



Table of Contents

- 1 数学函数
 - 1.1 三角 / 双曲函数
 - 1.2 指数和对数
 - 1.3 算术操作
 - 1.4 自动域
- 2 数值计算
 - 2.1 舍入
 - 2.2 和积差
 - 2.3 符号函数
 - 2.4 截断
 - 2.5 插值
- 3 导数和微积分
 - 3.1 梯度
 - 3.2 梯形公式
- 4 多项式
 - 4.1 简介
 - 4.2 便捷类
- 5 关系运算
 - 5.1 真值测试
 - 5.2 值和类型
 - 5.3 逻辑运算
 - 5.4 比较
- 6 二进制运算
 - 6.1 位运算
 - 6.2 左右移
 - 6.3 打包解包
- 7 字符串
 - 7.1 基本操作
 - 7.2 比较
 - 7.3 基本信息
- 8 小结
- 9 参考

```
In [1]: import numpy as np
        np.__version__
```

```
Out[1]: '1.22.3'
```

文档阅读说明：

-  表示 Tip
-  表示注意事项

数学函数

NumPy内置了很多数学函数，包括：

- 三角/双曲函数
- 四舍五入
- 和、积、差
- 导数和微积分
- 指数和对数
- 算术操作
- 综合

关于这一部分，我们主要介绍一些特殊的方法，对比较简单的就跳过了，可以参考：

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

三角/双曲函数

三角函数和双曲中大部分都很容易理解，也都是通函数，我们主要介绍一个看起来不太明显的：`unwrap`，它的主要目的是对周期取大增量补码。参数如下：

- `p`: 数组
- `discont`: 数值间的最大中断，默认 `period/2`，低于该值的被设为该值
- `axis`: 轴，默认最后一个轴
- `period`: 周期范围，默认 `2pi`

```
In [42]: phase = np.linspace(0, np.pi, num=5)
phase[3:] += np.pi
phase
```

```
Out[42]: array([0.          , 0.78539816, 1.57079633, 5.49778714, 6.28318531])
```

```
In [43]: # 超过pi的，剪掉period
np.unwrap(phase)
```

```
Out[43]: array([ 0.          , 0.78539816, 1.57079633, -0.78539816,  0.          ])
```

```
In [67]: # 要减掉 1 个周期
np.unwrap([1, 5]), 5 - 2*np.pi
```

```
Out[67]: (array([ 1.          , -1.28318531]), -1.2831853071795862)
```

```
In [66]: # 要减掉3个周期
np.unwrap([1, 20]), 20 - 3*2*np.pi
```

```
Out[66]: (array([1.          , 1.15044408]), 1.1504440784612413)
```

```
In [72]: # 超过 pi 的处理掉！
np.unwrap([1, 1.1+np.pi]), 1.1+np.pi-2*np.pi
```

```
Out[72]: (array([ 1.          , -2.04159265]), -2.0415926535897935)
```

再看几个例子：

```
In [77]: # 超过4/2, 加4  
np.unwrap([0, 1, 2, -1, 0], period=4)
```

```
Out[77]: array([0, 1, 2, 3, 4])
```

```
In [83]: # 为什么要加而不是减, 因为只有加才能满足条件  
np.unwrap([1, -2, -1, 0], period=4)
```

```
Out[83]: array([1, 2, 3, 4])
```

```
In [85]: # 同上, 5 后面的数字都要加 4  
np.unwrap([2, 3, 4, 5, 2, 3, 4, 5], period=4)
```

```
Out[85]: array([2, 3, 4, 5, 6, 7, 8, 9])
```

另外需要注意的是, `deg2rad == radians`, `rad2deg == degrees`, 前面的表达更加清晰一些。

更多细节可查阅：

- <https://numpy.org/devdocs/reference/routines.math.html>

指数和对数

大部分的API都比较容易理解, 比如自然指数 `np.exp`, 2为底指数 `np.exp2` 等, 对应的log也有 `np.log`, `np.log2`, `np.log10` 等, 而且所有API都是通函数。

另外, 也有 `np.expm1` 表示exp后减1, 对应的就是加1后log的 `np.log1p` :

```
In [242...] np.log(np.exp(2)), np.log1p(np.expm1(2))
```

```
Out[242...] (2.0, 2.0)
```

还有两个求和的 `np.logaddexp` 和以2为底的 `np.logaddexp2`, 计算公式:

```
log(exp(x1) + exp(x2))
```

```
In [247...] np.logaddexp([1], [2]), np.log(np.exp(1) + np.exp(2))
```

```
Out[247...] (array([2.31326169]), 2.3132616875182226)
```

```
In [253...] np.logaddexp2([1], [2]), np.log2(np.exp2(1) + np.exp2(2))
```

```
Out[253...] (array([2.5849625]), 2.584962500721156)
```

`np.frexp` 和 `np.ldexp` 是一对操作, 后者等于 `x1 * 2**x2`, 而前者是将一个数组分解成mantissa和exponent, 而依据就是 `x = mantissa * 2**exponent`, 就是对应前面的x1和x2了。

```
In [269...] np.ldexp(2, np.arange(5)), 2 * 2**np.arange(5)
```

```
Out[269...] (array([ 2., 4., 8., 16., 32.], dtype=float16), array([ 2, 4, 8, 16, 32]))
```

```
In [283...] np.frexp(np.arange(2, 5))
```

```
Out[283...] (array([0.5 , 0.75, 0.5 ]), array([2, 2, 3], dtype=int32))
```

```
In [285...] np.array([0.5, 0.75, 0.5]) * 2 ** np.array([2, 2, 3]), np.arange(2, 5)
```

```
Out[285...] (array([2., 3., 4.]), array([2, 3, 4]))
```

算术操作

主要是中小学学过的加减乘除、乘方、开方、取余、倒数、绝对值，以及对应的一些特殊方法等，它们也都是通函数。按照惯例，我们主要介绍特殊的。

关于除法和Python的类似：

```
In [326...] # 地板除，等价于python的 //  
np.floor_divide(5, 2)
```

```
Out[326...] 2
```

```
In [327...] np.floor_divide([7, 8], [3, 5])
```

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

绝对值有相应的兼容复数的方法。

```
In [411...] np.abs(-1-1j)
```

```
Out[411...] 1.4142135623730951
```

```
In [412...] np.fabs(-1-1j)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-412-0b4518d62178> in <module>  
----> 1 np.fabs(-1-1j)  
  
TypeError: ufunc 'fabs' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule ''safe''
```

还有几个关于取余的方法：

```
In [341...] # 等价于Python的 x1%x2  
np.remainder(np.arange(3), 2)
```

```
Out[341...] array([0, 1, 0])
```

```
In [344...] # 和 np.reminder 一样  
np.mod([12, 13], [4, 5])
```

```
Out[344...] array([0, 3])
```

```
In [367... # mod 结果的符号是x2的符号
np.mod([-3, -2, -1, 1, 2, 3], 2)
```

```
Out[367... array([1, 0, 1, 1, 0, 1])
```

```
In [368... np.mod([2, 3], -2)
```

```
Out[368... array([ 0, -1])
```

```
In [369... # 而fmod 结果的符号是x1的符号
np.fmod([-3, -2, -1, 1, 2, 3], 2)
```

```
Out[369... array([-1,  0, -1,  1,  0,  1])
```

```
In [370... np.fmod([2, 3], -2)
```

```
Out[370... array([0, 1])
```

接下来这两个稍微不一样些。

```
In [358... # 按元素返回数组的小数部分和整数部分。
np.modf([0, 3.5, 2.0])
```

```
Out[358... (array([0. , 0.5, 0. ]), array([0., 3., 2.]))
```

```
In [359... np.modf(-1)
```

```
Out[359... (-0.0, -1.0)
```

```
In [371... # 同时返回(x // y, x % y)
np.divmod([12, 13, 15], 2)
```

```
Out[371... (array([6, 6, 7]), array([0, 1, 1]))
```

```
In [372... np.divmod([-3, -2, -1, 1, 2, 3], 2)
```

```
Out[372... (array([-2, -1, -1,  0,  1,  1]), array([1, 0, 1, 1, 0, 1]))
```

还有两个小学数学用过的：最大公倍数和最小公约数。

```
In [296... # 最小公倍数
np.lcm(12, 20)
```

```
Out[296... 60
```

```
In [298... # 多个值可以用reduce
np.lcm.reduce([2, 3, 5, 8])
```

```
Out[298... 120
```

```
In [302... # 同时求多个
np.lcm([2, 3, 5, 8], 3)
```

```
Out[302... array([ 6,  3, 15, 24])
```

```
In [304... # 最大公约数
np.gcd(12, 20)
```

```
Out[304... 4
```

```
In [306... np.gcd.reduce([15, 20, 30])
```

```
Out[306... 5
```

```
In [312... np.gcd([2, 3, 5, 8], 20)
```

```
Out[312... array([2, 1, 5, 4])
```

自动域

函数的输出数据类型与输入的某些域中的输入数据类型不同时，可以使用 `np.emath`。

支持以下API:

- `sqrt`, `power`
- `log`, `log2`, `log10`, `logn`
- `arccos`, `arcsin`, `arctan`

```
In [564... np.emath.sqrt(-1)
```

```
Out[564... 1j
```

```
In [565... np.sqrt(-1)
```

```
<ipython-input-565-597592b72a04>:1: RuntimeWarning: invalid value encountered in sqrt
  np.sqrt(-1)
```

```
Out[565... nan
```

```
In [573... import math
np.emath.log(-math.exp(1)) == 1+1j*math.pi
```

```
Out[573... True
```

```
In [575... np.power([2, 4], -2)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-575-f02dcfa4fde4> in <module>
----> 1 np.power([2, 4], -2)

ValueError: Integers to negative integer powers are not allowed.
```

```
In [576... np.emath.power([2, 4], -2)
```

```
Out[576... array([0.25 , 0.0625])
```

```
In [581... # 如果x包含负数则转为复数
np.emath.power([-2, 4], 1)
```

```
Out[581...] array([-2.+0.j,  4.+0.j])
```

数值计算

舍入

然后是四舍五入，`round` 和 `around` 一样。

```
In [105...] rng = np.random.default_rng(42)
arr = rng.random((2,3))
arr
```

```
Out[105...] array([[0.77395605, 0.43887844, 0.85859792],
 [0.69736803, 0.09417735, 0.97562235]])
```

```
In [106...] np.around(arr, 2)
```

```
Out[106...] array([[0.77, 0.44, 0.86],
 [0.7 , 0.09, 0.98]])
```

```
In [107...] np.array(arr).round(2)
```

```
Out[107...] array([[0.77, 0.44, 0.86],
 [0.7 , 0.09, 0.98]])
```

其他的接口都比较类似，除了 `fix` 的其他函数都是通函数。如下，不再赘述：

```
In [155...] lst = [2.1, -1.5, 3.2, 4.9]
```

```
In [156...] np.fix(lst)
```

```
Out[156...] array([ 2., -1.,  3.,  4.])
```

```
In [157...] np.trunc(lst)
```

```
Out[157...] array([ 2., -1.,  3.,  4.])
```

```
In [139...] np rint(lst)
```

```
Out[139...] array([ 2., -2.,  3.,  5.])
```

```
In [140...] np.floor(lst)
```

```
Out[140...] array([ 2., -2.,  3.,  4.])
```

```
In [141...] np.ceil(lst)
```

```
Out[141...] array([ 3., -1.,  4.,  5.])
```

和积差

这里包含了基础和积和累积和积，以及对应的有空值（`nan`）版本，API都比较简单，此处就不再赘述。主要介绍剩下的几个不太熟悉的。

首先是 `diff`，它包含以下参数：

- 数组
- 计算次数，就是计算几次diff
- 维度
- `prepend/append`：沿着维度放在原数组前面/后面然后再计算

```
In [210...] rng = np.random.default_rng(42)
a = rng.integers(0, 10, (3, 4))
a
```

```
Out[210...] array([[0, 7, 6, 4],
        [4, 8, 0, 6],
        [2, 0, 5, 9]])
```

```
In [211...] np.diff(a)
```

```
Out[211...] array([[ 7, -1, -2],
        [ 4, -8,  6],
        [-2,  5,  4]])
```

```
In [237...] # 注意，连续算两次
np.diff(a, 2)
```

```
Out[237...] array([[ -8,  -1],
        [-12, 14],
        [ 7,  -1]])
```

```
In [238...] np.diff(a, 2, append=[[0], [0], [0]])
```

```
Out[238...] array([[ -8,  -1,  -2],
        [-12, 14, -12],
        [ 7,  -1, -13]])
```

```
In [240...] # 等价于
pd = np.full((3, 1), 0)
cct = np.concatenate((a, pd), axis=1)
np.diff(cct, 2)
```

```
Out[240...] array([[ -8,  -1,  -2],
        [-12, 14, -12],
        [ 7,  -1, -13]])
```

```
In [241...] # prepend 同理
np.diff(a, 2, prepend=[[0], [0], [0]])
```

```
Out[241...] array([[ 7,  -8,  -1],
        [ 0, -12, 14],
        [-4,  7,  -1]])
```

值得一提的是也可以对时间进行处理：

```
In [252...] dts = np.arange("2022-01-02", "2022-01-05", dtype=np.datetime64)
np.diff(dts, 1)
```

```
Out[252...] array([1, 1], dtype='timedelta64[D]')
```

`np.ediff1d` 会先flatten再diff，所以返回的是一维。

符号函数

```
In [11]: # 小于0为-1, 等于0, 为0, 大于0为1  
np.sign([-5, 0, 5])
```

```
Out[11]: array([-1,  0,  1])
```

```
In [12]: # x1<0时为0, x1=0时为x2, x1>0时为1  
np.heaviside([-5, 0, 5], 0.5)
```

```
Out[12]: array([0. , 0.5, 1. ])
```

截断

```
In [36]: # 截断  
np.clip(np.arange(10).reshape(2,5), a_min=3, a_max=7)
```

```
Out[36]: array([[3, 3, 3, 3, 4],  
                [5, 6, 7, 7, 7]])
```

插值

`np.interp` 是一维线性插值方法, 支持弧度。

```
In [29]: def func(x):  
         return 2 * x + 3
```

```
In [30]: x = np.arange(1, 5)  
y = func(x)  
x, y
```

```
Out[30]: (array([1, 2, 3, 4]), array([ 5,  7,  9, 11]))
```

```
In [31]: np.interp([2.5, 8], x, y), func(2.5), func(8)
```

```
Out[31]: (array([ 8., 11.]), 8.0, 19)
```

```
In [32]: # 指定左右边界  
np.interp([0.5, 2, 5.5], x, y)
```

```
Out[32]: array([ 5.,  7., 11.])
```

```
In [33]: np.interp([0.5, 2, 5.5], x, y, left=1, right=60)
```

```
Out[33]: array([ 1.,  7., 60.])
```

导数和微积分

梯度

梯度是使用内部点中的二阶精确中心差和边界处的一阶精确单侧（向前或向后）差异计算的。

主要是利用泰勒二阶展开计算导数：

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \dots$$



等价于：

$$f(x_0 + h) \approx f(x_0) + f'(x_0)(h) + \frac{f''(x_0)}{2!}h^2 + O(h^3)$$

又有：

$$f(x_0 - h) \approx f(x_0) - f'(x_0)(h) + \frac{f''(x_0)}{2!}h^2 + O(h^3)$$

两式相减，得到：

$$f(x_0 + h) - f(x_0 - h) = 2 \cdot f'(x_0)(h)$$

即：

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + O(h^2)$$

接下来就是利用上式来计算梯度（导数）。

参数如下：

- f, 就是f(x), 数组
- varargs: f值的间隔，有多种可能的取值
- edge_order: 在边界时采用顺序1或2计算
- axis: 坐标轴

```
In [144... # 表示f(0)=1, f(1)=2 ...  
fx = np.array([1, 2, 4, 7, 11, 16])
```

```
In [145... # 默认 h=1, 第一个和最后一个（边界）有点特殊  
# f'(0) = (f(0+1) - f(0)) / 1 = (2-1)/1 = 1  
# f'(1) = (f(1+1) - f(1-1)) / 2 = (4-1)/2*1 = 1.5  
# f'(2) = (f(2+1) - f(2-1)) / 2 = (7-2)/2*1 = 2.5  
# ...  
# f'(5) = (f(5) - f(5-1)) / 1 = (16-11)/1 = 5  
np.gradient(fx)
```

```
Out[145... array([1. , 1.5, 2.5, 3.5, 4.5, 5. ])
```

```
In [146... # h 这个可以为其他数，比如0.5，此时表示  $f(0)=1, f(0.5)=2 \dots$ 
#  $f'(0.0) = (f(0+0.5) - f(0)) / 0.5 = (2-1)/0.5 = 2$ 
#  $f'(0.5) = (f(0.5+0.5) - f(0.5-0.5)) / 2*0.5 = (4-1)/1.0 = 3$ 
#  $f'(1.0) = (f(1.0+0.5) - f(1.0-0.5)) / 2*0.5 = (7-2)/1.0 = 5$ 
# ...
#  $f'(2.5) = (f(2.5) - f(2.5-0.5)) / 2*0.5 = (16-11)/0.5 = 10$ 
np.gradient(fx, 0.5)
```

```
Out[146... array([ 2.,  3.,  5.,  7.,  9., 10.])
```

输入数组，会按照行列分别计算后返回，也可以指定坐标轴。

```
In [155... rng = np.random.default_rng(42)
arr = rng.integers(1, 10, (2, 3))
arr
```

```
Out[155... array([[1, 7, 6],
        [4, 4, 8]])
```

```
In [162... np.gradient(arr)
```

```
Out[162... [array([[ 3., -3.,  2.],
        [ 3., -3.,  2.]]),
 array([[ 6. ,  2.5, -1. ],
        [ 0. ,  2. ,  4. ]])]
```

```
In [163... np.gradient(arr[:, 0]), np.gradient(arr[:, 1]), np.gradient(arr[:, 2]))
```

```
Out[163... (array([3., 3.]), array([-3., -3.]), array([2., 2.]))
```

```
In [164... np.gradient(arr[0,:]), np.gradient(arr[1,:])
```

```
Out[164... (array([ 6. ,  2.5, -1. ]), array([0., 2., 4.]))
```

```
In [165... # 指定坐标轴
np.gradient(arr, axis=0)
```

```
Out[165... array([[ 3., -3.,  2.],
        [ 3., -3.,  2.]])
```

第二个参数 `varargs` 控制 `fx` 值之间的间隔，但它有多种方式：

- 1. 单个标量，用于指定所有尺寸的样本距离
- 2. N 个标量，用于为每个维度指定恒定的采样距离，即 “dx”, “dy”, “dz”, ...
- 3. N 个数组，用于指定值沿 F 的每个维度的坐标，数组的长度必须与相应尺寸的大小匹配
- 4. N 个标量/数组的任意组合，含义为 2 和 3

上面的例子是最简单的「单个标量」的情况，接下来看N个标量的情况。

$$a = \frac{\frac{-dx_2}{dx_1}}{dx_1 + dx_2}$$

$$b = \frac{1}{dx_1} - \frac{1}{dx_2}$$

$$c = \frac{\frac{dx_1}{dx_2}}{dx_1 + dx_2}$$

$$a + b + c = 0$$

```
In [219... # N 个标量
x = np.array([0, 1, 1.5, 3.5, 4, 6])
fx = np.array([ 1, 2, 4, 7, 11, 16])
np.gradient(fx, x)
```

```
Out[219... array([1. , 3. , 3.5, 6.7, 6.9, 2.5])
```

```
In [220... ax_dx = np.diff(x)

dx1 = ax_dx[0:-1]
dx2 = ax_dx[1:]
a = -dx2 / (dx1 * (dx1 + dx2))
b = (dx2 - dx1) / (dx1 * dx2)
c = dx1 / (dx2 * (dx1 + dx2))

N = fx.ndim

slice1 = [slice(None)]*N
slice2 = [slice(None)]*N
slice3 = [slice(None)]*N
slice4 = [slice(None)]*N
axis = 0
slice1[axis] = slice(1, -1)
slice2[axis] = slice(None, -2)
slice3[axis] = slice(1, -1)
slice4[axis] = slice(2, None)
```

```
In [221... a, b, c
```

```
Out[221... (array([-0.33333333, -1.6       , -0.1       , -1.6       ]),
 array([-1. , 1.5, -1.5, 1.5]),
 array([1.33333333, 0.1       , 1.6       , 0.1       ]))
```

```
In [222... fx[tuple(slice2)], fx[tuple(slice3)], fx[tuple(slice4)]
```

```
Out[222... (array([1, 2, 4, 7]), array([ 2, 4, 7, 11]), array([ 4, 7, 11, 16]))
```

```
In [223... # out[1:-1] = a * f[:-2] + b * f[1:-1] + c * f[2:]
out = np.empty_like(fx, dtype=np.float16)
out[tuple(slice1)] = a * fx[tuple(slice2)] + b * fx[tuple(slice3)] + c * fx[tuple(slice4)]
out
```

```
Out[223... array([0. , 3. , 3.5, 6.7, 6.9, 0. ], dtype=float16)
```

多个数组或数组与标量的组合，此时会分别对应到各自轴上计算：

In [189...

```
arr
```

Out[189...

```
array([[1, 7, 6],  
       [4, 4, 8]])
```

In [188...

```
np.gradient(arr, [3,5], [1,2,3])
```

Out[188...

```
[array([[ 1.5, -1.5,  1. ],  
        [ 1.5, -1.5,  1. ]]),  
 array([[ 6. ,  2.5, -1. ],  
        [ 0. ,  2. ,  4. ]])]
```

In [194...

```
np.gradient(arr, [3,5], axis=0)
```

Out[194...

```
array([[ 1.5, -1.5,  1. ],  
       [ 1.5, -1.5,  1. ]])
```

In [196...

```
np.gradient(arr, [1,2,3], axis=1)
```

Out[196...

```
array([[ 6. ,  2.5, -1. ],  
       [ 0. ,  2. ,  4. ]])
```

还有一个 `edge_order` 参数需要说明一下，它主要控制在边界位置如何计算梯度。

In [216...

```
x = np.array([1, 2, 4, 7])  
y = x ** 2 + 2 * x + 1  
y
```

Out[216...

```
array([ 4,  9, 25, 64])
```

In [205...

```
np.gradient(y, x), 2*x + 2., np.gradient(y, x, edge_order=2)
```

Out[205...

```
(array([ 5.,  6., 10., 13.]),  
 array([ 4.,  6., 10., 16.]),  
 array([ 4.,  6., 10., 16.]))
```

In [218...

```
(9 - 4)/(2-1), (64-25)/(7-4)
```

Out[218...

```
(5.0, 13.0)
```

具体公式如下：

$$f'(x_l) = \frac{f(x_l + h) - f(x_l)}{h}$$
$$f'(x_r) = \frac{f(x_r) - f(x_r - h)}{h}$$

梯形公式

另一个API是梯形公式，可用来求积分，原理就是把被积函数切成很多个小的梯形。

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + 2f(x_4) + \cdots + 2f(x_{n-1}) + f(x_n))$$

参数如下：

- y, 就是f(x)
- x, 默认为None, 如果指定则根据x的元素计算dx
- dx, 默认1.0, 如果没有x, 则使用dx
- axis, 坐标轴

更多参考：

- [Trapezoidal rule - Wikipedia](#)

```
In [100... y = np.array([1, 2, 3])
x = np.array([4, 6, 8])
```

```
In [101... np.trapz(y), 1/2 * (1 + 2*2 + 3)
```

```
Out[101... (4.0, 4.0)
```

```
In [117... diff = np.diff(x)
np.trapz(y, x), 1/2*((1+2)*diff[0] + (2+3)*diff[1])
```

```
Out[117... (8.0, 8.0)
```

```
In [119... z = np.array([1, 2, 4])
diff = np.diff(z)
np.trapz(y, z), 1/2*((1+2)*diff[0] + (2+3)*diff[1])
```

```
Out[119... (6.5, 6.5)
```

```
In [138... y = np.array([[1, 2, 3], [4, 5, 6]])
y
```

```
Out[138... array([[1, 2, 3],
        [4, 5, 6]])
```

```
In [139... # 按行
np.trapz(y, axis=1)
```

```
Out[139... array([ 4., 10.])
```

```
In [140... # 列
np.trapz(y, axis=0)
```

```
Out[140... array([2.5, 3.5, 4.5])
```

多项式

多项式的在新的版本有专门的library: `polynomial`。其中包括：

- 幂级数
- 切比雪夫多项式
- 埃尔米特多项式 (物理学)
- 埃尔米特多项式 (概率学)
- 拉盖尔多项式
- 勒让德多项式

具体概念可参考：

- [幂级数 - 维基百科，自由的百科全书](#)
- [切比雪夫多项式 - 维基百科，自由的百科全书](#)
- [埃尔米特多项式 - 维基百科，自由的百科全书](#)
- [拉盖尔多项式 - 维基百科，自由的百科全书](#)
- [勒让德多项式 - 维基百科，自由的百科全书](#)

```
In [117... from numpy.polynomial import Polynomial as P
```

简介

```
In [987... # 幂序列
p = np.polynomial.Polynomial([3, 2, 1])
p
```

```
Out[987...  $x \mapsto 3.0 + 2.0x + 1.0x^2$ 
```

```
In [995... p(3)
```

```
Out[995... 18.0
```

```
In [988... rng = np.random.default_rng(42)
x = np.arange(10)
y = x + rng.standard_normal(10)
```

```
In [100... # 拟合
fitted = np.polynomial.Polynomial.fit(x, y, deg=1)
fitted
```

```
Out[100...  $x \mapsto 4.1644286915941295 + 4.216899419361024 (-1.0 + 0.2222222222222222x)$ 
```



```
In [100... fitted.convert()
```

```
Out[100...  $x \mapsto -0.05247072776689432 + 0.9370887598580052x$ 
```

```
In [116... # 根据根得到表达式
p = P.fromroots([1, 2])
p
```

```
Out[116...  $x \mapsto 2.0 - 3.0x + 1.0x^2$ 
```

```
In [116... p.convert()
```

Out[116... $x \mapsto 2.0 - 3.0x + 1.0x^2$

```
In [117... p.convert(domain=[0,1])
```

Out[117... $x \mapsto 0.75 - 1.0(-1.0 + 2.0x) + 0.25(-1.0 + 2.0x)^2$

类型之间转换也可以，不过不建议使用。当级数增加时会造成精度损失严重。

```
In [117... from numpy.polynomial import Chebyshev as T
```

```
In [118... T.cast(p)
```

Out[118... $x \mapsto 2.5T_0(x) - 3.0T_1(x) + 0.5T_2(x)$

```
In [100... c1 = (1,2,3)
c2 = (3,2,1)
sum = P.polyadd(c1,c2)
```

```
In [100... P.polyval(2, sum)
```

Out[100... 28.0

便捷类

NumPy提供了不同类型多项式的便捷使用方式，提供了统一的创建、操作、拟合接口。以下的介绍以幂级数为例，更多可访问文档。

- [Power Series \(numpy.polynomial.polynomial\) — NumPy v1.23.dev0 Manual](#)
- [Chebyshev Series \(numpy.polynomial.chebyshev\) — NumPy v1.23.dev0 Manual](#)
- [Hermite Series, "Physicists" \(numpy.polynomial.hermite\) — NumPy v1.23.dev0 Manual](#)
- [HermiteE Series, "Probabilists" \(numpy.polynomial.hermite_e\) — NumPy v1.23.dev0 Manual](#)
- [Laguerre Series \(numpy.polynomial.laguerre\) — NumPy v1.23.dev0 Manual](#)
- [Legendre Series \(numpy.polynomial.legendre\) — NumPy v1.23.dev0 Manual](#)

```
In [101... # 初始化一个实例
p = P([1, 2, 3])
p
```

Out[101... $x \mapsto 1.0 + 2.0x + 3.0x^2$

```
In [101... p.coef, p.domain, p.window
```

Out[101... (array([1., 2., 3.]), array([-1, 1]), array([-1, 1]))

```
In [108... # x_new = -1+x
p1 = P([1, 2, 3], domain=[0, 1], window=[-1, 0])
p1
```


Out[108...] $x \mapsto 1.0 + 2.0(-1.0 + x) + 3.0(-1.0 + x)^2$

```
In [108...] # x_new = -1+x + x
p2 = P([1, 2, 3], domain=[0, 1], window=[-1, 1])
p2
```

Out[108...] $x \mapsto 1.0 + 2.0(-1.0 + 2.0x) + 3.0(-1.0 + 2.0x)^2$

```
In [110...] p3 = P([1, 2, 3], domain=[2, 5], window=[-1, 1])
p3
```

Out[110...] $x \mapsto 1.0 + 2.0(-2.3333333333333335 + 0.6666666666666666x) + 3.0(-2.3333333333333335 + 0.6666666666666666x)^2$



```
In [110...] from numpy.polynomial import polyutils as pu
```

```
In [112...] # 映射 domain
pu.mapparms([2, 5], [-1, 1])
```

Out[112...] $(-2.3333333333333335, 0.6666666666666666)$

```
In [111...] (5*-1 - 2*1)/(5-2), (1--1)/(5-2)
```

Out[111...] $(-2.3333333333333335, 0.6666666666666666)$

```
In [101...] print(p)
```

$1.0 + 2.0 \cdot x^1 + 3.0 \cdot x^2$

可以选择不同的打印风格:

```
In [102...] np.polynomial.set_default_printstyle("ascii")
```

```
In [102...] print(p)
```

$1.0 + 2.0 x^{*1} + 3.0 x^{*2}$

```
In [102...] # 或
print(f"{p:unicode}")
```

$1.0 + 2.0 \cdot x^1 + 3.0 \cdot x^2$

多项式的基本运算:

```
In [102...] p + p
```

Out[102...] $x \mapsto 2.0 + 4.0x + 6.0x^2$

```
In [102...] p - p
```

Out[102...] $x \mapsto 0.0$

```
In [102...] p * p
```

Out[102...] $x \mapsto 1.0 + 4.0x + 10.0x^2 + 12.0x^3 + 9.0x^4$

In [102... `p ** 2`

Out[102... $x \mapsto 1.0 + 4.0x + 10.0x^2 + 12.0x^3 + 9.0x^4$

In [103... `p // P([-1, 1])`

Out[103... $x \mapsto 5.0 + 3.0x$

In [103... `p`

Out[103... $x \mapsto 1.0 + 2.0x + 3.0x^2$

In [103... `P([-1, 1]) * P([5, 3])`

Out[103... $x \mapsto -5.0 + 2.0x + 3.0x^2$

In [104... `# 可以整除（因式分解）
P([2, 3, 1]) == P([1, 1]) * P([2, 1])`

Out[104... `True`

In [104... `# 取余
p % P([-1, 1])`

Out[104... $x \mapsto 6.0$

In [105... `# 分解+余
divmod(p, P([-1, 1]))`

Out[105... `(Polynomial([5., 3.], domain=[-1., 1.], window=[-1., 1.]),
Polynomial([6.], domain=[-1., 1.], window=[-1., 1.]))`

In [105... `# 求值
x = np.arange(5)
p(x)`

Out[105... `array([1., 6., 17., 34., 57.])`

In [105... `3*x**2 + 2*x + 1.`

Out[105... `array([1., 6., 17., 34., 57.])`

In [105... `# 嵌套
p(p)`

Out[105... $x \mapsto 6.0 + 16.0x + 36.0x^2 + 36.0x^3 + 27.0x^4$

In [105... `# 根
p.roots()`

Out[105... `array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])`

In [106... `# 有有理根
P([2, -3, 1]).roots()`

Out[106...] `array([1., 2.])`

In [106...] `p`

Out[106...] $x \mapsto 1.0 + 2.0x + 3.0x^2$

In [106...] `p + [1, 2, 3]`

Out[106...] $x \mapsto 2.0 + 4.0x + 6.0x^2$

In [106...] `p + [1, 2]`

Out[106...] $x \mapsto 2.0 + 4.0x + 3.0x^2$

In [107...] `p / 2`

Out[107...] $x \mapsto 0.5 + 1.0x + 1.5x^2$

注意：上面的运算在不同domain、window或类型时无法使用。

In [107...] `# 不同domain
p + P([1], domain=[0, 1])`

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-1073-c9477ae81d90> in <module>  
      1 # 不同domain  
>>> 2 p + P([1], domain=[0, 1])  
  
/usr/local/lib/python3.8/site-packages/numpy/polynomial/_polybase.py in __add__(self, other)  
    498  
    499     def __add__(self, other):  
--> 500         othercoef = self._get_coefficients(other)  
    501         try:  
    502             coef = self._add(self.coef, othercoef)  
  
/usr/local/lib/python3.8/site-packages/numpy/polynomial/_polybase.py in _get_coefficients(self, other)  
    282             raise TypeError("Polynomial types differ")  
    283             elif not np.all(self.domain == other.domain):  
--> 284                 raise TypeError("Domains differ")  
    285             elif not np.all(self.window == other.window):  
    286                 raise TypeError("Windows differ")  
  
TypeError: Domains differ
```

计算积分：

In [112...] `p = P([3, 2, 1])
p`

Out[112...] $x \mapsto 3.0 + 2.0x + 1.0x^2$

In [112...] `# 定积分
p.integ()`

Out[112...] $x \mapsto 0.0 + 3.0x + 1.0x^2 + 0.3333333333333333x^3$

```
In [113...] # 指定积分次数
p.integ(m=2)
```

Out[113...] $x \mapsto 0.0 + 0.0x + 1.5x^2 + 0.3333333333333333x^3 + 0.08333333333333333x^4$

```
In [113...] # 指定下界（默认是0）为-1，常数项发生变化
p.integ(lbnd=-1)
```

Out[113...] $x \mapsto 2.333333333333333 + 3.0x + 1.0x^2 + 0.3333333333333333x^3$

```
In [113...] p.integ(k=[1], lbnd=-1)
```

Out[113...] $x \mapsto 3.333333333333333 + 3.0x + 1.0x^2 + 0.3333333333333333x^3$

计算导数：

```
In [115...] p.deriv()
```

Out[115...] $x \mapsto 2.0 + 2.0x$

```
In [115...] p.deriv(2)
```

Out[115...] $x \mapsto 2.0$

```
In [115...] p.deriv()(1)
```

Out[115...] 4.0

```
In [115...] p.deriv(2)(10)
```

Out[115...] 2.0

关系运算

NumPy中的关系运算一般用于判断数组是否满足指定条件，返回结果为布尔数组。这部分的API大多都是通函数。

真值测试

常用于判断元素是否全部满足或任一满足条件。

```
In [37]: a = np.array([
          [1, 0, 2],
          [2, 3, 0]
        ])
```

```
In [38]: np.all(a)
```

Out[38]: False

```
In [39]: np.all(a, axis=0)
```

```
Out[39]: array([ True, False, False])
```

```
In [40]: np.any(a)
```

```
Out[40]: True
```

```
In [41]: np.any(a, axis=0)
```

```
Out[41]: array([ True,  True,  True])
```

```
In [42]: np.any([0, 0, 0])
```

```
Out[42]: False
```

```
In [43]: np.alltrue(a)
```

```
Out[43]: False
```

```
In [44]: np.alltrue([1, 2, 3])
```

```
Out[44]: True
```

值和类型

判断数组的值是否符合条件：

- `isfinite`：不是无穷并且不是非数字
- `isnan`：非数字
- `isnat`：非时间
- `isinf/isneginf/isposinf`：正/负无穷

类型：

- `iscomplex`：复数
- `iscomplexobj`：复数类型
- `isfortran`：F-Style
- `isreal`：实数
- `isrealobj`：不是复数类型
- `isscalar`：标量

```
In [45]: np.isfinite([np.nan, 0, np.inf, 1])
```

```
Out[45]: array([False,  True, False,  True])
```

```
In [46]: np.isnan([np.nan, 2, np.inf])
```

```
Out[46]: array([ True, False, False])
```

```
In [47]: np.isnat([np.datetime64("2016-01-01")])
```

Out[47]: array([False])

```
In [48]: # 只支持时间格式
np.isnat([2])
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-48-b44857751f20> in <module>
      1 # 只支持时间格式
----> 2 np.isnat([2])

TypeError: ufunc 'isnat' is only defined for datetime and timedelta.
```

```
In [49]: np.isinf([np.nan, np.inf])
```

Out[49]: array([False, True])

```
In [50]: np.isneginf([np.inf, -np.inf, np.NINF])
```

Out[50]: array([False, True, True])

```
In [51]: a = np.array([[2, 3], [1+1j, 0]])
a
```

Out[51]: array([[2.+0.j, 3.+0.j],
 [1.+1.j, 0.+0.j]])

```
In [52]: np.iscomplex(a)
```

Out[52]: array([[False, False],
 [True, False]])

```
In [53]: np.iscomplexobj(a)
```

Out[53]: True

```
In [54]: np.isreal(a)
```

Out[54]: array([[True, True],
 [False, True]])

```
In [55]: np.isrealobj(a)
```

Out[55]: False

```
In [56]: np.isfortran(a)
```

Out[56]: False

```
In [57]: np.isscalar(np.array([2, 3]))
```

Out[57]: False

```
In [58]: np.isscalar(2)
```

Out[58]: True

```
In [59]: np.isscalar("fdf")
```

```
Out[59]: True
```

逻辑运算

包括与、或、异或、非，以与为例。

```
In [60]: np.logical_and(True, False)
```

```
Out[60]: False
```

```
In [61]: np.logical_and([2, 3], [4, False])
```

```
Out[61]: array([ True, False])
```

```
In [62]: a = np.arange(5)
np.logical_and(a>1, a<4)
```

```
Out[62]: array([False, False,  True,  True, False])
```

```
In [63]: # & 等价
np.array([1, 0]) & np.array([0, 1])
```

```
Out[63]: array([0, 0])
```

比较

相近判断：

`allclose/isclose` 用于判断所有值是否在阈值范围内：

$$|a - b| \leq (\text{atol} + \text{rtol} * |b|)$$

- `atol`默认值为`1e-08`
- `rtol`默认值为`1e-05`

常用于验证精度。

```
In [64]: np.allclose(1.0000089, 1.000009)
```

```
Out[64]: True
```

```
In [65]: np.allclose([1e10, 1e-9], [1.000001e10, 1e-8])
```

```
Out[65]: True
```

```
In [66]: np.allclose([1, np.nan], [1, np.nan])
```

```
Out[66]: False
```

```
In [67]: np.allclose([1, np.nan], [1, np.nan], equal_nan=True)
```

```
Out[67]: True
```

```
In [68]: np.isclose(1.0000089, 1.000009)
```

```
Out[68]: True
```

```
In [69]: np.isclose([1e10, 1e-9], [1.000001e10, 1e-8])
```

```
Out[69]: array([ True,  True])
```

```
In [70]: np.isclose([1, np.nan], [1, np.nan])
```

```
Out[70]: array([ True, False])
```

```
In [71]: np.isclose([1, np.nan], [1, np.nan], equal_nan=True)
```

```
Out[71]: array([ True,  True])
```

相等判断:

```
In [72]: a = np.array([[1, 2], [3, 4]])  
b = np.array([[1, 2], [3, 4]])  
c = np.array([[1, np.nan], [3, np.nan]])  
d = np.array([[1, np.nan], [3, np.nan]])
```

```
In [73]: np.array_equal(a, b)
```

```
Out[73]: True
```

```
In [74]: np.array_equal(a, c)
```

```
Out[74]: False
```

```
In [75]: np.array_equal(c, d)
```

```
Out[75]: False
```

```
In [76]: np.array_equal(c, d, equal_nan=True)
```

```
Out[76]: True
```

```
In [77]: a = np.array([1, 2])  
b = np.array([[1, 2], [1, 2]])  
c = np.array([[1, 2]])
```

```
In [78]: np.array_equal(a, b)
```

```
Out[78]: False
```

```
In [79]: # shape一致值相等  
np.array_equiv(a, b)
```

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



```
In [80]: np.array_equiv(a, c)
```

```
Out[80]: True
```

```
In [81]: np.array_equiv(b, c)
```

```
Out[81]: True
```

注意，这里的shape一致是指一个可以广播到另一个上。

最后是比较运算，包括：>, >=, <, <=, ==, !=。

```
In [82]: a = np.array([[1, 2], [4, 2]])  
b = np.array([[1, 3], [2, 2]])
```

```
In [83]: np.greater(a, b)
```

```
Out[83]: array([[False, False],  
               [ True, False]])
```

```
In [84]: np.greater_equal(a, b)
```

```
Out[84]: array([[ True, False],  
               [ True,  True]])
```

```
In [85]: np.less(a, b)
```

```
Out[85]: array([[False,  True],  
               [False, False]])
```

```
In [86]: np.less_equal(a, b)
```

```
Out[86]: array([[ True,  True],  
               [False,  True]])
```

```
In [87]: np.equal(a, b)
```

```
Out[87]: array([[ True, False],  
               [False,  True]])
```

```
In [88]: np.not_equal(a, b)
```

```
Out[88]: array([[False,  True],  
               [ True, False]])
```

二进制运算

主要是位运算相关API。

首先是逐位运算，均为通函数，包括：与、或、异或、非、左移、右移等。

位运算

```
In [487... int("000011", base=2), int("001100", base=2)
```

Out[487... (3, 12)

```
In [489... # 逐位与
np.bitwise_and(3, 12), 3 & 12
```

Out[489... (0, 0)

```
In [501... # 非
x = np.invert(np.array(13, dtype=np.uint8))
x, 2**7+2**6+2**5+2**4+2
```

Out[501... (242, 242)

```
In [502... np.binary_repr(x, width=8), int("00001101", base=2)
```

Out[502... ('11110010', 13)

左右移

或、异或和与类似，不再赘述。接下来的按位移动，以左移为例。

```
In [505... # 左移
```

```
In [508... 2 << 1
```

Out[508... 4

```
In [511... np.left_shift(2, 1)
```

Out[511... 4

```
In [512... np.left_shift(2, [1, 2, 3])
```

Out[512... array([4, 8, 16])

```
In [514... np.left_shift([1, 2, 3], 2)
```

Out[514... array([4, 8, 12])

打包解包

最后介绍下打包和解包，分别将二进制转为 uint8 数组或反之。

```
In [521... arr = np.array([[1, 1, 0], [1, 0, 1]])
```

```
In [546... # 打包
np.packbits(arr)
```

Out[546... array([212], dtype=uint8)

```
In [527... np.packbits(np.ravel(arr))
```

Out[527... array([212], dtype=uint8)

```
In [537... # 110101
int("11010100", base=2)
```

```
Out[537... 212
```

```
In [540... # 2**7+2**6 2**7 2**6
np.packbits(arr, axis=0)
```

```
Out[540... array([[192, 128, 64]], dtype=uint8)
```

```
In [552... # 解包, 注意, dtype 须为 uint8
b = np.array([2], dtype=np.uint8)
```

```
In [553... np.unpackbits(b)
```

```
Out[553... array([0, 0, 0, 0, 0, 0, 1, 0], dtype=uint8)
```

字符串

字符串在 NumPy 中也有很好的支持。所有的API在 `np.char` 下面, 针对以下两种数据类型:

```
In [611... np.str_, np.unicode_, np.str0
```

```
Out[611... (numpy.str_, numpy.str_, numpy.str_)
```

```
In [613... np.bytes_, np.string_
```

```
Out[613... (numpy.bytes_, numpy.bytes_)
```

基本操作

首先是一些常用的字符串操作, 与Python自带的类似。

```
In [614... a = np.array(["1", "2"], dtype=np.str_)
b = np.array(["a", "b"], dtype=np.str_)
```

加法就是字符的拼接:

```
In [624... # 加法
np.char.add(a, b)
```

```
Out[624... array(['1a', '2b'], dtype='<U2')]
```

```
In [617... np.array([1, 2], dtype=np.bytes_) + np.array([1, 2], dtype=np.bytes_)
```

```
-----
UFuncTypeError                                Traceback (most recent call last)
<ipython-input-617-49d28b8ba097> in <module>
----> 1 np.array([1, 2], dtype=np.bytes_) + np.array([1, 2], dtype=np.bytes_)

UFuncTypeError: ufunc 'add' did not contain a loop with signature matching types
(dtype('S1'), dtype('S1')) -> None
```

```
In [621... np.char.add(np.array([1, 2], dtype=np.bytes_), np.array([1, 2], dtype=np.bytes_))
```

```
Out[621... array([b'11', b'22'], dtype='<S2'))
```

```
In [622... np.char.add(np.array([1, 2], dtype=np.bytes_), np.array([1, 2], dtype=np.str_))
```

```
Out[622... array(['11', '22'], dtype='<U2')
```

乘法是字符的重复：

```
In [631... np.char.multiply(np.array([1], dtype=np.str_), 3)
```

```
Out[631... array(['111'], dtype='<U3')
```

```
In [632... # 次数小于0时为0  
np.char.multiply(np.array([1], dtype=np.str_), -3)
```

```
Out[632... array([''], dtype='<U1')
```

其他API也与str自带的类似：

- `capitalize`：首字母大写
- `title`：按标题大写
- `center`：给定长度居中填充
- `ljust/rjust`：给定长度左右填充
- `zfill`：0左填充
- `decode/encode`：解编码
- `expandtabs`：tab会被替换成一个或多个空格
- `join`：拼接
- `lower/upper`：大小写
- `swapcase`：大小写互换
- `lstrip/rstrip/strip`：strip
- `replace`：替换
- `translate`：转换
- `partition/rpartition`：分为三元组（左/右）
- `split/splitlines`：切分

```
In [666... # 首字母大写  
np.char.capitalize("ab b c")
```

```
Out[666... array('Ab b c', dtype='<U6')
```

```
In [667... # 标题大写  
np.char.title("ab b c")
```

```
Out[667... array('Ab B C', dtype='<U6')
```

```
In [687... # 给定长度居中  
(np.char.center("ab", 5, "~"),  
 np.char.ljust("a", 5, "~"),  
 np.char.rjust("a", 5, "~"),
```

```
np.char.zfill("a", 6)
)
```

```
Out[687...] (array('~ab~', dtype='<U5'),
array('a~~~~', dtype='<U5'),
array('~~~~a', dtype='<U5'),
array('00000a', dtype='<U6'))
```

```
In [644...] # 编解码
np.char.encode("abc", encoding="utf8")
```

```
Out[644...] array(b'abc', dtype='|S3')
```

```
In [651...] # 替换tab
val = np.char.expandtabs("\ta", tabsize=1)
val
```

```
Out[651...] array(' a', dtype='<U2')
```

```
In [658...] val.tolist(), val.tolist()[0] == " "
```

```
Out[658...] (' a', True)
```

```
In [704...] # 拼接
np.char.join("a", "12345")
```

```
Out[704...] array('1a2a3a4a5', dtype='<U9')
```

```
In [709...] # 大小写
np.char.lower(np.array(["A"], dtype=np.str_)), np.char.upper("a")
```

```
Out[709...] (array(['a'], dtype='<U1'), array('A', dtype='<U1'))
```

```
In [741...] # 互换
np.char.swapcase("aBc")
```

```
Out[741...] array('AbC', dtype='<U3')
```

```
In [715...] # strip
np.char.strip("abc "), np.char.strip("abc", "c")
```

```
Out[715...] (array('abc', dtype='<U4'), array('ab', dtype='<U3'))
```

```
In [723...] # replace
np.char.replace("aaabc", "a", "A", count=2)
```

```
Out[723...] array('AAabc', dtype='<U5')
```

```
In [824...] # 转换
np.char.translate(["abc", "a"], "1"*255, deletechars=None)
```

```
Out[824...] array(['111', '1'], dtype='<U3')
```

```
In [831...] # 非unicode时才会删除
np.char.translate(
    np.array(["abc", "a"], dtype=np.bytes_), b"1"*256, deletechars=b"a")
```

```
Out[831...] array([b'11', b''], dtype='<S3')
```

```
In [738...] # partition
(
    np.char.partition("abc", "b"),
    np.char.rpartition("abc", "b"),
    np.char.partition("abca", "a"),
    np.char.rpartition("abca", "a")
)
```

```
Out[738...] (array(['a', 'b', 'c'], dtype='<U1'),
array(['a', 'b', 'c'], dtype='<U1'),
array(['', 'a', 'bca'], dtype='<U3'),
array(['abc', 'a', ''], dtype='<U3'))
```

```
In [841...] # split
np.char.split("a b c", " "), np.char.splitlines("a\nb\nc")
```

```
Out[841...] (array(list(['a', 'b', 'c']), dtype=object),
array(list(['a', 'b', 'c']), dtype=object))
```

比较

主要比较字符串大小相等。

```
In [867...] np.char.equal(["abc", "ab"], ["abd", "ab"])
```

```
Out[867...] array([False,  True])
```

```
In [868...] np.char.not_equal(["abc", "ab"], ["abd", "ab"])
```

```
Out[868...] array([ True, False])
```

```
In [880...] # >=
np.char.greater_equal(["abc", "ab"], ["abd", "ab"]), "abc">"abd"
```

```
Out[880...] (array([False,  True]), False)
```

```
In [881...] np.char.greater(["abc", "ab"], ["abd", "ab"])
```

```
Out[881...] array([False, False])
```

```
In [882...] # <=
np.char.less_equal(["abc", "ab"], ["abd", "ab"])
```

```
Out[882...] array([ True,  True])
```

```
In [883...] np.char.less(["abc", "ab"], ["abd", "ab"])
```

```
Out[883...] array([ True, False])
```

```
In [890...] # 比较
# cmp可以取 < <= == >= > !=
np.char.compare_chararrays(
    ["abc", "ab", "a"],
    ["ab", "ad", "ae"],
```

```
    cmp="<",  
   .rstrip=True  
)
```

Out[890...] array([False, True, True])

基本信息

包括基本的判断和统计。

```
In [962...] np.char.count("abcab", "a", start=0, end=None)
```

Out[962...] array(2)

```
In [964...] np.char.str_len(["abcab", "a"])
```

Out[964...] array([5, 1])

```
In [959...] (  
    np.char.find("abcab", "a", start=0, end=None),  
    np.char.rfind("abcab", "a", start=0, end=None)  
)
```

Out[959...] (array(0), array(3))

```
In [906...] # 找不到返回 -1  
np.char.find("abcab", "d", start=2, end=None)
```

Out[906...] array(-1)

```
In [961...] (  
    np.char.index("abcab", "a", start=0),  
    np.char.rindex("abcab", "a", start=0)  
)
```

Out[961...] (array(0), array(3))

```
In [908...] # 找不到抛出异常  
np.char.index("abcab", "d", start=2)
```

```

-----
ValueError                                Traceback (most recent call last)
<ipython-input-908-6904110ea61f> in <module>
      1 # 找不到抛出异常
----> 2 np.char.index("abcab", "d", start=2)

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

/usr/local/lib/python3.8/site-packages/numpy/core/defchararray.py in index(a, su
b, start, end)
    742
    743     """
--> 744     return _vec_string(
    745         a, int_, 'index', [sub, start] + _clean_args(end))
    746

ValueError: substring not found

```

```

In [955... # starts/ends
(
    np.char.endswith(["a", "ba"], "a", start=0, end=1),
    np.char.startswith(["a", "ba"], "a", start=1, end=3)
)

```

```

Out[955... (array([ True, False]), array([False,  True]))

```

```

In [952... # 只有空格
np.char.isspace(["\t\n", "a"])

```

```

Out[952... array([ True, False])

```

```

In [951... # 所有字符小/大写，首字母大写
(
    np.char.islower(["a", "Ab"]),
    np.char.isupper(["a", "Ab", "AB"]),
    np.char.istitle(["Aa", "aB", "AB"])
)

```

```

Out[951... (array([ True, False]),
array([False, False,  True]),
array([ True, False, False]))

```

```

In [979... # 判断
lst = ["a", "1", "01", " 0 3 ", " 3.8.", "a1", "1.1", ""]
(
    # 每个元素的所有字符都为字母，至少一个字符
    np.char.isalpha(lst),
    # 同上，字母或数字
    np.char.isalnum(lst),
    "",
    # 只有decimal（小数点不算）
    np.char.isdecimal(lst),
    # 只有digit
    np.char.isdigit(lst),
    # 只有numeric
    np.char.isnumeric(lst)
)

```



```
Out[979... (array([ True, False, False, False, False, False, False, False]),
array([ True,  True,  True,  True,  True,  True, False, False]),
'',
array([False,  True,  True,  True, False, False, False, False]),
array([False,  True,  True,  True,  True, False, False, False]),
array([False,  True,  True,  True,  True, False, False, False]))
```

关于decimal、digit和numeric的区别可参考：

- [string - What's the difference between str.isdigit, isnumeric and isdecimal in python? - Stack Overflow](#)

它们的主要区别在处理unicode的方式上。

小结

参考

- [NumPy documentation — NumPy v1.23.dev0 Manual](#)
- [python - Memory growth with broadcast operations in NumPy - Stack Overflow](#)

In []: