Pandas基础 （C05）

2020/10/28 胡俊峰

北京大学信息科学技术学院

主要内容

- Numpy（补充）
- Pandas 基础
- 数据特征分析与 PCA、SVD

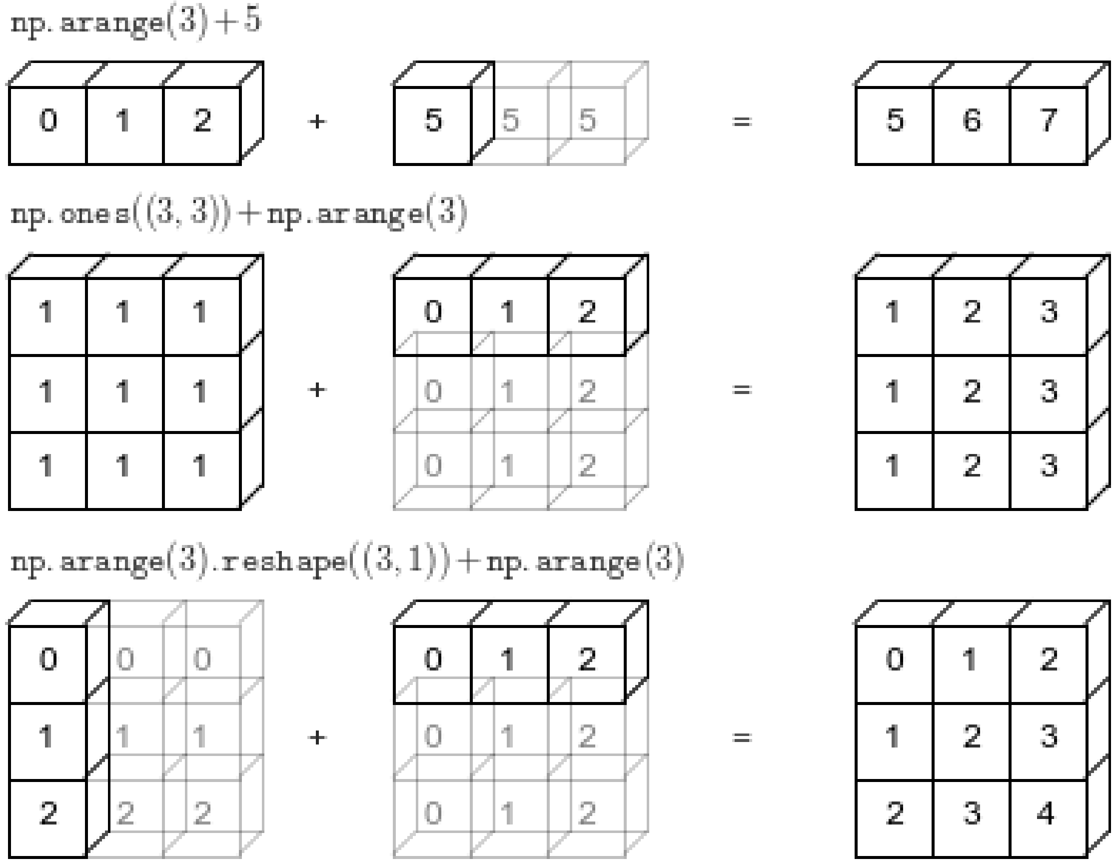
Universal functions (ufunc)

A universal function (or ufunc for short) is a function that operates on **ndarrays** in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a "vectorized" wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the **numpy.ufunc** class. Many of the built-in functions are implemented in compiled C code. The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions. One can also produce custom **ufunc** instances using the **frompyfunc** factory function.

Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs (where an element is generally a scalar, but can be a vector or higher-order sub-array for generalized ufuncs). Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on.



```python
import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]  # 求倒数
    return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)
```

```
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
```

```python
1.0 / values          # 效率高3倍
```

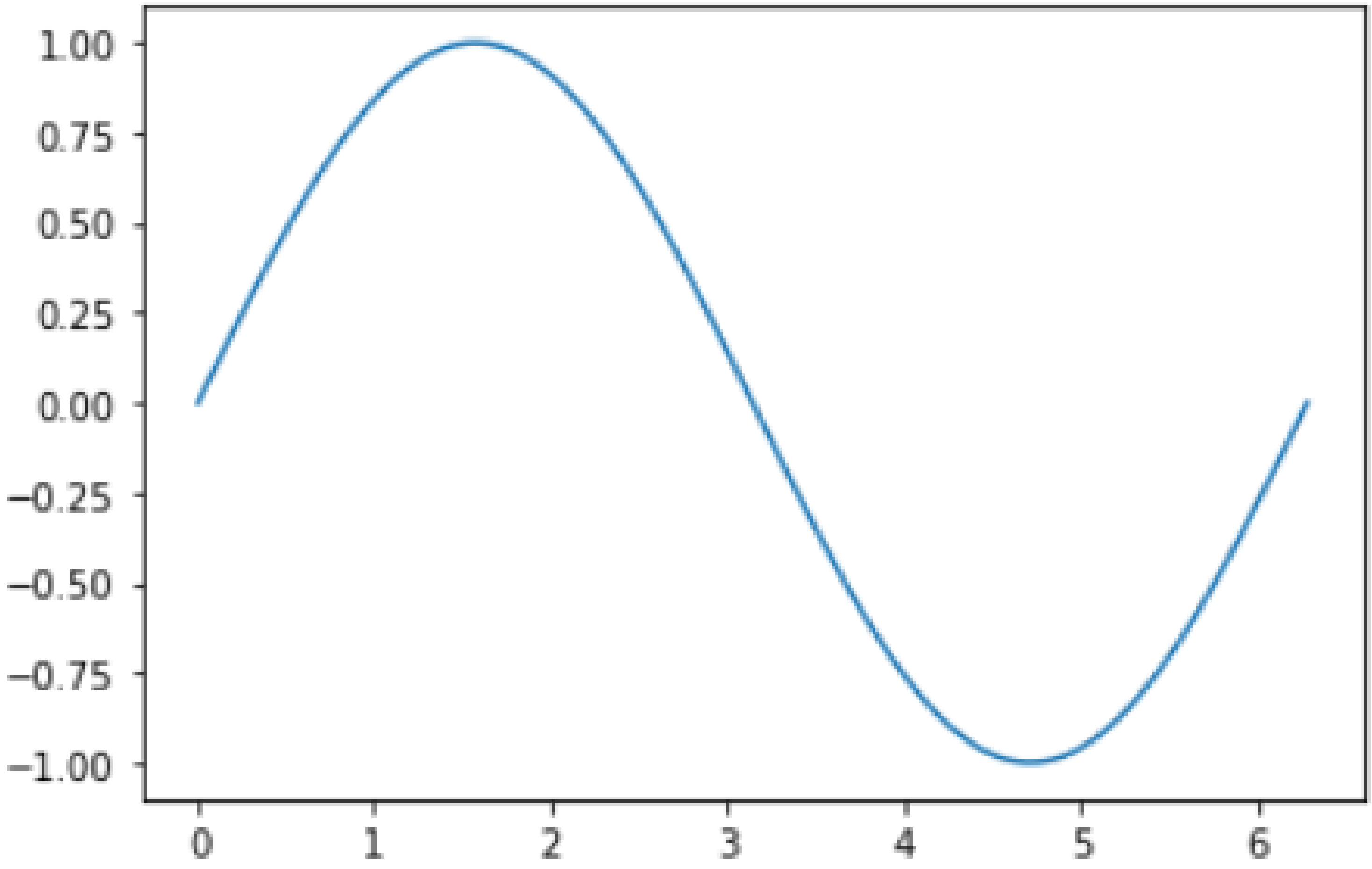
https://docs.python.org/3/library/timeit.html

```
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
```

```python
theta = np.linspace(0, np.pi * 2, 100)
y = np.sin(theta)   # 单目函数 广播

plt.plot(theta,y)
```

Out[4]: [<matplotlib.lines.Line2D at 0x23bc9911fc8>]

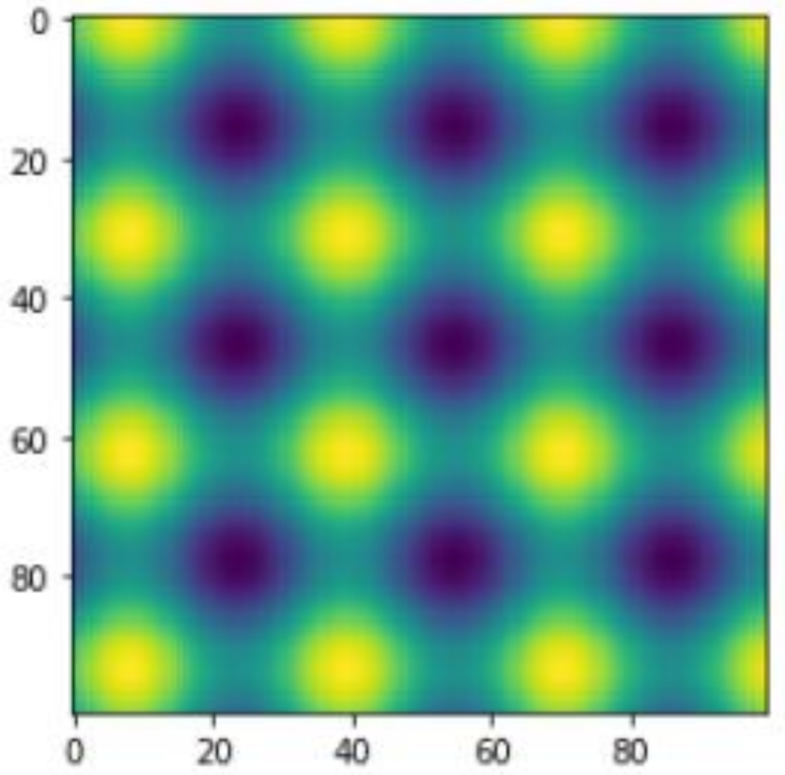


```
# define a function z = f(x, y)
# x and y have 100 steps from 0 to 10

x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)[:, np.newaxis]  # 增加一个维度

z = np.sin(2*x) + np.cos(2*y) # 广播合成2维度矩阵
plt.imshow(z)
```

<matplotlib.image.AxesImage at 0x23bca086c88>




```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

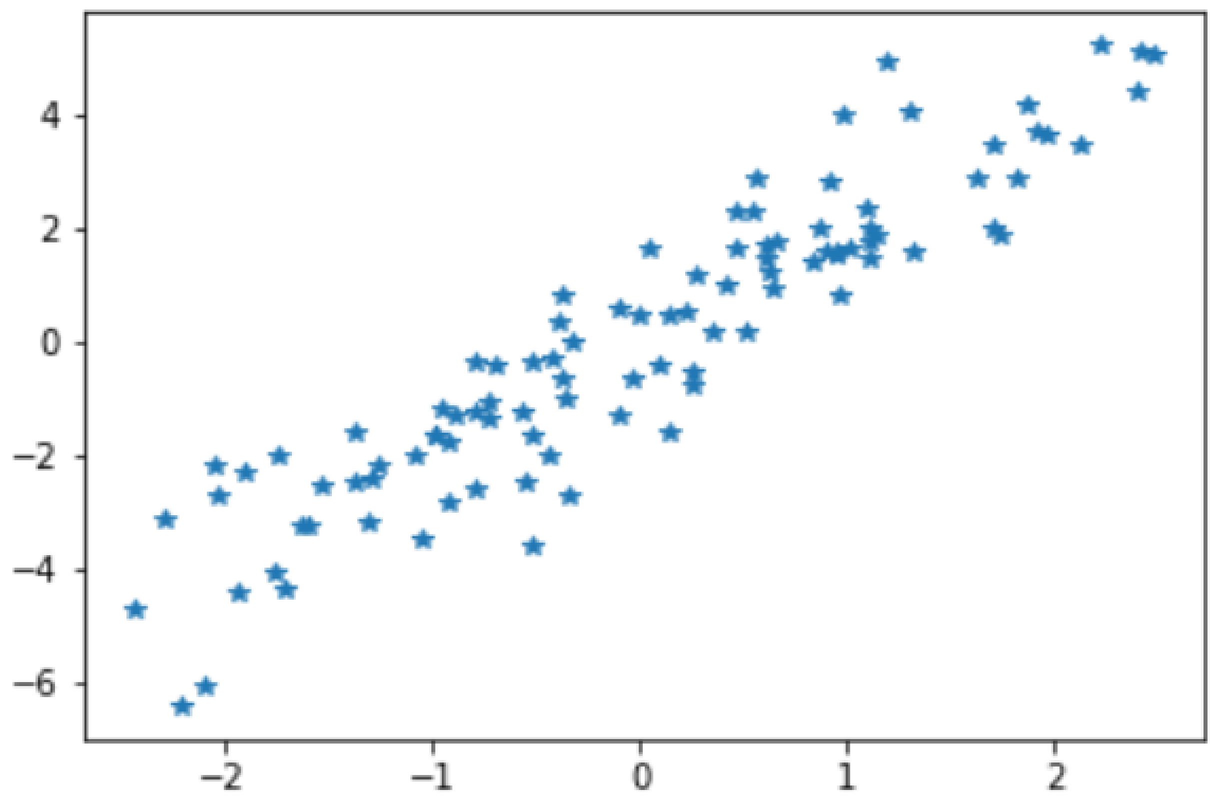
Aggregation functions（聚合函数）

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements

中心化

```python
def centerData(X):
    X = X.copy()
    X -= np.mean(X, axis = 0)
    return X

X_centered = centerData(X)
plt.plot(X_centered[:,0], X_centered[:,1], '*')
plt.show()
```



Pandas —— Panel data analysis

- 序列：indexed list
- 多通道序列：record list
- 多字段二维表
- 表关联运算

The Pandas Series Object —— 序列

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

缺省情况类似excel的表格，自动维护标号索引

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
data.index
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

与数组类似，支持下标访问操作

```
data.values
```

```
array([ 0.25,  0.5 ,  0.75,  1.  ])
```

The `index` is an array-like object of type `pd.Index`

```
data[1]
```

```
0.5
```

```
data[1:3]
```

```
1    0.50
2    0.75
dtype: float64
```

也可以指定可哈希的索引项，类似dict

```
data = pd.Series([0.5, 0.25, 1.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print(data)
print(data.sort_values())
data['b']
```

```
a    0.50
b    0.25
c    1.75
d    1.00
dtype: float64
b    0.25
a    0.50
d    1.00
c    1.75
dtype: float64

0.25
```

We can even use non-contiguous or non-sequential indices:

```python
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data[5]
```

```
0.5
```

```python
data[data>0.7] * 2
```

```
3    1.5
7    2.0
dtype: float64
```

```python
print(0.75 in data)
0.75 in data.values
```

```
False
```

```
True
```

```python
for i in data.values:
    print(i)
```

```
0.25
0.5
0.75
1.0
```

```
True
```

```python
a = pd.Series([2, 4, 6])
b = pd.Series({2:'a', 1:'b', 3:'c'})
print(b[1])
2 in b
```

```
b
```

```
True
```

```python
for i in b:
    print (i)
```

```
a
b
c
```

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
print (obj3)
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

插入索引

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

由于"California"所对应的sdata值找不到，所以其结果就为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值）。因为‘Utah’不在states中，它被从结果中除去。

```python
# Series最重要的一个功能是，它会根据运算的索引标签自动对齐数据
# 关于数据对齐功能如果你使用过数据库，可以认为是类似join的操作
obj3+obj4
```

```
California          NaN
Ohio            70000.0
Oregon          32000.0
Texas          142000.0
Utah                NaN
dtype: float64
```

```python
obj3 - obj4
```

```
California    NaN
Ohio          0.0
Oregon        0.0
Texas         0.0
Utah          NaN
dtype: float64
```

The Pandas DataFrame Object

- 视角1：多个对齐的序列（series）的组合
- 视角2：多维度的 Numpy array 支持 **多维度索引**
- 视角3：多帧数据的序列，每帧数据是一个Numpy array

索引-数据 与 索引合并

```python
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

```python
states = pd.DataFrame({'population': population,'area': area})
states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

```python
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127, 'W.DC': 11000000}
population = pd.Series(population_dict)

area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

```python
states = pd.DataFrame({'population': population,'area': area})
states
```

	population	area
California	38332521.0	423967.0
Florida	NaN	170312.0
Illinois	NaN	149995.0
New York	19651127.0	141297.0
Texas	26448193.0	695662.0
W.DC	11000000.0	NaN

索引-数据 与 索引合并（续）

```python
print(states.index)
print(states.columns)
for i in states.columns:
    print(states[i])
```

多重索引

```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas', 'W.DC'], dtype='ob
ject')
Index(['population', 'area'], dtype='object')
California    38332521.0
Florida              NaN
Illinois             NaN
New York      19651127.0
Texas         26448193.0
W.DC          11000000.0
Name: population, dtype: float64
California    423967.0
Florida       170312.0
Illinois      149995.0
New York      141297.0
Texas         695662.0
W.DC               NaN
Name: area, dtype: float64
```

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA/缺失值

词典的列表生成dataframe：

If some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values:

```python
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

	a	b
0	0	0
1	1	2
2	2	4

```python
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

Pandas 进阶：Index Object 与 表间关联

- 不可修改的数组
- 有序
- 支持可重复 key
- 表关联操作

Index as ordered set（支持表关联计算的基础）

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

```
indA & indB  # intersection
```

```
Int64Index([3, 5, 7], dtype='int64')
```

```
indA | indB  # union
```

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

```
indA ^ indB  # symmetric difference
```

```
Int64Index([1, 2, 9, 11], dtype='int64')
```

Indexers: loc, iloc, and ix

按位置索引

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
# explicit index when indexing
data[1]
```

```
'a'
```

```
# implicit index when slicing
data[1:3]
```

```
3    b
5    c
dtype: object
```

```
print(data.loc[1])
print(data.iloc[1])

```

```
a
b
```

Data Selection in DataFrame

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

```
1  data['area']
```

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

用法：字段名 类比 属性

Equivalently, we can use attribute-style access with column names that are strings:

```
1  data.area
```

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

```python
data['density'] = data['pop'] / data['area']
data
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

行列互换

```
s = data.T
print(s)
s['California']
```

```
         California         Texas      New York       Florida      Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05  1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07  1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02  8.588376e+01
```

```
area       4.239670e+05
pop        3.833252e+07
density    9.041393e+01
Name: California, dtype: float64
```

筛选，赋值：

```python
data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

```python
data.iloc[0, 2] = 90
data
```

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Working with NumPy ufunc

```
df = pd.DataFrame(rng.randint(0, 10, (3,4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	9	2	6	7
1	4	3	7	7
2	2	5	4	1

```
np.sin(df * np.pi / 4)
```

采用Numpy的Ufunc —— 广播机制

	A	B	C	D
0	7.071068e-01	1.000000	-1.000000e+00	-0.707107
1	1.224647e-16	0.707107	-7.071068e-01	-0.707107
2	1.000000e+00	-0.707107	1.224647e-16	0.707107

Dataframe之间的运算自动进行索引对齐-补足（out join）

Out[22]:

	A	B
0	2	4
1	18	6

In [23]:

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

Out[23]:

	B	A	C
0	4	8	6
1	1	3	8
2	1	9	8

In [24]:

```
A + B
```

Out[24]:

	A	B	C
0	10.0	8.0	NaN
1	21.0	7.0	NaN
2	NaN	NaN	NaN

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Frame 与 series 计算，按行broadcasting

```
A = rng.randint(10, size=(3, 4))
A
```

```
array([[9, 4, 1, 3],
       [6, 7, 2, 0],
       [3, 1, 7, 3]])
```

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	-3	3	1	-3
2	-6	-3	6	0

```
df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	5	0	-3	-1
1	-1	0	-5	-7
2	2	0	6	2

运算过程中类型自适应转换

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in Data Indexing and Selection, Boolean masks can be used directly as a `Series` or `DataFrame` index:

```
data[data.notnull()]
```

```
0        1
2    hello
dtype: object
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill
data.fillna(method='ffill')
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

层次-组合 索引（ Hierarchical-Indexing ）

```python
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

类似二维表切片

```python
pop[:, 2010]
```

```
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

MultiIndex VS extra dimension

```
#unstack() method will quickly convert a multiply indexed Series
#into a conventionally indexed DataFrame:
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

```
#unstack() method will quickly convert a multiply indexed Series into a conventi
pop_df.stack()
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

```python
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

```python
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

```
]:  df = pd.DataFrame(np.random.rand(4, 2),
                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                      columns=['data1', 'data2'])
    df
```

]:

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

MultiIndex constructor

```
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples giving the multiple index values of each point:

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

数据特征与主成分分解

- 特征的信息量与区分度
- 特征正交化与PCA降维
- 数据分析应用与数据可视化

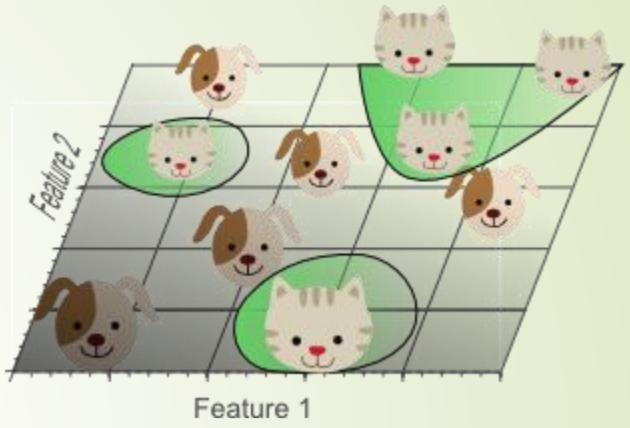
数据特征与数据维度

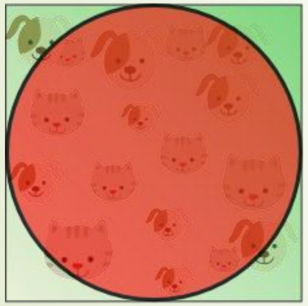
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
50	7.0	3.2	4.7	1.4	versicolor
51	6.4	3.2	4.5	1.5	versicolor
100	6.3	3.3	6.0	2.5	virginica
101	5.8	2.7	5.1	1.9	virginica

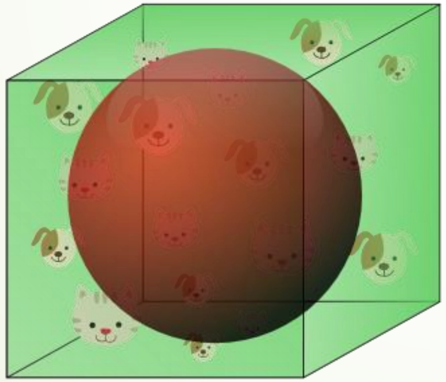
数据特征维度是否是越多越好呢？

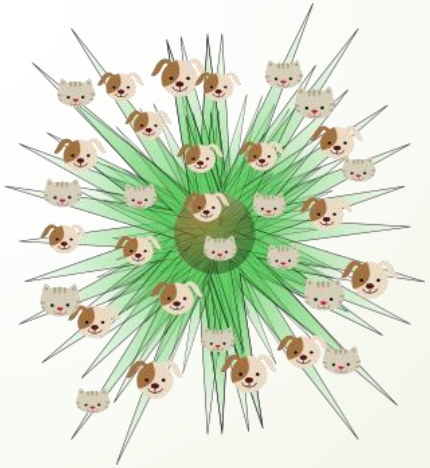
维度灾难

1. 高维度下，数据样本稀疏，难以做到密采样，易过拟合
2. 在高维空间，特征间的某些距离测量逐渐失效









回顾两个数学概念：概率 - 信息量

- 概率： $P_i = F_i / \sum_i F_i$ （古典概型，也称频率模型）
- 信息量：$H_i = -\log P_i$

概率越小，信息量越大，概率越大，信息量越小

再进一步：编码系统的信息量（信息熵）

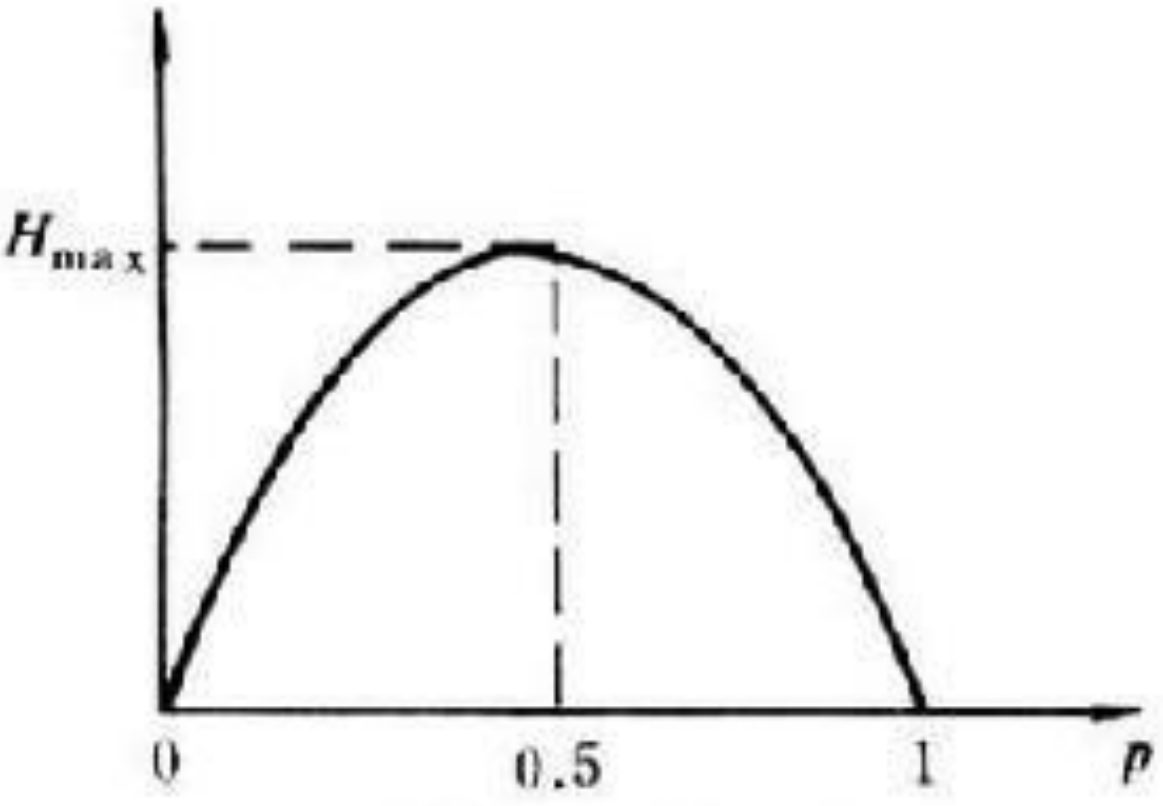
$$H(U) = E[-\log p_i] = -\sum_{i=1}^{n} p_i \log p_i$$

2元编码系统均匀分布的信息熵：

H2 = 2*（- 1/2 log（1/2））= 1 bit

4元编码系统均匀分布的信息熵

= ？

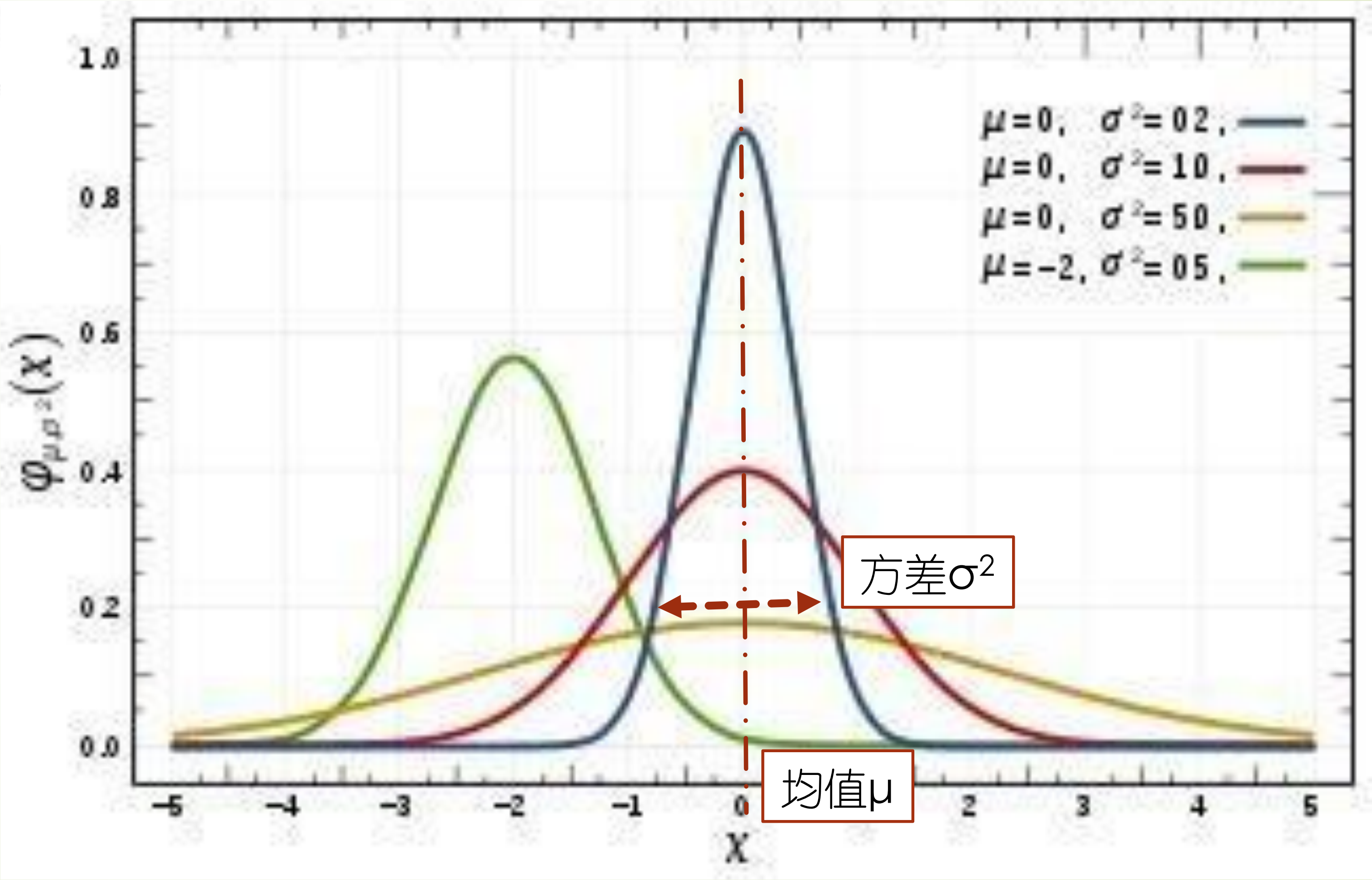


二元信源的熵函数

概率分布 - 方差 – 特征区分度

$$p(y)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

一个特征分布的方差越大，
其信息量即区分度也就越高



特征的协方差矩阵

- $C_{\vec{X}}(i;j) = Cov_{X_i,X_j} = E\left(\left(X_i - E(X_i)\right)\left(X_j - E(X_j)\right)\right)$
- 对于样本$\vec{X}^{(1)},\dots,\vec{X}^{(N)}$,常常先让$\vec{X}^{(i)} \leftarrow \vec{X}^{(i)} - \frac{1}{N}\sum_{i=1}^{N}\vec{X}^{(i)}$ （中心化）
- 记样本矩阵为$X = \begin{pmatrix} -\vec{X}^{(1)}- \\ -\vec{X}^{(2)}- \\ \dots \\ -\vec{X}^{(N)}- \end{pmatrix}$
- 可以估计协方差矩阵如下：

  $C(p;q) = \frac{1}{N}\sum_{k=1}^{N} X_p^{(k)} X_q^{(k)}$ （向量两两相乘）

则

$$C = X^T X / N$$

协方差矩阵的物理意义

- $C(p;q)=\frac{1}{N}\sum_{k=1}^{N}X_p^{(k)}X_q^{(k)}$

  对角线$(p;p)$上的元素：第p维特征的方差

  矩阵$(p;q)$元的大小反映了所有样本第p维和第q维数据的相关性（若不相关，则为0）