In [1]:
```
import pandas as pd     #pandas擅长处理表格和异质型数据，numpy适合同质型数值类数组数据
from pandas import Series, DataFrame    #Series：一维 DataFrame：多维
obj = pd.Series([4,7,-5,3])
obj    #会显示dtype
```

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

In [2]:
```
obj2 = pd.Series([4,7,-5,3], index=['d','b','a','c'])    #padas.Series建立数组可建立对应的索引
obj2
```

Out[2]:
```
d    4
b    7
a   -5
c    3
dtype: int64
```

In [3]:
```
obj2.values    #.values显示数组的值
```

Out[3]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [4]:
```
obj2.index    #.index显示数组索引，字符串作为索引不受数学运算影响（不变）
```

Out[4]:
```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

In [5]:
```
obj2['a']    #[]中加入索引可以输出对应的值
```

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

In [6]:
```
obj2['d'] = 6    #选中索引后可替换值
obj2[['c','a','d']]    #中括号中套中括号可以一次性调用多个索引
```

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

In [7]:
```
obj2[obj2>0]
```

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

In [8]:
```
obj2*2    #pandas中数组也是逐元素自带for循环操作
```

Out[8]:
```
d    12
b    14
a   -10
c     6
dtype: int64
```

In [9]:
```
'e' in obj2    #用in判断是否含有元素，可以把Series看作一个长度固定且有序的字典
```

Out[9]:
```
False
```

In [10]:
```
sdata = {'ohio':35000, 'texas':71000, 'oregon':16000, 'utah':5000}
obj3 = pd.Series(sdata)    #将python中的字典导入pandas形成数组
obj3
```

Out[10]:
```
ohio      35000
texas     71000
oregon    16000
utah       5000
dtype: int64
```

In [11]:
```
states = ['california', 'ohio', 'oregon','texas']
obj4 = pd.Series(sdata, index=states)
obj4    #新旧索引中有重复的，不变，新索引对应不到旧值显示NaN, 对应不到新索引的旧值被舍弃
```

Out[11]:
```
california        NaN
ohio          35000.0
oregon        16000.0
texas         71000.0
dtype: float64
```

In [12]:
```
pd.isnull(obj4)      #判断哪些值是null
```

Out[12]:
```
california     True
ohio          False
oregon        False
texas         False
dtype: bool
```

In [13]:
```
pd.notnull(obj4)     #判断哪些值不是null
```

Out[13]:
```
california    False
ohio           True
oregon         True
texas          True
dtype: bool
```

In [14]:
```
obj3+obj4     #series自动对齐索引
```

Out[14]:
```
california         NaN
ohio           70000.0
oregon         32000.0
texas         142000.0
utah               NaN
dtype: float64
```

In [15]:
```
obj4.name = 'population'     #series对象自身和其索引都有name属性
obj4.index.name = 'state'
obj4
```

Out[15]:
```
state
california        NaN
ohio          35000.0
oregon        16000.0
texas         71000.0
Name: population, dtype: float64
```

In [16]:
```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']    #series的索引可以通过按位置赋值的方式进行改变
obj
```

Out[16]:
```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

In [17]:
```
data = {'state':['ohio', 'ohio', 'ohio', 'nevada', 'nevada', 'nevada'], 'year':[2000, 2001, 2002, 2001, 2002, 2003],'pop':[1.5, 1.
frame = pd.DataFrame(data)     #将python中的字典导入frame形成可视化表格（二维）
frame
```

Out[17]:

	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
5	nevada	2003	3.2

In [18]:
```
frame.head()     #head只选出头部的5行
```

Out[18]:

	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 [19]:
```
pd.DataFrame(data, columns=['year', 'state', 'pop'])     #指定dataframe的列顺序
```

Out[19]:

	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

In [20]:
```
frame2 = pd.DataFrame(data, columns=['year','state','pop','debt'], index=['one','two','three','four','five','six'])
frame2    #如果你传的列不包含在字典中，将会出现一列缺失值，用index对值对应地赋予行索引
```

Out[20]:

	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 [21]:
```
frame2.columns     #用colums调用列索引，缺失值列依旧在列
```

Out[21]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

In [22]:
```
frame2.state    #.列索引输出列的全部值
```

Out[22]:
```
one        ohio
two        ohio
three      ohio
four     nevada
five     nevada
six      nevada
Name: state, dtype: object
```

In [23]:
```
frame2.loc['three']   #用loc【】调用行索引，输出一行中所有值组成的一个series，name为行索引
```

Out[23]:
```
year     2002
state    ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

In [24]:
```
frame2['debt'] = 16.5     #列的引用是可以用来修改列值的
frame2
```

Out[24]:

	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 [25]:
```
import numpy as np
frame2['debt'] = np.arange(6.)     #利用numpy对frame2的列值进行逐元素处理，numpy是专项功能（偏科生）
frame2
```

Out[25]:

	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

In [26]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame2['debt'] = val
frame2      #series赋值给一列时，series的索引将会按照dataframe的索引重新排列，并在空缺地方填充缺失值
```

Out[26]:

	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

In [27]:
```
frame2['eastern'] = frame2.state == 'ohio'    #新增一列
frame2
```

Out[27]:

	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

In [28]:
```
del frame2['eastern']    #删除一列
frame2.columns
```

Out[28]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

In [29]:
```
pop = {'nevada':{2001:2.4, 2002:2.9}, 'ohio':{2000:1.5, 2001:1.7, 2002:3.6}}
frame3 = pd.DataFrame(pop)
frame3    #第二层内嵌的索引将成为行索引
```

Out[29]:

	nevada	ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [30]:
```
frame3.T    #可以用类似numpy的T转置矩阵
```

Out[30]:

	2001	2002	2000
nevada	2.4	2.9	NaN
ohio	1.7	3.6	1.5

In [31]:
```
pd.DataFrame(pop, index=[2001, 2002, 2003])     #可直接编辑已经写好的索引，相同索引对应的值还是会保留
```

Out[31]:

	nevada	ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

In [32]:
```
pdata = {'ohio':frame3['ohio'][:-1], 'nevada':frame3['nevada'][:2]}     #包含series的字典也可以用于构造dataframe
                                                                        #ohio从1.5（-1位）开始往前取（不包含1.5），nevada从nan（2位）开始往前取（不包含
pd.DataFrame(pdata)
```

Out[32]:

	ohio	nevada
2001	1.7	2.4
2002	3.6	2.9

In [107]:
```
frame3.index.name = 'year';frame3.columns.name = 'state'     #行索引和列索引都可以被命名
frame3
```

Out[107]:

state	nevada	ohio
year		
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [34]:
```
frame3.values     #dataframe的values属性会将包含的数据以二维数组ndarray形式输出
```

Out[34]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

In [35]:
```
frame2.values     #如果dataframe的列是不一样的dtypes，则会自动选择适合所有值的dtypes
```

Out[35]:
```
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)
```

In [36]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index      #数组或标签序列都可以被内部转换成object，索引对象是不可变的
index
```

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

In [37]:
```
labels = pd.Index(np.arange(3))     #索引对象的不变性使得在多种数据中分享索引更为安全
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
obj2
```

Out[37]:
```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

In [38]:
```
obj2.index is labels     #is判断两者是否一致
```

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

In [39]:
```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])     #不同于python，pandas索引对象可以包含重复标签
dup_labels
```

Out[39]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

In [40]:
```
#一些索引对象的方法和属性
#append     将额外的索引对象粘贴到原索引后，产生一个新的索引
#difference     计算两个索引的差集
#intersection      计算两个索引的交集
#union     计算两个素引的并集
#isin     计算表示每一个值是否在传值容器中的布尔数组
#delete     将位置1的元素删除，并产生新的索引
#drop      根据传参删除指定索引值，并产生新的素引
#insert     在位置1插入元素，并产生新的索引
#is_monotonic      如果索引序列递增则返回True
#is_unique     如果索引序列唯一则返回 True
#unique     计算索引的唯一值序列
```

In [41]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d','b','a','c'])
obj2 = obj.reindex(['a','b','c','d','e'])     #reindex将数据按照新索引进行排列，值不存在则引入缺失值
obj2
```

Out[41]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

In [42]:
```
obj3 = pd.Series(['blue','purple','yellow'], index=[0,2,4])
obj3
```

Out[42]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [43]:
```
obj3.reindex(range(6), method='ffill')     #ffill方法会根据range将值向前填充，保证每个索引都有值，bfill则是向后填充
```

Out[43]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

In [102]:
```
frame = pd.DataFrame(np.arange(9).reshape((3,3)),index=['a','c','d'], columns=['ohio','texas','california'])
frame
```

Out[102]:

	ohio	texas	california
a	0	1	2
c	3	4	5
d	6	7	8

In [103]:
```
frame = frame.reindex(['a','b','c','d'])     #reindex可以改变行索引、列索引，也可以同时改变二者
frame                                        #reindex无说明则是改行索引，说了columns=则是改列标签
```

Out[103]:

	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 [109]:
```
states = ['texas', 'utah', 'california']
frame = frame.reindex(columns=states)     #没有赋值操作的话，只是改变视图，没改本体
frame
```

Out[109]:

	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

In [110]:
```
frame.loc[['a','b','c','d'], states]    #loc进行更为简洁的标签索引
```

Out[110]:

	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

In [51]:
```
obj = pd.Series(np.arange(5.), index=['a','b','c','d','e'])
obj
```

Out[51]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [52]:
```
obj.drop(['d','c'])    #series中drop根据行标签删掉行
```

Out[52]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

In [53]:
```
data = pd.DataFrame(np.arange(16).reshape((4,4)), index=['ohio','colorado','utah','new york'], columns=['one','two','three','four'
data
```

Out[53]:

	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 [54]:
```
data.drop(['colorado','ohio'])   #删行直接删，形似reindex
```

Out[54]:

	one	two	three	four
utah	8	9	10	11
new york	12	13	14	15

In [55]:
```
data.drop(['two','four'], axis='columns')     #删列需要赋值参数=columns，形似reindex
```

Out[55]:

	one	three
ohio	0	2
colorado	4	6
utah	8	10
new york	12	14

In [56]:
```
data.drop(['two','four'], axis=1)     #axis=0 指行，axis=1 指列
```

Out[56]:

	one	three
ohio	0	2
colorado	4	6
utah	8	10
new york	12	14

In [57]:
```
obj.drop('c', inplace=True)    #参数 inplace=True 表示直接在原 DataFrame 或 Series 上进行修改，而不是返回一个新对象
obj
```

Out[57]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [58]:
```
obj = pd.Series(np.arange(4.), index=['a','b','c','d'])
obj['b':'c']    #python切片不包含尾部，series的切片与之不同
```

Out[58]:
```
b    1.0
c    2.0
dtype: float64
```

In [59]:
```
obj['b':'c'] = 5
obj
```

Out[59]:
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

In [60]:
```
data[:2]    #取到第2位，但是不包括第2位，python经典前闭后开
```

Out[60]:

	one	two	three	four
ohio	0	1	2	3
colorado	4	5	6	7

In [61]:
```
data[data['three'] > 5]    #在行或列中筛选值
```

Out[61]:

	one	two	three	four
colorado	4	5	6	7
utah	8	9	10	11
new york	12	13	14	15

In [62]:
```
data < 5      #对data操作，就是对整个表格进行逐元素操作，每个元素在各自的位置返回结果
              #语法上与numpy二维数组相似
```

Out[62]:

	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 [63]:
```
data.loc['colorado', ['two','three']]    #loc通过标签直接选出数据
```

Out[63]:
```
two      5
three    6
Name: colorado, dtype: int32
```

In [64]:
```
data.iloc[2, [3,0,1]]      #iloc通过标签的位置数选出数据，2即行索引，301各为列索引
```

Out[64]:
```
four    11
one      8
two      9
Name: utah, dtype: int32
```

In [65]:
```
data.loc[:'utah','two']    #索引功能还能用于切片
```

Out[65]:
```
ohio        1
colorado    5
utah        9
Name: two, dtype: int32
```

In [ ]:
```
#df.loc[val]    根据标签选择DataFrame的单行或多行
#df.loc[:, val]    根据标签选择单列或多列
#df.loc[val1, val2]    同时选择行和列中的一部分
#df.iloc[where]    根据整数位置选择单行或多行
#df.iloc[:, where]    根据整数位置选择单列或多列
#df.iloc[where_i,where_j]    根据整数位置选择行和列
#df.at[label_i, label _j]    根据行、列标签选择单个标量值
#df.iat[i,j]    根据行、列整数位置选择单个标量值
#reindex方法    通过标签选择行或列
#get_value, set_value 方法    根据行和列的标签设置单个值
```

In [66]:
```
s1 = pd.Series([7.3,-2.5,3.4,1.5], index=['a','c','d','e'])
s2 = pd.Series([-2.1,3.6,-1.5,4,3.1], index=['a','c','e','f','g'])
s1 + s2    #对象相加时，如果存在某个索引对不相同，则返回的索引将是索引对的并集，但是不匹配的索引对应的值为缺失值
           #无论是series还是dataframe相加都会执行行列对齐并用缺失值填充未匹配的索引所对的值
```

Out[66]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [67]:
```
import numpy as np
df1 = pd.DataFrame(np.arange(12.).reshape((3,4)), columns=list('abcd'))
df2 = pd.DataFrame(np.arange(20.).reshape((4,5)), columns=list('abcde'))
df1, df2
```

Out[67]:
```
(     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,
       a     b     c     d     e
 0   0.0   1.0   2.0   3.0   4.0
 1   5.0   6.0   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)
```

In [68]:
```
df1 + df2    #用+会导致不重叠的位置出现NA值
```

Out[68]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

In [69]:
```
df1.add(df2, fill_value=0)    #用add就能自动补充不重叠的值，不会出现nan
                              #是指将两个DataFrame df1 和 df2 对应位置的元素相加，不重叠的值加上fill_value(这里加0等于原元素)
```

Out[69]:

	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 [70]:
```
df1.rdiv(1)    #表示1/df1，r开头的算术方法的参数都是倒置的
```

Out[70]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

In [71]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4,3)), columns=list('bde'), index=['utah','ohio','texas','oregon'])
series = frame.iloc[0]
frame - series    #形似numpy，在列表中减去一行，该减法在每一行都进行了操作，这就是所谓的广播机制
```

Out[71]:

	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

In [72]:
```
frame
```

Out[72]:

	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 [73]:
```
series3 = frame['d']
frame.sub(series3, axis='index')    #改在列上进行广播，必须使用算术方法，axis值用于匹配轴，=‘inde’或=0，都是在列上广播
```

Out[73]:

	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

In [74]:
```
frame = pd.DataFrame(np.random.randn(4,3), columns=list('bde'), index=['utah','ohio','texas','oregon'])
np.abs(frame)
```

Out[74]:

	b	d	e
utah	0.051711	0.614977	0.035341
ohio	2.316939	0.582167	0.427940
texas	0.185560	0.916395	1.414069
oregon	0.913377	1.451845	0.058976

In [75]:
```
f = lambda x:x.max()-x.min()     #用lambda定义一个函数
frame.apply(f)      #用apply应用函数，默认函数对行操作
```

Out[75]:
```
b    2.502500
d    2.368241
e    1.842009
dtype: float64
```

In [76]:
```
frame.apply(f, axis='columns')     #同样地，对列操作要定义axis='columns'
```

Out[76]:
```
utah      0.579636
ohio      2.899106
texas     2.330464
oregon    1.510821
dtype: float64
```

In [77]:
```
#def 和 lambda的不同
#语法形式不同：def 是一个完整的语句，用于定义一个常规的函数，而 lambda 是一个表达式，用于创建一个匿名函数。
#函数体大小不同：def 可以包含多个语句和复杂的逻辑，而 lambda 只能包含一个表达式。
#返回值不同：def 函数可以使用 return 语句返回一个值，而 lambda 表达式的结果就是该表达式的返回值。
#函数名称不同：def 定义的函数具有名称，可以在程序的其他地方被调用，而 lambda 表达式是匿名的，没有名称，只能在其定义的位置被引用。
```

In [78]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min','max'])     #将最大最小结果以表格的方式return
frame.apply(f)
```

Out[78]:

	b	d	e
min	-0.185560	-0.916395	-0.427940
max	2.316939	1.451845	1.414069

In [79]:
```
format = lambda x: '%.2f' % x    #'%.2f' 是格式化字符串，表示将 x 格式化为一个浮点数，并保留两位小数。 % 是格式化字符串的占位符，
frame.applymap(format)
```

Out[79]:

	b	d	e
utah	-0.05	-0.61	-0.04
ohio	2.32	-0.58	-0.43
texas	-0.19	-0.92	1.41
oregon	0.91	1.45	-0.06

In [80]:
```
frame['e'].map(format)     #series有map方法，可以将逐元素的函数应用在单独一列的series上
```

Out[80]:
```
utah      -0.04
ohio      -0.43
texas      1.41
oregon    -0.06
Name: e, dtype: object
```

In [81]:
```
frame = pd.DataFrame(np.arange(8).reshape((2,4)), index=['three','one'], columns=['d','a','b','c'])
frame.sort_index()     #sort_index 使行根据行索引升序排序
```

Out[81]:

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

In [82]:
```
frame.sort_index(axis=1, ascending=False)      #使轴（axis）=1，使对列标签升序排列，ascending=false 表示倒叙排列
```

Out[82]:

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

In [83]:
```
obj.sort_values()     #series可以用sort_values对值升序排列
```

Out[83]:
```
a    0.0
d    3.0
b    5.0
c    5.0
dtype: float64
```

In [84]:
```
frame = pd.DataFrame({'b':[4,7,-3,2], 'a':[0,1,0,1]})
frame
```

Out[84]:

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

In [85]:
```
frame.sort_values(by='b')     #可以用by选择一列进行值排列，不能对整个表格进行值排列，nan一般排到最后
```

Out[85]:

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

In [86]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj.rank()     #rank通过将平均排名分配到每个组来打破平级关系
```

Out[86]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

In [87]:
```
obj.rank(method='first')     #根据数据观察顺序进行排名分配
```

Out[87]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

In [88]:
```
obj.rank(ascending=False, method='max')     #倒叙排名，并将值分配给最大排名（并列第1就都变成2，取大的）
```

Out[88]:
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

In [89]:
```
import numpy as np
df = pd.DataFrame(np.random.randn(4,3), index=['a','a','b','b'])
df
```

Out[89]:

	0	1	2
a	0.549243	0.172364	-0.012139
a	0.249199	-1.941124	1.008628
b	0.163263	0.168224	0.381620
b	0.526940	0.688168	-0.430099

In [90]:
```
df.loc['b']     #标签可重复，索引时会输出这一标签对的所有值
```

Out[90]:

	0	1	2
b	0.163263	0.168224	0.381620
b	0.526940	0.688168	-0.430099

In [91]:
```
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'])
df
```

Out[91]:

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

In [92]:
```
df.sum()    #sum()返回一个一列上各行加和的series（加的还是各行，只是位置处于列上）
```

Out[92]:
```
one    9.25
two   -5.80
dtype: float64
```

In [93]:
```
df.sum(axis='columns')    #axis='columns'或1，则会将一行上各列加和的series
```

Out[93]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

In [94]:
```
df.mean(axis='columns', skipna=False)    #NA值一般是自动排除，禁用skipna可不排除NA值
```

Out[94]:
```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

In [95]:
```
df.idxmax()    #返回最大值对应的索引值，idmin()显示最小值
```

Out[95]:
```
one    b
two    d
dtype: object
```

In [96]:
```
df.cumsum()    #在一列上的每行的值累加，并显示每一步累加的过程结果
```

Out[96]:

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

In [97]:
```
df.describe()    #在一列上对每行做多种汇总统计
```

Out[97]:

	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 [98]:
```
obj = pd.Series(['a', 'a', 'b', 'c']*4)
obj.describe()    #对非数值对象则返回一下统计数值
```

Out[98]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

In [ ]:
```
#count      非NA值的个数
#describe     计算Series 或DataFrame 各列的汇总统计集合
#min, max    计算最小值、最大值
#argmin, argmax      分别计算最小值、最大值所在的索引位置（整数）
#idxmin,idxmax      分别计算最小值或最大值所在的索引标签
#quantile    计算样本的从0到1间的分位数
#sum       加和
#mean      均值
#median    中位数（50%分位数）
#mad       平均值的平均绝对偏差
#prod      所有值的积
#var       值的样本方差
#std       值的样本标准差
#skew      样本偏度（第三时刻）值
#kurt      样本峰度（第四时刻）的值
#cumsum    累计值
#cummin, cummax      累计值的最小值或最大值
#cumprod      值的累计积
#diff     计算第一个算术差值（对时间序列有用））
#pct_change      计算百分比
```

In [ ]:
```
conda install pandas-datareader
```

In [ ]:
```

```

In [145]:
```
import pandas_datareader as web

stocks = ['AAPL', 'IBM', 'MSFT', 'GOOG']

# 使用 Alpha Vantage 数据源
data = {ticker: web.DataReader(ticker, data_source="av-daily", start='2023-09-18', end='2023-09-25', api_key='RM2FEYF4GJFUQCV1')
data     #利用for循环（每一次循环，换一个索引和一个对应的数据源）和字典的特性，巧妙地一步将多个股票的数据合成一个字典
```

Out[145]:
```
{'AAPL':                open      high       low    close     volume
 2023-09-18  176.48  179.380  176.17  177.97   67257573
 2023-09-19  177.52  179.630  177.13  179.07   51826941
 2023-09-20  179.26  179.695  175.40  175.49   58436181
 2023-09-21  174.55  176.300  173.86  173.93   63149116
 2023-09-22  174.67  177.079  174.05  174.79   56725385
 2023-09-25  174.20  176.970  174.15  176.08   46172740,
 'IBM':                 open       high       low    close    volume
 2023-09-18  145.77  146.4800  145.06  145.09   2508062
 2023-09-19  145.00  146.7200  144.66  146.52   3945423
 2023-09-20  148.36  151.9299  148.13  149.83   9636681
 2023-09-21  149.00  149.2500  147.31  147.38   4944786
 2023-09-22  147.41  148.1000  146.82  146.91   2562216
 2023-09-25  146.57  147.4300  146.25  146.48   2694245,
 'MSFT':                open      high       low    close     volume
 2023-09-18  327.80  330.4000  326.36  329.06   16834208
 2023-09-19  326.17  329.3900  324.51  328.65   16514487
 2023-09-20  329.51  329.5900  320.51  320.77   21436525
 2023-09-21  319.26  325.3499  315.00  319.53   35560362
 2023-09-22  321.32  321.4500  316.15  317.01   21447887
 2023-09-25  316.59  317.6700  315.00  317.54   17835964,
 'GOOG':                open      high       low    close     volume
 2023-09-18  137.63  139.930  137.63  138.96   16233590
 2023-09-19  138.25  139.175  137.50  138.83   15484644
 2023-09-20  138.83  138.840  134.52  134.59   21473533
 2023-09-21  132.39  133.190  131.09  131.36   22058375
 2023-09-22  131.68  133.010  130.51  131.25   17355284
 2023-09-25  130.77  132.220  130.03  132.17   14650032}
```

In [146]:
```
price = pd.DataFrame({ticker: data['close'] for ticker, data in data.items()})
volume = pd.DataFrame({ticker: data['volume'] for ticker, data in data.items()})
price     ##再根据不同的索引，从字典中for循环抽取所需要的数据组成一个表格
```

Out[146]:

	AAPL	IBM	MSFT	GOOG
2023-09-18	177.97	145.09	329.06	138.96
2023-09-19	179.07	146.52	328.65	138.83
2023-09-20	175.49	149.83	320.77	134.59
2023-09-21	173.93	147.38	319.53	131.36
2023-09-22	174.79	146.91	317.01	131.25
2023-09-25	176.08	146.48	317.54	132.17

In [147]:
```
volume
```

Out[147]:

	AAPL	IBM	MSFT	GOOG
2023-09-18	67257573	2508062	16834208	16233590
2023-09-19	51826941	3945423	16514487	15484644
2023-09-20	58436181	9636681	21436525	21473533
2023-09-21	63149116	4944786	35560362	22058375
2023-09-22	56725385	2562216	21447887	17355284
2023-09-25	46172740	2694245	17835964	14650032

In [148]:
```
returns = price.pct_change()  #计算出每个元素与其前一个元素之间的百分比变化，并返回一个新的 DataFrame，其中包含这些百分比变化值
returns.tail()    #打印出 returns 数据的最后5行
```

Out[148]:

	AAPL	IBM	MSFT	GOOG
2023-09-19	0.006181	0.009856	-0.001246	-0.000936
2023-09-20	-0.019992	0.022591	-0.023977	-0.030541
2023-09-21	-0.008889	-0.016352	-0.003866	-0.023999
2023-09-22	0.004945	-0.003189	-0.007887	-0.000837
2023-09-25	0.007380	-0.002927	0.001672	0.007010

In [144]:
```
returns['MSFT'].corr(returns['IBM'])
                                    #corr() 方法用于计算两个序列之间的相关系数，范围从 -1 到 1。相关系数衡量了两个序列之间的线性关系
                                    #如果相关系数接近 1，表示两个序列呈正相关关系；如果相关系数接近 -1，表示两个序列呈负相关关系；
                                    #如果相关系数接近 0，则表示两个序列之间没有线性关系。
```

Out[144]: -0.8480267357878247

In [149]:
```
returns['MSFT'].cov(returns['IBM'])
                                    #cov() 方法用于计算两个序列之间的协方差，衡量了它们的联合变化程度。
                                    #协方差可以显示两个变量的变化趋势是否一致，以及变化幅度的大小。
                                    #正值表示正向相关，负值表示负向相关，而数值的绝对大小反映了变量之间的关联程度
```

Out[149]: -0.00010000266447960678

In [150]:
```
returns.corr()
```

Out[150]:

	AAPL	IBM	MSFT	GOOG
AAPL	1.000000	-0.369910	0.839848	0.972393
IBM	-0.369910	1.000000	-0.670709	-0.233583
MSFT	0.839848	-0.670709	1.000000	0.751010
GOOG	0.972393	-0.233583	0.751010	1.000000

In [151]:
```
returns.cov()
```

Out[151]:

	AAPL	IBM	MSFT	GOOG
AAPL	0.000144	-0.000066	0.000102	0.000191
IBM	-0.000066	0.000218	-0.000100	-0.000057
MSFT	0.000102	-0.000100	0.000102	0.000124
GOOG	0.000191	-0.000057	0.000124	0.000268

In [152]:
```
returns.corrwith(returns.IBM)    #corrwith可以计算行/列与另外一个表格或序列的相关性
```

Out[152]:
```
AAPL   -0.369910
IBM     1.000000
MSFT   -0.670709
GOOG   -0.233583
dtype: float64
```

In [153]:
```
returns.corrwith(volume)     #传入axis='columns'会逐行进行计算
```

Out[153]:
```
AAPL   -0.689321
IBM     0.655114
MSFT   -0.048805
GOOG   -0.960061
dtype: float64
```

In [154]:
```
data1 = pd.DataFrame({'Qu1':[1,3,4,3,4], 'Qu2':[2,3,1,2,3], 'Qu3':[1,5,2,4,4]})
data1
```

Out[154]:

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

In [156]:
```
result = data1.apply(pd.value_counts).fillna(0)    #将每个列中的唯一值进行计数,其中每列的索引是 data1 中的唯一值，而值是该值在相应列
result                                             #fillna(0) 是对结果 DataFrame 执行的操作，它将 DataFrame 中的缺失值（NaN）替换为 0
```

Out[156]:

	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

In [ ]:
```

```