Python 数据分析实践（Data Analysis Action）

Chap 4 pandas数据探索 Data Exploration

内容：

- 数据质量分析：缺失、异常、不一致、重复或特殊符号
- 数据特征分析：分布分析、汇总和描述统计分析、相关性分析

实践：

- pandas的Sries和DataFrame
- pandas进行数据探索

这节课是学习pandas对数据的各种分析探索，目的是深入了解数据。下节课对数据的内容进行预处理以及后面的课程对数据进行挖掘，从目前已有数据中挖掘出知识，用于预测新的数据，都是从深入了解数据开始的。

pandas介绍

pandas是基于NumPy构建的，含有使数据分析工作变得更快更简单的高级数据结构和操作工具。从2008年后，pandas逐渐成长为一个非常大的库，能解决越来越多的数据分析问题。

pandas名字源于panel data（面板数据，是计量经济学中关于多维结构化数据集的一个术语）以及Python data analysis（Python数据分析），很适合金融数据分析应用的工具

- 具有高性能的数组计算功能以及电子表格和关系型数据库（如SQL）灵活的数据处理能力
- 提供了复杂精细的索引功能，更便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作
- 对于金融行业的用户，pandas提供了大量适用于金融数据的高性能时间序列功能和工具

我们使用下面的pandas引入约定：

```
import pandas as pd
from pandas import Series, DataFrame
```

因为Series和DataFrame用的次数非常多，因此将其引入本地命名空间中会更方便

准备工作：导入库，配置环境等

In [1]:
```
from __future__ import division
import os, sys

# 导入库并为库起个别名
import numpy as np
import pandas as pd
from pandas import Series, DataFrame

# 启动绘图
%matplotlib inline
import matplotlib.pyplot as plt

# 常用全局配置
np.random.seed(12345)
np.set_printoptions(precision=4)
plt.rc('figure', figsize=(10, 6))
```

pandas的数据结构

pandas中两个最主要的数据结构就是：Series 和 DataFrame，为大多数应用提供了一种可靠的、易于使用的基础。

1. Series

Series是一种类似于**一维数组**的对象，它由一组数据（各种NumPy数据类型）以及一组与之相关的数据标签（即索引）组成。如下，仅由一组数据即可产生最简单的Series：

- Series的字符串表现形式为：索引在左边，值在右边
- 如果没有为数据指定索引，会自动创建一个0到N-1的（N为数据的长度）的整数型索引。

In [2]:
```
a=[1,3,5]
print(a, type(a))

a = Series(a)
print(a, type(a))
```

```
[1, 3, 5] <class 'list'>
0    1
1    3
2    5
dtype: int64 <class 'pandas.core.series.Series'>
```

In [3]:
```
# 由列表创建一个Series序列，对比下面两个Series序列的不同
s = Series([4, 7, -5, 3])  # 没有指定索引
s = Series([4, 7, -5, 3], index=['1','2','3','4'])  # 指定索引
s
```

Out[3]:
```
1    4
2    7
3   -5
4    3
dtype: int64
```

In [4]:
```
# 可以通过Series的values和index属性获取其数组表示形式和索引对象：
print(s.values)
print(s.index)
```

```
[ 4  7 -5  3]
Index(['1', '2', '3', '4'], dtype='object')
```

In [5]:
```
# 希望所创建的Series带有一个可以对各个数据点进行标记的索引。
# 所以，Series可以看做是一个定长的有序字典。
s2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c']) # 指定索引
print(s2.values)
print(s2.index)
```

```
[ 4  7 -5  3]
Index(['d', 'b', 'a', 'c'], dtype='object')
```

In [6]:
```
# 与普通NumPy数组相比，可以通过索引的方式选取Series中的单个或一组值：
s2['a']
```

Out[6]:
```
-5
```

In [7]:
```
# 修改d索引的值
print(s2['d'])
s2['d'] = 6
s2
```

```
4
```

Out[7]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [8]:
```
# 按索引切片数据
print(s2[['c', 'a', 'd']])
```

```
c    3
a   -5
d    6
dtype: int64
```

- NumPy数组运算（如根据布尔型数组进行过滤、标量乘法、应用数学函数等）都会保留索引和值之间的链接：

In [9]:
```
s2
```

Out[9]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [10]:
```
# 数据过滤、标量乘法等数组运算都会保留索引和值之间的链接：
s2[s2 > 0]
```

Out[10]:
```
d    6
b    7
c    3
dtype: int64
```

In [11]:
```
s2 * 2
```

Out[11]:
```
d    12
b    14
a   -10
c     6
dtype: int64
```

In [12]:
```
np.exp(s2)
```

Out[12]:
```
d      403.428793
b     1096.633158
a        0.006738
c       20.085537
dtype: float64
```

- 还可以把Series看成是一个定长的有序字典，因为它是索引值到数据值的一个映射。它可以用在许多原本需要字典参数的函数中：

In [13]:
```
print('b' in s2)
print('e' in s2)
```

```
True
False
```

In [14]:
```
# 可以直接通过字典来创建Series：# sdata是Python的字典对象
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
s3 = Series(sdata)  # Series中的索引就是原字典的键key（有序排列）

print(s3)
print(s3.index, s3.values)
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
Index(['Ohio', 'Texas', 'Oregon', 'Utah'], dtype='object') [35000 71000 16000  5000]
```

在下面这个例子中， sdata跟states索引相匹配的那3个值会被找出来并放到相应的位置上，但由于'California'所对应的sdata值找不到，所以其结果就是NaN（即"非数字"，not a number，在pandas中它用于表示**缺失**或**NA**值）。

In [15]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
# 寻找跟states索引相匹配的那3个值会被找出来并放到相应的位置上，创建Series序列
s4 = Series(sdata, index=states)
s4
```

Out[15]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [16]:
```
# pandas 的isnull用于检测缺失数据missing data，返回布尔型数组的值
pd.isnull(s4)
```

Out[16]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [17]:
```
# pandas的notnull用于检测非空数据
pd.notnull(s4)
```

Out[17]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series也有类似的实例方法：

- obj.isnull() 等同于 pd.isnull(ojb)
- obj.notnull() 等同于 pd.notnull(ojb)

In [18]:
```
# Series的isnull()方法也可以用于检测缺失数据missing data，返回布尔型数组的值
s4.isnull()
```

Out[18]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [19]:
```
# Series的notnull()方法也可以用于检测非空数据
s4.notnull()
```

Out[19]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

In [20]:
```
# Series 最重要的一个功能是，它在算术运算中会自动对齐不同索引的数据
print(s3, '\n', s4)
# Series在算术运算中自动对齐不同索引的数据
s3 + s4
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
 California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Out[20]:
```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

In [21]:
```
# Series对象本身及其索引都有一个name属性，该属性跟pandas其他的关键功能关系非常密切
s4.name = 'population'
s4.index.name = 'state'
s4
```

Out[21]:
```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

In [22]:
```
# Series的索引可以通过赋值的方式就地修改：
print(s)
print(s.index)
s.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
s
```

```
1    4
2    7
3   -5
4    3
dtype: int64
Index(['1', '2', '3', '4'], dtype='object')
```

Out[22]:
```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

2. DataFrame

> DataFrame是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。

> DataFrame既有行索引也有列索引，它可以被看做由Series组成的字典（共用同一个索引，如序列号，可以认为是行索引）。每列可以看作一个Series。

> 跟其他类似的数据结构相比（如R的data.frame）， DataFrame中面向行和面向列的操作基本上是平衡的。其实，DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或别的一维数据结构）。

注意： 虽然DataFrame是以二维结构保存数据的，仍然可以轻松地将其表示为更高维度的数据（层次化索引的表格型结构，这是pandas中许多高级数据处理功能的关键要素）。

构建DataFrame的方法有很多：（1）最常用的一种是直接传入一个由等长列表或NumPy数组组成的字典。结果DataFrame会自动加上索引（跟Series一样），且全部列会被有序排列。

In [23]:
```
# data是字典，每个key值对应的value是一个list，每个key值和value list对应一列
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data) # DataFrame会自动加上索引（跟Series一样），且全部列会被有序排列
frame
```

Out[23]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

In [24]:
```
# 如果指定了列序列，则DataFrame的列就会按照指定顺序进行排列。
DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[24]:

	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

In [25]:
```
# 与Series一样，如果传入的列在数据中找不到，就会产生NA值：
frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                   index=['one', 'two', 'three', 'four', 'five'])
frame2  # columns是列索引，index是行索引
```

Out[25]:

	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

通过类似字典标记的方法或属性的方式，可以将DataFrame的列获取为一个Series，以下两种方式结果相同，都是将DataFrame的列取出为一个Series：

- 1) 字典标记的方式 `frame['state']`
- 2) 属性的方式 `frame.state`

*注意： *返回的Series用于原DataFrame相同的索引，且其name属性也已经被相应地设置好了。

每个列可以看作是一个Series，即column:Series，其中column是列索引

In [26]:
```
print(frame2.columns)  # 返回所有的列索引
print(frame2.index)  # 返回所有的行索引
```

```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
```

In [27]:
```
frame2['state']  # 使用列索引返回指定列， 对行操作无效
#frame2.state  #  相同
```

Out[27]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

In [28]:
```
frame2.year  # 返回指定列
```

Out[28]:
```
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```

In [29]:
```
# 返回行只能通过位置或名称的方式进行获取，比如用索引字段ix获取某个行：
print(frame2.index)  # 返回所有的行索引
frame2.loc['three'] # 返回某特定索引的行数据
```

```
Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
```

Out[29]:
```
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

In [30]:
```
# 列可以通过赋值的方式进行就地修改。下面的方式对整个列的所有行的值都进行了就地修改
print(frame2)
frame2['debt'] = 16.5 # 给空的 “debt” 列附上一个标量值或一组值，就地修改
frame2
```

```
       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
```

Out[30]:

	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

In [31]:
```
frame2['debt'] = np.arange(5.)*2 # 给debt列附上一组值，就地修改
frame2
```

Out[31]:

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	2.0
three	2002	Ohio	3.6	4.0
four	2001	Nevada	2.4	6.0
five	2002	Nevada	2.9	8.0

注意： 将列表或数组赋值给某个列时，其长度必须跟DataFrame的长度相匹配。如果赋值的是一个Series，就会精确匹配DataFrame的索引，所有的空位都将被填上缺失值：

In [32]:
```
val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
print(frame2)
frame2['debt'] = val  # 精确匹配DataFrame的索引，所有的空位都将被填上缺失值
frame2  # 原来有值的位置也被填上了缺失值，说明Series整体覆盖原值
```

```
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   2.0
three  2002    Ohio  3.6   4.0
four   2001  Nevada  2.4   6.0
five   2002  Nevada  2.9   8.0
```

Out[32]:

	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

In [33]:
```
# 获取DataFrame表格中的值
print(frame2.loc['one']['state']) # 先按行索引，再按列索引
print(frame2['debt'][1])  # 先按列索引，再按行索引
```

```
Ohio
-1.2
```

In [34]:
```
# 按布尔值增加新的列
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

Out[34]:

	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

In [35]:
```
# 关键词del用于删除列
del frame2['eastern']
frame2.columns
```

Out[35]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

In [36]:
```
frame2
```

Out[36]:

	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

注意：通过索引方式返回的列是相应数据的视图，并不是副本。因此，对返回的Series所做的任何就地修改全部都会反映到源DataFrame上。通过Series的copy方法即可显式地复制列。

另一种常见的数据形式是嵌套字典（也就是字典的字典）。如果把它传给DataFrame，它就会被解释为：外层字典的键作为列，内层键则作为行索引。

In [37]:
```
# 嵌套字典（也就是字典的字典）
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
print(pop)
```

```
{'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [38]:
```
# 把嵌套字典传给DataFrame：外层字典的键作为列，内层键则作为行索引。
frame3 = DataFrame(pop)
frame3
```

Out[38]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [39]:
```
# 可以对DataFrame进行转置
frame3.T
```

Out[39]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

In [40]:
```
# 由Series组成的字典差不多也是一样的用法
print(frame3)
print(frame3['Ohio'][:-1])
pdata = {'Ohio': frame3['Ohio'][:-1],
         'Nevada': frame3['Nevada'][:2]}
DataFrame(pdata)
```

```
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
2001    1.7
2002    3.6
Name: Ohio, dtype: float64
```

Out[40]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

In [41]:
```
# 设置frame的index和column的name属性，则会显式出来行和列的名字
print(frame3.index.name)
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

```
None
```

Out[41]:

state	Nevada	Ohio
year		
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [42]:
```
# 跟Series一样，values属性也会以二维ndarray的形式返回DataFrame中的数据
frame3.values
```

Out[42]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

In [43]:
```
# 如果DataFrame的各列的数据类型不同，则值数组的数据类型就会选用能兼容所有列的数据类型。
frame2.values
```

Out[43]:
```
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]], dtype=object)
```

下面的表列出了DataFrame构造函数所能接受的各种数据

表：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元祖组成的字典	每个序列会变成DataFrame中的一列，所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”

类型	说明
由Series组成的字典	每个Series会成为一列。如果没有显式索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元祖组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA或缺失值。

索引对象（Index objects）

pandas的索引对象负责管理轴标签和其他元数据（比如轴名称等）。构建Series或DataFrame时，所用到的任何数组或其他序列的标签都会被转换成一个Index：

- Index对象是不可修改的（immutable），因此用户不能对其进行修改。这个属性非常重要，因为这样才可以使得Index对象在多个数据结构之间实现安全共享。
- Index不可修改的属性非常重要，因为这样才能使得Index对象在多个数据结构之间安全共享

In [44]:
```
obj = Series(list(range(3)), index=['a', 'b', 'c'])
index = obj.index
index
```

Out[44]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [45]:
```
index[1:]
```

Out[45]:
```
Index(['b', 'c'], dtype='object')
```

In [46]:
```
# index是不能修改的，下面的语句会报错，Index对象是不可修改的（immutable）
#index[1] = 'd'
```

In [47]:
```
index = pd.Index(np.arange(3))
s2 = Series([1.5, -2.5, 0], index=index)
s2.index is index
```

Out[47]:
```
True
```

In [48]:
```
index
```

Out[48]:
```
Int64Index([0, 1, 2], dtype='int64')
```

pandas中主要的Index对象

下面的表列出了pandas库中内置的Index类。Index甚至可以被继承从而实现特别的轴索引功能。

类	说明
Index	最泛化的Index对象，将轴标签表示为一个由Python对象组成的NumPy数组
Int64Index	针对整数的特殊Index
MultiIndex	“层次化”索引对象，表示单个轴上的多层索引。可以看做由元组组成的数组
DatetimeIndex	存储纳秒级时间戳（用NumPy的datetime64类型表示）
PeriodIndex	针对Period数据（时间间隔）的特殊Index

除了长得像数组，Index的功能也类似一个固定大小的集合。

- 每个索引都有一些方法和属性，可用于设置逻辑并回答有关该索引所包含数据的常见问题。
- 索引类似集合，集合的方法和属性也基本可以适用Index对象。

Index的方法和属性

方法	说明
append	连接另一个Index对象，产生一个新的Index
diff	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
in/is	计算一个索引各值是否都包含着参数集合中的布尔型数组
delete	删除索引 i 处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引 i 处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is_unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

In [49]:
```
# 计算一个索引各值是否包含参数集合中数值的布尔型数组
print(frame3)
print('Ohio' in frame3.columns)  # 是否在数据的列索引名中
print(2.4 in frame3.values)  # 是否在数据的值中
print(2003 in frame3.index)   # 是否在数据的行索引名中
```

```
state  Nevada  Ohio
year
2001      2.4   1.7
2002      2.9   3.6
2000      NaN   1.5
True
True
False
```

pandas的基本功能（Essential functionality）

下面介绍操作Series和DataFrame中的数据的基本手段，来探索pandas在数据分析和处理方面的功能。

重新索引（Reindexing）

pandas对象的一个重要方法是 reindex， 其作用是创建一个适应新索引的新对象。

- 调用该Series的reindex会根据新索引进行重排。如果某个索引值不存在，就引入缺失值
- fill_value=0 对missing value进行填充，缺失值填充值=0

In [50]:
```
s = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])  # 创建序列s
s2 = s.reindex(['a', 'b', 'c', 'd', 'e']) # 对序列s重排索引得到s2，默认缺失值填充NaN
s2
```

Out[50]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

In [51]:
```
# fill_value=0 对missing value进行填充，缺失值填充值=0
s3 = s.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=-1.)
s3
```

Out[51]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e   -1.0
dtype: float64
```

In [52]:
```
# fill_value=0 对missing value进行填充，缺失值填充值=5.0
s4 = s.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=5.0)
s4
```

Out[52]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    5.0
dtype: float64
```

In [53]:
```
#s2.reindex?
```

- 对于时间序列这样的有序数据，重新索引时需要做插值处理，method选项可达到这个目的。
- reindex()的参数 method='ffill'表示可以实现前向值填充，将前一个值传播给下一个元素，propagate last valid observation forward to next valid

In [54]:
```
s3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
s3.reindex(list(range(6)), method='ffill') # 前向值填充，即将前一个值传播给下一个元素
```

Out[54]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

下面的表格列出了reindex可用的method选项。

有时我们可能需要比前向和后向填充更为精准的插值方式。

```
Series.reindex(range(6), method='ffill')

method= {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}
```

参数	说明
default	不填充（dont't fill gaps）

参数	说明
ffill或pad	前向填充（或搬运）值(propagate last valid observation forward to next valid)
bfill或backfill	后向填充（或搬运）值 (use next valid observation to fill gap)
nearest	使用最近的值填充（use nearest valid observations to fill gap）

- 对于DataFrame， reindex 可以修改（行）索引、列，或两个都修改。
- 如果仅传入一个序列，则会重新索引行。
- reindex 并没有就地修改，即，没有改变源数据

In [55]:
```
frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
                  columns=['Ohio', 'Texas', 'California'])
frame
```

Out[55]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [56]:
```
# 只传入一个索引序列，则 reindex只修改行索引，即添加一个新行
frame2 = frame.reindex(['a', 'b', 'c', 'd']) # 默认添入新行的值为NaN
frame2
```

Out[56]:

	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

In [57]:
```
# 使用columns关键字，reindex 可以重新索引列，即实现修改列
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states)
```

Out[57]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

In [58]:
```
# 也可以同时对行和列进行重新索引，而插值则只能按行应用（即轴0）
# the use of sort ref to
# https://stackoverflow.com/questions/37982170/pandas-reindex-and-fill-missing-values-index-must-b
frame.reindex(index=['a','b','c','d'],method='ffill',columns=states.sort())
```

Out[58]:

	Ohio	Texas	California
a	0	1	2
b	0	1	2
c	3	4	5
d	6	7	8

In [59]:
```
# 注意：reindex并没有修改源数据
frame
```

Out[59]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

下表列出reindex函数的参数和说明

参数	说明
index	用作索引的新序列。即可以是Index实例，也可以是其他序列型的Python数据结构。Index会被完全使用，就像没有任何复制一样
method	插值（填充）方式，具体可以参见前面的表格
fill_value	在重新索引的过程中，需要引入缺失值时使用的替代值
limit	前向或后向填充时的最大填充量
level	在MultiIndex的指定级别上匹配简单索引，否则选取其子集
copy	默认为True，无论如何都复制；如果为False，则新旧相等就不复制

删除指定轴上的项（Dropping entries from an axis）

删除某条轴上的一个或多个项很简单，只要一个索引数组或列表即可。由于需要执行一些数据整理和集合逻辑，所以drop方法返回的是一个在指定轴上删除了指定值的新对象(没有改变源数据，即，没有对源数据删除)。

In [60]:
```
# 对series序列，只能删除指定行上的值
s = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_s = s.drop('c') # 丢弃指定的c行
new_s
```

Out[60]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [61]:
```
s.drop(['d', 'c'])# 删除指定的d和c行
```

Out[61]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

In [62]:
```
# 对于DataFrame，可以删除任意轴上的索引值。axis = 1即删除列轴
data = DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

Out[62]:

	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 [63]:
```
data.drop('Colorado') # 删除指定的索引值，axis默认为0，即行轴
```

Out[63]:

	one	two	three	four
Ohio	0	1	2	3
Utah	8	9	10	11
New York	12	13	14	15

In [64]:
```
data.drop(['Colorado', 'Ohio']) # 删除指定的两个索引值，axis默认为0，即行轴
```

Out[64]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

In [65]:
```
data.drop('two', axis=1) # 删除指定的列索引值，必须指定axis=1，即列轴
```

Out[65]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [66]:
```
data.drop(['two', 'four'], axis=1)
```

Out[66]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

In [67]:
```
#data.drop('two') # 报错：删除指定的列索引值，必须增加axis=1，即列轴
```

索引、选取和过滤（Indexing, selection, and filtering）

Series索引（obj[...]）的工作方式类似于NumPy数组的索引，只不过Series的索引值不只是整数。

In [68]:
```
s = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(s['b']) # 直接返回b索引对应的value
print(s[['b','c']])  # 返回b和c索引对应的行，即 key 和 itme的value
```

```
1.0
b    1.0
c    2.0
dtype: float64
```

In [69]:
```
print(s[1])  # series的索引只有行索引，可以使用整数索引代替索引值
print(s[2:4]) # 选取片段
print(s[['b', 'a', 'd']]) # 对多个行进行索引，需要使用list[]指明索引
print(s[[1, 3]])  # 选取多个行
```

```
1.0
c    2.0
d    3.0
dtype: float64
b    1.0
a    0.0
d    3.0
dtype: float64
b    1.0
d    3.0
dtype: float64
```

In [70]:
```
s[s < 2] # 按布尔值选取
```

Out[70]:
```
a    0.0
b    1.0
dtype: float64
```

- 利用标签的切片运算与普通的Python切片运算不同，其末端是包含的闭区间（inclusive）

In [71]:
```
s['b':'c'] #利用标签的切片与普通的Python不同，末端是包含闭区间,比较s[1:2]
```

Out[71]:
```
b    1.0
c    2.0
dtype: float64
```

In [72]:
```
s['b':'c'] = 5  # 给b和c索引的项赋值，改变源数据的值
s
```

Out[72]:
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

DataFrame中获取数据，无论是行切片或者列切片，都需要进行数据切片。

要从DataFrame选取列：

- 直接使用标签索引，获取一个或多个列， data['column1']或data[['column2','column3']]或data.column1

要从DataFrame选取行的方式有三种：

- 1) 使用数字索引，获取一个或多个行，data[2:4](选取第3和第4行数据),即使一行，也需要使用切片[i:i+1]获取第i行
- 2) 通过布尔型数组选取行 data[data[column] > 5]

- 3) 为了在行上进行标签索引，引入专门的索引字段 ix ，可以通过NumPy式的标记法以及轴标签从DataFrame中选取行和列的子集。这也是一种重新索引的简单手段。

索引字段ix可以按照标签索引选取行和列的子集 `data.ix['Colorado', ['two', 'three']]`

根据整数位置选取单列或单行，并返回一个Series：

- print data.iloc[:,1] 根据整数位置选取单列，并返回一个Series
- print data.iloc[3] 根据整数位置选取单行，并返回一个Series

根据标签选取单行或单列，并返回一个Series

- print data.xs('Utah')
- print data.xs('one', axis=1)

In [73]:
```
data = DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

Out[73]:

	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 [74]:
```
data['two'] # 列索引就是获取一个列，进行切片
data.two  # 同上，获取一个列，进行切片
data[['three', 'one']] # 获取多个列，对列索引使用list[]进行切块
```

Out[74]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

In [75]:
```
data.loc['Colorado',['three', 'one']]
```

Out[75]:
```
three    6
one      4
Name: Colorado, dtype: int64
```

In [76]:
```
# 要从DataFrame选取行，使用切片方式
print(data[0:1]) # 使用数字索引，默认就是行索引（轴为0）的获取行
print(data[:2]) #
print(data[1:2]) # 如果只获取一行，不能直接使用这行的数字索引，而要使用[i:i+1]获取第i行数据
```

```
      one  two  three  four
Ohio    0    1      2     3
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
          one  two  three  four
Colorado    4    5      6     7
```

In [77]:
```
print(data)
data[data['three'] > 5] # 对DataFrame也可以通过布尔型数组选取数据
```

```
          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
```

Out[77]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [78]:
```
print(data)
print(data < 5) # 由标量比较运算获得布尔型DataFrame
```

```
          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
            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 [79]:
```
data[data < 5] = 0  # 利用标量比较运算得到的布尔型DataFrame进行索引
data
```

Out[79]:

	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

In [80]:
```
#data.loc['Colorado', ['two', 'three']] # 索引字段loc可以按照标签索引选取行和列的子集
#data.loc['Colorado', ['two', 'three']] # 同上
#data.iloc[2] # 返回行索引（轴=0），第3行
#data.iloc[2][1]# 返回第3行的第2个项，采用切块的方法取单个元素项
#data.ix[['Colorado', 'Utah'], [3, 0, 1]] # 索引字段ix可以按照标签索引选取行和列的子集
#data.iloc[2][1]# 返回第3行的第2个项，采用切块的方法取单个元素项
#data.loc[:'Utah', 'two'] # 切块操作取得单个元素，包含Utah的闭区间之前的行，列=two
#data.ix[data.three > 5, :3] # 结合比较运算切块
```

In [81]:
```
print(data)
# 根据整数位置选取单列或单行，并返回一个Series
print(data.iloc[:,1])  # 根据整数位置选取单列，并返回一个Series
print(data.iloc[3]) # 根据整数位置选取单行，并返回一个Series
# 根据标签选取单行或单列，并返回一个Series
print(data.xs('Utah'))  # 根据标签选取单列，默认 axis=0
print(data.xs('one', axis=1)) # 根据标签选取单列，axis=1
```

```
          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
Ohio        0
Colorado    5
Utah        9
New York   13
Name: two, dtype: int64
one      12
two      13
three    14
four     15
Name: New York, dtype: int64
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64
Ohio        0
Colorado    0
Utah        8
New York   12
Name: one, dtype: int64
```

下面的表格列出了针对DataFrame数据的选取和重排方式

对pandas对象中的数据的选取和重排方式有很多，在使用层次化索引时还能用到一些别的方法。

对于pandas对象，必须输入obj[:,col]才能选取列实在有些啰嗦，而且很容易出错，因为列的选取是一种最常见的操作，因此，在设计pandas的时候就把所有的标签索引功能都放到 ix 中了。如下表显示的选项。

下表列出DataFrame的索引选项

类型	说明
obj.val	获取DataFrame的单个列，返回的一个Series
obj[val]	选取DataFrame的单个列或一组列。在一些特殊情况下会比较便利：布尔型数组（过滤行）、切片（行切片）、布尔型DataFrame（根据条件设置值）
obj.ix[val]	选取DataFrame的单个行或一组行
obj.ix[:, val]	选取单个列或列子集
obj.ix[val1, val2]	同时选取行和列 （切块操作）
reindex方法	将一个或多个轴匹配到新索引
xs 方法	根据标签选取单行或单列，并返回一个Series
icol、irow方法	根据整数位置选取单列或单行，并返回一个Series
get_value、set_vlaue方法	根据行标签和列标签选取/设置单个值

算术运算和数据对齐（Arithmetic and data alignment）

- pandas最重要的一个功能是，它可以对不同索引的对象进行算术运算。 在将对象相加时，如果存在不同的索引对，则结果的索引就是该索引对的并集。
- 自动的数据对齐操作中不重叠的索引处引入了NA值。缺失值会在算术运算过程中传播。
- 对于DataFrame， 对齐操作会同时发生在行和列上。

In [82]:
```
s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
s1 + s2  # 自动数据对齐，在不重叠的索引处引入NA值
```

Out[82]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [83]:
```
list('bcd') #  把字符串转为list
df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                index=['Ohio', 'Texas', 'Colorado'])
df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                index=['Utah', 'Ohio', 'Texas', 'Oregon'])
# 对于DataFrame对齐操作会同时发生在行和列上,相加后返回一个新的DataFrame，
# 其索引和列为原来那两个DataFrame的并集。
df1 + df2
```

Out[83]:

	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

在算术方法中填充值（Arithmetic methods with fill values）

- 在对不同索引的对象进行算术运算时，可能希望当一个对象中某个轴标签中另一个对象中找不到时填充一个特殊值（比如0），而不是默认的NA值。
- 使用df1的add方法，传入df2以及一个fill_value参数, fill_value=0 表示对于找不到的对象时默认值为0，
  `df1.add(df2, fill_value=0)`

In [84]:
```
df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
df1 + df2  # 将它们相加，没有重叠的位置就会产生NA值
df1.add(df2, fill_value=0)  # 使用df1的add方法，传入df2以及一个fill_value参数
```

Out[84]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	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

In [85]:
```
# 类似的，在对Series或DataFrame重新索引时，也可以指定一个填充值：
df1.reindex(columns=df2.columns, fill_value=0)
```

Out[85]:

	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

下表列出灵活的算术方法

方法	说明
add	用于加法（+）的方法
sub	用于减法（-）的方法
div	用于除法（/）的方法
mul	用于乘法（*）的方法

DataFrame和Series之间的运算（Operations between DataFrame and Series）

DataFrame和Series之间算术运算也是有明确规定的。下面的例子是计算一个二维数组与其某行之间的差，这叫做广播（broadcasting）。

In [86]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```

Out[86]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [87]:
```
arr[0]
```

Out[87]:
```
array([0., 1., 2., 3.])
```

In [88]:
```
# 二维数组与其第一行之间的差，这样的运算叫做广播（broadcasting）
arr - arr[0]
```

Out[88]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

- 在DataFrame 和 Series之间的运算也是类似的广播

In [89]:
```
frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[89]:

	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 [90]:
```
series = frame.iloc[0] # 取出第一行
series
```

Out[90]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [91]:
```
# 减法运算也进行广播
frame - series
```

Out[91]:

	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

- 默认情况下，DataFrame和Series之间的算术运算将会将Series 的索引匹配到DataFrame的列，然后沿着行一直向下广播。
- 如果某个索引值在DataFrame的列或Series的索引中找不到，则参与运算的两个对象就会被重新索引以形成并集。
- 如果希望匹配行并在列上广播，则必须使用算术运算方法。

In [92]:
```
# 如果某个索引值在DataFrame的列或Series的索引中找不到，
# 则参与运算的两个对象就会被重新索引以形成并集。
series2 = Series(list(range(3)), index=['b', 'e', 'f'])
frame + series2
```

Out[92]:

	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

In [93]:
```
frame - series2
```

Out[93]:

	b	d	e	f
Utah	0.0	NaN	1.0	NaN
Ohio	3.0	NaN	4.0	NaN
Texas	6.0	NaN	7.0	NaN
Oregon	9.0	NaN	10.0	NaN

In [94]:
```
# 如果希望匹配行并在列上广播，则必须使用算术运算*方法*，即，不能直接 A - B。
series3 = frame['d']
frame
```

Out[94]:

	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 [95]:
```
# sub 是减法的算术方法,即frame-series3，传入的轴号0就是希望匹配的轴，
# 这里希望匹配frame的行索引并进行广播
frame.sub(series3, axis=0)
```

Out[95]:

	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

函数应用和映射（Function application and mapping）

- NumPy的ufunc（元素级数组方法）也可以用于操作pandas对象。

In [96]:
```
frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[96]:

	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 [97]:
```
np.abs(frame)
```

Out[97]:

	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

- 另一个常用的操作是，将函数应用到各列或行所形成的一维数组上。DataFrame的apply方法即可实现此功能。
- 许多最为常见的数组统计功能都被实现成DataFrame的方法（如sum 和 mean），因此无需使用apply方法。

In [98]:
```
# 定义一个lambda 函数
f = lambda x: x.max() - x.min()
```

In [99]:
```
# 默认函数是在DataFrame的各个列上apply，即轴=0
print(frame)
frame.apply(f)
```

```
               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
```

Out[99]:
```
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

In [100]:
```
# 如果指定了应用的轴（axis=1），则按行应用
frame.apply(f, axis=1)
```

Out[100]:
```
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

- 除标量值外，传递给apply的函数还可以返回由多个值组成的Series。

In [101]:
```
def f(x):
    return Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
```

Out[101]:

	b	d	e
min	-0.555730	0.281746	-1.296221
max	1.246435	1.965781	1.393406

- 元素级的Python函数也是可以用的。想得到frame中各个浮点值的格式化字符串，使用applymap即可：

In [102]:
```
format = lambda x: '%.2f' % x
frame.applymap(format)
```

Out[102]:

	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

- 之所以叫applymap，是因为Series有一个用于应用元素级函数的map方法：

In [103]:
```
frame['e'].map(format)
```

Out[103]:
```
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

排序（Sorting）

1. 按索引排序： 可使用sort_index方法，它将返回一个已排序的新对象。对DataFrame，则可以根据任意一个轴上的索引进行排序（axis=0即行，axis=1即列）

2. 按值排序： 可使用使用sort_values()方法（order方法过时）。任何缺失值默认都会被放到Series的末尾。对DataFrame上进行按值排序，如果希望根据一个或多个列中的值进行排序，将一个或多个列的名字传递给by选项即可。

对DataFrame，按行只能进行索引排序，按列可以进行索引排序，也可以按照指定列进行按值排序

In [104]:
```
s = Series(list(range(4)), index=['d', 'a', 'b', 'c'])
s.sort_index() # 对Series的行索引排序（按照索引由小到大的顺序排序）
```

Out[104]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

In [105]:
```
# 对于DataFrame，则可以根据任意一个轴上的索引进行排序。默认是轴0
frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                  columns=['d', 'a', 'b', 'c'])
frame.sort_index() # 默认在轴0，即行上进行行索引排序（按照索引由小到大的顺序排序）
frame.sort_index(axis=1) # 选择在轴1，即列上进行列索引排序
frame.sort_index(axis=1, ascending=False)# 数据默认是按升序排序，也可以降序排序
```

Out[105]:

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

** 按值排序**

- 要按值对Series进行排序，使用sort_values()方法。(order方法过时)。排序时，任何缺失值默认都会被放到Series的末尾。

- 在DataFrame上进行按值排序，使用sort_values()方法。(sort_index方法过时)，如果希望根据一个或多个列中的值进行排序，将一个或多个列的名字传递给by选项即可。

In [106]:
```
s = Series([4, 7, -3, 2])
s.sort_values()   # sort_values按值排序，sort_index()是按索引排序
```

Out[106]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

In [107]:
```
frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]}, index=list('3012'))
print(frame)
frame.sort_index() # 默认在axis=0 即行轴上进行行索引排序
frame.sort_index(axis=1) # 在axis=1 即列轴上进行列索引排序
frame.sort_values(by='b') # by选项指定列，按照该列进行按值排序
frame.sort_values(by=['a', 'b']) # 要按照多个列的值排序，就把多个列的名字传递给by选项
frame.sort_values(by=['a', 'b'],ascending=False) # 按降序排列
```

```
   b  a
3  4  0
0  7  1
1 -3  0
2  2  1
```

Out[107]:

	b	a
0	7	1
2	2	1
3	4	0
1	-3	0

排名（Ranking）

- 排名（ranking）跟排序关系密切，且它会增设一个排名值（从1开始，直到数组中有效数据的数量）。跟 numpy.argsort 产生的间接排序索引差不多，但可根据某种规则破坏平级关系
- 默认情况下，rank是通过“为各组分配一个平均排名”的方式破坏平级关系的。如果有两个相同的值都排在第三位，则rank排名值会对它们的排名进行平均，即返回3.5
- 也可以根据值在原数据中出现的顺序给出排名（即，相同的值就按照它们出现的先后顺序而排序），method='first'
- 也可以按照降序进行排名，平级关系使用max即最大排名值

In [108]:
```
s = Series([7, -5, 7, 4, 2, 0, 4])
s.rank()  # 对Series对象的rank排名(从1开始)，排名值对于两个平级的排名会进行平均
s.rank(method='first') # 也可以平级关系按照出现的先后顺序排序，method='first'
s.rank(ascending=False, method='max') # 也可以按照降序进行排名，平级关系使用max即最大排名值
```

Out[108]:
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

In [109]:
```
frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                   'c': [-2, 5, 8, -2.5]})
#print frame
frame.rank() # 默认在每列上按照Series进行rank排名
frame.rank(axis=1) # 在轴1上排序，即每行上按照Series进行rank排名
```

Out[109]:

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

下表列出排名时用于破坏平级关系的method选项

method	说明
'average'	默认：在相等分组中，为各个值分配平均排名
'min'	使用整个分组的最小排名
'max'	使用整个分组的最大排名
'first'	按值在原始数据中的出现顺序分配排名

带有重复值的轴索引（Axis indexes with duplicate values）

到目前为止，前面所有数据都有唯一的轴标签（索引值）。虽然很多pandas函数（如reindex）都要求标签唯一，但这并不是强制性的。

In [110]:
```
s = Series(list(range(5)), index=['a', 'a', 'b', 'b', 'c']) # 带有重复索引值的Series
```

- index.is_unique属性可以判断索引的值是否是唯一的
- 对于带有重复值的索引，数据选取的行为将会有些不同。如果某个索引对应多个值，则返回一个Series；而对应单个值，则返回一个标量值。

In [111]:
```
s.index.is_unique  # index的is_unique 属性可以说明索引的值是否唯一
```

Out[111]:
```
False
```

In [112]:
```
print(s['a'], type(s['a'])) # 如果某个索引对s应多个值，则返回一个Series
print(s['c'], type(s['c'])) # 如果索引对应单个值，则返回一个标量值
```

```
a    0
a    1
dtype: int64 <class 'pandas.core.series.Series'>
4 <class 'numpy.int64'>
```

- 对DataFrame的行进行索引时也是类似的。索引对应多个值，则返回DataFrame；如果对应单个值，则返回一个Series。

In [113]:
```
df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'c'])
print(df.loc['a'], type(df.loc['a']))  # 索引对应多个值，则返回一个DataFrame
print(df.loc['c'], type(df.loc['c'])) # 索引对应单个值，则返回一个Series
```

```
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843 <class 'pandas.core.frame.DataFrame'>
0    0.476985
1    3.248944
2   -1.021228
Name: c, dtype: float64 <class 'pandas.core.series.Series'>
```

数据的汇总和描述统计

pandas对象拥有一组常用的数学和统计方法。它们大部分都属于约简和汇总统计，用于从Series中提取单个值（如 sum 和 mean ）或从DataFrame的行或列中提取一个Series。跟对应的NumPy数组相比，它们都是基于没有缺失数据的假设而构建的。

- NA值会自动被排除，除非整个切片（行或列）都是NA。通过skipna选项可以禁用该功能。

In [114]:
```
df = 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'])
df
```

Out[114]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [115]:
```
# 默认，一个列是一个series。调用DataFrame的sum方法将会返回一个含有列小计的Series：
df.sum()
```

Out[115]:
```
one    9.25
two   -5.80
dtype: float64
```

In [116]:
```
# 传入axis=1将会按行进行求和运算
df.sum(axis=1)
```

Out[116]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

- NA值会自动被排除，除非整个切片（行或列）都是NA。通过skipna选项可以禁用该功能。

In [117]:
```
# NA值会自动被排除，除非整个切片（行或列）都是NA。通过skipna选项可以禁用该功能。
df.mean(axis=1, skipna=False)
```

Out[117]:
```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

下表列出了这些约简方法的常用选项。

选项	说明
axis	约简的轴。DataFrame的行用0， 列用1
skipna	排除缺失值，默认值为True
level	如果轴是层次化索引的（即MultiIndex），则根据level分组约简

- 有些方法（如idxmin 和 idxmax）返回的是间接统计（比如达到最小值或最大值的索引）

In [118]:
```
# 返回的是间接统计，即，达到最大值的索引
df.idxmax()
```

Out[118]:
```
one    b
two    d
dtype: object
```

- 另一些方法是累计型的：

In [119]:
```
# 累计型方法
df.cumsum()
```

Out[119]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

- 还有一种方法，既不是约简型的，也不是累计型的。如 describe，用于一次性产生多个汇总统计的描述，返回值也是DataFrame型。
- 对于非数值型数据，describe会产生另外一种汇总统计。

In [120]:
```
# describe方法用于一次性产生多个汇总统计，返回的也是DataFrame型描述
print(type(df.describe()))
df.describe()
```

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

Out[120]:

	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

In [121]:
```
# 对于非数值型数据，describe会产生另外一种汇总统计
obj = Series(['a', 'a', 'b', 'c'] * 4)
# print obj
obj.describe()
```

Out[121]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

下表列出了所有与描述统计有关的方法

表列出描述和汇总统计

方法	说明
count	非NA值的数量
describe	针对Series或各DataFrame列计算汇总统计
min、max	计算最小值和最大值
argmin、argmax	计算能够获取到最小值和最大值的索引位置（整数）
idxmin、idxmax	计算能够获取到最小是和最大值的索引值
quantile	计算样本的分位数（0到1）
sum	值的总和
mean	值的平均数
median	值的算术中位数（50%分位数）
mad	根据平均值计算平均绝对离差
var	样本值的方差
std	样本值的标准差
skew	样本值的偏度（三阶矩）
kurt	样本值的峰度（四阶矩）

方法	说明
cumsucm	样本值的累计和
cummin、cummax	样本值的累计最大值和累计最小值
cumprod	样本值的累计积
diff	计算一阶差分（对时间序列很有用）
pct_change	计算百分数变化

相关系数与协方差（Correlation and covariance）

有些汇总统计（如相关系数和协方差）是通过参数对计算出来的。

- 下面的几个DataFrame的数据是来自Yahoo！Finance的股票价格和成交量。

采集股票数据

使用 Yahoo！Finance 提供的API

安装库

```
- conda install pandas-datareader
```

加载库

```
import pandas_datareader as pdr
```

或者

```
from pandas_datareader import data, wb
```

如果遇到版本不兼容无法安装问题，建议更新 GitHub 最新版本

```
pip install git+https://github.com/pydata/pandas-datareader
```

In [124]:
```
%system conda list pandas-datareader
```

Out[124]:
```
['# packages in environment at /home/lanman/anaconda3:',
 '#',
 '# Name                    Version                   Build  Channel',
 'pandas-datareader         0.9.0                      py_0  ']
```

In [125]:
```
# 方式： 使用Yahoo Finance的API获取四个公司的股票数据
import pandas as pd
import numpy as np
from pandas import Series, DataFrame
from pandas_datareader import data

all_stock = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_stock[ticker] = data.get_data_yahoo(ticker,start='12/1/2018')# 默认从2010年1月起始, start=
# print(all_stock.items())

price = DataFrame({tic: data['Close']
                   for tic, data in all_stock.items()})
volume = DataFrame({tic: data['Volume']
                   for tic, data in all_stock.items()})
open = DataFrame({tic: data['Open']
                   for tic, data in all_stock.items()})
high = DataFrame({tic: data['High']
                   for tic, data in all_stock.items()})
low =  DataFrame({tic: data['Low']
                   for tic, data in all_stock.items()})
```

In [126]:
```
# 最前面五条交易量数据
volume.head()
```

Out[126]:

Date	AAPL	IBM	MSFT	GOOG
2018-11-30	158126000.0	7251600.0	33665600.0	2580200
2018-12-03	163210000.0	5846400.0	34732800.0	1991200
2018-12-04	165377200.0	6000700.0	45197000.0	2345200
2018-12-06	172393600.0	6938000.0	49107400.0	2769200
2018-12-07	169126400.0	7019600.0	45044900.0	2101200

In [127]:
```
price.head()
```

Out[127]:

Date	AAPL	IBM	MSFT	GOOG
2018-11-30	44.645000	124.269997	110.889999	1094.430054
2018-12-03	46.205002	125.309998	112.089996	1106.430054
2018-12-04	44.172501	121.599998	108.519997	1050.819946
2018-12-06	43.680000	123.910004	109.190002	1068.729980
2018-12-07	42.122501	119.339996	104.820000	1036.579956

In [128]:
```
# 最后面五条价格数据
price.tail()
```

Out[128]:

Date	AAPL	IBM	MSFT	GOOG
2021-03-05	121.419998	122.830002	231.600006	2108.540039
2021-03-08	116.360001	124.809998	227.389999	2024.170044
2021-03-09	121.089996	124.180000	233.779999	2052.699951
2021-03-10	119.980003	127.870003	232.419998	2055.030029
2021-03-11	121.959999	127.139999	237.130005	2114.770020

- 接下来计算价格的百分数变化

In [129]:
```
# 计算价格的百分数变化
returns = price.pct_change()
# 价格百分数变化的最后五条记录
returns.tail()
```

Out[129]:

Date	AAPL	IBM	MSFT	GOOG
2021-03-05	0.010738	0.022646	0.021479	0.029013
2021-03-08	-0.041674	0.016120	-0.018178	-0.040013
2021-03-09	0.040650	-0.005048	0.028101	0.014095
2021-03-10	-0.009167	0.029715	-0.005817	0.001135
2021-03-11	0.016503	-0.005709	0.020265	0.029070

In [130]:
```
# 获得DataFrame的一个列MSFT, 得到一个Series
print(type(returns.MSFT))
# 获得MSFT最后五条数据,返回一个Series
returns.MSFT.tail()
```

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

Out[130]:
```
Date
2021-03-05    0.021479
2021-03-08   -0.018178
2021-03-09    0.028101
2021-03-10   -0.005817
2021-03-11    0.020265
Name: MSFT, dtype: float64
```

- Series的 corr 方法用于计算两个Series中重叠的、非NA的、按索引对齐的值的相关系数。
- 与此类似，cov 用于计算协方差。

In [131]:
```
# 计算 微软 与 IBM 之间的股票价格百分数变化的相关系数
returns.MSFT.corr(returns.IBM)
```

Out[131]:
```
0.5872341626468094
```

In [132]:
```
returns.AAPL.corr(returns.MSFT)
```

Out[132]:
```
0.7900276535241794
```

In [133]:
```
# 计算 微软 与 IBM 之间的股票价格百分数变化的协方差
returns.MSFT.cov(returns.IBM)
```

Out[133]: 0.00025478705484060105

- 对于DataFrame的corr和cov方法将以DataFrame的形式返回完整的相关系数或协方差矩阵：

In [134]:
```
# 计算DataFrame的corr，对角线为1.0
price.corr()
```

Out[134]:

	AAPL	IBM	MSFT	GOOG
AAPL	1.000000	-0.423082	0.970510	0.932889
IBM	-0.423082	1.000000	-0.409839	-0.290526
MSFT	0.970510	-0.409839	1.000000	0.916762
GOOG	0.932889	-0.290526	0.916762	1.000000

In [135]:
```
# 计算DataFrame的cov，协方差矩阵
returns.cov()
```

Out[135]:

	AAPL	IBM	MSFT	GOOG
AAPL	0.000570	0.000250	0.000402	0.000335
IBM	0.000250	0.000415	0.000255	0.000228
MSFT	0.000402	0.000255	0.000454	0.000344
GOOG	0.000335	0.000228	0.000344	0.000416

- 利用DataFrame的corrwith方法，可以计算其列或行跟另一个Series或DataFrame之间的相关系数。
- 传入一个Series将会返回一个相关系数值Series（针对各列进行计算）
- 传入一个DataFrame则会计算按列名配对的相关系数；如果传入axis=1，则计算按行的相关系数。

In [136]:
```
# 计算DataFrame与Series之间的相关系数
returns.corrwith(returns.IBM)
```

Out[136]:
```
AAPL    0.515208
IBM     1.000000
MSFT    0.587234
GOOG    0.547609
dtype: float64
```

In [137]:
```
volume.tail()
```

Out[137]:

Date	AAPL	IBM	MSFT	GOOG
2021-03-05	153590400.0	6944900.0	41842100.0	2193800
2021-03-08	153918600.0	7236600.0	35245900.0	1646000
2021-03-09	129159600.0	5608200.0	33034000.0	1696400
2021-03-10	111760400.0	7243500.0	29733000.0	1267800
2021-03-11	99598709.0	5147977.0	29907586.0	1208604

In [138]:
```
# 计算价格的百分数变化(DataFrame)与交易量(DataFrame)之间的相关系数，产生按列名匹配的相关系数
returns.corrwith(volume)
```

Out[138]:
```
AAPL   -0.086781
IBM    -0.111067
MSFT   -0.075950
GOOG   -0.077023
dtype: float64
```

In [139]:
```
# 传入axis=1，则计算按行的相关系数
returns.corrwith(volume, axis=1).tail()
```

Out[139]:
```
Date
2021-03-05   -0.947703
2021-03-08   -0.497354
2021-03-09    0.822087
2021-03-10   -0.582067
2021-03-11    0.105391
dtype: float64
```

唯一值、值计数以及成员资格（Unique values, value counts, and membership）

还有一类方法可以从一维Series的值中抽取信息，例如：

- 第一个函数是unique，可以得到Series中的唯一值数组numpy.ndarray类型，返回的唯一值是未排序的数组numpy.ndarray类型，如果需要，还可以对结果再次进行排序（unique.sort()）
- value_counts 用于计算一个Series中各值出现的频率。结果Series是按值频率降序排列的。obj.value_counts()

**value_counts* 还是一个顶级*pandas*方法，可用于任何数组或序列。 *pd.value_counts(obj.values, sort=False)*
*

- 最后一个是isin， 它用于判断矢量化集合的成员资格，可用于选取Series中或DataFrame列中数据的子集。

In [140]:
```
obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

In [141]:
```
# unique函数得到Series中的唯一值数组，返回的唯一值是未排序的数组numpy.ndarray类型
uniques = obj.unique()
uniques
```

Out[141]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

In [142]:
```
# value_counts函数用于计算一个Series中各值出现的频率，返回的结果Series是按值频率降序排列的
obj.value_counts()
```

Out[142]:
```
c    3
a    3
b    2
d    1
dtype: int64
```

In [143]:
```
# value_counts函数还是pandas的顶级方法，可以用于任何数组或序列
pd.value_counts(obj.values, sort=False)
```

Out[143]:
```
d    1
c    3
a    3
b    2
dtype: int64
```

In [144]:
```
# isin函数用于判断矢量化集合的成员资格，返回一个布尔型数组，用于选取Series或DataFrame列中数据的子集
mask = obj.isin(['b', 'c'])
mask
```

Out[144]:
```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

In [145]:
```
# 选取Series中数据的子集
obj[mask]
```

Out[145]:
```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

In [146]:
```
data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
                  'Qu2': [2, 3, 1, 2, 3],
                  'Qu3': [1, 5, 2, 4, 4]})
data
```

Out[146]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

In [147]:
```
# value_counts只能对Series计算，因此对于DataFrame需要指定在哪个列上进行值计数
data.Qu1.value_counts()
```

Out[147]:
```
3    2
4    2
1    1
Name: Qu1, dtype: int64
```

In [148]:
```
# value_counts也是pandas的顶级方法
pd.value_counts(data.Qu2, sort=True)
```

Out[148]:
```
2    2
3    2
1    1
Name: Qu2, dtype: int64
```

In [149]:
```
print(data)
result = data.apply(pd.value_counts).fillna(0)
result
```

```
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Out[149]:

	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

下表列出这些方法的一些参考信息

唯一值、值计数、成员资格方法

方法	说明
isin	计算一个表示“Series各值是否包含于传入的值序列中”的布尔型数组
unique	计算Series中的唯一值数组，按发现的顺序返回
value_counts	返回一个Series，其索引为唯一值，其值为频率，按计数值降序排列

处理缺失值（Handling missing data）

缺失数据（missing data）在大部分数据分析应用中都很常见。pandas的设计目标之一就是让缺失数据的处理任务尽量轻松。例如，pandas对象上的所有描述统计都排除了缺失数据。

- pandas使用浮点值 NaN（Not a Number）表示浮点和非浮点数组中的缺失数据, np.nan, 是一个便于被检测出来的标记而已。
- Python内置的None值也会被当做NA处理, value = None
- pandas 的NA表现形式很简单也很可靠。由于NumPy的数据类型体系中缺乏真正的NA数据类型或位模式，所以目前最佳的解决方案可能就是pandas的NA（一套简单API以及足够全面的性能特征）。以后随着NumPy的发展，也许问题会变化。

In [150]:
```
string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
string_data
```

Out[150]:
```
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

In [151]:
```
string_data.isnull()
```

Out[151]:
```
0    False
1    False
2     True
3    False
dtype: bool
```

In [152]:
```
string_data[0] = None
print(string_data[0])
string_data.isnull()
```

```
None
```

Out[152]:
```
0     True
1    False
2     True
3    False
dtype: bool
```

表列出了NA处理方法

方法	说明
dropna	根据各标签的值中是否存在缺失数据对轴标签进行过滤，可通过阈值调节对缺失值的容忍度
fillna	用指定值或插值方法（如ffill或bfill）填充缺失数据
isnull	返回一个含有布尔值的对象，这些布尔值表示哪些值是缺失值/NA， 该对象的类型与源类型一样
notnull	isnull的否定式

滤除缺失数据（Filtering out missing data）

过滤掉缺失数据的方法有很多种。

- 对于一个Series，dropna方法返回一个仅含非空数据和索引值的Series；也可以通过布尔型索引达到这个目的
- 但是对于一个DataFrame对象，事情有点复杂。dropna默认丢弃任何含有缺失值的行。

In [153]:
```
from numpy import nan as NA
data = Series([1, NA, 3.5, NA, 7])
# 对于一个Series，dropna方法返回一个仅含非空数据和索引值的Series
data.dropna()
```

Out[153]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [154]:
```
# 通过布尔型索引过滤掉空值
#data.notnull()
data[data.notnull()]
```

Out[154]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

- 对于一个DataFrame对象，事情有点复杂。dropna默认丢弃任何含有缺失值的行；
- 传入how='all'将只丢弃全为NA的那些行；
- 如果要丢弃列，只需传入axis=1即可；
- 另一个滤除DataFrame行的问题涉及时间序列数据。如果只想留下部分观测数据，可以用thresh参数实现此目的。

In [155]:
```
# 对于一个DataFrame对象，事情有点复杂,dropna默认丢弃任何含有缺失值的行
data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
                  [NA, NA, NA], [NA, 6.5, 3.]])
cleaned = data.dropna()
data
```

Out[155]:

	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 [156]:
```
# 丢弃任何含有缺失值的行
cleaned
```

Out[156]:

	0	1	2
0	1.0	6.5	3.0

In [157]:
```
# 传入how='all'将只丢弃全为 NA 的那些行；how='any'是默认值
data.dropna(how='all')
```

Out[157]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

In [158]:
```
# 为data增加第四列
data[4] = NA
data
```

Out[158]:

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

In [159]:
```
#  如果要丢弃列，只需传入axis=1即可
data.dropna(axis=1, how='all')
```

Out[159]:

	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 [160]:
```
df = DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA; df.iloc[:2, 2] = NA
df
```

Out[160]:

	0	1	2
0	-0.577087	NaN	NaN
1	0.523772	NaN	NaN
2	-0.713544	NaN	-2.370232
3	-1.860761	NaN	0.560145
4	-1.265934	0.119827	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

In [161]:
```
# 另一个滤除DataFrame行的问题涉及时间序列数据。如果只想留下部分观测数据，可以用thresh参数实现此目的
# thresh=2 表示保留具有2个及以上非NA的行
df.dropna(thresh=2)
```

Out[161]:

	0	1	2
2	-0.713544	NaN	-2.370232
3	-1.860761	NaN	0.560145
4	-1.265934	0.119827	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

填充缺失数据（Filling in missing data）

我们可能不想滤除缺失数据（有可能会丢弃跟它有关的其他数据），而是希望通过其他方式填补那些“空洞”。

- 对于大多数情况而言， fillna 方法是最主要的函数。通过一个常数调用fillna就会将缺失值替换为那个常数值，df.fillna(C)
- 若是通过一个字典调用fillna，就可以实现对不同的列填充不同的值
- fillna默认会返回新对象，但也可以对现有对象进行就地修改
- 对reindex有效的那些插值方法也可用于fillna
- 填充缺失值，也可以使用更聪明一些的办法，比如，可以传入Series的平均值或中位数，data.fillna(data.median())

In [162]:
```
print(df)
# 将缺失值替换为0
df.fillna(0)
```

```
          0         1         2
0 -0.577087       NaN       NaN
1  0.523772       NaN       NaN
2 -0.713544       NaN -2.370232
3 -1.860761       NaN  0.560145
4 -1.265934  0.119827 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Out[162]:

	0	1	2
0	-0.577087	0.000000	0.000000
1	0.523772	0.000000	0.000000
2	-0.713544	0.000000	-2.370232
3	-1.860761	0.000000	0.560145
4	-1.265934	0.119827	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

In [163]:
```
# 若是通过一个字典调用fillna，就可以实现对不同的列填充不同的值
df.fillna({1: 0.5, 2: -1})
```

Out[163]:

	0	1	2
0	-0.577087	0.500000	-1.000000
1	0.523772	0.500000	-1.000000
2	-0.713544	0.500000	-2.370232
3	-1.860761	0.500000	0.560145
4	-1.265934	0.119827	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

In [164]:
```
# fillna默认会返回新对象，但也可以对现有对象进行就地修改
# always returns a reference to the filled object
# 总是返回被填充对象的引用
print(df)
_ = df.fillna(0, inplace=True)
df
```

```
          0         1         2
0 -0.577087       NaN       NaN
1  0.523772       NaN       NaN
2 -0.713544       NaN -2.370232
3 -1.860761       NaN  0.560145
4 -1.265934  0.119827 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Out[164]:

	0	1	2
0	-0.577087	0.000000	0.000000
1	0.523772	0.000000	0.000000
2	-0.713544	0.000000	-2.370232
3	-1.860761	0.000000	0.560145
4	-1.265934	0.119827	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

In [165]:
```
df = DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA; df.iloc[4:, 2] = NA
df
```

Out[165]:

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	NaN	1.004812
3	1.327195	NaN	-1.549106
4	0.022185	NaN	NaN
5	0.862580	NaN	NaN

In [166]:
```
# 对reindex有效的那些插值方法也可用于fillna
df.fillna(method='ffill')
```

Out[166]:

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	1.349742	-1.549106
5	0.862580	1.349742	-1.549106

In [167]:
```
# limit限定了前向或后向填充时可以连续填充的最大数量
df.fillna(method='ffill', limit=2)
```

Out[167]:

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	NaN	-1.549106
5	0.862580	NaN	-1.549106

- 填充缺失值，也可以使用更聪明一些的办法，比如，可以传入Series的平均值或中位数

In [168]:
```
data = Series([1., NA, 3.5, NA, 7])
# 使用平均值去填充缺失值
data.fillna(data.mean())
```

Out[168]:
```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

In [169]:
```
# 使用中位数去填充缺失值
data.fillna(data.median())
```

Out[169]:
```
0    1.0
1    3.5
2    3.5
3    3.5
4    7.0
dtype: float64
```

下表列出fillna函数的参数参考

表：fillna函数的参数

参数	说明
value	用于填充缺失值的标量值或字典对象
method	插值方式。如果函数调用时未指定其他参数的话，默认为“ffill”
axis	待填充的轴， 默认axis=0
inplace	修改调用者对象而不产生副本
limit	（对于前向和后向填充）可以连续填充的最大数量

层次化索引（Hierarchical indexing）

层次化索引（hierarchical indexing）是pandas的一项重要功能，它使你能做一个轴上拥有多个（两个以上）索引级别。抽象点说，它使你能以低维度形式处理高维度数据。

下面一个简单的例子，创建一个Series，并用一个由列表或数组组成的列表作为索引。

In [170]:
```
np.random.randn(3)
```

Out[170]:
```
array([ 0.6702,  0.853 , -0.9559])
```

In [171]:
```
data = Series(np.random.randn(10),
              index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
                     [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])
data
```

Out[171]:
```
a  1   -0.023493
   2   -2.304234
   3   -0.652469
b  1   -1.218302
   2   -1.332610
   3    1.074623
c  1    0.723642
   2    0.690002
d  2    1.001543
   3   -0.503087
dtype: float64
```

- 这就是带有MultiIndex索引的Series的格式化输出形式。索引之间的“间隔”表示“直接使用上面的标签”

In [172]:
```
data.index
```

Out[172]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 2),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

- 对于一个层次化索引的对象，选取数据子集的操作很简单：

In [173]:
```
# 选择一行
data['b']
```

Out[173]:
```
1   -1.218302
2   -1.332610
3    1.074623
dtype: float64
```

In [174]:
```
# 选择几行的第一种方式
data['b':'c']
```

Out[174]:
```
b  1   -1.218302
   2   -1.332610
   3    1.074623
c  1    0.723642
   2    0.690002
dtype: float64
```

In [175]:
```
# 选择几行的第二种方式
data.loc[['b', 'd']]
```

Out[175]:
```
b  1   -1.218302
   2   -1.332610
   3    1.074623
d  2    1.001543
   3   -0.503087
dtype: float64
```

In [176]:
```
# 按照“内层”另外一种索引选择数据
data[:, 2]
```

Out[176]:
```
a   -2.304234
b   -1.332610
c    0.690002
d    1.001543
dtype: float64
```

- 层次化索引在数据重塑和基于分组的操作（如透视表生成）中扮演着重要的角色。比如下面这段数据可以通过其unstack方法被重新安排到一个DataFrame中。
- unstack，即旋转（pivot）,可以将带有MultiIndex索引的Series生成DataFrame，原有的外索引作为行索引保留，内层索引旋转为列索引（即轴发生旋转）。旋转后的层自动被排序。
- unstack的逆运算是stack
- 后面章节会继续详细讲解stack和unstack

In [177]:
```
#data.unstack?
# Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.
# The level involved will automatically get sorted.
```

In [178]:
```
data.unstack()
```

Out[178]:

	1	2	3
a	-0.023493	-2.304234	-0.652469
b	-1.218302	-1.332610	1.074623
c	0.723642	0.690002	NaN
d	NaN	1.001543	-0.503087

In [179]:
```
# unstack的逆运算
data.unstack().stack()
```

Out[179]:
```
a  1   -0.023493
   2   -2.304234
   3   -0.652469
b  1   -1.218302
   2   -1.332610
   3    1.074623
c  1    0.723642
   2    0.690002
d  2    1.001543
   3   -0.503087
dtype: float64
```

- 对于一个DataFrame，每条轴都可以有分层索引
- 各层都可以有名字（可以是字符串，也可以是别的Python对象）。如果指定了名称，它们就会显示在控制台输出中（**不要将索引名称和轴标签混为一谈！**）

In [180]:
```
frame = DataFrame(np.arange(12).reshape((4, 3)),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=[['Ohio', 'Ohio', 'Colorado'],
                           ['Green', 'Red', 'Green']])
frame
```

Out[180]:

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

- 以上的是轴标签，每个轴没有名字(name) **不要将索引名称和轴标签混为一谈!**

In [181]:
```
frame.index
```

Out[181]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

In [182]:
```
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']
frame
```

Out[182]:

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

- 有了分级的列索引，可以轻松选取列分组

In [183]:
```
frame['Ohio']
```

Out[183]:

color		Green	Red
key1	key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

- 可以单独创建MultiIndex然后复用。因此，前面的DataFrame中的分级列可以如下方式创建:

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

重排分级顺序（Reordering and sorting levels）

有时需要重新调整某条轴上各级别的顺序，或根据指定级别上的值对数据进行排序。

- swaplevel接受两个级别编号或名称，并返回一个互换了级别的新对象（但数据不会发生变化）
- sortlevel根据单个级别中的值对数据进行排序（稳定的）。交换级别时，常常也会用到sortlevel，这样最终结果就是有序的了。

In [184]:
```
print(frame)
frame.swaplevel('key1', 'key2')
# 在轴1上进行互换级别
# frame.swaplevel(0,1,axis=1)
```

```
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
```

Out[184]:

	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

In [185]:
```
# sortlevel根据单个级别中的值对数据进行排序（稳定的）。
# 交换级别时，常常也会用到sortlevel，这样最终结果就是有序的了。
# 默认axis=0； level=1是指第二个索引层即key2作为primary key
frame.sort_index(level=1)
# 在轴1上进行level=0的排序
#frame.sort_index(axis=1, level=0)
```

Out[185]:

	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 [186]:
```
# 交换第0层和第1层的索引，然后在第0层排序
frame.swaplevel(0, 1).sort_index(level=0)
```

Out[186]:

	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

In [187]:
```
# 在轴1上交换第0层和第1层的索引，然后在轴1上的第0层排序
frame.swaplevel(0, 1, axis=1).sort_index(level=0,axis=0)
```

Out[187]:

	color	Green	Red	Green
	state	Ohio	Ohio	Colorado
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

根据级别汇总统计（Summary statistics by level）

许多对DataFrame和Series的描述和汇总统计都有一个level选项，它用于指定在某条轴上求和的级别。可以根据行或列上的级别来进行求和。

In [188]:
```
# 在轴0的levle=1上求和
frame.sum(level='key2')
```

Out[188]:

state	Ohio		Colorado
color	Green	Red	Green
key2			
1	6	8	10
2	12	14	16

- 前面的操作只能一次指定在某个轴的一个级别上进行汇总
- 如果想在多个轴的多个级别上进行汇总，可以采用分步汇总，即先在某个指定轴的级别上汇总，得到一个新的DataFrame，然后针对新的DataFrame再次指定进行汇总的轴和级别。

In [189]:
```
# 在轴1的level=color上求和
f2=frame.sum(level='color', axis=1)
f2
```

Out[189]:

	color	Green	Red
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

In [190]:
```
f3=f2.sum(level='key2',axis=0)
f3
```

Out[190]:

color	Green	Red
key2		
1	16	8
2	28	14

使用DataFrame的列（Using a DataFrame's columns）

想要将DataFrame的一个或多个列当做行索引来用，或者可能希望将行索引变成DataFrame的列。

- DataFrame的set_index函数会将其一个或多个列转换为行索引，并创建一个新的DataFrame。
- 默认情况下，那些列会从DataFrame中移除，但也可以将其保留下来,drop=False。

In [191]:
```
frame = DataFrame({'a': list(range(7)), 'b': list(range(7, 0, -1)),
                   'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
                   'd': [0, 1, 2, 0, 1, 2, 3]})
frame
```

Out[191]:

	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

In [192]:
```
# DataFrame的set_index函数会将其一个或多个列转换为行索引，并创建一个新的DataFrame。
frame2 = frame.set_index(['c', 'd'])
frame2
```

Out[192]:

		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

In [193]:
```
# 默认情况下，那些列会从DataFrame中移除，但也可以将其保留下来drop=False
frame.set_index(['c', 'd'], drop=False)
```

Out[193]:

		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

- reset_index的功能和set_index刚好相反，层次化索引的级别会被转移到列里面

In [194]:
```
frame2.reset_index()
```

Out[194]:

	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

其他有关pandas的话题（Other pandas topics）

整数索引（Integer indexing）

操作由整数索引的pandas对象常常会让新手抓狂，因为它们跟内置的Python数据结构（例如列表和元组）在索引语义上有些不同。

- 例如，下面这段代码虽然不会有错，在这种情况下，虽然pandas会求助于整数索引，因为这里我们有一个含有0、1、2的索引，但是pandas很难推断出用户想要什么（基于标签或位置的索引）

In [195]:
```
#
ser = Series(np.arange(3.))
ser.iloc[-1]
```

Out[195]:
```
2.0
```

In [196]:
```
ser
```

Out[196]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

- 相反，对于一个非整数索引，就没有这样的歧义：

In [197]:
```
ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])
ser2[-1]
```

Out[197]:
```
2.0
```

- 为了保持良好的一致性，如果你的轴索引含有索引器，那么根据整数进行数据选取的操作将总是面向标签的。这也包括用loc或iloc切片：

In [198]:
```
ser.iloc[:1]
```

Out[198]:
```
0    0.0
dtype: float64
```

In [199]:
```
ser.iloc[0:2]
```

Out[199]:
```
0    0.0
1    1.0
dtype: float64
```

- 如果需要可靠的、不必考虑索引类型的、基于位置的索引，可以采用如下的方法。
- 对于Series，可以使用（1）iget_value()方法和(2)iloc[]方法，两者等同
- 对于DataFrame的irow和icol方法：

In [200]:
```
ser3 = Series(list(range(3)), index=[-5, 1, 3])
ser3
#下面两个方法等同
```

Out[200]:
```
-5    0
 1    1
 3    2
dtype: int64
```

In [201]:
```
ser3.iloc[2]
```

Out[201]:
```
2
```

In [202]:
```
ser3.iat[2] # iget_value(i) is deprecated, Use .iat[i, j] instead
```

Out[202]:
```
2
```

In [203]:
```
frame = DataFrame(np.arange(6).reshape((3, 2)), index=[2, 0, 1])
frame
```

Out[203]:

	0	1
2	0	1
0	2	3
1	4	5

In [204]:
```
# 默认返回第一行的值
frame.iloc[0]
```

Out[204]:
```
0    0
1    1
Name: 2, dtype: int64
```

In [205]:
```
# 返回第i+1行的值
#frame.irow(0) # irow(i) is deprecated
frame.iloc[0]
```

Out[205]:
```
0    0
1    1
Name: 2, dtype: int64
```

In [206]:
```
# 返回第i+1列的值
#frame.icol(1) # icol(i) is deprecated
frame.iloc[:,1]
```

Out[206]:
```
2    1
0    3
1    5
Name: 1, dtype: int64
```

结论

- Pandas 提供了对数据分析和探索的重要类型 Series和DataFrame
- 初步下载获取股票数据