前几篇介绍了 NumPy、pandas、matplotlib 三大库的基本操作。掌握了这些库，也就掌握了用 Python 对数据进行整合、清洗、分析及可视化的技能。

趁热打铁，本文继续来讲，针对数据分析过程中的常见数据操作，Python 是怎么用的。主要包括两大块：

- 数据规整化：清理、转换、合并、重塑
- 数据聚合与分组运算

一些朋友反映数据结构看不懂，那需要巩固一下数据库的知识。

数据规整化：清理、转换、合并、重塑

合并数据集

- pandas.merge：可根据一个或多个键将不同 DataFrame 中的行链接起来。
- pandas.concat：可沿着一条轴将多个对象堆叠到一起。
- combine_first：可将重复数据编接在一起，用一个对象中的值填充另一个对象中的缺失值。

1. 数据库风格的 DataFrame 合并

数据集的合并或连接运算：通过一个或多个键将行链接起来。

```
In [3]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                         'data1': range(7)})

In [4]: df2 = DataFrame({'key': ['a', 'b', 'd'],
                         'data2': range(3)})

In [5]: df1
Out[5]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

In [6]: df2
Out[6]:
   data2 key
0      0   a
1      1   b
```

多对一的合并：

```
In [7]: pd.merge(df1, df2)
```

Out[7]:

	data1	key	data2
0	0	b	1
1	1	b	1
2	6	b	1
3	2	a	0
4	4	a	0
5	5	a	0

若没有指定用哪个列进行连接，merge 会将重叠列名当做键，指定如下：

```
In [8]: pd.merge(df1, df2, on='key')
```

Out[8]:

	data1	key	data2
0	0	b	1
1	1	b	1
2	6	b	1
3	2	a	0
4	4	a	0
5	5	a	0

若两个对象的列名不同，可分别进行指定：

```
In [11]: df3 = DataFrame({'lkey':['b','b','a','c','a','a','b'],
                          'data1':range(7)})

In [12]: df4 = DataFrame({'rkey':['a','b','d'],
                          'data2':range(3)})

In [13]: pd.merge(df3,df4,left_on='lkey',right_on='rkey')
```

Out[13]:

	data1	lkey	data2	rkey
0	0	b	1	b
1	1	b	1	b
2	6	b	1	b
3	2	a	0	a
4	4	a	0	a
5	5	a	0	a

默认情况下，merge 做 inner 连接，结果中的键是交集。外连接求取的是键的并集：

```
In [14]: pd.merge(df1,df2,how='outer')
```

Out[14]:

	data1	key	data2
0	0.0	b	1.0
1	1.0	b	1.0
2	6.0	b	1.0
3	2.0	a	0.0
4	4.0	a	0.0
5	5.0	a	0.0
6	3.0	c	NaN
7	NaN	d	2.0

多对多的合并操作：

```
In [15]: pd.merge(df1,df2, on='key',how='left')
```

Out[15]:

	data1	key	data2
0	0	b	1.0
1	1	b	1.0
2	2	a	0.0
3	3	c	NaN
4	4	a	0.0
5	5	a	0.0
6	6	b	1.0

```
In [16]: pd.merge(df1,df2, on='key',how='right')
```

Out[16]:

	data1	key	data2
0	0.0	b	1
1	1.0	b	1
2	6.0	b	1
3	2.0	a	0
4	4.0	a	0
5	5.0	a	0

连接方式只影响出现在结果中的键。

根据多个键进行合并，传入一个由列名组成的列表：

```
In [29]: left = DataFrame({'key1':['foo','foo','bar'],
                           'key2':['one','two','one'],
                           'lval':[1,2,3]})

In [30]: right = DataFrame({'key1':['foo','foo','bar','bar'],
                            'key2':['one','one','one','two'],
                            'rval':[4,5,6,7]})

In [31]: pd.merge(left,right,on=['key1','key2'],how='outer')
Out[31]:
```

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

在进行列-列连接时，DataFrame 对象中的索引会被丢弃。

suffixes 选项：指定附加到左右两个 DataFrame 对象的重叠列名上的字符串。

```
In [32]: pd.merge(left,right,on='key1')
Out[32]:
```

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

```
In [33]: pd.merge(left,right,on='key1',suffixes=('_left','_right'))
Out[33]:
```

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

2. 索引上的合并

当 DataFrame 中的连接键位于其索引中时，传入 left_index=True、right_index=True，以说明索引应该被用作连接键：

```
In [34]: left1 = DataFrame({'key': ['a','b','a','a','b','c'],
                            'value': range(6)})

In [35]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a','b'])

In [36]: left1
Out[36]:
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

```
In [37]: right1
Out[37]:
```

	group_val
a	3.5
b	7.0

```
In [38]: pd.merge(left1, right1, left_on='key', right_index=True)
```

Out[38]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

对于层次化索引的数据：

```
In [47]: lefth = DataFrame({'key1':['Ohio','Ohio','Ohio','Nevada','Nevada'],
                            'key2':[2000,2001,2002,2001,2002],
                            'data':np.arange(5.)})

In [49]: righth = DataFrame(np.arange(12).reshape((6,2)),
                            index=[['Nevada','Nevada','Ohio','Ohio','Ohio','Ohio'],
                                   [2001,2000,2000,2000,2001,2002]],
                            columns=['event1','event2'])

In [50]: lefth
```

Out[50]:

	data	key1	key2
0	0.0	Ohio	2000
1	1.0	Ohio	2001
2	2.0	Ohio	2002
3	3.0	Nevada	2001
4	4.0	Nevada	2002

```
In [51]: righth
```

Out[51]:

		event1	event2
Nevada	2001	0	1
	2000	2	3

必须以列表的形式指明用作合并键的列（注意对重复索引值的处理）：

```
In [53]: pd.merge(lefth,righth, left_on=['key1','key2'], right_index=True)
```

Out[53]:

	data	key1	key2	event1	event2
0	0.0	Ohio	2000	4	5
0	0.0	Ohio	2000	6	7
1	1.0	Ohio	2001	8	9
2	2.0	Ohio	2002	10	11
3	3.0	Nevada	2001	0	1

使用合并双方的索引：

```
In [62]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

Out[62]:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

DataFrame 的 join 实例方法：

```
In [63]: left2.join(right2, how='outer')
```

Out[63]:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

```
In [56]: left2
```

Out[56]:

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

```
In [57]: right2
```

Out[57]:

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

更方便的实现按索引合并，不管有没有重叠的列。在连接键上作左连接。

支持参数 DataFrame 的索引跟调用者 DataFrame 的某个列之间的连接：

```
In [64]: left1.join(right1, on='key')
```

Out[64]:

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0
5	c	5	NaN

```
In [44]: left1
```

Out[44]:

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

```
In [45]: right1
```

Out[45]:

	group_val
a	3.5
b	7.0

对于简单的索引合并，可以向 join 传入一组 DataFrame（concat 函数也是这个功能）：

```
In [65]: another = DataFrame([[7.,8.],[9.,10.],[11.,12.],[16.,17.]],
                             index=['a','c','e','f'], columns=['New York', 'Oregon'])

In [66]: left2.join([right2, another])
```

Out[66]:

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0	2.0	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0	9.0	10.0
e	5.0	6.0	13.0	14.0	11.0	12.0

```
In [67]: another
```

Out[67]:

	New York	Oregon
a	7.0	8.0
c	9.0	10.0
e	11.0	12.0
f	16.0	17.0

3. 轴向连接

数据合并运算：

- 连接(concatenation)
- 绑定(binding)
- 堆叠(stacking)

NumPy 有一个用于合并原始 NumPy 数组的 concatenation 函数：

```
In [68]: arr = np.arange(12).reshape((3,4))

In [69]: arr
Out[69]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [70]: np.concatenate([arr,arr], axis=1)
Out[70]: array([[ 0,  1,  2,  3,  0,  1,  2,  3],
                [ 4,  5,  6,  7,  4,  5,  6,  7],
                [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

pandas 的 concat 函数：

```
In [71]: s1 = Series([0,1], index=['a','b'])

In [72]: s2 = Series([2,3,4], index=['c','d','e'])

In [73]: s3 = Series([5,6], index=['f','g'])

In [74]: pd.concat([s1,s2,s3])
Out[74]: a    0
         b    1
         c    2
         d    3
         e    4
         f    5
         g    6
         dtype: int64
```

默认情况下，concat 在 axis=0 上工作，产生一个新 Series。传入 axis=1，产生一个 DataFrame：

```
In [75]: pd.concat([s1,s2,s3], axis=1)
Out[75]:
```

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

这种情况下，另外一条轴上没有重叠，传入 join = 'inner' 得到它们的交集：

```
In [80]: pd.concat([s1,s4], axis=1, join='inner')
Out[80]:
```

	0	1
a	0	0
b	1	5

使用 key 参数，在连接轴上创建一个层次化索引：

```
In [83]: result = pd.concat([s1,s1,s3], keys=['one','two','three'])

In [84]: result
Out[84]: one    a    0
         b    1
two      a    0
         b    1
three    f    5
         g    6
dtype: int64

In [85]: result.unstack()
Out[85]:
```

	a	b	f	g
one	0.0	1.0	NaN	NaN
two	0.0	1.0	NaN	NaN
three	NaN	NaN	5.0	6.0

沿着 axis=1 对 Series 进行合并，keys 就会成为 DataFrame 的列头：

```
In [86]: pd.concat([s1,s1,s3], axis=1, keys=['one','two','three'])
Out[86]:
```

	one	two	three
a	0.0	0.0	NaN
b	1.0	1.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

对 DataFrame 对象也是如此：

```
In [87]: df1 = DataFrame(np.arange(6).reshape(3,2),index=['a','b','c'],
                         columns=['one','two'])

In [90]: df2 = DataFrame(5 + np.arange(4).reshape(2,2), index=['a','c'],
                         columns=['three','four'])

In [91]: pd.concat([df1,df2], axis=1, keys=['level1','level2'])
Out[91]:
```

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

传入一个字典，则字典的键会被当做 keys 选项的值：

```
In [99]: pd.concat({'level1':df1, 'level':df2}, axis=1)
Out[99]:
```

	level		level1	
	three	four	one	two
a	5.0	6.0	0	1
b	NaN	NaN	2	3
c	7.0	8.0	4	5

用于管理层次化索引创建方式的参数：

```
In [100]: pd.concat([df1,df2], axis=1, keys=['level1','level2'],
                    names=['upper','lower'])
```

Out[100]:

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

跟当前分析工作无关的 DataFrame 行索引：

```
In [101]: df1 = DataFrame(np.random.randn(3,4), columns=['a','b','c','d'])
In [102]: df2 = DataFrame(np.random.randn(2,3), columns=['b','d','a'])
In [103]: df1
```

Out[103]:

	a	b	c	d
0	-2.412248	1.413948	1.167305	1.314592
1	-0.322154	0.690420	0.772318	0.684792
2	0.702342	-0.282768	-1.260187	-0.694695

```
In [104]: df2
```

Out[104]:

	b	d	a
0	-1.506173	-0.360153	-0.173009
1	0.161619	0.431895	-0.027429

传入 ignore_index = True：

```
In [105]: pd.concat([df1,df2], ignore_index=True)
```

Out[105]:

	a	b	c	d
0	-2.412248	1.413948	1.167305	1.314592
1	-0.322154	0.690420	0.772318	0.684792
2	0.702342	-0.282768	-1.260187	-0.694695
3	-0.173009	-1.506173	NaN	-0.360153
4	-0.027429	0.161619	NaN	0.431895

```
In [106]: pd.concat([df1,df2])
```

Out[106]:

	a	b	c	d
0	-2.412248	1.413948	1.167305	1.314592
1	-0.322154	0.690420	0.772318	0.684792
2	0.702342	-0.282768	-1.260187	-0.694695
0	-0.173009	-1.506173	NaN	-0.360153
1	-0.027429	0.161619	NaN	0.431895

4. 合并重叠数据

关于有索引全部或部分重叠的两个数据集。

NumPy 的 where 函数，用于表达一种矢量化的 if-else：

```
In [108]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
                     index=['f','e','d','c','b','a'])

In [109]: b = Series(np.arange(len(a), dtype=np.float64),
                     index=['f','e','d','c','b','a'])

In [110]: b[-1] = np.nan

In [111]: a
Out[111]: f    NaN
          e    2.5
          d    NaN
          c    3.5
          b    4.5
          a    NaN
          dtype: float64

In [112]: b
Out[112]: f    0.0
          e    1.0
          d    2.0
          c    3.0
          b    4.0
          a    NaN
          dtype: float64
```

```
In [113]: np.where(pd.isnull(a), b, a)
Out[113]: array([0. , 2.5, 2. , 3.5, 4.5, nan])
```

Series 的 combine_first 方法，实现与上面一样的功能，并会进行数据对齐：

```
In [114]: b[:-2].combine_first(a[2:])
Out[114]: a    NaN
          b    4.5
          c    3.0
          d    2.0
          e    1.0
          f    0.0
          dtype: float64
```

对于 DataFrame 一样：

```
In [115]: df1 = DataFrame({'a':[1., np.nan, 5., np.nan],
                           'b':[np.nan, 2., np.nan, 6.],
                           'c':range(2,18,4)})

In [120]: df2 = DataFrame({'a':[5., 4, np.nan, 3., 7.],
                           'b':[np.nan, 3., 4., 6., 8.]})

In [121]: df1.combine_first(df2)
```

Out[121]:

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

```
In [123]: df1
```

Out[123]:

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

```
In [124]: df2
```

Out[124]:

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

可以看作用参数对象中的数据为调用者对象的缺失数据“打补丁”。

重塑和轴向旋转

用于重新排列表格型数据的基础运算：**重塑**(reshape)或**轴向旋转**(pivot)。

1.重塑层次化索引

- stark：将数据的列“旋转”为行
- unstark：将数据的行“旋转”为列

```
In [125]: data = DataFrame(np.arange(6).reshape((2,3)),
                           index=pd.Index(['Ohio', 'Colorado'], name='state'),
                           columns=pd.Index(['one','two','three'], name='number'))

In [126]: data
```

Out[126]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

用 stack 方法将行转为列，得到一个 Series：

```
In [127]: result = data.stack()

In [128]: result
Out[128]: state     number
          Ohio      one       0
                    two       1
                    three     2
          Colorado  one       3
                    two       4
                    three     5
          dtype: int64
```

对层次化索引的 Series，可以用 unstack 将其重新排为一个 DataFrame：

```
In [129]: result.unstack()
```

Out[129]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

默认情况下，unstack 操作最内层。

传入分层级别的编号或名称可对其他级别进行 unstack 操作：

```
In [130]: result.unstack(0)
```

Out[130]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

```
In [131]: result.unstack('state')
```

Out[131]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

如果不是所有的级别值都能在各分组找到的话，unstack 操作可能会引入缺失数据：

```
In [132]: s1 = Series([0,1,2,3], index=['a','b','c','d'])
          s2 = Series([4,5,6], index=['c','d','e'])

In [133]: data2 = pd.concat([s1,s2], keys=['one','two'])

In [135]: data2.unstack()
```

Out[135]:

	a	b	c	d	e
one	0.0	1.0	2.0	3.0	NaN
two	NaN	NaN	4.0	5.0	6.0

stack 默认会滤除缺失数据，因此该运算是可逆的：

```
In [136]: data2.unstack().stack()
Out[136]: one  a    0.0
               b    1.0
               c    2.0
               d    3.0
          two  c    4.0
               d    5.0
               e    6.0
          dtype: float64

In [137]: data2.unstack().stack(dropna=False)
Out[137]: one  a    0.0
               b    1.0
               c    2.0
               d    3.0
               e    NaN
          two  a    NaN
               b    NaN
               c    4.0
               d    5.0
               e    6.0
          dtype: float64
```

对 DataFrame 进行 unstack 操作时，作为旋转轴的级别将会成为结果中的最低级别：

```
In [138]: df = DataFrame({'left':result, 'right':result+5},
                         columns=pd.Index(['left', 'right'], name='side'))

In [139]: df
Out[139]:
```

state	number	side	left	right
Ohio	one		0	5
	two		1	6
	three		2	7
Colorado	one		3	8
	two		4	9
	three		5	10

```
In [140]: df.unstack('state')
Out[140]:
```

side	left		right	
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

```
In [141]: df.unstack('state').stack('side')
Out[141]:
```

	state	Colorado	Ohio
number	side		
one	left	3	0
	right	8	5
two	left	4	1
	right	9	6
three	left	5	2
	right	10	7

1. 将“长格式”旋转为“宽格式”

时间序列数据通常以 “长格式(long)”或“堆叠格式(stacked)”存储在数据库和 CSV 中。

```
In [207]: ldata
Out[207]:
```

	date	item	value
0	1959/3/31	r	2710
1	1959/3/31	i	0
2	1959/3/31	u	5
3	1959/6/30	r	2778
4	1959/6/30	i	2
5	1959/6/30	u	5

转成 DataFrame，用 pivot 方法：

```
In [220]: pivoted = pd.pivot_table(ldata, index=['date', 'item'])

In [221]: pivoted.head()
```

```
In [240]: pivoted.head()
```

Out[240]:

		value
date	item	
1959/3/31	i	0
	r	2710
	u	5
1959/6/30	i	2
	r	2778

得到的 DataFrame 带有层次化的列：

```
In [225]: pivoted = ldata.pivot('date','item')

In [226]: pivoted
```

Out[226]:

	value			value2		
item	i	r	u	i	r	u
date						
1959/3/31	0	2710	5	-0.856683	0.725284	-0.619792
1959/6/30	2	2778	5	-1.392365	-0.680428	-0.055720

```
In [227]: pivoted['value2']
```

Out[227]:

item	i	r	u
date			
1959/3/31	-0.856683	0.725284	-0.619792
1959/6/30	-1.392365	-0.680428	-0.055720

假设有两个需要参与重塑的数据列：

```
In [222]: ldata['value2'] = np.random.randn(len(ldata))

In [223]: ldata
```

Out[223]:

	date	item	value	value2
0	1959/3/31	r	2710	0.725284
1	1959/3/31	i	0	-0.856683
2	1959/3/31	u	5	-0.619792
3	1959/6/30	r	2778	-0.680428
4	1959/6/30	i	2	-1.392365
5	1959/6/30	u	5	-0.055720

pivot 其实只是一个快捷方式：用 set_index 创建层次化索引，再用 unstack 重塑。

```
In [229]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [230]: unstacked
```

Out[230]:

	value			value2		
item	i	r	u	i	r	u
date						
1959/3/31	0	2710	5	-0.856683	0.725284	-0.619792
1959/6/30	2	2778	5	-1.392365	-0.680428	-0.055720

以上是数据的重排，下面是过滤、清理及其他转换工作。

数据转换

1. 移除重复数据

DataFrame 中出现的重复行：

```
In [241]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
                            'k2': [1, 1, 2, 3, 3, 4, 4]})

In [242]: data
```

Out[242]:

	k1	k2
0	one	1
1	one	1
2	one	2
3	two	3
4	two	3
5	two	4
6	two	4

DataFrame 的 duplicated 方法返回一个布尔型 Series，表示各行是否是重复行，drop_duplicates 方法返回一个移除了重复行的 DataFrame：

```
In [243]: data.duplicated()
```

Out[243]:

```
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

```
In [244]: data.drop_duplicates()
```

Out[244]:

	k1	k2
0	one	1
2	one	2
3	two	3
5	two	4

指定部分列进行重复项判断，如只希望根据 k1 列过滤重复项：

```
In [245]: data['v1'] = range(7)

In [246]: data.drop_duplicates(['k1'])
```

Out[246]:

	k1	k2	v1
0	one	1	0
3	two	3	3

duplicated 和 drop_duplicates 默认保留重复数值里第一次出现的组合，传入 keep = last 则保留最后一个：

```
In [250]: data.drop_duplicates(['k1','k2'], keep='last')
```

Out[250]:

	k1	k2	v1
1	one	1	1
2	one	2	2
4	two	3	4
6	two	4	6

```
In [251]: data.drop_duplicates(['k1','k2'])
```

Out[251]:

	k1	k2	v1
0	one	1	0
2	one	2	2
3	two	3	3
5	two	4	5

2. 利用函数或映射进行数据转换

根据数组、Series 或 DataFrame 列中的值来实现转换。

```
In [256]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                                        'Pastrami', 'corned beef', 'Bacon',
                                        'pastrami', 'honey ham', 'nova lox'],
                               'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
          data
```

Out[256]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

编写一个肉类到动物的映射：

```
In [257]: meat_to_animal = {
            'bacon': 'pig',
            'pulled pork': 'pig',
            'pastrami': 'cow',
            'corned beef': 'cow',
            'honey ham': 'pig',
            'nova lox': 'salmon'
          }
```

Series 的 map 方法：可以接受一个函数或含有映射关系的字典型对象，用于修改对象的数据子集。

```
In [258]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)

In [259]: data
Out[259]:
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

也可以传入一个能够完成全部这些工作的函数：

```
In [260]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[260]: 0       pig
          1       pig
          2       pig
          3       cow
          4       cow
          5       pig
          6       cow
          7       pig
          8    salmon
          Name: food, dtype: object
```

3. 替换值

replace 方法 ： 替换

利用 fillna 方法填充缺失数据可以看作替换的一种特殊情况。

替换一个值和一次性替换多个值：

```
In [263]: data = Series([1., -999., 2., -999., -1000., 3.])
          data

Out[263]: 0       1.0
          1    -999.0
          2       2.0
          3    -999.0
          4   -1000.0
          5       3.0
          dtype: float64

In [264]: data.replace(-999, np.nan)

Out[264]: 0       1.0
          1       NaN
          2       2.0
          3       NaN
          4   -1000.0
          5       3.0
          dtype: float64

In [265]: data.replace([-999, -1000], np.nan)

Out[265]: 0    1.0
          1    NaN
          2    2.0
          3    NaN
          4    NaN
          5    3.0
          dtype: float64
```

对不同的值进行不同的替换：

```
In [266]: data.replace([-999, -1000], [np.nan, 0])

Out[266]: 0    1.0
          1    NaN
          2    2.0
          3    NaN
          4    0.0
          5    3.0
          dtype: float64
```

传入的参数也可以是字典：

```
In [267]: data.replace({-999: np.nan, -1000: 0})

Out[267]: 0    1.0
          1    NaN
          2    2.0
          3    NaN
          4    0.0
          5    3.0
          dtype: float64
```

4. 重命名轴索引

轴标签有一个 map 方法：

```
In [268]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                              index=['Ohio', 'Colorado', 'New York'],
                              columns=['one', 'two', 'three', 'four'])

In [269]: data.index.map(str.upper)
Out[269]: Index(['OHIO', 'COLORADO', 'NEW YORK'], dtype='object')
```

对函数或映射进行转换，从而得到一个新对象。

将其值赋给 index，就可以对 DataFrame 进行就地修改了：

```
In [270]: data.index = data.index.map(str.upper)

In [271]: data
Out[271]:
```

	one	two	three	four
OHIO	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

要创建数据集的转换版，而不是修改原始数据，用 rename：

```
In [272]: data.rename(index=str.title, columns=str.upper)
Out[272]:
```

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

rename 结合字典型对象可以实现对部分轴标签的更新：

```
In [273]: data.rename(index={'OHIO': 'INDIANA'},
                      columns={'three': 'peekaboo'})
Out[273]:
```

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

rename 实现了复制 DataFrame 并对其索引和列标签进行赋值，就地修改某个数据集，传入 inplace=True：

```
In [275]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
          data
```

Out[275]:

	one	two	three	four
INDIANA	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

5. 离散化和面元划分

为了便于分析，连续数据常常被离散化或拆分为“面元（bin)”。

用 pandas 的 cut 函数：

```
In [276]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

In [277]: bins = [18,25,35,60,100]

In [279]: cats = pd.cut(ages, bins)
          cats

Out[279]: [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 6
          0], (25, 35]]
          Length: 12
          Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

pandas 返回的是一个特殊的 Categorical 对象，它含有一个表示不同分类名称的数组和一个为年龄数据进行标号的属性：

```
In [281]: cats.categories

Out[281]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
                        closed='right',
                        dtype='interval[int64]')

In [283]: cats.codes

Out[283]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

哪边是闭端可以通过 right=False 进行修改：

```
In [285]: pd.cut(ages, [18,26,36,61,100], right=False)

Out[285]: [[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 6
          1), [26, 36)]
          Length: 12
          Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

设置自己的面元名称：

```
In [288]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
          pd.cut(ages, bins, labels=group_names)

Out[288]: [Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, You
          ngAdult]
          Length: 12
          Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

将 labels 选项设置为一个列表或数组即可。

如果向 cut 传入的是面元的数量而不是确切的面元边界，则它会根据数据的最小值和最大值计算等长面元：

```
In [289]: data = np.random.rand(20)

In [292]: pd.cut(data, 4, precision=2)

Out[292]: [(0.0095, 0.26], (0.75, 0.99], (0.5, 0.75], (0.75, 0.99], (0.26, 0.5], ..., (0.5, 0.75], (0.
          5, 0.75], (0.26, 0.5], (0.5, 0.75], (0.26, 0.5]]
          Length: 20
          Categories (4, interval[float64]): [(0.0095, 0.26] < (0.26, 0.5] < (0.5, 0.75] < (0.75, 0.9
          9]]
```

将一些均匀分布的数据分成了四组。

qcut 函数：根据样本分位数对数据进行面元划分。

由于 qcut 使用的是样本分位数，可以得到大小基本相等的面元（而 cut 根据数据的分布情况，可能无法使各个面元中含有相同数量的数据点）。

```
In [293]: data = np.random.randn(1000)    #正态分布

In [294]: cats = pd.qcut(data ,4)    #按四分位数进行切割
          cats

Out[294]: [(-2.967, -0.687], (0.58, 3.149], (-2.967, -0.687], (-0.0153, 0.58], (-2.967, -0.687], ...,
          (0.58, 3.149], (0.58, 3.149], (-0.687, -0.0153], (-0.687, -0.0153], (-2.967, -0.687]]
          Length: 1000
          Categories (4, interval[float64]): [(-2.967, -0.687] < (-0.687, -0.0153] < (-0.0153, 0.58] <
          (0.58, 3.149]]
```

```
In [296]: pd.value_counts(cats)

Out[296]: (0.58, 3.149]        250
          (-0.0153, 0.58]      250
          (-0.687, -0.0153]    250
          (-2.967, -0.687]     250
          dtype: int64
```

设置自定义的分位数：

```
In [295]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1])

Out[295]: [(-2.967, -1.248], (1.131, 3.149], (-2.967, -1.248], (-0.0153, 1.131], (-2.967, -1.248], ...,
          (-0.0153, 1.131], (1.131, 3.149], (-1.248, -0.0153], (-1.248, -0.0153], (-2.967, -1.248]]
          Length: 1000
          Categories (4, interval[float64]): [(-2.967, -1.248] < (-1.248, -0.0153] < (-0.0153, 1.131] <
          (1.131, 3.149]]
```

在聚合和分组运算时会再次用到 cut 和 qcut 这两个离散化函数。

6.检测和过滤异常值

判断是否存在异常值（outlier)：

```
In [297]: np.random.seed(12345)

In [298]: data = DataFrame(np.random.randn(1000, 4))

In [299]: data.describe()

Out[299]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067684	0.067924	0.025598	-0.002298
std	0.998035	0.992106	1.006835	0.996794
min	-3.428254	-3.548824	-3.184377	-3.745356
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.366626	2.653656	3.260383	3.927528

找出某列中绝对值大小超过 3 的值：

```
In [300]: col = data[3]

In [302]: col[np.abs(col) > 3]

Out[302]: 97     3.927528
          305   -3.399312
          400   -3.745356
          Name: 3, dtype: float64
```

选出全部含有“超过 3 或 -3 的值”的行：

```
In [303]: data[(np.abs(data) > 3).any(1)]

Out[303]:
```

	0	1	2	3
5	-0.539741	0.476985	3.248944	-1.021228
97	-0.774363	0.552936	0.106061	3.927528
102	-0.655054	-0.565230	3.176873	0.959533
305	-2.315555	0.457246	-0.025907	-3.399312
324	0.050188	1.951312	3.260383	0.963301
400	0.146326	0.508391	-0.196713	-3.745356
499	-0.293333	-0.242459	-3.056990	1.918403
523	-3.428254	-0.296336	-0.439938	-0.867165
586	0.275144	1.179227	-3.184377	1.369891
808	-0.362528	-3.548824	1.553205	-2.186301
900	3.366626	-2.372214	0.851010	1.332846

将值限制在区间 -3 到 3 以内：

```
In [304]: data[np.abs(data) > 3] = np.sign(data) * 3

In [305]: data.describe()
```

Out[305]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067623	0.068473	0.025153	-0.002081
std	0.995485	0.990253	1.003977	0.989736
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.000000	2.653656	3.000000	3.000000

np.sign 这个 ufunc 返回的是一个由 1 和 -1 组成的数组，表示原始值的符号。

7. 排列和随机采样

numpy.random.permutation 函数：对 Series 和 DataFrame 的列排列。

```
In [315]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))

In [317]: sampler = np.random.permutation(5)
          sampler

Out[317]: array([1, 3, 4, 0, 2])
```

Permutation(5)：需要排列的轴的长度。

然后就可以在基于 ix 的索引操作或 take 函数中使用该数组了：

```
In [320]: df
```

Out[320]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

```
In [324]: df.take(sampler)
```

Out[324]:

	0	1	2	3
1	4	5	6	7
4	16	17	18	19
0	0	1	2	3
2	8	9	10	11
3	12	13	14	15

选取随机子集（非替换）：

```
In [325]: df.take(np.random.permutation(len(df))[:3])
```

Out[325]:

	0	1	2	3
0	0	1	2	3
3	12	13	14	15
2	8	9	10	11

用替换的方式产生样本：

```
In [326]: bag = np.array([5, 7, -1, 6, 4])
In [327]: sampler = np.random.randint(0, len(bag), size=10)
In [328]: sampler
Out[328]: array([3, 0, 1, 2, 2, 3, 2, 1, 2, 0])
In [329]: draws = bag.take(sampler)
In [331]: draws
Out[331]: array([ 6,  5,  7, -1, -1,  6, -1,  7, -1,  5])
```

8. 计算指标 / 哑变量

将分类变量(Categorical)转换为“哑变量矩阵(dummy matrix)”或“指标矩阵(indicator matrix)”。

```
In [332]: df = DataFrame({'key': ['b','b','a','c','a','b'],
                          'data1': range(6)})

In [333]: pd.get_dummies(df['key'])
Out[333]:
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

```
In [334]: df
Out[334]:
```

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

给 DataFrame 的列加上一个前缀，以便能够跟其他数据进行合并：

```
In [335]: dummies = pd.get_dummies(df['key'], prefix='key')
In [336]: df_with_dummy = df[['data1']].join(dummies)
In [337]: df_with_dummy
Out[337]:
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

用 get_dummies 的 prefix 参数。

DataFrame 中的某行同属于多个分类的情况，举个例子：

```
In [339]: mnames = ['movie_id', 'title', 'genres']
In [340]: vies = pd.read_table('/Users/liguanwei/Desktop/movies.dat', sep='::', header=None, names=mnames)
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: ParserWarning: Falling back t
o the 'python' engine because the 'c' engine does not support regex separators (separators >
1 char and different from '\s+' are interpreted as regex); you can avoid this warning by spec
ifying engine='python'.
  """Entry point for launching an IPython kernel.
In [341]: movies[:10]
Out[341]:
```

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

要为每个genre添加指标变量就需要做一些数据规整操作，构建多成员指标变量：

```
In [342]: genre_iter = (set(x.split('|')) for x in movies.genres)

In [343]: genres = sorted(set.union(*genre_iter))

In [346]: dummies = pd.DataFrame(np.zeros((len(movies), len(genres))), columns=genres)

In [351]: for i, gen in enumerate(movies.genres):
              dummies.loc[i,gen.split('|')] = 1

In [352]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [355]: movies_windic.iloc[0]

Out[355]: movie_id                                1
          title                     Toy Story (1995)
          genres           Animation|Children's|Comedy
          Genre_Action                            0
          Genre_Adventure                         0
          Genre_Animation                         1
          Genre_Children's                        1
          Genre_Comedy                            1
          Genre_Crime                             0
          Genre_Documentary                       0
          Genre_Drama                             0
          Genre_Fantasy                           0
          Genre_Film-Noir                         0
          Genre_Horror                            0
          Genre_Musical                           0
```

对于很大的数据，这种方式会变得非常慢，需要编写一个能够利用 DataFrame 内部机制的更低级的函数：

```
In [356]: values = np.random.rand(10)

In [357]: values

Out[357]: array([0.90953758, 0.38441329, 0.08127486, 0.95663486, 0.04387138,
                 0.14476865, 0.11766477, 0.13018305, 0.62275769, 0.08497892])

In [358]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [359]: pd.get_dummies(pd.cut(values, bins))

Out[359]:
```

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	0	0	0	1
4	1	0	0	0	0
5	1	0	0	0	0
6	1	0	0	0	0
7	1	0	0	0	0
8	0	0	0	1	0
9	1	0	0	0	0

用 get_dummies 和 cut 之类的离散化函数。

字符串操作

1. 字符串对象方法

Python 字符串对象的内置方法：

```
In [361]: val = 'a,b,guido'
          val.split(',')
Out[361]: ['a', 'b', 'guido']

In [362]: pieces = [x.strip() for x in val.split(',')]
          pieces
Out[362]: ['a', 'b', 'guido']

In [363]: first, second, third = pieces
          first + '::' + second + '::' + third
Out[363]: 'a::b::guido'

In [364]: '::'.join(pieces)
Out[364]: 'a::b::guido'
```

find 找不到返回 -1，index 找不到引发一个异常

```
In [365]: 'guido' in val
Out[365]: True

In [366]: val.index(',')
Out[366]: 1

In [367]: val.find(':')
Out[367]: -1

In [368]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-368-2c016e7367ac> in <module>()
----> 1 val.index(':')

ValueError: substring not found
```

传入空字符串常常用于删除模式：

```
In [369]: val.count(',')
Out[369]: 2

In [370]: val.replace(',', '::')
Out[370]: 'a::b::guido'

In [372]: val.replace(',', ' ')
Out[372]: 'a b guido'
```

2. 正则表达式(regex)

提供了一种灵活的在文本中搜索或匹配字符串模式的方式。python 内置的 re 模块负责对字符串应用正则表达式。

re 模块的函数分为三个大类：模式匹配、替换、拆分。

```
In [373]: import re

In [374]: text = "foo    bar\t baz  \tqux"

In [375]: re.split('\s+', text)
Out[375]: ['foo', 'bar', 'baz', 'qux']

In [376]: regex = re.compile('\s+')

In [377]: regex.split(text)
Out[377]: ['foo', 'bar', 'baz', 'qux']
```

描述一个或多个空白符的 regex 是 \s+。

调用 re.split('\s+', text)时，正则表达式会先被编译，然后再在 text 上调用其 split 方法。

可以用 re.compile 自己编译一个 regex，以得到一个可重用的 regex 对象，如上所示。如果打算对许多字符串应用同一条正则表达式，强烈建议通过这种方法，可以节省大量的 CPU 时间。

得到匹配 regex 的所有模式：

```
In [379]: regex.findall(text)
Out[379]: ['    ', '\t ', '  \t']
```

- findall：返回字符串中所有的匹配项。
- search：只返回第一个匹配项。
- match：只匹配字符串的首部。

```
In [381]: text = """Dave dave@google.com
          Steve steve@gmail.com
          Rob rob@gmail.com
          Ryan ryan@yahoo.com
          """

          pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
          regex = re.compile(pattern, flags=re.IGNORECASE)    #re.IGNORECASE 使正则表达式对大小写不敏感

In [382]: regex.findall(text)
Out[382]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']

In [384]: m = regex.search(text)
          m
Out[384]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [388]: text[m.start():m.end()]
Out[388]: 'dave@google.com'

In [387]: print(regex.match(text))
          None
```

sub 方法：将匹配到的模式替换为指定字符串，并返回所得到的新字符串。

```
In [389]: print(regex.sub('REDACTED', text))

Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

不仅想找出电子邮件地址，还想将各个地址分为 3 个部分，只需将待分段的模式的各部分用圆括号包起来：

```
In [392]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
          regex = re.compile(pattern, flags=re.IGNORECASE)

In [394]: m = regex.match('wesm@bright.net')
          m.groups()

Out[394]: ('wesm', 'bright', 'net')
```

通过 groups 方法返回一个由模式各段组成的元组。

对于带有分组功能的模式，findall 会返回一个元组列表：

```
In [395]: regex.findall(text)

Out[395]: [('dave', 'google', 'com'),
           ('steve', 'gmail', 'com'),
           ('rob', 'gmail', 'com'),
           ('ryan', 'yahoo', 'com')]
```

sub 还能通过诸如\1, \2 之类的特殊符号访问各匹配项中的分组：

```
In [396]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))

Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

为各个匹配分组加上一个名称，由这种正则表达式所产生的匹配对象可以得到一个简单易用的带有分组名称的字典：

```
In [397]: regex = re.compile(r"""
              (?P<username>[A-Z0-9._%+-]+)
              @
              (?P<domain>[A-Z0-9.-]+)
              \.
              (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)

In [398]: m = regex.match('wesm@bright.net')
          m.groupdict()

Out[398]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

3. pandas 中矢量化的字符串函数

通过 data.map，所有字符串和正则表达式方法都能被应用于各个值，但如存在 NA 就会报错，为了解决这个问题，Series 有一些能够跳过 NA 值的字符串操作方法，通过 Series 的 str 属性即可访问这些方法：

```
In [401]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
                  'Rob': 'rob@gmail.com', 'Wes': np.nan}
          data = Series(data)
          data

Out[401]: Dave     dave@google.com
          Rob        rob@gmail.com
          Steve    steve@gmail.com
          Wes                  NaN
          dtype: object

In [402]: data.isnull()

Out[402]: Dave     False
          Rob      False
          Steve    False
          Wes       True
          dtype: bool

In [403]: data.str.contains('gmail')

Out[403]: Dave     False
          Rob       True
          Steve     True
          Wes        NaN
          dtype: object
```

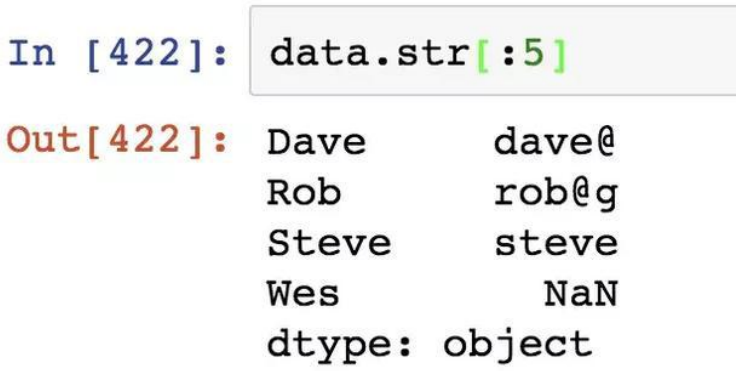
也可以用正则表达式：

```
In [404]: pattern

Out[404]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [405]: data.str.findall(pattern, flags=re.IGNORECASE)

Out[405]: Dave     [(dave, google, com)]
          Rob        [(rob, gmail, com)]
          Steve    [(steve, gmail, com)]
          Wes                        NaN
          dtype: object
```

实现矢量化的元素获取操作，对 str.get/str 属性上使用索引：

```
In [419]: matches = data.str.match(pattern, flags=re.IGNORECASE)
          matches

Out[419]: Dave     True
          Rob      True
          Steve    True
          Wes       NaN
          dtype: object

In [420]: matches.str.get(1)

Out[420]: Dave    NaN
          Rob     NaN
          Steve   NaN
          Wes     NaN
          dtype: float64

In [421]: matches.str[0]

Out[421]: Dave    NaN
          Rob     NaN
          Steve   NaN
          Wes     NaN
          dtype: float64
```

对字符串进行子串截取：



数据聚合与分组运算

对数据集进行分组并对各组应用一个函数。

在将数据集准备好之后，通常的任务就是**计算分组统计**或**生成透视表**。pandas 提供了一个灵活高效的 gruopby 功能，对数据集进行切片、切块、摘要等操作。

用 python 和 pandas 强大的表达能力可以执行复杂的多的分组运算：利用任何可以接受 pandas 对象或 NumPy 数组的函数。

GroupBy 技术

分组运算：split（拆分）--apply（应用）--combine（合并）。

分组键的形式：

- 列表或数组，其长度与待分组的轴一样。

- 表示 DataFrame 某个列名的值。
- 字典或 Series，给出待分组轴上的值与分组名之间的对应关系。
- 函数，用于处理轴索引或索引中的各个标签。

```
In [3]: df = DataFrame({'key1': ['a','a','b','b','a'],
                        'key2': ['one','two','one','two','one'],
                        'data1':np.random.randn(5),
                        'data2':np.random.randn(5)})
        df
```

Out[3]:

	data1	data2	key1	key2
0	-0.022058	-0.950728	a	one
1	0.836629	-1.254127	a	two
2	-0.183069	0.378230	b	one
3	-0.858412	0.771402	b	two
4	-0.923308	-0.411253	a	one

```
In [4]: grouped = df['data1'].groupby(df['key1'])
        grouped
Out[4]: <pandas.core.groupby.SeriesGroupBy object at 0x11113f4a8>
```

访问 data1，并根据 key1 调用 groupby。

变量 grouped 是一个 GroupBy 对象，它实际上还没有进行任何计算，只是含有一些有关分组键 df['key1']的中间数据。

例如，调用 GroupBy 的 mean 方法来计算分组平均值：

```
In [5]: grouped.mean()
Out[5]: key1
        a   -0.036246
        b   -0.520740
        Name: data1, dtype: float64

In [6]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()
        means
Out[6]: key1  key2
        a     one    -0.472683
              two     0.836629
        b     one    -0.183069
              two    -0.858412
        Name: data1, dtype: float64
```

Series 根据分组键进行了聚合，产生了一个新的 Series，其索引为 key1 列中的唯一值。

通过两个键对数据进行了分组后，得到的 Series 具有一个层次化索引：

```
In [7]: means.unstack()
```

Out[7]:

key1	key2 one	two
a	-0.472683	0.836629
b	-0.183069	-0.858412

分组键可以是任何长度适当的数组：

```
In [8]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
        years = np.array([2005, 2005, 2006, 2005, 2006])

In [9]: df['data1'].groupby([states, years]).mean()
Out[9]: California  2005    0.836629
                    2006   -0.183069
        Ohio        2005   -0.440235
                    2006   -0.923308
        Name: data1, dtype: float64
```

将列名用作分组键：

```
In [10]: df.groupby('key1').mean()
```

Out[10]:

key1	data1	data2
a	-0.036246	-0.872036
b	-0.520740	0.574816

```
In [11]: df.groupby(['key1','key2']).mean()
```

Out[11]:

key1	key2	data1	data2
a	one	-0.472683	-0.680991
	two	0.836629	-1.254127
b	one	-0.183069	0.378230
	two	-0.858412	0.771402

GroupBy 的 size 方法返回一个含有分组大小的 Series：

```
df.groupby(['key1','key2']).size()
```

```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

1. 对分组进行迭代

GroupBy 对象支持迭代，可以产生一组二元元组（由分组名和数据块组成）。

```
In [15]: for name, group in df.groupby('key1'):
             print(name)
             print(group)
```

```
a
      data1     data2 key1 key2
0 -0.022058 -0.950728    a  one
1  0.836629 -1.254127    a  two
4 -0.923308 -0.411253    a  one
b
      data1     data2 key1 key2
2 -0.183069  0.378230    b  one
3 -0.858412  0.771402    b  two
```

```
In [16]: for (k1, k2), group in df.groupby(['key1', 'key2']):
             print(k1, k2)
             print(group)
```

```
a one
      data1     data2 key1 key2
0 -0.022058 -0.950728    a  one
4 -0.923308 -0.411253    a  one
a two
      data1     data2 key1 key2
1  0.836629 -1.254127    a  two
b one
      data1    data2 key1 key2
2 -0.183069  0.37823    b  one
b two
      data1     data2 key1 key2
3 -0.858412  0.771402    b  two
```

对于多重键，元组的第一个元素将会是由键值组成的元组。

对数据片段进行操作，如将这些数据片段做成一个字典：

```
In [17]: pieces = dict(list(df.groupby('key1')))
         pieces['b']
```

Out[17]:

	data1	data2	key1	key2
2	-0.183069	0.378230	b	one
3	-0.858412	0.771402	b	two

groupby 默认在 axis=0 上进行分组，通过设置可以在其它任何轴上进行分组，如可以根据 dtype 对列进行分组：

```
In [18]: df.dtypes
Out[18]: data1    float64
         data2    float64
         key1      object
         key2      object
         dtype: object

In [19]: grouped = df.groupby(df.dtypes, axis=1)

In [20]: dict(list(grouped))
Out[20]: {dtype('float64'):       data1     data2
          0 -0.022058 -0.950728
          1  0.836629 -1.254127
          2 -0.183069  0.378230
          3 -0.858412  0.771402
          4 -0.923308 -0.411253, dtype('O'):   key1 key2
          0    a  one
          1    a  two
          2    b  one
          3    b  two
          4    a  one}
```

2. 选取一个或一组列

对于由 DataFrame 产生的 GroupBy 对象，用一个或一组（单个字符串或字符串数组）列名对其进行索引，就能实现选取部分列进行聚合的目的：

```
In [19]: df.groupby('key1')['data1']
         df.groupby('key1')[['data2']]
Out[19]: <pandas.core.groupby.DataFrameGroupBy object at 0x10d0e83c8>

In [20]: df['data1'].groupby(df['key1'])
         df[['data2']].groupby(df['key1'])
Out[20]: <pandas.core.groupby.DataFrameGroupBy object at 0x10d1b2208>
```

例如，对部分列进行聚合：计算 data2 列的平均值并以 DataFrame 形式得到结果：

```
In [21]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[21]:
```

		data2
key1	key2	
a	one	0.692382
	two	-1.221548
b	one	0.130620
	two	1.130298

返回一个已分组的 DataFrame（传入的是列表或数组）或 Series（传入的是标量形式的单个列名）：

```
In [22]: s_grouped = df.groupby(['key1', 'key2'])['data2']
         s_grouped
Out[22]: <pandas.core.groupby.SeriesGroupBy object at 0x10d1b26d8>

In [23]: s_grouped.mean()
Out[23]: key1  key2
         a     one      0.692382
               two     -1.221548
         b     one      0.130620
               two      1.130298
         Name: data2, dtype: float64
```

3. 通过字典或 Series 进行分组

除数组以外，分组信息还可以其他形式存在

```
In [23]: people = DataFrame(np.random.randn(5,5),
                            columns=['a','b','c','d','e'],
                            index=['Joe','Steve','Wes','Jim','Travis'])

In [25]: people.loc[2:3,['b','c']] = np.nan    #添加几个NA值，2:3 包括第二行 但不包括第三行

In [27]: people
```

Out[27]:

	a	b	c	d	e
Joe	-0.862212	-0.664210	0.373806	0.456015	-1.357138
Steve	-0.206403	-0.867255	0.860986	0.943550	0.326979
Wes	-1.821028	NaN	NaN	-0.126827	-2.517637
Jim	2.219069	0.145352	-1.877708	0.980218	0.638752
Travis	0.948515	-0.909773	0.187461	-0.520768	-0.406566

根据分组计算列的 sum：

```
In [28]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
                    'd': 'blue', 'e': 'red', 'f': 'orange'}

In [30]: by_column = people.groupby(mapping, axis=1)
         by_column.sum()
```

Out[30]:

	blue	red
Joe	0.829821	-2.883560
Steve	1.804536	-0.746679
Wes	-0.126827	-4.338665
Jim	-0.897490	3.003173
Travis	-0.333307	-0.367824

将 mapping 这个字典传给 groupby 即可。

用 Series 作为分组键：

```
In [31]: map_series = Series(mapping)
         map_series
Out[31]: a       red
         b       red
         c      blue
         d      blue
         e       red
         f    orange
         dtype: object
```

```
In [33]: people.groupby(map_series, axis=1).sum()
```

Out[33]:

	blue	red
Joe	0.829821	-2.883560
Steve	1.804536	-0.746679
Wes	-0.126827	-4.338665
Jim	-0.897490	3.003173
Travis	-0.333307	-0.367824

这里 Series 可以被看做一个固定大小的映射。pandas 会检查 Series 以确保其索引根分组轴是对齐的。

4. 通过函数进行分组

任何被当做分组键的函数都会在各个索引值上被调用一次，其返回值就会被用作分组名称。

```
In [34]: people.groupby(len).sum()
```

Out[34]:

	a	b	c	d	e
3	-0.464172	-0.518858	-1.503902	1.309406	-3.236022
5	-0.206403	-0.867255	0.860986	0.943550	0.326979
6	0.948515	-0.909773	0.187461	-0.520768	-0.406566

将函数根数组、列表、字典、Series 混合使用（任何东西最终都会被转换为数组）：

```
In [39]: key_list = ['one','one','one','two','two']
```

```
In [40]: people.groupby([len, key_list]).min()
```

Out[40]:

		a	b	c	d	e
3	one	-1.821028	-0.664210	0.373806	-0.126827	-2.517637
	two	2.219069	0.145352	-1.877708	0.980218	0.638752
5	one	-0.206403	-0.867255	0.860986	0.943550	0.326979
6	two	0.948515	-0.909773	0.187461	-0.520768	-0.406566

Key_list 和人名对应，再在相同长度的对应一列里选 min 的值。

5. 根据索引级别分组

层次化索引数据集通过 level 关键字传入级别编号或名称：

```
In [41]: columns = pd.MultiIndex.from_arrays([['US','US','US','JP','JP'],
                                              [1,3,5,1,3]], names=['cty', 'tenor'])
```

```
In [42]: hier_df = DataFrame(np.random.randn(4, 5), columns=columns)
         hier_df
```

Out[42]:

cty	US			JP	
tenor	1	3	5	1	3
0	1.462119	1.692673	-1.128897	0.599128	-0.197796
1	-0.462765	-0.669173	0.886197	1.464755	-0.067525
2	-1.235286	-1.122775	0.442953	-1.254059	0.941880
3	-0.272517	-1.185479	-1.563661	0.998094	-2.154077

```
In [43]: hier_df.groupby(level='cty', axis=1).count()
```

Out[43]:

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3