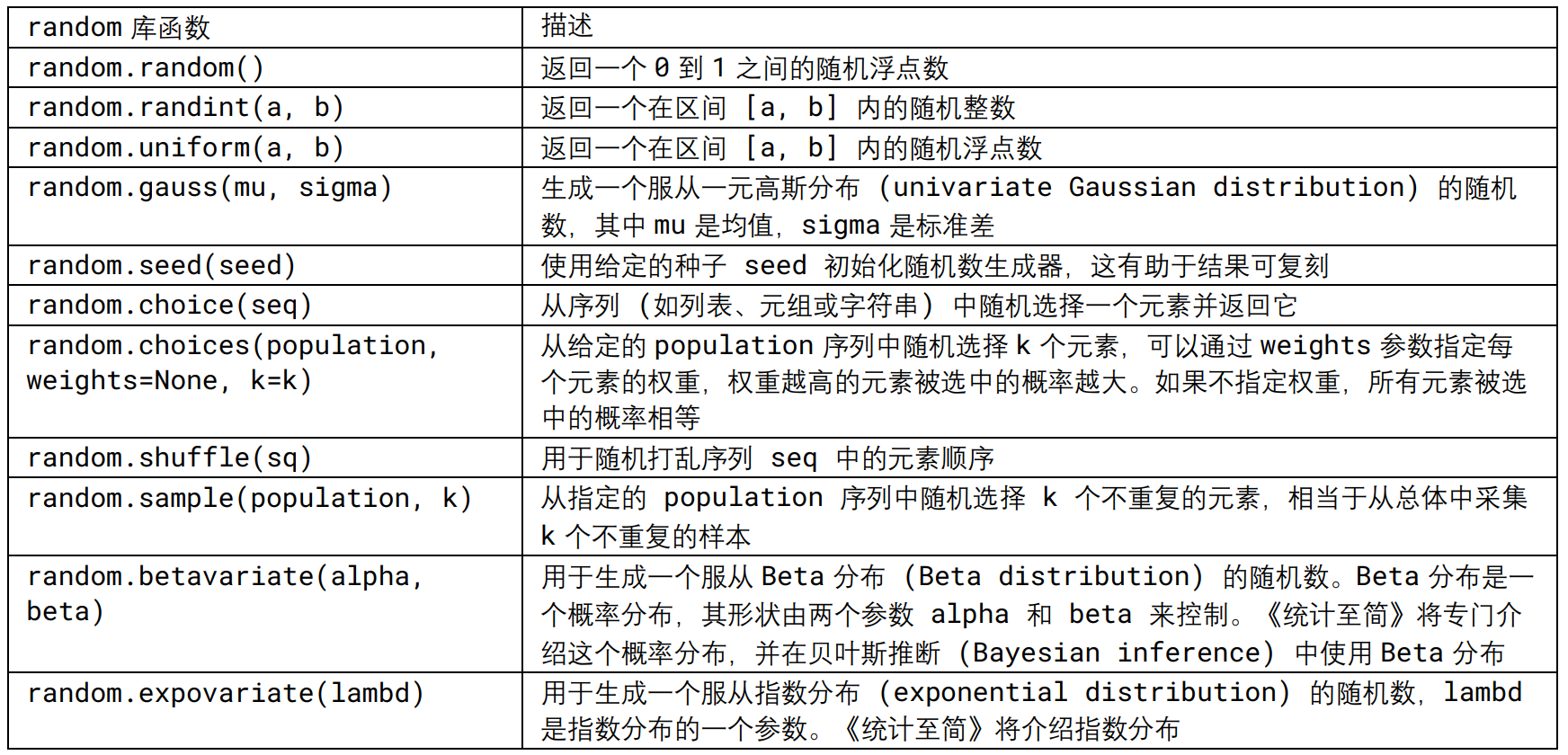
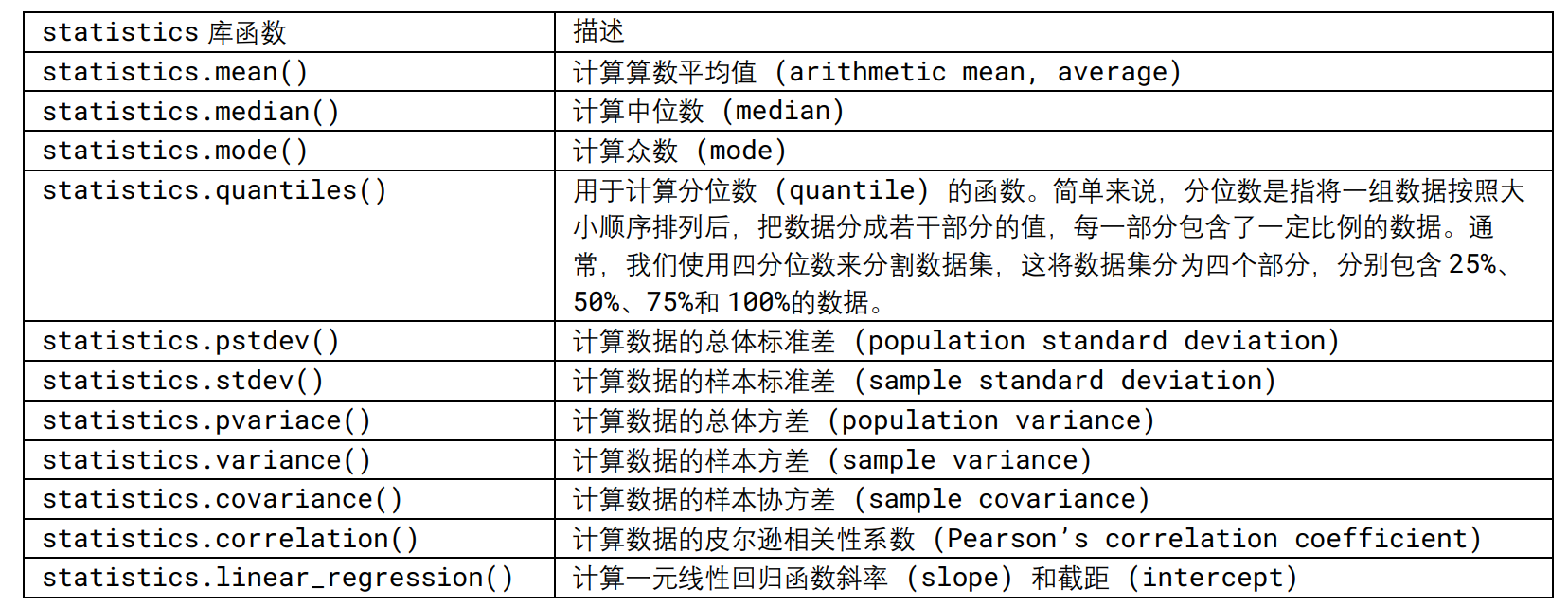
[1.代码部分](#代码部分)

①[NumPy](#numpy) ②[Pandas](#pandas) ③[SciPy](#Scipy) ④[Matplotlib](#Matplotlib)

**代码部分**

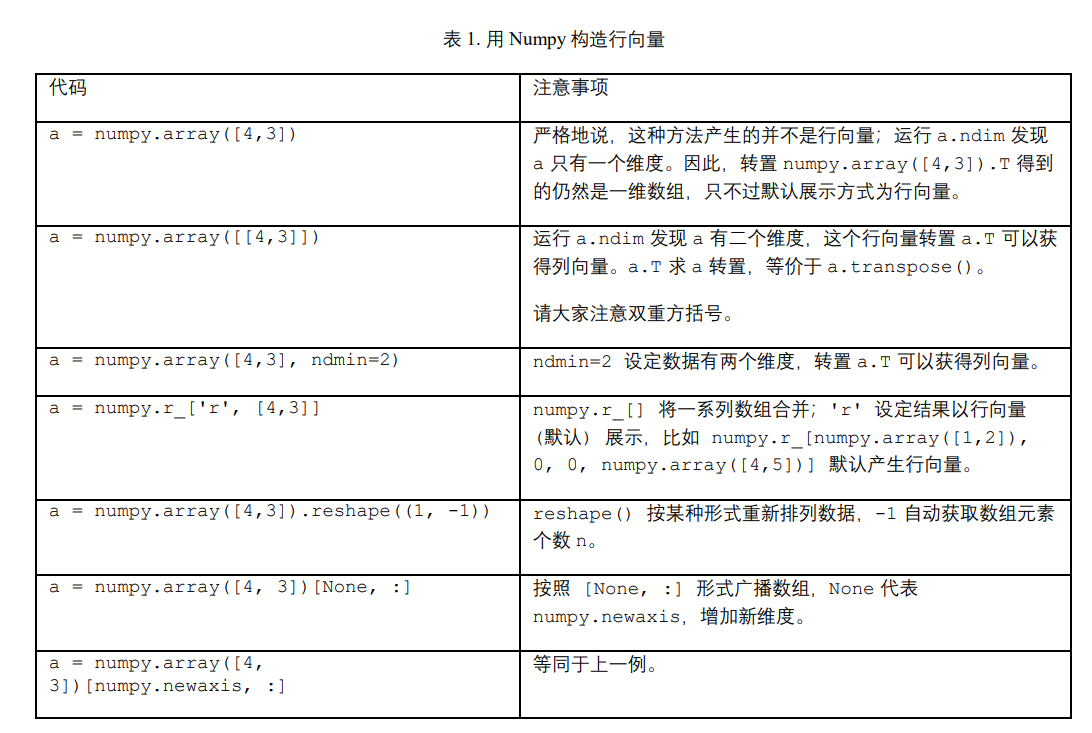


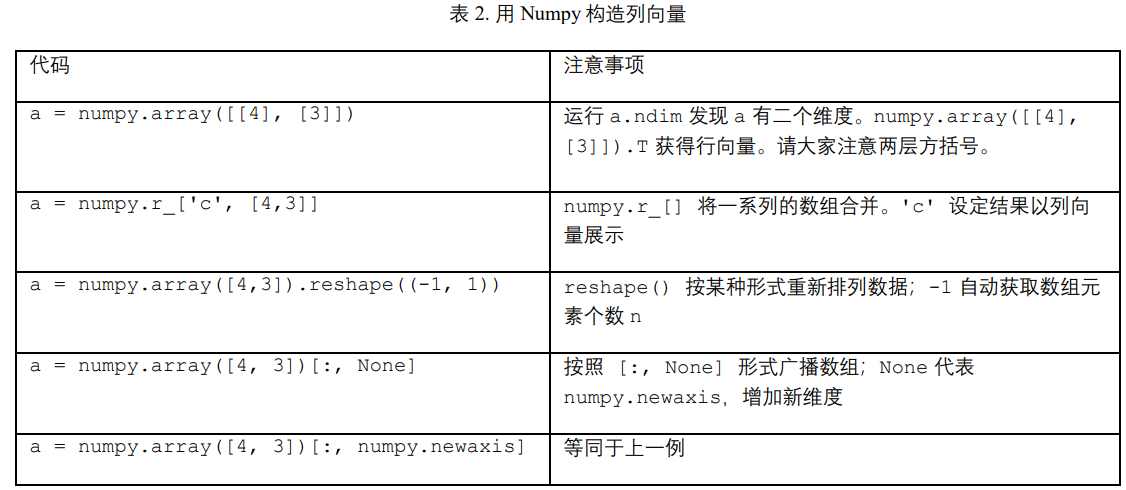


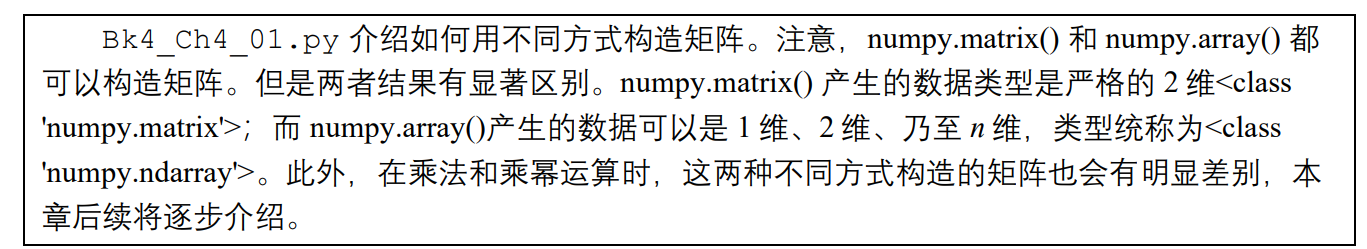
**一、基础部分**

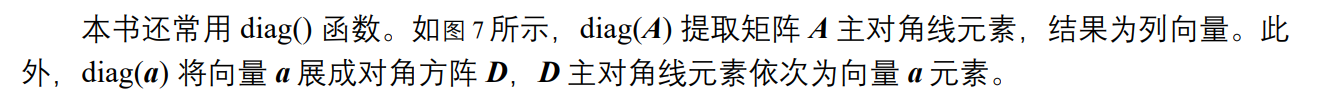
**1.NumPy**

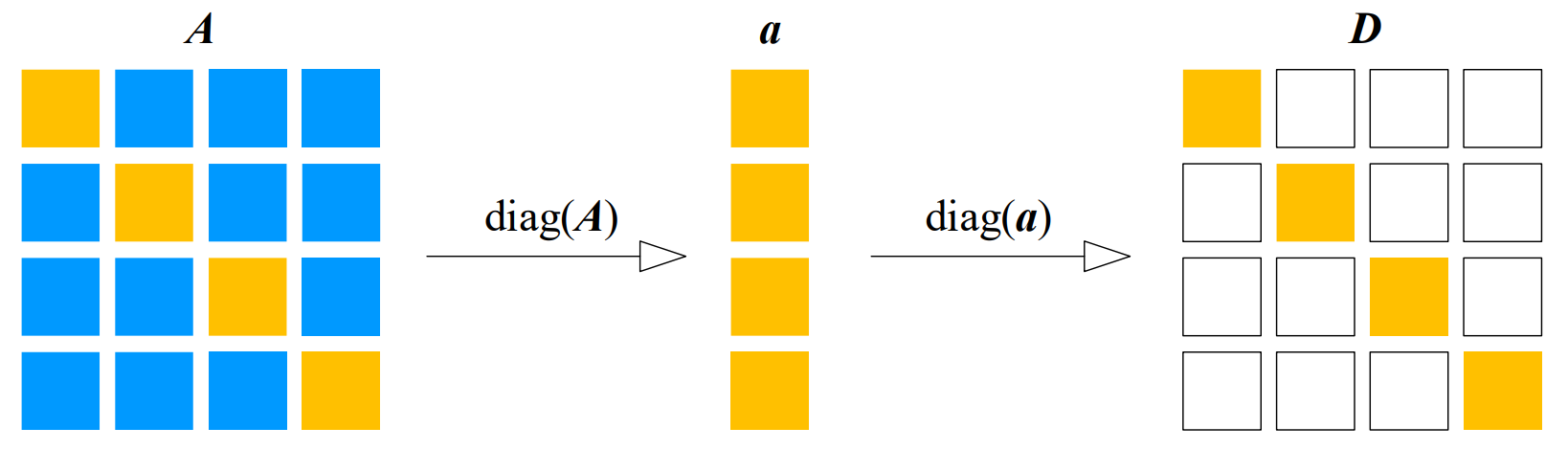
NumPy是Numerical Python的缩写。numerical ：relating to numbers; expressed in numbers



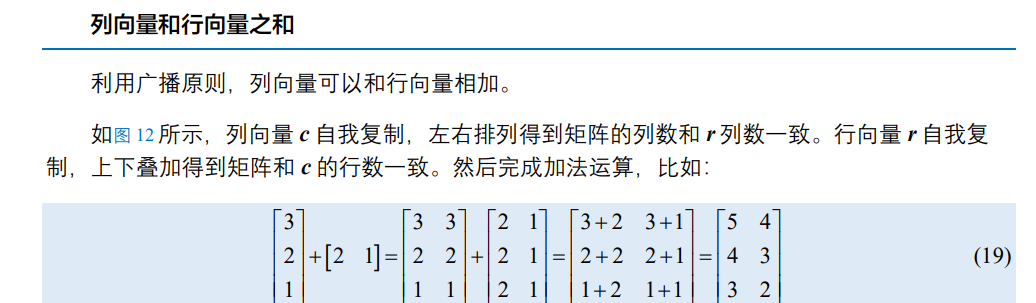


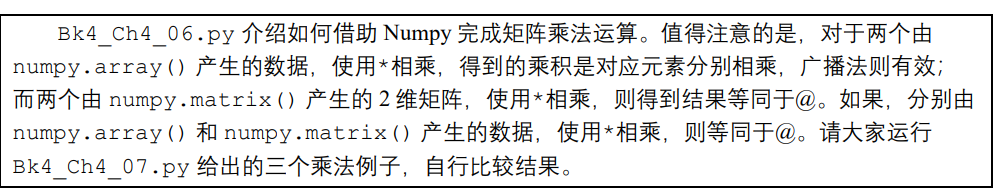


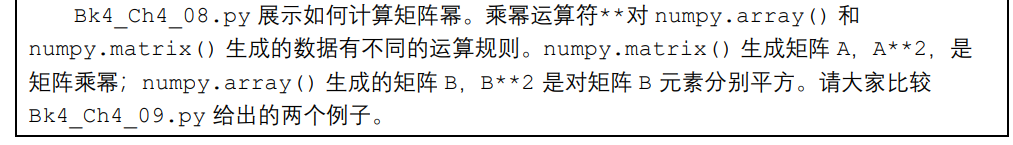




NumPy 中的矩阵加减运算常使用广播原则 (broadcasting)。当两个数组的形状并不相同的时 候，可以通过广播原则扩展数组来实现相加、相减等操作







**初识numpy——列表/元组转为数组**

import numpy as np  
  
list\_demo = [1, 2, 3, 4]  
a = np.array(list\_demo)  
print(a)  
  
tuple\_demo = ('1', '2', '3', '4')  
b = np.asarray(tuple\_demo)  
print(b)  
  
list\_tuple = [('胡桃', '可莉', '纳西妲'), ('7.15', '7.27', '10.27')]  
c = np.asarray(list\_tuple)  
print(c)

输出

[1 2 3 4]

['1' '2' '3' '4']

[['胡桃' '可莉' '纳西妲']

['7.15' '7.27' '10.27']]

第三个的输出：将列表转换为NumPy数组时，NumPy会将列表中的元组视为一个整体，并将该整体作为数组的一个元素。

**初识numpy——创建数值数组**

array1 = np.arange(0, 10, 2, int)  
print(array1)  
  
array2 = np.arange(0, 1, 0.2, float)  
print(array2)  
  
array3 = np.linspace(0, 1, num=5, endpoint=False, retstep=True)  
print(array3)

输出

[0 2 4 6 8]

[0. 0.2 0.4 0.6 0.8]

(array([0. , 0.2, 0.4, 0.6, 0.8]), 0.2)

函数**arange**用于创建一个等差数列的一维数组，参数包括起始值、结束值（不包含在数组中）、步长和数据类型，其中起始值默认0，步长默认为1，数据类型参数可省略。

函数**linspace**用于在一定范围内生成等间隔的数值序列，其中参数num表示生成数组的元素个数，endpoint表示是否包含结束点，retstep表示是否返回步长。

**其他方法：**

**ones(shape,dtype)**：指定数组的形状，生成一个包含全1元素的数组。数据类型为dtype，可省略。

**eye(N)**：指定方阵的大小N，生成一个单位矩阵。

**zeros(shape,dtype)**：指定数组的形状，生成一个包含全0元素的数组。数据类型为dtype，可省略。

注：dtype可省略，shape可以只有一个，默认为行数。

比如

print(np.eye(2))  
print(np.ones((3, 2)))  
print(np.zeros((3, 2)))

输出

[[1. 0.]

[0. 1.]]

[[1. 1.]

[1. 1.]

[1. 1.]]

[[0. 0.]

[0. 0.]

[0. 0.]]

注意到这里的ones与zeros都是用的((a,b))，不要只打一个括号。

事实上有**eye(N, M=None, k=0, dtype=float)**，这个比较复杂，单独说。它创建一个大小为(N, M)的二维数组，其中“对角线”上的元素为1，其他元素为0。偏移量k表示“对角线”向右的偏移量。比如：

e = np.eye(5, 6, -1, int)  
print(e)

输出

[[0 0 0 0 0 0]

[1 0 0 0 0 0]

[0 1 0 0 0 0]

[0 0 1 0 0 0]

[0 0 0 1 0 0]]

-1表示向左偏移1。

**Ndarray对象**

Ndarray对象是NumPy库中的多维数组对象，集合了N个类型相同的数据，shape与dtype分别反应了数组维度与数据类型情况。创建Ndarray可以用array()方法，上文已经使用过了。原来是嵌套格式的序列会被转换成多维数组。

对Ndarray对象使用**.ndim**与**.dtype**可以获取数组维度数与数据类型，用**.shape**与**.size**可以获取大小(行,列)与元素数量，用**. itemsize**可以获取元素的字节数。

从外向里依次“剥皮”，也就是去掉[]，就分别是数组的第1,2,3,......维度，对应的是shape[0],shape[1],shape[2]…。而shape[n]的值的计算方法是：去掉外面的中括号之后，内部的最大的中括号有几个。

比如下文中去掉最外面的[]后，得到的单位体的个数表示第0个维度(**axis=0**)的大小，剩下的是[[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]]，只有一个括号，所以shape[0]为1。

接着去掉[]后，剩下[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]，有三个括号，所以shape[1]为3。

然后内部元素是4个，shape[2]为4。

list\_demo = [[[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]]]  
a = np.array(list\_demo)  
print(a.ndim)  
print(a.dtype)  
print(a.shape)

输出

3

int32

(1, 3, 4)

上文提及了参数axis，该参数在sum求和里比较重要：

print(a.sum(axis=0))  
print(a.sum(axis=1))  
print(a.sum(axis=2))

输出

[[1 2 3 4]

[1 2 3 4]

[1 2 3 4]]

[[ 3 6 9 12]]

输出

[[[5 0 3]

[3 7 3]

[5 2 4]

[7 6 8]]

[[8 1 6]

[7 7 8]

[1 5 8]

[4 3 0]]]

[[13 1 9]

[10 14 11]

[ 6 7 12]

[11 9 8]]

[[20 15 18]

[20 16 22]]

[[ 8 13 11 21]

[15 22 14 7]]

[[10 10 10]]

其中axis=0说明去掉一个[]，这时候shape[0]=1，此时就是一个数组，sum运算后还是该数组。axis=1说明去掉两层[]，这时候shape[1]=3，此时就是三个数组[1,2,3,4]，将这三个数组相加得到sum。axis=2说明去掉三层[]，这时候shape[2]=4，将1,2,3,4相加就是10。

现在应该大致明白了计算方法，下面来谈谈这个输出里面的[]个数如何确定。因为shape[0]=1的时候不方便观察和(因为axis=0时没有相加过程)，所以这里取shape[0]=2的一个数组，比如shape=(2,4,3)。比如

arr = [[[5, 0, 3], [3, 7, 3], [5, 2, 4], [7, 6, 8]], [[8, 1, 6], [7, 7, 8], [1, 5, 8], [4, 3, 0]]]  
arr = np.asarray(arr)  
print(arr, end='\n\n')  
print(arr.sum(axis=0), end='\n\n')  
print(arr.sum(axis=1), end='\n\n')  
print(arr.sum(axis=2), end='\n\n')

输出见右边文本框

axis=0时，去掉最外层的[]，此时两个[[…]]的数组相加。

axis=1时，去掉中间的[]，此时[[[…]]]变为[[…]](但这与axis=0的时候的[[…]]是不一样的，因为去掉的那个[]不一样)。

axis=2时，去掉最内层的[]，此时是数字元素之间的相加了。

**显式转换**

arr = [[0.0, 1.0, 1.1], [-1.5, 1.5, -1.6]]  
arr = np.array(arr)  
print(arr)  
print(arr.astype(np.int32))

输出

[[ 0. 1. 1.1]

[-1.5 1.5 -1.6]]

[[ 0 1 1]

[-1 1 -1]]

这里类型转化的时候的小数点是被截断的。

**数组的操作**

NumPy数组与普通的列表的操作差不多，这里仅说明一些比较重要的。

**使用切片修改原数据**

arr = [1, 2, 3, 4, 5]  
arr = np.array(arr)  
arr\_slice = arr[2:4] # 截取的是元素3,4  
arr\_slice[1] = 44 # 把截取的元素中的第二个(这里是4)改为44  
print(arr)

输出[ 1 2 3 44 5]

如果想要arr\_slice里的元素全部更改，可以用arr\_slice[:]=44，不要忘记:了。

**多维下标索引**

和C里面的基本一样，比如

arr = [[1, 2, 3], [4, 5, 6]]  
arr = np.array(arr)  
print(arr[0], end='\t')  
print(arr[0][1], end='\t')  
print(arr[0, 1])

输出[1 2 3] 2 2

但是，如果需要的是某一部分的元素，难度比较大，需要索引与切片结合。比如

print(arr[:1, 2:])

输出[[3]]

这里输出的是行为:1，列为2:的元素。切片:1表示结束位置为1且不包括1，2:表示起始位置为2且包括2。又如

print(arr[1:, 1:2])

print(arr[:, 1:])

输出

[[5]]

[[2 3]

输出

[[[ 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]]]

(2, 4, 4)

[[[ 0 1 2 3]

[16 17 18 19]]

[[ 4 5 6 7]

[20 21 22 23]]

[[ 8 9 10 11]

[24 25 26 27]]

[[12 13 14 15]

[28 29 30 31]]]

(4, 2, 4)

[5 6]]

亦可对切片赋值：

arr[:, 1:] = 0  
print(arr)

输出

[[1 0 0]

[4 0 0]]

**转置**

arr = [[1, 2, 3], [4, 5, 6]]  
arr = np.array(arr)  
print(arr.T)

print(arr.transpose())

输出均为

[[1 4]

[2 5]

[3 6]]

arr = np.arange(32).reshape((2, 4, 4))  
print(arr)  
print(arr.shape)  
print(arr.transpose(1, 0, 2))  
print(arr.transpose(1, 0, 2).shape)

输出如右文本框

这里(1,0,2)表示第一个维度(0)与第二个维度(1)交换，第三个维度(2)不变。也就是shape从(2,4,4)变为(4,2,4)。之前axis=0的维度里0123后面的是16171819，所以现在axis=1的维度里0123后面的是16171819，以此类推。

**生成随机数组**

print(np.random.randn(3,4))  
print(np.random.choice([10,20,30]))  
print(np.random.beta(1,5,10))

输出

[[ 1.62816762 1.47448347 1.00854331 1.47853539]

[-0.19053414 -1.06565901 0.33980349 -1.31290175]

[-0.16290617 -0.5937015 0.82524027 1.56946873]]

10

[0.01761509 0.21731346 0.04291937 0.12523884 0.12166361 0.14969903

0.03939118 0.17141757 0.02659418 0.04199945]

np.random.randn函数生成的是服从标准正态分布的随机数。

np.random.choice从给的的数组里随机选一个元素。

np.random.beta是一个使用beta分布生成随机数的函数。前两个参数是beta分布的形状参数，第三个参数表示生成的随机数的数量。

Beta分布是一种连续型概率密度分布，表示为，由两个参数决定，称为形状参数。由于其定义域为(0,1)，一般被用于建模伯努利试验事件成功的概率的概率分布：为了测试系统的成功概率，我们做n次试验，统计成功的次数k，于是很直观地就可以计算出。然而由于系统成功的概率是未知的，这个公式计算出的只是系统成功概率的最佳估计。也就是说实际上也可能为其它的值，只是为其它的值的概率较小。因此我们并不能完全确定硬币出现正面的概率就是该值，所以也是一个随机变量，它符合Beta分布，其取值范围为0到1。参考<https://zhuanlan.zhihu.com/p/69606875，https://zhuanlan.zhihu.com/p/149964631>。

print(np.random.rand(2,4))  
print(np.random.rand())  
print(np.random.randint(1,10,3))

输出

[[0.46653209 0.35644468 0.83639798 0.27475056]

[0.68402447 0.93758497 0.44335841 0.22709871]]

0.6939882918736527

[5 5 1]

np.random.rand生成0~1的随机数

np.random.randint(1,10,3)生成[1,10]之间的长度为3的随机整数数组

**np.array()和np.asarray()的区别**

import numpy as np  
  
arr1 = np.ones((3, 3))  
arr2 = np.array(arr1)  
arr3 = np.asarray(arr1)  
arr1[1] = 2  
print('arr1:\n', arr1)  
print('arr2:\n', arr2)  
print('arr3:\n', arr3)

输出

arr1:

[[1. 1. 1.]

[2. 2. 2.]

[1. 1. 1.]]

arr2:

[[1. 1. 1.]

[1. 1. 1.]

[1. 1. 1.]]

arr3:

[[1. 1. 1.]

[2. 2. 2.]

[1. 1. 1.]]

发现arr3被arr1改了，这是因为：

np.array(默认情况下)将会copy该对象，而np.asarray 除非必要，否则不会copy该对象。(必要的意思是数据源是ndarray类型时，不会copy)。也就是说array和asarray都可以将结构数据转化为ndarray，但是主要区别就是当数据源是ndarray时，array仍然会copy出一个副本，占用新的内存，但asarray不会。

**矩阵的乘法——星乘\*与点乘dot**：

**1.星乘\***

①同型矩阵(哈达玛积)。对应位置的元素相乘：

如

②不同型，但两个矩阵行数相等，其中一个矩阵列数为1。单列矩阵的列与另一个矩阵的列分别相乘：

如

**2.点乘dot**

就是线代里的矩阵乘法

比如：

输出

[[ 5 12]

[21 32]]

[[ 5 6]

[14 16]]

[[ 5 6]

[14 16]]

[[19 22]

[43 50]]

import numpy as np  
  
a = np.array([[1, 2], [3, 4]])  
b = np.array([[5, 6], [7, 8]])  
c = np.array([[1], [2]])  
print(a \* b, '\n')  
print(c \* b, '\n')  
print(b \* c, '\n')  
print(np.dot(a, b))

**矩阵的逆矩阵：**

与线代的定义一样，，比如：

import numpy as np  
  
a = np.array([[1, 2], [3, 4]])  
a\_inv = np.linalg.inv(a)  
print(a\_inv)  
print(np.dot(a, a\_inv))

输出

[[-2. 1. ]

[ 1.5 -0.5]]

[[1.0000000e+00 0.0000000e+00]

[8.8817842e-16 1.0000000e+00]]

inv是inverse的缩写：相反的，倒转的；颠倒的，逆的

linalg是numpy库中的线性代数模块，可以用于进行矩阵和向量的运算，linalg是linear algebra(线性代数)的缩写。

这里有烦人的浮点数，可以这样修改代码：

a\_dot\_a\_inv = np.dot(a, a\_inv)  
a\_dot\_a\_inv = np.round(a\_dot\_a\_inv).astype(int) # 将a\_dot\_a\_inv中的元素四舍五入为整数  
print(a\_dot\_a\_inv)

输出

[[1 0]

[0 1]]

**矩阵的行列式**

A = np.array([[1, 0, 0], [0, 2, 5], [0, 0, 3]])  
print(np.linalg.det(A))

输出

6.0

**线性方程组**

A = np.array([[1, 0, 0], [0, 2, 5], [0, 0, 3]])  
b = np.array([1, 2, 3])  
x = np.linalg.solve(A, b)  
print(x)

输出

[ 1. -1.5 1. ]

**特征值和特征向量**

A = np.array([[1, 2], [0, -1]])  
Eigenvalue, Eigenvector = np.linalg.eig(A)  
print(Eigenvalue) # 特征值  
print(Eigenvector) # 特征向量

输出

[ 1. -1.]

[[ 1. -0.70710678]

[ 0. 0.70710678]]

**矩阵范数**

n = np.linalg.norm(A, ord=2) # 计算矩阵范数，类型n=2代表谱范数  
print(n)

输出

2.414213562373095

**2.****Pandas**

**Series数据结构**

**创建Series数组**

import pandas as pd  
  
obj = pd.Series([2, 4, 3, 1])  
print(obj)  
print(obj.values)  
print(obj.index)

输出

0 2

1 4

2 3

3 1

dtype: int64

[2 4 3 1]

RangeIndex(start=0, stop=4, step=1)

series类似一维数组，由一组数据与对应的数据标签组成。

索引在左，值在右，并注明dtype。还可以在创建时指定索引：

obj1 = pd.Series([2, 4, 3, 1], index=['b','D','C','a'])  
print(obj1)  
print(obj1.values)  
print(obj1.index)

输出

b 2

D 4

C 3

a 1

dtype: int64

[2 4 3 1]

Index(['b', 'D', 'C', 'a'], dtype='object')

**索引**

print(obj1[1])  
print(obj1['D'])

输出均为4

print(obj1[[1, 2]])

输出

D 4

C 3

dtype: int64

obj1.index = ["a", 'bb', 'cc', 'd']  
print(obj1)

输出

a 2

bb 4

cc 3

d 1

dtype: int64

用赋值的方式可以修改index

注意引用索引时，利用标签index的切片运算的末端是包含在内的，这与普通的切片运算不同。

**重新索引**

data = pd.Series([7, 1, 5, 2, 3], index=['H', 'u', 'T', 'a', 'o'])  
data = data.reindex(['H', 'u', 't', 'a', 'o', 'T'], fill\_value=0)  
print(data)

输出

H 7

u 1

t 0

a 2

o 3

T 5

dtype: int64

不指定fill\_value就默认为NAN。注意这个reindex可不是随意指定的，只要是与之前的index不同就会视为没有值，也就是NAN。

**丢弃**

drop()可以丢弃指定行/列

data = pd.Series([7, 1, 5, 2, 3], index=['H', 'u', 'T', 'a', 'o'])

data = data.drop('a')  
print(data)

输出

H 7

u 1

T 5

o 3

dtype: int64

**DataFrame数据结构**

**创建DataFrame**

DataFrame是表格类型，有多个列，每个列都能设置不同的类型(字符串、数值、布尔值等)，需要保证它们长度相等。

import pandas as pd  
  
data = {'name': ["Hu Tao", "Klee", "Keqing"],  
 'bir': [7.15, 7.27, 11.20],  
 'num': [1, 2, 3]} # 创建字典  
frame = pd.DataFrame(data) # 使用字典创建数据表  
print(frame)

输出

name bir num

0 Hu Tao 7.15 1

1 Klee 7.27 2

2 Keqing 11.20 3

也可以**指定列顺序**：

frame = pd.DataFrame(data, columns=['num', 'bir', 'name'])  
print(frame)

输出

num bir name

0 1 7.15 Hu Tao

1 2 7.27 Klee

2 3 11.20 Keqing

**获取DataFrame的某一列**作为一个Series：

print(frame['name'])

print(frame.name)

均输出

0 Hu Tao

1 Klee

2 Keqing

Name: name, dtype: object

**丢弃**

import pandas as pd  
  
data = {'name': ["Hu Tao", "Klee", "Keqing"],  
 'bir': [7.15, 7.27, 11.20],  
 'num': [1, 2, 3]}  
frame = pd.DataFrame(data)  
frame = frame.drop(1)  
frame = frame.drop('name', axis=1)  
print(frame)

输出

bir num

0 7.15 1

2 11.20 3

先删除index=1的行(第二行)[因为这里没有指定index，所以按默认的来]，再删除name列(要注明axis=1，指定在列上操作。默认值axis=0表示行)

指定index要按如下方法删除(再用1就会报错了KeyError: '[1] not found in axis')：

frame = pd.DataFrame(data,index=['a','b','c'])  
frame = frame.drop('b')

**索引**

data = pd.DataFrame(np.arange(16).reshape((4,4)),  
 index=['A', 'B ', 'C', 'D'],  
 columns=['a', 'b', 'c', 'd'])  
print(data)  
print(data.loc['C': 'D', 'b'])

输出

a b c d

A 0 1 2 3

B 4 5 6 7

C 8 9 10 11

D 12 13 14 15

C 9

D 13

Name: b, dtype: int32

这里如果截取的是某一列或者某一行，就会显示Name与dtype，否则就像全部输出一样只有index,columns与元素。

**算术运算**

如果索引相同则对对应内容加减，若有不同的索引，则该索引对应的值为NAN。

**以Series为例：**

obj1 = pd.Series([1, 2, 3, 4], index=['a','b','c','d'])  
obj2 = pd.Series([12, 34, 56, 78], index=['a','B','c','d'])  
print(obj1+obj2)

输出

B NaN

a 13.0

b NaN

c 59.0

d 82.0

dtype: float64

输出是浮点数因为在Panda中，当进行数据计算时，如果有一个操作数是浮点数，则结果的数据类型会自动转换为浮点数。NaN（Not a Number）值的数据类型是float。

**以DataFrame为例：**

df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)),  
 columns=list('abc'),  
 index=list('ABC'))  
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)),  
 columns=list('abc'),  
 index=list('ABCD'))  
print(df1+df2)

输出

a b c

A 0.0 2.0 4.0

B 6.0 8.0 10.0

C 12.0 14.0 16.0

D NaN NaN NaN

**以DataFrame与Series运算为例：**

series = df2.loc['A']  
print(df2-series)

输出

a b c

A 0.0 0.0 0.0

B 3.0 3.0 3.0

C 6.0 6.0 6.0

D 9.0 9.0 9.0

这里是将Series的列匹配到DataFrame的每一列，若有一方索引不对应，则值为NAN。

其中df2与series为：

a 0.0

b 1.0

c 2.0

Name: A, dtype: float64

a b c

A 0.0 1.0 2.0

B 3.0 4.0 5.0

C 6.0 7.0 8.0

D 9.0 10.0 11.0

与

用0,1,2对应去减a,b,c列。

**3.****SciPy**

**保存与读取矩阵文件**

from scipy import io  
import numpy as np  
arr = np.array([1,2,3,4,5,6])  
io.savemat('scipy\_test.mat',{'arr1':arr}) # 将数据保存到.mat文件，其中'arr1'是在.mat文件中存储的数组的键名  
loadArr = io.loadmat('scipy\_test.mat') # 从.mat文件中加载数据  
print(loadArr['arr1'])

输出[[1 2 3 4 5 6]]

在当前文件夹保存scipy\_test.mat文件，使用MATLAB打开如下：



**统计功能**

import scipy.stats as stats

**均匀分布**

x = stats.uniform.rvs(size=10)

**正态分布**

x = stats.norm.rvs(size=10)

**贝塔分布**

x = stats.beta.rvs(size=10, a=2, b=3)

**泊松分布**

x = stats.poisson.rvs(0.8, loc=0, size=10) # 0.8是泊松分布的参数λ，表示平均发生率或速率。loc=0表示事件发生的起始值，默认为0。

**均值与标准差计算**

x = np.array([0,1,2,3,4])  
print(stats.norm.fit(x))

输出

(2.0, 1.4142135623730951)

**偏度计算**

x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])  
print(stats.skewtest(x))

输出

SkewtestResult(statistic=-0.8048934446906357, pvalue=0.42088117154954274)

**峰度计算**

x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])  
print(stats.kurtosis(x))

输出-0.4344773135788995

**正态分布程度检验**

x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727, 0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])  
print(stats.normaltest(x))

输出NormaltestResult(statistic=0.9975897425378244, pvalue=0.6072620478568534)

该函数需要至少20个参数，否则给出warning：

UserWarning: kurtosistest only valid for n>=20 ... continuing anyway, n=10

warnings.warn("kurtosistest only valid for n>=20 ... continuing "

但还是能给出结果

**计算某一百分比处的数值**

x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])  
print(stats.scoreatpercentile(x, 50))

输出0.155

这里计算的是中位数。

**SciPy应用**

import numpy as np  
from scipy import stats  
  
arr = np.array(  
 [[0, 2], [2.5, 4], [5, 6], [7.5, 9], [10, 13], [12.5, 16], [15, 19], [17.5, 23], [20, 27], [22.5, 31], [25, 35],  
 [27.5, 40], [30, 53], [32.5, 68], [35, 90], [37.5, 110], [40, 130], [42.5, 148], [45, 165], [47.5, 182], [50, 195],  
 [52.5, 208], [55, 217], [57.5, 226], [60, 334], [62.5, 342], [65, 349], [67.5, 500], [70, 511], [72.5, 300],  
 [75, 200], [77.5, 80], [80, 20], [82.5, 50], [85, 6], [90, 3]])  
score, num = arr[:, 0], arr[:, 1] # 所有行第一列；所有行第二列  
All\_score = np.repeat(list(score), list(num))  
  
  
def count(score):  
 # 集中趋势度量  
 print('均值:', np.mean(score))  
 print('中位数:', np.median(score))  
 print('众数:', stats.mode(score))  
 # 离散趋势度量  
 print('极差:', np.ptp(score))  
 print('方差:', np.var(score))  
 print('标准差:', np.std(score))  
 print('变异系数:', np.mean(score) / np.std(score))  
 # 偏度与峰度的度量  
 print('偏度:', stats.skewtest(score))  
 print('峰度:', stats.kurtosis(score))  
  
  
count(All\_score)

输出

均值: 57.65014855687606

中位数: 62.5

众数: ModeResult(mode=70.0, count=511)

极差: 90.0

方差: 215.64357383192066

标准差: 14.684807585798346

变异系数: 3.9258361554991996

偏度: SkewtestResult(statistic=-23.201191283130118, pvalue=4.428865440236225e-119)

峰度: 0.7136422362619905

**Scipy实现优化算法**

**一元无约束优化：**

from scipy.optimize import minimize  
import numpy as np

func = lambda x: x \*\* 2  
x\_init = np.random.random(1)  
res = minimize(func, x\_init)  
print("最小值:", res.fun)  
print("最优解:", res.x)  
print("迭代终止是否成功", res.success)  
print("迭代终止原因", res.message)

输出：

最小值: 1.1488908661148124e-17

最优解: [3.38952927e-09]

迭代终止是否成功 True

迭代终止原因 Optimization terminated successfully.

注：之后不再显式给出这四句print。

**一元带约束优化：**

**使用constraints**

func = lambda x: x \*\* 2  
cons = ({'type': 'ineq', 'fun': lambda x: x - 2})  
x\_init = np.array(1)  
res = minimize(func, x\_init, constraints=cons) # 最优解x=2

**使用bounds**

func = lambda x: x \*\* 2  
x\_init = np.array(1)  
res = minimize(func, x\_init, bounds=((3, 5), )) # 最优解x=3

两种约束差不多，而constrains更适合复杂情况下的约束。

**一元带超参数优化：**

def func(x, a):  
 return x\*\*2 - 4\*a\*x  
a = 2  
x\_init = np.array(1)  
res = minimize(func, x\_init, args=(a,)) # 最优解x=4

这里也是可以用func = lambda x, a: x\*\*2 + 4\*a\*x的。而因为PEP 8: E731规范，不建议将匿名函数表达式赋值给一个变量再用这个变量调用函数，因此除非需要简短的代码，此时还是更建议用def，之后也以def为主。

**二元函数带约束和超参数优化：**

def func(x, a):  
 return x[0]\*\*2 + a\*x[1]  
cons = ({'type': 'eq', 'fun': lambda x: x[0] \* x[1] - 10}, # 约束条件：xy=10  
 {'type': 'ineq', 'fun': lambda x: x[0]}, # x>0  
 {'type': 'ineq', 'fun': lambda x: x[1]}) # y>0  
a = 2  
x\_init = np.random.random(2)  
res = minimize(func, x\_init, args=(a,), constraints=cons)

问题为的条件下求。

易得。

**不定积分的优化问题：**

def func(x):  
 x\_sym = sp.symbols('x')  
 expression = x\_sym \*\* 2 + 2 \* x\_sym + sp.sin(x\_sym)  
 result = xm\_mf.math\_integral(x\_sym, expression)  
 result = result.subs(x\_sym, x[0]) # 因为传入的x是np.array数组  
 return result  
x\_init = np.random.random(1)  
res = minimize(func, x\_init)

问题为。

导入的包是import easier\_excel.math\_formula as xm\_mf。

也可以使用scipy计算数值积分：

def fx(x):  
 return 1/x  
result, error = xm\_mf.value\_integral1(fx, 1, 2)  
print(result, error)  
  
def fxy(x, y):  
 return x\*\*2 + y\*\*2  
result, error = xm\_mf.value\_integral2(fxy, 0, 1, 0, 2)  
print(result, error)

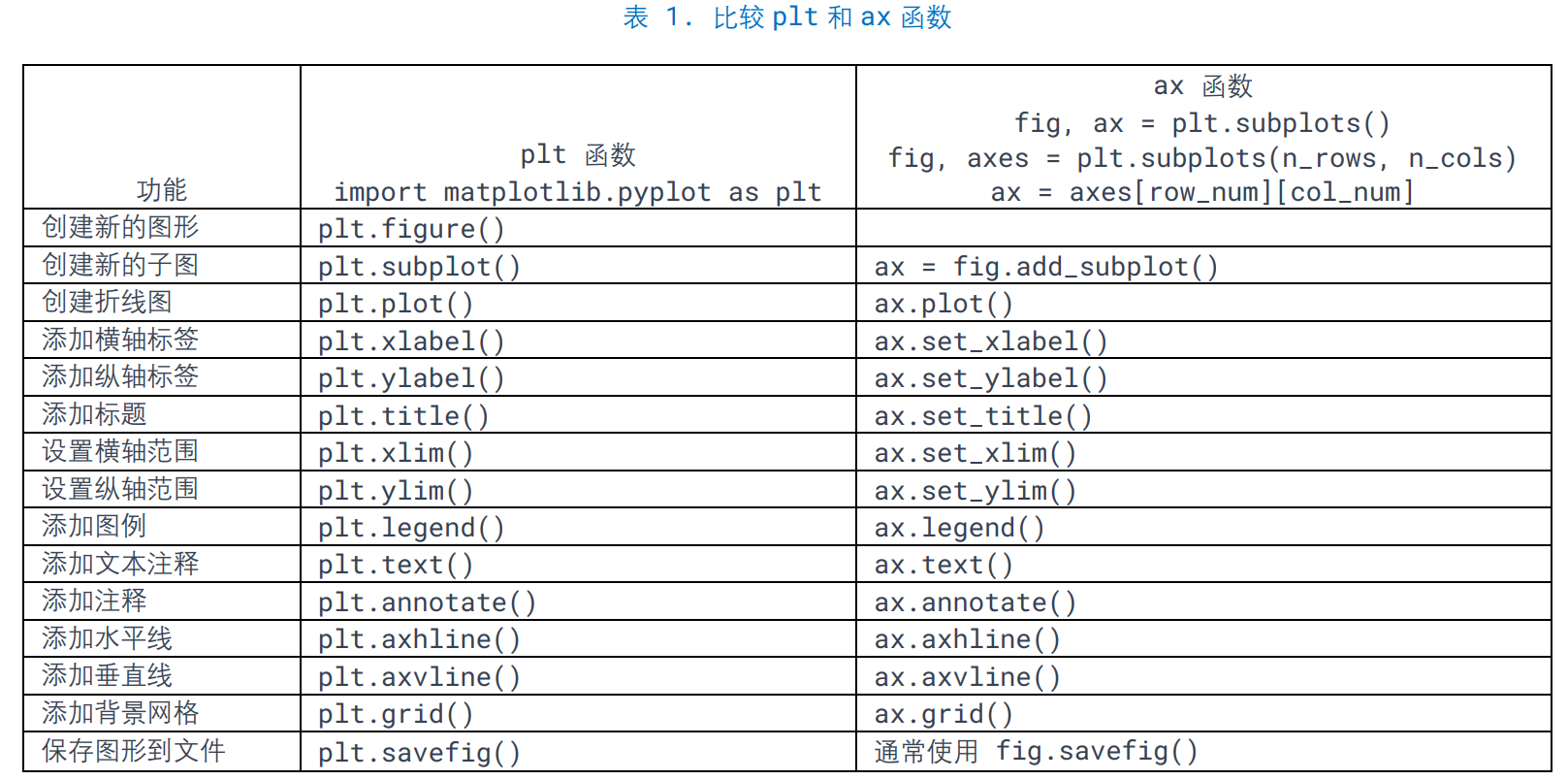
其它细节略去。

**4.****Matplotlib**

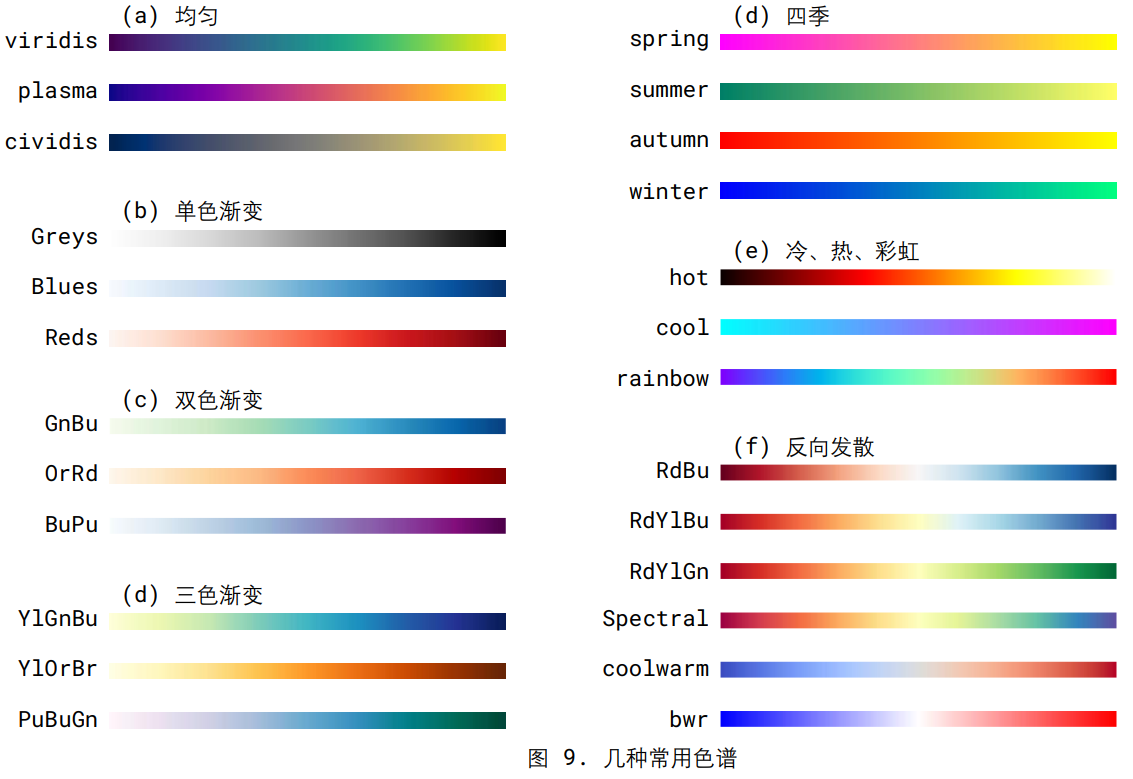
subplot(a,b,c)的意思是把窗口分为a行b列，选择第c块。

ax.set\_aspect('equal')设置横纵轴采用相同的比例，使图形在绘制时不会因为坐标轴的

比例问题产生形变(比如创建正方形子图fig,ax = plt.subplots(figsize=(6,6))会使得椭圆看起来像圆)。



colormap：颜色映射/色谱：



可视化方案的图片可参考https://matplotlib.org/stable/plot\_types/index.html

**绘制等高线图**matplotlib.pyplot.contour(X,Y,Z,levels,cmap)：

contourf是在contour的基础上进行了填充操作。

X：二维数组，表示数据点的横坐标。

Y：二维数组，表示数据点的纵坐标。

Z：二维数组，表示数据点对应的函数值或高度。

levels：用于指定绘制的等高线层级或数值列表。

colors：用于指定等高线的颜色，可以是单个颜色字符串、颜色序列或 colormap 对象。

cmap：颜色映射，用于将数值映射为颜色。可以是预定义的 colormap名称或colormap对象。

linestyles：用于指定等高线的线型，可以是单个线型字符串或线型序列。

linewidths：用于指定等高线的线宽，可以是单个线宽值或线宽序列。

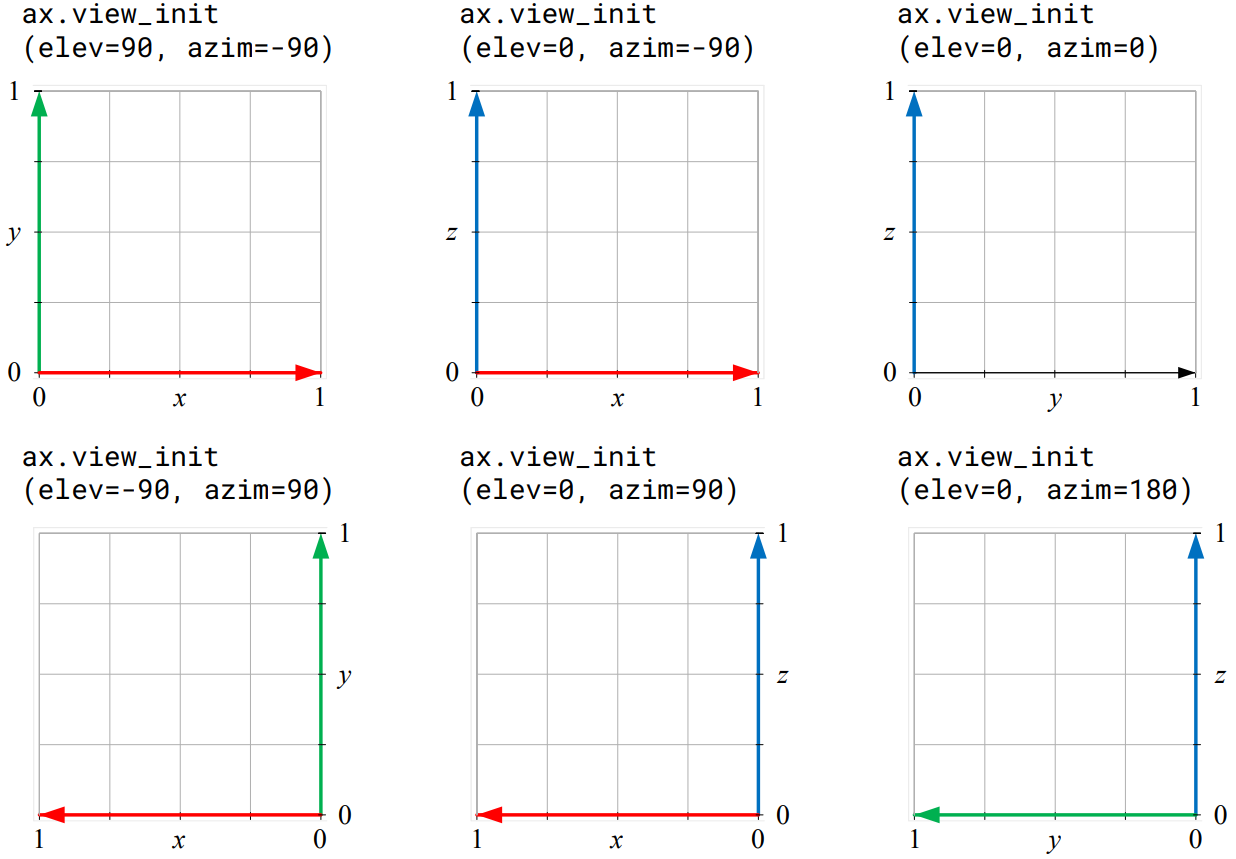
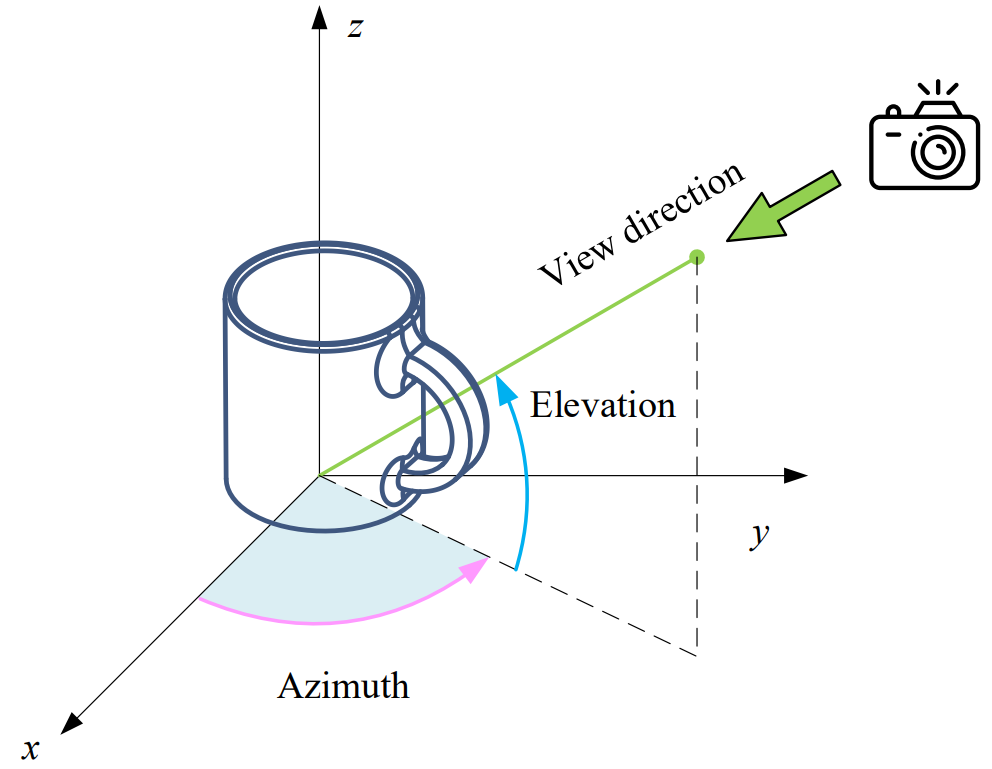
alpha：用于指定等高线的透明度。

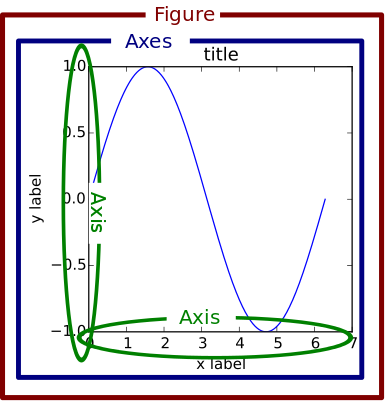
设置三维视角ax.view\_init(elev, azim, roll)：

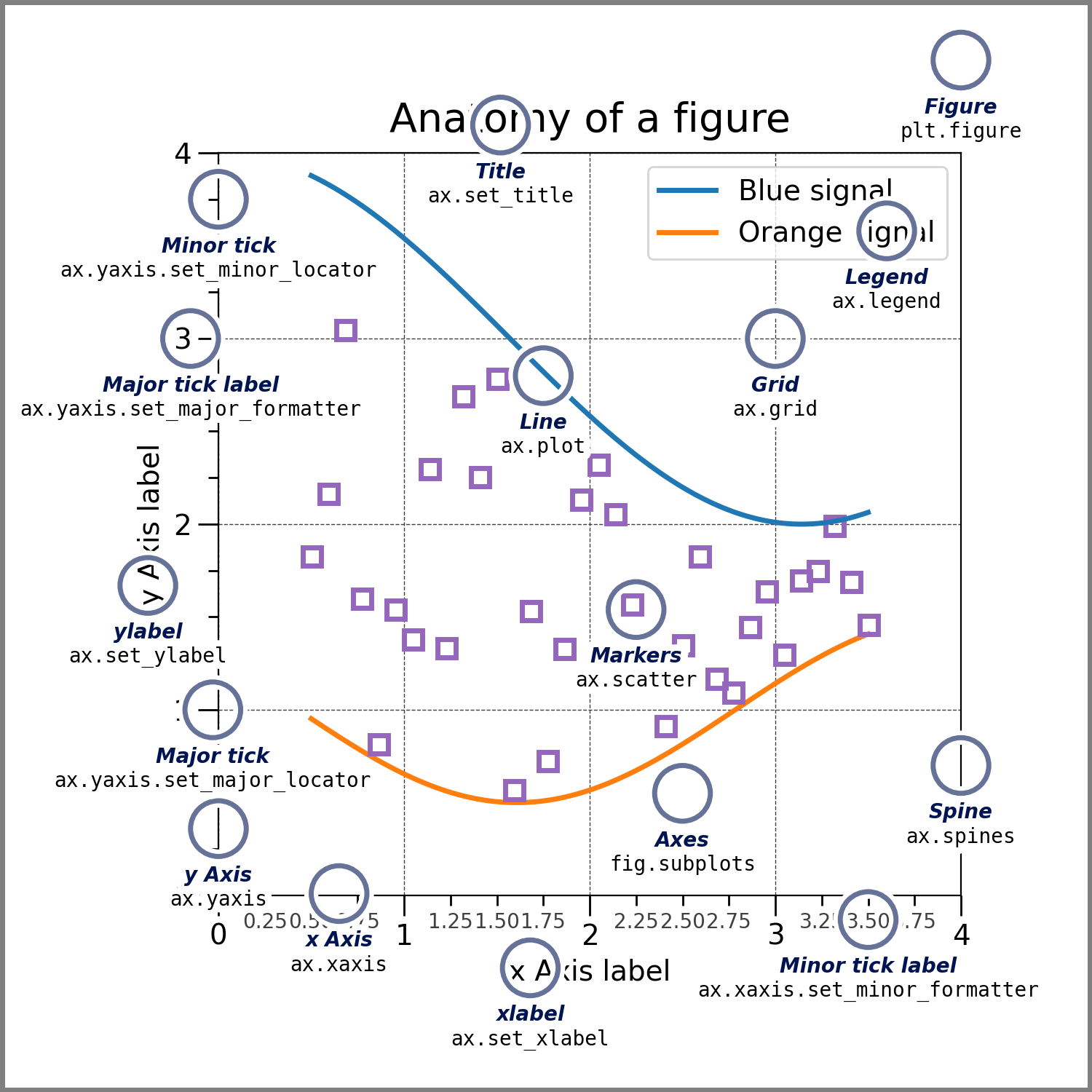
仰角(elevation)：观察者与xy平面之间的夹角。正值时观察者向上倾斜，负值表示向下倾斜。

方位角(azimuth)：观察者绕z轴旋转的角度。正值表示逆时针旋转，负值表示顺时针旋转。

滚动角(roll)：绕观察者视线方向旋转的角度，即观察者的头部倾斜程度。正值表示向右侧倾斜，负值表示向左侧倾斜。

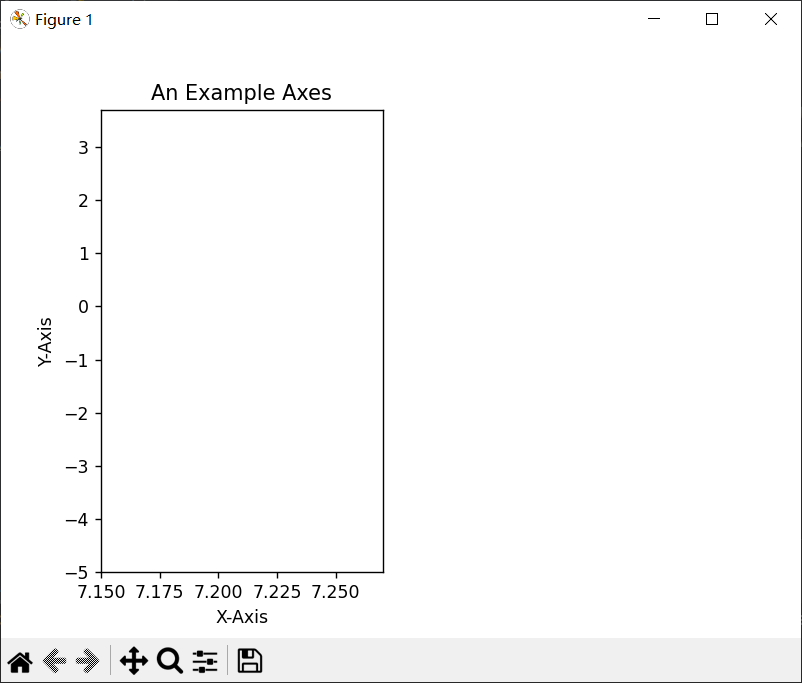
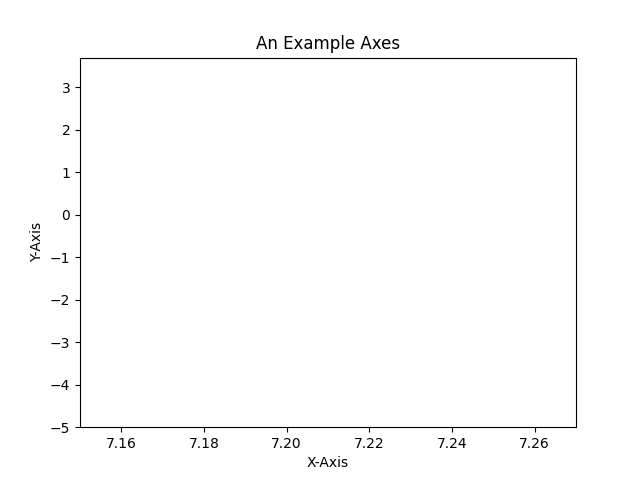






**绘制坐标轴**

import matplotlib.pyplot as plt  
  
fig = plt.figure()  
ax = fig.add\_subplot(111) # 在fig图形对象中添加一个子图，编号为1行1列的第1个子图。  
ax.set(xlim=[7.15, 7.27], ylim=[-5, 3.7], title='An Example Axes',  
 ylabel='Y-Axis', xlabel='X-Axis')  
plt.show()



add\_subplot用于创建和管理子图。参数改为121会输出右上图。

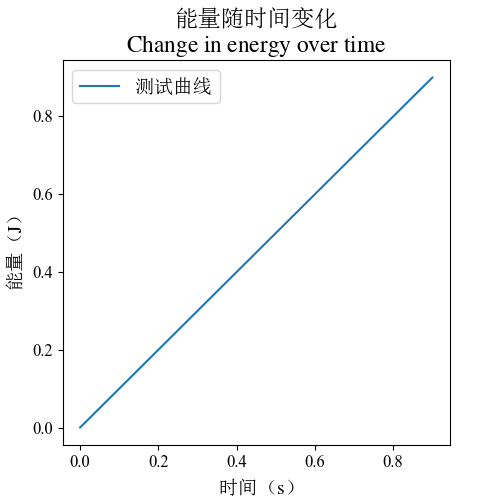
**使用双语双字体**

import matplotlib.pyplot as plt  
from matplotlib import rcParams  
  
config = {  
 "font.family": 'serif',  
 "font.size": 20,  
 "mathtext.fontset": 'stix',  
 "font.serif": ['SimSun'],  
}  
rcParams.update(config)  
  
plt.title(r'宋体 $\mathrm{Times \; New \; Roman}\/\/ \alpha\_i > \beta\_i$')  
plt.axis('off')  
plt.show()



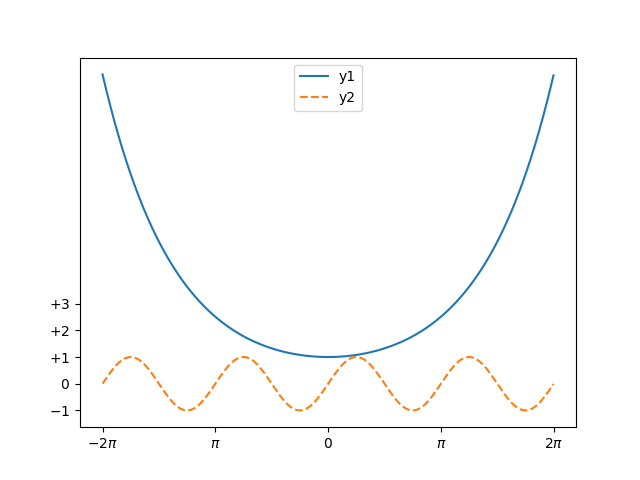
如果不使用第二句import就要把rcParams.update(config)改为plt.rcParams.update(config)

import matplotlib.pyplot as plt  
  
config = {  
 "font.family": "serif", # 使用衬线体  
 "font.serif": ["SimSun"], # 全局默认使用衬线宋体  
 "font.size": 14, # 五号，10.5磅  
 "axes.unicode\_minus": False,  
 "mathtext.fontset": "stix", # 设置 LaTeX 字体，stix 近似于 Times 字体  
}  
plt.rcParams.update(config)  
  
  
fig, ax = plt.subplots(figsize=(5, 5))  
ax.plot([i / 10.0 for i in range(10)], [i / 10.0 for i in range(10)])  
# 中西混排，西文使用 LaTeX 罗马体  
ax.set\_title("能量随时间变化\n$\mathrm{Change\ in\ energy\ over\ time}$")  
ax.set\_xlabel("时间（$\mathrm{s}$）")  
ax.set\_ylabel("能量（$\mathrm{J}$）")  
  
# 坐标系标签使用西文字体  
ticklabels\_style = {  
 "fontname": "Times New Roman",  
 "fontsize": 12, # 小五号，9磅  
}  
plt.xticks(\*\*ticklabels\_style)  
plt.yticks(\*\*ticklabels\_style)  
plt.legend(["测试曲线"])  
plt.show()



**绘制曲线**

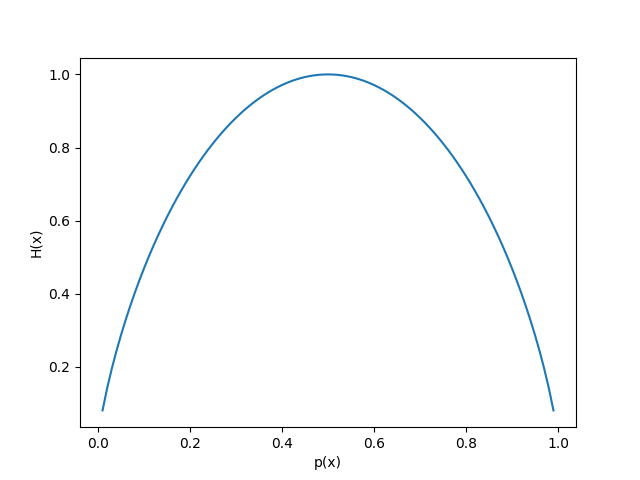
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.arange(-2 \* np.pi, 2 \* np.pi, 0.01)  
y1 = np.cosh(0.5 \* x)  
y2 = np.sin(2 \* x)  
plt.plot(x, y1)  
plt.plot(x, y2, '--')  
plt.xticks([-2 \* np.pi, -np.pi, 0, np.pi, 2 \* np.pi], [r'$-2\pi$', r'$\pi$', '$0$', '$\pi$', '$2\pi$'])  
plt.yticks([-1, 0, 1, 2, 3], [r'$-1$', '$0$', '$+1$', '$+2$', '$+3$'])  
plt.legend(['y1', 'y2'])  
plt.show()



**计算二元信源的熵**

**伯努利分布的熵**

import matplotlib.pyplot as plt  
from math import log  
import numpy as np  
  
  
# 计算二元信息熵  
def entropy(props, base=2):  
 sum = 0  
 for prop in props:  
 sum += prop \* log(prop, base)  
 return sum \* -1  
  
  
# 构造数据  
x = np.arange(0.01, 1, 0.01)  
props = []  
for i in x:  
 props.append([i, 1 - i])  
  
y = [entropy(i) for i in props]  
  
plt.plot(x, y)  
plt.xlabel("p(x)")  
plt.ylabel("H(x)")  
plt.show()



<https://segmentfault.com/a/1190000039676856> 两组离散数据求交点

<https://www.cnblogs.com/vamei/archive/2012/09/17/2689798.html> 饼状图

[https://matplotlib.org/stable/api/index.html 库的api](https://matplotlib.org/stable/api/index.html%20%20库的api)参考

<https://zhuanlan.zhihu.com/p/93423829> axes的讲解

**5. sympy**

**1.基本表达式**

from sympy import \*  
  
x = symbols('x') # 创建符号变量x  
a, b = symbols('a b') # 创建符号变量a和b  
  
expr = (x + 1)\*\*2  
expanded\_expr = expr.expand() # 展开表达式  
print(expanded\_expr) # x\*\*2 + 2\*x + 1  
  
expression = (x\*\*2 + 2\*x + 1) / (x + 1)  
simplified\_expr = simplify(expression) # 代数式简化  
print(simplified\_expr) # x + 1  
  
equation = (x + 1)\*\*2  
solutions1 = solve(equation, x) # 求解方程  
solutions2 = solve(Eq(equation, 2), x) # Eq代表等式  
print(solutions1, solutions2) # 返回list类型 [-1] 和 [-1 + sqrt(2), -sqrt(2) - 1]  
solutions = solveset(equation < 4, x, domain=S.Reals) # 求解不等式  
print(solutions) # Interval.open(-3, 1)  
  
limit\_val = equation.limit(x, 2) # 计算极限  
print(limit\_val) # 9  
  
integral = integrate(equation, x) # 积分  
print(integral) # x\*\*3/3 + x\*\*2 + x(这里忽略了常数C)

integral\_value = integrate(equation, (x, 0, 3)) # 积分  
print(integral\_value) # 21

r = diff(equation, x) # 求导  
print(r) # 2\*x + 2

**2.多个表达式**

from sympy import \*  
import numpy as np  
  
x = symbols('x')  
expr1 = x\*\*2  
expr2 = 2\*x + 12  
result = expr1 + expr2  
print(result) # 两个表达式相加，即x\*\*2 + 2\*x + 12  
  
x = np.array([1, 2, 4, 5, 7])  
y = np.array([2, 5, 7, 10, 13])  
a = symbols('a') # 95\*a\*\*2 - 362\*a + 347  
y\_predict = a\*x # 预测y=ax  
f\_a\_SSE = np.sum((y - y\_predict)\*\*2) # 定义平方和误差f(a)=这5个的(y-ax)^2之和  
f\_a\_SSE = simplify(f\_a\_SSE) # 化简表达式  
print(f\_a\_SSE)  
derivative\_f\_a = f\_a\_SSE.diff(a) # 求导得190\*a - 362  
a\_minpoint = solve(derivative\_f\_a, a) # [181/95]  
print(f"最优参数 a = {a\_minpoint[0]}")  
print(f"预测模型 y = {float(a\_minpoint[0])} \* x")  
  
a, b = symbols('a b')  
y\_predict = a\*x+b # 预测y=ax+b  
f\_ab\_SSE = np.sum((y - y\_predict)\*\*2) # 定义平方和误差f(a)=这5个的(y-(ax+b))^2之和  
f\_ab\_SSE = simplify(f\_ab\_SSE)  
print(f\_ab\_SSE)  
derivative\_f\_ab\_a = f\_ab\_SSE.diff(a) # 分别对a,b求偏导数  
derivative\_f\_ab\_b = f\_ab\_SSE.diff(b)  
minimization\_points = solve([derivative\_f\_ab\_a, derivative\_f\_ab\_b], (a, b)) # 解方程系统，使偏导数等于零  
print(f"最小值点:{minimization\_points}") # 最小值点: {a: 101/57, b: 2/3}  
print(f"最优参数 a = {minimization\_points[a]}, b = {minimization\_points[b]}")  
print(f"预测模型 y = {float(minimization\_points[a])} \* x + {float(minimization\_points[b])}")  
  
from sklearn.linear\_model import LinearRegression  
x = np.array([1, 2, 4, 5, 7]).reshape(-1, 1) # 将[1 2 4 5 7]转换为一个列向量[[1]\n[2]\n[4]\n[5]\n[7]](\n代表换行)  
y = np.array([2, 5, 7, 10, 13])  
model = LinearRegression()  
model.fit(x, y)  
print("权重参数:", model.coef\_[0])  
print("偏置参数:", model.intercept\_)

我们预测的是y = 1.7719298245614035 \* x + 0.6666666666666666，和

权重参数:1.7719298245614037，偏置参数: 0.666666666666667 可以说是一样的。

导数为0(驻点)是极值的必要条件，如果此点左右的导数同号则不是极值点。

**6.torch**

**基本信息**

import torch  
print(torch.cuda.is\_available()) # 是否可使用GPU（True）  
print(torch.cuda.device\_count()) # 可用GPU数量（1）  
print(torch.\_\_version\_\_) # 2.2.0+cu118

**基本的初始化方法**

x1 = torch.arange(12)  
print(x1)  
print(x1.shape)  
x2 = x1.reshape(3, 4)  
print(x2)  
print(x2.shape)  
print(x2.numel())  
x3 = torch.zeros((2, 2, 3))  
print(x3)  
x4 = torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])  
print(x4)

shape返回张量大小，也就是列出了张量沿每个轴的长度(维数)。numel()返回张量的元素总数(即shape的元素相乘的值)。因此A.mean() = A.sum() / A.numel()。

zeros全0张量，ones全1张量，randn张量的元素服从标准正态分布。

.reshape((3,4))和.reshape(3,4)都是可以的。

**.sum()用法（如何理解axis/dim）：**

.sum()求张量的元素之和，返回一个单元素张量。

使用X.item(),float(X),int(X)都能将单元素张量变成标量。另外，X.numpy()将torch张量X变成numpy数组，torch.tensor(X)将numpy数组X变成torch张量。

axis/dim设定了哪个轴，那对应的轴在拼接之后张量数会发生变化，也就是**遍历这个轴去做运算**，其他轴顺序不变：

一维张量：dim=0 表示沿着列的方向。

二维张量：dim=0 表示沿着行的方向，dim=1 表示沿着列的方向。

三维张量：dim=0 表示沿着矩阵的方向，dim=1 表示沿着行的方向，dim=2 表示沿着列的方向。

更高维度：dim=0 表示沿着最外层的维度，dim=1 表示下一层的维度，以此类推。

dim=-1 表示在张量的最后一个维度上进行操作。

可以这样理解，对于A=[[[1, 2]]]，A[0,0,1]能获取元素2，其中0,0,1分别对应着dim=0,1,2。

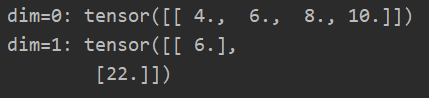
以二维张量X为例，X.sum(dim=[0,1])沿着行和列对矩阵求和，相当于X.sum()。

X.sum(dim=0)指定按矩阵X的行方向(相当于对全部样本的**某个属性**的值求和)，轴dim=0在输出中消失。

X.sum(dim=1)指定按矩阵X的列方向(相当于对**某个样本**的全部属性值求和)。

如果想要使用广播原则，如想要将全部样本的某个属性值除以该属性的平均值X / X.sum(axis=0)，可以设置参数keepdim=True，使得轴数不变。

下图是X，X.sum(dim=0/1)，X.sum(dim=0/1, keepdim=True)：

**向量或轴的维度**(dimension)既指向量或轴的**长度**(the length along a particular axis)，又指向量或轴的元素数量(the number of axes)。

**张量的维度**用来表示张量具有的**轴数，**张量的**某个轴的维数**就是这个**轴的长度**。

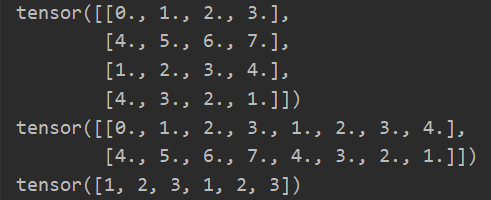
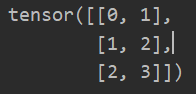
规范地，To avoid this confusion, we use *order* to refer to the number of axes and *dimensionality* exclusively to refer to the number of components.

.cumsum(dim : int)与sum类似，不过得到的结果的张量形状与之前保持一致，每个元素的值是是按这个dim的当前累加值。

**运算**

+,-,\*,/,\*\*,==,exp()都是按元素运算，都返回的是与之前大小相同的张量。

x = torch.tensor([0, 1, 2, 4])  
print(torch.exp(x)) # tensor([ 1.0000, 2.7183, 7.3891, 54.5981])  
  
X = torch.arange(8, dtype=torch.float32).reshape((2, 4))  
Y = torch.tensor([[1, 2, 3, 4], [4, 3, 2, 1]])  
Z = torch.tensor([1, 2, 3])  
**# 连结concatenate**  
print(torch.cat((X, Y), dim=0))  
print(torch.cat((X, Y), dim=1))  
print(torch.cat((Z, Z), dim=-1))  
  
**# 广播机制broadcasting mechanism：**通过适当复制元素来扩展一个或两个数组，以便在转换之后，两个张量具有相同的形状。对生成的数组执行按元素操作。  
a = torch.arange(3).reshape((3, 1))  
b = torch.arange(2).reshape((1, 2))  
print(a + b)

torch.dot(x, y)点积，torch.mm(A, B)矩阵乘法，torch.norm(X, p= )计算范数。

**内存**

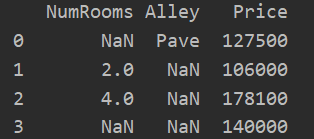
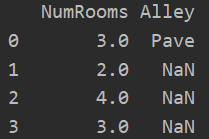
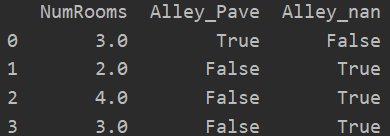
X = torch.tensor([0, 1, 2, 3])  
Y = torch.tensor([0, 3, 6, 9])  
Z = torch.tensor([1, 1, 2, 3])  
print("Y", id(Y)) # 1624687969968  
Y = Y + X  
print("Y", id(Y)) # 1624687970352  
print("Z", id(Z)) # 1624687970256  
Z[:] = Z + X  
print("Z", id(Z)) # 1624687970256  
Z += X  
print("Z", id(Z)) # 1624687970256

比如X[:]=<expression>或X +=<expression>这样的式子，不会为新结果分配内存，减少操作的内存开销。

**写入读入csv**

import os  
import pandas as pd  
import torch  
  
# 写入  
os.makedirs(os.path.join('..', 'data'), exist\_ok=True) # ..表示上一级目录  
data\_file = os.path.join('..', 'data', 'house\_tiny.csv') # E:\Py-Project\data\house\_tiny.csv  
print(f'文件的绝对路径为：{os.path.abspath(data\_file)}')  
with open(data\_file, 'w') as f:  
 f.write('NumRooms,Alley,Price\n') # 列名。房间数量、巷子类型、房屋价格  
 f.write('NA,Pave,127500\n') # 每行表示一个数据样本  
 f.write('2,NA,106000\n')  
 f.write('4,NA,178100\n')  
 f.write('NA,NA,140000\n')  
  
# 读取  
data = pd.read\_csv(data\_file)  
print(data)  
  
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]  
inputs['NumRooms'] = inputs['NumRooms'].fillna(inputs['NumRooms'].mean()) # 均值填补  
inputs = pd.get\_dummies(inputs, dummy\_na=True) # 将Alley的Pave和NaN变成两列Alley\_Pave和Alley\_nan  
X = torch.tensor(inputs.to\_numpy(dtype=float))  
y = torch.tensor(outputs.to\_numpy(dtype=float))

data，均值填补，get\_dummies依次如下：

**自动微分**

**标量、张量的不同调用方法**

如果y是一个**标量**，则不需要为backward()指定任何参数。

但是如果它有更多的元素(y是一个**张量**)，则需要指定一个gradient参数，该参数是**形状匹配**的张量。

比如X=[x1 x2], Z=X+2，可以求Z.sum().backward()。此时Z=x1+x2+4， Z对x1, x2的偏导值不变。因此也可以用一个额外的参数矩阵和需要求导的矩阵y做点乘使用全1矩阵，就等价于y.sum()了，也就是y.backward(torch.ones\_like(y))。

# y是标量  
x = torch.arange(4.0, requires\_grad=True)  
y = torch.dot(x, x)  
y.backward()  
print(x.grad) # y=x^T·x的导数应该是2x  
  
# y是张量  
x.grad.zero\_() # 在默认情况下会累积梯度，需要清除之前的梯度值  
y = x\*x+2 \* x  
y.backward(torch.ones\_like(y))  
print(x.grad)

得到的是tensor([0., 2., 4., 6.])和tensor([2., 4., 6., 8.])。

时，时。

**分离计算**

需要保留某一步的计算结果但不希望对其进行梯度计算。例如，在强调某一部分网络权重不发生变化的情况下，可以用detach来**断开梯度的计算**。

也就是：希望将y**视为一个常数**，并且只考虑x在y被计算后发挥的作用.

这里可以分离y来返回一个新变量u，该变量与y具有相同的值，但丢弃计算图中如何计算y的任何信息。换句话说，梯度不会向后流经u到x。

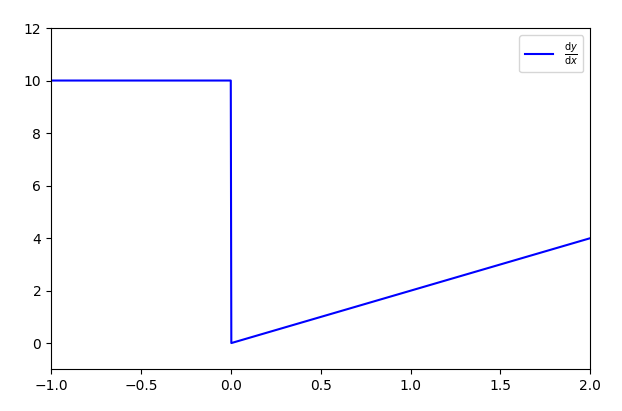
# 分离计算  
x.grad.zero\_()  
y = x \* x  
u = y.detach() # 分离y，创建一个新的张量u，与y数值相同，但与计算图分离，对u的梯度将不会在反向传播中进行计算。  
z = u \* x  
z.sum().backward()  
print(x.grad)  
x.grad.zero\_()  
y.sum().backward() # 由于记录了y的计算结果，可以在y上调用反向传播  
print(x.grad)

得到的是tensor([0., 1., 4., 9.])和tensor([0., 2., 4., 6.])

本来是，导数应该是，而断开了的信息，因此中的是一个系数而非自变量了。所以。

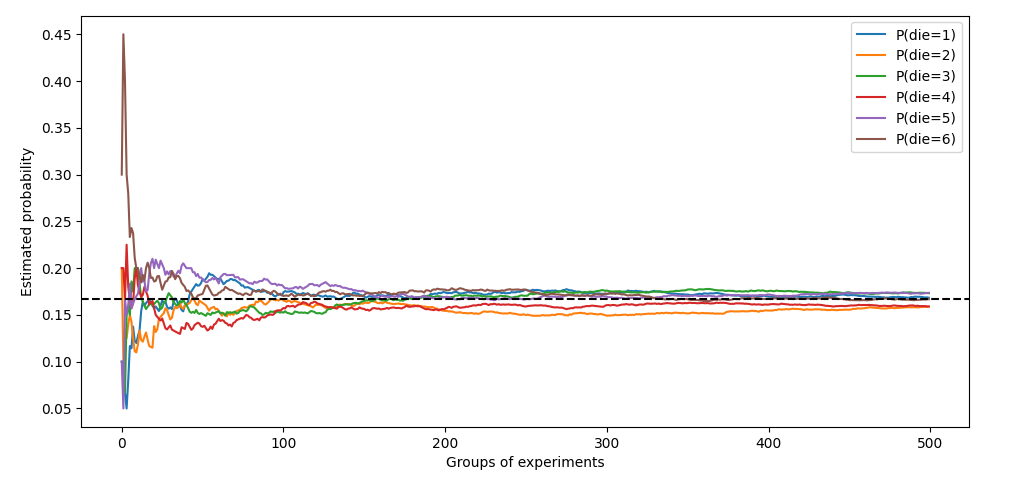
**分段函数计算**

# 分段函数计算  
# 即使构建函数的计算图需要通过Python控制流(例如，条件、循环或任意函数调用)，仍然可以计算得到的变量的梯度  
def f(a):  
 if a > 0:  
 return a \* a  
 else:  
 return 10 \* a  
a\_values = torch.randn(size=(5, 1), requires\_grad=True)  
for a\_val in a\_values:  
 a = a\_val.clone().detach().requires\_grad\_(True)  
 print("a =", a.item(), end='\t')  
 d = f(a)  
 d.backward()  
 print("Grad =", a.grad.item())  
  
# 分段函数求导绘图  
x = torch.linspace(-1, 2, 1000, requires\_grad=True)  
y = torch.where(x > 0, x \* x, 10 \* x) # 相当于将f改成def f(a): return torch.where(a > 0, a \* a, 10 \* a)  
y.backward(torch.ones\_like(y))  
xm\_draw.plot\_xy(x.detach().numpy(), x.grad.detach().numpy(), axes=(-1, 2, -1, 12), label=r'$\frac{\mathrm{d}y}{\mathrm{d}x}$')



**计算概率**

import torch  
from torch.distributions import multinomial  
import matplotlib.pylab as plt  
  
  
fair\_probs = torch.ones([6]) / 6 # tensor([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])  
# 生成多项式分布（Multinomial Distribution）的随机样本  
p1 = multinomial.Multinomial(1, fair\_probs).sample() # tensor([0., 0., 1., 0., 0., 0.])  
p10 = multinomial.Multinomial(10, fair\_probs).sample() # tensor([1., 2., 1., 2., 3., 1.])  
# 做500次实验，每次实验投掷10次骰子，记录下这10次投掷到的是1,2,3,4,5,6的次数  
counts = multinomial.Multinomial(10, fair\_probs).sample(sample\_shape=torch.Size([500])) # 大小[500, 6]  
cum\_counts = counts.cumsum(dim=0)  
estimates = cum\_counts / cum\_counts.sum(dim=1, keepdim=True)  
for i in range(6):  
 plt.plot(estimates[:, i].numpy(), label=("P(die=" + str(i + 1) + ")"))  
plt.gca().set\_xlabel('Groups of experiments') # gca:Get Current Axes 获取当前轴对象  
plt.gca().set\_ylabel('Estimated probability')  
plt.axhline(y=0.1667, color='black', linestyle='--')  
plt.legend()  
plt.show()



**计时功能**

n = 100000  
a = torch.ones([n])  
b = torch.ones([n])  
c = torch.zeros(n)  
timer = xm\_func.Timer()  
for i in range(n):  
 c[i] = a[i] + b[i]  
print(f'{timer.stop():.5f} sec')  
timer.start()  
d = a + b  
print(f'{timer.stop():.5f} sec')

**查阅文档**

查找模块中的所有函数和类：dir函数。如print(dir(torch.distributions))

查找特定函数和类的用法：help函数。如print(help(torch.ones))

**放在GPU上运行：**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

X\_train = X\_train.to(device)

……

net.to(device)  
loss.to(device)

优化器似乎不需要.to(device)。

**保存tensor、模型**

x = torch.arange(4)  
torch.save(x, 'x-file')  
y = torch.load('x-file')  
print(y)  
  
model = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))  
torch.save(model, 'model\_test.pt')  
m = torch.load('model\_test.pt')  
print(m)

**如何自定义网络继承nn.Module类**

以LeNet为例：

net = nn.Sequential(  
 nn.Conv2d(1, 6, kernel\_size=5, padding=2), nn.ReLU(),  
 nn.AvgPool2d(kernel\_size=2, stride=2),  
 nn.Conv2d(6, 16, kernel\_size=5), nn.ReLU(),  
 nn.AvgPool2d(kernel\_size=2, stride=2),  
 nn.Flatten(),  
 nn.Linear(16 \* 5 \* 5, 120), nn.ReLU(),  
 nn.Linear(120, 84), nn.ReLU(),  
 nn.Linear(84, 10))

改写成类的形式就是：

import torch.nn.functional as F

class MyNet(nn.Module):  
 def \_\_init\_\_(self):  
 super(MyNet, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 6, kernel\_size=5, padding=2)  
 self.conv2 = nn.Conv2d(6, 16, kernel\_size=5)  
 self.fc1 = nn.Linear(16 \* 5 \* 5, 120)  
 self.fc2 = nn.Linear(120, 84)  
 self.fc3 = nn.Linear(84, 10)  
 def forward(self, x):  
 x = F.relu(self.conv1(x))  
 x = F.avg\_pool2d(x, kernel\_size=2, stride=2)  
 x = F.relu(self.conv2(x))  
 x = F.avg\_pool2d(x, kernel\_size=2, stride=2)  
 x = x.view(-1, 16 \* 5 \* 5) # 将特征展平为一维向量，也就是Flatten。也可用torch.flatten(x, 1)  
 x = F.relu(self.fc1(x))  
 x = F.relu(self.fc2(x))  
 x = self.fc3(x)  
 return x  
  
net = nn.Sequential(MyNet())

然后可以使用X = torch.randn(64, 1, 28, 28)，net(X)这样的功能了。

另外，反向传播是pytorch自动实现的，不需要手动实现。

**神经网络中的shape变化：**

参考：https://pytorch.org/docs/stable/nn.html

首先约定我们输入的X统一是：

# 假设输入 X 是一个形状为 (batch\_size, channels, height, width) 的张量  
batch\_size, channels, height, width = 16, 3, 30, 30  
X = torch.randn(batch\_size, channels, height, width)

然后约定查看shape的代码是：

# 对输入 X 进行变换操作并打印形状  
for layer in net:  
 X = layer(X)  
 print(layer.\_\_class\_\_.\_\_name\_\_, 'output shape: \t', X.shape)

对于这里的输出，我们将其直接写在网络里。

**①对于简单的网络：**

net = nn.Sequential(  
 nn.Flatten(), # [16, 2700]  
 nn.Linear(channels\*height\*width, 114), # [16, 114]  
 nn.ReLU(), # [16, 114]  
 nn.Linear(114, 10), # [16, 10]  
 nn.Softmax(dim=1) # [16, 10]  
)

**展平层Flatten()**的默认值start\_dim=1, end\_dim=-1，表示dim=0(shape[0])保留，而后面的维度的维数(dimensionality)相乘，因此这里是[batch\_size, channels\*height\*width]。

**线性层Linear()**计算，填入参数是in\_features和out\_features，输入的维度[\*, in\_features]，输出维度是[\*, out\_features]，其中\*代表任意数量的维度(包括None)。

Relu、Softmax等**激活函数，**都是对数据进行数值变换，不改变shape。

**②CNN：**

net = nn.Sequential(  
 nn.Conv2d(channels, 16, kernel\_size=3, stride=1, padding=1), # [16, 16, 30, 30]  
 nn.ReLU(), # [16, 16, 30, 30]  
 nn.MaxPool2d(kernel\_size=2, stride=2), # [16, 16, 15, 15]  
 nn.BatchNorm2d(16), # [16, 16, 15, 15]  
 nn.Flatten(), # [16, 3600]  
 nn.Linear(16 \* (height//2) \* (width//2), 64), # [16, 64]  
 nn.ReLU(), # [16, 64]  
 nn.Linear(64, 10), # [16, 10]  
 nn.Softmax(dim=1) # [16, 10]  
)

**互相关Conv2d()**输入的是(N, Cin, Hin, Win)，输出的是(N, Cout, Hout, Wout)。对Hout, Wout，有：。其中指的是某**个**边上的填充，因此这里要乘2。一般上的是一样的。更详细的公式请见原文档。

Conv2d填入的参数是(in\_channels, out\_channels, kernel\_size)。

对于示例，kernel\_size=3卷积核的大小3x3，stride=1在上的步幅都为1，padding=1在上的填充都为1。代入得。又指定了Cin = channels=3，Cout=16，因此输出的维度是(16, 16, 30, 30)。

**汇聚Pool2d()**的输入输出与Conv2d()基本一致，因为汇聚也是对某个区域进行变换最后得到一个值，区域的移动过程也和互相关的移动一致。输入的是(N, C, Hin, Win)，输出的是(N, C, Hout, Wout)。对于示例，有。

**归一化BatchNorm2d()**，输入输出都是(N, C, H, W)，输入的参数是num\_features=C。

补充：因为参数已经指定了，代入得Linear的输入维度是。

**汇总表格：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **名称** | **nn.** | **输入参数** | **输入shape** | **输出shape** |
| **激活函数** | Relu、Softmax |  | \* | \* |
| **展平层** | Flatten | [start\_dim=1, end\_dim=-1] | [batch\_size, \*] | [batch\_size, ] |
| **线性层** | Linear | in\_features, out\_features | [\*, in\_features] | [\*, out\_features] |
| **互相关** | Conv2d | in\_channels, out\_channels, kernel\_size | [N, in\_channels,  h, w] | [N, out\_channels,  hout, wout] |
| **汇聚层** | Pool2d | kernel\_size | [N, C, h, w] | [N, C, hout, wout] |
| **归一化** | BatchNorm2d | channels | [N, C, H, W] | [N, C, H, W] |

表的一些约定：

①维度变量的表示：一般输入维度都是[batch\_size, channels, height, width]，偶尔使用in\_前缀来区分前后的维度名称，也会使用[N, C, H, W]，[N, C, h, w]等简化表示。

②符号：\*代表代表任意数量的维度，输入参数里的[]用于表示可选参数。

③计算：限于表格大小，公式在上文给出，表格内只给出大致流程。