pandas in python

Python Data Analysis Library 或 pandas 是基于NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。pandas提供了大量能使我们快速便捷地处理数据的函数和方法。接下来将介绍pandas的以下内容：

1. 数据结构简介：DataFrame和Series
2. 数据索引index
3. 利用pandas查询数据
4. 利用pandas的DataFrames进行统计分析
5. 利用pandas实现SQL操作
6. 利用pandas进行缺失值的处理

一、数据结构介绍

在pandas中有两类非常重要的数据结构，即序列Series和数据框DataFrame。Series类似于numpy中的一维数组，除了通吃一维数组可用的函数或方法，而且其可通过索引标签的方式获取数据，还具有索引的自动对齐功能；DataFrame类似于numpy中的二维数组，同样可以通用numpy数组的函数和方法，而且还具有其他灵活应用，后续会介绍到。

1、Series的创建

序列的创建主要有如下三种方式：

1）通过一维数组创建序列

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

```
In [3]: arr1 = np.arange(10)
```

```
In [4]: arr1
Out[4]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [5]: type(arr1)
Out[5]: numpy.ndarray
```

返回的是数组类型。

```
In [6]: s1 = pd.Series(arr1)
```

```
In [7]: s1
Out[7]: 0    0
```

```
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int32
```

```
In [8]: type(s1)
```

```
Out[8]: pandas.core.series.Series
```

返回的是序列类型。

2）通过字典的方式创建序列

```
In [9]: dic1 = {'a':10,'b':20,'c':30,'d':40,'e':50}
```

```
In [10]: dic1
```

```
Out[10]: {'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}
```

```
In [11]: type(dic1)
```

```
Out[11]: dict
```

返回的是字典类型。

```
In [12]: s2 = pd.Series(dic1)
```

```
In [13]: s2
```

```
Out[13]: a    10
b    20
c    30
d    40
e    50
dtype: int64
```

```
In [14]: type(s2)
```

```
Out[14]: pandas.core.series.Series
```

返回的是序列类型。

3）通过**DataFrame**中的某一行或某一列创建序列

这部分内容放在后面介绍，接下来开始讲一讲如何构造一个DataFrame。

2、DataFrame的创建

数据框的创建主要有三种方式：

1）通过二维数组创建数据框

In [15]:
```
arr2 = np.array(np.arange(12)).reshape(4,3)
```

In [16]:
```
arr2
```

Out[16]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [17]:
```
type(arr2)
```

Out[17]:
```
numpy.ndarray
```

返回的是数组类型。

In [18]:
```
df1 = pd.DataFrame(arr2)
```

In [19]:
```
df1
```

Out[19]:

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

In [20]:
```
type(df1)
```

Out[20]:
```
pandas.core.frame.DataFrame
```

2）通过字典的方式创建数据框

以下以两种字典来创建数据框，一个是字典列表，一个是嵌套字典。

In [21]:
```
dic2 = {'a':[1,2,3,4],'b':[5,6,7,8],
        'c':[9,10,11,12],'d':[13,14,15,16]}
```

In [22]:
```
dic2
```

Out[22]:
```
{'a': [1, 2, 3, 4],
 'b': [5, 6, 7, 8],
 'c': [9, 10, 11, 12],
 'd': [13, 14, 15, 16]}
```

In [23]:
```
type(dic2)
```

Out[23]:
```
dict
```

返回的是字典类型。

In [24]:
```
df2 = pd.DataFrame(dic2)
```

In [25]:
```
df2
```

Out[25]:

	a	b	c	d
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

In [26]:
```
type(df2)
```

Out[26]:
```
pandas.core.frame.DataFrame
```

返回的是数据框类型。

In [27]:
```
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}}
```

In [28]:
```
dic3
```

Out[28]:
```
{'one': {'a': 1, 'b': 2, 'c': 3, 'd': 4},
 'three': {'a': 9, 'b': 10, 'c': 11, 'd': 12},
 'two': {'a': 5, 'b': 6, 'c': 7, 'd': 8}}
```

返回的是字典类型。

In [29]:
```
df3 = pd.DataFrame(dic3)
```

In [30]:
```
df3
```

Out[30]:

	one	three	two
a	1	9	5
b	2	10	6
c	3	11	7
d	4	12	8

In [31]:
```
type(df3)
```

Out[31]:
```
pandas.core.frame.DataFrame
```

返回的是数据框类型。这里需要说明的是，如果使用嵌套字典创建数据框的话，嵌套字典的最外层键会形成数据框的列变量，而内层键则会形成数据框的行索引。

3）通过数据框的方式创建数据框

In [32]:
```
df4 = df3[['one','three']]
```

In [33]:
```
df4
```

Out[33]:

	one	three
a	1	9
b	2	10
c	3	11
d	4	12

In [34]:
```
type(df4)
```

Out[34]:
```
pandas.core.frame.DataFrame
```

返回的是数据框类型。

In [35]:
```
s3 = df3['one']
```

In [36]:
```
s3
```

Out[36]:
```
a    1
b    2
c    3
d    4
Name: one, dtype: int64
```

In [37]:
```
type(s3)
```

Out[37]:
```
pandas.core.series.Series
```

这里就是通过选择数据框中的某一列，返回一个序列的对象。

二、数据索引index

不论是序列也好，还是数据框也好，对象的最左边总有一个非原始数据对象，这就是索引。

序列或数据框的索引有两大用处，一个是通过索引值或索引标签获取目标数据，另一个是通过索引，可以使序列或数据框的计算、操作实现自动化对齐，下面我们就来看看这两个功能的应用。

1、通过索引值或索引标签获取数据

In [38]:
```
s4 = pd.Series(np.array([1,1,2,3,5,8]))
```

In [39]:
```
s4
```

Out[39]:
```
0    1
1    1
2    2
3    3
4    5
5    8
```

```
dtype: int32
```

如果不给序列一个指定的索引值，则序列自动生成一个从0开始的自增索引。可以通过index查看序列的索引：

In [40]:
```
s4.index
```

Out[40]:
```
RangeIndex(start=0, stop=6, step=1)
```

现在我们为序列设定一个自定义的索引值：

In [41]:
```
s4.index = ['a','b','c','d','e','f']
```

In [42]:
```
s4
```

Out[42]:
```
a    1
b    1
c    2
d    3
e    5
f    8
dtype: int32
```

序列有了索引，就可以通过索引值或索引标签进行数据的获取：

In [43]:
```
s4[3]
```

Out[43]:
```
3
```

In [44]:
```
s4['e']
```

Out[44]:
```
5
```

In [45]:
```
s4[[1,3,5]]
```

Out[45]:
```
b    1
d    3
f    8
dtype: int32
```

In [46]:
```
s4[['a','b','d','f']]
```

Out[46]:
```
a    1
b    1
d    3
f    8
dtype: int32
```

In [47]:
```
s4[:4]
```

Out[47]:
```
a    1
b    1
c    2
d    3
dtype: int32
```

In [48]:
```
s4['b':'e']
```

```
Out[48]: b    1
         c    2
         d    3
         e    5
         dtype: int32
```

千万注意：如果通过索引标签获取数据的话，末端标签所对应的值是可以返回的！在一维数组中，就无法通过索引标签获取数据，这也是序列不同于一维数组的一个方面。

2、自动化对齐

如果有两个序列，需要对这两个序列进行算术运算，这时索引的存在就体现的它的价值了--自动化对齐。

```
In [49]: s5 = pd.Series(np.array([10,15,20,30,55,80]),
                        index = ['a','b','c','d','e','f'])
```

```
In [50]: s5
```

```
Out[50]: a    10
         b    15
         c    20
         d    30
         e    55
         f    80
         dtype: int32
```

```
In [51]: s6 = pd.Series(np.array([12,11,13,15,14,16]),
                        index = ['a','c','g','b','d','f'])
```

```
In [52]: s6
```

```
Out[52]: a    12
         c    11
         g    13
         b    15
         d    14
         f    16
         dtype: int32
```

```
In [53]: s5 + s6
```

```
Out[53]: a    22.0
         b    30.0
         c    31.0
         d    44.0
         e     NaN
         f    96.0
         g     NaN
         dtype: float64
```

由于**s5**中没有对应的**g**索引，**s6**中没有对应的**e**索引，所以数据的运算会产生两个缺失值NaN。

注意，这里的算术结果就实现了两个序列索引的自动对齐，而非简单的将两个序列加总或相除。对于数据框的对齐，不仅仅是行索引的自动对齐，同时也会自动对齐列索引（变量名）。

数据框中同样有索引，而且数据框是二维数组的推广，所以数据框不仅有行索引，而且还存在列索引，关于数据框中的索引相比于序列的应用要强大的多，这部分内容将放在下面的数据查询中介绍。

三、利用**pandas**查询数据

这里的查询数据相当于R语言里的subset功能，可以通过索引有针对的选取原数据的子集、指定行、指定列等。我们先导入一个iris数据集：

In [54]:
```
iris = pd.io.parsers.read_csv('C:\\Users\\Lenovo\\Desktop\\iris.csv')
```

查询数据的前**5**行或末尾**5**行：

In [55]:
```
iris.head()
```

Out[55]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [56]:
```
iris.tail()
```

Out[56]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

查询指定的行：

In [57]:
```
iris.ix[[0,2,4,5,7]] #这里的ix索引标签函数必须是中括号[]
```

Out[57]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
7	5.0	3.4	1.5	0.2	setosa

查询指定的列：

In [58]:
```
iris[['Sepal.Length','Petal.Length','Species']].head() #如果多个列的话，必须使用双重中括号
```

Out[58]:

	Sepal.Length	Petal.Length	Species
0	5.1	1.4	setosa
1	4.9	1.4	setosa
2	4.7	1.3	setosa
3	4.6	1.5	setosa
4	5.0	1.4	setosa

也可以通过**ix**索引标签查询指定的列：

In [59]:
```
iris.ix[:,['Sepal.Length','Petal.Length','Species']].head()
```

Out[59]:

	Sepal.Length	Petal.Length	Species
0	5.1	1.4	setosa
1	4.9	1.4	setosa
2	4.7	1.3	setosa
3	4.6	1.5	setosa
4	5.0	1.4	setosa

查询指定的行和列：

In [60]:
```
iris.ix[[0,2,4,5,7],['Sepal.Length','Petal.Length','Species']]
```

Out[60]:

	Sepal.Length	Petal.Length	Species
0	5.1	1.4	setosa
2	4.7	1.3	setosa
4	5.0	1.4	setosa
5	5.4	1.7	setosa
7	5.0	1.5	setosa

这里简单说明一下**ix**的用法：df.ix[行索引,列索引]

1. ix后面必须是中括号
2. 多个行索引或列索引必须用中括号括起来
3. 如果选择所有行索引或列索引，则用英文状态下的冒号:表示

以上是从行或列的角度查询数据的子集，现在我们来看看如何通过因子索引实现数据的子集查询。

查询**virginica**的信息的前五行：

In [62]:
```
iris[iris['Species']=='virginica'].head()
```

Out[62]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
100	6.3	3.3	6.0	2.5	virginica
101	5.8	2.7	5.1	1.9	virginica
102	7.1	3.0	5.9	2.1	virginica
103	6.3	2.9	5.6	1.8	virginica
104	6.5	3.0	5.8	2.2	virginica

查询所有**Petal.Length**大于**6.0**的**virginica**的信息：

In [65]:
```
iris[(iris['Species']=='virginica') & (iris['Petal.Length']>6.0)]
```

Out[65]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
105	7.6	3.0	6.6	2.1	virginica
107	7.3	2.9	6.3	1.8	virginica
109	7.2	3.6	6.1	2.5	virginica
117	7.7	3.8	6.7	2.2	virginica
118	7.7	2.6	6.9	2.3	virginica
122	7.7	2.8	6.7	2.0	virginica
130	7.4	2.8	6.1	1.9	virginica
131	7.9	3.8	6.4	2.0	virginica
135	7.7	3.0	6.1	2.3	virginica

查询所有**Petal.Length**大于**6.0**的**virginica**的**Petal.Width**：

In [66]:
```
iris[(iris['Species']=='virginica') & (iris['Petal.Length']>6.0)]['Petal
.Width']
```

Out[66]:
```
105    2.1
107    1.8
109    2.5
117    2.2
118    2.3
122    2.0
130    1.9
131    2.0
135    2.3
Name: Petal.Width, dtype: float64
```

上面的查询逻辑其实非常的简单，需要注意的是，如果是多个条件的查询，必须在**&**（且）或者**|**（或）的两端条件用括号括起来。

四、统计分析

pandas模块为我们提供了非常多的描述性统计分析的指标函数，如总和、均值、最小值、最大值等，我们来具体看看这些函数：

首先随机生成三组数据

In [67]:
```
np.random.seed(1234)
```

In [68]:
```
d1 = pd.Series(2*np.random.normal(size = 100)+3)
```

In [69]:
```
d2 = np.random.f(2,4,size = 100)
```

In [70]:
```
d3 = np.random.randint(1,100,size = 100)
```

In [71]:
```
d1.count() #非空元素的计算
```

Out[71]: 100

In [73]:
```
d1.min() #最小值
```

Out[73]: -4.1270333212494705

In [74]:
```
d1.max() #最大值
```

Out[74]: 7.7819210309260658

In [75]:
```
d1.idxmin() #最小值的位置，类似于R中的which.min函数
```

Out[75]: 81

In [76]:
```
d1.idxmax() #最大值的位置，类似于R中的which.max函数
```

Out[76]: 39

In [77]:
```
d1.quantile(0.1) #10%分位数
```

Out[77]: 0.68701846440699277

In [78]:
```
d1.sum() #求和
```

Out[78]: 307.0224566250874

In [79]:
```
d1.mean() #均值
```

Out[79]: 3.070224566250874

In [80]:
```
d1.median() #中位数
```

Out[80]: 3.204555266776845

In [81]:
```
d1.mode() #众数
```

Out[81]: Series([], dtype: float64)

In [ ]:
```

```

In [82]:
```
d1.var() #方差
```

Out[82]:
```
4.005609378535085
```

In [83]:
```
d1.std() #标准差
```

Out[83]:
```
2.0014018533355777
```

In [84]:
```
d1.mad() #平均绝对偏差
```

Out[84]:
```
1.5112880411556109
```

In [85]:
```
d1.skew() #偏度
```

Out[85]:
```
-0.64947807604842933
```

In [86]:
```
d1.kurt() #峰度
```

Out[86]:
```
1.2201094052398012
```

In [87]:
```
d1.describe() #一次性输出多个描述性统计指标
```

Out[87]:
```
count    100.000000
mean       3.070225
std        2.001402
min       -4.127033
25%        2.040101
50%        3.204555
75%        4.434788
max        7.781921
dtype: float64
```

必须注意的是，**describe**方法只能针对序列或数据框，一维数组是没有这个方法的。

这里自定义一个函数，将这些统计描述指标全部汇总到一起：

In [88]:
```
def stats(x):
    return pd.Series([x.count(),x.min(),x.idxmin(),
                      x.quantile(.25),x.median(),
                      x.quantile(.75),x.mean(),
                      x.max(),x.idxmax(),
                      x.mad(),x.var(),
                      x.std(),x.skew(),x.kurt()],
                     index = ['Count','Min','Whicn_Min',
                              'Q1','Median','Q3','Mean',
                              'Max','Which_Max','Mad',
                              'Var','Std','Skew','Kurt'])
```

In [89]:
```
stats(d1)
```

Out[89]:
```
Count        100.000000
Min           -4.127033
Whicn_Min     81.000000
Q1             2.040101
Median         3.204555
Q3             4.434788
Mean           3.070225
Max            7.781921
Which_Max     39.000000
```

```
Mad             1.511288
Var             4.005609
Std             2.001402
Skew           -0.649478
Kurt            1.220109
dtype: float64
```

在实际的工作中，我们可能需要处理的是一系列的数值型数据框，如何将这个函数应用到数据框中的每一列呢？可以使用**apply**函数，这个非常类似于R中的apply的应用方法。

将之前创建的**d1,d2,d3**数据构建数据框：

In [90]:
```
df = pd.DataFrame(np.array([d1,d2,d3]).T,columns=['x1','x2','x3'])
```

In [91]:
```
df.head()
```

Out[91]:

	x1	x2	x3
0	3.942870	1.369531	55.0
1	0.618049	0.943264	68.0
2	5.865414	0.590663	73.0
3	2.374696	0.206548	59.0
4	1.558823	0.223204	60.0

In [92]:
```
df.apply(stats)
```

Out[92]:

	x1	x2	x3
Count	100.000000	100.000000	100.000000
Min	-4.127033	0.014330	3.000000
Whicn_Min	81.000000	72.000000	76.000000
Q1	2.040101	0.249580	25.000000
Median	3.204555	1.000613	54.500000
Q3	4.434788	2.101581	73.000000
Mean	3.070225	2.028608	51.490000
Max	7.781921	18.791565	98.000000
Which_Max	39.000000	53.000000	96.000000
Mad	1.511288	1.922669	24.010800
Var	4.005609	10.206447	780.090808
Std	2.001402	3.194753	27.930106
Skew	-0.649478	3.326246	-0.118917
Kurt	1.220109	12.636286	-1.211579

非常完美，就这样很简单的创建了数值型数据的统计性描述。如果是离散型数据呢？就不能用这个统计口径了，我们需要统计离散变量的观测数、唯一值个数、众数水平及个数。你只需要使

用describe方法就可以实现这样的统计了。

In [93]:
```
iris['Species'].describe()
```

Out[93]:
```
count        150
unique         3
top       setosa
freq          50
Name: Species, dtype: object
```

除以上的简单描述性统计之外，还提供了连续变量的相关系数（corr）和协方差矩阵（cov）的求解，这个跟R语言是一致的用法。

In [94]:
```
df.corr()
```

Out[94]:

	x1	x2	x3
x1	1.000000	0.136085	0.037185
x2	0.136085	1.000000	-0.005688
x3	0.037185	-0.005688	1.000000

关于相关系数的计算可以调用pearson方法或kendell方法或spearman方法，默认使用pearson方法。

In [95]:
```
df.corr('spearman')
```

Out[95]:

	x1	x2	x3
x1	1.00000	0.178950	0.006590
x2	0.17895	1.000000	-0.033874
x3	0.00659	-0.033874	1.000000

如果只想关注某一个变量与其余变量的相关系数的话，可以使用corrwith,如下方只关心x1与其余变量的相关系数：

In [96]:
```
df.corrwith(df['x1'])
```

Out[96]:
```
x1    1.000000
x2    0.136085
x3    0.037185
dtype: float64
```

数值型数据的协方差矩阵：

In [97]:
```
df.cov()
```

Out[97]:

	x1	x2	x3
x1	4.005609	0.870124	2.078596
x2	0.870124	10.206447	-0.507512
x3	2.078596	-0.507512	780.090808

五、类似于SQL的操作

在SQL中常见的操作主要是增、删、改、查几个动作，那么pandas能否实现对数据的这几项操作呢？答案是Of Course!

增：添加新行或增加新列

In [108]:
```
dic = {'Sepal.Length':[7.2,7.3],
       'Sepal.Width':[2.9,3.6],
       'Petal.Length':[6.2,6.6],
       'Petal.Width':[1.8,2.2],
       'Species':['virginica','setosa']}
```

In [109]:
```
iris2 = pd.DataFrame(dic)
```

In [113]:
```
iris2
```

Out[113]:

	Petal.Length	Petal.Width	Sepal.Length	Sepal.Width	Species
0	6.2	1.8	7.2	2.9	virginica
1	6.6	2.2	7.3	3.6	setosa

现在将iris2中的数据新增到iris中，可以通过**concat**函数实现：

In [114]:
```
iris3 = pd.concat([iris,iris2])
```

In [121]:
```
iris3.tail()
```

Out[121]:

	Petal.Length	Petal.Width	Sepal.Length	Sepal.Width	Species
147	5.2	2.0	6.5	3.0	virginica
148	5.4	2.3	6.2	3.4	virginica
149	5.1	1.8	5.9	3.0	virginica
0	6.2	1.8	7.2	2.9	virginica
1	6.6	2.2	7.3	3.6	setosa

在数据库中union必须要求两张表的列顺序一致，而这里concat函数可以自动对齐两个数据框的变量！

新增列的话，其实在pandas中就更简单了，例如在iris2中新增一列Status：

In [116]:
```
pd.DataFrame(iris2,columns=['Petal.Length','Petal.Width','Sepal.Length',
'Sepal.Width','Species','Status'])
```

Out[116]:

	Petal.Length	Petal.Width	Sepal.Length	Sepal.Width	Species	Status
0	6.2	1.8	7.2	2.9	virginica	NaN
1	6.6	2.2	7.3	3.6	setosa	NaN

对于新增的列没有赋值，就会出现空NaN的形式。

删：删除表、观测行或变量列

删除数据框iris2,通过del命令实现，该命令可以删除Python的所有对象。

In [117]:
```
del iris2
```

In [118]:
```
iris2
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-118-9522e1cbb712> in <module>()
----> 1 iris2

NameError: name 'iris2' is not defined
```

删除指定的行

In [120]:
```
iris.drop([0,1,2,3]).head() #删除第1,2,3,4行数据
```

Out[120]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa

原数据中的第1,2,3,4行的数据已经被删除了。

根据布尔索引删除行数据，其实这个删除就是保留删除条件的反面数据，例如删除所有非virginica的数据：

In [125]:
```
iris[iris['Species']=='virginica'].head()
```

Out[125]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
100	6.3	3.3	6.0	2.5	virginica
101	5.8	2.7	5.1	1.9	virginica
102	7.1	3.0	5.9	2.1	virginica
103	6.3	2.9	5.6	1.8	virginica
104	6.5	3.0	5.8	2.2	virginica

删除指定的列

In [126]:
```
iris.drop(['Petal.Length','Petal.Width'],axis=1).head()
```

Out[126]:

	Sepal.Length	Sepal.Width	Species
0	5.1	3.5	setosa
1	4.9	3.0	setosa
2	4.7	3.2	setosa
3	4.6	3.1	setosa
4	5.0	3.6	setosa

我们发现，不论是删除行还是删除列，都可以通过**drop**方法实现，只需要设定好删除的轴即可，即调整drop方法中的axis参数。默认该参数为0，表示删除行观测，如果需要删除列变量，则需设置为1。

改：修改原始记录的值

如果发现表中的某些数据错误了，如何更改原来的值呢？我们试试结合布尔索引和赋值的方法：

例如，把iris3中Sepal.Width大于3.5的记录的Sepal.Length值改为5

In [128]:
```
iris3.ix[iris3['Sepal.Width']>3.5,'Sepal.Length']=5
```

In [129]:
```
iris3[iris3['Sepal.Width']>3.5]
```

Out[129]:

	Petal.Length	Petal.Width	Sepal.Length	Sepal.Width	Species
4	1.4	0.2	5.0	3.6	setosa
5	1.7	0.4	5.0	3.9	setosa
10	1.5	0.2	5.0	3.7	setosa
14	1.2	0.2	5.0	4.0	setosa
15	1.5	0.4	5.0	4.4	setosa
16	1.3	0.4	5.0	3.9	setosa
18	1.7	0.3	5.0	3.8	setosa
19	1.5	0.3	5.0	3.8	setosa
21	1.5	0.4	5.0	3.7	setosa
22	1.0	0.2	5.0	3.6	setosa
32	1.5	0.1	5.0	4.1	setosa
33	1.4	0.2	5.0	4.2	setosa
37	1.4	0.1	5.0	3.6	setosa
44	1.9	0.4	5.0	3.8	setosa

46	1.6	0.2	5.0	3.8	setosa
48	1.5	0.2	5.0	3.7	setosa
109	6.1	2.5	5.0	3.6	virginica
117	6.7	2.2	5.0	3.8	virginica
131	6.4	2.0	5.0	3.8	virginica
1	6.6	2.2	5.0	3.6	setosa

关于索引的操作非常灵活、方便吧，就这样轻松搞定数据的更改。

查：有关数据查询部分，上面已经介绍过，下面重点讲讲聚合、排序和多表连接操作。

聚合：pandas模块中可以通过**groupby()**函数实现数据的聚合操作

聚合

根据Species分组，计算各组别中各个变量的平均值：

In [130]:
```
iris.groupby('Species').mean()
```

Out[130]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Species				
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

如果不对原始数据作限制的话，聚合函数会自动选择数值型数据进行聚合计算。如果不想对Sepal计算平均值的话，就需要剔除改变量：

In [132]:
```
iris.drop(['Sepal.Length','Sepal.Width'],axis=1).groupby('Species').mean
()
```

Out[132]:

	Petal.Length	Petal.Width
Species		
setosa	1.462	0.246
versicolor	4.260	1.326
virginica	5.552	2.026

groupby还可以使用多个分组变量，还可以对每个分组计算多个统计量：

In [133]:
```
iris.drop(['Sepal.Length','Sepal.Width'],axis=1).groupby('Species').agg(
[np.mean,np.median])
```

Out[133]:

	Petal.Length		Petal.Width	
	mean	**median**	**mean**	**median**

Species				
setosa	1.462	1.50	0.246	0.2
versicolor	4.260	4.35	1.326	1.3
virginica	5.552	5.55	2.026	2.0

In [134]:
```
iris.drop(['Sepal.Length','Sepal.Width'],axis=1).groupby('Species').describe()
```

Out[134]:

		Petal.Length	Petal.Width
Species			
setosa	**count**	50.000000	50.000000
	mean	1.462000	0.246000
	std	0.173664	0.105386
	min	1.000000	0.100000
	25%	1.400000	0.200000
	50%	1.500000	0.200000
	75%	1.575000	0.300000
	max	1.900000	0.600000
versicolor	**count**	50.000000	50.000000
	mean	4.260000	1.326000
	std	0.469911	0.197753
	min	3.000000	1.000000
	25%	4.000000	1.200000
	50%	4.350000	1.300000
	75%	4.600000	1.500000
	max	5.100000	1.800000
virginica	**count**	50.000000	50.000000
	mean	5.552000	2.026000
	std	0.551895	0.274650
	min	4.500000	1.400000
	25%	5.100000	1.800000
	50%	5.550000	2.000000
	75%	5.875000	2.300000
	max	6.900000	2.500000

是不是很简单，只需一句就能完成SQL中的SELECT...FROM...GROUP BY...功能，何乐而不为呢？

排序

排序在日常的统计分析中还是比较常见的操作，我们可以使用**order**、**sort_index**和**sort_values**实现序列和数据框的排序工作：

In [141]:
```
series = pd.Series(np.array(np.random.randint(1,20,10)))
```

In [143]:
```
series.sort_values()
```

Out[143]:
```
9     6
5     8
7    10
1    15
2    16
4    16
8    16
6    17
0    18
3    18
dtype: int32
```

我们再试试降序排序的设置：

In [144]:
```
series.sort_values(ascending=False)
```

Out[144]:
```
3    18
0    18
6    17
8    16
4    16
2    16
1    15
7    10
5     8
9     6
dtype: int32
```

在数据框中一般都是按值排序，例如：

In [147]:
```
iris.sort_values(by=['Sepal.Length','Species']).head(30)
```

Out[147]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
13	4.3	3.0	1.1	0.1	setosa
8	4.4	2.9	1.4	0.2	setosa
38	4.4	3.0	1.3	0.2	setosa
42	4.4	3.2	1.3	0.2	setosa
41	4.5	2.3	1.3	0.3	setosa
3	4.6	3.1	1.5	0.2	setosa
6	4.6	3.4	1.4	0.3	setosa
22	4.6	3.6	1.0	0.2	setosa
47	4.6	3.2	1.4	0.2	setosa

2	4.7	3.2	1.3	0.2	setosa
29	4.7	3.2	1.6	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa
24	4.8	3.4	1.9	0.2	setosa
30	4.8	3.1	1.6	0.2	setosa
45	4.8	3.0	1.4	0.3	setosa
1	4.9	3.0	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
34	4.9	3.1	1.5	0.2	setosa
37	4.9	3.6	1.4	0.1	setosa
57	4.9	2.4	3.3	1.0	versicolor
106	4.9	2.5	4.5	1.7	virginica
4	5.0	3.6	1.4	0.2	setosa
7	5.0	3.4	1.5	0.2	setosa
25	5.0	3.0	1.6	0.2	setosa
26	5.0	3.4	1.6	0.4	setosa
35	5.0	3.2	1.2	0.2	setosa
40	5.0	3.5	1.3	0.3	setosa
43	5.0	3.5	1.6	0.6	setosa
49	5.0	3.3	1.4	0.2	setosa

多表连接

表之间的连接也是非常常见的数据库操作，连接分内连接和外连接，在数据库语言中通过join关键字实现，**pandas**我比较建议使用**merger**函数实现数据的各种连接操作。

In [153]:
```
dic2 = {'ID':['0001','0002','0003','0004','0005','0006','0007','0008'],
        'SEX':['F','F','F','M','M','M','F','M'],
        'AGE':['21','23','25','21','22','25','24','25',]}
dic3 = {'ID':['0001','0002','0004','0005','0007','0008'],
         'SCORE':['92','93','100','96','95','99']}
student = pd.DataFrame(dic2)
score = pd.DataFrame(dic3)
```

In [154]:
```
student
```

Out[154]:

	AGE	ID	SEX
0	21	0001	F
1	23	0002	F

2	25	0003	F
3	21	0004	M
4	22	0005	M
5	25	0006	M
6	24	0007	F
7	25	0008	M

In [155]:
```
score
```

Out[155]:

	ID	SCORE
0	0001	92
1	0002	93
2	0004	100
3	0005	96
4	0007	95
5	0008	99

In [159]:
```
stu_score1 = pd.merge(student,score,on='ID')
```

In [160]:
```
stu_score1
```

Out[160]:

	AGE	ID	SEX	SCORE
0	21	0001	F	92
1	23	0002	F	93
2	21	0004	M	100
3	22	0005	M	96
4	24	0007	F	95
5	25	0008	M	99

注意，默认情况下，**merge**函数实现的是两个表之间的内连接，即返回两张表中共同部分的数据。可以通过how参数设置连接的方式，left为左连接；right为右连接；outer为外连接。

In [161]:
```
stu_score2 = pd.merge(student,score,on='ID',how='left')
```

In [162]:
```
stu_score2
```

Out[162]:

	AGE	ID	SEX	SCORE
0	21	0001	F	92
1	23	0002	F	93
2	25	0003	F	NaN
3	21	0004	M	100

4	22	0005	M	96
5	25	0006	M	NaN
6	24	0007	F	95
7	25	0008	M	99

左连接实现的是保留student表中的所有信息，同时将score表的信息与之配对，能配多少配多少，对于没有配对上的Name，将会显示成绩为NaN。

六、缺失值处理

现实生活中的数据是非常杂乱的，其中缺失值也是非常常见的，对于缺失值的存在可能会影响到后期的数据分析或挖掘工作，那么我们该如何处理这些缺失值呢？常用的有三大类方法，即删除法、填补法和插值法。

删除法：当数据中的某个变量大部分值都是缺失值，可以考虑删除改变量；当缺失值是随机分布的，且缺失的数量并不是很多是，也可以删除这些缺失的观测。

替补法：对于连续型变量，如果变量的分布近似或就是正态分布的话，可以用均值替代那些缺失值；如果变量是有偏的，可以使用中位数来代替那些缺失值；对于离散型变量，我们一般用众数去替换那些存在缺失的观测。

插补法：插补法是基于蒙特卡洛模拟法，结合线性模型、广义线性模型、决策树等方法计算出来的预测值替换缺失值。

我们这里就介绍简单的删除法和替补法：

In [163]:
```
s = stu_score2['SCORE']
```

In [165]:
```
s
```

Out[165]:
```
0     92
1     93
2    NaN
3    100
4     96
5    NaN
6     95
7     99
Name: SCORE, dtype: object
```

这是一组含有缺失值的序列，我们可以结合sum函数和isnull函数来检测数据中含有多少缺失值：

In [166]:
```
sum(pd.isnull(s))
```

Out[166]:
```
2
```

直接删除缺失值

In [168]:
```
s.dropna()
```

Out[168]:
```
0     92
```

```
1     93
3    100
4     96
6     95
7     99
Name: SCORE, dtype: object
```

默认情况下，**dropna**会删除任何含有缺失值的行，我们再构造一个数据框试试：

In [169]:
```
df = pd.DataFrame([[1,1,2],
                   [3,5,np.nan],
                  [13,21,34],
                  [55,np.nan,10],
                  [np.nan,np.nan,np.nan],
                  [np.nan,1,2]],
                 columns=['x1','x2','x3'])
```

In [170]:
```
df
```

Out[170]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	NaN
2	13.0	21.0	34.0
3	55.0	NaN	10.0
4	NaN	NaN	NaN
5	NaN	1.0	2.0

In [171]:
```
df.dropna()
```

Out[171]:

	x1	x2	x3
0	1.0	1.0	2.0
2	13.0	21.0	34.0

返回结果表明，数据中只要含有缺失值**NaN,**该数据行就会被删除，如果使用参数**how='all'**，则表明只删除那些所有变量值都为缺失值的观测。

In [173]:
```
df.dropna(how='all')
```

Out[173]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	NaN
2	13.0	21.0	34.0
3	55.0	NaN	10.0
5	NaN	1.0	2.0

使用一个常量来填补缺失值，可以使用fillna函数实现简单的填补工作：

1）用**0**填补所有缺失值

In [174]:
```
df.fillna(0)
```

Out[174]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	0.0
2	13.0	21.0	34.0
3	55.0	0.0	10.0
4	0.0	0.0	0.0
5	0.0	1.0	2.0

2）采用前项填充或后向填充

In [175]:
```
df.fillna(method='ffill') #用前一个观测值填充
```

Out[175]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	2.0
2	13.0	21.0	34.0
3	55.0	21.0	10.0
4	55.0	21.0	10.0
5	55.0	1.0	2.0

In [176]:
```
df.fillna(method='bfill') #用后一个观测值填充
```

Out[176]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	34.0
2	13.0	21.0	34.0
3	55.0	1.0	10.0
4	NaN	1.0	2.0
5	NaN	1.0	2.0

3）使用常量填充不同的列

In [177]:
```
df.fillna({'x1':1,'x2':2,'x3':3})
```

Out[177]:

	x1	x2	x3
0	1.0	1.0	2.0

1	3.0	5.0	3.0
2	13.0	21.0	34.0
3	55.0	2.0	10.0
4	1.0	2.0	3.0
5	1.0	1.0	2.0

4）用均值或中位数填充各自的列

In [178]:
```
df.fillna({'x1':df['x1'].median(),
           'x2':df['x2'].mean(),
           'x3':df['x3'].mean()})
```

Out[178]:

	x1	x2	x3
0	1.0	1.0	2.0
1	3.0	5.0	12.0
2	13.0	21.0	34.0
3	55.0	7.0	10.0
4	8.0	7.0	12.0
5	8.0	1.0	2.0

很显然，在使用填充法时，相对于常数填充或前项、后项填充，使用各列的众数、均值或中位数填充要更加合理一点，这也是工作中常用的一个快捷手段。

以上就是pandas的基本内容。