NumPy——快速处理数据

目录

- ndarray对象
- ufunc运算
- 矩阵运算
- 文件存取
- 函数库
- NumPy模块

NumPy

—ndarray对象

ndarray对象

- NumPy的导入
- 创建数组
- 存取元素
- 多维数组
- 结构数组

NumPy的导入

- 标准的Python中用列表(list)保存一组值，可以当作数组使用。但由于列表的元素可以是任何对象，因此列表中保存的是对象的指针。对于数值运算来说，这种结构显然比较浪费内存和CPU计算。

- Python 提供了array 模块，它和列表不同，能直接保存数值，但是由于它不支持多维数组，也没有各种运算函数，因此也不适合做数值运算。

NumPy的导入

- 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。

- 函数库的导入

```
import numpy as np
```

创建数组

在IPython 中输入函数名并添加一个“?”符号，就可以显示文档内容。例如，输入“np.array?”

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):

创建数组

```
>>> 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  #数组的元素类型可以通过dtype 属
性获得
dtype('int32')
```

创建数组

数组的大小可以通过其shape属性获得：

```
>>> a.shape #一维数组
(4L,)
>>> c.shape #二维数组其中第0 轴的长度为3，第1 轴的长度为4。
(3L, 4L)
```

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。

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

创建数组

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

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

创建数组

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。

```
>>> 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]])
```

创建数组

数组的元素类型可以通过dtype属性获得。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]])
```

创建数组

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。

- 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])
```

创建数组

- 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, 10) # 步长为1/9
array([ 0. , 0.11111111, 0.22222222, 0.33333333,
0.44444444,0.55555556, 0.66666667, 0.77777778,
0.88888889, 1. ])

>>> np.linspace(0, 1, 10, endpoint=False) # 步长为1/10
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

创建数组

- 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.        ])
```

创建数组

zeros()、ones()、empty()可以创建指定形状和类型的数组。

```
>>> np.empty((2,3),np.int) #只分配内存，不对其进行初始化
array([[ 32571594, 32635312, 505219724],
       [ 45001384, 1852386928, 665972]])

>>> np.zeros(4, np.float) #元素类型默认为np.float，因此这里可以省略
array([ 0., 0., 0., 0.])
```

此外，zeros_like()、ones_like()、empty_like()等函数可创建与参数数组的形状及类型相同的数组。因此，“zeros_like(a)”和“zeros(a.shape, a.dtype)”的效果相同。

创建数组

此外，使用frombuffer, fromstring, fromfile，fromfunction等函数可以从字节序列、文件创建数组，下面以fromfunction为例:np.fromfunction？

```
>>> 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)。

创建数组

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):
      #print i
       #print  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.]])
```

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
```

存取元素

```
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])
```

存取元素

- 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])
```

存取元素

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])
```

存取元素

```
>>> 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])
```

存取元素

- 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])
```

存取元素

```
>>> 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])]
#x[np.array([True, False, True, True,False])]
>>> # 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
# x[np.array([True, False, True, True,False])] = -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])
```

存取元素

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。

```
>>> 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])
```

存取元素

```
>>> 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])
```

存取元素

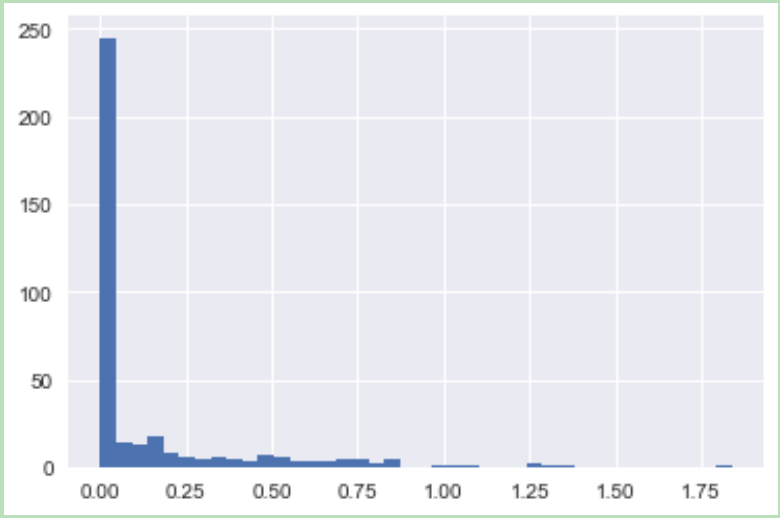
统计西雅图市下雨天数

```
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd

# 利用Pandas抽取降雨量，放入一个NumPy数组
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
inches = rainfall / 254.0 # 1英寸(inch) = 2.54厘米(cm)
inches.shape

#%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(inches, 40)
plt.show()

inches.max()
inches.min()
```



存取元素

该直方图表明了这些数据的大意：西雅图市 2014 年大多数时间的降水量都是接近 0 的。但是这样做并没有很好地传递出希望看到的某些信息，例如一年中有多少天在下雨，这些下雨天的平均降水量是多少，有多少天的降水量超过了半英寸？

回答以上问题的一种方法是通过传统的统计方式，即对所有数据循环，当碰到数据落在我们希望的区间时计数器便加 1。这种方法无论从编写代码的角度看，还是从计算结果的角度看，这都是一种浪费时间、非常低效的方法。

存取元素

通过将布尔操作、掩码操作和聚合结合，可以快速回答对数据集提出的这类问题。

```
rainy = (inches > 0)   # 为所有下雨天创建一个掩码，生成布尔数组。

# 构建一个包含整个夏季日期的掩码（6月21日是第172天）
summer = (np.arange(365) - 172 < 90) & (np.arange(365) - 172 > 0)

print("Median precip on rainy days in 2014 (inches): ", np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches):", np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):", np.median(inches[rainy & ~summer]))
```

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。如下图所示，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。

多维数组

如下图所示，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

```
>>> 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]])
```

多维数组

如何创建这个数组：

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。

```
>>> 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

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])
```

多维数组

- 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。

多维数组

- 多维数组索引的一个常见用途是从一个矩阵中选择行的子集。例如以下是一个二维正态分布的点组成的数组：

```
import numpy as np
rand = np.random.RandomState(42)

mean = [0, 0]
cov = [[1, 2],[2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape

import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1])
```

多维数组

利用多维数组的索引随机选取 20 个点——选择 20 个随机的、不重复的索引值，并利用这些索引值选取到原始数组对应的值：

```
indices = np.random.choice(X.shape[0], 20, replace=False)
indices

selection = X[indices] # 由索引取点取值
selection.shape

plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1], facecolor='none',
edgecolor='b', s=200)
```

现在来看哪些点被选中了，将选中的点在图上用大圆圈标示出来（如图所示）。

结构数组

假设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：

文件名：numpy_struct_array.py

```
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)

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

结构数组

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”
```

结构数组

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'
```

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
```

复制和视图

当运算和处理数组时，它们的数据有时被拷贝到新的数组有时不是。这有三种情况：

- 完全不拷贝

简单的赋值不拷贝数组对象或它们的数据。

```
>>> a = arange(12)
>>> b = a
>>> b is a
True
>>> b.shape = 3,4
>>> a.shape
(3L, 4L)
>>> id(a)
144004160L
>>> id(b)
144004160L
```

复制和视图

对**赋值**操作要有以下认识：

1、赋值是将一个对象的地址赋值给一个变量，让变量指向该地址（ 旧瓶装旧酒 ）。

2、修改不可变对象（str、tuple）需要开辟新的空间

3、修改可变对象（list等）不需要开辟新的空间

```
>>>c=['hello',[1,2,3]]
>>> d=c[:]
>>> [id(x) for x in c]
>>> [id(x) for x in d]
>>> c[0]='world'
>>> c[1].append(4)
>>> print(c)
>>> print(d)
>>> [id(x) for x in c]
>>> [id(x) for x in d]
```

复制和视图

视图(view)和浅复制

不同的数组对象分享同一个数据。视图方法创造一个新的数组对象指向同一数据。

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c is a view of the data owned by a
True
>>> c.flags.owndata      # c并不拥有数据
 False
>>> c.shape = 2,6 # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234 # a's data changes
>>> a
```

复制和视图

切片数组返回它的一个视图：

```
>>> s = a[ : , 1:3]
>>> s[:] = 10
# s[:] is a view of a. Note the difference between s=10
and s[:]=10
>>> a
array([[ 0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

浅拷贝是在另一块地址中创建一个新的变量或容器，但是容器内的元素的地址均是源对象的元素的地址的拷贝。也就是说新的容器中指向了旧的元素（ 新瓶装旧酒 ）。

复制和视图

深复制

这个复制方法完全复制数组和它的数据。

```
>>> d = a.copy()
# a new array object with new data is created
>>> d is a
False
>>> d.base is a
# d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[ 0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

复制和视图

python中的深拷贝和浅拷贝和java里面的概念是一样的，所谓浅拷贝就是对引用的拷贝，所谓深拷贝就是对对象的资源的拷贝。

深拷贝是在另一块地址中创建一个新的变量或容器，同时容器内的元素的地址也是新开辟的，仅仅是值相同而已，是完全的副本。也就是说（ 新瓶装新酒 ）。

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)
>>> y=x[[3, 3, 1, 8]]
>>> y[2]=100
>>> x
```

NumPy

—ufunc运算

目录

- ufunc运算简介
- 广播
- ufunc的方法

ufunc运算简介

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。

```
# 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
>>> x = np.linspace(0, 2*np.pi, 10)
>>> 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])
```

ufunc运算简介

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
```

ufunc运算简介

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：(numpy_speed_test.py)

```
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
```

ufunc运算简介

```
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

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
math.sin: 1.27905766614
numpy.sin: 0.126791054937
numpy.sin loop: 3.2112745202
```

ufunc运算简介

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。

针对数组的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，不过对单个数的计算math.sin则比numpy.sin快得多了。

ufunc运算简介

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'>
```

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，这样可以根据需要选择合适的函数调用。

ufunc运算简介

四则运算：

NumPy中有众多的ufunc函数提供各式各样的计算。

```
>>> 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) #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。
array([1, 3, 5, 7])
>>> a
array([1, 3, 5, 7])
```

ufunc运算简介

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
numpy. true_divide?
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])？？？

ufunc运算简介

```
#是否激活__future__.division:  激活  from __future__ import divide  整数/ 精确除；否则整数/取整用//.
Examples
--------
>>> x = np.arange(5)
>>> np.true_divide(x, 4)
array([ 0. , 0.25, 0.5 , 0.75, 1. ])

>>> x/4
array([0, 0, 0, 0, 1])
>>> x//4
array([0, 0, 0, 0, 1])

>>> from __future__ import division
>>> x/4
array([ 0. , 0.25, 0.5 , 0.75, 1. ])
>>> x//4
array([0, 0, 0, 0, 1])
```

ufunc运算简介

比较和布尔运算：

使用“==”、“>”等比较运算符对两个数组进行比较，将返回一个布尔数组，它的每个元素值都是两个数组对应元素的比较结果。例如：

```
>>> np.array([1,2,3]) < np.array([3,2,1])

array([ True, False, False], dtype=bool)
```

ufunc运算简介

每个比较运算符也与一个ufunc函数对应，下面是比较运算符和其ufunc函数：

y = x1 == x2	equal(x1, x2 [, y])
y = x1 != x2	not_equal(x1, x2 [, y])
y = x1 < x2	less(x1, x2, [, y])
y = x1 <= x2	less_equal(x1, x2, [, y])
y = x1 > x2	greater(x1, x2, [, y])
y = x1 >= x2	greater_equal(x1, x2, [, y])

ufunc运算简介

由于Python中的布尔运算使用and、or和not等关键字，它们无法被重载，因此数组的布尔运算只能通过相应的ufunc函数进行。这些函数名都以“logical_”开头，在IPython中使用自动补全即可找到它们.

```
>>> a = np.arange(5)
>>> b = np.arange(4,-1,-1)
>>> a == b
array([False, False, True, False, False], dtype=bool)
>>> a > b
array([False, False, False, True, True], dtype=bool)
>>> np.logical_or(a==b, a>b) # 和 a>=b 相同
array([False, False, True, True, True], dtype=bool)
```

ufunc运算简介

可以使用数组的any()或all()方法。只要数组中有一个值为True，则any()返回True；而只有数组的全部元素都为True，all()才返回True。

```
>>> np.any(a==b)
 True

>>> np.any(a==b) and np.any(a>b)
 True
```

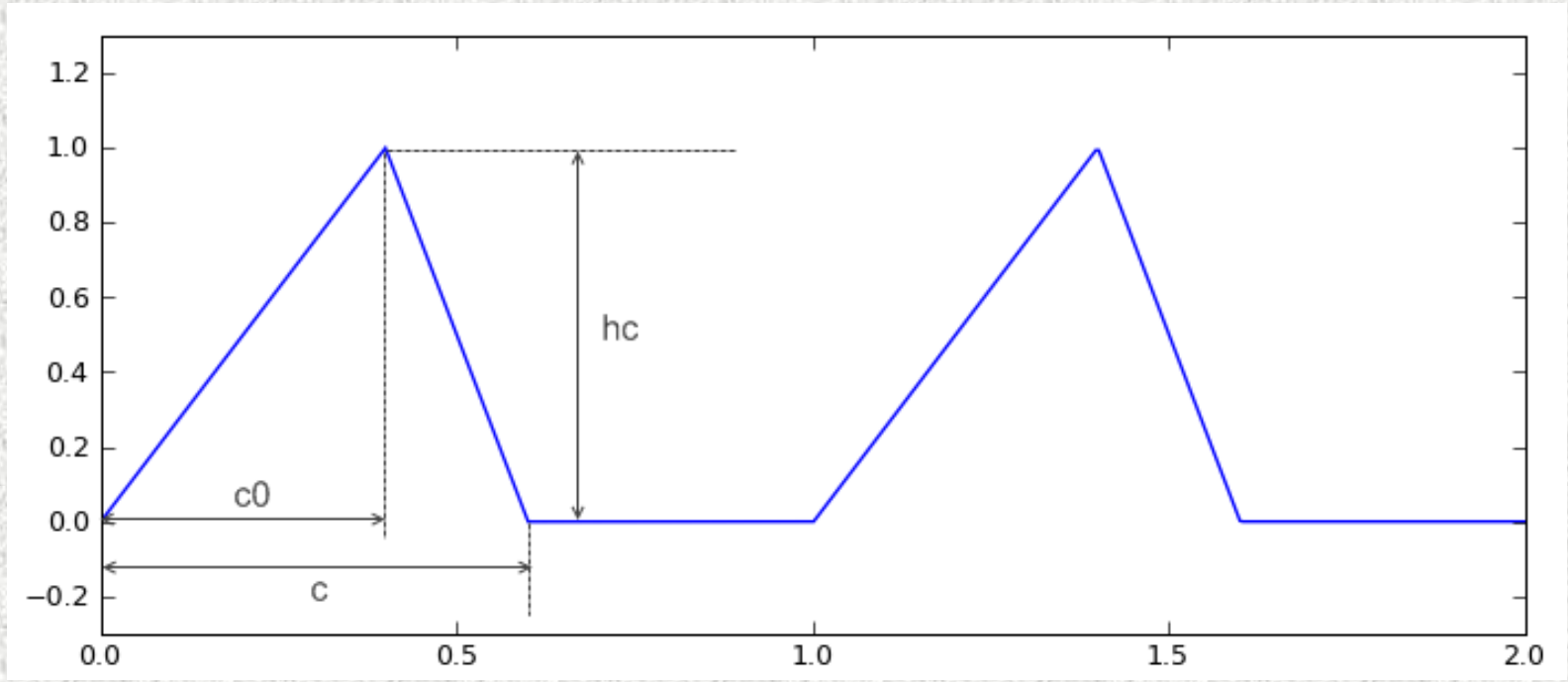
ufunc运算简介

自定义ufunc函数：

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。

ufunc运算简介

用一个分段函数描述三角波，三角波的样子如下：



ufunc运算简介

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。

ufunc运算简介

可以先使用计算出一个list，然后用array函数将列表转换为数组：

```
x = np.linspace(0, 2, 1000)
y1 = 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：

ufunc运算简介

通过frompyfunc()可以将计算单个值的函数转换为一个能对数组的每个元素进行计算的ufunc函数。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func的输入参数的个数，nout是func的返回值个数。下面的程序使用frompyfunc()将triangle_wave()转换为一个ufunc函数对象triangle_ufunc1：

ufunc运算简介

```
triangle_ufunc1 = np.frompyfunc(triangle_wave, 4, 1)
y2 = triangle_ufunc1(x, 0.6, 0.4, 1.0)
```

```
triangle_ufunc2 = np.frompyfunc( lambda x:
triangle_wave(x, 0.6, 0.4, 1.0), 1, 1)
y3 = triangle_ufunc2(x)
```

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，可用一个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。

ufunc运算简介

值得注意的是，triangle_ufunc1()所返回的数组的元素类型是object，因此还需要再调用数组的astype()方法将其转换为双精度浮点数组： numpy_frompyfunc.py

```
>>> y2.dtype
 dtype('object')
>>> y2 = y2.astype(np.float)
>>> y2.dtype
 dtype('float64')
>>> y3.dtype
 dtype('object')
>>> y3 = y3.astype(np.float)
>>> y3.dtype
 dtype('float64')
```

ufunc运算简介

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

np.vectorize?

下面的程序使用vectorize()计算三角波：

```
triangle_ufunc3 = np.vectorize(triangle_wave,
otypes=[np.float])
y4 = triangle_ufunc3(x, 0.6, 0.4, 1.0)
```

ufunc运算简介

最后验证一下结果：

```
>>> !python frompyfunc.py


>>> np.all(y1==y2)
 True
>>> np.all(y2==y3)
True
>>> np.all(y3==y4)
True
```

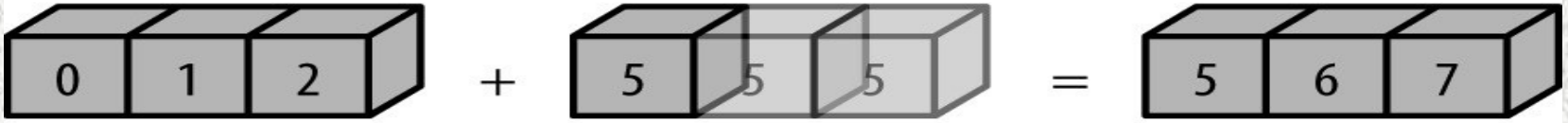
广播

对形状不同的数组的运算采取的操作。但是这个输入的数组中必须有一个某轴长度为1，或者缺少了一个维度（这个时候会自动的在shape属性前面补上）

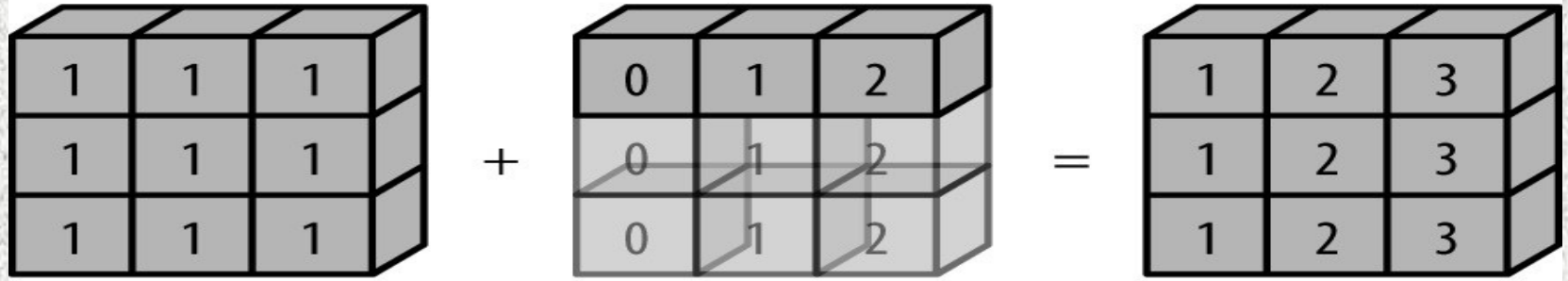
当我们使用ufunc函数对两个数组进行计算时，ufunc函数会对这两个数组的对应元素进行计算，因此它要求这两个数组有相同的大小(shape相同)。如果两个数组的shape不同的话，会进行如下的广播(broadcasting)处理：

广播

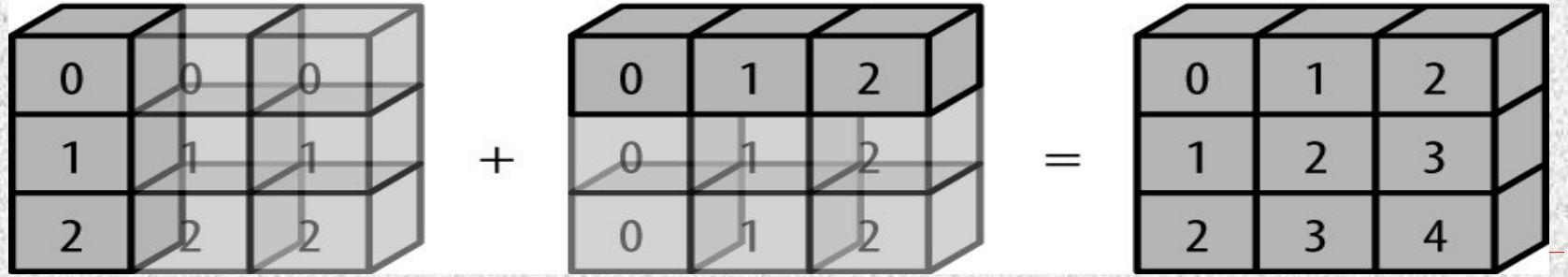
np. arange(3)+5



np. ones((3, 3))+np.arange(3)



np. ones((3, 1))+np.arange(3)



广播

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。

广播

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,)
```

广播

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)
```

广播

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)。

广播

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]])
```

广播

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。

广播

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]])
```

广播

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：

广播

```
>>> 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。

广播

利用ogrid的返回值，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：(numpy_orid_mlab.py)

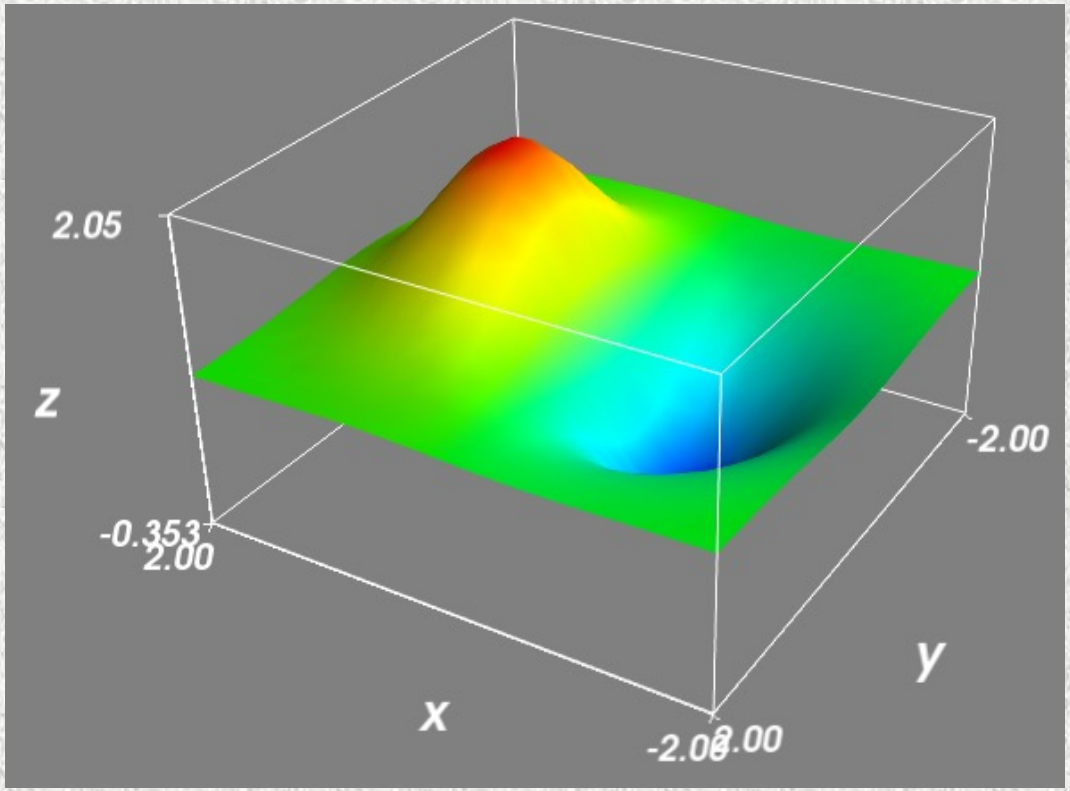
```
import numpy as np
from mayavi import mlab

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)
mlab.show
```

广播

此程序使用mayavi的mlab库快速绘制3D曲面。



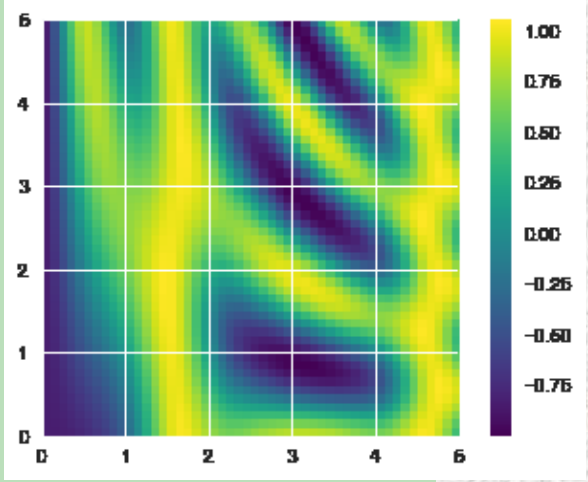
广播

```
# -*- coding: utf-8 -*-
import numpy as np
#%matplotlib inline


#x和y表示0~5区间50个步长的序列
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)

import matplotlib.pyplot as plt

plt.imshow(z, origin='lower', extent=[0, 5, 0,
5],cmap='viridis')
plt.colorbar()
plt.show()
```



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.

```
>>> np.add.reduce([1,2,3]) # 1 + 2 + 3
6
>>> np.add.reduce([[1,2,3],[4,5,6]], axis=1) #
(1+2+3),(4+5+6)
array([ 6, 15])
>>>np.add.reduce([[1,2,3],[4,5,6]], axis=0)
array([5, 7, 9])
```

ufunc的方法

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]])
```

ufunc的方法

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])
```

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

ufunc的方法

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

而最后一个元素如下计算：

```
<op>.reduce(a[indices[-1]:])
```

ufunc的方法

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[3] = 1+2+3+4 = 10
```

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

ufunc的方法

outer 方法,对其两个参数数组的每两对元素的组合进行运算。若数组a的维数为M，数组b的维数为N，则ufunc函数<op>的outer()方法对a、b数组计算所生成的数组c的维数为M+N。c的形状是a、b的形状的结合。例如a的形状为(2,3)，b的形状为(4,5)，则c的形状为(2,3,4,5)。<op>.outer(a,b)方法的计算等同于如下程序：

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

其中squeeze的功能是剔除数组a中长度为1的轴。

ufunc的方法

如果不太明白这个等同程序的话，可看一个例子：

```
>>> 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]])

>>> np.add.outer([1,2,3,4,5],[2,3,4])
```

ufunc的方法

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
2| 4 6 8
3| 6 9 12
4| 8 12 16
5|10 15 20
```

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。

```
>>>np.multiply.outer?
```

Python基础

—矩阵运算

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(np.mat)

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。

矩阵运算

下面是使用matrix的一个例子：

```
>>> a = np.matrix([[1,2,3],[5,5,6],[7,9,9]]) #np.mat
>>> 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。

矩阵运算

数组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])
>>> zz1=a.reshape((-1,1))
array([[1],
[2],
[3]])
>>> zz2=a.reshape((1,-1))
array([[1, 2, 3]])
>>> dot(zz1,zz2) # zz1.dot(zz2)
```

矩阵运算

实际上Python官方文档建议我们使用二维数组代替矩阵来进行矩阵运算；因为二维数组用得较多，而且基本可取代矩阵。

矩阵的转置：

```
>>> A = np.array([[1, 2, 3], [3, 4, 5], [6, 7, 8]])
>>> A

>>> A.T    #A的转置

>>> A.T.T    #A的转置的转置还是A本身

>>> A*A
>>> A.dot(A) #dot(A,A)
```

矩阵运算

方阵的迹:

```
>>>A = np.array([[1, 2, 3], [3, 4, 5], [6, 7, 8]])

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

>>>np.trace(A)
13

>>>np.trace(A.T)
13
```

矩阵运算

NumPy也允许你使用“点” (...)代表许多产生一个完整的索引元组必要的分号。如果x是秩为5的数组(即它有5个轴)，那么:

x[1,2,...] 等同于 x[1,2,:,:,:],

x[...,3] 等同于 x[:,:,:,:,3]

x[4,...,5,:] 等同 x[4,:,:,5,:].

```
# a 3D array (two stacked 2D arrays)
>>> c = array( [ [[ 0, 1, 2],
                  [ 10, 12, 13]],
                  [[100,101,102],
                  [110,112,113]] ] )
```

矩阵运算

```
# a 3D array (two stacked 2D arrays)
>>> c = array( [ [[ 0, 1, 2],
                  [ 10, 12, 13]],
                 [[100,101,102],
                  [110,112,113]] ] )

>>> c.shape
(2, 2, 3)

 >>> c[1,...]     # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])

>>> c[...,2]     # same as c[:,:,2]
array([[ 2, 13],
       [102, 113]])
```

矩阵运算

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的倒数第二维上的所有元素的乘积和：

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

矩阵运算

下面以两个3维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)
>>> c.shape
(2L, 3L, 2L, 3L)
```

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
```

矩阵运算

- 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,:])
```

矩阵运算

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
```

矩阵运算

- 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]) #np.outer?
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：

矩阵运算

```
>>> 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后面中详细介绍。

矩阵运算

python的常见矩阵运算

```
>>> a1=mat([[1,2],[2,4]])
>>> a2=mat([[5,6],[7,8]])
>>> a3=a1*a2  #矩阵相乘
>>> a4=np.multiply(a1,a2) #矩阵点乘
>>> a5=a4.I  #矩阵求逆
>>> a4*a5  #验证
>>> a6=a4.T  #转置
>>>a7=a4.sum(axis=0)  #列和
>>>a8=a4.sum(axis=1) #行和
>>>a9=np.sum(a4[1,:]) #计算第二行所有列的和，这里得到的是一个数值
>>> a4.max() #计算a4矩阵中所有元素的最大值,这里得到的结果是一个数值
>>> a10=np.max(a4[:,1]) #计算第二列的最大值，这里得到的是一个数值
>>> np.max(a4,1) #计算所有行的最大值，这里得到是一个矩阵
```

矩阵运算

- 操作符 *, dot(), 和 multiply():
  - 对于数组， * 表示逐元素乘法，而 dot() 函数用于矩阵乘法。
  - 对于矩阵， * 表示矩阵乘法， multiply() 函数用于逐元素乘法。
  - 使用数组类型从 Python3.5 之后，您就可以使用矩阵乘法 @ 运算符。

    dot(dot(A,B),C) =》A @ B @ C
- 处理更高维数组（ndim> 2）
  - 数组对象的维数可以> 2;
  - 矩阵对象总是具有两个维度。

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]])
```

文件存取

```
>>> 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]])
```

文件存取

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。 sep参数指定的是文本数据中数值的分隔符。

ndarray.tofile(file[, sep, format])

np.fromfile(file[, dtype, count, sep])

文件存取

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输出的文件很难被其它语言编写的程序读入：

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

文件存取

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：

文件存取

```
>>> 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。

文件存取

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'%.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.]])
```

文件存取

- 文件名和文件对象

  前面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：

文件存取

```
>>> 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])
```