Python 骑士

第五部：数学计算

——一个路过的假面骑士

第一章. math库的使用

1. 库的导入

import 库名 as xxx
from 库名 import xxxx

2.math库对数学运算的一些支持

math库的一些函数

```
math.pi #圆周率
math.e #自然对数底数
math.isnan() #判断是不是无穷
math.modf() #返回整数部分与小数部分的元组
math.factorial() #整数阶乘
math.gamma() #浮点阶乘
math.lgamma() #对数阶乘
math.pow() #乘方
math.sqrt() #根号
math.exp() #指数
math.log() #对数
math.log10() #对数
math.redians() #角度化弧度
math.degree() #弧度化角度
math.sin(),math.cos,math.tan() #三角
math.asin(),math.acos(),math.atan() #反三角
math.erf() #高斯误差
```

```
from math import *
>>> print(pi)
3.141592653589793
>>> print(e)
2.718281828459045
>>> print(isnan(2))
False
```

```
>>> modf(3.5)
(0.5, 3.0)
>>> factorial(4)
24
>>> gamma(2.3)
1.16671190519816
>>> lgamma(2.3)
0.15418945495963055
>>> pow(2,4)
16.0
>>> sqrt(5.2)
2.280350850198276
>>> exp(2)
7.38905609893065
>>> log(3,2)
1.5849625007211563
>>> log(4)
1.3862943611198906
>>> log10(2)
0.3010299956639812
```

第二章. random库的使用

1. 生成随机数指令random.random()
2. 生成一定范围内的随机数指令random.uniform()
3. random.seed()设定种子
4. randint()生成的是整数随机数
5. 选择随机数random.choice()
6. random.shuffle()洗牌，排列
7. normalvariate(), gauss()正态分布
8. expovariate()指数分布

```
import random
random.seed(100)#选择随机数种子，只要种子相同，每次产生的随机数相同
a = random.random()#生成0-1之间的小数
print(a)

b = random.randint(10, 100)#生成10-100之间的随机整数
print(b)

c = random.randrange(10, 100,10)#生成10-100之间，以10为步长的随机整数
print(c)

d = random.getrandbits(4)#生成一个4比特长的随机整数
print(d)

e = random.uniform(10, 100)#生成一个10-100之间的随机小数
print(e)

s = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
f = random.choice(s)#从s序列中随机选择一个元素
print(f)

random.shuffle(s)#打乱s
print(s)



import random

# 打乱排序
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

```
random.shuffle(items)
print(items)
#[7, 5, 6, 9, 4, 1, 3, 2, 0, 8]
```

基本原理就是用离散的点的个数模拟连续的概率分布。通过大量的测试，以频率估计概率，从而估计正确结果。当然，我们做的毕竟是有限次实验，估计值绝对有偏差。但是……额……其实也还好（尬笑）
少说废话，直接上代码吧（这里我用的python）：

```
>>> ##利用蒙特卡洛方法估计圆周率的值
>>> import random
>>> import math
>>> def montekarlo():
 pailist=[]
 for i in range(10):
  count=0
  for j in range(10000):
   x=random.random()    ##投掷一个点，计算它的坐标
   y=random.random()
   if x*x+y*y<=1:       ##利用距离公式
    count+=1            ##这里我们计算的是π/4，注意！
  pailist.append(4*count/10000)
  print(4*count/10000)
 sum=0.0000
 for pais in pailist:
  sum+=pais
 print('平均结果:%.4f'%(sum/10))
```

运行了几遍，我就只放一次运行的结果吧

```
>>> montekarlo()
3.1336
3.13
3.1352
3.1576
3.1416
3.138
3.1416
3.128
3.148
3.1572
平均结果:3.1411
```

NumPy 简单入门教程

NumPy是Python中的一个运算速度非常快的一个数学库，它非常重视数组。它允许你在Python中进行向量和矩阵计算，并且由于许多底层函数实际上是用C编写的，因此你可以体验在原生Python中永远无法体验到的速度。

NumPy绝对是科学Python成功的关键之一，如果你想要进入Python中的数据科学和/或机器学习，你就要必须学习它。在我看来，NumPy的API设计得很好，所以我们要开始使用它并不困难。

这是一系列关于科学Python的文章中的第二篇，别忘了看看其他的哟（译者注：并不会放出所有的文章，只摘取部分文章）。

哇哦，你能让NumPy成为你的伙伴并且一起做的事真是太棒了 好吗。

数组基础

创建一个数组

NumPy围绕这些称为数组的事物展开。实际上它被称之为 `ndarrays`，你不知道没事儿。使用NumPy提供的这些数组，我们就可以以闪电般的速度执行各种有用的操作，如矢量和矩阵、线性代数等数学运算！ （开个玩笑，本文章中我们不会做任何繁重的数学运算）

```
# 1D Array
a = np.array([0, 1, 2, 3, 4])
b = np.array((0, 1, 2, 3, 4))
c = np.arange(5)
d = np.linspace(0, 2*np.pi, 5)

print(a) # >>>[0 1 2 3 4]
print(b) # >>>[0 1 2 3 4]
print(c) # >>>[0 1 2 3 4]
print(d) # >>>[ 0.          1.57079633  3.14159265  4.71238898  6.28318531]
print(a[3]) # >>>3
```

上面的代码显示了创建数组的4种不同方法。最基本的方法是将序列传递给NumPy的array()函数; 你可以传递任何序列（类数组），而不仅仅是常见的列表（list）数据类型。

请注意，当我们打印具有不同长度数字的数组时，它会自动将它们填充出来。这对于查看矩阵很有用。对数组进行索引就像列表或任何其他Python序列一样。 你也可以对它们进行切片，我不打算在这里切片一维数组，如果你想了解切片的更多信息，请查看这篇文章索引与切片 - NumPy 中文文档。

上面的数组示例是如何使用NumPy表示向量的，接下来我们将看看如何使用多维数组表示矩阵和更多的信息。

```
# MD Array,
a = np.array([[11, 12, 13, 14, 15],
              [16, 17, 18, 19, 20],
              [21, 22, 23, 24, 25],
              [26, 27, 28 ,29, 30],
              [31, 32, 33, 34, 35]])

print(a[2,4]) # >>>25
```

为了创建一个2D（二维）数组，我们传递一个列表的列表（或者是一个序列的序列）给array()函数。如果我们想要一个3D（三维）数组，我们就要传递一个列表的列表的列表，如果是一个4D（四维）数组，那就是列表的列表的列表的列表，以此类推。

请注意2D（二维）数组（在我们的朋友空格键的帮助下）是如何按行和列排列的。要索引2D（二维）数组，我们只需引用行数和列数即可。

它背后的一些数学知识

要正确理解这一点，我们应该真正了解一下矢量和矩阵是什么。

矢量是具有方向和幅度的量。它们通常用于表示速度，加速度和动量等事物。向量可以用多种方式编写，尽管对我们最有用的是它们被写为n元组的形式，如（1,4,6,9）。这就是我们在NumPy中表示他们的方式。

矩阵类似于矢量，除了它由行和列组成; 很像一个网格。可以通过给出它所在的行和列来引用矩阵中的值。在NumPy中，我们通过传递一系列序列来制作数组，就像我们之前所做的那样。

多维数组切片

切片多维数组比1D数组复杂一点，并且在使用NumPy时你也会经常需要使用到。

```
# MD slicing
print(a[0, 1:4]) # >>>[12 13 14]
print(a[1:4, 0]) # >>>[16 21 26]
print(a[::2,::2]) # >>>[[11 13 15]
                  #     [21 23 25]
                  #     [31 33 35]]
print(a[:, 1]) # >>>[12 17 22 27 32]
```

如你所见，通过对每个以逗号分隔的维度执行单独的切片，你可以对多维数组进行切片。因此，对于2D数组，我们的第一片定义了行的切片，第二片定义了列的切片。

注意，只需输入数字就可以指定行或列。上面的第一个示例从数组中选择第0列。

下面的图表说明了给定的示例切片是如何进行工作的。

数组属性

在使用 NumPy 时，你会想知道数组的某些信息。很幸运，在这个包里边包含了很多便捷的方法，可以给你想要的信息。

```
# Array properties
a = np.array([[11, 12, 13, 14, 15],
              [16, 17, 18, 19, 20],
              [21, 22, 23, 24, 25],
              [26, 27, 28 ,29, 30],
              [31, 32, 33, 34, 35]])

print(type(a)) # >>><class 'numpy.ndarray'>
print(a.dtype) # >>>int64
print(a.size) # >>>25
print(a.shape) # >>>(5, 5)
print(a.itemsize) # >>>8
```

```
print(a.ndim) # >>>2
print(a.nbytes) # >>>200
```

正如你在上面的代码中看到的，NumPy数组实际上被称为ndarray。我不知道为什么他妈的它叫ndarray，如果有人知道请留言！我猜它代表n维数组。

数组的形状是它有多少行和列，上面的数组有5行和5列，所以它的形状是(5，5)。

`itemsize`属性是每个项占用的字节数。这个数组的数据类型是int 64，一个int 64中有64位，一个字节中有8位，除以64除以8，你就可以得到它占用了多少字节，在本例中是8。

`ndim`属性是数组的维数。这个有2个。例如，向量只有1。

`nbytes`属性是数组中的所有数据消耗掉的字节数。你应该注意到，这并不计算数组的开销，因此数组占用的实际空间将稍微大一点。

使用数组

基本操作符

只是能够从数组中创建和检索元素和属性不能满足你的需求，你有时也需要对它们进行数学运算。 你完全可以使用四则运算符 +、- 、/ 来完成运算操作。

```
# Basic Operators
a = np.arange(25)
a = a.reshape((5, 5))

b = np.array([10, 62, 1, 14, 2, 56, 79, 2, 1, 45,
              4, 92, 5, 55, 63, 43, 35, 6, 53, 24,
              56, 3, 56, 44, 78])
b = b.reshape((5,5))

print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a ** 2)
print(a < b) print(a > b)

print(a.dot(b))
```

除了 dot() 之外，这些操作符都是对数组进行逐元素运算。比如 (a, b, c) + (d, e, f) 的结果就是 (a+d, b+e, c+f)。它将分别对每一个元素进行配对，然后对它们进行运算。它返回的结果是一个数组。注意，当使用逻辑运算符比如 “<” 和 “>” 的时候，返回的将是一个布尔型数组，这点有一个很好的用处，后边我们会提到。

dot() 函数计算两个数组的点积。它返回的是一个标量（只有大小没有方向的一个值）而不是数组。

它背后的一些数学知识

dot()函数称为点积。理解这一点的最好方法是看下图，下图将表示它是如何进行计算的。

数组特殊运算符

NumPy还提供了一些别的用于处理数组的好用的运算符。

```
# dot, sum, min, max, cumsum
a = np.arange(10)

print(a.sum()) # >>>45
print(a.min()) # >>>0
print(a.max()) # >>>9
print(a.cumsum()) # >>>[ 0  1  3  6 10 15 21 28 36 45]
```

sum()、min()和max()函数的作用非常明显。将所有元素相加，找出最小和最大元素。

然而，cumsum()函数就不那么明显了。它将像sum()这样的每个元素相加，但是它首先将第一个元素和第二个元素相加，并将计算结果存储在一个列表中，然后将该结果添加到第三个元素中，然后再将该结果存储在一个列表中。这将对数组中的所有元素执行此操作，并返回作为列表的数组之和的运行总数。

索引进阶

花式索引

`花式索引` 是获取数组中我们想要的特定元素的有效方法。

```
# Fancy indexing
a = np.arange(0, 100, 10)
indices = [1, 5, -1]
b = a[indices]
print(a) # >>>[ 0 10 20 30 40 50 60 70 80 90]
print(b) # >>>[10 50 90]
```

正如你在上面的示例中所看到的，我们使用我们想要检索的特定索引序列对数组进行索引。这反过来返回我们索引的元素的列表。

布尔屏蔽

布尔屏蔽是一个有用的功能，它允许我们根据我们指定的条件检索数组中的元素。

```
# Boolean masking
import matplotlib.pyplot as plt

a = np.linspace(0, 2 * np.pi, 50)
b = np.sin(a)
plt.plot(a,b)
mask = b >= 0
plt.plot(a[mask], b[mask], 'bo')
mask = (b >= 0) & (a <= np.pi / 2)
plt.plot(a[mask], b[mask], 'go')
plt.show()
```

上面的示例显示了如何进行布尔屏蔽。你所要做的就是将数组传递给涉及数组的条件，它将为你提供一个值的数组，为该条件返回true。

该示例生成以下图：

我们利用这些条件来选择图上的不同点。蓝色点(在图中还包括绿点，但绿点掩盖了蓝色点)，显示值大于0的所有点。绿色点表示值大于0且小于一半π的所有点。

缺省索引

不完全索引是从多维数组的第一个维度获取索引或切片的一种方便方法。例如，如果数组a=[1，2，3，4，5]，[6，7，8，9，10]，那么[3]将在数组的第一个维度中给出索引为3的元素，这里是值4。

```
# Incomplete Indexing
a = np.arange(0, 100, 10)
b = a[:5]
c = a[a >= 50]
print(b) # >>>[ 0 10 20 30 40]
print(c) # >>>[50 60 70 80 90]
```

Where 函数

where() 函数是另外一个根据条件返回数组中的值的有效方法。只需要把条件传递给它，它就会返回一个使得条件为真的元素的列表。

```
# Where
a = np.arange(0, 100, 10)
b = np.where(a < 50)
c = np.where(a >= 50)[0]
print(b) # >>>(array([0, 1, 2, 3, 4]),)
print(c) # >>>[5 6 7 8 9]
```

这就是NumPy，没那么难吧？当然，这篇文章只涵盖了入门的基础知识，在NumPy中你还可以做许多其他好玩的事情，当你已经熟悉了NumPy的基础知识，你就可以开始自由的探索NumPy的世界了。

最后别忘了分享这篇文章噢

要记得，不要忘了分享这篇文章，这样其他人就也可以看到了！另外，记得订阅这个博客的邮件列表，关注我的Twitter和Google+，这样你就不会错过任何有价值的文章了！

我会阅读所有的评论，如果你有什么想说的，想分享的，想问的或者喜欢的，请在下边留言！

文章出处

由NumPy中文文档翻译，原作者为 Ravikiran Srinivasulu，翻译至：
https://www.pluralsight.com/guides/different-ways-create-numpy-arrays

理解 NumPy

在这篇文章中，我们将介绍使用NumPy的基础知识，NumPy是一个功能强大的Python库，允许更高级的数据操作和数学计算。

什么是 NumPy?

NumPy是一个功能强大的Python库，主要用于对多维数组执行计算。NumPy这个词来源于两个单词--`Numerical`和`Python`。NumPy提供了大量的库函数和操作，可以帮助程序员轻松地进行数值计算。这类数值计算广泛用于以下任务：

- **机器学习模型**：在编写机器学习算法时，需要对矩阵进行各种数值计算。例如矩阵乘法、换位、加法等。NumPy提供了一个非常好的库，用于简单(在编写代码方面)和快速(在速度方面)计算。NumPy数组用于存储训练数据和机器学习模型的参数。
- **图像处理和计算机图形学**：计算机中的图像表示为多维数字数组。NumPy成为同样情况下最自然的选择。实际上，NumPy提供了一些优秀的库函数来快速处理图像。例如，镜像图像、按特定角度旋转图像等。
- **数学任务**：NumPy对于执行各种数学任务非常有用，如数值积分、微分、内插、外推等。因此，当涉及到数学任务时，它形成了一种基于Python的MATLAB的快速替代。

NumPy 的安装

在你的计算机上安装NumPy的最快也是最简单的方法是在shell上使用以下命令：`pip install numpy`。

这将在你的计算机上安装最新/最稳定的NumPy版本。通过PIP安装是安装任何Python软件包的最简单方法。现在让我们来谈谈NumPy中最重要的概念，NumPy数组。

NumPy 中的数组

NumPy提供的最重要的数据结构是一个称为NumPy数组的强大对象。NumPy数组是通常的Python数组的扩展。NumPy数组配备了大量的函数和运算符，可以帮助我们快速编写上面讨论过的各种类型计算的高性能代码。让我们看看如何快速定义一维NumPy数组：

```
import numpy as np
my_array = np.array([1, 2, 3, 4, 5])
print my_array
```

在上面的简单示例中，我们首先使用import numpy作为np导入NumPy库。然后，我们创建了一个包含5个整数的简单NumPy数组，然后我们将其打印出来。继续在自己的机器上试一试。在看 “NumPy安装” 部分下面的步骤的时候，请确保已在计算机中安装了NumPy。

现在让我们看看我们可以用这个特定的NumPy数组能做些什么。

```
print my_array.shape
```

它会打印我们创建的数组的形状：`(5, )`。意思就是 my_array 是一个包含5个元素的数组。

我们也可以打印各个元素。就像普通的Python数组一样，NumPy数组的起始索引编号为0。

```
print my_array[0]
print my_array[1]
```

上述命令将分别在终端上打印1和2。我们还可以修改NumPy数组的元素。例如，假设我们编写以下2个命令：

```
my_array[0] = -1
print my_array
```

我们将在屏幕上看到：`[-1,2,3,4,5]`。

现在假设，我们要创建一个长度为5的NumPy数组，但所有元素都为0，我们可以这样做吗？是的。NumPy提供了一种简单的方法来做同样的事情。

```
my_new_array = np.zeros((5))
print my_new_array
```

我们将看到输出了`[0., 0., 0., 0., 0.]`。与`np.zeros`类似，我们也有`np.ones`。 如果我们想创建一个随机值数组怎么办？

```
my_random_array = np.random.random((5))
print my_random_array
```

我们得到的输出看起来像 [0.22051844 0.35278286 0.11342404 0.79671772 0.62263151] 这样的数据。你获得的输出可能会有所不同，因为我们使用的是随机函数，它为每个元素分配0到1之间的随机值。

现在让我们看看如何使用NumPy创建二维数组。

```
my_2d_array = np.zeros((2, 3)) print my_2d_array
```

这将在屏幕上打印以下内容：

```
[[0. 0. 0.]

[0. 0. 0.]]
```

猜猜以下代码的输出结果如何：

```
my_2d_array_new = np.ones((2, 4)) print my_2d_array_new
```

这里是：

```
[[1. 1. 1. 1.]

[1. 1. 1. 1.]]
```

基本上，当你使用函数`np.zeros()`或`np.ones()`时，你可以指定讨论数组大小的元组。在上面的两个例子中，我们使用以下元组，(2, 3) 和(2, 4) 分别表示2行，3列和4列。像上面那样的多维数组可以用`my_array[i][j]`符号来索引，其中i表示行号，j表示列号。i和j都从0开始。

```
my_array = np.array([[4, 5], [6, 1]])
print my_array[0][1]
```

上面的代码片段的输出是5，因为它是索引0行和索引1列中的元素。

你还可以按如下方式打印my_array的形状：

```
print my_array.shape
```

输出为(2, 2)，表示数组中有2行2列。

NumPy提供了一种提取多维数组的行/列的强大方法。例如，考虑我们上面定义的`my_array`的例子。

```
[[4 5] [6 1]]
```

假设，我们想从中提取第二列（索引1）的所有元素。在这里，我们肉眼可以看出，第二列由两个元素组成：`5`和`1`。为此，我们可以执行以下操作：

```
my_array_column_2 = my_array[:, 1]
print my_array_column_2
```

注意，我们使用了冒号(`:`)而不是行号，而对于列号，我们使用了值`1`，最终输出是：`[5, 1]`。

我们可以类似地从多维NumPy数组中提取一行。现在，让我们看看NumPy在多个数组上执行计算时提供的强大功能。

NumPy中的数组操作

使用NumPy，你可以轻松地在数组上执行数学运算。例如，你可以添加NumPy数组，你可以减去它们，你可以将它们相乘，甚至可以将它们分开。 以下是一些例子：

```
import numpy as np
a = np.array([[1.0, 2.0], [3.0, 4.0]])
b = np.array([[5.0, 6.0], [7.0, 8.0]])
sum = a + b
difference = a - b
product = a * b
quotient = a / b
print "Sum = \n", sum
print "Difference = \n", difference
print "Product = \n", product
print "Quotient = \n", quotient

# The output will be as follows:

Sum = [[ 6.  8.] [10. 12.]]
Difference = [[-4. -4.] [-4. -4.]]
Product = [[ 5. 12.] [21. 32.]]
Quotient = [[0.2 0.33333333] [0.42857143 0.5 ]]
```

如你所见，乘法运算符执行逐元素乘法而不是矩阵乘法。 要执行矩阵乘法，你可以执行以下操作：

```
matrix_product = a.dot(b)
print "Matrix Product = ", matrix_product
```

输出将是：

```
[[19. 22.]

[43. 50.]]
```

总结

如你所见，NumPy在其提供的库函数方面非常强大。你可以使用NumPy公开的优秀的API在单行代码中执行大型计算。这使它成为各种数值计算的优雅工具。如果你希望自己成为一名数学家或数据科学家，你一定要考虑掌握它。在熟练掌握NumPy之前，你需要了解Python。

你可以在 Hackr.io 上找到编程社区推荐的最佳Python 教程，愿上帝保佑你！

文章出处

由NumPy中文文档翻译，原作者为 Vijay Singh，翻译至：https://dzone.com/article/understanding-numpy。

Python、Numpy 教程

我们将在本课程的所有作业中使用Python编程语言。Python本身就是一种伟大的通用编程语言，并且它在一些其他流行的Python库(numpy、sciy、matplotlib)的帮助下，它成为了一个强大的科学计算环境。

我们希望你们中大部分人会有一点Python和numpy的使用经验；因为对于大部分人来说，本节将作为关于Python编程语言和使用Python进行科学计算的快速速成课程。

你们中的一些人可能以前学过Matlab接触过相关的知识，如果是这样的话，我推荐你们看一下这篇文章：Numpy对于Matlab用户。

你还可以在 这里找到 由 Volodymyr Kuleshov 和 Isaac Caswell 为 CS 228 创建的本教程的IPython笔记版本。

目录:

Python

Python是一种高级动态类型的多范式编程语言。Python代码通常被称为可运行的伪代码，因为它允许你在非常少的代码行中表达非常强大的想法，同时具有非常可读性。作为示例，这里是Python中经典快速排序算法的实现：

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
# Prints "[1, 1, 2, 3, 6, 8, 10]"
```

Python 的版本

目前有两种不同的受支持版本的Python，分别是2.7和3.5。有点令人困惑的是，Python 3.0引入了许多向后兼容的语言更改，因此为2.7编写的代码可能无法在3.5下运行，反之亦然。所以我们下面所有的示例的代码都使用Python 3.5来编程。

你可以通过运行 `python -version` 在命令行中查看Python的版本。

基本数据类型

与大多数语言一样，Python有许多基本类型，包括整数，浮点数，布尔值和字符串。这些数据类型的行为方式与其他编程语言相似。

Numbers(数字类型)：代表的是整数和浮点数，它原理与其他语言相同：

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

注意，与许多语言不同，Python没有一元增量(`x+`)或递减(`x-`)运算符。

Python还有用于复数的内置类型；你可以在这篇文档中找到所有的详细信息。

Booleans(布尔类型): Python实现了所有常用的布尔逻辑运算符，但它使用的是英文单词而不是符号(`&&`, `||`, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

Strings(字符串类型)：Python对字符串有很好的支持：

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 12"
```

String对象有许多有用的方法；例如：

```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "
hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with
another;
                                # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints
"world"
```

你可以在这篇文档中找到所有String方法的列表。

容器(Containers)

Python包含几种内置的容器类型：列表、字典、集合和元组。

列表(Lists)

列表其实就是Python中的数组，但是可以它可以动态的调整大小并且可以包含不同类型的元素：

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

像往常一样，你可以在这篇文档中找到有关列表的所有详细信息。

切片(Slicing): 除了一次访问一个列表元素之外，Python还提供了访问子列表的简明语法; 这被称为切片：

```
nums = list(range(5))     # range is a built-in function that creates a list of
integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "
[2, 3]"
print(nums[2:])           # Get a slice from index 2 to the end; prints "[2, 3,
4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive);
prints "[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3,
4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

我们将在numpy数组的上下文中再次看到切片。

(循环)Loops: 你可以循环遍历列表的元素，如下所示：

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

如果要访问循环体内每个元素的索引，请使用内置的 `enumerate` 函数：

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

列表推导式(List comprehensions): 编程时，我们经常想要将一种数据转换为另一种数据。 举个简单的例子，思考以下计算平方数的代码：

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

你可以使用 **列表推导式** 使这段代码更简单:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

列表推导还可以包含条件：

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```

字典

字典存储（键，值）对，类似于Java中的`Map`或Javascript中的对象。你可以像这样使用它：

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

你可以在这篇文档中找到有关字典的所有信息。

(循环)Loops: 迭代词典中的键很容易：

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

如果要访问键及其对应的值，请使用`items`方法：

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

字典推导式(Dictionary comprehensions): 类似于列表推导式，可以让你轻松构建词典数据类型。例如：

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

集合(Sets)

集合是不同元素的无序集合。举个简单的例子，请思考下面的代码：

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an element that is already in the set does
nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

与往常一样，你想知道的关于集合的所有内容都可以在这篇文档中找到。

循环(Loops): 遍历集合的语法与遍历列表的语法相同；但是，由于集合是无序的，因此不能假设访问集合元素的顺序：

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

集合推导式(Set comprehensions): 就像列表和字典一样，我们可以很容易地使用集合理解来构造集合：

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

元组(Tuples)

元组是（不可变的）有序值列表。 元组在很多方面类似于列表; 其中一个最重要的区别是元组可以用作字典中的键和集合的元素，而列表则不能。 这是一个简单的例子：

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

这篇文档包含有关元组的更多信息。

函数(Functions)

Python函数使用`def`关键字定义。例如：

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

我们经常定义函数来获取可选的关键字参数，如下所示：

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

这篇文档中有更多关于Python函数的信息。

类(Classes)

在Python中定义类的语法很简单：

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

你可以在这篇文档中阅读更多关于Python类的内容。

Numpy

Numpy是Python中科学计算的核心库。它提供了一个高性能的多维数组对象，以及用于处理这些数组的工具。如果你已经熟悉MATLAB，你可能会发现这篇教程对于你从MATLAB切换到学习Numpy很有帮助。

数组(Arrays)

numpy数组是一个值网格，所有类型都相同，并由非负整数元组索引。 维数是数组的排名; 数组的形状是一个整数元组，给出了每个维度的数组大小。

我们可以从嵌套的Python列表初始化numpy数组，并使用方括号访问元素：

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy还提供了许多创建数组的函数：

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

你可以在这篇文档中阅读有关其他数组创建方法的信息。

数组索引

Numpy提供了几种索引数组的方法。

切片(Slicing): 与Python列表类似，可以对numpy数组进行切片。由于数组可能是多维的，因此必须为数组的每个维指定一个切片：

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
```

```
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

你还可以将整数索引与切片索引混合使用。 但是，这样做会产生比原始数组更低级别的数组。 请注意，这与MATLAB处理数组切片的方式完全不同：

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

整数数组索引: 使用切片索引到numpy数组时，生成的数组视图将始终是原始数组的子数组。 相反，整数数组索引允许你使用另一个数组中的数据构造任意数组。 这是一个例子：

```
import numpy as np

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

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
```

```
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

整数数组索引的一个有用技巧是从矩阵的每一行中选择或改变一个元素：

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

布尔数组索引: 布尔数组索引允许你选择数组的任意元素。通常，这种类型的索引用于选择满足某些条件的数组元素。下面是一个例子：

```
import numpy as np

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

为简洁起见，我们省略了很多关于numpy数组索引的细节; 如果你想了解更多，你应该阅读这篇文档。

数据类型

每个numpy数组都是相同类型元素的网格。Numpy提供了一组可用于构造数组的大量数值数据类型。Numpy在创建数组时尝试猜测数据类型，但构造数组的函数通常还包含一个可选参数来显式指定数据类型。这是一个例子：

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

你可以在这篇文档中阅读有关numpy数据类型的所有信息。

数组中的数学

基本数学函数在数组上以元素方式运行，既可以作为运算符重载，也可以作为numpy模块中的函数：

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

请注意，与MATLAB不同，`*`是元素乘法，而不是矩阵乘法。 我们使用`dot`函数来计算向量的内积，将向量乘以矩阵，并乘以矩阵。 `dot`既可以作为numpy模块中的函数，也可以作为数组对象的实例方法：

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy为在数组上执行计算提供了许多有用的函数；其中最有用的函数之一是`SUM`：

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

你可以在这篇文档中找到numpy提供的数学函数的完整列表。

除了使用数组计算数学函数外，我们经常需要对数组中的数据进行整形或其他操作。这种操作的最简单的例子是转置一个矩阵；要转置一个矩阵，只需使用一个数组对象的`T`属性：

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

Numpy提供了许多用于操作数组的函数；你可以在这篇文档中看到完整的列表。

广播(Broadcasting)

广播是一种强大的机制，它允许numpy在执行算术运算时使用不同形状的数组。通常，我们有一个较小的数组和一个较大的数组，我们希望多次使用较小的数组来对较大的数组执行一些操作。

例如，假设我们要向矩阵的每一行添加一个常数向量。我们可以这样做：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

这会凑效; 但是当矩阵 `x` 非常大时，在Python中计算显式循环可能会很慢。注意，向矩阵 `x` 的每一行添加向量 `v` 等同于通过垂直堆叠多个 `v` 副本来形成矩阵 `vv`，然后执行元素的求和 `x` 和 `vv`。 我们可以像如下这样实现这种方法：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Numpy广播允许我们在不实际创建`v`的多个副本的情况下执行此计算。考虑这个需求，使用广播如下：

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

`y=x+v` 行即使 `x` 具有形状 `(4, 3)` 和 `v` 具有形状 `(3,)`，但由于广播的关系，该行的工作方式就好像 `v` 实际上具有形状 `(4, 3)`，其中每一行都是 `v` 的副本，并且求和是按元素执行的。

将两个数组一起广播遵循以下规则：

1. 如果数组不具有相同的rank，则将较低等级数组的形状添加1，直到两个形状具有相同的长度。
2. 如果两个数组在维度上具有相同的大小，或者如果其中一个数组在该维度中的大小为1，则称这两个数组在维度上是兼容的。
3. 如果数组在所有维度上兼容，则可以一起广播。
4. 广播之后，每个数组的行为就好像它的形状等于两个输入数组的形状的元素最大值。
5. 在一个数组的大小为1且另一个数组的大小大于1的任何维度中，第一个数组的行为就像沿着该维度复制一样

如果对于以上的解释依然没有理解，请尝试阅读这篇文档或这篇解释中的说明。

支持广播的功能称为通用功能。你可以在这篇文档中找到所有通用功能的列表。

以下是广播的一些应用：

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
```

```
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

广播通常会使你的代码更简洁，效率更高，因此你应该尽可能地使用它。

Numpy 的文档

这个简短的概述说明了部分numpy相关的重要事项。查看numpy参考手册以了解有关numpy的更多信息。

SciPy

Numpy提供了一个高性能的多维数组和基本工具来计算和操作这些数组。 而SciPy以此为基础，提供了大量在numpy数组上运行的函数，可用于不同类型的科学和工程应用程序。

熟悉SciPy的最佳方法是浏览它的文档。我们将重点介绍SciPy有关的对你有价值的部分内容。

图像操作

SciPy提供了一些处理图像的基本函数。例如，它具有将映像从磁盘读入numpy数组、将numpy数组作为映像写入磁盘以及调整映像大小的功能。下面是一个演示这些函数的简单示例：

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

左：原始图像。右：着色和调整大小的图像。

MATLAB 文件

函数 `scipy.io.loadmat` 和 `scipy.io.savemat` 允许你读取和写入MATLAB文件。你可以在这篇文档中学习相关操作。

点之间的距离

SciPy定义了一些用于计算点集之间距离的有用函数。

函数`scipy.spatial.distance.pdist`计算给定集合中所有点对之间的距离：

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

你可以在这篇文档中阅读有关此功能的所有详细信息。

类似的函数（`scipy.spatial.distance.cdist`）计算两组点之间所有对之间的距离; 你可以在这篇文档中阅读它。

Matplotlib

Matplotlib是一个绘图库。本节简要介绍 `matplotlib.pyplot` 模块，该模块提供了类似于MATLAB的绘图系统。

绘制

matplotlib中最重要的功能是`plot`，它允许你绘制2D数据的图像。这是一个简单的例子：

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

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

运行此代码会生成以下图表：

通过一些额外的工作，我们可以轻松地一次绘制多条线，并添加标题，图例和轴标签：

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

你可以在这篇文档中阅读有关 `绘图` 功能的更多信息。

子图

你可以使用 `subplot` 函数在同一个图中绘制不同的东西。 这是一个例子：

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
```

```
# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

你可以在这篇文档中阅读有关 `子图` 功能的更多信息。

图片

你可以使用 `imshow` 函数来显示一张图片。 这是一个例子：

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```

文章出处

由NumPy中文文档翻译，原作者为 Justin Johnson，翻译至：http://cs231n.github.io/python-numpy-tutorial/。

创建 Numpy 数组的不同方式

Numpy库的核心是数组对象或ndarray对象（n维数组）。你将使用Numpy数组执行逻辑，统计和傅里叶变换等运算。作为使用Numpy的一部分，你要做的第一件事就是创建Numpy数组。本指南的主要目的是帮助数据科学爱好者了解可用于创建Numpy数组的不同方式。

创建Numpy数组有三种不同的方法：

1. 使用Numpy内部功能函数
2. 从列表等其他Python的结构进行转换
3. 使用特殊的库函数

使用Numpy内部功能函数

Numpy具有用于创建数组的内置函数。 我们将在本指南中介绍其中一些内容。

创建一个一维的数组

首先，让我们创建一维数组或rank为1的数组。`arange`是一种广泛使用的函数，用于快速创建数组。将值20传递给`arange`函数会创建一个值范围为0到19的数组。

```
import Numpy as np
array = np.arange(20)
array
```

输出：

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

要验证此数组的维度，请使用shape属性。

```
array.shape
```

输出：

```
(20,)
```

由于逗号后面没有值，因此这是一维数组。 要访问此数组中的值，请指定非负索引。 与其他编程语言一样，索引从零开始。 因此，要访问数组中的第四个元素，请使用索引3。

```
array[3]
```

输出：

```
3
```

Numpy的数组是可变的，这意味着你可以在初始化数组后更改数组中元素的值。 使用print函数查看数组的内容。

```
array[3] = 100
print(array)
```

输出：

```
[  0   1   2 100
   4   5   6   7
   8   9  10  11
   12  13  14  15
   16  17  18  19]
```

与Python列表不同，Numpy数组的内容是同质的。 因此，如果你尝试将字符串值分配给数组中的元素，其数据类型为int，则会出现错误。

```
array[3] ='Numpy'
```

输出：

```
ValueError: invalid literal for int() with base 10: 'Numpy'
```

创建一个二维数组

我们来谈谈创建一个二维数组。 如果只使用arange函数，它将输出一维数组。 要使其成为二维数组，请使用reshape函数链接其输出。

```
array = np.arange(20).reshape(4,5)
array
```

输出：

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

首先，将创建20个整数，然后将数组转换为具有4行和5列的二维数组。 我们来检查一下这个数组的维数。

```
(4, 5)
```

由于我们得到两个值，这是一个二维数组。 要访问二维数组中的元素，需要为行和列指定索引。

```
array[3][4]
```

输出：

```
19
```

创建三维数组及更多维度

要创建三维数组，请为重塑形状函数指定3个参数。

```
array = np.arange(27).reshape(3,3,3)
array
```

输出：

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

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

需要注意的是：数组中元素的数量（27）必须是其尺寸（3 * 3 * 3）的乘积。 要交叉检查它是否是三维数组，可以使用shape属性。

```
array.shape
```

输出：

```
(3, 3, 3)
```

此外，使用`arange`函数，你可以创建一个在定义的起始值和结束值之间具有特定序列的数组。

```
np.arange(10, 35, 3)
```

输出：

```
array([10, 13, 16, 19, 22, 25, 28, 31, 34])
```

使用其他Numpy函数

除了arange函数之外，你还可以使用其他有用的函数（如`zeros`和`ones`）来快速创建和填充数组。

使用`zeros`函数创建一个填充零的数组。函数的参数表示行数和列数（或其维数）。

```
np.zeros((2,4))
```

输出：

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

使用`ones`函数创建一个填充了1的数组。

```
np.ones((3,4))
```

输出：

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

`empty`函数创建一个数组。它的初始内容是随机的，取决于内存的状态。

```
np.empty((2,3))
```

输出：

```
array([[0.65670626, 0.52097334, 0.99831087],
       [0.07280136, 0.4416958 , 0.06185705]])
```

`full`函数创建一个填充给定值的n * n数组。

```
np.full((2,2), 3)
```

输出：

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

`eye`函数可以创建一个n * n矩阵，对角线为1s，其他为0。

```
np.eye(3,3)
```

输出：

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

函数`linspace`在指定的时间间隔内返回均匀间隔的数字。 例如，下面的函数返回0到10之间的四个等间距数字。

```
np.linspace(0, 10, num=4)
```

输出：

```
array([ 0., 3.33333333, 6.66666667, 10.])
```

从Python列表转换

除了使用Numpy函数之外，你还可以直接从Python列表创建数组。将Python列表传递给数组函数以创建Numpy数组：

```
array = np.array([4,5,6])
array
```

输出：

```
array([4, 5, 6])
```

你还可以创建Python列表并传递其变量名以创建Numpy数组。

```
list = [4,5,6]
list
```

输出：

```
[4, 5, 6]
```

```
array = np.array(list)
array
```

输出：

```
array([4, 5, 6])
```

你可以确认变量`array`和`list`分别是Python列表和Numpy数组。

```
type(list)
```

> list

```
type(array)
```

> Numpy.ndarray

要创建二维数组，请将一系列列表传递给数组函数。

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

输出：

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

```
array.shape
```

输出：

```
(2, 3)
```

使用特殊的库函数

你还可以使用特殊库函数来创建数组。例如，要创建一个填充0到1之间随机值的数组，请使用`random`函数。这对于需要随机状态才能开始的问题特别有用。

```
np.random.random((2,2))
```

输出：

```
array([[0.1632794 , 0.34567049],
       [0.03463241, 0.70687903]])
```

总结

创建和填充Numpy数组是使用Numpy执行快速数值数组计算的第一步。使用不同的方式创建数组，你现在可以很好地执行基本的数组操作。

文章出处

由NumPy中文文档翻译，原作者为 Ravikiran Srinivasulu，翻译至：
https://www.pluralsight.com/guides/different-ways-create-numpy-arrays

NumPy 中的矩阵和向量

numpy的`ndarray`类用于表示矩阵和向量。
要在numpy中构造矩阵，我们在列表中列出矩阵的行，
并将该列表传递给numpy数组构造函数。

例如，构造与矩阵对应的numpy数组

我们会这样做

```
A = np.array([[1,-1,2],[3,2,0]])
```

向量只是具有单列的数组。 例如，构建向量

我们会这样做

```
v = np.array([[2],[1],[3]])
```

更方便的方法是转置相应的行向量。 例如，为了使上面的矢量，我们可以改为转置行向量

这个代码是

```
v = np.transpose(np.array([[2,1,3]]))
```

numpy重载数组索引和切片符号以访问矩阵的各个部分。 例如，要打印矩阵A中的右下方条目，我们会这样做

```
print(A[1,2])
```

要切出A矩阵中的第二列，我们会这样做

```
col = A[:,1:2]
```

第一个切片选择A中的所有行，而第二个切片仅选择每行中的中间条目。

要进行矩阵乘法或矩阵向量乘法，我们使用np.dot()方法。

```
w = np.dot(A,v)
```

用numpy求解方程组

线性代数中比较常见的问题之一是求解矩阵向量方程。 这是一个例子。 我们寻找解决方程的向量x

$A\mathbf{x} = \mathbf{b}$

当

我们首先构建A和b的数组。

```
A = np.array([[2,1,-2],[3,0,1],[1,1,-1]])
b = np.transpose(np.array([[-3,5,-2]]))
```

为了解决这个系统

```
x = np.linalg.solve(A,b)
```

应用：多元线性回归

在多元回归问题中，我们寻找一种能够将输入数据点映射到结果值的函数。 每个数据点是*特征向量（x1，x2，...，xm）*，由两个或多个捕获输入的各种特征的数据值组成。为了表示所有输入数据以及输出值的向量，我们设置了输入矩阵X和输出向量 **y**：

在简单的最小二乘线性回归模型中，我们寻找向量**β**，使得乘积Xβ最接近结果向量 **y**。

一旦我们构建了**β**向量，我们就可以使用它将输入数据映射到预测结果。 给定表单中的输入向量

我们可以计算预测结果值

计算β向量的公式是

$\boldsymbol{\beta} = (X^T X)^{-1} X^T \mathbf{y}$

在我们的下一个示例程序中，我将使用numpy构造适当的矩阵和向量并求解**β**向量。一旦我们解决了**β**，我们将使用它来预测我们最初从输入数据集中遗漏的一些测试数据点。

假设我们在numpy中构造了输入矩阵X和结果向量**y**，下面的代码将计算**β**向量：

```
Xt = np.transpose(X)
XtX = np.dot(Xt,X)
Xty = np.dot(Xt,y)
beta = np.linalg.solve(XtX,Xty)
```

最后一行使用`np.linalg.solve`计算**β**，因为等式是：

$\boldsymbol{\beta} = (X^T X)^{-1} X^T \mathbf{y}$

在数学上等价于方程组：

$(X^T X)\ \boldsymbol{\beta} = X^T \mathbf{y}$

我将用于此示例的数据集是Windsor房价数据集，其中包含有关安大略省温莎市区房屋销售的信息。 输入变量涵盖了可能对房价产生影响的一系列因素，例如批量大小，卧室数量以及各种设施的存在。此处提供具有完整数据集的CSV文件。我从这个网站下载了数据集，该网站提供了大量涵盖大量主题的数据集。

这里现在是示例程序的源代码。

```
import csv
import numpy as np

def readData():
    X = []
    y = []
    with open('Housing.csv') as f:
        rdr = csv.reader(f)
        # Skip the header row
        next(rdr)
        # Read X and y
        for line in rdr:
            xline = [1.0]
            for s in line[:-1]:
                xline.append(float(s))
            X.append(xline)
            y.append(float(line[-1]))
    return (X,y)

X0,y0 = readData()
# Convert all but the last 10 rows of the raw data to numpy arrays
d = len(X0)-10
X = np.array(X0[:d])
y = np.transpose(np.array([y0[:d]]))

# Compute beta
Xt = np.transpose(X)
XtX = np.dot(Xt,X)
Xty = np.dot(Xt,y)
beta = np.linalg.solve(XtX,Xty)
print(beta)

# Make predictions for the last 10 rows in the data set
for data,actual in zip(X0[d:],y0[d:]):
    x = np.array([data])
    prediction = np.dot(x,beta)
    print('prediction = '+str(prediction[0,0])+' actual = '+str(actual))
```

原始数据集包含500多个条目 为了测试线性回归模型所做预测的准确性，我们使用除最后10个数据条目之外的所有数据条目来构建回归模型并计算**β**。一旦我们构建了**β**向量，我们就用它来预测最后10个输入值，然后将预测的房价与数据集中的实际房价进行比较。

以下是该计划产生的产出：

```
[[ -4.14106096e+03]
 [  3.55197583e+00]
 [  1.66328263e+03]
 [  1.45465644e+04]
 [  6.77755381e+03]
 [  6.58750520e+03]
 [  4.44683380e+03]
 [  5.60834856e+03]
 [  1.27979572e+04]
 [  1.24091640e+04]
 [  4.19931185e+03]
```

```
 [  9.42215457e+03]]
prediction = 97360.6550969 actual = 82500.0
prediction = 71774.1659014 actual = 83000.0
prediction = 92359.0891976 actual = 84000.0
prediction = 77748.2742379 actual = 85000.0
prediction = 91015.5903066 actual = 85000.0
prediction = 97545.1179047 actual = 91500.0
prediction = 97360.6550969 actual = 94000.0
prediction = 106006.800756 actual = 103000.0
prediction = 92451.6931269 actual = 105000.0
prediction = 73458.2949381 actual = 105000.0
```

总体而言，预测并不是非常好，但是一些预测有点接近正确。从这些数据中做出更好的预测将成为机器学习冬季学期教程的主题。

文章出处

由NumPy中文文档翻译，原作者为 劳伦斯大学，翻译至：http://www2.lawrence.edu/fast/GREGGJ/Python/numpy/numpyLA.html。

与 Matlab 比较

介绍

MATLAB®和NumPy / SciPy有很多共同之处。但是有很多不同之处。创建NumPy和SciPy是为了用Python最自然的方式进行数值和科学计算，而不是MATLAB®克隆。本页面旨在收集有关差异的智慧，主要是为了帮助熟练的MATLAB®用户成为熟练的NumPy和SciPy用户。

一些关键的差异

MATLAB	NumPy
在MATLAB®中，基本数据类型是双精度浮点数的多维数组。大多数表达式采用这样的数组并返回这样的数 对这些数组的2-D实例的操作被设计成或多或少地像线性代数中的矩阵运算。	在NumPy中，基本类型是多维的array。包括2D在内的所有维度中对这些数组的操作是逐元素操作。人们需要使用线性代数的特定函数（尽管对于矩阵乘法，可以@在python 3.5及更高版本中使用运算符）。
MATLAB®使用基于1（一）的索引。使用（1）找到序列的初始元素。 请参阅备注	Python使用基于0（零）的索引。使用[0]找到序列的初始元素。
MATLAB®的脚本语言是为执行线性代数而创建的。基本矩阵操作的语法很好而且干净，但是用于添加GUI和制作完整应用程序的API或多或少都是事后的想法。	NumPy基于Python，它从一开始就被设计成一种优秀的通用编程语言。虽然Matlab的一些数组操作的语法比NumPy更紧凑，但NumPy（由于是Python的附加组件）可以做许多Matlab不能做的事情，例如正确处理矩阵堆栈。
在MATLAB®中，数组具有按值传递的语义，并具有惰性写入时复制方案，以防止在实际需要之前实际创建副本。切片操作复制数组的一部分。	在NumPy数组中有传递引用语义。切片操作是对数组的视图。

'array'或'matrix'？我应该使用哪个？

从历史上看，NumPy提供了一种特殊的矩阵类型 *np.matrix* ，它是ndarray的子类，它使二进制运算成为线性代数运算。您可能会在某些现有代码中看到它而不是 *np.array* 。那么，使用哪一个？

简答

使用数组。

- 它们是numpy的标准矢量/矩阵/张量类型。许多numpy函数返回数组，而不是矩阵。
- 元素操作和线性代数操作之间有明显的区别。
- 如果您愿意，可以使用标准向量或行/列向量。

在Python 3.5之前，使用数组类型的唯一缺点是你必须使用 `dot` 而不是 `*` 乘法（减少）两个张量（标量乘积，矩阵向量乘法等）。从Python 3.5开始，您可以使用矩阵乘法 `@` 运算符。

鉴于上述情况，我们打算 `matrix` 最终弃用。

长答案

NumPy包含`array`类和`matrix`类。所述
`array`类旨在是对许多种数值计算的通用n维数组中，而`matrix`意在具体促进线性代数计算。在实践中，两者之间只有少数关键差异。

- 运算符`*`和`@`函数`dot()`，以及`multiply()`：
  - 对于`数组`，`*`表示逐元素乘法，而`@`表示矩阵乘法; 它们具有相关的函数`multiply()`和`dot()`。（在python 3.5之前，`@`不存在，并且必须使用`dot()`进行矩阵乘法）。
  - 对于`矩阵`，`*`表示矩阵乘法，对于逐元素乘法，必须使用`multiply()`函数。
- 矢量处理（一维数组）
  - 对于`数组`，向量形状1xN，Nx1和N都是不同的东西。像`A[:, 1]`这样的操作返回形状N的一维数组，而不是形状Nx1的二维数组。在一维数组上转置什么都不做。
  - 对于`矩阵`，一维数组总是被上变频为1xN或Nx1矩阵（行或列向量）。`A[:, 1]`返回形状为Nx1的二维矩阵。
- 处理更高维数组（ndim> 2）
  - `数组`对象**的维数可以 > 2**；
  - `矩阵`对象**总是有两个维度**。
- 便利属性
  - `array` **有一个.T属性**，它返回数据的转置。
  - `matrix` **还有.H，.I和.A属性**，分别返回共轭转置，反转和`asarray()`矩阵。
- 便利构造函数
  - 该`array`构造**采用（嵌套）的Python序列初始化**。如：`array([[1,2,3],[4,5,6]])`。
  - 该`matrix`构造还**需要一个方便的字符串初始化**。如：`matrix("[1 2 3; 4 5 6]")`。

使用两者有利有弊：

- `array`
  - `:)` 元素乘法很容易：`A*B`。
  - `:(` 你必须记住，矩阵乘法有自己的运算符`@`。
  - `:)` 可以将一维数组视为行向量或列向量。`A @ v`将`v`视为列向量，而`v @ A`将`v`视为行向量。这可以节省您键入许多转置。
  - `:)` `array`是“默认”NumPy类型，因此它获得的测试最多，并且是使用NumPy的第三方代码最有可能返回的类型。
  - `:)` 非常擅长处理任何维度的数据。
  - `:)` 如果你熟悉那么语义学更接近张量代数。
  - `:)` *所有*操作（`*`，`/`，`+`，`-`等）逐元素。
  - `:(` 稀疏矩阵`scipy.sparse`不与数组交互。
- `matrix`
  - `:\\` 行为更像MATLAB®矩阵。
  - `<:(` 最大二维。要保存您需要的三维数据，`array`或者可能是Python列表`matrix`。
  - `<:(` 最小二维。你不能有载体。它们必须作为单列或单行矩阵进行转换。
  - `<:(` 由于`array`是NumPy中的默认值，因此`array`即使您将它们`matrix`作为参数给出，某些函数也可能返回。这不应该发生在NumPy函数中（如果它确实是一个错误），但基于NumPy的第三方代码可能不像NumPy那样遵守类型保存。
  - `:)` `A*B`是矩阵乘法，所以它看起来就像你在线性代数中写的那样（对于Python> = 3.5普通数组与`@`运算符具有相同的便利性）。
  - `<:(` 元素乘法需要调用函数，`multiply(A,B)`。
  - `<:(` 运算符重载的使用有点不合逻辑：`*`不能按元素操作，但`/`确实如此。

- 与之互动`scipy.sparse`有点清洁。

因此，使用`数组（array）`要明智得多。事实上，我们打算最终废除`矩阵（matrix）`。

MATLAB 和 NumPy粗略的功能对应表

下表给出了一些常见MATLAB®表达式的粗略等价物。**这些不是确切的等价物**，而应该作为提示让你朝着正确的方向前进。有关更多详细信息，请阅读NumPy函数的内置文档。

在下表中，假设您已在Python中执行以下命令：

```
from numpy import *
import scipy.linalg
```

另外如果下表中的**注释**这一列的内容是和 “matrix” 有关的话，那么参数一定是二维的形式。

一般功能的对应表

MATLAB	NumPy	注释
help func	info(func)或者help(func)或func?（在IPython的）	获得函数func的帮助
which func	请参阅备注	找出func定义的位置
type func	source(func)或者func??（在Ipython中）	func的打印源（如果不是本机函数）
a && b	a and b	短路逻辑AND运算符（Python本机运算符）；只有标量参数
a		b
1*i，1*j，1i，1j	1j	复数
eps	np.spacing(1)	1与最近的浮点数之间的距离。
ode45	scipy.integrate.solve_ivp(f)	将ODE与Runge-Kutta 4,5集成
ode15s	scipy.integrate.solve_ivp(f, method='BDF')	将ODE与BDF方法集成

线性代数功能对应表

MATLAB	NumPy	注释
ndims(a)	ndim(a) 要么 a.ndim	获取数组的维数
numel(a)	size(a) 要么 a.size	获取数组的元素数
size(a)	shape(a) 要么 a.shape	得到矩阵的“大小”
size(a,n)	a.shape[n-1]	获取数组第n维元素的数量a。（请注意，MATLAB®使用基于1的索引，而Python使用基于0的索引，请参阅备注）
[1 2 3; 4 5 6]	array([[1.,2.,3.], [4.,5.,6.]])	2x3矩阵文字
[a b; c d]	block([[a,b], [c,d]])	从块构造一个矩阵a，b，c，和d
a(end)	a[-1]	访问1xn矩阵中的最后一个元素 a
a(2,5)	a[1,4]	访问第二行，第五列中的元素
a(2,:)	a[1] 或者 a[1,:]	a的第二行
a(1:5,:)	a[0:5]或a[:5]或a[0:5,:]	前五行 a
a(end-4:end,:)	a[-5:]	a的最后五行
a(1:3,5:9)	a[0:3][:,4:9]	a的第一至第三行与第五至第九列交叉的元素。这提供了只读访问权限。
a([2,4,5],[1,3])	a[ix_([1,3,4],[0,2])]	第2,4,5行与第1,3列交叉的元素。这允许修改矩阵，并且不需要常规切片。
a(3:2:21,:)	a[2:21:2,:]	返回a的第3行与第21行之间每隔一行的行，即第3行与第21行之间的a的奇数行
a(1:2:end,:)	a[::2,:]	返回a的奇数行
a(end: -1:1,:) 要么 flipud(a)	a[::-1,:]	以相反的顺序排列的a的行
a([1:end 1],:)	a[r_[:len(a),0]]	a 的第一行添加到a的末尾行的副本
a.'	a.transpose() 要么 a.T	转置 a
a'	a.conj().transpose() 要么 a.conj().T	共轭转置 a
a * b	a @ b	矩阵乘法
a .* b	a * b	元素乘法
a./b	a/b	元素除法
a.^3	a**3	元素取幂

MATLAB	NumPy	注释
(a>0.5)	(a>0.5)	其i，jth元素为（a_ij> 0.5）的矩阵。Matlab结果是一个0和1的数组。NumPy结果是布尔值的数组False和True。
find(a>0.5)	nonzero(a>0.5)	找到a中所有大于0.5的元素的线性位置
a(:,find(v>0.5))	a[:,nonzero(v>0.5)[0]]	提取a中向量v> 0.5的对应的列
a(:,find(v>0.5))	a[:,v.T>0.5]	提取a中向量v> 0.5的对应的列
a(a<0.5)=0	a[a<0.5]=0	a中小于0.5的元素赋值为0
a .* (a>0.5)	a * (a>0.5)	返回一个数组，若a中对应位置元素大于0.5，取该元素的值；若a中对应元素<=0.5，取值0
a(:) = 3	a[:] = 3	将所有值设置为相同的标量值
y=x	y = x.copy()	numpy通过引用分配
y=x(2,:)	y = x[1,:].copy()	numpy切片是参考
y=x(:)	y = x.flatten()	将数组转换为向量（请注意，这会强制复制）
1:10	arange(1.,11.)或*r[1.:11.]或* r[1:10:10j]	创建一个增加的向量，步长为默认值1（参见备注）
0:9	arange(10.)或 r*[:10.]或* r[:9:10j]	创建一个增加的向量，步长为默认值1（参见注释范围）
[1:10]'	arange(1.,11.)[:, newaxis]	创建列向量
zeros(3,4)	zeros((3,4))	3x4二维数组，充满64位浮点零
zeros(3,4,5)	zeros((3,4,5))	3x4x5三维数组，全部为64位浮点零
ones(3,4)	ones((3,4))	3x4二维数组，充满64位浮点数
eye(3)	eye(3)	3x3单位矩阵
diag(a)	diag(a)	返回a的对角元素
diag(a,0)	diag(a,0)	方形对角矩阵，其非零值是元素 a
rand(3,4)	random.rand(3,4) 要么 random.random_sample((3, 4))	随机3x4矩阵
linspace(1,3,4)	linspace(1,3,4)	4个等间距的样本，介于1和3之间
[x,y]=meshgrid(0:8,0:5)	mgrid[0:9.,0:6.] 要么 meshgrid(r*[0:9.],r*[0:6.]	两个2D数组：一个是x值，另一个是y值

MATLAB	NumPy	注释
	ogrid[0:9.,0:6.] 要么 ix*r*[0:9.],r_[0:6.]	在网格上评估函数的最佳方法
[x,y]=meshgrid([1,2,4],[2,4,5])	meshgrid([1,2,4],[2,4,5])	
	ix_([1,2,4],[2,4,5])	在网格上评估函数的最佳方法
repmat(a, m, n)	tile(a, (m, n))	用n份副本创建m a
[a b]	concatenate((a,b),1)或者 hstack((a,b))或 column_stack((a,b))或c_[a,b]	连接a和的列b
[a; b]	concatenate((a,b))或 vstack((a,b))或r_[a,b]	连接a和的行b
max(max(a))	a.max()	最大元素a（对于matlab，ndims（a）<= 2）
max(a)	a.max(0)	每列矩阵的最大元素 a
max(a,[],2)	a.max(1)	每行矩阵的最大元素 a
max(a,b)	maximum(a, b)	比较a和b逐个元素，并返回每对中的最大值
norm(v)	sqrt(v @ v) 要么 np.linalg.norm(v)	L2矢量的规范 v
a & b	logical_and(a,b)	逐个元素AND运算符（NumPy ufunc）请参阅备注LOGICOPS
a	b	logical_or(a,b)
bitand(a,b)	a & b	按位AND运算符（Python native和NumPy ufunc）
bitor(a,b)	a	b
inv(a)	linalg.inv(a)	方阵的逆 a
pinv(a)	linalg.pinv(a)	矩阵的伪逆 a
rank(a)	linalg.matrix_rank(a)	二维数组/矩阵的矩阵秩 a
a\b	linalg.solve(a,b)如果a是正方形; linalg.lstsq(a,b) 除此以外	ax = b的解为x
b/a	解决aT xT = bT	xa = b的解为x
[U,S,V]=svd(a)	U, S, Vh = linalg.svd(a), V = Vh.T	奇异值分解 a

MATLAB	NumPy	注释
chol(a)	linalg.cholesky(a).T	矩阵的cholesky分解（chol(a)在matlab中返回一个上三角矩阵，但linalg.cholesky(a)返回一个下三角矩阵）
[V,D]=eig(a)	D,V = linalg.eig(a)	特征值和特征向量 a
[V,D]=eig(a,b)	D,V = scipy.linalg.eig(a,b)	特征值和特征向量a，b
[V,D]=eigs(a,k)		找到k最大的特征值和特征向量a
[Q,R,P]=qr(a,0)	Q,R = scipy.linalg.qr(a)	QR分解
[L,U,P]=lu(a)	L,U = scipy.linalg.lu(a) 要么 LU,P=scipy.linalg.lu_factor(a)	LU分解（注：P（Matlab）==转置（P（numpy）））
conjgrad	scipy.sparse.linalg.cg	共轭渐变求解器
fft(a)	fft(a)	傅立叶变换 a
ifft(a)	ifft(a)	逆傅立叶变换 a
sort(a)	sort(a) 要么 a.sort()	对矩阵进行排序
[b,I] = sortrows(a,i)	I = argsort(a[:,i]), b=a[I,:]	对矩阵的行进行排序
regress(y,X)	linalg.lstsq(X,y)	多线性回归
decimate(x, q)	scipy.signal.resample(x, len(x)/q)	采用低通滤波的下采样
unique(a)	unique(a)	
squeeze(a)	a.squeeze()	

备注

子矩阵：使用该 `ix_` 命令可以使用索引列表完成**对子**矩阵的分配。例如，对于2D数组 `a`，可能会做：`ind=[1,3]; a[np.ix_(ind,ind)]+=100`。

帮助：有MATLAB的没有直接等价 `which` 的命令，但命令 `help` 和 `source` 通常会列出其中函数所在的文件名。Python还有一个 `inspect` 模块（do `import inspect`），它提供了一个 `getfile` 经常工作的模块。

索引：MATLAB®使用一个基于索引，因此序列的初始元素具有索引1.Python使用基于零的索引，因此序列的初始元素具有索引0.出现混淆和火焰，因为每个元素都有优点和缺点。一种基于索引的方法与常见的人类语言使用一致，其中序列的"第一"元素具有索引1.基于零的索引简化了索引。另见prof.dr的文本。Edsger W. Dijkstra。

范围：在MATLAB®中，`0:5` 可以用作范围文字和"切片"索引（括号内）；然而，在Python，构建体等 `0:5` 可以 *仅* 被用作切片指数（方括号内）。因此，`r_` 创建了一些有点古怪的对象，以使numpy具有类似的简洁范围构造机制。请注意，`r_` 它不像函数或构造函数那样
被调用，而是
使用方括号进行 *索引*，这允许在参数中使用Python的切片语法。

逻辑运算：＆或| 在NumPy中是按位AND / OR，而在Matlab＆和|中 是逻辑AND / OR。任何具有重要编程经验的人都应该清楚这种差异。这两者似乎工作原理相同，但存在重要差异。如果您使用过Matlab的＆或| 运算符，您应该使用NumPy ufuncs logical_and / logical_or。Matlab和NumPy的＆和|之间的显着差异 运营商是：

- 非逻辑{0,1}输入：NumPy的输出是输入的按位AND。Matlab将任何非零值视为1并返回逻辑AND。例如，NumPy中的（3和4）是0，而在Matlab中，3和4都被认为是逻辑真，而（3和4）返回1。
- 优先级：NumPy的＆运算符优先于<和>之类的逻辑运算符; Matlab是相反的。

如果你知道你有布尔参数，你可以使用NumPy的按位运算符，但要注意括号，如：z =（x> 1）＆（x <2）。缺少NumPy运算符形式的logical_and和logical_or是Python设计的一个不幸结果。

重塑与线性索引：Matlab总是允许使用标量或线性索引访问多维数组，而NumPy则不然。线性索引在Matlab程序中很常见，例如矩阵上的find()返回它们，而NumPy的查找行为则不同。在转换Matlab代码时，可能需要首先将矩阵重新整形为线性序列，执行一些索引操作然后重新整形。由于重塑（通常）会在同一存储上生成视图，因此应该可以相当有效地执行此操作。请注意，在NumPy中重新整形使用的扫描顺序默认为'C'顺序，而Matlab使用Fortran顺序。如果您只是简单地转换为线性序列，那么这无关紧要。但是如果要从依赖于扫描顺序的Matlab代码转换重构，那么这个Matlab代码：z = reshape(x，3,4)应该变成 z = x.reshape(3,4,order='F').copy() 。

自定义您的环境

在MATLAB®中，可用于自定义环境的主要工具是使用您喜欢的功能的位置修改搜索路径。您可以将此类自定义项放入MATLAB将在启动时运行的启动脚本中。

NumPy，或者更确切地说是Python，具有类似的功能。

- 要修改Python搜索路径以包含您自己的模块的位置，请定义`PYTHONPATH`环境变量。
- 要在启动交互式Python解释器时执行特定的脚本文件，请定义`PYTHONSTARTUP`环境变量以包含启动脚本的名称。

与MATLAB®不同，可以立即调用路径上的任何内容，使用Python，您需要先执行“import”语句，以使特定文件中的函数可访问。

例如，您可能会创建一个如下所示的启动脚本（注意：这只是一个示例，而不是“最佳实践”的声明）：

```
# Make all numpy available via shorter 'np' prefix
import numpy as np
# Make all matlib functions accessible at the top level via M.func()
import numpy.matlib as M
# Make some matlib functions accessible directly at the top level via, e.g.
rand(3,3)
from numpy.matlib import rand,zeros,ones,empty,eye
# Define a Hermitian function
def hermitian(A, **kwargs):
    return np.transpose(A,**kwargs).conj()
# Make some shortcuts for transpose,hermitian:
#    np.transpose(A) --> T(A)
#    hermitian(A) --> H(A)
T = np.transpose
H = hermitian
```

链接

有关另一个MATLAB®/ NumPy交叉引用，请参见http://mathesaurus.sf.net/。

可以在主题软件页面中找到用于python科学工作的广泛工具列表。

MATLAB®和SimuLink®是The MathWorks的注册商标。