# Pandas数据处理

### Pandas中的数据结构

Series：一维数组，与Numpy中的一维array类似。二者与Python基本的数据结构List也很相近，其区别是：List中的元素可以是不同的数据类型，而Array和Series中则只允许存储相同的数据类型，这样可以更有效的使用内存，提高运算效率。

Time- Series：以时间为索引的Series。

DataFrame：二维的表格型数据结构。很多功能与R中的data.frame类似。可以将DataFrame理解为Series的容器。以下的内容主要以DataFrame为主。

Panel ：三维的数组，可以理解为DataFrame的容器。

首先引入Pandas

import pandas as pd

官方推荐的缩写形式为pd，你可以选择其他任意的名称。 DataFrame是二维的数据结构，其本质是Series的容器，因此，DataFrame可以包含一个索引以及与这些索引联合在一起的Series，由于一个Series中的数据类型是相同的，而不同Series的数据结构可以不同。因此对于DataFrame来说，每一列的数据结构都是相同的，而不同的列之间则可以是不同的数据结构。或者以数据库进行类比，DataFrame中的每一行是一个记录，名称为Index的一个元素，而每一列则为一个字段，是这个记录的一个属性。

## Series

**Series默认由Index和values组成。构建一个Series：**

Series1=pd.Series(np.random.randn(4))

print(Series1,type(Series1))

0 -0.410217

1 0.063657

2 -0.814011

3 0.192189

dtype: float64 <class 'pandas.core.series.Series'>

print Series1.index

RangeIndex(start=0, stop=4, step=1)

print(Series1.values)

[-0.41021744 0.06365672 -0.81401131 0.19218863]

**Series支持过滤：**

print(Series1>0)

0 False

1 True

2 False

3 True

dtype: bool

print(Series1[Series1>0])

1 0.063657

3 0.192189

dtype: float64

**Series支持Broadcasting(广播)：**

print(Series1\*2)

0 -0.820435

1 0.127313

2 -1.628023

3 0.384377

dtype: float64

print(Series1+5)

0 4.589783

1 5.063657

2 4.185989

3 5.192189

dtype: float64

**支持通用函数：**

print(round(Series1))

0 0.0

1 0.0

2 -1.0

3 0.0

dtype: float64

f\_np=np.frompyfunc(lambda x:round(x)+3,1,1)

print(f\_np(Series1))

0 3

1 3

2 2

3 3

dtype: object

**指定Index：**

Series2=pd.Series(Series1.values,index=['norm\_'+str(i) for i in range(4)])

print(Series2)

norm\_0 -0.410217

norm\_1 0.063657

norm\_2 -0.814011

norm\_3 0.192189

dtype: float64

print(Series2.index)

Index(['norm\_0', 'norm\_1', 'norm\_2', 'norm\_3'], dtype='object')

**通过索引访问数据：**

print(Series2[['norm\_1','norm\_3']])

norm\_1 0.063657

norm\_3 0.192189

dtype: float64

print('norm\_1' in Series2)

True

**从字典创建Series：**

Series3\_Dict={'Japan':'Tokyo','S.Korea':'Seoul','China':'Beijing'}

Series3=pd.Series(Series3\_Dict)

**为Series指定Index和数据列名：**

print(Series3.index.name)

None

print(Series3.name)

None

Series3.index.name='Nation'

Series3.name='Capital'

print(Series3)

Nation

China Beijing

Japan Tokyo

S.Korea Seoul

Name: Capital, dtype: object

## Dataframe

**创建Dataframe**

**从Numpy二维数组创建，别忘了定义列名：**

dataNumpy=np.asarray([('Japan','Tokyo',4000),('S.korea','Seoul',1300),('China','Beijing',9000)])

df1=pd.DataFrame(dataNumpy,columns=['nation','capital','GDP'])

df1

nation capital GDP

0 Japan Tokyo 4000

1 S.korea Seoul 1300

2 China Beijing 9000

**从字典定义，定义的Dataframe的列是无序的：**

Dict={'nation':['Japan','S.korea','China'],'capital':['Tokyo','Seoul','Beijing'],'GDP':[4900,1300,9000]}

df2=pd.DataFrame(Dict)

df2

GDP capital nation

0 4900 Tokyo Japan

1 1300 Seoul S.korea

2 9000 Beijing China

**从一个Dataframe定义一个Dataframe，使用这种方法可以调整列的顺序和索引顺序：**

df21=pd.DataFrame(df2,columns=['nation','capital','GDP'])

df21

nation capital GDP

0 Japan Tokyo 4900

1 S.korea Seoul 1300

2 China Beijing 9000

df22=pd.DataFrame(df2,columns=['nation','capital','GDP'],index=[2,0,1])

df22

nation capital GDP

2 China Beijing 9000

0 Japan Tokyo 4900

1 S.korea Seoul 1300

**从Dataframe中取出列，两种方法：**

df22.nation

2 China

0 Japan

1 S.korea

Name: nation, dtype: object

df22['GDP']

2 9000

0 4900

1 1300

Name: GDP, dtype: int64

**从Dataframe中取出行，四种方法：**

**df22[0:2] #输出实际的行**

nation capital GDP

2 China Beijing 9000

0 Japan Tokyo 4900

**ix：输入的参数可以是行索引，也可以是行标签，输出索引或标签对应的行。**

data = [[1,2,3],[4,5,6]]

index = ['d','e']

columns=['a','b','c']

df = pd.DataFrame(data=data, index=index, columns=columns)

df

a b c

d 1 2 3

e 4 5 6

df.ix['e']

a 4

b 5

c 6

Name: e, dtype: int64

df.ix[1] # df.ix['e']等同于df.ix[1]

a 4

b 5

c 6

Name: e, dtype: int64

**iloc:输入的参数是表示行或列的索引，输出索引所在的行或列，同时可对列进行选取，类似切片的效果。**

df22.iloc[0,:]

nation China

capital Beijing

GDP 9000

Name: 2, dtype: object

df22.iloc[:,1:3]

capital GDP

2 Beijing 9000

0 Tokyo 4900

1 Seoul 1300

**Loc：输入的是行或列的标签，输入标签所在的行或列。和iloc一样可以同时对行和对列进行选取。**

df.loc[['d','e']]

a b c

d 1 2 3

e 4 5 6

df.loc['d',['b','c']]

b 2

c 3

Name: d, dtype: int64

**增加列：**

df22['population']=[1600,130,33]

df22['region']='East\_Asian'

df22

nation capital GDP population region

2 China Beijing 9000 1600 East\_Asian

0 Japan Tokyo 4900 130 East\_Asian

1 S.korea Seoul 1300 33 East\_Asian

**增加行：**

**使用append方法，会改变索引和列的顺序：**

new=pd.DataFrame({'nation':'UA','capital':'HT','GDP':4000,'population':50,'region':'Africa'},index=[3])

new

GDP capital nation population region

3 4000 HT UA 50 Africa

df22.append(new,ignore\_index=True)

GDP capital nation population region

0 9000 Beijing China 1600 East\_Asian

1 4900 Tokyo Japan 130 East\_Asian

2 1300 Seoul S.korea 33 East\_Asian

3 4000 HT UA 50 Africa

**使用loc：**

new.loc[‘a’]=['1000','Beijing','China',60,'Asia']

new

GDP capital nation population region

3 4000 HT UA 50 Africa

a 1000 Beijing China 60 Asia

**删除列：**

df22.drop(['GDP','region'],axis=1)

nation capital population

2 China Beijing 1600

0 Japan Tokyo 130

1 S.korea Seoul 33

**删除行，根据索引的名称进行删除：**

df22.drop([1,2],axis=0)

nation capital GDP population region

0 Japan Tokyo 4900 130 East\_Asian

**修改数据：**

df22.iloc[1:3,2]=[2000,1000]

df22

df22

nation capital GDP population region

2 China Beijing 9000 1600 East\_Asian

0 Japan Tokyo 2000 130 East\_Asian

1 S.korea Seoul 1000 33 East\_Asian

## 数据读写

### 结构化数据输入输出

read\_csv和to\_csv

read\_table

read\_fwf:操作fixed width file

read\_excel和to\_excel

## 深入pandas数据操纵

### 数据整合：

**横向拼接（行拼接）。直接Dataframe:**

pd.DataFrame([np.random.rand(2),np.random.rand(2)],columns=['c1','c2'])

c1 c2

0 0.536172 0.799560

1 0.407922 0.749475

**使用Concatenate：**

pd.concat([df2,df2]) **#保留原始index**

GDP capital nation

0 4900 Tokyo Japan

1 1300 Seoul S.korea

2 9000 Beijing China

0 4900 Tokyo Japan

1 1300 Seoul S.korea

2 9000 Beijing China

pd.concat([df2,df2],ignore\_index=True) **#不保留原始index**

GDP capital nation

0 4900 Tokyo Japan

1 1300 Seoul S.korea

2 9000 Beijing China

3 4900 Tokyo Japan

4 1300 Seoul S.korea

5 9000 Beijing China

**纵向拼接（列拼接）：使用merge。根据列关联，使用on关键字(如果有相同的列名，则on省略的话，按照相同的列名的列进行拼接):**

pd.merge(df2,df2,on='nation')

GDP\_x capital\_x nation GDP\_y capital\_y

0 4900 Tokyo Japan 4900 Tokyo

1 1300 Seoul S.korea 1300 Seoul

2 9000 Beijing China 9000 Beijing

pd.merge(df2,df2,on=['nation','capital'])

GDP\_x capital nation GDP\_y

0 4900 Tokyo Japan 4900

1 1300 Seoul S.korea 1300

2 9000 Beijing China 9000

**根据index拼接：**

**增加how关键字，给定链接类型，how可以为：‘inner’，‘left’，‘right’，‘outer’：**

df3=pd.DataFrame({'name':[u'老王',u'老张',u'老李'],'sal':[5000,3000,1000]})

df4=pd.DataFrame({'name':[u'老王',u'老刘'],'job':['VP','Manager']})

df3

name sal

0 老王 5000

1 老张 3000

2 老李 1000

df4

job name

0 VP 老王

1 Manager 老刘

pd.merge(df3,df4,on='name',how='left')

name sal job

0 老王 5000 VP

1 老张 3000 NaN

2 老李 1000 NaN

pd.merge(df3,df4,on='name',how='inner')

name sal job

0 老王 5000 VP

pd.merge(df3,df4,on='name',how='outer')

name sal job

0 老王 5000.0 VP

1 老张 3000.0 NaN

2 老李 1000.0 NaN

3 老刘 NaN Manager

### 数据清洗

**数据清洗的三个方法：map，applymap和apply。**

**map：根据一列生成新的一列数据，对一列操作。Series>Series。**

**applymap：根据整张表生成新的一张表，对表操作。Dataframe>Dataframe。**

**apply：根据多列生成一列，或根据多行生成一行，对表操作。Dataframe>Series。**dataNumpy=np.asarray([('Japan','Tokyo',4000),('S.korea','Seoul',1300),('China','Beijing',9000)])

df1=pd.DataFrame(dataNumpy,columns=['nation','capital','GDP'])

df1['NATION']=df1.nation.map(str.upper)

df1

nation capital GDP NATION

0 Japan Tokyo 4000 JAPAN

1 S.korea Seoul 1300 S.KOREA

2 China Beijing 9000 CHINA

**map也可以使用自定义函数：**

def GDP\_Level(v): #GDP\_Level是自定义函数

v=np.float64(v)

if v>6000:

return 'High'

elif v<2000:

return 'Low'

else:

return 'Medium'

df1['GDP\_level']=df1['GDP'].map(GDP\_Level)

df1

nation capital GDP NATION GDP\_level

0 Japan Tokyo 4000 JAPAN Medium

1 S.korea Seoul 1300 S.KOREA Low

2 China Beijing 9000 CHINA High

df1.applymap(lambda x:float(x)\*2 if x.isdigit() else x.upper())

nation capital GDP NATION GDP\_level

0 JAPAN TOKYO 8000.0 JAPAN MEDIUM

1 S.KOREA SEOUL 2600.0 S.KOREA LOW

2 CHINA BEIJING 18000.0 CHINA HIGH

df1.apply(lambda x:x['nation']+x['capital']+'\_'+x['GDP'],axis=1) #axis=1表示对行进行处理。

0 JapanTokyo\_4000

1 S.koreaSeoul\_1300

2 ChinaBeijing\_9000

dtype: object

### 数据排序

**sort：按一列或多列进行行排序。**

**sort\_index:根据index里的值进行排序，可根据参数axis设置对行或列排序。**

df1.sort(['capital','nation'])

<string>:1: FutureWarning: sort(columns=....) is deprecated, use sort\_values(by=.....)

nation capital GDP NATION GDP\_level

2 China Beijing 9000 CHINA High

1 S.korea Seoul 1300 S.KOREA Low

0 Japan Tokyo 4000 JAPAN Medium

df1.sort('GDP',ascending=False)

nation capital GDP NATION GDP\_level

2 China Beijing 9000 CHINA High

0 Japan Tokyo 4000 JAPAN Medium

1 S.korea Seoul 1300 S.KOREA Low

df1.sort('GDP').sort(ascending=False) # sort(ascending=False)是对index进行降序排列

<string>:1: FutureWarning: sort(....) is deprecated, use sort\_index(.....)

nation capital GDP NATION GDP\_level

2 China Beijing 9000 CHINA High

1 S.korea Seoul 1300 S.KOREA Low

0 Japan Tokyo 4000 JAPAN Medium

df1.sort\_index(axis=1) #将列进行排序

GDP GDP\_level NATION capital nation

0 4000 Medium JAPAN Tokyo Japan

1 1300 Low S.KOREA Seoul S.korea

2 9000 High CHINA Beijing China

**rank的使用：**

df1.sort\_index(axis=1)

GDP GDP\_level NATION capital nation

0 4000 Medium JAPAN Tokyo Japan

1 1300 Low S.KOREA Seoul S.korea

2 9000 High CHINA Beijing China

df1.rank(ascending=False)

nation capital GDP NATION GDP\_level

0 2.0 1.0 2.0 2.0 1.0

1 1.0 2.0 3.0 1.0 2.0

2 3.0 3.0 1.0 3.0 3.0

**rank的method参数(默认average):**

**method=’average’：两条数据相等时，排序时用平均值。**

**method=’min’： 两条数据相等时，排序时用最小值。**

**method=’max’： 两条数据相等时，排序时用最大值。**

**method=’first’： 两条数据相等时，排序时谁先出现就先给谁较小值。**

df3=pd.DataFrame({'name':[u'老王',u'老张',u'老李',u'老刘'],'sal':np.array([5000,3000,5000,9000])})

df3

name sal

0 老王 5000

1 老张 3000

2 老李 5000

3 老刘 9000

df3.rank()

name sal

0 4.0 2.5

1 2.0 1.0

2 3.0 2.5

3 1.0 4.0

df3.sal.rank(method='min')

0 2.0

1 1.0

2 2.0

3 4.0

Name: sal, dtype: float64

df3.sal.rank(method='first')

0 2.0

1 1.0

2 3.0

3 4.0

Name: sal, dtype: float64

### 缺失数据处理

**Pandas使用浮点值NaN(Not a Number)表示缺失数据。numpy.nan可以定义确实值，Python中内置的None也被当作NA处理:**

string\_data=pd.Series(['abcd','efg',np.nan,None])

string\_data

0 abcd

1 efg

2 NaN

3 None

dtype: object

string\_data.isnull()

0 False

1 False

2 True

3 True

dtype: bool

**NA处理方法：**

**dropna：对一行或者一列进行过滤。**

**fillna：用指定值或插值方法(ffill或bfill)填充缺失数据。**

**isnull：返回含有布尔值的对象，表示哪些是/不是确实值。**

**notnull：isnull的否定式。**

**滤除缺失数据：**

对于Series：

data=pd.Series([1,np.nan,3.5,np.nan,7])

data

0 1.0

1 NaN

2 3.5

3 NaN

4 7.0

dtype: float64

data.dropna()

0 1.0

2 3.5

4 7.0

dtype: float64

data[data.notnull()]

0 1.0

2 3.5

4 7.0

dtype: float64

**对于Dataframe：**

**dropna默认丢弃任何含有确实值的行：**

from numpy import nan as NA

data=pd.DataFrame([[1,5,4,2],[1,NA,NA],[NA,NA,NA],[NA,5.5,2]])

data

0 1 2 3

0 1.0 5.0 4.0 2.0

1 1.0 NaN NaN NaN

2 NaN NaN NaN NaN

3 NaN 5.5 2.0 NaN

data.dropna()

0 1 2 3

1. 1.0 5.0 4.0 2.0

**how=’all’将只丢弃全为NA的行：**

data.dropna(how='all')

0 1 2 3

0 1.0 5.0 4.0 2.0

1 1.0 NaN NaN NaN

3 NaN 5.5 2.0 NaN

**如若在列上进行操作，只需传入axis=1：**

data[4]=NA

data

0 1 2 3 4

0 1.0 5.0 4.0 2.0 NaN

1 1.0 NaN NaN NaN NaN

2 NaN NaN NaN NaN NaN

3 NaN 5.5 2.0 NaN NaN

data.dropna(axis=1,how='all')

0 1 2 3

0 1.0 5.0 4.0 2.0

1 1.0 NaN NaN NaN

2 NaN NaN NaN NaN

3 NaN 5.5 2.0 NaN