Pandas基础教程三\_DataFrame数据结构

大家好，这里是Python知识学堂，在之前的基础知识中有提及到Pandas中有两个重要的数据结构，Series和DataFrame。也使用一些篇幅着重介绍了Series的基础知识，希望跟着学习的小伙伴好好复习一下知识点，本次推文我们一起来学习一下DataFrame的相关知识。

跟Series不同DataFrame表示的一种矩阵的数据结构(数据表)，在DataFrame中不同列的数据类型(数值，字符串，布尔值等等)。另外从DataFrame的数据形式可以看出。DataFrame既有行索引，也有列索引，这种形式的存在使得DataFrame的取数变得更加的灵活。

**注意：虽然DataFrame是一个二维数据，但是使用分层索引方式可以展现更高维度的数据，后期将以实例的形式说明分层索引方法的使用。**

1. **创建DataFrame**

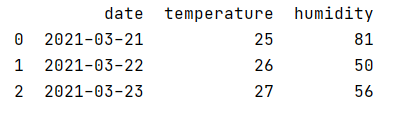
DataFrame的创建有很多种方式。在工程项目中，我们如果直接使用Pandas的方法pd.read\_csv(**'file.csv'**)和pd.read\_excel(**'file.xlsx'**)方法，这两个方法返回的数据就是DataFrame类型的数据，接下来我们来看看使用其他的方法如何进行DataFrame数据的创建。

使用字典

使用字典创建DataFrame是非常方便的，使用的方式如下：

import pandas as pd  
weather\_data = {  
 **'date'**:[**'2021-03-21'**,**'2021-03-22'**,**'2021-03-23'**],  
 **'temperature'**:[25,26,27],  
 **'humidity'**:[81, 50, 56]  
}  
weather\_df = pd.DataFrame(weather\_data)  
print(weather\_df)

输出结果如下：



date temperature humidity

0 2021-03-21 25 81

1 2021-03-22 26 50

2 2021-03-23 27 56

可以看出，使用字典的话是很方便创建DataFrame的，我们只需要给相应的方法传入一个字典就就可以，字典的数据我们使用列表。

使用元组列表

使用元组列表的方式很很方便，我们只需要传入一个列表，列表内使用元组。我们来看一下代码：

import pandas as pd  
weather\_data = [  
 (**'2021-03-21'**,25,81),  
 (**'2021-03-22'**,26,50),  
 (**'2021-03-23'**,27,56)  
]  
weather\_df = pd.DataFrame(data = weather\_data, columns=[**'date'**, **'temperature'**, **'humidity'**])  
print(weather\_df)

本次输出与使用字典创建的DataFrame一样，与上述不同的是，使用元组列表的时候，我们在使用pd.DataFrame()方法的时候需要传入参数columns以指定列名，columns列表的顺序也直接决定了生成的DataFrame列的顺序。

使用字典列表

跟使用元组列表一样，我们还可以使用字典列表进行DataFrame的创建：

import pandas as pd  
weather\_data = [  
 {**'date'**:**'2021-03-21'**,**'temperature'**:**'25'**,**'humidity'**:**'81'**},  
 {**'date'**:**'2021-03-22'**,**'temperature'**:**'26'**,**'humidity'**:**'50'**},  
 {**'date'**:**'2021-03-23'**,**'temperature'**:**'27'**,**'humidity'**:**'56'**}  
]  
weather\_df = pd.DataFrame(data = weather\_data,  
 columns=[**'date'**, **'temperature'**, **'humidity'**])  
print(weather\_df)

注意上述我们没有指定行索引，所以他们只有默认的所以0,1,和2.如果需要指定行索引的话，我们可以在加入index参数，如：

weather\_df = pd.DataFrame(data = weather\_data,  
 columns=[**'date'**, **'temperature'**, **'humidity'**],  
 index = [**'row\_index\_1'**,**'row\_index\_2'**,**'row\_index\_3'**]  
 )

还有一个要注意的就是，如果我们传入的列不在字典中，那么panda会默认加上一个缺失值(NaN的值)：

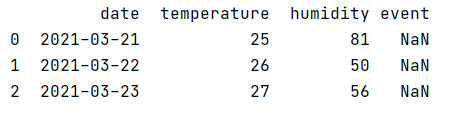
import pandas as pd  
weather\_data = {  
 **'date'**:[**'2021-03-21'**,**'2021-03-22'**,**'2021-03-23'**],  
 **'temperature'**:[25,26,27],  
 **'humidity'**:[81, 50, 56],  
}  
weather\_df = pd.DataFrame(weather\_data,columns=[**'date'**,**'temperature'**,**'humidity'**,**'event'**])  
print(weather\_df)

date temperature humidity event

0 2021-03-21 25 81 NaN

1 2021-03-22 26 50 NaN

2 2021-03-23 27 56 NaN



当然了，也可以使用列表列表的方式进行创建DataFrame。

**二、DataFrame子集操作**

在进行数据分析的时候，我们经常要对DataFrame的数据中每一列或者某一行进行一些诸如归一化等的操作，这个时候就要进行取数的操作，那么常见的一些子集的操作是怎样的呢？这里列举了一些方法：

|  |  |
| --- | --- |
| 子集操作方法 | 方法说明 |
| df.head() | 获取前5行数据 |
| df.tail() | 获取最后5行数据 |
| df[start:stop:end] | 使用切片方式获取数据 |
| df[column\_name] | 获取单列数据 |
| df[[column1,column2…]] | 获取多列数据 |
| df.loc[row\_label] | 使用行标签获取数据 |
| df.loc[[label1,label2,…]] | 使用行标签获取多行数据 |
| df.iloc[row] | 使用行号获取数据 |
| df.iloc[[row\_1,row\_2,…]] | 使用行号获取多行数据 |
| df.loc[[bool1,bool2,…]] | 使用bool值获取数据 |

我们使用实例的方式来说明这些操作实际的用法，假设我们的构建的DataFrame数据如下：

import pandas as pd  
data = [  
 [1,2,3,4,5],  
 [6,7,8,9,10],  
 [11,11,12,13,14],  
 [15,16,17,18,19],  
 [20,21,22,23,24],  
 [25,26,27,28,29],  
 [30,31,32,33,34],  
 [35,36,37,38,39],  
 [40,41,42,43,44],  
 [45,46,47,48,49]  
 ]  
index = [**'01'**, **'02'**, **'03'**, **'04'**, **'05'**,**'06'**, **'07'**, **'08'**, **'09'**, **'10'**]  
column=[**'C001'**, **'C002'**, **'C003'**,**'C004'**,**'C005'**]

df\_example = pd.DataFrame(data= data,index=index,columns=column)

我们先看head()和tail()方法：

print(df\_example.head())  
print(df\_example.tail())

将输出：

C001 C002 C003 C004 C005

01 1 2 3 4 5

02 6 7 8 9 10

03 11 11 12 13 14

04 15 16 17 18 19

05 20 21 22 23 24

C001 C002 C003 C004 C005

06 25 26 27 28 29

07 30 31 32 33 34

08 35 36 37 38 39

09 40 41 42 43 44

10 45 46 47 48 49

这里head()和tail()均可以传一个参数如：head(3)和tail(3)表示获取前三行和尾三行数据。切片方式获取数据跟列表的切片是一样的如：

print(df\_example[1:5:2])  
print(df\_example[1:5])

C001 C002 C003 C004 C005

02 6 7 8 9 10

04 15 16 17 18 19

C001 C002 C003 C004 C005

02 6 7 8 9 10

03 11 11 12 13 14

04 15 16 17 18 19

05 20 21 22 23 24

我们在来看一下获取单列数据和多列数据该怎样进行操作：  
print(df\_example[**'C001'**])  
print(df\_example[[**'C001'**,**'C002'**]])  
print(type(df\_example[**'C001'**]))  
print(type(df\_example[[**'C001'**]]))

01 1

02 6

03 11

04 15

05 20

06 25

07 30

08 35

09 40

10 45

Name: C001, dtype: int64

C001 C002

01 1 2

02 6 7

03 11 11

04 15 16

05 20 21

06 25 26

07 30 31

08 35 36

09 40 41

10 45 46

<class 'pandas.core.series.Series'>

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

这里我打印了df\_example[**'C001'**])和df\_example[[**'C001'**]])这两个数据的类型，发现一个是Series和DataFrame，大家注意这两个的不同，细细评味。

方法loc取数和iloc取数有一点相似性，在上篇关于Series的推文中也详细的介绍了这个方法，这里我就直接举例这两个方法的使用：

print(df\_example.loc[**'01'**])  
print(df\_example.loc[[**'01'**,**'02'**,**'03'**]])

输出：

C001 1

C002 2

C003 3

C004 4

C005 5

Name: 01, dtype: int64

C001 C002 C003 C004 C005

01 1 2 3 4 5

02 6 7 8 9 10

03 11 11 12 13 14

达到上述的输出形式我们也可以使用iloc方法：

print(df\_example.iloc[0])  
print(df\_example.iloc[[0,1,2]])

这两个输出的结果跟上述一样。

最后，我们来看一下bool值的使用：

print(df\_example.loc[[False,True,False,True,False,True,False,True,False,True]])

输出如下：

C001 C002 C003 C004 C005

02 6 7 8 9 10

04 15 16 17 18 19

06 25 26 27 28 29

08 35 36 37 38 39

10 45 46 47 48 49

除了一写子集的操作，在平时数据分析的时候，我们还可以对DataFrame进行转置的操作，即：

print(df\_example.T)

另外，我们可以使用Values属性里获取DataFrame中的数据：

print(df\_example.values)  
print(type(df\_example.values))

第一个输出为，即输出

[[ 1 2 3 4 5]

[ 6 7 8 9 10]

[11 11 12 13 14]

[15 16 17 18 19]

[20 21 22 23 24]

[25 26 27 28 29]

[30 31 32 33 34]

[35 36 37 38 39]

[40 41 42 43 44]

[45 46 47 48 49]]

对于这样的输出我们可以使用列表取数的方式进行子集数据的提取。

第二个输出为ndarray类型的数据，大家要做注意。

**三、DataFrame的描述性统计**

跟Series数据结构一样，Pandas中也有一些适用于DataFrame的适用于统计分析的属性和方法，我们来看一下，我们使用的DataFrame如下：

df = pd.DataFrame([[1.2,-2.5,0.3],[-4.2,1.5,np.nan],  
 [np.nan,np.nan,np.nan],[-0.5,2.3,0.6]],  
 index = [**'A'**,**'B'**,**'C'**,**'D'**],  
 columns= [**'num\_1'**,**'num\_2'**,**'num\_3'**])

在构建这个DataFrame的时候要导入numpy,即import numpy as np。

接下来我们列举几个常见的描述性统计方法：

|  |  |
| --- | --- |
| 描述性统计方法 | 方法说明 |
| Describe() | Series或DataFrame各列汇总统计集合 |
| sum()\mean()\median() | 求和\均值\中位数 |
| min()\max() | 最小值\最大值 |
| argmin()\argmax() | 最小值\最大值数据所在索引位置 |
| idxmin()\idxmax() | 最小值\最大值数据所在索引标签 |
| cummin()\cummax() | 累计值的最小\最大值 |
| count | 统计非nan值的数量 |
| cumsum | 累计值 |
| prod | 所有值的积 |
| var\std\skew\kurt | 方差\标准差\偏度\散度 |

print(df.describe())

num\_1 num\_2 num\_3

count 3.000000 3.000000 2.000000

mean -1.166667 0.433333 0.450000

std 2.761038 2.571640 0.212132

min -4.200000 -2.500000 0.300000

25% -2.350000 -0.500000 0.375000

50% -0.500000 1.500000 0.450000

75% 0.350000 1.900000 0.525000

max 1.200000 2.300000 0.600000

print(df.sum())

num\_1 -3.5

num\_2 1.3

num\_3 0.9

dtype: float64

如果想取出其中一个索引标签的值，可以这样：df.sum()[**'num\_1'**]

sum()有一个参数axix，可取值1或0:

print(df.sum(axis=1))  
print(df.sum(axis=0))

输出为：

A -1.0

B -2.7

C 0.0

D 2.4

dtype: float64

num\_1 -3.5

num\_2 1.3

num\_3 0.9

dtype: float64

可以看出，sum()默认的axis=0的：

默认为0则求和的逻辑为每列的各个行的值相加，

如果为1则求和的逻辑为每行的各个列的值相加。

sum(),mean(),median(),min(),max(),cummin(),cummax()这些方法均可以传入参数axis来获取不同方向上的描述统计量。

argmin()和argmax()方法是对Series而言的，实际上我们遇到的就是：获取某一列数据和某一行数据最小\最大值所在索引位置，我们来看两个实例：

print(df.num\_1)  
print(df.num\_1.argmin())  
  
print(df.loc[**'A'**])  
print(df.loc[**'A'**].argmax())

输出为：

A 1.2

B -4.2

C NaN

D -0.5

Name: num\_1, dtype: float64

1

num\_1 1.2

num\_2 -2.5

num\_3 0.3

Name: A, dtype: float64

0

对于idxmin()和idxmax()使用方法和argmin()和argmax()类似：

print(df.num\_1.idxmin())  
print(df.loc[**'A'**].idxmax())

cumin()和cummax()也是类似的用法:

print(df.cummin(axis = 0))  
print(df.cummin(axis = 1))  
print(df.num\_1.cummin(axis = 0))

其他的方法我这里就不先举例了，一般来说这些方法可以传入其他的参数，我这里就不再进行延伸了，大家在使用到的时候可以查询pandas的手册即可。