第3章

NumPy 科学计算

NumPy，全称为 Numerical Python，是一个用于科学计算的 Python 库。它提供了强大的多维数组对象以及用于处理这些数组的丰富函数。

NumPy 是 Python 科学计算生态系统中的核心库之一，许多其他科学计算库，如 SciPy、Pandas 和 Matplotlib，都基于它构建。使用 NumPy，用户可以高效地进行各种数值计算、数据分析和科学计算任务。

3.1 NumPy 数组

NumPy 的核心是 ndarray（*n*-dimensional Array，多维数组）对象，它是一个多维、同构的数据容器，只能存储相同类型的元素，使 NumPy 非常适合处理数值运算和大规模数据。它的特性和优势如下：

（1）支持多维数据存储。ndarray 对象可以存储多维数据，因此 NumPy 非常适合处理图像、音频、视频等需要高效存储和处理的数据。

（2）性能高效。NumPy 底层使用 C 语言编写，利用了向量化计算和优化的算法，具有出色的计算效率和性能，它可以快速执行数组操作，如数值计算和统计分析等。

（3）支持语义计算。NumPy 提供了简单易用的接口，使得数组操作更加直观和方便。例如，可以对整个数组进行数学运算和逻辑运算等操作，而无须使用循环。

例如，将 Python 列表[1，2，3]和[4，5，6]分别包装为 ndarray 数组后，可以直接对其进行加法运算。

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

对应的输出结果为

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

（4）支持广播机制。通过广播机制，NumPy 可以对形状不同的数组进行操作，使数组之间的计算更加灵活和方便。

(5) 高度优化的线性代数运算。NumPy 应用了并行化指令集和高度优化的算法,提供了丰富的线性代数运算功能。例如,可以进行矩阵乘法、求逆、特征值分解、奇异值分解等操作。

(6) 作为其他库的基础。NumPy 是许多数据处理和科学计算库的核心,例如 SciPy、Pandas 和 Scikit-learn 等。这些库都建立在 NumPy 之上,利用其高效的数组性能和灵活的接口执行数据分析、机器学习等任务。

NumPy 是 Python 的第三方库,使用前先通过 pip 包管理器完成安装。

在导入 NumPy 库时,通常按照惯例将其命名为 np,这是一个广泛使用的别名。

```
import numpy as np
```

https://numpy.org 是 NumPy 库的官网,包括 NumPy 的文档、教程、指南和示例等资源。官网是学习和使用 NumPy 的重要途径之一,可以帮助用户更全面地了解和掌握其强大的功能。

3.2 创建数组

NumPy 提供了多种创建数组的方法,以下是一些常见的方式。

3.2.1 array()函数

在 NumPy 中,创建 ndarray 对象使用 array()函数,它可以接收列表、元组等容器创建数组,将已有的数据转换为 NumPy 数组进行进一步的科学计算和数据处理。例如:

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

对应的输出结果为

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
<class 'numpy.ndarray'>
```

ndarray 对象有许多属性,用于描述数组各方面的信息,主要属性如表 3-1 所示。

表 3-1 ndarray 对象的主要属性

属 性	说 明
shape	数组的维度,返回一个元组,元组的每个元素代表对应维度的大小
dtype	元素的数据类型,例如整数、浮点数等
ndim	数组的维度数量,即轴的个数
size	数组中的元素总数
itemsize	数组中每个元素占用的字节数

例如，查看数组 arr 的属性。

```
In [2]  arr = np.array([[1, 2, 3], [4, 5, 6]])
        print("Shape:", arr.shape)
        print("Data type:", arr.dtype)
        print("Number of dimensions:", arr.ndim)
        print("Number of elements:", arr.size)
        print("Size of each element (in bytes):", arr.itemsize)
```

对应的输出结果为

```
Shape: (2, 3)
Data type: int32
Number of dimensions: 2
Number of elements: 6
Size of each element (in bytes): 4
```

dtype 代表数组对象的数据类型，数据类型定义了数组中元素的存储方式、占用的内存空间以及可以进行的操作。NumPy 的 ndarray 对象支持很多种数据类型，如表 3-2 所示。

表 3-2　ndarray 对象支持的数据类型

属　性	说　明
int	整数类型，可以是不同位数的有符号或无符号整数（如 int8、int16、uint32 等）
float	浮点数类型，包括单精度浮点数（float32）和双精度浮点数（float64）等
bool	布尔类型，表示 True 或 False
complex	复数类型，包括单精度复数（complex64）和双精度复数（complex128）
str	字符串类型，表示文本数据
object	Python 对象类型，可以包含任意类型的 Python 对象
datetime	日期和时间类型

NumPy 支持多种数据类型，主要目的是在不同的应用场景中提高计算速度并节省内存空间。例如，在进行简单的数学运算时，使用整数类型通常比浮点数类型更快、更高效；而在涉及大量小数运算时，浮点数类型则更为适用。

在创建数组时，如果没有显式指定数据类型，NumPy 会根据提供的数据自动推断出一个合适的数据类型。推断时遵循“就高不就低”的原则，例如，当数组中同时包含整数和浮点数元素时，NumPy 会选择一种能容纳所有元素的浮点数类型，以避免精度损失。如果希望避免 NumPy 意外选择错误的数据类型，或为了节省内存仅需达到特定的计算精度，可以显式指定 ndarray 的数据类型。例如：

```
In [3]  x = np.array([1, 2, 3], dtype = "int16")        #指定 dtype
        y = np.array([[4.0, 5, 6], [1.1, 2, 3]])        #未指定，由 NumPy 推断
        print("dtype: {}".format(x.dtype))
        print("dtype: {}".format(y.dtype))
```

对应的输出结果为

```
dtype: int16
dtype: float64
```

ndim 和 shape 是描述数组维度信息的重要属性，它们都与轴(axis)的概念密切相关。ndim 表示数组的轴数，shape 表示每个轴方向上元素的数量。每个维度的数组对轴都有统一的编号。在数组运算中，轴是常用的参数，用于指定沿着哪个维度进行各种计算，如图 3-1 所示。

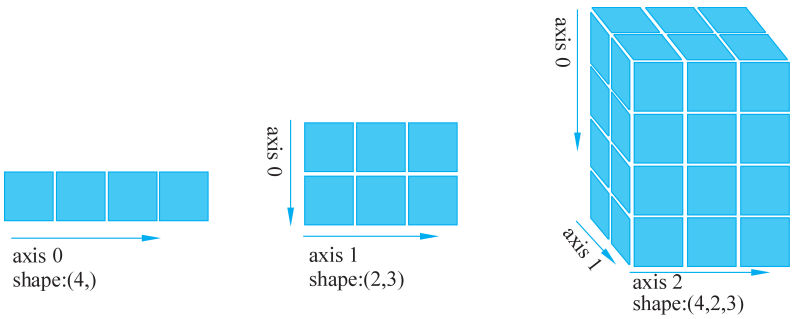

图 3-1 数组维度信息

3.2.2 数组维度变换

数组的维度是可以改变的，维度变换是数组非常重要的一个操作。通过 reshape()和 resize()方法改变数组的形状，通过 flatten()和 ravel()方法降维展开数组，通过 transpose()方法和 T 对数组进行转置，这些变换是数据处理、图像处理等领域的常见操作。

1. reshape()和 resize()方法

reshape()方法返回一个新数组，该数组是原数组按照指定形状变换后的结果，原数组保持不变。例如：

```
In [1]: arr = np.array([1, 2, 3, 4, 5, 6])
        new_arr = arr.reshape((2, 3))          #将一维数组变换为二维数组
        print(new_arr)
        print(arr)                             #原数组不变
```

对应的输出结果为

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

当指定某个维度的大小为 −1 时，NumPy 会根据其他维度的大小自动计算该维度的大小，以确保数组的总元素个数保持不变。例如：

```
In [2]: a.reshape((2,-1))
```

对应的输出结果为

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

resize()方法与 reshape()方法功能相似，但它直接修改原数组，并返回 None。reshape()

方法要求新形状的总元素数量与原数组相同，否则会引发错误。而 resize()方法则有不同的处理方式：如果新形状比原始形状大，它会用 0 填充额外的空间；如果新形状比原始形状小，则会截断数组。

总的来说，resize()方法更加灵活，因为它可以就地修改数组并处理形状不匹配的情况。而 reshape()方法则更为安全，它不会修改原数组，而是返回一个新的数组。

【说明】 在 NumPy 中，大多数功能既可以通过模块级调用（函数）实现，也可以通过对象级调用（方法）实现。模块级调用是直接使用 NumPy 中的函数对数组进行操作，而对象级调用则是通过 ndarray 数组对象调用其相关方法进行操作。

reshape()方法的模块级调用写法如下：

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

无论是模块级调用还是对象级调用，它们的功能都是相同的，都可以实现相应的数组操作。选择使用哪种方式主要取决于个人的编程习惯和具体需求。

2. flatten()和 ravel()方法

flatten()方法将多维数组变换为一维数组。例如：

```
In [1]: arr = np.array([[1, 2, 3], [4, 5, 6]])
        new_arr = arr.flatten()        #将二维数组变换为一维数组
        new_arr
```

对应的输出结果为

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

flatten()方法返回的数组是原数组的一个副本，二者相互独立。因此，对返回的数组所做的任何修改都不会影响原数组。

ravel()方法与 flatten()方法类似，都是将多维数组变换为一维数组。与 flatten()方法不同的是，ravel()方法返回的是原数组的一个视图（view），这意味着视图与原数组共享数据存储。因此，对返回的数组所做的修改会影响原数组。例如：

```
In [2]: arr = np.array([[1, 2, 3], [4, 5, 6]])
        new_arr = arr.ravel()          #返回一个视图，将二维数组变换为一维数组
        print(new_arr)
        new_arr[0] = 100               #修改新数组元素
        print(new_arr)
        print(arr)                     #原数组同步变化
```

对应的输出结果为

```
[1 2 3 4 5 6]
[100   2   3   4   5   6]
[[100   2   3]
[4   5   6]]
```

3. transpose()方法和 T

transpose()方法和 T 都可以用来对数组进行转置操作。它们的作用是相同的，都可以

将数组的行和列交换。例如：

```
In [1]  arr = np.array([[1, 2, 3], [4, 5, 6]])
        new_arr = arr.transpose()          #或者 arr.T
```

对应的输出结果为

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

4. np.newaxis 增加维度

np.newaxis 是 NumPy 中的一个特殊常量，用于增加数组的维度。当需要在特定位置插入一个新的维度时，可以使用 np.newaxis 在指定位置插入一个新的轴，从而将原数组变换为更高维度的数组。例如：

```
In [1]  arr = np.arange(6)                  #array([0, 1, 2, 3, 4, 5])
        arr.shape
```

对应的输出结果为

```
(6, )
```

```
In [2]  arr_new = arr[:, np.newaxis]        #在第二个位置(列方向)增加一个新的维度
        arr_new
```

对应的输出结果为

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

```
In [3]  arr_new.shape
```

对应的输出结果为

```
(6, 1)
```

3.2.3 NumPy 内置函数

在 NumPy 中，可以使用各种函数创建特殊的数组，常用函数如表 3-3 所示。

表 3-3 创建数组的常用函数

函　数	说　明
ones(shape)	根据 shape 创建一个全 1 数组，shape 为元组类型，dtype 默认为 float64
zeros(shape)	根据 shape 创建一个全 0 数组，shape 为元组类型，dtype 默认为 float64
eye(n)	创建一个 n 维单位数组，对角线为 1，其余为 0，dtype 默认为 float64
full(shape, val)	根据 shape 创建一个数组，每个元素值都是 val，shape 为元组类型

续表

函　数	说　明
arange ([start,]　stop [, step])	创建一个等差数列,类似 Python 的 range()函数,返回 ndarray 类型。元素默认从 0 到 stop－1,可以指定 start 和 step,用于整数情况
linspace(start, stop, num＝50, endpoint＝True)	创建一个等差数列,根据起止数据(默认含右侧端点 stop)等间距形成等差数组,用于浮点数情况

ones()、zeros()和 eye()函数创建全 1 数组、全 0 数组和单位数组。例如:

```
In [1]  np.ones((3, 3))
```

对应的输出结果为

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

```
In [2]  np.zeros((2, 3))
```

对应的输出结果为

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

```
In [3]  np.eye(3)
```

对应的输出结果为

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

full()函数用于将数组中的所有元素初始化为指定的值。

```
In [1]  np.full(3, 0)                  #创建形状为 (3,) 的全 0 数组
```

对应的输出结果为

```
array([0, 0, 0])
```

```
In [2]  np.full((2, 3), 1)             #创建形状为 (2,3) 的全 1 数组
```

对应的输出结果为

```
array([[1, 1, 1],
       [1, 1, 1]])
```

arange()函数与 Python 的 range()函数功能相似,用于产生整数序列。例如:

```
In [1]  np.arange(10)
```

对应的输出结果为

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

```
In [2]  np.arange(2, 10)
```

对应的输出结果为

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

```
In [3]  np.arange(1, 10, 3)
```

对应的输出结果为

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

linspace()函数用于产生浮点数序列，默认包含终止值，所有数字之间的间隔相等。例如：

```
In [1]  np.linspace(1, 10, 7)
```

对应的输出结果为

```
array([1. , 2.5, 4. , 6.5, 7. , 8.5, 10. ])
```

```
In [2]  np.linspace(1, 10, 7, endpoint = False)
```

对应的输出结果为

```
array([1. , 2.28571429, 3.57142857, 4.85714286, 6.14285714, 7.42857143,
8.71428571])
```

在 linspace()函数中，将 retstep 参数设置为 True 可以返回数列的步长。

```
In [3]  arr, step = np.linspace(2, 6, num = 5, retstep = True)
        print(arr)
        print(step)
```

对应的输出结果为

```
[2. 3. 4. 5. 6.]
1.0
```

linspace()函数是一个常用的数值计算工具，快速生成等间距样例点数组，并进行各种数值计算和科学计算。其主要应用场景如下：

(1) 生成等间距的样例点。例如，使用 linspace()函数在一定范围内生成若干等间距的样例点，用于曲线拟合、插值等数值计算问题。

(2) 生成网格点。在二维或三维空间中，使用 linspace()函数生成两个或 3 个等间距样例点组成的数组，并将它们作为坐标轴上的点，用于可视化、计算网格数据等领域。

(3) 计算采样步长。当需要在一定时间或空间范围内对某个函数进行采样时，可以使用 linspace()函数计算出样例点之间的距离，从而确定采样间隔。

linspace()是 NumPy 中必不可少的基础函数。

3.2.4 random 模块函数

在 NumPy 库中，numpy.random 模块提供了多种生成随机数的函数，以及与随机数相关的其他操作功能。random 模块中的常用函数如表 3-4 所示。

numpy.random 模块与 Python 标准库中的 random 模块的主要区别如下：

(1) 操作对象不同。numpy.random 模块针对 ndarray 数组进行随机数操作，Python 的 random 模块主要针对单个数值或列表进行操作。

表 3-4　random 模块中的常用函数

函　　数	说　　明
rand(d_0, d_1, …, d_n)	返回给定维度的随机样本，取值范围为[0, 1)
randn(d_0, d_1, …, d_n)	返回给定维度的标准正态分布样本
normal(loc=0.0, scale=1.0, size=None)	返回指定均值和标准差的正态分布样本
randint(low, high=None, size=None, dtype=int)	返回指定范围内的随机整数
uniform(low=0.0, high=1.0, size=None)	返回指定范围内的均匀分布的随机浮点数
permutation(x)	对给定的数组或者整数序列进行随机排列
shuffle(x)	随机打乱数组中的元素顺序
choice(a, size=None, replace=True, p=None)	从给定序列中随机选择元素
seed(seed=None)	初始化随机数生成器的种子

（2）增加维度信息。numpy.random 模块可以方便地生成高维随机数，Python 的 random 模块则无法直接生成高维随机数。

（3）效率更高。由于 NumPy 基于 C 语言实现，因此 numpy.random 模块在处理大量数据时通常比 Python 的 random 模块更高效。

（4）分布函数更丰富。numpy.random 模块提供了更多的分布函数选项，例如均匀分布、正态分布、指数分布、泊松分布等；Python 的 random 模块仅提供了一些基本的分布函数，如均匀分布和正态分布。

如果需要处理大量数据或进行多维数组的随机数操作，优先选择 numpy.random 模块。

1. rand()、randn()和 normal()函数

rand(d_0, d_1,…, d_n) 用于生成给定维度的、取值范围在 [0, 1) 随机样本数组。

其中，d_0,d_1,…,d_n是生成随机样本数组的维度，不指定维度则生成一个随机浮点数，可以指定一个或多个维度参数。例如：

```
In [1]  np.random.rand()                #一个随机浮点数,无维度
```

对应的输出结果为

```
0.666018692030193
```

```
In [2]  np.random.rand(3)               #3 个随机浮点数,shape = (3,)
```

对应的输出结果为

```
array([0.87655575, 0.61631995, 0.76611553])
```

```
In [3]  np.random.rand(3, 2)            #3 行 2 列随机浮点数,shape = (3,2)
```

对应的输出结果为

```
array([[0.99353118, 0.22584602],
       [0.1657952, 0.77387099],
       [0.24827181, 0.1565485]])
```

randn()函数与 rand()函数相似，但生成的样本满足正态分布。方差和均值是正态分布的两个重要参数。均值决定了分布的中心位置。方差决定了分布的形状：方差越大，分布越分散；方差越小，分布越集中。例如：

```
In [4]  np.random.randn(3, 2)          #正态分布抽样浮点数
```

对应的输出结果为

```
array([[-0.07865957, 0.50727013],
       [ 0.05368669, 0.6232558],
       [-0.62553655, -0.81609455]])
```

如果希望指定正态分布的随机数的均值和标准差(方差的平方根)，则可以使用 normal(loc=0.0，scale=1.0，size=None)函数，其中，参数 loc 为正态分布的均值，scale 为正态分布的标准差。例如：

```
In [5]  #在均值为 2、标准差为 0.5 的正态分布的随机数数组中抽样
        np.random.normal(2, 0.5, size = (3, 3))
```

对应的输出结果为

```
array([[2.27561314, 2.0997858, 2.47596742],
       [2.05368129, 1.46345549, 2.44633882],
       [1.57363747, 1.22854528, 2.00142761]])
```

2. randint()和 uniform()函数

randint()和 uniform()函数都用于生成指定范围内的随机数。

randint(low，high=None，size=None，dtype=int)用于生成指定范围内的随机整数，上限为开区间，不指定 size 参数时生成一个随机整数。例如：

```
In [1]  np.random.randint(0, 10)      #生成一个 [0,10)区间内的随机整数
```

对应的输出结果为

```
8
```

```
In [2]  #生成取值在[0,10)区间内、形状为(3,3)的随机整数数组
        np.random.randint(0, 10, (3,3))
```

对应的输出结果为

```
array([[8, 1, 2],
       [7, 3, 0],
       [8, 9, 3]])
```

uniform(low=0.0，high=1.0，size=None)用于生成指定范围内均匀分布的随机浮点数，默认范围为 [0，1)。例如：

```
In [3]  #生成取值在[20,30)区间内、形状为(3,2)的随机浮点数数组
        np.random.uniform(20, 30, (3,2))
```

对应的输出结果为

```
array([[22.21429951, 24.12252375],
```

```
       [24.96982382, 29.73791851],
       [24.75725006, 26.68149557]])
```

3. permutation()、shuffle()和 choice()函数

permutation()、shuffle()和 choice()函数都是 NumPy 库中用于生成随机序列或对序列进行随机操作的函数。

permutation()函数用于对给定的序列进行随机排列，并返回一个新的排列后的数组。它不会修改原数组，而是返回一个新的数组。可以接收一个序列数据作为参数；也可以接收一个整数 n 作为参数，n 表示要进行排列的范围，即 $0,1,2,\cdots,n-1$。例如：

```
In [1]: np.random.permutation(6)                    #对[0, 6)区间内的整数进行随机排列
```

对应的输出结果为

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

```
In [2]: x = [8, 5, 7, 6, 20, 13, 16, 19]
        x = np.random.permutation(x)                #对给定的序列进行随机排列,产生新数组
        print(x)
```

对应的输出结果为

```
[ 5 13 7 20 6 19 8 16]
```

shuffle()函数用于对给定的序列进行原地随机排列，直接修改原数组。它会打乱数组中元素的顺序，而不返回一个新的数组。例如：

```
In [3]: x = [8, 5, 7, 6, 20, 13, 16, 19]
        np.random.shuffle(x)                        #无返回值
        print(x)                                    #原数组被打乱
```

对应的输出结果为

```
[7, 6, 5, 19, 13, 16, 8, 20]
```

choice()函数用于从给定的序列中随机选择元素。可以接收一个数组、列表或整数 n 作为参数，还可以使用参数 replace=False 确保所选择的元素不重复。例如：

```
In [4]: x = np.arange(6)                            #array([0, 1, 2, 3, 4, 5])
        #从 x 数组中随机选择 3 个不重复的元素
        np.random.choice(x, size = 3, replace = False)
```

对应的输出结果为

```
array([2, 3, 1])
```

```
In [5]: #从[0,10)区间内随机选择 5 个元素,允许重复
        np.random.choice(10, size = 5)
```

对应的输出结果为

```
array([2, 8, 2, 4, 9])
```

3.2.5 数组拼接

数组拼接是指将多个数组沿着指定的轴方向合并成一个新的数组。数组拼接在数据处理和分析中非常常见，可用于以下应用场景：

（1）数据整合。将多个数据集合并为一个更大的数据集，以便进行整体分析和处理。

（2）特征工程。在机器学习中，将不同特征组合到一起形成新的特征向量，以提高模型的表现。

（3）数据处理。对不同部分的数据进行合并，以便进行更有效的数据清洗、转换和分析。

（4）图像处理。可以将多个图像数组按照需要的方式拼接在一起，实现图像合成或图像增广。

（5）时间序列处理。在时间序列分析中，可以将不同时间段的数据进行拼接，以获得全面的信息，构建趋势。

（6）模型输入准备。在深度学习中，将训练数据按照批次拼接成合适的输入形式，以供神经网络模型训练使用。

concatenate((a1, a2, a3,…), axis=0)函数将多个数组沿指定轴拼接在一起，它接收一个由要拼接的数组组成的元组作为参数，默认沿着第一个维度进行拼接，并返回拼接后的新数组。关于轴的概念见图 3-1。例如：

```
In [1]  arr1 = np.array([[1, 2], [3, 4]])
        arr2 = np.array([[5, 6], [7, 8]])
        np.concatenate((arr1, arr2), axis = 0)          #沿着纵轴合并
```

对应的输出结果为

```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

```
In [2]  np.concatenate((arr1, arr2), axis = 1)          #沿着横轴合并
```

对应的输出结果为

```
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

除此之外，vstack()和 hstack()函数专用于垂直（沿第一个维度）和水平（沿第二个维度）堆叠数组，它们不能沿其他轴拼接数组。例如：

```
In [3]  np.vstack((arr1, arr2))
```

对应的输出结果为

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

```
        [5, 6],
        [7, 8]])
```

```
In [4]: np.hstack((arr1, arr2))
```

对应的输出结果为

```
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

3.3　选取数组元素

在 NumPy 中，可以通过索引和切片访问和操作数组中的元素。

3.3.1　基本索引

基本索引用于从数组中获取单个元素，返回一个标量（单个数据）。

一维数组索引通过指定数组的索引位置访问元素，格式为“数组[索引]”。例如：

```
In [1]: a = np.arange(9)
        a
```

对应的输出结果为

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

```
In [2]: a[0]
```

对应的输出结果为

```
0
```

多维数组索引使用由逗号分隔的索引访问元素，格式为“数组[第一维索引，第二维索引，…]”。例如：

```
In [3]: b = np.array([[0,1,2],[3,4,5]])
        b
```

对应的输出结果为

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

```
In [4]: b[0, 1]          #二维数组，逗号分隔两个索引
```

对应的输出结果为

```
1
```

3.3.2　切片

切片操作通过原数组得到一个新的数组，且为原数组的视图。

一维数组的切片格式为“数组[起始索引：结束索引：步长]”，与 Python 列表的切片操

作相同。例如：

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

对应的输出结果为

```
array([0, 1, 2])
```

```
In [2]  a[3:]
```

对应的输出结果为

```
array([3, 4, 5, 6, 7, 8])
```

```
In [3]  a[1:7:2]
```

对应的输出结果为

```
array([1, 3, 5])
```

```
In [4]  a[::-1]
```

对应的输出结果为

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

```
In [5]  a[:5] = 10          #视图操作
        a                   #原数组改变
```

对应的输出结果为

```
array([10, 10, 10, 10, 10, 5, 6, 7, 8])
```

多维数组的切片格式为："数组[起始索引：结束索引：步长，起始索引：结束索引：步长，…]"，即将各维度的切片用逗号分隔。例如：

```
In [1]  b = np.array([[0,1,2],[3,4,5]])
```

对应的输出结果为

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

```
In [2]  b[1, :]
```

对应的输出结果为

```
array([3, 4, 5])
```

```
In [3]  b[:, 2]
```

对应的输出结果为

```
array([2, 5])
```

```
In [4]  b[:, 1:3]
```

对应的输出结果为

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

【例 3-1】　利用切片访问图像数组。

Matplotlib 库是一个用于绘制二维图像的第三方库，将在第 5 章详细介绍。imread() 是 Matplotlib 中用于读取图像文件并返回图像数组的函数，它返回一个表示图像的 NumPy 数组。下面通过图像演示切片操作的效果。

```
In [1]: import matplotlib.pyplot as plt
        img = plt.imread('./img/dog.jpg')        #读取./img/dog.jpg 文件
        print(type(img))
```

对应的输出结果为

```
<class 'numpy.ndarray'>
```

```
In [2]: plt.imshow(img)                          #如图 3-2(a)所示
        plt.imshow(img[300:, :])                 #行维度切片，如图 3-2(b)所示
        plt.imshow(img[ :, 300:])                #列维度切片，如图 3-2(c)所示
        img_rotate = img[::-1]                   #行维度逆置
        plt.imshow(img_rotate)                   #如图 3-2(d)所示
        img_rotate = img[ :, ::-1]               #列维度逆置
        plt.imshow(img_rotate)                   #如图 3-2(e)所示
        t = img.copy()                           #复制数组(图像)
        s = np.hstack((t,t))                     #拼接数组，沿 x 轴拼接图像
        u = np.vstack((s,s,s))                   #拼接数组，沿 y 轴拼接图像
        plt.imshow(u)                            #如图 3-2(f)所示
```

3.3.3　整数列表索引

在 NumPy 中，可以使用整数列表对数组进行选取操作。这种方式适用于同时选择多个不相邻的元素或根据特定位置选择元素。

整数列表可以出现在任意维度，用于指定要获取数据的位置。通过逐个提取列表中每个整数索引对应的元素，可以获得该元素的数组副本。例如：

```
In [1]: arr = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
        arr[[1, 3, 5, 6]]
```

对应的输出结果为

```
array([10, 30, 50, 60])
```

```
In [2]: arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8],
                        [9, 10, 11, 12], [13, 14, 15, 16]])
        arr[[1, 2]]          #行维度应用整数列表索引，列维度是切片
```

对应的输出结果为

```
array([[ 5, 6, 7, 8],
       [ 9, 10, 11, 12]])
```

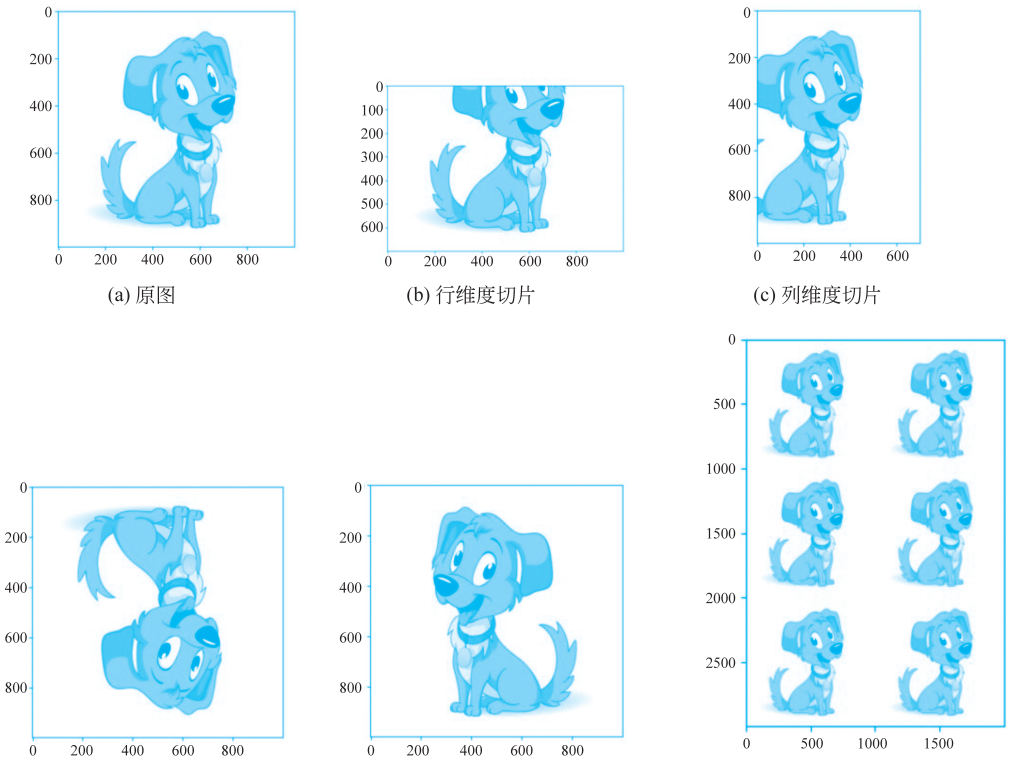

图 3-2 图像数组的切片操作

In [3]	arr[:, [0, 2]] #列维度应用整数列表索引,行维度是切片

对应的输出结果为

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15]])
```

In [4]	#不要全部都是整数列表索引,至少保证一个维度是切片 arr[[1, 2]][:, [0, 2]]

对应的输出结果为

```
array([[ 5,  7],
       [ 9, 11]])
```

【例 3-2】 使用整数列表索引筛选成绩。

设 grades 数组保存了几个学生的成绩,使用整数列表索引将不及格的成绩保存至另一个数组。

```
import numpy as np

#创建学生成绩数组
grades = np.array([75, 80, 45, 60, 90, 30, 55])
```

```
#找出不及格学生成绩的索引
pass_indices = [i for i in range(len(grades)) if grades[i] < 60]
#使用整数列表索引获取不及格学生成绩
passing_grades = grades[pass_indices]
print(passing_grades) #[45 30 55]
```

3.3.4　布尔数组索引

如果确切知道要选择的元素的索引，使用索引和切片的方法非常有效。然而，在许多情况下，可能并不知道要选择的元素的确切索引。例如，有一个形状为 10 000 × 10 000 的数组，取值为 [1, 15 000]区间的随机整数，现只想选择其中取值小于 20 的元素。在这种情况下，整数索引显然不再适用。

布尔数组索引是一种通过布尔数组(而非确切的索引值)选择数组元素的方法。布尔数组的每个元素与要索引的数组对应，每个位置的布尔值会匹配相应的元素(两者形状相同)。仅当布尔数组对应位置为 True 时，才会选择该元素。使用布尔数组索引得到的是数组的副本。

设有如下数组：

```
In [1]  arr = np.arange(25).reshape(5, 5)
```

对应的输出结果为

```
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

现在希望将数组中取值大于 10 的元素修改为 −1。首先，需要生成一个满足条件“取值＞10”的布尔数组。NumPy 数组支持广播机制(详见 3.6 节)，这意味着数组可以与标量直接进行运算，其结果是每个元素与该标量值进行运算后生成的新数组。例如：

```
In [2]  arr > 10        #数组每个元素与 10 比较，得到 True 或 False
```

对应的输出结果为

```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, True, True, True, True],
       [ True, True, True, True, True],
       [ True, True, True, True, True]])
```

显然，这个布尔数组可以作为索引筛选数组元素。在筛选出满足条件的元素后，可以对这些元素进行赋值。例如：

```
In [3]  arr[arr > 10] = -1
        arr
```

对应的输出结果为

```
array([[ 0, 1, 2, 3, 4],
```

```
       [ 5, 6, 7, 8, 9],
       [10, -1, -1, -1, -1],
       [-1, -1, -1, -1, -1],
       [-1, -1, -1, -1, -1]])
```

如果筛选条件由多个关系组成，就需要使用逻辑与、或、非运算。NumPy 对布尔值的逻辑与、或、非运算可以分别使用 &、| 和 ～ 运算符，它们会对数组中的每个布尔值进行相应的逻辑运算。需要注意的是，为了确保正确的计算顺序，使用 &、| 和 ～ 的布尔表达式中的每个关系运算都应分别用括号括起来。例如：

```
In [4]  (arr >10) & (arr <20)
```

对应的输出结果为

```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, True, True, True, True],
       [ True, True, True, True, True],
       [False, False, False, False, False]])
```

【例 3-3】 使用布尔数组索引进行数据清洗。

假设正在分析一组销售数据，其中包含了一些异常值和缺失值。

```
sales = np.array([100, 200, 300, 400, -999, np.nan, 500, 600, 700, 800])
```

其中，np.nan 是 NumPy 中表示缺失值或不可用值的特殊常量。NumPy 中还有一些常用的常量，如 np.pi(圆周率)、np.e(自然对数的底 e)、np.inf(正无穷大)等。

现使用布尔数组索引筛选这些异常值和缺失值，确保数据的质量和准确性。

np.isnan()函数用于判断取值是否为 np.nan，如果是返回 True，否则返回 False。这里的数据清洗希望筛选出非 np.nan 数据，所以在其前进行～运算。具体的数据清洗操作如下：

```
condition = (sales >0) & (～np.isnan(sales))
filtered_sales = sales[condition]
print(filtered_sales)
```

输出结果为

```
[100. 200. 300. 400. 500. 600. 700. 800.]
```

【例 3-4】 筛选鸢尾花数据。

鸢尾花数据集是 Scikit-learn 机器学习库中的经典数据集。该数据集包含 150 条鸢尾花样本数据。其中，特征值存储在名为 data 的 ndarray 数组中，形状为(150, 4)，即每个样本包含 4 个特征；目标值则存储在名为 target 的 ndarray 数组中，形状为(150,)，其取值为 0、1 和 2，分别对应 3 种鸢尾花：setosa、versicolor 和 virginica，如图 3-3 所示。

导入、获取数据的代码如下：

```
from sklearn.datasets import load_iris

iris = load_iris()              #导入数据集
data = iris.data                #特征值数据，ndarray 数组，形状为(150, 4)
target = iris.target            #目标值数据，ndarray 数组，形状为(150, )
```

图 3-3　鸢尾花数据集中的 3 种鸢尾花

设要根据目标值筛选出 3 种鸢尾花的数据，则可以使用布尔数组作为行索引实现，代码如下：

```
setosa = data[target == 0]
versicolor = data[target == 1]
virginica = data[target == 2]
```

3.4　NumPy 数组运算

NumPy 提供了大量的数组运算，使各种科学计算变得更加简单和高效。

3.4.1　基本运算

基本运算包括数组的标量运算和两个数组进行的双目运算。

数组的标量运算是指将一个标量与数组中的每个元素进行相同的运算，如加、减、乘、除、求余、求幂、取整、比较等。数组的标量运算对每个元素都执行相同的操作，使代码更加简洁高效，避免了循环操作每个数组元素的烦琐过程。

数组的双目运算指的是对两个数组进行逐元素操作的运算，它要求参与运算的两个数组具有相同的形状，以便能够按元素进行操作，返回值为相同形状的多维数组。常见的双目运算包括算术运算、比较运算和逻辑运算。例如：

```
In [1]: arr1 = np.random.randint(1, 10, (2,3))
        arr1
```

对应的输出结果为

```
array([[7, 3, 8],
       [7, 5, 1]])
```

```
In [2]: arr2 = np.random.rand(2,3)
        arr2
```

对应的输出结果为

```
array([[0.61908918, 0.77194627, 0.21670458],
       [0.08428996, 0.73259487, 0.8974566 ]])
```

```
In [3]: arr1 + arr2
```

对应的输出结果为

```
array([[7.61908918, 3.77194627, 8.21670458],
       [7.08428996, 6.73259487, 1.8974566 ]])
```

3.4.2 通用函数运算

通用函数(universal function,ufunc)对数组中的元素进行逐元素操作,并返回一个新的数组作为结果。NumPy 提供了许多内置的 ufunc 函数,包括常见的数学函数(如加法、减法、乘法、除法、幂运算、指数函数、对数函数等)、三角函数等。这些 ufunc 函数可以直接应用于数组,无须显式编写循环。NumPy 的常用通用函数如表 3-5 所示。

表 3-5 NumPy 的常用通用函数

函 数	说 明
数学函数	add(), subtract(), multiply(), divide(), power(), abs(), fabs(), sqrt(), square(), exp(), log(), log10(), log2(), ceil(), floor(), rint(), round(), maximum(x, y), minimum(x, y), mod(x, y)
三角函数	sin(), cos(), tan(), sinh(), cosh(), tanh()
逻辑判定函数	any(), all()

这些 ufunc 函数的计算结果均为新的 ndarray 数组。例如:

```
In [1]: amounts = np.array([19.99, 35.55, 42.22, 99.98])
        np.rint(amounts)            #将数组每个元素四舍五入为整数,返回新数组
```

对应的输出结果为

```
array([ 20., 36., 42., 100.])
```

```
In [2]: np.round(amounts, 2)        #将数组每个元素四舍五入到小数点后两位,返回新数组
```

对应的输出结果为

```
array([19.99, 35.55, 42.22, 99.98])
```

```
In [3]: arr1 = np.random.rand(2, 3)
        arr1
```

对应的输出结果为

```
array([[0.61908918, 0.77194627, 0.21670458],
       [0.08428996, 0.73259487, 0.8974566 ]])
```

```
In [4]: arr1 +arr2
```

对应的输出结果为

```
array([[0.6226222, 0.24758995, 0.07884157],
       [0.89670508, 0.51587071, 0.99272352]])
```

```
In [5]: np.maximum(arr1, arr2)      #求两个数组每个位置上的最大值,返回新数组
```