5.1 NumPy简介

NumPy是Numerical Python的缩写，它是Python中一个用于科学计算的扩展程序库。NumPy因其包含丰富的数学函数、支持高维数组与矩阵且运行速度快等特点而得到了广泛应用。具体而言，NumPy的主要功能有：

ndarray，一种支持矢量计算和广播能力的多维数组，运行速度快且节省空间；

能快速处理整组数据的数学函数；

整合C/C++/Fortran代码的工具；

线性代数、傅立叶变换、随机数生成等功能；

磁盘数据读写工具；

内存映射文件操作工具。

5.2 NumPy的安装

和许多扩展库一样，NumPy的安装可以通过pip进行。如果你已经安装了Python 3.x或以上版本，则安装Python时就已经自带了pip工具。打开命令行窗口，输入pip install numpy命令，即可自动下载并安装NumPy。

安装完成后，可以打开一个命令行窗口进入Python终端会话，对之前的安装进行测试。输入import numpy as np命令，尝试导入NumPy，如果没有出现任何错误信息提示，说明说明NumPy已经成功安装，可以使用。

如果你已经安装了Anaconda版本的Python，则Anaconda中已经集成了包括NumPy在内的多种科学包，无需重复安装。

5.3 NumPy数组简介

**5.3.1 NumPy数组的基本创建、访问与属性**

数组NumPy中所提供的最基本，也是最常用的一种数据类型。NumPy数组的维度数量称之为秩（rank），即n维数组的秩为n。对于多维数组而言，构成它的每一个单一维度的数组称之为一个轴（axis）或维度（dimension）。例如，一个二维数组可以看作两个一维数组；这两个一维数组中的每个元素又可以看做一个一维数组。

首先介绍如何直接创建一个NumPy数组并查看一些关键属性。可以使用数组对象的构造函数，即array()函数创建数组。下面这条语句便创建了一个简单的一维数组：

|  |
| --- |
| >>> a = np.array([0, 1, 2, 3]) |

输入这条语句后，再键入数组名a并按回车键，就会出现array([0, 1, 2, 3])字样，表明我们已经创建了数组 [0, 1, 2, 3]。

直接创建二维或更高维度的数组方法同样是使用array()函数，只是在参数的输入上略有区别：

|  |
| --- |
| >>> b = np.array([[1, 2, 3, 4],[11, 12, 13, 14]]) # 创建二维数组  >>> b  array([[ 1, 2, 3, 4],[11, 12, 13, 14]]) |

在5.5节，我们将进一步介绍array()函数及其他数组创建函数。

**5.3.2 NumPy数组的访问**

访问NumPy数组元素的方法与许多其他语言中对数组的访问类似，是通过指定数组元素的索引（下标）来访问某个位置的元素。对于一维数组而言，其访问方式几乎与Python中对列表元素的访问相同，而对于高维数组，则略有区别：

|  |
| --- |
| >>> a = np.array([1, 2, 4, 8, 16])  >>> b = np.array([[1, 2, 3, 4],[11, 12, 13, 14]])  >>> a[1] # 访问一维数组中的元素  2  >>> b[1, 3] # 访问二维数组中的元素  14  >>> b[1] # 获取二维数组中某一行元素  array([11, 12, 13, 14]) |

以如上的对二维数组b的访问为例，若要访问某个具体元素（如第1行第3列的元素），则通过形如a[1, 3]的方式指定其索引；而如果直接访问b[1]，则将取得该二维数组中的整行元素。

**5.3.3 NumPy数组的常用属性**

NumPy数组对象的属性非常丰富，接下来介绍一些常用的属性及其查看方式。

使用type(a)函数可以查看到，对象a的类型为ndarray：

|  |
| --- |
| >>> type(a)  <class 'numpy.ndarray'> |

而具体到数组中元素的数值类型，可以用a.dtype查看：

|  |
| --- |
| >>> a.dtype  dtype('int32') |

在数组中需要注意数据类型的强制转换问题。例如，若试图向int32类型的数组中赋一个浮点型数值，则小数部分会被丢弃：

|  |
| --- |
| >>> a.dtype  dtype('int32')  >>> a[0] = 3.14  >>> a  array([3, 3, 4, 5, 6, 7, 8, 9]) |

由于NumPy的最主要作用之一就是处理矩阵数组，因此了解一个数组的维度就很关键。在NumPy中，我们可以通过shape属性查看数组的维度，shape是一个整数元组，元组中每一个元素代表每个维度的大小。例如，假设对某个数组b查看属性b.shape，得到一元组(3,)，表示b是一个一维数组，包含3个元素；如果得到二元组(2, 3)，表示b是一个二维数组，在两个维度上分别含有2个元素和3个元素，以此类推。

NumPy中的数组对象ndarray有如下较为常用的属性：

|  |  |
| --- | --- |
| ndarray.shape | 数组的维度，或矩阵的行列数。它返回一个元组，元组的长度就是维度的数目 |
| ndarray.size | 数组中元素的总数 |
| ndarray.dtype | ndarray对象的元素类型 |
| ndarray.itemsize | ndarray对象中单个元素大小。单位为字节 |
| ndarray.ndim | 维度数 |

5.4 数组切片与花式索引

本节将介绍数组切片与花式索引这两种在NumPy中常用的数组访问技巧。相信列表的切片操作对于较有经验的Python开发者来说并不陌生，它进一步丰富了我们访问列表的方式。首先，本节将介绍NumPy数组的切片操作，供对Python列表切片较为生疏的读者了解，而对于较为熟悉列表切片的读者，也可作为一次快捷的复习；之后我们将介绍花式索引，这是一种利用整数列表进行索引访问数组的技巧。

**5.4.1 数组切片**

切片访问数组的基本语法格式如下：

|  |
| --- |
| **var[lower:upper:step]** |

数组的切片访问方式允许用户通过指定下界和上界获取数组中指定部分，通过指定步长，则可以进一步确定要获取的数组元素的下标间隔值。需要注意的是，通过切片方式获得的数组遵循“左闭右开”规则，即该数组包括下标恰好为下界值的元素，但不包括下标恰好为上界值的元素，或用数学的语言描述，切片数组的范围为[下界， 上界)。

需要注意的是，负数同样可以作为访问数组元素的索引值。数组最后一个元素的索引值为-1，倒数第二个元素为-2，以此类推。通过这种方式可以方便地从后向前访问数组。

|  |
| --- |
| # 负数索引 -5 -4 -3 -2 -1  # 对应正数索引： 0 1 2 3 4  >>> a = np.array([10, 11, 12, 13, 14])  # [10, 11, 12, 13, 14]  >>> a[1:3]  array([11, 12])  # 指定步长的切片访问  >>> a[1:4:2]  array([11, 13])  # 也可通过负数索引访问数组  >>> a[1:-2]  array([11, 12])  >>> a[-4:3]  array([11, 12]) |

在切片访问数组时，可以省略对下界或上界的指定。若省略下界，则在切片时则认为下界为数组起点；若省略上界，则在切片时认为上界为数组终点；步长显然也可以省略，默认的步长为1，即依次访问界限区域内所有元素。

|  |
| --- |
| # 获取前三个元素  >>> a[:3]  array([10, 11, 12])  # 获取后两个元素  >>> a[-2:]  array([13, 14])  # 从数组中每隔两个元素取一个元素  >>> a[::2]  array([10, 12, 14]) |

**5.4.2 高维数组的切片**

对高维数组的切片访问相比于列表切片访问，在格式上略有区别。用户可对每个维度指定具体的切片方式。接下来以对如下二维数组a的访问为例进行说明。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

|  |
| --- |
| >>> a = np.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]])  # （1）行索引固定，列索引指定切片起止点  >>> a[0, 3:5]  array([3, 4])  # （2）行、列索引仅指定起点  >>> a[4:, 4:]  array([[24]])  # （3）行索引不指定起止点（即选中所有行），列索引固定  >>> a[:, 2]  array([ 2, 7, 12, 17, 22])  # （4）指定步长的切片  >>> a[2::2, ::2]  array([[10, 12, 14],  [20, 22, 24]]) |

在第一种切片访问方式中，行索引固定为0，而指定了列索引的起止点（3:5），因此取得了a[0, 3]和a[0, 4]两个元素；

在第二种切片访问方式中，行列索引都仅指定了起点，但因为4已经是行与列的末位，因此仅取得a[4, 4]即24；

在第三种切片访问方式中，行索引采用了切片访问，但既没有指定起点，也没有指定终点，此时即选中了所有行，同时列索引固定为2。因此可以取得a[0, 2],

a[1, 2], a[2, 2], a[3, 2], a[4, 2]五个元素构成的数组；

在第四种切片访问方式中，行索引指定了起点为2，未指定终点，步长设定为2，即行索引实际指定范围为[2, 4]；列索引未指定起点和终点，步长设定为2，即列索引实际指定范围为[0, 2, 4]。故最终可以取得a[2, 0], a[2, 2], a[2, 4],

a[4, 0], a[4, 2], a[4, 4]这六个元素构成的二维数组。

**5.4.3 切片的本质是引用**

切片的本质是对内存中原始数组的引用。对数组切片进行的修改最终都会影响到原始数组。我们来看下面一个例子：

|  |
| --- |
| >>> a = np.array([0, 1, 2, 3, 4])  # 创建一个包含两个a中元素的数组切片  >>> b = a[2:4]  >>> b  array([2, 3])  >>> b[0] = 10  # 对b的修改也会修改a  >>> a  array([ 0, 1, 10, 3, 4]) |

在上例中可以看到，将a的切片a[2:4]赋给b之后，对数组b进行修改的同时，原始数组a对应位置的元素也发生了同样的变化。

**5.4.4 花式索引**

除了切片以外，在NumPy中还有一种被称作“花式索引”（fancy indexing）的访问方式，实质上就是利用整数列表作为索引。我们首先来看花式索引在一维数组中的使用情形：

|  |
| --- |
| >>> a = np.arange(0, 80, 10)  >>> a  array([0, 10, 20, 30, 40, 50, 60, 70])  # 花式索引  >>> indices = [1, 2, -3]  >>> y = a[indices]  >>> y  array([10, 20, 50]) |

我们创建了一个整数列表indices，其中包含三个元素1，2和-3。并直接将列表名indices作为数组a的索引，将a[indices]赋值给y。可以看到，最终y是一个包含了10（即a[1]），20（即a[2]）和50（即a[-3]）的数组，也就是说我们根据列表indices所储存的整数获取了对应索引的数组元素。

除了使用整数列表作为索引外，还可以用布尔值列表作为索引。在这种使用方式下，布尔值数组起到了类似掩码的作用，布尔值为真的对应位将被传递，而布尔值为假的对应位将被“掩盖”，不会传递：

|  |
| --- |
| # 创建一个掩码数组，dtype参数设为bool表示该数组数据类型为布尔型  >>> mask1 = np.array([0, 1, 1, 0, 0, 1, 0, 0], dtype=bool)  >>> y = a[mask1]  >>> y  array([10, 20, 50]) |

在上面的例子中，我们创建了一个掩码数组，共包含8个元素，其中第2，3，6个元素为1（True），其余为0（False）。这样，把mask1作为索引，将a[mask1]赋值给y时，y将获得数组a中与mask1里取值为True的位置对应的数组元素构成的数组。

我们还可以用条件表达式生成掩码，并将其作为索引：

|  |
| --- |
| >>> mask2 = a < 40  >>> z = a[mask2]  >>> z  array([ 0, 10, 20, 30]) |

通过这种方式，可以方便地将a中所有小于40的元素存入数组z中。

**5.4.5 二维数组中的花式索引**

对于二维数组，同样可以使用花式索引访问，具体而言又有若干种不同的访问方式。下面以对如下表所示的二维数组的访问为例，对这些访问方式作逐一说明。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

第一种方式是在访问时，对于二维数组的行索引和列索引都使用列表表示：

|  |
| --- |
| >>> a[[0, 1, 2, 3, 4],  [1, 2, 3, 4, 5]]  array([1, 12, 23, 34, 45]) |

此时，行索引列表为[0, 1, 2, 3, 4]，列索引列表为[1, 2, 3, 4, 5]，花式索引取得的元素下标为两个列表对应位置的元素组合，即a[0, 1], a[1, 2], a[2, 3], a[3, 4], a[4, 5]。

第二种方式是使用列表切片进行访问：

|  |
| --- |
| >>> a[3:, [0, 2, 5]]  array([[30, 32, 35],  [40, 42, 45],  [50, 52, 55]]) |

当使用这种方式进行访问时，我们可以看到生成了一个3行2列的数组，其中第一行数组的元素为a[3, 0], a[3, 2], a[3, 5]；第二行数组的元素为a[4, 0],

a[4, 2], a[4, 5]，第三行数组的元素为a[5, 0], a[5, 2], a[5, 5]。可见，在上述例子中，“3：”指定了每一个行数组里元素的行索引，而[0, 2, 5]指定了每一个行数组里元素的列索引。

第三种方式是使用掩码数组进行访问：

|  |
| --- |
| >>> mask = np.array([1, 0, 1, 0, 0, 1], dtype=bool)  >>> a[mask, 2]  array([2, 22, 52]) |

在上例中，掩码数组mask保留了第0，2，5号元素，屏蔽了其他元素，mask作为行索引；2作为列索引。最终，a[mask, 2]得到了一个包含三个元素的数组，这三个元素的行索引即为mask保留的0，2，5，列索引均为2，即分别为a[0, 2], a[2, 2], a[5, 2]。

2．5 用函数创建NumPy数组

在5.3节，我们简单介绍了用构造函数array()创建NumPy数组的方法。除了这种方式外，NumPy还提供了丰富的数组初始化函数。本节将介绍一些常用的NumPy数组创建函数。

**5.5.1 arange()函数**

arange()函数的常用语法格式如下：

|  |
| --- |
| **arange([start,] stop[, step], dtype=None)** |

其用法与Python中的range()函数基本相同。它能创建一个数值范围为[start, stop)的，步长为step的数组。其中，起点start，终点stop，步长step均允许非整型数值（如下例3）。参数dtype的取值来源于起点、终点和步长的数据类型，因此若在输入参数时在整数后额外添加小数点，则起点和终点的数据类型变为浮点型，数组中元素的数据类型也为浮点型（如下例4）。

|  |
| --- |
| # （1）仅有一个参数，则起点默认为0  >>> np.arange(4)  array([0, 1, 2, 3])  # （2）指定起点和终点，步长默认值为1  >>> a = np.arange(2, 10)  >>> a  array([2, 3, 4, 5, 6, 7, 8, 9])  # （3）起点、终点、步长非整型  >>> np.arange(0, 2\*pi, pi/4)  array([0.000, 0.785, 1.571, 2.356, 3.142, 3.927, 4,712, 5.497])  # （4）创建元素类型为浮点型的数组  >>> b = np.arange(2., 10.)  >>> b  array([ 2., 3., 4., 5., 6., 7., 8., 9.]) |

**5.5.2 ones()与zeros()函数**

ones()和zeros()函数能够分别将数组初始化为全1或全0数组。

|  |
| --- |
| **ones(shape, dtype='float64')** |
| **zeros(shape, dtype='float64')** |

其中，shape是一个用于确定数组形状的整数序列。若数据类型参数dtype未被指定，则其默认值为float64。

|  |
| --- |
| # 用ones()初始化一个两行三列的全1二维数组  >>> np.ones((2, 3), dtype=’float32’)  array([[1., 1., 1.], [1., 1., 1.]], dtype=float32)  # 用zeros初始化一个三列的一维数组  >>> np.zeros(3)  array([0., 0., 0.]) |

**5.5.3 identity(), empty()和fill()函数**

identity()函数可以创建一个n阶单位矩阵数组，其中n由参数指定。数组元素的数据类型也可以通过参数dtype设置，dtype的默认值是float64。

|  |
| --- |
| **identity(n, dtype=’float64’)** |

下面是一组identity()函数的应用实例：

|  |
| --- |
| >>> a = np.identity(4)  >>> a  array([[ 1., 0., 0., 0.],  [ 0., 1., 0., 0.],  [ 0., 0., 1., 0.],  [ 0., 0., 0., 1.]])  >>> a.dtype  dtype(‘float64’)  >> np.identity(4, dtype=int)  array([[1, 0, 0, 0],  [0, 1, 0, 0],  [0, 0, 1, 0],  [0, 0, 0, 1]]) |

empty()函数可以创建一个指定形状与数据类型的空数组：

|  |
| --- |
| **empty(shape, dtype=float64, order='C')** |

其中，shape是一个用于确定数组形状的整数序列。若数据类型参数dtype未被指定，则其默认值为float64。参数order有两个选项：’C’（行优先）和’F’（列优先），即在计算机内存中元素的存储顺序。

而fill()方法则可以将一个数组所有元素填充为指定取值：

|  |
| --- |
| **fill(value)** |

其中value即为填充值。

|  |
| --- |
| # 创建一个一行两列的空数组  >>> a = np.empty(2)  >>> a  array([ 5.43230922e-312, 7.29112202e-304])  # 将a中所有元素填充为5.0  >>> a.fill(5.0)  >>> a  array([ 5., 5.])  # 也可以用如下数组切片方式实现数组填充为统一值，但速度更慢  >>> a[:] = 4.0  >>> a  array([ 4., 4.]) |

**5.5.4 linspace()和logspace()函数**

linspace()的函数原型如下：

|  |
| --- |
| **linspace(start, stop, num=50, endpoint=True,**  **retstep=False, dtype=‘float64’)** |

使用linspace()函数，可以创建一个以start为起点值，stop为终点值，包含num个元素的等差数组。endpoint参数可以设置该数组是否包含终点值stop，当取值为True则包含，False则不包含，默认值为True。retstep参数可以设置结果中是否显示出步长，True为显示，False则不显示，默认值为False。数据类型dtype默认为float64型。

|  |
| --- |
| # 创建从1到10，共10个元素的等差数组  >>> a = np.linspace(1, 10, 10)  >>> a  array([ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])  # 在等差数列中不包含终点值  >>> a = np.linspace(1, 10, 10, endpoint=False)  >>> a  array([ 1. , 1.9, 2.8, 3.7, 4.6, 5.5, 6.4, 7.3, 8.2, 9.1])  # 在结果中显示步长  >>> a = np.linspace(1, 10, 10, retstep=True)  >>> a  (array([ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]), 1.0)  # 查看数组数据类型  >>> a.dtype  dtype('float64') |

5.6 数组计算方法

总体而言，对数组的计算来说，有以下四条主要法则：

1. 在多个数组之间的操作需要首先进行数组形状匹配检查；
2. 数学运算符（如+，-，\*，/，exp，log等）是逐个应用于元素进行计算的；
3. 归约操作（如求均值mean，标准化std，求偏度skew，求和sum等）应用于整个数组，除非指定了某个轴；
4. 丢失值会被传递，除非它们被显式忽略（如nanmean，nansum等）。

在本节中，我们将介绍求数组的和、最值、均值、标准差、方差的几种常用函数或方法。

**5.6.1 使用sum()方法求和**

数组对象的sum()方法作用于存储于数组中的数据。在默认情况下，它将求出数组中所有元素相加的和。若提供了轴关键字，则sum()方法将计算指定轴的元素之和。轴关键字和数组索引一样，支持使用负整数索引。例如，当指定axis=-1时，则可以求出最后一个轴的元素之和：

|  |
| --- |
| >>> a = np.array([1, 2, 3], [4, 5, 6])  # 默认情况下，sum()方法直接求出数组中所有元素之和  >>> a.sum()  21  # 指定轴关键字时，sum()方法将计算该指定轴上所有元素之和  >>> a.sum(axis=0)  array([5, 7, 9])  # 轴关键字支持使用负数索引。如axis=-1表示选中最后一个轴  >>> a.sum(axis=-1)  array([ 6, 15]) |

**5.6.2 使用NumPy函数求数组的最值、均值、标准差和方差**

NumPy提供了min()和max()函数用于求数组的最大值和最小值。通过设置参数axis，可以选择针对某个数轴进行求最值。使用NumPy的min()和max()函数来代替Python内置的min()和max()可以在多维数组运算方面有更快的速度。类似地，NumPy也提供了argmin()和argmax()函数用于找到最小值和最大值元素的索引。

|  |
| --- |
| >>> a = np.array([2., 3., 0., 1.])  # 使用Python内置的min()  >>> a.min(axis=0)  0.0  # 使用NumPy的min()  >>> np.min(a,axis=0)  0.0  # 使用Python内置的argmax()  >>> a.argmax(axis=0)  1  # 使用NumPy的argmin函数  >>> np.argmax(a, axis=0)  1  >>> |

NumPy提供了mean()函数用于求数组的均值：

|  |
| --- |
| >>> a = np.array([[1, 2, 3], [4, 5, 6]])  # 指定axis=0，求出每一列的均值并构成数组  >>> a.mean(axis=0)  array([ 2.5, 3.5, 4.5])  >>> np.mean(a, axis=0)  array([ 2.5, 3.5, 4.5]) |

std()方法和var()方法分别可以用于求数组的标准差和方差。需要注意的是，对于std()方法，其默认是除以数组元素个数n的，得到的是有偏的标准差。若想得到无偏的结果，可以将参数ddof设置为1：

|  |
| --- |
| >>> a = np.array([[1, 2, 3], [4, 5, 6]])  # 求标准差  >>> a.std(axis=0)  array([ 1.5, 1.5, 1.5])  # 设置ddof=1，求无偏标准差  >>> a.std(axis=0, ddof=1)  array([ 2.12132034, 2.12132034, 2.12132034])  # 求方差  >>> a.var(axis=0)  array([ 2.25, 2.25, 2.25])  >>> np.var(a, axis=0)  array([ 2.25, 2.25, 2.25])  >>> |

5.7 数组广播

具有不同维度的NumPy数组可以通过相同的表达式结合起来。此时，维度较小的数组将会被广播以与维度较大的数组相匹配（而不是复制大数组的数据）。

**5.7.1 数组广播的规则**

数组广播的规则主要有两条，如下所示：

1. 向形状（即shape）较小的数组开头补1；

|  |
| --- |
| >>> import numpy as np  >>> a = np.ones((3, 5)) # a.shape == (3, 5)  >>> b = np.ones((5, )) # b.shape == (5,)  >>> b.reshape(1, 5) # 得到一个(1, 5)形状的数组  >>> b[np.newaxis, :] # 与上一条语句等效 |

第二，维度大小为1的数组在广播时会重复自身内容，而不是复制大数组的内容。

|  |
| --- |
| >>> c = a + b # c.shape == (3, 5)  # 在逻辑上等效于……  >>> tmp\_b = b.reshape(1, 5)  >>> tmp\_b\_repeat = tmp\_b.repeat(3, axis=0)  >>> c = a + tmp\_b\_repeat  # 但广播时不会产生b中数据的副本 |

**5.7.2 数组广播的实例：到中心点的距离**

下面给出一个利用数组广播绘图的例子。我们将通过matplotlib库进行绘图，用色块深浅表示与中心距离的远近。

|  |
| --- |
| >>> import numpy as np  >>> import matplotlib.pyplot as plt # 导入matplotlib  >>> a = np.linspace(0, 1, 15) - 0.5  >>> b = a[:, np.newaxis] # b.shape == (15, 1)  >>> dist2 = a\*\*2 + b\*\*2 # 广播sum  >>> dist = np.sqrt(dist2)  >>> plt.imshow(dist)  >>> plt.show() |

绘图结果如下所示。距中心点越近的，色块颜色越深；反之则越浅。

图片包含 屏幕截图

描述已自动生成

5.8 NumPy运算符通用方法

对于数学、比较、逻辑和位运算等各类二元运算符而言，在它们应用于数组时，会有一些特殊的方法，本节将介绍这些方法。在后文中提到这些二元运算符时，我们将用op作为它们的泛称。

**5.8.1 op.reduce()**

op.reduce(a)会将运算符op应用于一维数组a中所有的单个元素，将a数组减少（reduce）到只有单个值。例如对于add.reduce(a)用数学语言可表达为：

y = add.reduce(a) = = a[0] + a[1] + … + a[N – 1]

对加法add调用add.reduce(a)，其中参数a为一维数组，则会通过将add（加法）运算应用于a中所有单个元素的方式（即将所有a的元素相加），对a求和，如此将a“减少到”单个值。

下面给出在NumPy中，运算符调用reduce()方法若干示例：

|  |
| --- |
| # 对add（加法）调用reduce()方法  >>> a = np.array([1, 2, 3, 4])  >>> np.add.reduce(a)  10  # 对add调用reduce()方法，并作用于字符串对象  >>> a = np.array(['ab', 'cd', 'ef'], dtype='object')  >>> np.add.reduce(a)  'abcdef'  # 对逻辑运算调用reduce()方法  >>> a = np.array([1, 1, 0, 1])  >>> np.logical\_and.reduce(a) # 逻辑和  False  >>> np.logical\_or.reduce(a) # 逻辑或  True  >>> np.logical\_xor.reduce(a) # 异或  True |

对多维数组而言，上述reduce()方法的一个重载，op.reduce(a, axis)方法可以将运算符op作用于数组a的轴axis上，从而得到一个比数组a维度小1的数组。参数axis的默认值是0。

|  |
| --- |
| >>> a = np.array([[0, 1, 2],[10, 11, 12],[20, 21, 22],[30, 31, 32]])  >>> np.add.reduce(a, 1)  array([ 3, 33, 63, 93]) # 从二维数组降维为一维数组 |

**5.8.2 op.accumulate()**

op.accumulate(a)创建一个新数组，该数组的元素为数组a中所有元素执行reduce运算以后的中值。例如，add.accumulate(a)用数学语言可表达为：

y = add.accumulate(a)

=

下面给出accumulate()方法在运算add，字符串列表，以及逻辑运算符中的应用实例。

|  |
| --- |
| # 加法add实例  >>> import numpy as np  >>> a = np.array([1, 2, 3, 4])  >>> np.add.accumulate(a)  array([ 1, 3, 6, 10])  # 字符串列表实例  >>> a = np.array([‘ab’,’cd’,’ef’], dtype=’object’)  >>> np.add.accumulate(a)  array([ab, abcd, abcdef], dtype=object)  # 逻辑运算符实例  >>> a = np.array([1,1,0])  >>> np.logical\_and.accumulate(a)  array([True, True, False])  >>> np.logical\_or.accumulate(a)  array([True, True, True]) |

对加法运算add调用accumulate()方法，将数组a（[1, 2, 3, 4]）作为参数，我们得到了一个包含4个元素的一维数组，其中每个元素依次为原数组a的前1个、2个、3个和4个元素之和。

对字符串列表调用accumulate()方法，将字符串列表a（[‘ab’,’cd’,’ef’]）作为参数，我们得到一个包含3个元素的一维数组，其中每个元素依次为原列表a的前1个、2个和3个字符串拼接所得总串。

对逻辑运算符逻辑与and和逻辑或or调用accumulate()方法，将数组a（[1,1,0]）作为参数，分别得到[True, True, False]和[True, True, True]。前者为对原数组a的前1个、2个、3个元素进行逻辑与/逻辑或操作所得布尔值。

**5.8.3 op.reduceat()**

op.reduceat(a, indices)接受两个参数输入：一维数组a和索引列表indices。该方法将得到一个长度与indices相同的数组y，其中每个元素y[i]用数学语言表示为（以下运算符op以加法add为例）：

y[i] = add.reduceat(a, indices)

=

下面给出一个reduceat()方法的应用实例：

|  |
| --- |
| >>> a = np.array([0, 10, 20, 30, 40, 50])  >>> indices = np.array([1, 4])  >>> np.add.reduceat(a, indices)  array([60, 90], dtype=int32) |

可以看到，原始数组a是一个长度为6的一维数组，而indices数组是长度为2的一维数组[1, 4]。add.reduceat(a, indices)方法返回了一个长度与indices相同的数组，其中第一个元素为a[1]到a[3]的和，第二个元素为a[4]到数组末位元素a[5]的和。

注意，对于高维数组而言，reduceat()总是应用于最后一个轴（例如对二维数组而言是行和）。这与reduce()和accumulate()的默认机制是不同的。

**5.8.4 op.outer()**

op.outer(a, b)将会用运算符op形成数组a和数组b元素之间所有可能的组合。所得结果数组的形状为数组a和b相连结的结果（a，b顺序不同，形状也将不同）。

例如对长度为4的一维数组a和长度为3的二维数组b而言，执行np.add.outer(a, b)所得数组如下：

|  |  |  |
| --- | --- | --- |
| a[0]+b[0] | a[0]+b[1] | a[0]+b[2] |
| a[1]+b[0] | a[1]+b[1] | a[1]+b[2] |
| a[2]+b[0] | a[2]+b[1] | a[2]+b[2] |
| a[3]+b[0] | a[3]+b[1] | a[3]+b[2] |

执行np.add.outer(b, a)所得数组如下：

|  |  |  |  |
| --- | --- | --- | --- |
| b[0]+a[0] | b[0]+a[1] | b[0]+a[2] | b[0]+a[3] |
| b[1]+a[0] | b[1]+a[1] | b[1]+a[2] | b[1]+a[3] |
| b[2]+a[0] | b[2]+a[1] | b[2]+a[2] | b[2]+a[3] |

2．9 数组结构操作方法

本节将介绍几个常用的数组结构操作方法。这些方法仅对数组的结构进行修改，而不改变数组内存储的数据。进行这类操作不需要复制内存中的数据。

**5.9.1 transpose()方法**

transpose()方法，顾名思义，将会返回当前数组的一个转置（transpose）数组。下面给出一个transpose()方法的应用实例：

|  |
| --- |
| >>> a = np.array([[0, 1, 2], [3, 4, 5]])  >>> a.shape # 转置前数组形状  (2, 3)  >>> a.transpose()  array([[0, 3],  [1, 4],  [2, 5]])  >>> a.transpose().shape # 转置数组形状  (3, 2) |

**5.9.2 reshape()方法与对shape属性赋值**

reshape()方法可以在不改变数组内数据的情况下按照给定的参数改变原数组的形状：

|  |
| --- |
| >>> a = np.array([[0, 1, 2], [3, 4, 5]])  >>> a.reshape(3, 2) # 返回一个指定形状的新数组  array([[0, 1],  [2, 3],  [4, 5]])  >>> a.reshape(4, 2) # reshape不能改变数组元素个数  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  ValueError: cannot reshape array of size 6 into shape (4,2) |

需要注意的是，在用参数指定数组的新形状时，必须保证新数组的元素个数与原数组相同。如上例中尝试将包含6个元素的数组a变形为4\*2数组时，解释器将会报错。

除调用reshape()方法外，另一种改变数组形状的方式是直接修改数组的shape属性从而改变数组形状：

|  |
| --- |
| >>> a = np.arange(6)  >>> a  array([0, 1, 2, 3, 4, 5])  >>> a.shape  (6,)  >>> a.shape = (2, 3) # 直接对shape属性赋值以修改之  >>> a  array([[0, 1, 2],  [3, 4, 5]]) |

在上例中，我们新建了一个一维数组a，其初始形状（shape属性值）为(6,)。随后我们直接对其shape属性进行赋值操作，将元组(2, 3)赋值给a.shape以修改a的形状，这是由于在Python中对元组只能通过重新赋值进行修改，而不能单独修改其中元素。再次调出a，可以看到其已经变成一个3行2列的数组。

**5.9.3 数组扁平化**

数组扁平化主要是指将高维度的数组通过某种方法转化为一维数组。常用的方法有两个，一种是较为安全的flatten()方法，一种是效率较高的ravel()方法。

flatten()方法可以将一个高维数组转化为一维数组。新数组的数据内容由原始数组复制而来。下面给出一个使用flatten()方法的代码实例。

|  |
| --- |
| # 创建一个二维数组a  >>> a = np.array([[0, 1], [2, 3]])  # 将二维数组a扁平化为一维数组b  >>> b = a.flatten()  >>> b  array([0, 1, 2, 3])  # 修改b并不会影响原始数组a  >>> b[0] = 10  >>> b  array([10, 1, 2, 3])  >>> a  array([[0, 1],  [2, 3]]) |

在上面的例子中，我们创建了一个二维数组a，并使用flatten()方法创建了一个a的扁平化一维数组b。由于b中的数据是由a复制而来的，所以我们在修改b时并不会影响a。

另一个将数组扁平化的方法是ravel()，其作用与flatten()相近，但在可能的情况下（内存相邻时），它将返回一个数组的引用。否则新数组将和flatten()一样是原始数组的拷贝：

|  |
| --- |
| # 将数组a扁平化为一维数组b  >>> a = np.array([[0, 1], [2, 3]])  >>> b = a.ravel()  # 对b进行修改也会修改a  >>> b[0] = 10  >>> b  array([10, 1, 2, 3])  >>> a  array([[10, 1],  [ 2, 3]]) |

观察上例可以看到，在使用ravel()方法对高维数组进行扁平化时，对扁平化后的数组b进行修改时，原始数组a也被相应修改，由此可知ravel()方法此时返回的扁平化数组是原始数组的一个引用。

#end