Python Numpy 入门教程

本教程最初由 Justin Johnson 撰写创立。现由Qi (Lewis) Liu 翻译整理。

在本门课程（CS231n）中， 我们将使用Python编程语言来提交所有的作业。Python就其本身而言是一门通用的编程语言，得益于众多受欢迎的开源库（numpy, scipy, matplotlib），其成为科学计算领域不可多得的强大工具。

我们希望你们对于Python和numpy的使用或多或少有些经验；如果你没有这方面的经验，那么本教程就是为你而生，你可以将其视作一个速成课程，你将快速学会Python编程语言以及Python在科学计算中的使用。

可能你们中有些人之前用过Mathlab，那么我们也推荐你去看 numpy for Matlab users .

你也可以找到 IPython notebook tutorial 早前版本的一个教程，由 Volodymyr Kuleshov 和 Isaac Caswell 撰写创立，用于先修课 CS 228.

目录:

Python

Python是一门高级的，动态类型，多参数的编程语言。Python程序被誉为“可执行的伪代码”，因为你允许你使用极少的几行代码就可以实现强大的逻辑。作为示例，下面是使用Python编写的经典快速排序实现。

```
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]))
# 输出:  "[1, 1, 2, 3, 6, 8, 10]"
```

Python 版本

目前有两个官方支持的Python版本，一个是2.7版本，另一个是3.5的。

那么该选择哪个呢？Python 3.0 引入了很多向后不兼容的语法，所以使用Python 2.7编写的代码可能在3.5版本之下无法运行，反之亦然。

对于本门课程来说，所有代码将采用Python 3.5. (译者注：事实上并非如此)

你可以通过在命令行上输入以下代码来查看自己的Python 版本： `python --version`.

基本数据类型

和大多数编程语言一样，Python有一些列自己的基本数据类型，包括整型，浮点型，布尔型，字符串类型。其属性和其他编程语言类似。

数字类型 Numbers: 整型和浮点型可能如你所想和其他编程语言中的用法无异。

```
x = 3
print(type(x)) # 输出 "<class 'int'>"
print(x)       # 输出 "3"
print(x + 1)   # 加; 输出 "4"
print(x - 1)   # 减; 输出 "2"
print(x * 2)   # 乘; 输出 "6"
print(x ** 2)  # 幂; 输出 "9"
x += 1
print(x)  # 输出 "4"
x *= 2
print(x)  # 输出 "8"
y = 2.5
print(type(y)) # 输出 "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # 输出 "2.5 3.5 5.0 6.25"
```

注意，和很多编程语言不一样的是，Python没有一元的增 (`x++`) 和一元减 (`x--`) 操作符。

对于更复杂的数字Python也提供了很多内置类型，具体细节参见这篇文档.

布尔类型 Booleans: Python实现了所有的常用的布尔逻辑操作符，但是，不是符号（`&&`, `||`, 等），取而代之的是英语单词（`and`, `or`, `not`,等）：

```
t = True
f = False
print(type(t)) # 输出 "<class 'bool'>"
print(t and f) # 逻辑 与; 输出 "False"
print(t or f)  # 逻辑 或; 输出 "True"
print(not t)   # 逻辑 非; 输出 "False"
print(t != f)  # 逻辑 异或; 输出 "True"
```

字符串类型 Strings: Python对于字符串类型有非常强大的支持性:

```
hello = 'hello'    # 字符串可以使用单引号阔起来,
world = "world"    # 也可以使用双引号; 两者完全等效
print(hello)       # 输出 "hello"
print(len(hello))  # 字符串长度; 输出 "5"
hw = hello + ' ' + world  # 字符串级联
print(hw)  # 输出 "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf 风格的字符串格式化
print(hw12)  # 输出 "hello world 12"
```

字符串对象有许多有用的方法；例如：

```
s = "hello"
print(s.capitalize())  # 将字符串首字母大写; 输出 "Hello"
print(s.upper())       # 将字符串转换成全大写; 输出 "HELLO"
print(s.rjust(7))      # 将字符串右对齐, 不够使用空格填充; 输出 "  hello"
print(s.center(7))     # 将字符串中间对其, 不够使用空白填充; 输出 " hello "
print(s.replace('l', '(ell)'))  # 将字符串中所有找到的字串使用另一个字串替换
                                # 输出 "he(ell)(ell)o"
print('  world '.strip())  # 跳过字符串首部和尾部的空字符串; 输出 "world"
```

想查看所有的字符串内置方法的列表，看 这篇文档.

容器 Containers

Python 中包含了好几种内置的容器类型：列表(lists), 字典(dictionaries), 集合(sets), 和元组(tuples).

列表 Lists

Python中列表(list) 相当于是阵列/数组(array), 但是Python列表是大小可变的

并且**可以存储不同类型的元素** :

```
xs = [3, 1, 2]    # 创建一个列表
print(xs, xs[2])  # 输出 "[3, 1, 2] 2"
print(xs[-1])     # 负的索引，将从列表的后边开始计数；输出 "2"
xs[2] = 'foo'     # 列表可以包含不同类型的元素
print(xs)         # 输出 "[3, 1, 'foo']"
xs.append('bar')  # 向列表中最后增加一个元素
print(xs)         # 输出 "[3, 1, 'foo', 'bar']"
x = xs.pop()      # 移除并返回列表中最后一个元素
print(x, xs)      # 输出 "bar [3, 1, 'foo']"
```

关于列表使用凶残的细节，我不会说可以在这篇文档中找到的。

切片 Slicing: 除了每次访问列表元素的一个元素外，Python提供了一种简洁的语法来访问列表中的一部分；这种方式叫做切片：

```
nums = list(range(5))     # range 为一个内置函数，用来创建一个整型的列表
print(nums)               # 输出 "[0, 1, 2, 3, 4]"
print(nums[2:4])          # 得到一个切片，元素为索引 2 到 4 (开) 的列表元素；输出 "[2, 3]"
print(nums[2:])           # 得到一个切片，元素为索引 2 到 结尾的元素；输出 "[2, 3, 4]"
print(nums[:2])           # 得到一个切片，元素为索引从开始(0) 到 2 (开)的元素；输出 "[0, 1]"
print(nums[:])            # 得到一个切片，元素为整个列表的全部元素；输出 "[0, 1, 2, 3, 4]"
print(nums[:-1])          # 切片的索引也可以为负值；输出 "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # 将一个列表对切片进行赋值
print(nums)               # 输出 "[0, 1, 8, 9, 4]"
```

我们在介绍 numpy arrays 的时候还会见到切片。

循环 Loops: 你可以像下面这样遍历列表元素:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# 输出 "cat", "dog", "monkey", 每个元素一行.
```

如果你想在循环体中获取其中元素的索引，需要使用内置函数`enumerate`l:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# 输出 "#1: cat", "#2: dog", "#3: monkey", 每个输出一行
```

列表解析 List comprehensions: 编程工程中，我梦经常需要将一种类型的数据转换成另外一种类型的数据。

下面是计算数字平方数的例子:

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

可以使用**列表解析 list comprehension**来更加优雅地完成上述任务：

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

列表解析中也可以包含条件表达式：

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

字典 Dictionaries

字典用来存储键值对——(key, value)，和Java中的`map`或者Javascript中的对象类似。其用法如下：

```
d = {'cat': 'cute', 'dog': 'furry'}  # 创建一个字典
print(d['cat'])       # 获取字典中的某项；输出 "cute"
print('cat' in d)     # 检查字典中是否有某个给定的键；输出 "True"
d['fish'] = 'wet'     # 修改字典中的某项
print(d['fish'])      # 输出 "wet"
# print(d['monkey'])  # 键的错误，KeyError: 'monkey' 不是字典 d 中的键
print(d.get('monkey', 'N/A'))  # 获取到一个默认的元素；输出 "N/A"
print(d.get('fish', 'N/A'))    # 获取到一个默认的元素；输出 "wet"
del d['fish']         # 从字典中移除一个元素
print(d.get('fish', 'N/A')) # "fish" 不再是其中的一个键；输出 "N/A"
```

关于字典的更多的用法，请阅读 这篇文档.

遍历: 字典很容易遍历其中的键:

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

如果你想在遍历字典过程中，同时获取字典中的键和键对应的值，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))
# 输出 "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

字典解析 Dictionary comprehensions: 字典解析和列表解析类似，能够以更快的方式建立字典。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)  # 输出 "{0: 0, 2: 4, 4: 16}"
```

集合 Sets

集合是不同无序元素组成的集体。其基本的用法，参见下方示例：

```
animals = {'cat', 'dog'}
print('cat' in animals)   # 检查某元素是否是集合中的元素；输出 "True"
print('fish' in animals)  # 输出 "False"
animals.add('fish')       # 向集合中增加一个元素
print('fish' in animals)  # 输出 "True"
print(len(animals))       # 集合中元素的数量；输出 "3"
animals.add('cat')        # 向集合中增加一个已经存在的元素
print(len(animals))       # 输出 "3"
animals.remove('cat')     # 从结合中移除一个元素
print(len(animals))       # 输出 "2"
```

同样，如果想知道关于集合的更多用法的介绍请阅读这篇文档.

遍历: 遍历集合和遍历列表的语法相同; 但是，由于集合是无序的，所以你没办法假设你访问的集合中元素就是按照显示的顺序访问的：

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

集合解析 Set comprehensions: 和集合以及字典类似，你也可以使用集合解析来构建集合:

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

元组 Tuples

元组是值不可更改的有序列表。

元祖在很多方面和列表一样, 和列表最大的不同点就是元祖可以作为字典以及集合中的键, 而列表是不可以这样的。

下面是一个简单的示例：

```
d = {(x, x + 1): x for x in range(10)}  # 创建一个带有元组键值的字典
t = (5, 6)        # 创建一个元组
print(type(t))    # 输出 "<class 'tuple'>"
print(d[t])       # 输出 "5"
print(d[(1, 2)])  # 输出 "1"
```

这个文档 有关于元组更多的信息。

函数 Functions

Python 函数使用 `def` 关机字来构建. 举例:

```
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))
# 输出 "negative", "zero", "positive"
```

我们常常定义一些包含一些可选参数的函数，就像下面这个一样：

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

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

这篇文档 中有关于Python 函数的更多用法.

类 Classes

Python 用来定义类的语法很简单:

```
class Greeter(object):

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

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

g = Greeter('Fred')  # 构建一个 Greeter 类的实例
g.greet()            # 调用一个实例方法；输出 "Hello, Fred"
g.greet(loud=True)   # 调用一个实例方法；输出 "HELLO, FRED!"
```

关于Python类，更多请阅读 这篇文档.

Numpy

Numpy 是Python中用于科学计算的核心库。它提供了高效的多维阵列/数组对象以及用来处理这种对象的工具。如果你已经对MATLAB比较熟悉，这篇文档 可能对于你开始使用Numpy有帮助。

阵列/数组 Arrays

一个 numpy 阵列/数组 可以视作是由值构成的网格，其中所有元素为同一类型，通过非负值元组来索引。阵列/数组的秩(rank)是维度数;阵列/数组的大小(*shape*)是一个整型的阵列/数组，其给出了每个维度下的阵列/数组的大小。

我们可以使用Python 嵌套列表来初始化一个 numpy 阵列/数组，并且使用方括号来方位其中的元素:

```
import numpy as np

a = np.array([1, 2, 3])   # 创建一个秩为 1 的阵列/数组
print(type(a))            # 输出 "<class 'numpy.ndarray'>"
print(a.shape)            # 输出 "(3,)"
print(a[0], a[1], a[2])   # 输出 "1 2 3"
a[0] = 5                  # 改变阵列/数组中的某个元素
print(a)                  # 输出 "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # 创建一个秩为 2 的阵列/数组
print(b.shape)                     # 输出 "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # 输出 "1 2 4"
```

Numpy 也提供了很多其他的方法来创建阵列/数组:

```
import numpy as np

a = np.zeros((2,2))   # 创建一个全 0 的阵列/数组
print(a)              # 输出 "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # 创建一个全 1 的阵列/数组
print(b)              # 输出 "[[ 1.  1.]]"

c = np.full((2,2), 7)  # 创建一个常阵列/数组
print(c)               # 输出 "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # 创建一个 a 2x2 单位矩阵
print(d)              # 输出 "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # 创建一个随机数阵列/数组
print(e)                     # 也许会输出 "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

想知道关于阵列/数组创建更多的内容请阅读 这篇文档.

阵列/数组索引(/访问) Array indexing

Numpy 提供了集中索引到阵列/数组中的方法。

切片 Slicing: 类似于 Python 列表, numpy 阵列/数组也支持切片。 由于numpy 的阵列/数组时多维的，所以你必须明确阵列/数组每个维度的切片。

```
import numpy as np

# 创建一个秩为 2 大小为 (3, 4) 的阵列/数组
# [[ 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]])

# 使用切片来抽取 a 的子阵列/数组，包含其前两行，第 1 到 2 列中的元素。
# b 为注释中 (2, 2) 的阵列/数组:
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# 一个数据的切片是阵列/数组时同一数据区的不同视图，所以对于切片修改也会修改原始阵列/数组的元素
print(a[0, 1])   # 输出 "2"
b[0, 0] = 77     # b[0, 0] 和 a[0, 1] 是完全相同的数据
print(a[0, 1])   # 输出 "77"
```

你可以将整数索引和切片索引混合使用。 但是，这样做会产生比原始阵列/数组秩/级(rank)耕地的阵列/数组。 注意，这和MATLAB中处理阵列/数组切片的方式完全不同:

```
import numpy as np

# 创建如下秩为 2 ，大小 (3, 4) 的阵列/数组
# [[ 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]])

# 阵列/数组中间行的两种访问方式
# 混合使用整数和切片作为索引的方式产生了秩较小的阵列/数组，
# 而仅使用切片索引的方式产生了和原始阵列/数组同样秩的阵列/数组:
row_r1 = a[1, :]    # 秩为 1，视作 a 的第二行
row_r2 = a[1:2, :]  # 秩为 2，视作 a 的第二行
print(row_r1, row_r1.shape)  # 输出 "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # 输出 "[[5 6 7 8]] (1, 4)"

# 当访问一个阵列/数组的列，同样可以用这两种方式来观察差别:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # 输出 "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # 输出 "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

整型阵列/数组索引 Integer array indexing: 当你采用切片的方式索引到 numpy 阵列/数组中，得到的结果形式上总是原始阵列/数组的一个子阵列/数组。相较而言，整型整列索引能够使用其他阵列/数组中的数据来构建你想要的阵列/数组。举例如下:

```
import numpy as np

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

# 整型整列索引举例
# 返回一个大小为 (3,) 的阵列/数组
print(a[[0, 1, 2], [0, 1, 0]])  # 输出 "[1 4 5]"

# 上面整型阵列/数组索引的示例等效于下面这种表达：
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # 输出 "[1 4 5]"

# 整型整列索引，可以重复使用原数组中的元素
print(a[[0, 0], [1, 1]])  # 输出 "[2 2]"

# 等效于下面这种表达
print(np.array([a[0, 1], a[0, 1]]))  # 输出 "[2 2]"
```

整型整列索引中有个实用的技巧，那就是选择或者更改矩阵每一行的一个元素：

```
import numpy as np

# 创建一个新的阵列/数组
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

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

# 创建一个索引值阵列/数组
b = np.array([0, 2, 0, 1])

# 使用 b 中的索引值来在矩阵 a 中的每一行中选择对应元素
print(a[np.arange(4), b])  # 输出 "[ 1  6  7 11]"

# 使用 b 中的索引值来更改矩阵 a 中的每一行中选择对应元素
a[np.arange(4), b] += 10

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

布尔型阵列/数组索引 Boolean array indexing: 布尔型阵列/数组缩影能够让你选择阵列/数组中的任意元素。这种类型的索引一般用来选择满足某些条件的阵列/数组中的元素。举例如下：

```
import numpy as np

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

bool_idx = (a > 2)   # 找出所有大于 2 的元素;
                     # 这句代码返回一个 numpy 阵列/数组，大小与 a 相等，
```

```
                    # bool_idx 中每一个位置对应于 a 相应位置元素是否大于 2

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

# 我们使用布尔型的阵列/数组来构建一个秩为 1 的阵列/数组，其由
# bool_idx 中值为 True 对应的 a 中的元素组成
print(a[bool_idx])  # 输出 "[3 4 5 6]"

# 上述操作我们也可以使用一句简单的表达来完成:
print(a[a > 2])     # 输出 "[3 4 5 6]"
```

对于numpy阵列/数组的索引技术，这里遗漏了很多具体细节，如果你有勇气了解它们，请阅读 这篇文档.

数据类型

每个numpy阵列/数组可以视作由相同数据类型醉成的网格。Numpy提供了一系列的数值型数据类型来构建阵列/数组。当你创建一个阵列/数组时，numpy会想假设其数据类型，但是构建阵列/数组的函数通常会包含一个参数来明确数据类型。下面为一个具体示例:

```
import numpy as np

x = np.array([1, 2])   # numpy 自行“选择”的数值类型
print(x.dtype)         # 输出 "int64"

x = np.array([1.0, 2.0])   # numpy 自行“选择”的数值类型
print(x.dtype)             # 输出 "float64"

x = np.array([1, 2], dtype=np.int64)   # 指定某种特定的数据类型
print(x.dtype)                         # 输出 "int64"
```

关于 numpy 的数据类型，你可以阅读这篇文档 了解更多。

阵列/数组数学运算

在numpy模块中，不单是提供了运算符重载还提供了许多函数，能够进行基本的数学运算:

```
import numpy as np

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

# 对应元素之和(sum); 下面两种方式结果一致
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# 对应元素之差(difference); 下面两种方式结果一致
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
```

```
print(x - y)
print(np.subtract(x, y))

# 对应元素之积(product); 下面两种方式结果一致
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# 对应元素之除(division); 下面两种方式结果一致
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# 逐元素求平方根(square root); 得到结果如下
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

注意和 MATLAB 不一样，这里`*`符号是对应元素之积，而不是矩阵乘法。这里我们使用`dot`函数来计算向量间，向量和矩阵，矩阵间的乘法。numpy模块中`dot`既可以作为内置函数也可以作为阵列/数组对象的实例方法来调用:

```
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])

# 向量间内积(Inner product) ; 两种方法均得到结果 219
print(v.dot(w))
print(np.dot(v, w))

# 矩阵向量乘法(product); 两种方法均得到秩/级为 1 的阵列/数组 [29 67]
print(x.dot(v))
print(np.dot(x, v))

# 矩阵间乘法(product); 两种方法均得到秩/级为 2 的阵列/数组
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy 对于基于阵列/数组的运算提供了很多有用的函数，其中一个最有用的方法就是`sum`:

```
import numpy as np

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

print(np.sum(x))  # 计算出所有元素的和；输出 "10"
print(np.sum(x, axis=0))  # 对每列(column)求和；输出 "[4 6]"
print(np.sum(x, axis=1))  # 对每行(row)求和；输出 "[3 7]"
```

阅读 这篇文档 你能够找到所有numpy支持的舒徐运算函数。

除了基于阵列/数组的数学运算函数，我们也经常需要对阵列/数组进行大小调整和其中数据的操作。这类操作中的代表就是对矩阵求转置；求一个矩阵的转置，最简单的方式是调用一个阵列/数组对象的 `T` 属性：

```
import numpy as np

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

# 注意对于秩为 1 的阵列/数组求转置，无变化：
v = np.array([1,2,3])
print(v)    # 输出 "[1 2 3]"
print(v.T)  # 输出 "[1 2 3]"
```

阅读 这篇文档 你可以看到Numpy提供的用来对阵列/数组进行更多操作的更多函数。

广播

广播是numpy中一种强大的机制，用于算数操作，其使得大小不同的阵列/数组进行计算操作。我们往往会遇到同时又两个一大一小的阵列/数组情况，并且期望使用更小的阵列/数组在相对大的阵列/数组上执行一些操作。

例如，我们要为一个矩阵的每一行加上一个常量向量，我们可以这样做：

```
import numpy as np

# 将矩阵 x 的每一行加上向量 v，
# 将结果存储到矩阵 y 中
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)   # 创建一个新矩阵，大小和 x 一致

# 使用明确的的(explicit)循环来对矩阵 x 的每一行加上向量 v
for i in range(4):
    y[i, :] = x[i, :] + v

# 结果 y 如下
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]

#  [11 11 13]]
```

```
print(y)
```

采用这种方式是可行的；然而，当矩阵 `x` 非常大的时候，这种采用明确循环的计算在Python中会运行得非常慢。注意，将向量 v 加到矩阵 x 的每行，相当于将多个相同的向量 v 摞(stack)起来组成 vv, 然后对 x 和 vv 相加。采用下面方法可以实现这一操作：

```
import numpy as np

# 将矩阵 x 的每一行加上向量 v,
# 将结果存储到矩阵 y 中
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))   # 将 4 个向量 v 垂直地摞起来
print(vv)                 # 输出 "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # x 和 vv 逐元素相加
print(y)  # 输出 "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Numpy 广播让我们不需要实际创建多个向量 `v` 副本来进行计算。下方为使用广播来计算地版本：

```
import numpy as np

# 将向量 v 加到矩阵 x 的每行
# 矩阵 y 用以存储计算结果
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # 通过广播机制，将 v 加到 x 矩阵的每行
print(y)  # 输出 "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

虽然其中 `x` 的大小是 `(4, 3)` ，而 `v` 的大小为 `(3,)` ,但是 `y = x + v` 这行代码能够正常运行得到正确的结果，这需要感谢Python的广播机制；这行代码之所以能够运行，是因为 `v` 的实际大小是 `(4, 3)` 其中每行就是 向量 `v` 的一个拷贝，之后逐元素进行运算。

广播两个阵列/数组遵循以下这些规则：

1. 如果两个阵列/数组的秩/级(rank)不一样，需要先以一为单位来扩展秩/级比小的阵列/数组知道两者的大小一致。
2. 如果两个阵列/数组的维度一致，或者其中一个维度数为1，我们就说这两个阵列/数组在维度上是相容的。
3. 如果两个阵列/数组在所有维度上是相容的，则说他们是可以广播的。
4. 广播之后，每个阵列/数组的大小将形如输入阵列/数组中每次对应项中最大的尺寸(译者注: 这段话不太好翻译)。(After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.)

5. 如果一个矩阵其任意一维上的长度为 1，而另外一个矩阵在该维下长度大于 1，那么在该维度上就形如该维长度为 1 的数组进行了复制。

如果这段说明你没有理解, 请阅读 这篇文档 或者 这个解释.

支持广播的函数被称为 *通用函数 (universal functions)*。想了解更多通用函数，请阅读 这篇文档.

这里又广播的几个应用：

```
import numpy as np

# 计算两向量的外积
v = np.array([1,2,3])  # v 大小为 (3,)
w = np.array([4,5])    # w 大小 (2,)
# 计算向量的外积，首先将向量 v 调整大小/重塑(resize)为一个列向量，
# 大小为(3, 1)，然后对 w 广播 v 得到 v 和 w 的外积，大小为(3, 2):
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# 将某向量加到矩阵每一行
x = np.array([[1,2,3], [4,5,6]])
# x 大小为 (2, 3) and v 大小为 (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# 将某向量加到矩阵每一列
# x 大小为 (2, 3) and w 大小为 (2,).
# x 的转置大小为 (3, 2) 然乎对 w 广播
# 得到的结果大小 (3, 2); 对该结果转置
# 得到的结果大小 (2, 3) , 得到矩阵 x
# 每列加上向量 w的计算结果。
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# 另一种方法是将 w 调整为大小为列向量，大小为(2, 1);
# 我们就可以对 x 直接广播 w ，得到和上述方法相同的结果。
print(x + np.reshape(w, (2, 1)))

# 将一个矩阵乘以一个常数:
# x 大小为 (2, 3). Numpy 将常量视作大小为 shape(x) 的阵列/数组;
# 例子中被广播成大小为shape(2, 3)，得到以下结果:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

广播机制一般情况下能使得你的代码变得简介而高效，所以你需要努力尽可能在能广播的时候尽量广播。

Numpy 文档

本文以及涉及了numpy中很多你需要掌握的内容，但是只是大概介绍了一下，还远远不够和完整。关于numpy，想了解更多，请阅读 numpy 文档。

SciPy

Numpy提供了高效的多维阵列/数组对象以及计算和处理阵列/数组的多种基础工具。 SciPy 就是基于此建立的强大工具，它提供了可以在 numpy 阵列/数组上操作的多种函数，并且在科学和工程领域有极其广泛的应用。

熟悉SciPy最好的方式就是阅读 这篇文档. z在这里我们着重挑选了 SciPy 中几个重要的模块，并将在这门课中练习使用。

图片操作

SciPy提供了一些基本的函数来处理图片(image). l例如，其提供了将图片从存储磁盘中读取到 numpy 阵列/数组的函数， 将 numpy 阵列/数组写入到图像的函数，重新调整图片大小的函数等等。 这里有一个简单示例，来展示上述功能：

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

# 读取一幅 JPEG 图片到 numpy 阵列/数组中
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # 输出 "uint8 (400, 248, 3)"

# 我们可以对图片的每个通道值乘以一个标量值来对图片进行着色
# 例子中图片大小为 (400, 248, 3);
# 将其乘以阵列/数组 [1, 0.95, 0.9] 大小为(3,);
# 由于 numpy 的广播机制，图片的红色通道值将不做改变，
# 绿色通道和蓝色通道分别乘以 0.95 和 0.9
img_tinted = img * [1, 0.95, 0.9]

# 将着色后的图片调整为 300 x 300 像素大小
img_tinted = imresize(img_tinted, (300, 300))

# 将着色后的图片写回磁盘存储
imsave('assets/cat_tinted.jpg', img_tinted)
```

左侧：元素图片。
右侧：重新着色和调整大小后的图片。

MATLAB 文件

使用 `scipy.io.loadmat` 和 `scipy.io.savemat` 函数能够读入和写回 MATLAB 文件 关于其具体用大请阅读 这篇文档.

点间距离

SciPy 定义了很多有用的函数，用来计算点集合中点(译者注：不只是二维空间的点)间距离。

`scipy.spatial.distance.pdist` 函数用来计算给定集合中每对点间距离：

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

# 创建如下发数据，其中每行代表二维平面空间中的一个点：
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# 计算 x 中每行间的欧几里得距离。
# d[i, j] 表示 x[i, :] 与 x[j, :] 间的距离；
# d 是以下矩阵：
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

关于本函数 这篇文档.

有一个类似的函数 (`scipy.spatial.distance.cdist`) 同样用来计算两个集合之间点对之间的距离；关于其用法请参见这篇文档 这篇文档.

Matplotlib

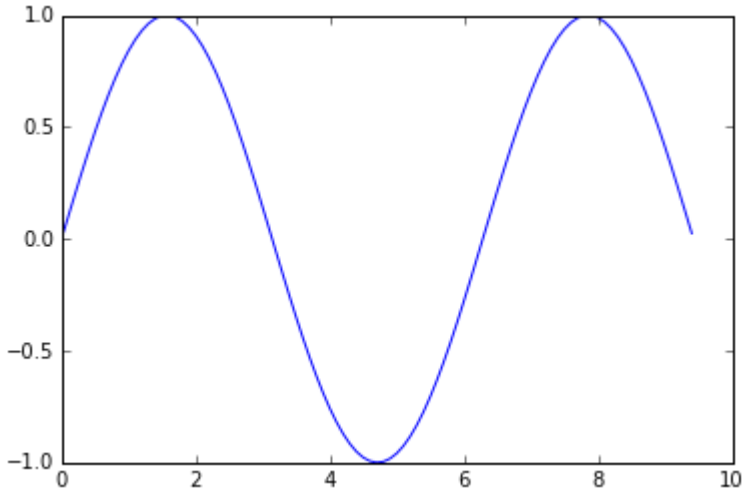
Matplotlib 是一个绘图库。 在这个部分，将对 `matplotlib.pyplot` 模块作简要的介绍，在MATLAB中有与之相似的模块。

绘图 Plotting

matplotlib 中最最重要的函数就是 `plot`, 它可以将你的数据绘制成 2D(平面) 图。下面是一个简单示例：

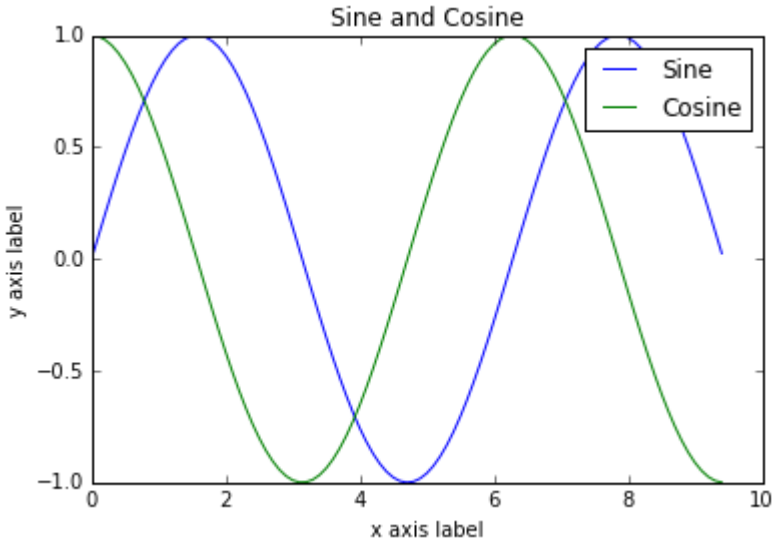
```
import numpy as np
import matplotlib.pyplot as plt

# 计算正弦曲线上点的 x 和 y 坐标
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# 使用 matplotlib 来绘制点
plt.plot(x, y)
plt.show()  # 必须调用 plt.show() 来显示图
```

运行上代码，最终会生成下图:



只需要很少量的工作，我们就能在一张图上绘制多条曲线，为图标增加标题，图例，坐标轴名称：

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

# 计算正弦曲线和余弦去线上点的 x 和 y 坐标
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# 使用 matplotlib 来绘制点
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()
```

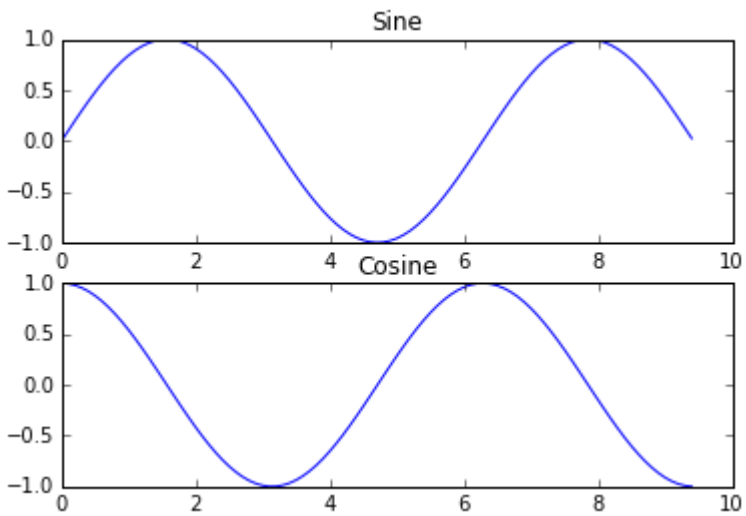


关于 `plot` 函数，想了解更多请阅读 这篇文档.

子图 Subplots

使用 `subplot` 函数能够在绘图(figure)中绘制几幅不同的图像。下方为一个示例：

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

# 计算正弦曲线和余弦去线上点的 x 和 y 坐标
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# 设置子图网格，高度为 2 宽度为 1，
# 并且将第一幅这样的子图设置为当前(activated)
plt.subplot(2, 1, 1)

# 制作第一幅
plt.plot(x, y_sin)
plt.title('Sine')

# 将第二幅这样的子图设置为当前(activated)，制作第二幅图
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# 显示全部绘图(figure)
plt.show()
```



关于 `subplot` 函数，你可以通过阅读 这篇文档 了解更多。

图片 Images

可以使用 `imshow` 函数来展示图片。下面是一个例子:

```
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]

# 显示原始图片
plt.subplot(1, 2, 1)
plt.imshow(img)

# 显示着色的图片
plt.subplot(1, 2, 2)

# 如果要显示的数据不是 uint8 类型编码的，imshow 的一个小毛病是
# 有时候它会输出很奇怪的结果。为了解决这个问题，我们在显示其之前
# 显式地将图片专户为 uint8 类型。
plt.imshow(np.uint8(img_tinted))
plt.show()
```