2 NumPy-快速处理数据

标准安装的Python中用列表(list)保存一组值，可以用来当作数组使用，不过由于列表的元素可以是任何对象，因此列表中所保存的是对象的指针。这样为了保存一个简单的[1,2,3]，需要有3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还提供了一个array模块，array对象和列表不同，它直接保存数值，和C语言的一维数组比较类似。但是由于它不支持多维，也没有各种运算函数，因此也不适合做数值运算。

NumPy的诞生弥补了这些不足，NumPy提供了两种基本的对象：ndarray（N-dimensional array object）和 ufunc（universal function object）。ndarray(下文统一称之为数组)是存储单一数据类型的多维数组，而ufunc则是能够对数组进行处理的函数。

2.1 ndarray对象

函数库的导入

本书的示例程序假设用以下推荐的方式导入NumPy函数库:

```
import numpy as np
```

2.1.1 创建

首先需要创建数组才能对其进行其它操作。

我们可以通过给array函数传递Python的序列对象创建数组，如果传递的是多层嵌套的序列，将创建多维数组(下例中的变量c):

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array((5, 6, 7, 8))
>>> c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
>>> b
array([5, 6, 7, 8])
>>> c
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])
>>> c.dtype
dtype('int32')
```

数组的大小可以通过其shape属性获得：

```
>>> a.shape
(4,)
```

```
>>> c.shape
(3, 4)
```

数组a的shape只有一个元素，因此它是一维数组。而数组c的shape有两个元素，因此它是二维数组，其中第0轴的长度为3，第1轴的长度为4。还可以通过修改数组的shape属性，在保持数组元素个数不变的情况下，改变数组每个轴的长度。下面的例子将数组c的shape改为(4,3)，注意从(3,4)改为(4,3)并不是对数组进行转置，而只是改变每个轴的大小，数组元素在内存中的位置并没有改变：

```
>>> c.shape = 4,3
>>> c
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

当某个轴的元素为-1时，将根据数组元素的个数自动计算此轴的长度，因此下面的程序将数组c的shape改为了(2,6)：

```
>>> c.shape = 2,-1
>>> c
array([[ 1,  2,  3,  4,  4,  5],
       [ 6,  7,  7,  8,  9, 10]])
```

使用数组的reshape方法，可以创建一个改变了尺寸的新数组，原数组的shape保持不变：

```
>>> d = a.reshape((2,2))
>>> d
array([[1, 2],
       [3, 4]])
>>> a
array([1, 2, 3, 4])
```

数组a和d其实共享数据存储内存区域，因此修改其中任意一个数组的元素都会同时修改另外一个数组的内容：

```
>>> a[1] = 100 # 将数组a的第一个元素改为100
>>> d # 注意数组d中的2也被改变了
array([[  1, 100],
       [  3,   4]])
```

数组的元素类型可以通过dtype属性获得。上面例子中的参数序列的元素都是整数，因此所创建的数组的元素类型也是整数，并且是32bit的长整型。可以通过dtype参数在创建时指定元素类型:

```
>>> np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]], dtype=np.float)
array([[  1.,   2.,   3.,   4.],
       [  4.,   5.,   6.,   7.],
       [  7.,   8.,   9.,  10.]])
```

```
>>> np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]], dtype=np.complex)
array([[  1.+0.j,   2.+0.j,   3.+0.j,   4.+0.j],
       [  4.+0.j,   5.+0.j,   6.+0.j,   7.+0.j],
       [  7.+0.j,   8.+0.j,   9.+0.j,  10.+0.j]])
```

上面的例子都是先创建一个Python序列，然后通过array函数将其转换为数组，这样做显然效率不高。因此NumPy提供了很多专门用来创建数组的函数。下面的每个函数都有一些关键字参数，具体用法请查看函数说明。

- arange函数类似于python的range函数，通过指定开始值、终值和步长来创建一维数组，注意数组不包括终值:

```
>>> np.arange(0,1,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

- linspace函数通过指定开始值、终值和元素个数来创建一维数组，可以通过endpoint关键字指定是否包括终值，缺省设置是包括终值:

```
>>> np.linspace(0, 1, 12)
array([ 0.        ,  0.09090909,  0.18181818,  0.27272727,  0.36363636,
        0.45454545,  0.54545455,  0.63636364,  0.72727273,  0.81818182,
        0.90909091,  1.        ])
```

- logspace函数和linspace类似，不过它创建等比数列，下面的例子产生1(10^0)到100(10^2)、有20个元素的等比数列:

```
>>> np.logspace(0, 2, 20)
array([   1.        ,    1.27427499,    1.62377674,    2.06913808,
          2.6366509 ,    3.35981829,    4.2813324 ,    5.45559478,
          6.95192796,    8.8586679 ,   11.28837892,   14.38449888,
         18.32980711,   23.35721469,   29.76351442,   37.92690191,
         48.32930239,   61.58482111,   78.47599704,  100.        ])
```

此外，使用frombuffer, fromstring, fromfile等函数可以从字节序列创建数组，下面以fromstring为例:

```
>>> s = "abcdefgh"
```

Python的字符串实际上是字节序列，每个字符占一个字节，因此如果从字符串s创建一个8bit的整数数组的话，所得到的数组正好就是字符串中每个字符的ASCII编码:

```
>>> np.fromstring(s, dtype=np.int8)
array([ 97,  98,  99, 100, 101, 102, 103, 104], dtype=int8)
```

如果从字符串s创建16bit的整数数组，那么两个相邻的字节就表示一个整数，把字节98和字节97当作一个16位的整数，它的值就是98*256+97 = 25185。可以看出内存中是以little endian(低位字节在前)方式保存数据的。

```
>>> np.fromstring(s, dtype=np.int16)
array([25185, 25699, 26213, 26727], dtype=int16)
>>> 98*256+97
25185
```

如果把整个字符串转换为一个64位的双精度浮点数数组，那么它的值是:

```
>>> np.fromstring(s, dtype=np.float)
array([  8.54088322e+194])
```

显然这个例子没有什么意义，但是可以想象如果我们用C语言的二进制方式写了一组double类型的数值到某个文件中，那们可以从此文件读取相应的数据，并通过fromstring函数将其转换为float64类型的数组。

我们可以写一个Python的函数，它将数组下标转换为数组中对应的值，然后使用此函数创建数组：

```
>>> def func(i):
...   return i%4+1
...
>>> np.fromfunction(func, (10,))
array([ 1.,  2.,  3.,  4.,  1.,  2.,  3.,  4.,  1.,  2.])
```

fromfunction函数的第一个参数为计算每个数组元素的函数，第二个参数为数组的大小(shape)，因为它支持多维数组，所以第二个参数必须是一个序列，本例中用(10,)创建一个10元素的一维数组。

下面的例子创建一个二维数组表示九九乘法表，输出的数组a中的每个元素a[i, j]都等于func2(i, j)：

```
>>> def func2(i, j):
...     return (i+1) * (j+1)
...
>>> a = np.fromfunction(func2, (9,9))
>>> a
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [  2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.],
       [  3.,   6.,   9.,  12.,  15.,  18.,  21.,  24.,  27.],
       [  4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.],
       [  5.,  10.,  15.,  20.,  25.,  30.,  35.,  40.,  45.],
       [  6.,  12.,  18.,  24.,  30.,  36.,  42.,  48.,  54.],
       [  7.,  14.,  21.,  28.,  35.,  42.,  49.,  56.,  63.],
       [  8.,  16.,  24.,  32.,  40.,  48.,  56.,  64.,  72.],
       [  9.,  18.,  27.,  36.,  45.,  54.,  63.,  72.,  81.]])
```

2.1.2 存取元素

数组元素的存取方法和Python的标准方法相同：

```
>>> a = np.arange(10)
>>> a[5]    # 用整数作为下标可以获取数组中的某个元素
5
>>> a[3:5]  # 用范围作为下标获取数组的一个切片，包括a[3]不包括a[5]
array([3, 4])
>>> a[:5]   # 省略开始下标，表示从a[0]开始
array([0, 1, 2, 3, 4])
>>> a[:-1]  # 下标可以使用负数，表示从数组后往前数
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> a[2:4] = 100,101    # 下标还可以用来修改元素的值
>>> a
array([  0,   1, 100, 101,   4,   5,   6,   7,   8,   9])
>>> a[1:-1:2]   # 范围中的第三个参数表示步长，2表示隔一个元素取一个元素
array([  1, 101,   5,   7])
>>> a[::-1] # 省略范围的开始下标和结束下标，步长为-1，整个数组头尾颠倒
array([  9,   8,   7,   6,   5,   4, 101, 100,   1,   0])
>>> a[5:1:-2] # 步长为负数时，开始下标必须大于结束下标
array([  5, 101])
```

和Python的列表序列不同，通过下标范围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空间：

```
>>> b = a[3:7] # 通过下标范围产生一个新的数组b，b和a共享同一块数据空间
>>> b
array([101,   4,   5,   6])
>>> b[2] = -10 # 将b的第2个元素修改为-10
>>> b
array([101,   4, -10,   6])
>>> a # a的第5个元素也被修改为10
array([  0,   1, 100, 101,   4, -10,   6,   7,   8,   9])
```

除了使用下标范围存取元素之外，NumPy还提供了两种存取元素的高级方法。

使用整数序列

当使用整数序列对数组元素进行存取时，将使用整数序列中的每个元素作为下标，整数序列可以是列表或者数组。使用整数序列作为下标获得的数组不和原始数组共享数据空间。

```
>>> 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
array([7, 7, 9, 2])
>>> b = x[np.array([3,3,-3,8])]  #下标可以是负数
>>> b[2] = 100
>>> b
array([7, 7, 100, 2])
>>> x   # 由于b和x不共享数据空间，因此x中的值并没有改变
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3,5,1]] = -1, -2, -3 # 整数序列下标也可以用来修改元素的值
>>> x
array([10, -3,  8, -1,  6, -2,  4,  3,  2])
```

使用布尔数组

当使用布尔数组b作为下标存取数组x中的元素时，将收集数组x中所有在数组b中对应下标为True的元素。使用布尔数组作为下标获得的数组不和原始数组共享数据空间，注意这种方式只对应于布尔数组，不能使用布尔列表。

```
>>> x = np.arange(5,0,-1)
>>> x
array([5, 4, 3, 2, 1])
>>> x[np.array([True, False, True, False, False])]
>>> # 布尔数组中下标为0，2的元素为True，因此获取x中下标为0,2的元素
array([5, 3])
>>> x[[True, False, True, False, False]]
>>> # 如果是布尔列表，则把True当作1, False当作0，按照整数序列方式获取x中的元素
array([4, 5, 4, 5, 5])
>>> x[np.array([True, False, True, True])]
>>> # 布尔数组的长度不够时，不够的部分都当作False
array([5, 3, 2])
>>> x[np.array([True, False, True, True])] = -1, -2, -3
>>> # 布尔数组下标也可以用来修改元素
>>> x
array([-1,  4, -2, -3,  1])
```

布尔数组一般不是手工产生，而是使用布尔运算的ufunc函数产生，关于ufunc函数请参照ufunc运算 一节。

```
>>> x = np.random.rand(10) # 产生一个长度为10，元素值为0-1的随机数的数组
>>> x
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.2055047 ,  0.17567449,
        0.95799412,  0.12015178,  0.7627083 ,  0.43260184,  0.91379859])
>>> x>0.5
>>> # 数组x中的每个元素和0.5进行大小比较，得到一个布尔数组，True表示x中对应的值大于0.5
array([ True,  True,  True, False, False,  True, False,  True, False,  True], dtype=bool)
>>> x[x>0.5]
>>> # 使用x>0.5返回的布尔数组收集x中的元素，因此得到的结果是x中所有大于0.5的元素的数组
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.95799412,  0.7627083 ,
        0.91379859])
```

2.1.3 多维数组

多维数组的存取和一维数组类似，因为多维数组有多个轴，因此它的下标需要用多个值来表示，NumPy采用组元(tuple)作为数组的下标。如图2.1所示，a为一个6x6的数组，图中用颜色区分了各个下标以及其对应的选择区域。

> 组元不需要圆括号
>
> 虽然我们经常在Python中用圆括号将组元括起来，但是其实组元的语法定义只需要用逗号隔开即可，例如 x,y=y,x 就是用组元交换变量值的一个例子。

```
>>> a[0,3:5]
array([3,4])
>>> a[4:,4:]
array([[44,45],[54,55]])
>>> a[:,2]
array([2,12,22,32,42,52])
>>> a[2::2,::2]
array([[20,22,24],
       [40,42,44]])
```

图**2.1** 使用数组切片语法访问多维数组中的元素

如何创建这个数组

你也许会对如何创建a这样的数组感到好奇，数组a实际上是一个加法表，纵轴的值为0, 10, 20, 30, 40, 50；横轴的值为0, 1, 2, 3, 4, 5。纵轴的每个元素都和横轴的每个元素求和，就得到图中所示的数组a。你可以用下面的语句创建它，至于其原理我们将在后面的章节进行讨论：

```
>>> np.arange(0, 60, 10).reshape(-1, 1) + np.arange(0, 6)
array([[ 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]])
```

多维数组同样也可以使用整数序列和布尔数组进行存取。

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([1,12,23,34,45])
>>> a[3:,[0,2,5]]
array([[30,32,35],
       [40,42,45],
       [50,52,55]])
>>> mask=np.array([1,0,1,0,0,1],
                  dtype=np.bool)
>>> a[mask,2]
array([2,22,52])
```

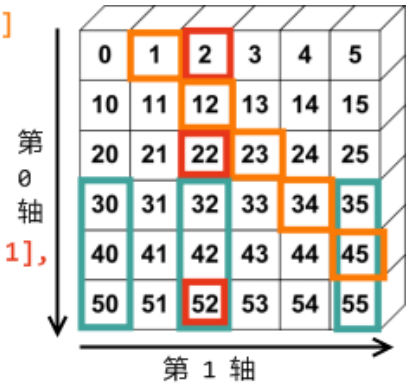

图**2.2** 使用整数序列和布尔数组访问多维数组中的元素

- a[(0,1,2,3,4),(1,2,3,4,5)] : 用于存取数组的下标和仍然是一个有两个元素的组元，组元中的每个元素都是整数序列，分别对应数组的第0轴和第1轴。从两个序列的对应位置取出两个整数组成下标： a[0,1], a[1,2], ..., a[4,5]。

- a[3:, [0, 2, 5]] : 下标中的第0轴是一个范围，它选取第3行之后的所有行；第1轴是整数序列，它选取第0, 2, 5三列。
- a[mask, 2] : 下标的第0轴是一个布尔数组，它选取第0，2，5行；第1轴是一个整数，选取第2列。

2.1.4 结构数组

在C语言中我们可以通过struct关键字定义结构类型，结构中的字段占据连续的内存空间，每个结构体占用的内存大小都相同，因此可以很容易地定义结构数组。和C语言一样，在NumPy中也很容易对这种结构数组进行操作。只要NumPy中的结构定义和C语言中的定义相同，NumPy就可以很方便地读取C语言的结构数组的二进制数据，转换为NumPy的结构数组。

假设我们需要定义一个结构数组，它的每个元素都有name, age和weight字段。在NumPy中可以如下定义：

```
import numpy as np
persontype = np.dtype({
    'names':['name', 'age', 'weight'],
    'formats':['S32','i', 'f']})
a = np.array([("Zhang",32,75.5),("Wang",24,65.2)],
    dtype=persontype)
```

我们先创建一个dtype对象persontype，通过其字典参数描述结构类型的各个字段。字典有两个关键字：names，formats。每个关键字对应的值都是一个列表。names定义结构中的每个字段名，而formats则定义每个字段的类型：

- **S32** : 32个字节的字符串类型，由于结构中的每个元素的大小必须固定，因此需要指定字符串的长度
- **i** : 32bit的整数类型，相当于np.int32
- **f** : 32bit的单精度浮点数类型，相当于np.float32

然后我们调用array函数创建数组，通过关键字参数 dtype=persontype， 指定所创建的数组的元素类型为结构persontype。运行上面程序之后，我们可以在IPython中执行如下的语句查看数组a的元素类型

```
>>> a.dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

这里我们看到了另外一种描述结构类型的方法： 一个包含多个组元的列表，其中形如 (字段名, 类型描述) 的组元描述了结构中的每个字段。类型描述前面为我们添加了 '|', '<' 等字符，这些字符用来描述字段值的字节顺序：

- **|** : 忽视字节顺序
- **<** : 低位字节在前
- **>** : 高位字节在前

结构数组的存取方式和一般数组相同，通过下标能够取得其中的元素，注意元素的值看上去像是组元，实际上它是一个结构：

```
>>> a[0]
('Zhang', 32, 75.5)
>>> a[0].dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

a[0]是一个结构元素，它和数组a共享内存数据，因此可以通过修改它的字段，改变原始数组中的对应字段：

```
>>> c = a[1]
>>> c["name"] = "Li"
>>> a[1]["name"]
"Li"
```

结构像字典一样可以通过字符串下标获取其对应的字段值：

```
>>> a[0]["name"]
'Zhang'
```

我们不但可以获得结构元素的某个字段，还可以直接获得结构数组的字段，它返回的是原始数组的视图，因此可以通过修改b[0]改变a[0]["age"]：

```
>>> b=a[:]["age"] # 或者a["age"]
>>> b
array([32, 24])
>>> b[0] = 40
>>> a[0]["age"]
40
```

通过调用a.tostring或者a.tofile方法，可以直接输出数组a的二进制形式：

```
>>> a.tofile("test.bin")
```

利用下面的C语言程序可以将test.bin文件中的数据读取出来。

> 内存对齐
>
> C语言的结构体为了内存寻址方便，会自动的添加一些填充用的字节，这叫做内存对齐。例如如果把下面的name[32]改为name[30]的话，由于内存对齐问题，在name和age中间会填补两个字节，最终的结构体大小不会改变。因此如果numpy中的所配置的内存大小不符合C语言的对齐规范的话，将会出现数据错位。为了解决这个问题，在创建dtype对象时，可以传递参数align=True，这样numpy的结构数组的内存对齐和C语言的结构体就一致了。

```
#include <stdio.h>
```

```
struct person
{
    char name[32];
    int age;
    float weight;
};

struct person p[2];

void main ()
{
    FILE *fp;
    int i;
    fp=fopen("test.bin","rb");
    fread(p, sizeof(struct person), 2, fp);
    fclose(fp);
    for(i=0;i<2;i++)
        printf("%s %d %f\n", p[i].name, p[i].age, p[i].weight);
    getchar();
}
```

结构类型中可以包括其它的结构类型，下面的语句创建一个有一个字段f1的结构，f1的值是另外一个结构，它有字段f2，其类型为16bit整数。

```
>>> np.dtype([('f1', [('f2', np.int16)])])
dtype([('f1', [('f2', '<i2')])])
```

当某个字段类型为数组时，用组元的第三个参数表示，下面描述的f1字段是一个shape为(2,3)的双精度浮点数组:

```
>>> np.dtype([('f0', 'i4'), ('f1', 'f8', (2, 3))])
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

用下面的字典参数也可以定义结构类型，字典的关键字为结构中字段名，值为字段的类型描述，但是由于字典的关键字是没有顺序的，因此字段的顺序需要在类型描述中给出，类型描述是一个组元，它的第二个值给出字段的字节为单位的偏移量，例如age字段的偏移量为25个字节：

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

2.1.5 内存结构

下面让我们来看看ndarray数组对象是如何在内存中储存的。如图**2.3**所示，关于数组的描述信息保存在一个数据结构中，这个结构引用两个对象：一块用于保存数据的存储区域和一个用于描述元素类型的dtype对象。

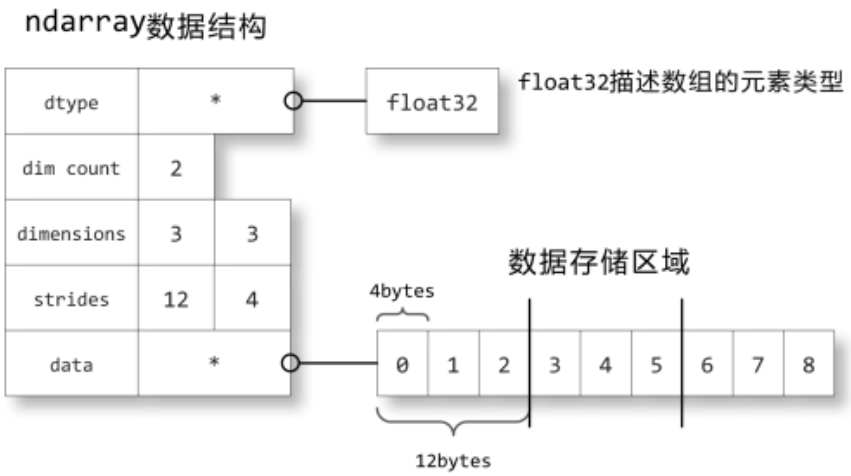

图**2.3** ndarray数组对象在内存中的储存方式

数据存储区域保存着数组中所有元素的二进制数据，dtype对象则知道如何将元素的二进制数据转换为可用的值。数组的维数、大小等信息都保存在ndarray数组对象的数据结构中。图中显示的是如下数组的内存结构：

```
>>> a = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32)
```

strides中保存的是当每个轴的下标增加1时，数据存储区中的指针所增加的字节数。例如图中的strides为12,4，即第0轴的下标增加1时，数据的地址增加12个字节：即a[1,0]的地址比a[0,0]的地址要高12个字节，正好是3个单精度浮点数的总字节数；第1轴下标增加1时，数据的地址增加4个字节，正好是单精度浮点数的字节数。

如果strides中的数值正好和对应轴所占据的字节数相同的话，那么数据在内存中是连续存储的。然而数据并不一定都是连续储存的，前面介绍过通过下标范围得到新的数组是原始数组的视图，即它和原始视图共享数据存储区域：

```
>>> b = a[::2,::2]
>>> b
array([[ 0.,  2.],
       [ 6.,  8.]], dtype=float32)
>>> b.strides
(24, 8)
```

由于数组b和数组a共享数据存储区，而b中的第0轴和第1轴都是数组a中隔一个元素取一个，因此数组b的strides变成了24,8，正好都是数组a的两倍。 对照前面的图很容易看出数据0和2的地址相差8个字节，而0和6的地址相差24个字节。

元素在数据存储区中的排列格式有两种：C语言格式和Fortan语言格式。在C语言中，多维数组的第0轴是最上位的，即第0轴的下标增加1时，元素的地址增加的字节数最多；而Fortan语言的多维数组的第0轴是最下位的，即第0轴的下标增加1时，地址只增加一个元素的字节数。在NumPy中，元素在内存中的排列缺省是以C语言格式存储的，如果你希望改为Fortan格式的话，只需要给数组传递order="F"参数：

```
>>> c = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32, order="F")
>>> c.strides
(4, 12)
```

2.2 ufunc运算

ufunc是universal function的缩写，它是一种能对数组的每个元素进行操作的函数。NumPy内置的许多ufunc函数都是在C语言级别实现的，因此它们的计算速度非常快。让我们来看一个例子：

```
>>> x = np.linspace(0, 2*np.pi, 10)
# 对数组x中的每个元素进行正弦计算，返回一个同样大小的新数组
>>> y = np.sin(x)
>>> y
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44921271e-16])
```

先用linspace产生一个从0到2*PI的等距离的10个数，然后将其传递给sin函数，由于np.sin是一个ufunc函数，因此它对x中的每个元素求正弦值，然后将结果返回，并且赋值给y。计算之后x中的值并没有改变，而是新创建了一个数组保存结果。如果我们希望将sin函数所计算的结果直接覆盖到数组x上去的话，可以将要被覆盖的数组作为第二个参数传递给ufunc函数。例如：：

```
>>> t = np.sin(x,x)
>>> x
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44921271e-16])
>>> id(t) == id(x)
True
```

sin函数的第二个参数也是x，那么它所做的事情就是对x中的每给值求正弦值，并且把结果保存到x中的对应的位置中。此时函数的返回值仍然是整个计算的结果，只不过它就是x，因此两个变量的id是相同的(变量t和变量x指向同一块内存区域)。

我用下面这个小程序，比较了一下numpy.math和Python标准库的math.sin的计算速度：：

```
import time
import math
import numpy as np

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = math.sin(t)
print "math.sin:", time.clock() - start
```

```
x = [i * 0.001 for i in xrange(1000000)]
x = np.array(x)
start = time.clock()
np.sin(x,x)
print "numpy.sin:", time.clock() - start

# 输出
# math.sin: 1.15426932753
# numpy.sin: 0.0882399858083
```

在我的电脑上计算100万次正弦值，numpy.sin比math.sin快10倍多。这得利于numpy.sin在C语言级别的循环计算。numpy.sin同样也支持对单个数值求正弦，例如：numpy.sin(0.5)。不过值得注意的是，对单个数的计算math.sin则比numpy.sin快得多了，让我们看下面这个测试程序：

```
x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = np.sin(t)
print "numpy.sin loop:", time.clock() - start

# 输出
# numpy.sin loop: 5.72166965355
```

请注意numpy.sin的计算速度只有math.sin的1/5。这是因为numpy.sin为了同时支持数组和单个值的计算，其C语言的内部实现要比math.sin复杂很多，如果我们同样在Python级别进行循环的话，就会看出其中的差别了。此外，numpy.sin返回的数的类型和math.sin返回的类型有所不同，math.sin返回的是Python的标准float类型，而numpy.sin则返回一个numpy.float64类型：

```
>>> type(math.sin(0.5))
<type 'float'>
>>> type(np.sin(0.5))
<type 'numpy.float64'>
```

通过上面的例子我们了解了如何最有效率地使用math库和numpy库中的数学函数。因为它们各有长短，因此在导入时不建议使用*号全部载入，而是应该使用import numpy as np的方式载入，这样我们可以根据需要选择合适的函数调用。

NumPy中有众多的ufunc函数为我们提供各式各样的计算。除了sin这种单输入函数之外，还有许多多个输入的函数，add函数就是一个最常用的例子。先来看一个例子:

```
>>> a = np.arange(0,4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(1,5)
>>> b
array([1, 2, 3, 4])
>>> np.add(a,b)
```

```
array([1, 3, 5, 7])
>>> np.add(a,b,a)
array([1, 3, 5, 7])
>>> a
array([1, 3, 5, 7])
```

add函数返回一个新的数组，此数组的每个元素都为两个参数数组的对应元素之和。它接受第3个参数指定计算结果所要写入的数组，如果指定的话，add函数就不再产生新的数组。

由于Python的操作符重载功能，计算两个数组相加可以简单地写为a+b，而np.add(a,b,a)则可以用a+=b来表示。下面是数组的运算符和其对应的ufunc函数的一个列表，注意除号"/"的意义根据是否激活__future__.division有所不同。

y = x1 + x2:	add(x1, x2 [, y])
y = x1 - x2:	subtract(x1, x2 [, y])
y = x1 * x2:	multiply (x1, x2 [, y])
y = x1 / x2:	divide (x1, x2 [, y]), 如果两个数组的元素为整数，那么用整数除法
y = x1 / x2:	true divide (x1, x2 [, y]), 总是返回精确的商
y = x1 // x2:	floor divide (x1, x2 [, y]), 总是对返回值取整
y = -x:	negative(x [,y])
y = x1**x2:	power(x1, x2 [, y])
y = x1 % x2:	remainder(x1, x2 [, y]), mod(x1, x2, [, y])

数组对象支持这些操作符，极大地简化了算式的编写，不过要注意如果你的算式很复杂，并且要运算的数组很大的话，会因为产生大量的中间结果而降低程序的运算效率。例如：假设a b c三个数组采用算式x=a*b+c计算，那么它相当于:

```
t = a * b
x = t + c
del t
```

也就是说需要产生一个数组t保存乘法的计算结果，然后再产生最后的结果数组x。我们可以通过手工将一个算式分解为x = a*b; x += c，以减少一次内存分配。

通过组合标准的ufunc函数的调用，可以实现各种算式的数组计算。不过有些时候这种算式不易编写，而针对每个元素的计算函数却很容易用Python实现，这时可以用frompyfunc函数将一个计算单个元素的函数转换成ufunc函数。这样就可以方便地用所产生的ufunc函数对数组进行计算了。让我们来看一个例子。

我们想用一个分段函数描述三角波，三角波的样子如图**2.4**所示：

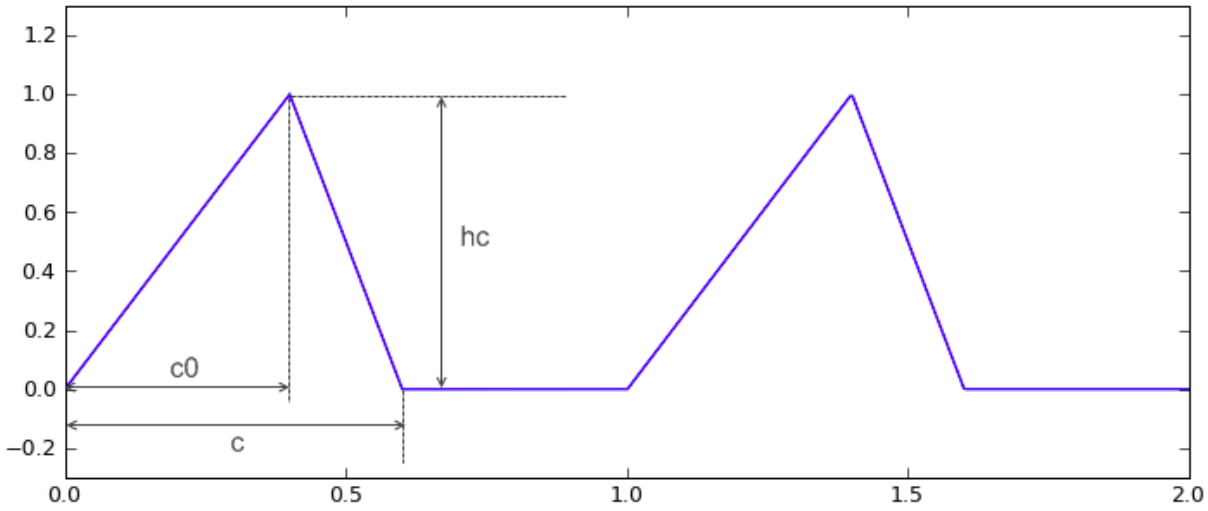

图**2.4** 三角波可以用分段函数进行计算

我们很容易根据上图所示写出如下的计算三角波某点y坐标的函数:

```
def triangle_wave(x, c, c0, hc):
    x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行计算
    if x >= c: r = 0.0
    elif x < c0: r = x / c0 * hc
    else: r = (c-x) / (c-c0) * hc
    return r
```

显然triangle_wave函数只能计算单个数值，不能对数组直接进行处理。我们可以用下面的方法先使用列表包容(List comprehension)，计算出一个list，然后用array函数将列表转换为数组：

```
x = np.linspace(0, 2, 1000)
y = np.array([triangle_wave(t, 0.6, 0.4, 1.0) for t in x])
```

这种做法每次都需要使用列表包容语法调用函数，对于多维数组是很麻烦的。让我们来看看如何用frompyfunc函数来解决这个问题：

```
triangle_ufunc = np.frompyfunc( lambda x: triangle_wave(x, 0.6, 0.4, 1.0), 1, 1)
y2 = triangle_ufunc(x)
```

frompyfunc的调用格式为frompyfunc(func, nin, nout)，其中func是计算单个元素的函数，nin是此函数的输入参数的个数，nout是此函数的返回值的个数。虽然triangle_wave函数有4个参数，但是由于后三个c, c0, hc在整个计算中值都是固定的，因此所产生的ufunc函数其实只有一个参数。为了满足这个条件，我们用一个lambda函数对triangle_wave的参数进行一次包装。这样传入frompyfunc的函数就只有一个参数了。这样子做，效率并不是太高，另外还有一种方法：

```
def triangle_func(c, c0, hc):
    def trifunc(x):
        x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行计算
```

```
        if x >= c: r = 0.0
        elif x < c0: r = x / c0 * hc
        else: r = (c-x) / (c-c0) * hc
        return r

    # 用trifunc函数创建一个ufunc函数，可以直接对数组进行计算，不过通过此函数
    # 计算得到的是一个Object数组，需要进行类型转换
    return np.frompyfunc(trifunc, 1, 1)

y2 = triangle_func(0.6, 0.4, 1.0)(x)
```

我们通过函数triangle_func包装三角波的三个参数，在其内部定义一个计算三角波的函数trifunc，trifunc函数在调用时会采用triangle_func的参数进行计算。最后triangle_func返回用frompyfunc转换结果。

值得注意的是用frompyfunc得到的函数计算出的数组元素的类型为object，因为frompyfunc函数无法保证Python函数返回的数据类型都完全一致。因此还需要再次 y2.astype(np.float64)将其转换为双精度浮点数组。

2.2.1 广播

当我们使用ufunc函数对两个数组进行计算时，ufunc函数会对这两个数组的对应元素进行计算，因此它要求这两个数组有相同的大小(shape相同)。如果两个数组的shape不同的话，会进行如下的广播(broadcasting)处理：

1. 让所有输入数组都向其中shape最长的数组看齐，shape中不足的部分都通过在前面加1补齐
2. 输出数组的shape是输入数组shape的各个轴上的最大值
3. 如果输入数组的某个轴和输出数组的对应轴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
4. 当输入数组的某个轴的长度为1时，沿着此轴运算时都用此轴上的第一组值

上述4条规则理解起来可能比较费劲，让我们来看一个实际的例子。

先创建一个二维数组a，其shape为(6,1)：

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1)
>>> a
array([[ 0], [10], [20], [30], [40], [50]])
>>> a.shape
(6, 1)
```

再创建一维数组b，其shape为(5,)：

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

计算a和b的和，得到一个加法表，它相当于计算a,b中所有元素组的和，得到一个shape为(6,5)的数组：

```
>>> c = a + b
>>> c
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44],
       [50, 51, 52, 53, 54]])
>>> c.shape
(6, 5)
```

由于a和b的shape长度(也就是ndim属性)不同，根据规则1，需要让b的shape向a对齐，于是将b的shape前面加1，补齐为(1,5)。相当于做了如下计算：

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

这样加法运算的两个输入数组的shape分别为(6,1)和(1,5)，根据规则2，输出数组的各个轴的长度为输入数组各个轴上的长度的最大值，可知输出数组的shape为(6,5)。

由于b的第0轴上的长度为1，而a的第0轴上的长度为6，因此为了让它们在第0轴上能够相加，需要将b在第0轴上的长度扩展为6，这相当于：

```
>>> b = b.repeat(6,axis=0)
>>> b
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

由于a的第1轴的长度为1，而b的第一轴长度为5，因此为了让它们在第1轴上能够相加，需要将a在第1轴上的长度扩展为5，这相当于：

```
>>> a = a.repeat(5, axis=1)
>>> a
array([[ 0,  0,  0,  0,  0],
       [10, 10, 10, 10, 10],
       [20, 20, 20, 20, 20],
       [30, 30, 30, 30, 30],
       [40, 40, 40, 40, 40],
       [50, 50, 50, 50, 50]])
```

经过上述处理之后，a和b就可以按对应元素进行相加运算了。

当然，numpy在执行a+b运算时，其内部并不会真正将长度为1的轴用repeat函数进行扩展，如果这样做的话就太浪费空间了。

由于这种广播计算很常用，因此numpy提供了一个快速产生如上面a,b数组的方法： ogrid对象：

```
>>> x,y = np.ogrid[0:5,0:5]
>>> x
array([[0],
       [1],
       [2],
       [3],
       [4]])
>>> y
array([[0, 1, 2, 3, 4]])
```

ogrid是一个很有趣的对象，它像一个多维数组一样，用切片组元作为下标进行存取，返回的是一组可以用来广播计算的数组。其切片下标有两种形式：

- 开始值:结束值:步长，和np.arange(开始值, 结束值, 步长)类似
- 开始值:结束值:长度j，当第三个参数为虚数时，它表示返回的数组的长度，和np.linspace(开始值, 结束值, 长度)类似：

  ```
  >>> x, y = np.ogrid[0:1:4j, 0:1:3j]
  >>> x
  array([[ 0.        ],
         [ 0.33333333],
         [ 0.66666667],
         [ 1.        ]])
  >>> y
  array([[ 0. ,  0.5,  1. ]])
  ```

ogrid为什么不是函数

根据Python的语法，只有在中括号中才能使用用冒号隔开的切片语法，如果ogrid是函数的话，那么这些切片必须使用slice函数创建，这显然会增加代码的长度。

利用ogrid的返回值，我能很容易计算x, y网格面上各点的值，或者x, y, z网格体上各点的值。下面是绘制三维曲面 x * exp(x**2 - y**2) 的程序：

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2)

pl = mlab.surf(x, y, z, warp_scale="auto")
```

```
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(pl)
```

此程序使用mayavi的mlab库快速绘制如图**2.5**所示的3D曲面，关于mlab的相关内容将在今后的章节进行介绍。

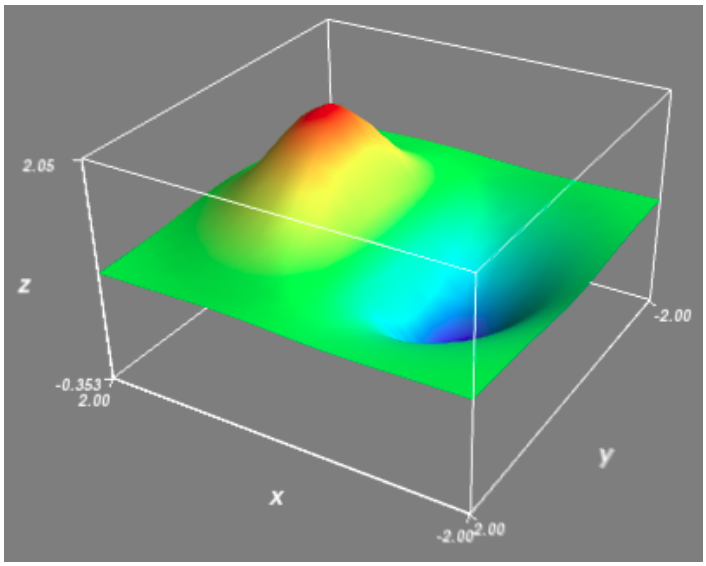

图**2.5** 使用ogrid创建的三维曲面

2.2.2 ufunc的方法

ufunc函数本身还有些方法，这些方法只对两个输入一个输出的ufunc函数有效，其它的ufunc对象调用这些方法时会抛出ValueError异常。

reduce 方法和Python的reduce函数类似，它沿着axis轴对array进行操作，相当于将<op>运算符插入到沿axis轴的所有子数组或者元素当中。

```
<op>.reduce (array=, axis=0, dtype=None)
```

例如：

```
>>> np.add.reduce([1,2,3]) # 1 + 2 + 3
6
>>> np.add.reduce([[1,2,3],[4,5,6]], axis=1) # 1,4 + 2,5 + 3,6
array([ 6, 15])
```

accumulate 方法和reduce方法类似，只是它返回的数组和输入的数组的shape相同，保存所有的中间计算结果：

```
>>> np.add.accumulate([1,2,3])
array([1, 3, 6])
>>> np.add.accumulate([[1,2,3],[4,5,6]], axis=1)
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

reduceat 方法计算多组reduce的结果，通过indices参数指定一系列reduce的起始和终了位置。reduceat的计算有些特别，让我们通过一个例子来解释一下：

```
>>> a = np.array([1,2,3,4])
>>> result = np.add.reduceat(a,indices=[0,1,0,2,0,3,0])
>>> result
array([ 1,  2,  3,  3,  6,  4, 10])
```

对于indices中的每个元素都会调用reduce函数计算出一个值来，因此最终计算结果的长度和indices的长度相同。结果result数组中除最后一个元素之外，都按照如下计算得出：

```
if indices[i] < indices[i+1]:
    result[i] = np.reduce(a[indices[i]:indices[i+1]])
else:
    result[i] = a[indices[i]
```

而最后一个元素如下计算:

```
np.reduce(a[indices[-1]:])
```

因此上面例子中，结果的每个元素如下计算而得：

```
1 : a[0] = 1
2 : a[1] = 2
3 : a[0] + a[1] = 1 + 2
3 : a[2] = 3
6 : a[0] + a[1] + a[2] =  1 + 2 + 3 = 6
4 : a[3] = 4
10: a[0] + a[1] + a[2] + a[4] = 1+2+3+4 = 10
```

可以看出result[::2]和a相等，而result[1::2]和np.add.accumulate(a)相等。

outer 方法，<op>.outer(a,b)方法的计算等同于如下程序：

```
>>> a.shape += (1,)*b.ndim
>>> <op>(a,b)
>>> a = a.squeeze()
```

其中squeeze的功能是剔除数组a中长度为1的轴。如果你看不太明白这个等同程序的话，让我们来看一个例子：

```
>>> np.multiply.outer([1,2,3,4,5],[2,3,4])
array([[ 2,  3,  4],
       [ 4,  6,  8],
       [ 6,  9, 12],
       [ 8, 12, 16],
       [10, 15, 20]])
```

可以看出通过outer方法计算的结果是如下的乘法表：

```
#    2, 3, 4
# 1
# 2
# 3
# 4
# 5
```

如果将这两个数组按照等同程序一步一步的计算的话，就会发现乘法表最终是通过广播的方式计算出来的。

2.3 矩阵运算

NumPy和Matlab不一样，对于多维数组的运算，缺省情况下并不使用矩阵运算，如果你希望对数组进行矩阵运算的话，可以调用相应的函数。

matrix对象

numpy库提供了matrix类，使用matrix类创建的是矩阵对象，它们的加减乘除运算缺省采用矩阵方式计算，因此用法和matlab十分类似。但是由于NumPy中同时存在ndarray和matrix对象，因此用户很容易将两者弄混。这有违Python的“显式优于隐式”的原则，因此并不推荐在较复杂的程序中使用matrix。下面是使用matrix的一个例子：

```
>>> a = np.matrix([[1,2,3],[5,5,6],[7,9,9]])
>>> a*a**-1
matrix([[  1.00000000e+00,   1.66533454e-16,  -8.32667268e-17],
        [ -2.77555756e-16,   1.00000000e+00,  -2.77555756e-17],
        [  1.66533454e-16,   5.55111512e-17,   1.00000000e+00]])
```

因为a是用matrix创建的矩阵对象，因此乘法和幂运算符都变成了矩阵运算，于是上面计算的是矩阵a和其逆矩阵的乘积，结果是一个单位矩阵。

矩阵的乘积可以使用dot函数进行计算。对于二维数组，它计算的是矩阵乘积，对于一维数组，它计算的是其点积。当需要将一维数组当作列矢量或者行矢量进行矩阵运算时，推荐先使用reshape函数将一维数组转换为二维数组：

```
>>> a = array([1, 2, 3])
>>> a.reshape((-1,1))
array([[1],
       [2],
       [3]])
>>> a.reshape((1,-1))
array([[1, 2, 3]])
```

除了dot计算乘积之外，NumPy还提供了inner和outer等多种计算乘积的函数。这些函数计算乘积的方式不同，尤其是当对于多维数组的时候，更容易搞混。

- **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；对于多维数组，它的通用计算公式如下，即结果数组中的每个元素都是：数组a的最后一维上的所有元素与数组b的倒数第二位上的所有元素的乘积和：

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

下面以两个3为数组的乘积演示一下dot乘积的计算结果：

首先创建两个3维数组，这两个数组的最后两维满足矩阵乘积的条件：

```
>>> a = np.arange(12).reshape(2,3,2)
>>> b = np.arange(12,24).reshape(2,2,3)
>>> c = np.dot(a,b)
```

dot乘积的结果c可以看作是数组a,b的多个子矩阵的乘积：

```
>>> np.alltrue( c[0,:,0,:] == np.dot(a[0],b[0]) )
True
>>> np.alltrue( c[1,:,0,:] == np.dot(a[1],b[0]) )
True
>>> np.alltrue( c[0,:,1,:] == np.dot(a[0],b[1]) )
True
>>> np.alltrue( c[1,:,1,:] == np.dot(a[1],b[1]) )
True
```

- **inner**：和dot乘积一样，对于两个一维数组，计算的是这两个数组对应下标元素的乘积和；对于多维数组，它计算的结果数组中的每个元素都是：数组a和b的最后一维的内积，因此数组a和b的最后一维的长度必须相同：

```
inner(a, b)[i,j,k,m] = sum(a[i,j,:]*b[k,m,:])
```

下面是inner乘积的演示：

```
>>> a = np.arange(12).reshape(2,3,2)
>>> b = np.arange(12,24).reshape(2,3,2)
>>> c = np.inner(a,b)
>>> c.shape
(2, 3, 2, 3)
>>> c[0,0,0,0] == np.inner(a[0,0],b[0,0])
True
>>> c[0,1,1,0] == np.inner(a[0,1],b[1,0])
True
>>> c[1,2,1,2] == np.inner(a[1,2],b[1,2])
True
```

- **outer**：只按照一维数组进行计算，如果传入参数是多维数组，则先将此数组展平为一维数组之后再进行运算。outer乘积计算的列向量和行向量的矩阵乘积：

```
>>> np.outer([1,2,3],[4,5,6,7])
array([[ 4,  5,  6,  7],
       [ 8, 10, 12, 14],
       [12, 15, 18, 21]])
```

矩阵中更高级的一些运算可以在NumPy的线性代数子库linalg中找到。例如inv函数计算逆矩阵，solve函数可以求解多元一次方程组。下面是solve函数的一个例子：

```
>>> a = np.random.rand(10,10)
>>> b = np.random.rand(10)
>>> x = np.linalg.solve(a,b)
>>> np.sum(np.abs(np.dot(a,x) - b))
3.1433189384699745e-15
```

solve函数有两个参数a和b。a是一个N*N的二维数组，而b是一个长度为N的一维数组，solve函数找到一个长度为N的一维数组x，使得a和x的矩阵乘积正好等于b，数组x就是多元一次方程组的解。

有关线性代数方面的内容将在今后的章节中详细介绍。

2.4 文件存取

NumPy提供了多种文件操作函数方便我们存取数组内容。文件存取的格式分为两类：二进制和文本。而二进制格式的文件又分为NumPy专用的格式化二进制类型和无格式类型。

使用数组的方法函数tofile可以方便地将数组中数据以二进制的格式写进文件。tofile输出的数据没有格式，因此用numpy.fromfile读回来的时候需要自己格式化数据：

```
>>> a = np.arange(0,12)
>>> a.shape = 3,4
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.tofile("a.bin")
>>> b = np.fromfile("a.bin", dtype=np.float) # 按照float类型读入数据
>>> b # 读入的数据是错误的
array([  2.12199579e-314,   6.36598737e-314,   1.06099790e-313,
         1.48539705e-313,   1.90979621e-313,   2.33419537e-313])
>>> a.dtype # 查看a的dtype
dtype('int32')
>>> b = np.fromfile("a.bin", dtype=np.int32) # 按照int32类型读入数据
>>> b # 数据是一维的
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b.shape = 3, 4 # 按照a的shape修改b的shape
>>> b # 这次终于正确了
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

从上面的例子可以看出，需要在读入的时候设置正确的dtype和shape才能保证数据一致。并且tofile函数不管数组的排列顺序是C语言格式的还是Fortran语言格式的，统一使用C语言格式输出。

此外如果fromfile和tofile函数调用时指定了sep关键字参数的话，数组将以文本格式输入输出。

numpy.load和numpy.save函数以NumPy专用的二进制类型保存数据，这两个函数会自动处理元素类型和shape等信息，使用它们读写数组就方便多了，但是numpy.save输出的文件很难和其它语言编写的程序读入：

```
>>> np.save("a.npy", a)
>>> c = np.load( "a.npy" )
>>> c
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

如果你想将多个数组保存到一个文件中的话，可以使用numpy.savez函数。savez函数的第一个参数是文件名，其后的参数都是需要保存的数组，也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, ...。savez函数输出的是一个压缩文件(扩展名为npz)，其中每个文件都是一个save函数保存的npy文件，文件名对应于数组名。load函数自动识别npz文件，并且返回一个类似于字典的对象，可以通过数组名作为关键字获取数组的内容：

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> b = np.arange(0, 1.0, 0.1)
>>> c = np.sin(b)
>>> np.savez("result.npz", a, b, sin_array = c)
>>> r = np.load("result.npz")
>>> r["arr_0"] # 数组a
array([[1, 2, 3],
       [4, 5, 6]])
>>> r["arr_1"] # 数组b
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
>>> r["sin_array"] # 数组c
array([ 0.        ,  0.09983342,  0.19866933,  0.29552021,  0.38941834,
        0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691])
```

如果你用解压软件打开result.npz文件的话，会发现其中有三个文件：arr_0.npy，arr_1.npy， sin_array.npy，其中分别保存着数组a, b, c的内容。

使用numpy.savetxt和numpy.loadtxt可以读写1维和2维的数组：

```
>>> a = np.arange(0,12,0.5).reshape(4,-1)
>>> np.savetxt("a.txt", a) # 缺省按照'%.18e'格式保存数据，以空格分隔
>>> np.loadtxt("a.txt")
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5],
       [  3. ,   3.5,   4. ,   4.5,   5. ,   5.5],
       [  6. ,   6.5,   7. ,   7.5,   8. ,   8.5],
```

```
       [  9. ,   9.5,  10. ,  10.5,  11. ,  11.5]])
>>> np.savetxt("a.txt", a, fmt="%d", delimiter=",") #改为保存为整数，以逗号分隔
>>> np.loadtxt("a.txt",delimiter=",") # 读入的时候也需要指定逗号分隔
array([[  0.,   0.,   1.,   1.,   2.,   2.],
       [  3.,   3.,   4.,   4.,   5.,   5.],
       [  6.,   6.,   7.,   7.,   8.,   8.],
       [  9.,   9.,  10.,  10.,  11.,  11.]])
```

文件名和文件对象

本节介绍所举的例子都是传递的文件名，也可以传递已经打开的文件对象，例如对于load和save函数来说，如果使用文件对象的话，可以将多个数组储存到一个npy文件中：

```
>>> a = np.arange(8)
>>> b = np.add.accumulate(a)
>>> c = a + b
>>> f = file("result.npy", "wb")
>>> np.save(f, a) # 顺序将a,b,c保存进文件对象f
>>> np.save(f, b)
>>> np.save(f, c)
>>> f.close()
>>> f = file("result.npy", "rb")
>>> np.load(f) # 顺序从文件对象f中读取内容
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> np.load(f)
array([ 0,  1,  3,  6, 10, 15, 21, 28])
>>> np.load(f)
array([ 0,  2,  5,  9, 14, 20, 27, 35])
```