说明：

1. 如下内容，是在《利用Python进行数据分析》总结笔记《DataAnalysis.docx》的基础上， 按照知识点的内容进行排列，并加入书本之外的知识点。
2. 使用的数据主要来源于《Data\_big.xlsx》和《Data\_small.xlsx》

《Data\_big.xlsx》包括21个字段，211291条数据：

订单编号、客户id、客户姓名、客户手机号、客户身份证、客户性别、客户年龄、车辆编号、车辆名称、车辆价格、车辆所属城市、商家ID、商家名称、首次上牌日期、首付刷卡日期、首付刷卡时间、业务期限、首付款金额、融资总额、首付车款、利息

《Data\_small.xlsx》包括20个字段，36548条数据（首付刷卡日期为2017年第三季度）：

订单编号、客户id、客户姓名、客户身份证、客户性别、客户年龄、车辆编号、车辆名称、车辆价格、车辆所属城市、商家ID、商家名称、首次上牌日期、首付刷卡日期、首付刷卡时间、业务期限、首付款金额、融资总额、首付车款、利息

3． 默认DataFrame

df1=pd.read\_excel('D:\\Data\_small.xlsx',encoding='gbk')

df2=df1[['车辆所属城市','车辆价格','融资总额']][:100]

df3=df1[['客户性别','车辆所属城市','车辆价格','融资总额']]

任务：

1. 提取车辆品牌
2. 取出车辆价格排前10%的订单
3. 时间序列
4. DataFrame转置
5. 正则
6. 类Vlookup和数据拼接

**关于DataFrame**

#DataFrame的构造及其索引 P119

#重新索引 P126

**# 生成DataFrame**

# Pandas读取Excel

from pandas import DataFrame,Series

import pandas as pd

df1=pd.read\_excel('D:\\Data\_small.xlsx',encoding='gbk')

df1[:20]

输出：df1的数据格式为DataFrame

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 订单编号 | 客户id | …… | 利息 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| …… |  |  |  |  |

# 选择DataFrame的某几列，形成新的DataFrame

df2=df1[['客户姓名','客户性别']]

df2

|  |  |  |
| --- | --- | --- |
|  | 客户姓名 | 客户性别 |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |

# 根据条件构造DataFrame的子集。实际上是给DataFrame传入bool型的Series

举例1：选取8万以上车

df\_chi=df3[df3.车辆价格>80000]

df\_chi

举例2：选取业务期限不正常的订单

abnormal=df1[df1.业务期限.isin(['33个月','21个月','34个月','23个月',])]

abnormal

# 选择表头

df1.columns

*Index(['订单编号', '客户id', '客户姓名', '客户身份证', '客户性别', '客户年龄', '车辆编号', '车辆名称', '车辆价格','车辆所属城市', '商家ID', '商家名称', '首次上牌日期', '首付刷卡日期', '首付刷卡时间', '业务期限', '首付款金额','融资总额', '首付车款', '利息'],dtype='object')*

# 选择第一行或者多行

data.ix[0] ——> 注意：ix在新版本的Pandas中已经不建议使用

data.iloc[0]

data.iloc[0:5]

# 去除缺失值。

举例：如果a列有缺失值，则将a列有缺失值的行删除。

实现原理：① frame.a.notnull()返回一个bool类型的Series。② DateFrame（Series）是DataFrame的一种筛选方法

cframe=frame[frame.a.notnull()]

cframe

# 统计DataFrame的行数（不包括表头）

len(df3)

**# DataFrame导出为Excel**

df3.to\_excel("D:\\to\_excel.xlsx",sheet\_name='Sheet1')

**# 数据聚合与分组运算**

# 描述统计

举例：计算关于车辆价格的描述统计

df1['车辆价格'].describe()

输出：

count 38965.000000

mean 157368.140639

std 107051.443548

min 30000.000000

25% 78900.000000

50% 125000.000000

75% 198000.000000

max 650000.000000

# 计算某列的平均值

df1['车辆价格'].mean()

# 分类计算。求和。

df1.groupby('首付刷卡日期').融资总额.sum()

# 分类计算。平均值

举例：计算不同性别的车辆价格平均值

price\_mean\_bysex=df1['车辆价格'].groupby(df1['客户性别']).mean()

price\_mean\_bysex

# 分类计算。计数

举例：计算各城市的销量。Series有value\_counts()方法，DataFrame没有

city\_counts=df1['车辆所属城市'].value\_counts()

city\_counts

|  |  |
| --- | --- |
| 乌鲁木齐 | 1280 |
| 银川 | 1103 |
| …… | …… |
| 阿拉善左旗 | 1 |

举例：计算不同客户性别的销量。思路：实际上是Series的值计数。

？

举例：计算车价8万以下的车辆数量

pd.value\_counts(df1['车辆价格']<=80000,sort=False)

输出：

False 28464

True 10505

Name: 经销商结算价, dtype: int64

？画图：每隔1万的区间，车辆数量的变化情况。包括累积和非累积。

# 分类计算。标准差

举例：计算各城市车辆价格的标准差。输出的结果是Series

price\_std\_by\_city=df1.groupby('车辆所属城市')['车辆价格'].std()

price\_std\_by\_city

# Series排序。和DataFrame的方法一样，只是参数不同。

sales\_std\_by\_city.sort\_values(ascending=False)

# 过滤

选出购买客户是女性的订单。

思路：使用DataFrame[ bool型Series] 的方法

order\_female=df1[df1.客户性别=='女']

order\_female

# 过滤

举例：从city\_counts的结果中，过滤掉销量不足500的城市。

方法：先对城市进行分组，然后利用size() 得到一个含有各城市分组大小的Series对象。输出结果和value\_counts()方法的不同在于，没有进行排序。——此方法也适用于DataFrame以及数据透视表

sales\_by\_city=df1.groupby('车辆所属城市').size()

sales\_by\_city

|  |  |
| --- | --- |
| 万州 | 25 |
| 三亚 | 25 |
| …… | …… |
| 齐齐哈尔 | 99 |
| 龙岩 | 30 |

goodSale\_city=sales\_by\_city.index[sales\_by\_city>=500] # 制作索引

goodSale\_city

*Index(['东莞', '乌鲁木齐', '包头', '北京', '南京', '南宁', '合肥', '呼和浩特', '唐山', '太原', '宁波','成都', '武汉', '沈阳', '深圳', '石家庄', '西安', '贵阳', '赣州', '银川', '长春', '长沙'],dtype='object', name='车辆所属城市')*

city\_counts=city\_counts.loc[goodSale\_city] # 过滤

city\_counts

|  |  |
| --- | --- |
| 东莞 | 715 |
| 乌鲁木齐 | 1280 |
| …… | …… |
| 长沙 | 608 |

# 过滤

生成实验用数据：

from pandas import DataFrame,Series

import pandas as pd

cencus\_df=pd.DataFrame({'city':['北京','上海','广州','深圳','宁波'],'cencus':[3000,25000,2000,1500,500]})

cencus\_df

|  |  |  |
| --- | --- | --- |
|  | city | cencus |
| 0 | 北京 | 3000 |
| 1 | 上海 | 2500 |
| 2 | 广州 | 2000 |
| 3 | 深圳 | 1500 |
| 4 | 宁波 | 500 |

举例：选出人口大于1500万成的城市

a\_df = cencus\_df[ cencus\_df['cencus'] > 1500 ]

|  |  |  |
| --- | --- | --- |
|  | city | cencus |
| 0 | 北京 | 3000 |
| 1 | 上海 | 2500 |
| 2 | 广州 | 2000 |

# 选择

举例：取出车辆价格排前10%的订单

思路：

**# 应用函数·apply()**

frame=DataFrame(np.random.randn(4,3),columns=list('bde'),index=['Utah','Ohio','Texas','Oregon'])

frame

输出：

|  | **b** | **d** | **e** |
| --- | --- | --- | --- |
| **Utah** | -0.101401 | -0.026205 | 0.618136 |
| **Ohio** | -2.209868 | -1.829319 | -0.850499 |
| **Texas** | -0.664304 | 0.107059 | 0.566694 |
| **Oregon** | 1.046030 | 1.037165 | 1.542936 |

#引用到行

f=lambda x:x.max()-x.min()

frame.apply(f)

输出：

b 3.255898

d 2.866484

e 2.393435

dtype: float64

备注：类似于groupby.apply()，也是将属于一行行传入函数进行处理

#应用到列

frame.apply(f,axis=1)

输出：

Utah 0.719537

Ohio 1.359369

Texas 1.230997

Oregon 0.505771

#传递给apply的函数，除了返回标量值，还可以返回由多个值组成的Series

#除了标量值外，传递给apply的函数还可以返回由多个值组成的Series

def f(x):

return Series([x.min(),x.max()],index=['min','max'])

frame.apply(f)

输出：

|  | **b** | **d** | **e** |
| --- | --- | --- | --- |
| **min** | -2.209868 | -1.829319 | -0.850499 |
| **max** | 1.046030 | 1.037165 | 1.542936 |

**# 应用函数·applymap()**

#应用元素级的python函数和applymap方法。将数字格式化到小数点后两位

format=lambda x:'%.2f' %x

frame.applymap(format)

输出：

|  | **b** | **d** | **e** |
| --- | --- | --- | --- |
| **Utah** | -0.10 | -0.03 | 0.62 |
| **Ohio** | -2.21 | -1.83 | -0.85 |
| **Texas** | -0.66 | 0.11 | 0.57 |
| **Oregon** | 1.05 | 1.04 | 1.54 |

**# 应用函数比较apply、applymap、map、transform**

apply 用在dataframe上，用于对row或者column进行计算

applymap 用于dataframe上，是元素级别的操作

map （其实是python自带的）用于series上，是元素级别的操作

**# 数据透视表**

**# 举例一**：计算分组平均数（pivot\_table的默认聚合类型）

df3.pivot\_table(index=['客户性别','车辆所属城市'])

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 融资总额 | 车辆价格 |
| 客户性别 | 车辆所属城市 |  |  |
| 女 | 万州 | 117082.6 | 134800 |
|  | 三亚 | 113885.1 | 138062.5 |
|  | 三明 | 94642.88 | 113137.5 |

**# 举例二**：计算每个城市按性别计算车辆价格的平均值。使用pivot\_table方法，返回DataFrame

mean\_price = df1.pivot\_table('车辆价格',index='车辆所属城市',columns='客户性别',aggfunc='mean')

mean\_price

|  |  |  |
| --- | --- | --- |
|  | 女 | 男 |
| 万州 | 134800.000000 | 95250.000000 |
| 三亚 | 138062.500000 | 83341.176471 |
| …… | …… | …… |
| 龙岩 | 83800.000000 | 82020.689655 |

——延伸：取出女性消费比男性消费高的城市

思路：属于DataFrame的过滤。可以先制作bool型Series，然后应用到DataFrame。

female\_higher = (price\_sum['女']-price\_sum['男']) > 0 # 原理：> 和加减乘除一样都是操作符

female\_higher\_city = mean\_price[female\_higher]

female\_higher\_city

——延伸：女性消费高、男性消费高的城市计数

思路：Series的值计数

female\_higher\_count = female\_higher.value\_counts()

female\_higher\_count

|  |  |
| --- | --- |
| True | 198 |
| False | 46 |

——排序

sort\_values() ——

举例：在mean\_price的基础上，按照女性客户降序排列。——适用于一般的DataFrame

top\_female\_sales=mean\_price.sort\_values(by='女',ascending=False)

top\_female\_sales

——反序。

举例：对top\_female\_sales排序结果进行反序，并取出前5行。

top\_female\_sales[::-1][:5]

——排序。复杂排序

方法一：apply函数

取出每个城市车价最高的3个订单——？如果一个城市没有3个订单咋办

def get\_top3(data):

return data.sort\_values(by='车辆价格',ascending=False)[:3]

top3=df1.groupby('车辆所属城市').apply(get\_top3)

top3

方法二：

利用groupby对象的可迭代性

pieces=[]

for name,data in df1.groupby('车辆所属城市'):

pieces.append(data.sort\_values(by='车辆价格',ascending=False)[:3])

top3=pd.concat(pieces,ignore\_index=True)

top3

取出其中北京的数据

top3[top3.车辆所属城市=='北京']

——DataFrame追加列。将计算结果追加为一列。思路：可将Series追加为列。

举例：DateFrame中追加一列，表示男女平均额的差值

mean\_price['差值']=mean\_price['女']-mean\_price['男']

mean\_price

|  |  |  |  |
| --- | --- | --- | --- |
|  | 女 | 男 | 差值 |
| 万州 | 134800.000000 | 95250.000000 | 39550.000000 |
| 三亚 | 138062.500000 | 83341.176471 | 54721.323529 |
| …… | …… | …… | …… |
| 龙岩 | 83800.000000 | 82020.689655 | 1779.310345 |

**# 举例三**：多值计算的透视表

df3.pivot\_table(['车辆价格','融资总额'],index='车辆所属城市',columns='客户性别')

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 融资总额 | | 车辆价格 | |
| 客户性别 | 女 | 男 | 女 | 男 |
| 车辆所属城市 |  |  |  |  |
| 万州 | 117082.6 | 76812.75 | 134800 | 95250 |
| 三亚 | 113885.1 | 70329.88 | 138062.5 | 83341.18 |
| 三明 | 94642.88 | 64895.78 | 113137.5 | 86955.56 |

**# 举例四**：添加分项小计。计算平均数

df3.pivot\_table(['车辆价格','融资总额'],index='车辆所属城市',columns='客户性别',**margins=True**)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 融资总额 | | | 车辆价格 | | |
| 客户性别 | 女 | 男 | All | 女 | 男 | All |
| 车辆所属城市 |  |  |  |  |  |  |
| 万州 | 117082.6 | 76812.75 | 84866.72 | 134800 | 95250 | 103160 |
| 三亚 | 113885.1 | 70329.88 | 84267.56 | 138062.5 | 83341.18 | 100852 |
| 三明 | 94642.88 | 64895.78 | 78894.41 | 113137.5 | 86955.56 | 99276.47 |
| …… |  |  |  |  |  |  |
| All |  |  |  |  |  |  |

**# 举例五**：使用其他聚合函数，将其传给aggfunc即可

# 使用count或者len计算有关分组大小的交叉表

df3.pivot\_table(['车辆价格','融资总额'],index='车辆所属城市',columns='客户性别',aggfunc=len,margins=True)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 融资总额 | | | 车辆价格 | | |
| 客户性别 | 女 | 男 | All | 女 | 男 | All |
| 车辆所属城市 |  |  |  |  |  |  |
| 万州 | 5 | 20 | 25 | 5 | 20 | 25 |
| 三亚 | 8 | 17 | 25 | 8 | 17 | 25 |
| 三明 | 8 | 9 | 17 | 8 | 9 | 17 |
| 三河 | 5 | 22 | 27 | 5 | 22 | 27 |
| 临夏 | NaN | 43 | 43 | NaN | 43 | 43 |
| …… |  |  |  |  |  |  |
| All |  |  |  |  |  |  |

# 将存在空的组合（也就是NA）设置一个fill\_value

df3.pivot\_table(['车辆价格','融资总额'],index='车辆所属城市',columns='客户性别',aggfunc=len,margins=True,fill\_value=0)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 融资总额 | | | 车辆价格 | | |
| 客户性别 | 女 | 男 | All | 女 | 男 | All |
| 车辆所属城市 |  |  |  |  |  |  |
| 万州 | 5 | 20 | 25 | 5 | 20 | 25 |
| 三亚 | 8 | 17 | 25 | 8 | 17 | 25 |
| 三明 | 8 | 9 | 17 | 8 | 9 | 17 |
| 三河 | 5 | 22 | 27 | 5 | 22 | 27 |
| 临夏 | 0 | 43 | 43 | 0 | 43 | 43 |
| …… |  |  |  |  |  |  |
| All |  |  |  |  |  |  |

**# 交叉表（计算分组频率的特殊透视表）**

# crosstab的前两个参数可以是数组、Series或数组列表

pd.crosstab([df1.车辆所属城市,df1.商家名称],df1.客户性别,margins=True)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 客户性别 |  | 女 | 男 | All |
| 车辆所属城市 | 商家名称 |  |  |  |
| 万州 | 万州区机电城宏睿精品二手车 | 0 | 1 | 1 |
|  | 万州天天好车 | 1 | 3 | 4 |
|  | 万州天子二手车 | 0 | 3 | 3 |
|  | 万州时尚车行 | 1 | 1 | 2 |
| …… |  |  |  |  |
| All |  |  |  |  |

**# 应用函数·groupby.apply()**

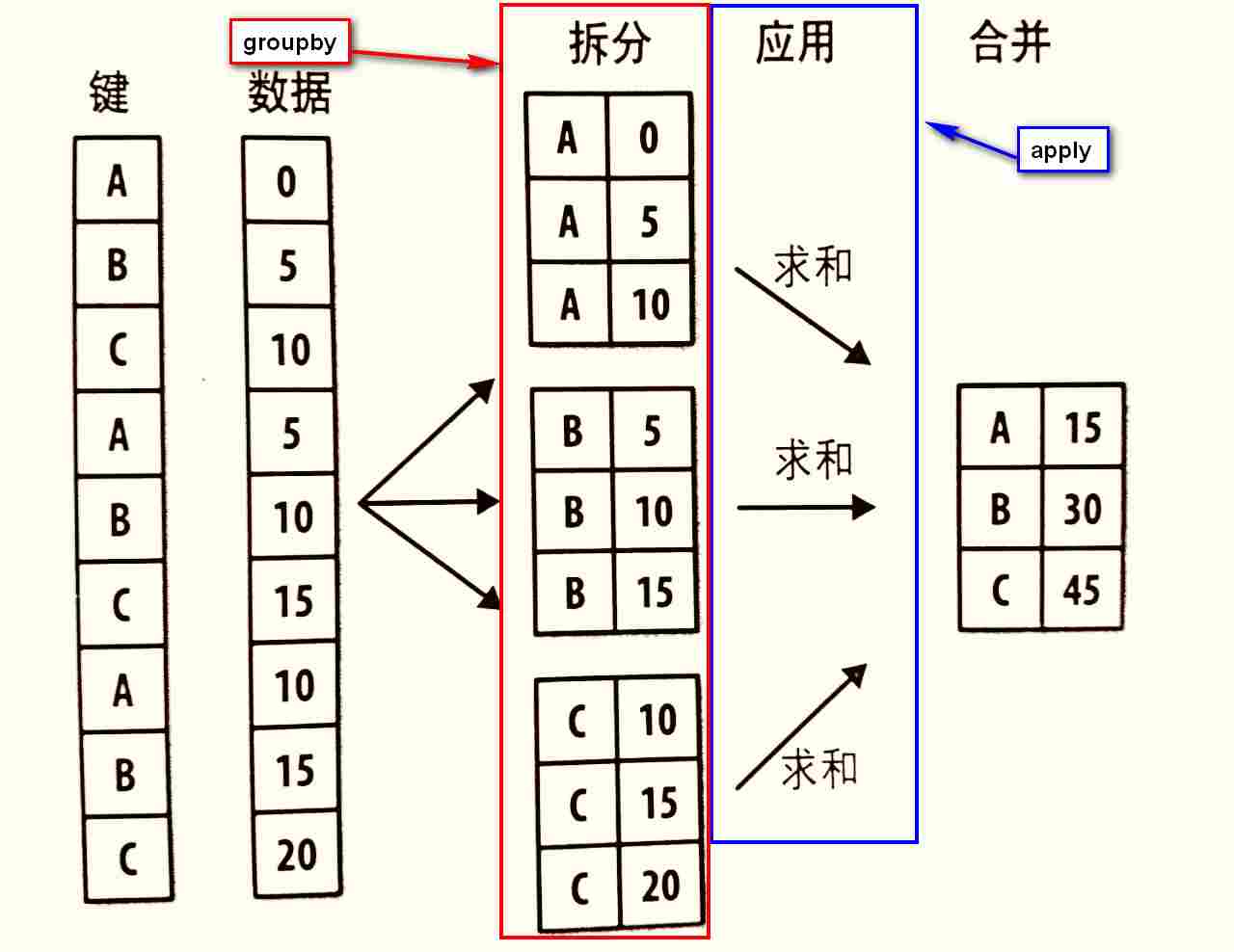
注意：此处的apply函数不是Python的apply函数，而是Pandas的apply函数

apply 是 pandas 库的一个很重要的函数，多和 groupby 函数一起用，也可以直接用于 DataFrame 和 Series 对象。主要用于数据聚合运算，可以很方便的对分组进行现有的运算和自定义的运算。参考：<https://blog.csdn.net/u010099080/article/details/53019844>

函数的传入参数根据axis来定，比如axis = 1，就会把一行数据作为Series的数据结构传入给自己实现的函数中，我们在函数中实现对Series不同属性之间的计算，返回一个结果，则apply函数会自动遍历每一行DataFrame的数据，最后将所有结果组合成一个Series数据结构并返回。

更准确的解释见官方文档：应用apply()的函数，第一个参数必须是DataFrame。返回的结果则是DataFrame、Series或者标量数据。然后将结果拼接起来。

理解：将data的每一行看作一个array，groupby()后如果不指定计算的字段的话，那么f(group)中的group默认传递的是data的每一行，所以书中例子输出结果是在整个names的dataframe形式上再加一列，1690784行5列而非我理解的262行3列了。



举例1 ：自定义函数，为DataFrame新增一列，代表融资额占总融资额（指该城市的总融资额）的比例，然后返回更新后的DataFrame

df2=df1[['车辆所属城市','车辆价格','融资总额']][:100]

df2

|  |  |  |  |
| --- | --- | --- | --- |
|  | 车辆所属城市 | 车辆价格 | 融资总额 |
| 0 | 烟台 | 170000.0 | 146460.0 |
| 1 | 承德 | 135000.0 | 116310.0 |
| …… | …… | …… | …… |
| 99 | 太原 | 62900.0 | 8300.0 |

def add\_prop(data):

loan = data.融资总额.astype(float)

data['prop']=loan/loan.sum()

return data

df3 = df2.groupby('车辆所属城市').apply(add\_prop)

df3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 车辆所属城市 | 车辆价格 | 融资总额 | prop |
| 0 | 烟台 | 170000.0 | 146460.0 | 1.000000 |
| 1 | 承德 | 135000.0 | 116310.0 | 1.000000 |
| …… | …… | …… | …… | …… |
| 99 | 太原 | 62900.0 | 8300.0 | 0.678240 |

解析①：prop=1代表该DataFrame中，该城市的数据就此一行。

如果想求该行数据在总数据（而不是本城市数据）中的比值，则用如下方式

df4=add\_prop(df2)

df4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 车辆所属城市 | 车辆价格 | 融资总额 | prop |
| 0 | 烟台 | 170000.0 | 146460.0 | 0.014831 |
| 1 | 承德 | 135000.0 | 116310.0 | 0.011778 |
| …… | …… | …… | …… | …… |
| 99 | 太原 | 62900.0 | 8300.0 | 0.006369 |

验证：

df4['prop'].sum()

out：1.0000000000000002

解析②：

def add\_prop2(data):

return data

test\_2 = df2.groupby('车辆所属城市').apply(add\_prop2)

test\_2

|  |  |  |  |
| --- | --- | --- | --- |
|  | 车辆所属城市 | 车辆价格 | 融资总额 |
| 0 | 烟台 | 170000.0 | 146460.0 |
| 1 | 承德 | 135000.0 | 116310.0 |
| …… | …… | …… | …… |
| 99 | 太原 | 62900.0 | 8300.0 |

解析③：

def add\_prop3(data):

loan = data.融资总额.astype(float)

return loan

test3=df2.groupby('车辆所属城市').apply(add\_prop3)

test3 （数据格式是Series，大小是100）

|  |  |  |
| --- | --- | --- |
| 上海 | 84 | 115210.0 |
|  | 85 | 131750.0 |
| 东莞 | 98 | 30650.0 |
| 临沂 | 27 | 58580.0 |
|  | 92 | 60308.0 |
| …… | …… | …… |
| 长沙 | 2 | 125538.0 |
|  | 12 | 92170.0 |
|  | 24 | 95460.0 |
|  | 44 | 85980.0 |

备注：说明如果参数只传入groupby()，则输出的还是原数据。但是如果传入计算值，即groupby().字段名，则输出结果采用的是groupby()后的形式。根据解析④，可以知道55个城市，每个城市一个Series

解析④：

def add\_prop4(data):

loan = data.融资总额.astype(float)

return type(loan)

test4=df2.groupby('车辆所属城市').apply(add\_prop4)

test4 （数据格式是Series，大小是55）

|  |  |
| --- | --- |
| 上海 | <class 'pandas.core.series.Series'> |
| 东莞 | <class 'pandas.core.series.Series'> |
| 临沂 | <class 'pandas.core.series.Series'> |
| …… | …… |
| 长春 | <class 'pandas.core.series.Series'> |
| 长沙 | <class 'pandas.core.series.Series'> |

备注：结果共55个城市

解析⑤：

def add\_prop5(data):

loan = data.融资总额.astype(float)

return loan.sum()

test5=df2.groupby('车辆所属城市').apply(add\_prop5)

test5 （数据格式是Series，大小是55）

|  |  |
| --- | --- |
| 上海 | 246960.0 |
| 东莞 | 30650.0 |
| 临沂 | 118888.0 |
| …… | …… |
| 长沙 | 399148.0 |

解析⑥：

prop=test3/test5

prop （数据格式是Series，长度是100）

|  |  |  |
| --- | --- | --- |
| 上海 | 84 | 0.466513 |
|  | 85 | 0.533487 |
| 东莞 | 98 | 1.000000 |
| 临沂 | 27 | 0.492733 |
|  | 92 | 0.507267 |
| …… | …… | …… |
| 长沙 | 2 | 0.314515 |
|  | 12 | 0.230917 |
|  | 24 | 0.239159 |
|  | 44 | 0.215409 |

备注：猜测——长Series/短Series=短Series

解析⑦

loan2=df2.groupby('车辆所属城市').融资总额.astype(float)

loan2

报错提示：

Cannot access callable attribute 'astype' of 'SeriesGroupBy' objects, try using the 'apply' method

备注：说明astype等方法不能应用在groupby()上，只用应用在groupby()里的Series上。而groupby转换为Series，只有传入apply()才行

举例2：选出每个城市中价格前5的车辆

第一步：编写函数

def top(df,n=5,column='车辆价格'):

return df.sort\_values(by=column)[-n:]

第二步：测试函数

top(df3,n=5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 客户性别 | 车辆所属城市 | 车辆价格 | 融资总额 |
| 车辆所属城市 |  |  |  |  |
| 北京 | 女 | 北京 | 830000 | 500000 |
| 鄂尔多斯 | 女 | 鄂尔多斯 | 840000 | 420000 |
| 北京 | 男 | 北京 | 850000 | 350000 |
| 北京 | 男 | 北京 | 850000 | 350000 |
| 河池 | 男 | 河池 | NaN | NaN |

第三步：应用到groupby上

df3.groupby('车辆所属城市').apply(top)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 客户性别 | 车辆所属城市 | 车辆价格 | 融资总额 |
| 车辆所属城市 |  |  |  |  |  |
| 万州 | 17725 | 女 | 万州 | 170000 | 146462 |
|  | 21619 | 女 | 万州 | 170000 | 146462 |
|  | 15770 | 女 | 万州 | 210000 | 180920 |
|  | 21821 | 男 | 万州 | 254800 | 219520 |
|  | 10865 | 男 | 万州 | 260000 | 224000 |
| 三亚 | 9168 | 男 | 三亚 | 140000 | 120615 |
|  | 4318 | 女 | 三亚 | 150000 | 120970 |
|  | 5409 | 男 | 三亚 | 158000 | 136120 |
|  | 14707 | 女 | 三亚 | 230000 | 198150 |
|  | 32060 | 女 | 三亚 | 315000 | 254032 |
| 三明 | 8108 | 女 | 三明 | 78800 | 67890 |
|  | 17929 | 男 | 三明 | 116500 | 100369 |
|  | 15839 | 女 | 三明 | 135000 | 116308 |
|  | 12137 | 男 | 三明 | 250000 | 150000 |
|  | 25419 | 女 | 三明 | 357000 | 287903 |

# 为apply() 传入参数

df3.groupby(['客户性别','车辆所属城市']).apply(top,n=1,column='车辆价格')

# 举例：计算各城市融资额最高的10个城市

def top\_dealer(df,key,n=2):

totals=df.groupby(key)['融资总额'].sum()

return totals.sort\_values(ascending=False)[:n]

grouped=df1.groupby('车辆所属城市')

grouped.apply(top\_dealer,'商家名称',n=2)

**# 关于groupby()**

**包括SeriesGroupBy,DataFrameGrouby**

# groupby之后的数据并不是DataFrame格式的数据，而是特殊的groupby类型，此时，可以通过size()方法返回分组后的记录数目统计结果，该结果是Series类型

# 通过groupby完成对数据的分组后，可以通过get\_group方法来获取某一制定分组的结果

对于多层分组后的数据，如果需要使用多重索引，此处需要传入一个元组

# 对于分组后的数据，如果还需要进行一系列查询，可以使用apply方法，传入一个处理的函数

# agg函数提供基于列的聚合操作。而groupby可以看做是基于行，或者说index的聚合操作

# GroupBy对象是一个迭代对象，每次迭代结果是一个元组，元组的第一个元素是该组的名称(就是groupby的列的元素名称)，第二个元素是该组的具体信息，是一个数据框，索引是以前的数据框的总索引

解析：

gb\_test = df2.groupby('车辆所属城市')

for name,data in test:

print(name) # 数据格式是 str

print(data) # 数据格式是 DataFrame

# 在GroupBy对象上调用descripe

# 原理：实际上只是应用了下面两条代码的快捷方式

f=lambda x:x.describe()

grouped.apply(f)

实际代码：

result=df3.groupby('车辆所属城市')['车辆价格'].describe()

result

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | count | mean | std | min | 25% | 50% | 75% | max |
| 车辆所属城市 |  |  |  |  |  |  |  |  |
| 万州 | 25 | 103160 | 63820.95 | 35000 | 65000 | 78000 | 119800 | 260000 |
| 三亚 | 25 | 100852 | 62806.93 | 39500 | 63000 | 88000 | 120000 | 315000 |
| 三明 | 17 | 99276.47 | 82221.85 | 50500 | 56800 | 68800 | 78800 | 357000 |
| 三河 | 27 | 82033.33 | 32138.27 | 36000 | 56000 | 75000 | 114000 | 145000 |

# 禁止分组键成为层次化索引

df3.groupby('车辆所属城市',**group\_keys=False**).apply(top)

**# 分位数和桶分析（groupby结合cut和qcut）P281**

# **桶（bucket）** 分位数（quantile）

# 举例：根据车辆价格将车辆拆分为4个**“面元”（bin）**。人为拆分

price=[0,80000,120000,200000,1000000]

factor=pd.cut(df3.车辆价格,price)

factor

|  |  |
| --- | --- |
| 0 | (120000, 200000] |
| 1 | (120000, 200000] |
| 2 | (120000, 200000] |
| 3 | (0, 80000] |
| …… | …… |
| 36548 | (0, 80000] |

# 由cut返回的Factor对象可直接用于groupby。

示例：

def get\_status(group):

return {'min':group.min(),'max':group.max(),'count':group.count(),'mean':group.mean()}

grouped=df3.车辆价格.groupby(factor)

grouped.apply(get\_status)

|  |
| --- |
| (0, 80000] count 14987.000000 |
| max 80000.000000 |
| mean 59636.291252 |
| min 20000.000000 |
| (80000, 120000] count 7781.000000 |
| max 120000.000000 |
| mean 101841.220666 |
| min 80100.000000 |

grouped.apply(get\_status).unstack()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | count | max | mean | min |
| 车辆价格 |  |  |  |  |
| (0, 80000] | 14987 | 80000 | 59636.29 | 20000 |
| (80000, 120000] | 7781 | 120000 | 101841.2 | 80100 |
| (120000, 200000] | 8128 | 200000 | 153130.6 | 120100 |
| (200000, 1000000] | 5652 | 850000 | 304685.1 | 200100 |

# 使用样本分位数来拆分面元

# 传入labels=False即可只获取分位数的编号

grouping=pd.qcut(df3.车辆价格,5,labels=False)

grouping

|  |  |
| --- | --- |
| 0 | 3.0 |
| 1 | 3.0 |
| 2 | 3.0 |
| …… | …… |
| 36548 | 1.0 |

grouping2=pd.qcut(df3.车辆价格,5)

grouping2

|  |  |
| --- | --- |
| 0 | (117000.0, 174100.0] |
| 1 | (117000.0, 174100.0] |
| 2 | (117000.0, 174100.0] |
| …… | …… |
| 36548 | (60000.0, 80000.0] |

grouped2=df3.车辆价格.groupby(grouping)

grouped2.apply(get\_status).unstack()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | count | max | mean | min |
| 车辆价格 |  |  |  |  |
| 0 | 7702 | 60000 | 48328.06 | 20000 |
| 1 | 7285 | 80000 | 71591.82 | 60100 |
| 2 | 6961 | 117000 | 99788.23 | 80100 |
| 3 | 7297 | 174100 | 141699.9 | 117200 |
| 4 | 7303 | 850000 | 278042.2 | 174200 |

备注：（更精确的情况下，count值应该都一样）

**# 关于索引**

# 添加索引

df3.index=df3['资金渠道'] ——同为列和索引，可能会造成问题

# 去除索引

df3=df3.reset\_index(drop=True)

df3

**# 层次化索引**

# 举例：统计不同性别不同城市的车辆平均价格（其实不如用pivot\_table）

df3=df1[['客户性别','车辆所属城市','车辆价格','融资总额']]

means=df3['车辆价格'].groupby([df3['客户性别'],df3['车辆所属城市']]).mean()

means

或者简化为：

df3.groupby(['客户性别','车辆所属城市'])['车辆价格'].mean()

|  |  |  |
| --- | --- | --- |
| 客户性别 | 车辆所属城市 |  |
| 女 | 万州 | 134800.00000 |
|  | 三亚 | 138062.50000 |
|  | 三明 | 113137.50000 |

备注：输出结果是Series（因为只有一列值）

# 将上述结果转换为一层索引，转换为DataFrame。

？是不是只有两层索引才能通过unstack()转换成DataFrame

means.unstack()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 车辆所属城市  客户性别 | 万州 | 三亚 | 三明 | …… | 龙岩 |
| 女 |  |  |  |  |  |
| 男 |  |  |  |  |  |

# 以无索引的形式返回聚合数据——即将层次化索引转为为普通的列

df1.groupby(['客户性别','车辆所属城市'],as\_index=False).mean()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 客户性别 | 车辆所属城市 | 订单编号 | 客户ID | …… | 利息 |
| 0 | 女 | 万州 |  |  |  |  |
| 1 | 女 | 三亚 |  |  |  |  |
| 2 | 女 | 三明 |  |  |  |  |
| 3 | 女 | 三河 |  |  |  |  |

df1.groupby(['客户性别','车辆所属城市'],as\_index=False)['车辆价格'].mean()

|  |  |  |  |
| --- | --- | --- | --- |
|  | 客户性别 | 车辆所属城市 | 车辆价格 |
| 0 | 女 | 万州 | 134800.00 |
| 1 | 女 | 三亚 | 138062.50 |
| 2 | 女 | 三明 | 113137.50 |
| 3 | 女 | 三河 | 85000.00 |

**# 面向列的多函数应用**

# 综合计算。计算根据省份分组的 “经销商结算价” 的总和、均值、标准差。也可传入自定义函数。（说白了还是针对同一列）

grouped=df1.groupby('车辆所属城市')

grouped\_car=grouped['车辆价格']

grouped\_car=grouped\_car.agg(['sum','mean','std'])

grouped\_car

|  |  |  |  |
| --- | --- | --- | --- |
| 车辆所属城市 | Sum | Mean | Std |
| 万州 |  |  |  |
| 三亚 |  |  |  |
| 三明 |  |  |  |

# 自定义列名称。传入由（name,function）元祖组成的列表

grouped\_car2=df1.groupby('车辆所属城市')['车辆价格'].agg([('车辆总价','sum'),('车辆平均价格','mean')])

grouped\_car2

|  |  |  |
| --- | --- | --- |
| 车辆所属城市 | 车辆总价 | 车辆平均价格 |
| 万州 |  |  |
| 三亚 |  |  |
| 三明 |  |  |

# 不同的列，应用相同的函数

grouped=df1.groupby(['客户性别','车辆所属城市'])

functions=['count','mean','max']

result=grouped['车辆价格','融资总额'].agg(functions)

result

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 车辆价格 | | | 融资总额 | | |
|  |  | count | mean | max | count | mean | max |
| 客户性别 | 车辆所属城市 |  |  |  |  |  |  |
| 女 | 万州 |  |  |  |  |  |  |
|  | 三亚 |  |  |  |  |  |  |
|  | 三明 |  |  |  |  |  |  |

# 相当于对各列进行聚合，然后用concat将结果组装到一起

result[‘车辆价格’]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | count | mean | max |
| 客户性别 | 车辆所属城市 |  |  |  |
| 女 | 万州 |  |  |  |
|  | 三亚 |  |  |  |
|  | 三明 |  |  |  |

# 也可以自定义列名：通过传入带有自定义名称的元组列表

ftuples=[('计数','count'),('平均数','mean')]

grouped['车辆价格','融资总额'].agg(ftuples)

# 对不同的列应用不同的函数。具体的办法是想agg传入一个从列名映射到函数的字典。

grouped.agg({'车辆价格':'mean','融资总额':['count','mean',’max’]})

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 车辆价格 | 融资总额 | | |
|  |  | mean | count | mean | max |
| 客户性别 | 车辆所属城市 |  |  |  |  |
| 女 | 万州 |  |  |  |  |
|  | 三亚 |  |  |  |  |
|  | 三明 |  |  |  |  |

**# 分组级运算和转换——即“面向分组”的计算**

**# 分类计算后与原数据合并**

# 先按照“资金渠道”分组，然后添加一列，为“首付款金额”和“经销商结算价”的平均值

# 方法一：先聚合再合并，即将计算的结果加在原来的数据上

new\_col=df3.groupby('车辆所属城市').mean().add\_prefix('mean\_')

new\_col

|  |  |  |
| --- | --- | --- |
|  | mean\_车辆价格 | mean\_融资总额 |
| 车辆所属城市 |  |  |
| 万州 |  |  |
| 三亚 |  |  |
| …… |  |  |
| 龙岩 |  |  |

备注：add\_prefix()应该可以应用于一般的DataFrame

pd.merge(df3,new\_col,left\_on='车辆所属城市',right\_index=True)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 客户性别 | 车辆所属城市 | 车辆价格 | 融资总额 | mean\_车辆价格 | mean\_融资总额 |
| 0 | 男 | 烟台 | 170000.0 | 146460.0 | 137798.333333 | 111537.042000 |
| 1105 | 男 | 烟台 | 110000.0 | 99580.0 | 137798.333333 | 111537.042000 |
| 1280 | 男 | 烟台 | 110000.0 | 80000.0 | 137798.333333 | 111537.042000 |
| 1355 | 男 | 烟台 | 94800.0 | 81670.0 | 137798.333333 | 111537.042000 |
| 1582 | 女 | 烟台 | 130000.0 | 112000.0 | 137798.333333 | 111537.042000 |

备注：可见是通过第一个DataFrame的列和第二个DataFrame的索引进行合并的

# 方法二：使用transform。但是这样没显示出对应的主键（解决办法：先将某一列置为索引）

df3.groupby('车辆所属城市').transform(np.mean)

输出：（所有的行）

|  |  |  |
| --- | --- | --- |
|  | 车辆价格 | 融资总额 |
| 0 | 137798.333333 | 111537.042000 |
| 1 | 138104.800000 | 108222.352000 |
| …… | …… | …… |
| 36548 | 131695.376247 | 106573.701723 |

# 使用transform（）应用函数

# 距平化函数；设置索引

输出：（所有行）

|  |  |  |
| --- | --- | --- |
|  | 车辆价格 | 融资总额 |
| 车辆所属城市 |  |  |
| 烟台 | 32201.666667 | 34922.958000 |
| 承德 | -3104.800000 | 8087.648000 |
| 长沙 | 17597.368421 | 3692.836546 |
| 沈阳 | -80513.921547 | -65664.352486 |
| 沈阳 | -52513.921547 | -41544.352486 |
| …… | …… | …… |

**关于Series**

# 创建Series

通过列表创建。创建带索引的Series

obj=Series([4,7,-5,3],index=['d','b','a','c'])

obj

输出：

d 4

b 7

a -5

c 3

#通过字典来创建Series

sdata={'Ohio':35000,'Texas':71000,'Oregon':16000,'Utah':5000}

obj2=Series(sdata)

obj2

输出：

Ohio 35000

Oregeon 16000

Texas 71000

Utah 5000

#通过字典创建Series时，可以单独提供索引

states=['Califonia','Ohio','Oregon','Texas']

obj3=Series(sdata,index=states)

obj3

输出：

Califonia NaN

Ohio 35000.0

Oregon 16000.0

Texas 71000.0

# Series导出为Excel （方法：先转换成DataFrame，再导出为Excel）

city\_counts = df3['车辆所属城市'].value\_counts()

city\_counts

pd.DataFrame(city\_counts).to\_excel("D:\\to\_excel.xlsx",sheet\_name='Sheet1')

#pandas的isnull和notnull函数

pd.isnull(obj3)

输出：

Califonia True

Ohio False

Oregon False

Texas False

# 计数

举例：统计8万以下/8万以上车的数量

(df3.车辆价格<=80000).value\_counts()

输出：

False 21562

True 14987

# 排重；唯一性

# 获取车商列表

unique\_dealer=df1.商家名称.unique()

unique\_dealer

输出：

array(['烟台港通商务服务有限公司', '承德天成精品二手车', '长沙中南市场深通名车车行', ..., '福州鼓山外围瑞杰精品车行', '太原万国二手车市场万成二部精品车行', '重庆八公里华骏龙二手车'], dtype=object)

**# 应用函数·map()** 有点类似于列表推导式。map的用法另见“映射”

map() 会根据提供的函数对指定序列做映射。第一个参数 function 以参数序列中的每一个元素调用 function 函数，返回包含每次 function 函数返回值的新列表。

举例：取出客户的姓（假设无复姓）

get\_surname=lambda x:x[0]

surname=df1.客户姓名.map(get\_surname)

surname

**其它**

# 禁用科学计数法

pd.set\_option('display.float\_format',lambda x:'%.5f' % x)

# 用特定于分组的值填充缺失值 P282

fillna

#结合fillna、groupby和lambda，为不同的分组填充不同的值

#也可以在代码中预定义各组的填充值，而不只是描述统计函数的值

# 随机采样和排列

np.random.permutation(N)与take()函数

# 分组加权平均数和相关系数

np.average()

corrwith()

# 面向分组的线性回归

statsmodels库

# 映射

# 举例：将“业务期限”映射如下，并添加到最后一列

|  |  |
| --- | --- |
| 业务期限 | 是否正常 |
| 24/36 | true |
| 33/21/34/23 | false |

先利用字典进行说明

isNormal={'24个月':'true','36个月':'true','33个月':'false','21个月':'false','34个月':'false','23个月':'false'}

df1.业务期限.map(isNormal)

|  |  |
| --- | --- |
| 0 | true |
| 1 | true |
| …… | …… |
| 36548 | true |

# 将其添加为一个新列

df1['isNormal']=df1.业务期限.map(isNormal)

df1['isNormal'].value\_counts()

输出：

true 36499

false 16

**应用举例**

1. Excel中有一列‘对话’，统计‘对话’中包含‘再见’字样的总数

|  |  |
| --- | --- |
| **姓名** | 对话 |
| 林炳科 | 机器人：您好，请问您是机主本人吗 |
| 林炳科 | 客户：啊 喂 |
| 林炳科 | 机器人：这里是玖富借贷服务平台，您在我平台申请的万卡借款已经逾期。想问一下，您是什么原因没有处理欠款呢 |
| 林炳科 | 客户：我已经知道了 我等下到家了 十一点多就处理掉。 |

方法一：列表推导式

Import panda as pd

Result = pd.read\_excel(“D:\\test.xlsx”,encoding=’gbk’)

bye = [dialogue for dialogue in result['对话'] if '再见' in dialogue]

len(bye)

方法二：groupby

contain\_bye = np.where(df2['对话'].str.contains('再见'),'包含','不包含')

contain\_bye

输出：array(['不包含', '不包含', '不包含', ..., '不包含', '不包含', '包含'], dtype='<U3')

group\_bye = df2.groupby([contain\_bye])

group\_bye

输出：<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x00000145D5833E48>

bye\_counts = group\_bye.size()

bye\_counts

输出：

不包含 16363

包含 3015

方法三：

直接将包含“再见”字样的数据提取出来

contain\_bye = np.where(result[‘对话’].str.contains(‘再见’),True,False)

contain\_bye

输出：

array([ True, False, True, ..., True, False, True])

df\_contain\_bye = result[contain\_bye]

len(df\_contain\_bye)

**关于NumPy**

**# 创建ndarray**

# 使用列表创建

data1=[[1,2,3,4],[5,6,7,8]]

arr1=np.array(data1)

arr1

输出：

array([[1, 2, 3, 4],

[5, 6, 7, 8]])

# 制定数据格式

arr2=np.array([1,2,3],dtype=np.float64)

arr3=np.array([1,2,3],dtype=np.int32)

#显示转换dtype

arr5=np.array([1,2,3,4,5])

arr5.dtype

float\_arr=arr5.astype(np.float64)

float\_arr.dtype

# 使用数组创建函数（数组创建函数见P85）

np.zeros(10)

np.zeros((3,6)) # 传入元组

np.ones(10)

np.ones((3,6))

np.empty((2,3,2)) # 返回未初始化的垃圾值

np.arange(15)

# 使用numpy.random中的randn函数生成一些正态分布的随机数据

data\_randn = np.random.randn(7,3)

data\_randn

输出：

array([[ 0.08106514, -0.00402809, -0.15916213],

[ 0.11194186, 0.08986413, -0.51107223],

[-0.38693212, -0.79643107, -0.00954642],

[ 0.528887 , 0.13452351, -0.35322715],

[ 0.29913157, -0.79868638, 1.07677309],

[-1.7277721 , 0.09533197, 0.77298737],

[-0.78740315, -0.77150805, -1.18332798]])

# 生成bool型数组

names=np.array(['Bob','Joe','Will','Bob','Will','Joe','Joe'])

names=='Bob'

输出：

array([ True, False, False, True, False, False, False])

# 数组的维数、大小、类型

arr1.ndim

arr1.shape

arr1.dtype

**# 数组的计算**

# 二位数组相乘

arr=np.array([[1,2,3],[4,5,6]])

array([[ 1, 4, 9],

[16, 25, 36]])

**# 索引——布尔型索引**

# 布尔值索引

data\_randn[names==’Bob’]

输出：

array([[ 0.08106514, -0.00402809, -0.15916213],

[ 0.528887 , 0.13452351, -0.35322715]])

# 布尔型索引和切片相结合。（切片切的是列而不是行）

data\_randn[names==’Bob’,1:]

输出：

array([[-0.00402809, -0.15916213],

[ 0.13452351, -0.35322715]])

# 布尔型数组与整数组合（降维）

data\_randn[names=='Bob',2]

输出：

array([-0.15916213, -0.35322715])

# 用!=或者~号进行否定

names != 'Bob'

data[~(names=='Bob')]

# 用逻辑运算符组合多个布尔条件

mask = (names=='Bob')|(names=='Will')

mask

输出：

array([ True, False, True, True, True, False, False])

data\_randn[mask]

输出：

array([[ 0.08106514, -0.00402809, -0.15916213],

[-0.38693212, -0.79643107, -0.00954642],

[ 0.528887 , 0.13452351, -0.35322715],

[ 0.29913157, -0.79868638, 1.07677309]])

# 通过布尔型数组设置值

data\_randn[data\_randn<0]=0

data\_randn

输出：

array([[0.08106514, 0. , 0. ],

[0.11194186, 0.08986413, 0. ],

[0. , 0. , 0. ],

[0.528887 , 0.13452351, 0. ],

[0.29913157, 0. , 1.07677309],

[0. , 0.09533197, 0.77298737],

[0. , 0. , 0. ]])

# 通过一维布尔数组设置整行或者整列的值

data\_randn[names!='Joe']=7

data\_randn

输出：

array([[7. , 7. , 7. ],

[0.11194186, 0.08986413, 0. ],

[7. , 7. , 7. ],

[7. , 7. , 7. ],

[7. , 7. , 7. ],

[0. , 0.09533197, 0.77298737],

[0. , 0. , 0. ]])

**# 索引——花式索引**（即利用整数数组进行索引）

arr=np.empty((8,4))

for i in range(8):

arr[i]=i

arr

输出：

array([[0., 0., 0., 0.],

[1., 1., 1., 1.],

[2., 2., 2., 2.],

[3., 3., 3., 3.],

[4., 4., 4., 4.],

[5., 5., 5., 5.],

[6., 6., 6., 6.],

[7., 7., 7., 7.]])

# 以特定顺序选取子集，只需传入一个用于指定顺序的整数列表或ndarray即可

# 选取行子集

arr[[4,3,0,6]]

输出：

array([[4., 4., 4., 4.],

[3., 3., 3., 3.],

[0., 0., 0., 0.],

[6., 6., 6., 6.]])

# 负数索引

arr[[-3,-5,-7]]

输出：

array([[5., 5., 5., 5.],

[3., 3., 3., 3.],

[1., 1., 1., 1.]])

# 一次传入多个索引数组，返回一维数组，其中的元素对应各个索引元祖

（类似于通过行和列选出某一个单元格）

# arange：创建等差数组 reshape：用来改变数组的形状

arr=np.arange(32).reshape((8,4))

arr

输出：

array([[ 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]])

# 选出（1,0）（5，3）（7,1）（2,2）

arr[[1,5,7,2],[0,3,1,2]]

输出：

array([ 4, 23, 29, 10])

# 按顺序选取行子集

arr[[1,5,7,2]]

输出：

array([[ 4, 5, 6, 7],

[20, 21, 22, 23],

[28, 29, 30, 31],

[ 8, 9, 10, 11]])

# 根据特定顺序选取行列子集。

方法一：

arr[[1,5,7,2]][:,[0,3,1,2]]

输出：

array([[ 4, 7, 5, 6],

[20, 23, 21, 22],

[28, 31, 29, 30],

[ 8, 11, 9, 10]])

方法二：np.ix\_函数

arr[np.ix\_([1,5,7,2],[0,3,1,2])]

输出：

array([[ 4, 7, 5, 6],

[20, 23, 21, 22],

[28, 31, 29, 30],

[ 8, 11, 9, 10]])

**# 数组转置和轴对换**

arr=np.arange(15).reshape((3,5))

arr

输出：

array([[ 0, 1, 2, 3, 4],

[ 5, 6, 7, 8, 9],

[10, 11, 12, 13, 14]])

——转置

arr.T

输出：

array([[ 0, 5, 10],

[ 1, 6, 11],

[ 2, 7, 12],

[ 3, 8, 13],

[ 4, 9, 14]])

# 对于高维数组，transpose需要得到一个由轴编号组成的元祖才能对这些轴进行转置（比较费脑子）

arr=np.arange(16).reshape((2,2,4))

arr

输出：

array([[[ 0, 1, 2, 3],

[ 4, 5, 6, 7]],

[[ 8, 9, 10, 11],

[12, 13, 14, 15]]])

——转置

arr.transpose((1,0,2))

输出：

array([[[ 0, 1, 2, 3],

[ 8, 9, 10, 11]],

[[ 4, 5, 6, 7],

[12, 13, 14, 15]]])

# 用ndarray的swapaxes方法进行转置，它需要接受一对轴编号。 ***视图***

arr.swapaxes(1,2)

输出：

array([[[ 0, 4],

[ 1, 5],

[ 2, 6],

[ 3, 7]],

[[ 8, 12],

[ 9, 13],

[10, 14],

[11, 15]]])

**# 通用函数（ufunc）：快速的元素级数组函数**

# 可将其看作简单函数（接受一个或多个标量值，并产生一个或多个标量值）的矢量化包装器

**# 一元ufunc。即参数是一个数组**

arr=np.arange(10)

arr

输出：

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# 平方根（一元ufunc）

np.sqrt(arr)

输出：

array([0. , 1. , 1.41421356, 1.73205081, 2. 2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.])

# 计算各元素的指数值（一元ufunc）

np.exp(arr)

输出：

array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,

5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,

2.98095799e+03, 8.10308393e+03])

**# 二元ufunc：接受2个数组，并返回一个结果数组。即参数是2个数组**

# 比较两个数组

x=np.random.randn(8)

y=np.random.randn(8)

x：

array([-1.68098552, -0.224441 , -1.20091248, 0.16692036, 0.5718666 ,

-0.21989944, -0.67545525, 0.53567895])

y:

array([-1.83208736, 2.0931789 , -0.20409594, 0.38107422, 0.23477104,

0.2576702 , 0.03032071, 1.07594208])

np.maximum(x,y)

输出：

array([-1.68098552, 2.0931789 , -0.20409594, 0.38107422, 0.5718666 ,

0.2576702 , 0.03032071, 1.07594208])

**# 矩阵**

# 计算矩阵內积 P97

**字符串操作**

**# 正则表达式**

（re模块的三大类函数：模式匹配、替换、拆分）

**读取文件**

# 读取CSV文件

举例：读取前5行

方法一：直接取出前5行

Result=pd.read\_csv(‘G:\\xin.csv’,encoding=’gbk’,**nrows=5**)

Result

方法二：取出全部，只展示前5行

Result=pd.read\_csv(‘G:\\xin.csv’,encoding=’gbk’)

Result[:5]

# 读取txt文件

f = open('D:/corpus.txt', 'r')

lines = f.readlines()

for line in lines:

line\_list = line.split(',')

row = lines.index(line)

**其它**

# 为节省read\_csv的开销，可提前制定dtype

import pandas as pd

from StringIO import StringIO

csvdata = """user\_id,username

1,Alice

3,Bob

4,Caesar"""

sio = StringIO(csvdata)

pd.read\_csv(sio, dtype={"user\_id": int, "username": object})