a = np.ones((2, 3, 4))
b = np.ones((2, 3))
a + b
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-475-796f45d71953> in <module>
      2 a = np.ones((2, 3, 4))
      3 b = np.ones((2, 3))
----> 4 a + b

ValueError: operands could not be broadcast together with shapes (2,3,4) (2,3)
```

```
In [476... # 当然一维向量数字的数量就是维度
# 这样是不行的
a = np.ones((2, 3))
b = np.ones((2, ))
b.shape
```

```
Out[476... (2,)
```

```
In [472... a + b
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-472-bd58363a63fc> in <module>
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

```
In [473... # 这样就可以
b = np.ones((3, ))
a + b
```

```
Out[473... array([[2., 2., 2.],
        [2., 2., 2.]])
```

大部分时候我们可以无伤地使用它，但当数据量很大时，广播会复制数组，可能会导致内存溢出。

```
In [477... rng = np.random.default_rng(42)
a = rng.random((102400, 256, 64))
b = rng.random((102400, 256))
c = rng.random((256, 64))
```

```
In [478... # a=12.5G
102400*256*64*8/1024/1024/1024
```

```
Out[478... 12.5
```

```
In [ ]: # 如果内存小于25G，这里会超出内存，因为差的结果还是会存在临时空间
a[:, :] = b[:, :, np.newaxis] - c
```

```
In [ ]: # 可以用之前提到的 out 参数，不会额外增加内存
d = np.subtract(b[:, :, np.newaxis], c, out=a)
```

不过，一般会使用虚拟内存，不用过于担心。

通函数

通函数 (ufunc) 是以逐个元素的方式对ndarray进行操作的函数，支持数组广播、类型转换等。在NumPy中，通用函数都是 `np.ufunc` 的实例。

```
In [479... isinstance(np.add, np.ufunc)
```

```
Out[479... True
```

+ 是 `np.add` 的快捷方式，其他的函数也类似：

```
In [481... x1 = np.arange(9).reshape((3, 3))
x2 = np.arange(3)
```

```
In [482... x1 + x2
```

```
Out[482... array([[ 0,  2,  4],
        [ 3,  5,  7],
        [ 6,  8, 10]])
```

```
In [483... np.add(x1, x2)
```

```
Out[483... array([[ 0,  2,  4],  
        [ 3,  5,  7],  
        [ 6,  8, 10]])
```

```
In [484... x1 * x2
```

```
Out[484... array([[ 0,  1,  4],  
        [ 0,  4, 10],  
        [ 0,  7, 16]])
```

```
In [485... np.multiply(x1, x2)
```

```
Out[485... array([[ 0,  1,  4],  
        [ 0,  4, 10],  
        [ 0,  7, 16]])
```

out 参数主要是用来存储计算结果:

```
In [486... x3 = np.zeros((3, 3), dtype=np.int8)  
x3
```

```
Out[486... array([[0, 0, 0],  
        [0, 0, 0],  
        [0, 0, 0]], dtype=int8)
```

```
In [487... np.add(x1, x2, out=x3)
```

```
Out[487... array([[ 0,  2,  4],  
        [ 3,  5,  7],  
        [ 6,  8, 10]], dtype=int8)
```

```
In [488... x3
```

```
Out[488... array([[ 0,  2,  4],  
        [ 3,  5,  7],  
        [ 6,  8, 10]], dtype=int8)
```

where 参数确定哪些可以存储:

```
In [489... x4 = np.zeros((3, 3), dtype=np.int8)  
x4
```

```
Out[489... array([[0, 0, 0],  
        [0, 0, 0],  
        [0, 0, 0]], dtype=int8)
```

```
In [490... np.add(x1, x2, out=x4, where=[True, False, False])
```

```
Out[490... array([[0, 0, 0],  
        [3, 0, 0],  
        [6, 0, 0]], dtype=int8)
```

```
In [491... x4
```

```
Out[491... array([[0, 0, 0],  
        [3, 0, 0],  
        [6, 0, 0]], dtype=int8)
```

NumPy中的ufunc数量很多，包括数学运算、三角函数、位操作、逻辑函数、浮点函数等，具体可查看：

- <https://numpy.org/devdocs/reference/ufuncs.html#available-ufuncs>

ufunc可以被 `__array_ufunc__` 方法覆盖，具体可参考第一章《核心概念：自定义数组容器》。

ufunc支持以下方法：

- `reduce`：沿着某个维度累积
- `accumulate`：所有元素累积
- `reduceat`：沿着某个维度指定的slice累积
- `outer`：对A和B中所有元素对运算
- `at`：对指定索引的元素的执行无缓冲的就地运算

```
In [492...] a = np.arange(12).reshape(4, 3)
a
```

```
Out[492...] array([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

`reduce` 的参数和初级课程介绍的很多接口差不多：

- `array`：数组
- `axis`：维度
- `dtype`：数据类型
- `out`：同上
- `where`：同上
- `keepdims`：是否保持维度，《基础教程》中有介绍
- `initial`：初始值

```
In [493...] np.add.reduce(a, axis=0, initial=10)
```

```
Out[493...] array([28, 32, 36])
```

```
In [494...] np.add.reduce(a, axis=1, initial=10, keepdims=True)
```

```
Out[494...] array([[13],
        [22],
        [31],
        [40]])
```

`accumulate` 的参数少很多：

- `array`：数组
- `axis`：维度
- `dtype`：数据类型
- `out`：同上

In [495...

```
a
```

Out[495...

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [496...

```
# 沿着行
np.multiply.accumulate(a, axis=1)
```

Out[496...

```
array([[ 0,  0,  0],
       [ 3, 12, 60],
       [ 6, 42, 336],
       [ 9, 90, 990]])
```

In [497...

```
np.multiply.accumulate(a)
```

Out[497...

```
array([[ 0,  1,  2],
       [ 0,  4, 10],
       [ 0, 28, 80],
       [ 0, 280, 880]])
```

`reduceat` 相比 `accumulate` 增加了索引位置:

- `indices`: `index`的复数
- 其他参数同 `accumulate`

计算 `array[indices[i]:indices[i+1]]`, `i` 表示第*i*行/列, 计算规则如下:

- 当 `i = len(indices) - 1` (最后一个index) : `indices[i+1] = array.shape[axis]`
- 当 `indices[i] >= indices[i + 1]`, 第*i*个就是 `array[indices[i]]`
- 当 `indices[i] >= len(array)` 或 `indices[i] < 0`, 错误

In [498...

```
a
```

Out[498...

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [499...

```
np.add.reduceat(a, [1,2])
```

Out[499...

```
array([[ 3,  4,  5],
       [15, 17, 19]])
```

In [500...

```
# 四列
# 第0列: indices[0]: indices[1], 即第0:2累积 (0+1列)
# 第1列: indices[1] > indices[2], 等于第2列
# 第2列: indices[2] > indices[3], 等于第1列
# 第3列: 最后一个, indices[3]: indices[a.shape[1]=3], 等于0:3累积 (0,1,2列)
np.add.reduceat(a, [0, 2, 1, 0], axis=1)
```

Out[500...

```
array([[ 1,  2,  1,  3],
       [ 7,  5,  4, 12],
       [13,  8,  7, 21],
       [19, 11, 10, 30]])
```

`outer` 接受两个数组，等价于以下结果：

```
r = 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])
```

```
In [501... np.add.outer(a, a).shape
```

```
Out[501... (4, 3, 4, 3)
```

```
In [502... np.add.outer([1,2,3], [4,5,6])
```

```
Out[502... array([[5, 6, 7],
        [6, 7, 8],
        [7, 8, 9]])
```

`at` 参数如下：

- a: 数组
- indices: 索引
- b: 两个操作对象时另一个操作对象

```
In [503... a = np.arange(12).reshape(4, 3)
a
```

```
Out[503... array([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

```
In [504... # 对第0和1行加1
np.add.at(a, [0, 1], [1])
```

```
In [505... a
```

```
Out[505... array([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

```
In [506... # 4x3 和 1x3 可以通过广播运算
a = np.arange(12).reshape(4, 3)
a
```

```
Out[506... array([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

```
In [507... np.add.at(a, [0, 1], np.array([[1, 2, 3]]))
a
```

```
Out[507... array([[ 1,  3,  5],
        [ 4,  6,  8],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

可以使用 `np.frompyfunc` 来生成通函数实例。本章后面会进一步介绍，不再赘述。

基本操作

其中不少常用的操作我们在《从入门到小白》中已有介绍，比如 `shape` , `reshape` , `squeeze` , `expand_dims` , `stack` , `concatenate` , `split` , `repeat` 等，接下来就介绍下剩下的一些使用频率稍低但也比较重要的操作。

Shape

```
In [194...] rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 4))
a
```

```
Out[194...] array([[0, 7, 6, 4],
        [4, 8, 0, 6],
        [2, 0, 5, 9]])
```

```
In [202...] # flat
list(a.flat)
```

```
Out[202...] [0, 7, 6, 4, 4, 8, 0, 6, 2, 0, 5, 9]
```

```
In [204...] # 返回copy
a.flatten()
```

```
Out[204...] array([0, 7, 6, 4, 4, 8, 0, 6, 2, 0, 5, 9])
```

```
In [208...] # 不同的Style
a.flatten("F")
```

```
Out[208...] array([0, 4, 2, 7, 8, 0, 6, 0, 5, 4, 6, 9])
```

```
In [211...] # 返回view
np.ravel(a)
```

```
Out[211...] array([0, 7, 6, 4, 4, 8, 0, 6, 2, 0, 5, 9])
```

```
In [212...] np.ravel(a, "F")
```

```
Out[212...] array([0, 4, 2, 7, 8, 0, 6, 0, 5, 4, 6, 9])
```

轴变换

```
In [216...] a = np.ones((3, 4, 5))
```

```
In [217...] np.moveaxis(a, 0, 1).shape
```

```
Out[217...] (4, 3, 5)
```

```
In [229...] # 坐标轴的值可以是数组
np.moveaxis(a, [0, 2], [1, 0]).shape
```


Out[229...] (5, 3, 4)

```
In [230...] # 坐标轴的值必须是整数
np.swapaxes(a, 0, 1).shape
```

Out[230...] (4, 3, 5)

维度

```
In [245...] np.atleast_1d(1)
```

Out[245...] array([1])

```
In [246...] np.atleast_1d(1, 2)
```

Out[246...] [array([1]), array([2])]

```
In [252...] np.atleast_2d(1, 2)
```

Out[252...] [array([[1]]), array([[2]])]

```
In [247...] a = np.ones((2, 3))
```

```
In [255...] np.atleast_3d(a).shape
```

Out[255...] (2, 3, 1)

构造

`block` 经常被用于构建block矩阵。

```
In [301...] a = np.eye(2)*2
b = np.eye(3)*3
```

```
In [303...] np.block([
    [a, np.zeros((2, 3))],
    [np.ones((3, 2)), b]
])
```

Out[303...] array([[2., 0., 0., 0., 0.],
 [0., 2., 0., 0., 0.],
 [1., 1., 3., 0., 0.],
 [1., 1., 0., 3., 0.],
 [1., 1., 0., 0., 3.]])

另外，`block`在某些情况下等价于其他几个API：

- `depth=1`时，可以作为 `hstack`
- `depth=2`时，可以作为 `vstack`
- 可以替换 `atleast_1d` 和 `atleast_2d`

《从小白到入门》中我们提过，`concatenate` 和 `hstack`，`vstack`，`dstack` 是可以通用的，前者加上不同的维度就可以达到后几者的效果。

不过后几个是可以处理0维的（也就是整数），前者不可以；处理向量时也有一些不同。如果是二维以上的，建议还按照之前的做法；对于一维或零维的，也可以先处理一下再按之前的做法。

```
In [323... a = np.array([0, 1, 2])
```

```
In [324... np.hstack((a, 3))
```

```
Out[324... array([0, 1, 2, 3])
```

```
In [327... # 这样方可以  
np.concatenate((a, [3]))
```

```
Out[327... array([0, 1, 2, 3])
```

```
In [333... np.vstack((a, (3,4,5)))
```

```
Out[333... array([[0, 1, 2],  
        [3, 4, 5]])
```

```
In [351... np.concatenate((np.atleast_2d(a), [[3,4,5]]))
```

```
Out[351... array([[0, 1, 2],  
        [3, 4, 5]])
```

`tile` 用于构建对输入数组值重复给定次数的数组。

```
In [355... a = np.array([0, 1, 2])
```

```
In [357... np.tile(a, 2)
```

```
Out[357... array([0, 1, 2, 0, 1, 2])
```

注意与 `repeat` 的区别：

```
In [359... np.repeat(a, 2)
```

```
Out[359... array([0, 0, 1, 1, 2, 2])
```

可以指定重复多个维度：

```
In [364... np.tile(a, (2, 2, 2)).shape
```

```
Out[364... (2, 2, 6)
```

对于多维数组也是类似的：

```
In [366... b = np.array([[1,2], [3,4]])
```

```
In [368... np.tile(b, 2)
```

```
Out[368... array([[1, 2, 1, 2],  
        [3, 4, 3, 4]])
```

```
In [374... np.repeat(b, 2, axis=1)
```

```
Out[374...] array([[1, 1, 2, 2],
          [3, 3, 4, 4]])
```

```
In [377...] np.tile(b, (4, 1)).shape
```

```
Out[377...] (8, 2)
```

增删元素

`delete` 用来删除元素，如果不指定`axis`，则会被打平；否则会删除对应`index`的所有元素。

```
In [396...] a = np.arange(6).reshape(3, 2)
a
```

```
Out[396...] array([[0, 1],
          [2, 3],
          [4, 5]])
```

```
In [397...] np.delete(a, 5)
```

```
Out[397...] array([0, 1, 2, 3, 4])
```

```
In [398...] np.delete(a, [3, 5])
```

```
Out[398...] array([0, 1, 2, 4])
```

```
In [400...] # 指定axis=0, 删除行
np.delete(a, 1, 0)
```

```
Out[400...] array([[0, 1],
          [4, 5]])
```

`insert` 用来插入元素，与删除类似。

```
In [402...] a
```

```
Out[402...] array([[0, 1],
          [2, 3],
          [4, 5]])
```

```
In [405...] # 在index=1的位置插入-1, 不指定axis
np.insert(a, 1, -1)
```

```
Out[405...] array([ 0, -1,  1,  2,  3,  4,  5])
```

```
In [406...] # 在index=[1,2]的位置插入-1, 不指定axis
np.insert(a, [1,2], -1)
```

```
Out[406...] array([ 0, -1,  1, -1,  2,  3,  4,  5])
```

```
In [413...] # 在index=1的位置插入[-1, -2], 不指定axis
np.insert(a, 1, [-1, -2])
```

```
Out[413...] array([ 0, -1, -2,  1,  2,  3,  4,  5])
```

```
In [414... # 在index=[1,2]的位置插入[-1, -2], 不指定axis
np.insert(a, [1,2], [-1, -2])
```

```
Out[414... array([ 0, -1,  1, -2,  2,  3,  4,  5])
```

```
In [422... # 指定axis, index=1的位置插入
np.insert(a, 1, -1, axis=0)
```

```
Out[422... array([[ 0,  1],
        [-1, -1],
        [ 2,  3],
        [ 4,  5]])
```

```
In [423... # index=1的位置插入不同值
np.insert(a, 1, [-1, -2], axis=0)
```

```
Out[423... array([[ 0,  1],
        [-1, -2],
        [ 2,  3],
        [ 4,  5]])
```

```
In [426... np.insert(a, [1,2], -1, axis=0)
```

```
Out[426... array([[ 0,  1],
        [-1, -1],
        [ 2,  3],
        [-1, -1],
        [ 4,  5]])
```

```
In [425... np.insert(a, [1,2], [-1, -2], axis=0)
```

```
Out[425... array([[ 0,  1],
        [-1, -2],
        [ 2,  3],
        [-1, -2],
        [ 4,  5]])
```

```
In [428... np.insert(a, [1,2], [[-1, -2]], axis=0)
```

```
Out[428... array([[ 0,  1],
        [-1, -2],
        [ 2,  3],
        [-1, -2],
        [ 4,  5]])
```

append 类似于Python的append。

```
In [434... np.append(a, 1)
```

```
Out[434... array([0, 1, 2, 3, 4, 5, 1])
```

```
In [436... np.append(a, [1,2])
```

```
Out[436... array([0, 1, 2, 3, 4, 5, 1, 2])
```

```
In [445... a
```

```
Out[445... array([[0, 1],
          [2, 3],
          [4, 5]])
```

```
In [447... np.append(a, [[6,7]], axis=0)
```

```
Out[447... array([[0, 1],
          [2, 3],
          [4, 5],
          [6, 7]])
```

```
In [451... np.append(a, [[6],[7],[8]], axis=1)
```

```
Out[451... array([[0, 1, 6],
          [2, 3, 7],
          [4, 5, 8]])
```

`trim_zeros` 用于清理一维数组或序列的零值。

```
In [384... a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
np.trim_zeros(a)
```

```
Out[384... array([1, 2, 3, 0, 2, 1])
```

```
In [386... np.trim_zeros(a, "b")
```

```
Out[386... array([0, 0, 0, 1, 2, 3, 0, 2, 1])
```

```
In [387... np.trim_zeros(a, "f")
```

```
Out[387... array([1, 2, 3, 0, 2, 1, 0])
```

重排元素

`flip` 主要是翻转元素：

```
In [453... a = np.arange(8).reshape((2,2,2))
```

```
In [461... np.flip(a)
```

```
Out[461... array([[[7, 6],
          [5, 4]],

          [[3, 2],
          [1, 0]]])
```

```
In [462... np.flip(a, [0,1,2])
```

```
Out[462... array([[[7, 6],
          [5, 4]],

          [[3, 2],
          [1, 0]]])
```

```
In [455... np.flip(a, 0)
```

```
Out[455... array([[4, 5],
        [6, 7]],

        [[0, 1],
        [2, 3]]])
```

```
In [458... np.flip(a, 1)
```

```
Out[458... array([[2, 3],
        [0, 1]],

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

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

```
Out[460... array([[1, 0],
        [3, 2]],

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

```
In [464... np.flip(a, [0,1])
```

```
Out[464... array([[6, 7],
        [4, 5]],

        [[2, 3],
        [0, 1]]])
```

`roll` 和 `rot90` 用于旋转元素，不过两者并不一样。

`roll` 用于旋转给定轴的元素，参数包括：

- 数组
- `shift`: 整数或元组整数，旋转的数量，如果是tuple，要和对应的axis一样长
- `axis`: 坐标轴，整数或元组整数，默认会打平后shift，然后再reshape回去

```
In [469... a = np.arange(10)
```

```
In [470... np.roll(a, 3)
```

```
Out[470... array([7, 8, 9, 0, 1, 2, 3, 4, 5, 6])
```

```
In [471... np.roll(a, -2)
```

```
Out[471... array([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

```
In [475... np.roll(a, (1, 2))
```

```
Out[475... array([7, 8, 9, 0, 1, 2, 3, 4, 5, 6])
```

```
In [476... a = np.arange(10).reshape(2, 5)
a
```

```
Out[476... array([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])
```

```
In [479... np.roll(a, 1, axis=0)

Out[479... array([[5, 6, 7, 8, 9],
        [0, 1, 2, 3, 4]])
```

```
In [480... np.roll(a, 1, axis=1)

Out[480... array([[4, 0, 1, 2, 3],
        [9, 5, 6, 7, 8]])
```

```
In [509... np.roll(a, 1)

Out[509... array([[9, 0, 1, 2, 3],
        [4, 5, 6, 7, 8]])
```

```
In [511... # 等价于
np.roll(a.flatten(), 1).reshape((2, 5))

Out[511... array([[9, 0, 1, 2, 3],
        [4, 5, 6, 7, 8]])
```

```
In [490... np.roll(a, -1)

Out[490... array([[1, 2, 3, 4, 5],
        [6, 7, 8, 9, 0]])
```

```
In [508... # 两个tuple等长
np.roll(a, (1, 1), axis=(1, 0))

Out[508... array([[9, 5, 6, 7, 8],
        [4, 0, 1, 2, 3]])
```

```
In [503... # 上式等价于
np.roll(a, 1, axis=(1, 0))

Out[503... array([[9, 5, 6, 7, 8],
        [4, 0, 1, 2, 3]])
```

```
In [499... np.roll(a, (2, 1), axis=(1, 0))

Out[499... array([[8, 9, 5, 6, 7],
        [3, 4, 0, 1, 2]])
```

`rot90` 表示按指定轴旋转90°，方向是从第一个轴到第二个轴。参数包括：

- 数组：二维或更高维
- k：整数，旋转次数，默认1
- axes：两个以上元素的tuple，且元素必须不同，默认 (0,1)

```
In [514... a

Out[514... array([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])
```

```
In [515... np.rot90(a)
```

```
Out[515... array([[4, 9],
          [3, 8],
          [2, 7],
          [1, 6],
          [0, 5]])
```

```
In [517... # 180°
np.rot90(a, 2)
```

```
Out[517... array([[9, 8, 7, 6, 5],
          [4, 3, 2, 1, 0]])
```

```
In [519... # 转回来
np.rot90(a, 4)
```

```
Out[519... array([[0, 1, 2, 3, 4],
          [5, 6, 7, 8, 9]])
```

```
In [521... # 顺时针
np.rot90(a, 1, (1, 0))
```

```
Out[521... array([[5, 0],
          [6, 1],
          [7, 2],
          [8, 3],
          [9, 4]])
```

排序搜索

极值

关于最大最小值的方法不少，我们看看。

`np.maximum` 和 `np.minimum` 是通函数，以`max`为例，其他相关的还有 `np.max`（等价的 `np.amax`），`np.fmax` 和 `np.nanmax`。区分如下：

- `minimum`：两个数组逐元素比较
- `fmax`：同上，但忽略缺失值
- `amax`：沿着给定维度
- `nanmax`：同上，但忽略缺失值

我们以极大值为例。

```
In [15]: # 等价于 np.where(x1 >= x2, x1, x2)
np.fmax([np.nan, 2, 3], [1, 5, np.nan])
```

```
Out[15]: array([1., 5., 3.])
```

```
In [13]: np.nanmax([[np.nan, 3, 5], [2, 1, np.nan]], axis=0)
```

```
Out[13]: array([2., 3., 5.])
```

搜索

`argmax/argmin` 我们在《从小白到入门》中已经做了介绍，这里主要介绍带非数值 (NaN) 的版本，以极小值为例。

```
In [513... a = np.array([
    [np.nan, 2, 3],
    [1, np.nan, 4]
])
```

```
In [516... np.argmin(a, axis=0)
```

```
Out[516... array([0, 1, 0])
```

```
In [517... np.nanargmin(a, axis=0)
```

```
Out[517... array([1, 0, 0])
```

`argwhere` 和《从小白到入门》中介绍的 `where` 有所不同，它返回所有非0元素的 index，是 `where` 的弱化版本。

```
In [524... a = np.arange(6).reshape(2, 3)
a
```

```
Out[524... array([[0, 1, 2],
        [3, 4, 5]])
```

```
In [526... # 默认返回非0的元素 index
np.argwhere(a)
```

```
Out[526... array([[0, 1],
        [0, 2],
        [1, 0],
        [1, 1],
        [1, 2]])
```

```
In [529... # 这个不是指定条件，而是a>3本身是个数组
np.argwhere(a>3)
```

```
Out[529... array([[1, 1],
        [1, 2]])
```

对于非0位置的索引，还有两个API：

```
In [535... # 返回非0元素的索引
np.nonzero(a>2)
```

```
Out[535... (array([1, 1, 1]), array([0, 1, 2]))
```

```
In [537... # 返回打平后的索引
np.flatnonzero(a>2)
```

```
Out[537... array([3, 4, 5])
```

最后是 `searchsorted`，返回给定值应该插入的位置索引。给定的数组应为一维，但插入的值可以是数组。参数如下：

- 被插入的一维数组。

- 要插入的值，数组。
- side: 默认left, 第一个合适的位置; right为最后一个合适的位置。
- sorter: 一维数组, 可选的索引, 将要插入的数组排序为升序。

```
In [558... a = np.arange(1, 6)
b = np.array([3, 2, 1, 5, 4])
a
```

```
Out[558... array([1, 2, 3, 4, 5])
```

```
In [547... np.searchsorted(a, 2)
```

```
Out[547... 1
```

```
In [548... np.searchsorted(b, 2)
```

```
Out[548... 3
```

```
In [554... # 要插入的值可以是多维数组
np.searchsorted(a, [[2,2],[2,2]])
```

```
Out[554... array([[1, 1],
        [1, 1]])
```

side控制在所有合适位置中插入位置的index:

```
In [564... np.searchsorted([1,1,1,1,1], 1, "right")
```

```
Out[564... 5
```

```
In [565... np.searchsorted([1,1,1,1,1], 1, "left")
```

```
Out[565... 0
```

sorter表示被插入数组元素的索引, 该索引将数组按升序排列。

```
In [578... b
```

```
Out[578... array([3, 2, 1, 5, 4])
```

```
In [579... np.searchsorted(b, 2)
```

```
Out[579... 3
```

```
In [583... # 1 2 3 4 5
np.searchsorted(b, 2, sorter=[2,1,0,4,3])
```

```
Out[583... 1
```

```
In [587... np.searchsorted([1,2,3,4,5],2)
```

```
Out[587... 1
```

```
In [585... # 5 4 3 2 1
np.searchsorted(b, 2, sorter=[3,4,0,1,2])
```

Out[585... 0

```
In [586... np.searchsorted([5,4,3,2,1],2)
```

Out[586... 0

排序

`argsort` 在《从小白到入门》中已有介绍，此处不再赘述。这里介绍其他几个和排序相关的API：

- `sort/lexsort/ndarray.sort` : 数组排序
- `msort` : 排序第一轴
- `sort_complex` : 复数排序
- `partition/argpartition/ndarray.partition` : 部分排序

```
In [633... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 4))
a
```

Out[633... array([[0, 7, 6, 4],
 [4, 8, 0, 6],
 [2, 0, 5, 9]])

`sort` 包括以下参数：

- 数组
- `axis`: 默认最后一个轴 (-1)
- `kind`: `kind`是排序算法，支持：`quicksort`、`mergesort`、`heapsort`和`stable`，默认`quicksort`
- `order`: 用来排序的field（数组有field时）

沿着最后一个维度（默认）排序时不创建临时复制，因此速度最快，不占用额外空间。

`stable`会根据被排序的数据类型自动选择最稳定的排序算法。

```
In [634... # 默认axis=-1
np.sort(a)
```

Out[634... array([[0, 4, 6, 7],
 [0, 4, 6, 8],
 [0, 2, 5, 9]])

```
In [635... # 指定轴
np.sort(a, axis=0)
```

Out[635... array([[0, 0, 0, 4],
 [2, 7, 5, 6],
 [4, 8, 6, 9]])

```
In [636... # 指定排序算法
np.sort(a, kind="stable")
```

```
Out[636... array([[0, 4, 6, 7],
        [0, 4, 6, 8],
        [0, 2, 5, 9]])
```

```
In [637... # 指定order
# String, float16, int32
dtype = [("name", "U10"), ("height", "f2"), ("age", "i4")]
values = [
    ("Arthur", 1.8, 41),
    ("Lancelot", 1.9, 38),
    ("Galahad", 1.7, 38)
]
b = np.array(values, dtype=dtype)
np.sort(b, order=["height"])
```

```
Out[637... array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41), ('Lancelot', 1.9, 38)],
        dtype=[('name', '<U10'), ('height', '<f2'), ('age', '<i4')])
```

`lexsort` , 后面的key优先。

```
In [638... surnames = ('Zertz', 'Halilei', 'Halilei')
first_names = ('Heinrich', 'Gzlileo', 'Gustav')
# 先surname, 在按firstname
ind = np.lexsort((first_names, surnames))
ind
```

```
Out[638... array([2, 1, 0])
```

```
In [639... [surnames[i] + ", " + first_names[i] for i in ind]
```

```
Out[639... ['Halilei, Gustav', 'Halilei, Gzlileo', 'Zertz, Heinrich']
```

`ndarray.sort` 是in-place排序。其余参数与 `np.sort` 一样。

```
In [645... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 4))
a
```

```
Out[645... array([[0, 7, 6, 4],
        [4, 8, 0, 6],
        [2, 0, 5, 9]])
```

```
In [646... a.sort()
```

```
In [647... a
```

```
Out[647... array([[0, 4, 6, 7],
        [0, 4, 6, 8],
        [0, 2, 5, 9]])
```

`msort` 与 `np.sort(a, axis=0)` 等价。

```
In [650... rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 4))
a
```

```
Out[650...] array([[0, 7, 6, 4],
          [4, 8, 0, 6],
          [2, 0, 5, 9]])
```

```
In [651...] np.msort(a)
```

```
Out[651...] array([[0, 0, 0, 4],
          [2, 7, 5, 6],
          [4, 8, 6, 9]])
```

`sort_complex` 先用实部排，再用虚部。

```
In [657...] np.sort_complex([1 + 2j, 1+1j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
```

```
Out[657...] array([1.+1.j, 1.+2.j, 2.-1.j, 3.-3.j, 3.-2.j, 3.+5.j])
```

`partition` 和 `argpartition` 的关系和 `sort` 与 `argsort` 类似。参数包括：

- 数组
- `kth`: 切分的位置，整数或整数序列
- `axis`: 轴，默认-1
- `kind`: 选择算法，默认 `introselect`
- `order`: `str`或`List[str]`，和前面介绍的 `sort` 中的参数用法一样。

注意，返回的结果中，第`k`个元素的位置是排好序时的位置（不是原始数组中的`index`）。

```
In [723...] a = [100, 99, 87, 101, 88, 78, 98]
# 88 排好时在index=2的数字
# 比88小的在左边，比88大或相等的在右边
np.partition(a, 2)
```

```
Out[723...] array([ 78,  87,  88, 101,  99, 100,  98])
```

```
In [724...] np.argpartition(a, 2)
```

```
Out[724...] array([5, 2, 4, 3, 1, 0, 6])
```

```
In [725...] a = [3, 4, 2, 1]
# 4 是排好序时index=3的数字
np.partition(a, 3)
```

```
Out[725...] array([2, 1, 3, 4])
```

```
In [726...] np.argpartition(a, 3)
```

```
Out[726...] array([2, 3, 0, 1])
```

```
In [727...] rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 8))
a
```

```
Out[727...] array([[0, 7, 6, 4, 4, 8, 0, 6],
          [2, 0, 5, 9, 7, 7, 7, 7],
          [5, 1, 8, 4, 5, 3, 1, 9]])
```

```
In [728... # 注意第一行，比4大或相等的在右边
np.partition(a, 2)
```

```
Out[728... array([[0, 0, 4, 6, 4, 8, 7, 6],
        [0, 2, 5, 9, 7, 7, 7, 7],
        [1, 1, 3, 4, 5, 8, 5, 9]])
```

```
In [729... np.argpartition(a, 2)
```

```
Out[729... array([[0, 6, 3, 2, 4, 5, 1, 7],
        [1, 0, 2, 3, 4, 5, 6, 7],
        [1, 6, 5, 3, 4, 2, 0, 7]])
```

集合操作

包含

第一个数组的元素是否在第二个数组：

```
In [192... a = np.array([[1, 2], [2, 3]])
np.unique(a)
```

```
Out[192... array([1, 2, 3])
```

```
In [193... np.in1d([1,2,3,4], a)
```

```
Out[193... array([ True,  True,  True, False])
```

```
In [194... # 翻转
np.in1d([1,2,3,4], a, invert=True)
```

```
Out[194... array([False, False, False,  True])
```

```
In [195... # 两个数组里的值均unique
# 可以加速计算
np.in1d([1,2,3,4], a, assume_unique=True)
```

```
Out[195... array([ True,  True,  True, False])
```

```
In [200... # 打平
np.in1d([1,2],[3,4], a)
```

```
Out[200... array([ True,  True,  True, False])
```

```
In [201... # 不打平
np.isin([1,2,3,4], a)
```

```
Out[201... array([ True,  True,  True, False])
```

```
In [198... np.isin([1,2],[3,4], a)
```

```
Out[198... array([[ True,  True],
        [ True, False]])
```

其余参数和 `in1d` 一样。

交集

交集，输入的数组会打平：

```
In [210...] a
Out[210...] array([[1, 2],
                  [2, 3]])

In [211...] b = np.array([[2, 4, 1]])

In [212...] np.intersect1d(a, b)
Out[212...] array([1, 2])

In [213...] # 如果假设为unique, 其实不是, 结果会有误
np.intersect1d(a, b, assume_unique=True)
Out[213...] array([1, 2, 2])

In [214...] # 返回索引, 如果有多个, 返回第一个
np.intersect1d(a, b, return_indices=True)
Out[214...] (array([1, 2]), array([0, 1]), array([2, 0]))
```

并集

并集，输入的数组会打平：

```
In [215...] a
Out[215...] array([[1, 2],
                  [2, 3]])

In [218...] b
Out[218...] array([[2, 4, 1]])

In [216...] np.union1d(a, b)
Out[216...] array([1, 2, 3, 4])
```

差集

差集，返回在数组1不在数组2的：

```
In [219...] np.setdiff1d(a, b)
Out[219...] array([3])

In [220...] np.setdiff1d(b, a)
```

Out[220...] array([4])

异或集

异或集：

```
In [224...] np.setxor1d(a, b)
```

Out[224...] array([3, 4])

对于多个数组，可以使用 `reduce` 方法：

```
In [206...] from functools import reduce
```

```
In [208...] reduce(  
    np.union1d,  
    ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2])  
)
```

Out[208...] array([1, 2, 3, 4, 6])

函数式编程

NumPy的向量化计算也可以被自定义使用。下面我们就看看如何在NumPy上使用自定义方法。

`apply_along_axis` 是沿着某个维度应用一个函数，等于是沿着某个维度对原数组值进行映射。

```
In [230...] def func(a):  
    return (a[0] + a[-1]) / 2
```

```
In [232...] b = np.arange(1, 10).reshape(3, 3)  
b
```

Out[232...] array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])

```
In [233...] np.apply_along_axis(func, 0, b)
```

Out[233...] array([4., 5., 6.])

```
In [236...] np.apply_along_axis(func, 1, b)
```

Out[236...] array([2., 5., 8.])

```
In [242...] b = np.array([[8,1,7], [4,3,9], [5,2,6]])  
b
```

Out[242...] array([[8, 1, 7],
 [4, 3, 9],
 [5, 2, 6]])


```
In [240...] np.apply_along_axis(sorted, 1, b)
```

```
Out[240...] array([[1, 7, 8],  
        [3, 4, 9],  
        [2, 5, 6]])
```

```
In [245...] np.apply_along_axis(sorted, 0, b)
```

```
Out[245...] array([[4, 1, 6],  
        [5, 2, 7],  
        [8, 3, 9]])
```

```
In [253...] np.apply_along_axis(np.diag, 1, b)
```

```
Out[253...] array([[[8, 0, 0],  
        [0, 1, 0],  
        [0, 0, 7]],  
  
        [[4, 0, 0],  
        [0, 3, 0],  
        [0, 0, 9]],  
  
        [[5, 0, 0],  
        [0, 2, 0],  
        [0, 0, 6]]])
```

```
In [254...] np.apply_along_axis(np.diag, 0, b).T
```

```
Out[254...] array([[[8, 0, 0],  
        [0, 4, 0],  
        [0, 0, 5]],  
  
        [[1, 0, 0],  
        [0, 3, 0],  
        [0, 0, 2]],  
  
        [[7, 0, 0],  
        [0, 9, 0],  
        [0, 0, 6]]])
```

`apply_over_axes` 会将一个函数重复多次应用到多个轴。

注意：函数接受两个参数（和 `apply_along_axis` 的一个参数区别），分别是数组和维度。

```
In [268...] b
```

```
Out[268...] array([[8, 1, 7],  
        [4, 3, 9],  
        [5, 2, 6]])
```

```
In [266...] np.apply_over_axes(np.sum, b, 1)
```

```
Out[266...] array([[16],  
        [16],  
        [13]])
```

```
In [267...] np.apply_over_axes(np.sum, b, 0)
```

```
Out[267...] array([[17,  6, 22]])
```

```
In [265...] np.apply_over_axes(np.sum, b, [0, 1])
```

```
Out[265...] array([[45]])
```

`vectorize` 是通用类，将自定义函数向量化，主要是方便，而不是为了提高性能，内部实际是一个for循环。

```
In [270...] def func(a, b):  
    if a > b:  
        return a - b  
    else:  
        return a + b
```

```
In [271...] vfunc = np.vectorize(func)
```

```
In [272...] vfunc(np.arange(10), 3)
```

```
Out[272...] array([3, 4, 5, 6, 1, 2, 3, 4, 5, 6])
```

可以指定输出的类型：

```
In [274...] vfunc = np.vectorize(func, otypes=[np.float16])
```

```
In [275...] vfunc(np.arange(10), 3)
```

```
Out[275...] array([3., 4., 5., 6., 1., 2., 3., 4., 5., 6.], dtype=float16)
```

或者指定不进行向量化的参数：

```
In [290...] def func(a, b):  
    res = np.zeros_like(b)  
    for i in a:  
        res += i * b  
    return res
```

```
In [291...] func([1,2,3], np.array([2,4]))
```

```
Out[291...] array([12, 24])
```

```
In [305...] vfunc = np.vectorize(func, excluded=["a"])
```

```
In [306...] vfunc(a=[1,2,3], b=[2,4])
```

```
Out[306...] array([12, 24])
```

或事后指定：

```
In [328...] vfunc = np.vectorize(func)  
# 第一个参数排除  
vfunc.excluded.add(0)
```

```
In [329...] vfunc([1,2,3], [2,4])
```

```
Out[329...] array([12, 24])
```

如果输入的参数不能直接执行，则需要签名：

```
In [426...] conv = np.vectorize(np.convolve, signature="(n),(m)->(k)")
```

```
In [427...] conv([[1,2,3],[4,5,6],[7,8,9]], [1, 0.5])
```

```
Out[427...] array([[ 1. ,  2.5,  4. ,  1.5],  
          [ 4. ,  7. ,  8.5,  3. ],  
          [ 7. , 11.5, 13. ,  4.5]])
```

```
In [428...] np.convolve([1,2,3,4,5,6,7,8,9], [1, 0.5])
```

```
Out[428...] array([ 1. ,  2.5,  4. ,  5.5,  7. ,  8.5, 10. , 11.5, 13. ,  4.5])
```

相比 `vectorize`，`frompyfunc` 更加通用，它可以将任意的Python函数转为通函数。

```
In [433...] def func(x):  
            return str(x)
```

```
In [439...] ufunc = np.frompyfunc(func, 1, 1)
```

```
In [441...] ufunc(np.arange(5))
```

```
Out[441...] array(['0', '1', '2', '3', '4'], dtype=object)
```

`piecewise` 会执行一组条件和对应的函数，当条件为真时执行该函数。

```
In [447...] a = np.arange(10)
```

```
In [448...] np.piecewise(a, [a<5, a==5, a>5], [-1, 0, 1])
```

```
Out[448...] array([-1, -1, -1, -1, -1,  0,  1,  1,  1,  1])
```

```
In [453...] def func(a, b):  
            return a + b
```

```
In [463...] # 注意后面的参数是所有函数通用的  
np.piecewise(a, [a<5, a==5, a>5], [func, 0, lambda x, b: x**2, b=5])
```

```
Out[463...] array([ 5,  6,  7,  8,  9,  0, 36, 49, 64, 81])
```

测试

主要介绍与Asserts相关的API，与第三节中的《逻辑运算》有一定联系。

相等

相等的Asserts：

- `assert_equal`
- `assert_array_equal`

- `assert_string_equal`

三者API有点类似，前两个除了实际值和期望值，还多了：

- `err_msg`：失败时打印的消息
- `verbose`：为True时，冲突的值会append到消息后面

`assert_equal` 比较两个对象（标量，数组，元组，字典，NumPy数组等）。如果其中一个标量，另一个是数组，则会比较标量与数组的每一个元素。如果同样位置都是非数值（NaN），也算相等。

```
In [750... np.testing.assert_equal([1,2], [1,4], "msg: not equal")
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-750-bfa692ba2952> in <module>
----> 1 np.testing.assert_equal([1,2], [1,4], "msg: not equal")

[... skipping hidden 1 frame]

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
equal(actual, desired, err_msg, verbose)
    423         # Explicitly use __eq__ for comparison, gh-2552
    424         if not (desired == actual):
--> 425             raise AssertionError(msg)
    426
    427     except (DeprecationWarning, FutureWarning) as e:

AssertionError:
Items are not equal:
item=1
msg: not equal
ACTUAL: 2
DESIRED: 4
```

```
In [2]: # 标量与NumPy array_like 数组
np.testing.assert_equal(3, np.array([3, 3, 3]))
```

```
In [144... # 含有非数值（NaN）的情况
np.testing.assert_equal([1, np.nan], [1, np.nan])
```

`assert_array_equal` 的范围比 `assert_equal` 小，后者内部调用了前者。如果输入是 `array_like`，则两者无区别。后者在复数、时间、非数值（NaN）等方面有所不同。

```
In [4]: np.testing.assert_array_equal(np.nan, [np.nan])
```

```
In [9]: np.testing.assert_equal(np.nan, [np.nan])
```

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-9-ff23e2edb3c8> in <module>
----> 1 np.testing.assert_equal(np.nan, [np.nan])

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
equal(actual, desired, err_msg, verbose)
    375     # isscalar test to check cases such as [np.nan] != np.nan
    376     if isscalar(desired) != isscalar(actual):
--> 377         raise AssertionError(msg)
    378
    379     try:

AssertionError:
Items are not equal:
ACTUAL: nan
DESIRED: [nan]

```

`assert_string_equal` 用以比较字符串，输入两个字符串即可。

```
In [19]: np.testing.assert_string_equal("abc", "abc")
```

```
In [20]: np.testing.assert_string_equal("abc", "abC")
```

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-20-c25a53711d83> in <module>
----> 1 np.testing.assert_string_equal("abc", "abC")

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
string_equal(actual, desired)
    1204     msg = f"Differences in strings:\n{''.join(diff_list).rstrip()}"
    1205     if actual != desired:
-> 1206         raise AssertionError(msg)
    1207
    1208

AssertionError: Differences in strings:
- abc+ abC

```

相近

相近的Asserts:

- `assert_allclose` : 使用最多
- `assert_array_almost_equal_nulp`
- `assert_array_max_ulp`

`assert_allclose` 包含以下参数:

- 实际值
- 期望值
- `rtol=1e-07`
- `atol=0`
- `equal_nan=True`

- `err_msg=""`
- `verbose=True`

后两个参数和前面一样，不再赘述。`equal_nan` 为True表示非数值 (NaN) 会认为是一样的。`rtol`和`atol`用于评估精度，如果两者相差小于：`atol + rtol*|desired|` 就认为足够接近。

```
In [40]: np.testing.assert_allclose(1+1e-7, 1+1e-8)
```

```
In [41]: 1e-7-1e-8 < (1+1e-8)*1e-7
```

```
Out[41]: True
```

```
In [43]: np.testing.assert_allclose(5, 4, atol=1)
```

```
In [51]: np.testing.assert_allclose([5, np.nan], [4, np.nan],
                                     atol=1, equal_nan=True)
```

`assert_array_almost_equal_nulp` 可用于比较两个幅度可变队列相对稳健。

- `x`
- `y`
- `nulp=1`，最后一位公差的最大单位数，满足下式时：

$$|x-y| \leq nulp * spacing(\max(|x|, |y|))$$

算几乎相等。

```
In [79]: np.spacing(np.float64(1)) == np.finfo(np.float64).eps
```

```
Out[79]: True
```

```
In [88]: x = np.array([1, 0.1, 1e-10, 1e-20])
         x.dtype
```

```
Out[88]: dtype('float64')
```

```
In [89]: eps = np.finfo(x.dtype).eps
         eps
```

```
Out[89]: 2.220446049250313e-16
```

```
In [90]: x * eps <= 1 * np.spacing(x + x*eps)
```

```
Out[90]: array([ True, False, False, False])
```

```
In [91]: np.testing.assert_array_almost_equal_nulp(
         x, x + x*eps, nulp=1
         )
```

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-91-4807c4e32d47> in <module>
----> 1 np.testing.assert_array_almost_equal_nulp(
      2     x, x + x*eps, nulp=1
      3 )

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
array_almost_equal_nulp(x, y, nulp)
    1592         max_nulp = np.max(nulp_diff(x, y))
    1593         msg = "X and Y are not equal to %d ULP (max is %g)" % (nulp,
max_nulp)
-> 1594         raise AssertionError(msg)
    1595
    1596
AssertionError: X and Y are not equal to 1 ULP (max is 2)

```

```

In [93]: np.testing.assert_array_almost_equal_nulp(
        .1, .1+eps, nulp=1
        )

```

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-93-7d1275786d24> in <module>
----> 1 np.testing.assert_array_almost_equal_nulp(
      2     .1, .1+eps, nulp=1
      3 )

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
array_almost_equal_nulp(x, y, nulp)
    1592         max_nulp = np.max(nulp_diff(x, y))
    1593         msg = "X and Y are not equal to %d ULP (max is %g)" % (nulp,
max_nulp)
-> 1594         raise AssertionError(msg)
    1595
    1596
AssertionError: X and Y are not equal to 1 ULP (max is 16)

```

```

In [92]: np.testing.assert_array_almost_equal_nulp(
        1, 1+eps, nulp=1
        )

```

```

In [127... np.testing.assert_array_almost_equal_nulp(
        4, 8, nulp=1e16
        )

```

```

In [128... 1e16 * np.spacing(8)

```

```

Out[128... 17.763568394002505

```

`assert_array_max_ulp` 用于检查数组中所有元素最多相差N个单位。

```

In [142... np.testing.assert_array_max_ulp(1, 1)

```

```

Out[142... array([0.])

```

```
In [132... a = np.linspace(0., 1., 10)
np.testing.assert_array_max_ulp(a, np.arcsin(np.sin(a)))
```

```
Out[132... array([[0., 0., 0., 0., 0., 0., 0., 0., 1., 0.]])
```

```
In [138... np.testing.assert_array_max_ulp(1, 2, maxulp=1e16)
```

```
Out[138... array([4.50359963e+15])
```

```
In [140... # NaN不区分
np.testing.assert_array_max_ulp(np.nan, np.nan)
```

```
Out[140... array([0.])
```

小于

小于:

- `assert_array_less` : 参数与 `assert_array_equal` 一样, 用来比较是否小于, NaN也会比较, 如果同样位置都是NaN, 则跳过 (不抛出异常)。

```
In [147... np.testing.assert_array_less(
    [1, 1, np.nan],
    [2, 2, np.nan]
)
```

```
In [148... np.testing.assert_array_less(
    3, [1, 4]
)
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-148-4b4915dff584> in <module>
----> 1 np.testing.assert_array_less(
      2     3, [1, 4]
      3 )

[... skipping hidden 1 frame]

/usr/local/lib/python3.8/site-packages/numpy/testing/_private/utils.py in assert_
array_compare(comparison, x, y, err_msg, verbose, header, precision, equal_nan, e
qual_inf)
      842             verbose=verbose, header=header,
      843             names=('x', 'y'), precision=precision)
--> 844         raise AssertionError(msg)
      845     except ValueError:
      846         import traceback

AssertionError:
Arrays are not less-ordered

Mismatched elements: 1 / 2 (50%)
Max absolute difference: 2
Max relative difference: 2.
x: array(3)
y: array([1, 4])
```


异常

异常：

- `assert_raises`
- `assert_raises_regex`
- `assert_warns`

这三个API都可以作为上下文管理器使用。

`assert_raises` 有两种使用方法：

- `assert_raises(exception_class, callable, *args, **kwargs)`
- `assert_raises(exception_class)`

In [158... `1/0`

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-158-9e1622b385b6> in <module>  
----> 1 1/0  
  
ZeroDivisionError: division by zero
```

In [163... `with np.testing.assert_raises(ZeroDivisionError):`
`1/0`

In [162... `def div(a,b):a/b`
`np.testing.assert_raises(ZeroDivisionError, div, 1, 0)`

In [164... `with np.testing.assert_raises(ZeroDivisionError):`
`1/1`

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-164-3bd112b16131> in <module>
      1 with np.testing.assert_raises(ZeroDivisionError):
----> 2     1/1

/usr/local/Cellar/python@3.8/3.8.10/Frameworks/Python.framework/Versions/3.8/lib/
python3.8/unittest/case.py in __exit__(self, exc_type, exc_value, tb)
    225                                     self.obj_
name))
    226             else:
--> 227                 self._raiseFailure("{} not raised".format(exc_name))
    228         else:
    229             traceback.clear_frames(tb)

/usr/local/Cellar/python@3.8/3.8.10/Frameworks/Python.framework/Versions/3.8/lib/
python3.8/unittest/case.py in _raiseFailure(self, standardMsg)
    162     def _raiseFailure(self, standardMsg):
    163         msg = self.test_case._formatMessage(self.msg, standardMsg)
--> 164         raise self.test_case.failureException(msg)
    165
    166 class _AssertRaisesBaseContext(_BaseTestCaseContext):

AssertionError: ZeroDivisionError not raised

```

`assert_raises_regex` 相比前者可以指定正则。

```

In [174... with np.testing.assert_raises_regex(
            ZeroDivisionError, "division by zero"
        ):
            1/0

```

```

In [179... with np.testing.assert_raises_regex(
            ZeroDivisionError, "[a-z]+"
        ):
            1/0

```

`assert_warns` 和 `assert_raises` 一样的使用方式，只不过会警告而不是抛出异常。

```

In [182... import warnings

```

```

In [187... def func(n):
            warnings.warn("msg", UserWarning)
            return n * n

```

```

In [188... np.testing.assert_warns(UserWarning, func, 2)

```

```

Out[188... 4

```

```

In [189... func(2)

```

```

<ipython-input-187-23e97a00966d>:2: UserWarning: msg
  warnings.warn("msg", UserWarning)

```

```

Out[189... 4

```

关于警告的类型，可以参考文档：

- [warnings — Warning control — Python 3.10.4 documentation](#)

小结

参考

- [NumPy documentation — NumPy v1.23.dev0 Manual](#)

In []: