

# ch13 Numpy-快速数据处理

标准的 Python 中用列表(list)保存一组值，可以用来当作数组使用。但由于列表的元素可以是任何对象，因此列表中保存的是对象的指针。这样的话，为了保存一个简单的列表，比如[1, 2, 3]，需要三个指针和三个整数对象。对于数值运算来说，这种结构显然比较浪费内存和 CPU 计算时间。

此外，Python 还提供了 array 模块，它所提供的 array 对象和列表不同，能直接保存数值，和 C 语言的一维数组类似。但是由于它不支持多维数组，也没有各种运算函数，因此也不适合做数值运算。

NumPy (Numerical Python)的诞生弥补了这些不足。NumPy 是 Python 语言的一个扩展程序库，支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。NumPy 是一个运行速度非常快的数学库，主要用于数组计算，包含：

- 一个强大的 N 维数组对象 ndarray (n-dimensional array object)。
- 广播功能函数 ufunc (universal function object)，是一种能够对数组进行处理的特殊函数。
- 整合 C/C++/Fortran 代码的工具。
- 线性代数、傅里叶变换、随机数生成等功能。

## 一、关于 Numpy 应用

NumPy 通常与 SciPy (Scientific Python) 和 Matplotlib (绘图库) 一起使用，这种组合广泛用于替代 MatLab，是一个强大的科学计算环境，有助于我们通过 Python 学习数据科学或者机器学习。

- SciPy 是一个开源的 Python 算法库和数学工具包。
- SciPy 包含的模块有最优化、线性代数、积分、插值、特殊函数、快速傅里叶变换、信号处理和图像处理、常微分方程求解和其他科学与工程中常用的计算。
- Matplotlib 是 Python 编程语言及其数值数学扩展包 NumPy 的可视化操作界面。它为利用通用的图形用户界面工具包，如 Tkinter, wxPython, Qt 或 GTK+ 向应用程序嵌入式绘图提供了应用程序接口 (API)。

## 二、安装 Numpy

### 1. 使用已有的发行版本

对于许多用户，尤其是在 Windows 上，最简单的方法是下载以下的 Python 发行版，它们包含了所有的关键包（包括 NumPy, SciPy, matplotlib, IPython, SymPy 以及 Python 核心自带

的其它包)：

- Anaconda: 免费 Python 发行版, 用于进行大规模数据处理、预测分析, 和科学计算, 致力于简化包的管理和部署。支持 Linux, Windows 和 Mac 系统。
- Enthought Canopy: 提供了免费和商业发行版。支持 Linux, Windows 和 Mac 系统。
- Python(x,y): 免费的 Python 发行版, 包含了完整的 Python 语言开发包 及 Spyder IDE。支持 Windows, 仅限 Python 2 版本。
- WinPython: 另一个免费的 Python 发行版, 包含科学计算包与 Spyder IDE。支持 Windows。
- Pyzo: 基于 Anaconda 的免费发行版本及 IEP 的交互开发环境, 超轻量级。支持 Linux, Windows 和 Mac 系统。

## 2. 使用 pip 安装

安装 NumPy 最简单的方法就是使用 pip 工具:

```
python -m pip install numpy scipy matplotlib pandas sympy
```

## 3. 安装验证

测试是否安装成功:

```
>>> import numpy as np
>>> np.eye(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

## 三、Ndarray 对象

Python 官网上的发行版是不包含 NumPy 模块的。NumPy 最重要的一个特点是其 N 维数组对象 ndarray, 它是一系列同类型数据的集合, 以 0 下标为开始进行集合中元素的索引。

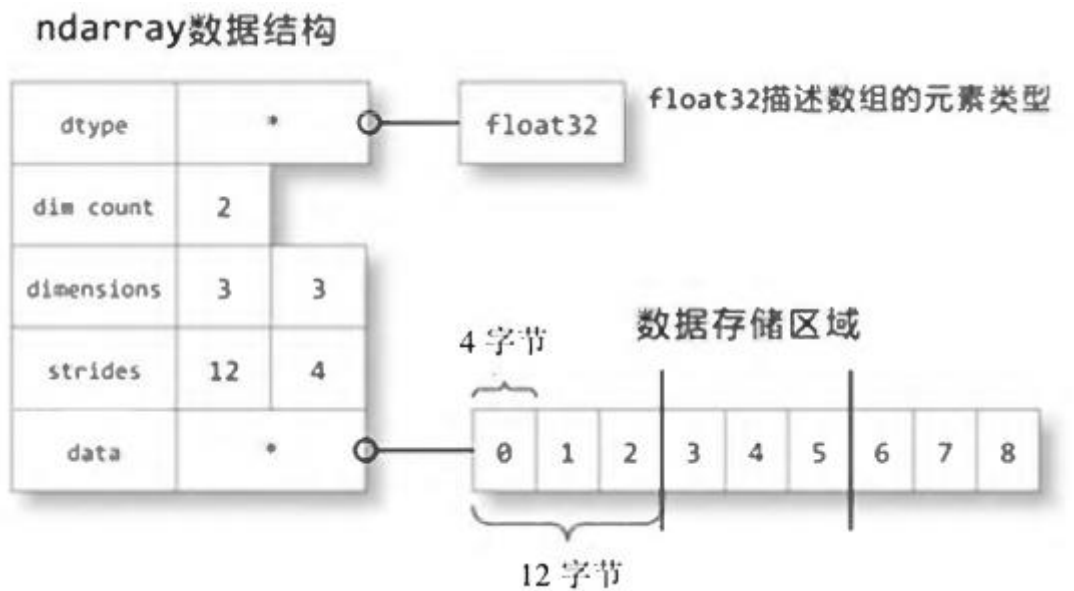
ndarray 对象是用于存放同类型元素的多维数组。

ndarray 中的每个元素在内存中都有相同存储大小的区域。

### 1. ndarray 数据结构

ndarray 内部由以下内容组成:

- 一个指向数据（内存或内存映射文件中的一块数据）的指针。
- 数据类型或 `dtype`，描述在数组中的固定大小值的格子。
- 一个表示数组形状（`shape`）的元组，表示各维度大小的元组。
- 一个跨度元组（`stride`），其中的整数指的是为了前进到当前维度下一个元素需要"跨过"的字节数。



3x3 的 ndarray 数组对象在内存中的存储方式

2. ndarray 构造器

创建一个 ndarray 只需调用 NumPy 的 `array` 函数即可：

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

参数说明：

名称	描述
object	数组或嵌套的数列（嵌套的序列将创建多维数组）
dtype	数组元素的数据类型，可选
copy	对象是否需要复制，可选
order	创建数组的样式，C 为行方向，F 为列方向，A 为任意方向（默认）
subok	默认返回一个与基类类型一致的数组
ndmin	指定生成数组的最小维度

实例 1：一维数组

```
import numpy as np
a = np.array([1,2,3]) # 传入序列的也可以是元组 (1,2,3)
print(a)
```

输出结果如下：

```
[1, 2, 3]
```

### 实例 2：二维数组即矩阵

```
# 多于一个维度
import numpy as np
a = np.array([[1, 2], [3, 4]])
print(a)
```

输出结果如下：

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

### 实例 3：指定数组最小维度

```
# 最小维度
import numpy as np
a = np.array([1, 2, 3,4,5], ndmin = 2)
print(a)
```

输出如下：

```
[[1, 2, 3, 4, 5]]
```

### 实例 4：指定元素数据类型

```
# dtype 参数
import numpy as np
a = np.array([1, 2, 3], dtype = complex)
print(a)
```

输出结果如下：

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

ndarray 对象由计算机内存的连续一维部分组成，并结合索引模式，将每个元素映射到内存块中的一个位置。内存块以行顺序(C 样式)或列顺序(FORTRAN 或 MatLab 风格，即前述的 F 样式)来保存元素。

## 3. Numpy 元素的数据类型

numpy 支持的数据类型比 Python 内置的类型要多很多，基本上可以和 C 语言的数据类型对

应上，其中部分类型对应为 Python 内置的类型。下表列举了常用 NumPy 基本类型。

名称	描述
bool_	布尔型数据类型（True 或者 False）
int_	默认的整数类型（类似于 C 语言中的 long，int32 或 int64）
intc	与 C 的 int 类型一样，一般是 int32 或 int 64
intp	用于索引的整数类型（类似于 C 的 ssize_t，一般情况下仍然是 int32 或 int64）
int8	字节（-128 to 127）
int16	整数（-32768 to 32767）
int32	整数（-2147483648 to 2147483647）
int64	整数（-9223372036854775808 to 9223372036854775807）
uint8	无符号整数（0 to 255）
uint16	无符号整数（0 to 65535）
uint32	无符号整数（0 to 4294967295）
uint64	无符号整数（0 to 18446744073709551615）
float_	float64 类型的简写
float16	半精度浮点数，包括：1 个符号位，5 个指数位，10 个尾数位
float32	单精度浮点数，包括：1 个符号位，8 个指数位，23 个尾数位
float64	双精度浮点数，包括：1 个符号位，11 个指数位，52 个尾数位
complex_	complex128 类型的简写，即 128 位复数
complex64	复数，表示双 32 位浮点数（实数部分和虚数部分）
complex128	复数，表示双 64 位浮点数（实数部分和虚数部分）

numpy 的数值类型实际上是 dtype 对象的实例，并对应唯一的字符，包括 np.bool\_，np.int32，np.float32，等等。

数据类型对象 (dtype)

数据类型对象是用来描述与数组对应的内存区域如何使用，这依赖如下几个方面：

- 数据的类型（整数，浮点数或者 Python 对象）
- 数据的大小（例如， 整数使用多少个字节存储）
- 数据的字节顺序（小端法或大端法）
- 在结构化类型的情况下，字段的名称、每个字段的数据类型和每个字段所取的内存块的部分
- 如果数据类型是子数组，它的形状和数据类型

字节顺序是通过对数据类型预先设定"<"或">"来决定的。"<"意味着小端法(最小值存储在最小的地址，即低位组放在最前面)。">"意味着大端法(最重要的字节存储在最小的地址，即高位组放在最前面)。

dtype 对象是使用以下语法构造的：

```
numpy.dtype(object, align, copy)
```

- object - 要转换为的数据类型对象
- align - 如果为 true，填充字段使其类似 C 的结构体。
- copy - 复制 dtype 对象，如果为 false，则是对内置数据类型对象的引用

#### 实例 1

```
import numpy as np
# 使用标量类型
dt = np.dtype(np.int32)
print(dt)
```

输出结果为：

```
int32
```

#### 实例 2

```
import numpy as np
# int8, int16, int32, int64 四种数据类型可以使用字符串 'i1', 'i2', 'i4', 'i8' 代替
dt = np.dtype('i4')
print(dt)
```

输出结果为：

```
int32
```

#### 实例 3

```
import numpy as np
# 字节顺序标注
dt = np.dtype('<i4')
print(dt)
```

输出结果为：

```
int32
```

下面实例展示结构化数据类型的使用，类型字段和对应的实际类型将被创建。

#### 实例 4

```
# 首先创建结构化数据类型
import numpy as np
dt = np.dtype([('age', np.int8)])
```

```
print(dt)
```

输出结果为:

```
[('age', 'i1')]
```

### 实例 5

```
# 将数据类型应用于 ndarray 对象
import numpy as np
dt = np.dtype([('age', np.int8)])
a = np.array([(10,), (20,), (30,)], dtype = dt)
print(a)
```

输出结果为:

```
[(10,) (20,) (30,)]
```

### 实例 6

```
# 类型字段名可以用于存取实际的 age 列
import numpy as np
dt = np.dtype([('age', np.int8)])
a = np.array([(10,), (20,), (30,)], dtype = dt)
print(a['age'])
```

输出结果为:

```
[10 20 30]
```

下面的示例定义一个结构化数据类型 `student`，包含字符串字段 `name`，整数字段 `age`，及浮点字段 `marks`，并将这个 `dtype` 应用到 `ndarray` 对象。

### 实例 7

```
import numpy as np
student = np.dtype([('name', 'S20'), ('age', 'i1'), ('marks', 'f4')])
print(student)
a = np.array([('Tom', 21, 50.), ('Mark', 18, 75)], dtype = student)
print(a)
print(a["name"])
```

输出结果为:

```
[(b'Tom', 21, 50.) (b'Mark', 18, 75.)]
[b'Tom' b'Mark']
```

每个内建类型都有一个唯一定义它的字符代码，如下:

字符	对应类型
b	布尔型

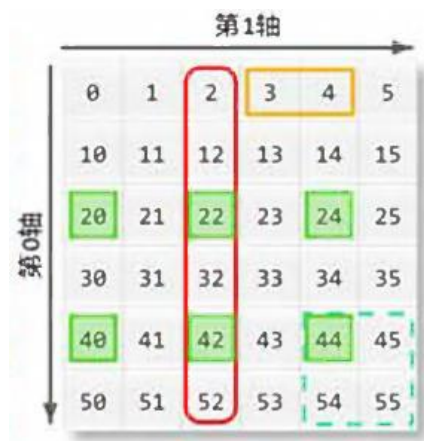
字符	对应类型
i	(有符号) 整型
u	无符号整型 integer
f	浮点型
c	复数浮点型
m	timedelta（时间间隔）
M	datetime（日期时间）
O	(Python) 对象
S, a	(byte-)字符串
U	Unicode
V	原始数据 (void)

## 4. NumPy 数组属性

NumPy 数组的维数称为秩（rank），秩就是轴的数量，即数组的维度，一维数组的秩为 1，二维数组的秩为 2，以此类推。

在 NumPy 中，每一个线性的数组称为是一个轴（axis），也就是维度（dimensions）。比如说，二维数组相当于是两个一维数组，其中第一个一维数组中每个元素又是一个一维数组。所以一维数组就是 NumPy 中的轴（axis），第一个轴相当于是底层数组，第二个轴是底层数组里的数组。而轴的数量——秩，就是数组的维数。

很多时候可以声明 axis。axis=0，表示沿着第 0 轴进行操作，即对每一列进行操作；axis=1，表示沿着第 1 轴进行操作，即对每一行进行操作。



NumPy 的数组中比较重要 ndarray 对象属性有：



属性	说明
<code>ndarray.ndim</code>	秩，即轴的数量或维度的数量
<code>ndarray.shape</code>	数组的维度，对于矩阵，n 行 m 列
<code>ndarray.size</code>	数组元素的总个数，相当于 <code>.shape</code> 中 <code>n*m</code> 的值
<code>ndarray.dtype</code>	<code>ndarray</code> 对象的元素类型
<code>ndarray.itemsize</code>	<code>ndarray</code> 对象中每个元素的大小，以字节为单位
<code>ndarray.flags</code>	<code>ndarray</code> 对象的内存信息
<code>ndarray.real</code>	<code>ndarray</code> 元素的实部
<code>ndarray.imag</code>	<code>ndarray</code> 元素的虚部
<code>ndarray.data</code>	包含实际数组元素的缓冲区，由于一般通过数组的索引获取元素，所以通常不需要使用这个属性。

#### (1) `ndarray.ndim`

`ndarray.ndim` 用于返回数组的维数，等于秩。

#### 实例

```
import numpy as np
a = np.arange(24)
print(a.ndim)          # a 现只有一个维度
b = a.reshape(2,4,3)  # 现在调整其大小
print(b.ndim)          # b 现在拥有三个维度
```

输出结果为：

```
1
3
```

#### (2) `ndarray.shape`

`ndarray.shape` 表示数组的维度，返回一个元组，这个元组的长度就是维度的数目，即 `ndim` 属性(秩)。比如，一个二维数组，其维度表示"行数"和"列数"。

`ndarray.shape` 也可以用于调整数组大小。

#### 实例

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a.shape)
```

输出结果为：

```
(2, 3)
```

调整数组大小。

## 实例

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
a.shape = (3,2)
print(a)
```

输出结果为:

```
[[1 2]
 [3 4]
 [5 6]]
```

当设置某个轴的元素个数为 -1 时，将自动计算此轴的长度。例如 `a.shape = (-1, 2)`，由于数组 `a` 有 6 个元素，因此 `a` 的 `shape` 属性改成了 `(3, 2)`。

NumPy 也提供了 `reshape` 函数来调整数组大小，但是返回一个新数组。

## 实例

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2) # 注意，这样不会改变 a 的 shape
print(b)
```

输出结果为:

```
[[1, 2]
 [3, 4]
 [5, 6]]
```

`b = a.reshape((-1,))` ?

`a.resize(3,2)` 等效于 `a.shape = (3, 2)`

### (3) ndarray.itemsize

`ndarray.itemsize` 以字节的形式返回数组中每一个元素的大小。

例如，一个元素类型为 `float64` 的数组 `itemsiz` 属性值为 8(`float64` 占用 64 个 bits，每个字节长度为 8，所以 `64/8`，占用 8 个字节)，又如，一个元素类型为 `complex32` 的数组 `item` 属性为 4 (`32/8`)。

## 实例

```
import numpy as np
# 数组的 dtype 为 int8 (一个字节)
x = np.array([1,2,3,4,5], dtype = np.int8)
print(x.itemsize)
# 数组的 dtype 现在为 float64 (八个字节)
y = np.array([1,2,3,4,5], dtype = np.float64)
print(y.itemsize)
```

输出结果为:

```
1
8
```

## 5. 从数值范围创建数组

numpy 的 `arange()`、`linspace()`、`logspace()`等方法用于从数值范围创建一维数组（`ndarray` 对象）。

### （1）`numpy.arange()`方法

`numpy.arange()`方法创建数值范围并返回 `ndarray` 对象，函数格式如下：

```
numpy.arange(start, stop, step, dtype)
```

根据 `start` 与 `stop` 指定的范围以及 `step` 设定的步长，生成一个 `ndarray`。

参数	描述
<code>start</code>	起始值，默认为 0
<code>stop</code>	终止值（不包含）
<code>step</code>	步长，默认为 1
<code>dtype</code>	返回 <code>ndarray</code> 的数据类型，如果没有提供，则会使用输入数据的类型。

注意与 `python` 内置函数 `range` 的区别：

`range(start, stop[, step])` -> range object

生成表示整数序列的 `range` 对象，各参数均为整数。

实例：生成 0 到 5 的数组

```
import numpy as np
x = np.arange(5)
print(x)
```

输出结果如下：

```
[0 1 2 3 4]
```

实例：设置返回类型为 `float`

```
import numpy as np
x = np.arange(5, dtype = float) # 设置了 dtype
print(x)
```

输出结果如下：

```
[0. 1. 2. 3. 4.]
```

实例：设置了起始值、终止值及步长

```
import numpy as np
x = np.arange(10, 20, 2)
print(x)
```

输出结果如下：

```
[10 12 14 16 18]
```

## (2) numpy.linspace

numpy.linspace 函数用于创建一个一维数组，数组是一个等差数列构成的，格式如下：

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

参数	描述
start	序列的起始值
stop	序列的终止值，如果 endpoint 为 true，该值包含于数列中
num	要生成的等步长的样本数量，默认为 50
endpoint	该值为 true 时，数列中包含 stop 值，反之不包含，默认是 True。
retstep	如果为 True 时，生成的数组中会显示间距，反之不显示。
dtype	ndarray 的数据类型

实例：设置起始点为 1，终止点为 10，数列个数为 10

```
import numpy as np
a = np.linspace(1,10,10)
print(a)
```

输出结果为：

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

实例：设置元素全部是 1 的等差数列

```
import numpy as np
a = np.linspace(1,1,10)
print(a)
```

输出结果为：

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

实例：将 endpoint 设为 false，不包含终止值

```
import numpy as np
a = np.linspace(10, 20, 5, endpoint = False)
print(a)
```

输出结果为：

```
[10. 12. 14. 16. 18.]
```

如果将 endpoint 设为 true，则会包含 20。

以下实例设置间距。

实例：显示设置间距

```
import numpy as np
a = np.linspace(1,10,10,retstep= True)
print(a)
# 拓展例子
b = np.linspace(1,10,10).reshape([10,1])
print(b)
```

输出结果为:

```
(array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]), 1.0)
[[ 1.]
 [ 2.]
 [ 3.]
 [ 4.]
 [ 5.]
 [ 6.]
 [ 7.]
 [ 8.]
 [ 9.]
[10.]]
```

### (3) numpy.logspace

numpy.logspace 函数用于创建一个等比数列。格式如下:

```
np.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
```

base 参数意思是取对数的时候 log 的下标。

参数	描述
start	序列的起始值为: $\text{base}^{\text{start}}$
stop	序列的终止值为: $\text{base}^{\text{stop}}$ 。如果 endpoint 为 true, 该值包含于数列中
base	对数 log 的底数。

该函数先将 start~stop 区间等距 (假定为 dx) 划分为  $x_0=\text{start}$ ,  $x_1=\text{start}+\text{dx}$ ,  $x_2=\text{start}+2\text{dx}$ , ..., 然后以  $\text{base}=y$  为底, 分别以  $\text{start}$ ,  $\text{start}+\text{dx}$ ,  $\text{start}+2\text{dx}$ , ... 为幂生成等比数列:

$$y^{x_0}, y^{x_1}, y^{x_2}, \dots$$

数列的公比为  $y^{\text{dx}}$ 。显然, 对该数列取以  $y$  为底的对数后, 得到的是等差数列  $x_0, x_1, x_2, \dots$

实例

```
import numpy as np
# 默认底数是 10
a = np.logspace(1.0, 2.0, num = 10)
print(a)
```

输出结果为(公比为  $10^{(2-1)/9} = 10^{1/9}$ ):

```
[ 10.          12.91549665   16.68100537   21.5443469   27.82559402
```

```
35.93813664 46.41588834 59.94842503 77.42636827 100. ]
```

实例：将对数的底数设置为 2

```
import numpy as np
a = np.logspace(0,9,10,base=2)
print(a)
```

输出如下(公比为  $2^{(9-0)/9} = 2$ ):

```
[ 1.  2.  4.  8. 16. 32. 64. 128. 256. 512.]
```

也就是生成了公比为 2 的等比数列。

## 6. 使用 `numpy.asarray()` 方法从已有的数组创建数组

`numpy.asarray` 类似 `numpy.array`，但 `numpy.asarray` 参数只有三个。

```
numpy.asarray(a, dtype = None, order = None)
```

参数	描述
a	任意形式的输入参数，可以是，列表，列表的元组，元组，元组的元组，元组的列表，多维数组
dtype	数据类型，可选
order	可选，有"C"和"F"两个选项,分别代表，行优先和列优先，在计算机内存中的存储元素的顺序。

实例：将列表转换为 `ndarray`

```
import numpy as np
x = [1,2,3]
a = np.asarray(x)
print(a)
```

输出结果为：

```
[1 2 3]
```

实例：将元组转换为 `ndarray`

```
import numpy as np
x = (1,2,3)
a = np.asarray(x)
print(a)
```

输出结果为：

```
[1 2 3]
```

实例：将元组列表转换为 `ndarray`

```
import numpy as np
x = [(1,2,3),(4,5)] # 高版本不再支持 !!!!!!!!!!!
a = np.asarray(x)
```

```
print(a)
print(a.shape)
print(a[0])
```

输出结果为:

```
[(1, 2, 3) (4, 5)]
(2, )
(1, 2, 3)
```

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

**E:\Python38\lib\site-packages\numpy\core\\_asarray.py:102: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.**

实例：将嵌套列表转换为矩阵（二维 ndarray）

```
import numpy as np
x = [[1,2,3],[4,5,6]]
a = np.asarray(x)
print(a)
print(a.shape)
print(a[1,2])
```

输出结果为:

```
[[1, 2, 3]
 [4, 5, 6]]
(2, 3)
6
```

实例：设置了 dtype 参数

```
import numpy as np
x = [1,2,3]
a = np.asarray(x, dtype = float)
print(a)
```

输出结果为:

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

## 7. 创建特殊数组

(1) `numpy.empty()`方法创建空数组

`numpy.empty` 方法用来创建一个指定形状 (`shape`)、数据类型 (`dtype`) 且未初始化的数组:

```
numpy.empty(shape, dtype = float, order = 'C')
```

参数	描述
shape	数组形状
dtype	数据类型, 可选
order	有"C"和"F"两个选项, 分别代表, 行优先和列优先, 在计算机内存中的存储元素的顺序。

实例: 创建空数组

```
import numpy as np
x = np.empty([3,2], dtype = int)
print(x)
```

输出结果为:

```
[[ 6917529027641081856  5764616291768666155]
 [ 6917529027641081859 -5764598754299804209]
 [          4497473538          844429428932120]]
```

注意 - 数组元素为随机值, 因为它们未初始化。

(2) `numpy.zeros()`方法创建全 0 数组

创建指定大小的数组, 数组元素以 0 来填充:

```
numpy.zeros(shape, dtype = float, order = 'C')
```

实例

```
import numpy as np
x = np.zeros(5) # 默认为浮点数
print(x)
y = np.zeros((5,), dtype = np.int) # 设置类型为整数
print(y)
z = np.zeros((2,3), dtype = [('x', 'i4'), ('y', 'f')]) # 自定义类型
print(z)
print(z['y'])
```

输出结果为:

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



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

### (3) numpy.ones()方法创建全 1 数组

创建指定形状的数组，数组元素以 1 来填充：

```
numpy.ones(shape, dtype = None, order = 'C')
```

#### 实例

```
import numpy as np
x = np.ones(5) # 默认为浮点数
print(x)
x = np.ones([2,2], dtype = int) # 自定义类型
print(x)
```

输出结果为：

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

### (4) numpy.eye()方法创建单位矩阵

创建指定对角元为 1，其他元素为 0 的二维数组即单位矩阵：

```
numpy.eye(N, M = None, k= 0, dtype = float, order = 'C')
```

参数	描述
N	数组形状
M	可选，默认矩阵为方阵
k	指定设置元素为 1 的对角线，k=0 表示主对角，k 为正表示上三角的第 k 条对角，k 为负表示下三角的第 k 条对角。

#### 实例

```
import numpy as np
A = np.eye(4)
print(A)
B = np.eyes(4, k = 1) # 上三角的第一条对角线全为 1，其他元素为 0
print(B)
```

输出结果为：

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

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

(5) `numpy.diag()`方法创建对角矩阵或取矩阵对角元素

Extract a diagonal or construct a diagonal array。

```
numpy.diag(v, k= 0)
```

参数	描述
v	列表、元组或矩阵
k	指定对角线位置。

实例

```
import numpy as np
A = np.diag([1, 2, 3]) # 创建三角矩阵
print(A)
a = np.diag(A) # 取对角元素
print(a)
b = np.diag([[1, 2, 3] , [4, 5, 6] , [7, 8, 9]]) # 取对角元素
print(b)
```

输出结果为:

```
[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]]
[1, 2, 3]
[1, 5, 9]
```

## 8. 数组切片和索引

`ndarray` 对象的内容可以通过索引或切片来访问和修改，与 `Python` 中 `list` 的切片操作一样。

`ndarray` 数组可以基于 `0 - n` 的下标进行索引，切片对象可以通过内置的 `slice` 函数，并设置 `start`, `stop` 及 `step` 参数进行，从原数组中切割出一个新数组。

实例

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2) # 从索引 2 开始到索引 7 停止，间隔为 2
print(a[s])
```

输出结果为:

```
[2 4 6]
```

以上实例中，首先通过 `arange()` 函数创建 `ndarray` 对象。然后，分别设置起始，终止和步长的参数为 2, 7 和 2。

也可以通过冒号分隔切片参数 `start:stop:step` 来进行切片操作。

#### 实例

```
import numpy as np
a = np.arange(10)
b = a[2:7:2] # 从索引 2 开始到索引 7 停止，间隔为 2
print(b)
```

输出结果为：

```
[2 4 6]
```

冒号 `:` 的解释：如果只放置一个参数，如 `[2]`，将返回与该索引相对应的单个元素。如果为 `[2:]`，表示从该索引开始以后的所有项都将被提取。如果使用了两个参数，如 `[2:7]`，那么则提取两个索引(不包括停止索引)之间的项。

#### 实例

```
import numpy as np
a = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
b = a[5]
print(b)
print(a[2:])
print(a[2:5])
```

输出结果为：

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

可以使用和列表相同的方式对数组的元素进行存取：

```
a = np.arange(10)
a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- `a[5]`: 用整数作为下标可以获取数组中的某个元素。
- `a[3:5]`: 用切片作为下标获取数组的一部分，包括 `a[3]`但不包括 `a[5]`。
- `a[:5]`: 切片中省略开始下标，表示从 `a[0]`开始。
- `a[:-1]`: 下标可以使用负数，表示从数组最后往前数。

<code>a[5]</code>	<code>a[3:5]</code>	<code>a[:5]</code>	<code>a[:-1]</code>
5	[3, 4]	[0, 1, 2, 3, 4]	[0, 1, 2, 3, 4, 5, 6, 7, 8]

- `a[1:-1:2]`: 切片中的第三个参数表示步长，2表示隔一个元素取一个元素。
- `a[::-1]`: 省略切片的开始下标和结束下标，步长为-1，整个数组头尾颠倒。
- `a[5:1:-2]`: 步长为负数时，开始下标必须大于结束下标。

<code>a[1:-1:2]</code>	<code>a[::-1]</code>	<code>a[5:1:-2]</code>
[1, 3, 5, 7]	[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]	[5, 3]

下标还可以用来修改元素的值：

```
a[2:4] = 100, 101
a
array([ 0,  1, 100, 101,  4,  5,  6,  7,  8,  9])
```

**注意**，和列表不同的是，通过切片获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据存储空间。下面的程序将 `b` 的第 2 个元素修改为-10, `a` 的第 5 个元素也同时被修改为-10，因为它们在内存中的地址相同。

```
b = a[3:7] # 通过切片产生一个新的数组 b, b 和 a 共享同一块数据存储空间
b[2] = -10 # 将 b 的第 2 个元素修改为-10
```

<code>b</code>	<code>a</code>
[101, 4, -10, 6]	[0, 1, 100, 101, 4, -10, 6, 7, 8, 9]

多维数组同样适用上述索引提取方法。

#### 实例

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

```
print(a)
print('从数组索引 a[1:] 处开始切割')
print(a[1:]) # 从某个索引处开始切割
```

输出结果为:

```
[[1 2 3]
 [3 4 5]
 [4 5 6]]
从数组索引 a[1:] 处开始切割
[[3 4 5]
 [4 5 6]]
```

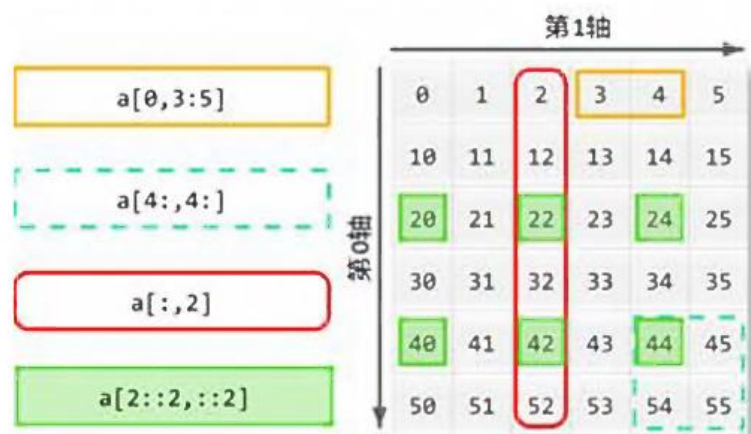
切片还可以包括省略号 **...或冒号:**，来使选择元组的长度与数组的维度相同。如果在行位置使用省略号，它将返回包含行中元素的 ndarray。

实例

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a[...,1]) # 第2列元素
print(a[1,:]) # 第2行元素
print(a[...,1:]) # 第2列及剩下的所有元素
```

输出结果为:

```
[2 4 5]
[3 4 5]
[[2 3]
 [4 5]
 [5 6]]
```



使用数组切片语法访问多维数组中的元素

## 9. 高级索引

除了使用切片下标存取元素之外，NumPy 还提供了整数列表、整数数组和布尔数组等几种高级

下标存取方法。当使用整数列表对数组元素进行存取时，将使用列表中的每个元素作为下标。使用列表作为下标得到的数组不和原始数组共享数据。

### (1) 整数数组索引

```
x = np.arange(10, 1, -1)
x
array([10, 9, 8, 7, 6, 5, 4, 3, 2])
```

- `x[[3,3,1,8]]`: 获取 `x` 中的下标为 3、3、1、8 的 4 个元素，组成一个新的数组。
- `x[[3,3,-3,8]]`: 下标可以是负数，-3 表示取倒数第 3 个元素(从 1 开始计数)。

```
a = x[[3, 3, 1, 8]]
b = x[[3, 3, -3, 8]]

      a              b
-----
[7, 7, 9, 2] [7, 7, 4, 2]
```

下面修改 `b[2]` 的值，但是由于它和 `x` 不共享内存，因此 `x` 的值不变：

```
b[2] = 100

      b              x
-----
[ 7,  7, 100,  2] [10, 9, 8, 7, 6, 5, 4, 3, 2]
```

整数序列下标也可以用来修改元素的值：

```
x[[3,5,1]] = -1, -2, -3
x
array([10, -3, 8, -1, 6, -2, 4, 3, 2])
```

实例：获取二维数组中(0,0)，(1,1)和(2,0)位置处的元素

```
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
y = A[[0,1,2], [0,1,0]]
print(y)
```

输出结果为：

```
[[1  2]
 [3  4]
 [5  6]]
[1  4  5]
```

## （2）布尔索引

可以通过一个布尔数组来索引目标数组。

布尔索引通过布尔运算（如：比较运算符）来获取符合指定条件的元素的数组。

实例：获取大于 5 的元素

```
import numpy as np
x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])
print('我们的数组是： ', x)
print('\n')
print('大于 5 的元素是： ')
print(x[x > 5])
```

输出结果为：

我们的数组是：

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

大于 5 的元素是：

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

实例：使用了 ~（取补运算符）来过滤 NaN

```
import numpy as np
a = np.array([np.nan, 1, 2, np.nan, 3, 4, 5])
print(a[~np.isnan(a)])
```

输出结果为：

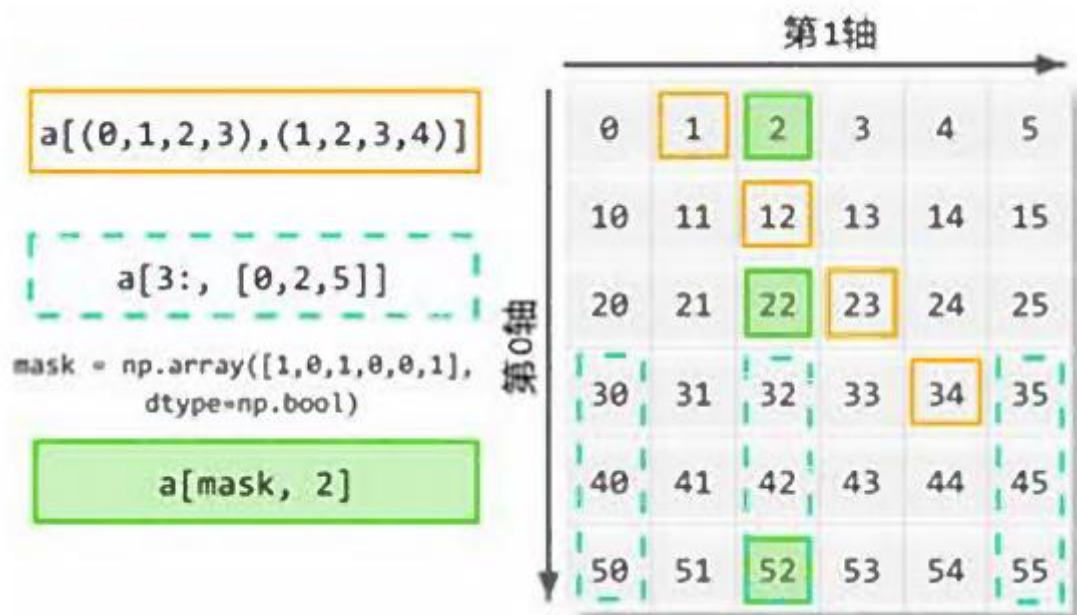
```
[ 1.  2.  3.  4.  5.]
```

实例：从数组中过滤掉非复数元素

```
import numpy as np
a = np.array([1, 2+6j, 5, 3.5+5j])
print(a[np.iscomplex(a)])
```

输出如下：

```
[2.0+6.j 3.5+5.j]
```



使用整数序列和布尔数组访问多维数组中的元素

### (3) 花式索引

花式索引指的是利用整数数组进行索引。

花式索引根据索引数组的值作为目标数组的某个轴的下标来取值。对于使用一维整型数组作为索引，如果目标是一维数组，那么索引的结果就是对应位置的元素；如果目标是二维数组，那么就是对应下标的行。

花式索引跟切片不一样，它总是将数据复制到新数组中。

实例：传入顺序索引数组

```
import numpy as np
x=np.arange(32).reshape((8,4))
print(x)
print(x[[4,2,1,7]])
```

输出结果为：

```
[[ 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]]
[[16 17 18 19]
 [ 8  9 10 11]
 [ 4  5  6  7]
 [28 29 30 31]]
```



## 2、传入倒序索引数组

实例：

```
import numpy as np
x=np.arange(32).reshape((8,4))
print(x)
print(x[[-4,-2,-1,-7]])
```

输出结果为：

```
[[ 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]]
[[16 17 18 19]
 [24 25 26 27]
 [28 29 30 31]
 [ 4  5  6  7]]
```

## 10. 广播(Broadcast)

广播(Broadcast)是 numpy 对不同形状(shape)的数组进行数值计算的方式，对数组的算术运算通常在相应的元素上进行。

如果两个数组 **a** 和 **b** 形状相同，即满足 **a.shape == b.shape**，那么 **a\*b** 的结果就是 **a** 与 **b** 数组对应位相乘。这要求维数相同，且各维度的长度相同。

实例：

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
print(c)
```

输出结果为：

```
[ 10  40  90 160]
```

当运算中的 2 个数组的形状不同时，numpy 将自动触发广播机制。例如：

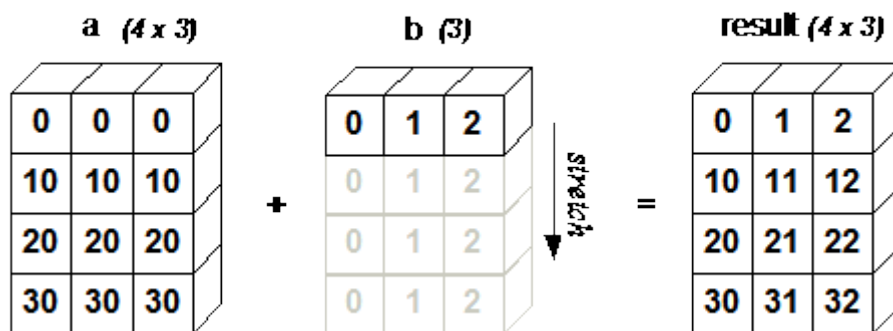
```
import numpy as np
a = np.array([
[ 0, 0, 0],
```

```
[10,10,10],
[20,20,20],
[30,30,30]])
b = np.array([1,2,3])
print(a + b)
```

输出结果为：

```
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]
```

下面的图片展示了数组 `b` 如何通过广播来与数组 `a` 兼容。



4x3 的二维数组与长为 3 的一维数组相加，等效于把数组 `b` 在二维上重复 4 次再运算：

```
import numpy as np
a = np.array([[ 0, 0, 0], [10,10,10], [20,20,20], [30,30,30]])
b = np.array([1,2,3])
bb = np.tile(b, (4, 1)) # 重复 b 的各个维度，得到 4x1 的分块矩阵
print(bb)
print(a + bb)
```

输出结果为：

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

[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]
```

思考：下面的代码实现了什么功能？

```
sqDiffMat = (bb-a)**2
```

```
sqDiffMat.sum(axis=1)
```

```
distances = sqDiffMat ** 0.5
```

广播的规则:

- 让所有输入数组都向其中形状最长的数组看齐，形状中不足的部分都通过在前面加 1 补齐。
- 输出数组的形状是输入数组形状的各个维度上的最大值。
- 如果输入数组的某个维度和输出数组的对应维度的长度相同或者其长度为 1 时，这个数组能够用来计算，否则出错。
- 当输入数组的某个维度的长度为 1 时，沿着此维度运算时都用此维度上的第一组值。

**简单理解:** 对两个数组，分别比较其每一个维度（若其中一个数组没有当前维度则忽略），满足：

- 数组拥有相同形状。
- 当前维度的值相等。
- 当前维度的值有一个是 1。

若条件不满足，抛出 **"ValueError: frames are not aligned"** 异常。

## 11. 迭代数组

数组是可选代对象。

```
A=array([[5, 6, 7],
         [6, 7, 8],
         [7, 8, 9]])
for x in A:
    print(x)
```

```
[5 6 7]
[6 7 8]
[7 8 9]
```

NumPy 迭代器对象 `numpy.nditer` 提供了一种灵活访问一个或者多个数组元素的方式。

迭代器最基本的功能是可以完成对数组元素的访问。

接下来我们使用 `arange()` 函数创建一个 2X3 数组，并使用 `nditer` 对它进行迭代。

实例

```
import numpy as np
a = np.arange(6).reshape(2, 3)
print('原始数组是: \n', a)
print('迭代输出元素: ')
for x in np.nditer(a):
    print(x, end=", ")
```

输出结果为:

原始数组是:

```
[[0 1 2]
 [3 4 5]]
```

迭代输出元素:

```
0, 1, 2, 3, 4, 5,
```

以上实例不是使用标准 C 或者 Fortran 顺序，选择的顺序是和数组内存布局一致的，这样做是为了提升访问的效率，默认是行序优先（row-major order，或者说是 C-order）。

这反映了默认情况下只需访问每个元素，而无需考虑其特定顺序。我们可以通过迭代上述数组的转置来看到这一点，并与以 C 顺序访问数组转置的 copy 方式做对比，如下实例：

实例

```
import numpy as np
a = np.arange(6).reshape(2, 3)
for x in np.nditer(a.T):
    print(x, end=", ")
print('\n')
for x in np.nditer(a.T.copy(order='C')):
    print(x, end=", ")
```

输出结果为：

```
0, 1, 2, 3, 4, 5,
0, 3, 1, 4, 2, 5,
```

从上述例子可以看出，a 和 a.T 的遍历顺序是一样的，也就是他们在内存中的存储顺序也是一样的，但是 a.T.copy(order = 'C') 的遍历结果是不同的，那是因为它和前两种的存储方式是不一样的，默认是按行访问。

#### （1）控制遍历顺序

- for x in np.nditer(a, order='F'):Fortran order，即是列序优先；
- for x in np.nditer(a.T, order='C'):C order，即是行序优先；

实例

```
import numpy as np
a = np.arange(1, 13).reshape(3, 4)
print('原始数组是: \n', a)
print('原始数组的转置是: \n', a.T)
print('以 C 风格顺序排序: ')
b = a.copy(order='C') # 存储顺序为行优先，但矩阵元素的相对位置改变
print(b)
for x in np.nditer(b):
    print(x, end=", ")
print('\n 以 F 风格顺序排序: ')
```

```
c = a.copy(order='F') # 存储顺序为列优先，但矩阵元素的相对位置改变
print(c)
for x in np.nditer(c):
    print(x, end=", ")
```

输出结果为：

原始数组是：

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

原始数组的转置是：

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

以 C 风格顺序排序：

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

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

以 F 风格顺序排序：

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

1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12,

可以通过显式设置，来强制 `nditer` 对象使用某种顺序。

实例

```
import numpy as np
a = np.arange(1, 13).reshape(3, 4)
print('原始数组是： \n', a)
print('以 C 风格顺序排序： ')
for x in np.nditer(a, order='C'):
    print(x, end=", ")
print('\n 以 F 风格顺序排序： ')
for x in np.nditer(a, order='F'):
    print(x, end=", ")
```

输出结果为：

原始数组是：

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

```
[ 9 10 11 12]]
```

以 C 风格顺序排序：  
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  
以 F 风格顺序排序：  
1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12,

## 12. 数组操作

### (1) numpy.reshape

numpy.reshape 函数可以在不改变数据的条件下修改形状，格式如下：

numpy.reshape(arr, newshape, order='C')

#### 实例

<pre>import numpy as np a = np.arange(8) print('原始数组: ') print(a) b = a.reshape(4,2) # 行优先 print('修改后的数组: ') print(b)</pre>	<pre>b = a.reshape(4,2, order='F') # 列优先</pre> <pre>[[0 4]  [1 5]  [2 6]  [3 7]]</pre>
---	--

输出结果如下：

原始数组：  
[0 1 2 3 4 5 6 7]  
修改后的数组：  
[[0 1]  
 [2 3]  
 [4 5]  
 [6 7]]

### (2) numpy.transpose 与 numpy.ndarray.T

numpy.transpose 函数用于对换数组的维度，格式如下：

numpy.transpose(arr, axes)

#### 实例

```
import numpy as np
a = np.arange(12).reshape(3,4)
print('原数组: ')
print(a)
print('对换数组: ')
print(np.transpose(a))
```

输出结果如下：

原数组：

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

对换数组:

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

`numpy.ndarray.T` 类似 `numpy.transpose`:

实例

```
import numpy as np
a = np.arange(12).reshape(3,4)
print('原数组: ')
print(a)
print('转置数组: ')
print(a.T)
```

输出结果如下:

原数组:

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

转置数组:

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

### (3) `numpy.squeeze`

`squeeze` 函数从数组的形状中移除长度为 1 的轴（例如单行矩阵转化为向量），函数格式如下:

```
numpy.squeeze(arr, axis)
```

实例: 移除长度为 1 的轴

```
import numpy as np
x = np.arange(9).reshape(1,3,3)
print('数组 x: ')
print(x)
y = np.squeeze(x)
print('数组 y: ')
print(y)
print('数组 x 和 y 的形状: ')
print(x.shape, y.shape)
```

输出结果为:

数组 x:

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

数组 y:

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

数组 x 和 y 的形状:

```
(1, 3, 3) (3, 3)
```

```
Python 控制台 x
>>> np.reshape(a,(2,-1))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> B=np.reshape(a,(2,-1))
>>> C=np.reshape(a,(1,10))
>>> d=np.squeeze(C)
>>>
```

```
> A = {ndarray: (3, 2)} [[1 2], [3 4], [5 6]] ...作为Array查看
> B = {ndarray: (2, 5)} [[0 1 2 3 4], [5 6 7 8 9]] ...作为Array查看
> C = {ndarray: (1, 10)} [[0 1 2 3 4 5 6 7 8 9]] ...作为Array查看
> a = {ndarray: (10,)} [0 1 2 3 4 5 6 7 8 9] ...作为Array查看
> b = {list: 3} [2, 7, 8]
01 c = {int} 5
> d = {ndarray: (10,)} [0 1 2 3 4 5 6 7 8 9] ...作为Array查看
> id = {ndarray: (10,)} [False False False False False False True True] ...作为Array查看
01 x = {int} 6
01 y = {int} 5
01 z = {int} 5
> 特殊变量
```

单行矩阵  
向量

#### (4) 连接数组

函数	描述
concatenate	连接沿现有轴的数组序列
stack	沿着新的轴加入一系列数组。
hstack	水平堆叠序列中的数组（列方向）
vstack	竖直堆叠序列中的数组（行方向）

numpy.concatenate 函数用于沿指定轴连接相同形状的两个或多个数组，格式如下:

```
numpy.concatenate((a1, a2, ...), axis)
```

参数说明:

- a1, a2, ...: 相同类型的数组
- axis: 沿着它连接数组的轴，默认为 0

实例

```
import numpy as np
a = np.array([[1,2],[3,4]])
print('第一个数组: ')
print(a)
b = np.array([[5,6],[7,8]])
print('第二个数组: ')
print(b)
# 两个数组的维度相同
print('沿轴 0 连接两个数组: ')
```



```
print(np.concatenate((a,b)))
print('\n')
print('沿轴 1 连接两个数组: ')
print(np.concatenate((a,b),axis = 1))
```

输出结果为:

第一个数组:

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

第二个数组:

```
[[5 6]
 [7 8]]
```

沿轴 0 连接两个数组:

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

0  
轴  
↓

沿轴 1 连接两个数组:

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

numpy.stack 函数用于沿新轴连接数组序列, 格式如下:

```
numpy.stack(arrays, axis)
```

参数说明:

- arrays 相同形状的数组序列
- axis: 返回数组中的轴, 输入数组沿着它来堆叠

实例

```
import numpy as np
a = np.array([[1,2],[3,4]])
print('第一个数组: ')
print(a)
b = np.array([[5,6],[7,8]])
print('第二个数组: ')
print(b)
print('沿轴 0 堆叠两个数组: ')
print(np.stack((a,b),0))
print('沿轴 1 堆叠两个数组: ')
print(np.stack((a,b),1))
print('沿轴 2 堆叠两个数组: ')
print(np.stack((a,b),2))
```

输出结果如下:

第一个数组:

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

第二个数组:

```
[[5 6]
 [7 8]]
```

沿轴 0 堆叠两个数组:

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

  [[5 6]
   [7 8]]]
```

沿轴 1 堆叠两个数组:

```
[[[1 2]
   [5 6]]

  [[3 4]
   [7 8]]]
```

沿轴 2 堆叠两个数组:

```
[[[1 5]
   [2 6]]

  [[3 7]
   [4 8]]]
```

`numpy.hstack` 是 `numpy.stack` 函数的变体，它通过水平堆叠来生成数组。

`numpy.vstack` 是 `numpy.stack` 函数的变体，它通过垂直堆叠来生成数组

实例

```
import numpy as np
a = np.array(
    [[1,2],
     [3,4]])
print('第一个数组: ')
print(a)
b = np.array(
    [[5,6],
     [7,8]])
print('第二个数组: ')
print(b)
print('水平堆叠: ')
c = np.hstack((a,b))
print(c)
d = np.vstack((a,b))
```

```
print(d)
```

输出结果如下：

第一个数组：

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

第二个数组：

```
[[5 6]
 [7 8]]
```

水平堆叠：

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

竖直堆叠：

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

### (5) numpy.append

`numpy.append` 函数在数组的末尾添加值。追加操作会分配整个数组，并把原来的数组复制到新数组中。此外，输入数组的维度必须匹配否则将生成 `ValueError`。

`append` 函数返回的始终是一个一维数组。

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

参数说明：

- `arr`：输入数组
- `values`：要向 `arr` 添加的值，需要和 `arr` 形状相同（除了要添加的轴）
- `axis`：默认为 `None`。当 `axis` 无定义时，是横向加成，返回总是为一维数组！当 `axis` 为 0 时，数组是加在下边（列数要相同）；当 `axis` 为 1 时，数组是加在右边（行数要相同）。

实例

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print('第一个数组: ')
print(a)
print('向数组添加元素: ')
print(np.append(a, [7,8,9]))
print('沿轴 0 添加元素: ')
print(np.append(a, [[7,8,9]],axis = 0))
print('沿轴 1 添加元素: ')
print(np.append(a, [[5,5,5],[7,8,9]],axis = 1))
```

输出结果为：

第一个数组：

```
[[1 2 3]
 [4 5 6]]
```

向数组添加元素：

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

沿轴 0 添加元素：

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

沿轴 1 添加元素：

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

## 四、ufunc 函数

ufunc 是 universal function 的缩写，它是一种能对数组的每个元素进行运算的函数。NumPy 内置的许多 ufunc 函数都是用 C 语言实现的，因此它们的计算速度非常快。让我们先看一个例子：

```
import numpy as np
x=np.arange(1,5)
print('x=',x)
y=np.sin(x)
print('sin(x)=',y)
y
```

输出结果为：

```
x= [1 2 3 4]
sin(x)= [ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

可见，`np.sin()`函数计算了一维数组 `x` 中每个元素的正弦值，并以一维数组形式返回。

由于 `np.sin()` 是一个 ufunc 函数，因此在其内部对数组 `x` 的每个元素进行循环，分别计算它们的正弦值，并返回一个保存各个计算结果的数组。运算之后数组 `x` 中的值并没有改变，而是新建了一个数组来保存结果。

对于数组运算来说，`np.sin()`比循环调用 `math.sin()`快很多，这得益于 `np.sin()`在 C 语言级别的循环计算。但对单个数值的计算，`math.sin()`则比 `np.sin()`快很多。在 Python 级别进行循环时，

`np.sin()`的计算速度只有 `math.sin()`的 1/6。这是因为：`np.Sin()`为了同时支持数组和单个数值的计算，其 C 语言的内部实现要比 `math.sin()`复杂很多。此外，对于单个数值的计算，`np.sin()`的返回值类型和 `math.sin()`的不同，`math.sin()`返回的是 Python 的标准 `float` 类型，而 `np.sin()`返回 `float64` 类型。

NumPy 提供了许多 `ufunc` 函数。

## 1. 四则运算函数及其运算符

表达式	对应的 <code>ufunc</code> 函数
<code>y = x1 + x2</code>	<code>add(x1, x2[, y])</code>
<code>y = x1 - x2</code>	<code>subtract(x1, x2[, y])</code>
<code>y = x1 * x2</code>	<code>multiply(x1, x2[, y])</code>
<code>y = x1 / x2</code>	<code>divide(x1, x2[, y])</code> ，如果两个数组的元素为整数，那么用整数除法
<code>y = x1 / x2</code>	<code>true_divide(x1, x2[, y])</code> ，总是返回精确的商
<code>y = x1 // x2</code>	<code>floor_divide(x1, x2[, y])</code> ，总是对返回值取整
<code>y = -x</code>	<code>negative(x[, y])</code>
<code>y = x1 ** x2</code>	<code>power(x1, x2[, y])</code>
<code>y = x1 % x2</code>	<code>remainder(x1, x2[, y])</code> ， <code>mod(x1, x2[, y])</code>

实例：一维数组的四则运算

```
import numpy as np
a = np.arange(0, 4)
b = np.arange(1, 5)
c = np.add(a, b) # c=a+b
d = a + b
e = a + 10 # 广播后运算:a 各元素+10
f = a*b
g = 10*a # 广播后运算:a 各元素*10
h = b**a
i = b**2 # 广播后运算:b 各元素**2
print('a=', a)
print('b=', b)
print('c=', c)
print('d=', d)
print('e=', e)
print('f=', f)
```

```
a= [0 1 2 3]
b= [1 2 3 4]
c= [1 3 5 7]
d= [1 3 5 7]
e= [10 11 12 13]
f= [ 0  2  6 12]
g= [ 0 10 20 30]
h= [ 1  2  9 64]
i= [ 1  4  9 16]

a= [1 3 5 7]
```

```
print('g=', g)
print('h=', h)
print('i=', i)
np.add(a, b, a)  # a += b
print('a=', a)
```

如果参与运算的两个对象形状不同，必须满足广播条件，否则无法运算。

#### 实例

```
import numpy as np
a = np.arange(1, 4)
b = np.arange(1, 2)  # b 只有一个元素
c = np.arange(1, 5)
print('a=', a)
print('b=', b)
print('c=', c)
print('a + b =', a + b)  # b 沿轴0 广播后运算
print('a + c =', a + c)  # 无法广播
```

输出结果为：

```
Traceback (most recent call last):
  File "C:/Users/ruanz/PycharmProjects/new/ch13/demo_13_03.py", line 14, in
<module>
    print('a + c =', a + c)  # 无法广播
ValueError: operands could not be broadcast together with shapes (3,) (4,)
a= [1 2 3]
b= [1]
c= [1 2 3 4]
a + b = [2 3 4]
```

#### 实例：二维数组的四则运算

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[1, 1, 1], [2, 2, 2]])
print('A=\n', A)
print('B=\n', B)
print('A + B=\n', A + B)
print('A * B=\n', A * B)
e = np.array([1, 2, 3])
```

```
print('e=\n', e)
print('A + e =\n', A + e) # 沿轴0广播后运算
f = np.array([1, 2, 3, 4])
print('f=\n', f)
print('A + f =\n', A + f) # 无法广播
```

输出结果为：

```
A=
[[1 2 3]
 [4 5 6]]
B=
[[1 1 1]
 [2 2 2]]
A + B=
[[2 3 4]
 [6 7 8]]
A * B=
[[ 1  2  3]
 [ 8 10 12]]
e=
[1 2 3]
A + e =
[[2 4 6]
 [5 7 9]]
f=
[1 2 3 4]
Traceback (most recent call last):
  File "C:/Users/ruanz/PycharmProjects/new/ch13/demo_13_04.py", line 19, in
<module>
    print('A + f =\n', A + f) # 无法广播
ValueError: operands could not be broadcast together with shapes (2,3) (4,)
```

## 2. 比较运算函数及其运算符

表达式	对应的 ufunc 函数
$y = x1 == x2$	<code>equal(x1, x2[, y])</code>
$y = x1 != x2$	<code>not_equal(x1, x2[, y])</code>
$y = x1 < x2$	<code>less(x1, x2[, y])</code>
$y = x1 <= x2$	<code>less_equal(x1, x2[, y])</code>
$y = x1 > x2$	<code>greater(x1, x2[, y])</code>
$y = x1 >= x2$	<code>greater_equal(x1, x2[, y])</code>

实例：一维数组的比较运算

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([3, 0, 3])
print('a=', a)
print('b=', b)
print('a > b=', a > b) # 返回一个布尔数组
```

输出结果为:

```
a= [1 2 3]
b= [3 0 3]
a > b= [False True False]
```

### 3. 逻辑运算函数

由于 Python 中的逻辑运算使用 `and`、`or` 和 `not` 等关键字，它们无法被重载，因此布尔数组的逻辑运算只能通过相应的 `ufunc` 函数进行。这些函数名都以 `logical_` 开头:

**logical\_and   logical\_or   logical\_not   logical\_xor**

实例：一维布尔数组的逻辑运算

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([3, 0, 3])
print('a=', a)
print('b=', b)
print('a > b:\n', a > b) # 返回一个布尔数组
print('a == b:\n', a == b) # 返回一个布尔数组
c = np.logical_and(a > b, a == b)
print('np.logical_and(a>b,a==b):\n', c) # 返回一个布尔数组
```

输出结果为:

```
a= [1 2 3]
b= [3 0 3]
a > b:
[False True False]
a == b:
[False False True]
np.logical_and(a>b,a==b):
[False False False]
```



如果对两个布尔数组使用 `and`、`or` 和 `not` 等进行布尔运算，将抛出 `ValueError` 异常。因为布尔数组中有 `True` 也有 `False`，所以 NumPy 无法确定用户的运算目的。

#### 4. 位运算函数及其运算符

表达式	对应的 <code>ufunc</code> 函数
<code>y = x1 &amp; x2</code>	<code>bitwise_and(x1, x2[, y])</code>
<code>y = x1   x2</code>	<code>bitwise_or (x1, x2 [, y])</code>
<code>y = ~x1</code>	<code>bitwise_not (x1, [, y])</code>
<code>y = ^x1</code>	<code>bitwise_xor (x1, [, y])</code>

实例：布尔数组的按位运算与逻辑运算的结果相同

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([3, 0, 3])
print('a=', a)
print('b=', b)
print('a > b:\n', a > b) # 返回一个布尔数组
print('a == b:\n', a == b) # 返回一个布尔数组
c = np.logical_and(a > b, a == b)
print('np.logical_and(a>b,a==b):\n', c) # 返回一个布尔数组
d = (a > b) & (a == b) # 注意运算符优先级
print('(a > b) & (a == b):\n', d) # 返回一个布尔数组
```

输出结果为：

```
a= [1 2 3]
b= [3 0 3]
a > b:
[False  True False]
a == b:
[False False  True]
np.logical_and(a>b,a==b):
[False False False]
(a > b) & (a == b):
[False False False]
```

注意，对于布尔数组，按位运算与逻辑运算的结果相同。

整数数组按位取反运算时，注意元素类型的符号，有符号与无符号的运算结果不一样。例如对于有符号整数，对正数取反将得到负数：

`~np.array([0, 1, 2, 3, 4])`

`array([-1, -2, -3, -4, -5], dtype=int32)`

对于无符号整数:

`~np.array([0, 1, 2, 3, 4], dtype=np.uint8)`

`array([255, 254, 253, 252, 251], dtype=uint8)`

0 的 2 进制: 00000000  
~0 : 11111111 → -1 的 2 进制

-1 的 2 进制: 10000001  
-1 的 2 进制: 11111110  
-1 的 2 进制: 11111111 → +1

3 的 2 进制: 00000011  
~3 : 11111100 → -4 的 2 进制

## 5. 自定义 ufunc 函数

通过 `np.frompyfunc()` 可以将计算单个值的函数转换为能对数组的每个元素进行计算的 ufunc 函数, 其调用格式为:

`f_ufunc=np.frompyfunc(f, nin, nout)`

`f` 是计算单个元素的函数, `nin` 是 `func` 的输入参数的个数, `nout` 是 `func` 的返回值的个数, `f_ufunc` 是 ufunc 函数对象, 作为一个新的函数, 它实现了 `f` 函数的 ufunc 功能。`f_ufunc()` 的调用格式为:

`y= f_ufunc(与 f 中定义的形参对应的实际参数)`

实例: 定义一个计算二次多项式的函数 `f` (计算单个值), 然后定义其 ufunc 函数并调用

```
import numpy as np
def f(a, b, c, x):
    """ 计算二次函数值 """
    return a * x ** 2 + b * x + c

a, b, c = 1, 0, 7 # 对应: y(x)=x**2+7
x = np.array([0, 1, 2])
f_ufunc = np.frompyfunc(f, 4, 1) # 生成 ufunc 函数对象
y = f_ufunc(a, b, c, x) # 调用 f_ufunc() 函数
print(y.dtype)
y = y.astype(np.float) # 注意: f_ufunc() 所返回的数组的元素类型是 object,
                        # 因此还需要调用数组的 astype() 方法, 以将其转换为双精度浮点数组
print(y.dtype)
print(y)
```

输出结果为:

object  
float64

使用 `np.vectorize()` 也可以实现和 `np.frompyfunc()` 类似的功能，但它可以通过 `otypes` 参数指定返回的数组的元素类型。`otypes` 参数可以是一个表示元素类型的字符串，也可以是一个类型列表，使用列表可以描述多个返回多个数组的元素类型。

例如在上例中添加代码：

```
f_ufunc2 = np.vectorize(f, otypes=[np.float]) # 生成 ufunc 函数对象
y2 = f_ufunc2(a, b, c, x)
print(np.all(y == y2)) # 验证
```

则输出结果为：

```
object
float64
[ 7.  8. 11.]
True
```

## 6. 数学函数

NumPy 包含大量的各种数学运算的函数，包括三角函数，算术运算的函数，复数处理函数等。

- 算术运算函数：`abs()`、`sqrt()`、`power()`、`exp()`等
- 标准三角函数：`sin()`、`cos()`、`tan()`等，角度单位为弧度，角度  $x$  转化为弧度  $y$ ：  
 $y = x/180 * \pi$
- 反三角函数：`arcsin()`、`arccos()`和 `arctan()`等，函数的结果可以通过 `degrees()`函数将弧度转换为角度。
- 复数处理函数：`real()`、`imag()`、`angle()`、`conj()`等
- `around(a, decimals)` 函数返回指定数字的四舍五入值。
- `floor()` 返回小于或者等于指定表达式的最大整数，即向下取整。
- `ceil()` 返回大于或者等于指定表达式的最小整数，即向上取整。
- `ceil()` 返回整数部分。

这些 `ufunc` 函数可以对标量进行运算，也可以对列表、元组、数组等集合进行运算。

实例：

```
import numpy as np
a = np.abs(-np.pi)
print(a)
b = [1, 3 + 4j, -8]
print(np.real(b)) # 传入列表，返回一维数组
```

```
print(np.abs(b)) # 传入列表, 返回一维数组
print(np.real(np.array(b))) # 传入一维数组, 返回一维数组
A = np.array([[1.6, 3.3], [-1.6, -3.3]])
print(A)
print(np.floor(A)) # 传入二维数组, 返回二维数组
print(np.fix(A)) # 传入二维数组, 返回二维数组
```

输出结果为:

```
3.141592653589793
[ 1.  3. -8.]
[1. 5. 8.]
[ 1.  3. -8.]
[[ 1.6  3.3]
 [-1.6 -3.3]]
[[ 1.  3.]
 [-2. -4.]]
[[ 1.  3.]
 [-1. -3.]]
```

## 7. 统计函数

NumPy 提供了很多统计函数, 用于从数组中计算元素和、元素积、最小元素、最大元素、中位数、百分位、平均值、标准差和方差等。统计中可以指定数组的轴, 即在指定的轴上统计, 否则所有元素一起统计。

- `sum()`:
- `min()/amin()`、`max()/amax()`:
- `mean()`、`var()`、`std()`:
- `median()`、`percentile()`:

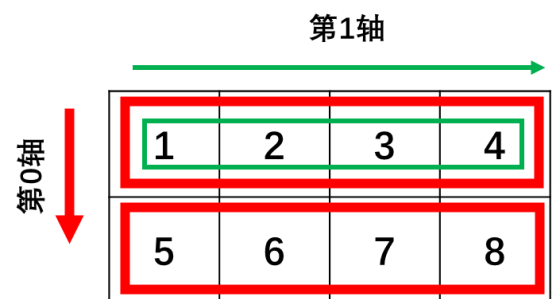
实例: 一维数组的统计

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a.sum()) # 求和: 10
print(a.prod()) # 求积: 24
print(a.mean()) # 求均值: 2.5
print(a.var()) # 求方差: 1.25
print(a.std()) # 求标准差  $std=\sqrt{var}$ : 1.118033988749895
print(a.min()) # 求最小值: 1
```

```
print(a.max()) # 求最大值: 4
```

实例：二维数组的统计

```
import numpy as np
a = [[1, 2, 3, 4],
      [5, 6, 7, 8]]
A = np.array(a)
print(A)
# 不指定轴，则所有元素一起统计，否则指定轴统计
print(A.sum()) # 求和: 36
print(A.mean()) # 求均值: 4.5
print(A.var()) # 求方差: 5.25
print(A.min()) # 求最小值: 1
# 指定轴0，即按列
print(A)
print(A.sum(axis=0)) # 求和: [ 6  8 10 12]
print(A.mean(axis=0)) # 求均值: [3.  4.  5.  6.]
print(A.var(axis=0)) # 求方差: [4.  4.  4.  4.]
print(A.min(axis=0)) # 求最小值: [1  2  3  4]
```



该二维数组是由 2 个一维数组构成的一维数组，而每个一维数组由 4 个元素，即二维数组的第 1 个大元素是  $A[0]=[1\ 2\ 3\ 4]$ ，第 2 个大元素是  $A[1]=[5\ 6\ 7\ 8]$ ，大元素对应第 0 轴，索引号包括 0, 1。

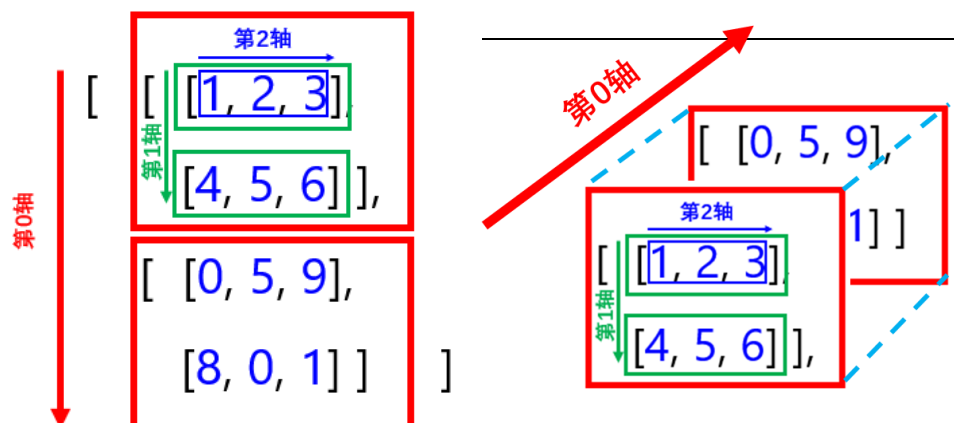
每个一维数组内部对应第 1 轴，其元素索引号包括 0, 1, 2, 3。

对第 0 轴操作，就是各行之间的对应元素的操作，例如找最大值，得到的是各列的最大值；

对第 1 轴操作，就是每个行自身内部元素的操作，例如找最大值，得到的是各行的最大值。

实例：三维数组的统计

```
import numpy as np
a = [[[1, 2, 3],
      [4, 5, 6]],
      [[0, 5, 9],
      [8, 0, 1]]]
A = np.array(a)
print(A)
```



# 不指定轴，则所有元素一起统计

```
print(A.min()) # 求最小值: 0
```

# 指定轴 0, 三维数组由 2 个 2x3 的二维数组 A[0]、A[1] 构成，第 0 轴的索引号包括 0,1

# 完成的是两个 2x3 的数组之间对应元素的比较，因此得到 2x3 的数组

```
print(A.min(axis=0)) # 求最小值:
```

```
# [[0 2 3]
```

```
# [4 0 1]]
```

# 指定轴 1，每个二维数组由 2 个一维数组构成，包括了两个轴，第 1 轴对应各一维数组，第 2 轴对应一维数组内部

# 完成的是各个二维数组内部列操作，每个二维数组按列操作后，分别得到含 3 个元素的行向量，最后返回 2x3 的数组

```
print(A.min(axis=1)) # 求最小值:
```

```
# [[1 2 3]
```

```
# [0 0 1]]
```

# 指定轴 2，

# 完成的是各个二维数组内部行操作，每个二维数组按行操作后，分别得到含 2 个元素的行向量，最后返回 2x2 的数组

```
print(A.min(axis=2)) # 求最小值:
```

```
# [[1 4]
```

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

## 五、 排序和条件筛选

### 1. 排序 `numpy.sort()`

`numpy.sort()` 函数返回输入数组的排序副本。函数格式如下：

```
numpy.sort(a, axis, kind, order)
```

- `a`: 要排序的数组
- `axis`: 沿着它排序数组的轴，如果没有数组会被展开，沿着最后的轴排序，`axis=0` 按列排序，

axis=1 按行排序

- kind: 指定排序算法，默认为'quicksort'（快速排序），其他算法包括'mergesort'（归并排序）、'heapsort'（堆排序）等
- order: 如果数组包含字段，则是要排序的字段

例子：

```
import numpy as np
a = np.array([[3, 7], [9, 1]])
print('我们的数组是：')
print(a)
print('调用 sort() 函数：')
print(np.sort(a))
print('按列排序：')
print(np.sort(a, axis=0))
# 在 sort 函数中排序字段
dt = np.dtype([('name', 'S10'), ('age', int)])
a = np.array([(b'raju', 21), (b'anil', 25), (b'ravi', 17), (b'amar', 27)], dtype=dt)
print('我们的数组是：')
print(a)
print('按 name 排序：')
print(np.sort(a, order='name'))
```

输出结果为：

```
我们的数组是：
[[3 7]
 [9 1]]
调用 sort() 函数：
[[3 7]
 [1 9]]
按列排序：
[[3 1]
 [9 7]]
我们的数组是：
[(b'raju', 21) (b'anil', 25) (b'ravi', 17) (b'amar', 27)]
按 name 排序：
[(b'amar', 27) (b'anil', 25) (b'raju', 21) (b'ravi', 17)]
```

## 2. 获取非零元素索引排序 `numpy.nonzero()`

`numpy.nonzero()` 函数返回输入数组中非零元素的索引。

例子：

```
import numpy as np
a = np.array([[30, 40, 0], [0, 20, 10], [50, 0, 60]])
print('我们的数组是：')
print(a)
print('\n')
print('调用 nonzero() 函数：')
index = np.nonzero(a) # 返回一个元组，元素类型为数组
print(index)
print(index[0])
print(index[1])
```

输出结果为：

```
我们的数组是：
[[30 40  0]
 [ 0 20 10]
 [50  0 60]]
调用 nonzero() 函数：
(array([0, 0, 1, 1, 2, 2], dtype=int64), array([0, 1, 1, 2, 0, 2],
dtype=int64))
[0 0 1 1 2 2]
[0 1 1 2 0 2]
```

## 3. 矢量化判断表达式函数 `numpy.where()`

Python 中的判断表达式语法为：

**`x = y if condition else z`**

当 `condition` 条件为 `True` 时，表达式的值为 `y`，否则为 `z`。

NumPy 中的 `where()` 函数可以看作判断表达式的数组版本：

**`x = np.where(condition[, y, z])`**

其中 `condition`、`y` 和 `z` 都是数组，它的返回值是一个形状与 `condition` 相同的数组。当 `condition` 中的某个元素为 `True` 时，`x` 中对应下标的值从数组 `y` 获取，否则从数组 `z` 获取。

```
x = np.arange(10)
```

```
np.where(x < 5, 9 - x, x)
```



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

因此，`where()`函数可以用来计算分段函数的值。

如果 `y` 和 `z` 是单个数值或者它们的形状与 `condition` 的不同，将先通过广播运算使其形状一致：

```
x = np.arange(10)
np.where(x > 6, 2 * x, 0)
array([0, 0, 0, 0, 0, 0, 0, 14, 16, 18])
```

`where()`函数也可以嵌套调用，以完成多重分支函数的计算。例如：

$$f(x) = \begin{cases} e^x + 2, & x < 0 \\ 3 & 0 \leq x < 2 \\ x + 1 & x \geq 2 \end{cases}$$

```
x = np.arange([-2, -1, 0, 1, 2, 3, 4])
np.set_printoptions(2) # 设置输出小数位
np.where(x < 0,
        np.exp(x) + 2,
        np.where(x >= 2,
                x + 1,
                3))
```

```
array([2.14, 2.37, 3. , 3. , 3. , 4. , 5. ])
```

如果没有指定参数 `y` 和 `z`，则 `np.where(condition)` 函数返回的是输入数组中满足给定条件 `condition` 的元素的索引。

例子：

```
import numpy as np
x = np.arange(9.).reshape(3, 3)
print('我们的数组是：')
print(x)
print('大于 3 的元素的索引：')
y = np.where(x > 3)
print(y)
print('使用这些索引来获取满足条件的元素：')
print(x[y])
```

输出结果为：

```
我们的数组是：
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

大于 3 的元素的索引:

```
(array([1, 1, 2, 2, 2], dtype=int64), array([1, 2, 0, 1, 2], dtype=int64))
```

使用这些索引来获取满足条件的元素:

```
[4. 5. 6. 7. 8.]
```

#### 4. 矢量化多重分支判断选择函数 `numpy.select()`

随着分段函数的分段数量的增加, 需要嵌套更多层 `where()`。这样不便于程序的编写和阅读。可以用 `select()` 解决这个问题, 它的调用形式如下:

```
select(condlist, choicelist, default=0)
```

其中 `condlist` 是一个长度为 `N` 的布尔数组列表, `choicelist` 是一个长度为 `N` 的存储候选值的数组列表, 所有数组的长度都为 `M`。如果列表元素不是数组而是单个数值, 那么它相当于元素值都相同、长度为 `M` 的数组。

例如:

$$f(x) = \begin{cases} e^x + 2, & x < 0 \\ 3 & 0 \leq x < 2 \\ x + 1 & x \geq 2 \end{cases}$$

```
np.select([x < 0,          x >= 2],  
          [np.exp(x) + 2, x + 1],  
          default=3)  
array([2.14, 2.37, 3.  , 3.  , 3.  , 4.  , 5.  ])
```

与前面的 `where()` 嵌套是等价的。

#### 5. 矢量化高效多重分支判断选择函数 `numpy.piecewise()`

`where()` 和 `select()` 的所有参数都需要在调用它们之前完成计算, 例如:

```
np.select([x < 0,          x >= 2],  
          [np.exp(x) + 2, x + 1],  
          default=3)
```

NumPy 会先计算 4 个数组: `x < 0`, `x < 2`, `np.exp(x) + 2` 和 `x + 1`。

在计算时还会产生许多保存中间结果的数组, 因此如果输入的数组 `X` 很大, 将会发生大量内存分配和释放。

为了解决这个问题, NumPy 提供了 `piecewise()` 专门用于计算分段函数, 它的调用参数如下:

```
piecewise(x, condlist, funclist)
```

参数 `x` 是一个保存自变量值的数组, `condlist` 是一个长度为 `M` 的布尔数组列表, 其中的每个布

尔数组的长度都和数组 `x` 相同。 `funclist` 是一个长度为 `M` 或 `M+1` 的函数列表，这些函数的输入和输出都是数组。它们计算分段函数中的每个片段。如果不是函数而是数值，则相当于返回此数值的函数。每个函数与 `condlist` 中下标相同的布尔数组对应，如果 `funclist` 的长度为 `M+1`，则最后一个函数计算所有条件都为 `False` 时的值。

例如：

$$f(x) = \begin{cases} e^x + 2, & x < 0 \\ 3 & 0 \leq x < 2 \\ x+1 & x \geq 2 \end{cases}$$

```
x = x.astype(float) # 这里 x 应该为浮点类型，否则有可能出错!!! why?
np.piecewise(x,
             [x < 0, x >= 2], # condlist
             [lambda x: np.exp(x) + 2, # x < 0
              lambda x: x + 1,          # x >= 2
              lambda x: 3])            # else
array([2.14, 2.37, 3. , 3. , 3. , 4. , 5. ])
```

使用 `piecewise()` 的好处在于它只计算需要计算的值。因此在上面的例子中，表达式 `np.exp(x) + 2` 只对输入数组 `x` 中满足条件的部分进行计算。

## 六、 NumPy 随机数模块(random)

`numpy.random` 模块中提供了大量的随机数相关的函数。

函数名	功能	函数名	功能
<b>rand</b>	0 到 1 之间的随机数	<b>randn</b>	标准正态分布 $N(0, 1)$ 的随机数
<b>randint</b>	指定范围内的随机整数	<b>normal</b>	正态分布 $N(\mu, \sigma)$
<b>uniform</b>	均匀分布	<b>poisson</b>	泊松分布
<b>permutation</b>	随机排列	<b>shuffle</b>	随机打乱顺序
<b>choice</b>	随机抽取样本	<b>seed</b>	设置随机数种子

### 1. `rand()`、`randn()` 与 `randint()`

- `rand()` 产生 0 到 1 之间的随机浮点数，它的所有参数用于指定所产生的数组的形状。
- `randn()` 产生标准正态分布的随机数，参数的含义与 `rand()` 相同。
- `randint()` 产生指定范围的随机整数，包括起始值，但是不包括终值。

例子：

```
import numpy as np
from numpy import random as nr
np.set_printoptions(precision=2) # 为了节省篇幅, 只显示小数点后两位数字
r1 = nr.rand(2, 4)
r2 = nr.randn(2, 4)
r3 = nr.randint(0, 10, (2, 4))
```

某次运行的对应结果为:

<b>r1</b>	array([[0.11, 0.21, 0.09, 0.28], [0.19, 0.99, 0.75, 0.18]])
<b>r2</b>	array([[ 0.19, 1.44, 0.57, -0.39], [ 0.82, 1.63, 0.71, 1.12]])
<b>r3</b>	array([[7, 7, 6, 9], [0, 6, 6, 8]])

## 2. normal()、uniform() 与 poisson() 等

random 模块提供了许多产生符合特定随机分布的随机数的函数, 它们的最后一个参数 **size** 都用于指定输出数组的形状, 而其他参数都是分布函数的参数。例如:

- **normal()**: 正态分布, 前两个参数分别为期望值和标准差。
- **uniform()**: 均匀分布, 前两个参数分别为区间的起始值和终值。
- **poisson()**: 泊松分布, 第一个参数指定  $\lambda$  系数, 它表示单位时间(或单位面积)内随机事件的平均发生率。由于泊松分布是一个离散分布, 因此它输出的数组是一个整数数组。

例子:

```
r1 = nr.normal(100, 10, (2, 4))
r2 = nr.uniform(10, 20, (2, 4))
r3 = nr.poisson(2.0, (2, 4))
```

某次运行的对应结果为:

<b>r1</b>	array([[ 88.4 , 89.05, 107.22, 121.06], [103.76, 106.86, 82.34, 107.76]])
<b>r2</b>	array([[12.03, 15.88, 15.67, 17.66], [11.23, 18.33, 16.57, 11.4 ]])
<b>r3</b>	array([[3, 4, 1, 2], [1, 2, 2, 3]])

### 3. permutation()、shuffle() 与 poisson() 等

permutation()可以用于产生一个乱序数组，当参数为整数 n 时，它返回[0, n)这 n 个整数的随机排列；当参数为一个序列时，它返回一个随机排列之后的序列。

例子：

```
import numpy as np
from numpy import random as nr
a = np.array([1, 10, 20, 30, 40])
print(nr.permutation(10))
print(nr.permutation(a))
```

输出结果为：

```
[4 8 6 3 1 9 2 0 7 5]
[10 40 1 20 30]
```

permutation()返回一个新数组，而 shuffle()则直接将参数数组的顺序打乱：

```
nr.shuffle(a)
print(a) # [10 40 30 20 1]
```

choice()从指定的样本中随机进行抽取。

- size 参数用于指定输出数组的形状。
- replace 参数为 True 时，进行可重复抽取，而为 False 时进行不重复抽取，默认值为 True。
- p 参数指定每个元素对应的抽取概率，如果不指定，所有的元素被抽取到的概率相同。
- 例子：

```
a = np.arange(10, 25, dtype=float)
c1 = nr.choice(a, size=(4, 3))
c2 = nr.choice(a, size=(4, 3), replace=False) # 不允许不重复抽样
c3 = nr.choice(a, size=(4, 3), p=a / np.sum(a)) # 值越大的元素被抽到的概率越大，因此数值较大的元素比较多
```

某次运行的对应结果为：

c1	c2	c3
array([[13., 11., 20.], [12., 15., 15.], [22., 19., 15.], [24., 20., 10.]])	array([[21., 17., 15.], [24., 12., 13.], [14., 16., 19.], [18., 22., 11.]])	array([[11., 10., 20.], [15., 23., 22.], [18., 18., 21.], [10., 24., 22.]])

## 七、 NumPy 矩阵库(Matrix)

NumPy 中包含了一个矩阵库 `numpy.matlib`，该模块中的函数返回的是一个矩阵（matrix）对象，而不是 ndarray 对象，类似于 MATLAB 中的矩阵，支持矩阵乘法  $A*B$ 、求逆  $A.I$ 、转置  $A.T$  等运算。

一个  $m \times n$  的矩阵是一个由行  $m$ （row）列  $n$ （column）元素排列成的矩形阵列。矩阵里的元素可以是数字、符号或数学式。以下是一个由 6 个数字元素构成的 2 行 3 列的矩阵：

$$\begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}$$

### 1. numpy.matrix()与 numpy.asmatrix() 生成矩阵

这两个函数可以用生成 matrix 对象。例如：`np.asmatrix([[1, 2, 3], [4, 5, 6]])`、`np.matrix([[1, 2, 3], [4, 5, 6]])` 或 `np.mat([[1, 2, 3], [4, 5, 6]])` 都将产生：

```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

`A=np.mat(np.random.rand(4,4))`生成随机矩阵：

```
matrix([[0.07711176, 0.52762327, 0.56875003, 0.40327313],
        [0.9522149 , 0.67086255, 0.85330884, 0.3936621 ],
        [0.20392387, 0.93722227, 0.42586811, 0.99516942],
        [0.28388984, 0.14701266, 0.41383118, 0.4226833 ]])
```

### 2. 矩阵求逆、乘法、转置、迹

`B=A.I` # 求矩阵 A 的逆矩阵，等同于 `A.getI()`

```
matrix([[ -1.99019934,  1.10187596,  0.30889676,  0.14531567],
        [ 0.5589437 ,  0.65373992,  0.73495266, -2.87251001],
        [ 2.28781452, -0.22857687, -1.29279598,  1.07389846],
        [-1.09761469, -0.74364708,  0.80263234,  2.21591205]])
```

`E=A*B` # 矩阵相乘

```
matrix([[ 1.00000000e+00, -4.05719353e-17,  2.99017698e-17,  1.01630458e-16],
        [ 5.82376543e-17,  1.00000000e+00, -5.56746906e-17, -4.49257273e-17],
        [-8.35020425e-17,  4.25491059e-17,  1.00000000e+00,  1.95803985e-17],
        [-3.33998641e-17, -2.46244071e-17,  1.83359155e-17,  1.00000000e+00]])
```

`A.T` # 矩阵的转置，等同于 `A.getT()`。共轭转置则调用 `A.getH()` 或 `A.H`

```
matrix([[0.07711176, 0.9522149 , 0.20392387, 0.28388984],
```

```
[0.52762327, 0.67086255, 0.93722227, 0.14701266],  
[0.56875003, 0.85330884, 0.42586811, 0.41383118],  
[0.40327313, 0.3936621 , 0.99516942, 0.4226833 ]])
```

```
x=A.trace() # 矩阵的迹
```

```
matrix([[1.59652572]])
```

```
int(x)
```

```
1
```

### 3. `matlib.empty()`

`matlib.empty()` 函数返回一个新的矩阵，元素值随机产生，语法格式为：

```
numpy.matlib.empty(shape, dtype, order)
```

- **shape**: 定义新矩阵形状的整数或整数元组
- **Dtype**: 可选，数据类型
- **order**: C（行序优先） 或者 F（列序优先）

实例

```
import numpy as np  
import numpy.matlib  
print (np.matlib.empty((2,2))) # 填充为随机数据
```

某次运行的输出结果为：

```
[[1.24489502e-311 1.24489502e-311]  
 [1.24489502e-311 1.24489502e-311]]
```

### 4. `matlib.zeros()`

`matlib.zeros()` 函数创建一个以 0 填充的矩阵。

实例

```
print(np.matlib.zeros((2, 2)))
```

输出结果为：

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

### 5. `matlib.ones()`

`numpy.matlib.ones()`函数创建一个以 1 填充的矩阵。

## 实例

```
print(np.matlib.ones((2, 2)))
```

输出结果为：

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

## 6. matlib.eye()

`matlib.eye()` 函数返回一个矩阵，对角线元素为 1，其他位置为零。

```
numpy.matlib.eye(n, M, k, dtype)
```

- **n**: 返回矩阵的行数
- **M**: 返回矩阵的列数，默认为 n
- **k**: 对角线的索引
- **dtype**: 数据类型

## 实例

```
print(np.matlib.eye(n=3, M=4, k=0, dtype=float))
```

输出结果为：

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

## 7. matlib.identity()

`matlib.identity()` 函数返回给定大小的单位矩阵，可以看做是 `eye()` 方法的特例。

单位矩阵是个方阵，从左上角到右下角的对角线（称为主对角线）上的元素均为 1，除此以外全都为 0。

$$I_1 = [1], I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \dots, I_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

## 实例

```
print(np.matlib.identity(5, dtype=float)) # 单位矩阵，eye()的特例
```



输出结果为：

```
[[ 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.]]
```

注意：np.zeros()、np.ones()、np.eye()等方法得到的数组而不是 matrix:

np.zeros((2,2))	np.ones((2,2))	np.eye(2,2)	np.empty((2,2))
array([[0., 0.], [0., 0.]])	array([[1., 1.], [1., 1.]])	array([[1., 0.], [0., 1.]])	array([[1., 0.], [0., 1.]])

## 8. matplotlib.rand()和 matplotlib.randn()

numpy.matlib.rand() 函数创建一个给定大小的矩阵，数据是随机填充的，服从均匀分布  $U(0,1)$ 。

numpy.matlib.randn() 函数创建一个给定大小的矩阵，数据是随机填充的，服从  $N(\mu=0, \sigma=1)$  的标准正态分布（高斯分布），即标准正态分布。

实例

```
print(np.matlib.rand(2, 3)) # 服从 U(0,1) 的均匀分布
print(np.matlib.randn(2, 3)) # 服从 N(μ=0,σ=1) 的标准正态分布（高斯分布），即标准正态分布
```

输出结果为：

```
[[0.144228  0.75322429 0.79386176]
 [0.52244889 0.38165559 0.10907001]]

[[ 2.35518613  2.35572584  1.06214054]
 [-0.89680691 -0.59180813 -0.05902507]]
```

## 9. ndarray 对象与 matrix 对象相互转换

矩阵总是二维的，而 ndarray 是一个 n 维数组。两个对象都是可互换的。

实例

```
import numpy as np
A = np.matrix([[1, 2, 3], [4, 5, 6]])
print(A)
B = np.asarray(A) # 等同于 A.getA()
```

```
print(B)
C = np.asmatrix(B)
print(C)
```

输出结果为：

```
[[1 2 3]
 [4 5 6]]
[[1 2 3]
 [4 5 6]]
[[1 2 3]
 [4 5 6]]
```

## 10. ndarray 对象转换为列表

`numpy.tolist()`将数组转换为列表。例如：

```
a = np.array([[1, 2], [3, 4]])
b = a.tolist() # [[1, 2], [3, 4]]
```

a 是数组，而 b 是列表。

## 八、 NumPy 线性代数

numpy 及其 linalg 模块提供了线性代数所需的所有功能。

Please note that the most-used linear algebra functions in NumPy are present in the main ``numpy`` namespace rather than in ``numpy.linalg``. There are: ``dot``, ``vdot``, ``inner``, ``outer``, ``matmul``, ``tensordot``, ``einsum``, ``einsum\_path`` and ``kron``.

### 1. numpy.dot()、 numpy.vdot()、 numpy.inner()和 numpy.matmul()

`numpy.dot()` 对于两个一维的数组，计算的是这两个数组对应下标元素的乘积和(数学上称之为内积)；对于二维数组，计算的是两个数组的矩阵乘积。

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

- **a** : ndarray 数组或列表
- **b** : ndarray 数组或列表
- **out** : ndarray, 可选，用来保存 `dot()` 的计算结果

实例

```
import numpy as np
print(np.dot([1, 2], [3, 4])) # 向量内积
```

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(a)
print(b)
print(np.dot(a, b)) # 矩阵相乘
```

输出结果为：

```
11
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
[[19 22]
 [43 50]]
```

`numpy.vdot()` 函数是两个向量的点积。如果第一个参数是复数，那么它的共轭复数会用于计算。如果参数是多维数组，它会被展开成一维数组。

实例

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.vdot(a, b)) # vdot 将数组展开计算内积
# 等价于
v1=a.reshape((4,)) # 转化为一维数组
v2=b.reshape((4,))
print(v1)
print(v2)
print(np.dot(v1,v2)) # 向量内积
```

输出结果为：

```
70
[1 2 3 4]
[5 6 7 8]
70
```

`numpy.inner()` 函数返回一维数组的向量内积。例如：

```
print(np.inner(np.array([1, 2, 3]), np.array([0, 1, 0]))) # 2
```

`numpy.matmul` 函数返回两个数组的矩阵乘积。例如：

### 实例

```
import numpy as np
a = [[1, 0],
      [0, 1]]
b = [[4, 1],
      [2, 2]]
print(np.matmul(a, b))
A = np.array(a)
B = np.array(b)
print(np.matmul(A, B))
print(A * B)  # 注意，得到的是矩阵对应元素相乘而不是矩阵相乘
```

输出结果为：

```
[[4 1]
 [2 2]]
[[4 1]
 [2 2]]
[[4 0]
 [0 2]]
```

## 2. numpy.trace()、numpy.diag()、numpy.tril()等

numpy.trace ()用于计算矩阵的迹(Trace)； numpy.diag()用于获取矩阵的对角线元素或构造对角矩阵； numpy.tril()用于返回下三角矩阵。

### 实例

```
import numpy as np
A = np.arange(1, 10).reshape((3, 3))
print(A)
print('trace(A): ', np.trace(A))  # 矩阵的迹
print('diag(A): ', np.diag(A))  # 取矩阵对角元素
print(np.tril(A))  # 取矩阵下三角
```

输出结果为：

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
trace(A): 15
diag(A): [1 5 9]
```

```
[[1 0 0]
 [4 5 0]
 [7 8 9]]
```

### 3. 矩阵行列式、秩、逆、特征值分解、奇异值分解等

numpy.linalg 模块用于求矩阵行列式、秩、逆、特征值分解、奇异值分解、解线性方程组等。

- numpy.linalg.det(): 求行列式
- numpy.linalg.matrix\_rank(): 求矩阵的秩
- numpy.linalg.inv(): 求矩阵的逆
- numpy.linalg.solve(): 解线性方程组
- numpy.linalg.eig(): 特征值分解
- numpy.linalg.svd(): 奇异值分解

实例：求矩阵的行列式、秩、逆

```
import numpy as np
A = np.array([[1, 1, 1], [0, 2, 5], [2, 5, -1]])
print(A)
print('matrix_rank(A): ', np.linalg.matrix_rank(A)) # 矩阵的秩
print('det(A): ', np.linalg.det(A)) # 矩阵的行列式
print(np.linalg.inv(A)) # 矩阵逆
print(np.matmul(A, np.linalg.inv(A))) # 验证 A*inv(A)=I
```

输出结果为：

```
[[ 1  1  1]
 [ 0  2  5]
 [ 2  5 -1]]
matrix_rank(A):  3
det(A):  -21.0
[[ 1.29 -0.29 -0.14]
 [-0.48  0.14  0.24]
 [ 0.19  0.14 -0.1 ]]
[[ 1.00e+00 -5.55e-17  0.00e+00]
 [ 0.00e+00  1.00e+00  0.00e+00]
 [-1.11e-16  5.55e-17  1.00e+00]]
```

实例：考虑以下线性方程：

```
x + y + z = 6
2y + 5z = -4
2x + 5y - z = 27
```

可以使用矩阵表示为：

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}$$

已知该方程组的解为  $x = 5, y = 3, z = -2$  。

```
import numpy as np
from numpy import linalg as LA
A = np.array([[1, 1, 1], [0, 2, 5], [2, 5, -1]])
print('矩阵 A: ')
print(A)
print('系数列向量 b: ')
b = np.array([[6, -4, 27]]).T
print(b)
print('调用 LA.solve(A, b): ')
x = LA.solve(A, b)
print(x)
print('计算 A^(-1)b: ')
A_inv = LA.inv(A)
x2 = np.matmul(A_inv, b)
print(x2)
```

输出结果为：

```
矩阵 A:
[[ 1  1  1]
 [ 0  2  5]
 [ 2  5 -1]]
系数列向量 b:
[[ 6]
 [-4]
 [27]]
LA.solve(A, b):
[[ 5.]
 [ 3.]
 [-2.]]
A^(-1)b:
[[ 5.]
 [ 3.]
 [-2.]]
```

numpy.linalg.eig() 对方阵进行特征值分解，返回特征值和右特征向量。语法格式为：

```
d,U=numpy.linalg.eig(A)
```

- **A**：待分解的方阵
- **d**：一维数组，包含矩阵特征值
- **U**：右特征向量构成的矩阵，为正交矩阵或酉矩阵。

实例

```
import numpy as np
from numpy import linalg as LA
np.set_printoptions(2)
A = np.array([[6, 7, 5], [7, 13, 8], [5, 8, 6]]) # 对称正定矩阵，可对角化

print('矩阵 A: \n', A)
d, U = LA.eig(A) # 特征分解
print('矩阵 A 的秩: ', LA.matrix_rank(A))
print('特征值: \n', d)
print('右特征向量构成的矩阵: \n', U)

# 验证:  $A=UDU^H$ 。不可对角化的矩阵不满足!
print('U^H=\n', np.conj(U).T) # 共轭转置
print('UU^H=\n', np.matmul(U, np.conj(U).T))
print("UDU^H=")
D = np.diag(d)
UD = np.matmul(U, D)
print(np.matmul(UD, np.conj(U).T))

#  $AU=DU$  总是满足
test = np.matmul(A, U) - np.matmul(D, U)
print('AU-DU:\n', test)
```

输出结果为：

```
矩阵 A:
[[ 6  7  5]
 [ 7 13  8]
 [ 5  8  6]]
矩阵 A 的秩: 3
特征值:
```

```

[22.67  1.68  0.66]
右特征向量构成的矩阵:
[[-0.46 -0.84 -0.29]
 [-0.74  0.54 -0.4 ]
 [-0.49 -0.03  0.87]]
U^(H)=
[[-0.46 -0.74 -0.49]
 [-0.84  0.54 -0.03]
 [-0.29 -0.4   0.87]]
UU^(H)=
[[ 1.00e+00  4.02e-16 -6.66e-16]
 [ 4.02e-16  1.00e+00  7.22e-16]
 [-6.66e-16  7.22e-16  1.00e+00]]
UDU^(H)=
[[ 6.  7.  5.]
 [ 7. 13.  8.]
 [ 5.  8.  6.]]
AU-DU:
[[ 1.78e-15  1.76e+01  6.45e+00]
 [-1.55e+01 -2.33e-15  4.04e-01]
 [-1.08e+01 -3.54e-02  0.00e+00]]

```

如果矩阵 A 换成：

```
A = np.array([[3, 1, 0], [0, 3, 1], [0, 0, 3]]) # 不可对角化（亏损）矩阵的特征值
```

A 包含重复特征值[3 3 3]，且特征向量线性相关。这意味着 A 不可对角化，因此为亏损矩阵。此时仍然满足  $A^*U=D^*U$ ，但由于 U 不再正交 ( $U^*U^{(H)} \neq I$ )，因此不满足  $A=U^*D^*U^{(H)}$ 。

任意矩阵 A 奇异值分解可表示为  $A=U^*S^*V^H$ ，其中 S 是与 A 同大小的对角矩阵，矩阵的奇异值在 S 的对角线上，U 和 V 是两个酉矩阵。若 A 为  $m \times n$  阵，则 U 为  $m \times m$  阵，V 为  $n \times n$  阵。

`numpy.linalg.svd()` 对任意矩阵进行奇异值分解，返回奇异值和奇异向量，奇异值非负且按降序排列。语法格式为：

```
U, s, Vh=numpy.linalg.svd(A)
```

- A：待分解的矩阵
- s：一维数组，包含矩阵奇异值
- U：左奇异向量构成酉矩阵
- Vh：右奇异向量构成酉矩阵的共轭转置

实例

```

import numpy as np
from numpy import linalg as LA
np.set_printoptions(2)

```



```

A = np.arange(1, 9).reshape(-1, 2) # 4x2 矩阵
print('矩阵 A: \n', A)
U, s, Vh = LA.svd(A) # svd 分解
print('U: \n', U)
print('奇异值 s: \n', s)
print('Vh: \n', Vh)

L = len(s) # 奇异值个数
S = np.zeros(A.shape) # shape 与 A 相同
R = np.diag(s)
S[0:L, 0:L] = R # 构造 S 矩阵
print('S 矩阵: \n', S)

# 验证:  $A=USV^H$ , U 为 m 阶酉矩阵  $UU^H=I_m$ , Vh 为 n 阶酉矩阵  $Vh^H Vh=I_n$ 
print('UU^H=\n', np.matmul(U, np.conj(U).T)) # mxm 的单位阵
print('VV^H=\n', np.matmul(np.conj(Vh).T, Vh)) # nxn 的单位阵
US = np.matmul(U, S)
print("USV^H=\n", np.matmul(US, Vh)) # 验证:  $A=USV^H$ 

```

输出结果为:

```

矩阵 A:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
U:
[[-0.15 -0.82 -0.39 -0.38]
 [-0.35 -0.42  0.24  0.8 ]
 [-0.55 -0.02  0.7  -0.46]
 [-0.74  0.38 -0.55  0.04]]
奇异值 s:
[14.27  0.63]
Vh:
[[-0.64 -0.77]
 [ 0.77 -0.64]]
S 矩阵:
[[14.27  0. ]
 [ 0.    0.63]
 [ 0.    0. ]
 [ 0.    0. ]]

```

```

UU^(H)=
[[ 1.00e+00  1.07e-17  6.13e-17  3.49e-16]
 [ 1.07e-17  1.00e+00 -1.23e-17  3.76e-17]
 [ 6.13e-17 -1.23e-17  1.00e+00  1.68e-16]
 [ 3.49e-16  3.76e-17  1.68e-16  1.00e+00]]

VV^(H)=
[[ 1.00e+00 -9.58e-19]
 [-9.58e-19  1.00e+00]]

USV^(H)=
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]

```

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix}$$

——SVD——>

$$A = U\Sigma V^T = \begin{pmatrix} 1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \\ 2/\sqrt{6} & 0 & -1/\sqrt{3} \\ 1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \end{pmatrix} \begin{pmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}$$

## 九、NumPy IO

与 list 等任意 Python 一样，ndarray 对象可以时使用 pickle 模块写入文件或从文件读取。

例如：用 pickle 模块将 ndarray 对象用写入文件或从文件读取 ndarray 对象。

```

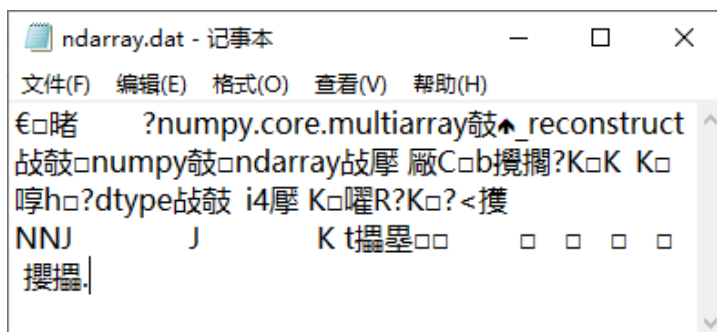
import numpy as np
import pickle

A = np.arange(1, 7).reshape(-1, 3) # 2x3 矩
# 序列化并写入文件
f = open(r'ndarray.dat', 'wb')
pickle.dump(A, f, protocol=0)
f.close()

# 从文件读取并反序列化
f = open(r'ndarray.dat', 'rb')
B = pickle.load(f)
f.close()

print('pickle.load():\n', B)

```



输出结果为：

```

pickle.load():
[[1 2 3]
 [4 5 6]]

```

此外，Numpy 可以读写磁盘上的文本数据或二进制数据，它为 ndarray 对象引入了一个简单的文件格式 npy。npy 文件用于存储与重建 ndarray 所需的数据、图形、dtype 和其他信息。

常用的 IO 函数有：

load() 和 save() 函数是读写文件数组数据的两个主要函数，默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为 .npy 的文件中。

savze() 函数用于将多个数组写入文件，默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为 .npz 的文件中。

loadtxt() 和 savetxt() 函数处理正常的文本文件(.txt 等)

## 1. numpy.save()和 numpy.load()

numpy.save()函数将数组保存到以.npy 为扩展名的文件中。

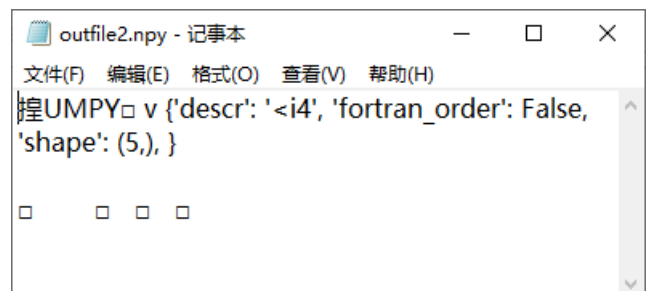
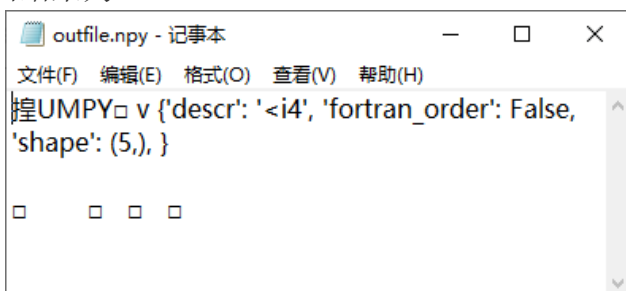
**numpy.save(file, arr, allow\_pickle=True, fix\_imports=True)**

- file: 要保存的文件，扩展名为.npy，如果扩展名.npy 省略则会被自动加上。
- arr: 要保存的数组
- allow\_pickle: 可选，布尔值，允许使用 Python pickles 保存对象数组，Python 中的 pickle 用于在保存到磁盘文件或从磁盘文件读取之前，对对象进行序列化和反序列化。
- fix\_imports: 可选，为了方便 Pyhton2 中读取 Python3 保存的数据。

例如：使用 numpy.save()一维数组保存至.npy 文件。

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
# 保存到 outfile.npy 文件上
np.save('outfile.npy', a)
# 保存到 outfile2.npy 文件上，如果文件路径末尾没有扩展名 .npy，该扩展名会被自动加上
np.save('outfile2', a)
```

输出结果为：



可以看出文件是乱码的，因为它们都是 Numpy 专用的二进制格式后的数据。

使用 load() 函数来读取数据就可以正常显示了。如果写入的是一个数组，load()返回一个数组；如

果写入的是多个数组（用 `savez()` 保存），则返回的是一个 `NpzFile` 对象，该对象保存了个数组对象并有一个默认的名字 `arr_0`、`arr_1`、...，也可是在 `savez()` 写入数组时给各个数组取一个名字。

例如：使用 `numpy.load()` 从 `.npy` 文件读取一维数组。

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
# 保存到 outfile.npy 文件上
np.save('outfile.npy', a)
# 保存到 outfile2.npy 文件上，如果文件路径末尾没有扩展名 .npy，该扩展名会被自动加上
np.save('outfile2', a)
```

输出结果为：

```
[1 2 3 4 5]
```

## 2. `np.savez()`

`numpy.savez()` 函数将多个数组保存到以 `npz` 为扩展名的文件中。

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

- **file**: 要保存的文件，扩展名为 `.npz`，未加扩展名 `.npz` 则会被自动加上。
- **args**: 要保存的数组，可以使用关键字参数为数组起一个名字，非关键字参数传递的数组会自动起名为 `arr_0`, `arr_1`, ...。
- **kwargs**: 要保存的数组使用关键字名称。

例如：使用 `numpy.savez()` 将三个数组写入 `.npz` 文件，然后读取还原。

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.arange(0, 1.0, 0.1)
c = np.sin(b)
# c 使用了关键字参数 sin_array
np.savez("a.npz", a, b, sin_array=c) # 数组保存到文件

r = np.load("a.npz") # 从文件读取数组
print(r.files) # files 属性中保存了个数组对应的关键字
# 查看各个数组名称
print(r["arr_0"]) # 数组 a
```

```
print(r["arr_1"]) # 数组 b
print(r["sin_array"]) # 数组 c
```

输出结果为：

```
['sin_array', 'arr_0', 'arr_1']
[[1 2 3]
 [4 5 6]]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
[0.          0.09983342 0.19866933 0.29552021 0.38941834 0.47942554
 0.56464247 0.64421769 0.71735609 0.78332691]
```

### 3. savetxt()和 loadtxt()

savetxt() 函数是以简单的文本文件格式存储数据，对应的使用 loadtxt() 函数来获取数据。

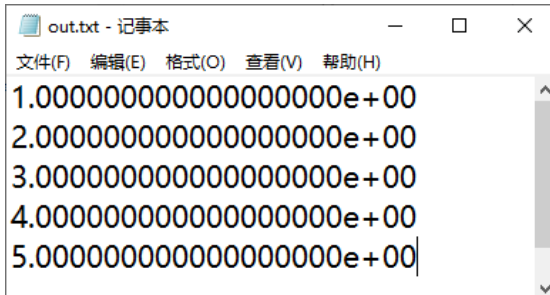
```
np.savetxt(fname, X, fmt='%.18e', delimiter=' ')
np.loadtxt(fname, dtype=float, delimiter=' ', skiprows=0, usecols=None)
```

参数 delimiter 可以指定各种分隔符（注意，**只能是单个字符**），默认是空格；fmt 指定写入格式，'%d'、'%.5f'、'%.4e'等。

参数 skiprows 指定跳过的行数，参数 usecols 指定读取那些列，是一个列索引列表，例如[0,2,3]表示读取索引号为 0，2 和 3 三列。

例如

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
np.savetxt('out.txt', a)
b = np.loadtxt('out.txt')
print(b)
```

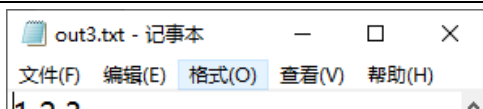
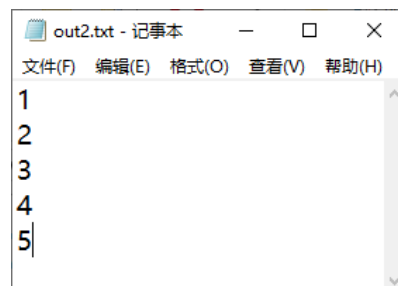


输出结果为：

```
[1. 2. 3. 4. 5.]
```

实例：指定 delimiter 参数、fmt 参数和 dtype 参数。

```
# 写入和读取一维数组
a = np.array([1, 2, 3, 4, 5])
np.savetxt('out2.txt', a, fmt='%d', delimiter=',')
b = np.loadtxt('out2.txt', dtype=int, delimiter=',')
print(b)
# 写入和读取二维数组
```



```
A = np.arange(1, 7).reshape(-1, 3)
np.savetxt('out3.txt', A, fmt='%d', delimiter=',')
B = np.loadtxt('out3.txt', dtype=int, delimiter=',')
print(B)
```

输出结果为：

```
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
```

NumPy 通常与 SciPy（Scientific Python）和 Matplotlib（绘图库）一起使用，这种组合广泛用于替代 MATLAB，是一个强大的科学计算环境，有助于我们通过 Python 学习数据科学或者机器学习。