pandas 入门

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

pandas 的数据结构 —— *Series*

pd.Series(data, index)

In [2]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

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

In [3]:
```
obj2 = pd.Series([4, 7, -5, 3], index=['q', 'w', 'e', 'r'])
obj2
```

Out[3]:
```
q    4
w    7
e   -5
r    3
dtype: int64
```

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

In [4]:
```
print(obj.values)
print(obj.index)
print(obj2.values)
print(obj2.index)
```

```
[ 4  7 -5  3]
RangeIndex(start=0, stop=4, step=1)
[ 4  7 -5  3]
Index(['q', 'w', 'e', 'r'], dtype='object')
```

Series 的 *index* 可以通过赋值的方式就地修改

In [5]:
```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
obj
```

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

pd.Series(dict) : 通过字典创建 Series

In [6]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
obj3
```

Out[6]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [7]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4 # 没有 'Utah', 但是有 'California'
```

Out[7]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

pd.isnull(obj), pd.notnull(obj)：检测缺失数据

In [8]:
```
pd.isnull(obj4), pd.notnull(obj4)
```

Out[8]:
```
(California     True
 Ohio          False
 Oregon        False
 Texas         False
 dtype: bool,
 California    False
 Ohio           True
 Oregon         True
 Texas          True
 dtype: bool)
```

Series 的索引与运算

In [9]:
```
obj2['w'], obj2[['w']], obj2[['r', 'q', 'e']] #['r', 'q', 'e']: 索引表
```

Out[9]:
```
(7,
 w    7
 dtype: int64,
 r    3
 q    4
 e   -5
 dtype: int64)
```

In [10]:
```
obj2[obj2 > 0]
```

Out[10]:
```
q    4
w    7
r    3
dtype: int64
```

In [11]:
```
obj2 * 2, np.exp(obj2)
```

Out[11]:
```
(q     8
 w    14
 e   -10
 r     6
 dtype: int64,
 q       54.598150
 w     1096.633158
 e        0.006738
 r       20.085537
 dtype: float64)
```

In [12]:
```
'q' in obj2, 'a' in obj2
```

Out[12]:
```
(True, False)
```

运算时，索引标签会自动对齐

In [13]:
```
obj3, obj4, obj3+obj4
```

Out[13]:
```
(Ohio      35000
 Texas     71000
 Oregon    16000
 Utah       5000
 dtype: int64,
 California        NaN
 Ohio          35000.0
 Oregon        16000.0
 Texas         71000.0
 dtype: float64,
 California         NaN
 Ohio           70000.0
 Oregon         32000.0
 Texas         142000.0
 Utah              NaN
 dtype: float64)
```

Series 本身及索引都有 *name* 属性

In [14]:
```
obj4.name = 'population'
obj4.index.name = 'state'
obj4
```

Out[14]:
```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

pandas 的数据结构 —— *DataFrame*

pd.DataFrame(object) : DataFrame 的建立

In [15]:
```
# 直接传入一个由等长列表或 NumPy 数组组成的字典
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)
frame
```

Out[15]:

	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.head(n) : 对于特别大的 DataFrame,方法会选取前 n 行

In [16]:
```
frame.head(3)
```

Out[16]:

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002

DataFrame.columns : 返回 DataFrame 的列名

In [17]:
```
frame.columns
```

Out[17]:
```
Index(['pop', 'state', 'year'], dtype='object')
```

pd.DataFrame(object , columns)：指定列序列, DataFrame会按照指定顺序进行排列

In [18]:
```
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]
        }
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[18]:

	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

pd.DataFrame(object , columns , index)：指定索引

In [19]:
```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                      index=data['year'])
frame2
```

Out[19]:

	year	state	pop	debt
2000	2000	Ohio	1.5	NaN
2001	2001	Ohio	1.7	NaN
2002	2002	Ohio	3.6	NaN
2001	2001	Nevada	2.4	NaN
2002	2002	Nevada	2.9	NaN
2003	2003	Nevada	3.2	NaN

pd.DataFrame(dict)：嵌套字典：*{ 'out1' : {'in1': v11, 'in2': v12}, 'out2': {...} }*

如果嵌套字典传给DataFrame, 字典外层的键作为列, 内层的键则作为行索引

In [20]:
```
pop = {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
pop
```

Out[20]:
```
{'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [21]:
```
frame3 = pd.DataFrame(pop)
frame3
```

Out[21]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

pd.DataFrame(dict , index)：指定索引

In [22]:
```
pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[22]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

DataFrame 的索引和切片

In [23]:
```
frame2['state']
```

Out[23]:
```
2000      Ohio
2001      Ohio
2002      Ohio
2001    Nevada
2002    Nevada
2003    Nevada
Name: state, dtype: object
```

In [24]:
```
frame2.state
```

Out[24]:
```
2000      Ohio
2001      Ohio
2002      Ohio
2001    Nevada
2002    Nevada
2003    Nevada
Name: state, dtype: object
```

In [25]:
```
frame2.loc[2002]
```

Out[25]:

	year	state	pop	debt
2002	2002	Ohio	3.6	NaN
2002	2002	Nevada	2.9	NaN

In [26]:
```
frame2['state'].loc[2002]
```

Out[26]:
```
2002      Ohio
2002    Nevada
Name: state, dtype: object
```

DataFrame 的赋值

数组或列表：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

In [27]:
```
frame2['debt'] = np.arange(6.)
frame2
```

Out[27]:

	year	state	pop	debt
2000	2000	Ohio	1.5	0.0
2001	2001	Ohio	1.7	1.0
2002	2002	Ohio	3.6	2.0
2001	2001	Nevada	2.4	3.0
2002	2002	Nevada	2.9	4.0
2003	2003	Nevada	3.2	5.0

Series：如果赋值的是一个 Series，就会精确匹配 DataFrame 的索引

In [28]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=[2000, 2001, 2004])
frame2['debt'] = val
frame2
```

Out[28]:

	year	state	pop	debt
2000	2000	Ohio	1.5	-1.2
2001	2001	Ohio	1.7	-1.5
2002	2002	Ohio	3.6	NaN
2001	2001	Nevada	2.4	-1.5
2002	2002	Nevada	2.9	NaN
2003	2003	Nevada	3.2	NaN

In [29]:
```
frame2['debt_null'] = pd.notnull(frame2.debt)
## 注意 : 不能用frame2.debt_null创建新的列
frame2
```

Out[29]:

	year	state	pop	debt	debt_null
2000	2000	Ohio	1.5	-1.2	True
2001	2001	Ohio	1.7	-1.5	True
2002	2002	Ohio	3.6	NaN	False
2001	2001	Nevada	2.4	-1.5	True
2002	2002	Nevada	2.9	NaN	False
2003	2003	Nevada	3.2	NaN	False

del : DataFrame 删除某列

In [30]:
```
del frame2['debt_null']
frame2.columns
```

Out[30]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame 的转置

In [31]:
```
frame3.T
```

Out[31]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

可以输入给 DataFrame 进行构造的数据：

类型	说明
二维 ndarray	数据矩阵, 还可以传入行标和列标
由数组、列表和元组组成的字典	字典的每个键对应的序列会成为 DataFrame 的一列, 所有序列的长度必须相同
Numpy 的结构化 / 记录数组	类似于有数组组成的字典
由 Series 组成的字典	字典的每个键对应的 Series 会成为一列
由字典组成的字典	外层字典的键为列头, 内层字典的键为行索引
字典或 Series 列表	列表各项会成为 DataFrame 的各行, 各项字典/ Series 对应的键/索引会成为列头
由列表或元组组成的列表	类似于二维 ndarray
另一个 DataFrame	该 DataFrame 的索引将被沿用

类型	说明
Numpy 的 MaskedArray	类似于二维 ndarray, 掩码值在 DataFrame 会变成NA/缺失值

DataFrame 的 *name* 和 *values* 属性

如果设置了 DataFrame 的 index 和 columns 的 name 属性, 则这些信息也会被显示出来

In [32]:
```
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

Out[32]:

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

values 属性也会以二维 ndarray 的形式返回 DataFrame 中的数据

In [33]:
```
frame3.values
```

Out[33]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

In [34]:
```
frame2.values
```

Out[34]:
```
array([[2000, 'Ohio', 1.5, -1.2],
       [2001, 'Ohio', 1.7, -1.5],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, nan],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

pandas 的索引对象 —— *index*

index 对象

Index([...], dtype='object')

In [35]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index, obj.index
```

Out[35]:
```
(Index(['a', 'b', 'c'], dtype='object'),
 Index(['a', 'b', 'c'], dtype='object'))
```

In [36]:
```
index[1], index[1:]
```

Out[36]:
```
('b', Index(['b', 'c'], dtype='object'))
```

Index 对象不可进行切片赋值

In [37]:
```
#index[1:] = pd.Index(['d', 'e'])  # Error
```

pd.Index(object)

In [38]:
```
labels = pd.Index(np.arange(3))
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
obj2.index is labels
```

Out[38]:
```
True
```

frame.index , frame.columns

In [39]:
```
print(frame3)
frame3.index, frame3.columns
```

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

Out[39]:
```
(Int64Index([2001, 2002, 2000], dtype='int64', name='year'),
 Index(['Nevada', 'Ohio'], dtype='object', name='state'))
```

index1.append(index2) : 连接 index1 和 index2 , 产生一个新的 Index 对象 [index1, index2]

In [40]:
```
index1 = pd.Index(list(range(2, 5)))
index2 = pd.Index(list(range(0, 3)))
index3 = index1.append(index2)
index1, index2, index3
```

Out[40]:
```
(Int64Index([2, 3, 4], dtype='int64'),
 Int64Index([0, 1, 2], dtype='int64'),
 Int64Index([2, 3, 4, 0, 1, 2], dtype='int64'))
```

index1.difference(index2) : 计算两个 Index 的差值 index1-index2, 得到一个 Index

In [41]:
```
index1 = pd.Index(list(range(2, 5)))
index2 = pd.Index(list(range(0, 3)))
index3 = index1.difference(index2)  # index1-index2
index1, index2, index3
```

Out[41]:
```
(Int64Index([2, 3, 4], dtype='int64'),
 Int64Index([0, 1, 2], dtype='int64'),
 Int64Index([3, 4], dtype='int64'))
```

index1.intersection(index2) : 计算交集

In [42]:
```
index1 = pd.Index(list(range(2, 5)))
index2 = pd.Index(list(range(0, 3)))
index3 = index1.intersection(index2)
index1, index2, index3
```

Out[42]:
```
(Int64Index([2, 3, 4], dtype='int64'),
 Int64Index([0, 1, 2], dtype='int64'),
 Int64Index([2], dtype='int64'))
```

index1.union(index2) : 计算并集

In [43]:
```
index1 = pd.Index(list(range(2, 5)))
index2 = pd.Index(list(range(0, 3)))
index3 = index1.union(index2)
index1, index2, index3
```

Out[43]:
```
(Int64Index([2, 3, 4], dtype='int64'),
 Int64Index([0, 1, 2], dtype='int64'),
 Int64Index([0, 1, 2, 3, 4], dtype='int64'))
```

index.isin(object)：判断 index 中的各值是否在 object 中

In [44]:
```
index = pd.Index(list(range(2, 5)))
obj = list(range(4))
index, obj, index1.isin(obj)
```

Out[44]:
```
(Int64Index([2, 3, 4], dtype='int64'),
 [0, 1, 2, 3],
 array([ True,  True, False]))
```

index.delete(loc)：删除 index 中 loc 处的元素，并返回一个新的索引

In [45]:
```
index = pd.Index(list(range(2, 5)))
index2 = index.delete(1)
index, index2
```

Out[45]:
```
(Int64Index([2, 3, 4], dtype='int64'), Int64Index([2, 4], dtype='int64'))
```

index.drop(labels)：删除传入的值 labels，并返回一个新的索引

In [46]:
```
index = pd.Index(list(range(2, 5)))
index2 = index.drop([2, 3])
index, index2
```

Out[46]:
```
(Int64Index([2, 3, 4], dtype='int64'), Int64Index([4], dtype='int64'))
```

index.insert(loc, item)：将元素 item 插入到位置 loc 处，并返回一个新的索引

In [47]:
```
index = pd.Index(list(range(2, 5)))
index2 = index.insert(2, 2)
index, index2
```

Out[47]:
```
(Int64Index([2, 3, 4], dtype='int64'), Int64Index([2, 3, 2, 4], dtype='int64'))
```

index.is_monotonic：判断 index 是否是升序排列

In [48]:
```
index = pd.Index(list(range(2, 5)))
index, index.is_monotonic
```

Out[48]:
```
(Int64Index([2, 3, 4], dtype='int64'), True)
```

index.is_unique：判断 index 是否有重复值

In [49]:
```
index = pd.Index(list(range(2, 5)))
index, index.is_unique
```

Out[49]:
```
(Int64Index([2, 3, 4], dtype='int64'), True)
```

index.unique( )：计算 index 中唯一值的数组

In [50]:
```
index = pd.Index([1, 1, 2, 2, 3, 3, 5])
index, index.unique()
```

Out[50]:
```
(Int64Index([1, 1, 2, 2, 3, 3, 5], dtype='int64'),
 Int64Index([1, 2, 3, 5], dtype='int64'))
```

重新索引：*reindex*

obj.reindex(index)：根据新索引重新排列

In [51]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj, obj2
```

Out[51]:
```
(d    4.5
 b    7.2
 a   -5.3
 c    3.6
 dtype: float64,
 a   -5.3
 b    7.2
 c    3.6
 d    4.5
 e    NaN
 dtype: float64)
```

obj.reindex(index, method)：填充方式

method = 'ffill'：前向值填充

In [52]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3, obj3.reindex(range(7), method='ffill')
```

Out[52]:
```
(0      blue
 2    purple
 4    yellow
 dtype: object,
 0      blue
 1      blue
 2    purple
 3    purple
 4    yellow
 5    yellow
 6    yellow
 dtype: object)
```

method = 'bfill'：后向值填充

In [53]:
```
obj3.reindex(range(7), method='bfill')
```

Out[53]:
```
0      blue
1    purple
2    purple
3    yellow
4    yellow
5       NaN
6       NaN
dtype: object
```

frame.reindex(index, columns)：会重新索引行和列，其中列用 columns 重新索引

In [54]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                     columns=['Ohio', 'Texas', 'California'])
frame
```

Out[54]:

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

In [55]:
```
frame.reindex(['a', 'b', 'c', 'd'])
```

Out[55]:

	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 [56]:
```
frame.reindex(['a', 'b', 'c', 'd'], columns=['California', 'Los', 'Ohio', 'Texas'])
```

Out[56]:

	California	Los	Ohio	Texas
a	2.0	NaN	0.0	1.0
b	NaN	NaN	NaN	NaN
c	5.0	NaN	3.0	4.0
d	8.0	NaN	6.0	7.0

frame.reindex(..., fill_value)：设置缺失值

In [57]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'e'],
                     columns=['Ohio', 'Texas', 'California'])
frame.reindex(index     = ['a', 'b', 'c', 'd', 'e'],
              columns   = ['California', 'Los', 'Ohio', 'Texas'],
              fill_value= -9999)
```

Out[57]:

	California	Los	Ohio	Texas
a	2	-9999	0	1
b	-9999	-9999	-9999	-9999
c	5	-9999	3	4
d	-9999	-9999	-9999	-9999
e	8	-9999	6	7

frame.reindex(..., method, limit)：前向或后向填充时的最大填充量

In [58]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'd', 'e'],
                     columns=['Ohio', 'Texas', 'California'])
frame.reindex(index = ['a', 'b', 'c', 'd', 'e'],
              method= 'ffill',
              limit = 1)
```

Out[58]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	0.0	1.0	2.0
c	NaN	NaN	NaN
d	3.0	4.0	5.0
e	6.0	7.0	8.0

丢弃指定轴：*drop*

obj.drop(index)：删除指定的索引行

In [59]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj, obj.drop(index=['b', 'd'])
```

Out[59]:
```
(a    0.0
 b    1.0
 c    2.0
 d    3.0
 e    4.0
 dtype: float64,
 a    0.0
 c    2.0
 e    4.0
 dtype: float64)
```

In [60]:
```
frame = pd.DataFrame(np.arange(16).reshape((4, 4)),
                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
                     columns=['one', 'two', 'three', 'four'])
frame
```

Out[60]:

	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

frame.drop(labels, axis) : 删除指定轴 axis 的索引项 labels

In [61]:
```
frame.drop(labels=['Colorado', 'Ohio'], axis='index')
```

Out[61]:

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

In [62]:
```
frame.drop(labels=['two', 'four'], axis='columns')
```

Out[62]:

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

frame.drop(index, columns) : 删除指定的索引行 index 和索引列 columns

In [63]:
```
frame.drop(index=['Colorado'], columns=['two'])
```

Out[63]:

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

frame.drop(..., inplace) : inplace 设置为 True 时就地修改对象 frame

In [64]:
```
print(obj)
obj.drop('c', inplace=True)
print(obj)
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

切片：*obj[...]*, *loc* 和 *iloc*, 整数索引

直接索引：*obj [...]*

Series :

In [65]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj[ ['b', 'a'] ]
```

Out[65]:
```
b    1.0
a    0.0
dtype: float64
```

In [66]:
```
obj[ 2:4 ]
```

Out[66]:
```
c    2.0
d    3.0
dtype: float64
```

In [67]:
```
obj[ [1, 3] ]
```

Out[67]:
```
b    1.0
d    3.0
dtype: float64
```

In [68]:
```
obj['b':'c']  # 包含末端
```

Out[68]:
```
b    1.0
c    2.0
dtype: float64
```

In [69]:
```
obj[ obj<2 ]
```

Out[69]:
```
a    0.0
b    1.0
dtype: float64
```

DataFrame :

In [70]:
```
frame = pd.DataFrame(np.arange(16).reshape((4, 4)),
                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
                     columns=['one', 'two', 'three', 'four'])
frame
```

Out[70]:

	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

传入单一的元素或者列表可以索引列

In [71]:
```
frame[ ['three', 'one'] ]  # 索引列
```

Out[71]:

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

传入整数可以索引行，但是不能单独索引（需要有冒号：）

In [72]:
```
frame[ 2:3 ]  #索引行
```

Out[72]:

	one	two	three	four
Utah	8	9	10	11

布尔型索引

In [73]:
```
frame[ frame['three'] > 6 ]
```

Out[73]:

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

利用 *loc* 和 *iloc* 进行索引

frame.iloc(index_number, columns_number)
frame.loc(index_labels, columns_labels)

In [74]:
```
frame = pd.DataFrame(np.arange(16).reshape((4, 4)),
                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
                     columns=['one', 'two', 'three', 'four'])
frame
```

Out[74]:

	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 [75]:
```
frame.iloc[2:4, [3, 0, 1]]  # 选取第2、3行，第3、0、1列
```

Out[75]:

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

In [76]:
```
frame.loc['Utah':'New York', ['four', 'one', 'two']]
```

Out[76]:

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

这两个索引函数也适用于一个标签或 **多个标签** 的切片

In [77]:
```
frame.iloc[:, :][frame.three > 5]
```

Out[77]:

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

数据索引方式的总结

类型	说明
_df[col_labs] _	通过标签 col_labs , 选取对应的列
_df.loc[ind_labs]_	通过标签 ind_labs , 选取对应的行
_df.loc[:, col_labs]_	通过标签 col_labs , 选取对应的行
_df.loc[ind_labs, col_labs]_	通过标签, 选取对应的行 ind_labs , 列 col_labs
_df.iloc[ind_nums]_	通过整数 ind_nums , 选取对应的行
_df.iloc[:, col_nums]_	通过整数 col_nums , 选取对应的列
_df.iloc[ind_nums, col_nums]_	通过整数, 选取对应的行 ind_nums , 列 col_nums
_df.at[ind_lab, col_lab]_	通过行列的标签, 选取单一的标量
df.iat[i, j]	通过行列的位置（整数）, 选取单一的标量
reindex	通过行列标签, 重塑索引
_get_value, set_value_	通过行列标签, 选取单一值

算术运算与数据对齐

+, -, *, /

> 在将对象相加时如果索引对不同, 则结果的索引就是索引对的并集

自动的数据对齐操作在不重叠的索引处引入了 NaN 值

In [78]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=['c', 'd', 'e', 'f', 'g'])
s1, s2
```

Out[78]:
```
(a    7.3
 b   -2.5
 c    3.4
 d    1.5
 dtype: float64,
 c   -2.1
 d    3.6
 e   -1.5
 f    4.0
 g    3.1
 dtype: float64)
```

In [79]:
```
s1+s2
```

Out[79]:
```
a    NaN
b    NaN
c    1.3
d    5.1
e    NaN
f    NaN
g    NaN
dtype: float64
```

对于 DataFrame，对齐操作会同时发生在行和列上

In [80]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('cde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df1, df2
```

Out[80]:
```
(            b    c    d
 Ohio      0.0  1.0  2.0
 Texas     3.0  4.0  5.0
 Colorado  6.0  7.0  8.0,
           c     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 [81]:
```
df1+df2
```

Out[81]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	NaN	4.0	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	NaN	10.0	12.0	NaN
Utah	NaN	NaN	NaN	NaN

df1.add(df2, fill_value) 等算术运算函数

> 算术运算中的缺失值：***fill_value***：当一个对象中某个位置在另一个对象中找不到时填充一个指定值

In [82]:
```
df1.add(df2, fill_value=0)
```

Out[82]:

	b	c	d	e
Colorado	6.0	7.0	8.0	NaN
Ohio	0.0	4.0	6.0	5.0
Oregon	NaN	9.0	10.0	11.0
Texas	3.0	10.0	12.0	8.0
Utah	NaN	0.0	1.0	2.0

算数算法

类型	说明
_add, radd _	用于加法（+）运算
sub, rsub	用于减法（-）运算
div, rdiv	用于除法（/）运算
floordiv, rfloordiv	用于整除（//）运算
mod, rmod	用于取余（%）运算

类型	说明
divmod, rdivmod	用于取整和取余（// 和 %）运算
mul, rmul	用于乘法（*）运算
pow, rpow	用于指数（**）运算

div 与 *rdiv* 的区别：

> ***df1.div(df2)*** : df1/df2 ; ***df1.rdiv(df2)*** : df2/df1

In [83]:
```
s1 = pd.Series(list(range(1,6)), index=list('abcde'))
s2 = pd.Series(list(range(4,9)), index=list('cdefg'))
s1, s2
```

Out[83]:
```
(a    1
 b    2
 c    3
 d    4
 e    5
 dtype: int64,
 c    4
 d    5
 e    6
 f    7
 g    8
 dtype: int64)
```

In [84]:
```
s1.mod(s2), s1.rmod(s2)  # s1/s2, s2/s1
```

Out[84]:
```
(a    NaN
 b    NaN
 c    3.0
 d    4.0
 e    5.0
 f    NaN
 g    NaN
 dtype: float64,
 a    NaN
 b    NaN
 c    1.0
 d    1.0
 e    1.0
 f    NaN
 g    NaN
 dtype: float64)
```

DataFrame 与 Series 之间的运算

> 广播

匹配列索引 columns，在行上广播 ↓

In [85]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
series = frame.iloc[0]
frame, series
```

Out[85]:
```
(          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,
 b    0.0
 d    1.0
 e    2.0
 Name: Utah, dtype: float64)
```

In [86]:
```
frame-series
```

Out[86]:

	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

参与运算的两个对象的索引不重合时, 形成并集

In [87]:
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
frame + series2
```

Out[87]:

	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

匹配行索引 index , 在列上广播 →

frame.sub(series, axis) : 匹配轴 axis 进行广播, 其他算数运算函数同理

In [88]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
series3 = frame['d']
frame, series3
```

Out[88]:
```
(          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,
 Utah      1.0
 Ohio      4.0
 Texas     7.0
 Oregon   10.0
 Name: d, dtype: float64)
```

In [89]:
```
frame.sub(series3, axis='index')
```

Out[89]:

	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

函数应用与映射

Numpy 函数的应用

In [90]:
```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[90]:

	b	d	e
Utah	1.014427	-0.171110	-0.209578
Ohio	0.384411	0.098477	0.425979
Texas	0.618812	-0.087757	0.498332
Oregon	0.329589	-1.004679	0.205972

In [91]:
```
np.abs(frame)
```

Out[91]:

	b	d	e
Utah	1.014427	0.171110	0.209578
Ohio	0.384411	0.098477	0.425979
Texas	0.618812	0.087757	0.498332
Oregon	0.329589	1.004679	0.205972

frame.apply(func, axis)：将函数应用到各列或各行所形成的一维数组上

> 默认对 index 进行操作 ↓，axis='columns' 时对 columns 进行操作 →

In [92]:
```
frame =  pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[92]:

	b	d	e
Utah	-0.504649	-0.773222	3.765686
Ohio	-1.138202	0.029167	0.841352
Texas	-0.649582	0.596585	-0.854243
Oregon	-0.353114	1.173491	-1.507888

In [93]:
```
f = lambda x: x.max() - x.min()  # func_name = lambda vars_in: vars_out
frame.apply(f), frame.apply(f, axis='columns')  # f(frame)
```

Out[93]:
```
(b    0.785088
 d    1.946713
 e    5.273573
 dtype: float64,
 Utah      4.538908
 Ohio      1.979554
 Texas     1.450828
 Oregon    2.681378
 dtype: float64)
```

In [94]:
```
def f(x):
    return x.max()-x.min()
frame.apply(f)
```

Out[94]:
```
b    0.785088
d    1.946713
e    5.273573
dtype: float64
```

计算各列/各行的最大值和最小值

> 对各行/列操作的返回的结果可以是由多个值组成的 Series

In [95]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
```

Out[95]:

	b	d	e
min	-1.138202	-0.773222	-1.507888
max	-0.353114	1.173491	3.765686

frame.applymap(func)：将函数映射到对象中的各个元素进行操作

In [96]:
```
frame
```

Out[96]:

	b	d	e
Utah	-0.504649	-0.773222	3.765686
Ohio	-1.138202	0.029167	0.841352
Texas	-0.649582	0.596585	-0.854243
Oregon	-0.353114	1.173491	-1.507888

In [97]:
```
fmap = lambda x: '%.2f' % (x*10) # 只保留两位小数
frame.applymap(fmap)
```

Out[97]:

	b	d	e
Utah	-5.05	-7.73	37.66
Ohio	-11.38	0.29	8.41
Texas	-6.50	5.97	-8.54
Oregon	-3.53	11.73	-15.08

排序

obj.sort_index(axis, ascending)：按索引排序

> ascending : 是否按升序排序

In [98]:
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()
```

Out[98]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

In [99]:
```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
frame
```

Out[99]:

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

In [100]:
```
frame.sort_index()
```

Out[100]:

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

In [101]:
```
frame.sort_index(axis=1)
```

Out[101]:

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

降序

In [102]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[102]:

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

obj.sort_values(by) : 按值排序

In [103]:
```
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

Out[103]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

在排序时, 任何缺失值默认都会被放到末尾

In [104]:
```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

Out[104]:
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

排序DataFrame时, 将一个或多个列的名字传递给 *by* 选项可以 根据一个或多个列的值排序

In [105]:
```
frame = pd.DataFrame({'b': [4, 7, 2, 2], 'a': [0, 1, 0, 1], 'c': [4, 3, 2, 1]})
frame
```

Out[105]:

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

In [106]:
```
frame.sort_values(by='b')
```

Out[106]:

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

In [107]:
```
frame.sort_values(by=['b', 'c'])
```

Out[107]:

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

obj.rank(axis, ascending, method)：返回各组的平均排名

In [108]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj.rank()
```

Out[108]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

method='first'：根据值在原数据中出现的顺序给出排名

In [109]:
```
obj.rank(method='first')
```

Out[109]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

按降序排序

In [110]:
```
obj.rank(ascending=False, method='min')  # 并列第一、并列第三...
```

Out[110]:
```
0    1.0
1    7.0
2    1.0
3    3.0
4    5.0
5    6.0
6    3.0
dtype: float64
```

In [111]:
```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                      'c': [-2, 5, 8, -2.5]})
frame
```

Out[111]:

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

In [112]:
```
frame.rank(axis='columns')
```

Out[112]:

	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

method 选项

类型	说明
_average _	给相等的分组分配平均排名（默认）
min	给相等的分组分配最小排名
max	给相等的分组分配最大排名
first	对相等的分组按出现顺序排名
dense	类似于 min，但是组间排名总是相差1

统计计算

In [113]:
```
frame = pd.DataFrame([[1, np.nan], [2, 4],
                      [np.nan, np.nan], [3, 5]],
                     index=['a', 'b', 'c', 'd'],
                     columns=['one', 'two'])
```

obj.sum(axis, skipna, level)：求和

In [114]:
```
frame
```

Out[114]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	3.0	5.0

In [115]:
```
frame.sum()
```

Out[115]:
```
one    6.0
two    9.0
dtype: float64
```

In [116]:
```
frame.sum(axis=1)
```

Out[116]:
```
a    1.0
b    6.0
c    0.0
d    8.0
dtype: float64
```

In [117]:
```
frame.sum(axis=1, skipna=False)
```

Out[117]:
```
a    NaN
b    6.0
c    NaN
d    8.0
dtype: float64
```

obj.mean(axis, skipna, level)：求平均数

In [118]:
```
frame.mean()
```

Out[118]:
```
one    2.0
two    4.5
dtype: float64
```

obj.median(axis, skipna, level)：求中位数

In [119]:
```
frame.median()
```

Out[119]:
```
one    2.0
two    4.5
dtype: float64
```

obj.mad(axis, skipna, level)：求绝对离差 $\frac{1}{n}\sum|x-\bar{x}|$

In [120]:
```
frame.mad()
```

Out[120]:
```
one    0.666667
two    0.500000
dtype: float64
```

obj.var(axis, skipna, level)：求方差 $\frac{1}{n}\sum(x-\bar{x})^2$

In [121]:
```
frame.var()
```

Out[121]:
```
one    1.0
two    0.5
dtype: float64
```

obj.std(axis, skipna, level)：求标准差 $\sqrt{\frac{1}{n}\sum(x-\bar{x})^2}$

In [122]:
```
frame.std()
```

Out[122]:
```
one    1.000000
two    0.707107
dtype: float64
```

obj.skew(axis, skipna, level)：求偏度 $\frac{\frac{1}{n}\sum(x-\bar{x})^3}{\sigma^3}$

In [123]:
```
frame.skew()
```

Out[123]:
```
one    0.0
two    NaN
dtype: float64
```

obj.kurt(axis, skipna, level)：求峰度 $\frac{\frac{1}{n}\sum(x-\bar{x})^4}{\sigma^4}$

In [124]:
```
pd.DataFrame([1,2,3,4,3,2,1,0]).kurt()
```

Out[124]:
```
0   -0.7
dtype: float64
```

obj.quantile(q, axis)：求分位数（二分位、四分位、四分之三分位 ... ）

In [125]:
```
frame, frame.quantile(0.5), frame.mean()
```

Out[125]:
```
(   one  two
 a  1.0  NaN
 b  2.0  4.0
 c  NaN  NaN
 d  3.0  5.0,
 one    2.0
 two    4.5
 Name: 0.5, dtype: float64,
 one    2.0
 two    4.5
 dtype: float64)
```

In [126]:
```
frame.quantile(0.75) - frame.quantile(0.25) ## 离差IQR
```

Out[126]:
```
one    1.0
two    0.5
dtype: float64
```

obj.cumsum(axis, skipna, level), obj.cumprod(axis, skipna, level)：累加, 累乘

In [127]:
```
frame
```

Out[127]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	3.0	5.0

In [128]:
```
frame.cumsum()
```

Out[128]:

	one	two
a	1.0	NaN
b	3.0	4.0
c	NaN	NaN
d	6.0	9.0

In [129]:
```
frame.cumprod()
```

Out[129]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	6.0	20.0

obj.cummin(axis, skipna, level), obj.cummax(axis, skipna, level)：累计最小值和累计最大值

In [130]:
```
frame.cummax()
```

Out[130]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	3.0	5.0

obj.idmax(axis, skipna, level), obj.idmin(axis, skipna, level)：返回最大值或最小

值的索引

In [131]:
```
frame
```

Out[131]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	3.0	5.0

In [132]:
```
frame.idxmax()
```

Out[132]:
```
one    d
two    d
dtype: object
```

In [133]:
```
frame.idxmax(axis=1)
```

Out[133]:
```
a    one
b    two
c    NaN
d    two
dtype: object
```

obj.describe( )：返回多个汇总统计

In [134]:
```
frame
```

Out[134]:

	one	two
a	1.0	NaN
b	2.0	4.0
c	NaN	NaN
d	3.0	5.0

In [135]:
```
frame.describe()
```

Out[135]:

	one	two
count	3.0	2.000000
mean	2.0	4.500000
std	1.0	0.707107
min	1.0	4.000000
25%	1.5	4.250000
50%	2.0	4.500000
75%	2.5	4.750000
max	3.0	5.000000

对于非数值类型：

In [136]:
```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj.head(), obj.describe()
```

Out[136]:
```
(0    a
 1    a
 2    b
 3    c
 4    a
 dtype: object,
 count     16
 unique     3
 top        a
 freq       8
 dtype: object)
```

series1.cov(series2), frame.cov( )：计算协方差

$$Cov(X,Y) = \frac{1}{n}\sum(x-\overline{x})(y-\overline{y})$$

In [137]:
```
# 数据准备
price = pd.read_pickle('pydata-book-2nd-edition/examples/yahoo_price.pkl') #股票价格
volume = pd.read_pickle('pydata-book-2nd-edition/examples/yahoo_volume.pkl') #股票成交量
returns = price.pct_change() #计算变化率 : (后一个值 - 前一个值) / 前一个值
returns
```

Out[137]:

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	NaN	NaN	NaN	NaN
2010-01-05	0.001729	-0.004404	-0.012080	0.000323
2010-01-06	-0.015906	-0.025209	-0.006496	-0.006137
2010-01-07	-0.001849	-0.023280	-0.003462	-0.010400
2010-01-08	0.006648	0.013331	0.010035	0.006897
...	...	...	...	...
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

1714 rows × 4 columns

In [138]:
```
returns['MSFT'].cov(returns['IBM'])
```

Out[138]:
```
8.870655479703546e-05
```

In [139]:
```
returns.cov()
```

Out[139]:

	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

series1.corr(series2), frame.corr( )：计算相关系数

$$\rho = \frac{Cov(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum(x-\overline{x})(y-\overline{y})}{\sqrt{\sum(x-\overline{x})^2}\cdot\sqrt{\sum(y-\overline{y})^2}}$$

In [140]:
```
returns['MSFT'].corr(returns['IBM'])
```

Out[140]:
```
0.4997636114415114
```

In [141]:
```
returns.corr()
```

Out[141]:

	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

obj1.corrwith(obj2, axis)：计算 obj1 与 obj2 对应行或列的相关系数

In [142]:
```
returns.corrwith(returns.IBM)
```

Out[142]:
```
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

In [143]:
```
returns.corrwith(volume) # 一一对应
```

Out[143]:
```
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

In [144]:
```
returns.T.corrwith(volume.T, axis=1)
```

Out[144]:
```
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

series.unique(series2)：得到 series 唯一值数组

In [145]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
obj.unique()
```

Out[145]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

obj.isin(list)：判断 obj 的元素是否在 list 中

In [146]:
```
frame = pd.DataFrame([[1, np.nan], [2, 4],
                      [np.nan, np.nan], [3, 5]],
                     index=['a', 'b', 'c', 'd'],
                     columns=['one', 'two'])
frame, frame.isin([1, 2])
```

Out[146]:
```
(   one  two
 a  1.0  NaN
 b  2.0  4.0
 c  NaN  NaN
 d  3.0  5.0,
      one    two
 a   True  False
 b   True  False
 c  False  False
 d  False  False)
```

obj.value_counts(normalize, sort)：计算 obj 中各值出现的频率

In [147]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
obj.value_counts()
```

Out[147]:
```
c    3
a    3
b    2
d    1
dtype: int64
```

In [148]:
```
frame = pd.DataFrame([[1, 3, 5], [2, 4, 6],
                      [1, 3, 6], [2, 4, 6]],
                     index=['a', 'b', 'c', 'd'],
                     columns=['one', 'two', 'thr'])
frame, frame.value_counts()
```

Out[148]:
```
(   one  two  thr
 a    1    3    5
 b    2    4    6
 c    1    3    6
 d    2    4    6,
 one  two  thr
 2    4    6      2
 1    3    5      1
           6      1
 dtype: int64)
```

Type *Markdown* and LaTeX: α^2