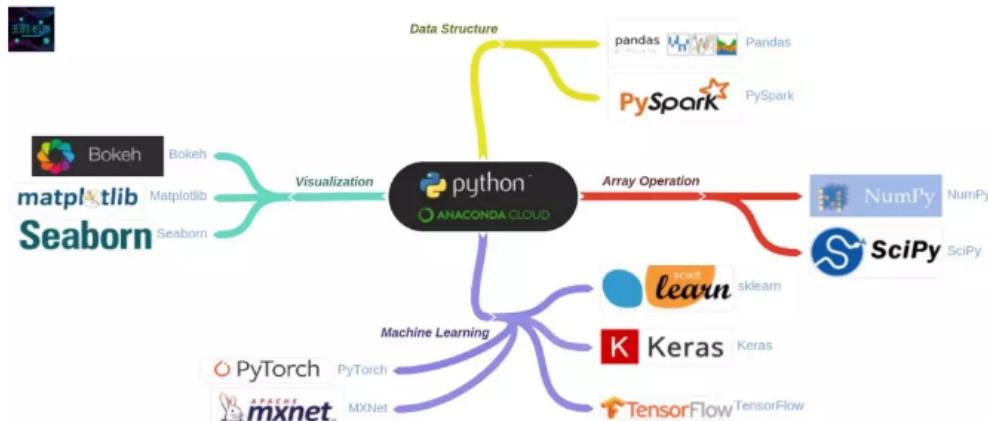


盘一盘 Python 系列 4 - Pandas (下)

From:王圣元 王的机器 4/17



全文共 14270 字, 73 幅图或表,
预计阅读时间 36 分钟。

0 引言 <

本文是 Python 系列的第七篇

- [Python 入门篇 \(上\)](#)
- [Python 入门篇 \(下\)](#)
- [数组计算之 NumPy \(上\)](#)
- [数组计算之 NumPy \(下\)](#)
- [科学计算之 SciPy](#)
- [数据结构之 Pandas \(上\)](#)
- [数据结构之 Pandas \(下\)](#)
- [基本可视化之 Matplotlib](#)
- [统计可视化之 Seaborn](#)
- [交互可视化之 Bokeh](#)

- 炫酷可视化之 PyEcharts
- 机器学习之 Sklearn
- 深度学习之 TensorFlow
- 深度学习之 Keras
- 深度学习之 PyTorch
- 深度学习之 MXnet

接着[上篇](#)继续后面三个章节

- 数据表的合并和连接
- 数据表的重塑和透视
- 数据表的分组和整合

目录

1

数据表的创建

1 维 Series, 2 维 DataFrame, 3 维 Panel

2

数据表的存载

保存和加载 (csv 和 excel 格式)

3

数据表的获取

索引和切片 (at, iat, loc, iloc), 多层索引

4

数据表的合并和连接

按“键”来合并, 按“轴”来连接

5

数据表的重塑和透视

用 stack, unstack 来重塑, 用 pivot, melt 来透视

6

数据表的分组和整合

数据分析的大杀器: **split-apply-combine**



4

数据表的合并和连接

数据表可以按「键」合并，用 `merge` 函数；可以按「轴」来连接，用 `concat` 函数。

4.1 合并

按键 (key) 合并可以分「单键合并」和「多键合并」。

单键合并

单键合并用 `merge` 函数，语法如下：

```
pd.merge( df1, df2, how=s, on=c )
```

`c` 是 `df1` 和 `df2` 共有的一栏，合并方式 (`how=s`) 有四种：

1. 左连接 (left join): 合并之后显示 `df1` 的所有行
2. 右连接 (right join): 合并之后显示 `df2` 的所有行
3. 外连接 (outer join): 合并 `df1` 和 `df2` 共有的所有行
4. 内连接 (inner join): 合并所有行 (默认情况)

首先创建两个 DataFrame：

- `df_price`: 4 天的价格 (2019-01-01 到 2019-01-04)
- `df_volume`: 5 天的交易量 (2019-01-02 到 2019-01-06)

```
1 df_price = pd.DataFrame( {'Date': pd.date_range('2019-1-1', periods=4),
2                               'Adj Close': [24.42, 25.00, 25.25, 25.64]})  
3 df_price
```

	Date	Adj Close
0	2019-01-01	24.42
1	2019-01-02	25.00
2	2019-01-03	25.25
3	2019-01-04	25.64

```
1 df_volume = pd.DataFrame( {'Date': pd.date_range('2019-1-2', periods=5),
2                               'Volume' : [56081400, 99455500, 83028700, 100  
3 df_volume
```

	Date	Volume
0	2019-01-02	56081400
1	2019-01-03	99455500
2	2019-01-04	83028700
3	2019-01-05	100234000
4	2019-01-06	73829000

接下来用 `df_price` 和 `df_volume` 展示四种合并。

left join

```
1 pd.merge( df_price, df_volume, how='left' )
```

	Date	Adj Close	Volume
0	2019-01-01	24.42	NaN
1	2019-01-02	25.00	56081400.0
2	2019-01-03	25.25	99455500.0
3	2019-01-04	25.64	83028700.0

按 `df_price` 里 Date 栏里的值来合并数据

- `df_volume` 里 Date 栏里没有 2019-01-01，因此 Volume 为 NaN
- `df_volume` 里 Date 栏里的 2019-01-05 和 2019-01-06 不在 `df_price` 里 Date 栏，因此丢弃

right join

```
1 pd.merge( df_price, df_volume, how='right' )
```

	Date	Adj Close	Volume
0	2019-01-02	25.00	56081400
1	2019-01-03	25.25	99455500
2	2019-01-04	25.64	83028700
3	2019-01-05	NaN	100234000
4	2019-01-06	NaN	73829000

按 `df_volume` 里 Date 栏里的值来合并数据

- `df_price` 里 Date 栏里没有 2019-01-05 和 2019-01-06，因此 Adj Close 为 NaN
- `df_price` 里 Date 栏里的 2019-01-01 不在 `df_volume` 里 Date 栏，因此丢弃

outer join

```
1 pd.merge( df_price, df_volume, how='outer' )
```

	Date	Adj Close	Volume
0	2019-01-01	24.42	NaN
1	2019-01-02	25.00	56081400.0
2	2019-01-03	25.25	99455500.0
3	2019-01-04	25.64	83028700.0
4	2019-01-05	NaN	100234000.0
5	2019-01-06	NaN	73829000.0

按 `df_price` 和 `df_volume` 里 Date 栏里的**所有值**来合并数据

- `df_price` 里 Date 栏里没有 2019-01-05 和 2019-01-06，因此 Adj Close 为 NaN
- `df_volume` 里 Date 栏里没有 2019-01-01，因此 Volume 为 NaN

inner join

```
1 pd.merge( df_price, df_volume, how='inner' )
```

	Date	Adj Close	Volume
0	2019-01-02	25.00	56081400
1	2019-01-03	25.25	99455500
2	2019-01-04	25.64	83028700

按 `df_price` 和 `df_volume` 里 Date 栏里的**共有值**来合并数据

- `df_price` 里 Date 栏里的 2019-01-01 不在 `df_volume` 里 Date 栏，因此丢弃
- `df_volume` 里 Date 栏里的 2019-01-05 和 2019-01-06 不在 `df_price` 里 Date 栏，因此丢弃

多键合并

多键合并用的语法和单键合并一样，只不过 `on=c` 中的 c 是多栏。

```
pd.merge( df1, df2, how=s, on=c )
```

首先创建两个 DataFrame：

- portfolio1：3 比产品 FX Option, FX Swap 和 IR Option 的数量
- portfolio2：4 比产品 FX Option (重复名称), FX Swap 和 IR Swap 的数量

```
1 portfolio1 = pd.DataFrame({'Asset': ['FX', 'FX', 'IR'],
2                             'Instrument': ['Option', 'Swap', 'Option'],
```

```
3                                     'Number': [1, 2, 3]})  
4 portfolio1
```

	Asset	Instrument	Number
0	FX	Option	1
1	FX	Swap	2
2	IR	Option	3

```
1 portfolio2 = pd.DataFrame({'Asset': ['FX', 'FX', 'FX', 'IR'],  
2                             'Instrument': ['Option', 'Option', 'Swap', 'Swap'],  
3                             'Number': [4, 5, 6, 7]})  
4 portfolio2
```

	Asset	Instrument	Number
0	FX	Option	4
1	FX	Option	5
2	FX	Swap	6
3	IR	Swap	7

在 'Asset' 和 'Instrument' 两个键上做外合并。

```
1 pd.merge( portfolio1, portfolio2,  
2           on=['Asset','Instrument'],  
3           how='outer')
```

	Asset	Instrument	Number_x	Number_y
0	FX	Option	1.0	4.0
1	FX	Option	1.0	5.0
2	FX	Swap	2.0	6.0
3	IR	Option	3.0	NaN
4	IR	Swap	NaN	7.0

df1 和 df2 中两个键都有 FX Option 和 FX Swap，因此可以合并它们中 number 那栏。

- df1 中有 IR Option 而 df2 中没有，因此 Number_y 栏下的值为 NaN
- df2 中有 IR Swap 而 df1 中没有，因此 Number_x 栏下的值为 NaN

当 df1 和 df2 有两个相同的列 (Asset 和 Instrument) 时，单单只对一列 (Asset) 做合并产出的 DataFrame 会有另一列 (Instrument) 重复的名称。这时 `merge` 函数给重复的名称加个后缀 `_x`, `_y` 等等。

```
1 pd.merge( portfolio1, portfolio2,
2           on='Asset' )
```

	Asset	Instrument_x	Number_x	Instrument_y	Number_y
0	FX	Option	1	Option	4
1	FX	Option	1	Option	5
2	FX	Option	1	Swap	6
3	FX	Swap	2	Option	4
4	FX	Swap	2	Option	5
5	FX	Swap	2	Swap	6
6	IR	Option	3	Swap	7

当没设定 `merge` 函数里参数 `how` 时，默认为 `inner` (内合并)。在 Asset 列下，df1 有 2 个 FX 和 1 个 IR，df2 有 3 个 FX 和 1 个 IR，内合并完有 8 行 ($2 \times 3 + 1 \times 1$)。

如果觉得后缀 `_x`, `_y` 没有什么具体含义时，可以设定 `suffixes` 来改后缀。比如 df1 和 df2 存储的是 portoflio1 和 portfolio2 的产品信息，那么将后缀该成 '1' 和 '2' 更贴切。

```
1 pd.merge( portfolio1, portfolio2,
2           on='Asset',
3           suffixes=['1', '2'])
```

	Asset	Instrument1	Number1	Instrument2	Number2
0	FX	Option	1	Option	4
1	FX	Option	1	Option	5
2	FX	Option	1	Swap	6
3	FX	Swap	2	Option	4
4	FX	Swap	2	Option	5
5	FX	Swap	2	Swap	6
6	IR	Option	3	Swap	7

Numpy 数组可相互连接，用 `np.concatenate`；同理，Series 也可相互连接，DataFrame 也可相互连接，用 `pd.concat`。

连接 Series

在 `concat` 函数也可设定参数 `axis`,

- `axis = 0` (默认)，沿着轴 0 (行) 连接，得到一个更长的 Series
- `axis = 1`，沿着轴 1 (列) 连接，得到一个 DataFrame

被连接的 Series 它们的 index 可以重复 (overlapping)，也可以不同。

overlapping index

先定义三个 Series，它们的 index 各不同。

```
1 s1 = pd.Series([0, 1], index=['a', 'b'])
2 s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
3 s3 = pd.Series([5, 6], index=['f', 'g'])
```

沿着「轴 0」连接得到一个更长的 Series。

```
1 pd.concat([s1, s2, s3])
```

```
a 0
b 1
c 2
d 3
e 4
f 5
g 6
dtype: int64
```

沿着「轴 1」连接得到一个 DataFrame。

```
1 pd.concat([s1, s2, s3], axis=1)
```

	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

non-overlapping index

将 s1 和 s3 沿「轴 0」连接来创建 s4，这样 s4 和 s1 的 index 是有重复的。

```
1 s4 = pd.concat([s1, s3])
2 s4
```

```
a 0
b 1
f 5
g 6
dtype: int64
```

将 s1 和 s4 沿「轴 1」内连接 (即只连接它们共有 index 对应的值)

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

	0	1
a	0	0
b	1	1

hierarchical index

最后还可以将 n 个 Series 沿「轴 0」连接起来，再赋予 3 个 keys 创建多层 Series。

```
1 pd.concat( [s1, s1, s3],  
2             keys=['one','two','three'])
```

```
one  a 0  
      b 1  
two  a 0  
      b 1  
three f 5  
      g 6  
dtype: int64
```

连接 DataFrame

连接 DataFrame 的逻辑和连接 Series 的一模一样。

沿着行连接 (axis = 0)

先创建两个 DataFrame, df1 和 df2。

```
1 df1 = pd.DataFrame( np.arange(12).reshape(3,4),  
2                      columns=['a','b','c','d'])  
3 df1
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
1 df2 = pd.DataFrame( np.arange(6).reshape(2,3),  
2                      columns=['b','d','a'])  
3 df2
```

	b	d	a
0	0	1	2
1	3	4	5

沿着行连接分两步

1. 先把 df1 和 df2 **列标签**补齐
2. 再把 df1 和 df2 **纵向**连起来

```
1 pd.concat( [df1, df2] )
```

	a	b	c	d
0	0	1	2.0	3
1	4	5	6.0	7
2	8	9	10.0	11
0	2	0	NaN	1
1	5	3	NaN	4

得到的 DataFrame 的 index = [0,1,2,0,1], 有重复值。如果 index 不包含重要信息 (如上例)，可以将 ignore_index 设置为 True，这样就得到默认的 index 值了。

```
1 pd.concat( [df1, df2], ignore_index=True )
```

	a	b	c	d
0	0	1	2.0	3
1	4	5	6.0	7
2	8	9	10.0	11
3	2	0	NaN	1
4	5	3	NaN	4

沿着列连接 (axis = 1)

先创建两个 DataFrame，df1 和 df2。

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

	one	two
a	0	1
b	2	3
c	4	5

```
1 df2 = pd.DataFrame( 5 + np.arange(4).reshape(2,2),
2                         index=['a', 'c'],
3                         columns=['three', 'four'])
4 df2
```

	three	four
a	5	6
c	7	8

沿着列连接分两步

1. 先把 df1 和 df2 行标签补齐
2. 再把 df1 和 df2 横向连起来

```
1 pd.concat( [df1, df2], axis=1 )
```

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



5 数据表的重塑和透视

重塑 (reshape) 和透视 (pivot) 两个操作只改变数据表的布局 (layout):

- 重塑用 `stack` 和 `unstack` 函数 (互为逆转操作)
- 透视用 `pivot` 和 `melt` 函数 (互为逆转操作)

在【[数据结构之 Pandas \(上\)](#)】提到过，DataFrame 和「多层索引的 Series」其实维度是一样，只是展示形式不同。而重塑就是通过改变数据表里面的「行索引」和「列索引」来改变展示形式。

- **列索引 → 行索引**，用 `stack` 函数
- **行索引 → 列索引**，用 `unstack` 函数

单层 DataFrame

创建 DataFrame df (1 层行索引，1 层列索引)

```

1 symbol = ['JD', 'AAPL']
2 data = {'行业': ['电商', '科技'],
3          '价格': [25.95, 172.97],
4          '交易量': [27113291, 18913154]}
5 df = pd.DataFrame( data, index=symbol )
6 df.columns.name = '特征'
7 df.index.name = '代号'
8 df

```

特征	行业	价格	交易量
代号			
JD	电商	25.95	27113291
AAPL	科技	172.97	18913154

从上表中可知：

- **行索引** = [JD, AAPL]，名称是**代号**
- **列索引** = [行业, 价格, 交易量]，名称是**特征**

stack: 列索引 → 行索引

列索引 (特征) 变成了**行索引**，原来的 DataFrame df 变成了两层 Series (第一层索引是

代号, 第二层索引是特征)。

```
1 c2i_Series = df.stack()  
2 c2i_Series
```

```
代号 特征  
JD 行业 电商  
      价格 25.95  
      交易量 27113291  
AAPL 行业 科技  
      价格 172.97  
      交易量 18913154  
dtype: object
```

思考: 变成行索引的特征和原来行索引的代号之间的层次是怎么决定的? 好像特征更靠内一点, 代号更靠外一点。

unstack: 行索引 → 列索引

行索引(代号)变成了列索引, 原来的 DataFrame df 也变成了两层 Series (第一层索引是特征, 第二层索引是代号)。

```
1 i2c_Series = df.unstack()  
2 i2c_Series
```

```
特征 代号  
行业 JD 电商  
      AAPL 科技  
价格 JD 25.95  
      AAPL 172.97  
交易量 JD 27113291  
      AAPL 18913154  
dtype: object
```

思考: 变成列索引的特征和原来列索引的代号之间的层次是怎么决定的? 这时好像代号更靠内一点, 特征更靠外一点。

规律总结

对 df 做 `stack` 和 `unstack` 都得到了「两层 Series」，但是索引的层次不同，那么在背后的规律是什么？首先我们先来看看两个「两层 Series」的 index 包含哪些信息 (以及 df 的 index 和 columns)。

```
1 df.index, df.columns
```

```
(Index(['JD', 'AAPL'], dtype='object', name='代号'),  
 Index(['行业', '价格', '交易量'], dtype='object', name='特征'))
```

```
1 c2i_Series.index
```

```
MultiIndex(levels=[[['JD', 'AAPL'], ['行业', '价格', '交易量']],  
                   [[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]]],  
                   names=['代号', '特征'])
```

```
1 i2c_Series.index
```

```
MultiIndex(levels=[[['行业', '价格', '交易量'], ['JD', 'AAPL']],  
                   [[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]]],  
                   names=['特征', '代号'])
```

定义

- **r** = [JD, AAPL]，名称是**代号**
- **c** = [行业, 价格, 交易量]，名称是**特征**

那么

- df 的**行索引** = **r**
- df 的**列索引** = **c**
- **c2i_Series** 的索引 = [**r**, **c**]
- **i2c_Series** 的索引 = [**c**, **r**]

现在可以总结规律：

1. 当用 `stack` 将 df 变成 **c2i_Series** 时，df 的**列索引 c** 加在其**行索引 r** 后面得到 [**r**, **c**] 做为 **c2i_Series** 的**多层索引**

2. 当用 `unstack` 将 df 变成 i2c_Series 时, df 的行索引 r 加在其列索引 c 后面得到 [c, r] 做为 i2c_Series 的多层索引

基于层和名称来 unstack

对于多层索引的 Series, `unstack` 哪一层有两种方法来确定:

1. 基于层 (level-based)
2. 基于名称 (name-based)

拿 c2i_Series 举例 (读者也可以尝试 i2c_Series):

```
代号 特征
JD 行业 电商
    价格 25.95
    交易量 27113291
AAPL 行业 科技
    价格 172.97
    交易量 18913154
dtype: object
```

1. 基于层来 `unstack()` 时, 没有填层数, 默认为最后一层。

```
1 c2i_Series.unstack()
```

特征	行业	价格	交易量
代号			
JD	电商	25.95	27113291
AAPL	科技	172.97	18913154

c2i_Series 的最后一层 (看上面它的 MultiIndex) 就是 [行业, 价格, 交易量], 从行索引转成列索引得到上面的 DataFrame。

2. 基于层来 `unstack()` 时, 选择第一层 (参数放 0)

```
1 c2i_Series.unstack(0)
```

代号	JD	AAPL
特征		
行业	电商	科技
价格	25.95	172.97
交易量	27113291	18913154

c2i_Series 的第一层 (看上面它的 MultiIndex) 就是 [JD, AAPL]，从行索引转成列索引得到上面的 DataFrame。

3. 基于名称来 unstack

```
1 c2i_Series.unstack('代号')
```

代号	JD	AAPL
特征		
行业	电商	科技
价格	25.95	172.97
交易量	27113291	18913154

c2i_Series 的代号层 (看上面它的 MultiIndex) 就是 [JD, AAPL]，从行索引转成列索引得到上面的 DataFrame。

多层 DataFrame

创建 DataFrame df (2 层行索引, 1 层列索引)

```
1 data = [ ['电商', 101550, 176.92],
2           ['电商', 175336, 25.95],
3           ['金融', 60348, 41.79],
4           ['金融', 36600, 196.00] ]
5
6 midx = pd.MultiIndex( levels=[[ '中国', '美国'],
7                         ['BABA', 'JD', 'GS', 'MS']],
8                         labels=[[0,0,1,1],[0,1,2,3]],
```

```
9             names = ['地区', '代号'])
10
11 mcol = pd.Index(['行业', '雇员', '价格'], name='特征')
12
13 df = pd.DataFrame( data, index=midx, columns=mcol )
14 df
```

	特征	行业	雇员	价格
地区	代号			
中国	BABA	电商	101550	176.92
	JD	电商	175336	25.95
美国	GS	金融	60348	41.79
	MS	金融	36600	196.00

从上表中可知：

- 行索引第一层 = **r1** = [中国, 美国], 名称是**地区**
- 行索引第二层 = **r2** = [BABA, JD, GS, MS], 名称是**代号**
- 列索引 = **c** = [行业, 雇员, 价格], 名称是**特征**

查看 df 的 index 和 columns 的信息

```
1 df.index, df.columns

(MultiIndex(levels=[[ '中国', '美国'], ['BABA', 'JD', 'GS', 'MS']],
           labels=[[0, 0, 1, 1], [0, 1, 2, 3]],
           names=[['地区', '代号']]),
Index(['行业', '雇员', '价格'], dtype='object', name='特征'))
```

那么

- df 的行索引 = [**r1**, **r2**]
- df 的列索引 = **c**

1. 基于层来 `unstack()` 时, 选择第一层 (参数放 0)

```
1 df.unstack(0)
```

特征	行业		雇员		价格		
	地区	中国	美国	中国	美国	中国	美国
代号							
BABA	电商	NaN	101550.0	NaN	176.92	NaN	
JD	电商	NaN	175336.0	NaN	25.95	NaN	
GS	NaN	金融	NaN	60348.0	NaN	41.79	
MS	NaN	金融	NaN	36600.0	NaN	196.00	

df 被 `unstack(0)` 之后变成 (行 → 列)

- 行索引 = `r2`
- 列索引 = `[c, r1]`

重塑后的 DataFrame 这时行索引只有一层 (**代号**)，而列索引有两层，第一层是**特征**，第二层是**地区**。

2. 基于层来 `unstack()` 时，选择第二层 (参数放 1)

```
1 df.unstack(1)
```

特征	行业				雇员				价格			
	代号	BABA	JD	GS	MS	BABA	...	MS	BABA	JD	GS	MS
地区												
中国	电商	电商	NaN	NaN	101550.0	...	NaN	176.92	25.95	NaN	NaN	NaN
美国	NaN	NaN	金融	金融	NaN	...	36600.0	NaN	NaN	41.79	196.0	

2 rows × 12 columns

df 被 `unstack(1)` 之后变成 (行 → 列)

- 行索引 = `r1`
- 列索引 = `[c, r2]`

重塑后的 DataFrame 这时行索引只有一层 (**地区**)，而列索引有两层，第一层是**地区**，第二层是**代号**。

3. 基于层先 `unstack(0)` 再 `stack(0)`

```
1 df.unstack(0).stack(0)
```

	地区	中国	美国
代号	特征		
BABA	价格	176.92	NaN
	行业	电商	NaN
	雇员	101550	NaN
JD	价格	25.95	NaN
	行业	电商	NaN
...
GS	行业	NaN	金融
	雇员	NaN	60348
MS	价格	NaN	196
	行业	NaN	金融
	雇员	NaN	36600

12 rows × 2 columns

df 被 `unstack(0)` 之后变成 (行 → 列)

- 行索引 = `r2`
- 列索引 = `[c, r1]`

再被 `stack(0)` 之后变成 (列 → 行)

- 行索引 = `[r2, c]`
- 列索引 = `r1`

重塑后的 DataFrame 这时行索引有两层，第一层是 **代号**，第二层是 **特征**，而列索引只有一层 (**地区**)。

4. 基于层先 `unstack(0)` 再 `stack(1)`

```
1 df.unstack(0).stack(1)
```

	特征	行业	雇员	价格
代号	地区			

BABA	中国	电商	101550.0	176.92
JD	中国	电商	175336.0	25.95
GS	美国	金融	60348.0	41.79
MS	美国	金融	36600.0	196.00

df 被 `unstack(0)` 之后变成 (行 → 列)

- 行索引 = `r2`
- 列索引 = [`c, r1`]

再被 `stack(1)` 之后变成 (列 → 行)

- 行索引 = [`r2, r1`]
- 列索引 = `c`

重塑后的 DataFrame 这时行索引有两层，第一层是代号，第二层是地区，而列索引只有一层 (特征)。

5. 基于层先 `unstack(1)` 再 `stack(0)`

```
1 df.unstack(1).stack(0)
```

	代号	BABA	GS	JD	MS
地区	特征				
中国	价格	176.92	NaN	25.95	NaN
	行业	电商	NaN	电商	NaN
	雇员	101550	NaN	175336	NaN
美国	价格	NaN	41.79	NaN	196
	行业	NaN	金融	NaN	金融
	雇员	NaN	60348	NaN	36600

df 被 `unstack(1)` 之后变成 (行 → 列)

- 行索引 = `r1`
- 列索引 = [`c, r2`]

再被 `stack(0)` 之后变成 (列 → 行)

- 行索引 = [r1, c]
- 列索引 = r2

重塑后的 DataFrame 这时行索引有两层，第一层是地区，第二层是特征，而列索引只有一层（代号）。

6. 基于层先 unstack(1) 再 stack(1)

```
1 df.unstack(1).stack(1)
```

	特征	行业	雇员	价格
地区	代号			
中国	BABA	电商	101550	176.92
	JD	电商	175336	25.95
美国	GS	金融	60348	41.79
	MS	金融	36600	196.00

df 被 unstack(1) 之后变成 (行 → 列)

- 行索引 = r1
- 列索引 = [c, r2]

再被 stack(1) 之后变成 (列 → 行)

- 行索引 = [r1, r2]
- 列索引 = c

重塑后的 DataFrame 这时行索引有两层，第一层是地区，第二层是特征，而列索引只有一层（代号）。还原成原来的 df 了。

7. 基于层被 stack(), 没有填层数，默認為最后一层。

```
1 df.stack()
```

```
地区 代号 特征
中国 BABA 行业 电商
```

```
雇员 101550
      价格 176.92
      JD 行业 电商
      雇员 175336
      ...
      美国 GS 雇员 60348
          价格 41.79
          MS 行业 金融
          雇员 36600
          价格 196
Length: 12, dtype: object
```

df 被 `stack()` 之后变成 (列 → 行)

- 行索引 = [r1, r2, c]
- 列索引 = []

重塑后的 Series 只有行索引，有三层，第一层是地区，第二层是代号，第三层是特征。

8. 基于层被 `unstack()` 两次，没有填层数，默认为最后一层。

```
1 df.unstack().unstack()
```

```
特征 代号 地区
行业 BABA 中国 电商
      美国 NaN
      JD 中国 电商
      美国 NaN
      GS 中国 NaN
      ...
      价格 JD 美国 NaN
      GS 中国 NaN
          美国 41.79
      MS 中国 NaN
          美国 196
Length: 24, dtype: object
```

df 被第一次 `unstack()` 之后变成 (行 → 列)

- 行索引 = r1

- 列索引 = [c, r2]

df 被第二次 `unstack()` 之后变成 (行 → 列)

- 行索引 = []
- 列索引 = [c, r2, r1]

重塑后的 Series 只有列索引 (实际上是个转置的 Series)，有三层，第一层是特征，第二层是代号，第三层是地区。

5.2 透视

数据源表通常只包含行和列，那么经常有重复值出现在各列下，因而导致源表不能传递有价值的信息。这时可用「透视」方法调整源表的布局用作更清晰的展示。

知识点

本节「透视」得到的数据表和 Excel 里面的透视表 (pivot table) 是一样的。透视表是用来汇总其它表的数据：

1. 首先把源表分组，将不同值当做行 (row)、列 (column) 和值 (value)
2. 然后对各组内数据做汇总操作如排序、平均、累加、计数等

这种动态将「源表」得到想要「终表」的旋转 (pivoting) 过程，使透视表得以命名。

在 Pandas 里透视的方法有两种：

- 用 `pivot` 函数将「一张长表」变「多张宽表」，
- 用 `melt` 函数将「多张宽表」变「一张长表」，

本节使用的数据描述如下：

- 5 只股票：AAPL, JD, BABA, FB, GS

- 4 个交易日：从 2019-02-21 到 2019-02-26

```
1 data = pd.read_csv('Stock.csv', parse_dates=[0], dayfirst=True)
2 data
```

	Date	Symbol	Open	High	Low	Close	Adj Close	Volume
0	2019-02-21	AAPL	171.800003	172.369995	170.300003	171.059998	171.059998	17249700
1	2019-02-21	JD	24.820000	24.879999	24.010000	24.270000	24.270000	13542600
2	2019-02-21	BABA	171.000000	171.779999	169.800003	171.660004	171.660004	8434800
3	2019-02-21	GS	198.970001	199.449997	195.050003	196.360001	196.360001	2785900
4	2019-02-21	FB	161.929993	162.240005	159.589996	160.039993	160.039993	15607800
...
15	2019-02-26	AAPL	173.710007	175.300003	173.169998	174.330002	174.330002	17006000
16	2019-02-26	JD	25.980000	26.820000	25.660000	26.590000	26.590000	20264100
17	2019-02-26	BABA	179.789993	184.350006	179.369995	183.539993	183.539993	13857900
18	2019-02-26	GS	198.470001	200.559998	196.550003	198.899994	198.899994	2498000
19	2019-02-26	FB	164.339996	166.240005	163.800003	164.130005	164.130005	13645200

20 rows × 8 columns

从上表看出有 20 行 (5×4) 和 8 列，在 Date 和 Symbol 那两列下就有重复值，4 个日期和 5 个股票在 20 行中分别出现了 5 次和 4 次。

从长到宽 (pivot)

当我们做数据分析时，只关注不同股票在不同日期下的 Adj Close，那么可用 pivot 函数将原始 data 「透视」成一个新的 DataFrame，起名 close_price。在 pivot 函数中

- 将 index 设置成 'Date'
- 将 columns 设置成 'Symbol'
- 将 values 设置 'Adj Close'

close_price 实际上把 data['Date'] 和 data['Symbol'] 的唯一值当成支点(pivot 就是支点的意思) 创建一个 DataFrame，其中

- 行标签 = 2019-02-21, 2019-02-22, 2019-02-25, 2019-02-26
- 列标签 = AAPL, JD, BABA, FB, GS

在把 `data['Adj Close']` 的值放在以如上的行标签和列标签创建的 `close_price` 来展示。

代码如下：

```
1 close_price = data.pivot( index='Date',
2                             columns='Symbol',
3                             values='Adj Close' )
4 close_price
```

Symbol	AAPL	BABA	FB	GS	JD
Date					
2019-02-21	171.059998	171.660004	160.039993	196.360001	24.270000
2019-02-22	172.970001	176.919998	161.889999	196.000000	25.950001
2019-02-25	174.229996	183.250000	164.619995	198.649994	26.190001
2019-02-26	174.330002	183.539993	164.130005	198.899994	26.590000

如果觉得 `Adj Close` 不够，还想加个 `Volume` 看看，那么就把 `values` 设置成 `['Adj Close', 'Volume']`。这时支点还是 `data['Date']` 和 `data['Symbol']`，但是要透视的值增加到 `data[['Adj Close', 'Volume']]` 了。`pivot` 函数返回的是两个透视表。

```
1 data.pivot( index='Date',
2             columns='Symbol',
3             values=['Adj Close', 'Volume'] )
```

Symbol	Adj Close					Volume				
	AAPL	BABA	FB	GS	JD	AAPL	BABA	FB	GS	JD
Date										
2019-02-21	171.059998	171.660004	160.039993	196.360001	24.270000	17249700.0	8434800.0	15607800.0	2785900.0	13542600.0
2019-02-22	172.970001	176.919998	161.889999	196.000000	25.950001	18913200.0	16175600.0	15858500.0	2626600.0	27113300.0
2019-02-25	174.229996	183.250000	164.619995	198.649994	26.190001	21873400.0	22831800.0	18737100.0	3032200.0	29338500.0
2019-02-26	174.330002	183.539993	164.130005	198.899994	26.590000	17006000.0	13857900.0	13645200.0	2498000.0	20264100.0

如果不设置 `values` 参数，那么 `pivot` 函数返回的是六个透视表。(源表 `data` 有八列，两列当了支点，剩下六列用来透视)

```
1 all_pivot = data.pivot( index='Date',
2                         columns='Symbol' )
3 all_pivot
```

Symbol	Open					High					...	Adj Close
	AAPL	BABA	FB	GS	JD	AAPL	BABA	FB	GS	JD	...	AAPL
Date												
2019-02-21	171.800003	171.000000	161.929993	198.970001	24.820000	172.369995	171.779999	162.240005	199.449997	24.879999	...	171.059998
2019-02-22	171.580002	172.800003	160.580002	196.600006	24.549999	173.000000	177.020004	162.410004	197.750000	25.959999	...	172.970001
2019-02-25	174.160004	181.259995	163.070007	198.000000	27.110001	175.869995	183.720001	166.070007	201.500000	27.379999	...	174.229996
2019-02-26	173.710007	179.789993	164.339996	198.470001	25.980000	175.300003	184.350006	166.240005	200.559998	26.820000	...	174.330002

再继续观察下，`all_pivot` 实际上是个多层 DataFrame (有多层 columns)。假设我们要获取 2019-02-25 和 2019-02-26 两天的 BABA 和 FB 的开盘价，用以下「多层索引和切片」的方法。

```
1 all_pivot['Open'].iloc[2:,1:3]
```

Symbol	BABA	FB
Date		
2019-02-25	181.259995	163.070007
2019-02-26	179.789993	164.339996

从宽到长 (melt)

`pivot` 逆反操作是 `melt`。

- 前者将「一张长表」变成「多张宽表」
- 后者将「多张宽表」变成「一张长表」

具体来说，函数 `melt` 实际是将「源表」转化成 `id-variable` 类型的 DataFrame，下例将

- `Date` 和 `Symbol` 列当成 `id`
- 其他列 `Open`, `High`, `Low`, `Close`, `Adj Close` 和 `Volume` 当成 `variable`，而它们对应的值当成 `value`

代码如下：

```
1 melted_data = pd.melt( data, id_vars=['Date','Symbol'] )
2 melted_data.head(5).append(melted_data.tail(5))
```

	Date	Symbol	variable	value
0	2019-02-21	AAPL	Open	1.718000e+02
1	2019-02-21	JD	Open	2.482000e+01
2	2019-02-21	BABA	Open	1.710000e+02
3	2019-02-21	GS	Open	1.989700e+02
4	2019-02-21	FB	Open	1.619300e+02
115	2019-02-26	AAPL	Volume	1.700600e+07
116	2019-02-26	JD	Volume	2.026410e+07
117	2019-02-26	BABA	Volume	1.385790e+07
118	2019-02-26	GS	Volume	2.498000e+06
119	2019-02-26	FB	Volume	1.364520e+07

新生成的 DataFrame 有 120 行 ($4 \times 5 \times 6$)

- 4 = data['Date'] 有 4 个日期
- 5 = data['Symbol'] 有 5 只股票
- 6 = Open, High, Low, Close, Adj Close 和 Volume 这 6 个变量

在新表 melted_data 中

- 在参数 id_vars 设置的 Date 和 Symbol 还保持为 columns
- 此外还多出两个 columns, 一个叫 variable, 一个叫 value
 - variable 列下的值为 Open, High, Low, Close, Adj Close 和 Volume
 - value 列下的值为前者在「源表 data」中的值

函数 melt 可以生成一张含有多个 id 的长表, 然后可在 id 上筛选出我们想要的信息, 比如

```
1 melted_data[ lambda x: (x.Date=='25/02/2019') ]
```

2

```
& ((x.Symbol=='BABA')|(x.Symbol=='FB')) ]
```

	Date	Symbol	variable	value
12	2019-02-25	BABA	Open	1.812600e+02
14	2019-02-25	FB	Open	1.630700e+02
32	2019-02-25	BABA	High	1.837200e+02
34	2019-02-25	FB	High	1.660700e+02
52	2019-02-25	BABA	Low	1.807300e+02
...
74	2019-02-25	FB	Close	1.646200e+02
92	2019-02-25	BABA	Adj Close	1.832500e+02
94	2019-02-25	FB	Adj Close	1.646200e+02
112	2019-02-25	BABA	Volume	2.283180e+07
114	2019-02-25	FB	Volume	1.873710e+07

12 rows × 4 columns

在 `melted_data` 上使用调用函数 (callable function) 做索引，我们得到了在 2019-02-25 那天 BABA 和 FB 的信息。



6

数据表的分组和整合

DataFrame 中的数据可以根据某些规则分组，然后在每组的数据上计算出不同统计量。这种操作称之为 split-apply-combine，

6.1 数据准备

本节使用的数据描述如下：

- 5 只股票：AAPL, JD, BABA, FB, GS
- 1 年时期：从 2018-02-26 到 2019-02-26

```
1 data = pd.read_csv('1Y Stock Data.csv', parse_dates=[0], dayfirst=True)
2 data.head(3).append(data.tail(3))
```

	Date	Symbol	Open	High	Low	Close	Adj Close	Volume
0	2018-02-26	AAPL	176.350006	179.389999	176.210007	178.970001	176.285675	38162200
1	2018-02-27	AAPL	179.100006	180.479996	178.160004	178.389999	175.714386	38928100
2	2018-02-28	AAPL	179.259995	180.619995	178.050003	178.119995	175.448410	37782100
1257	2019-02-22	GS	196.600006	197.750000	195.199997	196.000000	196.000000	2626600
1258	2019-02-25	GS	198.000000	201.500000	197.710007	198.649994	198.649994	3032200
1259	2019-02-26	GS	198.470001	200.559998	196.550003	198.899994	198.899994	2498000

我们目前只对 `Adj Close` 感兴趣，而且想知道在哪一年份或哪一月份每支股票的 `Adj Close` 是多少。因此我们需要做两件事：

1. 只保留 '`Date`', '`Symbol`' 和 '`Adj Close`'
2. 从 '`Date`' 中获取 '`Year`' 和 '`Month`' 的信息并插入表中

将处理过后的数据存在 `data1` 中。

```

1 data1 = data[['Date', 'Symbol', 'Adj Close']]
2 data1.insert( 1, 'Year', pd.DatetimeIndex(data1['Date']).year )
3 data1.insert( 2, 'Month', pd.DatetimeIndex(data1['Date']).month )
4 data1.head(3).append(data1.tail(3))

```

	Date	Year	Month	Symbol	Adj Close
0	2018-02-26	2018	2	AAPL	176.285675
1	2018-02-27	2018	2	AAPL	175.714386
2	2018-02-28	2018	2	AAPL	175.448410
1257	2019-02-22	2019	2	GS	196.000000
1258	2019-02-25	2019	2	GS	198.649994
1259	2019-02-26	2019	2	GS	198.899994

6.2 分组 (grouping)

用某一特定标签 (`label`) 将数据 (`data`) 分组的语法如下：

```
data.groupby( label )
```

单标签分组

首先我们按 `Symbol` 来分组：

```
1 grouped = data1.groupby('Symbol')
2 grouped
```

```
<pandas.core.groupby.groupby.DataFrameGroupBy
object at 0x7fbbe7248d68>
```

又要提起那句说了无数遍的话「万物皆对象」了。这个 `grouped` 也不例外，当你对如果使用某个对象感到迷茫时，用 `dir()` 来查看它的「属性」和「内置方法」。以下几个属性和方法是我们感兴趣的：

- `ngroups`: 组的个数 (`int`)
- `size()`: 每组元素的个数 (`Series`)
- `groups`: 每组元素在原 `DataFrame` 中的索引信息 (`dict`)
- `get_group(label)`: 标签 `label` 对应的数据 (`DataFrame`)

下面看看这些属性和方法的产出结果。

数据里有 5 只股票，因此有 5 组。

```
1 grouped.ngroups
```

```
5
```

一年有 252 个交易日，因此每只股票含 252 条信息。

```
1 grouped.size()
```

```
Symbol
AAPL 252
BABA 252
FB 252
GS 252
JD 252
```

```
dtype: int64
```

苹果股票 (AAPL) 的索引从 0 到 251, ..., 一直到高盛股票 (GS) 的索引从 1008 到 1259。

```
1 grouped.groups
```

```
{'AAPL': Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
...                   242, 243, 244, 245, 246, 247, 248, 249, 250, 251],
dtype='int64', length=252),
'BABA': Int64Index([252, 253, 254, 255, 256, 257, 258, 259, 260, 261,
...                   494, 495, 496, 497, 498, 499, 500, 501, 502, 503],
dtype='int64', length=252),
'FB': Int64Index([ 756,  757,  758,  759,  760,  761,  762,  763,  764,  765,
...                   998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007],
dtype='int64', length=252),
'GS': Int64Index([1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017,
...                   1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259],
dtype='int64', length=252),
'JD': Int64Index([504, 505, 506, 507, 508, 509, 510, 511, 512, 513,
...                   746, 747, 748, 749, 750, 751, 752, 753, 754, 755],
dtype='int64', length=252)}
```

查查 'GS' 组里的数据的前五行。

```
1 grouped.get_group('GS').head()
```

	Date	Year	Month	Symbol	Adj Close
1008	2018-02-26	2018	2	GS	267.574249
1009	2018-02-27	2018	2	GS	264.289459
1010	2018-02-28	2018	2	GS	260.085419
1011	2018-03-01	2018	3	GS	254.001984
1012	2018-03-02	2018	3	GS	255.327515

接下来定义个 `print_groups` 函数便于打印组的名字和前五行信息。

```
1 def print_groups( group_obj ):
2     for name, group in group_obj:
```

```
3     print( name )
4     print( group.head() )
```

用这个函数来调用 grouped (上面用 `groupBy` 得到的对象)

```
1 print_groups( grouped )

AAPL
    Date  Year  Month Symbol  Adj Close
0 2018-02-26  2018      2   AAPL  176.285675
1 2018-02-27  2018      2   AAPL  175.714386
2 2018-02-28  2018      2   AAPL  175.448410
3 2018-03-01  2018      3   AAPL  172.375214
4 2018-03-02  2018      3   AAPL  173.567078
BABA
    Date  Year  Month Symbol  Adj Close
252 2018-02-26  2018      2   BABA  194.190002
253 2018-02-27  2018      2   BABA  188.259995
254 2018-02-28  2018      2   BABA  186.139999
255 2018-03-01  2018      3   BABA  181.990005
256 2018-03-02  2018      3   BABA  179.759995
FB
    Date  Year  Month Symbol  Adj Close
756 2018-02-26  2018      2     FB  184.929993
757 2018-02-27  2018      2     FB  181.460007
758 2018-02-28  2018      2     FB  178.320007
759 2018-03-01  2018      3     FB  175.940002
760 2018-03-02  2018      3     FB  176.619995
GS
    Date  Year  Month Symbol  Adj Close
1008 2018-02-26  2018      2     GS  267.574249
1009 2018-02-27  2018      2     GS  264.289459
1010 2018-02-28  2018      2     GS  260.085419
1011 2018-03-01  2018      3     GS  254.001984
1012 2018-03-02  2018      3     GS  255.327515
JD
    Date  Year  Month Symbol  Adj Close
504 2018-02-26  2018      2     JD  48.799999
505 2018-02-27  2018      2     JD  47.040001
506 2018-02-28  2018      2     JD  47.150002
507 2018-03-01  2018      3     JD  46.209999
508 2018-03-02  2018      3     JD  43.799999
```

这个 `print_groups` 函数在下面也多次被用到。

多标签分组

`groupBy` 函数除了支持单标签分组，也支持多标签分组 (将标签放入一个列表中)。

```
1 grouped2 = data1.groupby(['Symbol', 'Year', 'Month'])
2 print_groups( grouped2 )
```

```

('AAPL', 2018, 2)
    Date Year Month Symbol   Adj Close
0 2018-02-26 2018      2   AAPL 176.285675
1 2018-02-27 2018      2   AAPL 175.714386
2 2018-02-28 2018      2   AAPL 175.448410
('AAPL', 2018, 3)
    Date Year Month Symbol   Adj Close
3 2018-03-01 2018      3   AAPL 172.375214
4 2018-03-02 2018      3   AAPL 173.567078
5 2018-03-05 2018      3   AAPL 174.167923
6 2018-03-06 2018      3   AAPL 174.020172
7 2018-03-07 2018      3   AAPL 172.404770
...
('JD', 2019, 1)
    Date Year Month Symbol   Adj Close
718 2019-01-02 2019      1     JD 21.270000
719 2019-01-03 2019      1     JD 20.350000
720 2019-01-04 2019      1     JD 22.270000
721 2019-01-07 2019      1     JD 22.760000
722 2019-01-08 2019      1     JD 22.950001
('JD', 2019, 2)
    Date Year Month Symbol   Adj Close
739 2019-02-01 2019      2     JD 24.629999
740 2019-02-04 2019      2     JD 24.290001
741 2019-02-05 2019      2     JD 25.420000
742 2019-02-06 2019      2     JD 25.110001
743 2019-02-07 2019      2     JD 23.980000

```

不难看出在**每组左上方**，有一个(*Symbol, Year, Month*)元组型的标识：

- 第一组：('AAPL', 2018, 2)
- 最后一组：('JD', 2019, 2)

还记得【[数据结构之 Pandas \(上\)](#)】提到的重设索引(`set_index`)的操作么？

```

1 data2 = data1.set_index(['Symbol', 'Year', 'Month'])
2 data2.head().append(data2.tail())

```

			Date	Adj Close
Symbol	Year	Month		
AAPL	2018	2	2018-02-26	176.285675
		2	2018-02-27	175.714386
		2	2018-02-28	175.448410
		3	2018-03-01	172.375214
		3	2018-03-02	173.567078
GS	2019	2	2019-02-20	198.600006
		2	2019-02-21	196.360001
		2	2019-02-22	196.000000

	2	2019-02-25	198.649994
	2	2019-02-26	198.899994

对 `data1` 重设索引之后，产出是一个有 multi-index 的 DataFrame，记做 `data2`。由于有多层索引，这时我们根据索引的 `level` 来分组，下面 `level = 1` 就是对第一层 (`Year`) 进行分组。

```
1 grouped3 = data2.groupby(level=1)
2 print_groups( grouped3 )
```

2018			
Symbol	Year	Month	Date
AAPL	2018	2	2018-02-26 176.285675
		2	2018-02-27 175.714386
		2	2018-02-28 175.448410
		3	2018-03-01 172.375214
		3	2018-03-02 173.567078

2019			
Symbol	Year	Month	Date
AAPL	2019	1	2019-01-02 157.245605
		1	2019-01-03 141.582779
		1	2019-01-04 147.626846
		1	2019-01-07 147.298264
		1	2019-01-08 150.106216

注意**每组左上方**的标识是 `Year`。

多层索引中的任意个数的索引也可以用来分组，下面 `level = [0,2]` 就是对第零层 (`Symbol`) 和第二层 (`Month`) 进行分组。

```
1 grouped4 = data2.groupby(level=[0, 2])
2 print_groups( grouped4 )
```

('AAPL', 1)			
Symbol	Year	Month	Date
AAPL	2019	1	2019-01-02 157.245605
		1	2019-01-03 141.582779
		1	2019-01-04 147.626846
		1	2019-01-07 147.298264
		1	2019-01-08 150.106216

('AAPL', 2)			
Symbol	Year	Month	Date
AAPL	2018	2	2018-02-26 176.285675
		2	2018-02-27 175.714386
		2	2018-02-28 175.448410
	2019	2	2019-02-01 165.808884
		2	2019-02-04 170.518677

```
('JD', 11)
      Date  Adj Close
Symbol Year Month
JD    2018 11    2018-11-01  25.450001
          11    2018-11-02  24.049999
          11    2018-11-05  24.150000
          11    2018-11-06  23.600000
          11    2018-11-07  24.070000
('JD', 12)
      Date  Adj Close
Symbol Year Month
JD    2018 12    2018-12-03  22.010000
          12    2018-12-04  21.450001
          12    2018-12-06  21.230000
          12    2018-12-07  20.930000
          12    2018-12-10  20.510000
```

注意每组左上方的标识是 (Symbol, Month)。

6.3 整合 (aggregating)

做完分组之后 so what? 当然是在每组做点数据分析再整合啦。

一个最简单的例子就是上节提到的 `size()` 函数，用 grouped 对象 (上面根据 `Symbol` 分组得到的) 来举例。

```
1 grouped.size()
```

```
Symbol
AAPL 252
BABA 252
FB 252
GS 252
JD 252
dtype: int64
```

一个更实际的例子是用 `mean()` 函数计算每个 `Symbol` 下 1 年时期的股价均值。在获取任意信息就用 DataFrame 的索引或切片那一套方法。

```
1 grouped.mean()
```

	Year	Month	Adj Close
Symbol			
AAPL	2018.150794	6.488095	186.022309
BABA	2018.150794	6.488095	171.780992
FB	2018.150794	6.488095	167.244841
GS	2018.150794	6.488095	221.593412
JD	2018.150794	6.488095	31.340754

除了上述方法，整合还可以用内置函数 `aggregate()` 或 `agg()` 作用到「组对象」上。用 `grouped4` 对象（上面根据 *Symbol, Year, Month* 分组得到的）来举例。

```
1 result = grouped4.agg( np.mean )
2 result.head().append(result.tail())
```

			Adj Close	
Symbol	Year	Month		
AAPL	2018	2	175.816157	
		3	171.878962	
		4	167.286981	
		5	183.207503	
		6	186.508769	
		10	23.570435	
JD		11	22.139048	
		12	21.186842	
2019	1	22.747619		
	2	24.800000		

函数 `agg()` 其实是一个高阶函数（见【[Python 入门篇 \(下\)](#)】），里面的参数可以是另外一个函数，比如上例的 `np.mean`。上面代码对每只股票在每年每个月上求均值。

那么参数可以是另外**一组**函数么？可以的！

```
1 result = grouped4.agg( [np.mean, np.std] )
2 result.head().append(result.tail())
```

			Adj Close

			mean	std
Symbol	Year	Month		
AAPL	2018	2	175.816157	0.427810
		3	171.878962	4.734514
		4	167.286981	4.921071
		5	183.207503	5.115191
		6	186.508769	3.590954
		10	23.570435	0.984542
JD	2018	11	22.139048	1.690337
		12	21.186842	0.764831
		1	22.747619	1.085647
	2019	2	24.800000	0.795495

将 `np.mean` 和 `np.std` 放进列表中，当成是高阶函数 `agg()` 的参数。上面代码对每只股票在每年每个月上求均值和标准差。

既然 `agg()` 是高阶函数，参数当然也可以是匿名函数 (`lambda` 函数)，下面我们定义一个对 `grouped` 里面每个标签下求最大值和最小值，再求差。注意 `lambda` 函数里面的 `x` 就是 `grouped`。

```
1 result = grouped.agg( lambda x: np.max(x)-np.min(x) )
2 result.head().append(result.tail())
```

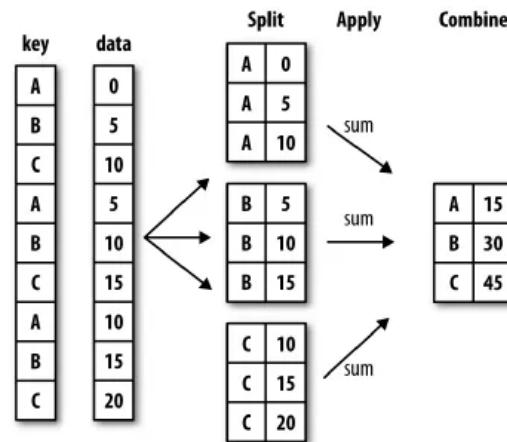
	Date	Year	Month	Adj Close
Symbol				
AAPL	365 days	1	11	88.692703
BABA	365 days	1	11	80.259995
FB	365 days	1	11	93.440002
GS	365 days	1	11	114.072418
JD	365 days	1	11	29.529999
AAPL	365 days	1	11	88.692703
BABA	365 days	1	11	80.259995
FB	365 days	1	11	93.440002
GS	365 days	1	11	114.072418
JD	365 days	1	11	29.529999

上面代码对每只股票在 `Date`, `Year`, `Month` 和 `Adj Close` 上求「最大值」和「最小值」的差。真正有价值的信息在 `Adj Close` 那一栏，但我们来验证一下其他几栏。

- Date: 365 days, 合理, 一年数据
- Year: 1, 合理, 2019 年和 2018 年
- Month: 11, 合理, 12 月和 1 月。

6.4 split-apply-combine

前几节做的事情的实质就是一个 split-apply-combine 的过程, 如下图所示:



该 split-apply-combine 过程有三步:

1. 根据 key 来 split 成 n 组
2. 将函数 apply 到每个组
3. 把 n 组的结果 combine 起来

在看具体例子之前, 我们先定一个 top 函数, 返回 DataFrame 某一栏中 n 个最大值。

```
1 def top( df, n=5, column='Volume' ):
2     return df.sort_values(by=column)[-n:]
```

将 top 函数用到最原始的数据 (从 csv 中读取出来的) 上。

```
1 top( data )
```

	Date	Symbol	Open	High	Low	Close	Adj Close	Volume
145	2018-09-21	AAPL	220.779999	221.360001	217.289993	217.660004	215.976913	96246700
773	2018-03-21	FB	164.800003	173.399994	163.300003	169.389999	169.389999	105920200
776	2018-03-26	FB	160.820007	161.100006	149.020004	160.059998	160.059998	126116600
772	2018-03-20	FB	167.470001	170.199997	161.949997	168.149994	168.149994	129851800
861	2018-07-26	FB	174.889999	180.130005	173.750000	176.259995	176.259995	169803700

从上表可看出，在 Volume 栏取 5 个最大值。

Apply 函数

在 split-apply-combine 过程中，apply 是核心。Python 本身有高阶函数 apply() 来实现它，既然是高阶函数，参数可以是另外的函数了，比如刚定义好的 top()。

将 top() 函数 apply 到按 Symbol 分的每个组上，按每个 Symbol 打印出来了 Volume 栏下的 5 个最大值。

```
1 data.groupby('Symbol').apply(top)
```

		Date	Symbol	Open	High	Low	Close	Adj Close	Volume
Symbol									
AAPL	109	2018-08-01	AAPL	199.130005	201.759995	197.309998	201.500000	199.243088	67935700
	215	2019-01-03	AAPL	143.979996	145.720001	142.000000	142.190002	141.582779	91312200
	175	2018-11-02	AAPL	209.550003	213.649994	205.429993	207.479996	205.875610	91328700
	208	2018-12-21	AAPL	156.860001	158.160004	149.630005	150.729996	150.086304	95744600
	145	2018-09-21	AAPL	220.779999	221.360001	217.289993	217.660004	215.976913	96246700
BABA	427	2018-11-02	BABA	152.559998	154.360001	146.279999	147.589996	147.589996	45985800
	426	2018-11-01	BABA	144.979996	152.317993	138.619995	151.250000	151.250000	47039300
	410	2018-10-10	BABA	142.500000	144.000000	137.919998	138.289993	138.289993	55828800
	300	2018-05-04	BABA	180.399994	190.600006	178.619995	188.889999	188.889999	57788300
	377	2018-08-23	BABA	184.970001	186.500000	171.910004	172.229996	172.229996	78843400
FB	771	2018-03-19	FB	177.009995	177.169998	170.059998	172.559998	172.559998	88140100
	773	2018-03-21	FB	164.800003	173.399994	163.300003	169.389999	169.389999	105920200
	776	2018-03-26	FB	160.820007	161.100006	149.020004	160.059998	160.059998	126116600
	772	2018-03-20	FB	167.470001	170.199997	161.949997	168.149994	168.149994	129851800
	861	2018-07-26	FB	174.889999	180.130005	173.750000	176.259995	176.259995	169803700
GS	1216	2018-12-21	GS	168.250000	169.630005	159.419998	160.050003	160.050003	8960000
	1106	2018-07-17	GS	231.479996	233.229996	226.869995	231.020004	229.328781	9280200

	1043	2018-04-17	GS	261.600006	262.250000	252.339996	253.630005	250.886078	10134400
	1189	2018-11-12	GS	222.000000	222.309998	205.130005	206.050003	205.218948	11019400
	1232	2019-01-16	GS	187.000000	198.149994	185.600006	197.080002	197.080002	15194200
JD	624	2018-08-16	JD	32.000000	33.200001	31.180000	31.969999	31.969999	46628200
	623	2018-08-15	JD	32.130001	32.560001	31.480000	32.360001	32.360001	46803300
	636	2018-09-04	JD	30.010000	30.030001	29.059999	29.430000	29.430000	47842200
	712	2018-12-21	JD	20.090000	21.990000	19.520000	21.080000	21.080000	49686400
	637	2018-09-05	JD	27.959999	28.049999	26.000000	26.299999	26.299999	82488000

上面在使用 `top()` 时，对于 `n` 和 `column` 我们都只用的默认值 5 和 'Volume'。如果用自己设定的值 `n = 1, column = 'Adj Close'`，写法如下(下面使用在元数据上插入 Year 和 Month 的数据)：

```
1 data1.groupby(['Symbol','Year']).apply(top, n=1, column='Adj Close')
```

			Date	Year	Month	Symbol	Adj Close
Symbol	Year						
AAPL	2018	153	2018-10-03	2018	10	AAPL	230.275482
	2019	251	2019-02-26	2019	2	AAPL	174.330002
BABA	2018	328	2018-06-14	2018	6	BABA	210.860001
	2019	503	2019-02-26	2019	2	BABA	183.539993
FB	2018	860	2018-07-25	2018	7	FB	217.500000
	2019	993	2019-02-05	2019	2	FB	171.160004
GS	2018	1018	2018-03-12	2018	3	GS	270.422424
	2019	1234	2019-01-18	2019	1	GS	202.539993
JD	2018	504	2018-02-26	2018	2	JD	48.799999
	2019	755	2019-02-26	2019	2	JD	26.590000

按每个 `Symbol` 和 `Year` 打印出来了 `Adj Close` 栏下的最大值。



	代号	价格	Merge		代号	雇员	
df1	0	JD	25.95	df2	0	JD	多
	1	BABA	176.92		1	BABA	中
	2	PDD	22.50		2	BIDU	多

`pd.merge(df1, df2, how='left' on='代号')` `pd.merge(df1, df2, how='inner' on='代号')`

	代号	价格	雇员
df1	0	JD	25.95
	1	BABA	176.92
	2	PDD	22.50

	代号	价格	雇员
df2	0	JD	多
	1	BABA	中
	2	BIDU	多

`pd.merge(df1, df2, how='right' on='代号')`

	代号	价格	雇员
df1	0	JD	25.95
	1	BABA	176.92
	2	BIDU	NaN

`pd.merge(df1, df2, how='outer' on='代号')`

	代号	价格	雇员
df1	0	JD	25.95
	1	BABA	176.92
	2	PDD	22.50
	3	BIDU	NaN

【连接数据表】用 `concat` 函数对 Series 和 DataFrame 沿着不同轴连接。

【重塑数据表】用 `stack` 函数将「列索引」变成「行索引」，用 `unstack` 函数将「行索引」变成「列索引」。它们只是改变数据表的布局和展示方式而已。

Stack

df

		代号	价格
行业	雇员		
电商	多	JD	25.95
	中	BABA	176.92
金融	多	BAC	29.88
	中	GS	196.00

Multi-Index

`df.stack()`

行业	雇员	代号	JD
电商	多	价格	25.95
	中	价格	BABA
金融	多	代号	BAC
	中	价格	29.88
金融	多	代号	GS
	中	价格	196.00

Multi-Index

Unstack(0)

series

行业	雇员		
电商	多	代号	JD
		价格	25.95
	中	价格	BABA
		价格	176.92
	多	代号	BAC
		价格	29.88
金融	中	代号	GS
		价格	196.00

Multi-Index

series.unstack(0)

series.unstack('行业')

	行业	电商	金融
雇员			
多	代号	JD	BAC
	价格	25.95	29.88
中	代号	BABA	GS
	价格	176.92	196.00

Multi-Index

Unstack(1)

series

行业	雇员		
电商	多	代号	JD
		价格	25.95
	中	价格	BABA
		价格	176.92
	多	代号	BAC
		价格	29.88
金融	中	代号	GS
		价格	196.00

Multi-Index

series.unstack(1)

series.unstack('雇员')

	雇员	多	中
行业			
电商	代号	JD	BABA
	价格	25.95	176.92
金融	代号	BAC	GS
	价格	29.88	196.00

Multi-Index

【透视数据表】用 `pivot` 函数将「一张长表」变成「多张宽表」，用 `melt` 函数将「多张宽表」变成「一张长表」。它们只是改变数据表的布局和展示方式而已。

Pivot

df

	行业	雇员	价格	代号
0	电商	多	25.95	JD
1	电商	中	176.92	BABA
2	电商	少	22.50	PDD
3	金融	多	29.88	BAC
4	金融	中	196.00	GS
5	金融	小	41.79	MS

df.pivot(index='行业', columns='雇员', values='价格')



雇员	多	中	少
行业			
电商	25.95	176.92	22.50
金融	29.88	196.00	41.79

Melt

df

	行业	雇员	价格	代号
0	电商	多	25.95	JD
1	金融	中	196.00	GS

df.melt(id_vars=['行业', '雇员'])



	行业	雇员	variable	value
0	电商	多	价格	25.95
1	金融	中	价格	196.00
2	电商	多	代号	JD
3	金融	中	代号	GS

【分组数据表】用 `groupBy` 函数按不同「列索引」下的值分组。一个「列索引」或多个「列索引」就可以。

【整合数据表】用 `agg` 函数对每个组做整合而计算统计量。

【split-apply-combine】用 `apply` 函数做数据分析时美滋滋。

Split-Apply-Combine

```
df.groupby('行业')  
    .apply(sum, columns='价格')
```



至此，我们已经打好 Python Basics 的基础，能用 NumPy 做数组计算，能用 SciPy 做插值、积分和优化，能用 Pandas 做数据分析，现在已经搞很多事情了。现在我们唯一欠缺的是如何画图或可视化数据，下帖从最基础的可视化工具 Matplotlib 开始讲。Stay Tuned!

• END •