1. 代码部分

①NumPy　②Pandas　③SciPy　④Matplotlib

代码部分

random 库函数	描述
random.random()	返回一个 0 到 1 之间的随机浮点数
random.randint(a, b)	返回一个在区间 [a, b] 内的随机整数
random.uniform(a, b)	返回一个在区间 [a, b] 内的随机浮点数
random.gauss(mu, sigma)	生成一个服从一元高斯分布 (univariate Gaussian distribution) 的随机数，其中 mu 是均值，sigma 是标准差
random.seed(seed)	使用给定的种子 seed 初始化随机数生成器，这有助于结果可复刻
random.choice(seq)	从序列 (如列表、元组或字符串) 中随机选择一个元素并返回它
random.choices(population, weights=None, k=k)	从给定的 population 序列中随机选择 k 个元素，可以通过 weights 参数指定每个元素的权重，权重越高的元素被选中的概率越大。如果不指定权重，所有元素被选中的概率相等
random.shuffle(sq)	用于随机打乱序列 seq 中的元素顺序
random.sample(population, k)	从指定的 population 序列中随机选择 k 个不重复的元素，相当于从总体中采集 k 个不重复的样本
random.betavariate(alpha, beta)	用于生成一个服从 Beta 分布 (Beta distribution) 的随机数。Beta 分布是一个概率分布，其形状由两个参数 alpha 和 beta 来控制。《统计至简》将专门介绍这个概率分布，并在贝叶斯推断 (Bayesian inference) 中使用 Beta 分布
random.expovariate(lambd)	用于生成一个服从指数分布 (exponential distribution) 的随机数，lambd 是指数分布的一个参数。《统计至简》将介绍指数分布

statistics 库函数	描述
statistics.mean()	计算算数平均值 (arithmetic mean, average)
statistics.median()	计算中位数 (median)
statistics.mode()	计算众数 (mode)
statistics.quantiles()	用于计算分位数 (quantile) 的函数。简单来说，分位数是指将一组数据按照大小顺序排列后，把数据分成若干部分的值，每一部分包含了一定比例的数据。通常，我们使用四分位数来分割数据集，这将数据集分为四个部分，分别包含 25%、50%、75%和 100%的数据。
statistics.pstdev()	计算数据的总体标准差 (population standard deviation)
statistics.stdev()	计算数据的样本标准差 (sample standard deviation)
statistics.pvariace()	计算数据的总体方差 (population variance)
statistics.variance()	计算数据的样本方差 (sample variance)
statistics.covariance()	计算数据的样本协方差 (sample covariance)
statistics.correlation()	计算数据的皮尔逊相关性系数 (Pearson's correlation coefficient)
statistics.linear_regression()	计算一元线性回归函数斜率 (slope) 和截距 (intercept)

一、基础部分

1.NumPy

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

表 1. 用 Numpy 构造行向量

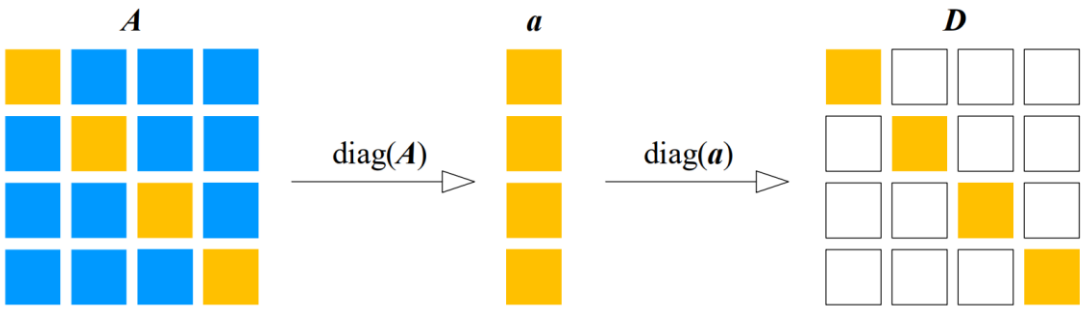
代码	注意事项
`a = numpy.array([4,3])`	严格地说，这种方法产生的并不是行向量；运行 `a.ndim` 发现 a 只有一个维度。因此，转置 `numpy.array([4,3]).T` 得到的仍然是一维数组，只不过默认展示方式为行向量。
`a = numpy.array([[4,3]])`	运行 `a.ndim` 发现 a 有二个维度，这个行向量转置 `a.T` 可以获得列向量。`a.T` 求 a 转置，等价于 `a.transpose()`。 请大家注意双重方括号。
`a = numpy.array([4,3], ndmin=2)`	`ndmin=2` 设定数据有两个维度，转置 `a.T` 可以获得列向量。
`a = numpy.r_['r', [4,3]]`	`numpy.r_[]` 将一系列数组合并；`'r'` 设定结果以行向量（默认）展示，比如 `numpy.r_[numpy.array([1,2]), 0, 0, numpy.array([4,5])]` 默认产生行向量。
`a = numpy.array([4,3]).reshape((1, -1))`	`reshape()` 按某种形式重新排列数据，`-1` 自动获取数组元素个数 n。
`a = numpy.array([4, 3])[None, :]`	按照 `[None, :]` 形式广播数组，None 代表 `numpy.newaxis`，增加新维度。
`a = numpy.array([4, 3])[numpy.newaxis, :]`	等同于上一例。

表 2. 用 Numpy 构造列向量

代码	注意事项
`a = numpy.array([[4], [3]])`	运行 `a.ndim` 发现 a 有二个维度。`numpy.array([[4], [3]]).T` 获得行向量。请大家注意两层方括号。
`a = numpy.r_['c', [4,3]]`	`numpy.r_[]` 将一系列的数组合并。`'c'` 设定结果以列向量展示
`a = numpy.array([4,3]).reshape((-1, 1))`	`reshape()` 按某种形式重新排列数据；`-1` 自动获取数组元素个数 n
`a = numpy.array([4, 3])[:, None]`	按照 `[:, None]` 形式广播数组；None 代表 `numpy.newaxis`，增加新维度
`a = numpy.array([4, 3])[:, numpy.newaxis]`	等同于上一例

`Bk4_Ch4_01.py` 介绍如何用不同方式构造矩阵。注意，numpy.matrix() 和 numpy.array() 都可以构造矩阵。但是两者结果有显著区别。numpy.matrix() 产生的数据类型是严格的 2 维<class 'numpy.matrix'>；而 numpy.array()产生的数据可以是 1 维、2 维、乃至 *n* 维，类型统称为<class 'numpy.ndarray'>。此外，在乘法和乘幂运算时，这两种不同方式构造的矩阵也会有明显差别，本章后续将逐步介绍。

本书还常用 diag() 函数。如图 7 所示，diag(***A***) 提取矩阵 ***A*** 主对角线元素，结果为列向量。此外，diag(***a***) 将向量 ***a*** 展成对角方阵 ***D***，***D*** 主对角线元素依次为向量 ***a*** 元素。



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

列向量和行向量之和

利用广播原则，列向量可以和行向量相加。

如图 12 所示，列向量 $\boldsymbol{c}$ 自我复制，左右排列得到矩阵的列数和 $\boldsymbol{r}$ 列数一致。行向量 $\boldsymbol{r}$ 自我复制，上下叠加得到矩阵和 $\boldsymbol{c}$ 的行数一致。然后完成加法运算，比如：

$$\begin{bmatrix}3\\2\\1\end{bmatrix}+\begin{bmatrix}2 & 1\end{bmatrix}=\begin{bmatrix}3 & 3\\2 & 2\\1 & 1\end{bmatrix}+\begin{bmatrix}2 & 1\\2 & 1\\2 & 1\end{bmatrix}=\begin{bmatrix}3+2 & 3+1\\2+2 & 2+1\\1+2 & 1+1\end{bmatrix}=\begin{bmatrix}5 & 4\\4 & 3\\3 & 2\end{bmatrix} \tag{19}$$

Bk4_Ch4_06.py 介绍如何借助 Numpy 完成矩阵乘法运算。值得注意的是，对于两个由 numpy.array() 产生的数据，使用*相乘，得到的乘积是对应元素分别相乘，广播法则有效；而两个由 numpy.matrix() 产生的 2 维矩阵，使用*相乘，则得到结果等同于@。如果，分别由 numpy.array() 和 numpy.matrix() 产生的数据，使用*相乘，则等同于@。请大家运行 Bk4_Ch4_07.py 给出的三个乘法例子，自行比较结果。

Bk4_Ch4_08.py 展示如何计算矩阵幂。乘幂运算符**对 numpy.array() 和 numpy.matrix() 生成的数据有不同的运算规则。numpy.matrix() 生成矩阵 A，A**2，是矩阵乘幂；numpy.array() 生成的矩阵 B，B**2 是对矩阵 B 元素分别平方。请大家比较 Bk4_Ch4_09.py 给出的两个例子。

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]]
[[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。

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]]
```

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]
 [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

```
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)
```

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 $N \sim (0,1)$ 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 $x \sim Beta(a,b)$，由两个参数 a,b 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 $c_{ij}=a_{ij}\times b_{ij}$：

$$如\begin{pmatrix} a & b \\ c & d \end{pmatrix}*\begin{pmatrix} A & B \\ C & D \end{pmatrix}=\begin{pmatrix} aA & bB \\ cC & dD \end{pmatrix}$$

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：

$$如\begin{pmatrix} a \\ b \end{pmatrix}*\begin{pmatrix} A & B \\ C & D \end{pmatrix}=\begin{pmatrix} aA & aB \\ bC & bD \end{pmatrix}$$

2. 点乘 dot

就是线代里的矩阵乘法

比如：

```
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))
```

输出

```
[[ 5 12]
 [21 32]]

[[ 5  6]
 [14 16]]
```

矩阵的逆矩阵：

与线代的定义一样，$AA^{-1}=E$，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     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

```
a    0.0
b    1.0
c    2.0
Name: A, dtype: float64
```

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 $xy=10, x>0, y>0$ 的条件下求 $\min x^2+2y$。

易得 $f(x)=x^2+2y=x^2+\frac{20}{x} \Rightarrow \frac{\partial f(x)}{\partial x}=2x-\frac{20}{x^2}=0 \Rightarrow x=\sqrt[3]{10}$。

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 $\min\int(x^2+2x+\sin x)dx \Rightarrow \frac{\partial f(x)}{\partial x}=x^2+2x+\sin x=0 \Rightarrow x=0$。

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)。

表 1. 比较 plt 和 ax 函数

功能	plt 函数 import matplotlib.pyplot as plt	ax 函数 fig, ax = plt.subplots() fig, axes = plt.subplots(n_rows, n_cols) ax = axes[row_num][col_num]
创建新的图形	plt.figure()	
创建新的子图	plt.subplot()	ax = fig.add_subplot()
创建折线图	plt.plot()	ax.plot()
添加横轴标签	plt.xlabel()	ax.set_xlabel()
添加纵轴标签	plt.ylabel()	ax.set_ylabel()
添加标题	plt.title()	ax.set_title()
设置横轴范围	plt.xlim()	ax.set_xlim()
设置纵轴范围	plt.ylim()	ax.set_ylim()
添加图例	plt.legend()	ax.legend()
添加文本注释	plt.text()	ax.text()
添加注释	plt.annotate()	ax.annotate()
添加水平线	plt.axhline()	ax.axhline()
添加垂直线	plt.axvline()	ax.axvline()
添加背景网格	plt.grid()	ax.grid()
保存图形到文件	plt.savefig()	通常使用 fig.savefig()

colormap：颜色映射/色谱：

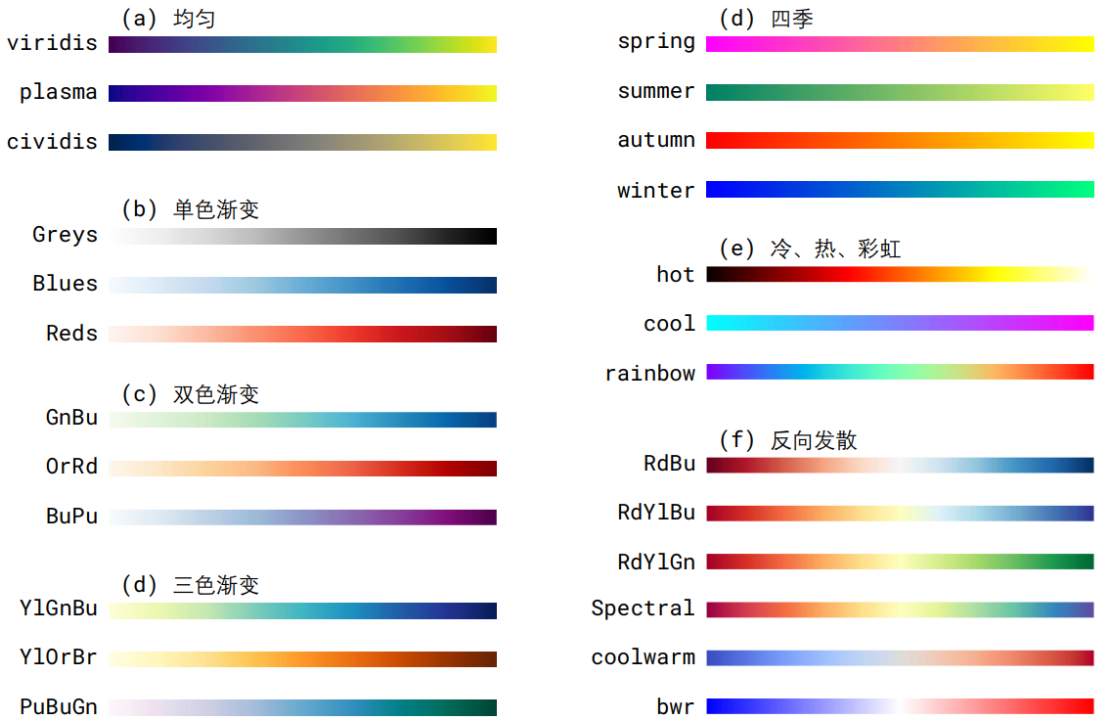


图 9. 几种常用色谱

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)：

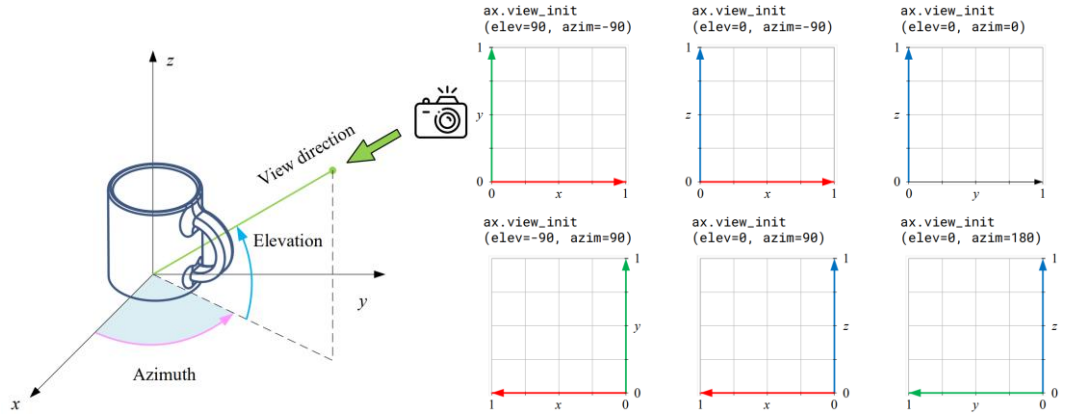
contourf 是在 contour 的基础上进行了填充操作。

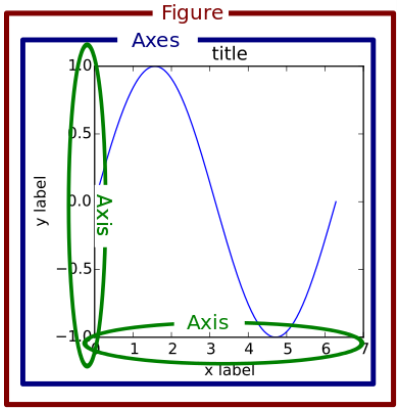
X：二维数组，表示数据点的横坐标。
Y：二维数组，表示数据点的纵坐标。
Z：二维数组，表示数据点对应的函数值或高度。
levels：用于指定绘制的等高线层级或数值列表。

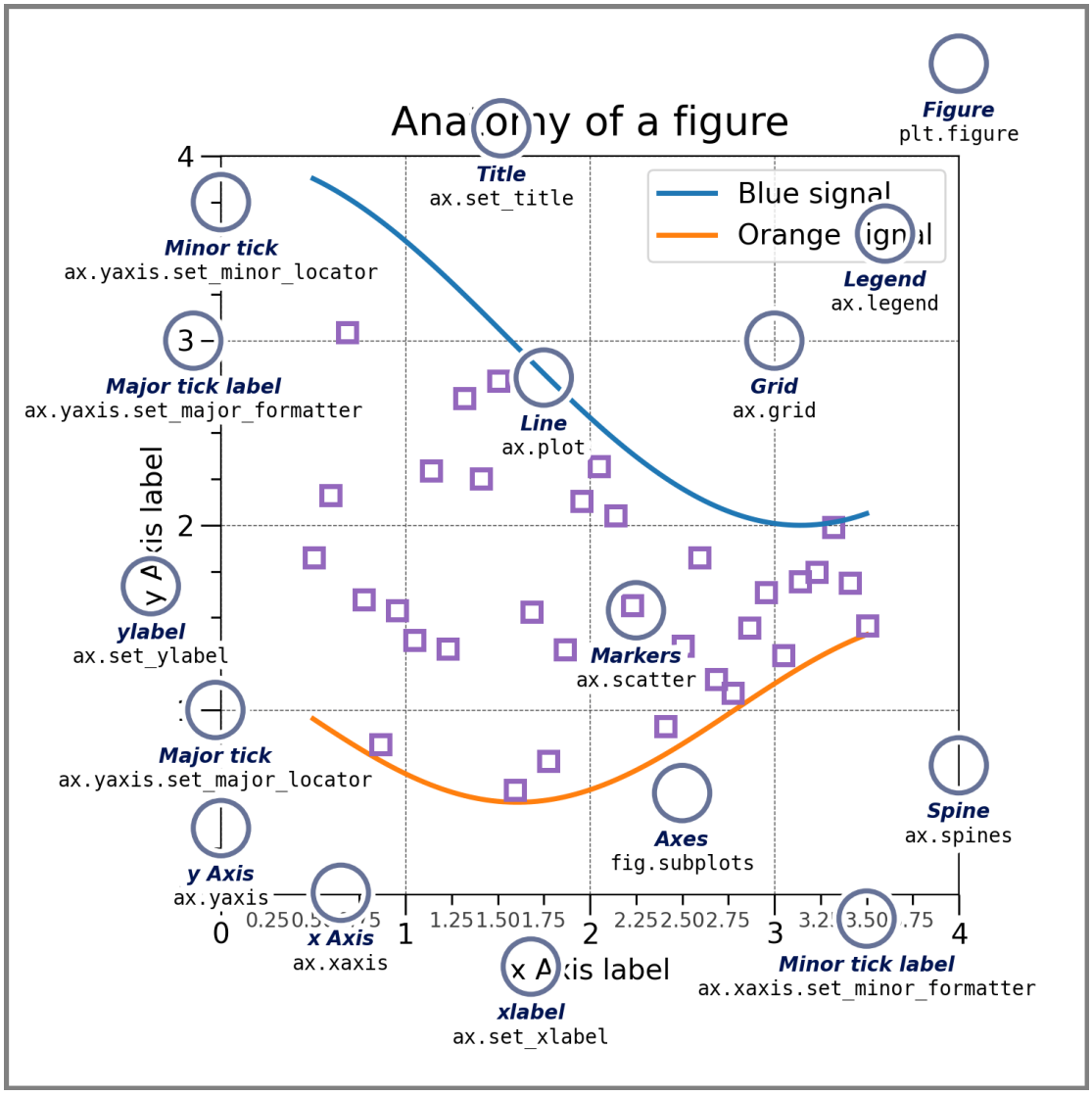
colors：用于指定等高线的颜色，可以是单个颜色字符串、颜色序列或 colormap 对象。
cmap：颜色映射，用于将数值映射为颜色。可以是预定义的 colormap 名称或 colormap 对象。
linestyles：用于指定等高线的线型，可以是单个线型字符串或线型序列。
linewidths：用于指定等高线的线宽，可以是单个线宽值或线宽序列。
alpha：用于指定等高线的透明度。

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。

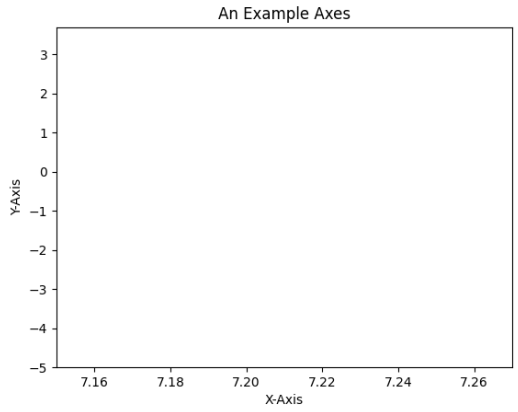


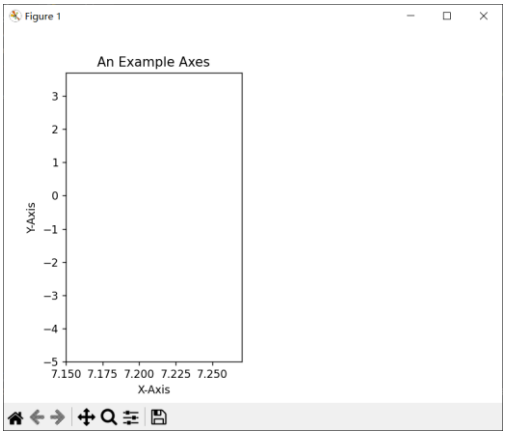




绘制坐标轴

```
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

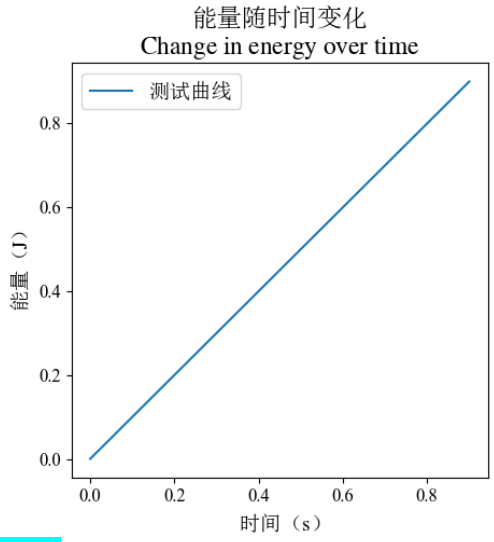
```
import matplotlib.pyplot as plt
from matplotlib import rcParams

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()
```

宋体 Times New Roman $\alpha_i > \beta_i$

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

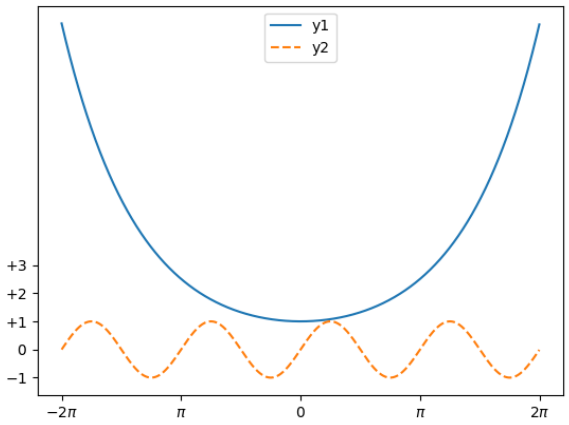
```
import matplotlib.pyplot as plt

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()
```

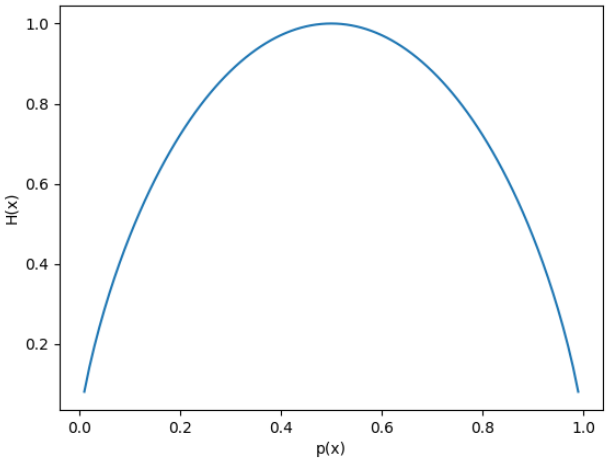


计算二元信源的熵

伯努利分布的熵 $H(U) = -P\log P - (1-P)\log(1-P)$

```
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 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 $N(0,1)$ 。

.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):

```
tensor([[0., 1., 2., 3.],
        [4., 5., 6., 7.]])
```

```
dim=0: tensor([ 4.,  6.,  8., 10.])
dim=1: tensor([ 6., 22.])
```

```
dim=0: tensor([[ 4.,  6.,  8., 10.]])
dim=1: tensor([[ 6.],
        [22.]])
```

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)
```

```
tensor([[0., 1., 2., 3.],
        [4., 5., 6., 7.],
        [1., 2., 3., 4.],
        [4., 3., 2., 1.]])
tensor([[0., 1., 2., 3., 1., 2., 3., 4.],
        [4., 5., 6., 7., 4., 3., 2., 1.]])
tensor([1, 2, 3, 1, 2, 3])
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

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：

	NumRooms	Alley	Price
0	NaN	Pave	127500
1	2.0	NaN	106000
2	4.0	NaN	178100
3	NaN	NaN	140000

	NumRooms	Alley
0	3.0	Pave
1	2.0	NaN
2	4.0	NaN
3	3.0	NaN

	NumRooms	Alley_Pave	Alley_nan
0	3.0	True	False
1	2.0	False	True
2	4.0	False	True
3	3.0	False	True

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.])。

$y=\boldsymbol{x}^T\boldsymbol{x}$ 时 $\frac{\mathrm{d}y}{\mathrm{d}\boldsymbol{x}}=2\boldsymbol{x}$， $y=\boldsymbol{x}^2+2\boldsymbol{x}$ 时 $\frac{\mathrm{d}y}{\mathrm{d}\boldsymbol{x}}=2\boldsymbol{x}+2$。

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
```

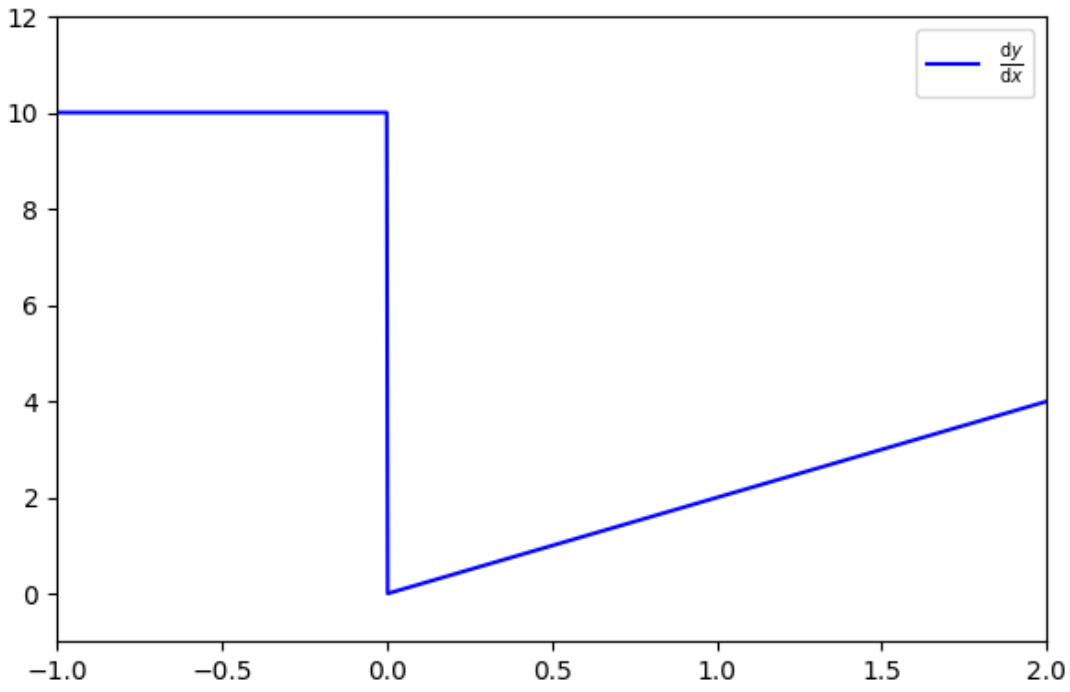
```
会在反向传播中进行计算。
z = u * x
z.sum().backward()
print(x.grad)
x.grad.zero_()
y.sum().backward()  # 由于记录了 y 的计算结果，可以在 y 上调用反向传播
print(x.grad)
```

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

本来是 $z=ux=yx=x^2x=x^3$，导数应该是 $3x^2$，而 u 断开了 y 的信息，因此 $z=x^2x$ 中的 x^2 是一个系数而非自变量了。所以 $\frac{\mathrm{d}z}{\mathrm{d}x}=x^2$。

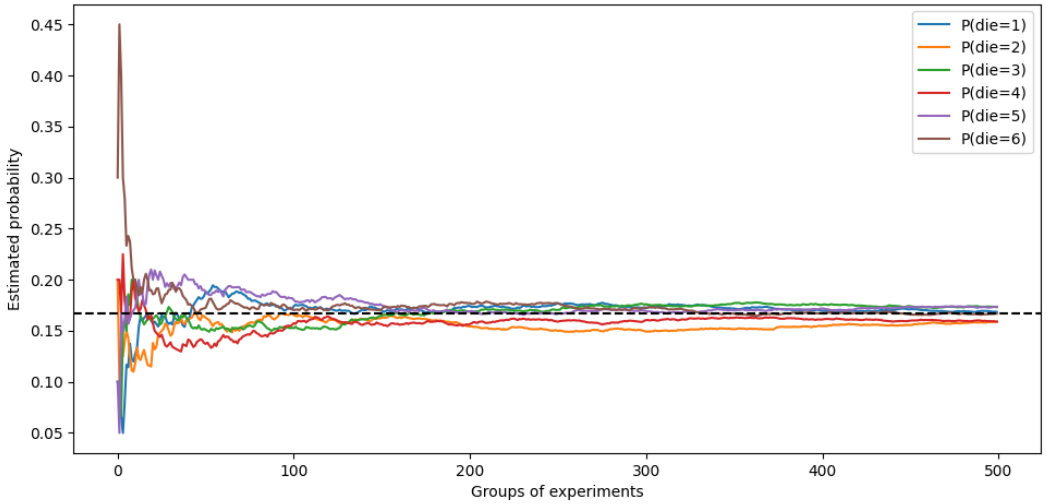
分段函数计算

```
# 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))

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 $y = xA^T + b$，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, C_{in}, H_{in}, W_{in})，输出的是(N, C_{out}, H_{out}, W_{out})。对 H_{out}, W_{out}，有：$H_{out} = \left\lfloor \frac{H_{in} - k_h + 2p_h + s_h}{s_h} \right\rfloor, W_{out} = \left\lfloor \frac{W_{in} - k_w + 2p_w + s_w}{s_w} \right\rfloor$。其中 $p_{h,w}$ 指的是某个边上的填充，因此这里要乘 2。一般 h,w 上的 k, p, s 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 h,w 上的步幅都为 1，padding=1 在 h,w 上的填充都为 1。代入得 $H_{out} = W_{out} = \left\lfloor \frac{30-3+2+1}{1} \right\rfloor = 30$。又指定了 C_{in} = channels=3，C_{out}=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, H_{in}, W_{in})，输出的是(N, C, H_{out}, W_{out})。对于示例，有 $H_{out} = W_{out} = \left\lfloor \frac{30-2+2+1}{2} \right\rfloor = 15$。

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 k, p, s 已经指定了，代入得 Linear 的输入维度是 $C_{out} \times \lfloor h/2 \rfloor \times \lfloor w/2 \rfloor$。

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, $\prod$*]
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, h_{out}, w_{out}]
汇聚层	Pool2d	kernel_size	[N, C, h, w]	[N, C, h_{out}, w_{out}]
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。