

Python常用库介绍

Guangrui Qian

Anaconda和python库管理

- ▶ **Anaconda**包含了**conda**、**Python**在内的超过**180**个科学包（库）及其依赖项。可以便捷获取包（库）且对包能够进行管理，同时对环境可以统一管理。
- ▶ **Anaconda**自带包含的科学包包括：**conda**, **numpy**, **scipy**, **ipython notebook**等。
- ▶ **conda**是包及其依赖项和环境的管理工具。
- ▶ **conda**为**Python**项目而创造，但可适用于**Python**, **R**, **Ruby**, **Lua**, **Scala**, **Java**, **JavaScript**, **C/C++**, **FORTRAN**等多种语言环境和库管理。
- ▶ 需要注意**Python pip** 与 **conda**的功能区别

GPU

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow-2.2.0	2.7, 3.5-3.8	GCC 7.3.1	Bazel 0.27.1	7.6	10.1
tensorflow-2.1.0	2.7, 3.5-3.7	GCC 7.3.1	Bazel 0.27.1	7.6	10.1
tensorflow-2.0.0	2.7, 3.3-3.7	GCC 7.3.1	Bazel 0.26.1	7.4	10.0
tensorflow_gpu-1.14.0	2.7, 3.3-3.7	GCC 4.8	Bazel 0.24.1	7.4	10.0
tensorflow_gpu-1.13.1	2.7, 3.3-3.7	GCC 4.8	Bazel 0.19.2	7.4	10.0
tensorflow_gpu-1.12.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.11.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.9.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.11.0	7	9
tensorflow_gpu-1.8.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.10.0	7	9

Tensorflow不同版本对应的库版本

Pip和Conda的功能差距

► pip：维护多个环境难度较大。

- 不一定会展示所需其他依赖包。安装包时或许会直接忽略依赖项而安装，仅在结果中提示错误
- 在系统自带Python中包的“更新/回退版本/卸载”将影响其他程序。仅适用于Python。
- pip从PyPI (Python Package Index) 上拉取数据，它的Repo在PyPI上。绝大多数的Python包会优先发布到PyPI上。截止2020年5月，PyPI上的项目有23万之多。

► conda：比较方便地不同环境之间进行切换，环境管理较为简单。

- 列出所需其他依赖包、安装包时自动安装其依赖项。不会影响系统自带Python
- 可以便捷地在包的不同版本中自由切换，提供了环境隔离，可以使用conda命令创建多个环境
- 适用于Python, R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN

	conda	pip
Repo	Anaconda, 包数量远少于PyPI	PyPI, Python包会被优先发布到PyPI上
包内容	二进制	源码和二进制
支持语言	Python、R、C/C++等	只支持Python
多环境管理	可以创建多个环境，环境内包含Python解释器	本身不支持，需要依赖其他工具
依赖检查	严格的依赖检查	依赖检查不严格

pip install + package名称

conda install + package名称

conda中tensorflow安装和使用

► 创建env环境

```
conda create -n dl python=3.6
```

► 激活(或切换不同python版本)的虚拟环境

```
source activate dl
```

► 安装tesnorflow-gpu

```
conda install tensorflow-gpu==1.14.0
```

```
pip install tensorflow (cpu版本, 注意版本号)
```

► import导入python模块

clean	Remove unused packages and caches.
config	Modify configuration values in .condarc. This is modeled after the git config command. Writes to the user .condarc file (/Users/grqian/.condarc) by default.
create	Create a new conda environment from a list of specified packages.
help	Displays a list of available conda commands and their help strings.
info	Display information about current conda install.
init	Initialize conda for shell interaction. [Experimental]
install	Installs a list of packages into a specified conda environment.
list	List linked packages in a conda environment.
package	Low-level conda package utility. (EXPERIMENTAL)
remove	Remove a list of packages from a specified conda environment.
uninstall	Alias for conda remove.
run	Run an executable in a conda environment. [Experimental]
search	Search for packages and display associated information. The input is a MatchSpec, a query language for conda packages. See examples below.
update	Updates conda packages to the latest compatible version.
upgrade	Alias for conda update.

Python导入模块

- ▶ Python提供了强大的模块支持，主要体现在，不仅 Python 标准库中包含了大量的模块（称为标准模块），还有大量的第三方模块，开发者自己也可以开发自定义模块。通过这些强大的模块可以极大地提高开发者的开发效率。
- ▶ import导入模块

- import 模块名 as 别名

```
# 导入sys整个模块
import sys
# 使用sys模块名作为前缀来访问模块中的成员
print(sys.argv[0])
```

- from 模块名 import 成员名 as 别名

```
# 导入sys模块的argv成员
from sys import argv
# 使用导入成员的语法，直接使用成员名访问
print(argv[0])
```

```
# 导入sys、os两个模块
import sys,os
# 使用模块名作为前缀来访问模块中的成员
print(sys.argv[0])
# os模块的sep变量代表平台上的路径分隔符
print(os.sep)
```

```
# 导入sys模块的argv成员，并为其指定别名v
from sys import argv as v
# 使用导入成员（并指定别名）的语法，直接使用成员的别名访问
print(v[0])
```

Python常用库介绍

- ▶ NumPy科学计算库
- ▶ SciPy-Python科学算法库
- ▶ Pandas数据分析支持库
- ▶ Matplotlib绘图工具库
- ▶ cv2 (opencv-python) 图像处理cv库



<https://www.datacamp.com/community/data-science-cheatsheets>

NumPy科学计算库

- ▶ **NumPy**是**Python**中科学计算的核心库。它提供一个高性能多维数据对象，以及操作这个对象的工具。**NumPy**的全名为**Numeric Python**，是一个开源的**Python**科学计算库。
- ▶ **NumPy**是一个运行速度非常快的数学库，主要用于**数组**计算：
 - 一个强大的**N**维数组对象**ndarray**；
 - 比较成熟的（广播）函数库；
 - 用于整合**C/C++**和**Fortran**代码的工具包；
 - 实用的线性代数、傅里叶变换和随机数生成函数
- ▶ **NumPy** 通常与 **SciPy**（**Scientific Python**）和 **Matplotlib**（绘图库）一起使用，这种组合广泛用于替代 **MatLab**，是一个强大的科学计算环境，有助于我们通过 **Python** 学习数据科学或者机器学习。

Python特性

► NumPy的优点：

- 对于同样的数值计算任务，使用NumPy要比直接编写Python代码便捷得多，是基于向量化的运算；
- NumPy中的数组的存储效率和输入输出性能均远远优于Python中等价的基本数据结构，且其能够提升的性能是与数组中的元素成比例的；
- NumPy的大部分代码都是用C语言写的，其底层算法在设计时就有着优异的性能，这使得NumPy比纯Python代码高效得多

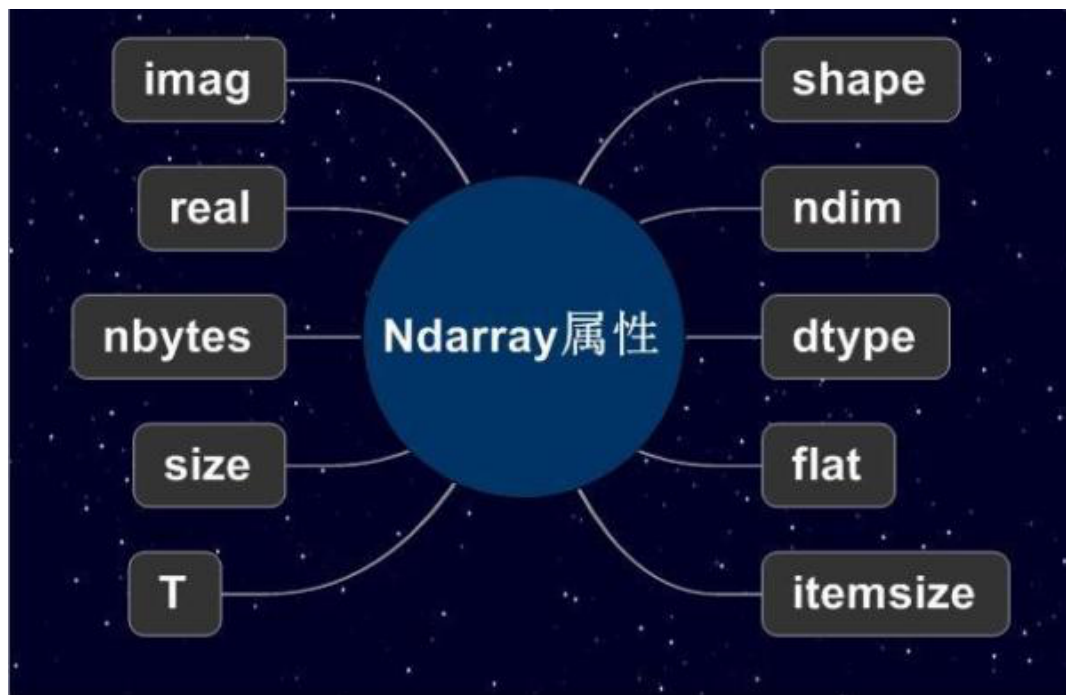
► NumPy的不足：

- 由于NumPy使用内存映射文件以达到最优的数据读写性能，而内存的大小限制了其对TB级大文件的处理；
- NumPy数组的通用性不及Python提供的list容器。因此，在科学计算之外的领域，NumPy的优势也就不那么明显。

numpy的许多函数不仅是用C实现了，还使用了BLAS（一般Windows下link到MKL的，Linux下link到OpenBLAS）。基本上那些BLAS实现在每种操作上都进行了高度优化，例如使用AVX向量指令集，像求平均值这种vector operation，很容易使用multi-threading或者vectorization来加速。

数组ndarray (N-dimensional Array)

- ▶ NumPy最重要的一个特点就是其N维数组对象（即**ndarray**），该对象是一个快速而灵活的大数据集容器，该对象由两部分组成：
 - 实际的数据；
 - 描述这些数据的元数据；



▶ 常用ndarray属性：

- **dtype** 描述数组元素的类型
- **shape** 以**tuple**表示的数组形状
- **ndim** 数组的维度
- **size** 数组中元素的个数
- **itemsizes** 数组中的元素在内存所占字节数
- **T** 数组的转置
- **flat** 返回一个数组的迭代器，对**flat**赋值将导致整个数组的元素被覆盖
- **real/imag** 给出复数数组的实部/虚部
- **nbytes** 数组占用的存储空间

以 0 下标为开始进行集合中元素的索引

NumPy Narray 对象

► **ndarray** 内部由以下内容组成：

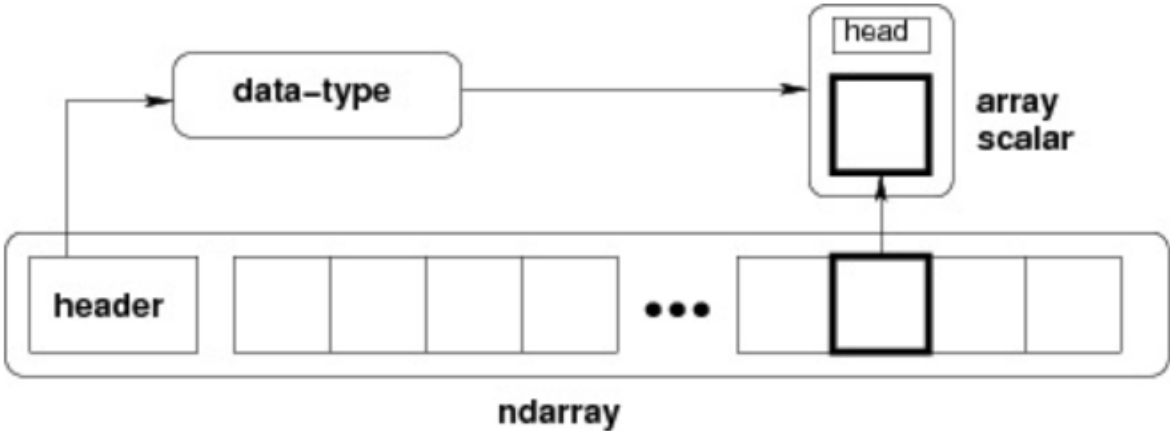
- 一个指向数据（内存或内存映射文件中的一块数据）的指针。
- 数据类型或 **dtype**，描述在数组中的固定大小值的格子。
- 一个表示数组形状（**shape**）的元组，表示各维度大小的元组。
- 一个跨度元组（**stride**），其中的整数指的是为了前进到当前维度下一个元素需要"跨过"的字节数。

参数说明：

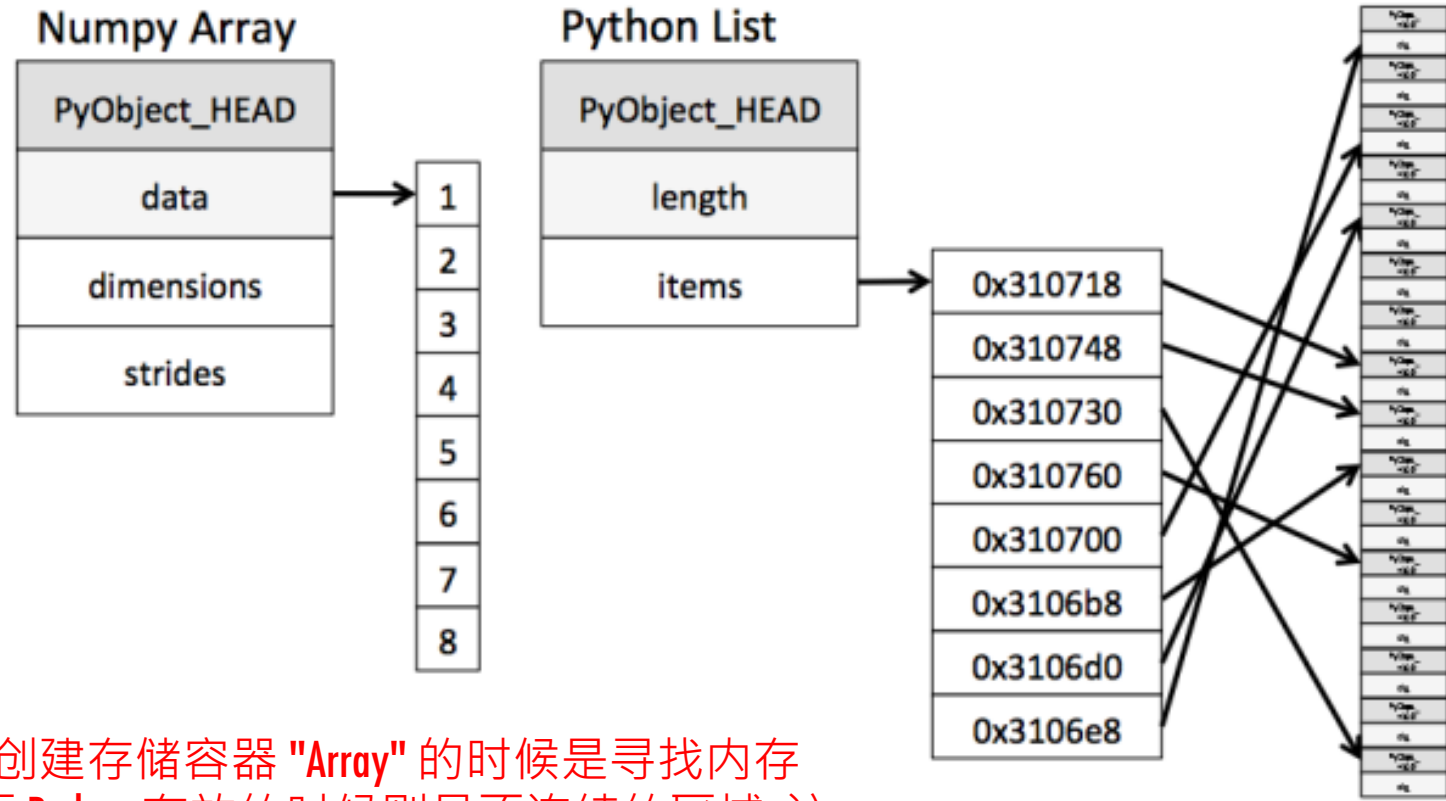
名称	描述
object	数组或嵌套的数列
dtype	数组元素的数据类型，可选
copy	对象是否需要复制，可选
order	创建数组的样式，C为行方向，F为列方向，A为任意方向（默认）
subok	默认返回一个与基类类型一致的数组
ndmin	指定生成数组的最小维度

```
numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)
```

ndarray 的内部结构：



创建Numpy Array



其实 **Numpy** 就是 **C** 的逻辑, 创建存储容器 "**Array**" 的时候是寻找内存上的一连串区域来存放, 而 **Python** 存放的时候则是不连续的区域, 这使得 **Python** 在索引这个容器里的数据时不是那么有效率. **Numpy** 只需要再这块固定的连续区域前后走走就能不费吹灰之力拿到数据.

NumPy数据类型

- ▶ **numpy** 支持的数据类型比 **Python** 内置的类型要多很多，基本上可以和 **C** 语言的数据类型对应上，其中部分类型对应为 **Python** 内置的类型。
- ▶ **dtype**数据类型对象是用来描述与数组对应的内存区域如何使用
 - **object** - 要转换为的数据类型对象
 - **align** - 如果为 **true**，填充字段使其类似 **C** 的结构体。
 - **copy** - 复制 **dtype** 对象，如果为 **false**，则是对内置数据类型对象的引用

```
numpy.dtype(object, align, copy)
```

数据类型	说明
bool	布尔类型，True或False，占用1比特
inti	其长度取决于平台的整数，一般是int32或int64
int8	字节长度的整数，取值：[-128, 127]
int16	16位长度的整数，取值：[-32768, 32767]
int32	32位长度的整数，取值：[-2 ³¹ , 2 ³¹ - 1]
int64	64位长度的整数，取值：[-2 ⁶³ , 2 ⁶³ - 1]
uint8	8位无符号整数，取值：[0, 255]
uint16	16位无符号整数，取值：[0, 65535]
uint32	32位无符号整数，取值：[0, 2 ³² - 1]
uint64	32位无符号整数，取值：[0, 2 ⁶⁴ - 1]
float16	16位半精度浮点数：1位符号位，5位指数，10位尾数
float32	32位半精度浮点数：1位符号位，8位指数，23位尾数
float64或float	双精度浮点数：1位符号位，11位指数，52位尾数
complex64	复数类型，实部和虚部都是32位浮点数
complex128或complex	复数类型，实部和虚部都是64位浮点数

NumPy 创建数组

- ▶ **numpy.empty** 创建一个指定形状 (**shape**) 、数据类型 (**dtype**) 且未初始化的数组：

```
numpy.empty(shape, dtype = float, order = 'C')
```

- ▶ **numpy.zeros** 创建指定大小的数组，数组元素以 0 来填充：

```
numpy.zeros(shape, dtype = float, order = 'C')
```

- ▶ **numpy.ones** 创建指定形状的数组，数组元素以 1 来填充：

```
numpy.ones(shape, dtype = None, order = 'C')
```

NumPy 从数值范围创建数组

- ▶ **numpy.arange** 使用 **arange** 函数创建数值范围并返回 **ndarray** 对象

```
numpy.arange(start, stop, step, dtype)
```

- ▶ **numpy.linspace** 函数用于创建一个一维数组，数组是一个等差数列构成的

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

- ▶ **numpy.logspace** 函数用于创建一个等比数列

```
np.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
```

NumPy 切片和索引

- ▶ **ndarray**对象的内容可以通过索引或切片来访问和修改，与 **Python** 中 **list** 的切片操作一样。
- ▶ **ndarray** 数组可以基于 **0 - n** 的下标进行索引，切片对象可以通过内置的 **slice** 函数和冒号分割切片参数，并设置 **start**, **stop** 及 **step** 参数进行，从原数组中切割出一个新数组。
 - **slice**函数方法: **slice(3,7,2)**
 - 冒号分隔切片参数 **start:stop:step** 来进行切片操作
 - 切片还可以包括省略号 **...**，来使选择元组的长度与数组的维度相同

冒号 **:** 的解释：如果只放置一个参数，如 **[2]**，将返回与该索引相对应的单个元素。如果为 **[2:]**，表示从该索引开始以后的所有项都将被提取。如果使用了两个参数，如 **[2:7]**，那么则提取两个索引(不包括停止索引)之间的项。

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

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])
```

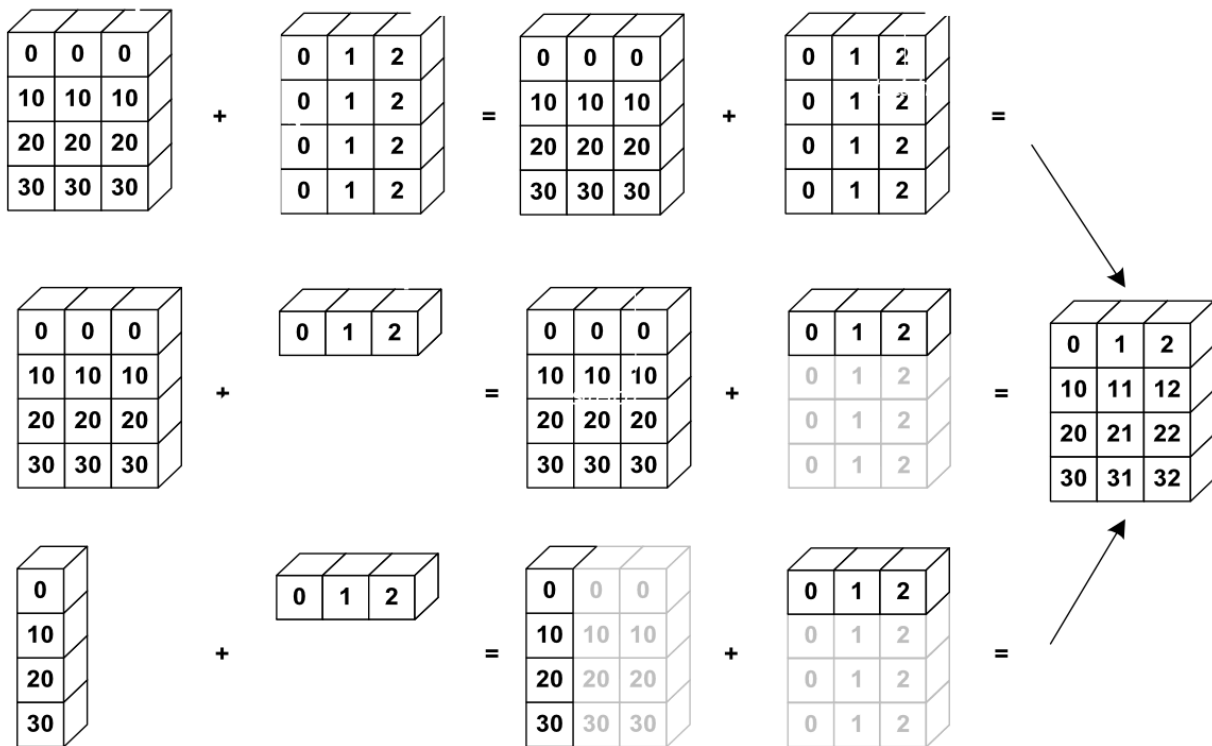
```
>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])
```

```
>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

数组广播

- ▶ 当数组跟一个标量进行数学运算时，标量需要根据数组的形状进行扩展，然后执行运算。
- ▶ 如果两个数组 **a** 和 **b** 形状相同，即满足 **a.shape == b.shape**，那么 **a*b** 的结果就是 **a** 与 **b** 数组对应位相乘。这要求维数相同，且各维度的长度相同。
- ▶ 这个扩展的过程称为“广播 (broadcasting)”



```
b
array([[ 0,  1, 20],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

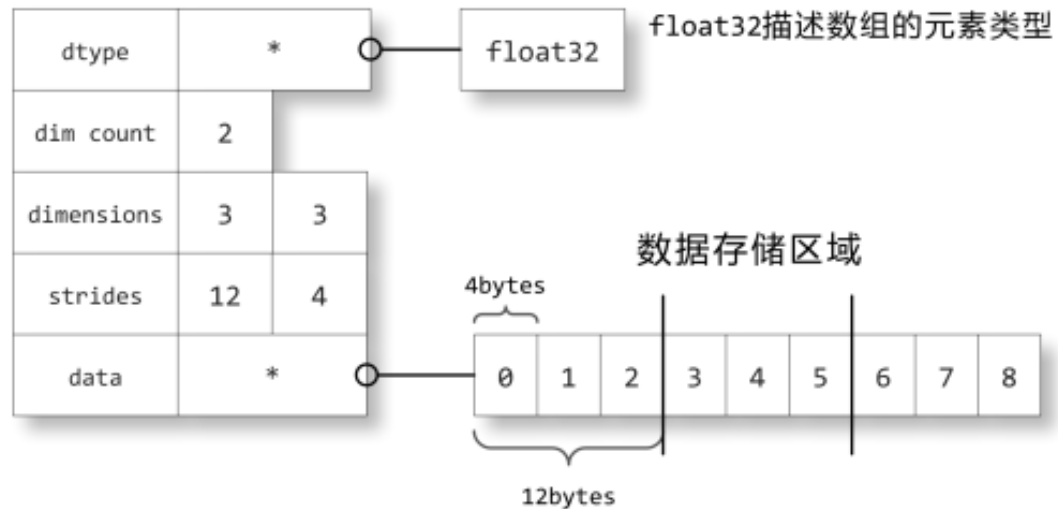
d = b + 2

d
array([[ 2,  3, 22],
       [ 5,  6,  7],
       [ 8,  9, 10],
       [11, 12, 13]])
```

内存结构

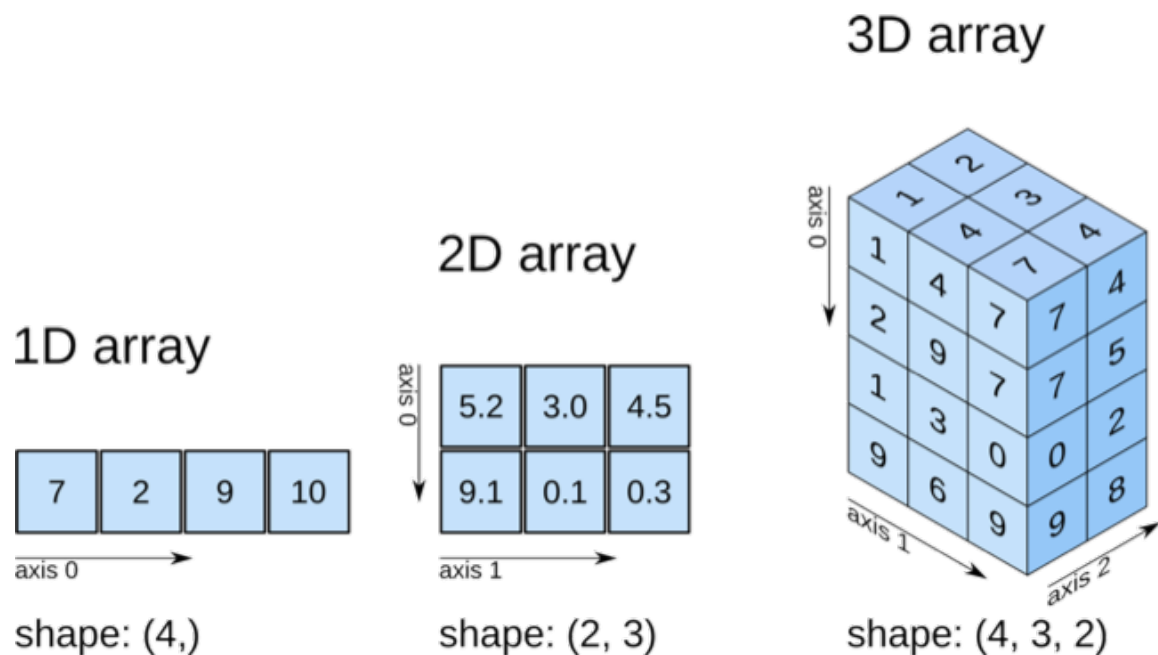
```
>>> a = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32)
```

ndarray数据结构



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

Array 数据存放



在我们看来的 **2D Array**, 如果追溯到计算机内存里, 它其实是储存在一个连续空间上的. 而对于这个连续空间, 我们如果创建 **Array** 的方式不同, 在这个连续空间上的排列顺序也有不同.

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

`arr.reshape((4, 3), order=?)`

C order (row major)

0	1	2
3	4	5
6	7	8
9	10	11

`order='C'`

Fortran order (column major)

0	4	8
1	5	9
2	6	10
3	7	11

`order='F'`

```
col_major = np.zeros((10,10), order='C') # C-type
row_major = np.zeros((10,10), order='F') # Fortran
```

Array 数据存放

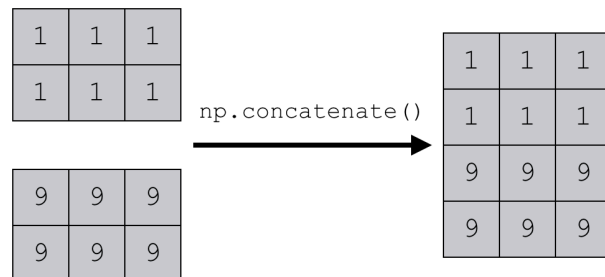
```
import numpy as np
import time

a = np.zeros((200, 200), order='C')
b = np.zeros((200, 200), order='F')
N=9999

def f(x, N):
    for _ in range(N):
        np.concatenate((x,x), axis=0)

t0 = time.time()
f(a, N)
t1 = time.time()
f(b, N)
t2 = time.time()

print((t1-t0)/N)
print((t2-t1)/N)
```



How the array is represented in Numpy

Row Major
Order (C)
(default in Numpy)



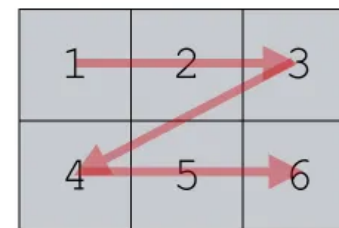
Column Major
Order (Fortran)



How the array is stored in memory



'C' flattens in row-first order (C-style)



'F' flattens in column-first order (Fortran-style)

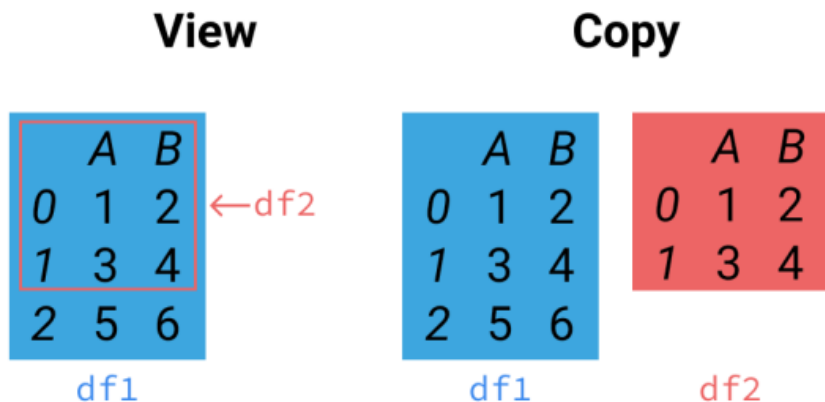


常用ndarray方法

- `reshape(...)` 返回一个给定shape的数组的副本
- `resize(...)` 返回给定shape的数组，原数组shape发生改变
- `flatten()/ravel()` 返回展平数组，原数组不改变
- `astype(dtype)` 返回指定元素类型的数组副本
- `fill()` 将数组元素全部设定为一个标量值
- `sum/Prod()` 计算所有数组元素的和/积
- `mean()/var()/std()` 返回数组元素的均值/方差/标准差
- `max()/min()/ptp()/median()` 返回数组元素的最大值/最小值/取值范围/中位数
- `argmax()/argmin()` 返回最大值/最小值的索引
- `sort()` 对数组进行排序，axis指定排序的轴；kind指定排序算法，默认是快速排序
- `view()/copy()` view创建一个新的数组对象指向同一数据；copy是深复制
- `tolist()` 将数组完全转为列表，注意与直接使用list(array)的区别
- `compress()` 返回满足条件的元素构成的数组

返回拷贝(copy)和返回视图(view)

- 在 Numpy 中, 有两个很重要的概念, **copy** 和 **view**.
- copy** 会将数据 **copy** 出来存放在内存中另一个地方, 而 **view** 则是不 **copy** 数据, 直接取源数据的索引部分.



```
a = np.arange(1, 7).reshape((3,2))
a_view = a[:2]
a_copy = a[:2].copy()
```

```
a_copy[1,1] = 0
print(a)
"""
[[1 2]
 [3 4]
 [5 6]]
"""
```

```
a_view[1,1] = 0
print(a)
"""
[[1 2]
 [3 0]
 [5 6]]
"""
```

a_view 的东西全部都是 **a** 的东西, 动 **a_view** 的任何地方, **a** 都会被动到, 因为他们在内存中的位置是一模一样的, 本质上就是自己. 而 **a_copy** 则是将 **a** **copy** 了一份, 然后把 **a_copy** 放在内存中的另外的地方, 这样改变 **a_copy**, **a** 是不会被改变的.

copy慢 view快

- `a*=2` 就是将这个 **view** 给赋值了, 和 `a[:] *= 2` 一个意思, 从头到尾没有创建新的东西.
- 而 `b = 2*b` 中, 我们将 **b** 赋值给另外一个新建的 **b**.

```
a = np.zeros((1000, 1000))
b = np.zeros((1000, 1000))
N = 9999

def f1(a):
    for _ in range(N):
        a *= 2          # same as a[:] *= 2

def f2(b):
    for _ in range(N):
        b = 2*b

print('%f' % ((t1-t0)/N))    # f1: 0.000837
print('%f' % ((t2-t1)/N))    # f2: 0.001346
```

选择数据

► view 的方式

```
a_view1 = a[1:2, 3:6]    # 切片 slice  
a_view2 = a[:100]        # 同上  
a_view3 = a[:, :2]       # 跳步
```

► copy 的方式

```
a_copy1 = a[[1,4,6], [2,4,6]] # 用 index 选  
a_copy2 = a[[True, True], [False, True]] # 用 mask  
a_copy3 = a[[1,2], :]        # 虽然 1,2 的确连在一起了, 但是他们确实是 copy  
a_copy4 = a[a[1,:] != 0, :]   # fancy indexing  
a_copy5 = a[np.isnan(a), :]  # fancy indexing
```

fatten和ravel

```
: # flatten() 返回的是拷贝, 不影响原始数组  
# 即数组 "b" 没有发生变化  
b.flatten()[2]=20  
b
```

```
: array([[ 0,  1,  2],  
        [ 3,  4,  5],  
        [ 6,  7,  8],  
        [ 9, 10, 11]])
```

```
: # ravel() 返回的是视图, 会影响原始数组  
# 即数组 "b" 会发生变化  
b.ravel()[2]=20  
b
```

```
: array([[ 0,  1, 20],  
        [ 3,  4,  5],  
        [ 6,  7,  8],  
        [ 9, 10, 11]])
```

```
def f1(a):  
    for _ in range(N):  
        a.flatten()
```

```
def f2(b):  
    for _ in range(N):  
        b.ravel()
```

```
print('%f' % ((t1-t0)/N))      # 0.001059  
print('%f' % ((t2-t1)/N))      # 0.000000
```

flatten() 返回一份拷贝, 需要分配新的内存空间, 对拷贝所做的修改不会影响原始矩阵, 而 **ravel()** 返回的是视图 (**view**), 会影响原始矩阵。

Numpy算数函数

- ▶ NumPy 算术函数包含简单的加减乘除: **add()** , **subtract()** , **multiply()** 和 **divide()** 。
- ▶ 需要注意的是数组必须具有相同的形状或符合数组广播规则。

```
import numpy as np

a = np.arange(9, dtype = np.float_).reshape(3,3)
print ('第一个数组: ')
print (a)
print ('\n')
print ('第二个数组: ')
b = np.array([10,10,10])
print (b)
print ('\n')
print ('两个数组相加: ')
print (np.add(a,b))
print ('\n')
print ('两个数组相减: ')
print (np.subtract(a,b))
print ('\n')
print ('两个数组相乘: ')
print (np.multiply(a,b))
print ('\n')
print ('两个数组相除: ')
print (np.divide(a,b))
```

第一个数组:

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

第二个数组:

```
[10 10 10]
```

两个数组相加:

```
[[10. 11. 12.]
 [13. 14. 15.]
 [16. 17. 18.]]
```

两个数组相减:

```
[[ -10.  -9.  -8.]
 [  -7.  -6.  -5.]
 [  -4.  -3.  -2.]]
```

两个数组相乘:

```
[[ 0. 10. 20.]
 [30. 40. 50.]
 [60. 70. 80.]]
```

两个数组相除:

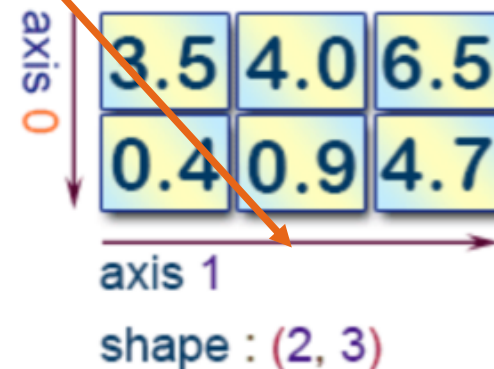
```
[[0.  0.1 0.2]
 [0.3 0.4 0.5]
 [0.6 0.7 0.8]]
```

numpy常用统计函数

- `np.sum()`，返回求和
- `np.mean()`，返回均值
- `np.max()`，返回最大值
- `np.min()`，返回最小值
- `np.ptp()`，数组沿指定轴返回最大值减去最小值，即 (max-min)
- `np.std()`，返回标准偏差 (standard deviation)
- `np.var()`，返回方差 (variance)
- `np.cumsum()`，返回累加值
- `np.cumprod()`，返回累乘积值

请注意函数在使用时需要指定axis轴的方向，若不指定，默认统计整个数组。

```
b
array([[ 0,  1, 20,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
np.max(b)
20
# 沿axis=1轴方向统计
np.max(b,axis=1)
array([20, 11])
# 沿axis=0轴方向统计
np.max(b,axis=0)
array([ 6,  7, 20,  9, 10, 11])
```



numpy.linalg 线性代数

```
inv(matrix)
diag(array/matrix)
trace(matrix)
det(matrix)
eig(matrix)
svd(matrix)
solve(matrix, array)
```

求逆

将array转换为对角方阵, 或者将matrix对角元素

求迹

求行列式

求方阵的本征值和本征向量

SVD分解

解matrix * x= array 的解

```
>>> np.linalg.
np.linalg.LinAlgError(      np.linalg.eigh(
np.linalg.absolute_import  np.linalg.eigvals(
np.linalg.bench(           np.linalg.eigvalsh(
np.linalg.cholesky(        np.linalg.info
np.linalg.cond(            np.linalg.inv(
np.linalg.det(             np.linalg.lapack_lite
np.linalg.division         np.linalg.linalg
np.linalg.eig(             np.linalg.lstsq(
np.linalg.eigvals(         np.linalg.lu(
np.linalg.eigvalsh(        np.linalg.lu_solve(
np.linalg.eigh(            np.linalg.matrix_power(
np.linalg.eigvals(         np.linalg.matrix_rank(
np.linalg.eigvalsh(        np.linalg.multi_dot(
np.linalg.info             np.linalg.norm(
np.linalg.inv(             np.linalg.pinv(
np.linalg.lapack_lite      np.linalg.print_function
np.linalg.linalg           np.linalg.qr(
np.linalg.lstsq(           np.linalg.slogdet(
np.linalg.lu(              np.linalg.solve(
np.linalg.lu_solve(        np.linalg.svd(
np.linalg.matrix_power(    np.linalg.tensorinv(
np.linalg.matrix_rank(    np.linalg.tensorsolve(
np.linalg.multi_dot(       np.linalg.test(
np.linalg.norm(
np.linalg.pinv(
np.linalg.print_function
np.linalg.qr(
np.linalg.slogdet(
```


NumPy的random子库

► NumPy的random子库

- `np.random.*`
- `np.random.rand()`
- `np.random.randn()`
- `np.random.randint()`

函数	说明
<code>rand(d0,d1,...,dn)</code>	根据d0-dn创建随机数数组，浮点数， $[0,1)$ ，均匀分布
<code>randn(d0,d1,...,dn)</code>	根据d0-dn创建随机数数组，标准正态分布
<code>randint(low[,high,shape])</code>	根据shape创建随机整数或整数数组，范围是 $[low, high)$
<code>seed(s)</code>	随机数种子，s是给定的种子值
函数	说明
<code>shuffle(a)</code>	根据数组a的第1轴进行随排列，改变数组x
<code>permutation(a)</code>	根据数组a的第1轴产生一个新的乱序数组，不改变数组x
<code>choice(a[,size,replace,p])</code>	从一维数组a中以概率p抽取元素，形成size形状新数组 replace表示是否可以重用元素，默认为False
函数	说明
<code>uniform(low,high,size)</code>	产生具有均匀分布的数组，low起始值，high结束值，size形状
<code>normal(loc,scale,size)</code>	产生具有正态分布的数组，loc均值，scale标准差，size形状
<code>poisson(lam,size)</code>	产生具有泊松分布的数组，lam随机事件发生率，size形状

NumPy 文件存取

- CSV文件: CSV (Comma-Separated Value, 逗号分隔值)

文件保存:

`np.savetxt(frame, array, fmt='%.18e', delimiter=None)`

- **frame** : 文件、字符串或产生器，可以是.gz或.bz2的压缩文件
- **array** : 存入文件的数组
- **fmt** : 写入文件的格式，例如：`%d %2f %.18e`
- **delimiter** : 分割字符串，默认是任何空格

```
a=np.arange(100).reshape(5,20)
```

```
np.savetxt('a.csv',a,fmt='%d',delimiter=',')
```

生成文件

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39
40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59
60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79
80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99
```

NumPy 文件读取

- CSV文件: CSV (Comma-Separated Value, 逗号分隔值)

文件读取:

`np.loadtxt(frame, dtype=np.float, delimiter=None, unpack=False)`

- **frame** : 文件、字符串或产生器，可以是.gz或.bz2的压缩文件
- **dtype** : 数据类型，可选
- **delimiter** : 分割字符串，默认是任何空格
- **unpack** : 如果True，读入属性将分别写入不同变量

CSV只能有效存储一维和二维数组

`np.savetxt()` `np.loadtxt()` 只能有效存取一维和二维数组

```
a = np.loadtxt('s.csv', dtype=np.float, delimiter=',', unpack=False)
print(a)
```

```
[[ 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. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37.
 38. 39.]
 [40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57.
 58. 59.]
 [60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77.
 78. 79.]
 [80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97.
 98. 99.]]
```

NumPy 便捷文件存取

`np.save(fname, array)` 或 `np.savez(fname, array)`

- **fname** : 文件名, 以`.npy`为扩展名, 压缩扩展名为`.npz`
- **array** : 数组变量

`np.load(fname)`

- **fname** : 文件名, 以`.npy`为扩展名, 压缩扩展名为`.npz`

```
a=np.arange(100).reshape(5,10,2)
```

```
np.save('a.npy',a)
```

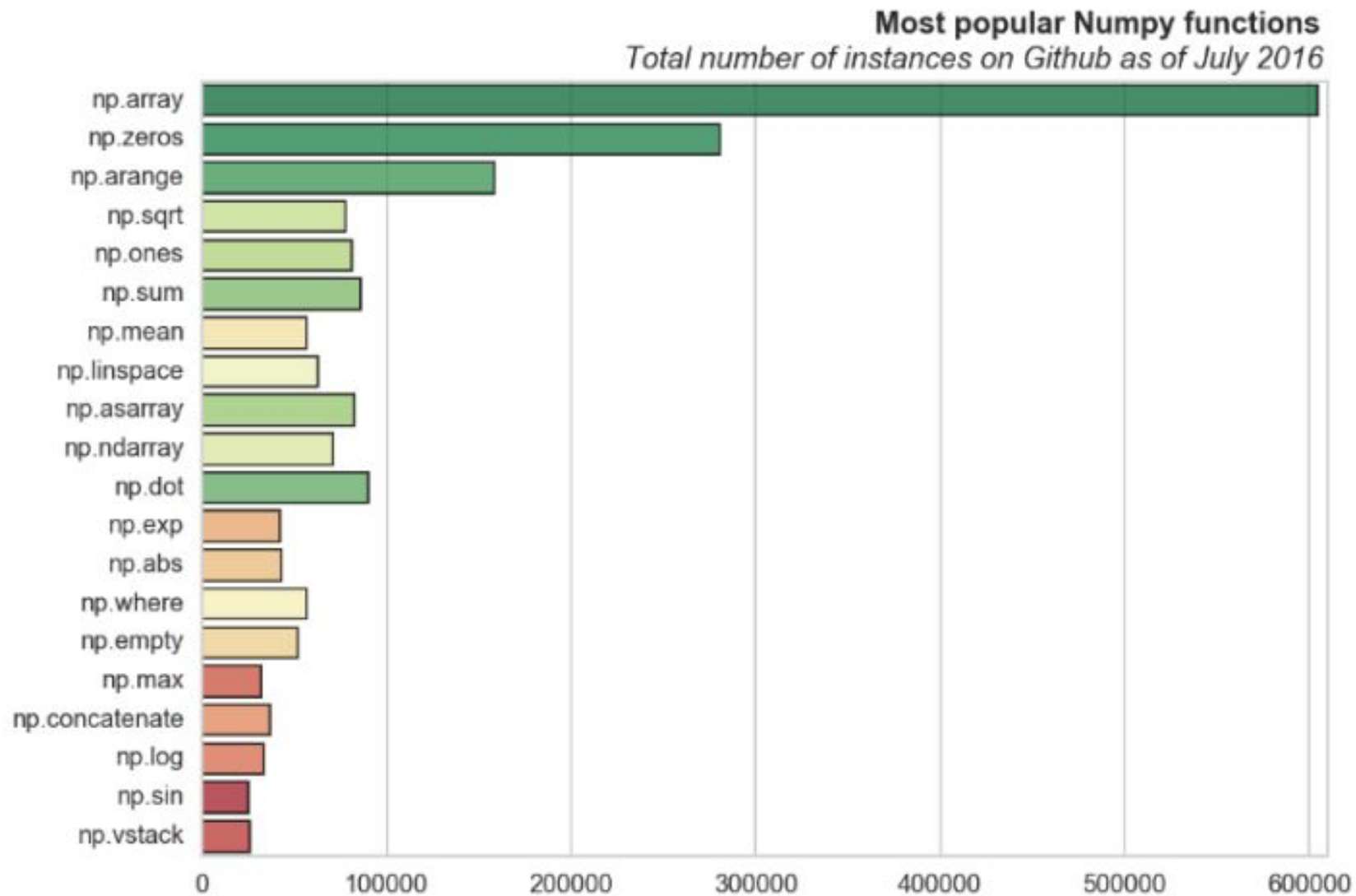
```
b=np.load('a.npy')
```

```
b
```

```
Out[80]:
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        ...,
        [14, 15],
        [16, 17],
        [18, 19]],
```

NumPy最常用函数



SciPy-Python 科学算法库

- ▶ **SciPy**库提供了大量有用的函数和类，用来解决各种专业领域的问题。
- ▶ **SciPy**框架建立于低一级的**NumPy**框架的多维数组之上，并且提供了大量的高级科学算法。
- ▶ **SciPy** 包含的模块有最优化、线性代数、积分、插值、特殊函数、快速傅里叶变换、信号处理和图像处理、常微分方程求解和其他科学与工程中常用的计算。

模块名	功能
scipy.cluster	向量量化
scipy.constants	数学常量
scipy.fftpack	快速傅里叶变换
scipy.integrate	积分
scipy.interpolate	插值
scipy.io	数据输入输出
scipy.linalg	线性代数
scipy.ndimage	N维图像
scipy.odr	正交距离回归
scipy.optimize	优化算法
scipy.signal	信号处理
scipy.sparse	稀疏矩阵
scipy.spatial	空间数据结构和算法
scipy.special	特殊数学函数
scipy.stats	统计函数

文件输入和输出：scipy.io

- 这个模块可以加载和保存matlab文件：

```
1 >>> from scipy import io as spio
2 >>> a = np.ones((3, 3))
3 >>> spio.savemat('file.mat', {'a': a}) # 保存字典到file.mat
4 >>> data = spio.loadmat('file.mat', struct_as_record=True)
5 >>> data['a']
6 array([[ 1.,  1.,  1.],
7         [ 1.,  1.,  1.],
8         [ 1.,  1.,  1.]])
```

线性代数操作：scipy.linalg

```
1 >>> from scipy import linalg
2 >>> arr = np.array([[1, 2],
3 ...                [3, 4]])
4 >>> linalg.det(arr)
5 -2.0
```

```
1 >>> arr = np.array([[1, 2],
2 ...                [3, 4]])
3 >>> iarr = linalg.inv(arr)
4 >>> iarr
5 array([[-2. ,  1. ],
6        [ 1.5, -0.5]])
```

统计模块操作：scipy.stats

► Stats提供了产生连续性分布的函数：

- 均匀分布(uniform)
- 正态分布(norm)
- 贝塔分布(beta)
- 离散分布
- 伯努利分布(bernoulli)
- 几何分布(geom)
- 泊松分布(poisson)

```
from scipy import stats
import matplotlib.pyplot as plt

#产生符合正态分布的随机数
x = stats.norm.rvs(size = 20)
print(x)

#用频率分布直方图反映一组数据的分布情况
plt.hist(x) #数据点比较少时，不一定符合正太分布
plt.show()

x = stats.norm.rvs(size = 20000) #取多数据点进行尝试

plt.hist(x)
plt.show() #多数据情况下可看出符合正态分布

#产生服从beta分布的数组
y = stats.beta.rvs(size=100,a=0.5,b=0.5)

plt.hist(y)
plt.show()
```


优化器：scipy.optimize

```
from scipy import optimize

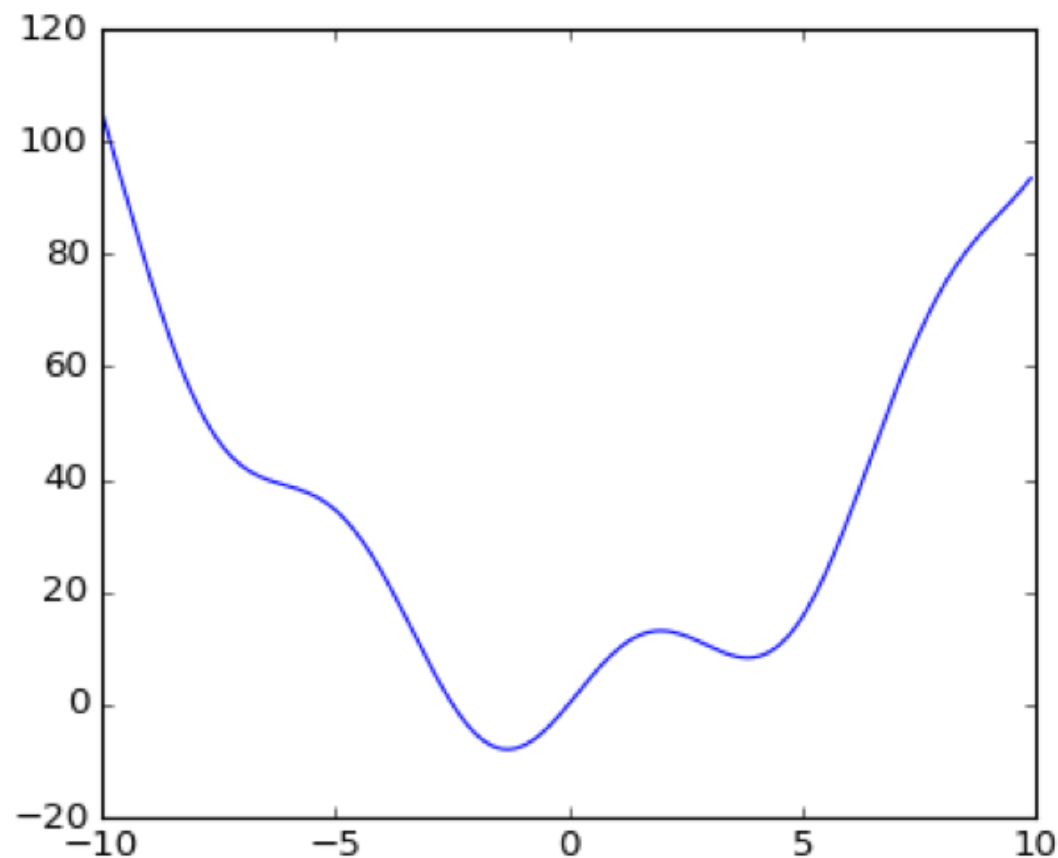
def f(x):
    return x**2 + 10*np.sin(x)

x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
plt.show()

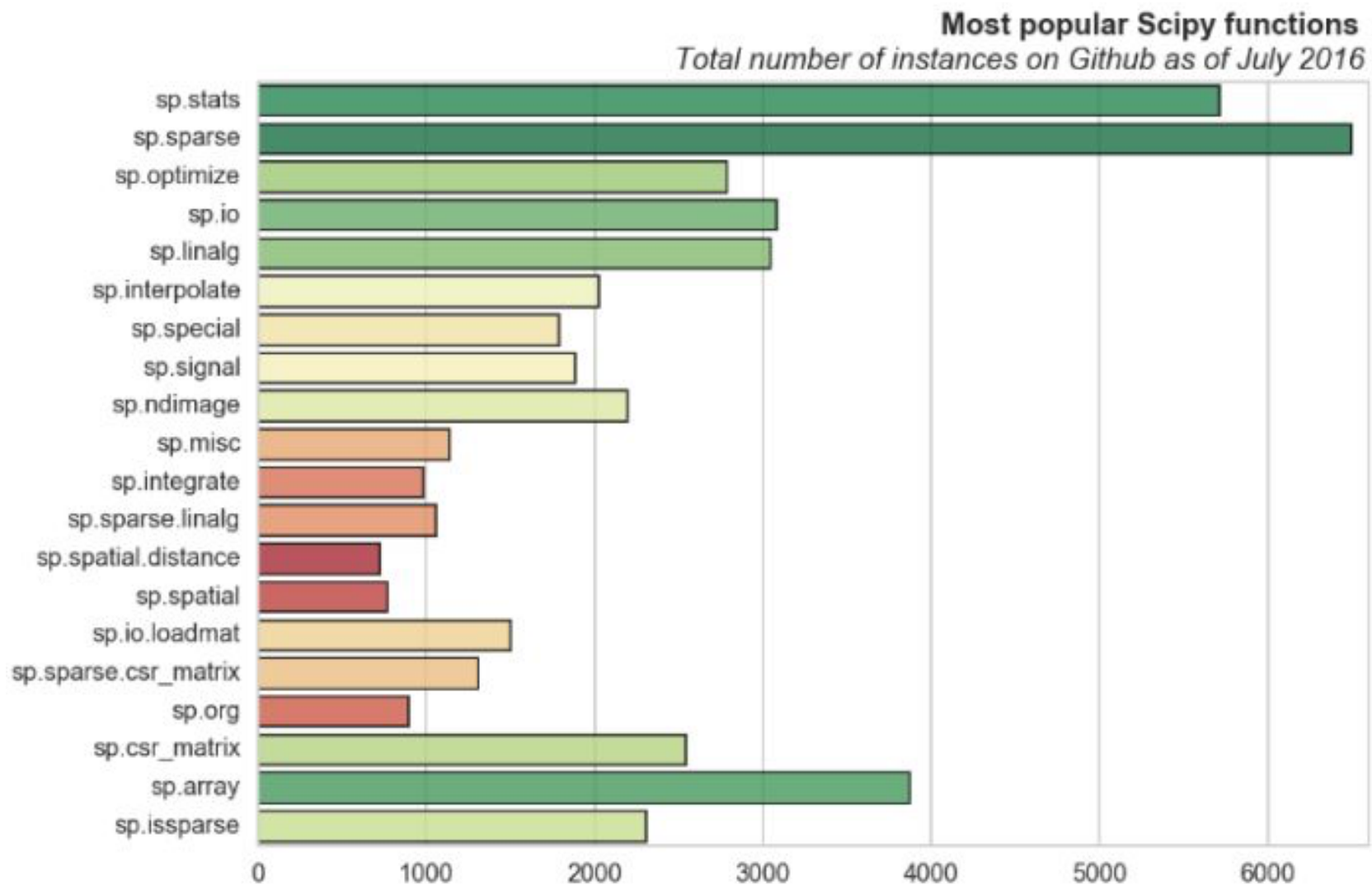
optimize.fmin_bfgs(f, 0)
```

要找出这个函数的最小值，也就是曲线的最低点。可以用BFGS优化算法(Broyden-Fletcher-Goldfarb-Shanno algorithm)：

```
1 >>> optimize.fmin_bfgs(f, 0)
2 Optimization terminated successfully.
3     Current function value: -7.945823
4     Iterations: 5
5     Function evaluations: 24
6     Gradient evaluations: 8
7 array([-1.30644003])
```



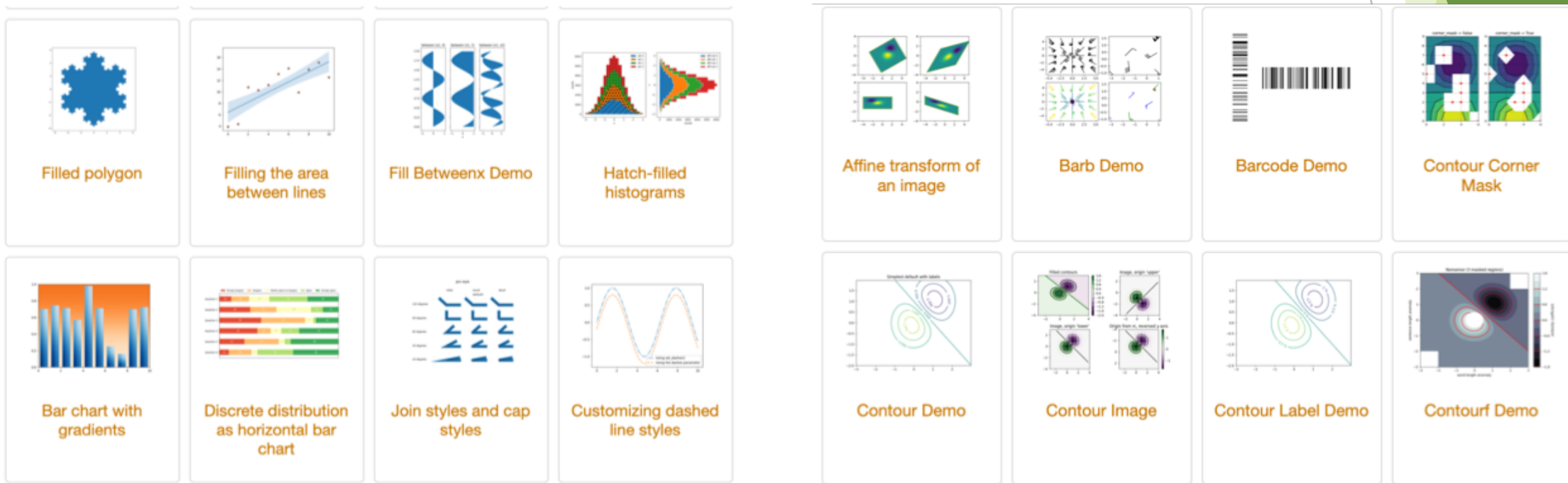
SciPy最常用函数



Matplotlib绘图可视化库



- **Matplotlib** 可能是 **Python 2D**-绘图领域使用最广泛的套件。它能让使用者很轻松地将数据图形化，并且提供多样化的输出格式。



<https://matplotlib.org/gallery/index.html>

Matplotlib线条相关属性标记设置

► 标记属性

标记maker	描述	标记	描述	
'o'	圆圈	'.'	点	
'D'	菱形	's'	正方形	
'h'	六边形1	'*'	星号	
'H'	六边形2	'd'	小菱形	
'_'	水平线	'v'	一角朝下的三角形	
'8'	八边形	'<'	一角朝左的三角形	
'p'	五边形	'>'	一角朝右的三角形	
','	像素	'^'	一角朝上的三角形	
'+'	加号	'\'	'	竖线
'None',' ',''	无	'x'	X	

颜色和线条属性

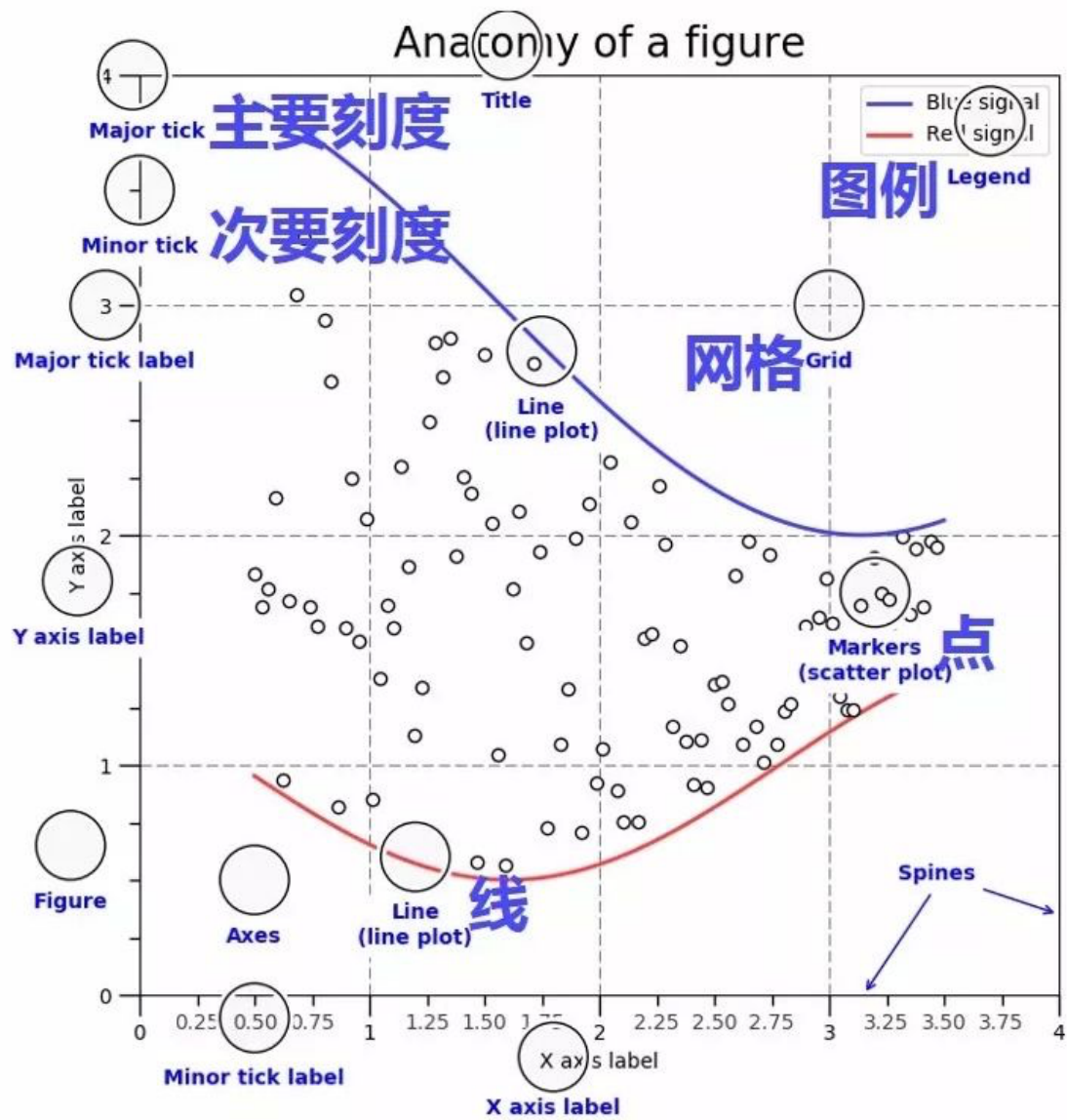
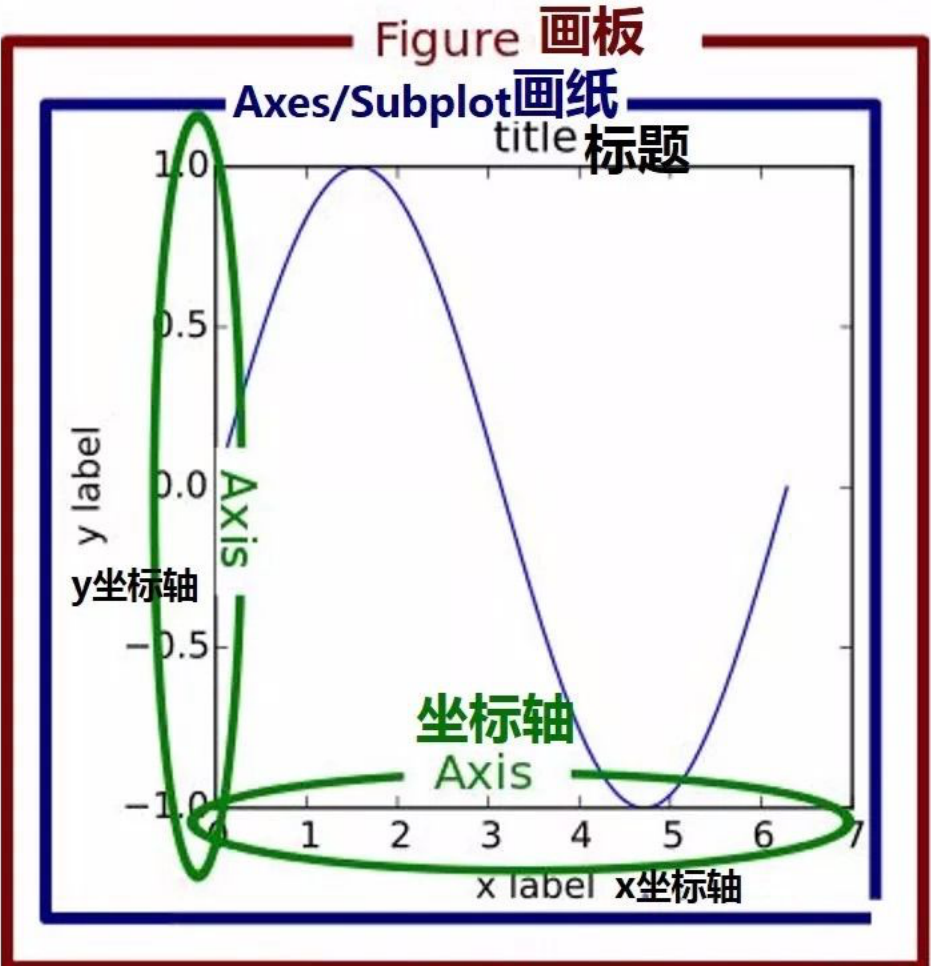
- 可以通过调用`matplotlib.pyplot.colors()`得到`matplotlib`支持的所有颜色

别名	颜色	别名	颜色
b	蓝色	g	绿色
r	红色	y	黄色
c	青色	k	黑色
m	洋红色	w	白色

- 表线条的属性

线条风格linestyle或ls	描述	线条风格linestyle或ls	描述
'-'	实线	'.'	虚线
'--'	破折线	'None',' '	什么都不画
'-.'	点划线		

Matplotlib图形基础



matplotlib示例1

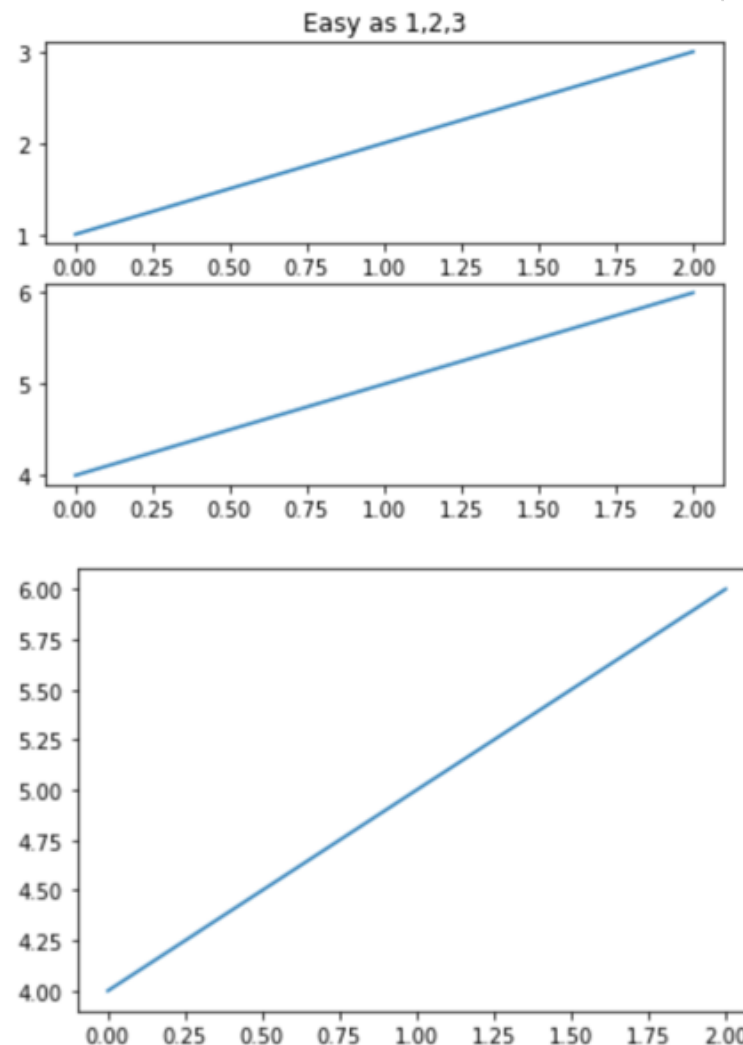
```
import matplotlib.pyplot as plt

plt.figure(1)
plt.subplot(211)
plt.plot([1,2,3])
plt.subplot(212)
plt.plot([4,5,6])

plt.figure(2)
plt.plot([4,5,6])

plt.figure(1)
plt.subplot(211)
plt.title('Easy as 1,2,3')

plt.show()
```



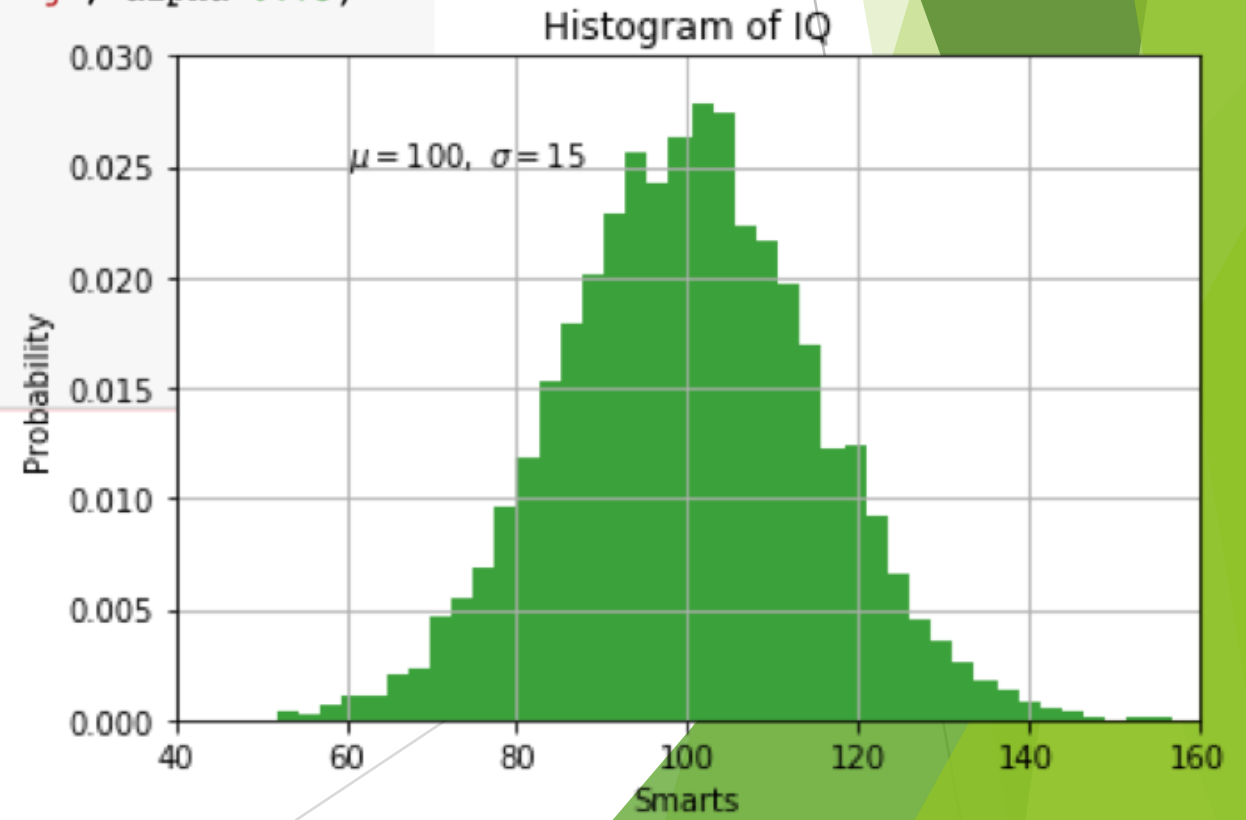
matplotlib示例2

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

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



matplotlib示例3

```
import matplotlib.pyplot as plt

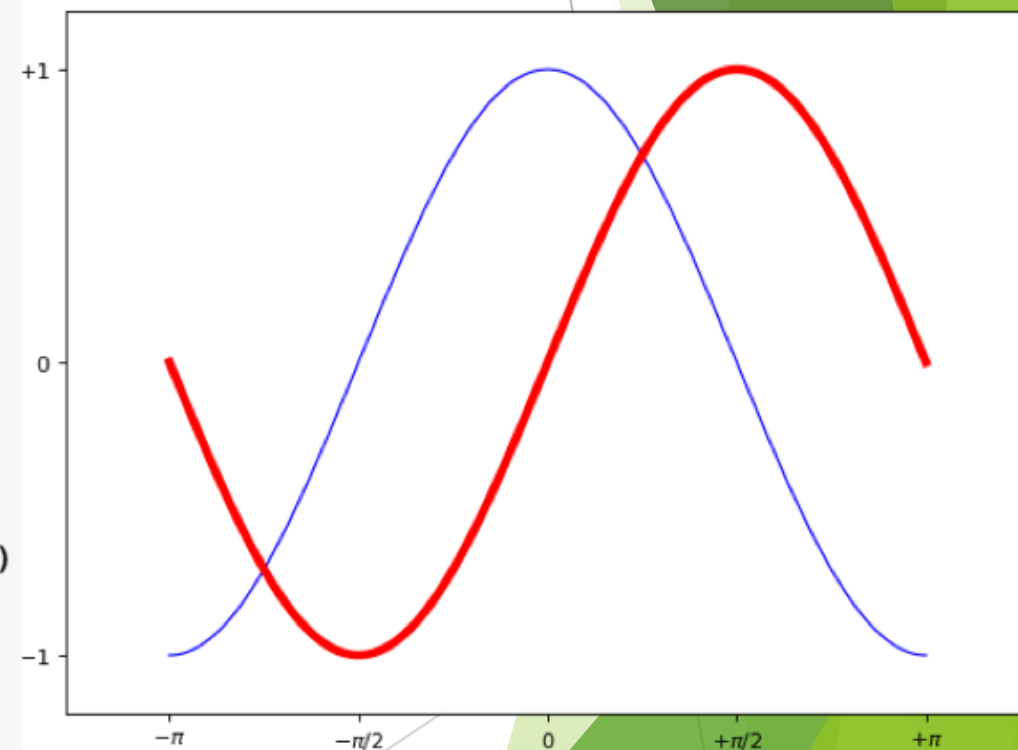
plt.figure(figsize=(8,6), dpi=80)
plt.subplot(1,1,1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C,S = np.cos(X), np.sin(X)

plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")
plt.plot(X, S, color="r", lw=4.0, linestyle="-")
plt.axis([-4,4,-1.2,1.2])

plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
           [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.yticks([-1, 0, +1], [r'$-1$', r'$0$', r'$+1$'])

plt.show()
```



matplotlib 示例pylib

```
import numpy as np
from pylab import *
```

```
figure(figsize=(8,6), dpi=80)
subplot(1,1,1)
```

```
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
```

```
plot(X, C, color="blue", linewidth=1.0, linestyle="-")
plot(X, S, color="r", lw=4.0, linestyle="-")
axis([-4,4,-1.2,1.2])
```

```
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
        [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
yticks([-1, 0, +1], [r'$-1$', r'$0$', r'$+1$'])
```

```
show()
```

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(8,6), dpi=80)
plt.subplot(1,1,1)
```

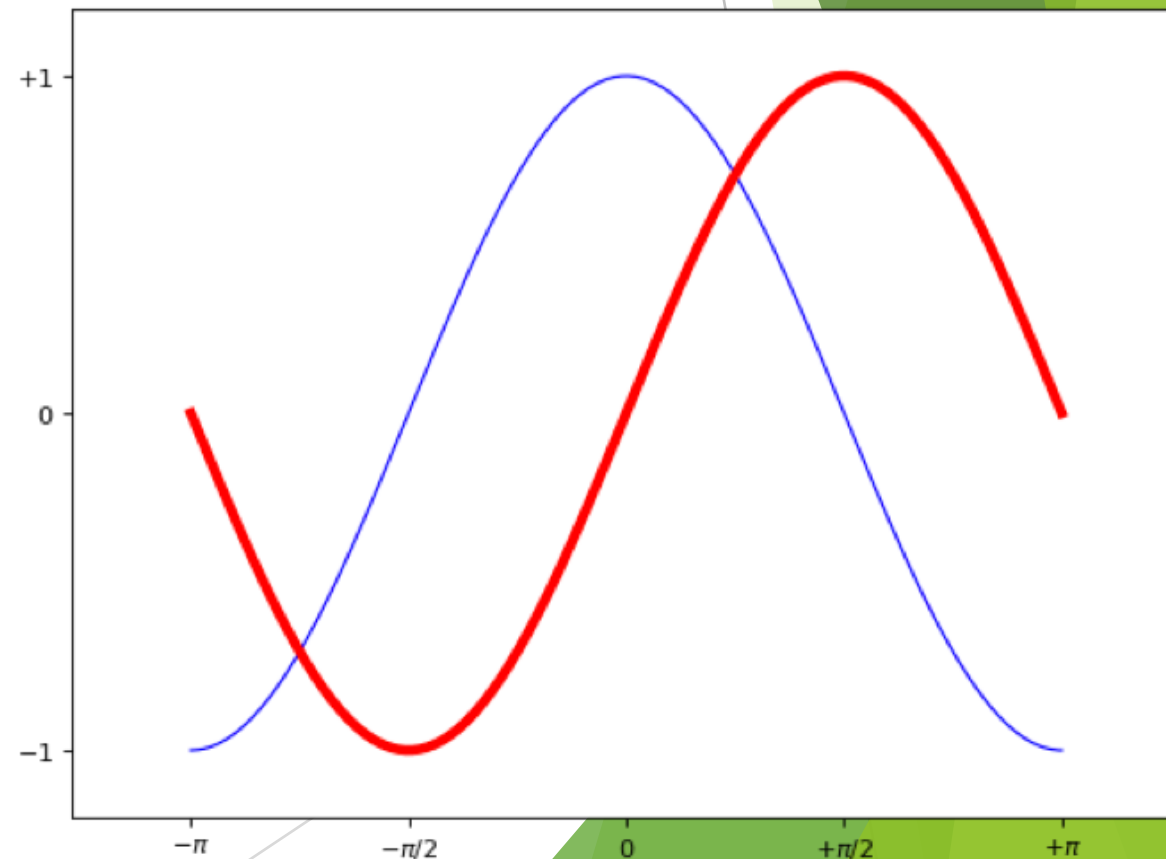
```
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
```

```
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")
plt.plot(X, S, color="r", lw=4.0, linestyle="-")
plt.axis([-4,4,-1.2,1.2])
```

```
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
            [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.yticks([-1, 0, +1], [r'$-1$', r'$0$', r'$+1$'])
```

```
plt.show()
```

pylab 是 **matplotlib** 面向对象绘图库的一个接口。它的语法和 **Matlab** 十分相近。也就是说，它主要的绘图命令和 **Matlab** 对应的命令有相似的参数



matplotlib用于Image图像

```
from pylab import *

def f(x,y): return (1-x/2+x**5+y**3)*np.exp(-x**2-y**2)

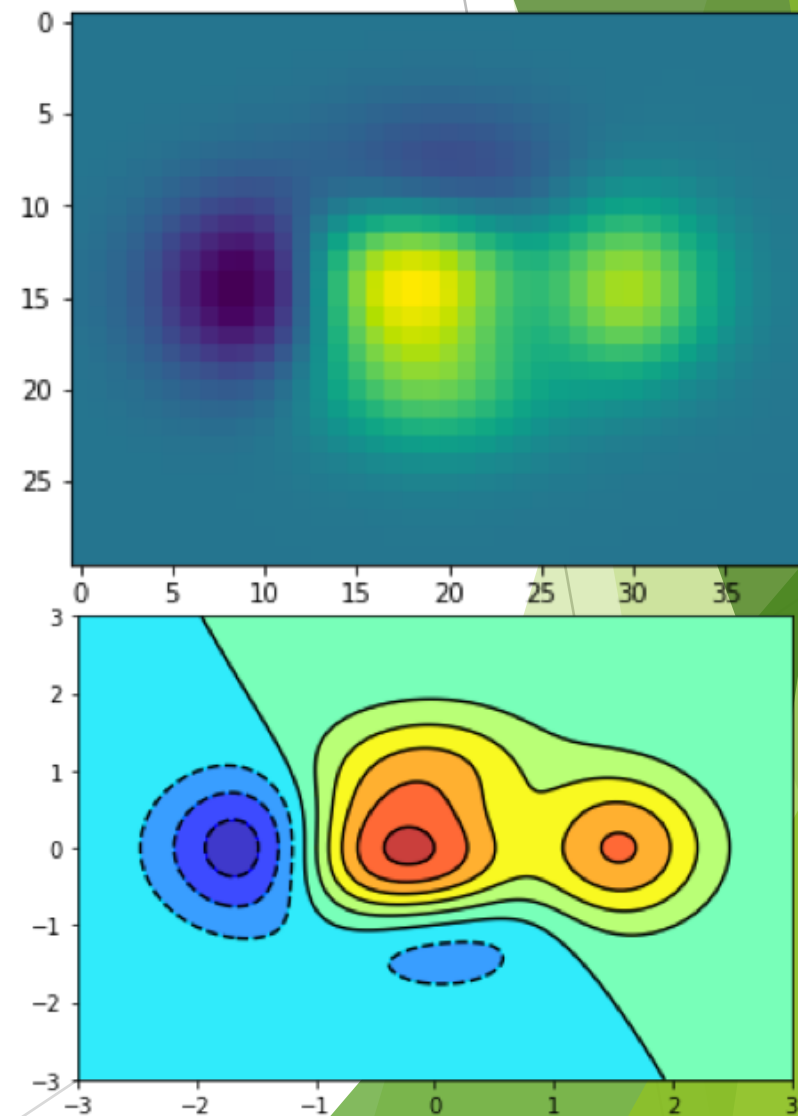
n = 10
x = np.linspace(-3,3,4*n)
y = np.linspace(-3,3,3*n)
X,Y = np.meshgrid(x,y)
imshow(f(X,Y)), show()
```

```
from pylab import *

def f(x,y): return (1-x/2+x**5+y**3)*np.exp(-x**2-y**2)

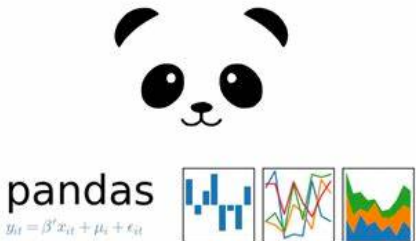
n = 256
x = np.linspace(-3,3,n)
y = np.linspace(-3,3,n)
X,Y = np.meshgrid(x,y)

contourf(X, Y, f(X,Y), 8, alpha=.75, cmap='jet')
C = contour(X, Y, f(X,Y), 8, colors='black', linewidth=.5)
show()
```



Pandas (Python Data Analysis Library)


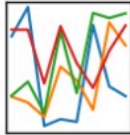

- ▶ 对于数据科学家，无论是数据分析还是数据挖掘来说，**Pandas**是一个非常重要的**Python**包。它不仅提供了很多方法，使得数据处理非常简单，同时在数据处理速度上也做了很多优化，使得和**Python**内置方法相比时有了很大的优势。
- ▶ **DataFrame**是**Pandas**中的一个表格型的数据结构，包含有一组有序的列，每列可以是不同的值类型(数值、字符串、布尔型等)，**DataFrame**既有行索引也有列索引，可以被看做是由**Series**组成的字典
- ▶ **Series**是一种类似于一维数组的对象，是由一组数据(各种**NumPy**数据类型)以及一组与之相关的数据标签(即索引)组成。仅由一组数据也可产生简单的**Series**对象。




Series			Series			DataFrame	
apples			oranges			apples	oranges
0	3	+	0	0	=	0	3
1	2		1	3		1	2
2	0		2	7		2	0
3	1		3	2		3	1


pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$





NumPy



python™

Pandas dtype	Python type	NumPy type	Usage
object	str	string_, unicode_	Text
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64	Integer numbers
float64	float	float_, float16, float32, float64	Floating point numbers
bool	bool	bool_	True/False values
datetime64	NA	NA	Date and time values
timedelta[ns]	NA	NA	Differences between two datetimes
category	NA	NA	Finite list of text values

Pandas示例

```
df = pd.read_csv('./uk_rain_2014.csv', header=0)
df.head(5)
df.tail(5)
# Finding out basic statistical information on your dataset. pd.options.display.float_format = '{:,.3f}'.format •df.de
df['Rain (mm) Oct-Sep']
df['Rain (mm) Oct-Sep'] < 1000
```

	Water Year	Rain (mm) Oct-Sep	Outflow (m3/s) Oct-Sep	Rain (mm) Dec-Feb	Outflow (m3/s) Dec-Feb	Rain (mm) Jun-Aug	Outflow (m3/s) Jun-Aug
0	1980/81	1182	5408	292	7248	174	2212
1	1981/82	1098	5112	257	7316	242	1936
2	1982/83	1156	5701	330	8567	124	1802
3	1983/84	993	4265	391	8905	141	1078
4	1984/85	1182					

	Water Year	Rain (mm) Oct-Sep	Outflow (m3/s) Oct-Sep	Rain (mm) Dec-Feb	Outflow (m3/s) Dec-Feb	Rain (mm) Jun-Aug	Outflow (m3/s) Jun-Aug
28	2008/09	1139	4941	268	6690	323	
29	2009/10	1103	4738	255	6435	244	
30	2010/11	1053	4521	265	6593	267	
31	2011/12	1285	5500	339	7630	379	
32	2012/13	1090	5329	350	9615	187	

Pandas数据结构

► Series (序列)

- Series是带有标签的一维数组，可以保存任何数据类型（整数，字符串，浮点数，Python对象等）。轴标签统称为索引。创建Series的基本方法是调用：

```
>>> s = pd.Series(data, index=index)
```

- data可以是许多不同的东西：Python dict（字典）、ndarray、标量值（如5）

► DataFrame (数据帧)

- DataFrame是带有标签的二维数据结构，列的类型可能不同。你可以把它想象成一个电子表格或SQL表，或者Series对象的字典。它一般是最常用的pandas对象。像Series一样，DataFrame接受许多不同类型的输入。

```
>>> pd.DataFrame(data=None, index=None, columns=None)
```

► Panel (面板)

- Panel是一个稍微不常用的容器，但是对于三维数据仍然重要。术语面板数据源自计量经济学，是pandas名称的部分来源：pan(el)-da(ta)-s。

Series类型

- ▶ **Series**类型索引、切片、运算的操作类似于**ndarray**，同样的类似**Python**字典类型的操作，包括保留字**in**操作、使用**.get()**方法。
- ▶ **Series**和**ndarray**之间的主要区别在于**Series**之间的操作会根据索引自动对齐数据。

```
import numpy as np, pandas as pd
arr1 = np.arange(10)
s1 = pd.Series(arr1)
print(s1)
```

```
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64
```

```
import numpy as np, pandas as pd

dic = {"A":1, "B":2, "C":3, "D":2}
s2 = pd.Series(dic)
print(s2)
```

```
df2 = pd.DataFrame([s2, s2])
df2
```

```
A    1
B    2
C    3
D    2
dtype: int64
```

	A	B	C	D
0	1	2	3	2
1	1	2	3	2

DataFrame(数据帧)

- ▶ **DataFrame** 接受的数据类型：
 - 一维数组，列表，字典或 **Series** 的字典
 - 二维 `numpy.ndarray`
 - 结构化或记录 `ndarray`
 - **Series**
 - 另一个 **DataFrame**
- ▶ 和数据一起，选择传递 **index** (行标签) 和 **columns** (列标签) 作为索引参数。

```
import pandas as pd
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
df = pd.DataFrame(data)

print(df)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
import numpy as np, pandas as pd

s3 = pd.Series(np.arange(4), index=['a', 'b', 'c', 'd'])
print(s3)

df3 = pd.DataFrame([s3, s3])
df3
```

```
a    0
b    1
c    2
d    3
dtype: int64
```

	a	b	c	d
0	0	1	2	3
1	0	1	2	3

数据清理

- `df.columns = ['a','b','c']` : 重命名列名
- `pd.isnull()` : 检查DataFrame对象中的空值，并返回一个Boolean数组
- `pd.notnull()` : 检查DataFrame对象中的非空值，并返回一个Boolean数组
- `df.dropna()` : 删除所有包含空值的行
- `df.dropna(axis=1)` : 删除所有包含空值的列
- `df.dropna(axis=1,thresh=n)` : 删除所有小于n个非空值的行
- `df.fillna(x)` : 用x替换DataFrame对象中所有的空值
- `df.rename(columns=lambda x: x + 1)` : 批量更改列名
- `df.rename(columns={'old_name': 'new_name'})` : 选择性更改列名
- `df.set_index('column_one')` : 更改索引列
- `df.rename(index=lambda x: x + 1)` : 批量重命名索引
- `s.astype(float)` : 将Series中的数据类型更改为float类型
- `s.replace(1,'one')` : 用'one'代替所有等于1的值
- `s.replace([1,3],['one','three'])` : 用'one'代替1，用'three'代替3

Pandas导入和导出数据

- `pd.read_csv(filename)` : 从CSV文件导入数据
 - `pd.read_table(filename)` : 从限定分隔符的文本文件导入数据
 - `pd.read_excel(filename)` : 从Excel文件导入数据
 - `pd.read_sql(query, connection_object)` : 从SQL表/库导入数据
 - `pd.read_json(json_string)` : 从JSON格式的字符串导入数据
 - `pd.read_html(url)` : 解析URL、字符串或者HTML文件，抽取其中的
 - `pd.read_clipboard()` : 从你的粘贴板获取内容，并传给`read_table()`
 - `pd.DataFrame(dict)` : 从字典对象导入数据，**Key**是列名，**Value**是数据
-
- `df.to_csv(filename)` : 导出数据到CSV文件
 - `df.to_excel(filename)` : 导出数据到Excel文件
 - `df.to_sql(table_name, connection_object)` : 导出数据到SQL表
 - `df.to_json(filename)` : 以Json格式导出数据到文本文件

查看、检查数据

- `df.head(n)` : 查看**DataFrame**对象的前n行
- `df.tail(n)` : 查看**DataFrame**对象的最后n行
- `df.shape()` : 查看行数和列数
- `df.info()` : 查看索引、数据类型和内存信息
- `df.describe()` : 查看数值型列的汇总统计
- `s.value_counts(dropna=False)` : 查看**Series**对象的唯一值和计数
- `df.apply(pd.Series.value_counts)` : 查看**DataFrame**对象中每一列的唯一值和计数

数据选取

- `df[col]`：根据列名，并以**Series**的形式返回列
- `df[[col1, col2]]`：以**DataFrame**形式返回多列
- `s.iloc[0]`：按位置选取数据
- `s.loc['index_one']`：按索引选取数据
- `df.iloc[0,:]`：返回第一行
- `df.iloc[0,0]`：返回第一列的第一个元素

数据统计

- **df.describe()** : 查看数据值列的汇总统计
- **df.mean()** : 返回所有列的均值
- **df.corr()** : 返回列与列之间的相关系数
- **df.count()** : 返回每一列中的非空值的个数
- **df.max()** : 返回每一列的最大值
- **df.min()** : 返回每一列的最小值
- **df.median()** : 返回每一列的中位数
- **df.std()** : 返回每一列的标准差

Pandas创建数据

创建数据集

1、引入pandas

```
import numpy as np
import pandas as pd
```

2、创建数据集

```
# 生成创建一个6*4的正数数据集
data2 = pd.DataFrame(np.random.rand(6,4), columns=list('ABCD'))
data2
```

	A	B	C	D
0	0.268960	0.944233	0.564392	0.241277
1	0.739152	0.786348	0.108528	0.062371
2	0.777080	0.272329	0.986666	0.559934
3	0.234701	0.300360	0.667984	0.946984
4	0.807806	0.979275	0.793605	0.172620
5	0.753253	0.046280	0.403574	0.417993

```
# 先创建一个时间索引
dates = pd.date_range('20170101', periods=6)

# 再创建一个6*4的数据
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
```

	A	B	C	D
2017-01-01	-0.188280	-0.512786	-1.507800	1.020410
2017-01-02	-1.699451	0.580669	1.000826	0.719016
2017-01-03	0.412229	-1.902585	-1.260546	1.153787
2017-01-04	0.679351	0.633701	0.500856	-0.771894
2017-01-05	-0.383847	0.676279	0.548778	-0.117328
2017-01-06	0.036585	0.429563	-1.769048	-0.351982

3、使用字典来创建数据

```
df2 = pd.DataFrame({'A':np.random.randn(3)})
print df2
```

	A
0	0.249433
1	1.048593
2	-0.170596

4、另一种字典创建数据的方法

```
# 另一种用字典创建数据的方法
df3 = pd.DataFrame({'A':pd.Timestamp('20170101'),'B':np.random.randn(3)})
print df3
```

	A	B
0	2017-01-01	-0.364829
1	2017-01-01	-0.487836
2	2017-01-01	1.686728

Pandas查看数据

查看数据

1、使用dtypes来查看各行的数据格式

```
# 使用dtypes来查看各行的数据格式
df3.dtypes
```

	A	datetime64[ns]
	B	float64
		dtype: object

2、查看数据框中所有的数据

```
# 查看数据框中所有的数据
df3
```

	A	B
0	2017-01-01	-0.545882
1	2017-01-01	-0.033721
2	2017-01-01	-0.388657

3、使用head查看前几行数据（默认是前5行）

```
# 使用head查看指定的前几行数据（不指定时默认是前5行）
df.head(3)
```

	A	B	C	D
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327

4、使用tail查看后5行数据

```
# 使用tail查看后指定的后几行数据（默认是5行）
df.tail(3)
```

	A	B	C	D
2017-01-04	0.198526	0.286079	1.907239	1.415250
2017-01-05	0.976051	-0.874763	0.236659	0.452393
2017-01-06	0.987277	0.104625	-2.007644	1.041692

5、查看数据框的索引

```
# 查看数据框的索引
df.index
```

```
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04',  
               '2017-01-05', '2017-01-06'],  
              dtype='datetime64[ns]', freq='D')
```

6、用columns查看列名

```
# 使用columns查看列名
df.columns
```

```
Index([u'A', u'B', u'C', u'D'], dtype='object')
```

7、用values查看数据值

```
# 用values查看数据值
df.values
```

```
array([[ 0.68806638,  0.81934586,  0.34296473,  1.73047519],  
       [ 0.0332296 ,  0.5998538 , -1.47720515, -0.237258  ],  
       [ 1.99659048, -1.91628982,  0.42948988,  1.00732749],  
       [ 0.19852554,  0.28607882,  1.90723864,  1.41525027],  
       [ 0.97605102, -0.87476293,  0.23665913,  0.45239314],  
       [ 0.98727715,  0.10462485, -2.00764438,  1.04169165]])
```

8、用describe查看描述性统计

```
# 用describe查看描述性统计
df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.813290	-0.163525	-0.094750	0.901647
std	0.701295	1.039315	1.425101	0.704338
min	0.033230	-1.916290	-2.007644	-0.237258
25%	0.320911	-0.629916	-1.048739	0.591127
50%	0.832059	0.195352	0.289812	1.024510
75%	0.984471	0.521410	0.407859	1.321861
max	1.996590	0.819346	1.907239	1.730475

9、用T转置，即行列转换

```
# 用T转置，即行列转换
df.T
```

	2017-01-01 00:00:00	2017-01-02 00:00:00	2017-01-03 00:00:00	2017-01-04 00:00:00	2017-01-05 00:00:00	2017-01-06 00:00:00
A	0.688066	0.033230	1.996590	0.198526	0.976051	0.987277
B	0.819346	0.599854	-1.916290	0.286079	-0.874763	0.104625
C	0.342965	-1.477205	0.429490	1.907239	0.236659	-2.007644
D	1.730475	-0.237258	1.007327	1.415250	0.452393	1.041692

10、用sort_values对数据进行排序

```
# 用到sort_values对数据进行排序
df.sort_values(by='C')
```

	A	B	C	D
2017-01-06	0.987277	0.104625	-2.007644	1.041692
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-05	0.976051	-0.874763	0.236659	0.452393
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-03	1.996590	-1.916290	0.429490	1.007327
2017-01-04	0.198526	0.286079	1.907239	1.415250

Pandas数据选择 - 1

1、选择A列的数据进行操作

```
# 选择A列的数据进行操作
df['A']
```

```
2017-01-01    0.688066
2017-01-02    0.033230
2017-01-03    1.996590
2017-01-04    0.198526
2017-01-05    0.976051
2017-01-06    0.987277
```

2、使用数组的切片操作得到行数据

```
# 使用数组的切片操作得到行数据
df[1:3]
```

	A	B	C	D
2017-01-02	0.03323	0.599854	-1.477205	-0.237258
2017-01-03	1.99659	-1.916290	0.429490	1.007327

3、使用行标签来指定输出的行(列不限定, 行限定)

```
# 使用行标签来指定输出的行(列不限定, 行限定)
df['20170102':'20170104']
```

	A	B	C	D
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327
2017-01-04	0.198526	0.286079	1.907239	1.415250

4、使用loc方法选择多列数据(列不限定, 行限定)

```
# 使用loc方法选择多列数据(列不限定, 行限定)
df.loc[dates[0]:dates[2],:]
```

	A	B	C	D
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327

5、使用loc方法选择多列数据(行不限定, 列限定)

```
# 使用loc方法选择多列数据(行不限定, 列限定)
df.loc[:, 'B':'D']
```

	B	C	D
2017-01-01	0.819346	0.342965	1.730475
2017-01-02	0.599854	-1.477205	-0.237258
2017-01-03	-1.916290	0.429490	1.007327
2017-01-04	0.286079	1.907239	1.415250
2017-01-05	-0.874763	0.236659	0.452393
2017-01-06	0.104625	-2.007644	1.041692

6、使用loc方法选择多列数据(行列都限定, 且列只要指定的某些)

```
# 使用loc方法选择多列数据(行列都限定, 且列只要指定的某些)
df.loc[dates[0]:dates[2], ['B', 'D']]
```

	B	D
2017-01-01	0.819346	1.730475
2017-01-02	0.599854	-0.237258
2017-01-03	-1.916290	1.007327

数据选择

7、使用loc方法选择某行数据(如第一行)

```
# 使用loc方法选择某行数据(如第一行)
df.loc[dates[0]]
```

```
A    0.688066
B    0.819346
C    0.342965
D    1.730475
```


Pandas数据选择 - 2

数据选择

7、使用loc方法选择某行数据(如第一行)

```
# 使用loc方法选择某行数据(如第一行)
df.loc[dates[0]]
```

A	0.688066
B	0.819346
C	0.342965
D	1.730475

8、使用loc方法只选择某一个数据, 可以指定行和列

```
# 使用loc方法只选择某一个数据, 可以指定行和列
df.loc[dates[0], 'B']
```

0.81934585546715122

9、at方法专门用于获取某个值

```
# at方法专门用于获取某个值
df.at[dates[0], 'B']
```

0.81934585546715122

10、使用iloc方法提取第四行数据

```
# 使用iloc方法提取第四行数据, 得到的返回值是一个series 数据
df.iloc[3]
```

A	1.658872
B	-0.250281
C	-0.531258
D	-0.091432

11、返回4-5行, 2-3列数据

```
# 返回4-5行, 2-3列数据
df.iloc[3:5, 1:3]
```

	B	C
2017-01-04	-0.250281	-0.531258
2017-01-05	0.123437	-1.455018

12、提取不连续行和列的数, 如2, 4, 6行的B列和D列

```
# 提取不连续行和列的数, 如2, 4, 6行的B列和D列
df.iloc[[1, 3, 5], [1, 3]]
```

	B	D
2017-01-02	-0.449626	-2.183514
2017-01-04	-0.250281	-0.091432
2017-01-06	-0.319806	0.200287

13、提取某一行或某几行数据, 保证所有列都在

```
# 提取某一行或某几行数据, 保证所有列都在(提取所有行的某些列类似)
df.iloc[1:3, :]
```

	A	B	C	D
2017-01-02	-0.813571	-0.449626	-0.548597	-2.183514
2017-01-03	-1.704362	0.218734	-1.052347	0.369618

14、提取某个值, 如第二行第二列

```
# 提取某个值, 如第二行第二列
df.iloc[1, 1]
```

-0.44962649890028467

15、iat是专门提取某个数的方法, 它的效率高更高

```
# iat是专门提取某个数的方法, 它的效率高更高
df.iat[1, 1]
```

-0.44962649890028467

Pandas数据库操作

数据库操作

1、用read_sql从sqlite数据库中读取数据

```
# 用read_sql从sqlite数据库中读取数据
import sqlite3
con = sqlite3.connect('user_information.sqlite')
sql = 'select * from user_information LIMIT 3'
df = pd.read_sql(sql, con)
```

2、用index_col参数来规定将那一列数据设置为index

```
# 使用index_col 参数来规定将那一列数据设置为index
df = pd.read_sql(sql, con, index_col='id')
```

3、可以设置多个index, 只要将index_col的值设置为列表

```
# 可以设置多个index, 只要将index_col的值设置为列表
df = pd.read_sql(sql, con, index_col=['id', 'bank_id'])
```

4、删除数据库中的某个表

```
# 删除数据库中的某个表
con.execute('DROP TABLE IF EXISTS user_information')
```

5、将df保存到数据库中的user_information表

```
# 将df保存到数据库中的user_information表
pd.io.sql.write_sql(df, 'user_information', con)
```

6、假设我们使用的是MySQL数据库, 同样可以

```
# 假设我们使用的是MySQL数据库, 同样可以
import MySQLdb
con = MySQLdb.connect(host='localhost', db='databasename')
```

GPU科学加速包RAPIDS

- ▶ **cuDF** : **DataFrame**操作库，包含对加载、过滤、数据操作等过程的加速，基于**cuda**内核加速的接口与**pandas**无缝衔接
- ▶ **cuML** : **GPU**加速的机器学习库，包括**scikit-learn**中的所有机器学习算法
- ▶ **cuGRAPH** : 图分析库

```
import cudf

gdf = cudf.read_csv('path/to/file.csv')
for column in gdf.columns:
    print(gdf[column].mean())
```

METHOD	Preferred		Advanced	
	Conda	Docker + Examples	Docker + Dev Env	Source
RELEASE	Legacy (0.14)		Stable (0.15)	
PACKAGES	All Packages	cuDF	cuML	cuGraph
LINUX	Ubuntu 16.04	Ubuntu 18.04	CentOS 7	RHEL 7
PYTHON	Python 3.6 (0.14 only)		Python 3.7	
CUDA	CUDA 10.0 (0.14 only)		CUDA 10.2	