NumPy

— 函数库

目录

- 求和、平均值、方差
- 更改数组的形状与数组堆叠
- 最值和排序
- 多项式函数
- 分段函数
- 统计函数
- 解线性方程组

函数与函数库

除了前面介绍的ndarray数组对象和ufunc函数之外，NumPy还提供了大量对数组进行处理的函数。充分利用这些函数，能够简化程序的逻辑，提高运算速度。

求和、平均值、方差

sum()计算数组元素之和，也可以对列表、元组等和数组类似的序列进行求和。当数组是多维时，它计算数组中所有元素的和：

```
>>> a =np.random.randint(0,10,size=(4,5))
>>> a
array([[7, 1, 9, 6, 3],
       [5, 1, 3, 8, 2],
       [9, 8, 9, 4, 0],
       [9, 5, 1, 7, 0]])
>>> np.sum(a)
97
```

求和、平均值、方差

如果指定axis参数，求和运算将沿着指定的轴进行。在上面的例子中，数组a的第0轴长度为4,第1轴长度为5。如果axis参数为1，就对每行上的5个数求和，所得的结果是长度为 4 的一维数组。如果参数axis为0,就对每列上的4个数求和，结果是长度为5的一维数组。 即结果数组的形状是原始数组的形状除去其第axis个元素：

```
>>> np.sum(a,axis=1)
array([26, 19, 30, 22])
>>> np.sum(a, axis=0)
array([30, 15, 22, 25, 5])
```

求和、平均值、方差

上面的例子将产生一个新的数组来保存求和结果，如果希望将结果直接保存到另外一个数组中，可以和ufunc函数一样使用out参数指定输出数组，它的形状必须和结果数组的形状相同。

```
>>> c=np.array([1,2,3,4])

>>> np.sum(a,axis=1,out=c)
```

求和、平均值、方差

sum()默认使用和数组的元素类型相同的累加变量进行计算，如果元素类型为整数，就使用系统的默认整数类型作为累加变量。在32位系统中，累加变量的类型为32 bit整型。因此对整数数组进行累加时可能会出现溢出问题，即数组元素的总和超过了累加变量的取值范围。而对很大的单精度浮点数类型数组进行计算时，也可能出现精度不够的现象，这时可以通过dtype 参数指定累加变量的类型。在下面的例子中，对一个元素都为1.1的单精度数组进行求和，比较单精度累加变量和双精度累加变量的计算结果：

求和、平均值、方差

```
>>> b =np.ones(1000000, dtype=np.float32) * 1.1
# 创建一个很大的单精度浮点数数组
>>> b       # 1.1无法使用浮点数精示，存在一些误差
array([ 1.10000002， 1.10000002,…,1.10000002],
dtype=float32)
>>> np.sum(b)
#使用单精度累加变量进行累加计算，误差将越来越大
1110920.5
>>>np.sum(b, dtype=np.double) #使用双精度浮点数则能够得到正确的值
1100000.0238418579
```

求和、平均值、方差

cumsum()求累积和；cumprod() 累计积。

```
>>> arr12 = np.random.randint(1,10,size = 12).reshape(4,3)
>>> np.cumsum(arr12) #对每一个元素求累积和（从左到右，从上到下的元素顺序）

>>> np.cumsum(arr12, axis = 0) #计算每一列的累积和，并返回二维数组

>>>np.cumprod(arr12, axis = 1) #计算每一行的累计积，并返回二维数组
```

求和、平均值、方差

mean()用于求数组的平均值（期望），也可以通过axis参数指定求平均值的轴，通过out参数指定输出数组。和sum()不同的是，对于整数数组，它使用双精度浮点数进行计算，而对于其他类型的数组，则使用和数组元素类型相同的累加变量进行计算：

```
>>>np.mean(a,axis=1) #整数数组使用双精度浮点数进行计算
array([ 5.2，3.8，6. , 4.4])
>>> np.mean(b) #单精度浮点数使用单精度浮点数进行计算
1.1109205
>>> np.mean(b, dtype=np.double) #用双稍度浮点数计算平均值
1.1000000238418579
```

求和、平均值、方差

除此之外， np.average(X, axis=0, weights=w) （加权平均值）也可以对数组进行平均计算。它没有out和dtype参数，但有一个指定每个元素权值的weights参数，可在IPython中输入“np.average?”来查看使用说明。

```
>>> X = np.array([[.9, .1],
                  [.8, .2],
                  [.4, .6]])
>>> w = np.array([.2, .2, .6])
>>> print(w.dot(X))
>>> print(np.average(X, axis=0, weights=w))
```

求和、平均值、方差

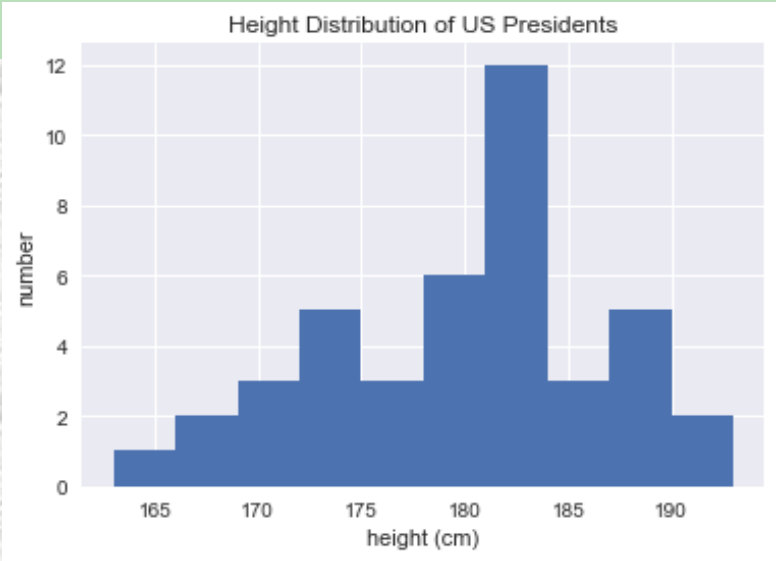
std()和var()分别计算数组的标准差和方差，有axis、out及dtype等参数。

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a) # 计算全局标准差
1.1180339887498949
>>> np.std(a, axis=0) # axis=0计算每一列的标准差
array([ 1.,  1.])
>>> np.std(a, axis=1) # 计算每一行的标准差
array([ 0.5,  0.5])
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1., 1.])
>>> np.var(a, axis=1)
array([ 0.25, 0.25])
```

美国总统的身高是多少

```
import numpy as np
import pandas as pd
#%matplotlib inline
import matplotlib.pyplot as plt

data = pd.read_csv('president_heights.csv‘)

heights = np.array(data['height(cm)'])
print(heights)

print("Mean height: ", heights.mean())
print("Standard deviation: ", heights.std())
print("Minimum height: ", heights.min())
print("Maximum height: ", heights.max())
```

```
print("25th percentile: ", np.percentile(heights, 25))
print("Median: ", np.median(heights))
print("75th percentile: ", np.percentile(heights, 75))

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number')
```



- 这些聚合是探索数据分析的一些最基本片段。

更改数组的形状与数组堆叠

更改数组的形状:

```
>>> a1 = floor(10*random.random((3,4)))
>>> a1
 array([[ 7., 5., 9., 3.],
        [ 7., 2., 7., 8.],
        [ 6., 8., 3., 2.]])
>>> a1.shape
(3, 4)
>>> a1.ravel() # flatten the array。flatten不会影响原始矩阵，返回的是一个副本，但是ravel是会修改数组
array([ 7., 5., 9., 3., 7., 2., 7., 8., 6., 8., 3., 2.])

>>> a1.shape = (6, 2)
>>> a1.T
>>> a1.transpose()
array([[ 7., 9., 7., 7., 6., 3.],
       [ 5., 3., 2., 8., 8., 2.]])
```

更改数组的形状与数组堆叠

对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置

```
>>> arr = np.arange(16).reshape((2,2,4))
>>> arr
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
>>> arr.transpose((1,0,2))
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

更改数组的形状与数组堆叠

还有一个swapaxes方法，它接受一对轴变换：

```
>>> arr
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
>>> arr.swapaxes(1,2)
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

更改数组的形状与数组堆叠

resize和reshape：两个函数都是改变数组的形状，但是resize是在本身上进行操作，reshape返回的是修改之后的参数。

```
>>> a1
array([[ 7., 5.],
       [ 9., 3.],
       [ 7., 2.],
       [ 7., 8.],
       [ 6., 8.],
       [ 3., 2.]])
>>> a1.resize(2,6)
>>> a1
array([[ 7., 5., 9., 3., 7., 2.],
       [ 7., 8., 6., 8., 3., 2.]])
```

更改数组的形状与数组堆叠

```
>>> a = np.arange(20).reshape(4,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> a.reshape(2,10)
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> a.resize(2,10)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

更改数组的形状与数组堆叠

沿不同轴将数组堆叠在一起：

```
>>> a2 = floor(10*random.random((2,2)))
>>> a2
array([[ 1., 1.],
       [ 5., 8.]])
 >>> b2 = floor(10*random.random((2,2)))
 >>> b2
array([[ 3., 3.],
       [ 6., 0.]])
>>> vstack((a2,b2))
array([[ 1., 1.],
       [ 5., 8.],
       [ 3., 3.],
       [ 6., 0.]])
>>> hstack((a2,b2))    # hstack？ hsplit？ concatenate?
array([[ 1., 1., 3., 3.],
       [ 5., 8., 6., 0.]])
#对那些维度比二维更高的数组，hstack沿着第二个轴组合，vstack沿着第一个轴组合,concatenate允许可选参数给出组合时沿着的轴。
```

更改数组的形状与数组堆叠

使用hsplit能将数组沿着水平轴分割，或者指定返回相同形状数组的个数，或者指定在哪些列后发生分割；vsplit沿着纵向的轴分割，split允许指定沿哪个轴分割。

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> dd,qq=np.hsplit(x,2)
>>> x
 >>>  dd
>>> qq
>>> rr,tt,ll=np.vsplit(x, np.array([3, 6]))
>>> rr,tt,ll=np.vsplit(x, np.array([2, 3]))
```

最值和排序

用min()和max()可以计算数组的最大值和最小值，而ptp()计算最大值和最小值之间的差。 它们都有axis和out两个参数。这些参数的用法和sum()相同。

```
>>>np.min(a2)
1.0
>>>np.max(a2)
9.0
>>>np.ptp(a2)
8.0
```

用argmax()和argmin()可以求最大值和最小值的下标。如果不指定axis参数，就返回平坦化之后的数组下标，例如：

最值和排序

```
>>> np.argmax(a) #找到数组a中最大值的下标，有多个最值时得到第一个最值的下标
2
>>> a.ravel()[2] #求平坦化之后的数组中的第二个元素
 9
```

可以通过unravel_index()将一维下标转换为多维数组中的下标，它的第一个参数为一维下标值，第二个参数是多维数组的形状：

```
>>> idx = np.unravel_index(2, a.shape)
>>> idx
(0, 2)
>>> a[idx]
9
```

最值和排序

当使用axis参数时，可以沿着指定的轴计算最大值的下标。例如下面的结果表示，在数组 a中，第0行中最大值的下标为2,第1行中最大值的下标为3:

```
>>> idx = np.argmax(a, axis=1)
>>> idx
array([2, 3, 0, 0])
```

下面的语句使用idx选择出每行的最大值:

```
>>> a[range(a.shape[0]),idx]
array([9, 8, 9, 9])
```

最值和排序

数组的sort()方法用于对数组进行排序，它将改变数组的内容。而sort()函数则返回一个新数组，不改变原始数组。它们的axis参数默认值都为-1,即沿着数组的最后一个轴进行排序。sort()函数的axis参数可以设置为None,此时它将得到平坦化之后进行排序的新数组。

```
>>> np.sort(a) #对每行的数据进行排序# b=sort(a)
 array([[1, 3, 6, 7, 9],
        [1, 2, 3,5, 8],
        [0, 4, 8, 9, 9],
        [0, 1, 5, 7, 9]])
>>> np.sort(a, axis=0) #对每列的数据进行排序
array([[5,1,1, 4, 0],
       [7, 1, 3, 6, 0],
       [9, 5, 9, 7, 2],
       [9, 8, 9'8, 3]])
>>> np.sort(a, axis=None)
```

最值和排序

argsort()返回数组的排序下标，axis参数的默认值为-1：

```
>>>idx = np.argsort(a)
>>> idx
array([[1, 4, 3, 0, 2],
       [1, 4, 2, 0, 3],
       [4, 3, 1, 0, 2],
       [4, 2, 1, 3, 0]])
```

用median()可以获得数组的中值，即对数组进行排序之后，位于数组中间位置的值，当长度是偶数时，得到正中间两个数的平均值。它也可以指定axis和out参数：

```
>>> np.median(a,axis=0)
array([ 8. , 3. , 6. , 6.5, 1. ])
```

最值和排序

np.partition：

如果不希望对整个数组进行排序，仅仅希望找到数组中第 *K* 小的值， np.partition 函数提供了该功能。该函数的输入是数组和数字 *K*，输出结果是一个新数组，最左边是第 *K* 小的值，往右是任意顺序的其他值：

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

结果数组中前三个值是数组中最小的三个值，剩下的位置是原始数组剩下的值。在这两个分隔区间中，元素都是任意排列的。

最值和排序

与排序类似，也可以沿着多维数组任意的轴进行分隔：

```
>>> rand = np.random.RandomState(42)
>>> X = rand.randint(0, 10, (4, 6))
>>> print(X)
>>> np.partition(X, 2, axis=1)
```

输出结果是一个数组，该数组每一行的前两个元素是该行最小的两个值，每行的其他值分布在剩下的位置。

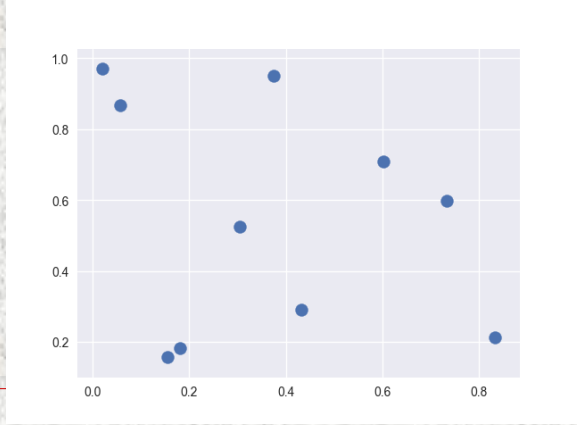
最后，正如 np.argsort 函数计算的是排序的索引值，也有一个np.argpartition 函数计算的是分隔的索引值。

最值和排序

- 示例展示的是如何利用 argsort、 argpartition 函数沿着多个轴快速找到集合中每个点的最近邻：

  首先，在二维平面上创建一个有 10 个随机点的集合。按照惯例，将这些数据点放在一个 10×2 的数组中：

```
>>> X = rand.rand(10, 2)
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:, 0], X[:, 1], s=100)
```



最值和排序

现在来计算两两数据点对间的距离。两点间距离的平方等于每个维度的距离差的平方的和。利用 NumPy 的广播和聚合功能，可以用一行代码计算矩阵的平方距离：

```
>>> dist_sq = np.sum((X[:,np.newaxis,:] -
X[np.newaxis,:,:]) ** 2, axis=2)
```

这个操作由很多部分组成。当你遇到这种代码时，将其各组件分解后再分析是非常有用的。应该看到该矩阵的对角线（也就是每个点到其自身的距离）的值都是 0：

```
>>> dist_sq.diagonal()
```

最值和排序

当有了这样一个转化为两点间的平方距离的矩阵，就可以使用 np.argsort 函数沿着每行进行排序。最左边的列给出的索引值就是最近邻：

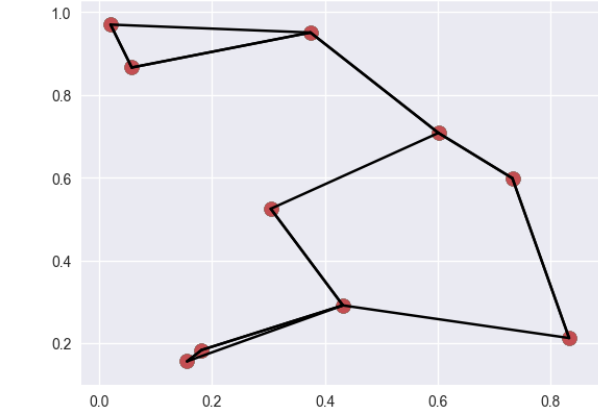
```
>>> nearest = np.argsort(dist_sq, axis=1)
>>> print(nearest)
```

第一列是按 0~9 从小到大排列的。这是因为每个点的最近邻是其自身。

如果仅仅关心 *k* 个最近邻，只需要做的是分隔，最小的 *k* + 1 的平方距离将排在最前面，可以用 np.argpartition 函数实现。（argsortpart.py）

最值和排序

```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1,
axis=1)

plt.scatter(X[:, 0], X[:, 1], s=100)

# 将每个点与它的两个最近邻连接
for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
# 画一条从X[i]到X[j]的线段

# 用zip方法实现：
        plt.plot(*zip(X[j], X[i]), color='black')
plt.show()
```



多项式函数

多项式函数是变量的整数次幂与系数的乘积之和，可以用下面的数学公式表示：

$$f(x)= a_nx^n+a_{n-1}x^{n-1} +...+a_2x^2+a_1x +a_0$$

由于多项式函数只包含加法和乘法运算，因此它很容易计算，并且可以用于计算其他数学函数的近似值。在NumPy中，多项式函数的系数可以用一维数组表示，例如$f(x) = x^3 -2x + 1$可以用下面的数组表示，其中a[0]是最高次的系数，a[-1]是常数项，注意x^2的系数为0。

```
>>>a= np.array([1.0, 0, -2, 1])
```

多项式函数

可以用poly1d()将系数转换为poly1d(一元多项式)对象，此对象可以像函数一样调用，它返回多项式函数的值：

```
>>> p = np.poly1d(a)

>>>type(p)
< numpy.lib.polynomial.poly1d>

>>> p(np.linspace(0,1, 5))
array([ 1., 0.515625,   0.125, -0.078125, 0. ])

# p.r ； p.c； p.order； print np.poly1d(p)；
np.square(p) ； print np.poly1d(p*p)
```

多项式函数

对poly1d对象进行加减乘除运算相当于对相应的多项式函数进行计算。例如：

```
>>> p + [-2, 1] # 和 p + np.poly1d([-2, 1])相同
poly1d([ 1., 0., -4., 2.])

>>> p*p #两个3次多项式相乘得到一个6次多项式
 poly1d([ 1.,       0., -4., 2., 4., -4., 1.])

>>> p / [1, 1] #除法返回两个多项式，分别为商式和余式 (poly1d([ 1., -1., -1.]), poly1d([ 2.]))
```

多项式函数

由于多项式的除法不一定能正好整除，因此它返回除法所得到的商式和余式。上面的例子中，商式为$x^2 - x - 1$,余式为2。因此将商式和被除式相乘后，再加上余式就等于原来的P:

```
>>> p==np.poly1d([ 1., -1., -1.]) * [1,1] + 2
True
```

多项式函数

多项式对象的deriv()和integ()方法分别计算多项式函数的微分和积分：

```
>>> p.deriv()  # polyder(p)
poly1d([ 3., 0., -2.])

>>> p.deriv(2)
poly1d([ 6.,  0.])

>>> p.integ()  #polyint(p)
poly1d([ 0.25, 0., -1., 1. , 0.])

>>> p.integ().deriv() == p
True
```

多项式函数

多项式函数的根可以使用roots()函数计算：

```
>>> r = np.roots(p)
>>> r
array([-1.61803399, 1.        , 0.61803399])

>>> p(r)  #将根带入多项式计算，得到的值近似为0
array([ -4.21884749e-15,   -4.44089210e-16,    -
2.22044605e-16])
```

而poly()函数可以将根转换回多项式的系数：

```
>>> np.poly(r)
array([ 1. 00000000e+00, 9.99200722e-16, -2.
000000000e+00 , 1.00000000e+00])
```

多项式函数

除了使用多项式对象之外，还可以直接使用NumPy提供的多项式函数对表示多项式系数的数组进行运算。可以在IPython中使用自动补全查看函数名：

```
>>> np.poly  # 按 Tab 键
np.poly np.polyadd np.polydiv np.polyint np.polysub
np.poly1d np.polyder np.polyfit np.polymul np.polyval

 >>> np.polyval([3,0,1], 5)
76
```

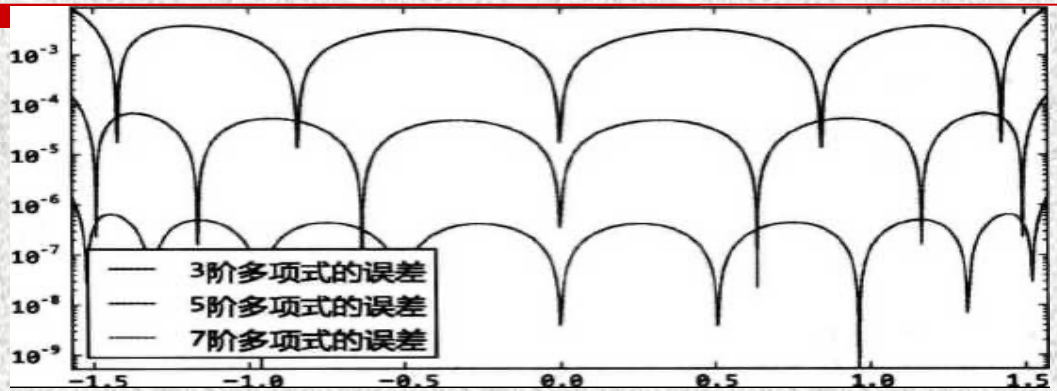
其中：polyfit()函数可以对一组数据使用多项式函数进行拟合，找到和这组数据最接近的多项式的系数。

多项式函数

计算-pi/2~ pi/2区间与sin(x)函数最接近的多项式的系数：(numpy_polyfit.py)

```
import numpy as np
import pylab as pl
pl.figure(figsize=(8,4))
x = np.linspace(-np.pi/2, np.pi/2, 1000)
y = np.sin(x)
for deg in [3,5,7]:
    a = np.polyfit(x, y, deg)
    error = np.abs(np.polyval(a, x)-y)
    print "poly %d:" % deg, a
    print "max error of order %d:" % deg , np.max(error)
    pl.semilogy(x, error, label=u"%d阶多项式的误差" % deg)
pl.legend(loc=3)
pl.axis('tight')
pl.show()
```

多项式函数




```
poly 3: [ -1.45021094e-01 -6.39518172e-16
          9.88749145e-01 -4.29811537e-15]
max error of order 3: 0.00894699976708
poly 5:  [7.57279944e-03 -2.50656614e-17
          -1.65823793e-01   -2.72346001e-18
           9.99770071e-01   7.17317591e-18]
max error of order 5: 0.00015740861417
poly 7: [ -1.84445514e-04  3.70441786e-17
          8.30942467e-03 -1.24633291e-16
           -1.66651593e-01   7.40085118e-17
          9.99997468e-01 -8.11201916e-18]
max error of order 7: 1.52682558119e-06
```

多项式函数

numpy.polynomial模块中提供了更丰富的多项式函数类，例如Polynomial、Chebyshev、 Legendre等。它们和前面介绍的numpy.poly1d相反，多项式各项的系数按照幂从小到大的顺序排列。

```
>>>  from numpy.polynomial import Polynomial, Chebyshev

>>> p = Polynomial([1, -2, 0, 1])
>>> print p(2.0)
5.0

>>> p.deriv()
Polynomial([-2.,  0.,  3.], [-1.,  1.], [-1.,  1.])
>>> Polynomial?
```

多项式函数

切比雪夫多项式是一个正交多项式序列Ti(x),一个n次多项式可以表示为多个切比雪夫多项式的加权和。在NumPy中，使用Chebyshev类表示由切比雪夫多项式组成的多项式p(x):

$$p(x) = \sum_{i=0}^{n} c_i T_i(x)$$

Ti(x)多项式可以通过Chebyshev.basis(i)获得。通过多项式类的convert()方法可以在不同类型的多项式之间相互转换，转换的目标类型由kind参数指定。

```
>>>  Chebyshev.basis(4).convert(kind=Polynomial)
Polynomial([ 1.,  0., -8.,  0.,  8.], [-1.,  1.], [-1.,  1.])
#T4(x) = 1- 8x^2 + 8x^4。
```

多项式函数

切比雪夫多项式的根被称为切比雪夫节点，可以用于多项式插值。相应的插值多项式能最大限度地降低龙格现象，并且提供多项式在连续函数的最佳一致逼近。

使用两种取样点分别对f（x）进行多项式插值，一是在[-1,1]区间上等距离取n个取样点，另一个是使用n阶切比雪夫多项式的根作为取样点。(Chebyshev.py)

多项式函数

```
import numpy as np
from numpy.polynomial import  Chebyshev


def f(x):
    return 1.0/ (1+25*x**2)

n = 11
x1 = np.linspace(-1, 1, n)
x2 = Chebyshev.basis(n).roots()
xd = np.linspace(-1,1, 200)


c1 = Chebyshev.fit(x1, f(x1), n - 1, domain=[-1,1])
c2 = Chebyshev.fit(x2, f(x2), n - 1, domain=[-1,1])


print u"插值多项式的最大误差：",
print u"等距离取样点：", abs(c1(xd) - f(xd)).max(),
print u"契比雪夫节点：", abs(c2(xd) - f(xd)).max()
```

分段函数

在上节中介绍了如何使用frompyfunc()函数计算三角波形。由于三角波形是分段函数，需要根据自变量的取值范围决定计算函数值的公式，因此无法直接通过ufunc函数计算它。

NumPy提供了一些计算分段函数的方法。

numpy_piecewise.py
用where()和piecewise()计算三角波形

Python 2.6中新增了如下的判断表达式语法，当condition条件为True时，表达式的值为y， 否则为z:

```
x = y if condition else z
```

分段函数

在NumPy中，where()函数可以看做判断表达式的数组版本：

```
x = where(condition, y, z)
```

其中，condition、y和z都是数组，它的返回值是一个形状与condition相同的数组。当condition中的某个元素为True时，数组x中对应下标的值从数组y获取，否则从数组z获取：

```
>>> x = np.arange(10)
>>> np.where(x<5, 9-x, x)
array([9，  8，  7，  6，  5，  5，  6，  7，  8，  9])
```

分段函数

如果y和z是单个的数值或者它们的形状与condition的不同,将先通过广播运算使其形状一致:

```
>>> np.where(x>6, 2*x, 0)
array([ 0, 0, 0, 0, 0, 0, 0, 14, 16, 18])
```

使用where()很容易计算上节介绍的三角波形。

```
def triangle_wave(x, c, c0, hc):
    x = x - x.astype(np.int) #三角波的周期为1，因此只取x坐标的小数部分进行计算
    return np.where(x>=c,0,np.where(x<c0, x/c0*hc, (c-x)/(c-c0)*hc))
```

分段函数

由于三角波形分为三段，因此需要两个嵌套的where()进行计算.由于所有的运算和循环都在C语言级别完成，因此它的计算效率比frompyfunc()高.

随着分段函数的分段数量的增加，需要嵌套更多层where(),但这样做不便于程序的编写和阅读。可以用select()解决这个问题，它的调用形式如下：

```
select(condlist, choicelist, default=0)
```

分段函数

```
select(condlist, choicelist, default=0)
# np.select?
>>> np.select([x<2,x>6,True],[7-x,x,2*x])
array([ 7,  6,  4,  6,  8, 10, 12,  7,  8,  9])
```

其中，condlist是一个长度为N的布尔数组列表，choicelist是一个长度为N的储存候选值的数组列表，所有数组的长度都为M.如果列表元素不是数组而是单个数值，那么它相当于元素值都相同且长度为M的数组。

分段函数

对于从0到M-1的数组下标i,从布尔数组列表中找出满足条件“condlist[j][i]=True”的 j 的最小值，则“out[i]=choicelist[j][i]”，其中out是select()的返回数组。可以使用select()计算三角波形：

```
def triangle_wave2(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.select([x>=c, x<c0, True], [0, x/c0*hc,
(c-x)/(c-c0)*hc])
```

分段函数

由于分段函数分为三段，因此每个列表都有三个元素。choicelist的最后一个元素为True,表示前面所有条件都不满足时，将使用choicelist的最后一个数组中的值。也可以用default参数指定条件都不满足时的候选值数组：

```
return np.select([x>=c, x<c0], [0, x/c0*hc],
default=(c-x)/(c-c0)*hc)
```

但是where()和select()的所有参数都需要在调用它们之前完成计算，因此NumPy会计算下面4个数组：

```
x>=c, x<c0, x/c0*hc, (c-x)/(c-c0)*hc
```

分段函数

在计算时还会产生许多保存中间结果的数组，因此如果输入的数组x很大，将会发生大量的内存分配和释放。为了解决这个问题,NumPy提供了 piecewise()专门用于计算分段函数，它的调用参数如下

```
piecewise(x, condlist, funclist)
```

分段函数

piecewise(x, condlist, funclist)

参数x是一个保存自变量值的数组.condlist是一个长度为N的布尔数组列表，其中的每个布尔数组的长度都和数组x相同。funclist是一个长度为N或N+1的函数列表，这些函数的输入和输出都是数组。它们计算分段函数中的每个片段。如果不是函数而是数值，就相当于返回此数值的函数。每个函数与condlist中下标相同的布尔数组对应，如果funclist的长度为N+1, 那么最后一个函数对应于所有条件都为False时。

分段函数

下面是使用piecewise()计算三角波形的程序:

```
def triangle_wave3(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.piecewise(x,
            [x>=c, x<c0],
            [0, # x>=c
            lambda x: x/c0*hc, # x<c0
            lambda x: (c-x)/(c-c0)*hc]) # else
```

使用piecewise()的好处在于它只计算需要计算的值.因此在上面的例子中，表达式“x/c0*hc”和“(c-x)/(c-c0)*hc”只对输入数组x中满足条件的部分进行计算。

分段函数

```
# -*- coding: utf-8 -*-
"""
使用where, select, piecewise等计算三角波形的分段函数。
"""
import numpy as np

def triangle_wave(x, c, c0, hc):
    x = x - x.astype(np.int) # 三角波的周期为1，因此只取x坐标的小数部分进行计算
    return np.where(x>=c, 0, np.where(x<c0, x/c0*hc, (c-x)/(c-c0)*hc))

def triangle_wave2(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.select([x>=c, x<c0, True], [0, x/c0*hc, (c-x)/(c-c0)*hc])
```

分段函数

```
def triangle_wave3(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.piecewise(x,
        [x>=c, x<c0],
        [0, # x>=c
        lambda x: x/c0*hc, # x<c0
        lambda x: (c-x)/(c-c0)*hc]) # else

def triangle_wave4(x, c, c0, hc):
"""显示每个分段函数计算的数据点数"""
    def f1(x):
        print "f1:", x.shape
        return x/c0*hc
    def f2(x):
        print "f2:", x.shape
        return (c-x)/(c-c0)*hc
    x = x - x.astype(np.int)
    return np.piecewise(x, [x>=c, x<c0], [0, f1, f2])
```

分段函数

```
x = np.linspace(0, 2, 1000)
y = triangle_wave(x, 0.6, 0.4, 1.0)
y2 = triangle_wave2(x, 0.6, 0.4, 1.0)
y3 = triangle_wave3(x, 0.6, 0.4, 1.0)
y4 = triangle_wave4(x, 0.6, 0.4, 1.0)
```

triangle_wave4()验证了每个函数所计算的数组的长度。

```
np.alltrue(y==y2) and np.alltrue(y2==y3) and
np.alltrue(y3==y4)
```

统计函数

随机数

numpy.random模块中提供了大量的随机数相关的函数

rand()产生0到1之间的随机浮点数；randn()产生标准正态分布的随机数；randint()产生指定范围的随机整数；

```
>>> from numpy import random as nr
>>> np.set_printoptions(precision=2) # 节省篇幅，只显小数点后两位数字
>>> r1=nr.rand(4, 3)
>>> r2 = nr.randn(4, 3)
>>> r3 = nr.randint(0, 10, (4, 3))
```

统计函数

random模块提供了许多产生符合特定随机分布的随机数的函数。

normal():正态分布，前两个参数分别为期望值和标准差。uniform():均匀分布，前两个参数分别为区间的起始值和终值。poisson():泊松分布，第一个参数指定λ系数,它表示单位时间(或单位面积)内随机事件的平均发生率。由于泊松分布是一个离散分布，因此它输出的数组是一个整数数组。。

```
>>> r1=nr.normal(100, 10, (4, 3))
>>> r2 = nr.uniform(10, 20, (4, 3))
>>> r3 = nr.poisson(2.0, (4, 3))
```

统计函数

permutation()可以用于产生一个乱序数组，当参数为整数n时，它返回[0,n)这n个整数的随机排列；当参数为一个序列时，它返回一个随机排列之后的序列。permutation()返回一个新数组，而shuffle()则直接将参数数组的顺序打乱：

```
>>> print nr.permutation(10)
[8 9 6 0 1 2 3 7 4 5]
>>> a = np.array([1, 10, 20, 30, 40])
>>> print nr.permutation(a)
[10 30 20  1 40]
>>>a
>>> nr.shuffle(a)
>>> a
```

统计函数

choice()从指定的样本中随机进行抽取：

size参数用于指定输出数组的形状；

replace参数为True时，进行可重复抽取，而为False时进行不重复抽取，默认为True；

p参数指定每个元素对应的抽取概率。

```
>>> a = np.arange(10, 25, dtype=float)

>>> c1=nr.choice(a, size=(4, 3))
>>> c2 = nr.choice(a, size=(4, 3), replace=False)
>>> c3 = nr.choice(a, size=(4, 3), p=a / np.sum(a))
#值越大的元素被抽到的概率越大
```

统计函数

为了保证每次运行时能出现相同的随机数，可通过seed()函数指定随机数的种子。 r3和r4之前，都使用41作为种子，因此得到的随机数组是相同的。

```
>>> r1 = nr.randint(0, 100, 3)
>>> r2 = nr.randint(0, 100, 3)
>>> nr.seed(41)
>>> r3 = nr.randint(0, 100, 3)
>>>nr.seed(41)
>>> r4 = nr.randint(0, 100, 3)
```

统计函数

unique():返回其参数数组中所有不同的值，并按从小到大的顺序排列。它有两个可选参数：

- return_index：Ture表示同时返回原始数组中的下标。
- return_inverse: True表示返回重建原始数组用的下标数组。

下面通过实例介绍unique()的用法。首先用randint()创建含有10个元素、值在0到9范围之内的随机整数数组：

```
>>> a =np.random.randint(0,10,10)
>>> a
array([1, 1, 9, 5, 2, 6, 7, 6, 2, 9])
```

统计函数

通过unique(a)可以找到数组a中所有的整数，并按照顺序排列：

```
>>> np.unique(a)
array([1, 2, 5, 6, 7, 9])
```

如果参数return_index为True,就返回两个数组,第二个数组是第一个数组在原始数组中的下标：

```
>>> x, idx = np.unique(a, return_index=True)
>>> x
array([1, 2, 5, 6, 7, 9])
>>> idx
array([0, 4, 3, 5, 6, 2])
```

统计函数

数组idx保存的是数组x中每个元素在数组a中的下标：

```
>>> a[idx]
array([1, 2, 5, 6, 7, 9])
```

如果参数return_inverse为True,那么返回的第二个数组是原始数组a中每个元素在数组x 中的下标：

```
>>> x, ridx = np.unique(a, return_inverse=True)
>>> ridx
array([0, 0, 5, 2, 1, 3, 4, 3, 1, 5])
>>>all(x[ridx]==a) #原始数组a和x[ridx]完全相同
True
```

统计函数

bincount():对整数数组中各个元素出现的次数进行统计，它要求数组中所有元素都是非负的。其返回数组中第i个元素的值表示整数i在参数数组中出现的次数。

```
>>> np.bincount(a)
array([0, 2, 2, 0, 0, 1, 2, 1, 0, 2])
```

由上面的结果可知,在数组a中有两个1、两个2、一个5、两个6、一个7和两个9,而 0、3、4、8等数没有在数组a中出现。

统计函数

通过weights参数可以指定每个数对应的权值。当指定weights参数时，bincount(x, weights=w)返回数组x中每个整数所对应的w中的权值之和。实例：

```
>>> x =np.array([0 , 1, 2, 2, 1, 1, 0])
>>> w = np.array([0.1, 0.3, 0.2,0.4,0.5,0.8,1.2])
>>> np.bincount(x, w)
array ([ 1.3,1.6,0.6])
```

上面的结果中，1.3是数组x中0所对应的w中的值(0.1和1.2)的和，1.6是1所对应的w 中的值(0.3、0.5和0.8)的和，而0.6是2所对应的w中的值(0.2和0.4)的和.

统计函数

如果要求权重平均值，可以用求和结果与次数相除：

```
>>> np.bincount(x,w)/np.bincount(x)
array([ 0.65,  0.53333333,  0.3])
```

统计函数

percentile()：百分位数是统计中使用的度量，表示小于这个值得观察值占某个百分比。

numpy.percentile(a, q, axis)

其中：a 输入数组；q 要计算的百分位数，在 0 ~ 100 之间；axis 沿着它计算百分位数的轴。

```
>>> a = np.array([[30,40,70],[80,20,10],[50,90,60]])
>>> print np.percentile(a,50)
50.0
>>> print np.percentile(a,50, axis =  1)
[ 40.  20.  60.]
>>> print np.percentile(a,70)
66.0
```

统计函数

histogram() :对一维数组进行直方图统计，其参数列表如下：

```
histogram(a,bins=10,range=None,normed=False,weights=None)
```

其中，a是保存待统计数据的数组，bins指定统计的区间个数，即对统计范围的等分数。range是一个长度为2的元组，表示统计范围的最小值和最大值，默认值为None,表示范围由数据的范围决定，即(a.min(), a.max()).当normed参数为False时，函数返回数组a中的数据在每个区间的个数，否则对个数进行正规化处理，使它等于每个区间的概率密度。weights参数和 bincount()的类似。

统计函数

histogram()返回两个一维数组--hist和bin_edges,第一个数组是每个区间的统计结果,第二个数组长度为len(hist)+1,每两个相邻的数值构成一个统计区间。下面我们看一个例子:

```
>>> a = np. random.rand (100)
>>> np.histogram(a,bins=5,range=(0,1))
(array([20,26,20,16,18]), array([ 0. , 0.2, 0.4, 0.6, 0.8, 1.])
```

首先创建了一个一维随机数组a,然后用histogram()对数组a中的数据进行直方图统计。结果显示有20个元素的值在0到0.2之间，26个元素的值在0.2到0.4之间. rand()所创建的随机数在0到1范围之间是平均分布的。

统计函数

如果需要统计的区间长度不等，可以将表示区间分隔位置的数组传递给bins参数，例如:

```
>>> np.histogram(a,bins=[0, 0.4, 0.8, 1.0], range=(0,1))
(array([46, 36, 18]), array([ 0. , 0.4, 0.8, 1.]))
```

结果表示：0到0.4之间有46个值，0.4對0.8之间有36个值。

如果用weights参数指定了数组a中每个元素对应的权值，那么histogram()将对区间中数值对应的权值进行求和.

统计函数

一个使用histogram()统计男青少年年龄和身高的例子。height.csv（ 100名年龄在7到20岁之间的男性青少年的身高统计数据）

首先用loadtxt()从数据文件载入数据:

```
>>> d = np.loadtxt("height.csv",delimiter=",")
>>> d.shape
(100, 2)
```

在数组d中，第0列是年龄，第1列是身高。可以看到年龄的范围在7到20之间:

```
>>> np.min(d[:,0])
 7.099999999999996
>>> np.max(d[:,0])
19.899999999999999
```

统计函数

下面对数据进行统计，sums是每个年龄段的身高总和，cnts是每个年龄段的数据个数，因此很容易计算出每个年龄段的平均身高：

```
>>> sums =
np.histogram(d[:,0],bins=range(7,21),range=(7,20),weig
hts=d[:,1])[0]
>>> cnts =np.histogram (d[:,0], bins=range(7,21),
range=(7,20))[0]
>>>sums/cnts
array([ 125.96,  132.06666667, 137.82857143, 143.8   ,
148. 14 ,153.44, 162.15555556, 166.86666667,
172.83636364, 173.3,175.275, 174.19166667,
        175.075])
```

解线性方程组

对矩阵的一些更高级运算可以在numpy的线性代数模块linalg中找到。例如inv()计算逆矩阵，solve()可以求解多元一次方程组.下面是solve()的例子：

```
>>> 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()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的一维数使，solve()找到一个长度为N的一维数组X，使得它们满足a • x = b,即x就是这个N元一次方程组的解.

。

解线性方程组

lstsq()比solve()更一般化，它不要求矩阵a是正方形的，也就是说方程的个数可以少于、 等于或多于未知数的个数。它找到一组解x,使得丨b-a •x丨的平方和最小。我们称得到的结果为最小二乘解，即它使得所有等式的误差的平方和最小。

numpy.linalg.lstsq(*a*, *b*, *rcond=-1*)

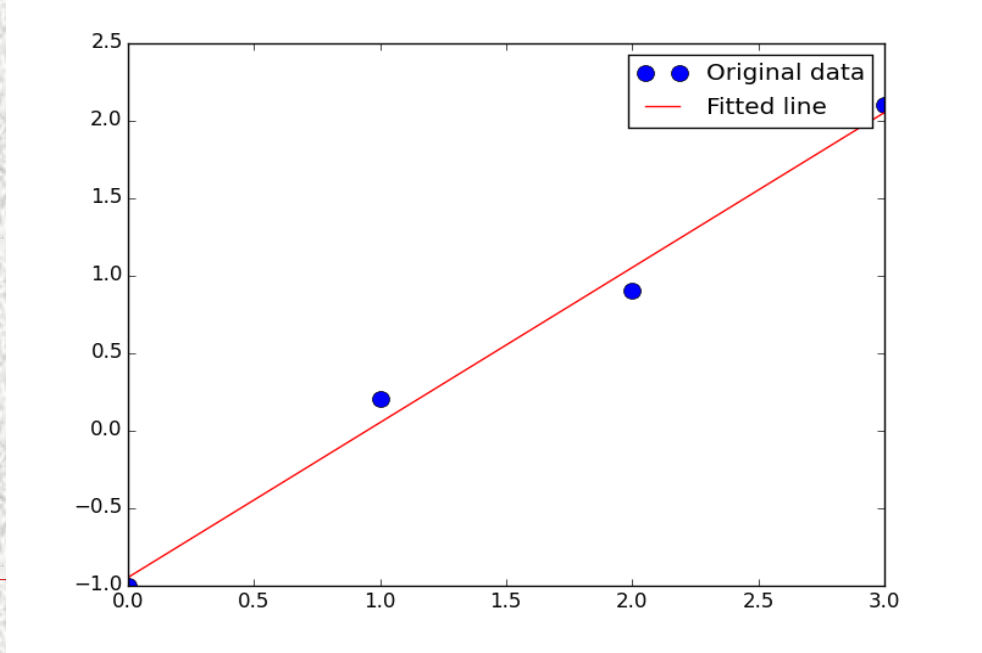
rcond *: float, optional*
Returns:
X*:ndarray* Least-squares solution.
residuals *: ndarray* Sums of residuals;
rank *: int* Rank of matrix *a*.
s *: (min(M, N),) ndarray* Singular values of *a*.

解线性方程组

```
#Examples
#Fit a line, y = mx + c, through some noisy data-points:
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A        # y= A*[m,c]^T
array([[ 0., 1.],
       [ 1., 1.],
       [ 2., 1.],
       [ 3., 1.]])

>>> m, c = np.linalg.lstsq(A, y)[0]
>>> print m, c
1.0 -0.95
```

解线性方程组

```
#Plot the data along with the fitted line:

import matplotlib.pyplot as plt

plt.plot(x, y, 'o', label='Original data', markersize=10)
plt.plot(x, m*x + c, 'r', label='Fitted line')
plt.legend()
plt.show()
```



NumPy

—NumPy模块

目录

- numpy.linalg模块
- numpy线性代数小结
- numpy.fft模块
- numpy.random模块
- 直方图(histogram)

numpy.linalg模块

numpy.linalg模块包含线性代数的函数。使用这个模块，可以计算逆矩阵、求特征值、解线性方程组以及求解行列式等.

- 计算逆矩阵

在线性代数中，矩阵A与其逆矩阵A^{-1}相乘后会得到一个单位矩阵I。该定义可以写为$A*A^{-1}=I$。

numpy.linalg模块中的inv函数可以计算逆矩阵。按如下步骤来对矩阵求逆。

numpy.linalg模块

使用numpy.linalg模块中的inv函数计算逆矩阵，并检查了原矩阵与求得的逆矩阵相乘的结果确为单位矩阵。

```
import numpy as np

A = np.mat("0 1 2;1 0 3;4 -3 8") #使用mat函数创建示例矩阵
#A = np.array([[0,1,2],[1,0,3],[4,-3,8]])
print "A\n", A

inverse = np.linalg.inv(A)
print "inverse of A\n", inverse

print "Check\n", A * inverse
#print "Check\n", np.dot(A,inverse)
```

numpy.linalg模块

- 求解线性方程组

numpy.linalg中的函数solve可以求解形如 Ax = b 的线性方程组，其中 A 为矩阵，b 为一维或二维的数组，x 是未知变量。

```
import numpy as np

A = np.mat("1 -2 1;0 2 -8;-4 5 9")
print "A\n", A
b = np.array([0, 8, -9])
print "b\n", b
x = np.linalg.solve(A, b)
print "Solution", x

print "Check\n", np.dot(A , x)
```

numpy.linalg模块

- 特征值和特征向量

  特征值（eigenvalue）即方程 Ax = ax 的根，a是一个标量。其中，A 是一个二维矩阵，x 是一个一维向量。特征向量（eigenvector）是关于特征值的向量。在numpy.linalg模块中，eigvals函数可以计算矩阵的特征值，而eig函数可以返回一个包含特征值和对应的特征向量的元组。

numpy.linalg模块

```
import numpy as np

A = np.mat("3 -2;1 0")
print "A\n", A

print "Eigenvalues", np.linalg.eigvals(A)

eigenvalues, eigenvectors = np.linalg.eig(A)
print "First tuple of eig", eigenvalues
print "Second tuple of eig\n", eigenvectors

for i in range(len(eigenvalues)):
    print "Left", np.dot(A, eigenvectors[:,i])
    print "Right", eigenvalues[i] * eigenvectors[:,i]
```

numpy.linalg模块

- 奇异值分解

在numpy.linalg模块中的svd函数可以对矩阵进行奇异值分解。该函数返回3个矩阵U、Sigma和V，其中U和V是正交矩阵，Sigma包含输入矩阵的奇异值。

```
import numpy as np
A = np.mat("4 11 14;8 7 -2")
print "A\n", A
U, Sigma, V = np.linalg.svd(A, full_matrices=False)
print U
print Sigma
print V
print "Product\n", U * np.diag(Sigma) * V
print U*U.T
print V*V.T
```

numpy.linalg模块

- 广义逆矩阵

  摩尔·彭罗斯广义逆矩阵（Moore-Penrose pseudoinverse）可以使用numpy.linalg模块中的pinv函数进行求解。计算广义逆矩阵需要用到奇异值分解。inv函数只接受方阵作为输入矩阵，而pinv函数则没有这个限制。

```
import numpy as np
A = np.mat("4 11 14;8 7 -2")
print "A\n", A
pseudoinv = np.linalg.pinv(A)
print "Pseudo inverse\n", pseudoinv
print "Check", A * pseudoinv
```

numpy.linalg模块

- 行列式

  numpy.linalg模块中的det函数可以计算矩阵的行列式。

```
import numpy as np

A = np.mat("3 4;5 6")
print "A\n", A

print "Determinant", np.linalg.det(A)
```

numpy.linalg模块

- 矩阵的秩

```
>>> import numpy as np

>>> I = np.eye(3)#先创建一个单位阵
>>> I
array([[ 1., 0., 0.],  [ 0., 1., 0.],  [ 0., 0., 1.]])

>>> np.linalg.matrix_rank(I)#秩
3

>>> I[1, 1] = 0#将该元素置为0
>>> I
array([[ 1., 0., 0.],  [ 0., 0., 0.], [ 0., 0., 1.]])

>>> np.linalg.matrix_rank(I)#此时秩变成2
2
```

Numpy线性代数小结

1.建立矩阵

a1=np.array([1,2,3],dtype=int)　#建立一个一维数组，数据类型是int。也可以不指定数据类型，使用默认。几乎所有的数组建立函数都可以指定数据类型，即dtype的取值。

a2=np.array([[1,2,3],[2,3,4]])　#建立一个二维数组。此处和MATLAB的二维数组（矩阵）的建立有很大差别。

同样，numpy中也有很多内置的特殊矩阵：
b1=np.zeros((2,3))　#生成一个2行3列的全0矩阵。注意，参数是一个tuple：(2,3)，所以有两个括号。完整的形式为：zeros(shape,dtype=)。相同的结构，有ones()建立全1矩阵。empty()建立一个空矩阵，使用内存中的随机值来填充这个矩阵。

b2=identity(n)　#建立n*n的单位阵，这只能是一个方阵。

Numpy线性代数小结

b3=eye(N,M=None,k=0)　#建立一个对角线是1其余值为0的矩阵，用k指定对角线的位置。M默认None。

此外，numpy中还提供了几个like函数，即按照某一个已知的数组的规模（几行几列）建立同样规模的特殊数组。这样的函数有zeros_like()、empty_like()、ones_like()，它们的参数均为如此形式：zeros_like(a,dtype=)，其中，a是一个已知的数组。

c1=np.arange(2,3,0.1)　#起点，终点，步长值。含起点值，不含终点值。

c2=np.linspace(1,4,10)　#起点，终点，区间内点数。起点终点均包括在内。同理，有logspace()函数，等比数组。

Numpy线性代数小结

d1=np.linalg.companion(a)　#伴随矩阵

d2=np.linalg.triu()/tril()　#作用同MATLAB中的同名函数

np.fliplr()/np.flipud()/np.rot90()　#功能类似MATLAB同名函数，翻转。

xx=np.roll(x,2)　#np.roll? roll()是循环移位函数。此调用表示向右循环移动2位。

Numpy线性代数小结

- 2.数组的特征信息：

  先假设已经存在一个N维数组X了，那么可以得到X的一些属性，这些属性可以在输入X和一个.之后，按tab键查看提示。这里明显看到了Python面向对象的特征。

```
X.flags     #数组的存储情况信息。

X.shape     #结果是一个tuple，返回本数组的行数、列数、……

X.ndim    #数组的维数，结果是一个数。

X.size     #数组中元素的数量

X.itemsize     #数组中的数据项的所占内存空间大小
```

Numpy线性代数小结

```
X.dtype    #数据类型

X.T   #如果X是矩阵，发挥的是X的转置矩阵

X.trace()    #计算X的迹

np.linalg.det(a)   #返回的是矩阵a的行列式

np.linalg.norm(a,ord=None)    #计算矩阵a的范数

np.linalg.eig(a)    #矩阵a的特征值和特征向量

np.linalg.cond(a,p=None)    #矩阵a的条件数

np.linalg.inv(a)    #矩阵a的逆矩阵
```

Numpy线性代数小结

- 3.矩阵分解

  常见的矩阵分解函数，numpy.linalg均已经提供。比如cholesky() /qr() /svd() /lu() /schur()（舒尔分解）等。某些算法为了方便计算或者针对不同的特殊情况，还给出了多种调用形式，以便得到最佳结果。

- 4.矩阵运算

  np.dot(a,b)用来计算数组的点积；
  vdot(a,b)专门计算矢量的点积，和dot()的区别在于对complex数据类型的处理不一样；
  inner(a,b)用来计算内积；
  outer(a,b)计算外积。

Numpy线性代数小结

- 5.解方程

np.linalg.lstsq

np.linalg.solve

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> np.allclose(np.dot(a, x), b)
True
```

Numpy线性代数小结

- 6.索引

  numpy中的数组索引形式和Python是一致的。

```
x=np.arange(10)

print x[2]    #单个元素，从前往后正向索引。注意下标是从0开始的。

print x[-2]    #从后往前索引。最后一个元素的下标是-1

print x[2:5]    #多个元素，左闭右开，默认步长值是1

print x[:-7]    #多个元素，从后向前，制定了结束的位置，使用默认步长值
```

Numpy线性代数小结

```
print x[1:7:2]    #指定步长值

x.shape=(2,5)     #x的shape属性被重新赋值，要求就是元素个数不变。2*5=10

print x[1,3]      #二维数组索引单个元素，第2行第4列的那个元素

print x[0]    #第一行所有的元素

y=np.arange(35).reshape(5,7)     #reshape()函数用于改变数组的维度

print y[1:5:2,::2]     #选择二维数组中的某些符合条件的元素
```

numpy.fft模块

- 快速傅里叶变换

  在NumPy中，有一个名为fft的模块提供了快速傅里叶变换的功能。在这个模块中，许多函数都是成对存在的，也就是说许多函数存在对应的逆操作函数。例如，fft和ifft函数就是其中的一对。

```
import numpy as np
from matplotlib.pyplot import plot, show

x = np.linspace(0, 2 * np.pi, 30) #创建一个包含30个点的余弦波信号
wave = np.cos(x)
```

numpy.fft模块

```
transformed = np.fft.fft(wave)  #使用fft函数对余弦波信号进行傅里叶变换。

print np.all(np.abs(np.fft.ifft(transformed) - wave) < 10 ** -9)  #对变换后的结果应用ifft函数，应该可以近似地还原初始信号。

plot(transformed)  #使用Matplotlib绘制变换后的信号。
show()
```

numpy.fft模块

- 移频

numpy.fft模块中的fftshift函数可以将FFT输出中的直流分量移动到频谱的中央。ifftshift函数则是其逆操作。

```
import numpy as np
from matplotlib.pyplot import plot, show

x = np.linspace(0, 2 * np.pi, 30)
wave = np.cos(x)  #创建一个包含30个点的余弦波信号。

transformed = np.fft.fft(wave)  #使用fft函数对余弦波信号进行傅里叶变换。
```

numpy.fft模块

```
shifted = np.fft.fftshift(transformed) #使用fftshift函数进行移频操作。

print np.all((np.fft.ifftshift(shifted) - transformed) < 10 ** -9)  #用ifftshift函数进行逆操作，这将还原移频操作前的信号。

plot(transformed, lw=2)
plot(shifted, lw=3)
show()    #使用Matplotlib分别绘制变换和移频处理后的信号。
```

numpy.random模块

- 二项分布函数

随机数的函数可以在NumPy的random模块中找到。

numpy.random. binomial(n, p, size=None)

其中n,p,size分别是每轮试验次数、概率、轮数，函数的返回值表示成功的次数。

```
>>> n, p = 2, .5              # 两枚都是正面
>>> sum(np.random.binomial(n, p, size=20000)==2)/20000.
0.25490000000000002

# 其中一个为反面
>>> sum(np.random.binomial(n, p, size=20000)==1)/20000.
0.501
```

numpy.random模块

```
>>> n, p = 2, .5                # 两个都是反面
>>> sum(np.random.binomial(n, p, size=20000)==0)/20000.
0.24940000000000001
```

硬币赌博游戏：

设想你来到了一个一世纪的赌场，正在对一个硬币赌博游戏下若干份赌注。每一轮抛9枚硬币，如果少于5枚硬币正面朝上，你将损失赌注中的1份；否则，你将赢得1份赌注。模拟一下赌博的过程，初始资本为1000份赌注。为此，需要使用random模块中的binomial函数(二项分布函数)。

numpy.random模块

```
import numpy as np
from matplotlib.pyplot import plot, show

#初始化一个全0的数组来存放剩余资本。以参数10000调用
binomial函数，意味着将在赌场中玩10000轮硬币赌博游戏。
cash = np.zeros(10000)
cash[0] = 1000
outcome = np.random.binomial(9, 0.5, size=len(cash))

# np.random.binomial(n,p,size=N),函数的返回值表示n中成功
（success）的次数.
```

numpy.random模块

```
#模拟每一轮抛硬币的结果并更新cash数组。打印出outcome的最小值和最大值，以检查输出中是否有任何异常值：将在赌场中玩10000轮硬币赌博游戏。
for i in range(1, len(cash)):
        if outcome[i] < 5:
                cash[i] = cash[i - 1] - 1
        elif outcome[i] < 10:
                cash[i] = cash[i - 1] + 1
        else:
                raise AssertionError("Unexpected outcome " + outcome)
print outcome.min(), outcome.max()

#使用Matplotlib绘制cash数组：
plot(np.arange(len(cash)), cash)
show()
```

numpy.random模块

- 超几何分布

超几何分布（hypergeometric distribution）是一种离散概率分布，它描述的是一个罐子里有两种物件，无放回地从中抽取指定数量的物件后，抽出指定种类物件的数量。NumPy random模块中的hypergeometric函数可以模拟这种分布。

模拟游戏秀节目

设想有这样一个游戏秀节目，每当参赛者回答对一个问题，他们可以从一个罐子里

numpy.random模块

摸出3个球并放回。罐子里有一个“倒霉球”，一旦这个球被摸出，参赛者会被扣去6分。而如果他们摸出的3个球全部来自其余的25个普通球，那么可以得到1分。因此，如果一共有100道问题被正确回答，得分情况会是怎样的呢？

```
import numpy as np
from matplotlib.pyplot import plot, show
```

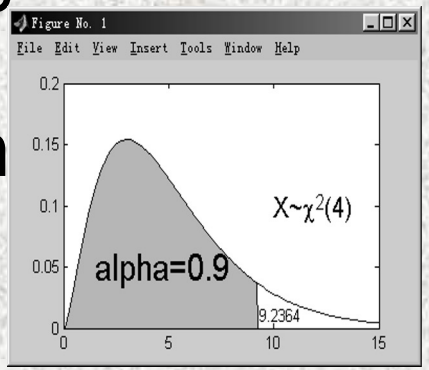
numpy.random模块

```
points = np.zeros(100)
outcomes = np.random.hypergeometric(25, 1, 3,
size=len(points))
#使用hypergeometric函数初始化游戏的结果矩阵。该函数的第一个参数为罐中普通球的数量，第二个参数为“倒霉球”的数量，第三个参数为每次采样（摸球）的数量。

for i in range(len(points)):  #产生的游戏结果计算相应的得分。
    if outcomes[i] == 3:
        points[i] = points[i - 1] + 1
    elif outcomes[i] == 2:
        points[i] = points[i - 1] - 6
    else:
        print outcomes[i]

plot(np.arange(len(points)), points)
show()
```

numpy.random模块

- 连续分布

  连续分布可以用PDF（Probability Density Function，概率密度函数）来描述。随机变量落在某一区间内的概率等于概率密度函数在该区间的曲线下方的面积。NumPy的random模块中有一系列连续分布的函数——beta、chisquare、exponential、f、gamma、gumbel、laplace、lognormal、logistic、multivariate_normal、noncentral_chisquare、noncen
  normal等。



numpy.random模块

绘制正态分布

随机数可以从正态分布中产生，它们的直方图能够直观地刻画正态分布。

```
import numpy as np
import matplotlib.pyplot as plt


#使用NumPy random模块中的normal函数产生指定数量的随机数。
N=10000
normal_values = np.random.normal(size=N)
```

numpy.random模块

#绘制分布直方图和理论上的概率密度函数（均值为0、方差为1的正态分布）曲线。将使用Matplotlib进行绘图。

```
dummy, bins, dummy11 = plt.hist(normal_values,
np.sqrt(N), normed=True, lw=1)     #np.sqrt(N)需用100替换
sigma = 1
mu = 0
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( -
(bins -mu)**2 / (2 *
sigma**2) ),lw=2)
plt.show()
```

numpy.random模块

对数正态分布： 绘制出对数正态分布的概率密度函数以及对应的分布直方图。

```
import numpy as np
import matplotlib.pyplot as plt
N=10000
lognormal_values = np.random.lognormal(size=N)
#绘制分布直方图和理论上的概率密度函数（均值为0、方差为1）
dummy, bins, dummy12 = plt.hist(lognormal_values,
100,normed=True, lw=1)
sigma = 1
mu = 0
x = np.linspace(min(bins), max(bins), len(bins))
pdf = np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))/
(x * sigma * np.sqrt(2 * np.pi))
plt.plot(x, pdf,lw=3)
plt.show()
```

直方图(histogram)

NumPy中histogram函数应用到一个数组返回一对变量：直方图数组。

注意：matplotlib也有一个用来建立直方图的函数(叫作hist,正如matlab中一样)与NumPy中的不同。主要的差别是pylab.hist自动绘制直方图，而numpy.histogram仅仅产生数据。

直方图(histogram)

```
import numpy
import pylab
# Build a vector of 10000 normal deviates with variance
0.5^2 and mean 2
mu, sigma = 2, 0.5
v = numpy.random.normal(mu,sigma,10000)
# Plot a normalized histogram with 50 bins
pylab.hist(v, bins=50, normed=1)
# matplotlib version (plot)
pylab.show()
 # Compute the histogram with numpy and then plot it
(n, bins) = numpy.histogram(v, bins=50, normed=True)
# NumPy version (no plot)
pylab.plot(.5*(bins[1:]+bins[:-1]), n)
pylab.show()
```

直方图(histogram)

- 数据区间划分

可以有效地将数据进行区间划分并手动创建直方图。例如，假定有 1000 个值，希望快速统计分布在每个区间中的数据频次，可以用

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
x = np.random.randn(1000)

bins = np.linspace(-5, 5, 20) # 手动计算直方图
counts = np.zeros_like(bins)

i = np.searchsorted(bins, x) # 为每个x找到合适的区间
```

直方图(histogram)

```
np.add.at(counts, i, 1) # 为每个区间加上1

plt.plot(bins, counts, linestyle='steps') # 画出结果
```

当然，Matplotlib 提供了 plt.hist() 方法，仅用一行代码就实现了上述功能：

```
plt.hist(x, bins, histtype='step')
```

比较一下这两种方法：

```
x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)
print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)

np.histogram??   #查看源代码
```