创建数据集

1、引入pandas

```
import numpy as np
import pandas as pd
```

2、创建数据集

```
# 生成创建一个6*4的正数数据集
data2 = pd.DataFrame(np.random.rand(6,4),columns=list('ABCD'))
data2
```

	A	B	C	D
0	0.268960	0.944233	0.564392	0.241277
1	0.739152	0.786348	0.108528	0.062371
2	0.777080	0.272329	0.986666	0.559934
3	0.234701	0.300360	0.667984	0.946984
4	0.807806	0.979275	0.793605	0.172620
5	0.753253	0.046280	0.403574	0.417993

```
# 先创建一个时间索引
dates = pd.date_range('20170101',periods=6)

# 再创建一个6*4的数据
df = pd.DataFrame(np.random.randn(6,4),index=dates,columns=list('ABCD'))
```

	A	B	C	D
2017-01-01	-0.188280	-0.512786	-1.507800	1.020410
2017-01-02	-1.699451	0.580669	1.000826	0.719016
2017-01-03	0.412229	-1.902585	-1.260546	1.153787
2017-01-04	0.679351	0.633701	0.500856	-0.771894
2017-01-05	-0.383847	0.676279	0.548778	-0.117328
2017-01-06	0.036585	0.429563	-1.769048	-0.351982

3、使用字典来创建数据

```
df2 = pd.DataFrame({'A':np.random.randn(3)})
print df2
```

	A
0	0.249433
1	1.048593
2	-0.170596

4、另一种字典创建数据的方法

```
# 另一种用字典创建数据的方法
df3 = pd.DataFrame({'A':pd.Timestamp('20170101'),'B':np.random.randn(3)})
print df3
```

	A	B
0	2017-01-01	-0.364829
1	2017-01-01	-0.487836
2	2017-01-01	1.686728

查看数据

1、使用dtypes来查看各行的数据格式

```
# 使用dtypes来查看各行的数据格式
df3.dtypes
```

```
A    datetime64[ns]
B           float64
dtype: object
```

2、查看数据框中所有的数据

```
# 查看数据框中所有的数据
df3
```

	A	B
0	2017-01-01	-0.545882
1	2017-01-01	-0.033721
2	2017-01-01	-0.388657

3、使用head查看前几行数据（默认是前5行）

```
# 使用head查看指定的前几行数据（不指定时默认是前5行）
df.head(3)
```

	A	B	C	D
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327

4、使用tail查看后5行数据

```
# 使用tail查看后指定的后几行数据（默认是5行）
df.tail(3)
```

	A	B	C	D
2017-01-04	0.198526	0.286079	1.907239	1.415250
2017-01-05	0.976051	-0.874763	0.236659	0.452393
2017-01-06	0.987277	0.104625	-2.007644	1.041692

5、查看数据框的索引

```
# 查看数据框的索引
df.index
```

```
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04',
               '2017-01-05', '2017-01-06'],
              dtype='datetime64[ns]', freq='D')
```

6、用columns查看列名

```
# 使用columns查看列名
df.columns
```

```
Index([u'A', u'B', u'C', u'D'], dtype='object')
```

7、用values查看数据值

```
# 用values查看数据值
df.values
```

```
array([[ 0.68806638,  0.81934586,  0.34296473,  1.73047519],
       [ 0.0332296 ,  0.5998538 , -1.47720515, -0.237258  ],
       [ 1.99659048, -1.91628982,  0.42948988,  1.00732749],
       [ 0.19852554,  0.28607882,  1.90723864,  1.41525027],
       [ 0.97605102, -0.87476293,  0.23665913,  0.45239314],
       [ 0.98727715,  0.10462485, -2.00764438,  1.04169165]])
```

8、用describe查看描述性统计

```
# 用describe查看描述性统计
df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.813290	-0.163525	-0.094750	0.901647
std	0.701295	1.039315	1.425101	0.704338
min	0.033230	-1.916290	-2.007644	-0.237258
25%	0.320911	-0.629916	-1.048739	0.591127
50%	0.832059	0.195352	0.289812	1.024510
75%	0.984471	0.521410	0.407859	1.321861
max	1.996590	0.819346	1.907239	1.730475

9、用T转置，即行列转换

```
# 用T转置，即行列转换
df.T
```

	2017-01-01 00:00:00	2017-01-02 00:00:00	2017-01-03 00:00:00	2017-01-04 00:00:00	2017-01-05 00:00:00	2017-01-06 00:00:00
A	0.688066	0.033230	1.996590	0.198526	0.976051	0.987277
B	0.819346	0.599854	-1.916290	0.286079	-0.874763	0.104625
C	0.342965	-1.477205	0.429490	1.907239	0.236659	-2.007644
D	1.730475	-0.237258	1.007327	1.415250	0.452393	1.041692

10、用sort_values对数据进行排序

```
# 用到sort_values对数据进行排序
df.sort_values(by='C')
```

	A	B	C	D
2017-01-06	0.987277	0.104625	-2.007644	1.041692
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-05	0.976051	-0.874763	0.236659	0.452393
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-03	1.996590	-1.916290	0.429490	1.007327
2017-01-04	0.198526	0.286079	1.907239	1.415250

数据选择

1、选择A列的数据进行操作

```
# 选择A列的数据进行操作
df['A']
```

```
2017-01-01    0.688066
2017-01-02    0.033230
2017-01-03    1.996590
2017-01-04    0.198526
2017-01-05    0.976051
2017-01-06    0.987277
```

2、使用数组的切片操作得到行数据

```
# 使用数组的切片操作得到行数据
df[1:3]
```

	A	B	C	D
2017-01-02	0.03323	0.599854	-1.477205	-0.237258
2017-01-03	1.99659	-1.916290	0.429490	1.007327

3、使用行标签来指定输出的行(列不限定，行限定)

```
# 使用行标签来指定输出的行(列不限定，行限定)
df['20170102':'20170104']
```

	A	B	C	D
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327
2017-01-04	0.198526	0.286079	1.907239	1.415250

4、使用loc方法选择多列数据（列不限定,行限定）

```
# 使用loc方法选择多列数据（列不限定,行限定）
df.loc[dates[0]:dates[2],:]
```

	A	B	C	D
2017-01-01	0.688066	0.819346	0.342965	1.730475
2017-01-02	0.033230	0.599854	-1.477205	-0.237258
2017-01-03	1.996590	-1.916290	0.429490	1.007327

5、使用loc方法选择多列数据（行不限定,列限定）

```
# 使用loc方法选择多列数据（行不限定,列限定）
df.loc[:,'B':'D']
```

	B	C	D
2017-01-01	0.819346	0.342965	1.730475
2017-01-02	0.599854	-1.477205	-0.237258
2017-01-03	-1.916290	0.429490	1.007327
2017-01-04	0.286079	1.907239	1.415250
2017-01-05	-0.874763	0.236659	0.452393
2017-01-06	0.104625	-2.007644	1.041692

6、使用loc方法选择多列数据（行列都限定，且列只要指定的某些）

```
# 使用loc方法选择多列数据（行列都限定，且列只要指定的某些）
df.loc[dates[0]:dates[2],['B','D']]
```

	B	D
2017-01-01	0.819346	1.730475
2017-01-02	0.599854	-0.237258
2017-01-03	-1.916290	1.007327

7、使用loc方法选择某行数据(如第一行)

```
# 使用loc方法选择某行数据(如第一行)
df.loc[dates[0]]
```

```
A    0.688066
B    0.819346
C    0.342965
D    1.730475
```

8、使用loc方法只选择某一个数据，可以指定行和列

```
# 使用loc方法只选择某一个数据，可以指定行和列
df.loc[dates[0],'B']
```

```
0.81934585546715122
```

9、at方法专门用于获取某个值

```
# at方法专门用于获取某个值
df.at[dates[0],'B']
```

```
0.81934585546715122
```

10、使用iloc方法提取第四行数据

```
# 使用iloc方法提取第四行数据，得到的返回值是一个series 数据
df.iloc[3]
```

```
A    1.658872
B   -0.250281
C   -0.531258
D   -0.091432
```

11、返回4-5行，2-3列数据

```
# 返回4-5行，2-3列数据
df.iloc[3:5,1:3]
```

	B	C
2017-01-04	-0.250281	-0.531258
2017-01-05	0.123437	-1.455018

12、提取不连续行和列的数,如2,4,6行的B列和D列

```
# 提取不连续行和列的数,如2,4,6行的B列和D列
df.iloc[[1,3,5],[1,3]]
```

	B	D
2017-01-02	-0.449626	-2.183514
2017-01-04	-0.250281	-0.091432
2017-01-06	-0.319806	0.200287

13、提取某一行或某几行数据，保证所有列都在

```
# 提取某一行或某几行数据，保证所有列都在(提取所有行的某些列类似)
df.iloc[1:3,:]
```

	A	B	C	D
2017-01-02	-0.813571	-0.449626	-0.548597	-2.183514
2017-01-03	-1.704362	0.218734	-1.052347	0.369618

14、提取某个值,如第二行第二列

```
# 提取某个值,如第二行第二列
df.iloc[1,1]
```

```
-0.44962649890028467
```

15、iat是专门提取某个数的方法，它的效率高更高

```
# iat是专门提取某个数的方法，它的效率高更高
df.iat[1,1]
```

```
-0.44962649890028467
```

读取和保存

读取CSV文件

```
# read_csv方法读取csv文件，参数是文件的路径
df4 = pd.read_csv('C:/Users/Administrator/Desktop/Rong360_credit_forecasting/train/user_info.csv',encoding='gbk')
# 显示前3行数据
df4.head(3)
```

	ID	gender	job	edu	marriage	family_type
0	2583	2	2	2	1	1
1	34764	1	2	3	3	1
2	9554	1	2	4	2	2

读取Excel文件

```
# read_excel()方法读取Excel文件为DataFrame
excel_content = pd.read_excel(r'C:/Users/Administrator/Desktop/user_info.xlsx',encoding='gbk')
excel_content
```

	性别	edu	age
0	1	1	16
1	2	2	27
2	5	1	34

读取.dat文件

```
# read_table()函数读取.dat文件
content = pd.read_table(r'C:/Users/Administrator/Desktop/user_info.dat',encoding='gbk')
content
```

我爱Python！

保存为CSV文件

```
# DataFrame可以使用to_csv方法方便地导出到csv文件中
excel_content.to_csv(r'C:/Users/Administrator/Desktop/user_info.csv', encoding='utf-8', index=False)
```

筛选数据

1、筛选D列数据中大于0的行

```
# 筛选D列数据中大于0的行
df[df.D>0]
```

	A	B	C	D
2017-01-03	-1.704362	0.218734	-1.052347	0.369618
2017-01-05	0.687470	0.123437	-1.455018	0.240838
2017-01-06	-0.461552	-0.319806	0.252722	0.200287

2、使用&符号可以实现多条件筛选

```
# 使用&符号可以实现多条件筛选（比如筛选出D列中大于0且C列中小于0的所有行）
df[(df.D>0)&(df.C<0)]
```

	A	B	C	D
2017-01-03	-1.704362	0.218734	-1.052347	0.369618
2017-01-05	0.687470	0.123437	-1.455018	0.240838

3、使用|符号可以实现多条件筛选

```
# 使用|符号可以实现多条件筛选（比如筛选出D列中大于0或C列中大于0的所有行）
df[(df.D>0)|(df.C>0)]
```

	A	B	C	D
2017-01-01	0.426529	0.193027	0.090471	-0.472144
2017-01-03	-1.704362	0.218734	-1.052347	0.369618
2017-01-05	0.687470	0.123437	-1.455018	0.240838
2017-01-06	-0.461552	-0.319806	0.252722	0.200287

4、我们也可以先限定我们需要的列或行，然后将其他行作为筛选条件

```
# 我们也可以先限定我们需要的列或行，然后将其他行作为筛选条件
df[['A','C']][(df.B>0)&(df.D>0)]
```

	A	C
2017-01-03	-1.704362	-1.052347
2017-01-05	0.687470	-1.455018

5、使用isin方法来筛选特定的值，把要筛选的值写到一个列表里

```
# 使用isin方法来筛选特定的值，把要筛选的值写到一个列表里
goal_list = [0.369618,0.200287,0.240838]
df['D'].isin(goal_list)
```

```
2017-01-01    False
2017-01-02    False
2017-01-03    False
2017-01-04    False
2017-01-05    False
2017-01-06    False
```

增加和删除

0、原数据

df

	A	B	C	D
2017-01-01	-1.074422	0.260438	0.247010	-0.751180
2017-01-02	1.422712	1.080154	1.022007	-1.535193
2017-01-03	-0.212700	-0.252385	-0.757682	1.264396
2017-01-04	-1.200412	0.795977	0.580242	-0.891465
2017-01-05	-0.817875	0.986500	-1.249263	2.026642
2017-01-06	1.245176	0.887597	1.482469	-1.186458

1、增加列

```
# 在df数据中增加一列
df['E'] = pd.Series(np.random.randn(6),index=df.index)
df
```

	A	B	C	D	E
2017-01-01	-1.074422	0.260438	0.247010	-0.751180	-1.628038
2017-01-02	1.422712	1.080154	1.022007	-1.535193	-1.008447
2017-01-03	-0.212700	-0.252385	-0.757682	1.264396	0.816551
2017-01-04	-1.200412	0.795977	0.580242	-0.891465	0.516260
2017-01-05	-0.817875	0.986500	-1.249263	2.026642	0.014172
2017-01-06	1.245176	0.887597	1.482469	-1.186458	-1.342964

```
# 还可以插入一列数据到任意位置，如第2列
df.insert(1,'a',np.random.randn(6))
df
```

	A	a	B	C	D	E
2017-01-01	-1.074422	1.003573	0.260438	0.247010	-0.751180	-1.628038
2017-01-02	1.422712	-0.654633	1.080154	1.022007	-1.535193	-1.008447
2017-01-03	-0.212700	1.799204	-0.252385	-0.757682	1.264396	0.816551
2017-01-04	-1.200412	0.472393	0.795977	0.580242	-0.891465	0.516260
2017-01-05	-0.817875	-0.812925	0.986500	-1.249263	2.026642	0.014172
2017-01-06	1.245176	1.807374	0.887597	1.482469	-1.186458	-1.342964

2、删除列

```
# 永久删除一列数据用del
del df['a']
df
```

	A	B	C	D	E
2017-01-01	-1.074422	0.260438	0.247010	-0.751180	-1.628038
2017-01-02	1.422712	1.080154	1.022007	-1.535193	-1.008447
2017-01-03	-0.212700	-0.252385	-0.757682	1.264396	0.816551
2017-01-04	-1.200412	0.795977	0.580242	-0.891465	0.516260
2017-01-05	-0.817875	0.986500	-1.249263	2.026642	0.014172
2017-01-06	1.245176	0.887597	1.482469	-1.186458	-1.342964

```
# 用drop不改变原有的df中的数据，而是返回另一个dataframe来存放删除后的数据
df9 = df.drop(['D','E'],axis=1)
df9
```

	A	B	C
2017-01-01	-1.074422	0.260438	0.247010
2017-01-02	1.422712	1.080154	1.022007
2017-01-03	-0.212700	-0.252385	-0.757682
2017-01-04	-1.200412	0.795977	0.580242
2017-01-05	-0.817875	0.986500	-1.249263
2017-01-06	1.245176	0.887597	1.482469

```
# 可以看到df仍然没有改变
df
```

	A	B	C	D	E
2017-01-01	-1.074422	0.260438	0.247010	-0.751180	-1.628038
2017-01-02	1.422712	1.080154	1.022007	-1.535193	-1.008447
2017-01-03	-0.212700	-0.252385	-0.757682	1.264396	0.816551
2017-01-04	-1.200412	0.795977	0.580242	-0.891465	0.516260
2017-01-05	-0.817875	0.986500	-1.249263	2.026642	0.014172
2017-01-06	1.245176	0.887597	1.482469	-1.186458	-1.342964

计数统计

1、读取数据

```
# 读取数据
df5 = pd.read_csv('C:/Users/Administrator/Desktop/Rong360_credit_forecasting/train/bill_detail.csv',encoding='gbk')
# 显示前3行数据
df5.head(3)
```

	ID	bill_time	bank_id	amount_of_pre_bill	account_of_repay	credit_card_capacity	bill_balance	min_repay
0	3150	5926182226	14	20.416610	18.851841	21.580708	19.754017	19.372499
1	3150	5926182246	6	20.701662	19.579708	21.000890	19.904695	18.771018
2	3150	5926182267	16	19.193763	19.193763	19.460445	19.290633	16.294952

2、统计银行id在1到16之间的数量

```
# 统计银行id在1到16之间的数量
df6 = df5[df5['bank_id']<17]
id_count = df6['bank_id'].value_counts()
id_count
```

```
7     397436
14    272940
4     223082
16    217683
6     214527
3     189765
10    185577
2     116923
8     116776
11    113551
13     82780
15     81857
9      77616
5      41948
1       3441
12      1579
```

3、统计amount_of_pre_bill的总数

```
# 统计amount_of_pre_bill的总数
df7 = df6['amount_of_pre_bill']
df7.sum()
```

```
33370373.327058665
```

数据分组

1、groupby()实现分组

```
# 首先以ID列进行分组,使用grouped.first()打印出来的为每一组的第一行数据,head()显示前5行
group_ID = df5.groupby('ID')
group_ID.first().head(3)
```

	bill_time	bank_id	amount_of_pre_bill	account_of_repay	credit_card_capacity	bill_balance
ID						
2	0	4	17.389606	17.389606	19.460445	16.656269
3	0	2	18.371030	19.495329	18.361833	18.325511
4	5894225007	7	16.791567	0.000000	19.460445	16.854184

```
# 以两列以上进行分组，groupby参数为一个列表
group_id = df5.groupby(['ID','bank_id'])
group_id.first().head()
```

		bill_time	amount_of_pre_bill	account_of_repay	credit_card_capacity	bill_balance
ID	bank_id					
2	4	0	17.389606	17.389606	19.460445	16.656269
	9	5910282927	20.210676	19.484314	19.971271	18.503332
	16	5908250547	21.580308	21.600903	0.000000	21.578907
3	2	0	18.371030	19.495329	18.361833	18.325511
4	3	5894272767	0.000000	0.000000	20.441274	18.772836

2、aggregate()实现数据分组计算

```
# groupby.aggregate()实现数据分组计算
df5.groupby(['ID','bank_id']).aggregate(np.sum).head()
```

		bill_time	amount_of_pre_bill	account_of_repay	credit_card_capacity	bill_balance
ID	bank_id					
2	4	47322120516	108.590843	110.127042	176.165657	167.369954
	9	23657746668	79.698032	74.339122	79.885084	74.119572
	16	59192890830	172.445595	166.921073	172.645664	215.567977
3	2	0	36.733542	37.160348	36.723666	36.696541
4	3	82759309458	0.000000	0.000000	286.177836	276.793989

3、size()查看各组数据量

```
# 用size()方法返回各组数据量
df6.size().head(8)
```

```
ID  bank_id
2   4          9
    9          4
    16        10
3   2          2
4   3         14
    7         14
    11         8
5   7          2
```

4、describe()对各组数据进行描述性统计

```
# describe()方法对各组数据进行描述性统计
df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	-0.106253	0.626380	0.220797	-0.178876
std	1.167593	0.517443	1.047104	1.458594
min	-1.200412	-0.252385	-1.249263	-1.535193
25%	-1.010285	0.394323	-0.506509	-1.112710
50%	-0.515287	0.841787	0.413626	-0.821322
75%	0.880707	0.961774	0.911565	0.760502
max	1.422712	1.080154	1.482469	2.026642

5、agg分组多种计算

```
# 将读取的CSV文件内容df5按银行id进行排序
df8 = df5.groupby('bank_id')
# 计算各组数据的总数、平均数、标准差
df8.agg([np.sum, np.mean, np.std]).head(3)
```

	ID			bill_time		
	sum	mean	std	sum	mean	std
bank_id						
1	86439920	25120.581226	14833.499135	561895090725	163294126	9.692482e+08
2	3023938231	25862.646622	16441.065768	539447088500488	4613695239	2.448435e+09
3	5277649807	27811.502685	15814.808398	925879022790042	4879082142	2.243280e+09

3 rows × 42 columns

```
# 也可以只选择某列进行统计,如只对amount_of_pre_bill项进行统计
df8['amount_of_pre_bill'].agg([np.sum, np.mean, np.std]).head(3)
```

	sum	mean	std
bank_id			
1	9.185189e+03	2.669337	6.329477
2	1.802219e+06	15.413722	10.367568
3	5.463976e+04	0.287934	2.468761

```
# 假如我们需要定制显示的标题可以如此设置
df8['amount_of_pre_bill'].agg({'总计':np.sum,'均值':np.mean,'标准差':np.std}).head(3)
```

	标准差	均值	总计
bank_id			
1	6.329477	2.669337	9.185189e+03
2	10.367568	15.413722	1.802219e+06
3	2.468761	0.287934	5.463976e+04

缺失值处理

1、创建数据

```
# 在简单的运算中，如果遇到缺失值，运算结果在相应的位置也是缺失值；
# 在描述性统计中，Nan都是作为0进行运算；
df10 = pd.DataFrame(np.random.randn(4,3),index=list('abcd'),columns=['one','two','three'])
df10.ix[1,:-1] = np.nan
df10.ix[1:-1,2] = np.nan
df10
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	NaN	NaN	NaN
c	-0.007093	-0.285383	NaN
d	1.767437	0.166128	0.059600

2、用固定数替代确实值

```
# DataFrame.fillna()方法填充缺失值（如使用0替代缺失值）
df10.fillna(0)
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	0.000000	0.000000	0.000000
c	-0.007093	-0.285383	0.000000
d	1.767437	0.166128	0.059600

```
# 用一个字符串代替缺失值
df10.fillna('missing')
```

	one	two	three
a	0.325564	-1.42049	2.55105
b	missing	missing	missing
c	-0.00709253	-0.285383	missing
d	1.76744	0.166128	0.0596

3、用临近值替代缺失值

```
# 用前一个数据代替NaN：method='pad'
df10.fillna(method='pad')
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	0.325564	-1.420485	2.551049
c	-0.007093	-0.285383	2.551049
d	1.767437	0.166128	0.059600

```
# 与pad相反，bfill表示用后一个数据代替NaN
df10.fillna(method='bfill')
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	-0.007093	-0.285383	0.059600
c	-0.007093	-0.285383	0.059600
d	1.767437	0.166128	0.059600

```
# 用limit限制每列可以替代NaN的数目
df10.fillna(method='bfill',limit=1)
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	-0.007093	-0.285383	NaN
c	-0.007093	-0.285383	0.059600
d	1.767437	0.166128	0.059600

4、用统计值替代缺失值

```
# 还可以使用平均数或者其他描述性统计量来代替NaN
df10.fillna(df10.mean())
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	0.695303	-0.513247	2.551049
c	-0.007093	-0.285383	1.305325
d	1.767437	0.166128	0.059600

```
# 还可以选择哪一列进行缺失值的处理
df10.fillna(df10.mean()['one':'two'])
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	0.695303	-0.513247	NaN
c	-0.007093	-0.285383	NaN
d	1.767437	0.166128	0.059600

5、用插值法填补缺失值

```
# 插值法就是通过两点（x0，y0），（x1，y1）估计中间点的值
df10.interpolate()
```

	one	two	three
a	0.325564	-1.420485	2.551049
b	0.159236	-0.852934	1.720566
c	-0.007093	-0.285383	0.890083
d	1.767437	0.166128	0.059600

```
# 假如index是数字，我们还可以根据数字来进行插值，用到参数method='values'
df10.index = [1,2,3,4]
df10.interpolate(method='values')
```

	one	two	three
1	0.325564	-1.420485	2.551049
2	0.159236	-0.852934	1.720566
3	-0.007093	-0.285383	0.890083
4	1.767437	0.166128	0.059600

```
# 同样，如果index是时间，我们可以用method=time来达到同样的效果
df10.index = pd.date_range(20170101,periods=4)
df10.interpolate(method='time')
```

	one	two	three
1970-01-01 00:00:00.020170101	0.325564	-1.420485	2.551049
1970-01-02 00:00:00.020170101	0.159236	-0.852934	1.720566
1970-01-03 00:00:00.020170101	-0.007093	-0.285383	0.890083
1970-01-04 00:00:00.020170101	1.767437	0.166128	0.059600

6、删除缺失值

```
# 可以选择删除行或者删除列，用的都是DataFrame.dropna()
# 通常情况下，我们选择删除行，使用参数axis=0
df10.dropna(axis=0)
```

	one	two	three
a	0.325564	-1.420485	2.551049
d	1.767437	0.166128	0.059600

```
# 有时候也会选择删除列
df10.dropna(axis=1)
```

a
b
c
d

7、值替换

```
# 创建一个Series
ser = pd.Series([0,1,2,3,4])
```

```
# 具体数值的替换
print ser.replace(0,8)
```

```
0    8
1    1
2    2
3    3
4    4
```

```
# 列表到列表的替换
ser.replace([0,1,2,3,4],[4,3,2,1,0])
```

```
0    4
1    3
2    2
3    1
4    0
```

```
# 使用字典映射，如：将1替换为11，将2替换为12
ser.replace({1:11,2:12})
```

```
0     0
1    11
2    12
3     3
4     4
```

```
# 以上方法同样适用于DataFrame对象
df11 = pd.DataFrame({'a':[0,1,2,3],'b':[2,6,7,8]})
df11.replace(2,20)
```

	a	b
0	0	20
1	1	6
2	20	7
3	3	8

```
# 假如DataFrame中只有一列数据需要替换数值，我们可以单独操作者一列
df11['a'].replace([0,1,2,3],[5,4,3,2])
```

```
0    5
1    4
2    3
3    2
```

排序与合并

1、排序sort_values(by=' ')

```
# 读取数据源over_due.csv文件
import pandas as pd
over_due = pd.read_csv("C:/Users/Administrator/Desktop/Rong360_credit_forecasting/train/over_due.csv")
over_due.head()
```

	ID	overdue
0	1	0
1	2	0
2	3	0
3	4	1
4	5	0

```
# 读取数据源 user_info.csv文件
user_info = pd.read_csv("C:/Users/Administrator/Desktop/Rong360_credit_forecasting/train/user_info.csv")
user_info.head()
```

	ID	gender	job	edu	marriage	family_type
0	2583	2	2	2	1	1
1	34764	1	2	3	3	1
2	9554	1	2	4	2	2
3	6720	1	2	3	3	2
4	29165	1	2	4	1	4

```
# 将数据表按ID号进行排序
over_due2 = over_due.sort_values(by='ID')
user_info2 = user_info.sort_values(by='ID')
user_info2.head()
```

	ID	gender	job	edu	marriage	family_type
134	1	1	2	3	1	3
213	2	1	2	3	2	1
370	3	1	4	4	1	4
397	4	1	4	4	3	2
499	5	1	2	2	3	1

2、合并--merge ()

```
# pandas提供了一个类似于关系数据库的连接join操作的方法merage
# 将over_due和user_info两张表按ID列进行合并,当遇到某行ID不一样时，以右边（即over_due）为基准
data_train = pd.merge(user_info2,over_due2,how='right',on='ID')
data_train.head()
```

	ID	gender	job	edu	marriage	family_type	overdue
0	1.0	1.0	2.0	3.0	1.0	3.0	0
1	2.0	1.0	2.0	3.0	2.0	1.0	0
2	3.0	1.0	4.0	4.0	1.0	4.0	0
3	4.0	1.0	4.0	4.0	3.0	2.0	1
4	5.0	1.0	2.0	2.0	3.0	1.0	0

3、合并--concat ()

```
import numpy as np
# 创建数据集df1(3*4)
df1 = pd.DataFrame(np.random.rand(3,4),columns=['A','B','C','D'])
df1
```

	A	B	C	D
0	0.931738	0.176487	0.366165	0.187267
1	0.231919	0.322953	0.116273	0.283859
2	0.509275	0.320339	0.937341	0.346943

```
# 创建数据集df2(2*4)
df2 = pd.DataFrame(np.random.rand(2,4),columns=['B','D','A','E'])
df2
```

	B	D	A	E
0	0.127387	0.299934	0.596534	0.472561
1	0.651947	0.353489	0.092125	0.963692

```
# 直接使用concat，默认按列索引进行拼接
# 缺失的部分直接用NaN填充
pd.concat([df1,df2])
```

	A	B	C	D	E
0	0.931738	0.176487	0.366165	0.187267	NaN
1	0.231919	0.322953	0.116273	0.283859	NaN
2	0.509275	0.320339	0.937341	0.346943	NaN
0	0.596534	0.127387	NaN	0.299934	0.472561
1	0.092125	0.651947	NaN	0.353489	0.963692

```
# 直接使用concat，默认按列索引进行拼接
# 缺失的部分直接用NaN填充
# 行索引忽略之前的重新生成
pd.concat([df1,df2],ignore_index=True)
```

	A	B	C	D	E
0	0.931738	0.176487	0.366165	0.187267	NaN
1	0.231919	0.322953	0.116273	0.283859	NaN
2	0.509275	0.320339	0.937341	0.346943	NaN
3	0.596534	0.127387	NaN	0.299934	0.472561
4	0.092125	0.651947	NaN	0.353489	0.963692

```
# 指定axis=1，表示按行索引进行拼接
# 同样缺失部分用NaN填充
pd.concat([df1,df2],axis=1)
```

	A	B	C	D	B	D	A	E
0	0.931738	0.176487	0.366165	0.187267	0.127387	0.299934	0.596534	0.472561
1	0.231919	0.322953	0.116273	0.283859	0.651947	0.353489	0.092125	0.963692
2	0.509275	0.320339	0.937341	0.346943	NaN	NaN	NaN	NaN

```
# 指定axis=1，表示按行索引进行拼接
# 同样缺失部分用NaN填充
# 列索引忽略之前的重新生成
pd.concat([df1,df2],axis=1,ignore_index=True)
```

	0	1	2	3	4	5	6	7
0	0.931738	0.176487	0.366165	0.187267	0.127387	0.299934	0.596534	0.472561
1	0.231919	0.322953	0.116273	0.283859	0.651947	0.353489	0.092125	0.963692
2	0.509275	0.320339	0.937341	0.346943	NaN	NaN	NaN	NaN

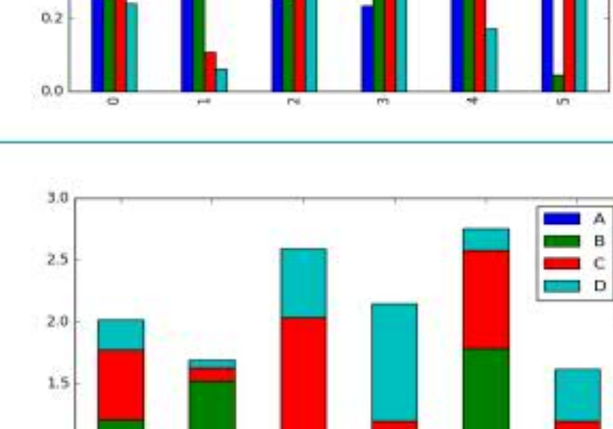
可视化操作

1、散点图

```
# 生成数据源
data = pd.DataFrame(np.random.randn(100,2),columns=list('XY'))
data.head()
```

	X	Y
0	0.398123	-1.047429
1	-1.271065	1.690244
2	0.420676	-2.323342
3	-0.331538	0.555148
4	1.254394	0.643024

```
# 绘制散点图
plt = data.plot(kind='scatter',x='X',y='Y').get_figure()
plt.savefig('C:/Users/Administrator/Desktop/plot.png')
```

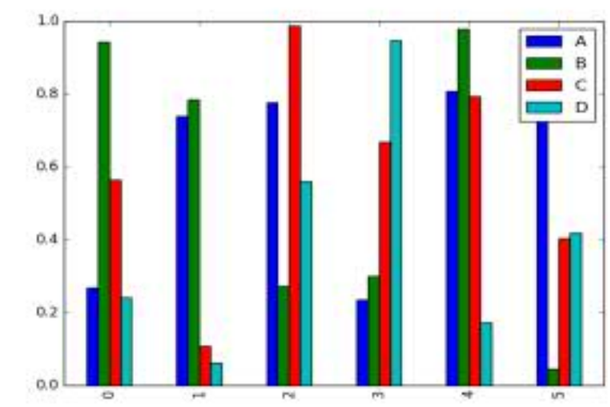


2、柱状图

```
# 生成创建一个6*4的正数数据集
data2 = pd.DataFrame(np.random.rand(6,4),columns=list('ABCD'))
data2
```

	A	B	C	D
0	0.268960	0.944233	0.564392	0.241277
1	0.739152	0.786348	0.108528	0.062371
2	0.777080	0.272329	0.986666	0.559934
3	0.234701	0.300360	0.667984	0.946984
4	0.807806	0.979275	0.793605	0.172620
5	0.753253	0.046280	0.403574	0.417993

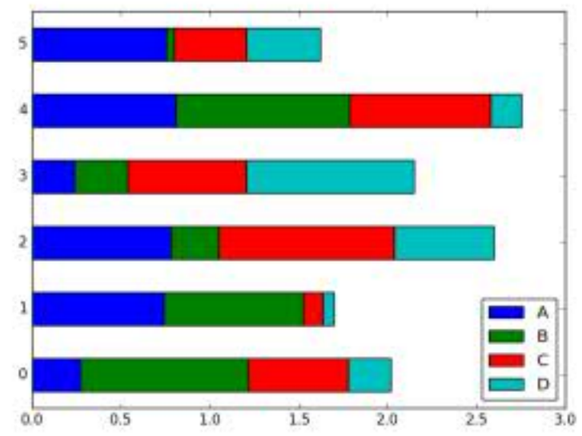
```
# 绘制柱状图
plt2 = data2.plot(kind='bar').get_figure()
plt2.savefig('C:/Users/Administrator/Desktop/plot2.png')
```



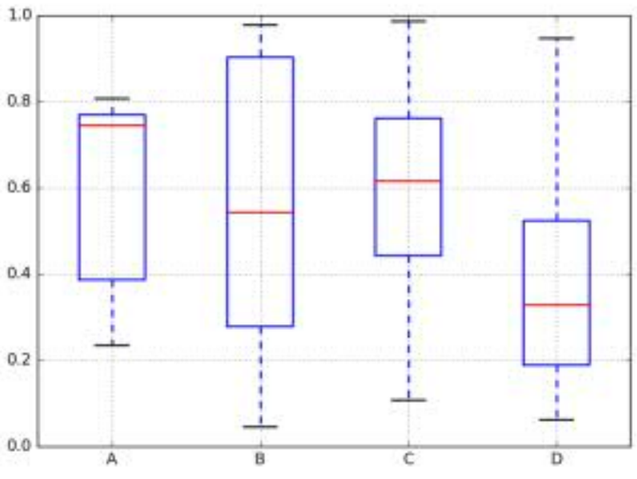

```
# 绘制堆积直方图
plt3 = data2.plot(kind='bar',stacked=True).get_figure()
plt3.savefig('C:/Users/Administrator/Desktop/plot3.png')
```


3、箱形图

```
# 箱形图用于显示数据的一些基本的统计量：中位数、平均数、四分位数等
import matplotlib.pyplot as plt
data2.boxplot()
plt.savefig('C:/Users/Administrator/Desktop/plot5.png')
```




```
# 箱形图用于显示数据的一些基本的统计量：中位数、平均数、四分位数等
import matplotlib.pyplot as plt
data2.boxplot()
plt.savefig('C:/Users/Administrator/Desktop/plot5.png')
```



字符串操作

1、生成字符串

```
# 生成字符串
s = pd.Series(list('ABCDE'))
print s
```

```
0    A
1    B
2    C
3    D
4    E
```

2、字符串大小写转换

```
# 将字符串转化为小写
s.str.lower()
```

```
0    a
1    b
2    c
3    d
4    e
```

```
# 转化为大写
s.str.upper()
```

```
0    A
1    B
2    C
3    D
4    E
```

3、获取字符串长度

```
# 获取字符串长度
s.str.len()
```

```
0    1
1    1
2    1
3    1
4    1
```

4、切割字符串，将字符串转换为list

```
# 切割字符串，将字符串转换为list
s.str.split()
```

```
0    [A]
1    [B]
2    [C]
3    [D]
4    [E]
```

5、获取字符串中的第一个字符

```
# 获取字符串中的第一个字符
s.str.get(0)
```

```
0    A
1    B
2    C
3    D
4    E
```

6、替换字符串

```
# 替换字符串(replace的第一个参数是正则表达式，第二个参数是要替换成的字符串。)
s.str.replace('A','a')
```

```
0    a
1    B
2    C
3    D
4    E
```

6、字符串匹配

```
# 字符串匹配
s2 = pd.Series(['a1','A1','B1','a2c',np.nan])
pattern = r'[a-z][0-9]'
print s2.str.contains(pattern)
```

```
0     True
1    False
2    False
3     True
4      NaN
```

```
# 们可以使用na参数来规定出现NaN数据的时候匹配成True 还是False
print s2.str.contains(pattern,na=False)
```

```
0     True
1    False
2    False
3     True
4    False
```

```
# match方法可以严格匹配字符串
s2.str.match(pattern,na=False)
```

```
0     True
1    False
2    False
3     True
4    False
```

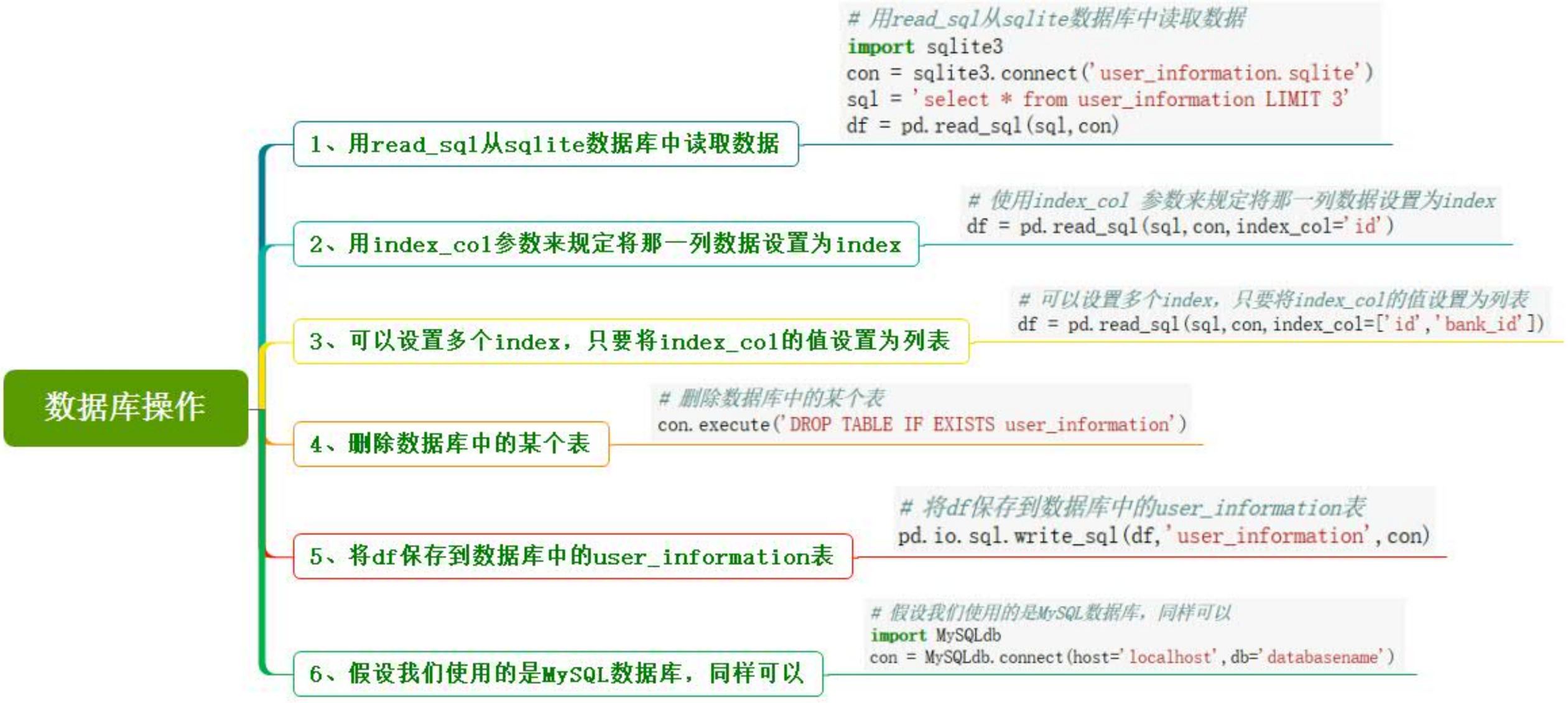
7、判断字符串开头结尾

```
# 用startswith判断是否以某个字母开头
s2.str.startswith('A',na=False)
```

```
0    False
1     True
2    False
3    False
4    False
```

```
# 类似可以用endswith判断字符串是否以某某结尾
s2.str.endswith('c',na=False)
```

```
0    False
1    False
2    False
3     True
4    False
```


数据库操作
1、用read_sql从sqlite数据库中读取数据
# 用read_sql从sqlite数据库中读取数据
import sqlite3
con = sqlite3.connect('user_information.sqlite')
sql = 'select * from user_information LIMIT 3'
df = pd.read_sql(sql,con)
2、用index_col参数来规定将那一列数据设置为index
# 使用index_col 参数来规定将那一列数据设置为index
df = pd.read_sql(sql,con,index_col='id')
3、可以设置多个index，只要将index_col的值设置为列表
# 可以设置多个index，只要将index_col的值设置为列表
df = pd.read_sql(sql,con,index_col=['id','bank_id'])
4、删除数据库中的某个表
# 删除数据库中的某个表
con.execute('DROP TABLE IF EXISTS user_information')
5、将df保存到数据库中的user_information表
# 将df保存到数据库中的user_information表
pd.io.sql.write_sql(df,'user_information',con)
6、假设我们使用的是MySQL数据库，同样可以
# 假设我们使用的是MySQL数据库，同样可以
import MySQLdb
con = MySQLdb.connect(host='localhost',db='databasename')

一个矩阵或者向量减去一个常数，那么通常是矩阵中的每一个元素减去这个常数，这就是广播

广播

```
# 一个矩阵或者向量减去一个常数，那么通常是矩阵中的每一个元素减去这个常数，这就是广播
# 通过参数axis可指定广播的维度,axis=1 或者axis='column'这两种写法是相同的
row = df.ix[1]
# 将df中每一行与row做减法
df.sub(row,axis='columns')
```

	A	B	C	D	E
2017-01-01	-2.497134	-0.819716	-0.774996	0.784014	-0.619590
2017-01-02	0.000000	0.000000	0.000000	0.000000	0.000000
2017-01-03	-1.635412	-1.332539	-1.779688	2.799590	1.824998
2017-01-04	-2.623123	-0.284177	-0.441765	0.643729	1.524707
2017-01-05	-2.240586	-0.093654	-2.271269	3.561835	1.022619
2017-01-06	-0.177536	-0.192557	0.460463	0.348735	-0.334517