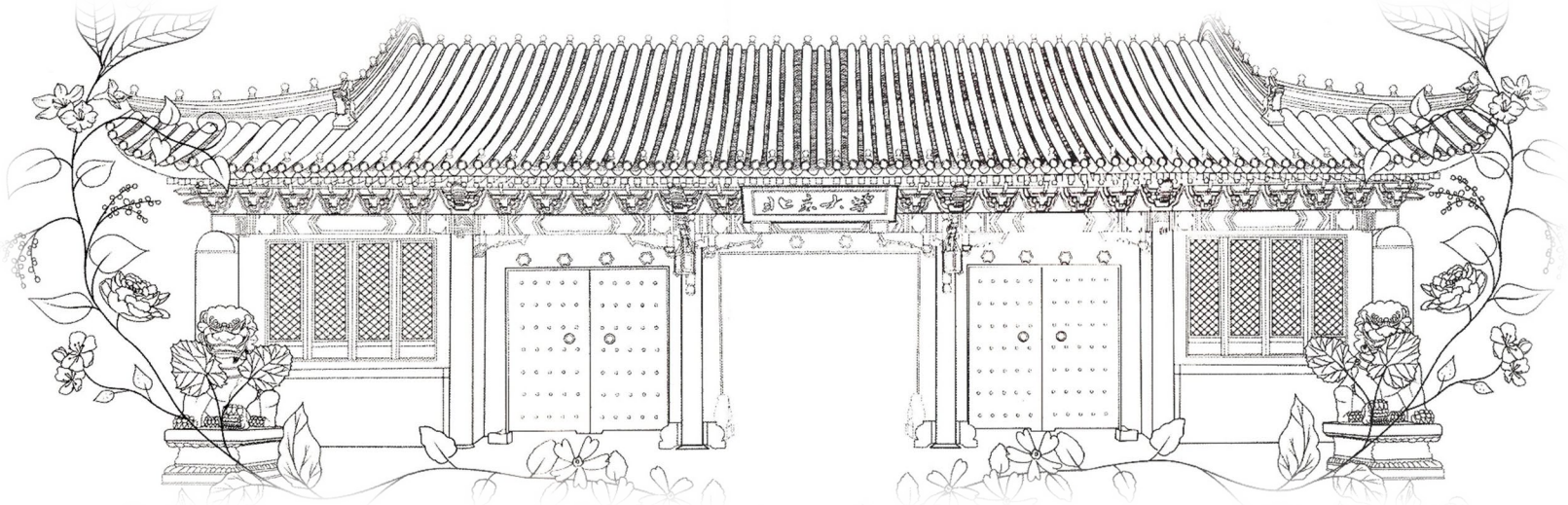
第 1 章 Python数据分析基础 II

北京大学信息管理系

2021/3/7



Pandas简介

Pandas (Python Data Analysis Library) 是基于NumPy 的一种工具，该工具是**为了解决数据分析任务而创建**的。

Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。Pandas提供了大量能使我们快速便捷地处理数据的函数和方法。

它是使Python成为强大而高效的数据分析环境的重要因素之一。

Pandas简介

Pandas致力于解决的问题包括：

- 支持自动或明确的数据对齐的带有标签轴的数据结构。这可以防止由数据不对齐引起的常见错误，并可以处理不同来源的不同索引数据。
- 整合的时间序列功能。
- 以相同的数据结构来处理时间序列和非时间序列。
- 支持传递元数据（坐标轴标签）的算术运算和缩减。
- 灵活处理丢失数据。
- 在常用的基于数据库（例如基于SQL）中的合并和其它关系操作。

本节内容

01 Pandas数据结构

02 基本功能

03 描述性统计

04 缺失值处理

05 层次化索引

06 其他技巧

本节内容

01 Pandas数据结构

02 基本功能

03 描述性统计

04 缺失值处理

05 层次化索引

06 其他技巧

Pandas数据结构

为了开始使用Pandas，需要熟悉它的两个重要的数据结构：

- **Series**
- **DataFrame**

虽然它们不是每一个问题的通用解决方案，但提供了一个坚实的，易于使用的大多数应用程序的基础。

引用规范：

```
In [1]: from pandas import Series, DataFrame
In [2]: import pandas as pd
```

Series

- Series是一个一维的**类似数组的对象**，包含一个数组的**数据**（任何NumPy的数据类型）和一个与数组关联的**数据标签**，被叫做 **索引** 。最简单的Series是由一个数组的数据构成：

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series

- Series的交互式显示的字符串表示形式是**索引在左边，值在右边**。因为我们没有给数据指定索引，一个包含整数0到 N-1 （这里N是数据的长度）的默认索引被创建。 可以分别的通过它的 values 和 index 属性来获取Series的数组表示和索引对象：

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Series

- 可以创建一个**带有索引来确定每一个数据点的Series**：

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Series

- 与正规的NumPy数组相比，**可以使用索引里的值来选择**一个单一值或一个值集：

```
In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64
```

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

Series

- NumPy数组操作，例如通过一个布尔数组过滤，纯量乘法，或使用数学函数，将会**保持索引和值间的关联**：

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64
```

```
In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64
```

```
In [23]: np.exp(obj2)
Out[23]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Series

- 另一种思考的方式是，Series是一个定长的，有序的字典，因为它把索引和值映射起来了。它可以适用于许多期望一个字典的函数：

```
In [24]: 'b' in obj2
Out[24]: True


In [25]: 'e' in obj2
Out[25]: False
```

Series

- 如果你有一些数据在一个Python字典中，你**可以通过传递字典来从这些数据创建一个Series**：

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

Series

➢ 只传递一个字典的时候，结果Series中的**索引将是排序后的字典的键**。

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

在这种情况下，**sdata** 中的3个值被放在了合适的位置，但因为没有发现对应于 **‘California’** 的值，就出现了 **NaN** （不是一个数），这在pandas中被用来标记数据缺失或 NA 值。

Series

- 在pandas中用函数 isnull 和 notnull 来检测数据丢失：

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series

- Series也提供了这些函数的**实例方法**：

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

Series

- 在许多应用中，Series的一个重要功能是在算术运算中它会**自动对齐不同索引的数据**：

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California        NaN
Ohio          70000.0
Oregon        32000.0
Texas        142000.0
Utah              NaN
dtype: float64
```

Series

- Series对象本身和它的索引都有一个 name 属性，它和pandas的其它一些关键功能整合在一起：

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series

- Series的索引可以通过赋值就地更改：

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64
```

```
In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

- 一个**DataFrame表示一个表格**，类似电子表格的数据结构，包含一个经过排序的列表集，它们每一格都可以有不同的类型值（数字，字符串，布尔等等）。
- DataFrame有**行和列的索引**；它**可以被看作是一个Series的字典**（每个Series共享一个索引）。与其它类似data frame的结构相比（如 R 的 data.frame），**在DataFrame里的面向行和面向列的操作大致是对称的**。
- 在底层，数据是作为一个或多个二维数组存储的，而不是列表，字典，或其它一维的数组集合。
- 因为DataFrame在内部把数据**存储为一个二维数组的格式**，因此**可以采用分层索引以表格格式来表示高维的数据**。分层索引是pandas中许多更先进的数据处理功能的关键因素。

DataFrame

- 由此产生的DataFrame和Series一样，它的索引会自动分配，并且对列进行了排序：

```
data = {‘pop’: [1.5, 1.7, 3.6, 2.4, 2.9, 3.2],
'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003]}
frame = pd.DataFrame(data)
```

```
In [45]: frame
Out[45]:
   pop    state  year
0  1.5     Ohio  2000
1  1.7     Ohio  2001
2  3.6     Ohio  2002
3  2.4   Nevada  2001
4  2.9   Nevada  2002
5  3.2   Nevada  2003
```

DataFrame

- 对于较大的DataFrame，可以调用head方法只显示前五行数据：

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

DataFrame

- 如果设定了一个列的顺序，DataFrame的列将会精确地按照你所传递的顺序排列：

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

DataFrame

- 和Series一样，如果传递了一个行，但不包括在 data 中，在结果中它会表示为NaN值：

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])
In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN
In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame

- 和Series一样，在DataFrame中的一列可以通过字典记法或属性来检索：

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

```
In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

注意，返回的Series包含和DataFrame相同的索引，并它们的 **name** 属性也被直接设置为索引名字。

frame2[column]方式对所有列名都适用，但frame2.column方式则要求列名为有效的python变量名。

DataFrame

- 行也可以使用一些方法通过位置或名字来检索，例如 **loc属性**索引成员：

```
In [53]: frame2.loc['three']
Out[53]:
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three, dtype: object
```

DataFrame

- **列可以通过赋值来修改**。例如，空的‘debt’列可以通过一个纯量或一个数组来赋值：

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5
```

```
In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

DataFrame

- 通过**列表或数组**给一列赋值时，所赋的值的长度必须和DataFrame的**长度相匹配**。
- 如果你使用**Series**来赋值，它会代替在DataFrame中**精确匹配的索引的值**，并在所有的空缺位置插入NaN：

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

DataFrame

- **给一个不存在的列赋值，将会创建一个新的列。** 像字典一样 del 关键字将会删除列：

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

DataFrame

- 像字典一样 del 关键字将会删除列：

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

- 索引DataFrame时返回的列是底层数据的一个视图（view），而不是一个拷贝。因此，任何在Series上的就地修改都会影响DataFrame。列可以使用Series的 **copy** 函数来显式的拷贝。

DataFrame

- 另一种通用的数据形式是一个字典嵌套格式：

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
    ....:       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

- 如果被传递到DataFrame，它的**外部键**会被解释为**列索引**，**内部键**会被解释为**行索引**：

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

DataFrame

- 内部字典的键被结合并排序来形成结果的索引。如果指定了一个特定的索引，则根据指定的索引生成。

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

- 总是可以对DataFrame**转置**：

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

DataFrame

- 如果一个DataFrame的 index 和 columns 有它们的 name属性，也会被显示出来：

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

DataFrame

- 像Series一样， values 属性返回一个包含在DataFrame中的数据的二维ndarray：

```
In [74]: frame3.values
Out[74]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

- 如果DataFrame的列有不同的dtypes，返回值数组将会给所有的列选择一个合适的dtypes：

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

DataFrame

- 可以传递到DataFrame构造器的对象的完整列表如下

可能的传递到DataFrame的构造器	
二维ndarray	一个数据矩阵，有可选的行标和列标
数组，列表或元组的字典	每一个序列成为DataFrame中的一列。所有的序列必须有相同的长度。
NumPy的结构/记录数组	和“数组字典”一样处理
Series的字典	每一个值成为一列。如果没有明显的传递索引，将结合每一个Series的索引来形成结果的行索引。
字典的字典	每一个内部的字典成为一列。和“Series的字典”一样，结合键值来形成行索引。
字典或Series的列表	每一项成为DataFrame中的一列。结合字典键或Series索引形成DataFrame的列标。
列表或元组的列表	和“二维ndarray”一样处理
另一个DataFrame	DataFrame的索引将被使用，除非传递另外一个
NumPy伪装数组（MaskedArray）	除了蒙蔽值在DataFrame中成为N/A丢失数据之外，其它的和“二维ndarray”一样

索引对象

- Pandas的索引（Index）对象用来保存坐标轴标签和其它元数据（如坐标轴名或名称）。构建一个Series或DataFrame时任何数组或其它序列标签在内部转化为索引：

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

索引对象

- 索引对象是不可变的，因此不能由用户改变：

```
index[1] = 'd'  # TypeError
```

```
---------------------------------------------------------------------------
Exception Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302 def __setitem__(self, key, value):
    303 """Disable the setting of values."""
--> 304 raise Exception(str(self.__class__) + ' object is immutable')
    305
    306 def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

索引对象

- 索引对象的不可变性非常重要，这样它可以在数据结构中结构中安全的共享：

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64
```

```
In [84]: obj2.index is labels
Out[84]: True
```

索引对象

- Index 对象除了像数组一样访问，还可以像**固定大小的集合**：

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

```
In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

索引对象

- 需要注意的是，与Python内置的集合不同，pandas的index可以包含重复值。

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

索引对象

- 每个索引都有许多关于集合逻辑的方法和属性，且能够解决它所包含的数据的常见问题。详见下表：

索引方法和属性	
append	链接额外的索引对象，产生一个新的索引
diff	计算索引的差集
intersection	计算交集
union	计算并集
isin	计算出一个布尔数组表示每一个值是否包含在所传递的集合里
delete	计算删除位置i的元素的索引
drop	计算删除所传递的值后的索引
insert	计算在位置i插入元素后的索引
is_monotonic	返回True，如果每一个元素都比它前面的元素大或相等
is_unique	返回True，如果索引没有重复的值
unique	计算索引的唯一值数组

- **一般不必要知道许多索引对象的知识**，但是它们仍然是pandas数据模型的重要部分。

本章内容

基本功能

- 在本节中，将介绍Series或DataFrame所包含的数据的基础结构的相互关系。
- 在接下来的章节中，将要更深入的探究使用pandas进行数据分析和处理的主题。
- 本节的注意力集中在最重要的特性上
- 不常见的内容，可以在使用中查询学习。

重新索引

- pandas对象的一个关键的方法是 reindex，意味着**使数据符合一个新的索引来构造一个新的对象**。来看一下下面一个简单的例子：

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
Out[92]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

重新索引

- 在Series上调用 reindex 重排数据，使得它符合新的索引，如果那个索引的值不存在就引入缺失数据值：

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

重新索引

- 为了对时间序列这样的数据排序，当重建索引的时候可能想要对值进行插值或填充。这时可以选用**method** 选项，使用一个如 **ffill** 的方法来向前填充值：

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

```
In [96]: obj3
Out[96]:
0      blue
2    purple
4    yellow
dtype: object
```

```
In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

重新索引

- 下表是可用的 method 选项的清单。在此，内差比正向和反向填充更复杂。

reindex 的 method（内插）选项	
参数	描述
ffill或pad	前向（或进位）填充
bfill或backfill	后向（或进位）填充

重新索引

- 使用DataFrame时，reindex可以改变（行）索引，列（索引），或者都改变。当只传一个序列参数时，将改变（行）索引：

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8
```

重新索引

- 使用DataFrame时，reindex可以改变（行）索引，列（索引），或者都改变。当只传一个序列参数时，将改变（行）索引：

```
In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2
Out[101]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

重新索引

- 使用 columns 关键字则可以是列重新索引：

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

重新索引

- 另外同样可以使用loc 方式进行重新索引：

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
   Texas  Utah  California
a    1.0   NaN         2.0
b    NaN   NaN         NaN
c    4.0   NaN         5.0
d    7.0   NaN         8.0
```

重新索引

reindex 函数的参数	
index	作为索引的新序列。可以是索引实例或任何类似序列的Python数据结构。一个索引被完全使用，没有任何拷贝。
method	插值（填充）方法，见表中的选项
fill_value	代替重新索引时引入的缺失数据值
limit	当前向或后向填充时，最大的填充间隙
level	在多层索引上匹配简单索引，否则选择一个子集
copy	如果新索引与就的相等则底层数据不会拷贝。默认为True(即始终拷贝）

从一个坐标轴删除条目

- 从坐标轴删除一个多或多个条目是很容易的，当有一个索引数组或列表且存储需要删除当条目或需要保留当条目，但是这可能需要一点修改和集合逻辑。 drop 方法将会返回一个新的对象并从坐标轴中删除指定的一个或多个值：

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

```
In [109]: obj.drop(['d', 'c'])
Out[109]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

从一个坐标轴删除条目

- 对于DataFrame，可以从任何坐标轴删除索引值：

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

从一个坐标轴删除条目

- 对于DataFrame，删除列需要传递参数axis=1或axis="columns"：

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

从一个坐标轴删除条目

- 像drop一样的很多函数，**可以原地修改dataframe**，进而**不返回任何值**。这时需要指定inplace参数为True：

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

- 使用inplace时**一定要谨慎，因为这会改变原始数据**。

索引、选择和过滤

- Series索引(obj[...])的工作原理类似与NumPy索引，除了可以使用Series的**索引值**，也可以仅使用**整数**来索引。下面是关于这一点的一些例子：

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

索引、选择和过滤

- 使用标签来切片和正常的Python切片并不一样，它会把结束点也包括在内：

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

- 使用这些函数来复制，其工作方法和你想象的一样：

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

索引、选择和过滤

- 采用索引方式访问DataFrame将返回若干列：

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

索引、选择和过滤

- 采用索引方式访问DataFrame将返回若干列：

```
In [130]: data['two']
Out[130]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

```
In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

索引、选择和过滤

- 像这样的索引有一些特殊的情况。首先，可以通过一个布尔数组来切片或者选择：

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
In [133]: data[data['three'] > 5]
Out[133]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

索引、选择和过滤

- 像这样的索引有一些特殊的情况。首先，可以通过一个布尔数组来切片或者选择：

```
In [134]: data < 5
Out[134]:
             one    two  three   four
Ohio        True   True   True   True
Colorado    True  False  False  False
Utah       False  False  False  False
New York   False  False  False  False
```

```
In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

索引、选择和过滤

- **使用loc 和iloc进行索引**:
- 这样可以采用类似numpy的访问方式进行数据切片。
- loc通过轴标签访问

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

索引、选择和过滤

- **使用loc 和iloc进行索引**:
- iloc通过整数下标访问

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

```
In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64
```

索引、选择和过滤

- **使用loc 和iloc进行索引**:
- loc 和 iloc 都像切片一样工作，且能传递单标签或者一系列标签:

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64
```

```
In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

索引与重排小结

- 因此，有很多方法来选择和重排包含在pandas对象中的数据。对于DataFrame， 下表是这些方法的简短概要。

df[val]	从DataFrame选择单一列或连续列。特殊情况下的便利：布尔数组（过滤行），切片（行切片），或布尔DataFrame（根据一些标准来设置值）。
df.loc[label]	通过标签从DataFrame的行集选择单行或多行
df.loc[:, label]	通过标签从DataFrame的行集选择单列或多列
df.loc[label_1, label_2]	通过标签同时选择行和列
df.iloc[where]	通过数字下标选择单行或多行
df.iloc[:, where]	通过数字下标选择单列或多列
df.iloc[where_i, where_j]	通过数字下标选择若干行和若干列
df.at[label_i, label_j]	通过标签获取某个位置的值
df.iat[i, j]	通过数字下标获取某个位置的值
reindex 方法	通过标签访问行或者列

算术和数据对齐

- Pandas的最重要的特性之一是在具有不同索引的对象间进行算术运算的行为。当把对象加起来时，如果有任何的索引对不相同的话，在结果中将会把各自的索引联合起来。让我们看一个简单的例子：

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64

In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

内部数据对其，在索引不重合的地方引入了NA值。数据缺失在算术运算中会传播。

算术和数据对齐

- 对于DataFrame，对其在行和列的对齐上都表现都很好：

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                    index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [157]: df1
Out[157]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
```

```
In [158]: df2
Out[158]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

```
In [159]: df1 + df2
Out[159]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

把这些加起来返回一个DataFrame，它的索引和列是每一个DataFrame对应的索引和列的联合，缺失值也传播

算术和数据对齐

- 如果将两个没有任何重复列或行的DataFrame相加，将得到一个空DataFrame：

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})
```

```
In [162]: df1
Out[162]:
   A
0  1
1  2
```

```
In [163]: df2
Out[163]:
   B
0  3
1  4
```

```
In [164]: df1 - df2
Out[164]:
     A   B
0 NaN NaN
1 NaN NaN
```

带填充值的算术方法

- 在不同索引对象间的算术运算，当一个轴标签在另一个对象中找不到时，可以选择填充一个特定的值，如0:

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [169]: df2
Out[169]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

带填充值的算术方法

- 在不同索引对象间的算术运算，当一个轴标签在另一个对象中找不到时，可以选择填充一个特定的值，如0：

```
In [170]: df1 + df2
Out[170]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

带填充值的算术方法

- 内置算数方法列表

Method	Description
add, radd	Methods for addition (+)
sub, rsub	Methods for subtraction (-)
div, rdiv	Methods for division (/)
floordiv, rfloordiv	Methods for floor division (//)
mul, rmul	Methods for multiplication (*)
pow, rpow	Methods for exponentiation (**)

- 每两个算法方式是一对，表示相同的算数计算逻辑
- 其中每个以r开头的，表示交换运算变量的顺序

带填充值的算术方法

- 内置算数方法列表

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

带填充值的算术方法

- 使用reindex时，可以调用fill_value进行填充

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

DataFrame 和 Series 间的操作

- 就像具有不同维度的NumPy数组一样，DataFrame和Series间的算术操作也有类似的原则。首先，作为一个启发性的例子，考虑一个二维数组和它的一个行间的减法：

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])
```

```
In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

这种操作被称为广播 (broadcasting)

DataFrame 和 Series 间的操作

- 在一个DataFrame和一个Series间的操作是类似的：

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]
```

```
In [181]: frame
Out[181]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

```
In [182]: series
Out[182]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

DataFrame 和 Series 间的操作

- 默认的，DataFrame和Series间的算术运算Series的索引将匹配DataFrame的列，并在行上扩展：

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

- 如果一个索引值在DataFrame的列和Series的索引里都找不着，对象将会从它们的联合重建索引：

```
In [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [185]: frame + series2
Out[185]:
          b   d    e   f
Utah    0.0 NaN  3.0 NaN
Ohio    3.0 NaN  6.0 NaN
Texas   6.0 NaN  9.0 NaN
Oregon  9.0 NaN 12.0 NaN
```

DataFrame 和 Series 间的操作

- 如果想在行上而不是列上进行扩展，要使用一个算术方法，并指定轴。例

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

```
In [188]: series3
Out[188]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

```
In [189]: frame.sub(series3, axis='index')
Out[189]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

- 或者指定axis=0

函数应用和映射

- Numpy中元素级别的函数（ufun）可以直接应用于pandas对象

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [191]: frame
Out[191]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221
```

```
In [192]: np.abs(frame)
Out[192]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

函数应用和映射

➢ 可以将函数应用于某一个维度。这时需要利用DataFrame的apply方法：

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

- 默认情况下，函数f作用于与DataFrame的每一列，
- 所得的结果为一个以DataFrame的columns为index的Series

- 如果想作用于每一行，可以再调用apply时，设定参数axis='colums'或者axis=1

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

- DataFrame中内置有很多通用的函数，如sum、mean等，
- 这些函数不需要apply，直接调用即可。

函数应用和映射

- 传入apply方法的函数同样可以返回Series：

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
Out[197]:
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

```
In [191]: frame
Out[191]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221
```

函数应用和映射

- 如果函数要应用于DataFrame的每一个元素，则可采用applymap方法：

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
             b     d      e
Utah     -0.20  0.48  -0.52
Ohio     -0.56  1.97   1.39
Texas     0.09  0.28   0.77
Oregon    1.25  1.01  -1.30
```

- 对于Series，有类似的map方法作用于每个元素：

```
In [200]: frame['e'].map(format)
Out[200]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

排序与排名

- Pandas内置了排序（Sorting）功能。如可通过sort_index按索引排序：

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```

排序与排名

➢ DataFrame可以用根据任意一个轴的索引进行排序：

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
In [205]: frame.sort_index(axis=1)
Out[205]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

➢ 排序默认是按升序，但也可以采用降序：

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

排序与排名

- 对Series的值进行排序，可以采用order方法：

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2   -3
3    2
0    4
1    7
dtype: int64
```

- 缺失值将默认排在最后：

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

排序与排名

- 在DataFrame中，有序需要根据一个或多个列排序，这是可以使用参数by将列名作为参数值传递：

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [212]: frame
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

- 注意将列名放在一个list中

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

排序与排名

- 在排名（Ranking）时，使用rank方法，会增加一个从1开始，到有效数据数量的排名值（默认对于大小相同的值，取该值的所有样本取排名的平均值）：

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

- 也可用method参数指定按照出现的顺序对相同大小给出排名值

```
In [217]: obj.rank(method='first')
Out[217]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

排序与排名

- 同样可以进行倒序排名，max参数还能指定最后出现位置作为排名：

```
# Assign tie values the maximum rank in the group
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

- 以下表列出了所有平级关系的method选项

Table 5-6. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group
'min'	Use the minimum rank for the whole group
'max'	Use the maximum rank for the whole group
'first'	Assign ranks in the order the values appear in the data
'dense'	Like method='min', but ranks always increase by 1 in between groups rather than the number of equal elements in a group

排序与排名

- DataFrame可以在行或者列上计算排名：

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})
```

```
In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5
```

```
In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

有重复值的轴索引

- 在Pandas中，并不强制限制索引值的唯一性：

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

- 可以通过is_unique属性判断索引是否唯一：

```
In [224]: obj.index.is_unique
Out[224]: False
```

- 不唯一的索引返回Series，唯一对应的返回标量：

```
In [225]: obj['a']
Out[225]:
a    0
a    1
dtype: int64
```

```
In [226]: obj['c']
Out[226]: 4
```

有重复值的轴索引

- DataFrame的行索引也有类似的操作逻辑：

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

本节内容

01 Pandas数据结构

02 基本功能

03 描述性统计

04 缺失值处理

05 层次化索引

06 其他技巧

汇总和描述性统计

- Pandas对象有一组常用的数学和统计方法。其中大多数属于数据归约和描述性统计，即从Series提取单个值（如min、max等），或从DataFrame的行或列中提取出一个Series。与numpy对比，Pandas的这些方法都内置的对缺失值的自动处理：

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                    [np.nan, np.nan], [0.75, -1.3]],
   .....:                   index=['a', 'b', 'c', 'd'],
   .....:                   columns=['one', 'two'])

In [231]: df
Out[231]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

汇总和描述性统计

- DataFrame的sum方法默认返回一个统计每列之和的Series：

```
In [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

- 可以通过axis参数改变轴：

```
In [233]: df.sum(axis='columns')
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

- 可以通过skipna选择是否跳过NaN：

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

汇总和描述性统计

- 数据归约计算的参数列表如下：

Table 5-7. Options for reduction methods

Method	Description
`axis`	Axis to reduce over; 0 for DataFrame's rows and 1 for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

汇总和描述性统计

- 有些归约方法，返回间接统计（如最大值和最小值的索引）：

```
In [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

- 有些方法属于累积方法：

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

- 此外，还有非归约非累积方法，如describe：

```
In [237]: df.describe()
Out[237]:
             one       two
count   3.000000  2.000000
mean    3.083333 -2.900000
std     3.493685  2.262742
min     0.750000 -4.500000
25%     1.075000 -3.700000
50%     1.400000 -2.900000
75%     4.250000 -2.100000
max     7.100000 -1.300000
```

汇总和描述性统计

- 对于非数组型数据，describe返回相应的统计量：

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count     16
unique     3
top        a
freq       8
dtype: object
```

汇总和描述性统计

- 汇总和描述性统计相关方法列表如右表所示：

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

汇总和描述性统计

- 相关性和协方差：
  - 引入数据

pip install pandas_datareader

>>>import yfinance as yf
>>>yf.pdr_override()

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.items()})
```

汇总和描述性统计

- 相关性和协方差：
  - 计算价格变化

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

汇总和描述性统计

- 相关性和协方差：
  - Series的corr方法返回两个Series中，重叠的、非NaN的对齐值的相关系数；
  - Series的cov方法返回相应的协方差；

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05

In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

汇总和描述性统计

- 相关性和协方差：
  - DataFrame的corr方法和cov方法返回相关系数矩阵和协方差矩阵：

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000
In [248]: returns.cov()
Out[248]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

汇总和描述性统计

- 相关性和协方差：
  - DataFrame的corrwith方法返回DataFrame的每行或每列与另一个Series或DataFrame的相关系数。
  - 传入Series则计算原DataFrame与新Series的相关系数，并返回一个Series：

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

汇总和描述性统计

- 相关性和协方差：
  - DataFrame的corrwith方法返回DataFrame的每行或每列与另一个Series或DataFrame的相关系数。
  - 传入DataFrame则默认计算按列配对的相关系数（可通过axis参数进行行列指定，如axis='columns'）：

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

唯一值、计数和成员判断

- 唯一值unique()：

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

- 计算Series每个元素出现次数，采用value_counts()：

```
In [254]: obj.value_counts()
Out[254]:
c    3
a    3
b    2
d    1
dtype: int64
```

- value_counts()作为Pandas顶层方法，可以作用与任何数组或序列：

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a    3
b    2
c    3
d    1
dtype: int64
```

唯一值、计数和成员判断

- isin判断矢量化集合的成员资格，并可利用它来进行DataFrame的下采样：

```
In [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

```
In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

```
In [259]: obj[mask]
Out[259]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

汇总和描述性统计

- 与isin相似的是Index.get_indexed方法，可以进行有重复到无重复序列的转换：

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

汇总和描述性统计

- 唯一值、计数、分箱方法汇总：

Table 5-9. Unique, value counts, and set membership methods

Method	Description
`isin`	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
`match`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

汇总和描述性统计

- 计算关于列的柱状图的例子：

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})
```

```
In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

- 每一row代表一个DataFrame中出现过的唯一值；
- 每个的取值为该位置的取值出现的次数。

本节内容

01 Pandas数据结构

02 基本功能

03 描述性统计

04 缺失值处理

05 层次化索引

06 其他技巧

缺失值处理

- 缺失值在大多数据数据中都存在，Pandas的设计目标之一就是让缺失值处理更加轻松。例如，很多方法已经自动的排除了缺失值。
- 此外，还提供了一些根据针对性的的缺失值处理方法，如缺失值监测：

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

```
In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

```
In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

缺失值处理

- Python内置的None值，也被当做NaN处理：

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
1    False
2     True
3    False
dtype: bool
```

缺失值处理

- Pandas缺失值处理方法汇总：

Table 7-1. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

缺失值处理

- Series采用dropna()滤除缺失值：

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

缺失值处理

- DataFrame则默认剔除所有包含缺失值的行：

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
    ....:                     [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()
```

```
In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

```
In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

- 使用参数how=‘all’只剔除全为NaN的行：

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

缺失值处理

- 可以采用axis参数确定按某个轴剔除：

```
In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA
```

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

缺失值处理

- 缺失值填充：

```
In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

缺失值处理

- 缺失值填充：
  - 可以对每列填充不同值：

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

- 默认返回新对象，但也可以指定原地修改：

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

缺失值处理

- 缺失值填充：
  - 默认返回新对象，但也可以指定原地修改：

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA
```

缺失值处理

- 缺失值填充：
  - 默认返回新对象，但也可以指定原地修改：

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232
```

```
In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

缺失值处理

- 对reindex有效的插值方法也可用于fillna：

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

缺失值处理

- fillna()函数的常见参数列表如下：

Table 7-2. fillna function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

本节内容

层次化索引

- 层次化索引（Hierarchical indexing）是Pandas的一项重要功能，使得可以在一个轴上有多个索引级别，从而使得Pandas可以在二维表上处理高维数据：

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])
```

```
In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
```

- 这是一个具有多层次索引的index的格式化输出

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

层次化索引

- 对于层次化索引，可以很方便地选取部分数据：

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64
```

```
In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

- 甚至可以在内层进行选取：

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

层次化索引

- 层次化索引在数据重塑和基于分组的操作（如透视表生成）中扮演重要角色。如可以通过unstack方法重新安排到DataFrame中，相应的逆操作为stack：

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

层次化索引

- 在DataFrame中，每一个轴都可以有层次化索引：

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....:                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   ....:                      columns=[['Ohio', 'Ohio', 'Colorado'],
   ....:                               ['Green', 'Red', 'Green']])

In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

层次化索引

- 每一个层次级别都能有一个名字，并能用于抽取部分数据：

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state       Ohio     Colorado
color      Green Red    Green
key1 key2
a    1         0   1         2
     2         3   4         5
b    1         6   7         8
     2         9  10        11
```

层次化索引

- 使用部分的列索引，同样可以选择部分列：

```
In [23]: frame['Ohio']
Out[23]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

- 多层索引也能直接创建：

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                       names=['state', 'color'])
```

层次化索引

- 重排索引顺序，swaplevel改变级别，不变数据：

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state       Ohio     Colorado
color      Green Red    Green
key2 key1
1    a         0   1         2
2    a         3   4         5
1    b         6   7         8
2    b         9  10        11
```

层次化索引

- Sortlevel则会根据其中某一个level中的值对数据排序：

```
In [25]: frame.sort_index(level=1)
Out[25]:
state       Ohio     Colorado
color      Green Red    Green
key1 key2
a    1         0   1         2
b    1         6   7         8
a    2         3   4         5
b    2         9  10        11
```

```
In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state       Ohio     Colorado
color      Green Red    Green
key2 key1
1    a         0   1         2
     b         6   7         8
2    a         3   4         5
     b         9  10        11
```

层次化索引

- 根据层级进行统计汇总：

```
In [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1          6   8        10
2         12  14        16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

层次化索引

- 使用DataFrame的多个列进行索引：

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
   ....:                       'c': ['one', 'one', 'one', 'two', 'two',
   ....:                             'two', 'two'],
   ....:                       'd': [0, 1, 2, 0, 1, 2, 3]})

In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

层次化索引

- 使用DataFrame的多个列：
  - 使用set_index可以指定若干columns作为index：

```
In [31]: frame2 = frame.set_index(['c', 'd'])
```

```
In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

- 默认会将那些列从DataFrame中，但也可以选择保留移除：

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

层次化索引

- 使用DataFrame的多个列：
  - reset_index和set_index作用相反，可以把层次化索引的级别转化为列：

```
In [34]: frame2.reset_index()
Out[34]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

本节内容

01 Pandas数据结构

02 基本功能

03 描述性统计

04 缺失值处理

05 层次化索引

06 其他技巧