Pandas 的基础操作

Guangyao Zhao

2022-12-02

Contents

```
import numpy as np
import pandas as pd

data = np.random.randint(low=1, high=10, size=(3, 5))
df = pd.DataFrame(data, columns=list("abcde"), index=list("ABC"))
df
```

	a	b	c	d	e
A	8	9	4	7	4
B	2	7	6	2	1
C	6	3	4	2	3

索引操作

set_index()

设置新的索引：

```
new_index = list("abc")
df.index = new_index
df
```

	a	b	c	d	e
a	8	9	4	7	4
b	2	7	6	2	1
c	6	3	4	2	3

将原数据的某一列设置为新的索引，可以看到被设置为索引后原列消失：

```
# 将某列设置为索引
df.set_index("a")  # 或者：df.set_index('a', inplace=True)，此处的 inplace 指的是原对象 df
```

	b	c	d	e
a				
8	9	4	7	4
2	7	6	2	1
6	3	4	2	3

可以替换后保持原列：

```
df.set_index("b", drop=False)
```

	a	b	c	d	e
b					
9	8	9	4	7	4
7	2	7	6	2	1
3	6	3	4	2	3

可以保持原列且保持原索引，此处可以看出 drop 针对的是列，append 针对的是索引：

```
df.set_index("a", drop=False, append=True)  # append 指的是不是追加到原 index
```

		a	b	c	d	e
	a					
a	8	8	9	4	7	4
b	2	2	7	6	2	1
c	6	6	3	4	2	3

reset_index()

重置索引：

```
df.reset_index()  # 重制索引，原索引归到列中
```

	index	a	b	c	d	e
0	a	8	9	4	7	4
1	b	2	7	6	2	1
2	c	6	3	4	2	3

如果想替换掉原索引：

```
df.reset_index(drop=True)  # 重制索引，原索引归到列中
```

	a	b	c	d	e
0	8	9	4	7	4
1	2	7	6	2	1
2	6	3	4	2	3

rename()

用 mapper 修改索引名或列名:

```
df.rename(mapper=str.upper, axis=1)  # 内置函数
```

	A	B	C	D	E
a	8	9	4	7	4
b	2	7	6	2	1
c	6	3	4	2	3

```
df.rename(mapper=lambda x: "pre_" + x, axis=1)  # 自定义匿名函数
```

	pre_a	pre_b	pre_c	pre_d	pre_e
a	8	9	4	7	4
b	2	7	6	2	1
c	6	3	4	2	3

其它

数据类型

```
df.index.dtype
```

```
dtype('O')
```

排序:

```
df.index.sort_values(ascending=False)
```

```
Index(['c', 'b', 'a'], dtype='object')
```

函数:

```
df.index.map(lambda x: "pre_" + x)
```

```
Index(['pre_a', 'pre_b', 'pre_c'], dtype='object')
```

数据的信息

此处主要介绍数据框的基础信息和统计信息，验证下读取数据是否和原数据信息大概一致，比如行名，列名，数据量是否缺失，各列的类型数据等等。

```
data = np.random.randint(low=1, high=10, size=(6, 5))
df = pd.DataFrame(data, columns=list("abcde"), index=list("ABCDEF"))
df
```

	a	b	c	d	e
A	5	7	9	8	3
B	4	4	2	4	5
C	3	3	8	5	4
D	3	5	7	6	7
E	4	1	1	1	8
F	1	7	5	8	3

head(), tail(), sample()

头部数据：

```
df.head(3)  # 默认是前 5 个数据
```

	a	b	c	d	e
A	5	7	9	8	3
B	4	4	2	4	5
C	3	3	8	5	4

尾部数据：

```
df.tail(4)  # 默认是后 5 个数据
```

	a	b	c	d	e
C	3	3	8	5	4
D	3	5	7	6	7
E	4	1	1	1	8
F	1	7	5	8	3

随机采样：

```
df.sample(3)  # 默认 1 条数据，可以指定条数
```

	a	b	c	d	e
A	5	7	9	8	3
F	1	7	5	8	3
D	3	5	7	6	7

shape

和 numpy 一样，pandas 也内置了数据形状查找功能：

```
df.shape
```

```
(6, 5)
```

info()

Pandas 有一个很好用的功能，直接可以显示出数据框的常规信息：

- 数据类型
- 索引情况
- 行数列数
- 各字段数据类型

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6 entries, A to F
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   a       6 non-null      int64
 1   b       6 non-null      int64
 2   c       6 non-null      int64
 3   d       6 non-null      int64
```

```
 4   e       6 non-null      int64
dtypes: int64(5)
memory usage: 288.0+ bytes
```

统计计算

describe()

Pandas 可以直接批量给出数据框的统计信息：

- 行数量
- 平均数
- 标准差
- 最小值
- 最大值

```
df.describe()
```

	a	b	c	d	e
count	6.000000	6.000000	6.000000	6.000000	6.000000
mean	3.333333	4.500000	5.333333	5.333333	5.000000
std	1.366260	2.345208	3.265986	2.658320	2.097618
min	1.000000	1.000000	1.000000	1.000000	3.000000
25%	3.000000	3.250000	2.750000	4.250000	3.250000
50%	3.500000	4.500000	6.000000	5.500000	4.500000
75%	4.000000	6.500000	7.750000	7.500000	6.500000
max	5.000000	7.000000	9.000000	8.000000	8.000000

corr()

在此强调一点，pandas 的重点在于列，也就是所谓的特征列，所以默认的情况和 numpy 强调数据的整体不同，pandas 更注重的是每列的情况，默认自然而然也就是以列为整体

```
df.corr()
```

	a	b	c	d	e
a	1.000000	-0.249675	0.014940	-0.312045	0.209359
b	-0.249675	1.000000	0.574456	0.962415	-0.731804
c	0.014940	0.574456	1.000000	0.744833	-0.554680
d	-0.312045	0.962415	0.744833	1.000000	-0.789076
e	0.209359	-0.731804	-0.554680	-0.789076	1.000000

同理还有 `df.max()`, `df.min()`, `df.std()`, `df.corr()` 等常用统计函数。

idxmax()

还有一些更有用的信息，比如每列的最大值索引：

```
df.idxmax()  # 同理也有 df.xmin()
```

	0
a	A
b	A
c	A
d	A
e	E

nunique()

还有个很好用的功能，查询每列的集合：

```
df.nunique()
```

	0
a	4
b	5
c	6
d	5
e	5

非统计计算

all(), any()

判断列数据是否都大于某个值，计算分为两部：

- 判断数据大小，将结果表示为 True，False
- 如果该列所有都为 True，则结果为 True

```
df > 3
```

	a	b	c	d	e
A	True	True	True	True	False
B	True	True	False	True	True
C	False	False	True	True	True
D	False	True	True	True	True
E	True	False	False	False	True
F	False	True	True	True	False

```
(df > 3).all()
```

	o
a	False
b	False
c	False
d	False
e	False

any() 用法和 all() 一致，只是前者是只要有一个结果为 True 则为 True。

round()

```
data = np.random.randn(6, 5)
df = pd.DataFrame(data, columns=list("abcde"), index=list("ABCDEF"))
df
```

	a	b	c	d	e
A	0.753506	-1.156491	0.195253	-2.057953	-1.100952
B	-0.546977	-0.564385	0.219106	-0.483550	-0.809611
C	-1.271007	0.584289	0.561312	-0.949898	0.429530
D	2.852030	1.317620	0.814921	-2.237400	0.815241
E	0.557681	-0.249468	1.270958	-0.039905	0.625712
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

将整个数据框四舍五入到小数点两位：

```
df.round(2)
```

	a	b	c	d	e
A	0.75	-1.16	0.20	-2.06	-1.10
B	-0.55	-0.56	0.22	-0.48	-0.81
C	-1.27	0.58	0.56	-0.95	0.43
D	2.85	1.32	0.81	-2.24	0.82
E	0.56	-0.25	1.27	-0.04	0.63
F	1.96	1.49	-0.40	-1.59	-0.43

指定特定列的有效数字：

```
df.round({"a": 1, "b": 2, "c": 3, "d": 4, "e": 5})
```

	a	b	c	d	e
A	0.8	-1.16	0.195	-2.0580	-1.10095
B	-0.5	-0.56	0.219	-0.4836	-0.80961
C	-1.3	0.58	0.561	-0.9499	0.42953
D	2.9	1.32	0.815	-2.2374	0.81524
E	0.6	-0.25	1.271	-0.0399	0.62571
F	2.0	1.49	-0.398	-1.5851	-0.43362

运算

```
df.add()
df.sub()
```

```
df.mul()
df.div()
df.mod() # 模
df.pow()
df.dot(df2) # 矩阵运算
```

value_counts()

该函数是 Series 的专有函数，在统计某列的分布时很好用

```
df["a"].value_counts(normalize=True)
```

	a
0.557681	0.166667
-0.546977	0.166667
2.852030	0.166667
1.957232	0.166667
-1.271007	0.166667
0.753506	0.166667

位置计算

diff()

Pandas 提供了增量计算，比如上一个数据和本数据的差值，当然了，也可以计算下一个和本数据的差值：

```
df.diff(1)  # 上一个和本数据的差值，因第一行的前面没有数据，所以第一行为 NaN
```

	a	b	c	d	e
A	NaN	NaN	NaN	NaN	NaN
B	-1.300483	0.592106	0.023854	1.574403	0.291340
C	-0.724030	1.148674	0.342205	-0.466347	1.239141
D	4.123038	0.733331	0.253609	-1.287503	0.385711
E	-2.294350	-1.567088	0.456037	2.197495	-0.189529
F	1.399551	1.743364	-1.668783	-1.545159	-1.059331

```
df.diff(-1)  # 下一个和本数据差值，因最后一行的后面没有数据，所以最后一行为 NaN
```

	a	b	c	d	e
A	1.300483	-0.592106	-0.023854	-1.574403	-0.291340
B	0.724030	-1.148674	-0.342205	0.466347	-1.239141
C	-4.123038	-0.733331	-0.253609	1.287503	-0.385711
D	2.294350	1.567088	-0.456037	-2.197495	0.189529
E	-1.399551	-1.743364	1.668783	1.545159	1.059331
F	NaN	NaN	NaN	NaN	NaN

shift()

对数据进行移位，不做任何计算。注意，索引和列保持不变：

```
df.shift(periods=1, axis=0, fill_value=0)  # 移位后会出现空值，可以选择填充值
```

	a	b	c	d	e
A	0.000000	0.000000	0.000000	0.000000	0.000000
B	0.753506	-1.156491	0.195253	-2.057953	-1.100952
C	-0.546977	-0.564385	0.219106	-0.483550	-0.809611
D	-1.271007	0.584289	0.561312	-0.949898	0.429530
E	2.852030	1.317620	0.814921	-2.237400	0.815241
F	0.557681	-0.249468	1.270958	-0.039905	0.625712

rank()

该函数是将数据的数据大小序列替换到原数据中：

```
df.rank(axis=1)  # 行排序
```

	a	b	c	d	e
A	5.0	2.0	4.0	1.0	3.0
B	3.0	2.0	5.0	4.0	1.0
C	1.0	5.0	4.0	2.0	3.0
D	5.0	4.0	2.0	1.0	3.0
E	3.0	1.0	5.0	2.0	4.0
F	5.0	4.0	3.0	1.0	2.0

以上是绝对顺序，也可以转化为分位数：

```
df.rank(axis=1, pct=True)
```

	a	b	c	d	e
A	1.0	0.4	0.8	0.2	0.6
B	0.6	0.4	1.0	0.8	0.2
C	0.2	1.0	0.8	0.4	0.6
D	1.0	0.8	0.4	0.2	0.6
E	0.6	0.2	1.0	0.4	0.8
F	1.0	0.8	0.6	0.2	0.4

数据选择

数据选择是 pandas 最重要的一部分，方式分为三种：

- `df.a`：取出列，推荐
- `df['a']`，`df[0]`：推荐前者，不太推荐后者
- `df.loc['a', 'c']`，`df.loc[0, 10]`：轴标签，推荐
- `df.iloc[0, 5]`，`df.iloc[0: 3, 1: 2]`：轴索引，推荐

> Tip
>
> `.loc()` 和 `.iloc()` 只相差个 `i`，那是 `index` 的意思。

从上面可以看出，在 pands 中同样支持切片。

loc()

某列的所有行：

```
df.loc[:, "a"]
```

	a
A	0.753506
B	-0.546977
C	-1.271007
D	2.852030
E	0.557681
F	1.957232

切片：

```
df.loc["A":"D", "a":"c"]
```

	a	b	c
A	0.753506	-1.156491	0.195253
B	-0.546977	-0.564385	0.219106
C	-1.271007	0.584289	0.561312
D	2.852030	1.317620	0.814921

某几列的某些行：

```
df.loc["A":"D", ["a", "c"]]
```

	a	c
A	0.753506	0.195253
B	-0.546977	0.219106
C	-1.271007	0.561312
D	2.852030	0.814921

iloc()

第 1 列的所有行：

```
df.iloc[:, 0]
```

	a
A	0.753506
B	-0.546977
C	-1.271007
D	2.852030
E	0.557681
F	1.957232

第 1 和 4 列的所有行：

```
df.iloc[:, [0, 3]]
```

	a	d
A	0.753506	-2.057953
B	-0.546977	-0.483550
C	-1.271007	-0.949898
D	2.852030	-2.237400
E	0.557681	-0.039905
F	1.957232	-1.585064

第 1 行的第 2 列；第 3 行的第 3 列：

```
df.iloc[[0, 2], [1, 2]]
```

	b	c
A	-1.156491	0.195253
C	0.584289	0.561312

高级操作

复杂查询

数据查询是 pandas 的最重要的功能之一，Sec. 的内容远远达不到要求，所以就需要更复杂的逻辑条件。和 Sec. 不同，此处支持以下两种形式：

- `df[]`：不筛选列，只筛选行时使用
- `df.loc()`：同时筛选行和列时使用

> Tip
>
> - 含有多逻辑时，单逻辑要加上 ()。
> - 无论哪种筛选方法，本质上都是将符合条件的标记为 True，反之标记为 False。

`df[]`

这种形式的筛选只能筛选出特定行。

```
df[(df["a"] > 0) & (df["b"] < 0.8)]
```

	a	b	c	d	e
A	0.753506	-1.156491	0.195253	-2.057953	-1.100952
E	0.557681	-0.249468	1.270958	-0.039905	0.625712

```
df[~(df["a"] > 0)]  # 非运算，即 df['a'] < 0
```

	a	b	c	d	e
B	-0.546977	-0.564385	0.219106	-0.483550	-0.809611
C	-1.271007	0.584289	0.561312	-0.949898	0.429530

```
df[df["a"] > df["b"]]
```

	a	b	c	d	e
A	0.753506	-1.156491	0.195253	-2.057953	-1.100952
B	-0.546977	-0.564385	0.219106	-0.483550	-0.809611
D	2.852030	1.317620	0.814921	-2.237400	0.815241
E	0.557681	-0.249468	1.270958	-0.039905	0.625712
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

`df.loc()`

这种形式不仅能筛选出特定行，还能指定特定列。

```
df.loc[df["a"] > 0.1, "b":]  # a 列 大于 0.1 且只看 b 列以后的列
```

	b	c	d	e
A	-1.156491	0.195253	-2.057953	-1.100952
D	1.317620	0.814921	-2.237400	0.815241
E	-0.249468	1.270958	-0.039905	0.625712
F	1.493896	-0.397825	-1.585064	-0.433619

```
df.loc[(df["a"] > 0.1) | (df["b"] < 0.3), ["a", "b"]]
```

	a	b
A	0.753506	-1.156491
B	-0.546977	-0.564385
D	2.852030	1.317620
E	0.557681	-0.249468
F	1.957232	1.493896

```
df[(df.loc[:, ["a", "b"]] > 0.1).all(axis=1)]  # a,b 列同时大于 0.1 的行
```

	a	b	c	d	e
D	2.852030	1.317620	0.814921	-2.237400	0.815241
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

query()

当逻辑非常复杂的时候，利用以上方法将会显得特别杂乱，pandas 也提供了类似 SQL 的查询语句，针对数据框的列进行查询，筛选出符合条件的行。非常推荐这种写法。

```
query = "a > b > 0.0"  # 选出 a > b 同时两列都大于 0.2 的行
df.query(query)
```

	a	b	c	d	e
D	2.852030	1.317620	0.814921	-2.237400	0.815241
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

```
quary = "(a > -0.1) & ( b< 0.3) & (c >= a + b)"
df.query(query)
```

	a	b	c	d	e
D	2.852030	1.317620	0.814921	-2.237400	0.815241
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

```
a_mean = df["a"].mean()
query = "b >= @a_mean + 0.1"  # 支持参数传递
df.query(query)
```

	a	b	c	d	e
D	2.852030	1.317620	0.814921	-2.237400	0.815241
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

filter()

此函数支持对**行名和列名**进行筛选，支持模糊匹配，正则表达式：

```
df.filter(items=["a", "b"])  # 选择特定列
```

	a	b
A	0.753506	-1.156491
B	-0.546977	-0.564385
C	-1.271007	0.584289
D	2.852030	1.317620
E	0.557681	-0.249468
F	1.957232	1.493896

```
df.filter(regex="a", axis=1)  # 列名包含 a 的列
```

	a
A	0.753506
B	-0.546977
C	-1.271007
D	2.852030
E	0.557681
F	1.957232

```
df.filter(regex="a$", axis=1)  # 筛选出以 a 结尾的列
```

	a
A	0.753506
B	-0.546977
C	-1.271007
D	2.852030
E	0.557681
F	1.957232

> **Warning**
>
> `filter()` 函数仅支持对索引和列名称进行过滤，不针对具体数据。

数据类型转换

`convert_dtypes()`

在开始数据分析之前，要清楚地了解数据类型，一般也无非是整型，浮点型，字符串，时间这几种。

```
df.convert_dtypes()  # 推断后的数据
```

	a	b	c	d	e
A	0.753506	-1.156491	0.195253	-2.057953	-1.100952
B	-0.546977	-0.564385	0.219106	-0.48355	-0.809611
C	-1.271007	0.584289	0.561312	-0.949898	0.42953
D	2.85203	1.31762	0.814921	-2.2374	0.815241
E	0.557681	-0.249468	1.270958	-0.039905	0.625712
F	1.957232	1.493896	-0.397825	-1.585064	-0.433619

```
df.convert_dtypes().dtypes  # 推断后的数据类型
```

	0
a	Float64
b	Float64
c	Float64
d	Float64
e	Float64

to_xxx(), astype()

强制转换数据类型

```
df.iloc[0, 0] = np.nan  # 将该元素设置为异常值
pd.to_numeric(df["a"], errors="coerce").fillna(0)  # 如果存在异常值，将使用 NaN 填充，然后将其转换为 0
```

	a
A	0.000000
B	-0.546977
C	-1.271007
D	2.852030
E	0.557681
F	1.957232

```
df.iloc[0, 0] = -1
df.astype("float32").dtypes
```

	0
a	float32
b	float32
c	float32
d	float32
e	float32

```
df = df.astype("int64")  # 将 float 转换为 int
df
```

	a	b	c	d	e
A	-1	-1	0	-2	-1
B	0	0	0	0	0
C	-1	0	0	0	0
D	2	1	0	-2	0
E	0	0	1	0	0
F	1	1	0	-1	0

```
df["a"] = df["a"].astype("float64")  # 将 a 列单独设置为 float
df.dtypes
```

	0
a	float64
b	int64
c	int64
d	int64
e	int64

排序

`sort_index()`

按行索引:

```
df.sort_index(ascending=False, axis=0)
```

	a	b	c	d	e
F	1.0	1	0	-1	0
E	0.0	0	1	0	0
D	2.0	1	0	-2	0
C	-1.0	0	0	0	0
B	0.0	0	0	0	0
A	-1.0	-1	0	-2	-1

按列索引:

```
df.sort_index(ascending=False, axis=1)
```

	e	d	c	b	a
A	-1	-2	0	-1	-1.0
B	0	0	0	0	0.0
C	0	0	0	0	-1.0
D	0	-2	0	1	2.0
E	0	0	1	0	0.0
F	0	-1	0	1	1.0

将索引设置为 0-(n-1)，类似于 df.reset_index(drop = True)：

```
df.sort_index(ignore_index=True, axis=0)
```

	a	b	c	d	e
A	-1.0	-1	0	-2	-1
B	0.0	0	0	0	0
C	-1.0	0	0	0	0
D	2.0	1	0	-2	0
E	0.0	0	1	0	0
F	1.0	1	0	-1	0

> **Warning**
>
> 此处需要注意，当操作对象是索引和列而不是具体数据内容时，两者均可称之为索引，即行索引和列索引。但需要注意的是对列索引排序没有什么实际意义。

sort_values()

```
df.sort_values(by="a", ascending=True, ignore_index=True)  # 忽视 index，即将索引初始化为 0 - (n-1)
```

	a	b	c	d	e
0	-1.0	-1	0	-2	-1
1	-1.0	0	0	0	0
2	0.0	0	0	0	0
3	0.0	0	1	0	0
4	1.0	1	0	-1	0
5	2.0	1	0	-2	0

> Tip
>
> 为什么要有 ignore_index 选项？因为根据值排序以后，索引的顺序会显得很凌乱，如果不使用索引的信息时可以添加此选项初始化索引值。

也可多列混合排序：

```
df.sort_values(by=["a", "b"])
```

	a	b	c	d	e
A	-1.0	-1	0	-2	-1
C	-1.0	0	0	0	0
B	0.0	0	0	0	0
E	0.0	0	1	0	0
F	1.0	1	0	-1	0
D	2.0	1	0	-2	0

更进一步，按照索引和某列混合排序，比如班级里要按成绩和人名排名，其中人名是索引：

```
df1 = pd.DataFrame({"score": [99, 99, 97, 86, 86, 86, 90, 97]}, index=list("qisdndod"))
df1.index.name = "name"  # 给索引添加名字，方便进行下一步使用
df1.sort_values(by=[df1.index.name, "score"])
```

name	score
d	86
d	86
d	97
i	99
n	86
o	90
q	99
s	97

高级过滤

`where()`

满足条件的维持原值不变，不满足的赋予新的值：

```
df.where(cond=df["a"] >= 0.05, other="one")
```

	a	b	c	d	e
A	one	one	one	one	one
B	one	one	one	one	one
C	one	one	one	one	one
D	2.0	1	0	-2	0
E	one	one	one	one	one
F	1.0	1	0	-1	0

`mask()`

该函数和 `where()` 相反，满足条件的赋予新值，不满足的维持不变：

```
df.mask(cond=df["a"] >= 0.05, other="one")
```

	a	b	c	d	e
A	-1.0	-1	0	-2	-1
B	0.0	0	0	0	0
C	-1.0	0	0	0	0
D	one	one	one	one	one
E	0.0	0	1	0	0
F	one	one	one	one	one

数据迭代

原则上将，无论 numpy 还是 pandas 都是矢量化运算，但难免也要用到迭代处理。

常规方法：

```
for i, n, s in zip(df.index, df["a"], df["b"]):
    print(i, n, s)
```

```
A -1.0 -1
B 0.0 0
C -1.0 0
D 2.0 1
E 0.0 0
F 1.0 1
```

```
df.iterrows()
```

生成一个可迭代对象，将数据框的行作为组成的 Series 数据对进行迭代。

```
for index, row in df.iterrows():
    print(index, row["a"])
```

```
A -1.0
B 0.0
C -1.0
D 2.0
E 0.0
F 1.0
```

`df.itertuples()`

封装程度更高，会将列名返回：

```
for row in df.itertuples(index=False):
    print(row)
```

```
Pandas(a=-1.0, b=-1, c=0, d=-2, e=-1)
Pandas(a=0.0, b=0, c=0, d=0, e=0)
Pandas(a=-1.0, b=0, c=0, d=0, e=0)
Pandas(a=2.0, b=1, c=0, d=-2, e=0)
Pandas(a=0.0, b=0, c=1, d=0, e=0)
Pandas(a=1.0, b=1, c=0, d=-1, e=0)
```

函数应用

函数的目的就是为了重复利用代码，同时让整个代码结构显得更加清晰。在此主要函数有以下几种：

- `apply()`：应用在行或者列中，可以传递多参数。推荐
- `map()`：应用在列中的每个元素，只能传递一个参数。推荐
- `applymap()`：应用在整个数据框中，只能传递一个参数。推荐
- `pipe()`：应用在整个数据框中，不太推荐

> Tip
>
> 还是那句话，在 pandas 中数据处理主要针对的是列，行处理使用频率不高。

`apply()`

可以针对某列：

```
df["a"].apply(lambda x: x * 2)
```

	a
A	-2.0
B	0.0
C	-2.0
D	4.0
E	0.0
F	2.0

也可以针对所有列：

```
df.apply(lambda x: x * 2)
```

	a	b	c	d	e
A	-2.0	-2	0	-4	-2
B	0.0	0	0	0	0
C	-2.0	0	0	0	0
D	4.0	2	0	-4	0
E	0.0	0	2	0	0
F	2.0	2	0	-2	0

自定义函数：

```
# 去掉最高和最低，然后求平均值，最后添加一个 bias 项
def my_means(s, bias):
    max_min_ser = pd.Series([-s.max(), -s.min()])
    tmp = s.append(max_min_ser).sum()  # 去掉最大值和最小值
    res = tmp / (s.count() - 2)
    return res + bias



df.apply(my_means, args=[0.01], axis=0)
```

	0
a	0.01
b	0.26
c	0.01
d	-0.74
e	0.01

筛选出数型列进行运算：

```
df.select_dtypes(include="number").apply(my_means, args=[0.00], axis=0)
```

	0
a	0.00
b	0.25
c	0.00
d	-0.75
e	0.00

map()

和 apply() 不同，map() 只能针对某列数据进行某种操作，不可以一次性对全体列，另外 map() 的局限性在于只能传递入一个参数：

```
def func_map(x):
    return x * 2



df["a"].map(func_map)
```

	a
A	-2.0
B	0.0
C	-2.0
D	4.0
E	0.0
F	2.0

applymap()

和之前的不同，此函数可以做到针对每个元素进行操作：

```
def func_applymap(x):
    return x * 2 + 1



df.applymap(func_applymap)
```

	a	b	c	d	e
A	-1.0	-1	1	-3	-1
B	1.0	1	1	1	1
C	-1.0	1	1	1	1
D	5.0	3	1	-3	1
E	1.0	1	3	1	1
F	3.0	3	1	-1	1