pandas数据处理

1、删除重复元素

使用duplicated()函数检测重复的行，返回元素为布尔类型的Series对象，每个元素对应一行，如果该行不是第一次出现，则元素为True

- keep参数：指定保留哪一重复的行数据
- 创建具有重复元素行的DataFrame

In [1]:
```
import numpy as np
import pandas as pd
from pandas import DataFrame,Series
```

In [2]:
```
#创建一个df
np.random.seed(1)
df = DataFrame(data=np.random.randint(1,100,size=(8,6)))
df
```

Out[2]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
1	80	65	17	2	77	72
2	7	26	51	21	19	85
3	12	29	30	15	51	69
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

In [3]:
```
# 手动将df的某几行设置成相同的内容
```

In [4]:
```
df.iloc[1] = [6,6,6,6,6,6]
df.iloc[2] = [6,6,6,6,6,6]
df.iloc[3] = [6,6,6,6,6,6]
df
```

Out[4]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
1	6	6	6	6	6	6
2	6	6	6	6	6	6
3	6	6	6	6	6	6
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

- 使用duplicated查看所有重复元素行

In [5]:
```
df.duplicated(keep='last')  #保留最后一个 重复的元素
```

Out[5]:
```
0    False
1     True
2     True
3    False
4    False
5    False
6    False
7    False
dtype: bool
```

- 删除重复元素的行

In [6]:
```
# 删除重复的行
indexs = df[df.duplicated(keep='last')].index
df.drop(labels=indexs,axis=0)
```

Out[6]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
3	6	6	6	6	6	6
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

- 使用drop_duplicates()函数删除重复的行
  - drop_duplicates(keep='first/last'/False)

In [7]:
```
df
```

Out[7]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
1	6	6	6	6	6	6
2	6	6	6	6	6	6
3	6	6	6	6	6	6
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

In [8]:
```
df.drop_duplicates() # 默认保留 最前的一个重复
```

Out[8]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
1	6	6	6	6	6	6
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

2. 映射

1) replace()函数：替换元素

使用replace()函数，对values进行映射操作

Series替换操作

- 单值替换
  - 普通替换
  - 字典替换(推荐)
- 多值替换
  - 列表替换
  - 字典替换（推荐）
- 参数
  - to_replace:被替换的元素

In [9]:
```
df
```

Out[9]:

	0	1	2	3	4	5
0	38	13	73	10	76	6
1	6	6	6	6	6	6
2	6	6	6	6	6	6
3	6	6	6	6	6	6
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

In [10]:
```
df.replace(to_replace=6,value='six')  # 把所有的6替换 成 six
```

Out[10]:

	0	1	2	3	4	5
0	38	13	73	10	76	six
1	six	six	six	six	six	six
2	six	six	six	six	six	six
3	six	six	six	six	six	six
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

In [11]:
```
df.replace(to_replace={5:6},value='six')  # 把索引为 5列  的6 改为 six
```

Out[11]:

	0	1	2	3	4	5
0	38	13	73	10	76	six
1	6	6	6	6	6	six
2	6	6	6	6	6	six
3	6	6	6	6	6	six
4	88	88	95	97	87	14
5	10	8	64	62	23	58
6	2	1	61	82	9	89
7	14	48	73	31	72	4

注意：DataFrame中，无法使用method和limit参数

2) map()函数：新建一列， map函数并不是df的方法，而是series的方法

- map()可以映射新一列数据
- map()中可以使用lambd表达式
- map()中可以使用方法，可以是自定义的方法

  eg:map({to_replace:value})
- **注意** map()中不能使用sum之类的函数，for循环

- 新增一列：给df中，添加一列，该列的值为中文名对应的英文名

In [12]:
```
dic = {
    'name':['周杰伦','李四','王五'],
    'salary':[1000,2000,3000]
}
df = DataFrame(data=dic)
df
```

Out[12]:

	name	salary
0	周杰伦	1000
1	李四	2000
2	王五	3000

In [13]:
```
# 封装一个 映射关系表
dic = {
    '周杰伦':'jay',
    '王五':'wangwu',
    '李四':'lisi'
}
df['ename'] = df['name'].map(dic)
```

In [14]:
```
df
```

Out[14]:

	name	salary	ename
0	周杰伦	1000	jay
1	李四	2000	lisi
2	王五	3000	wangwu

map当做一种运算工具，至于执行何种运算，是由map函数的参数决定的（参数：lambda，函数）

- 使用自定义函数

In [15]:
```
def after_salary(s):
    if s<=500:
        return s
    else:
        return s - (s-500)*0.5
```

In [16]:
```
# 超过500 部分的钱 缴纳50%的税
```

In [17]:
```
after_sal = df['salary'].map(after_salary)  # df['salary'] 中的值传入after_salary
df['after_salary'] = after_sal
df
```

Out[17]:

	name	salary	ename	after_salary
0	周杰伦	1000	jay	750.0
1	李四	2000	lisi	1250.0
2	王五	3000	wangwu	1750.0

- 使用lambda表达式

In [18]:
```
df['lambda_xxxx'] = df['salary'].map(lambda x:x if x>2000 else x-500)
```

In [19]:
```
df
```

Out[19]:

	name	salary	ename	after_salary	lambda_xxxx
0	周杰伦	1000	jay	750.0	500
1	李四	2000	lisi	1250.0	1500
2	王五	3000	wangwu	1750.0	3000

注意：并不是任何形式的函数都可以作为map的参数。只有当一个函数具有一个参数且有返回值，那么该函数才可以

作为map的参数。

3. 使用聚合操作对数据异常值检测和过滤

使用df.std()函数可以求得DataFrame对象每一列的标准差

- 创建一个1000行3列的df 范围（0-1），求其每一列的标准差

In [20]:
```
np.random.seed(1)
df = DataFrame(np.random.random(size=(1000,3)),columns=['A','B','C'])
df.head()
```

Out[20]:

	A	B	C
0	0.417022	0.720324	0.000114
1	0.302333	0.146756	0.092339
2	0.186260	0.345561	0.396767
3	0.538817	0.419195	0.685220
4	0.204452	0.878117	0.027388

In [21]:
```
value = df['C'].std()*2
value
```

Out[21]: 0.573996473439927

In [22]:
```
df['C'] <= value
# 0      True
# 1      True
# 2      True
# 3     False
# 4      True
# 5      True
# 6     False
```

...

对df应用筛选条件,去除标准差太大的数据:假设过滤条件为 C列数据大于两倍的C列标准差

In [23]:
```
df.loc[df['C']<=value]
```

	A	B	C
0	0.417022	0.720324	0.000114
1	0.302333	0.146756	0.092339
2	0.186260	0.345561	0.396767
4	0.204452	0.878117	0.027388
5	0.670468	0.417305	0.558690

...

4. 排序

使用.take()函数排序

- take()函数接受一个索引列表，用数字表示,使得df根据列表中索引的顺序进行排序
- eg:df.take([1,3,4,2,5])

可以借助np.random.permutation()函数随机排序

In [24]:
```
df.head()
```

Out[24]:

	A	B	C
0	0.417022	0.720324	0.000114
1	0.302333	0.146756	0.092339
2	0.186260	0.345561	0.396767
3	0.538817	0.419195	0.685220
4	0.204452	0.878117	0.027388

In [25]:
```
df.take([0,2,1],axis=1).head()  # axis=1列索引 按 0 2 1 排序
```

Out[25]:

	A	C	B
0	0.417022	0.000114	0.720324
1	0.302333	0.092339	0.146756
2	0.186260	0.396767	0.345561
3	0.538817	0.685220	0.419195
4	0.204452	0.027388	0.878117

In [26]:
```
values = df.take(np.random.permutation(1000),axis=0).take(np.random.permutation(3),axis=1)
```

In [27]:
```
values.head() # 先进行 随机排序 在进行列随机排序
```

Out[27]:

	C	A	B
217	0.106584	0.108065	0.787552
741	0.029783	0.545471	0.606535
745	0.809217	0.366732	0.574940
385	0.815477	0.391099	0.320565
610	0.770132	0.622313	0.231695

- np.random.permutation(x)可以生成x个从0-(x-1)的随机数列

In [28]:
```
np.random.permutation(100)
```

Out[28]:
```
array([30, 88, 21, 74, 31, 43, 96, 52, 71,  4, 14, 26, 47, 10,  0, 25, 16,
       42, 75, 53, 20, 57, 35, 92, 99, 68, 39, 80, 51, 94, 93, 61, 49,  6,
       60, 64, 56, 24, 89, 19, 46, 18, 83, 55, 76,  3, 29, 37, 28, 87, 63,
       98, 86, 59, 22, 90, 70, 81, 15, 82, 72, 23, 40, 12, 91, 69, 11, 54,
       79, 36, 95, 27, 78,  1, 13, 48, 84, 66, 32,  7, 62, 77, 73, 50, 41,
        9,  5, 44,  8, 33, 17, 58, 45,  2, 38, 85, 97, 34, 65, 67])
```

随机抽样

当DataFrame规模足够大时，直接使用np.random.permutation(x)函数，就配合take()函数实现随机抽样

5. 数据分类处理【重点】

数据聚合是数据处理的最后一步，通常是要使每一个数组生成一个单一的数值。

数据分类处理：

- 分组：先把数据分为几组
- 用函数处理：为不同组的数据应用不同的函数以转换数据

- 合并：把不同组得到的结果合并起来

数据分类处理的核心：

- groupby()函数
- groups属性查看分组情况
- eg: df.groupby(by='item').groups

分组

In [29]:
```
df = DataFrame({'item':['Apple','Banana','Orange','Banana','Orange','Apple'],
                'price':[4,3,3,2.5,4,2],
                'color':['red','yellow','yellow','green','green','green'],
                'weight':[12,20,50,30,20,44]})
df
```

Out[29]:

	item	price	color	weight
0	Apple	4.0	red	12
1	Banana	3.0	yellow	20
2	Orange	3.0	yellow	50
3	Banana	2.5	green	30
4	Orange	4.0	green	20
5	Apple	2.0	green	44

- 使用groupby实现分组

  该函数可以进行数据的分组，但是不显示分组情况

In [30]:
```
df.groupby(by='color').groups
```

Out[30]:
```
{'green': Int64Index([3, 4, 5], dtype='int64'),
 'red': Int64Index([0], dtype='int64'),
 'yellow': Int64Index([1, 2], dtype='int64')}
```

- 使用groups查看分组情况

In [31]:
```
df.groupby(by='item').groups
```

Out[31]:
```
{'Apple': Int64Index([0, 5], dtype='int64'),
 'Banana': Int64Index([1, 3], dtype='int64'),
 'Orange': Int64Index([2, 4], dtype='int64')}
```

- 分组后的聚合操作：分组后的成员中可以被进行运算的值会进行运算，不能被运算的值不进行运算

In [32]:
```
df
```

Out[32]:

	item	price	color	weight
0	Apple	4.0	red	12
1	Banana	3.0	yellow	20
2	Orange	3.0	yellow	50
3	Banana	2.5	green	30
4	Orange	4.0	green	20
5	Apple	2.0	green	44

In [33]:
```
#给df创建一个新列，内容为各个水果的平均价格

df.groupby(by='item').mean()['price']
```

Out[33]:
```
item
Apple     3.00
Banana    2.75
Orange    3.50
Name: price, dtype: float64
```

In [34]:
```
df.groupby(by='item')['price'].mean()
```

Out[34]:
```
item
Apple     3.00
Banana    2.75
Orange    3.50
Name: price, dtype: float64
```

In [35]:
```
dic = {
        'Apple':3,
    'Banana':2.75,
    'Orange':3.5
}
df['mean_price'] = df['item'].map(dic)
df
```

Out[35]:

	item	price	color	weight	mean_price
0	Apple	4.0	red	12	3.00
1	Banana	3.0	yellow	20	2.75
2	Orange	3.0	yellow	50	3.50
3	Banana	2.5	green	30	2.75
4	Orange	4.0	green	20	3.50
5	Apple	2.0	green	44	3.00

计算出苹果的平均价格

In [36]:
```
df.groupby(by='item')['price'].mean()['Apple']
```

Out[36]:
```
3.0
```

按颜色查看各种颜色的水果的平均价格

In [37]:
```
df.groupby(by='color')['price'].mean()
```

Out[37]:
```
color
green     2.833333
red       4.000000
yellow    3.000000
Name: price, dtype: float64
```

In [38]:
```
dic = {
      'green':2.83,
    'red':4,
    'yellow':3
}
df['color_mean_price'] = df['color'].map(dic)
df
```

Out[38]:

	item	price	color	weight	mean_price	color_mean_price
0	Apple	4.0	red	12	3.00	4.00
1	Banana	3.0	yellow	20	2.75	3.00
2	Orange	3.0	yellow	50	3.50	3.00
3	Banana	2.5	green	30	2.75	2.83
4	Orange	4.0	green	20	3.50	2.83
5	Apple	2.0	green	44	3.00	2.83

6.0 高级数据聚合

使用groupby分组后，也可以使用transform和apply提供自定义函数实现更多的运算

- df.groupby('item')['price'].sum() <==> df.groupby('item')['price'].apply(sum)
- transform和apply都会进行运算，在transform或者apply中传入函数即可
- transform和apply也可以传入一个lambda表达式

In [39]:
```
#求出各种水果价格的平均值
df.groupby(by='item')['price'].mean()
```

Out[39]:
```
item
Apple     3.00
Banana    2.75
Orange    3.50
Name: price, dtype: float64
```

In [40]:
```
df
```

Out[40]:

	item	price	color	weight	mean_price	color_mean_price
0	Apple	4.0	red	12	3.00	4.00
1	Banana	3.0	yellow	20	2.75	3.00
2	Orange	3.0	yellow	50	3.50	3.00
3	Banana	2.5	green	30	2.75	2.83
4	Orange	4.0	green	20	3.50	2.83
5	Apple	2.0	green	44	3.00	2.83

In [41]:
```
#使用apply函数求出水果的平均价格
def fun(s):
    sum = 0
    for i in s :
        sum+=i
    return sum/s.size
```

In [42]:
```
df.groupby(by='item')['price'].apply(fun)
```

Out[42]:
```
item
Apple     3.00
Banana    2.75
Orange    3.50
Name: price, dtype: float64
```

In [43]:
```
#使用transform函数求出水果的平均价格
df.groupby(by='item')['price'].transform(fun)
```

Out[43]:
```
0    3.00
1    2.75
2    3.50
3    2.75
4    3.50
5    3.00
Name: price, dtype: float64
```

In [ ]:
```

```