1. Numpy库简介

1.1 概念

NumPy(Numerical Python)是一个开源的Python科学计算库，旨在为Python提供高性能的多维数组对象和一系列工具。NumPy数组是Python数据分析的基础，许多其他的数据处理库（如Pandas、SciPy）都依赖于NumPy。以下是NumPy的一些主要特点：

1. 高性能：NumPy底层主要采用C语言编写，相比于python来说运算速度快，性能优越，并且保留了python的易用性。
2. 多维数组：NumPy提供了强大的n维数组对象ndarray，可进行高效的数据处理，这是Numpy进行数据处理的**核心对象。**
3. 丰富的函数：NumPy内置了大量数学、统计和线性代数函数，方便进行数据计算，除此之外还具有广播功能，可以允许不同形状的数组进行算术运算。
4. 广泛的接口：NumPy与许多其他科学计算库（如Matplotlib、SciPy）兼容，可轻松实现数据交换和集成，此外，在人工智能领域中也可以很方便的和神经网络中使用的张量进行结构转换。

1.2 安装

Numpy库是一个第三方库，因此在使用时需要我们提前安装，安装的命令是：

```
pip install numpy
```

2. Ndarray与list的区别

在Python中具有list这样的数据类型，它也可以非常灵活的处理列表中具有的多个元素，为什么不用list而是用Ndarray?

是因为Python中的list虽然可以灵活的处理多个元素，但它的效率很低，一般情况下的科学运算的数据量是非常庞大的，所以list的效率低会导致整个科学运算的过程变得非常慢。与之相比，Ndarray数组具有以下特点：

1. Ndarray数组所有元素的数据类型相同、数据地址连续，批量操作数组元素时速度更快，而list中元素的数据类型可能不同，需要通过寻址的方式找到下一个元素。

2. Ndarray数组支持广播机制，矩阵运算时不需要写for循环。
3. 底层主要使用C语言实现，运行速度远高于Python代码。

下面举两个例子来体会一下Ndarray数组和list的差异：

1. 实现a+1的运算

使用python中的list完成对列表中的每一个元素进行加1的操作：

```
# 使用python中的list
a = [1, 2, 3, 4, 5]

# python语法不支持直接对列表进行加1操作，这样会报错
# a = a + 1

# 使用循环
for i in range(len(a)):
    a[i] += 1

print(a)
```

使用Ndarray数组完成对数组中的每一个元素进行加1的操作：

```
import numpy as np

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

# Numpy数组可以直接进行加1操作，它会令数组中的所有的元素都去进行加1
arr = arr + 1

print(arr)
```

2. 实现a+b的运算

使用list完成两个列表元素的算术运算：

```
# 使用list完成两个列表元素的加法
a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]

# python中支持两个列表直接相加，但相加的结果是两个列表拼接的结果
# 不是我们想要的对应元素进行数值运算的结果
# c = a + b
```

```
# print(c)

# 只能是通过遍历的形式去拿到每个元素再去进行算术运算
c = []
for i in range(len(a)):
    c.append(a[i] + b[i])
print(c)
```

使用Ndarray数组完成两个数组元素的算术运算：

```
import numpy as np

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

# Ndarray数组可以通过两个数组变量之间的运算直接对数组中的元素进行算术运算
c = a + b
print(c)
```

通过上面的两个案例可以看出，在不写for循环的情况下，Ndarray数组就可以非常方便的完成数学计算。在编写矢量或者矩阵的程序时，可以像编写普通数值一样，使得代码极其简洁。

3. Ndarray数组的创建

3.1 array函数

创建Ndarray数组最简单的方式就是使用array函数，它接受一切序列型的对象，然后产生一个新的含有传入数据的Numpy数组。

函数原型：

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

1. **object**:可以是列表、元组、数组等可迭代对象，表示要转换为数组的数据。
2. **dtype**:可以指定数组的数据类型，如果未指定，则根据输入数据自动推断。
3. **copy**:如果为True，则复制输入数据；如果为False，则只有必要时才复制，这意味着如果 object 已经是一个 NumPy 数组并且满足其他条件（如 dtype、order 等），则不会进行复制。默认为True。

4. **order**:指定数组在内存中的存储顺序，可以是‘C’（C语言风格,行优先）、‘F’（Fortran风格,列优先）或‘K’（尽可能与输入数据的顺序一致）。默认为‘K’。
5. **subok**:如果为True，则返回的数组将是输入对象的子类；如果为False，则返回的数组将始终是基类数组。默认值为False。
6. **ndmin**:指定数组的最小维度。如果输入数据的维度小于ndmin，则 NumPy 会自动在前面添加维度，以使数组的维度至少为ndmin。

附：数据类型

类型	描述
np.int8	8 位有符号整数
np.int16	16 位有符号整数
np.int32	32 位有符号整数
np.int64	64 位有符号整数
np.uint8	8 位无符号整数
np.uint16	16 位无符号整数
np.uint32	32 位无符号整数
np.uint64	64 位无符号整数
np.float16	16 位半精度浮点数
np.float32	32 位单精度浮点数
np.float64	64 位双精度浮点数
np.complex64	由两个 32 位单精度浮点数表示的复数
np.complex128	由两个 64 位双精度浮点数表示的复数
np.bool_	用于存储布尔值（True 或 False）
np.string_	固定长度的字符串类型，字节串
np.unicode_	固定长度的 Unicode 字符串类型
np.object_	可以存储任意 Python 对象的类型，但会降低数组操作的性能，不推荐使用

> 不同的数据类型在内存占用和数值范围上有所不同，选择合适的数据类型可以提高内存使用效率和计算性能。例如，如果数据范围较小，可以选择较小的整数类型；如果需要高精度的浮点数计算，可以选择双精度浮点数类型。

附：copy参数说明

如果 `copy` 设置为 `True`，则在创建新数组时，无论是否需要，都会复制原始数据。这意味着新数组和原始数组在内存中是完全独立的，对其中一个数组的修改不会影响另一个数组。
如果 `copy` 设置为 `False`，则 `NumPy` 尝试避免不必要的数据复制。但是当源数组的数据类型（`dtype`）与通过 `dtype` 参数指定的数据类型不匹配，需要进行类型转换时，就必须复制数据。

附：order参数说明

在NumPy中，“行优先”（row-major）和"列优先"（column-major）是描述多维数组（如矩阵）在内存中如何存储数据的方式。
行优先（Row-major）：
在行优先的存储顺序中，数组的行是连续存储在内存中的。
这意味着，当你按顺序遍历数组时，首先遍历第一行的所有元素，然后是第二行，依此类推。
在C语言中，默认的数组存储方式是行优先。
列优先（Column-major）：
在列优先的存储顺序中，数组的列是连续存储在内存中的。
这意味着，当你按顺序遍历数组时，首先遍历第一列的所有元素，然后是第二列，依此类推。
在Fortran语言中，默认的数组存储方式是列优先。
NumPy默认使用行优先的存储方式，这与C语言中的数组存储方式一致。但是，NumPy也提供了方法来改变存储顺序，或者从列优先的数据源（如Fortran程序或MATLAB生成的数据）中读取数据。
假设有一个2x3的矩阵：

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

在行优先存储中，内存中的数据顺序是：1, 2, 3, 4, 5, 6
在列优先存储中，内存中的数据顺序是：1, 4, 2, 5, 3, 6
行优先和列优先存储顺序的不同会影响切片和索引操作的性能。例如，在行优先的数组中，连续地访问行元素通常会比访问列元素更快，因为在内存中它们是连续存储的。相反，在列优先的数组中，连续地访问列元素会更快。

附：subok参数说明

如果 subok 设置为 True，则当输入数据是一个 NumPy 数组的子类时，返回的数组将保留这个子类的类型。这意味着，如果你有一个自定义的数组子类，并且你想保留这个子类的特性，你可以设置 subok=True 来确保返回的数组仍然是这个子类的实例。
如果 subok 设置为 False（默认值），即使输入数据是一个数组子类，返回的数组也会被强制转换为基类数组，即标准的 numpy.ndarray 类型。这会丢失子类的任何特殊行为或属性。

```
# 创建一维数组
import numpy as np


# 可以直接使用array函数进行创建
arr1 = np.array([1, 2, 3, 4, 5])
print(type(arr1))
print(arr1)

# 也可以将列表、元组等数据转化为Ndarray数组
ls1 = [1, 2, 3, 4, 5]
arr2 = np.array(ls1)
print(type(ls1))
print(type(arr2))
print(ls1)
print(arr2)
```

```
# 创建二维数组
import numpy as np

# 可以直接使用array函数进行创建
arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(type(arr1))
print(arr1)
```

```
# 创建三维数组
import numpy as np

# 可以直接使用array函数进行创建
arr1 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]])

print(type(arr1))
print(arr1)
```

3.2 arange函数

该函数的作用与Python自带的range()类似，其函数原型为：

```
numpy.arange([start], stop[, step,], dtype=None)
```

1. `start`：可选参数，表示数组的起始值。默认值为 0。
2. `stop`：必须参数，表示数组的结束值（不包含该值）。
3. `step`：可选参数，表示相邻两个值的差值。默认值为 1。
4. `dtype`：可选参数，表示返回数组的数据类型。如果没有指定，则根据其他参数推断数据类型。

该函数会返回一个一维数组（ndarray），其中包含了从 `start` 到 `stop`（不包括 `stop`），以 `step` 为间隔的一系列值。

```
import numpy as np

# 创建一个从0到9的数组
arr1 = np.arange(10)
print(arr1)

# 创建一个从5到14的数组，步长为2
arr2 = np.arange(5, 15, 2)
print(arr2)

# 创建一个从0到1的数组，包含10个值（步长为0.1）
arr3 = np.arange(0, 1, 0.1)
print(arr3)

# 指定数据类型
arr4 = np.arange(10, dtype=np.float32)
```

```
print(arr4)
```

3.3 zeros函数

该函数的作用是创建指定长度或者形状的全0数组，函数原型为：

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

- `shape`：一个整数或整数元组，用于指定输出数组的形状。
- `dtype`：可选参数，指定数组元素的数据类型。默认为 `float`。
- `order`：可选参数，指定数组数据在内存中的存储顺序。'C' 表示按行（C语言风格），'F' 表示按列（Fortran风格）。

```
# 创建一维全0数组
import numpy as np

arr1 = np.zeros(5)

print(arr1)
```

```
# 创建二维全0数组

import numpy as np

arr1 = np.zeros((3, 2))

print(arr1)
```

```
# 创建三维全0数组

import numpy as np

arr1 = np.zeros((3, 2, 4))

print(arr1)
```

3.4 ones函数

该函数的作用是创建指定长度或者形状的全1数组，函数原型为：

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

- `shape`：一个整数或整数元组，用于指定输出数组的形状。
- `dtype`：可选参数，指定数组元素的数据类型。默认为 `float`。
- `order`：可选参数，指定数组数据在内存中的存储顺序。'C' 表示按行（C语言风格），'F' 表示按列（Fortran风格）。

```
# 创建一维全1数组
import numpy as np

arr1 = np.ones(5)

print(arr1)
```

```
# 创建二维全1数组

import numpy as np

arr1 = np.ones((3, 2))

print(arr1)
```

```
# 创建三维全1数组

import numpy as np

arr1 = np.ones((3, 2, 4))

print(arr1)
```


3.5 empty函数

该函数的作用就是创建一个指定形状、数据类型且未初始化的数组，函数原型是：

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

- `shape`：整数或整数元组，用于指定输出数组的形状。
- `dtype`：可选，指定数组元素的数据类型。默认是 `float`。
- `order`：可选，指定数组数据在内存中的存储顺序。'C' 表示按行优先顺序；'F' 表示按列优先顺序。

需要注意的是：

1. 由于 `numpy.empty` 不初始化数组元素，因此它比 `numpy.zeros` 或 `numpy.ones` 更快，但数组中的值是未定义的，可能包含任何值。
2. 在使用 `numpy.empty` 创建数组后，应立即用有效值填充数组，以避免使用未定义的数据。

```
import numpy as np

# 创建一个形状为 (2, 3) 的空数组
x = np.empty((2, 3))
print(x)

# 创建一个形状为 (2, 3) 的空数组，数据类型为整型
y = np.empty((2, 3), dtype=int)
print(y)
```

3.6 full函数

该函数的作用是创建指定形状、指定元素的数组，函数原型为：

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

- `shape`：整数或整数元组，用于指定输出数组的形状。
- `fill_value`：用于填充数组的值。
- `dtype`：可选，指定数组元素的数据类型。如果未指定，则从 `fill_value` 推断。
- `order`：可选，指定数组数据在内存中的存储顺序。'C' 表示按行优先顺序；'F' 表示按列优先顺序。

需要注意的是：

1. `numpy.full` 与 `numpy.zeros` 和 `numpy.ones` 类似，但允许指定任何填充值，而不仅仅是 0 或 1。
2. 如果 `fill_value` 不能被转换成指定的 `dtype`，则会引发错误。

```
import numpy as np

# 创建一个形状为 (2, 3) 的数组，所有元素初始化为 7
x = np.full((2, 3), 7)
print(x)

# 创建一个形状为 (2, 3) 的数组，所有元素初始化为 5.5，数据类型为 float
y = np.full((2, 3), 5.5, dtype=float)
print(y)
```

4. Ndarray数组的索引和切片

Ndarray数组中的元素是可以被修改的，如果需要访问或者修改Ndarray数组某个位置的元素，则需要使用Ndarray数组的索引来完成；如果需要访问或者修改一些区域的元素，则需要使用Ndarray数组的切片。

4.1 一维数组的索引与切片

4.1.1 索引

一维数组的索引方式与Python列表的索引方式类似，Ndarray数组使用方括号`[]`进行索引，索引值从左向右从0开始，从右向左从-1开始。

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

4.1.2 切片

与Python列表的切片方式类似，使用冒号`:`进行切片，格式为`start:stop:step`，其中start、stop、step可根据情况省略。

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

arr1 = arr[4:7]
print('arr1数组元素为', arr1)
```

```
arr2 = arr[1:6:2]
print('arr2数组元素为', arr2)

# 将一个标量值赋值给一个切片时，该值会自动传播到整个选区。
arr[4:7] = 6
print('arr数组元素为：', arr)

# 也可以使用:代表全部元素
arr[:] = 10
print('arr数组元素为：', arr)
```

4.2 多维数组的索引与切片

4.2.1 索引

对于多维数组，可以通过逗号分隔的索引来访问特定元素，也可以使用连续的[]来访问特定元素，例如，对于一个二维数组，第一个索引表示行，第二个索引表示列。

```
import numpy as np

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

print('arr数组为：\n', arr)

# 只有一个索引指标时，会在第0维上索引，后面的维度保持不变
print('arr[0]为：', arr[0])

# 两个索引指标
print('arr[0][0]为：', arr[0][0])

# 两个索引指标
print('arr[0, 1]为：', arr[0, 1])
```

4.2.2 切片

多维数组切片时，可以分别为每个维度指定切片。对于一个二维数组`arr`，切片操作的语法如下：

```
arr[row_slice, column_slice]
```

其中，`row_slice`用于选择行的范围，`column_slice`用于选择列的范围，如果不切列，列可以省略，如果不切行，需要使用`:`代替。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# 选择特定的一行
row1 = arr[1]
print(row1)

# 选择连续的多行
rows = arr[1:3]
print(rows)

# 选择不连续的多行
rows = arr[[0, 2]]
print(rows)

# 选择特定的一列
col1 = arr[:, 1]
print(col1)

# 选择连续的多列
cols = arr[:, 1:3]
print(cols)

# 选择不连续的多列
cols = arr[:, [0, 2]]
print(cols)

# 同时选择行和列
subset = arr[1:3, 1:3]
print(subset)
```


5. Ndarray数组的属性

Ndarray数组有很多的属性，但我们一般最关注的属性有以下几个：

5.1 shape

通过调用Ndarray.shape即可返回一个表示数组维度大小的元组。例如，对于一个 2 行 3 列的数组，`shape` 将返回 `(2, 3)`。

```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.shape)
```

5.2 dtype

通过调用Ndarray.dtype即可返回一个Ndarray数组中元素的数据类型。

```
import numpy as np

# 可以在创建的时候去指定数据类型
arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.dtype)
```

5.3 size

该属性是指数组中包含的元素个数，其大小等于调用shape函数返回的元组中的所有元素的乘积。

```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.shape)
print(arr1.size)
```

5.4 ndim

该属性是指数组的维度大小，其大小等于调用shape函数返回的元组中的元素的个数。

```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.shape)
print(arr1.ndim)
```

6. 改变Ndarray数组的数据类型和形状

Ndarray数组创建之后，数组的形状和数据类型是可以进行改变的。

6.1 数据类型的修改

函数原型：

```
numpy.ndarray.astype(dtype, order='K', casting='unsafe',
subok=True, copy=True)
```

- `dtype`：表示新数据类型的对象。这可以是 NumPy 的 dtype 对象，也可以是 Python 的数据类型，例如 `int` 或 `float`。
- `order`：一个字符，指定结果的内存布局。`'C'` 表示按行（C风格），`'F'` 表示按列（Fortran风格），`'A'` 或 `'K'` 表示在内存中的顺序与原数组保持一致。
- `casting`：控制数据类型转换的安全性。它可以是 `'no'`、`'equiv'`、`'safe'`、`'same_kind'` 或 `'unsafe'` 之一。
- `subok`：如果为 `True`，则子类将被传递，否则返回的数组将被强制转换为基类数组。
- `copy`：布尔值，指定是否复制数据。

> 需要注意的是，默认情况下，该函数会返回一个新的Ndarray数组，且该数组与原数组没有任何关系。

附：casting参数说明

```
casting参数说明：
    'no'：表示根本不应该进行转换数据类型。
    'equiv'：允许数值上等价的类型转换，即转换前后数据的位表示相同。这意味着转换不会导致数据丢失。
```

'safe'：允许安全的类型转换，即转换过程中不会丢失信息。

'same_kind'：允许相同数据类型类别内的转换。例如，允许整型和整型之间、浮点型和浮点型之间的转换，但是不允许整型和浮点型之间的转换。

'unsafe'：允许任何类型的转换，不考虑是否会导致数据丢失或改变。这是最不安全的选项，因为它可能会静默地丢弃数据。

比如：

从 int64 到 int32 的转换：

'no'：不允许。

'equiv'：不允许。

'safe'：不允许。

'same_kind'：允许。

'unsafe'：允许。

从 float64 到 int32 的转换：

'no'：不允许，因为会丢失小数部分。

'equiv'、'safe'、'same_kind'：不允许，因为这不是数值上等价或安全的转换。

'unsafe'：允许，但会丢失小数部分。

```
import numpy as np

# 创建一个浮点数数组
arr = np.array([1.1, 2.2, 3.3])

# 使用 astype 方法将数组转换为整数类型
new_arr = arr.astype(np.int32)

print("Original array:", arr)
print("old type:", arr.dtype)
print("New array:", new_arr)
print("new type:", new_arr.dtype)
```


6.2 形状的修改

6.2.1 reshape

reshape方法用于给数组一个新的形状而不改变其数据，它返回一个新的数组，但是如果给定的形状与原始数组的数据不兼容，会抛出异常，函数原型为：

```
numpy.ndarray.reshape(newshape, order='C')
```

- `newshape`：整数或整数元组，新的形状应该与原始数组中的元素数量相匹配。
- `order`：可选参数，'C' 表示按行（C-style），'F' 表示按列（Fortran-style），'A' 表示原数组内存顺序，'K' 表示元素在内存中的出现顺序。

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


print(b)
print('shape--b',b.shape)
```

6.2.2 resize

resize 方法用于改变数组的大小，与 reshape类似，但它会直接修改调用它的原始数组。如果新形状大于原始形状，则会在数组的末尾添加新的元素，这些元素的值未定义；如果新形状小于原始形状，则会截断数组。函数原型为：

```
numpy.ndarray.resize(newshape)
```

- `newshape`：整数或整数元组，新的形状。

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

a.resize((3, 2))
print('after:', a.shape)

a.resize((2, 2))
print(a)

a.resize((5, 5))
print(a)
```

6.2.3 flatten

flatten方法返回一个一维数组，它是原始数组的拷贝，它默认按行顺序展平数组，但可以通过参数 `order` 来指定展平顺序。函数原型为：

```
numpy.ndarray.flatten(order='C')
```

- `order`：可选参数，'C' 表示按行，'F' 表示按列，'A' 或 'K' 表示与原数组相同的顺序。

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

6.2.4 ravel

ravel方法返回一个连续的数组，它尝试以最低的复制操作来返回展平后的数组，函数原型为：

```
numpy.ndarray.ravel(order='C')
```

- `order`：可选参数，'C' 表示按行，'F' 表示按列。

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

ravel和flatten的区别：

- `flatten` 方法返回的是原数组的副本。这意味着返回的新数组与原数组是两个独立的对象，对新数组的修改不会影响原数组。
- `ravel` 方法返回的是原数组的视图（view）或副本（copy），这取决于数组的顺序。如果数组是连续的（C-style，行优先），则 `ravel` 返回的是视图，这意味着对返回数组的修改会影响到原数组。如果数组不是连续的，则 `ravel` 返回的是副本。可以通过传递 `order='F'` 参数来请求列优先的视图。

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
b = a.flatten()
c = a.ravel()
d = a.ravel(order='F')

a[0][0] = 100
print('a', a)
print('b', b)
print('c', c)
print('d', d)
```

6.2.5 T

`Ndarray` 对象的 `T` 属性是一个特殊的属性，它返回数组的转置（transpose）。

转置是一个操作，它将数组的维度进行交换。对于二维数组，这意味着行变为列，列变为行。对于更高维度的数组，转置操作涉及到交换数组的轴。

```
import numpy as np

# 创建一个二维数组
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])

# 获取转置
transposed_arr = arr_2d.T

print("Original array:")
print(arr_2d)
print("Transposed array:")
print(transposed_arr)
```

7. 随机Ndarray数组的创建

Ndarray数组在创建时除了可以使用上面章节的函数创建指定元素的数组之外，还可以创建具有随机元素的数组。

在创建之前，需要了解一个概念：随机数种子。

随机数种子（random seed）是一个用于初始化随机数生成器（random number generator, RNG）的值。在计算机科学中，大多数的随机数生成器实际上是伪随机数生成器（pseudo-random number generators, PRNGs），它们通过一个算法来生成一系列看似随机的数字。伪随机数生成器的特点是可以重现生成的随机数序列，这是通过设置相同的随机数种子来实现的。

以下是关于随机数种子的几个要点：

1. **可重现性**：通过设置相同的随机数种子，每次程序运行时生成的随机数序列都是相同的。这对于调试程序、进行科学计算或模拟时保持实验结果的一致性非常有用。
2. **算法确定性**：伪随机数生成器是确定性的，这意味着给定的种子会总是产生相同的随机数序列，这与真正的随机数生成器不同，后者总会生成不同的随机数。
3. **种子来源**：随机数种子的值可以是任意的。在许多编程环境中，如果不显式设置种子，通常会使用当前时间作为种子，这样每次程序运行时都会产生不同的随机数序列。
4. **跨平台差异**：不同的操作系统或硬件平台可能会产生不同的随机数序列，即使种子相同。这是因为不同的平台可能有不同的PRNG算法。

在Numpy库中，可以使用numpy.random.seed()函数来设置随机数种子。

```
import numpy as np

# 设置随机数种子，种子值可以是任何整数，通常是正整数。
np.random.seed(42)
```

7.1 random.rand

该函数的作用是返回一个或一组服从“0到1”均匀分布的随机样本，函数原型为：

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

- `d0, d1, ..., dn`：这些参数定义了输出数组的形状。它们是整数，指定了每个维度的大小。例如，`d0` 是第一个维度的大小，`d1` 是第二个维度的大小，依此类推。

该函数会返回一个形状为 `(d0, d1, ..., dn)` 的数组，其中的元素是在 [0.0, 1.0) 区间内均匀分布的随机浮点数。

```
# 生成一个0~1之间的随机数
import numpy as np
```

```
# # 设置随机数种子
# np.random.seed(10)

arr1 = np.random.rand()
print('arr1:', arr1)

# 生成1个一行三列的随机数组
import numpy as np
arr2 = np.random.rand(3)
print('arr2:', arr2)

# 生成一个3行2列的随机数组
import numpy as np
arr3 = np.random.rand(3, 2)
print('arr3:', arr3)
```

7.2 random.random

与random.rand函数的功能类似，但参数不太一样，其函数原型为：

```
numpy.random.random(size=None)
```

- `size`：这是一个可选参数，用于指定输出数组的形状。它可以是一个整数，也可以是一个元组。如果 `size` 是一个整数，则返回一个一维数组；如果 `size` 是一个元组，则返回一个多维数组，其形状与元组指定的一致。

该函数会返回一个形状为 `size` 的数组，其中的元素是在 [0.0, 1.0) 区间内均匀分布的随机浮点数。

```
import numpy as np

# # 设置随机数种子
# np.random.seed(10)

# 生成一个一维数组，包含5个随机数
array_1d = np.random.random(5)
print(array_1d)

# 生成一个二维数组，形状为 (2, 3)
array_2d = np.random.random((2, 3))
print(array_2d)
```

与random.rand函数的区别：

- `numpy.random.rand(d0, d1, ..., dn)` 需要一个或多个整数参数，这些参数定义了返回数组的形状。例如，`np.random.rand(2, 3)` 会返回一个 2x3 的二维数组。
- `numpy.random.random(size=None)` 接受一个可选的 `size` 参数，这个参数可以是整数或者元组。如果 `size` 是一个整数，它会返回一个一维数组；如果 `size` 是一个元组，它会返回一个多维数组。例如，`np.random.random((2, 3))` 也会返回一个 2x3 的二维数组。如果没有提供 `size` 参数，`numpy.random.random` 会返回一个单一的随机浮点数。

7.3 random.randn

该函数用于从标准正态分布中抽取样本。这意味着它返回的随机数将具有平均值（mean）为 0 和标准差（standard deviation）为 1 的正态分布。函数原型为：

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

- `d0, d1, ..., dn`：这些参数指定了输出数组的维度。如果你只提供一个数字，它将返回一个一维数组；如果你提供多个数字，它将返回一个多维数组，其中每个维度的大小由对应的参数指定。

```
import numpy as np


# # 设置随机数种子
# np.random.seed(10)
```

```
# 返回一个从标准正态分布中抽取的单个随机数
print(np.random.randn())

# 返回一个一维数组，包含5个从标准正态分布中抽取的随机数
print(np.random.randn(5))

# 返回一个二维数组，形状为2x3，元素从标准正态分布中抽取
print(np.random.randn(2, 3))
```

7.4 random.normal

该函数用于从具有指定平均值（mean）和标准差（standard deviation）的正态分布（也称为高斯分布）中抽取样本。函数原型为：

```
numpy.random.normal(loc=0.0, scale=1.0, size=None)
```

- `loc`：正态分布的均值，对应于分布的中心位置，默认值为 0.0。
- `scale`：正态分布的标准差，对应于分布的宽度，默认值为 1.0。
- `size`：输出数组的形状。如果 `size` 是一个整数，则返回一个一维数组；如果 `size` 是一个元组，则返回一个多维数组。默认值为 `None`，此时返回一个单个的随机数。

```
import numpy as np

# # 设置随机数种子
# np.random.seed(10)

# 返回一个均值为0.0，标准差为1.0的正态分布中的单个随机数
print(np.random.normal())

# 返回一个均值为0.0，标准差为1.0的正态分布中的5个随机数
print(np.random.normal(size=5))

# 返回一个均值为5.0，标准差为2.0的正态分布中的5个随机数
print(np.random.normal(loc=5, scale=2, size=5))

# 返回一个2x3数组，其元素来自均值为10.0，标准差为3.0的正态分布
print(np.random.normal(loc=10, scale=3, size=(2, 3)))
```

7.5 random.randint

该函数用于生成随机整数，这些整数在指定的范围内均匀分布。函数原型为：

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

- `low`：生成随机数的起始点（包含）。如果只提供了`low`参数而没有提供`high`参数，那么随机整数的范围将是从 0 到`low`（不包含`low`本身）。
- `high`（可选）：生成随机数的结束点（不包含）。
- `size`（可选）：定义输出数组形状的整数或元组。例如，`size=(m,n)`将生成一个`m`行`n`列的数组。
- `dtype`（可选）：指定返回数组的数据类型，默认为int。

需要注意的是：

- 如果只提供了`low`参数而没有提供`high`参数，那么随机整数的范围将是从 0 到`low`（不包含`low`本身）。
- 如果同时提供了`low`和`high`参数，那么随机整数的范围将是从`low`到`high`（不包含`high`本身）。

```
import numpy as np

# # 设置随机数种子
# np.random.seed(10)

# 从 0 到 10（不包含）之间随机生成一个整数
print(np.random.randint(10))

# 从 1 到 10（不包含）之间随机生成一个整数
print(np.random.randint(1, 10))

# 从 1 到 10（不包含）之间随机生成一个 3x3 的整数数组
print(np.random.randint(1, 10, size=(3, 3)))

# 从 1 到 10（不包含）之间随机生成一个 3x3 的整数数组，数据类型为 'int32'
print(np.random.randint(1, 10, size=(3, 3), dtype='int32'))
```

7.6 random.uniform

该函数用于从均匀分布中抽取一个浮点数。函数原型为：

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

- `low`：浮点数或类似浮点数的数组，表示样本抽取区间的下限，默认值为 0。
- `high`：浮点数或类似浮点数的数组，表示样本抽取区间的上限，默认值为 1。
- `size`：整数或元组，可选参数，表示输出的形状。如果未提供，则返回单个浮点数。如果提供了一个整数，则返回一个一维数组；如果提供了一个元组，则返回一个多维数组。

```python
import numpy as np

# # 设置随机数种子
# np.random.seed(10)

# 在 [0, 1) 范围内抽取一个浮点数
print(np.random.uniform())

# 在 [5, 10) 范围内抽取一个浮点数
print(np.random.uniform(5, 10))

# 在 [0, 1) 范围内抽取一个 3x3 的浮点数数组
print(np.random.uniform(size=(3, 3)))

# 在 [5, 10) 范围内抽取一个 2x3 的浮点数数组
print(np.random.uniform(5, 10, size=(2, 3)))
```

7.7 random.shuffle

该函数是用来随机打乱数组元素的顺序的，函数原型为：

```
numpy.random.shuffle(x)
```

- `x`：要打乱的数组。

需要注意的是：

1. `shuffle`函数只适用于一维数组。
2. `shuffle`函数是在原数组上进行操作，不会创建一个新的数组。

```
import numpy as np

# # 设置随机数种子
# np.random.seed(10)

# 创建一个numpy数组
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# 打乱数组
np.random.shuffle(arr)

print(arr)
```

8. Ndarray数组的运算

8.1 标量和数组的运算

在Numpy中，标量和数组的运算内部其实是依靠广播机制进行的。

8.1.1 加法运算

```
import numpy as np

# 生成一个2行3列的Ndarray数组
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 数组加上标量，会使数组的每一个元素都加上该标量得到一个新数组
b = arr + 2

print('original: \n', arr)
print('new: \n', b)
```

8.1.2 减法运算

```
import numpy as np

# 生成一个2行3列的Ndarray数组
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 数组减去标量，会使数组的每一个元素都减去该标量得到一个新数组
b = arr - 2

print('original: \n', arr)
print('new: \n', b)
```

8.1.3 乘法运算

```
import numpy as np

# 生成一个2行3列的Ndarray数组
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 数组乘以标量，会使数组的每一个元素都乘以该标量得到一个新数组
b = arr * 2

print('original: \n', arr)
print('new: \n', b)
```

8.1.4 除法运算

```
import numpy as np

# 生成一个2行3列的Ndarray数组
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 数组除以标量，会使数组的每一个元素都除以该标量得到一个新数组
b = arr / 2

print('original: \n', arr)
print('new: \n', b)
```

8.2 数组和数组的运算

这里分为直接使用符号进行运算和使用Numpy库的函数进行运算。

8.2.1 加法运算

1. 使用符号运算

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# 数组加上数组，会使对应位置的元素相加
arr3 = arr1 + arr2

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)
```

2. 使用函数运算

在Numpy中，使用numpy.add函数来完成两个数组之间的加法运算，如果两个数组的形状相同，那么它就会将对应位置的元素相加。函数原型为：

```
numpy.add(x1, x2, /, out=None, *, where=True, casting='same_kind',
order='K', dtype=None, subok=True)
```

- `x1`：第一个输入数组或标量。
- `x2`：第二个输入数组或标量。
- `out`：输出数组，数据类型必须与预期输出相符。如果提供，它必须具有与输出相同形状的适当类型来接收结果。
- `where`：表示计算加法的条件，如果设置为True，则在相应位置进行计算；如果设置为False，则在相应位置不进行计算。
- `casting`：定义如何处理数据类型转换，例如 'no', 'equiv', 'safe', 'same_kind', 'unsafe'。
- `order`：指定结果的内存布局，'C' 表示 C 风格，'F' 表示 Fortran 风格，'A' 表示原数组样式，'K' 表示元素在内存中的出现顺序。
- `dtype`：指定输出数组的类型。
- `subok`：控制返回数组是否可以是输入数组的子类。

```
import numpy as np
```

```
# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# add会使arr1和arr2中的每个对应位置的元素相加
arr3 = np.add(arr1, arr2)

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)

condition = np.array([[True, True, True], [False, False, False]],
dtype=np.bool_)
arr4 = np.add(arr1, arr2, where=condition)
print('运算的结果为：\n', arr4)
```

8.2.2 减法运算

1. 使用符号运算

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# 数组减去数组，会使对应位置的元素相减
arr3 = arr1 - arr2

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)
```

2. 使用函数运算

在Numpy中，使用numpy.subtract函数来完成两个数组之间的减法运算，如果两个数组的形状相同，那么它就会将对应位置的元素相减。函数原型为：

```
numpy.subtract(x1, x2, /, out=None, *, where=True,
casting='same_kind', order='K', dtype=None, subok=True)
```

- `x1`：第一个输入数组或标量。

- `x2`: 第二个输入数组或标量。
- `out`: 可选输出数组，用于放置运算结果，其数据类型必须与预期输出相符。
- `where`: 表示计算减法的条件，如果设置为 True，则在相应位置进行计算；如果设置为 False，则在相应位置不进行计算。
- `casting`: 定义如何处理数据类型转换，例如 'no', 'equiv', 'safe', 'same_kind', 'unsafe'。
- `order`: 指定结果的内存布局，'C' 表示 C 风格，'F' 表示 Fortran 风格，'A' 表示原数组样式，'K' 表示元素在内存中的出现顺序。
- `dtype`: 指定输出数组的类型。
- `subok`: 控制返回数组是否可以是输入数组的子类。

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# subtract会使arr1中的每个元素减去arr2中的对应位置的元素
arr3 = np.subtract(arr1, arr2)

print(arr1)
print(arr2)
print('运算的结果为: \n', arr3)
```

8.2.3 乘法运算

1. 使用符号运算

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# 数组乘以数组，会使对应位置的元素相乘
arr3 = arr1 * arr2

print(arr1)
print(arr2)
print('运算的结果为: \n', arr3)
```

2. 使用函数运算

在Numpy中，使用numpy.multiply函数来完成两个数组之间的乘法运算，如果两个数组的形状相同，那么它就会将对应位置的元素相乘。函数原型为：

```
numpy.multiply(x1, x2, /, out=None, *, where=True,
casting='same_kind', order='K', dtype=None, subok=True)
```

- `x1`: 第一个输入数组或标量。
- `x2`: 第二个输入数组或标量。
- `out`: 可选输出数组，用于放置运算结果，其数据类型必须与预期输出相符。
- `where`: 表示计算乘法的条件，如果设置为 True，则在相应位置进行计算；如果设置为 False，则在相应位置不进行计算。
- `casting`: 定义如何处理数据类型转换，例如 'no', 'equiv', 'safe', 'same_kind', 'unsafe'。
- `order`: 指定结果的内存布局，'C' 表示 C 风格，'F' 表示 Fortran 风格，'A' 表示原数组样式，'K' 表示元素在内存中的出现顺序。
- `dtype`: 指定输出数组的类型。
- `subok`: 控制返回数组是否可以是输入数组的子类。

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# multiply会使arr1中的每个元素乘以arr2中的对应位置的元素
arr3 = np.multiply(arr1, arr2)

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)
```

除了进行对应元素之间的相乘之外，更重要的是数组之间的点积，即：

$$A_{MN} \times B_{NP} = C_{MP}$$

在Numpy中，符合该公式的函数为`numpy.dot`，使用该函数即可得到符合点积要求的两个数组的点积结果，函数原型为：

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

- a：第一个输入数组。
- b：第二个输入数组。
- out：可选输出数组，用于放置运算结果。

```
import numpy as np

# 二维数组矩阵乘法
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 3]])
matrix_product = np.dot(A, B)
print(matrix_product)
```

8.2.4 除法运算

1. 使用符号运算

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# 数组除以数组，会使对应位置的元素相除
arr3 = arr1 / arr2

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)
```

2. 使用函数运算

在Numpy中，使用numpy.divide函数来完成两个数组之间的除法运算，如果两个数组的形状相同，那么它就会将对应位置的元素相除。函数原型为：

```
numpy.divide(x1, x2, /, out=None, *, where=True,
casting='same_kind', order='K', dtype=None, subok=True)
```

- x1：第一个输入数组或标量。
- x2：第二个输入数组或标量。
- out：可选输出数组，用于放置运算结果，其数据类型必须与预期输出相符。

- `where`: 表示计算除法的条件，如果设置为 True，则在相应位置进行计算；如果设置为 False，则在相应位置不进行计算。
- `casting`: 定义如何处理数据类型转换，例如 'no', 'equiv', 'safe', 'same_kind', 'unsafe'。
- `order`: 指定结果的内存布局，'C' 表示 C 风格，'F' 表示 Fortran 风格，'A' 表示原数组样式，'K' 表示元素在内存中的出现顺序。
- `dtype`: 指定输出数组的类型。
- `subok`: 控制返回数组是否可以是输入数组的子类。

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# divide会使arr1中的每个元素除以arr2中的对应位置的元素
arr3 = np.divide(arr1, arr2)

print(arr1)
print(arr2)
print('运算的结果为：\n', arr3)
```

8.2.5 开根号

1. 使用符号运算

```
import numpy as np

# 生成一个2行3列的数组
arr1 = np.array([[4, 9, 16], [25, 36, 49]])

# 开根号操作就是让一个数组进行0.5次方的幂运算
arr2 = arr1 ** 0.5

print(arr1)
print('运算的结果为：\n', arr2)
```

2. 使用函数运算

在Numpy中，使用numpy.power函数来完成两个数组之间的幂运算，如果两个数组的形状相同，那么它就会将对应位置的元素进行幂运算。函数原型为：

```
numpy.power(x1, x2, /, out=None, *, where=True,
casting='same_kind', order='K', dtype=None, subok=True)
```

- `x1`: 第一个输入数组或标量。
- `x2`: 第二个输入数组或标量，表示幂。
- `out`: 可选输出数组，用于放置运算结果，其数据类型必须与预期输出相符。
- `where`: 表示计算幂运算的条件，如果设置为 True，则在相应位置进行计算；如果设置为 False，则在相应位置不进行计算。
- `casting`: 定义如何处理数据类型转换，例如 'no', 'equiv', 'safe', 'same_kind', 'unsafe'。
- `order`: 指定结果的内存布局，'C' 表示 C 风格，'F' 表示 Fortran 风格，'A' 表示原数组样式，'K' 表示元素在内存中的出现顺序。
- `dtype`: 指定输出数组的类型。
- `subok`: 控制返回数组是否可以是输入数组的子类。

```
import numpy as np

# 生成两个2行3列的数组
arr1 = np.array([[11, 12, 13], [14, 15, 16]])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# power会使arr1中的每个元素为底数，arr2中的对应位置的元素为指数进行运算
arr3 = np.power(arr1, arr2)

print(arr1)
print(arr2)
print('运算的结果为: \n', arr3)
```

9. Ndarray数组的广播机制

通常情况下，为了进行元素级的算术运算，两个数组的形状必须完全一致，比如上面数组和数组的运算案例中，运算的两个数组的形状是一模一样的。而上面的标量与数组的运算案例中，运算的两个对象的形状并不相同，那它们为什么就可以直接运算呢?

其实是因为Numpy的广播机制，它提供了一种规则，能够将不同形状的两个计算数组广播到相同形状，然后再去进行元素级的算术运算，这种规则使得形状不完全匹配的数组也能相互运算。通过这个机制，Numpy能够在保持效率的同时，扩展数组的操作范围，从而无需显式地扩展数组维度或进行循环遍历以实现元素级的计算，极大

的增强了数组的处理能力。

其具体规则为：

1. 如果两个数组的维数不同，形状较小的数组会在前面补1。
2. 如果两个数组的形状在某个维度上不匹配，且其中一个维度长度为1，则会沿该维度复制扩展以匹配另一个数组的形状。
3. 如果在任一维度上不匹配且没有维度等于1，则会引发异常。

比如前面的标量与数组的加法运算中，`arr = np.array([[1, 2, 3], [4, 5, 6]])`，标量是2。

首先，标量 2 的形状可以看作是（），而二维数组`arr`的形状是（2, 3）。根据广播规则，维度较低的标量会在其形状左边添加与高维数组相同的维度数，使其维度与高维数组相同。这里标量 2 被扩展为形状（1,1）的数组，再进一步扩展为形状（2,3）的数组，即`[[2,2,2],[2,2,2]]`。然后，这个扩展后的数组与原二维数组`arr`进行对应元素的运算。所以最终结果为：

`arr + 2 = [[1+2,2+2,3+2],[4+2,5+2,6+2]] = [[3,4,5],[6,7,8]]`。

```
import numpy as np

arr1 = np.zeros((3, 3))
arr2 = np.ones((3, 1))
print('arr1: \n', arr1)
print('arr2: \n', arr2)

print(arr1 + arr2)
```

10. 其他操作

10.1 修改维度

在Numpy中，使用squeeze和expand_dims可以去修改数组的维度。

10.1.1 squeeze

该函数的作用是从数组形状中删除所有单维度的条目，即把形状中为1的维度去掉。函数原型为：

```
numpy.squeeze(a, axis=None)
```

- `a`: 输入数组。它可以是任何形状的数组，但至少有一个维度的大小为1。
- `axis`: 可选参数。一个整数或整数元组。如果指定了该参数，则只压缩指定的轴。如果该轴在数组 `a` 中不是单维度的，则不会进行压缩。如果未指定，则所有单维度的轴都将被压缩。

```
import numpy as np

# 创建一个具有单维度的数组
arr = np.array([[[1], [2], [3]]])
print(arr)
print("原始数组形状:", arr.shape)

# 使用squeeze函数去掉所有单维度的条目
squeezed_arr = np.squeeze(arr)
print(squeezed_arr)
print("压缩后的数组形状:", squeezed_arr.shape)
```

10.1.2 expand_dims

该函数的作用是在指定的位置增加一个单一维度，即在指定的位置增加一个形状为1的维度。函数原型为：

```
numpy.expand_dims(a, axis)
```

- `a`: 输入数组。
- `axis`: 整数或者整数元组。表示在哪个位置增加单一维度。

```
import numpy as np

# 创建一个一维数组
arr = np.array([1, 2, 3])
print("原始数组形状:", arr.shape)
print(arr)

# 在位置0增加一个维度
expanded_arr = np.expand_dims(arr, axis=0)
print("增加维度后的数组形状:", expanded_arr.shape)
print(expanded_arr)

# 在位置1增加一个维度
expanded_arr = np.expand_dims(arr, axis=1)
```

```
print("增加维度后的数组形状:", expanded_arr.shape)
print(expanded_arr)
```

10.2 连接数组

10.2.1 concatenate

该函数用于将多个数组沿指定的轴连接起来，形成一个更大的数组。函数原型为：

```
numpy.concatenate((a1, a2, ..., arr_n), axis=0, out=None)
```

- `(a1, a2, ..., arr_n)`：这是一个包含数组的元组，这些数组需要被连接。所有数组在除了连接轴之外的其他维度上必须有相同的形状。
- `axis`：整数，指定沿着哪个轴进行连接。如果不指定，默认为 0，表示第一个轴。
- `out`：可选参数，如果提供，结果将直接存储在这个数组中。这个数组必须具有与输入数组相同的形状和数据类型。

工作原理为：

1. **输入验证**：`concatenate` 接受一个元组或列表作为输入，其中包含一系列数组。这些数组可以是任意维度，但它们在除了要连接的轴之外的所有维度上必须具有相同的形状。
2. **轴参数**：`concatenate` 有一个必需的参数 `axis`，它指定了沿着哪个轴进行连接。轴的编号从 0 开始，对于一维数组，只有一个轴（轴 0）。对于二维数组，轴 0 是行，轴 1 是列。
3. **形状兼容性检查**：所有输入数组在 `axis` 参数指定的轴上的维度大小可以不同，但在其他轴上的维度大小必须相同。例如，如果 `axis=1`，则所有输入数组的行数必须相同。
4. **内存分配**：在连接之前，`concatenate` 会计算输出数组的大小，并分配足够的内存空间来存储结果。
5. **数据复制**：`concatenate` 会将输入数组的数据复制到新分配的内存空间中。具体来说，它将沿着指定的轴顺序地复制每个数组的数据，从而形成一个新的数组。
6. **结果数组**：输出结果是一个新的数组，它在 `axis` 指定的轴上的维度大小是所有输入数组在该轴上维度大小的总和，而在其他轴上则与输入数组相同。

```
import numpy as np
```

```
# 创建两个数组
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])

print(f'数组a的形状为{a.shape}, 数组a为: \n', a)
print(f'数组b的形状为{b.shape}, 数组b为: \n', b)

# 沿着第一个轴（垂直方向）连接数组
c = np.concatenate((a, b), axis=0)
print(f'数组c的形状为{c.shape}, 数组c为: \n', c)

# 沿着第二个轴（水平方向）连接数组
d = np.concatenate((a, b.T), axis=1)
print(f'数组b的转置的形状为{b.T.shape}, 数组b的转置为: \n', b.T)
print(f'数组d的形状为{d.shape}, 数组d为: \n', d)
```

10.2.2 stack

该函数用于沿着新的轴连接一系列数组。与 `numpy.concatenate` 不同，`numpy.stack` 总是创建一个新的轴，而 `numpy.concatenate` 则是在现有轴上进行数组的连接。函数原型为：

```
numpy.stack(arrays, axis=0, out=None)
```

- `arrays`：一系列数组，它们将被堆叠在一起。所有数组必须具有相同的形状。
- `axis`：整数，表示新轴的位置。默认值为 0。
- `out`：可选参数，如果提供，结果将直接存储在这个数组中。这个数组必须具有与输出数组相同的形状和数据类型。

工作原理为：

1. **输入验证**：`stack` 接受一个数组序列（例如一个列表或元组）作为输入。所有输入数组必须具有相同的形状。
2. **轴参数**：`stack` 有一个必需的参数 `axis`，它指定了新轴的位置。
3. **形状兼容性检查**：所有输入数组在除了要创建的新轴外的所有维度上必须具有相同的形状。
4. **内存分配**：`stack` 会计算输出数组的形状，并在内存中为这个新数组分配空间。新数组的形状将比输入数组的形状多一个维度，这个新增的维度的大小等于输入数组的数量。

5. **数据复制**：`stack` 将输入数组的数据复制到新分配的内存空间中，每个输入数组在新轴上占据一个位置。
6. **结果数组**：输出结果是一个新的数组，其形状在 `axis` 参数指定的位置上增加了一个维度。

```
import numpy as np

# 创建三个一维数组
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(f'数组a的形状为{a.shape}, 数组a为：\n', a)
print(f'数组b的形状为{b.shape}, 数组b为：\n', b)

# 沿着新的轴堆叠数组
d = np.stack((a, b), axis=0)
print(f'数组d的形状为{d.shape}, 数组d为：\n', d)

# 沿着第二个轴堆叠数组
e = np.stack((a, b), axis=1)
print(f'数组e的形状为{e.shape}, 数组e为：\n', e)
```

10.2.3 hstack

在 NumPy 中，`numpy.hstack`（水平堆叠）函数用于沿着水平方向堆叠数组序列。这个函数实际上是 `numpy.concatenate` 函数的一个特化版本，其中 `axis` 参数固定为 1。函数原型为：

```
numpy.hstack(tup)
```

- `tup`：一个数组序列，它们将被水平堆叠在一起。所有数组在除了第二个轴之外的轴上必须具有相同的形状。

其工作原理为：

1. **输入验证**：首先，`hstack` 会检查所有输入数组是否都是二维的，或者至少可以被视为二维的。如果输入数组是多维的，它们将被重新塑造为二维数组。
2. **形状兼容性检查**：所有输入数组在除了第二个轴（列）之外的所有其他轴上必须有相同的形状。这意味着，如果输入数组是一系列的矩阵，那么这些矩阵的行数必须相同。例如，如果你有两个矩阵 A 和 B，它们分别是 3x2 和 3x3 的，那么它们可以在水平方向上堆叠，因为它们都有 3 行。

3. **堆叠操作**：如果输入数组通过了形状兼容性检查，`hstack` 将沿着第二个轴（列）将它们连接起来。这意味着，对于每个输入数组，它们的列将被依次排列在一起。
4. **结果数组**：输出结果是一个新的二维数组，其列数是所有输入数组列数的总和，而行数与输入数组中的任何一个相同。

```
import numpy as np

# 创建两个 2x2 的矩阵
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

print(f'数组a的形状为{a.shape}, 数组a为：\n', a)
print(f'数组b的形状为{b.shape}, 数组b为：\n', b)

# 使用 hstack 进行水平堆叠
c = np.hstack((a, b))

# 输出结果
print(f'数组c的形状为{c.shape}, 数组c为：\n', c)
```

10.2.4 vstack

`numpy.vstack` 函数用于将多个数组沿垂直方向（即沿着第一个轴，axis=0）堆叠成一个单一的数组。函数原型为：

```
numpy.vstack(tup)
```

- `tup`：一个包含数组的元组或列表。所有数组在除了第一个轴外的维度上必须具有相同的形状。

工作原理为：

1. **输入验证**：`vstack` 接受一个数组序列（例如一个列表或元组）作为输入。所有输入数组在除了第一个轴外的所有其他维度上必须具有相同的形状。
2. **内存分配**：`vstack` 会计算输出数组的形状，并在内存中为这个新数组分配空间。输出数组的第一个维度将是所有输入数组第一个维度大小的总和，其他维度与输入数组相同。
3. **数据复制**：`vstack` 将输入数组的数据复制到新分配的内存空间中，每个输入数组在新数组中占据连续的块。

4. **结果数组**：输出结果是一个新的数组，其中输入数组沿第一个轴（垂直方向）堆叠。

```
import numpy as np

# 创建两个二维数组
a = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([[7, 8, 9]])

print(f'数组a的形状为{a.shape}, 数组a为: \n', a)
print(f'数组b的形状为{b.shape}, 数组b为: \n', b)


# 使用 vstack 进行垂直堆叠
c = np.vstack((a, b))

# 输出结果
print(f'数组c的形状为{c.shape}, 数组c为: \n', c)
```

10.3 分割数组

10.3.1 split

该函数用于沿着指定的轴将数组分割成多个子数组，可以指定要分割的数组、分割的位置或子数组的数量。函数原型为：

```
numpy.split(ary, indices_or_sections, axis=0)
```

- `ary`：要分割的数组。
- `indices_or_sections`：可以是一个整数，表示要将数组平均分割成多少个子数组；也可以是一个整数数组，表示分割的位置。
- `axis`：沿着哪个轴进行分割，默认为 0，即第一个轴。

工作原理为：

1. **输入验证**：`split` 接受一个数组 `ary` 作为输入，以及一个指示如何分割数组的参数 `indices_or_sections`。如果提供了 `axis` 参数，它指定了沿哪个轴分割数组，默认为 0。
2. **解析分割参数**：如果 `indices_or_sections` 是一个整数，它表示要将数组分割成多少个大小相等的子数组。如果是一个序列，它表示沿指定轴的分割点。

3. **计算分割点**：如果 `indices_or_sections` 是整数，`split` 会计算每个子数组应该包含的元素数量。如果 `indices_or_sections` 是一个序列，则直接使用这些值作为分割点。
4. **执行分割**：`split` 使用计算出的分割点在指定轴上将原始数组分割成多个子数组。这通常涉及到创建原始数组的视图，而不是复制数据。
5. **返回结果**：`split` 返回一个列表，其中包含所有分割后的子数组。

```
import numpy as np

# 创建一个一维数组
a = np.array([1, 2, 3, 4, 5, 6])

# 使用 split 平均分割数组为 3 个子数组
result1 = np.split(a, 3)

# 使用 split 按位置分割数组
result2 = np.split(a, [2, 4])

# 输出结果
print("平均分割为 3 个子数组：", result1)
print("按位置分割：", result2)
```

10.3.2 hsplit

`numpy.hsplit` 用于沿着横向（水平方向）将数组分割成多个子数组。这个函数是 `numpy.split` 的一个特化版本，专门用于沿着第二个轴（axis=1）进行分割。以下是 `numpy.hsplit` 函数的介绍：

函数原型为：

```
numpy.hsplit(ary, indices_or_sections)
```

- `ary`：要分割的数组，通常是一个二维数组。
- `indices_or_sections`：可以是一个整数，表示要将数组平均分割成多少个子数组；也可以是一个整数数组，表示分割的位置。

工作原理为：

1. **输入验证**：`hsplit` 接受一个数组 `ary` 作为输入，以及一个指示如何分割数组的参数 `indices_or_sections`。与 `split` 不同，`hsplit` 不接受 `axis` 参数，因为它默认沿着第二个轴（axis=1）进行分割。

2. **解析分割参数**：如果 `indices_or_sections` 是一个整数，它表示要将数组在水平方向上平均分割成多少个子数组。如果是一个序列，它表示沿第二个轴的分割点。
3. **计算分割点**：如果 `indices_or_sections` 是整数，`hsplit` 会计算每个子数组在水平方向上应该包含的列数。如果 `indices_or_sections` 是一个序列，则直接使用这些值作为分割点。
4. **执行分割**：`hsplit` 使用计算出的分割点在第二个轴上将原始数组分割成多个子数组。与 `split` 类似，这通常涉及到创建原始数组的视图，而不是复制数据。
5. **返回结果**：`hsplit` 返回一个列表，其中包含所有分割后的子数组。每个子数组都是原始数组的一部分，沿第二个轴分割而成。

```
import numpy as np

# 创建一个 6x4 的二维数组
arr = np.arange(24).reshape(6, 4)

# 打印原始数组
print("原始数组:")
print(arr)

# 指定每部分应该包含的列数
col_counts = [1, 2]

# 使用 hsplit 分割数组
subarrays = np.hsplit(arr, col_counts)

# 打印分割后的子数组
print("\n分割后的子数组:")
print(subarrays)
```

10.3.3 vsplit

`numpy.vsplit` 用于沿着纵向（垂直方向）将数组分割成多个子数组。这个函数是 `numpy.split` 的一个特化版本，专门用于沿着第一个轴（axis=0）进行分割。以下是 `numpy.vsplit` 函数的介绍：

函数原型为：

```
numpy.vsplit(ary, indices_or_sections)
```

- `ary`：要分割的数组，通常是一个二维数组。
- `indices_or_sections`：可以是一个整数，表示要将数组平均分割成多少个子数组；也可以是一个整数数组，表示分割的位置。

工作原理为：

1. **输入验证**：`vsplit` 接受一个数组 `ary` 作为输入，以及一个指示如何分割数组的参数 `indices_or_sections`。与 `split` 不同，`vsplit` 不接受 `axis` 参数，因为它默认沿着第一个轴（axis=0）进行分割。
2. **解析分割参数**：如果 `indices_or_sections` 是一个整数，它表示要将数组在垂直方向上平均分割成多少个子数组。如果是一个序列，它表示沿第一个轴的分割点。
3. **计算分割点**：如果 `indices_or_sections` 是整数，`vsplit` 会计算每个子数组在垂直方向上应该包含的行数。如果 `indices_or_sections` 是一个序列，则直接使用这些值作为分割点。
4. **执行分割**：`vsplit` 使用计算出的分割点在第一个轴上将原始数组分割成多个子数组。与 `split` 类似，这通常涉及到创建原始数组的视图，而不是复制数据。
5. **返回结果**：`vsplit` 返回一个列表，其中包含所有分割后的子数组。每个子数组都是原始数组的一部分，沿第一个轴分割而成。

```
import numpy as np

# 创建一个 6x4 的二维数组
arr = np.arange(24).reshape(6, 4)

# 打印原始数组
print("原始数组:")
print(arr)

# 指定每部分应该包含的列数
col_counts = [1, 2]

# 使用 hsplit 分割数组
subarrays = np.vsplit(arr, col_counts)

# 打印分割后的子数组
print("\n分割后的子数组:")
print(subarrays)
```

10.4 数组元素的添加与删除

10.4.1 append

该函数用于将值追加到数组的末尾。函数原型为：

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

- `arr`：原始数组，可以是任何形状。
- `values`：要追加的值，它们会被追加到 `arr` 的末尾。`values` 的形状必须与 `arr` 在除了要追加的轴之外的所有轴上兼容。
- `axis`：可选参数，指定要追加值的轴。如果 `axis` 没有被指定，`arr` 会被展平，`values` 会被追加到结果数组的末尾。

注意事项：

- `numpy.append` 在追加元素时，如果 `values` 是一个标量，它会自动广播以匹配 `arr` 的形状。
- 如果 `values` 是一个数组，它的形状必须与 `arr` 在除了要追加的轴之外的所有轴上匹配。
- `numpy.append` 可能不是最高效的操作，因为它涉及到创建一个新的数组。如果频繁追加元素，考虑使用 `numpy.concatenate` 或 `numpy.append` 的替代方法，如使用 `numpy.resize` 和直接赋值。

```
import numpy as np

# 创建一个一维数组
arr = np.array([1, 2, 3])

# 要追加的值
values = np.array([4, 5, 6])

# 使用 numpy.append 追加值
result = np.append(arr, values)

# 输出结果
print(result)

# 创建一个二维数组
arr_2d = np.array([[1, 2], [3, 4]])

# 要追加的值
values_2d = np.array([[5], [6]])
```

```
# 使用 numpy.append 追加值，沿着列方向
result_2d = np.append(arr_2d, values_2d, axis=1)

# 输出结果
print(result_2d)
```

10.4.2 insert

该函数用于在数组的指定位置插入元素。与 `numpy.append` 不同的是，`numpy.insert` 允许用户在数组的任意位置插入元素，而不仅仅是数组的末尾。函数原型为：

```
numpy.insert(arr, obj, values, axis=None)
```

- `arr`：原始数组，可以是任何形状。
- `obj`：表示插入位置的索引。它可以是整数或者整数数组。如果是整数，则表示在哪个位置插入。如果是整数数组，则表示在哪些位置插入。
- `values`：要插入的值。如果 `arr` 是多维数组，则 `values` 的形状必须在除了 `axis` 指定的轴以外的所有轴上与 `arr` 保持一致。
- `axis`：可选参数，指定插入操作的轴。如果未指定，则 `arr` 会被展平，然后 `values` 会被插入到一维数组中。

注意事项：

- 如果 `obj` 是一个整数数组，`values` 也会被分割成多个部分，每个部分被插入到对应的索引位置。
- 如果 `values` 是一个标量，它会自动广播以匹配需要插入的位置。
- 插入操作可能会比追加操作更加低效，因为它涉及到数组元素的移动。

```
import numpy as np

# 创建一个一维数组
arr = np.array([1, 2, 3, 4, 5])
print(arr)


# 在索引 2 的位置插入值 10
result = np.insert(arr, 2, 10)

# 输出结果
```

```
print(result)


# 在索引 [1, 3] 的位置插入值 [20, 30]
result_2 = np.insert(arr, [1, 3], [20, 30])

# 输出结果
print(result_2)


# 在二维数组中插入值
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
print(arr_2d)
# 在索引 1 的位置沿着列方向插入值 [[10], [20]]
result_2d = np.insert(arr_2d, [0], [10], 1)

# 输出结果
print(result_2d)
```

10.4.3 delete

该函数用于从数组中删除指定的子数组，并返回一个新的数组。这个函数允许用户删除数组中的单个元素或多个连续的元素。函数原型为：

```
numpy.delete(arr, obj, axis=None)
```

- `arr`：原始数组，可以是任何形状。
- `obj`：表示要删除的子数组的索引。它可以是单个整数、一个整数列表或一个整数数组。如果是一个整数，它表示要删除单个元素的位置；如果是列表或数组，则表示要删除多个元素的位置。
- `axis`：可选参数，指定删除操作的轴。如果未指定，则 `arr` 会被展平，然后根据 `obj` 指定的索引删除元素。

注意事项：

- 如果 `obj` 是一个列表或数组，删除操作会按照索引的顺序进行，这意味着删除操作可能会影响后续索引的位置。
- 如果 `obj` 中的索引超出数组的范围，`numpy.delete` 会抛出一个异常。

```
import numpy as np
```

```
# 创建一个一维数组
arr = np.array([1, 2, 3, 4, 5])

# 删除索引 2 的元素
result = np.delete(arr, 2)

# 输出结果
print(result)


# 删除索引 [1, 3] 的元素
result_2 = np.delete(arr, [1, 3])

# 输出结果
print(result_2)


# 在二维数组中删除元素
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# 删除索引 1 的列
result_2d = np.delete(arr_2d, 1, axis=1)

# 输出结果
print(result_2d)


# 删除索引 [0, 2] 的行
result_2d_row = np.delete(arr_2d, [0, 2], axis=0)

# 输出结果
print(result_2d_row)
```


10.5 统计计算

10.5.1 mean

该函数用于计算数组中元素的平均值。它可以计算整个数组的平均值，也可以沿着指定的轴（axis）计算平均值。函数原型为：

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

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算平均值的轴。如果没有指定，则计算整个数组的平均值。如果指定了轴，则计算该轴上元素的平均值。
- `dtype`：可选参数，用于指定返回的平均值的数据类型。如果没有指定，则通常返回与输入数组相同的数据类型。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的平均值会保留原始数组的维度，其值默认为 `False`。

注意事项：

- 如果 `a` 是一个空数组，或者指定轴上的所有元素都是空值（如 `NaN`），则 `numpy.mean` 返回 `NaN`。
- 如果 `keepdims=True`，则计算出的平均值将具有与输入数组相同的维度数，但沿着指定轴的维度大小为1。

```python
import numpy as np

# 创建一个一维数组
arr = np.array([1, 2, 3, 4, 5])

# 计算整个数组的平均值
mean_arr = np.mean(arr)

# 输出结果
print(mean_arr)

# 创建一个二维数组
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr_2d)

# 计算整个二维数组的平均值
mean_arr_2d = np.mean(arr_2d)

# 输出结果
print(mean_arr_2d)
```

```
# 计算二维数组沿着列的平均值
mean_arr_2d_col = np.mean(arr_2d, axis=0)

# 输出结果
print(mean_arr_2d_col)

# 计算二维数组沿着行的平均值
mean_arr_2d_row = np.mean(arr_2d, axis=1)

# 输出结果
print(mean_arr_2d_row)

# 使用 keepdims=True 保留维度
mean_arr_2d_row_keepdims = np.mean(arr_2d, axis=1, keepdims=True)

# 输出结果
print(mean_arr_2d_row_keepdims)
```

10.5.2 sum

该函数用于计算数组中所有元素的和，或者沿指定轴计算元素的和。函数原型为：

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

下面是各个参数的说明：

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算和的轴。如果没有指定，则计算整个数组的和。如果指定了轴，则计算该轴上元素的和。可以是整数或元组，用于指定多个轴。
- `dtype`：可选参数，指定返回和的数据类型。如果未指定，则通常与输入数组的类型相同，除非输入数组为布尔型，此时默认返回 `int64` 或 `int32`。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的和会保留原始数组的维度，其值默认为 `False`。
- `initial`：可选参数，如果提供，则用于指定当数组为空时的初始值。
- `where`：可选参数，用于指定计算和的条件。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 0, 6], [7, 8, 9]])
print(arr)

# 计算整个数组的和
total_sum = np.sum(arr)
print(total_sum)

# 计算每一列的和
sum_col = np.sum(arr, axis=0)
print(sum_col)

# 计算每一行的和
sum_row = np.sum(arr, axis=1)
print(sum_row)

# 保留原始维度
sum_row_keep = np.sum(arr, axis=1, keepdims=True)
print(sum_row_keep)
```

10.5.3 max和min

`numpy.max` 和 `numpy.min` 是 NumPy 库中用于计算数组中元素最大值和最小值的函数。

`numpy.max` 函数返回数组中的最大值或沿指定轴的最大值。函数原型为：

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

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算最大值的轴。如果没有指定，则计算整个数组的最大值。如果指定了轴，则计算该轴上元素的最大值。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的最大值会保留原始数组的维度，其值默认为 `False`。
- `initial`：可选参数，如果提供，则用于指定当数组为空时的初始值。
- `where`：可选参数，用于指定计算最大值的条件。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 0, 6], [7, 8, 9]])
print(arr)

# 计算整个数组的最大值
max_val = np.max(arr)

# 计算每一列的最大值
max_val_col = np.max(arr, axis=0)

# 计算每一行的最大值
max_val_row = np.max(arr, axis=1)

print(max_val)
print(max_val_col)
print(max_val_row)
```

`numpy.min` 函数返回数组中的最小值或沿指定轴的最小值。函数原型为：

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

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算最小值的轴。如果没有指定，则计算整个数组的最小值。如果指定了轴，则计算该轴上元素的最小值。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的最小值会保留原始数组的维度，其值默认为 `False`。
- `initial`：可选参数，如果提供，则用于指定当数组为空时的初始值。
- `where`：可选参数，用于指定计算最小值的条件。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 0, 6], [7, 8, 9]])
print(arr)

# 计算整个数组的最小值
min_val = np.min(arr)
```

```
# 计算每一列的最小值
min_val_col = np.min(arr, axis=0)

# 计算每一行的最小值
min_val_row = np.min(arr, axis=1)

print(min_val)
print(min_val_col)
print(min_val_row)
```

10.5.4 var

`numpy.var` 用于计算数组中所有元素或沿指定轴的方差。方差是衡量一组数值分散程度的统计量，它是各个数值与其平均数差的平方的平均数。函数原型为：

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

下面是各个参数的说明：

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算方差的轴。如果没有指定，则计算整个数组的方差。如果指定了轴，则计算该轴上元素的方差。可以是整数或元组，用于指定多个轴。
- `dtype`：可选参数，指定返回方差的数据类型。如果未指定，则通常与输入数组的类型相同。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的方差会保留原始数组的维度，其值默认为 `False`。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr)

# 计算整个数组的方差
total_var = np.var(arr)

# 计算每一列的方差
var_col = np.var(arr, axis=0)
```

```
# 计算每一行的方差
var_row = np.var(arr, axis=1)

print(total_var)
print(var_col)
print(var_row)
```

10.5.5 std

`numpy.std` 用于计算数组中所有元素的标准差，或者沿着指定轴计算标准差。标准差是方差的平方根，它也是一种衡量数据分散程度的统计量。函数原型为：

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

下面是各个参数的说明：

- `a`：输入数组，可以是任何形状的数组。
- `axis`：可选参数，用于指定计算标准差的轴。如果没有指定，则计算整个数组的标准差。如果指定了轴，则计算该轴上元素的标准差。可以是整数或元组，用于指定多个轴。
- `dtype`：可选参数，指定返回标准差的数据类型。如果未指定，则通常与输入数组的类型相同。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。
- `keepdims`：可选参数，如果设置为 `True`，则计算后的标准差会保留原始数组的维度，其值默认为 `False`。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# 计算整个数组的标准差
total_std = np.std(arr)

# 计算每一列的标准差
std_col = np.std(arr, axis=0)

# 计算每一行的标准差
std_row = np.std(arr, axis=1)

print(total_std)
```

```
print(std_col)
print(std_row)
```

10.5.6 argmax和argmin

`numpy.argmax` 和 `numpy.argmin` 是 NumPy 库中的两个函数，它们分别用于找出数组中最大值和最小值的索引位置。

函数原型：

```
numpy.argmax(a, axis=None, out=None)
numpy.argmin(a, axis=None, out=None)
```

这两个函数的参数是相同的：

- `a`：输入数组。如果数组是多维的，则沿着指定的轴搜索最大值或最小值。
- `axis`：可选参数，用于指定搜索最大值或最小值的轴。如果没有指定，则在展平的数组中进行搜索。如果指定了轴，则沿着该轴搜索，并返回该轴上的索引位置。
- `out`：可选参数，如果提供，结果会直接写入到这个数组中。这个数组必须具有正确的形状。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 0, 6], [7, 8, 9]])
print(arr)

# 找出整个数组中的最大值和最小值的索引位置
max_index = np.argmax(arr)
min_index = np.argmin(arr)

# 找出每一列中的最大值和最小值的索引位置
max_index_col = np.argmax(arr, axis=0)
min_index_col = np.argmin(arr, axis=0)

# 找出每一行中的最大值和最小值的索引位置
max_index_row = np.argmax(arr, axis=1)
min_index_row = np.argmin(arr, axis=1)

print(max_index)
print(min_index)
```

```
print(max_index_col)
print(min_index_col)

print(max_index_row)
print(min_index_row)
```

10.6 where

`numpy.where` 是 NumPy 库中的一个非常有用的函数，它可以根据指定的条件返回满足该条件的元素的索引。

当 `numpy.where` 接受一个条件作为参数时，它会返回一个元组，其中包含满足该条件的元素的索引。函数原型：

```
numpy.where(condition)
```

参数：

- `condition`：一个布尔数组或条件表达式。

返回值：

- 一个元组，其中包含满足条件的元素的索引。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
condition = arr > 5

result = np.where(condition)
print(result)
```

在这个例子中，`condition` 是一个布尔数组，它表示 `arr` 中哪些元素大于 5。`numpy.where` 返回一个元组，其中包含满足条件的元素的行索引和列索引。