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

在日常的数据处理中，经常会对一个DataFrame进行逐行、逐列和逐元素的操作，对应这些操作，Pandas中的map、apply和applymap可以解决绝大部分这样的数据处理需求

三种方法的使用和区别：

```
apply：应用在DataFrame的行或列中；
map：应用在单独一列（Series）的每个元素中。
applymap：应用在DataFrame的每个元素中；
```

apply()方法

前面也说了apply方法是一般性的“拆分-应用-合并”方法。
apply()将一个函数作用于DataFrame中的每个行或者列
它既可以得到一个经过广播的标量值，也可以得到一个相同大小的结果数组。我们先来看下函数形式：

`df.apply(func, axis=0, raw=False, result_type=None, args=(), **kwds)`

- `func` ：函数应用于每一列或每一行
- `axis`：
  0或“索引”：将函数应用于每一列。
  1或“列”：将函数应用于每一行。
- ``

```
df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
df
```

	A	B
0	4	9
1	4	9
2	4	9

```
df.apply(np.sum)
```

```
A    12
B    27
dtype: int64
```

```
# 1或“列”：将函数应用于每一行。
df.apply(np.sum, axis=1)
```

```
0    13
1    13
2    13
dtype: int64
```

或者使用 lambda 函数做简单的运算：

```
df.apply(lambda x: x + 1)
```

	A	B
0	5	10
1	5	10
2	5	10

但是这样使用起来非常不方便，每次都要定义 lambda 函数。因此可以通过 def 定义一个函数，然后再调用该函数，在实际处理中都是定义自己所需要的函数完成操作：

```
def cal_result(df, x, y):
    df['C'] = (df['A'] + df['B']) * x
    df['D'] = (df['A'] + df['B']) * y
    return df
df.apply(cal_result, x=3, y=8, axis=1)
```

```

```

	A	B	C	D
0	4	9	39	104
1	4	9	39	104
2	4	9	39	104

```
df.apply(cal_result, args=(3, 8), axis=1)
```

```
df.apply(cal_result, **{'x': 3, 'y': 8}, axis=1)
```

在这里我们先定义了一个 cal_result 函数，它的作用是计算 A,B 列和的 x 倍和 y 倍添加到 C,D 列中。这里有三种方式可以完成参数的赋值，

第一种方式直接通过关键字参数赋值，指定参数的值；

第二种方式是使用 args 关键字参数传入一个包含参数的元组；

第三种方式传入通过 ** 传入包含参数和值的字典

apply的使用是很灵活的，再举一个例子，配合 loc 方法我们能够在最后一行得到一个总和：

```
df.loc[2] = df.apply(np.sum)
df
```

applymap()方法

该方法针对DataFrame中的元素进行操作，还是使用这个数据：

`df.applymap(func)`

```
df
```

```

```

	A	B
0	4	9
1	4	9
2	4	9

```
df.applymap(lambda x: '%.1f'%x)
```

```

```

	A	B
0	4.0	9.0
1	4.0	9.0
2	4.0	9.0

在这里可以看到applymap方法操作的是其中的元素，并且是对整个DataFrame进行了格式化，我们也可以选择行或列中的元素：

```
df[['A']]
```

```

```

	A
0	4
1	4
2	4

```
type(df[['A']])
```

```
pandas.core.frame.DataFrame
```

```
df['A']
```

```
0    4
1    4
2    4
Name: A, dtype: int64
```

```
type(df['A'])
```

```
pandas.core.series.Series
```

```
# 取列
df[['A']].applymap(lambda x: '%.2f'%x)
```

```

```

	A
0	4.00
1	4.00
2	4.00

```
df['A'].applymap(lambda x: '%.2f'%x)  # 异常
```

```
---------------------------------------------------------------------
---------

AttributeError                            Traceback (most recent
call last)

<ipython-input-16-585649caf30e> in <module>
----> 1 df['A'].applymap(lambda x: '%.2f'%x)  # 异常
```

```
D:\Anaconda3\lib\site-packages\pandas\core\generic.py in __getattr__(self, name)
   5272             if self._info_axis._can_hold_identifiers_and_holds_name(name):
   5273                 return self[name]
-> 5274             return object.__getattribute__(self, name)
   5275
   5276     def __setattr__(self, name: str, value) -> None:
```

```
AttributeError: 'Series' object has no attribute 'applymap'
```

需要注意的是这里必须使用 df[['A']] ，表示这是一个DataFrame，而不是一个Series，如果使用 df['A'] 就会报错。同样从行取元素也要将它先转成DataFrame。还需要注意apply方法和applymap的区别：

```
apply方法操作的是行或列的运算，而不是元素的运算，比如在这里使用格式化操作就会报错；
applymap方法操作的是元素，因此没有诸如axis这样的参数，它只接受函数传入。
```

map()方法

如果你对applymap方法搞清楚了，那么map方法就很简单，说白了map方法是应用在Series中的，还是举上面的例子：

```
df[['A']].applymap(lambda x: '%.2f'%x)
```

	A
0	4.00
1	4.00
2	4.00

```
df['A'].map(lambda x: '%.2f'%x)
```

```
0    4.00
1    4.00
2    4.00
Name: A, dtype: object
```

```

```