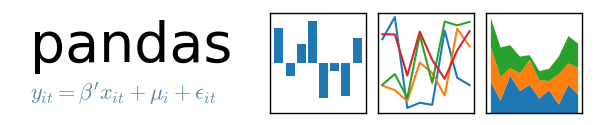
## Pandas简介

### Pandas是什么？

[](http://pandas.pydata.org/index.html)

Pandas是Panel data and data analysis的简写，是一个专门用来数据处理和数据分析的Python工具库。

### 特点

基于numpy库

底层C实现，效率极高

对矩阵(高维数组)做计算

使用起来用在excel/R/SQL，快速入手

### 相关资源：

pandas官方网站《10 Minutes to pandas》， [这里](http://pandas.pydata.org/pandas-docs/stable/10min.html)。

pandas速查表

## Pandas中重要的数据类型：series与dataframe。

### Series

一个series是一个一维的数据类型，其中每一个元素都有一个标签。series类似于Numpy中元素带标签的数组。其中，index标签可以是数字或者字符串。用以作为行号来标识一行数据。

#导入相应库

import numpy as np

import pandas as pd

#### 从列表初始化series[默认数字为index]

my\_list=[7,'Beijing','19',3.1415,-10000,'Happy']

series1=pd.Series(my\_list)

#从列表初始化Series

series1[1]#结果为'Beijing'

series1[1:4]#Series这种数据类型默认用0到N这种格式做为索引index

b Beijing

c 19

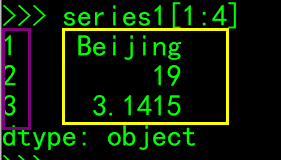
d 3.1415

dtype: object

什么是Pandas里的index呢？

可以理解为Python字典中的key,可以通过这个key访问对应的值。

那Series就可以想象成为一个KV对应的结构。



#### 从列表初始化series[指定index]

my\_list=[7,'Beijing','19',3.1415,-10000,'Happy']

series1=pd.Series(my\_list, index=['a','b','c','d','e','f'])

#从列表初始化Series，并自定义index

a 7

b Beijing

c 19

d 3.1415

e -10000

f Happy

dtype: object

#使用自定义index访问Series

series1['a']

7

series1['f']

'Happy'

即使是自定义索引来初始化Series了，依旧可以通过数字 来访问。也就是说数字做为index始终是默认的。

series1[1:4]

b Beijing

c 19

d 3.1415

dtype: object

s1=pd.Series([1,2,3,4,5,6]

index=pd.date\_range('20130101',periods=6))#带标签，标签是时间序列

s2=pd.Series(list('china! '))#不指定标签，系统自动是从0开始，

#### 从字典初始化series

因为字典这种数据结构本身就是kv对应型的，因为可以直接用来构建series。index自动为字典的key

cities={'Beijing':55000,'Shanghai':60000,'Shenzhen':50000,'Hangzhou':20000,'Guangzhou':45000,'Suzhou':None}

series3=pd.Series(cities)

series3

Beijing 55000.0

Guangzhou 45000.0

Hangzhou 20000.0

Shanghai 60000.0

Shenzhen 50000.0

Suzhou NaN

dtype: float64

#### 从numpy对象初始化series

series4=pd.Series(np.random.rand (5),index=[list('abcde')])

series4

a -0.362932

b 1.670863

c -0.665555

d 0.402865

e 1.789551

dtype: float64

#### 快速理解：

**我们可以快速理解Series为一个列,我们通过index去指定访问某些行。在创建Series我们也可以指定它的index，数字index始终是默认的。**

### DataFrame

一个dataframe是一个二维的表结构。Pandas的dataframe可以存储许多种不同的数据类型，并且每一个坐标轴都有自己的标签。你可以把它想象成一个series的字典项。

一个DataFrame就是一张表格。Series可以理解为一维数组，DataFrame是一个二维数据，DataFrame是由多个Series组成的（可以理解为Series的集合）

#### 从已有Series构建

s1=pd.Series(list('china'))

s2=pd.Series(list('powers'))

df2=pd.DataFrame({'s1':s1,'s2':s2})

df2#两个Series按index对齐（这里的index为默认的0至N），如果对应的index没有，则填为NaN

s1 s2

0 c p

1 h o

2 i w

3 n e

4 a r

5 NaN s

#### 从已有字典构建

population={'city':['Beijing','Shanghai','Guangzhou','Shenzhen','Hangzhou','Chongqi'],'year':[2016,2017,2016,2017,2016,2016],

'population':[2100,2300,1000,700,500,500]

}

df1=pd.DataFrame(population)

df1

city population year

0 Beijing 2100 2016

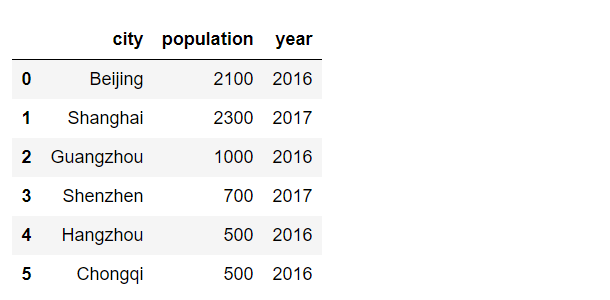
1 Shanghai 2300 2017

2 Guangzhou 1000 2016

3 Shenzhen 700 2017

4 Hangzhou 500 2016

5 Chongqi 500 2016



#### 从list的字典中构建

l1=[{'Jack':100000,'Han':80000,'David':10000},{'Jack':9999999,'Han':90000}]

df3=pd.DataFrame(l1)

df3

David Han Jack

0 10000.0 80000 100000

1 NaN 90000 9999999

## Series与DataFrame相应操作

可以把series理解为一列，dataframe可以理解为由很多个列（series）构成的表格。

我们可以像对待一个list一样series，完成各类操作

cities={'Beijing':55000,'Shanghai':60000,'Shenzhen':50000,'Hangzhou':20000,'Guangzhou':45000,'Suzhou':None }

series5=pd.Series(cities,name='incomes')

### 访问

#### Series切片访问

series5[:5]

Beijing 55000.0

Guangzhou 45000.0

Hangzhou 20000.0

Shanghai 60000.0

Shenzhen 50000.0

Name: incomes, dtype: object

#### 指定索引访问，注意这里位置可以任意调换

series5[[0,4,3,1,2]]

Beijing 55000.0

Shenzhen 50000.0

Shanghai 60000.0

Guangzhou 45000.0

Hangzhou 20000.0

Name: incomes, dtype: object

series5[['Shanghai','Suzhou']]

*Shanghai 60000.0*

*Suzhou NaN*

*dtype: float64*

### 元素检查

#### in关键字，类似于检查元素是否在列表中一样的操作

'Suzhou' in series5

True

'Nanjing' in series5

False

#### get方法访问，如果指定的index不存在，则返回nan

series5.get('Suzhou')

nan

series5.get('Beijing')

55000.0

### 条件boolean indexing

conditions=(series5==60000)

conditions

Beijing False

Guangzhou False

Hangzhou False

Shanghai True

Shenzhen False

Suzhou False

Name: incomes, dtype: object

按条件进行对series过滤后，会得到一个boolean型变量组成的Series对象,叫做**boolean indexing**。这个series可以传递作为条件，过滤掉所有相应位置为False的记录，得到最终结果。

series5[conditions]

Shanghai 60000.0

Name: incomes, dtype: object

#### 非空与非空值过滤

**找出所有的空值 或 非空值 的 boolean indexing#bool条件**

series5.notnull()#或-series5.isnull()

Beijing True

Guangzhou True

Hangzhou True

Shanghai True

Shenzhen True

Suzhou False

Name: incomes, dtype: bool

series5.isnull()#或-series5.notnull()

Beijing False

Guangzhou False

Hangzhou False

Shanghai False

Shenzhen False

Suzhou True

### 简单数学运算

#### 求均值，中位数，最大最小，

series5.mean()

46000.0

series5.median()

50000.0

series5[series5.isnull]=series5.mean()#把空项填充为均值

### 元素赋值

#### 指定Series的index赋值

series5['Shenzhen']=70000

series5['Shenzhen']

70000

#### 按boolean indexing条件过滤后赋值

series5[series5<50000]=50001

series5

Beijing 55000.0

Guangzhou 50001.0

Hangzhou 50001.0

Shanghai 60000.0

Shenzhen 50000.0

Suzhou NaN

Name: incomes, dtype: float64

### 数学运算

series5\*\*0.5#算开方，列表可不支持这样的操作，这也是pandas强大的地方

Beijing 234.520788

Guangzhou 223.609034

Hangzhou 223.609034

Shanghai 244.948974

Shenzhen 223.606798

Suzhou NaN

Name: incomes, dtype: float64

np.log(series5)

Beijing 10.915088

Guangzhou 10.819798

Hangzhou 10.819798

Shanghai 11.002100

Shenzhen 10.819778

Suzhou NaN

Name: incomes, dtype: float64

#### 运算->自动indexing对齐

Pandas有个强大的功能，叫做对齐。我们新建两个series,列的顺序是不是对齐的，而且两个series中都有另一个series中没有项，但依旧可以进行运算，结果为NaN

s1=pd.Series({'Beijing':55000,'Shanghai':60000,'Hangzhou':20000})

s2=pd.Series({'Shenzhen':50000,'Beijing':55000,'Shanghai':60000})

s1

Beijing 55000

Hangzhou 20000

Shanghai 60000

dtype: int64

s2

Beijing 55000

Shanghai 60000

Shenzhen 50000

dtype: int64

s1+s2

Beijing 110000.0

Hangzhou NaN

Shanghai 120000.0

Shenzhen NaN

dtype: float64

**总结：pandas自动对齐两个series的index，进行运算。**

## Index索引

理解index就是对应 每个行value的一个key。这样去理解简单些。

### 默认类型

population={'city':['Beijing','Shanghai','Guangzhou','Shenzhen','Hangzhou','Chongqi'],'year':[2016,2017,2016,2017,2016,2016],

'population':[2100,2300,1000,700,500,500]

}

df4.index#查看索引

RangeIndex(start=0, stop=6, step=1)

df默认索引均为从0开始，步长为1，总长为总行数减1的序列。

### 自定义索引并查看类型

pd.index(list('china'))

Index(['c', 'h', 'i', 'n', 'a'], dtype='object')

type(pd.Index(list('china')))

<class 'pandas.core.indexes.base.Index'>

### 使用自定义索引

df4.index=list('china!')

df4#可以看到这时的索引已经变成指定的字母了

city population year

c Beijing 2100 2016.0

h Shanghai 2300 NaN

i Guangzhou 1000 2016.0

n Shenzhen 700 2017.0

a Hangzhou 500 2016.0

! Chongqi 500 2016.0

df4.ix[['n','a'],[0,1]]#正常使用并访问

city population

n Shenzhen 700

a Hangzhou 500

### 多层index

series6=pd.Series(np.random.randn(10),index=[['a','a','a','b','b','c','c','d','d','d'],[1,2,3,1,2,1,2,1,2,3]])

series6.index

MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],

labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 0, 1, 0, 1, 2]])

多层index的访问

series6['b':'c']

b 1 0.198059

2 0.418935

c 1 -0.146183

2 0.134893

dtype: float64

series6[:2]

a 1 0.898547

2 0.132076

dtype: float64

### unstack

series6

a 1 0.898547

2 0.132076

3 -0.911465

b 1 0.198059

2 0.418935

c 1 -0.146183

2 0.134893

d 1 0.925714

2 0.657029

3 -0.237139

dtype: float64

series6.unstack()

1 2 3

a 0.898547 0.132076 -0.911465

b 0.198059 0.418935 NaN

c -0.146183 0.134893 NaN

d 0.925714 0.657029 -0.237139

type(series6.unstack())#unstack一个Series后，变为DataFrame了。

<class 'pandas.core.frame.DataFrame'>

千万不要因为pandas功能强大就给自己挖坑，开始的时候就老老实实地用二维表格，或者把多维表格降为二维。

## 导入与导出

### 中文问题

pd.read\_csv('d:\dell2.txt',sep='\t',encoding =**'gb18030'**)

### 从CSV开始

import pandas as pd

df6=pd.read\_csv('d:/ibm.txt',sep='\t')

### 从numpy开始

df7=pd.DataFrame(np.random.randn(6,4),columns=list('ABCD'))

### mysql

import pymysql

import pandas as pd

conn=pymysql.connect(host=,user=,passwd=,db=)

sql=''

df=pd.read\_sql(sql,conn)

### read\_html

需要beautifulsoup4，html5lib库支持

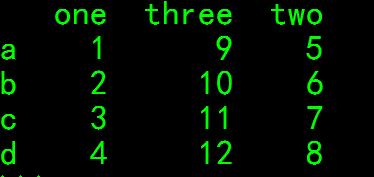
pd.read\_html('c:/1.html')

### 从Python字典中

dic3 = {'one':{'a':1,'b':2,'c':3,'d':4},'two':{'a':5,'b':6,'c':7,'d':8},'three':{'a':9,'b':10,'c':11,'d':12}}

df3 = pd.DataFrame(dic3)

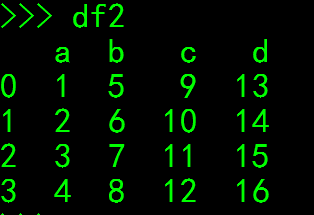
df3



dic2 = {'a':[1,2,3,4],'b':[5,6,7,8],'c':[9,10,11,12],'d':[13,14,15,16]}

df2 = pd.DataFrame(dic2)

df2



df=pd.DataFrame(google['chartlist'][0:3015],columns=google['chartlist'][1].keys())

### 从已有DF（数据框）中创建数据框

df4 = df3[['one','three']]

### 保存至csv

df.to\_csv('d:/ss.txt')

## DataFrame基础信息

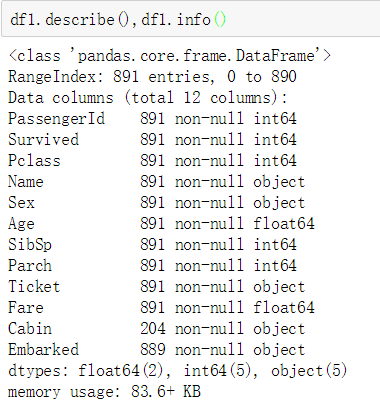
### DataFrame描述性命令

info()##了解dataframe整体结构，列，数据类型

describe()#了解数据分布

head()

tail()

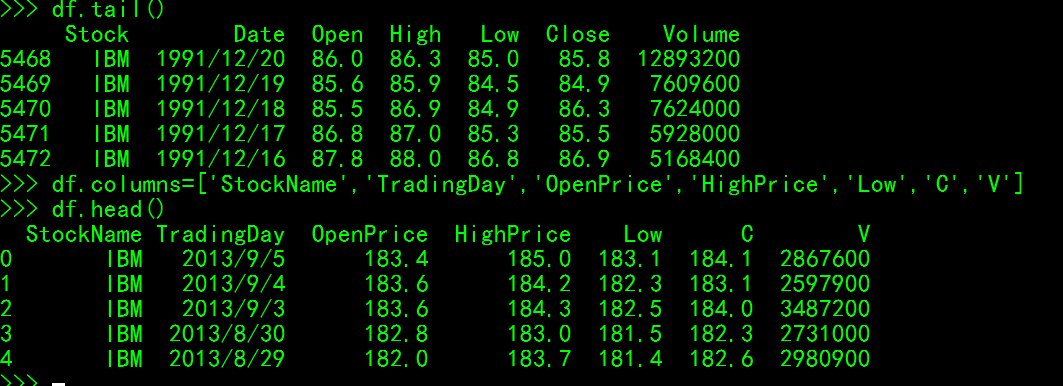


### 查看df对象大小

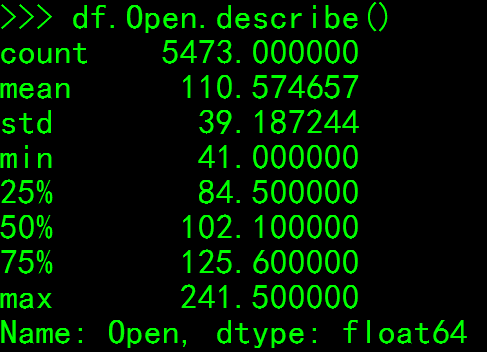
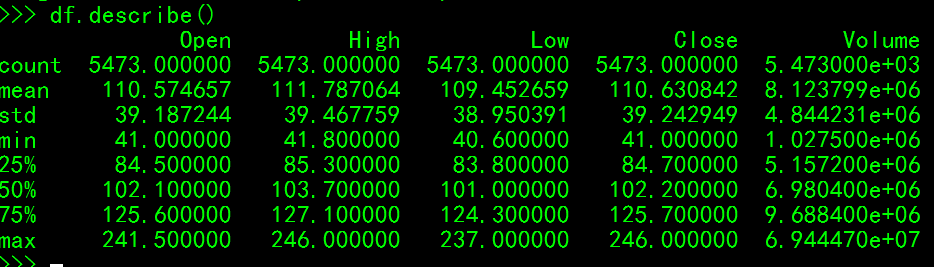
import sys

sys.getsizeof(df)

### head与tail



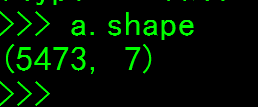
### 快速统计describe()



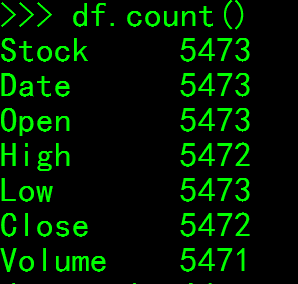
读取与columns列重命名



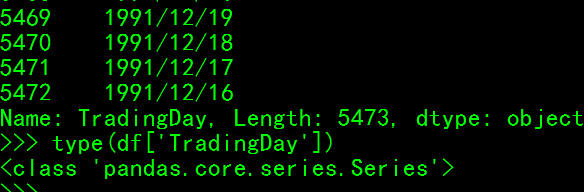
### Shape（返回行列元组）



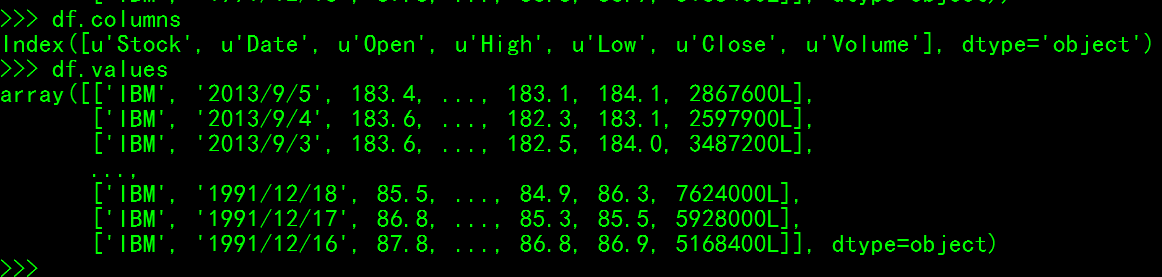
### df.count()统计非空个数



### 不同列的数据类型：dtypes

### df.columns,df.values,df,index



### 常用统计

#### max,min,mean,median,std,var,sum

## 访问方式

### 列访问

df.column\_name或df['column\_name']

### loc

按indx取出Series,通过行标签索引行列数据,有标签时可以通过标签没标签时可以通过行号。

df4.loc[0:2]

city population year bonus average

0 Beijing 2100 2016 6666 3.174286

1 Shanghai 2300 2017 6666 2.898261

2 Guangzhou 1000 2016 6666 6.666000

df4.loc[[0,3],['city','average']]#前边[0,3]代表选行，[‘city’,’average’]代表选列

city average

0 Beijing 3.174286

3 Shenzhen 9.522857

df4.loc[df4.average>10,['city','average']]#前边代表按列条件去过滤后剩下的行，['city', 'average']代表选列

### iloc

只能通过数字索引获来取数据

df4.iloc[1:,1:]

population year

1 2300 NaN

2 1000 2016.0

3 700 2017.0

4 500 2016.0

5 500 2016.0

### ix

混合索引，可以和标签，索引，行号混用。

df4.ix[:4,['city','population']]

city population

0 Beijing 2100

1 Shanghai 2300

2 Guangzhou 1000

3 Shenzhen 700

4 Hangzhou 500

### 条件组合与判断访问

#### 条件组合

(df4.population==2100)|(df4.population==1000)

0 True

1 False

2 True

3 False

4 False

5 False

df4.loc[((df4.populatin==2100)|(df4.population==1000)),:]

city population year

0 Beijing 2100 2016

2 Guangzhou 1000 2016

#### 指定条件

df4.population.isin([2100,1000])

0 True

1 False

2 True

3 False

4 False

5 False

可以看到，使用isin函数得到的是一个boolean indexing

Pandas里的条件否定，可以简单前边加一个负号即可。

-df4.population.isin([2100,1000])

0 False

1 True

2 False

3 True

4 True

5 True

Name: population, dtype: bool

df4.loc[df4.population.isin([2100,1000]),:]#isin的传入参数可以是一个列表,ndarray,Series

city population year

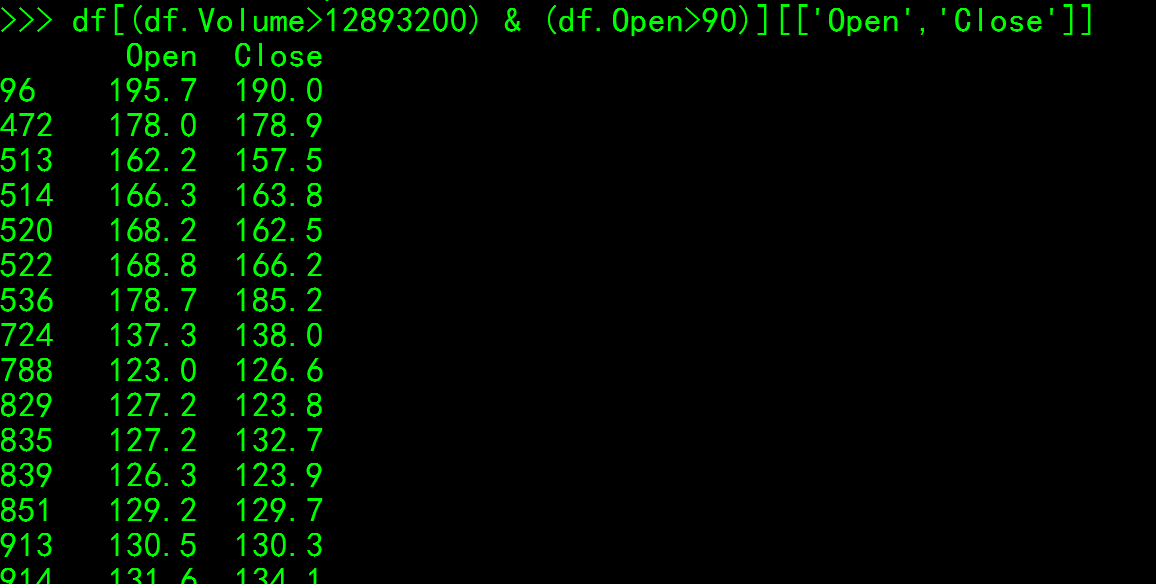
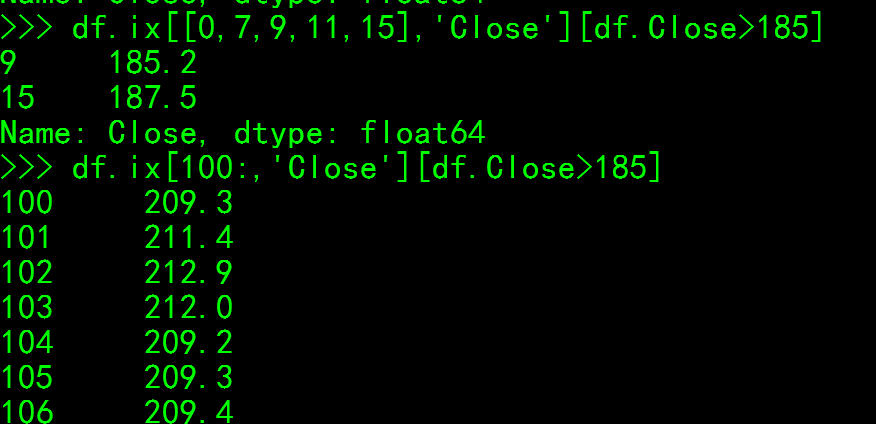
0 Beijing 2100 2016

2 Guangzhou 1000 2016

### 多列过滤&（且）或者|（或）

df6[(df6.Volume>12893200) & (df6.Open>90)]

df6[(df6.Volume>12893200) & (df6.Open>90)][['Open','Close']]

df6.ix[[0,7,9,11,15],'Close'][df6.Close>185]

### 数据透视

pd.pivot\_table(df1,columns=['city'],aggfunc=np.sum)

city Beijing Chongqi Guangzhou Hangzhou Shanghai Shenzhen

population 2100 500 1000 500 2300 700

year 2016 2016 2016 2016 2017 2017

### 按列进行排序

df6.sort\_values(by='Close',ascending=False).head()

## 广播特性Broadcasting

df4=pd.DataFrame(population)

df4['bonus']=6666

df4#这里的6666为标量，要应用到整个series，，就会使用到广播

df4

city population year bonus

0 Beijing 2100 2016 6666

1 Shanghai 2300 2017 6666

2 Guangzhou 1000 2016 6666

3 Shenzhen 700 2017 6666

4 Hangzhou 500 2016 6666

5 Chongqi 500 2016 6666

### 并行进行的广播，非循环

df4['average']=df4['bonus']/df4['population']

df4

city population year bonus average

0 Beijing 2100 2016 6666 3.174286

1 Shanghai 2300 2017 6666 2.898261

2 Guangzhou 1000 2016 6666 6.666000

3 Shenzhen 700 2017 6666 9.522857

4 Hangzhou 500 2016 6666 13.332000

5 Chongqi 500 2016 6666 13.332000

## Pandas高级应用

### Group

#### 数据准备

#导入相关库

import pandas as pd

import numpy as np

#建立实验数据

salaries = pd.DataFrame({

'name':['BOSS', 'Lilei', 'Lilei', 'Han', 'BOSS', 'BOSS', 'Han', 'BOSS'],

'Year':[2016,2016,2016,2016,2017,2017,2017,2017],

'Salary':[999999,20000,25000,3000,9999999,999999,3500,999999],

'Bonus':[100000,20000,20000,5000,200000,300000,3000,400000]

})



#简单描述,看到索引，列，数据类型等DF结构，以及是否有空值等

salaries.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 8 entries, 0 to 7

Data columns (total 4 columns):

Bonus 8 non-null int64

Salary 8 non-null int64

Year 8 non-null int64

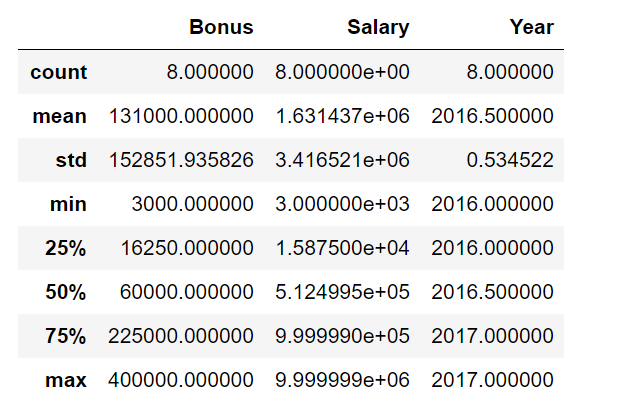
name 8 non-null object

dtypes: int64(3), object(1)

memory usage: 336.0+ bytes

#主要对数据进行计数，看数据列的均值，离散程度，最大最小，以4分位数

salaries.describe()



salaries = salaries[['name','Year', 'Salary','Bonus']]#重排列

#### 按照单列--名字列来做个分组

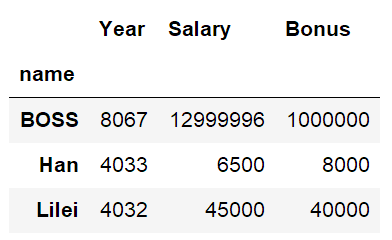
group\_by\_name = salaries.groupby('name')

type(group\_by\_name)

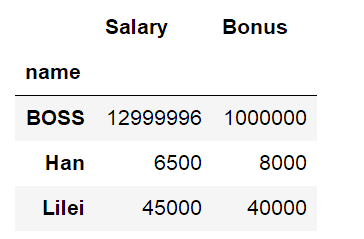
pandas.core.**groupby**.**DataFrameGroupBy**

#### 分组后运算#如sum, size，mean，median等

group\_by\_name.sum()

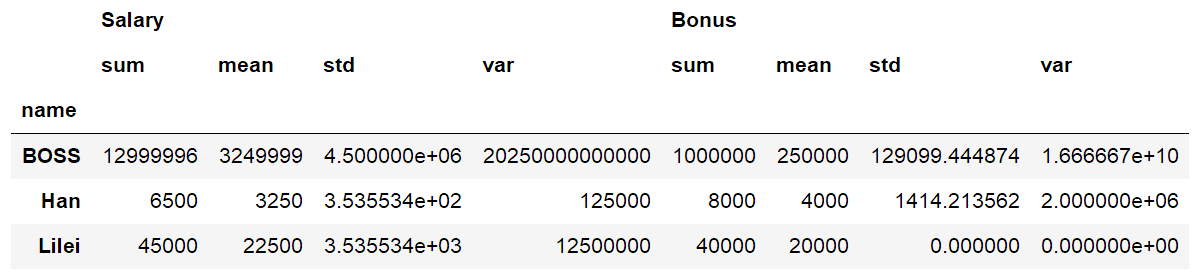


group\_by\_name[['Salary','Bonus']].sum()



#### agg-并行执行多种运算

group\_by\_name[['Salary','Bonus']].agg(['sum','mean','std','var'])



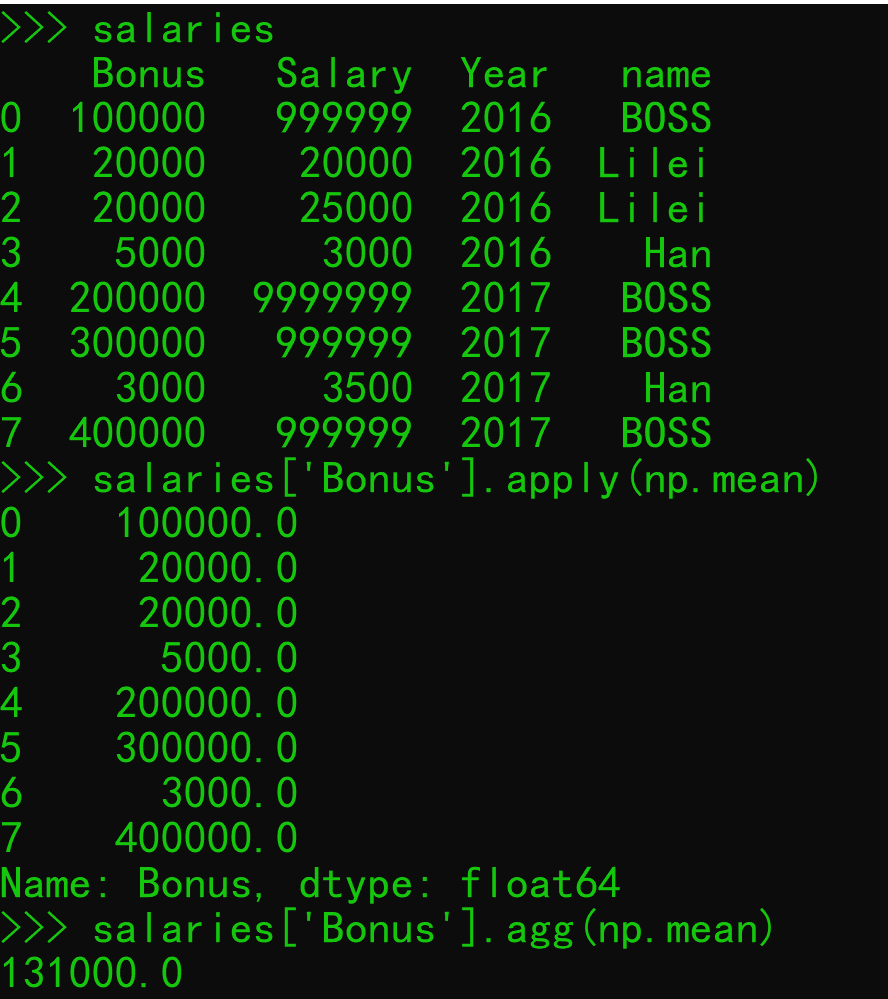
#### 也可以针对不同的列使用不同的函数，参数为字典形式

group\_by\_name[['Salary']].agg({'Salary':'mean', 'bonus':'std'})

agg与apply的没

agg将一个函数使用在一个数列上，然后返回一个标量的值。也就是说agg每次传入的是一列数据，对其聚合后返回标量。

apply将一个数据分拆-应用-汇总。会将当前分组后的数据一起传入，可以返回多维数据。



#### Group对象的构成

group\_by\_name.groups

{'BOSS': Int64Index([0, 4, 5, 7], dtype='int64'),

'Han': Int64Index([3, 6], dtype='int64'),

'Lilei': Int64Index([1, 2], dtype='int64')}

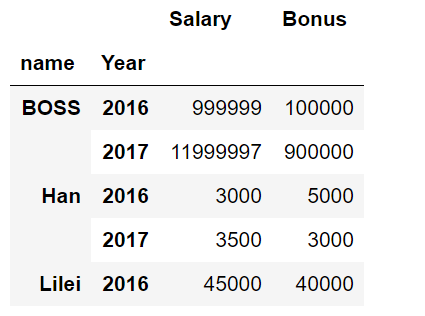
print(len(group\_by\_name.groups))

3

#### 按多列进行分组

group\_by\_name\_year = salaries.groupby(['name','Year'])

group\_by\_name\_year.sum()



#### 用迭代的方法查看分组详情

for name, group in group\_by\_name:

print(name)

print(group)

BOSS

name Year Salary Bonus

0 BOSS 2016 999999 100000

4 BOSS 2017 9999999 200000

5 BOSS 2017 999999 300000

7 BOSS 2017 999999 400000

Han

name Year Salary Bonus

3 Han 2016 3000 5000

6 Han 2017 3500 3000

Lilei

name Year Salary Bonus

1 Lilei 2016 20000 20000

2 Lilei 2016 25000 20000

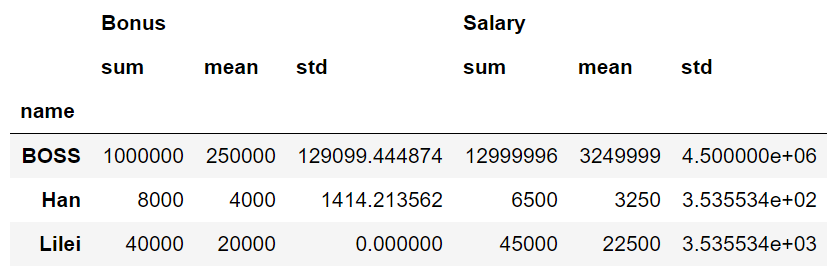
#### 选择不同的Group

group\_by\_name.get\_group("Lilei")

group\_by\_name.get\_group("BOSS")

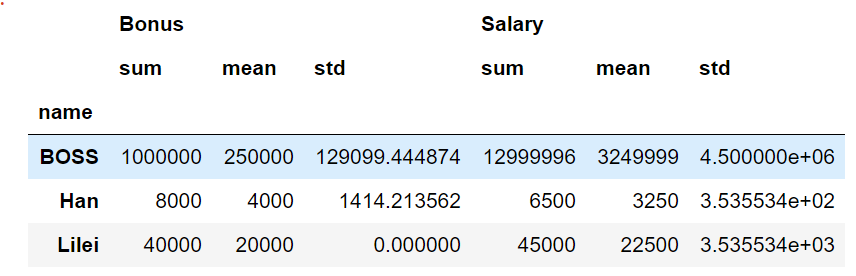
#### 同时在分组里做几个agg统计操作

group\_by\_name[['Bonus','Salary']].agg(['sum','mean','std'])



#### 使用numpy下的运算函数

group\_by\_name[['Bonus','Salary']].agg([np.sum, np.mean, np.std]



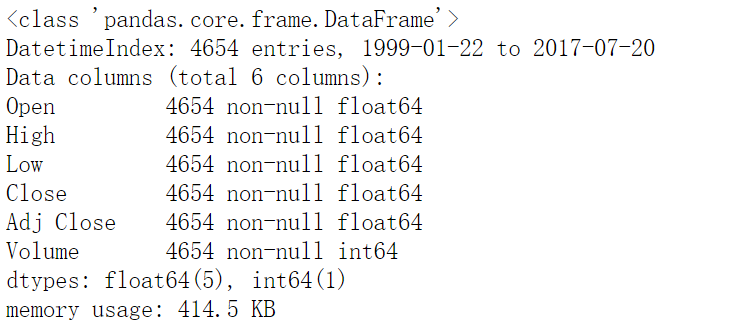
### Transform

#### 数据准备

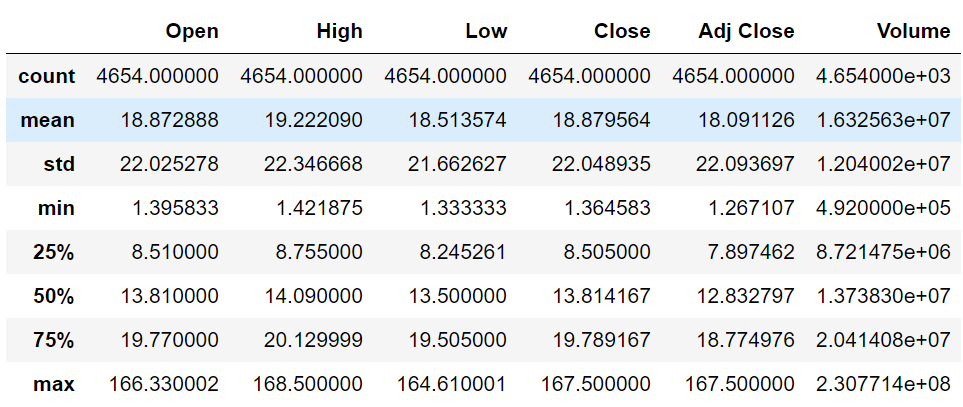
#使用Nvidia的股价数据

nvda = pd.read\_csv('d:/NVDA.csv', index\_col=0, parse\_dates=['Date'])

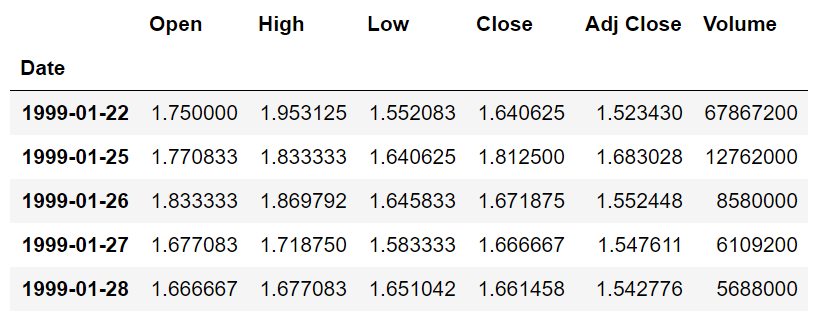
nvda.info()



nvda.describe()



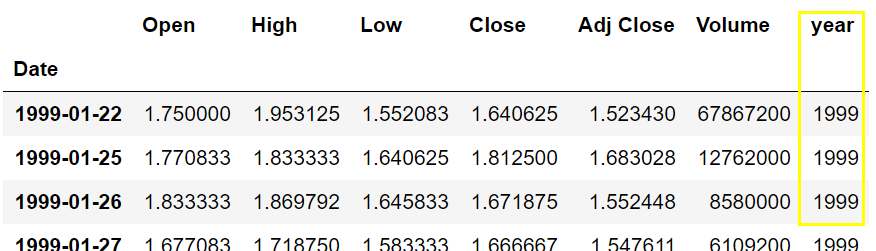
nvda.head()



#### 根据已有列构造新列

nvda['year'] = nvda.index.year

nvda.head()



#### 删除列

#注意inplace参数为True，代表的是原地删除。

nvda.drop(['year'], axis=1, inplace=True)

#### 索引列进行处理后排序

nvda.index.year#返回日期型索引的年

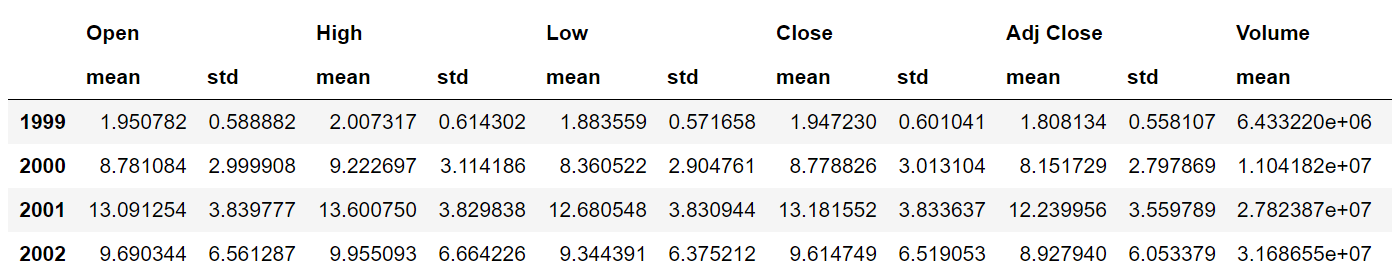
nvda.index.month#返回日期型索引的月

nvda.index.day#返回日期型索引的日

#如果我们想按年来先进行group，再进行统计，可以这样来进行

key = lambda x : x.year

nvda.groupby(key).agg(['mean','std']).head()

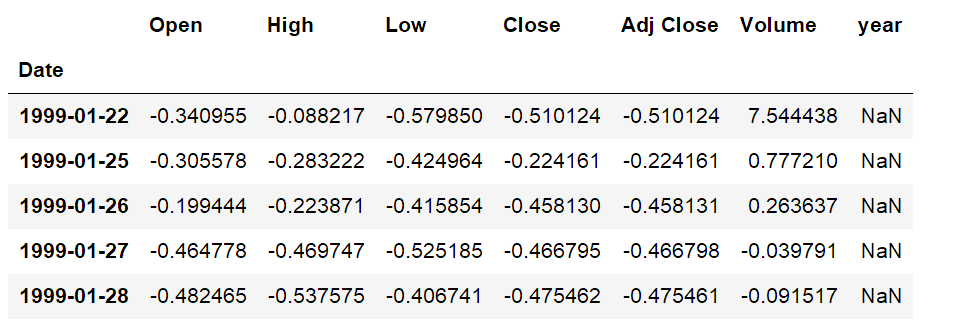


#### Transform变换

与apply,agg不同，变换函数transform针对的是整列数据,常用于求一列数据进行标准化

transform=nvda.groupby(key).transform(lambda x : (x-x.mean())/x.std()).head()

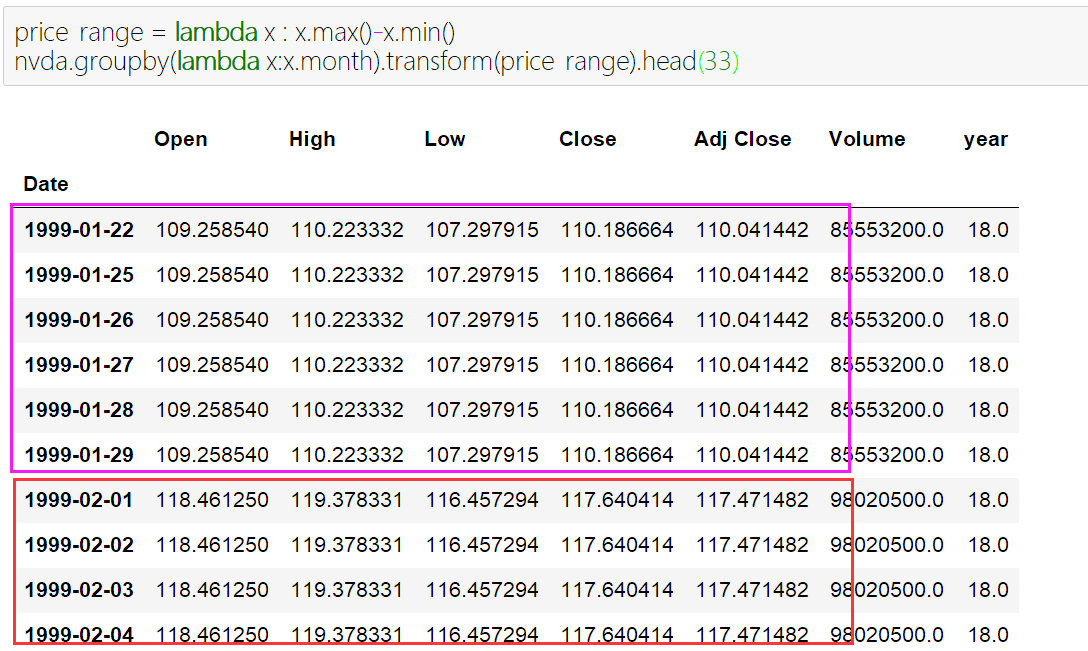
transform



#### #按月取得当月内价格波动范围

price\_range = lambda x : x.max()-x.min()

nvda.groupby(lambda x:x.month).transform(price\_range).head()



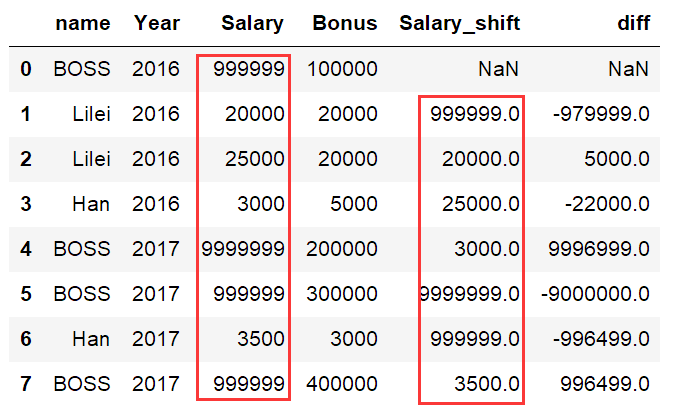
#### 尽量把pandas的运算变成列与列之间的运算

#使用**shift函数**对列数据进行上下平移后产生新的列，再与原列进行运算。

salaries['Salary\_shift'] = salaries['Salary'].shift(1)

salaries['diff'] = salaries['Salary'] - salaries['Salary\_shift']

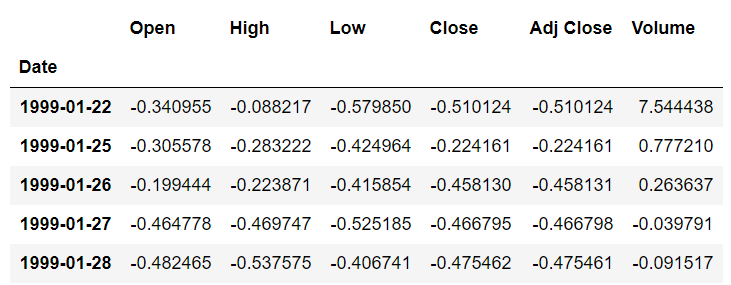
salaries



#### apply

对每列指定操作

nvda.groupby(key).apply(lambda x : x).head()



### 拼接concat

#### 数据准备

df1 = pd.DataFrame({'apts':[55000, 60000, 80000],

'cars':[200000, 300000, 400000]})

df2 = pd.DataFrame({'apts':[35000, 30000, 50000],

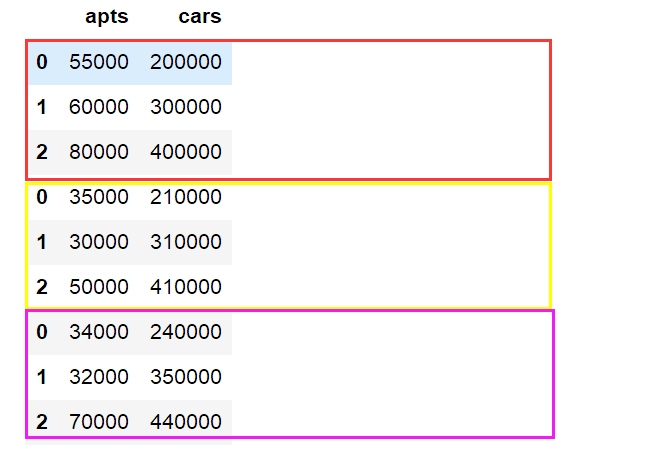
'cars':[210000,310000,410000]})

df3 = pd.DataFrame({'apts':[34000, 32000, 70000],

'cars':[240000,350000,440000]})

#### 纵向拼接

pd.concat([df1, df2, df3])



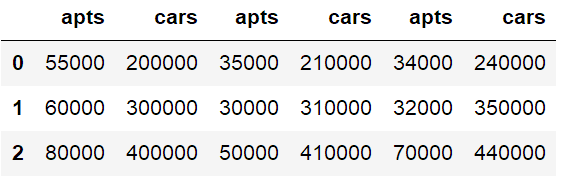
append一样可以完成

df1.append(df2)

df1.append([df1,df2])

#### 横向拼接

pd.concat([df1, df2, df3],axis=1)



### Merge(join)表关联

#### 数据准备

df4 = pd.DataFrame({'apts':[55000, 60000, 80000],

'cars':[200000, 300000, 400000]},

index = ['Beijing','Shanghai','Nanjing'])

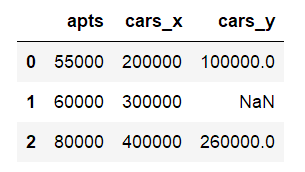
df5 = pd.DataFrame({'apts':[55000, 70000, 80000],

'cars':[100000, 330000, 260000]},

index = ['Beijing','Shanghai','Nanjing'])

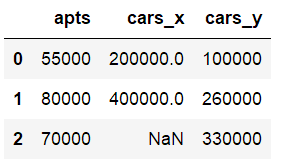
#### 左连

pd.merge(df4, df5, on=['apts'], how='left')



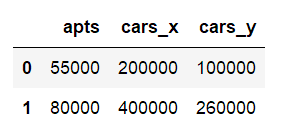
#### 右连

pd.merge(df4, df5, on=['apts'], how='right')



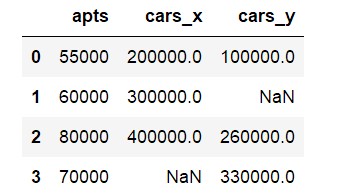
#### 内连

pd.merge(df4, df5, on=['apts'], how='inner')



#### 外联

pd.merge(df4, df5, on=['apts'], how='outer')



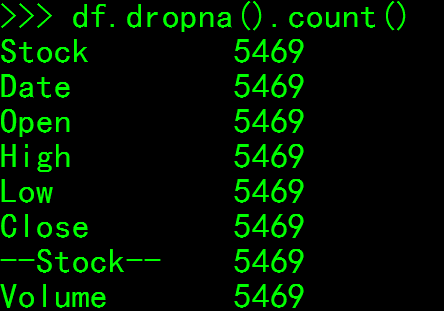
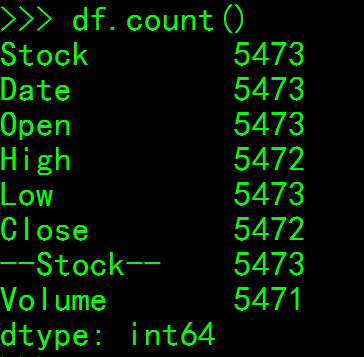
## Pandas数据预处理

### 空值与非空值

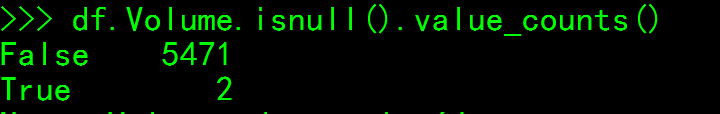
import numpy as np

#### 统计DF里所有非空值。

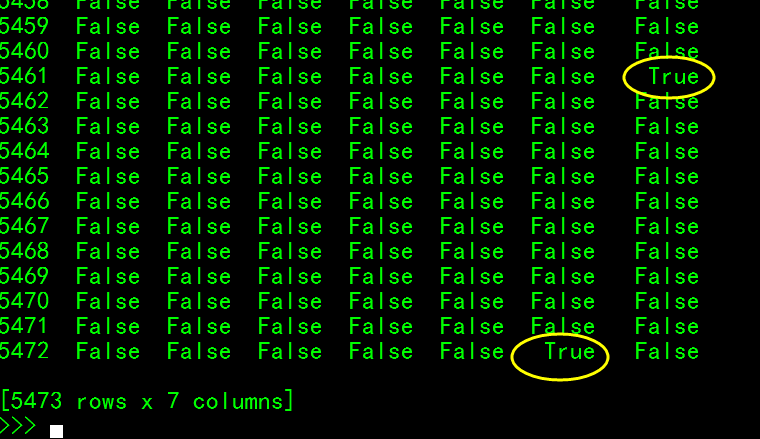
df.count()



#### np.nan



#### df.isnull()



#### dropna删除空行

df.dropna(how='any') #会去掉缺失值的行

#### fillna填充空值

不想滤除缺失的数据，而是通过其他方式填补“空洞”，fillna是最主要的函数。

df1['Age']=df1['Age'].fillna(df1['Age'].mean())

df.fillna(value=) #对缺失值进行填充

### 缺省值与缺少值填充

缺少值的类型是numpy中的nan

df4.iloc[1,2]=np.nan

city population year

0 Beijing 2100 2016.0

1 Shanghai 2300 NaN

2 Guangzhou 1000 2016.0

3 Shenzhen 700 2017.0

4 Hangzhou 500 2016.0

5 Chongqi 500 2016.0

df4.fillna(2017)#注意此时是返回一个修过的df，而原来的df不会被修改，如果要修改原来的df,指定inplace=True

city population year

0 Beijing 2100 2016.0

1 Shanghai 2300 2017.0

2 Guangzhou 1000 2016.0

3 Shenzhen 700 2017.0

4 Hangzhou 500 2016.0

5 Chongqi 500 2016.0

#### bfill与bfill

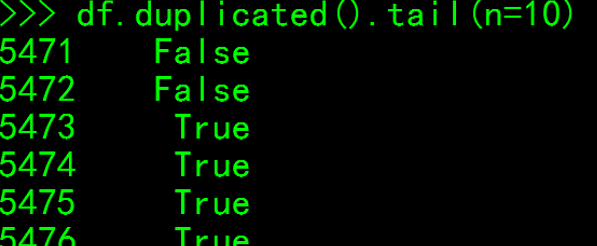
如果是像天气，股票价格的取样周期很短，如每半分钟一次的的数据，缺少值就适合用这种方法填充。但同样的场景，每半小时取样，就不适用这种方法了。先分析场景很重要。

df4.fillna(method='ffill') #按空值前一值填充

df4.fillna(method='bfill') #按空值后一值填充

### 去重

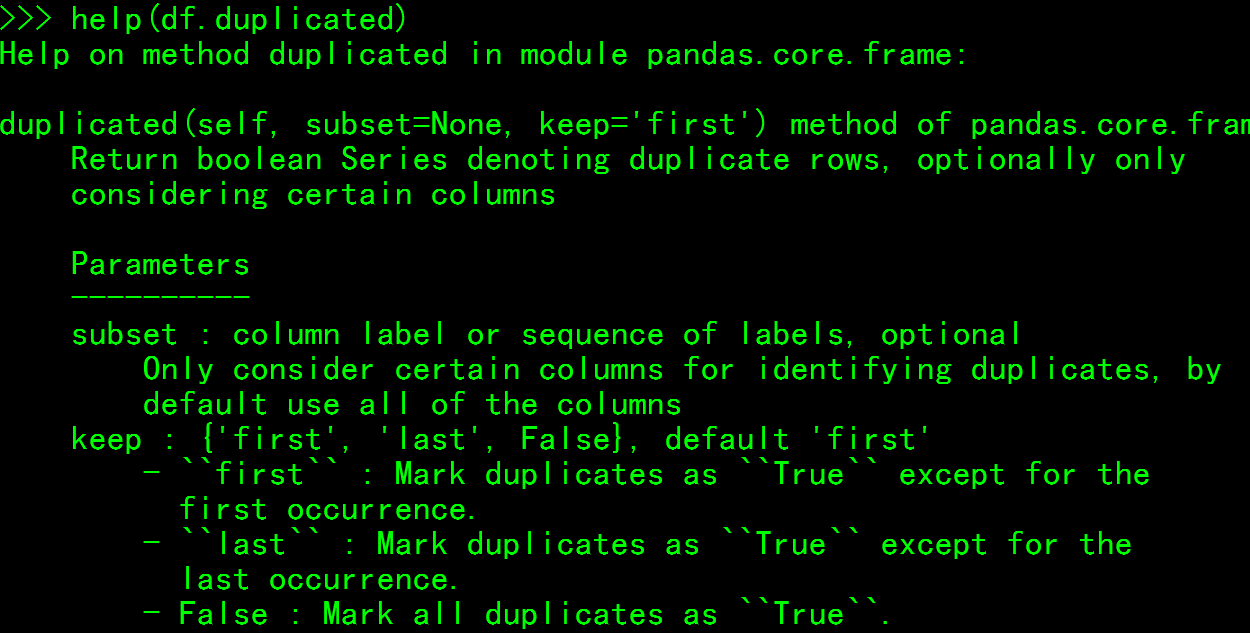
数据表中两个条目间所有列的内容都相等时duplicated才会判断为重复值。(Duplicated也可以单独对某一列进行重复值判断)。第二，duplicated支持从前向后(first)，和从后向前(last)两种重复值查找模式。默认是从前向后进行重复值的查找和判断



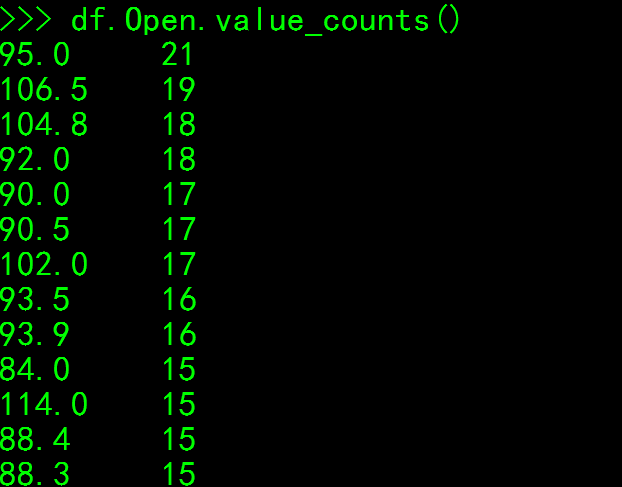
### drop\_duplicates用来删除数据表中的重复值，

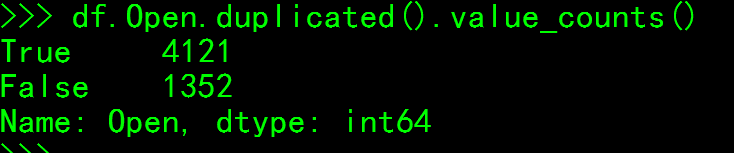
df.duplicated(subset='Stock')#指定列进行重复性检查

df.duplicated(subset='')



value\_counts()

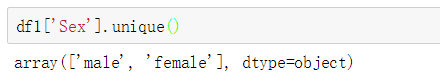


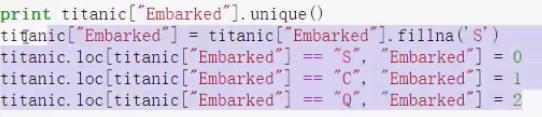


### 字符处理

#### 字符转换为数字

df1['Sex'].unique()





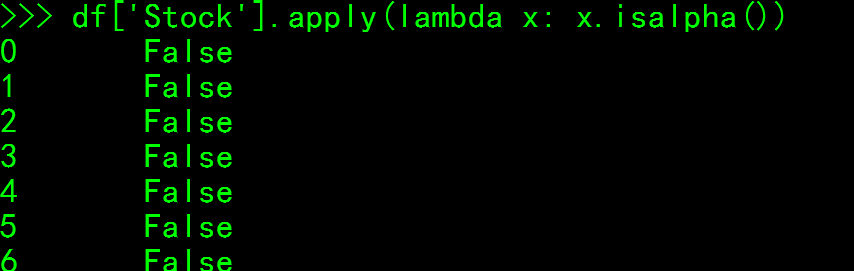
#### 空格，大小写，首写母大写title

df['Stock']=df['Stock'].map(str.strip)

df['Stock']=df['Stock'].map (str.lower)

df['Stock']=df['Stock'].map (str.title)

#### 检查列是否是数字，字符



### 数据分组标签

### 分组Categoricals

df = pd.DataFrame({"id":[1,2,3,4,5,6], "raw\_grade":['a', 'b', 'b', 'a', 'a', 'e']})

df["grade"] = df["raw\_grade"].astype("category")

cats=[0,50,80,100,150,200,300]

title=['A','B','C','D','E','F']

pd.cut(df['Close'],cats,labels=title)