Pandas Learning

参考：10 munites to pandas

引入包

首先引入如下两个Pandas和Numpy的包

In [1]:
```
import numpy as np
import pandas as pd
```

对象创建

创建series

通过传入一个值的列表创建一个序列（Series），让pandas创建一个整数索引：

In [2]:
```
s = pd.Series([1,3,5,np.nan,6,8])
s
```

Out[2]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

In [3]:
```
dates = pd.date_range('1/1/2022',periods=10)
dates
```

Out[3]:
```
DatetimeIndex(['2022-01-01', '2022-01-02', '2022-01-03', '2022-01-04',
               '2022-01-05', '2022-01-06', '2022-01-07', '2022-01-08',
               '2022-01-09', '2022-01-10'],
              dtype='datetime64[ns]', freq='D')
```

In [4]:
```
dates1 = pd.date_range('20220201',periods=10)
dates1
```

Out[4]:
```
DatetimeIndex(['2022-02-01', '2022-02-02', '2022-02-03', '2022-02-04',
               '2022-02-05', '2022-02-06', '2022-02-07', '2022-02-08',
               '2022-02-09', '2022-02-10'],
              dtype='datetime64[ns]', freq='D')
```

创建DataFrame

1. 通过传入一个Numpy数组创建一个DataFrame，使用datetime索引和标签列

In [5]:
```
df = pd.DataFrame(np.random.randn(10,4),index=dates,columns=['A','B','C','D'])
df
```

Out[5]:

	A	B	C	D
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542

	A	B	C	D
2022-01-02	-1.128372	-0.022418	0.773715	0.965569
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-04	0.996316	-1.430516	0.309446	-1.910570
2022-01-05	0.113685	0.119375	0.226008	-1.115882
2022-01-06	-1.151013	1.182598	0.833071	0.770879
2022-01-07	-0.062773	-0.374317	0.489827	0.146479
2022-01-08	1.255284	1.769874	1.303418	-0.343367
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141
2022-01-10	-0.727601	0.647371	-1.552739	1.156347

1. 通过传入一个可以转换为类似序列数据结构的字典对象创建一个DataFrame

In [6]:
```
df2 = pd.DataFrame({
    'A':1.0,
    'B':pd.Timestamp('20220201'),
    'C':pd.Series(1,index=list(range(6)),dtype='float32'), # 创建序列的一种方式
    'D': np.array([1,2,3,4,5,6],dtype='int32'),
    'E': pd.Categorical(['start','train','test','train','test','end']),
    'F':'lsy',
})
df2
```

Out[6]:

	A	B	C	D	E	F
0	1.0	2022-02-01	1.0	1	start	lsy
1	1.0	2022-02-01	1.0	2	train	lsy
2	1.0	2022-02-01	1.0	3	test	lsy
3	1.0	2022-02-01	1.0	4	train	lsy
4	1.0	2022-02-01	1.0	5	test	lsy
5	1.0	2022-02-01	1.0	6	end	lsy

In [7]:
```
df2.dtypes
```

Out[7]:
```
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

查看数据

查看前几行和后几行数据

In [8]:
```
df.head() # 查看前几行数据
```

Out[8]:

	A	B	C	D
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542
2022-01-02	-1.128372	-0.022418	0.773715	0.965569
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-04	0.996316	-1.430516	0.309446	-1.910570
2022-01-05	0.113685	0.119375	0.226008	-1.115882

In [9]:
```
df.tail(3) # 查看后几行数据,不传入参数值，则默认查看最后5行
```

Out[9]:

	A	B	C	D
2022-01-08	1.255284	1.769874	1.303418	-0.343367
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141
2022-01-10	-0.727601	0.647371	-1.552739	1.156347

查看索引（行）和列

In [10]:
```
df.index
```

Out[10]:
```
DatetimeIndex(['2022-01-01', '2022-01-02', '2022-01-03', '2022-01-04',
               '2022-01-05', '2022-01-06', '2022-01-07', '2022-01-08',
               '2022-01-09', '2022-01-10'],
              dtype='datetime64[ns]', freq='D')
```

In [11]:
```
df.columns
```

Out[11]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

DataFrame.to_numpy()

DataFrame.to_numpy()可以将数据转化为Numpy的数据表示形式。

> 注意： **当DataFrame中列之间数据类型不一样时，将DataFrame转换为Numpy的数据表示形式，会消耗比较多的性能**； 如果是DataFrame中每列都是相同类型，并且Numpy中支持，调用DataFrame.to_numpy()进行转换是比较快的并且不需要复制数据；

DataFrame和Numpy的典型区别： Numpy Array整个Array都是一种数据类型，而pandas的DataFrame可以每列一种数据类型；

当你调用DataFrame.to_numpy(), pandas将会查找一种可以兼容（容纳）DataFrame中所有数据类型的数据类型。这个数据最后可能是object类型，这个需要将每个值强转为Python object；

> 注意：DataFrame.to_numpy()输出是没有索引标签（index）和列标签(column)的

In [12]:
```
df.to_numpy()
```

Out[12]:
```
array([[-0.54927941, -1.03575309,  0.62150311, -1.18554168],
       [-1.1283715 , -0.02241765,  0.77371515,  0.96556903],
       [ 0.60347695,  0.56333619,  0.86259828, -1.44339315],
       [ 0.99631628, -1.43051588,  0.30944589, -1.91056965],
```

```
       [ 0.11368479,  0.11937531,  0.226008  , -1.11588236],
       [-1.15101308,  1.18259762,  0.83307135,  0.77087881],
       [-0.06277274, -0.37431695,  0.48982709,  0.14647896],
       [ 1.25528367,  1.76987423,  1.30341769, -0.34336673],
       [ 1.17508577, -0.45200321, -2.89156329, -0.86514132],
       [-0.72760075,  0.64737104, -1.55273912,  1.15634741]])
```

In [13]:
```
df2.to_numpy()
```

Out[13]:
```
array([[1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 1, 'start', 'lsy'],
       [1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 2, 'train', 'lsy'],
       [1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 3, 'test', 'lsy'],
       [1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 4, 'train', 'lsy'],
       [1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 5, 'test', 'lsy'],
       [1.0, Timestamp('2022-02-01 00:00:00'), 1.0, 6, 'end', 'lsy']],
      dtype=object)
```

查看数据的快速统计摘要

In [14]:
```
df.describe()
```

Out[14]:

	A	B	C	D
count	10.000000	10.000000	10.000000	10.000000
mean	0.052481	0.096755	0.097528	-0.382462
std	0.926875	0.980484	1.298960	1.088992
min	-1.151013	-1.430516	-2.891563	-1.910570
25%	-0.683020	-0.432582	0.246867	-1.168127
50%	0.025456	0.048479	0.555665	-0.604254
75%	0.898106	0.626362	0.818232	0.614779
max	1.255284	1.769874	1.303418	1.156347

矩阵转置

In [15]:
```
df.T
```

Out[15]:

	2022-01-01	2022-01-02	2022-01-03	2022-01-04	2022-01-05	2022-01-06	2022-01-07	2022-01-08	2022-01-09
A	-0.549279	-1.128372	0.603477	0.996316	0.113685	-1.151013	-0.062773	1.255284	1.175086
B	-1.035753	-0.022418	0.563336	-1.430516	0.119375	1.182598	-0.374317	1.769874	-0.452003
C	0.621503	0.773715	0.862598	0.309446	0.226008	0.833071	0.489827	1.303418	-2.891563
D	-1.185542	0.965569	-1.443393	-1.910570	-1.115882	0.770879	0.146479	-0.343367	-0.865141

按轴排序（X轴、Y轴...）

In [16]:
```
df.sort_index(axis=1,ascending=False) # 按横轴降序排列
```

Out[16]:

	D	C	B	A
2022-01-01	-1.185542	0.621503	-1.035753	-0.549279

	D	C	B	A
2022-01-02	0.965569	0.773715	-0.022418	-1.128372
2022-01-03	-1.443393	0.862598	0.563336	0.603477
2022-01-04	-1.910570	0.309446	-1.430516	0.996316
2022-01-05	-1.115882	0.226008	0.119375	0.113685
2022-01-06	0.770879	0.833071	1.182598	-1.151013
2022-01-07	0.146479	0.489827	-0.374317	-0.062773
2022-01-08	-0.343367	1.303418	1.769874	1.255284
2022-01-09	-0.865141	-2.891563	-0.452003	1.175086
2022-01-10	1.156347	-1.552739	0.647371	-0.727601

按值排序

In [17]:
```
df.sort_values(by='B',ascending=False) # 传入具体排序的列标签名或者行标签名，默认升序
```

Out[17]:

	A	B	C	D
2022-01-08	1.255284	1.769874	1.303418	-0.343367
2022-01-06	-1.151013	1.182598	0.833071	0.770879
2022-01-10	-0.727601	0.647371	-1.552739	1.156347
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-05	0.113685	0.119375	0.226008	-1.115882
2022-01-02	-1.128372	-0.022418	0.773715	0.965569
2022-01-07	-0.062773	-0.374317	0.489827	0.146479
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542
2022-01-04	0.996316	-1.430516	0.309446	-1.910570

查询值

注意： 尽管标准的Python/Numpy表达式在查询值和设置值是非常直观，并且可以在交互窗口派上用场；但是在生产代码，**我们推荐使用经过优化的pandas数据访问方式：.at, iat, .loc 和 .iloc。**

查询数据值

查询某一列的值，产生一个Series，相当于df.A

In [18]:
```
df['A']
```

Out[18]:
```
2022-01-01   -0.549279
2022-01-02   -1.128372
2022-01-03    0.603477
2022-01-04    0.996316
2022-01-05    0.113685
```

```
2022-01-06   -1.151013
2022-01-07   -0.062773
2022-01-08    1.255284
2022-01-09    1.175086
2022-01-10   -0.727601
Freq: D, Name: A, dtype: float64
```

In [19]:
```
df[0:3] # 查询索引范围内的行记录
```

Out[19]:

	A	B	C	D
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542
2022-01-02	-1.128372	-0.022418	0.773715	0.965569
2022-01-03	0.603477	0.563336	0.862598	-1.443393

In [20]:
```
df['20220101':'20220105'] # 通过索引标签值范围内的行记录
```

Out[20]:

	A	B	C	D
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542
2022-01-02	-1.128372	-0.022418	0.773715	0.965569
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-04	0.996316	-1.430516	0.309446	-1.910570
2022-01-05	0.113685	0.119375	0.226008	-1.115882

In [21]:
```
# 传入列标签范围，为啥不能查询出查询列标签范围内的列值？ 传入范围值只能用来查行记录
# df['A':'C']  报错
```

In [22]:
```
df[['A','B']] # 传入多个列标签名，可以查询出多列数据
```

Out[22]:

	A	B
2022-01-01	-0.549279	-1.035753
2022-01-02	-1.128372	-0.022418
2022-01-03	0.603477	0.563336
2022-01-04	0.996316	-1.430516
2022-01-05	0.113685	0.119375
2022-01-06	-1.151013	1.182598
2022-01-07	-0.062773	-0.374317
2022-01-08	1.255284	1.769874
2022-01-09	1.175086	-0.452003
2022-01-10	-0.727601	0.647371

In [23]:
```
# 这样传入多个行标签，为啥不可以查询出多行？ 传入标签值的列表只能用于查列值？
# df[['2022-01-03','2022-01-06']]   # 报错
```

In [24]:
```
df.loc[['2022-01-03','2022-01-06']]  # 取行索引为 2022-01-03 和 2022-01-06的两行数据
```

Out[24]:

	A	B	C	D
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-06	-1.151013	1.182598	0.833071	0.770879

通过标签名查询（.at .loc）

In [25]:
```
df.loc[dates[0]] # 通过行标签值查询某一行的值
```

Out[25]:
```
A   -0.549279
B   -1.035753
C    0.621503
D   -1.185542
Name: 2022-01-01 00:00:00, dtype: float64
```

In [26]:
```
df.loc[:,'B'] # 查询列标签值为‘B’的列值
```

Out[26]:
```
2022-01-01   -1.035753
2022-01-02   -0.022418
2022-01-03    0.563336
2022-01-04   -1.430516
2022-01-05    0.119375
2022-01-06    1.182598
2022-01-07   -0.374317
2022-01-08    1.769874
2022-01-09   -0.452003
2022-01-10    0.647371
Freq: D, Name: B, dtype: float64
```

In [27]:
```
df.loc[:,['B','C']] # 查询'B'列和'C'列的列值
```

Out[27]:

	B	C
2022-01-01	-1.035753	0.621503
2022-01-02	-0.022418	0.773715
2022-01-03	0.563336	0.862598
2022-01-04	-1.430516	0.309446
2022-01-05	0.119375	0.226008
2022-01-06	1.182598	0.833071
2022-01-07	-0.374317	0.489827
2022-01-08	1.769874	1.303418
2022-01-09	-0.452003	-2.891563
2022-01-10	0.647371	-1.552739

In [28]:
```
df.loc['20220103',['A','B']] # 查询‘20220103’行的‘A’列和‘B’值
```

Out[28]:
```
A    0.603477
B    0.563336
Name: 2022-01-03 00:00:00, dtype: float64
```

In [29]:
```
df.loc[dates[0],'B'] # 获取某个单元格的值
```

Out[29]:
```
-1.0357530888963922
```

In [30]:
```
df.at[dates[0],'B'] # 获取某个单元格的值，和上面的loc效果一样
```

Out[30]:
```
-1.0357530888963922
```

In [31]:
```
# df.at[dates[0],['A','B']] # 会报错；  .at只能查询单个单元格的值，不能查询多个单元格的值
```

通过定位查询（.iat .iloc）

In [32]:
```
df.iloc[3] # 查询第3行记录（起始行为0行）
```

Out[32]:
```
A    0.996316
B   -1.430516
C    0.309446
D   -1.910570
Name: 2022-01-04 00:00:00, dtype: float64
```

1. 通过整数范围切片，实现和Numpy或者Python类似 例如查询第3行到第5行之间的第0列到第2列之间的数据； 注意：这里的范围区间属于**左闭右开**

In [33]:
```
df.iloc[3:5,0:2] #通过整数切片，实现和Numpy或者Python类似的效果，查询第3行到第5行之间的第
```

Out[33]:

	A	B
2022-01-04	0.996316	-1.430516
2022-01-05	0.113685	0.119375

1. 通过传入整数类型位置的集合查询值，实现和Numpy或Python类似

In [34]:
```
df.iloc[[1,5,6],[0,2,3]]
```

Out[34]:

	A	C	D
2022-01-02	-1.128372	0.773715	0.965569
2022-01-06	-1.151013	0.833071	0.770879
2022-01-07	-0.062773	0.489827	0.146479

In [35]:
```
df.iloc[:,1:3]
```

Out[35]:

	B	C
2022-01-01	-1.035753	0.621503
2022-01-02	-0.022418	0.773715
2022-01-03	0.563336	0.862598
2022-01-04	-1.430516	0.309446
2022-01-05	0.119375	0.226008
2022-01-06	1.182598	0.833071
2022-01-07	-0.374317	0.489827
2022-01-08	1.769874	1.303418
2022-01-09	-0.452003	-2.891563

	B	C
2022-01-10	0.647371	-1.552739

In [36]:
```
df.iloc[1,1] # 通过单元格位置获取单元格值
```

Out[36]:
```
-0.022417648160720893
```

In [37]:
```
df.iat[1,1] # 实现效果和.iloc一致
```

Out[37]:
```
-0.022417648160720893
```

.at 和 .loc的区别：

- .at只能通过标签名获取一个单元格的值，不能获取某个标签名范围内（包含多个单元格）的值；
- .loc既可以获取一个单元格的值，也能获取某个范围内多个单元格的值；

.iat 和 .iloc的区别：

- .iat 只能通过位置索引获取一个单元格的值，不能获取某个位置索引范围内（包含多个单元格）的值；
- .iloc 既可以通过位置索引获取一个单元格的值，不能获取某个位置索引范围内（包含多个单元格）的值；

布尔索引（Boolean indexing）

使用某个列的数据值来过滤数据记录

In [38]:
```
df[df['A']>0] #查询'A'列值大于0的记录
```

Out[38]:

	A	B	C	D
2022-01-03	0.603477	0.563336	0.862598	-1.443393
2022-01-04	0.996316	-1.430516	0.309446	-1.910570
2022-01-05	0.113685	0.119375	0.226008	-1.115882
2022-01-08	1.255284	1.769874	1.303418	-0.343367
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141

In [39]:
```
df[df>0] # 查询所有列值大于0的记录，列值小于0则显示为NaN
```

Out[39]:

	A	B	C	D
2022-01-01	NaN	NaN	0.621503	NaN
2022-01-02	NaN	NaN	0.773715	0.965569
2022-01-03	0.603477	0.563336	0.862598	NaN
2022-01-04	0.996316	NaN	0.309446	NaN
2022-01-05	0.113685	0.119375	0.226008	NaN
2022-01-06	NaN	1.182598	0.833071	0.770879

	A	B	C	D
2022-01-07	NaN	NaN	0.489827	0.146479
2022-01-08	1.255284	1.769874	1.303418	NaN
2022-01-09	1.175086	NaN	NaN	NaN
2022-01-10	NaN	0.647371	NaN	1.156347

In [40]:
```
df2 = df.copy()
df2['E'] = ['one','one','two','three','four','three','five','six','seven','ten']
df2
```

Out[40]:

	A	B	C	D	E
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542	one
2022-01-02	-1.128372	-0.022418	0.773715	0.965569	one
2022-01-03	0.603477	0.563336	0.862598	-1.443393	two
2022-01-04	0.996316	-1.430516	0.309446	-1.910570	three
2022-01-05	0.113685	0.119375	0.226008	-1.115882	four
2022-01-06	-1.151013	1.182598	0.833071	0.770879	three
2022-01-07	-0.062773	-0.374317	0.489827	0.146479	five
2022-01-08	1.255284	1.769874	1.303418	-0.343367	six
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141	seven
2022-01-10	-0.727601	0.647371	-1.552739	1.156347	ten

In [41]:
```
df2[df2['E'].isin(['one','ten'])] # 通过列值是否存在于某个列表中过滤数据
```

Out[41]:

	A	B	C	D	E
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542	one
2022-01-02	-1.128372	-0.022418	0.773715	0.965569	one
2022-01-10	-0.727601	0.647371	-1.552739	1.156347	ten

设置值

设置新列时会自动按索引对齐数据

In [42]:
```
s1 = pd.Series([1,2,3,4,5,6,7,8,9,10],index=pd.date_range('20220101',periods=10))
s1
```

Out[42]:
```
2022-01-01     1
2022-01-02     2
2022-01-03     3
2022-01-04     4
2022-01-05     5
2022-01-06     6
2022-01-07     7
2022-01-08     8
2022-01-09     9
2022-01-10    10
Freq: D, dtype: int64
```

In [43]:
```
df['F'] = s1 # 会默认按照索引标签（行标签）进行匹配
df
```

Out[43]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542	1
2022-01-02	-1.128372	-0.022418	0.773715	0.965569	2
2022-01-03	0.603477	0.563336	0.862598	-1.443393	3
2022-01-04	0.996316	-1.430516	0.309446	-1.910570	4
2022-01-05	0.113685	0.119375	0.226008	-1.115882	5
2022-01-06	-1.151013	1.182598	0.833071	0.770879	6
2022-01-07	-0.062773	-0.374317	0.489827	0.146479	7
2022-01-08	1.255284	1.769874	1.303418	-0.343367	8
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141	9
2022-01-10	-0.727601	0.647371	-1.552739	1.156347	10

In [ ]:
```
```

通过标签设置值

In [44]:
```
df.at[dates[1],'B'] = 0.999
df
```

Out[44]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542	1
2022-01-02	-1.128372	0.999000	0.773715	0.965569	2
2022-01-03	0.603477	0.563336	0.862598	-1.443393	3
2022-01-04	0.996316	-1.430516	0.309446	-1.910570	4
2022-01-05	0.113685	0.119375	0.226008	-1.115882	5
2022-01-06	-1.151013	1.182598	0.833071	0.770879	6
2022-01-07	-0.062773	-0.374317	0.489827	0.146479	7
2022-01-08	1.255284	1.769874	1.303418	-0.343367	8
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141	9
2022-01-10	-0.727601	0.647371	-1.552739	1.156347	10

In [45]:
```
df.iat[1,1] = 1.999999
df
```

Out[45]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	-1.185542	1
2022-01-02	-1.128372	1.999999	0.773715	0.965569	2
2022-01-03	0.603477	0.563336	0.862598	-1.443393	3
2022-01-04	0.996316	-1.430516	0.309446	-1.910570	4

	A	B	C	D	F
2022-01-05	0.113685	0.119375	0.226008	-1.115882	5
2022-01-06	-1.151013	1.182598	0.833071	0.770879	6
2022-01-07	-0.062773	-0.374317	0.489827	0.146479	7
2022-01-08	1.255284	1.769874	1.303418	-0.343367	8
2022-01-09	1.175086	-0.452003	-2.891563	-0.865141	9
2022-01-10	-0.727601	0.647371	-1.552739	1.156347	10

In [46]:
```
df.loc[:,'D'] = np.array([5]*len(df))
df
```

Out[46]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	5	1
2022-01-02	-1.128372	1.999999	0.773715	5	2
2022-01-03	0.603477	0.563336	0.862598	5	3
2022-01-04	0.996316	-1.430516	0.309446	5	4
2022-01-05	0.113685	0.119375	0.226008	5	5
2022-01-06	-1.151013	1.182598	0.833071	5	6
2022-01-07	-0.062773	-0.374317	0.489827	5	7
2022-01-08	1.255284	1.769874	1.303418	5	8
2022-01-09	1.175086	-0.452003	-2.891563	5	9
2022-01-10	-0.727601	0.647371	-1.552739	5	10

In [47]:
```
df2 = df.copy()
df2[df2>0] = -df2
df2
```

Out[47]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	-0.621503	-5	-1
2022-01-02	-1.128372	-1.999999	-0.773715	-5	-2
2022-01-03	-0.603477	-0.563336	-0.862598	-5	-3
2022-01-04	-0.996316	-1.430516	-0.309446	-5	-4
2022-01-05	-0.113685	-0.119375	-0.226008	-5	-5
2022-01-06	-1.151013	-1.182598	-0.833071	-5	-6
2022-01-07	-0.062773	-0.374317	-0.489827	-5	-7
2022-01-08	-1.255284	-1.769874	-1.303418	-5	-8
2022-01-09	-1.175086	-0.452003	-2.891563	-5	-9
2022-01-10	-0.727601	-0.647371	-1.552739	-5	-10

缺失值处理

pandas使用np.nan代替缺失值，缺失值默认是不包含在计算过程中。 .reindex方法可以用于**修改、添加和删除指定轴上的索引**，这个将返回数据的副本：

In [48]:
```
df1 = df.reindex(index=dates[0:4],columns=list(df.columns)+['E'])
df1.loc[dates[0]:dates[2],'E'] = 2
df1
```

Out[48]:

	A	B	C	D	F	E
2022-01-01	-0.549279	-1.035753	0.621503	5	1	2.0
2022-01-02	-1.128372	1.999999	0.773715	5	2	2.0
2022-01-03	0.603477	0.563336	0.862598	5	3	2.0
2022-01-04	0.996316	-1.430516	0.309446	5	4	NaN

In [49]:
```
list(df.columns)
```

Out[49]:
```
['A', 'B', 'C', 'D', 'F']
```

In [50]:
```
df1.dropna(how='any') # 删除含有缺失值的数据记录
```

Out[50]:

	A	B	C	D	F	E
2022-01-01	-0.549279	-1.035753	0.621503	5	1	2.0
2022-01-02	-1.128372	1.999999	0.773715	5	2	2.0
2022-01-03	0.603477	0.563336	0.862598	5	3	2.0

In [51]:
```
df1.fillna(value=5) # 填充缺失值
```

Out[51]:

	A	B	C	D	F	E
2022-01-01	-0.549279	-1.035753	0.621503	5	1	2.0
2022-01-02	-1.128372	1.999999	0.773715	5	2	2.0
2022-01-03	0.603477	0.563336	0.862598	5	3	2.0
2022-01-04	0.996316	-1.430516	0.309446	5	4	5.0

In [52]:
```
df1.fillna(df1.mean()) #使用列的平均值填充
```

Out[52]:

	A	B	C	D	F	E
2022-01-01	-0.549279	-1.035753	0.621503	5	1	2.0
2022-01-02	-1.128372	1.999999	0.773715	5	2	2.0
2022-01-03	0.603477	0.563336	0.862598	5	3	2.0
2022-01-04	0.996316	-1.430516	0.309446	5	4	2.0

In [53]:
```
pd.isna(df1) #显示是否是NaN值
```

Out[53]:

	A	B	C	D	F	E
2022-01-01	False	False	False	False	False	False

	A	B	C	D	F	E
2022-01-02	False	False	False	False	False	False
2022-01-03	False	False	False	False	False	False
2022-01-04	False	False	False	False	False	True

操作（Operations）

- axis=0表示纵轴,类似axis='index'
- axis=1表示横轴,类似axis='columns'

统计

默认排除了缺失值（NaN）

In [54]:
```
df.mean() # 默认按列分组求值（Y轴 纵轴）
```

Out[54]:
```
A    0.052481
B    0.298996
C    0.097528
D    5.000000
F    5.500000
dtype: float64
```

In [55]:
```
df.mean(axis=1) # 按行求平均值（X轴 横轴）, 类似df.mean(axis='columns')
```

Out[55]:
```
2022-01-01    1.007294
2022-01-02    1.729069
2022-01-03    2.005882
2022-01-04    1.775049
2022-01-05    2.091814
2022-01-06    2.372931
2022-01-07    2.410547
2022-01-08    3.465715
2022-01-09    2.366304
2022-01-10    2.673406
Freq: D, dtype: float64
```

In [56]:
```
s = pd.Series([1,3,5,np.nan,6,8,9,10,13,2],index=dates).shift(2) #移动两位
s
```

Out[56]:
```
2022-01-01     NaN
2022-01-02     NaN
2022-01-03     1.0
2022-01-04     3.0
2022-01-05     5.0
2022-01-06     NaN
2022-01-07     6.0
2022-01-08     8.0
2022-01-09     9.0
2022-01-10    10.0
Freq: D, dtype: float64
```

In [57]:
```
df
```

Out[57]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	5	1

	A	B	C	D	F
2022-01-02	-1.128372	1.999999	0.773715	5	2
2022-01-03	0.603477	0.563336	0.862598	5	3
2022-01-04	0.996316	-1.430516	0.309446	5	4
2022-01-05	0.113685	0.119375	0.226008	5	5
2022-01-06	-1.151013	1.182598	0.833071	5	6
2022-01-07	-0.062773	-0.374317	0.489827	5	7
2022-01-08	1.255284	1.769874	1.303418	5	8
2022-01-09	1.175086	-0.452003	-2.891563	5	9
2022-01-10	-0.727601	0.647371	-1.552739	5	10

In [58]:
```
df.sub(s,axis=0) # 每列减去s列上对应的值，类似axis='index'
```

Out[58]:

	A	B	C	D	F
2022-01-01	NaN	NaN	NaN	NaN	NaN
2022-01-02	NaN	NaN	NaN	NaN	NaN
2022-01-03	-0.396523	-0.436664	-0.137402	4.0	2.0
2022-01-04	-2.003684	-4.430516	-2.690554	2.0	1.0
2022-01-05	-4.886315	-4.880625	-4.773992	0.0	0.0
2022-01-06	NaN	NaN	NaN	NaN	NaN
2022-01-07	-6.062773	-6.374317	-5.510173	-1.0	1.0
2022-01-08	-6.744716	-6.230126	-6.696582	-3.0	0.0
2022-01-09	-7.824914	-9.452003	-11.891563	-4.0	0.0
2022-01-10	-10.727601	-9.352629	-11.552739	-5.0	0.0

In [59]:
```
df
```

Out[59]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	5	1
2022-01-02	-1.128372	1.999999	0.773715	5	2
2022-01-03	0.603477	0.563336	0.862598	5	3
2022-01-04	0.996316	-1.430516	0.309446	5	4
2022-01-05	0.113685	0.119375	0.226008	5	5
2022-01-06	-1.151013	1.182598	0.833071	5	6
2022-01-07	-0.062773	-0.374317	0.489827	5	7
2022-01-08	1.255284	1.769874	1.303418	5	8
2022-01-09	1.175086	-0.452003	-2.891563	5	9
2022-01-10	-0.727601	0.647371	-1.552739	5	10

In [60]:
```
df.apply(np.cumsum)  # 列值累加
```

Out[60]:

	A	B	C	D	F
2022-01-01	-0.549279	-1.035753	0.621503	5	1
2022-01-02	-1.677651	0.964246	1.395218	10	3
2022-01-03	-1.074174	1.527582	2.257817	15	6
2022-01-04	-0.077858	0.097066	2.567262	20	10
2022-01-05	0.035827	0.216442	2.793270	25	15
2022-01-06	-1.115186	1.399039	3.626342	30	21
2022-01-07	-1.177959	1.024722	4.116169	35	28
2022-01-08	0.077325	2.794596	5.419587	40	36
2022-01-09	1.252411	2.342593	2.528023	45	45
2022-01-10	0.524810	2.989964	0.975284	50	55

In [61]:
```
df.apply(lambda x:x.max()-x.min()) # Lambda表达式
```

Out[61]:
```
A    2.406297
B    3.430515
C    4.194981
D    0.000000
F    9.000000
dtype: float64
```

直方图

直方图和离散化

In [62]:
```
s = pd.Series(np.random.randint(0,7,size=10))
s
```

Out[62]:
```
0    0
1    1
2    1
3    0
4    3
5    5
6    0
7    5
8    4
9    1
dtype: int32
```

In [63]:
```
s.value_counts() # 统计出现的个数
```

Out[63]:
```
0    3
1    3
5    2
3    1
4    1
dtype: int64
```

字符串方法

Series的str属性有一系列的字符串处理方法，可以非常方便地处理数组里的每一个元素； 注意： str中的pattern-matching通常默认使用正则表达式。

In [64]:
```
s = pd.Series(['A','B','C','Aaba','Baca',np.nan,'CABA','dog','cat'])
s.str.upper()
```

Out[64]:
```
0       A
1       B
2       C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object
```

合并

Concat

pandas提供了多种工具，可以在join/merge类型操作时，轻松将Series和DataFrame对象与索引和各种关系代数功能的各种逻辑组合在一起;

In [65]:
```
df = pd.DataFrame(np.random.randn(10,4))
df
```

Out[65]:

	0	1	2	3
0	0.336003	-0.408323	1.263011	-0.303722
1	0.241124	0.664029	1.840833	0.052391
2	-0.832435	-0.841934	0.598293	1.180171
3	-1.339389	0.542614	-0.485700	-1.584911
4	0.427082	-0.197518	1.309208	0.849893
5	0.213777	-0.536168	-0.690227	-2.351220
6	-0.172593	-0.205613	0.808136	-0.072774
7	0.994801	0.516240	-1.833451	-1.009345
8	1.309698	-0.495452	-0.050730	-0.212149
9	0.531532	0.208832	1.772995	-0.079708

In [66]:
```
pieces = [df[:3],df[3:7],df[7:]]
pd.concat(pieces) # 合并多个DataFrame
```

Out[66]:

	0	1	2	3
0	0.336003	-0.408323	1.263011	-0.303722
1	0.241124	0.664029	1.840833	0.052391
2	-0.832435	-0.841934	0.598293	1.180171
3	-1.339389	0.542614	-0.485700	-1.584911
4	0.427082	-0.197518	1.309208	0.849893

	0	1	2	3
5	0.213777	-0.536168	-0.690227	-2.351220
6	-0.172593	-0.205613	0.808136	-0.072774
7	0.994801	0.516240	-1.833451	-1.009345
8	1.309698	-0.495452	-0.050730	-0.212149
9	0.531532	0.208832	1.772995	-0.079708

注意： 向DataFrame中添加一列是相对比较快的。但是，添加一行是需要一个副本，比较损耗性能。 建议采用**传入一个预先创建好的记录集合到DataFrame构造器的方式**创建DataFrame，而不是**通过将记录迭代地添加到DataFrame中的方式**创建DataFrame；

Join

类似于数据库的Join连接

In [67]:
```
left = pd.DataFrame({'key':['foo','foo'],'lval':[1,2]})
right = pd.DataFrame({'key':['foo','foo'],'rval':[4,5]})
left
```

Out[67]:

	key	lval
0	foo	1
1	foo	2

In [68]:
```
right
```

Out[68]:

	key	rval
0	foo	4
1	foo	5

In [69]:
```
pd.merge(left,right,on='key') # 将left DataFrame和right DataFrame按照字段‘key’进行关联
```

Out[69]:

	key	lval	rval
0	foo	1	4
1	foo	1	5
2	foo	2	4
3	foo	2	5

In [70]:
```
left = pd.DataFrame({'key':['foo','bar'],'lval':[1,2]})
right = pd.DataFrame({'key':['foo','bar'],'lval':[4,5]})
left
```

Out[70]:

	key	lval
0	foo	1
1	bar	2

In [71]:
```
right
```

Out[71]:

	key	lval
0	foo	4
1	bar	5

In [72]:
```
pd.merge(left,right,on='key')
```

Out[72]:

	key	lval_x	lval_y
0	foo	1	4
1	bar	2	5

分组（Group）

group by是指一个包含一步或多步如下操作的过程：

- 按某些标准将数据分成若干分组；
- 在每个分组中单独使用函数；
- 合并一些结果到一个数据结构中；

In [73]:
```
df = pd.DataFrame({
     "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
     "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
     "C": np.random.randn(8),
     "D": np.random.randn(8),
})
df
```

Out[73]:

	A	B	C	D
0	foo	one	0.846625	-0.402053
1	bar	one	-1.164032	0.723914
2	foo	two	0.440119	-0.809135
3	bar	three	-0.835419	-0.905771
4	foo	two	-0.187825	0.995173
5	bar	two	-0.470598	1.331585
6	foo	one	-1.642231	1.049514
7	foo	three	-0.761584	-1.293312

In [74]:
```
df.groupby('A').sum() # 按字段‘A’进行分组求和
```

Out[74]:

A	C	D
bar	-2.470050	1.149728
foo	-1.304895	-0.459813

In [75]:
```
df.groupby(['A','B']).sum() # 按字段‘A’和‘B’进行分组求和
```

Out[75]:

A	B	C	D
bar	one	-1.164032	0.723914
	three	-0.835419	-0.905771
	two	-0.470598	1.331585
foo	one	-0.795605	0.647461
	three	-0.761584	-1.293312
	two	0.252294	0.186038

重组*（Reshape）

In [76]:
```
tuples = list(
    zip(
        *[
            ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
            ["one", "two", "one", "two", "one", "two", "one", "two"],
        ]
    )
)

index = pd.MultiIndex.from_tuples(tuples,names=['first','second'])
df = pd.DataFrame(np.random.randn(8,2),index=index,columns=['A','B'])
df2 = df[:6]
df2
```

Out[76]:

first	second	A	B
bar	one	1.267890	1.335531
	two	-0.238612	0.403715
baz	one	-0.234953	-1.375464
	two	1.578720	0.262375
foo	one	0.421873	-0.568166
	two	-0.230847	0.602096

zip()函数用于将可迭代的对象作为参数，将对象对应的元素打包成一个个元组，然后返回由这些元组组成的列表；

- 如果各个迭代器的元素个数不一致，则**返回列表长度与最短的对象相同**；

- 利用*号操作符，可以将元组解压为列表；

> zip方法在Python2和Python3中的不同： 在Python2.x中，zip()返回的是列表；在Python3.x中为了减少内存，zip()返回的是一个对象；如需展示列表，需要手动list()转换。

zip语法： zip([iterable,...]) 参数说明：

- iterable 表示一个或多个迭代器

In [77]:
```
a = [1,2,3]
b = [4,5,6]
c = [7,8,9,10]
zipped = zip(a,b) # python3开始返回的是一个对象
zipped
```

Out[77]:
```
<zip at 0x27b764f13c0>
```

In [78]:
```
list(zipped) # list() 转换为列表
```

Out[78]:
```
[(1, 4), (2, 5), (3, 6)]
```

In [79]:
```
list(zip(a,c)) # 返回元素个数与最短的列表一致
```

Out[79]:
```
[(1, 7), (2, 8), (3, 9)]
```

In [80]:
```
a1,a2 = zip(*zip(a,b)) # 与zip相反，zip(*)可理解为解压，返回二维矩阵式
list(a1)
```

Out[80]:
```
[1, 2, 3]
```

In [81]:
```
list(a2)
```

Out[81]:
```
[4, 5, 6]
```

stack() 方法在DataFrame列中压缩了一层

In [82]:
```
stacked = df2.stack()
stacked
```

Out[82]:
```
first  second
bar    one     A    1.267890
               B    1.335531
       two     A   -0.238612
               B    0.403715
baz    one     A   -0.234953
               B   -1.375464
       two     A    1.578720
               B    0.262375
foo    one     A    0.421873
               B   -0.568166
       two     A   -0.230847
               B    0.602096
dtype: float64
```

对于压缩的DataFrame或Series（使用MultiIndex作为索引），stack的逆向操作是unstack，默认情况下会解压最后一层：

In [83]:
```
stacked.unstack()
```

Out[83]:

		A	B
first	**second**		
bar	**one**	1.267890	1.335531
	two	-0.238612	0.403715
baz	**one**	-0.234953	-1.375464
	two	1.578720	0.262375
foo	**one**	0.421873	-0.568166
	two	-0.230847	0.602096

In [84]:
```
stacked.unstack(1) # 选择解压的方向，横轴方向
```

Out[84]:

	second	one	two
first			
bar	**A**	1.267890	-0.238612
	B	1.335531	0.403715
baz	**A**	-0.234953	1.578720
	B	-1.375464	0.262375
foo	**A**	0.421873	-0.230847
	B	-0.568166	0.602096

In [85]:
```
stacked.unstack(0) # 选择解压的方向，纵轴方向（默认）
```

Out[85]:

	first	bar	baz	foo
second				
one	**A**	1.267890	-0.234953	0.421873
	B	1.335531	-1.375464	-0.568166
two	**A**	-0.238612	1.578720	-0.230847
	B	0.403715	0.262375	0.602096

透视表（Pivot tables）

In [86]:
```
df = pd.DataFrame({
    'A':['one','one','two','three']*3,
    'B':['A','B','C']*4,
    "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 2,
    'D':np.random.randn(12),
    'E':np.random.randn(12)
})
df
```

Out[86]:

	A	B	C	D	E
0	one	A	foo	-0.238015	1.197977
1	one	B	foo	-1.349828	0.598853

	A	B	C	D	E
2	two	C	foo	-1.539470	-1.513162
3	three	A	bar	-1.079824	0.481829
4	one	B	bar	-1.112132	-0.029112
5	one	C	bar	-0.027423	1.320127
6	two	A	foo	1.037352	1.182141
7	three	B	foo	-0.760191	0.618513
8	one	C	foo	-1.626682	1.995568
9	one	A	bar	0.403204	0.645425
10	two	B	bar	0.118735	1.927009
11	three	C	bar	-0.501484	-1.543219

In [87]:
```
pd.pivot_table(df,values='D',index=['A','B'],columns=['C']) # 生成数据透视表
```

Out[87]:

	C	bar	foo
A	**B**		
one	**A**	0.403204	-0.238015
	B	-1.112132	-1.349828
	C	-0.027423	-1.626682
three	**A**	-1.079824	NaN
	B	NaN	-0.760191
	C	-0.501484	NaN
two	**A**	NaN	1.037352
	B	0.118735	NaN
	C	NaN	-1.539470

时间序列（Time Series）

pandas可以在频率转换期间执行重采样操作（例如，将秒级数据转换为5分钟级的数据）。这在金融应用程序等中极为常见。

In [88]:
```
rng = pd.date_range('20220201',periods=10,freq='S') #表示频率为秒(或者间隔为秒)
rng
```

Out[88]:
```
DatetimeIndex(['2022-02-01 00:00:00', '2022-02-01 00:00:01',
               '2022-02-01 00:00:02', '2022-02-01 00:00:03',
               '2022-02-01 00:00:04', '2022-02-01 00:00:05',
               '2022-02-01 00:00:06', '2022-02-01 00:00:07',
               '2022-02-01 00:00:08', '2022-02-01 00:00:09'],
              dtype='datetime64[ns]', freq='S')
```

Alias Description B business day frequency C custom business day frequency D calendar day frequency W weekly frequency M month end frequency SM semi-month end frequency (15th and end of month) BM business month end frequency CBM custom business month end

frequency MS month start frequency SMS semi-month start frequency (1st and 15th) BMS business month start frequency CBMS custom business month start frequency Q quarter end frequency BQ business quarter end frequency QS quarter start frequency BQS business quarter start frequency A, Y year end frequency BA, BY business year end frequency AS, YS year start frequency BAS, BYS business year start frequency BH business hour frequency H hourly frequency T, min minutely frequency S secondly frequency L, ms milliseconds U, us microseconds N nanoseconds

In [89]:
```
ts = pd.Series(np.random.randint(0,500,len(rng)),index=rng) # 随机产生和rng个数相同的0
```

In [90]:
```
ts
```

Out[90]:
```
2022-02-01 00:00:00    129
2022-02-01 00:00:01    417
2022-02-01 00:00:02    424
2022-02-01 00:00:03     93
2022-02-01 00:00:04    468
2022-02-01 00:00:05    401
2022-02-01 00:00:06    202
2022-02-01 00:00:07    194
2022-02-01 00:00:08    124
2022-02-01 00:00:09    487
Freq: S, dtype: int32
```

In [91]:
```
ts.resample('5Min').sum() # 每5分钟采样一次
```

Out[91]:
```
2022-02-01    2939
Freq: 5T, dtype: int32
```

In [92]:
```
ts.resample('5S').sum() # 每5秒钟采样一次
```

Out[92]:
```
2022-02-01 00:00:00    1531
2022-02-01 00:00:05    1408
Freq: 5S, dtype: int32
```

时区表示

In [93]:
```
rng = pd.date_range('3/1/2022',periods=10,freq='D')
rng
```

Out[93]:
```
DatetimeIndex(['2022-03-01', '2022-03-02', '2022-03-03', '2022-03-04',
               '2022-03-05', '2022-03-06', '2022-03-07', '2022-03-08',
               '2022-03-09', '2022-03-10'],
              dtype='datetime64[ns]', freq='D')
```

In [94]:
```
ts = pd.Series(np.random.randn(len(rng)),rng)
ts
```

Out[94]:
```
2022-03-01   -1.343030
2022-03-02    1.921904
2022-03-03    0.014999
2022-03-04    1.200884
2022-03-05    0.442425
2022-03-06    0.756571
2022-03-07   -1.131561
2022-03-08   -0.359986
2022-03-09    2.133910
2022-03-10   -1.008117
Freq: D, dtype: float64
```

In [95]:
```
ts_utc = ts.tz_localize('UTC') # 设置时区
ts_utc
```

Out[95]:
```
2022-03-01 00:00:00+00:00   -1.343030
2022-03-02 00:00:00+00:00    1.921904
2022-03-03 00:00:00+00:00    0.014999
2022-03-04 00:00:00+00:00    1.200884
2022-03-05 00:00:00+00:00    0.442425
2022-03-06 00:00:00+00:00    0.756571
2022-03-07 00:00:00+00:00   -1.131561
2022-03-08 00:00:00+00:00   -0.359986
2022-03-09 00:00:00+00:00    2.133910
2022-03-10 00:00:00+00:00   -1.008117
Freq: D, dtype: float64
```

In [96]:
```
ts_utc.tz_convert('Asia/Shanghai') #转换时区，转换为上海时区
```

Out[96]:
```
2022-03-01 08:00:00+08:00   -1.343030
2022-03-02 08:00:00+08:00    1.921904
2022-03-03 08:00:00+08:00    0.014999
2022-03-04 08:00:00+08:00    1.200884
2022-03-05 08:00:00+08:00    0.442425
2022-03-06 08:00:00+08:00    0.756571
2022-03-07 08:00:00+08:00   -1.131561
2022-03-08 08:00:00+08:00   -0.359986
2022-03-09 08:00:00+08:00    2.133910
2022-03-10 08:00:00+08:00   -1.008117
Freq: D, dtype: float64
```

在时间跨度中相互转换

In [97]:
```
rng = pd.date_range('5/1/2022',periods=10,freq='M')
rng
```

Out[97]:
```
DatetimeIndex(['2022-05-31', '2022-06-30', '2022-07-31', '2022-08-31',
               '2022-09-30', '2022-10-31', '2022-11-30', '2022-12-31',
               '2023-01-31', '2023-02-28'],
              dtype='datetime64[ns]', freq='M')
```

In [98]:
```
ts = pd.Series(np.random.randn(len(rng)),index=rng)
ts
```

Out[98]:
```
2022-05-31    0.503510
2022-06-30   -0.679366
2022-07-31    0.613947
2022-08-31    1.630310
2022-09-30    0.633477
2022-10-31   -0.400410
2022-11-30   -0.454036
2022-12-31    0.603057
2023-01-31    0.505252
2023-02-28    2.347496
Freq: M, dtype: float64
```

In [99]:
```
ps = ts.to_period()
ps
```

Out[99]:
```
2022-05    0.503510
2022-06   -0.679366
2022-07    0.613947
2022-08    1.630310
2022-09    0.633477
2022-10   -0.400410
2022-11   -0.454036
2022-12    0.603057
2023-01    0.505252
2023-02    2.347496
Freq: M, dtype: float64
```

In [100…
```
ps.to_timestamp()
```

Out[100…
```
2022-05-01    0.503510
2022-06-01   -0.679366
2022-07-01    0.613947
2022-08-01    1.630310
2022-09-01    0.633477
2022-10-01   -0.400410
2022-11-01   -0.454036
2022-12-01    0.603057
2023-01-01    0.505252
2023-02-01    2.347496
Freq: MS, dtype: float64
```

周期和时间戳之间的相互转换可以使用一些方便的算术函数； 在下面的的案例中，我们将一个在年末在11月的季度周期转换为季度末次月末的上午九点 In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end

In [101…
```
prng = pd.period_range('2019Q1','2020Q4',freq='Q-NOV')
ts = pd.Series(np.random.randn(len(prng)),prng)
ts.index = (prng.asfreq('M','e')+1).asfreq('H','s')+9
ts.head()
```

Out[101…
```
2019-03-01 09:00   -1.343457
2019-06-01 09:00    0.151337
2019-09-01 09:00   -1.412863
2019-12-01 09:00   -0.298861
2020-03-01 09:00    0.822182
Freq: H, dtype: float64
```

分类（Categoricals）

pandas可以在DataFrame中存储分类数据; 主要作用:为数据取见名知意的别名

In [102…
```
df = pd.DataFrame({
    "id":[1,2,3,4,5,6],
    "raw_grade":["a","b","b","a","a","e"]
})
```

In [103…
```
df['grade'] = df['raw_grade'].astype("category") #将'raw_grade'这个字段作为分类类型
df['grade']
```

Out[103…
```
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

通过重命名的方式给分类取更加形象的名称，一般通过Series.cat.categories()的方式就可以实现

In [104…
```
df['grade'].cat.categories = ['very good','good','very bad']
```

对类别进行重新排序，并且同时添加缺少的类别（Series.cat()下的方法默认返回一个新Series）

?: **缺少的类别和原本的数据又是如何进行绑定的?**

In [105…
```
df['grade'] = df['grade'].cat.set_categories([
    'very bad','bad','medium','good','very good'
```

```
])
df['grade']  # 如何知道a绑定到very good，而e绑定到very bad?
```

Out[105…
```
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

In [106…
```
df.sort_values(by='grade') # 按类别中的顺序排序，而不是词汇顺序
```

Out[106…

	id	raw_grade	grade
5	6	e	very bad
1	2	b	good
2	3	b	good
0	1	a	very good
3	4	a	very good
4	5	a	very good

In [107…
```
df.groupby('grade').size() # 按类别字段分组，空的类别也会显示
```

Out[107…
```
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

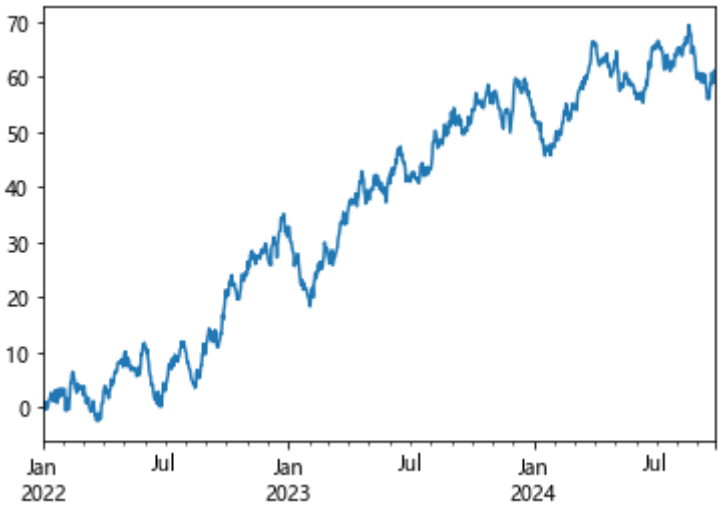
画图（Plotting）

一般通过matplotlib API画图

In [108…
```
import matplotlib.pyplot as plt
plt.close('all')
```

close()方法用于关闭图形窗口

In [109…
```
ts = pd.Series(np.random.randn(1000),index=pd.date_range('20220101',periods=1000))
ts = ts.cumsum()
ts.plot();
```



如果在Jupyter Notebook执行，画图将出现在plot()上。否则，使用**matplotlib.pyplot.show**显示画图，或者使用**matplotlib.pyplot.savefig**将画图保存到文件里。

In [110…
```
df = pd.DataFrame(
    np.random.randn(1000,4),index=ts.index,columns=['A','B','C','D']
)
```

In [111…
```
df = df.cumsum()
plt.figure()
df.plot()
plt.legend(loc='best') #设置图例的位置
```

Out[111…
```
<matplotlib.legend.Legend at 0x27b7768d790>
<Figure size 432x288 with 0 Axes>
```

loc='best'从其他9个位置中挑选一个和图重合最少的位置放置图例，当画图的数据很大时，画图速度会很慢；

Loction String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4

Loction String	Location Code
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

输入或输出数据

In [112…
```
df.to_csv("foo.csv") # 输出数据到CSV文件中
```

In [113…
```
pd.read_csv("foo.csv") #读取CSV文件数据
```

Out[113…

	Unnamed: 0	A	B	C	D
0	2022-01-01	-1.039265	0.848499	-0.672730	-0.576270
1	2022-01-02	0.216005	0.823903	-0.304602	-1.502944
2	2022-01-03	0.418521	0.405340	-0.369427	-1.444457
3	2022-01-04	0.330419	0.903617	0.822847	-1.507633
4	2022-01-05	-1.198590	1.048297	1.716102	-0.728068
...	...	...	...	...	...
995	2024-09-22	29.614938	82.030909	7.255583	24.149662
996	2024-09-23	30.931259	80.941661	6.365912	24.248240
997	2024-09-24	31.074078	82.781696	5.822364	23.667818
998	2024-09-25	31.384752	83.278335	5.785206	24.314173
999	2024-09-26	32.658524	82.922822	6.566700	24.464831

1000 rows × 5 columns

In [114…
```
df.to_hdf("foo.hf","df") # 输出数据到HDF5存储中
```

In [115…
```
pd.read_hdf("foo.hf","df") # 读取HDF5数据
```

Out[115…

	A	B	C	D
2022-01-01	-1.039265	0.848499	-0.672730	-0.576270
2022-01-02	0.216005	0.823903	-0.304602	-1.502944
2022-01-03	0.418521	0.405340	-0.369427	-1.444457
2022-01-04	0.330419	0.903617	0.822847	-1.507633
2022-01-05	-1.198590	1.048297	1.716102	-0.728068
...	...	...	...	...
2024-09-22	29.614938	82.030909	7.255583	24.149662

	A	B	C	D
2024-09-23	30.931259	80.941661	6.365912	24.248240
2024-09-24	31.074078	82.781696	5.822364	23.667818
2024-09-25	31.384752	83.278335	5.785206	24.314173
2024-09-26	32.658524	82.922822	6.566700	24.464831

1000 rows × 4 columns

In [116…
```
df.to_excel("foo.xlsx",sheet_name="Hello_Sheet") # 输出数据到Excel文件中
```

In [117…
```
pd.read_excel("foo.xlsx",sheet_name="Hello_Sheet") # 读取Excel文件数据
```

Out[117…

	Unnamed: 0	A	B	C	D
0	2022-01-01	-1.039265	0.848499	-0.672730	-0.576270
1	2022-01-02	0.216005	0.823903	-0.304602	-1.502944
2	2022-01-03	0.418521	0.405340	-0.369427	-1.444457
3	2022-01-04	0.330419	0.903617	0.822847	-1.507633
4	2022-01-05	-1.198590	1.048297	1.716102	-0.728068
...	...	...	...	...	...
995	2024-09-22	29.614938	82.030909	7.255583	24.149662
996	2024-09-23	30.931259	80.941661	6.365912	24.248240
997	2024-09-24	31.074078	82.781696	5.822364	23.667818
998	2024-09-25	31.384752	83.278335	5.785206	24.314173
999	2024-09-26	32.658524	82.922822	6.566700	24.464831

1000 rows × 5 columns

一些陷阱（Gotchas）

如果你试图执行一个操作，你可能会遇到如下异常：

In [118…
```
# if pd.Series([True,False,True]):
#     print("I was true")
```

可以查阅比较（Comparisons）或者陷阱（Gotchas），从中获得解释和处理的方法。

Really 10 minutes ???

group by

In [119…
```
df = pd.DataFrame(
    {
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
```

```
    }
)
df
```

Out[119…

	A	B	C	D
0	foo	one	-0.064059	0.362133
1	bar	one	-0.406412	-1.662427
2	foo	two	0.387281	-0.107811
3	bar	three	1.963167	-1.953017
4	foo	two	-1.544160	0.112308
5	bar	two	1.779932	-0.575129
6	foo	one	-0.692168	-0.925385
7	foo	three	-1.355639	1.886781

In [120…
```
group_single = df.groupby("A")
group_single
```

Out[120… `<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000027B7947F580>`

In [121…
```
group_multi = df.groupby(["A","B"])
group_multi
```

Out[121… `<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000027B7954AB50>`

设置按除指定列外的列进行分组

In [122…
```
df2 = df.set_index(["A", "B"])
grouped = df2.groupby(level=df2.index.names.difference(["B"]))
grouped.sum()
```

Out[122…

	C	D
A		
bar	3.336687	-4.190574
foo	-3.268745	1.328026

In [123…
```
def get_letter_type(letter):
    if letter.lower() in 'aeiou':
        return 'vowel'
    else:
        return 'consonant'
grouped = df.groupby(get_letter_type, axis=1)
grouped.head()
```

Out[123…

	A	B	C	D
0	foo	one	-0.064059	0.362133
1	bar	one	-0.406412	-1.662427
2	foo	two	0.387281	-0.107811
3	bar	three	1.963167	-1.953017

	A	B	C	D
4	foo	two	-1.544160	0.112308
5	bar	two	1.779932	-0.575129
6	foo	one	-0.692168	-0.925385
7	foo	three	-1.355639	1.886781

In [124…
```
grouped.groups
```

Out[124…
```
{'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

In [125…
```
lst = [1, 2, 3, 1, 2, 3, 1, 2, 3]
s = pd.Series([1, 2, 3, 10, 20, 30, 100, 200, 300],lst) # lst表示行索引（index）
grouped = s.groupby(level=0) # 对行索引进行分组

grouped.first() # 获取每个分组的第一条记录信息
```

Out[125…
```
1    1
2    2
3    3
dtype: int64
```

In [126…
```
grouped.first(2) # first里面加数值，并不生效，依然是返回每个分组的第一条记录信息
```

Out[126…
```
1    1
2    2
3    3
dtype: int64
```

In [127…
```
grouped.last() # 获取每个分组的最后一条记录信息
```

Out[127…
```
1    100
2    200
3    300
dtype: int64
```

In [128…
```
grouped.sum() # 针对行索引进行分组并求和
```

Out[128…
```
1    111
2    222
3    333
dtype: int64
```

In [129…
```
grouped.head(2) # 获取每个分组的前两条记录
```

Out[129…
```
1     1
2     2
3     3
1    10
2    20
3    30
dtype: int64
```

In [130…
```
df.groupby("A").groups # 返回一个字典，key为该分组的标签值，value为该分组的索引值
```

Out[130…
```
{'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}
```

问题：这三者有啥区别 grouped = df.groupby(['A']) grouped_C = grouped['C']

df['C'].groupby(df['A'])

grouped = df.groupby(['A']) grouped['C']

In [131…
```
df = pd.DataFrame(
    {
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
    }
)
df
```

Out[131…

	A	B	C	D
0	foo	one	0.400563	-1.023294
1	bar	one	0.720834	-0.291269
2	foo	two	0.911537	-0.531749
3	bar	three	0.088858	-1.017411
4	foo	two	0.086402	1.006197
5	bar	two	-0.732197	1.135159
6	foo	one	-0.725385	-1.155821
7	foo	three	0.142884	-1.775584

In [132…
```
grouped = df.groupby(['A'])
grouped_C = grouped['C']  # 和df['C'].groupby(df['A'])效果一样
grouped_C.head()
```

Out[132…
```
0    0.400563
1    0.720834
2    0.911537
3    0.088858
4    0.086402
5   -0.732197
6   -0.725385
7    0.142884
Name: C, dtype: float64
```

In [133…
```
df['C'].groupby(df['A']).head()
```

Out[133…
```
0    0.400563
1    0.720834
2    0.911537
3    0.088858
4    0.086402
5   -0.732197
6   -0.725385
7    0.142884
Name: C, dtype: float64
```

In [134…
```
df['C']
```

Out[134…
```
0    0.400563
1    0.720834
2    0.911537
3    0.088858
4    0.086402
5   -0.732197
6   -0.725385
```

```
7    0.142884
Name: C, dtype: float64
```

In [135…
```
grouped = df.groupby(['A'])
grouped['C'].head()
```

Out[135…
```
0    0.400563
1    0.720834
2    0.911537
3    0.088858
4    0.086402
5   -0.732197
6   -0.725385
7    0.142884
Name: C, dtype: float64
```

In [136…
```
grouped.get_group('foo') #获取分组后某个分组值的数据
```

Out[136…

	A	B	C	D
0	foo	one	0.400563	-1.023294
2	foo	two	0.911537	-0.531749
4	foo	two	0.086402	1.006197
6	foo	one	-0.725385	-1.155821
7	foo	three	0.142884	-1.775584

In [137…
```
df.groupby(['A','B']).get_group(('foo','one')) # 按A列和B列进行分组，然后取A列值为‘foo
```

Out[137…

	A	B	C	D
0	foo	one	0.400563	-1.023294
6	foo	one	-0.725385	-1.155821

In [138…
```
df
```

Out[138…

	A	B	C	D
0	foo	one	0.400563	-1.023294
1	bar	one	0.720834	-0.291269
2	foo	two	0.911537	-0.531749
3	bar	three	0.088858	-1.017411
4	foo	two	0.086402	1.006197
5	bar	two	-0.732197	1.135159
6	foo	one	-0.725385	-1.155821
7	foo	three	0.142884	-1.775584

In [139…
```
df.groupby(['A','B']).agg(np.sum) # 默认返回是多级索引（MultiIndex）的情况，也可以通过as
```

Out[139…

		C	D
A	**B**		
bar	**one**	0.720834	-0.291269

		C	D
A	**B**		
	three	0.088858	-1.017411
	two	-0.732197	1.135159
foo	**one**	-0.324822	-2.179115
	three	0.142884	-1.775584
	two	0.997939	0.474448

In [140…
```
df.groupby(['A','B'],as_index=False).agg(np.sum) # 可以通过as_index=False来改变，使返
```

Out[140…

	A	B	C	D
0	bar	one	0.720834	-0.291269
1	bar	three	0.088858	-1.017411
2	bar	two	-0.732197	1.135159
3	foo	one	-0.324822	-2.179115
4	foo	three	0.142884	-1.775584
5	foo	two	0.997939	0.474448

In [141…
```
df.groupby(['A','B']).sum().reset_index() # 通过reset_index也可以实现和as_index=False
```

Out[141…

	A	B	C	D
0	bar	one	0.720834	-0.291269
1	bar	three	0.088858	-1.017411
2	bar	two	-0.732197	1.135159
3	foo	one	-0.324822	-2.179115
4	foo	three	0.142884	-1.775584
5	foo	two	0.997939	0.474448

In [142…
```
grouped.describe() # 分组后的简单数据分析
```

Out[142…

									C	
	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**	**count**	**mean**
A										
bar	3.0	0.025832	0.728563	-0.732197	-0.321669	0.088858	0.404846	0.720834	3.0	-0.05784
foo	5.0	0.163200	0.594189	-0.725385	0.086402	0.142884	0.400563	0.911537	5.0	-0.69605

In [143…
```
ll = [['foo', 1], ['foo', 2], ['foo', 2], ['bar', 1], ['bar', 1]]
df4 = pd.DataFrame(ll, columns=["A", "B"])
df4
```

Out[143...

	A	B
0	foo	1
1	foo	2
2	foo	2
3	bar	1
4	bar	1

In [144...
```
df4.groupby('A').sum()
```

Out[144...

	B
A	
bar	2
foo	5

In [145...
```
df4.groupby('A').size() # 统计每个分组的记录数（包含NaN值）
```

Out[145...
```
A
bar    2
foo    3
dtype: int64
```

In [146...
```
df4.groupby('A').nunique() # 统计每个分组唯一值的数量，类似value_counts函数，只是nunique
```

Out[146...

	B
A	
bar	1
foo	2

In [147...
```
df4.groupby('A')['B'].nunique() # 统计每个分组唯一值的数量，类似value_counts函数，只是nu
```

Out[147...
```
A
bar    1
foo    2
Name: B, dtype: int64
```

Note:

- Aggregation functions will not return the groups that you are aggregating over if they are named columns, when as_index=True, the default. The grouped columns will be the indices of the returned object.
- Passing as_index=False will return the groups that you are aggregating over, if they are named columns.

In [148...
```
df4.groupby('A').size()
```

Out[148...
```
A
bar    2
foo    3
dtype: int64
```

In [149…
```
df4.groupby('A').count()
```

Out[149…

A	B
bar	2
foo	3

In [150…
```
ll = [['foo', ], ['foo', 2], ['foo', 3], ['bar', 1], ['bar', 2]]
df5 = pd.DataFrame(ll, columns=["A", "B"])
df5
```

Out[150…

	A	B
0	foo	NaN
1	foo	2.0
2	foo	3.0
3	bar	1.0
4	bar	2.0

In [151…
```
df5.groupby('A').size()
```

Out[151…
```
A
bar    2
foo    3
dtype: int64
```

In [152…
```
df5.groupby('A').count()
```

Out[152…

A	B
bar	2
foo	2

In [153…
```
df5.groupby('A').agg(np.size)
```

Out[153…

A	B
bar	2
foo	3

In [154…
```
df5.groupby('A').agg('size') # 使用agg函数计算分组的记录数（包含）
```

Out[154…
```
A
bar    2
foo    3
dtype: int64
```

In [155…
```
df5.groupby('A').agg('count')
```

Out[155…

A	B
bar	2
foo	2

In [156…
```
df5.groupby('A').agg('mean')
```

Out[156…

A	B
bar	1.5
foo	2.5

In [157…
```
df5.groupby('A').agg('sum')
```

Out[157…

A	B
bar	3.0
foo	5.0

In [158…
```
df
```

Out[158…

	A	B	C	D
0	foo	one	0.400563	-1.023294
1	bar	one	0.720834	-0.291269
2	foo	two	0.911537	-0.531749
3	bar	three	0.088858	-1.017411
4	foo	two	0.086402	1.006197
5	bar	two	-0.732197	1.135159
6	foo	one	-0.725385	-1.155821
7	foo	three	0.142884	-1.775584

In [159…
```
df.groupby(['A','B']).agg('size') # 返回的是Series类型
```

Out[159…
```
A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

In [160…
```
df.groupby(['A','B']).agg({'size'}) # 返回的是DataFrame类型
```

Out[160...

		C	D
		size	size
A	B		
bar	one	1	1
	three	1	1
	two	1	1
foo	one	2	2
	three	1	1
	two	2	2

In [161...
```
df.groupby(['A','B']).agg({'size'}).columns # 多级索引
```

Out[161...
```
MultiIndex([('C', 'size'),
            ('D', 'size')],
           )
```

In [162...
```
type(df.groupby(['A','B']))
```

Out[162...
```
pandas.core.groupby.generic.DataFrameGroupBy
```

In [163...
```
type(df.groupby(['A','B'])['C'])
```

Out[163...
```
pandas.core.groupby.generic.SeriesGroupBy
```

当对Series进行分组时，agg函数可以传入一个函数列表或者函数字典进行聚合，返回一个DataFrame

In [164...
```
df.groupby('A')['C'].agg([np.sum,np.mean,np.std])
```

Out[164...

	sum	mean	std
A			
bar	0.077496	0.025832	0.728563
foo	0.816001	0.163200	0.594189

In [165...
```
type(df.groupby(['A'])['C'].agg([np.sum,np.mean,np.std]))
```

Out[165...
```
pandas.core.frame.DataFrame
```

当对DataFrame进行分组时，agg函数可以传入用于每一列的函数列表进行聚合，这会产生具有分层索引的聚合效果

In [166...
```
df.groupby('A')['C','D'].agg([np.sum,np.mean,np.std])
```

```
<ipython-input-166-52c26b5adc30>:1: FutureWarning: Indexing with multiple keys (impl
icitly converted to a tuple of keys) will be deprecated, use a list instead.
  df.groupby('A')['C','D'].agg([np.sum,np.mean,np.std])
```

Out[166...

	C			D		
	sum	mean	std	sum	mean	std
A						
bar	0.077496	0.025832	0.728563	-0.173521	-0.05784	1.095106
foo	0.816001	0.163200	0.594189	-3.480251	-0.69605	1.049823

In [167...
```
type(df.groupby('A')['C','D'].agg([np.sum,np.mean,np.std]))
```

```
<ipython-input-167-0649c89ba0d9>:1: FutureWarning: Indexing with multiple keys (impl
icitly converted to a tuple of keys) will be deprecated, use a list instead.
  type(df.groupby('A')['C','D'].agg([np.sum,np.mean,np.std]))
```

Out[167...
```
pandas.core.frame.DataFrame
```

多级索引的Group By

创建多级索引的序列

In [192...
```
arrays = [
    ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
    ["lakers", "rockets", "76ers", "bulls", "heats", "worriors", "lakers", "heats"],
    ["one", "two", "one", "two", "one", "two", "one", "two"]
]
```

In [193...
```
index = pd.MultiIndex.from_arrays(arrays,names=['first','second','third'])
```

In [194...
```
s = pd.Series(np.random.randn(8),index)
```

In [195...
```
s
```

Out[195...
```
first  second    third
bar    lakers    one      0.192513
       rockets   two      0.080053
baz    76ers     one      2.057298
       bulls     two     -1.125576
foo    heats     one     -0.140655
       worriors  two     -0.663413
qux    lakers    one     -0.433718
       heats     two      1.441950
dtype: float64
```

In [196...
```
type(s)
```

Out[196...
```
pandas.core.series.Series
```

group 多级索引中的某一级索引

In [197...
```
grouped_first = s.groupby(level=0) # 通过索引级别值指定，比如一级索引的level=0
```

In [198...
```
grouped_first.sum()
```

Out[198...
```
first
bar    0.272566
baz    0.931721
foo   -0.804068
qux    1.008232
dtype: float64
```

In [199…
```
grouped_second = s.groupby(level=1) # 通过索引级别值指定，比如二级索引的level=1
```

In [200…
```
grouped_second.sum()
```

Out[200…
```
second
76ers       2.057298
bulls      -1.125576
heats       1.301295
lakers     -0.241205
rockets     0.080053
worriors   -0.663413
dtype: float64
```

In [201…
```
grouped_second_by_name = s.groupby(level='second') # 也可以通过索引名称指定
grouped_second_by_name.sum()
```

Out[201…
```
second
76ers       2.057298
bulls      -1.125576
heats       1.301295
lakers     -0.241205
rockets     0.080053
worriors   -0.663413
dtype: float64
```

In [203…
```
grouped_multi_ind = s.groupby(level=['second','third'])  # 按多级索引进行分组
grouped_multi_ind.sum()
```

Out[203…
```
second    third
76ers     one      2.057298
bulls     two     -1.125576
heats     one     -0.140655
          two      1.441950
lakers    one     -0.241205
rockets   two      0.080053
worriors  two     -0.663413
dtype: float64
```

按索引级别和列对DataFrame进行分组

In [210…
```
arrays = [
     ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
     ["one", "two", "one", "two", "one", "two", "one", "two"],
]
```

In [219…
```
index = pd.MultiIndex.from_arrays(arrays,names=['first','second'])
df = pd.DataFrame({'A':['A','B','C','D','A','B','C','D'],'B':np.random.randn(8)},ind
df
```

Out[219…

		A	B
first	**second**		
bar	**one**	A	-0.701906
	two	B	0.198589
baz	**one**	C	-0.672610
	two	D	1.046324
foo	**one**	A	0.809309
	two	B	-0.317249

		A	B
first	**second**		
qux	**one**	C	-0.830964
	two	D	-0.523737

按索引级别和列名进行分组

In [222…
```
df.groupby([pd.Grouper(level=1),'A']).sum() # 按索引级别和列名进行分组
```

Out[222…

		B
second	**A**	
one	**A**	0.107403
	C	-1.503575
two	**B**	-0.118660
	D	0.522587

In [224…
```
df.groupby([pd.Grouper(level='second'),'A']).sum() # 在Grouper中也可以使用索引名称
```

Out[224…

		B
second	**A**	
one	**A**	0.107403
	C	-1.503575
two	**B**	-0.118660
	D	0.522587

In [ ]:
```
df.groupby(['second','A']).sum() # 更简单点，也可以直接把索引名称列表传给其进行分组
```

pandas.Grouper

pandas.Grouper 是专门用来生成分组依据的工具，可以按列、按索引、按计算结果、时序中的频率等内容为依据进行分组。 参考： pandas.Grouper

In [ ]:
```
# 轴方向
# df.groupby(Grouper(level='date', freq='60s', axis=1))
# 多个列:
# df.groupby([pd.Grouper(freq='1M', key='Date'), 'Buyer']).sum()
# df.groupby([pd.Grouper('dt', freq='D'),
#             pd.Grouper('other_column')
#           ])
# 时序周期
# 按时间周期分组，需要使用时间字段，如果不是日期时间类型需要进行类型转换:
#
# df['column_name'] = pd.to_datetime(df['column_name'])
# df.groupby(pd.Grouper(key='column_name', freq="M")).mean()
# 可以自定义时间周期:
#
# # 10 年一个周期
# df.groupby(pd.cut(df.date,
#                   pd.date_range('1970', '2020', freq='10YS'),
```

```
#                   right=False)
#           ).mean()
```