

NumPy虽然提供了方便的数组处理功能，但它缺少数据处理、分析所需的许多快速工具。Pandas基于 NumPy开发，提供了众多更高级的数据处理功能，主要包括：

- 数据存储
- 数据清洗
- 数据变换
- 分组运算
- 数据可视化

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

Series 对象

Series和DataFrame是Pandas中最常用的两个对象。

Series是 Pandas中最基本的对象，它定义了NumPy的ndarray对象的接口 `__array__()`，因此可以用NumPy的数组处理函数直接对Series对象进行处理。Series对象除了支持使用**整数位置**作为下标存取元素之外，还可以使用**索引标签**作为下标存取元素，这个功能与字典类似。

每个Series对象实际上都由两个数组组成：

- **index**: 它是从ndarray数组继承的Index索引对象，**保存标签信息**。若创建Series对象时不指定index, 将自动创建一个表示位置下标的整数索引。
- **values**: **保存元素值**的ndarray数组。

```
import pandas as pd
s = pd.Series([90, 60, 36, 75, 87], index=["语文", "政治", "历史", "地理", "英语"])
print(" 索引: ", s.index)
print("值数组: ", s.values, type(s.values))
```

```
索引: Index(['语文', '政治', '历史', '地理', '英语'], dtype='object')
值数组: [90 60 36 75 87] <class 'numpy.ndarray'>
```

Series 对象

创建Series时未指定标签索引index，将自动创建一个表示位置下标的整数索引。

```
s2 = pd.Series([90, 60, 36, 75, 87])  
print(" 索引: ", s2.index)  
print("值数组: ", s2.values, type(s2.values))
```

```
索引: RangeIndex(start=0, stop=5, step=1)  
值数组: [90 60 36 75 87] <class 'numpy.ndarray'>
```

Series 对象

Series对象的下标运算同时支持**位置**和**标签**两种形式。

```
s = pd.Series([90, 60, 36, 75, 87], index=["语文", "政治", "历史", "地理", "英语"])
```

<code>print(" s[0]:", s[0])</code>	<code>print("s['语文']:", s['语文'])</code>
s[0]: 90	s['语文']: 90

Series对象还支持**位置切片**和**标签切片**。位置切片遵循Python的切片规则，包括起始位置，但**不包括结束位置**；但标签切片则**同时包括起始标签和结束标签**。

<code>print(s[1:3])</code>	<code>print(s['政治':'地理'])</code>
政治 60 历史 36 dtype: int64	政治 60 历史 36 地理 75 dtype: int64

Series 对象

和ndarray数组一样，还可以使用**位置列表**或位置数组存取元素，同样也可以使用标签列表和标签数组。

```
s = pd.Series([90, 60, 36, 75, 87], index=["语文", "政治", "历史", "地理", "英语"])
```

<code>print(s[[1,3,2]])</code>	<code>print(s[['政治','地理','历史']])</code>
政治 60	政治 60
地理 75	地理 75
历史 36	历史 36
dtype: int64	dtype: int64

Series对象同时具有数组和字典的功能，因此它也支持字典的一些方法，例如Series.iteritems():

<code>print(list(s.iteritems()))</code>
[('语文', 90), ('政治', 60), ('历史', 36), ('地理', 75), ('英语', 87)]

Series 对象

当两个Series对象进行操作符运算时，Pandas会**按照标签对齐**元素，也就是说运算操作符会对标签相同的两个元素进行计算。在下面的例子中，s 中标签为"语文"的元素和s2 中标签为"语文"的元素相加得到结果中的156。当某一方的标签不存在时，默认以 NaN (Not a Number)填充。由于NaN是浮点数中的一个特殊值，因此输出的Series对象的元素类型被转换为float64。

```
s = pd.Series([90, 60, 36, 75, 87], index=["语文", "政治", "历史", "地理", "英语"])
s2 = pd.Series([72, 83, 66, 80], index=["政治", "历史", "语文", "英语"])
```

print(s)	print(s2)	print(s+s2)
语文 90	政治 72	历史 119.0
政治 60	历史 83	地理 NaN
历史 36	语文 66	政治 132.0
地理 75	英语 80	英语 167.0
英语 87	dtype: int64	语文 156.0
dtype: int64		dtype: float64

DataFrame 对象

DataFrame(数据框/数据表)对象是Pandas中最常用的数据对象。Pandas提供了将许多数据结构（字典、numpy二维数组等）转换为DataFrame对象的方法，还提供了许多输入输出函数（read_csv()等）来将各种文件格式转换成DataFrame 对象。

DataFrame 对象：DataFrame的结构

DataFrame对象是一个二维表格。其中，**每列中的元素类型必须一致**，而**不同的列可以拥有不同的元素类型**。

每行和每列都有索引，默认是**位置索引**，但通常会指定**标签索引**（相当于表格的列名和行名），还可以给列索引和行索引取名。

索引通常只有一级，但是也可以建立多级索引（第0级、第1级、...），多级索引相当于数据分类。行索引和列索引都可以是多级的。

列索引名		列索引						
	Measures	pH	Dens	Ca	Conduc	Date	Name	
行索引名	Depth	Contour						
	0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
		Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
Top		5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy	
行索引	10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
		Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
		Top	4.8600	1.3325	10.2375	3.5825	2015-04-11	Diana
第0级索引		第1级	行	列	数据			

DataFrame 对象: DataFrame的结构

	A	B	C	D	E	F	G	H
1	Depth	Contour	pH	Dens	Ca	Conduc	Date	Name
2	0-10	Depression	5.3525	0.9775	10.685	1.4725	2015/5/26 0:00	Lois
3	0-10	Slope	5.5075	1.05	12.2475	2.05	2015/4/30 0:00	Roy
4	0-10	Top	5.3325	1.0025	13.385	1.3725	2015/5/21 0:00	Roy
5	10-30	Depression	4.88	1.3575	7.5475	5.48	2015/3/21 0:00	Lois
6	10-30	Slope	5.2825	1.3475	9.515	4.91	2015/2/6 0:00	Diana
7	10-30	Top	4.85	1.3325	10.2375	3.5825	2015/4/11 0:00	Diana

```
df_soil = pd.read_csv("data/Soils-simple.csv", index_col=[0, 1], parse_dates=["Date"])
df_soil.columns.name = "Measures" # 设置列索引名
print(df_soil)
```

通过index_col参数指定第0和第1列为行索引，用parse_dates参数指定进行日期转换的列，在指定列时可以使用列的序号(是文件中的列序号)或列名，例如这里也可以使用parse_dates=[6]。

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

DataFrame 对象：DataFrame的结构

DataFrame对象是一个二维表格。其中，**每列中的元素类型必须一致**，而**不同的列可以拥有不同的元素类型**。

在本例中，有4列浮点数类型、1列日期类型和1列object类型。object类型的列可以保存任何Python对象，在Pandas中字符串列使用object类型。

DataFrame对象的**dtypes属性**可以获得表示各个列类型的Series对象：

Measures		pH	Dens	Ca	Conduc	Date	Name
0-10	Depth						
	Contour						
	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
10-30	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

与数组类似，通过**shape属性**可以得到DataFrame的行数和列数：

```
print(df_soil.shape)
```

```
(6, 6)
```

```
print(df_soil.dtypes)
```

```
Measures
pH                float64
Dens              float64
Ca                float64
Conduc            float64
Date              datetime64[ns]
Name              object
dtype: object
```

DataFrame 对象：DataFrame的结构

DataFrame对象拥有行索引和列索引，可以通过索引标签对其中的数据进行存取。
index属性保存行索引，而**columns**属性保存列索引。在本例中列索引是一个Index对象，索引对象的名称可以通过其name属性存取：

```
print(df_soil.columns)
```

```
print(df_soil.columns.name)
```

```
Index(['pH', 'Dens', 'Ca', 'Conduc', 'Date', 'Name'], dtype='object', name='Measures')  
Measures
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth Contour							
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

DataFrame 对象: DataFrame的结构

- **values**属性是DataFrame对象中存储数据的numpy二维ndarray数组。由于本例中的列类型不统一，所以数组元素类型为object:

```
print(df_soil.values)
```

```
print(df_soil.values.dtype)
```

```
[[5.35250000000001 0.9775 10.685 1.4725 Timestamp('2015-05-26 00:00:00') 'Lois']  
[5.5075 1.05 12.2475 2.05 Timestamp('2015-04-30 00:00:00') 'Roy']  
[5.33250000000005 1.0025 13.3850000000002 1.3725 Timestamp('2015-05-21 00:00:00') 'Roy']  
[4.88 1.3575 7.5475 5.48 Timestamp('2015-03-21 00:00:00') 'Lois']  
[5.28250000000015 1.3475 9.515 4.91 Timestamp('2015-02-06 00:00:00') 'Diana']  
[4.85 1.3325 10.2375 3.58250000000005 Timestamp('2015-04-11 00:00:00') 'Diana']]
```

object

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth Contour							
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

DataFrame 对象：获取指定行或列

在本例中，行索引是一个表示多级索引的MultiIndex对象，每级的索引名可以通过names属性存取：

```
print(df_soil.index)
print(df_soil.index.names)

MultiIndex([( '0-10',  'Depression'),
            ( '0-10',  'Slope'),
            ( '0-10',  'Top'),
            ('10-30',  'Depression'),
            ('10-30',  'Slope'),
            ('10-30',  'Top')],
           names=['Depth', 'Contour'])
['Depth', 'Contour']
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

DataFrame 对象：获取指定行或列

与二维数组相同，DataFrame对象也有两个轴，它的第0轴为纵轴，第1轴为横轴。当某个方法或函数有axis、orient等参数时，该参数可以使用整数0和1或者'index'和'columns'来表示纵轴方向和横轴方向。

[]运算符可以通过**列索引标签****获取指定的列**，当下标是单个标签时，所得到的是Series对象，例如df_soil['pH']; 而当下标是**标签列表**时，则得到一个新的DataFrame对象，例如df_soil['pH', 'Ca']:

print(df_soil['pH'])

Depth	Contour	
0-10	Depression	5.3525
	Slope	5.5075
	Top	5.3325
10-30	Depression	4.8800
	Slope	5.2825
	Top	4.8500

Name: pH, dtype: float64

print(df_soil[['pH','Ca']])

Measures		pH	Ca
Depth	Contour		
0-10	Depression	5.3525	10.6850
	Slope	5.5075	12.2475
	Top	5.3325	13.3850
10-30	Depression	4.8800	7.5475
	Slope	5.2825	9.5150
	Top	4.8500	10.2375

DataFrame 对象：获取指定行或列

.loc[]可通过**行索引标签**获取指定的行，例如`df.loc["0-10", "Top"]`获得 Depth为“0-10”，Contour为“Top”的行，而 `df.loc["0-10"]`获取Depth为“0-10”的所有行。

当结果为一行时，得到的是Series对象，结果为多行时得到的是DataFrame对象。

`print(df_soil.loc['0-10', 'Top'])`

```
Measures
pH          5.3325
Dens         1.0025
Ca           13.385
Conduc       1.3725
Date      2015-05-21 00:00:00
Name              Roy
Name: (0-10, Top), dtype: object
```

`print(df_soil.loc['0-10'])`

Measures	pH	Dens	Ca	Conduc	Date	Name
Contour						
Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy

注意这里有两级行索引

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

创建DataFrame对象：将内存中的数据转换为DataFrame对象

调用DataFrame()可以将多种格式的数据转换成DataFrame对象，它的三个参数data、index和columns分别为数据、行索引和列索引。data参数可以是：

- **二维数组**或者能转换为二维数组的**嵌套列表**。
- **字典**：字典中的每对“键-值”将成为DataFrame对象的一列。值可以是一维数组、列表或Series对象。

当未指定索引标签index和columns时，采用整数位置索引。

```
A = np.random.randint(0, 10, (4, 2))
df1 = pd.DataFrame(data=A, index=['r1', 'r2', 'r3', 'r4'], columns=['c1', 'c2'])
df2 = pd.DataFrame(data=A, columns=['c1', 'c2'])
df3 = pd.DataFrame(data=A)
```

print(df1)	print(df2)	print(df3)																																													
<table><tr><th></th><th>c1</th><th>c2</th></tr><tr><th>r1</th><td>3</td><td>0</td></tr><tr><th>r2</th><td>2</td><td>6</td></tr><tr><th>r3</th><td>1</td><td>5</td></tr><tr><th>r4</th><td>0</td><td>8</td></tr></table>		c1	c2	r1	3	0	r2	2	6	r3	1	5	r4	0	8	<table><tr><th></th><th>c1</th><th>c2</th></tr><tr><th>0</th><td>3</td><td>0</td></tr><tr><th>1</th><td>2</td><td>6</td></tr><tr><th>2</th><td>1</td><td>5</td></tr><tr><th>3</th><td>0</td><td>8</td></tr></table>		c1	c2	0	3	0	1	2	6	2	1	5	3	0	8	<table><tr><th></th><th>0</th><th>1</th></tr><tr><th>0</th><td>3</td><td>0</td></tr><tr><th>1</th><td>2</td><td>6</td></tr><tr><th>2</th><td>1</td><td>5</td></tr><tr><th>3</th><td>0</td><td>8</td></tr></table>		0	1	0	3	0	1	2	6	2	1	5	3	0	8
	c1	c2																																													
r1	3	0																																													
r2	2	6																																													
r3	1	5																																													
r4	0	8																																													
	c1	c2																																													
0	3	0																																													
1	2	6																																													
2	1	5																																													
3	0	8																																													
	0	1																																													
0	3	0																																													
1	2	6																																													
2	1	5																																													
3	0	8																																													

创建DataFrame对象：将内存中的数据转换为DataFrame对象

调用DataFrame()可以将多种格式的数据转换成DataFrame对象，它的三个参数data、index和columns分别为数据、行索引和列索引。data参数可以是：

- **二维数组**或者能转换为二维数组的**嵌套列表**。
- **字典**：字典中的每对“键-值”将成为DataFrame对象的一列。值可以是一维数组、列表或Series对象。

当未指定索引标签index和columns时，采用整数位置索引。

```
stud_dict = {'name': ['张三', '李四'], 'sex': ['男', '女'], 'major': ['大数据', '应用数学']}  
df4 = pd.DataFrame(stud_dict)  
df5 = pd.DataFrame(stud_dict, index=['r1', 'r2'])
```

print(df4)

	name	sex	major
0	张三	男	大数据
1	李四	女	应用数学

print(df5)

	name	sex	major
r1	张三	男	大数据
r2	李四	女	应用数学

创建DataFrame对象：将 DataFrame对象转换为其他格式的数据

DataFrame对象提供了一系列to_*()方法将数据转换为其他格式，例如to_dict(), to_csv(), to_json(), to_excel(), to_record(), to_numpy(), to_string(), to_sql()。

to_dict()方法将DataFrame对象转换为字典，它的orient参数决定字典元素的类型：

<code>print(df4)</code>	# 列表字典 <code>print(df4.to_dict(orient="list"))</code>	# 字典列表 <code>print(df4.to_dict(orient="records"))</code>
<pre> name sex major 0 张三 男 大数据 1 李四 女 应用数学 </pre>	<pre> {'name': ['张三', '李四'], 'sex': ['男', '女'], 'major': ['大数据', '应用数学']} </pre>	<pre> [{'name': '张三', 'sex': '男', 'major': '大数据'}, {'name': '李四', 'sex': '女', 'major': '应用数学'}] </pre>

to_records()方法可以将DataFrame对象转换为结构化数组，若其 index参数为True(默认值)，则其返回的数组中包含行索引数据：

```

print(df4.to_records())
print(df4.to_records().dtype)
print(df4.to_records(index=False))
print(df4.to_records(index=False).dtype)

[(0, '张三', '男', '大数据') (1, '李四', '女', '应用数学')]
(numpy.record, [('index', '<i8'), ('name', 'O'), ('sex', 'O'), ('major', 'O')])
[('张三', '男', '大数据') ('李四', '女', '应用数学')]
(numpy.record, [('name', 'O'), ('sex', 'O'), ('major', 'O')])

```

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

Series和 DataFrame提供了丰富的下标存取方法，除了直接使用**[]运算符**之外，还可以使用**.loc[]**、**.iloc[]**、**.at[]**、**.iat[]**等存取器存取其中的元素。

存取方法	说明
[col_label]	以单个标签作为下标，获取与标签对应的列，返回Series对象
[col_labels]	以标签列表作为下标，获取对应的多个列，返回DataFrame对象
[row_slice]	整数切片或标签切片，得到指定范围之内的行
[row_bool_array]	选择布尔数组中True对应的行
.get(col_label, default)	与字典的get()方法的用法相同
.at[index_label, col_label]	选择行标签和列标签对应的值，返回单个元素
.iat[index, col]	选择行编号和列编号对应的值，返回单个元素
.loc[index_label, col_label]	通过单个标签值、标签列表、标签数组、布尔数组、标签切片等选择指定行与列上的数据
.iloc[index, col]	通过单个整数值、整数列表、整数数组、布尔数组、整数切片选择指定行与列上的数据
.lookup(row_labels, col_labels)	选择行标签列表与列标签列表中每对标签对应的元素
.query()	通过表达式选择满足条件的行
.head()	获取头部n行数据
.tail()	获取尾部n行数据
.nlargest(n,columns)	按某些列值降序排列后，获取前n行
.nsmallest(n,columns)	按某些列值升序排列后，获取前n行

以下面的数据框为例，说明数据框中元素的存取。

```
import pandas as pd
import numpy as np
```

```
np.random.seed(0) # 固定随机数种子
```

```
A = np.random.randint(0, 10, (5, 3)) # 数据
```

```
columns = ["c1", "c2", "c3"] # 列索引
```

```
index = ["r1", "r2", "r3", "r4", "r5"] # 行索引
```

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

```
print(df)
```

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

[]运算符

通过[]运算符对DataFrame对象进行存取时，支持以下5种下标对象：

- 存取列 {
- 单个索引标签：获取标签对应的列，返回一个Series对象。
 - 多个索引标签：获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。
- 存取行 {
- 整数切片：以整数下标获取切片对应的行。
 - 标签切片：当使用标签作为切片时**包含终值**。
 - 布尔数组：获取数组中True对应的行。
 - 布尔DataFrame：将 DataFrame对象中False对应的元素设置为NaN。

<code>print(df)</code>	<code>print(df['c2']) # print(df.c2)</code>	<code>print(df[['c1', 'c3']])</code>																																																				
<table><tr><th></th><th>c1</th><th>c2</th><th>c3</th></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	<table><tr><td>r1</td><td>0</td></tr><tr><td>r2</td><td>7</td></tr><tr><td>r3</td><td>5</td></tr><tr><td>r4</td><td>7</td></tr><tr><td>r5</td><td>8</td></tr></table> <p>Name: c2, dtype: int32</p>	r1	0	r2	7	r3	5	r4	7	r5	8	<table><tr><th></th><th>c1</th><th>c3</th></tr><tr><td>r1</td><td>5</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>1</td></tr></table>		c1	c3	r1	5	3	r2	3	9	r3	3	2	r4	4	6	r5	8	1
	c1	c2	c3																																																			
r1	5	0	3																																																			
r2	3	7	9																																																			
r3	3	5	2																																																			
r4	4	7	6																																																			
r5	8	8	1																																																			
r1	0																																																					
r2	7																																																					
r3	5																																																					
r4	7																																																					
r5	8																																																					
	c1	c3																																																				
r1	5	3																																																				
r2	3	9																																																				
r3	3	2																																																				
r4	4	6																																																				
r5	8	1																																																				

[]运算符

通过[]运算符对DataFrame对象进行存取时，支持以下5种下标对象：

存取列

- 单个**索引标签**：获取标签对应的列，返回一个Series对象。
- 多个**索引标签**：获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。

存取行

- **整数切片**：以整数下标获取切片对应的行。
- **标签切片**：当使用标签作为切片时**包含终值**。
- **布尔数组**：获取数组中True对应的行。
- 布尔DataFrame: 将 DataFrame对象中False对应的元素设置为**NaN**。

<code>print(df)</code>	<code>print(df[2:4])</code>	<code>print(df['r2':'r4'])</code>	<code>print(df.c1>3)</code>	<code>print(df[df.c1>3])</code>																																																																																
<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr></table>		c1	c2	c3	r3	3	5	2	r4	4	7	6	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr></table>		c1	c2	c3	r2	3	7	9	r3	3	5	2	r4	4	7	6	<table><tr><td>r1</td><td>True</td></tr><tr><td>r2</td><td>False</td></tr><tr><td>r3</td><td>False</td></tr><tr><td>r4</td><td>True</td></tr><tr><td>r5</td><td>True</td></tr><tr><td colspan="2">Name: c1, dtype: bool</td></tr></table>	r1	True	r2	False	r3	False	r4	True	r5	True	Name: c1, dtype: bool		<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r4	4	7	6	r5	8	8	1
	c1	c2	c3																																																																																	
r1	5	0	3																																																																																	
r2	3	7	9																																																																																	
r3	3	5	2																																																																																	
r4	4	7	6																																																																																	
r5	8	8	1																																																																																	
	c1	c2	c3																																																																																	
r3	3	5	2																																																																																	
r4	4	7	6																																																																																	
	c1	c2	c3																																																																																	
r2	3	7	9																																																																																	
r3	3	5	2																																																																																	
r4	4	7	6																																																																																	
r1	True																																																																																			
r2	False																																																																																			
r3	False																																																																																			
r4	True																																																																																			
r5	True																																																																																			
Name: c1, dtype: bool																																																																																				
	c1	c2	c3																																																																																	
r1	5	0	3																																																																																	
r4	4	7	6																																																																																	
r5	8	8	1																																																																																	

[]运算符

通过[]运算符对DataFrame对象进行存取时，支持以下5种下标对象：

存取列

- 单个**索引标签**：获取标签对应的列，返回一个Series对象。
- 多个**索引标签**：获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。

存取行

- **整数切片**：以整数下标获取切片对应的行。
- **标签切片**：当使用标签作为切片时**包含终值**。
- **布尔数组**：获取数组中True对应的行。
- 布尔DataFrame: 将 DataFrame对象中False对应的元素设置为**NaN**。

print(df)	print(df>2)	print(df[df>2])																																																																								
<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>True</td><td>False</td><td>True</td></tr><tr><td>r2</td><td>True</td><td>True</td><td>True</td></tr><tr><td>r3</td><td>True</td><td>True</td><td>False</td></tr><tr><td>r4</td><td>True</td><td>True</td><td>True</td></tr><tr><td>r5</td><td>True</td><td>True</td><td>False</td></tr></table>		c1	c2	c3	r1	True	False	True	r2	True	True	True	r3	True	True	False	r4	True	True	True	r5	True	True	False	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>NaN</td><td>3.0</td></tr><tr><td>r2</td><td>3</td><td>7.0</td><td>9.0</td></tr><tr><td>r3</td><td>3</td><td>5.0</td><td>NaN</td></tr><tr><td>r4</td><td>4</td><td>7.0</td><td>6.0</td></tr><tr><td>r5</td><td>8</td><td>8.0</td><td>NaN</td></tr></table>		c1	c2	c3	r1	5	NaN	3.0	r2	3	7.0	9.0	r3	3	5.0	NaN	r4	4	7.0	6.0	r5	8	8.0	NaN
	c1	c2	c3																																																																							
r1	5	0	3																																																																							
r2	3	7	9																																																																							
r3	3	5	2																																																																							
r4	4	7	6																																																																							
r5	8	8	1																																																																							
	c1	c2	c3																																																																							
r1	True	False	True																																																																							
r2	True	True	True																																																																							
r3	True	True	False																																																																							
r4	True	True	True																																																																							
r5	True	True	False																																																																							
	c1	c2	c3																																																																							
r1	5	NaN	3.0																																																																							
r2	3	7.0	9.0																																																																							
r3	3	5.0	NaN																																																																							
r4	4	7.0	6.0																																																																							
r5	8	8.0	NaN																																																																							

.loc[]和.iloc[]存取器

.loc[]的下标对象是一个元组，其中的两个元素分别与DataFrame的两个轴相对应。若下标不是元组，则该下标对应第0轴（即获取行）。每个轴的下标对象都支持**单个标签**、**标签列表**、**标签切片**以及**布尔数组**。

如果是获取单行（ $1 < \text{长度} \leq \text{数据框列数}$ ）或单列（ $1 < \text{长度} \leq \text{数据框行数}$ ），则返回一个Series对象。如果是获取多行多列中的元素，则返回DataFrame对象。

<code>print(df)</code>	<i># r2行</i> <code>print(df.loc['r2'])</code>	<i># r2和r4行</i> <code>print(df.loc[['r2', 'r4']])</code>	<i># 切片: r2~r4行</i> <code>print(df.loc['r2':'r4'])</code>																																																										
<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	<table><tr><td>c1</td><td>3</td></tr><tr><td>c2</td><td>7</td></tr><tr><td>c3</td><td>9</td></tr></table> <p>Name: r2, dtype: int32</p>	c1	3	c2	7	c3	9	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr></table>		c1	c2	c3	r2	3	7	9	r4	4	7	6	<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr></table>		c1	c2	c3	r2	3	7	9	r3	3	5	2	r4	4	7	6
	c1	c2	c3																																																										
r1	5	0	3																																																										
r2	3	7	9																																																										
r3	3	5	2																																																										
r4	4	7	6																																																										
r5	8	8	1																																																										
c1	3																																																												
c2	7																																																												
c3	9																																																												
	c1	c2	c3																																																										
r2	3	7	9																																																										
r4	4	7	6																																																										
	c1	c2	c3																																																										
r2	3	7	9																																																										
r3	3	5	2																																																										
r4	4	7	6																																																										

.loc[]和.iloc[]存取器

df	#单个元素r2行c3列 df.loc['r2', 'c3']	# r2行, c1、 c3列 df.loc['r2', ['c1', 'c3']]	# r2行, c1~c3列 df.loc['r2', 'c1':'c3']
<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	9	<pre> c1 3 c3 9 Name: r2, dtype: int32 </pre>	<pre> c1 3 c2 7 c3 9 Name: r2, dtype: int32 </pre>

# r2~r4行、 c3列 df.loc['r2':'r4', 'c3']	# r2~r4行、 c2~c3列 df.loc['r2':'r4', 'c2':'c3']
<pre> r2 9 r3 2 r4 6 Name: c3, dtype: int32 </pre>	<pre> c2 c3 r2 7 9 r3 5 2 r4 7 6 </pre>

.loc[]和.iloc[]存取器

df	#r2行 df.loc['r2', :]	# r2~r4行 df.loc['r2':'r4', :]	# c1列 print(df.loc[:, 'c1'])
<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	<pre> c1 3 c2 7 c3 9 Name: r2, dtype: int32 </pre>	<pre> c1 c2 c3 r2 3 7 9 r3 3 5 2 r4 4 7 6 </pre>	<pre> r1 5 r2 3 r3 3 r4 4 r5 8 Name: c1, dtype: int32 </pre>

# c1、c3列 df.loc[:, ['c1', 'c3']]	# 所有行、所有列 df.loc[:, :]	# c2>5的行 df.loc[df.c2>5]	# c2>5的行, c2、c3列 df.loc[df.c2>5, 'c2':'c3']
<pre> c1 c3 r1 5 3 r2 3 9 r3 3 2 r4 4 6 r5 8 1 </pre>	<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	<pre> c1 c2 c3 r2 3 7 9 r4 4 7 6 r5 8 8 1 </pre>	<pre> c2 c3 r2 7 9 r4 7 6 r5 8 1 </pre>

.loc[]和.iloc[]存取器

.iloc[]的和.loc[]类似，不过它使用整数下标。

<code>print(df)</code>	# 第2行 <code>df.loc[1]</code>	# 第2行和第4行 <code>df.loc[[1, 3]]</code>	# 切片:第2~4行 <code>df.loc[1 : 4]</code>
<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	<pre> c1 3 c2 7 c3 9 Name: r2, dtype: int32 </pre>	<pre> c1 c2 c3 r2 3 7 9 r4 4 7 6 </pre>	<pre> c1 c2 c3 r2 3 7 9 r3 3 5 2 r4 4 7 6 </pre>

.loc[]和.iloc[]存取器

df	#单个元素第2行3列 df.iloc[1, 2]	第2行, 第1、3列 df.iloc[1, [0, 2]]	# 第2行, 第1~3列 df.iloc[1, 0:3])
<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	9	<pre> c1 3 c3 9 Name: r2, dtype: int32 </pre>	<pre> c1 3 c2 7 c3 9 Name: r2, dtype: int32 </pre>

# 第2~4行、第3列 df.iloc[1:4, 2])	# 第2~4行、第2~3列 df.iloc[1:4, 1:3])
<pre> r2 9 r3 2 r4 6 Name: c3, dtype: int32 </pre>	<pre> c2 c3 r2 7 9 r3 5 2 r4 7 6 </pre>

.loc[]和.iloc[]存取器

df	# 第2行 df.iloc[1, :]	# 第2~4行 df.iloc[1:4, :]	# 第1列 df.iloc[:, 0]
<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	<pre> c1 3 c2 7 c3 9 Name: r2, dtype: int32 </pre>	<pre> c1 c2 c3 r2 3 7 9 r3 3 5 2 r4 4 7 6 </pre>	<pre> r1 5 r2 3 r3 3 r4 4 r5 8 Name: c1, dtype: int32 </pre>

# 第1、3列 df.iloc[:, [0, 2]]	# 所有行和列 df.iloc[:, :]	# 第2列>5的行 df.iloc[df.c2.values>5]	# 第2列>5的行, 第2、3列 df.iloc[df.c2.values>5, 1:3]
<pre> c1 c3 r1 5 3 r2 3 9 r3 3 2 r4 4 6 r5 8 1 </pre>	<pre> c1 c2 c3 r1 5 0 3 r2 3 7 9 r3 3 5 2 r4 4 7 6 r5 8 8 1 </pre>	<pre> c1 c2 c3 r2 3 7 9 r4 4 7 6 r5 8 8 1 </pre>	<pre> c2 c3 r2 7 9 r4 7 6 r5 8 1 </pre>

获取单个值

`.at[]`和`.iat[]`分别使用标签和整数下标获取单个值。

<code>print(df)</code>	<code>df.at['r2','c3']</code>	<code>df.iat[1,2]</code>																								
<table><tr><td></td><td>c1</td><td>c2</td><td>c3</td></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	9	9
	c1	c2	c3																							
r1	5	0	3																							
r2	3	7	9																							
r3	3	5	2																							
r4	4	7	6																							
r5	8	8	1																							

多级标签的存取

`.loc[]`和`.at[]`的下标可以指定多级索引中每级索引上的标签。这时多级索引轴对应的下标是一个下标元组，该元组中的每个元素与索引中的每级索引对应。

指定了第0级和第1级行索引标签

缺失第1级行索引标签，自动转换为元组 ('10-30', `slice(None)`)

缺失第0级行索引标签，使用`np.s_`对象创建元组 (`slice(None)`, 'Slope')

`df_soil.loc[('10-30','Slope'),
['pH','Ca']]`

```
Measures
pH    5.2825
Ca     9.515
Name: (10-30, Slope),
dtype: object
```

`df_soil.loc['10-30',
['pH','Ca']]`

```
Measures      pH      Ca
Contour
Depression    4.8800  7.5475
Slope         5.2825  9.5150
Top           4.8500 10.2375
```

`df_soil.loc[np.s_[:,'Slope'],
['pH','Ca']]`

```
Measures      pH      Ca
Depth Contour
0-10  Slope    5.5075 12.2475
10-30  Slope    5.2825  9.5150
```

	Measures	pH	Dens	Ca	Conduc	Date	Name
	Depth Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

多级标签的存取

缺失的索引标签也可以直接用`slice(None)`指定，表示本级索引的所有标签。

直接指定第1级行索引
标签为 `slice(None)`

直接指定第0级行索引
标签为 `slice(None)`

```
df_soil.loc[('10-30', slice(None)), ['pH', 'Ca']]
```

Measures	pH	Ca
Contour		
Depression	4.8800	7.5475
Slope	5.2825	9.5150
Top	4.8500	10.2375

```
df_soil.loc[(slice(None), 'Slope'), ['pH', 'Ca']]
```

Measures	pH	Ca
Depth Contour		
0-10 Slope	5.5075	12.2475
10-30 Slope	5.2825	9.5150

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth Contour							
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

head()方法和tail()方法

head()方法和tail()方法分别用于获取数据框头部n行数据和末尾n行数据。

df	df.head(2)	df.tail(2)																																																
<table><tr><th></th><th>c1</th><th>c2</th><th>c3</th></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr><tr><td>r3</td><td>3</td><td>5</td><td>2</td></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	r3	3	5	2	r4	4	7	6	r5	8	8	1	<table><tr><th></th><th>c1</th><th>c2</th><th>c3</th></tr><tr><td>r1</td><td>5</td><td>0</td><td>3</td></tr><tr><td>r2</td><td>3</td><td>7</td><td>9</td></tr></table>		c1	c2	c3	r1	5	0	3	r2	3	7	9	<table><tr><th></th><th>c1</th><th>c2</th><th>c3</th></tr><tr><td>r4</td><td>4</td><td>7</td><td>6</td></tr><tr><td>r5</td><td>8</td><td>8</td><td>1</td></tr></table>		c1	c2	c3	r4	4	7	6	r5	8	8	1
	c1	c2	c3																																															
r1	5	0	3																																															
r2	3	7	9																																															
r3	3	5	2																																															
r4	4	7	6																																															
r5	8	8	1																																															
	c1	c2	c3																																															
r1	5	0	3																																															
r2	3	7	9																																															
	c1	c2	c3																																															
r4	4	7	6																																															
r5	8	8	1																																															

query()方法

当需要根据一定的条件对行进行过滤时，通常可以先创建一个布尔数组，使用该数组获取True对应的行，例如下面的程序获得pH值大于5、Ca含量小于11%的行。

由于Python中无法自定义not、and和or等关键字的行为，因此需要改用~、&、|等位运算符。然而这些运算符的优先级比比较运算符要高，因此需要用括号将比较运算括起来。

```
(df_soil.pH>5) & (df_soil.Ca<11)
```

```
df_soil[(df_soil.pH>5) & (df_soil.Ca<11)]
```

Depth	Contour		Measures	pH	Dens	Ca	Conduc	Date	Name	
0-10	Depression	True	Depth	Contour						
	Slope	False	0-10	Depression	5.3525	0.9775	10.685	1.4725	2015-05-26	Lois
	Top	False	10-30	Slope	5.2825	1.3475	9.515	4.9100	2015-02-06	Diana
10-30	Depression	False								
	Slope	True								
	Top	False								
dtype: bool										

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

query()方法

使用 query()方法可以简化上述程序：

```
df_soil.query("pH>5 and Ca<11")
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.685	1.4725	2015-05-26	Lois
10-30	Slope	5.2825	1.3475	9.515	4.9100	2015-02-06	Diana

query()方法的参数是一个运算表达式字符串。其中可以使用not、and和or等关键字进行向量布尔运算，表达式中的变量名表示与其对应的列。如果希望在表达式中使用其他全局或局域变量的值，可以在变量名之前添加@。例如：

```
a=5
```

```
b=11
```

```
print(df_soil.query("pH>@a and Ca<@b"))
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

```
df = pd.DataFrame({'population': [59000000, 65000000, 434000, 434000, 434000, 337000,
                                   11300, 11300, 11300],
                   'GDP': [1937894, 2583560, 12011, 4520, 12128, 17036, 182, 38, 311],
                   'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN', 'IS', 'NR', 'TV', 'AI'] },
                  index=['Italy', 'France', 'Malta', 'Maldives', 'Brunei', 'Iceland',
                        'Nauru', 'Tuvalu', 'Anguilla'])
```

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

`nlargest(n, columns) / nsmallest(n, columns)`方法

根据指定列降序/升序排列后，返回数据框的前n行。

`nlargest(n, columns)`方法等同于`df.sort_values(columns, ascending=False).head(n)`

`nsmallest(n, columns)`方法等同于`df.sort_values(columns, ascending=True).head(n)`

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

按population降序排列后的前3行
`print(df.nlargest(3, 'population'))`

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Malta	434000	12011	MT

nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

如果population有相同的行，保留最后一个行
print(df.nlargest(3, 'population', keep='last'))

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Brunei	434000	12128	BN

如果population有相同的行，保留所有相同行
print(df.nlargest(3, 'population', keep='all'))

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN

nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

按 **population** 降序、**GDP** 降序排列后的前3行
print(df.nlargest(3, ['population', 'GDP']))

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Brunei	434000	12128	BN

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

pandas常用的文件输入函数

函数名	说明
<code>pd.read_csv()</code>	从CSV 格式的文本文件读取数据至DataFrame
<code>pd.read_excel()</code>	从Excel文件读入数据至DataFrame
<code>pd.HDFStore()</code>	使用HDF5文件读数据至DataFrame
<code>pd.read_sql()</code>	从SQL数据库的查询结果载入数据至DataFrame
<code>pd.read_pickle()</code>	读入Pickle序列化之后的数据至DataFrame

pandas常用的文件输出函数

函数名	说明
<code>DataFrame.to_csv()</code>	DataFrame数据写入CSV 格式的文本文件
<code>DataFrame.to_excel()</code>	DataFrame数据写入Excel文件
<code>DataFrame.to_hdf()</code>	DataFrame数据写入HDF5文件
<code>DataFrame.to_sql()</code>	DataFrame数据写入SQL数据库
<code>DataFrame.to_pickle()</code>	DataFrame数据Pickle序列化并保存至文件

CSV 文件的读写

`read_csv()`从文本文件读入数据，它的可选参数非常多，下面只简要介绍一些常用参数：

- `sep`参数指定数据的分隔符号，可以使正则表达式，默认值为逗号。有时CSV 文件为了便于阅读，在逗号之后添加了一些空格以对齐每列的数据。如果希望忽略这些空格，可以将`skipinitialspace`参数设置为True。
- 如果数据使用空格或制表符分隔，可以不设置`sep`参数，而将`delim_whitespace`参数设置为True。
- 默认情况下第一行文本被作为列索引标签，如果数据文件中没有保存列名的行，可以设置`header`参数为0。
- 如果数据文件之前包含一些说明行，可以使用`skiprows`参数指定数据开始的行号。

CSV 文件的读写

`read_csv()`从文本文件读入数据，它的可选参数非常多，下面只简要介绍一些常用参数：

- `na_values`、`true_values` 和 `false_values` 等参数分别指定NaN、True和False对应的字符串列表。
- 如果希望将子字符串转换为时间，可以使用`parse_dates`指定转换为时间的列。
- 如果数据中包含中文，可以使用`encoding`参数指定文件的编码，例如utf-8、gbk等。指定编码之后得到的字符串列为Unicode字符串。
- 可以使用`usecols`参数指定需要读入的列。
- 当文件很大时，可以用`chunksize`参数指定一次读入的行数。当使用`chunksize`时，`read_csv()`返回一个迭代器。
- 当文件名包含中文时，需要使用Unicode字符串指定文件名。

CSV 文件的读写

例如：从csv文件读取土壤监测数据

	A	B	C	D	E	F	G	H
1	Depth	Contour	pH	Dens	Ca	Conduc	Date	Name
2	0-10	Depression	5.3525	0.9775	10.685	1.4725	2015/5/26 0:00	Lois
3	0-10	Slope	5.5075	1.05	12.2475	2.05	2015/4/30 0:00	Roy
4	0-10	Top	5.3325	1.0025	13.385	1.3725	2015/5/21 0:00	Roy
5	10-30	Depression	4.88	1.3575	7.5475	5.48	2015/3/21 0:00	Lois
6	10-30	Slope	5.2825	1.3475	9.515	4.91	2015/2/6 0:00	Diana
7	10-30	Top	4.85	1.3325	10.2375	3.5825	2015/4/11 0:00	Diana

```
df_soil = pd.read_csv("data/Soils-simple.csv", index_col=[0, 1], parse_dates=["Date"])
```

```
df_soil.columns.name = "Measures" # 设置列索引名
```

```
print(df_soil)
```

通过index_col参数指定第0和第1列为行索引，用 parse_dates参数指定进行日期转换的列，在指定列时可以使用列的序号(是文件中的列序号)或列名，例如这里也可以使用parse_dates=[6]。

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

CSV 文件的读写

例如：从csv文件读取空气质量数据

	A	B	C	D	E	F	G	H	I	J	K	L
1	时间	类型	城市	监测点	AQI	质量等级	Level	PM2.5	PM2.5_24h	PM10	pm10_24h	CO
2	2014/6/1 0:00	国控	北京		165	中度污染	四级	70	105	72	199	0.729
3	2014/6/1 0:00	国控	北京	万寿西宫	90	良		67	111	0	218	0.8
4	2014/6/1 0:00	国控	北京	定陵	0			0	88	0	130	0
5	2014/6/1 0:00	国控	北京	东四	95	良		71	110	0	0	0.9
6	2014/6/1 0:00	国控	北京	天坛	99	良		74	112	0	214	0.7

```
df = pd.read_csv(
    u"data/aqi/上海市_201406.csv",
    encoding="utf-8-sig", # 文件编码
    usecols=[u"时间", u"监测点", "AQI", "PM2.5", "PM10"], # 只读入这些列
    na_values=['-', '—'], # 这些字符串表示缺失数据
    parse_dates=[0]) # 第一列为时间列

print(df.count())
print(df.dtypes)
print(df.head(5))
```

一次读入所有数据
至一个数据框中。

时间	6999	时间	datetime64[ns]	时间	监测点	AQI	PM2.5	PM10
监测点	6306	监测点	object	0	2014-06-01	NaN	63	31 45.0
AQI	6999	AQI	int64	1	2014-06-01	普陀	50	35 50.0
PM2.5	6999	PM2.5	int64	2	2014-06-01	十五厂	54	37 58.0
PM10	6835	PM10	float64	3	2014-06-01	虹口	0	0 0.0
dtype: int64		dtype: object		4	2014-06-01	徐汇上师大	55	39 54.0

CSV 文件的读写

```

df_list = [] # 数据框列表
for df in pd.read_csv(
    u"data/aqi/上海市_201406.csv",
    encoding="utf-8-sig", # 文件编码
    chunksize=100, # 一次读入的行数
    usecols=[u"时间", u"监测点", "AQI", "PM2.5", "PM10"], # 只读入这些列
    na_values=['-', '—'], # 这些字符串表示缺失数据
    parse_dates=[0]): # 第一列为时间列
    df_list.append(df) # 在这里处理数据

print(df_list[0].count())
print(df_list[0].dtypes)
print(df_list[0].head(5))

```

一次读入100行数据到一个数据框中，循环读取所有数据到多个数据框中。

时间	100
监测点	90
AQI	100
PM2.5	100
PM10	98
dtype:	int64

时间	datetime64[ns]
监测点	object
AQI	int64
PM2.5	int64
PM10	float64
dtype:	object

	时间	监测点	AQI	PM2.5	PM10
0	2014-06-01	NaN	63	31	45.0
1	2014-06-01	普陀	50	35	50.0
2	2014-06-01	十五厂	54	37	58.0
3	2014-06-01	虹口	0	0	0.0
4	2014-06-01	徐汇上师大	55	39	54.0

CSV 文件的读写

to_csv()方法能将数据框数据保存至csv文件。例如，保存空气质量数据至csv文件：

```
path = u"data/aqi/上海市_201406_tmp.csv"
df.to_csv(path, # 文件名
          sep=',', # 分隔符
          header=True, # 列索引作为表头输出
          index=False, # 行索引不输出
          na_rep='-', # 缺失值的替代字符
          encoding='utf-8-sig') # 字符编码。
```

#如果用utf-8，记事本能正确显示中文，但Excel打开中文显示乱码

	A	B	C	D	E	F
1	时间	监测点	AQI	PM2.5	PM10	
2	2014/6/1 0:00	-	63	31	45	
3	2014/6/1 0:00	普陀	50	35	50	
4	2014/6/1 0:00	十五厂	54	37	58	
5	2014/6/1 0:00	虹口	0	0	0	
6	2014/6/1 0:00	徐汇上师大	55	39	54	
7	2014/6/1 0:00	杨浦四漂	61	33	72	
8	2014/6/1 0:00	青浦淀山湖	23	9	9	
9	2014/6/1 0:00	静安监测站	54	34	58	
10	2014/6/1 0:00	浦东川沙	43	30	35	
11	2014/6/1 0:00	浦东新区监测站	49	34	37	
12	2014/6/1 0:00	浦东张江	46	32	33	

Excel文件的读写

`read_excel()`方法从Excel文件读取数据，参数与`read_csv()`类似。

但需要安装xlrd模块：C:\WINDOWS\system32>python -m pip install xlrd

```
管理员: 命令提示符
Microsoft Windows [版本 10.0.18362.959]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\WINDOWS\system32>python -m pip install xlrd
Collecting xlrd
  Downloading xlrd-1.2.0-py2.py3-none-any.whl (103 kB)
    |#####| 102 kB 172 kB/s eta 0:00
    |#####| 103 kB 172 kB/s
Installing collected packages: xlrd
Successfully installed xlrd-1.2.0
WARNING: You are using pip version 20.1.1; however, version 20.2 is available.
You should consider upgrading via the 'C:\Program Files\Python38\python.exe -m pip install --up
grade pip' command.
C:\WINDOWS\system32>
```

例如：从Excel文件读取空气质量数据

```
df = pd.read_excel(
    u"data/aqi/上海市_201406.xlsx",
    usecols=[u"时间", u"监测点", "AQI", "PM2.5", "PM10"], # 只读入这些列
    na_values=['-', '—'], # 这些字符串表示缺失数据
    parse_dates=[0]) # 第一列为时间列
print(df.head(5))
```

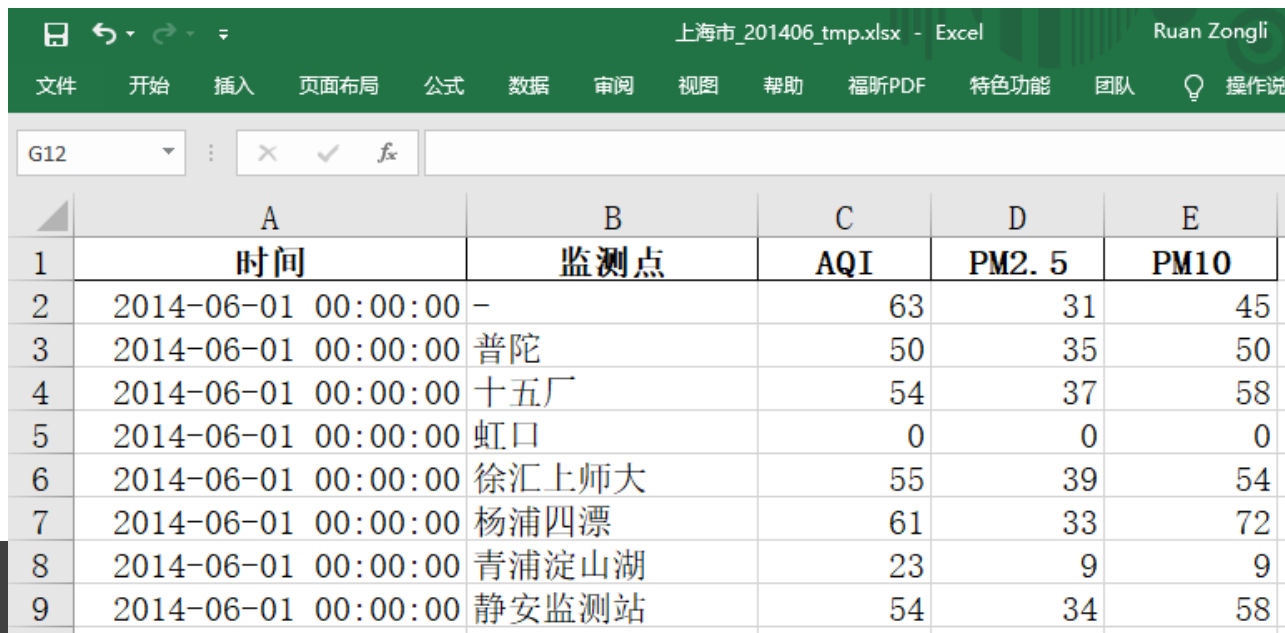
	时间	监测点	AQI	PM2.5	PM10
0	2014-06-01	NaN	63	31	45.0
1	2014-06-01	普陀	50	35	50.0
2	2014-06-01	十五厂	54	37	58.0
3	2014-06-01	虹口	0	0	0.0
4	2014-06-01	徐汇上师大	55	39	54.0

Excel 文件的读写

to_excel()方法能将数据框数据保存至excel文件。但需要安装openpyxl模块：

```
C:\WINDOWS\system32>python -m pip install openpyxl
```

```
path = u"data/aqi/上海市_201406_tmp2.xlsx"
df.to_excel(path, # 文件名
            header=True, # 列索引作为表头输出
            index=False, # 行索引不输出
            na_rep='-') # 缺失值的替代字符
```



The screenshot shows an Excel spreadsheet titled "上海市_201406_tmp.xlsx" with the following data:

	A	B	C	D	E
1	时间	监测点	AQI	PM2.5	PM10
2	2014-06-01 00:00:00	-	63	31	45
3	2014-06-01 00:00:00	普陀	50	35	50
4	2014-06-01 00:00:00	十五厂	54	37	58
5	2014-06-01 00:00:00	虹口	0	0	0
6	2014-06-01 00:00:00	徐汇上师大	55	39	54
7	2014-06-01 00:00:00	杨浦四漂	61	33	72
8	2014-06-01 00:00:00	青浦淀山湖	23	9	9
9	2014-06-01 00:00:00	静安监测站	54	34	58

读写数据库

to_sql()方法可以用于将DataFrame中数据写入SQL数据库。其语法如下：

```
DataFrame.to_sql(  
    name, # 数据库表名称  
    con, # SQLAlchemy ( sqlalchemy库) 的Engine对象, 或sqlite3的connect对象  
    if_exists = 'fail', # {'fail', 'replace', 'append'}, 记录存在时的处理方式  
    index=True, # 数据框的索引是否作为新列写入数据库, 新列名则由index_label参数指定  
    index_label=None,  
    chunksize=None,  
    dtype=None) # 用字典指定某些列对应的数据库字段类型, 例如dtype={"A": sqlalchemy.Integer()}
```

read_sql()方法可以用于从SQL数据库读数据, 返回DataFrame。其语法如下：

```
df=pd.read_sql(  
    sql, # SQL查询语句字符串  
    con, # SQLAlchemy的Engine对象, 或sqlite3的connect对象  
    index_col=None,  
    coerce_float=True, # 默认将数值的字符串转换为浮点数  
    parse_dates=None,  
    columns=None,  
    chunksize =None)
```

安装sqlalchemy库:

```
C:\WINDOWS\system32>python -m pip install sqlalchemy
```

读写数据库

例子：从csv文件读入数据至数据框，然后写入sqlite数据库，再从数据库查询读取。

```
import pandas as pd
import sqlalchemy
```

```
engine = sqlalchemy.create_engine('sqlite:///data/aqi/my_aqi.db')
```

```
# 数据库中aqi存在则删除它
```

```
try:
```

```
    engine.execute('DROP TABLE aqi')
```

```
except:
```

```
    pass
```

```
df = pd.read_csv(
    u"data/aqi/上海市_201406.csv",
    encoding="utf-8-sig", # 文件编码
    usecols=[u"时间", u"监测点", "AQI", "PM2.5", "PM10"], # 只读入这些列
    na_values=['-', '—'], # 这些字符串表示缺失数据
    parse_dates=[0]) # 第一列为时间列
```

读写数据库

例子：从csv文件读入数据至数据框，然后写入sqlite数据库，再从数据库查询读取。

重命名数据框的列索引

```
dict_name = {'时间': 'Time', '监测点': 'Position',
             'AQI': 'AQI', 'PM2.5': 'PM2_5', 'PM10': 'PM10'}
```

```
df = df.rename(columns=dict_name) # 重命名,返回数据框
```

```
print(df.head(2))
```

df写入数据库

```
df.to_sql(name='aqi', con=engine, if_exists='append', index=False)
```

从数据库查询，结果存入df_polluted

```
df_polluted = pd.read_sql(sql="select * from aqi where PM2_5>150", con=engine)
```

转换Time列的数据类型 (object->datetime64[D])

```
df_polluted['Time']=df_polluted['Time'].astype('datetime64[D])
```

```
print(df_polluted.head(2))
```

```
print(df_polluted.dtypes)
```

```
print(len(df_polluted))
```

	Time	Position	AQI	PM2_5	PM10
0	2014-06-01	NaN	63	31	45.0
1	2014-06-01	普陀	50	35	50.0
	Time	Position	AQI	PM2_5	PM10
0	2014-06-28	杨浦四漂	202	152	184.0
1	2014-06-28	青浦淀山湖	201	151	170.0

Time	datetime64[ns]
Position	object
AQI	int64
PM2_5	int64
PM10	float64
dtype:	object
32	

使用pickle序列化与反序列化

使用to_pickle()和 read_pickle()方法可以对DataFrame对象进行序列化和反序列化。Pickle是 Python特有的对象序列化格式，因此很难使用其他软件、程序设计语言读取Pickle化之后的数据，但是作为临时保存运算的中间结果还是很方便的。

```
df = pd.read_csv(  
    u"data/aqi/上海市_201406.csv",  
    encoding="utf-8-sig", # 文件编码  
    usecols=[u"时间", u"监测点", "AQI", "PM2.5", "PM10"], # 只读入这些列  
    na_values=['-', '—'], # 这些字符串表示缺失数据  
    parse_dates=[0]) # 第一列为时间列
```

```
path="data/aqi/my_aqi.pickle"  
df.to_pickle(path=path,protocol=4) #序列化  
df2=pd.read_pickle(filepath_or_buffer=path)# 反序列化  
print(df2.head(5))
```

	时间	监测点	AQI	PM2.5	PM10
0	2014-06-01	NaN	63	31	45.0
1	2014-06-01	普陀	50	35	50.0
2	2014-06-01	十五厂	54	37	58.0
3	2014-06-01	虹口	0	0	0.0
4	2014-06-01	徐汇上师大	55	39	54.0

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

数值运算函数

Series和DataFrame对象都支持NumPy的数组接口，因此可以直接使用NumPy提供的ufunc函数对它们进行运算。例如：

```
np.set_printoptions(2)
```

```
f'%.2f' % (np.mean(df_soil.pH)) # pH列的平均
```

```
np.mean(df_soil[["pH", "Ca"]], axis=0) # 列平均
```

```
np.mean(df_soil.loc[:, "pH":"Ca"], axis=0) # 同上，列平均
```

```
np.mean(df_soil.loc[:, "pH":"Ca"], axis=1) # 行平均
```

5.20

Measures

pH 5.20

Dens 1.18

Ca 10.60

dtype: float64

Depth Contour

0-10 Depression 5.67

Slope 6.27

Top 6.57

10-30 Depression 4.59

Slope 5.38

Top 5.47

dtype: float64

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

但pandas自身也提供各种运算方法，例如 max()、min()、mean()、std()等。这些函数都有如下3个常用参数：

- axis: 指定运算对应的轴。
- level: 指定运算对应的索引级别。
- skipna: 运算是否自动跳过NaN。

Measures	
pH	5.20
Dens	1.18
Ca	10.60
Conduc	3.14
dtype: float64	

pandas DataFrame控制台打印输出设置浮点数小数位数
pd.options.display.float_format = '{:.2f}'.format

或者

pd.options.display.float_format = lambda x:'%.2f'%x

print(df_soil.mean()) # 在第0轴上计算平均值，即每列的平均值

print(df_soil.mean(level=0)) # 第0级行索引上分组计算各列平均值

print(df_soil.mean(level=1)) # 在第1级行索引上分组计算各列平均值

print(df_soil.mean(axis=1)) # 第1轴上计算平均值，即每行的平均值

Measures	pH	Dens	Ca	Conduc
Depth				
0-10	5.40	1.01	12.11	1.63
10-30	5.00	1.35	9.10	4.66

Measures	pH	Dens	Ca	Conduc
Contour				
Depression	5.12	1.17	9.12	3.48
Slope	5.40	1.20	10.88	3.48
Top	5.09	1.17	11.81	2.48

Depth	Contour	
0-10	Depression	4.62
	Slope	5.21
	Top	5.27
10-30	Depression	4.82
	Slope	5.26
	Top	5.00
dtype: float64		

除了支持**加减乘除等运算符**之外，Pandas还提供了 `add()`、`sub()`、`mul()`、`div()`、`mod()`等与二元运算符对应的函数。这些函数可以通过**axis**、**level**和 **fill_value**等参数控制其运算行为，`fill_value`参数用于指定不存在的值或NaN时使用的默认值。

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

Depth	Contour	
0-10	Depression	6.35
	Slope	6.51
	Top	6.33
10-30	Depression	5.88
	Slope	6.28
	Top	5.85
Name: pH, dtype: float64		

`df_soil.pH + 1` # 单列+1

`df_soil.loc[:, ['pH', 'Ca']] * 0.5` # 两列*0.5

`df_soil.loc[(slice(None), 'Top'), ['pH', 'Ca']] * 0.5` # 第1级行索引为Top的行中两列*0.5

`df_soil.iloc[0:2, 0:2] * 0.5` # 头两行中前2列*0.5 (注意文本和日期类型不知支持该概运算)

Measures		pH	Ca
Depth	Contour		
0-10	Depression	2.68	5.34
	Slope	2.75	6.12
	Top	2.67	6.69
10-30	Depression	2.44	3.77
	Slope	2.64	4.76
	Top	2.42	5.12

Measures		pH	Ca
Depth	Contour		
0-10	Top	2.67	6.69
10-30	Top	2.42	5.12

Measures		pH	Dens
Depth	Contour		
0-10	Depression	2.68	0.49
	Slope	2.75	0.53

除了支持**加减乘除等运算符**之外，Pandas还提供了 `add()`、`sub()`、`mul()`、`div()`、`mod()`等与二元运算符对应的函数。这些函数可以通过**axis**、**level**和 **fill_value**等参数控制其运算行为，`fill_value`参数用于指定不存在的值或NaN时使用的默认值。

Depth	Contour	
0-10	Depression	9.62
	Slope	14.70
	Top	13.39
10-30	Depression	6.79
	Slope	11.42
	Top	10.24

dtype: float64

Depth	Contour	
0-10	Depression	9.62
	Slope	14.70
	Top	0.00
10-30	Depression	6.79
	Slope	11.42
	Top	0.00

dtype: float64

Measures	pH	Ca
Depth Contour		
0-10 Depression	2.68	21.37
Slope	2.75	24.50
Top	2.67	26.77
10-30 Depression	2.44	15.10
Slope	2.64	19.03
Top	2.42	20.48

```
s = pd.Series(dict(Depression=0.9, Slope=1.2))
df_soil.Ca.mul(s, level=1, fill_value=1)
df_soil.Ca.mul(s, level=1, fill_value=0)
s2 = pd.Series(dict(pH=0.5, Ca=2))
df_soil[['pH', 'Ca']].mul(s2) # 两列乘不同的系数
```

Measures	pH	Dens	Ca	Conduc	Date	Name
Depth Contour						
0-10 Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30 Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

字符串处理

Series对象提供了大量的字符串处理方法，例如upper(), split(), len(), cat(), +运算符, *运算符, replace(), map()等。由于数量众多，因此Pandas使用了一个类似名称空间的对象str来包装这些字符串相关的方法。

```
s = pd.Series(['Python Programming', 'thank you!', '我爱UPC'])
print(s.str.upper()) # 每个元素转化为大写字母
print(s.str.len()) # Unicode字符数(实际字符个数)
print(s.str.encode('GBK').str.len()) # 编码后的字节数(一个汉字占用2字节)
print(s.str.encode('UTF-8').str.len()) # 编码后的字节数(一个汉字占用3字节)
print(s.str.split(' ')) # 每个元素分割成列表
# 也可以在序列上使用map()函数,它将针对每个元素运算的函数运用到整个序列之上。
print(s.map(lambda x: x.capitalize())) # 每个元素的首字母大写
```

```
0    PYTHON PROGRAMMING
1           THANK YOU!
2           我爱UPC
dtype: object
```

```
0    18
1    10
2     5
dtype: int64
```

```
0    18
1    10
2     7
dtype: int64
```

```
0    18
1    10
2     9
dtype: int64
```

```
0    [Python, Programming]
1    [thank, you!]
2    [我爱UPC]
dtype: object
```

```
0    Python programming
1    Thank you!
2    我爱upc
dtype: object
```

与NaN相关的函数

Pandas使用NaN表示缺失的数据。与NaN相关的函数常用的有：

- **where()**方法：不满足条件的元素设置为NaN;
- **isnull()**和 **notnull()**方法：用于判断元素值是否为NaN;
- **count()**方法：返回每行或每列的非NaN元素的个数;
- **dropna()**方法：删除包含NaN的行或列;
- **ffill()**、**bfill()**和**interpolate()**方法：分别用NaN之前的元素值，之后的元素值，或前后元素的插值填充该NaN元素;
- **fillna()**方法：将NaN值填充为value参数指定的值。

与NaN相关的函数：where()方法

where()方法将**不满足**指定条件的元素设置为NaN，返回Series或DataFrame。由于整数列无法使用NaN，因此如果整数类型的列出现缺失数据，则会被自动转换为浮点数类型。

```
s = pd.Series([2, 0, 4, 8, 1])
print(s)
print(s > 1)
s_nan = s.where(s > 1) # 小于等于1的元素置为NaN, 返回一个Series
print(s_nan)
```

s	s > 1	s_nan = s.where(s > 1)
0 2	0 True	0 2.0
1 0	1 False	1 NaN
2 4	2 True	2 4.0
3 8	3 True	3 8.0
4 1	4 False	4 NaN
dtype: int64	dtype: bool	dtype: float64

与NaN相关的函数：where()方法

where()方法将**不满足**指定条件的元素设置为NaN，返回Series或DataFrame。
由于整数列无法使用NaN，因此如果整数类型的列出现缺失数据，则会被自动转换为浮点数类型。

```
np.random.seed(41)
```

```
A = np.random.randint(0, 10, (10, 3))
```

```
df_int = pd.DataFrame(data=A, columns=list("ABC"))
```

```
df_int["A"] += 10 # 修改第一列元素：各元素+10
```

```
df_nan = df_int.where(df_int > 2) # 小于等于2的元素全部置为NaN，返回一个DataFrame
```

df_int				df_nan = df_int.where(df_int > 2)			
	A	B	C		A	B	C
0	10	3	2	0	10	3.0	NaN
1	10	1	3	1	10	NaN	3.0
2	19	7	5	2	19	7.0	5.0
3	18	3	3	3	18	3.0	3.0
4	12	6	0	4	12	6.0	NaN
5	14	6	9	5	14	6.0	9.0
6	13	8	4	6	13	8.0	4.0
7	17	6	1	7	17	6.0	NaN
8	15	2	1	8	15	NaN	NaN
9	15	3	2	9	15	3.0	NaN

与NaN相关的函数：isnull()和notnull()方法

isnull()和 notnull()方法用于判断元素值是否为NaN，返回布尔型Series或DataFrame。

s_nan	s_nan.isnull()
0 2.0	0 False
1 NaN	1 True
2 4.0	2 False
3 8.0	3 False
4 NaN	4 True
dtype: float64	dtype: bool

df_nan	df_nan.isnull()		
	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

与NaN相关的函数：count()方法

count()方法返回每行或非NaN元素的个数，可以指定axis参数表示在0轴（对应列）或1轴（对应行）上统计。

s_nan	s_nan.count()
0 2.0	3
1 NaN	
2 4.0	
3 8.0	
4 NaN	
dtype: float64	

df_nan	df_nan.count()	df_nan.count(axis=1)
	A 10	0 2
0 10 3.0 NaN	B 8	1 2
1 10 NaN 3.0	C 5	2 3
2 19 7.0 5.0	dtype: int64	3 3
3 18 3.0 3.0		4 2
4 12 6.0 NaN		5 3
5 14 6.0 9.0		6 3
6 13 8.0 4.0		7 2
7 17 6.0 NaN		8 1
8 15 NaN NaN		9 2
9 15 3.0 NaN		dtype: int64

与NaN相关的函数：dropna()方法

对于包含NaN元素的数据，最简单的办法就是调用dropna()以删除包含NaN的行或列，当全部使用默认参数时，将删除包含NaN的所有行。

可以通过thresh参数指定NaN个数的阈值，删除所有NaN个数大于等于该阈值的行。

```
print(df_nan.dropna()) # 删除包含NaN的行
print(df_nan.dropna(axis=1)) # 删除包含NaN的列
print(df_nan.dropna(thresh=2)) # 删除包含大于等于2个NaN的行
```

df_nan	df_nan.dropna()	df_nan.dropna(axis=1)	df_nan.dropna(thresh=2)																																																																																																																												
<table><tr><td></td><td>A</td><td>B</td><td>C</td></tr><tr><td>0</td><td>10</td><td>3.0</td><td>NaN</td></tr><tr><td>1</td><td>10</td><td>NaN</td><td>3.0</td></tr><tr><td>2</td><td>19</td><td>7.0</td><td>5.0</td></tr><tr><td>3</td><td>18</td><td>3.0</td><td>3.0</td></tr><tr><td>4</td><td>12</td><td>6.0</td><td>NaN</td></tr><tr><td>5</td><td>14</td><td>6.0</td><td>9.0</td></tr><tr><td>6</td><td>13</td><td>8.0</td><td>4.0</td></tr><tr><td>7</td><td>17</td><td>6.0</td><td>NaN</td></tr><tr><td>8</td><td>15</td><td>NaN</td><td>NaN</td></tr><tr><td>9</td><td>15</td><td>3.0</td><td>NaN</td></tr></table>		A	B	C	0	10	3.0	NaN	1	10	NaN	3.0	2	19	7.0	5.0	3	18	3.0	3.0	4	12	6.0	NaN	5	14	6.0	9.0	6	13	8.0	4.0	7	17	6.0	NaN	8	15	NaN	NaN	9	15	3.0	NaN	<table><tr><td></td><td>A</td><td>B</td><td>C</td></tr><tr><td>2</td><td>19</td><td>7.0</td><td>5.0</td></tr><tr><td>3</td><td>18</td><td>3.0</td><td>3.0</td></tr><tr><td>5</td><td>14</td><td>6.0</td><td>9.0</td></tr><tr><td>6</td><td>13</td><td>8.0</td><td>4.0</td></tr></table>		A	B	C	2	19	7.0	5.0	3	18	3.0	3.0	5	14	6.0	9.0	6	13	8.0	4.0	<table><tr><td>0</td><td>10</td></tr><tr><td>1</td><td>10</td></tr><tr><td>2</td><td>19</td></tr><tr><td>3</td><td>18</td></tr><tr><td>4</td><td>12</td></tr><tr><td>5</td><td>14</td></tr><tr><td>6</td><td>13</td></tr><tr><td>7</td><td>17</td></tr><tr><td>8</td><td>15</td></tr><tr><td>9</td><td>15</td></tr></table>	0	10	1	10	2	19	3	18	4	12	5	14	6	13	7	17	8	15	9	15	<table><tr><td></td><td>A</td><td>B</td><td>C</td></tr><tr><td>0</td><td>10</td><td>3.0</td><td>NaN</td></tr><tr><td>1</td><td>10</td><td>NaN</td><td>3.0</td></tr><tr><td>2</td><td>19</td><td>7.0</td><td>5.0</td></tr><tr><td>3</td><td>18</td><td>3.0</td><td>3.0</td></tr><tr><td>4</td><td>12</td><td>6.0</td><td>NaN</td></tr><tr><td>5</td><td>14</td><td>6.0</td><td>9.0</td></tr><tr><td>6</td><td>13</td><td>8.0</td><td>4.0</td></tr><tr><td>7</td><td>17</td><td>6.0</td><td>NaN</td></tr><tr><td>9</td><td>15</td><td>3.0</td><td>NaN</td></tr></table>		A	B	C	0	10	3.0	NaN	1	10	NaN	3.0	2	19	7.0	5.0	3	18	3.0	3.0	4	12	6.0	NaN	5	14	6.0	9.0	6	13	8.0	4.0	7	17	6.0	NaN	9	15	3.0	NaN
	A	B	C																																																																																																																												
0	10	3.0	NaN																																																																																																																												
1	10	NaN	3.0																																																																																																																												
2	19	7.0	5.0																																																																																																																												
3	18	3.0	3.0																																																																																																																												
4	12	6.0	NaN																																																																																																																												
5	14	6.0	9.0																																																																																																																												
6	13	8.0	4.0																																																																																																																												
7	17	6.0	NaN																																																																																																																												
8	15	NaN	NaN																																																																																																																												
9	15	3.0	NaN																																																																																																																												
	A	B	C																																																																																																																												
2	19	7.0	5.0																																																																																																																												
3	18	3.0	3.0																																																																																																																												
5	14	6.0	9.0																																																																																																																												
6	13	8.0	4.0																																																																																																																												
0	10																																																																																																																														
1	10																																																																																																																														
2	19																																																																																																																														
3	18																																																																																																																														
4	12																																																																																																																														
5	14																																																																																																																														
6	13																																																																																																																														
7	17																																																																																																																														
8	15																																																																																																																														
9	15																																																																																																																														
	A	B	C																																																																																																																												
0	10	3.0	NaN																																																																																																																												
1	10	NaN	3.0																																																																																																																												
2	19	7.0	5.0																																																																																																																												
3	18	3.0	3.0																																																																																																																												
4	12	6.0	NaN																																																																																																																												
5	14	6.0	9.0																																																																																																																												
6	13	8.0	4.0																																																																																																																												
7	17	6.0	NaN																																																																																																																												
9	15	3.0	NaN																																																																																																																												

与NaN相关的函数：ffill()、bfill()和interpolate()方法

ffill()、bfill()和interpolate()方法分别使用NaN之前的元素值，之后的元素值，或前后元素的插值填充该NaN元素。填充顺序，默认bfill()为backward, 其他为forward。

```
print(df_nan.ffill()) # 在0轴（列）上填充NaN之前的元素值
print(df_nan.ffill(axis=1)) # 在1轴（行）上填充NaN之前的元素值
print(df_nan.bfill()) # 在0轴（列）上填充NaN之后的元素值
print(df_nan.interpolate()) # 在0轴（列）上填充NaN之前后元素的插值
```

df_nan	df_nan.ffill()	df_nan.ffill(axis=1)	df_nan.bfill()	df_nan.interpolate()
A B C	A B C	A B C	A B C	A B C
0 10 3.0 NaN	0 10 3.0 NaN	0 10.0 3.0 3.0	0 10 3.0 3.0	0 10 3.0 NaN
1 10 NaN 3.0	1 10 3.0 3.0	1 10.0 10.0 3.0	1 10 7.0 3.0	1 10 5.0 3.0
2 19 7.0 5.0	2 19 7.0 5.0	2 19.0 7.0 5.0	2 19 7.0 5.0	2 19 7.0 5.0
3 18 3.0 3.0	3 18 3.0 3.0	3 18.0 3.0 3.0	3 18 3.0 3.0	3 18 3.0 3.0
4 12 6.0 NaN	4 12 6.0 3.0	4 12.0 6.0 6.0	4 12 6.0 9.0	4 12 6.0 6.0
5 14 6.0 9.0	5 14 6.0 9.0	5 14.0 6.0 9.0	5 14 6.0 9.0	5 14 6.0 9.0
6 13 8.0 4.0	6 13 8.0 4.0	6 13.0 8.0 4.0	6 13 8.0 4.0	6 13 8.0 4.0
7 17 6.0 NaN	7 17 6.0 4.0	7 17.0 6.0 6.0	7 17 6.0 NaN	7 17 6.0 4.0
8 15 NaN NaN	8 15 6.0 4.0	8 15.0 15.0 15.0	8 15 3.0 NaN	8 15 4.5 4.0
9 15 3.0 NaN	9 15 3.0 4.0	9 15.0 3.0 3.0	9 15 3.0 NaN	9 15 3.0 4.0

与NaN相关的函数： ffill()、bfill()和interpolate()方法

interpolate()方法的插值方法由**method**参数指定，默认为linear，也可以是其他插值方法，例如quadratic, cubic, polynomial, spline, index, pad等。

- line: 线性插值的结果为前后元素的平均值；
- polynomial: 多项式插值，需要指定多项式次数。例如：
df.interpolate(method='polynomial', order=5)
- quadratic: 二次多项式（抛物线）插值，等价于：
df.interpolate(method='polynomial', order=2)
- cubic: 三次多项式插值，等价于：
df.interpolate(method='polynomial', order=3)
- index: 使用索引值进行插值运算：索引值的差分比值等于数据差分的比值
- pad: 用以已有值填充，可以不是数值类型；

<code>s = pd.Series([0, 1, np.nan, 3])</code>	<code>s.interpolate() # method='linear'</code>
0 0.0	0 0.0
1 1.0	1 1.0
2 NaN	2 2.0
3 3.0	3 3.0
dtype: float64	dtype: float64

与NaN相关的函数： ffill()、bfill()和interpolate()方法

<pre>s = pd.Series([0, 2, np.nan, 8])</pre>	<pre>s.interpolate(method='polynomial', order=2) s.interpolate(method='quadratic') # 二次多项式插值, 同上</pre>
<pre>0 0.0 1 2.0 2 NaN 3 8.0 dtype: float64</pre>	<pre>0 0.000000 1 2.000000 2 4.666667 3 8.000000 dtype: float64</pre>
<pre>s = pd.Series([3, np.nan, 7], index=[0,8,9])</pre>	<pre>s.interpolate(method='index') # 使用索引值进行插值运算: 索引值的差分比值等于数据差分的比值</pre>
<pre>0 3.0 8 NaN 9 7.0 dtype: float64</pre>	<pre>0 3.000000 8 6.555556 9 7.000000 dtype: float64</pre>

与NaN相关的函数: ffill()、bfill()和interpolate()方法

```
s = pd.Series(  
    [np.nan, "single_one", np.nan,  
     "fill_two_more", np.nan, np.nan, np.nan,  
     4.71, np.nan])
```

```
0      NaN  
1  single_one  
2      NaN  
3  fill_two_more  
4      NaN  
5      NaN  
6      NaN  
7      4.71  
8      NaN  
dtype: object
```

```
s.interpolate(method='pad',  
               limit=2)
```

```
0      NaN  
1  single_one  
2  single_one  
3  fill_two_more  
4  fill_two_more  
5  fill_two_more  
6      NaN  
7      4.71  
8      4.71  
dtype: object
```

与NaN相关的函数：fillna()方法

fillna()方法：将NaN值填充为value参数指定的值。value参数若为字典，则让fillna()对不同的列使用不同的值填充NaN。

```
np.random.seed(41)
```

```
A = np.random.randint(0, 10, (10, 3))
```

```
df_int = pd.DataFrame(data=A, columns=list("ABC"))
```

```
df_int["A"] += 10 # 修改第一列元素：各元素+10
```

```
df_nan = df_int.where(df_int > 2) # 小于等于2的元素全部置为NaN，返回一个DataFrame
```

df_nan	df_nan.fillna(value='-9999')			df_nan.fillna(value={'B':-9999,'C':0})		
A B C	A B C			A B C		
0 10 3.0 NaN	0 10 3 -9999			0 10 3.0 0.0		
1 10 NaN 3.0	1 10 -9999 3			1 10 -9999.0 3.0		
2 19 7.0 5.0	2 19 7 5			2 19 7.0 5.0		
3 18 3.0 3.0	3 18 3 3			3 18 3.0 3.0		
4 12 6.0 NaN	4 12 6 -9999			4 12 6.0 0.0		
5 14 6.0 9.0	5 14 6 9			5 14 6.0 9.0		
6 13 8.0 4.0	6 13 8 4			6 13 8.0 4.0		
7 17 6.0 NaN	7 17 6 -9999			7 17 6.0 0.0		
8 15 NaN NaN	8 15 -9999 -9999			8 15 -9999.0 0.0		
9 15 3.0 NaN	9 15 3 -9999			9 15 3.0 0.0		

各种聚合方法的skipna参数默认为True, 因此计算时将忽略NaN元素, 注意每行或每列是单独运算的。如果需要忽略包含NaN的整行, 需要先调用dropna()。若将skipna参数设置为False, 则包含NaN的行或列的运算结果为NaN。

df_nan	df_nan.sum()	df_nan.sum(skipna=False)	df_nan.dropna(axis=0).sum()
<pre>A B C 0 10 3.0 NaN 1 10 NaN 3.0 2 19 7.0 5.0 3 18 3.0 3.0 4 12 6.0 NaN 5 14 6.0 9.0 6 13 8.0 4.0 7 17 6.0 NaN 8 15 NaN NaN 9 15 3.0 NaN</pre>	<pre>A 143.0 B 42.0 C 24.0 dtype: float64</pre>	<pre>dtype: float64 A 143.0 B NaN C NaN dtype: float64</pre>	<pre>A 64.0 B 24.0 C 21.0 dtype: float64</pre>

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

与DataFrame结构相关的操作主要包括：

函数或运算符	功能	函数	功能
shape属性	获取形状，同Numpy二维数组	set_index()	设置索引，即列转换为行索引
rename()	索引标签重命名	reset_index()	将行索引转换为列
df['col_label']	添加、修改或删除列，或调整列序	stack()	将列索引转换为行索引
insert()	插入新列	uastack()	将行索引转换为列索引
assign()	返回添加新列之后的数据	reorder_levels()	设置索引级别的顺序
drop()	删除行或列(del df['col_label'], del df.col_label))	sort_index()	对索引排序
append()	添加行	reorder_levels()	设置索引级别的顺序
concat()	拼接多块数据	swaplevel()	交换索引中两个级别的顺序
		sort_values()	对值排序
		pivot()	创建透视表
		melt()	透视表的逆变换

修改index、columns名

一般常用的有两个方法：

■ 修改index或columns属性

- DataFrame.index = [newName]
- DataFrame.columns = [newName]

■ 使用rename()方法 (推荐)

DataFrame.rename mapper = None , index = None, columns = None, copy = True, inplace = False, level = None)

参数：

- mapper, index, columns：映射函数，或旧名与新名的映射关系的字典。
- axis：int或str，可以是轴名称('index', 'columns')或数字(0,1)。默认为' index'。
- copy：默认为True，是否复制基础数据。
- **inplace**：默认为False，返回新的DataFrame。否则为True，则忽略copy参数。
- level：在多级索引中，指定要修改的索引的级别。

修改index、columns名

几个示例：

修改方式	修改后结构
<pre>df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]}) df.columns = ["AA", "BB"] print(df)</pre>	<pre>AA BB 0 1 4 1 2 5 2 3 6</pre>
<pre># Rename columns using a mapping df = df.rename(columns={"AA": "a", "BB": "c"}) #或者 df.rename(columns={"AA": "a", "BB": "c"}, inplace=True) print(df)</pre>	<pre>a c 0 1 4 1 2 5 2 3 6</pre>
<pre># Rename index using a mapping: df = df.rename(index={0: "x", 1: "y", 2: "z"}) print(df)</pre>	<pre>a c x 1 4 y 2 5 z 3 6</pre>
<pre># Using mapper and axis-style parameters df = df.rename(str.upper, axis='columns') print(df)</pre>	<pre>A C x 1 4 y 2 5 z 3 6</pre>

添加列

由于DataFrame可以看作一个Series对象的字典，因此通过DataFrame[colname] = values即可添加新列。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  
df["C"] = [7, 8, 9]  
print(df)
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

有时新添加的列是从已经存在的列计算而来，这时可以使用**eval()方法**计算。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  
df["C"] = df.eval('B*10')  
print(df)
```

	A	B	C
0	1	4	40
1	2	5	50
2	3	6	60

assign()方法添加由关键字参数指定的列，它返回一个新的DataFrame对象，**原数据的内容保持不变**。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  
print(df.assign(C=df.B+2))
```

	A	B	C
0	1	4	6
1	2	5	7
2	3	6	8

插入新列

DataFrame的insert()方法用于在指定列序号插入新列：

df.insert(col_id, col_index, value)

col_id: 插入列的位置序号, col_index: 列标签

value: 新列, 可以是列表, 也可以是序列

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  
df.insert(0, 'C', [7, 8, 9])  
print(df)
```

	C	A	B
0	7	1	4
1	8	2	5
2	9	3	6

```
df.insert(2, 'D', pd.Series([1, 0, 5]))  
print(df)
```

	C	A	D	B
0	7	1	1	4
1	8	2	0	5
2	9	3	5	6

调整列顺序

有两种方法:

(1) 读取某列保存到变量s (即用s引用它), 然后从df中移除该列, 最后s插入df中指定位置。----适合单列调整

(2) 指定具有新顺序的列标签列表order, 然后从df中获取这些列并覆盖df。---适合调整多列

<pre>C A D B 0 7 1 1 4 1 8 2 0 5 2 9 3 5 6</pre>	<pre>s=df['C'] del df['C'] df.insert(1,'C',s) print(df)</pre>	<pre>A C D B 0 1 7 1 4 1 2 8 0 5 2 3 9 5 6</pre>
--	---	--

<pre>order=['A','B','D','C'] df=df[order] print(df)</pre>	<pre>A B D C 0 1 4 1 7 1 2 5 0 8 2 3 6 5 9</pre>
---	--

删除列

使用**drop()方法**可以删除指定的列，参数与rename()方法类似，默认返回一个新的数据框。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6],  
                  "C": [7, 8, 9], "D": [0, 0, 0]})  
df = df.drop(columns=["C", "D"])  
# 等价于 df.drop(columns=["C", "D"], inplace=True)  
print(df)
```

	A	B
0	1	4
1	2	5
2	3	6

也可以使用**del 命令**删除一列。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6],  
                  "C": [7, 8, 9], "D": [0, 0, 0]})  
  
del df["D"]  
print(df)
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

添加行

append()方法用于添加行，它没有inplace参数，只能返回一个全新对象。

添加的数据可以源自另一个数据框，也可以来自字典或Series，但此时ignore_index参数必须设置为True，表示忽略行索引，从而自动添加新行的索引。

从另一个数据框中添加(相当于合并两个结构相同的数据框)

```
df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))  
df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))  
print(df.append(df2))
```

	A	B
0	1	2
1	3	4
0	5	6
1	7	8

With ignore_index set to True

```
print(df.append(df2, ignore_index=True))
```

	A	B
0	1	2
1	3	4
2	5	6
3	7	8

忽略行索引时，还可以从字典添加行

```
row_dict = {"A": 5, "B": 6}  
print(df.append(row_dict, ignore_index=True))
```

	A	B
0	1	2
1	3	4
2	5	6

忽略行索引时，还可以从Series添加行

```
row_series = pd.Series([5, 6], index=['A', 'B'])  
print(df.append(row_series, ignore_index=True))
```

	A	B
0	1	2
1	3	4
2	5	6

添加行

由于每次调用append()都会复制所有的数据，因此在循环中使用append()添加数据会极大地降低程序的运算速度。可以使用一个列表缓存所有的分块数据，然后调用pd.concat()方法将所有这些数据沿着指定轴拼贴到一起。

从另一个数据框中添加(相当于合并两个结构相同的数据框)

```
df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
```

```
df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
```

```
df_list = [df, df2] # 数据缓冲列表
```

在0轴(即列上)拼接数据，并忽略行索引

```
df3 = pd.concat(df_list, axis=0, ignore_index=True)
```

```
print(df3)
```

	A	B
0	1	2
1	3	4
2	5	6
3	7	8

添加行

示例：对比append()方法与pd.concat()方法合并数据框的运算速度。

```
import pandas as pd
import numpy as np
import time
```

```
def random_dataframe(n):
```

```
    """
```

定义生成器 (generator)，调用时不会立即被执行，而是先生成一个对象genertor(假定为x)，通过next(x)生成一个元素，该元素由yield语句产生；

yield语句执行完毕后，函数代码暂停执行，直到下一次调用next(x)，代码才继续执行。

```
    """
```

```
    print('The generator is starting...')
```

```
    columns=list('ABC') #
```

```
    for i in range(n):
```

```
        m=np.random.randint(10,20) #矩阵的函数为随机数整数，10~20之间
```

```
        A=np.random.randint(0,10,size=(m,3)) # mx3的随机矩阵
```

```
        yield pd.DataFrame(data=A,columns=columns) # 产生元素：数据框
```

```
    print('The generator ends.') # 生成器函数中的这条语句可能永远不会被执行！
```

```
generatorObj=random_dataframe(1000) # 这里仅仅是创建了一个生成器，函数并没有开始执行
df_list=list(generatorObj) # 生成1000个元素（自动执行next(generatorObj)1000次）并存入列表
```

添加行

示例：对比append()方法与pd.concat()方法合并数据框的运算速度。

```
print('1、使用append()方法合并数据框')
time_start = time.time() # 记录开始时间
df1=pd.DataFrame([ ]) # 创建一个空的数据框

for df in df_list:
    df1.append(df)

time_end = time.time() # 记录结束时间
elapsed = time_end - time_start # 总共用时（秒）
print(f"Time used:{1000*elapsed}毫秒")
```

```
print('2、使用pd.concat()方法合并数据框')
time_start = time.time()
df2=pd.DataFrame([ ]) # 创建一个空的数据框

df2=pd.concat(df_list,axis=0)

time_end = time.time()
elapsed = time_end - time_start
print(f"Time used:{1000*elapsed}毫秒")
```

The generator is starting...

The generator ends.

1、使用append()方法合并数据框

Time used:101.72915458679199毫秒

2、使用pd.concat()方法合并数据框

Time used:43.94197463989258毫秒

删除行

drop()方法用于删除指定标签对应的行或列。删除行时指定**labels**参数为要删除的行索引，**axis**参数默认为0（即对行操作）。

```
df = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7, 8]], columns=list('AB'))  
print(df)
```

	A	B
0	1	2
1	3	4
2	5	6
3	7	8

```
print(df.drop(labels=[2,3]))  
print(df.drop([2,3]))  
print(df.drop([2,3],axis=0))
```

	A	B
0	1	2
1	3	4

何删除满足条件的行呢？

办法：先使用**query()**方法查询满足条件的数据行（返回一个数据框），然后提取该数据框的行索引**index**对象，最后使用**drop()**方法删除由这些行索引指定的行。

```
index=df.query('A<4').index # df中A列<4的行的行索引  
print(df.drop(labels=index))
```

	A	B
2	5	6
3	7	8

排序--按索引排序

sort_index()方法用于对行索引或列索引排序。参数axis = 0（默认值）时，对行索引排序（即数据行按索引顺序升序或降序排列）；axis = 1时，对列索引排序（即数据列按列索引顺序升序或降序排列）。默认升序（ascending = True），默认不修改自身（inplace = False）。

```
df = pd.DataFrame({'Name': ['Tom', 'emily', 'Fred'],  
                  'Sex':   ['M', 'F', 'M'],  
                  'Age':   [35, 16, 28]})
```

df	df.sort_index()	df.sort_index(ascending=False)	df.sort_index(axis=1)																																																																
<table><tr><th></th><th>Name</th><th>Sex</th><th>Age</th></tr><tr><td>0</td><td>Tom</td><td>M</td><td>35</td></tr><tr><td>1</td><td>emily</td><td>F</td><td>16</td></tr><tr><td>2</td><td>Fred</td><td>M</td><td>28</td></tr></table>		Name	Sex	Age	0	Tom	M	35	1	emily	F	16	2	Fred	M	28	<table><tr><th></th><th>Name</th><th>Sex</th><th>Age</th></tr><tr><td>0</td><td>Tom</td><td>M</td><td>35</td></tr><tr><td>1</td><td>emily</td><td>F</td><td>16</td></tr><tr><td>2</td><td>Fred</td><td>M</td><td>28</td></tr></table>		Name	Sex	Age	0	Tom	M	35	1	emily	F	16	2	Fred	M	28	<table><tr><th></th><th>Name</th><th>Sex</th><th>Age</th></tr><tr><td>2</td><td>Fred</td><td>M</td><td>28</td></tr><tr><td>1</td><td>emily</td><td>F</td><td>16</td></tr><tr><td>0</td><td>Tom</td><td>M</td><td>35</td></tr></table>		Name	Sex	Age	2	Fred	M	28	1	emily	F	16	0	Tom	M	35	<table><tr><th></th><th>Age</th><th>Name</th><th>Sex</th></tr><tr><td>0</td><td>35</td><td>Tom</td><td>M</td></tr><tr><td>1</td><td>16</td><td>emily</td><td>F</td></tr><tr><td>2</td><td>28</td><td>Fred</td><td>M</td></tr></table>		Age	Name	Sex	0	35	Tom	M	1	16	emily	F	2	28	Fred	M
	Name	Sex	Age																																																																
0	Tom	M	35																																																																
1	emily	F	16																																																																
2	Fred	M	28																																																																
	Name	Sex	Age																																																																
0	Tom	M	35																																																																
1	emily	F	16																																																																
2	Fred	M	28																																																																
	Name	Sex	Age																																																																
2	Fred	M	28																																																																
1	emily	F	16																																																																
0	Tom	M	35																																																																
	Age	Name	Sex																																																																
0	35	Tom	M																																																																
1	16	emily	F																																																																
2	28	Fred	M																																																																

排序—按值排序

sort_values()方法用于按值排序。参数axis = 0（默认值）时，对行按某些列排序（即**数据行**按一个或多个列升序或降序排列）；axis = 1时，对列按某些行索引排序（即**数据列**按一个或多个行值顺序升序或降序排列）。默认升序（ascending = True），默认不修改自身（inplace = False）。

df	df.sort_values(by=['Name']) # 值按姓名升序排列	df.sort_values(by=['Name'], ascending=False) # 值按姓名降序排列
<pre> Name Sex Age 0 Tom M 35 1 emily F 16 2 Fred M 28 </pre>	<pre> Name Sex Age 1 emily F 16 2 Fred M 28 0 Tom M 35 </pre>	<pre> Name Sex Age 0 Tom M 35 2 Fred M 28 1 emily F 16 </pre>

df.sort_values(by=['Sex','Age']) # 值按性别、年龄升序排列	df.sort_values(by=['Sex', 'Age'], ascending=[True, False]) # 值按性别升序、年龄降序排列
<pre> Name Sex Age 1 emily F 16 2 Fred M 28 0 Tom M 35 </pre>	<pre> Name Sex Age 1 emily F 16 0 Tom M 35 2 Fred M 28 </pre>

列转化为行索引

下面首先从CSV文件读入数据，并使用groupby()计算分组的平均值。关于groupby在后面的分组运算中还会详细介绍。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Group	Contour	Depth	Gp	Block	pH	N	Dens	P	Ca	Mg	K	Na	Conduc
2	1	1	Top	0-10	T0	1	5.4	0.188	0.92	215	16.35	7.65	0.72	1.14	1.09
3	2	1	Top	0-10	T0	2	5.65	0.165	1.04	208	12.25	5.15	0.71	0.94	1.35
4	3	1	Top	0-10	T0	3	5.14	0.26	0.95	300	13.02	5.68	0.68	0.6	1.41
5	4	1	Top	0-10	T0	4	5.14	0.169	1.1	248	11.92	7.88	1.09	1.01	1.64
6	5	2	Top	10-30	T1	1	5.14	0.164	1.12	174	14.17	8.12	0.7	2.17	1.85
7	6	2	Top	10-30	T1	2	5.1	0.094	1.22	129	8.55	6.92	0.81	2.67	3.18
8	7	2	Top	10-30	T1	3	4.7	0.1	1.52	117	8.74	8.16	0.39	3.32	4.16
9	8	2	Top	10-30	T1	4	4.46	0.112	1.47	170	9.49	9.16	0.7	3.76	5.14
10	9	3	Top	30-60	T3	1	4.37	0.112	1.07	121	8.85	10.35	0.74	5.74	5.73
11	10	3	Top	30-60	T3	2	4.39	0.058	1.54	115	4.73	6.91	0.77	5.85	6.45

列转化为行索引

注意下面的soil_mean对象的行索引是两级索引：

```
df_soils = pd.read_csv("Soils.csv", index_col=0) # csv的第1列作为行索引
soils=df_soils[["Depth", "Contour", "Group", "pH", "N"]] # 取若干列
# 先按Depth分组，组内再按Contour分组，然后各列按分组求平均值
soils_mean = soils.groupby(["Depth", "Contour"]).mean()
```

`print(soils.head())`

	Depth	Contour	Group	pH	N
1	0-10	Top	1	5.40	0.188
2	0-10	Top	1	5.65	0.165
3	0-10	Top	1	5.14	0.260
4	0-10	Top	1	5.14	0.169
5	10-30	Top	2	5.14	0.164

`print(soils_mean.head())`

Depth	Contour	Group	pH	N
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

列转化为行索引

set_index()方法将列转换为行索引，如果append参数为False(默认值)，则删除当前的行索引；若为True，则为当前的索引添加新的级别。

例如，将soils_mean中的Group列设置为行索引，返回数据框具有了3级行索引。

soils_mean.head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

soils_mean.set_index('Group',
append=True).head()

			pH	N
Depth	Contour	Group		
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

行索引转化为列

reset_index()方法将行索引转换为列，通过level参数可以指定被转换为列的级别（默认所有级别）。若只希望从索引中删除某个级别，可以设置drop参数为True。例如，将soils_mean中的1级行索引Contour转化为列，返回的数据框为单级行索引。

soils_mean.head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

soils_mean.reset_index(level='Contour',
drop=True).head()

		Group	pH	N
Depth				
0-10		9	5.3525	0.17825
0-10		5	5.5075	0.21900
0-10		1	5.3325	0.19550
10-30		10	4.8800	0.08025
10-30		6	5.2825	0.10100

soils_mean.reset_index(
level='Contour').head()

	Contour	Group	pH	N
Depth				
0-10	Depression	9	5.3525	0.17825
0-10	Slope	5	5.5075	0.21900
0-10	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
10-30	Slope	6	5.2825	0.10100

行索引和列索引的相互转换

stack()方法把指定级别的列索引转换为行索引，而**unstack()**则把行索引转换为列索引。下面的程序将行索引中的第1级转换为列索引的第1级，所得到的结果中行索引为单级索引，而列索引为两级索引。

soils_mean.head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

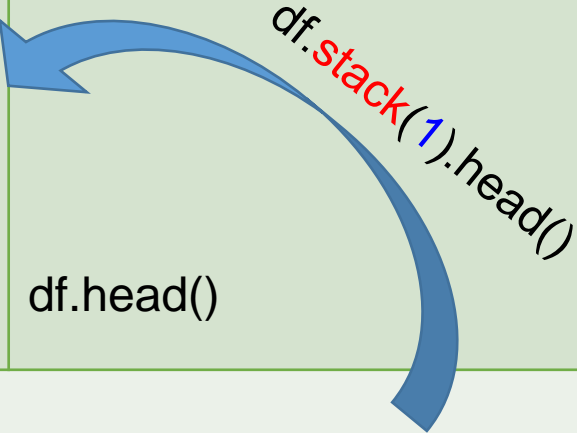
将行索引中的第1级转换为列索引的第1级
df=soils_mean.unstack(level=1)
print(df.head())

	Group			pH			N		
	Depression	Slope	Top	Depression	Slope	Top	Depression	Slope	Top
Contour									
Depth									
0-10	9	5	1	5.3525	5.5075	5.3325	0.17825	0.21900	0.19550
10-30	10	6	2	4.8800	5.2825	4.8500	0.08025	0.10100	0.11750
30-60	11	7	3	4.3625	4.2675	4.2050	0.05050	0.06075	0.07950
60-90	12	8	4	4.1725	3.9275	3.8925	0.04025	0.04300	0.05775

行索引和列索引的相互转换

对前面得到的df调用stack(), 将其列索引中的第1级转换为行索引的第1级, 所得到的结果中列索引为单级索引, 而行索引为两级索引。返回的数据框与soils_mean完全相同。

df.stack(1).head()									
		Group	pH	N					
Depth 0-10	Contour Depression	9	5.3525	0.17825					
	Slope	5	5.5075	0.21900					
	Top	1	5.3325	0.19550					
10-30	Depression	10	4.8800	0.08025					
	Slope	6	5.2825	0.10100					



Contour	Group			pH			N		
	Depression	Slope	Top	Depression	Slope	Top	Depression	Slope	Top
Depth 0-10	9	5	1	5.3525	5.5075	5.3325	0.17825	0.21900	0.19550
10-30	10	6	2	4.8800	5.2825	4.8500	0.08025	0.10100	0.11750
30-60	11	7	3	4.3625	4.2675	4.2050	0.05050	0.06075	0.07950
60-90	12	8	4	4.1725	3.9275	3.8925	0.04025	0.04300	0.05775

行索引和列索引的相互转换

无论是`stack()`还是`unstack()`，当所有的索引被转换到同一个轴上时，将得到一个Series对象。

`soils_mean.head()`

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

`soils_mean.stack().head(10)`

Depth	Contour		
0-10	Depression	Group	9.00000
		pH	5.35250
		N	0.17825
	Slope	Group	5.00000
		pH	5.50750
		N	0.21900
	Top	Group	1.00000
		pH	5.33250
		N	0.19550
10-30	Depression	Group	10.00000
dtype: float64			

交换索引的等级

`reorder_levels()`和 `swaplevel()`用于交换指定轴的索引级别。

下面调用 `swaplevel()`交换行索引的两个级别，然后调用`sort_index()`对新的索引进行排序：

soils_mean					soils_mean.swaplevel(0,1,axis=0).sort_index()				
		Group	pH	N			Group	pH	N
0-10	Depression	9	5.3525	0.17825	Depression	0-10	9	5.3525	0.17825
	Slope	5	5.5075	0.21900		10-30	10	4.8800	0.08025
	Top	1	5.3325	0.19550		30-60	11	4.3625	0.05050
10-30	Depression	10	4.8800	0.08025	Slope	60-90	12	4.1725	0.04025
	Slope	6	5.2825	0.10100		0-10	5	5.5075	0.21900
	Top	2	4.8500	0.11750		10-30	6	5.2825	0.10100
30-60	Depression	11	4.3625	0.05050	Top	30-60	7	4.2675	0.06075
	Slope	7	4.2675	0.06075		60-90	8	3.9275	0.04300
	Top	3	4.2050	0.07950		0-10	1	5.3325	0.19550
60-90	Depression	12	4.1725	0.04025		10-30	2	4.8500	0.11750
	Slope	8	3.9275	0.04300		30-60	3	4.2050	0.07950
	Top	4	3.8925	0.05775		60-90	4	3.8925	0.05775

透视表

`pivot()`可以将DataFrame中的三列数据分别作为行索引、列索引和元素值，将这三列数据转换为二维表格。

例如，分别显示soil_mean中pH指标与N指标在Depth和Contour索引上的透视图。

```
df1 = soils_mean.reset_index()# 将所有行索引都转换为列
```

```
df=df1[["Depth", "Contour", "pH", "N"]]
```

```
df_pivot_pH = df.pivot("Depth", "Contour", "pH")
```

```
df_pivot_N = df.pivot("Depth", "Contour", "N")
```

df					df_pivot_pH 和 df_pivot_N				
	Depth	Contour	pH	N		Contour	Depression	Slope	Top
0	0-10	Depression	5.3525	0.17825	Depth				
1	0-10	Slope	5.5075	0.21900	0-10		5.3525	5.5075	5.3325
2	0-10	Top	5.3325	0.19550	10-30		4.8800	5.2825	4.8500
3	10-30	Depression	4.8800	0.08025	30-60		4.3625	4.2675	4.2050
4	10-30	Slope	5.2825	0.10100	60-90		4.1725	3.9275	3.8925
5	10-30	Top	4.8500	0.11750					
6	30-60	Depression	4.3625	0.05050	Contour	Depression	Slope	Top	
7	30-60	Slope	4.2675	0.06075	Depth				
8	30-60	Top	4.2050	0.07950	0-10		0.17825	0.21900	0.19550
9	60-90	Depression	4.1725	0.04025	10-30		0.08025	0.10100	0.11750
10	60-90	Slope	3.9275	0.04300	30-60		0.05050	0.06075	0.07950
11	60-90	Top	3.8925	0.05775	60-90		0.04025	0.04300	0.05775

透视表

`pivot()`的三个参数`index`、`columns`和`values`只支持指定一列数据。若不指定`values`参数，就将剩余的列都当作元素值列，得到多级列索引。

```
df_pivot = df.pivot("Depth", "Contour")
pd.options.display.float_format = '{:.2f}'.format
print(df_pivot)
```

df					df_pivot						
	Depth	Contour	pH	N							
0	0-10	Depression	5.3525	0.17825							
1	0-10	Slope	5.5075	0.21900							
2	0-10	Top	5.3325	0.19550							
3	10-30	Depression	4.8800	0.08025							
4	10-30	Slope	5.2825	0.10100							
5	10-30	Top	4.8500	0.11750							
6	30-60	Depression	4.3625	0.05050							
7	30-60	Slope	4.2675	0.06075							
8	30-60	Top	4.2050	0.07950							
9	60-90	Depression	4.1725	0.04025							
10	60-90	Slope	3.9275	0.04300							
11	60-90	Top	3.8925	0.05775							

				pH		
				N		
Contour	Depression	Slope	Top	Depression	Slope	Top
Depth						
0-10	5.35	5.51	5.33	0.18	0.22	0.20
10-30	4.88	5.28	4.85	0.08	0.10	0.12
30-60	4.36	4.27	4.21	0.05	0.06	0.08
60-90	4.17	3.93	3.89	0.04	0.04	0.06

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

分组运算是指使用特定的条件将数据分为多个分组，然后对每个分组进行某种运算，最后再将结果整合起来。Pandas中的分组运算由DataFrame或 Series对象的 **groupby()方法**实现。

以某种药剂的实验数据”dose.asv”为例介绍如何使用分组运算分析数据。在该数据集中使用了”ABCD”4种不同的药剂处理方式(Tmt), 针对不同同性别(Gender)、不同年龄(Age)的患者进行药剂实验，记录下药剂的投药量(Dose)与两种药剂反应(Response)。

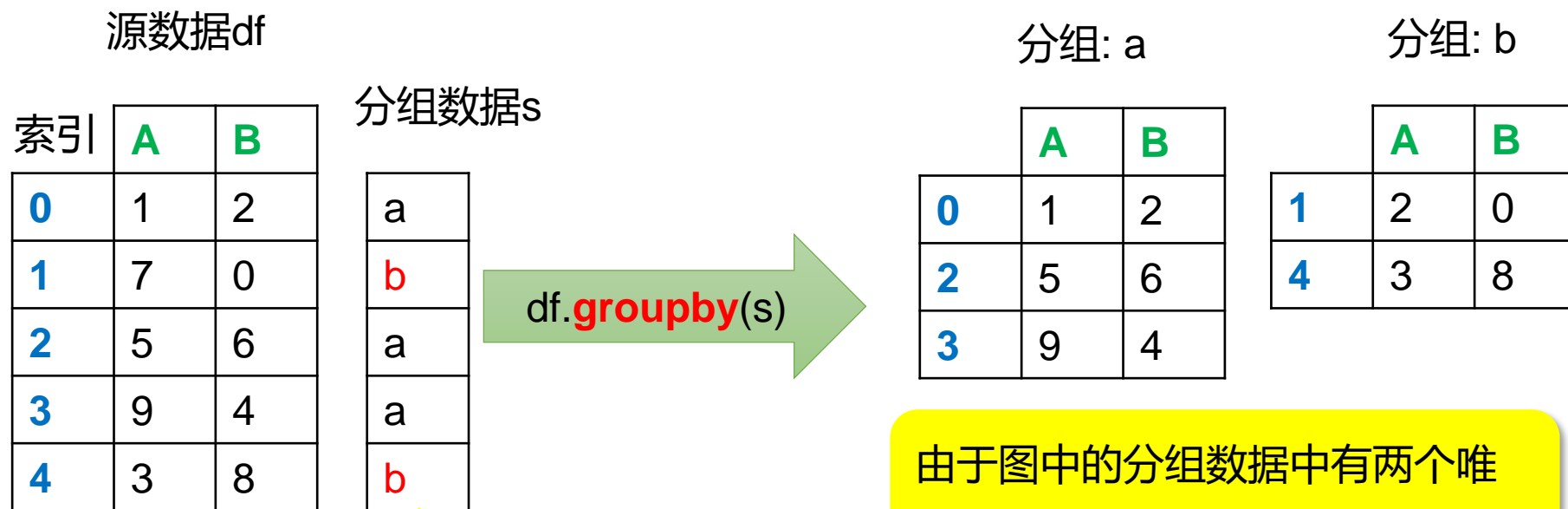
	A	B	C	D	E	F
1	Dose	Response1	Response2	Tmt	Age	Gender
2	50	9.872	10.032	C	60s	F
3	15	0.002	0.004	D	60s	F
4	25	0.626	0.803	C	50s	M
5	25	1.372	1.557	C	60s	F
6	15	0.01	0.02	C	60s	F
7	20	0.007	0.079	D	50s	F
8	1	0	0	A	50s	F
9	20	0.038	0.033	C	60s	M
10	15	0.001	0.001	D	50s	F
11	40	10.598	10.319	B	60s	F

```
df_dose = pd.read_csv("dose.csv")  
print(df_dose.head())
```

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
1	15.0	0.002	0.004	D	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F

groupby()方法

分组操作中涉及两组数据：**源数据**和**分组数据（分组依据）**。将分组数据传递给源数据的groupby()方法以完成分组。groupby()的axis参数默认为0，表示对源数据的行进行分组。源数据中的每行与分组数据中的每个元素对应，分组数据中的每个唯一值对应一个分组。



由于图中的分组数据中有两个唯一值a和b，因此得到两个分组。

给df中的每一行贴一个标签

groupby()方法：3种分组情况

(1) 当分组用的数据在源数据中时，可以直接通过列名指定分组数据。

当源数据是DataFrame类型时，groupby()方法返回一个DataFrameGroupBy对象；
当源数据是Series类型，则返回SeriesGroupBy对象。

例如：使用Tmt列对源数据分组

```
tmt_group=df_dose.groupby('Tmt')  
print(type(tmt_group))
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

分组数

len(tmt_group)

4

#分组聚合运算：分组平均

tmt_group.agg(np.mean)

注意到分组平均后，
Tmt成了行索引

	Dose	Response1	Response2
Tmt			
A	33.546154	6.728985	6.863185
B	33.546154	5.573354	5.456415
C	33.546154	4.040415	4.115323
D	33.546154	3.320646	3.188369

groupby()方法：3种分组情况

还可以使用列表传递多组分组数据给groupby(),从而新进行多级分组。

例如, 先按Tmt分成 m 组, 然后在每一组内再按性别Gender分成若干小组 (每组 n_i 行), 总共的分组数为: $\sum_{i=1}^m n_i$

多级分组

```
tmt_gender_group = df_dose.groupby(["Tmt", "Gender"])
```

分组数

`len(tmt_gender_group)`

8

分组聚合运算: 分组平均

`tmt_gender_group.agg(np.mean)`

注意到分组平均后, Tmt
和Gender分别成了数据
框的第0级和第1级行索引

		Dose	Response1	Response2
Tmt	Gender			
A	F	33.546154	6.774795	6.964769
	M	33.546154	6.660269	6.710808
B	F	33.546154	5.538205	5.383692
	M	33.546154	5.626077	5.565500
C	F	33.546154	4.231808	4.269577
	M	33.546154	3.912821	4.012487
D	F	33.546154	3.452487	3.363590
	M	33.546154	3.122885	2.925538

groupby()方法：3种分组情况

(2) 当分组数据不在源数据中时，可以直接传递分组数据。

例如，对长度与源数据的行数相同、取值范围为[0,5)的随机整数数组进行分组，这样就将源数据随机分成了5组：

```
m = df_dose.shape[0] # df_dose的行数
random_values = np.random.randint(0, 5, m) # m个0-4的随机整数
random_group = df_dose.groupby(random_values)
```

df_dose.head(10)

	Dose	Response1	Response2	Tmt	Age	Gender	
0	50.0	9.872	10.032	C	60s	F	[[3]
1	15.0	0.002	0.004	D	60s	F	[3]
2	25.0	0.626	0.803	C	50s	M	[0]
3	25.0	1.372	1.557	C	60s	F	[4]
4	15.0	0.010	0.020	C	60s	F	[0]
5	20.0	0.007	0.079	D	50s	F	[2]
6	1.0	0.000	0.000	A	50s	F	[2]
7	20.0	0.038	0.033	C	60s	M	[1]
8	15.0	0.001	0.001	D	50s	F	[2]
9	40.0	10.598	10.319	B	60s	F	[1]]

random_values.
reshape(-1,1)[0:10]

#分组聚合运算：分组平均
random_group.agg(np.mean)

	Dose	Response1	Response2
0	35.943103	5.525000	5.540603
1	32.371154	4.993462	5.036077
2	29.890385	4.458231	4.431615
3	30.851923	4.244135	4.274769
4	39.030435	5.336696	5.207630

groupby()方法：3种分组情况

(3) 当分组数据可以通过源数据的行索引计算时，可以将**计算函数**传递给groupby()。groupby()中传入一个计算函数时，数据框的行索引号将**逐个传递**给该函数，**函数返回值**将作为各行在分组数据中的**对应标签**。

例如，对于行索引为0开始的整数序列的数据框，使用行索引值除以3 的余数进行分组，因此将源数据的每行交替地分为3 组：

```
alternating_group = df_dose.groupby(lambda n: n % 3)
```

df_dose.head(10)

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
1	15.0	0.002	0.004	D	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F
5	20.0	0.007	0.079	D	50s	F
6	1.0	0.000	0.000	A	50s	F
7	20.0	0.038	0.033	C	60s	M
8	15.0	0.001	0.001	D	50s	F
9	40.0	10.598	10.319	B	60s	F

#分组聚合运算：分组平均

alternating_group.agg(np.mean)

	Dose	Response1	Response2
0	31.843678	4.764322	4.730977
1	31.170115	4.422460	4.379506
2	37.672093	5.568267	5.615140

GroupBy 对象

使用**len()方法**可以获取分组数，例如len(tmt_group)。

使用**get_group()方法**可以获得与指定的**分组键**对应的数据。例如：

df_dose.head()

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
1	15.0	0.002	0.004	D	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F

tmt_group.get_group("C").head()

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F
7	20.0	0.038	0.033	C	60s	M

GroupBy 对象

使用**len()**方法可以获取分组数，例如len(tmt_group)。

使用**get_group()**方法可以获得与指定的**分组键**对应的数据。又如：

df_dose.head()

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
1	15.0	0.002	0.004	D	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F

tmt_gender_group.
get_group(("C","F")).head()

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F
18	30.0	4.892	4.851	C	60s	F
21	0.1	0.000	0.000	C	60s	F

注意这里传入的是由两个分组键构成的**元组**

分组—运算—合并

通过GroupBy对象提供的**agg()**、**transform()**、**filter()**以及**apply()**等方法可以实现各种分组运算。

- 每个方法的第一个参数都是一个**回调函数**，该函数对每个分组的数据进行运算并返回结果。(运算)
- 这些方法根据回调函数的返回结果生成最终的分组运算结果。(合并)

分组—运算—合并：agg()—聚合(aggregate)

agg()对每个分组中的数据进行聚合运算。所谓聚合运算是指将一组由N个数值组成的数据转换为单个数值的运算，例如求和、平均值、中间值甚至随机取值等都是聚合运算。

- 其回调函数接收的数据是表示**每个分组中每列数据的Series对象**，若回调函数不能处理Series对象（例如，函数中需要访问多个列），则agg()会接着尝试将整个分组的数据作为DataFrame对象传递给回调函数。
- 回调函数对其参数进行聚合运算，将Series对象转换为单个数值，或将DataFrame对象转换为Series对象。
- agg()返回一个DataFrame对象，其**行索引为每个分组的键**，而列索引为源数据的列索引。

分组—运算—合并: `agg()`—聚合Groupby
对象: g

分组: a

	A	B
0	1	2
2	5	6
3	9	4

分组: b

	A	B
1	7	0
4	3	8

`g.agg(np.max)`

	A	B
a	9	6
b	7	8

`g.agg(lamda df:
df.loc[(df.A+df.B).idxmax()])`

	A	B
a	9	4
b	3	8

对分组中的每列做
运算, 然后聚合对整个分组做运算,
然后聚合

分组—运算—合并: agg()—聚合

例如:

```
agg_res1=tmt_group.agg(np.mean)
```

```
agg_res2=tmt_group.agg(lambda df: df.loc[df.Response1.idxmax()])
```

agg_res1

	Dose	Response1	Response2
Tmt			
A	33.546154	6.728985	6.863185
B	33.546154	5.573354	5.456415
C	33.546154	4.040415	4.115323
D	33.546154	3.320646	3.188369

注意到结果中自动剔除了无法求平均值的字符串列

agg_res2

	Dose	Response1	Response2	Age	Gender
Tmt					
A	80.0	11.226	10.132	60s	F
B	100.0	10.824	10.158	50s	M
C	60.0	10.490	11.218	50s	M
D	80.0	10.911	9.854	60s	F

找到每个分组中 Response1最大的那一行, 结果中包含了源数据中的所有列

分组—运算—合并：transform()—转换

transform()对每个分组中的数据进行转换运算。

- 它将表示每列的Series对象传递给回调函数。
- **回调函数的返回结果与参数的形状相同**，transform()将这些结果**按照源数据的顺序合并**在一起。

分组—运算—合并: transform()—转换

Groupby
对象: g

分组: a

	A	B
0	1	2
2	5	6
3	9	4

分组: b

	A	B
1	7	0
4	3	8

对分组a运算

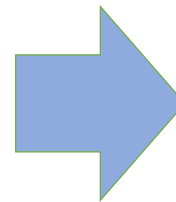
	A	B
0	0	0
2	4	4
3	8	2

`g.transform(lambda s: s - s.min())`

分组中每列元素减去
本列中最小的元素

对分组b运算

	A	B
1	4	0
4	0	8



	A	B
0	0	0
1	4	0
2	4	4
3	8	2
4	0	8

分组—运算—合并: transform()—转换

例如:

分组中各列中的每个元素减去本列平均值

```
transform_res1 = tmt_group.transform(lambda s: s-s.mean())
```

transform_res1.head()

	Dose	Response1	Response2
0	16.453846	5.831585	5.916677
1	-18.546154	-3.318646	-3.184369
2	-8.546154	-3.414415	-3.312323
3	-8.546154	-2.668415	-2.558323
4	-18.546154	-4.030415	-4.095323

df_dose.head()

	Dose	Response1	Response2	Tmt	Age	Gender
0	50.0	9.872	10.032	C	60s	F
1	15.0	0.002	0.004	D	60s	F
2	25.0	0.626	0.803	C	50s	M
3	25.0	1.372	1.557	C	60s	F
4	15.0	0.010	0.020	C	60s	F

tmt_group.agg(np.mean) # 分组平均

	Dose	Response1	Response2
Tmt			
A	33.546154	6.728985	6.863185
B	33.546154	5.573354	5.456415
C	33.546154	4.040415	4.115323
D	33.546154	3.320646	3.188369

分组—运算—合并：filter()—分组过滤

filter()方法对每个分组进行条件判断并过滤分组。

- 它将表示每个分组的**DataFrame对象传递给回调函数**，该函数返回True或False，以决定是否保留该分组。
- filter()的返回结果是过滤掉一些行之后的DataFrame对象，其行索引与源数据的行索引的顺序一致。

例如：筛选出Response1列的最大值大于11的分组。

```
filter_res1 = tmt_group.filter(lambda df: df.Response1.max() > 11)
```

filter_res1.head()							df_dose.head()						
	Dose	Response1	Response2	Tmt	Age	Gender		Dose	Response1	Response2	Tmt	Age	Gender
6	1.0	0.000	0.000	A	50s	F	0	50.0	9.872	10.032	C	60s	F
10	15.0	5.225	5.163	A	60s	F	1	15.0	0.002	0.004	D	60s	F
12	5.0	0.000	0.001	A	60s	F	2	25.0	0.626	0.803	C	50s	M
17	5.0	0.000	0.003	A	50s	M	3	25.0	1.372	1.557	C	60s	F
32	100.0	9.295	10.103	A	60s	F	4	15.0	0.010	0.020	C	60s	F

各分组最大值

```
tmt_group.agg(np.max)
```

	Dose	Response1	Response2	Age	Gender
Tmt					
A	100.0	11.226	10.745	60s	M
B	100.0	10.824	10.340	60s	M
C	100.0	10.490	11.246	60s	M
D	100.0	10.911	9.863	60s	M

分组—运算—合并：apply ()—应用（某种运算函数）

apply()方法将表示每个分组的DataFrame对象传递给回调函数并收集其返回值，并将这些返回值按照某种规则合并。

- apply()的用法十分灵活，可以实现上述agg()、transform()和filter()方法的功能。
- 它会根据回调函数的返回值的类型选择恰当的合并方式。

例如：

- **g.apply(pd.DataFrame.max)**：回调函数为DataFrame.max，它计算DataFrame对象中每列的最大值，返回一个以列名为索引的Series对象，因此对于所有的分组数据返回的索引都是相同的。这种情况下apply()的结果与**g.agg(np.max)**相同。
- 若数据框各列全部为数值类型时，**g.apply(lambda s: s - s.min())**等效于 **g.transform(lambda s: s - s.min())**

```
df = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7, 8]],
```

```
                  columns=list('AB'))
```

```
g = df.groupby(list('abba')) # 分成a、b两组
```

```
print(g.apply(lambda s: s - s.min()))
```

```
print(g.transform(lambda s: s - s.min())) # 效果同上
```

df

	A	B
0	1	2
1	3	4
2	5	6
3	7	8

g.apply
→
g.transform

	A	B
0	0	0
1	0	0
2	2	2
3	6	6

分组—运算—合并：apply ()—应用（某种运算函数）

当回调函数返回None时，将忽略该返回值，因此可以实现filter()的功能。

例如：

- `tmt_group.apply(lambda df: df if df.Response1.max() > 11 else None)`
等效于 `tmt_group.filter(lambda df: df.Response1.max() > 11)`
- `tmt_group.apply(lambda df: df.sample(2) if df.Response1.max()>11 else None)`

从Response1的最大值大于11的分组中随机取两行数据

		Dose	Response1	Response2	Age	Gender
Tmt						
A	235	60.0	9.825	10.465	50s	M
	35	60.0	10.399	10.131	60s	F

apply()可以实现更加灵活的功能。

例如：

`tmt_group.apply(lambda df:
df.sample(2))`

从各分组中随机抽取两行

		Dose	Response1	Response2	Age	Gender
Tmt						
A	213	10.0	0.264	0.239	50s	M
	135	100.0	8.934	10.217	50s	M
B	13	40.0	9.676	10.218	60s	F
	31	15.0	0.398	0.528	60s	F
C	79	30.0	2.390	2.487	60s	M
	165	60.0	10.490	11.218	50s	M
D	49	50.0	9.219	9.255	60s	F
	85	10.0	0.000	0.000	50s	M

第十五章 Pandas数据分析

15.1 Pandas中的数据对象

15.2 下标存取

15.3 文件输入输出

15.4 数值运算函数、字符串处理及NaN处理

15.5 改变DataFrame的形状

15.6 分组运算

15.7 数据处理和可视化实例

两个用Pandas分析实际数据的例子。可以使用 Pandas提供的绘图方法 `plot()` 将计算结果显示为图表，其内部使用 `matplotlib` 绘图。

- Case 1: 分析Pandas项目的提交历史
- Case 2: 分析城市空气质量数据

感谢聆听

