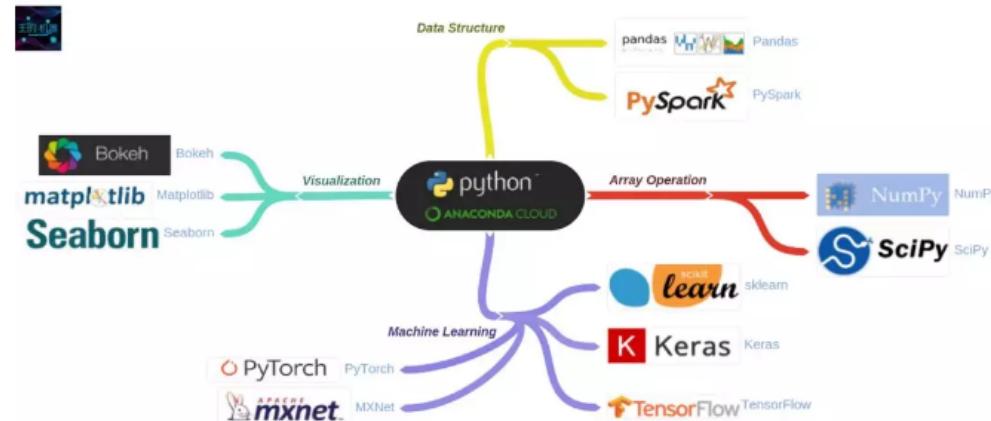


# 盘一盘 Python 系列 4 - Pandas (上)

From:王圣元 王的机器 4/11



全文共 20592 字, 63 幅图,

预计阅读时间 52 分钟。

【注: 本帖小节 2.2 用万矿里的 WindPy 来下载金融数据】

## 0 引言

本文是 Python 系列的第六篇

- [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

Pandas 是 Python 为解决数据分析而创建的，详情看官网 (<https://pandas.pydata.org/>)。在使用 *pandas* 之前，需要引进它，语法如下：

```
1 import pandas
```

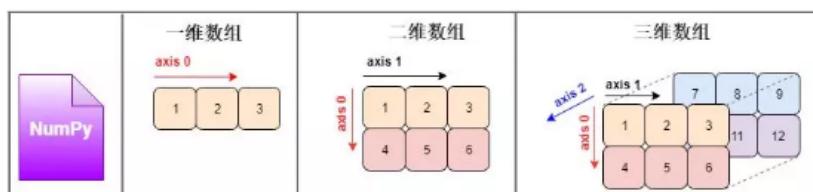
这样你就可以用 *pandas* 里面所有的内置方法 (build-in methods) 了，比如创建一维的 *Series* 和二维的 *DataFrame*。

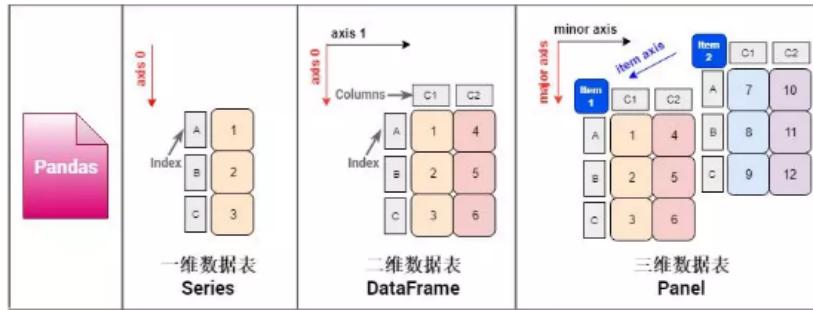
```
1 pandas.Series()  
2 pandas.DataFrame()
```

但是每次写 *pandas* 字数有点多，通常我们给 *pandas* 起个别名 pd，用以下语法，这样所有出现 *pandas* 的地方都可以用 pd 替代。

```
1 import pandas as pd
```

Pandas 里面的数据结构是「多维数据表」，学习它可以类比这 NumPy 里的「多维数组」。1/2/3 维的「多维数据表」分别叫做 Series (系列), DataFrame (数据帧) 和 Panel (面板)，和1/2/3 维的「多维数组」的类比关系如下。





由于「系列」、「数据帧」和「面板」这些直译过来的中文名词听起来有些奇怪，在本帖还是直接用 Series, DataFrame 和 Panel。

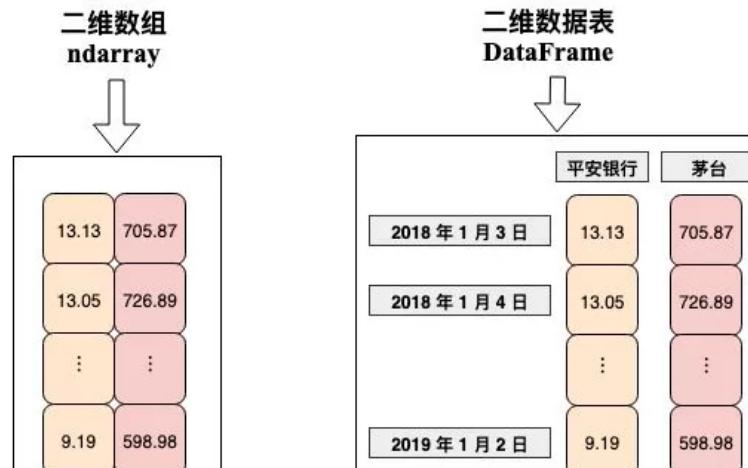
对比 NumPy (np) 和 Pandas (pd) 每个维度下的数据结构，不难看出

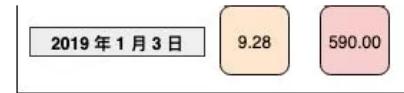
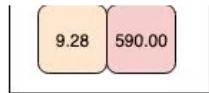
pd 多维数据表 = np 多维数组 + 描述

其中

- Series = 1darray + index
- DataFrame = 2darray + index + columns
- Panel = 3darray + index + columns + item

每个维度上的「索引」使得「多维数据表」比「多维数组」涵盖更多的信息，如下图，左边的 2d array 仅仅储存了一组数值（具体代表什么意思却不知道），而右边的 DataFrame 一看就知道这是平安银行和茅台从 2018-1-3 到 2019-1-3 的价格。





和学习 *numpy*一样，学习 *pandas* 还是遵循的 Python 里「万物皆对象」的原则，既然把数据表当对象，我们就按着**数据表的创建、数据表的存载、数据表的获取、数据表的合并和连接、数据表的重塑和透视、和数据表的分组和整合**来盘一盘 Pandas，目录如下：



由于篇幅原因，Pandas 系列分两贴，上贴讲前三节的内容，下贴讲后三节的内容。



## 1

## 数据表的创建

数据表有三大类型

- **Series**: 一维数据，类似于 python 中的基本数据的 list 或 NumPy 中的 1D array。Pandas 里最基本的数据结构
- **DataFrame**: 二维数据，类似于 R 中的 data.frame 或 Matlab 中的 Tables。DataFrame 是 Series 的容器
- **Panel**: 三维数据。Panel 是 DataFrame 的容器

### 知识点

最常见的数据类型是二维的 *DataFrame*，其中

- 每行代表一个示例 (instance)
- 每列代表一个特征 (feature)

*DataFrame* 可理解成是 *Series* 的容器，每一列都是一个 *Series*，或者 *Series* 是只有一列的 *DataFrame*。

*Panel* 可理解成是 *DataFrame* 的容器。

接下来我们用代码来创建 *pandas* 数据表，有两种方式：

1. 按步就班的用 `pd.Series()`, `pd.DataFrame()` 和 `pd.Panel()`
2. 一步登天的用万矿里面的 WindPy API 读取

## 一维 Series

创建 Series 只需用下面一行代码

```
pd.Series(x, index=idx)
```

其中 x 可以是

1. 列表 (list)
2. *numpy* 数组 (ndarray)
3. 字典 (dict)

回顾在 [【Python 入门篇 \(下\)】](#) 讲的函数里可以设定不同参数，那么

- x 是位置参数
- index 是默认参数，默认值为 `idx = range(0, len(x))`

## 用列表

```
1 s = pd.Series([27.2, 27.65, 27.70, 28])
2 s
```

```
0 27.20
1 27.65
2 27.70
3 28.00
dtype: float64
```

打印出来并不仅仅是列表里面的浮点数，每个浮点数前面还有一个索引，在本例中是 0, 1, 2, 3。

因此在创建 Series 时，如果不显性设定 index，那么 Python 给定一个默认从 0 到 N-1 的值，其中 N 是 x 的长度。

Series s 也是一个对象，用 `dir(s)` 可看出关于 Series 所有的属性和内置函数，其中最重要的是

- 用 `s.values` 打印 s 中的元素
- 用 `s.index` 打印 s 中的元素对应的索引

```
1 s.values
```

```
array([27.2 , 27.65, 27.7 , 28. ])
```

```
1 s.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

不难发现，以上创建的 Series 和 numpy 数组比多了「索引」，但这种 0,1,2,3 的索引是在没有什么描述意义。实际上我们定义的 s 是海底捞在 2019 年 4 月 1 日到 2019 年 4 月 4 日的股价，那么用日期来当索引是不是更好些？

```
1 dates = pd.date_range('20190401', periods=4)
2 s2 = pd.Series( [27.2, 27.65, 27.7, 28], index=dates )
3 s2
```

```
2019-04-01 27.20
2019-04-02 27.65
2019-04-03 27.70
2019-04-04 28.00
Freq: D, dtype: float64
```

显然，s2 比 s 包含的信息更多，这是 s2 的索引是一组日期对象，数据类型是 `datetime64`，频率是 D (天)。

```
1 s2.index
```

```
DatetimeIndex(['2019-04-01', '2019-04-02', '2019-04-03', '2019-04-04'],
dtype='datetime64[ns]', freq='D')
```

你甚至还可以给 s2 命名，就叫[海底捞股价](#)如何？

```
1 s2.name = '海底捞股价'  
2 s2
```

```
2019-04-01 27.20  
2019-04-02 27.65  
2019-04-03 27.70  
2019-04-04 28.00  
Freq: D, Name: 海底捞股价, dtype: float64
```

## 用 numpy 数组

除了用列表，我们还可以用 numpy 数组来生成 Series。在下例中，我们加入缺失值 np.nan，并分析一下 Series 中另外 5 个属性或内置函数的用法：

- `len`: s 里的元素个数
- `shape`: s 的形状 (用元组表示)
- `count`: s 里不含 nan 的元素个数
- `unique`: 返回 s 里不重复的元素
- `value_counts`: 统计 s 里非 nan 元素的出现次数

对照上面函数的用法，下面的输出一看就懂了吧。

```
1 s = pd.Series( np.array([27.2, 27.65, 27.7, 28, 28, np.nan]) )  
2 print('The length is', len(s))  
3 print('The shape is', s.shape)  
4 print('The count is', s.count())
```

```
The length is 6  
The shape is (6,)  
The count is 5
```

```
1 s.unique()
```

```
array([27.2, 27.65, 27.7, 28., nan])
```

```
1 s.value_counts()
```

```
28.00 2  
27.70 1  
27.65 1  
27.20 1  
dtype: int64
```

## 用字典

创建 Series 还可以用字典。字典的「键值对」的「键」自动变成了 Series 的索引 (index)，而「值」自动变成了 Series 的值 (values)。代码如下 (下列用 name 参数来对 s3 命名)

```
1 data_dict = { 'BABA': 187.07, 'PDD': 21.83, 'JD': 30.79, 'BIDU': 184.77 }  
2 s3 = pd.Series(data_dict, name='中概股')  
3 s3.index.name = '股票代号'  
4 s3
```

```
股票代号  
BABA 187.07  
PDD 21.83  
JD 30.79  
BIDU 184.77  
Name: 中概股, dtype: float64
```

给 s3 起名中概股是因为阿里巴巴 (BABA)、拼多多 (PDD)、京东 (JD) 和百度 (BIDU) 都是中国公司但在美国上市的。此外还可以给 index 命名为 '股票代号'。

现在假设我们的股票代号为

```
1 stock = ['FB', 'BABA', 'PDD', 'JD']  
2 s4 = pd.Series( sdata, index=stock )  
3 s4
```

```
FB NaN  
BABA 160.0  
PDD 28.0  
JD 25.0  
dtype: float64
```

代号里多加了脸书 (FB)，而 sdata 字典中没有 FB 这个键，因此生成的 s4 在 FB 索引下对应的值为 NaN。再者，代号里没有百度 (BIDU)，因此 s4 里面没有 BIDU 对应的值 (即便 sdata 里面有)。

当两个 Series 进行某种操作时，比如相加，Python 会自动对齐不同 Series 的 index，如下面代码所示：

```
1 s3 + s4
```

```
BABA 320.0
BIDU NaN
FB NaN
JD 50.0
PDD 56.0
dtype: float64
```

Series 是 Pandas 里面最基本的数据结构，但是对应每个索引只有一个元素 (比如一个日期对应一个股价)，因此 Series 处理不了每个索引对应多个元素 (比如一个日期对应一个开盘价、收盘价、交易量等等)。而 DataFrame 可以解决这个问题。

## 「二维 DataFrame」

创建 DataFrame 只需用下面一行代码

```
pd.DataFrame( x, index=idx,
               columns=col )
```

其中 x 可以是

1. 二维列表 (list)
2. 二维 numpy 数组 (ndarray)
3. 字典 (dict)，其值是一维列表、numpy 数组或 Series
4. 另外一个 DataFrame

回顾在【[Python 入门篇 \(下\)](#)】讲的函数里可以设定不同参数，那么

- `x` 是位置参数
- `index` 是默认参数，默认值为 `idx = range(0, x.shape[0])`
- `columns` 是默认参数，默认值为 `col = range(0, x.shape[1])`

### 用列表或 numpy 数组

```
1 # df1 = pd.DataFrame( [[1, 2, 3], [4, 5, 6]] )
2 df1 = pd.DataFrame( np.array([[1, 2, 3], [4, 5, 6]]) )
3 df1
```

	0	1	2
0	1	2	3
1	4	5	6

在创建 DataFrame 时，如果不显性设定 `index` 和 `columns` 时，那么 Python 给它们默认值，其中

- `index = 0 到 r-1, r 是 x 的行数`
- `columns = 0 到 c-1, c 是 x 的列数`

### 用对象为列表的字典

```
1 symbol = ['BABA', 'JD', 'AAPL', 'MS', 'GS', 'WMT']
2 data = {'行业': ['电商', '电商', '科技', '金融', '金融', '零售'],
3         '价格': [176.92, 25.95, 172.97, 41.79, 196.00, 99.55],
4         '交易量': [16175610, 27113291, 18913154, 10132145, 2626634, 8086946],
5         '雇员': [101550, 175336, 100000, 60348, 36600, 2200000]}
6 df2 = pd.DataFrame( data, index=symbol )
7 df2.name='美股'
8 df2.index.name = '代号'
9 df2
```

	行业	价格	交易量	雇员
代号				
<b>BABA</b>	电商	176.92	16175610	101550
<b>JD</b>	电商	25.95	27113291	175336

AAPL	科技	172.97	18913154	100000
MS	金融	41.79	10132145	60348
GS	金融	196.00	2626634	36600
WMT	零售	99.55	8086946	2200000

字典的「键值对」的「键」自动变成了 DataFrame 的栏 (columns)，而「值」自动变成了 DataFrame 的值 (values)，而其索引 (index) 需要另外定义。

分别来看 df2 的 values, columns 和 index。

```
1 df2.values
```

```
array([['电商', 176.92, 16175610, 101550],  
       ['电商', 25.95, 27113291, 175336],  
       ['科技', 172.97, 18913154, 100000],  
       ['金融', 41.79, 10132145, 60348],  
       ['金融', 196.0, 2626634, 36600],  
       ['零售', 99.55, 8086946, 2200000]], dtype=object)
```

```
1 df2.columns
```

```
Index(['行业', '价格', '交易量', '雇员'], dtype='object')
```

```
1 df2.index
```

```
Index(['BABA', 'JD', 'AAPL', 'MS', 'GS', 'WMT'],  
      dtype='object', name='代号')
```

## A 查看 DataFrame

我们可以从头或从尾部查看 DataFrame 的 n 行，分别用 `df2.head()` 和 `df2.tail(n)`，如果没有设定 n，默认值为 5 行。

```
1 df2.head()
```

	行业	价格	交易量	雇员
代码				

<b>BABA</b>	电商	176.92	16175610	101550
<b>JD</b>	电商	25.95	27113291	175336
<b>AAPL</b>	科技	172.97	18913154	100000
<b>MS</b>	金融	41.79	10132145	60348
<b>GS</b>	金融	196.00	2626634	36600

```
1 df2.tail(3)
```

	行业	价格	交易量	雇员
代码				
<b>MS</b>	金融	41.79	10132145	60348
<b>GS</b>	金融	196.00	2626634	36600
<b>WMT</b>	零售	99.55	8086946	2200000

## B 统计 DataFrame

我们用 `df2.describe()` 还可以看看 DataFrame 每栏的统计数据。

```
1 df2.describe()
```

	价格	交易量	雇员
<b>count</b>	6.000000	6.000000e+00	6.000000e+00
<b>mean</b>	118.863333	1.384130e+07	4.456390e+05
<b>std</b>	73.748714	8.717312e+06	8.607522e+05
<b>min</b>	25.950000	2.626634e+06	3.660000e+04
<b>25%</b>	56.230000	8.598246e+06	7.026100e+04
<b>50%</b>	136.260000	1.315388e+07	1.007750e+05
<b>75%</b>	175.932500	1.822877e+07	1.568895e+05
<b>max</b>	196.000000	2.711329e+07	2.200000e+06

函数 `describe()` 只对「数值型变量」有用 (没有对「字符型变量」行业栏做统计)，统计量分别包括个数、均值、标准差、最小值，25-50-75 百分数值，最大值。一般做数据分析第一步会用这个表大概看看

- 数据是否有缺失值 (每个栏下的 `count` 是否相等)?
- 数据是否有异常值 (最小值 `min` 和最大值 `max` 是否太极端)?

## C 升维 DataFrame

我们用 `MultiIndex.from_tuples()` 还可以赋予 DataFrame 多层索引 (实际上增加了维度，多层索引的 DataFrame 实际上是三维数据)。

```
1 df2.index = pd.MultiIndex.from_tuples(  
2     [(['中国公司', 'BABA'), ('中国公司', 'JD'),  
3      ('美国公司', 'AAPL'), ('美国公司', 'MS'),  
4      ('美国公司', 'GS'), ('美国公司', 'WMT')])  
5 df2
```

		行业	价格	交易量	雇员
中国公司	BABA	电商	176.92	16175610	101550
	JD	电商	25.95	27113291	175336
美国公司	AAPL	科技	172.97	18913154	100000
	MS	金融	41.79	10132145	60348
	GS	金融	196.00	2626634	36600
	WMT	零售	99.55	8086946	2200000

在 `MultiIndex.from_tuples()` 中传递一个「元组的列表」，每个元组，比如 `('中国公司', 'BABA')`，第一个元素 `中国公司` 是第一层 index，第二个元素 `BABA` 是第二层 index。

DataFrame 是 Series 的容器，那什么是 DataFrame 的容器？Panel!

## 三维 Panel

首先需要指出的是 Panel 在未来版本中会被废除，因此不想花时间看的同学可跳过。

创建 Panel 只需用下面一行代码

```
pd.Panel( x, item=item,  
          major_axis=n1,
```

```
minor_axis=n2 )
```

其中 x 可以是

1. 三维列表 (list)
2. 三维 numpy 数组 (ndarray)
3. 字典 (dict), 其值是 DataFrame

回顾在 [【Python 入门篇 \(下\)】](#) 讲的函数里可以设定不同参数, 那么

- x 是位置参数
- items 是默认参数 (axis 0), 默认值为 `itm = range(0, number of DataFrame)`
- major\_axis 是默认参数 (axis 1), 默认值和 DataFrame 的默认 index 一样
- minor\_axis 是默认参数 (axis 2), 默认值和 DataFrame 的默认 columns 一样

## 用 numpy 数组

```
1 pn = pd.Panel(np.random.randn(2, 5, 4))
2 pn
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 4
Minor_axis axis: 0 to 3
```

Panel pn 含有 2 个 DataFrame, items 为 0, 1; 每个 DataFrame 有 5 行 4 列, 因此 major\_axis 为 0,1,2,3,4, 而 minor\_axis 为 0,1,2,3。

## 用对象为 DataFrame 的字典

```
1 dates = pd.date_range('20190401', periods=4)
2
3 data = {'开盘价': [27.2, 27.65, 27.70, 28],
4         '收盘价': [27.1, 27.55, 27.45, 28.1]}
5 df1 = pd.DataFrame(data, index=dates )
6
```

```
7 data = {'开盘价': [367, 369.8, 378.2, 380.6],  
8         '收盘价': [369.5, 370.1, 380, 382.1]}  
9 df2 = pd.DataFrame( data, index=dates )  
10  
11 p_data = {'海底捞' : df1, '腾讯' : df2}  
12 pn = pd.Panel(p_data)  
13 pn
```

```
<class 'pandas.core.panel.Panel'>  
Dimensions: 2 (items) x 4 (major_axis) x 2 (minor_axis)  
Items axis: 海底捞 to 腾讯  
Major_axis axis: 2019-04-01 00:00:00 to 2019-04-04 00:00:00  
Minor_axis axis: 开盘价 to 收盘价
```

分析上面的 Panel pn

- 有 2 个 DataFrame, items 为 '海底捞' 和 '腾讯'
- 每个 DataFrame 有 4 行 2 列
  - major\_axis 从 2019-04-01 到 2019-04-04
  - minor\_axis 为 '开盘价' 和 '收盘价'

让我们来查看两个 DataFrame 的内容

```
1 pn['海底捞']
```

	开盘价	收盘价
2019-04-01	27.20	27.10
2019-04-02	27.65	27.55
2019-04-03	27.70	27.45
2019-04-04	28.00	28.10

```
1 pn['腾讯']
```

	开盘价	收盘价
2019-04-01	367.0	369.5
2019-04-02	369.8	370.1
2019-04-03	378.2	380.0

上面这种 Panel 类型的数据在量化投资中还蛮常见，比如我们需要 **10 个股票在 1 年时期** 的 **OHLC 价格** (Open, High, Low, Close)，Panel 的 `Items`, `Major_axis` 和 `Minor_axis` 正好可以存储这样的三维数据。如果 Panel 要废掉，那用什么容器来储存三维数据呢？

用多层索引 (Multi-index) 的 DataFrame！

```

1 df = pd.concat([df1, df2])
2 code = ['海底捞', '腾讯']
3 midx = [ (c, d) for c in code for d in dates ]
4 df.index =pd.MultiIndex.from_tuples( midx )
5 df

```

		开盘价	收盘价
海底捞	2019-04-01	27.20	27.10
	2019-04-02	27.65	27.55
	2019-04-03	27.70	27.45
	2019-04-04	28.00	28.10
腾讯	2019-04-01	367.00	369.50
	2019-04-02	369.80	370.10
	2019-04-03	378.20	380.00
	2019-04-04	380.60	382.10

首先用 `concat()` 函数 (下帖的内容) 将 `df1` 和 `df2` 连接起来；再用「列表解析法」生成 `midx`，它是一个元组的列表，`c` 是股票代码，`d` 是日期；最后放入 `MultiIndex.from_tuples()` 生成有多层索引的 DataFrame。

## 2.2 一步登天法

不喜欢量化的读者可跳过本节，不影响本帖的完整性。

上节都是手敲一些数据来创建「多维数据表」的，现实中做量化分析时，数据量都会很大，一般都是从量化平台中或者下载好的 csv 中直接读取。本节介绍如何从量化平台

「**万矿**」中读取数据来创建「多维数据表」的。

首先在 <https://www.windquant.com> 注册一个账号，点击「研究」后在点开一个 Notebook 作为你的研究环境 (这是要夸奖一下万矿的 Notebook 体验真的不错，而且数据质量方面还有万德保证)。



接着必须加载 WindPy，然后执行 `w.start()` 启动 API 接口：

```
1 from WindPy import *
2 w.start()
```

```
COPYRIGHT (C) 2017 Wind Information Co., Ltd. ALL RIGHTS RESERVED.
IN NO CIRCUMSTANCE SHALL WIND BE RESPONSIBLE FOR ANY DAMAGES OR LOSSES
CAUSED BY USING WIND QUANT API FOR PYTHON.
```

WindPy 里面有几个获取数据的核心函数，分别是

- 日期序列函数 `wsd`
- 多维数据函数 `wss`
- 行情数据函数 `wsq`
- 分钟序列数据函数 `wsi`

### 日期序列函数 `wsd`

该函数支持股票、债券、基金、期货、指数等多种证券的基本资料、股东信息、市场行情、证券分析、预测评级、财务数据等各种数据，可以支持取**单品种单指标、多品种单指标和单品种多指标**的时间序列数据 (注：不支持**多品种多指标**)。函数定义如下

```
w.wsd(security, fields, startdate, enddate, options)
```

- `security` = 证券代号，可以是 str 或 list
- `fields` = 指标，可以是 str 或 list
- `startdate` = 起始日，可以是 str 或 datetime

- enddate = 起始日, 可以是 str 或 datetime
- options = 一些特定设置

## 单品种单指标

获取平安银行在 2019-04-01 到 2019-04-04 的收盘价。

```
1 code = "000001.SZ"
2 factors = ["close"]
3 startDate = "2019-04-01"
4 endDate = "2019-04-04"
5 data = w.wsd(code, factors, startDate, endDate, usedf=True )
6 data
```

```
(0, CLOSE
2019-04-01 00:00:00.005 13.18
2019-04-02 00:00:00.005 13.36
2019-04-03 00:00:00.005 13.44
2019-04-04 00:00:00.005 13.86)
```

## 知识点

当 usedf=True 时返回元组

- 元组第一个元素为 ErrorCode, 其为 0 时表示数据获取正常
- 元组第二个元素为获取的数据 DataFrame, 其中 index 列为时间, columns 为参数 Fields 各指标

上面结果 errorcode = 0, 要获取 DataFrame 只需访问 data[1]

```
1 data[1]
```

	CLOSE
2019-04-01 00:00:00.005	13.18
2019-04-02 00:00:00.005	13.36
2019-04-03 00:00:00.005	13.44
2019-04-04 00:00:00.005	13.86

## 单品种多指标

获取平安银行在 2019-04-01 到 2019-04-04 的开盘价、最低价、最高价和收盘价。

```
1 code = "000001.SZ"
2 factors = "open,low,high,close"
3 startDate = "2019-04-01"
4 endDate = "2019-04-04"
5 data = w.wsd(code, factors, startDate, endDate, usedf=True )
6 data[1]
```

	OPEN	LOW	HIGH	CLOSE
2019-04-01 00:00:00.005	12.83	12.83	13.55	13.18
2019-04-02 00:00:00.005	13.28	13.23	13.48	13.36
2019-04-03 00:00:00.005	13.21	13.15	13.45	13.44
2019-04-04 00:00:00.005	13.43	13.43	14.00	13.86

## 多品种单指标

获取平安银行、万科、茅台在 2019-04-01 到 2019-04-04 的收盘价。

```
1 code = ["000001.SZ","000002.SZ","600519.SH"]
2 factors = "close"
3 startDate = "2019-04-01"
4 endDate = "2019-04-04"
5 data = w.wsd(code, factors, startDate, endDate, usedf=True )
6 data[1]
```

	000001.SZ	000002.SZ	600519.SH
2019-04-01 00:00:00.005	13.18	32.07	859.0
2019-04-02 00:00:00.005	13.36	31.99	850.0
2019-04-03 00:00:00.005	13.44	32.49	844.5
2019-04-04 00:00:00.005	13.86	32.22	865.0

## 多维数据函数 wss

该函数同样支持股票、债券、基金、期货、指数等多种证券的基本资料、股东信息、市场行情、证券分析、预测评级、财务数据等各种数据。但是 wss 支持取**多品种多指标某个时间点**的截面数据。函数定义如下

```
w.wss(security, fields, option)
```

- security = 证券代号，可以是 str 或 list
- fields = 指标，可以是 str 或 list
- options = 一些特定设置

获取平安银行、万科、茅台在 2018-12-31 的收盘价、交易量、每股盈余和 profit/GR。

```
1 date = "2018-12-31"
2 codes = ["000001.SZ","000002.SZ","600519.SH"]
3 factors = "close, volume, eps_basic, profitogr"
4 data = w.wss( codes, factors,
5                 "rptDate="+date+";currencyType=", usedf=True)
6 data[1]
```

	CLOSE	VOLUME	EPS_BASIC	PROFITTOGR
000001.SZ	13.54	65535662.0	1.39	21.2636
000002.SZ	31.84	57378188.0	3.06	16.5521
600519.SH	925.20	6110254.0	28.02	49.0025

如果要看财务数据，万矿是取每个季度最后一天作为报告期，如取 2018 年的四个定期报告数据，那报告期设置分别为：

- 一季报：2018-03-31
- 半年报：2018-06-30
- 三季报：2018-09-30
- 年报： 2018-12-31

本例 2018-12-31 是年报的数据。

## 行情数据函数 wsq

该函数支持股票、债券、基金、期货、指数等多种证券品种的实时行情数据。函数定义如下

```
w.wsq(security, fields, func=None)
```

- security = 证券代号, 可以是 str 或 list
- fields = 指标, 可以是 str 或 list
- func = 回调函数

获取易方达深证 100ETF 里所有成分中的各种行情指标。

```
1 ETF = w.wset("allfundhelddetail", "rptdate=20181231;windcode=159901.OF")
2 codes = ETF.Data[2]
3 fields = "rt_last,rt_vol,rt_chg,rt_pct_chg,rt_vwap,rt_ask1,rt_bid1"
4 data = w.wsq(codes, fields)
5
6 data = pd.DataFrame(data.Data,
7                      index=data.Fields,
8                      columns=data.Codes).T
9 data.head(3).append(data.tail(3))
```

	RT_LAST	RT_VOL	RT_BID1	RT_ASK1	RT_VWAP	RT_CHG	RT_PCT_CHG
000001.SZ	13.54	65535662.0	13.54	13.55	13.693	-0.19	-0.0138
000002.SZ	31.84	57378188.0	31.84	31.85	32.109	-0.93	-0.0284
000050.SZ	16.08	63751861.0	16.08	16.09	16.001	0.28	0.0177
300676.SZ	72.01	3987272.0	72.01	72.02	72.530	-1.92	-0.0260
300750.SZ	80.72	8953765.0	80.72	80.73	81.359	-1.52	-0.0185
300760.SZ	128.98	2516269.0	128.96	128.98	128.782	-1.51	-0.0116

读者肯定好奇第一行代码怎么来的? 这里 wset 是专门收集数据集信息的函数, 万矿做的好的东西是又一套 GUI 帮你生成第一行代码, 展示如下:

The screenshot shows a software interface with a toolbar at the top containing buttons for '运行' (Run), '中断' (Interrupt), '保存' (Save), '折叠' (Collapse), '块复制' (Copy Block), '块剪切' (Cut Block), and 'API函数' (API Functions). Below the toolbar, there is a code editor window with the following content:

```
1] 1 ETF = w.wset("allfundhelddetail", "rptdate=20181231;windcode=159901.OF")
2 codes = ETF.Data[2]
3 fields = "rt_last,rt_vol,rt_chg,rt_pct_chg,rt_vwap,rt_ask1,rt_bid1"
4 data = w.wsq(codes, fields)
5
6 data = pd.DataFrame(data.Data, index=data.Fields,
7                      columns=data.Codes).T
8 data.head(3).append(data.tail(3))
```

To the right of the code editor, there is a panel titled 'API函数' (API Functions) which lists several data series and their descriptions:

- WSD 日期序列 59901.OF")
- WSS 多维数据
- WSI 分钟序列 bid1"
- WST 日内跳价
- WSQ 行情数据 > a.Codes).T
- WSEE 板块函数

WSET 数据集

TDays 日期函数

指标查询

	RT_LAST	RT_VOL	RT_BID1	RT_ASK	RT_PCT_CHG
000001.SZ	13.54	65535662.0	13.54	13.55	-0.0138
000002.SZ	31.84	57378188.0	31.84	31.85	-0.0284
000050.SZ	16.08	63751861.0	16.08	16.09	0.0177

点击「API 函数」下面的「WSET 数据集」会带给你以下界面。再选择「ETF 申购成分信息」。

WSET 数据集

Step 1 参数设置 Step 2 生成代码

设置参数

指标名称	中文名称	参数值	参数值释义	编辑
date	日期	2019-04-11	2019-04-11	
windcode	Wind代码	159901.OF	易方达深证100ETF	

显示输出字段

	中文名称	英文名称
<input checked="" type="checkbox"/>	日期	date
<input checked="" type="checkbox"/>	Wind代码	wind_code
<input checked="" type="checkbox"/>	证券名称	sec_name
<input checked="" type="checkbox"/>	数量	volume

按中文、拼音首字母查找

★ 灰色指标表示无权限，需要更多权限，请联系管理员。

上一步 下一步 关闭

点击下一步得到

WSET 数据集

Step 1 参数设置 Step 2 生成代码

```
w.wset("etfconstituent", "date=2019-04-11;windcode=159901.OF")
```

复制到剪切板  
(提示: 使用Ctrl + V或者“右键” 可粘贴到Notebook单元格中)

看到没有第一行代码就这样生成了，获取数据的门槛迅速降低了好多。

### 分钟序列数据函数 wsi

该函数获取选定证券品种的分钟线数据，包含基本行情和部分技术指标的分钟数据，分钟周期为 1-60 min，技术指标参数可以自定义设置。函数定义如下

```
w.wsi(security, fields, starttime = None, endtime = None, options = None)
```

- security = 证券代号，可以是 str 或 list
- fields = 指标，可以是 str 或 list
- startdate = 起始日，可以是 str 或 datetime
- enddate = 起始日，可以是 str 或 datetime
- options = 一些特定设置

获取中金所 IF 股指期货当月连续合约 2019-04-01 09:30:00 开始至 2019-04-01 09:40:00 的 1 分钟数据。

```
1 codes ='IF00.CFE'
2 fields ='open, high, low, close'
3 IF = w.wsi( codes, fields, '2019-04-01 09:30:00', '2019-04-01 09:40:00', "", )
4 IF[1]
```

	open	high	low	close
2019-04-01 09:30:00.005004	3915.0	3926.6	3914.6	3925.6
2019-04-01 09:31:00.005000	3926.2	3942.0	3925.8	3942.0
2019-04-01 09:32:00.004996	3939.8	3939.8	3933.0	3934.4
2019-04-01 09:33:00.005003	3934.4	3943.0	3933.8	3940.2

2019-04-01 09:34:00.004999	3940.2	3943.6	3937.6	3942.4
2019-04-01 09:35:00.004996	3942.4	3942.4	3935.2	3936.2
2019-04-01 09:36:00.005002	3936.4	3943.2	3935.8	3941.0
2019-04-01 09:37:00.004999	3941.4	3945.4	3939.2	3943.6
2019-04-01 09:38:00.005005	3943.6	3960.6	3942.4	3957.6
2019-04-01 09:39:00.005002	3957.6	3969.0	3957.6	3968.8



## 2 数据表的存载

本节讲数据表的「保存」和「加载」，在 NumPy 一贴已经提到过，数据的存载没什么技术含量

- 保存只是为了下次再用处理好的 DataFrame
- 加载可以不用重新再定义 DataFrame

DataFrame 可以被保存为 Excel, csv, SQL 和 HDF5 格式，其语句一看就懂，用 `to_` 数据格式，具体如下：

- `to_excel()`
- `to_csv()`
- `to_sql()`
- `to_hdf()`

如果要加载某种格式的数据到 DataFrame 里，用 `read_` 数据格式，具体如下：

- `read_excel()`
- `read_csv()`
- `read_sql()`
- `read_hdf()`

我们只用 excel 和 csv 格式举例。

### Excel 格式

用 `pd.to_excel` 函数将 DataFrame 保存为 .xlsx 格式，并保存到 ‘Sheet1’ 中，具体写

法如下：

```
pd.to_excel('文件名', '表名')
```

```
1 df = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]))
2 df.to_excel('pd_excel.xlsx', sheet_name='Sheet1')
```

用 `pd.read_excel('文件名', '表名')` 即可加载该文件并存成 DataFrame 形式

```
1 df1 = pd.read_excel('pd_excel.xlsx', sheet_name='Sheet1')
2 df1
```

	0	1	2
0	1	2	3
1	4	5	6

## csv 格式

用 `pd.to_csv` 函数将 DataFrame 保存为 .csv 格式，注意如果 index 没有特意设定，最后不要把 index 值存到 csv 文件中。具体写法如下：

```
pd.to_csv('文件名', index=False)
```

```
1 data = {'Code': ['BABA', '00700.HK', 'AAPL', '600519.SH'],
2         'Name': ['阿里巴巴', '腾讯', '苹果', '茅台'],
3         'Market': ['US', 'HK', 'US', 'SH'],
4         'Price': [185.35, 380.2, 197, 900.2],
5         'Currency': ['USD', 'HKD', 'USD', 'CNY']}
6 df = pd.DataFrame(data)
7 df.to_csv('pd_csv.csv', index=False)
```

用 `pd.read_csv('文件名')` 即可加载该文件并存成 DataFrame 形式

```
1 df2 = pd.read_csv('pd_csv.csv')
2 df2
```

	Code	Name	Market	Price	Currency
0	BABA	阿里巴巴	US	185.35	USD
1	00700.HK	腾讯	HK	380.20	HKD
2	AAPL	苹果	US	197.00	USD
3	600519.SH	茅台	SH	900.20	CNY

如果一开始储存 df 的时候用 `index=True`, 你会发现加载完后的 df2 是以下的样子。

	Unnamed: 0	Code	Name	Market	Price	Currency
0	0	BABA	阿里巴巴	US	185.35	USD
1	1	00700.HK	腾讯	HK	380.20	HKD
2	2	AAPL	苹果	US	197.00	USD
3	3	600519.SH	茅台	SH	900.20	CNY

df2 里面第一栏是 df 的 index, 由于没有具体的 columns 名称, 系统给它一个 "Unnamed: 0"。因此在存储 df 的时候, 如果 `df.index` 没有特意设定, 记住要在 `to_csv()` 中把 `index` 设置为 `False`。



### 3 数据表的索引和切片

由于索引/切片 Series 跟 numpy 数组很类似, 由于 Panel 在未来会被废掉, 因此本节只专注于对 DataFrame 做索引和切片。本节以下面 df 为例做展示。

```
1 symbol = ['BABA', 'JD', 'AAPL', 'MS', 'GS', 'WMT']
2 data = {'行业': ['电商', '电商', '科技', '金融', '金融', '零售'],
3         '价格': [176.92, 25.95, 172.97, 41.79, 196.00, 99.55],
4         '交易量': [16175610, 27113291, 18913154, 10132145, 2626634, 8086946],
5         '雇员': [101550, 175336, 100000, 60348, 36600, 2200000]}
6 df = pd.DataFrame( data, index=symbol )
7 df.name='美股'
8 df.index.name = '代号'
9 df
```

	行业	价格	交易量	雇员
代号				
BABA	电商	176.92	16175610	101550
JD	电商	25.95	27113291	175336
AAPL	科技	172.97	18913154	100000
MS	金融	41.79	10132145	60348
GS	金融	196.00	2626634	36600
WMT	零售	99.55	8086946	2200000

用不同颜色标注了 df 的 index, columns 和 values, 可视图如下:

df

	行业	价格	交易量	雇员
BABA	电商	176.92	16175610	101550
JD	电商	25.95	27113291	175336
AAPL	科技	172.97	18913154	100000
MS	金融	41.79	10132145	60348
GS	金融	196.00	2626634	36600
WMT	零售	99.55	8086946	2200000

DataFrame 的索引或切片可以基于标签 (label-based) , 也可以基于位置 (position-based), 不像 numpy 数组的索引或切片只基于位置。

DataFrame 的索引或切片有四大类:

- 索引单元素:

- 基于标签的 at
- 基于位置的 iat

- 切片 columns:

- 用 . 来切片单列
- 用 [] 来切片单列或多列
- 基于标签的 loc
- 基于位置的 iloc

- 切片 index:

- 用 [] 来切片单行或多行
- 基于标签的 loc
- 基于位置的 iloc

- 切片 index 和 columns:

- 基于标签的 loc
- 基于位置的 iloc

总体规律，基于标签就用 at 和 loc，基于位置就用 iat 和 iloc。下面我们来一类类分析：

### 3.1 索引单元素

两种方法来索引单元素，情况 1 基于标签 at，情况 2 基于位置 iat。

- 情况 1 - df.at['idx\_i', 'attr\_j']
- 情况 2 - df.iat[i, j]

Python 里的中括号 [] 会代表很多意思，比如单元素索引，多元素切片，布尔索引等等，因此让 Python 猜你用的 [] 意图会很低效。如果你想索引单元素，明明白白的用 at 和 iat 效率最高。

#### 情况 1

```
1 df.at['AAPL','价格']
```

172.97

用 `at` 获取「行标签」为 'AAPL' 和「列标签」为 '价格' 对应的元素。

情况 2

```
1 df.iat[2,1]
```

172.97

用 `iat` 获取第 3 行第 2 列对应的元素。

索引单元素的总结图：

iat		0	1	2	3
	at	行业	价格	交易量	雇员
0	BABA	电商	176.92	16175610	101550
1	JD	电商	25.95	27113291	175336
2	AAPL	科技	172.97	18913154	100000
3	MS	金融	41.79	10132145	60348
4	GS	金融	196.00	2626634	36600
5	WMT	零售	99.55	8086946	2200000

`df.at['AAPL', '价格']`

172.97

`df.iat[2,1]`

172.97

## 切片单个 columns

切片**单个** columns 会返回一个 Series，有以下四种情况。情况 1 用点 .；情况 2 用中括号 []；情况 3 基于标签 loc，情况 4 基于位置 iloc。

- 情况 1 - df.attr\_i
- 情况 2 - df['attr\_i']
- 情况 3 - df.loc[:, 'attr\_i']
- 情况 4 - df.iloc[:, i]

情况 1 记住就可以了，没什么可说的。

情况 2 非常像二维 numpy 数组 arr 的切片，用 arr[i] 就能获取 arr 在「轴 0」上的第 i 个元素（一个 1darray），同理 df['attr\_i'] 也能获取 df 的第 i 个 Series。

情况 3 和 4 的 loc 和 iloc 可类比于上面的 at 和 iat。带 i 的基于位置（位置用整数表示，i 也泛指整数），不带 i 的基于标签。里面的冒号 : 代表所有的 index（和 numpy 数组里的冒号意思相同）。

个人建议，如果追求简洁和方便，用 . 和 []；如果追求一致和清晰，用 loc 和 iloc。

### 情况 1

```
1 df.价格
```

```
代号
BABA 176.92
JD 25.95
AAPL 172.97
MS 41.79
GS 196.00
WMT 99.55
Name: 价格, dtype: float64
```

用 . 获取「价格」那一栏下的 Series。

## 情况 2

```
1 df['价格']
```

```
代号  
BABA 176.92  
JD 25.95  
AAPL 172.97  
MS 41.79  
GS 196.00  
WMT 99.55  
Name: 价格, dtype: float64
```

用 `[]` 获取「价格」属性下的 Series。

## 情况 3

```
1 df.loc[:, '交易量']
```

```
代号  
BABA 16175610  
JD 27113291  
AAPL 18913154  
MS 10132145  
GS 2626634  
WMT 8086946  
Name: 交易量, dtype: int64
```

用 `loc` 获取「交易量」属性下的 Series。

## 情况 4

```
1 df.iloc[:, 0]
```

```
代号  
BABA 电商
```

```
JD 电商  
AAPL 科技  
MS 金融  
GS 金融  
WMT 零售  
Name: 行业, dtype: object
```

用 `iLoc` 获取第 1 列下的 Series。

切片单个 `columns` 的总结图：

iloc		0	1	2	3
	loc	行业	价格	交易量	雇员
0	BABA	电商	176.92	16175610	101550
1	JD	电商	25.95	27113291	175336
2	AAPL	科技	172.97	18913154	100000
3	MS	金融	41.79	10132145	60348
4	GS	金融	196.00	2626634	36600
5	WMT	零售	99.55	8086946	2200000

df.价格	df['价格']	df.loc[:, '交易量']	df.iloc[:, 0]
BABA 176.92	BABA 176.92	BABA 16175610	BABA 电商
JD 25.95	JD 25.95	JD 27113291	JD 电商
AAPL 172.97	AAPL 172.97	AAPL 18913154	AAPL 科技
MS 41.79	MS 41.79	MS 10132145	MS 金融
GS 196.00	GS 196.00	GS 2626634	GS 金融
WMT 99.55	WMT 99.55	WMT 8086946	WMT 零售

### 切片多个 columns

切片 **多个** `columns` 会返回一个 sub-DataFrame (原 DataFrame 的子集)，有以下三种情况。情况 1 用中括号 `[]`；情况 2 基于标签 `loc`，情况 3 基于位置 `iloc`。

- 情况 1 - `df[['attr_i', 'attr_j']]`
- 情况 2 - `df.loc[:, 'attr_i':'attr_j']`
- 情况 3 - `df.iloc[:, i:j]`

和切片单个 columns 相比：

- 情况 1 用一个列表来储存一组属性 `'attr_i', 'attr_j'`，然后在放进中括号 `[]` 里获取它们
- 情况 2 用 `'attr_i':'attr_j'` 来获取从属性 i 到属性 j 的 sub-DataFrame
- 情况 3 用 `i:j` 来获取从列 i+1 到列 j 的 sub-DataFrame

个人建议，如果追求简洁和方便，用 `[]`；如果追求一致和清晰，用 `loc` 和 `iloc`。

### 情况 1

```
1 df[ ['雇员', '价格'] ]
```

	雇员	价格
代号		
<b>BABA</b>	101550	176.92
<b>JD</b>	175336	25.95
<b>AAPL</b>	100000	172.97
<b>MS</b>	60348	41.79
<b>GS</b>	36600	196.00
<b>WMT</b>	2200000	99.55

用 `[]` 获取「雇员」和「价格」两个属性下的 sub-DataFrame。

### 情况 2

```
1 df.loc[:, '行业':'交易量']
```

	行业	价格	交易量
代号			
<b>BABA</b>	电商	176.92	16175610
<b>JD</b>	电商	25.95	27113291
<b>AAPL</b>	科技	172.97	18913154
<b>MS</b>	金融	41.79	10132145
<b>GS</b>	金融	196.00	2626634
<b>WMT</b>	零售	99.55	8086946

用 `loc` 获取从属性 ‘行业’ 到 ‘交易量’ 的 sub-DataFrame。

### 情况 3

```
1 df.iloc[:, 0:2]
```

	行业	价格
代号		
<b>BABA</b>	电商	176.92
<b>JD</b>	电商	25.95
<b>AAPL</b>	科技	172.97
<b>MS</b>	金融	41.79
<b>GS</b>	金融	196.00
<b>WMT</b>	零售	99.55

用 `iloc` 获取第 1 和 2 列下的 sub-DataFrame。

切片多个 columns 的总结图：

iloc		0	1	2	3
	loc	行业	价格	交易量	雇员
0	<b>BABA</b>	电商	176.92	16175610	101550
1	<b>JD</b>	电商	25.95	27113291	175336
2	<b>AAPL</b>	科技	172.97	18913154	100000
3	<b>MS</b>	金融	41.79	10132145	60348
4	<b>GS</b>	金融	196.00	2626634	36600
5	<b>WMT</b>	零售	99.55	8086946	2200000

`df[['雇员', '价格']]`

	雇员	价格
<b>BABA</b>	101550	176.92
<b>JD</b>	175336	25.95
<b>AAPL</b>	100000	172.97
<b>MS</b>	60348	41.79
<b>GS</b>	36600	196.00
<b>WMT</b>	2200000	99.55

`df.loc[:, '行业':'交易量']`

	行业	价格	交易量
<b>BABA</b>	电商	176.92	16175610
<b>JD</b>	电商	25.95	27113291
<b>AAPL</b>	科技	172.97	18913154
<b>MS</b>	金融	41.79	10132145
<b>GS</b>	金融	196.00	2626634
<b>WMT</b>	零售	99.55	8086946

`df.iloc[:, 0:2]`

	行业	价格
<b>BABA</b>	电商	176.92
<b>JD</b>	电商	25.95
<b>AAPL</b>	科技	172.97
<b>MS</b>	金融	41.79
<b>GS</b>	金融	196.00
<b>WMT</b>	零售	99.55

## 切片单个 index

切片单个 index 有时会返回一个 Series，有以下两种情况。情况 1 基于标签 loc，情况 2 基于位置 iloc。

- 情况 1 - `df.loc['idx_i', :]`
- 情况 2 - `df.iloc[i, :]`

切片单个 index 有时会返回一个只有一行的 DataFrame，有以下两种情况。情况 3 用中括号 [] 加「位置」，情况 4 用中括号 [] 加「标签」。

- 情况 3 - `df[i:i+1]`
- 情况 4 - `df['idx_i':'idx_i']`

情况 1 和 2 的 loc 和 iloc 可类比于上面的 at 和 iat。带 i 的基于位置（位置用整数表示，i 也泛指整数），不带 i 的基于标签。里面的冒号 : 代表所有的 columns (和 numpy 数组里的冒号意思相同)。

情况 3 用中括号 [] 加「位置」，位置 `i:i+1` 有前闭后开的性质。如果要获取第 `i+1` 行，需要用 `i:i+1`。

情况 4 用中括号 [] 加「标签」，标签没有前闭后开的性质。如果要获取标签 i，只需要用 `'idx_i':'idx_i'`。为什么不能只用 `'idx_i'` 呢？原因是 Python 会把 `df['idx_i']` 当成切片 columns，然后发现属性中没有 `'idx_i'` 这一个字符，会报错的。

个人建议，只用 loc 和 iloc。情况 3 太麻烦，获取一行还要用 `i:i+1`。情况 4 的 `df['idx_i']` 很容易和切片 columns 中的语句 `df['attr_j']` 混淆。

## 情况 1

```
1 df.loc[ 'GS', : ]
```

```
行业 金融  
价格 196  
交易量 2626634  
雇员 36600  
Name: GS, dtype: object
```

用 `loc` 获取标签为 'GS' 的 Series。(GS = Goldman Sachs = 高盛)

#### 情况 2

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

```
行业 金融  
价格 41.79  
交易量 10132145  
雇员 60348  
Name: MS, dtype: object
```

用 `iloc` 获取第 4 行下的 Series。(MS = Morgan Stanley = 摩根斯坦利)

#### 情况 3

```
1 df[1:2]
```

	行业	价格	交易量	雇员
代号				
<b>JD</b>	电商	25.95	27113291	175336

用 `[1:2]` 获取第 2 行的 sub-DataFrame (只有一行)。

#### 情况 4

```
1 df['JD':'JD']
```

	行业	价格	交易量	雇员
代号				
<b>JD</b>	电商	25.95	27113291	175336

用 `[‘JD’ : ‘JD’]` 获取标签为 ‘JD’ 的 sub-DataFrame (只有一行)。

切片单个 index 的总结图:

The diagram illustrates four methods to slice a single index from a DataFrame:

- df[1:2]**: Returns a sub-DataFrame with rows 1 and 2.
- df.loc['GS', :]**: Returns a sub-DataFrame for the row labeled 'GS'.
- df.iloc[3, :]**: Returns a sub-DataFrame for the row at index 3.
- df['JD':'JD']**: Returns a sub-DataFrame for the row labeled 'JD'.

iloc		0	1	2	3
	loc	行业	价格	交易量	雇员
0	BABA	电商	176.92	16175610	101550
1	JD	电商	25.95	27113291	175336
2	AAPL	科技	172.97	18913154	100000
3	MS	金融	41.79	10132145	60348
4	GS	金融	196.00	2626634	36600
5	WMT	零售	99.55	8086946	2200000

行业	价格	交易量	雇员
JD	电商	25.95	27113291 175336

行业	金融
价格	196.00
交易量	2626634
雇员	36600

行业	金融
价格	41.79
交易量	10132145
雇员	60348

## 切片多个 index

切片 **多个** index 会返回一个 sub-DataFrame，有以下四种情况。情况 1 用中括号 `[]` 加「位置」，情况 2 用中括号 `[]` 加「标签」，情况 3 基于标签 `loc`，情况 4 基于位置 `iloc`。

- 情况 1 - `df[i:j]`
- 情况 2 - `df['idx_i':'idx_j']`
- 情况 3 - `df.loc['idx_i':'idx_j', :]`
- 情况 4 - `df.iloc[i:j, :]`

和切片单个 index 相比：

- 情况 1 用 `[i:j]` 来获取行  $i+1$  到行  $j$  的 sub-DataFrame
- 情况 2 用 `['idx_i': 'idx_j']` 来获取标签  $i$  到标签  $j$  的 sub-DataFrame
- 情况 3 用 `loc` 加 `'idx_i': 'idx_j'` 来获取从标签  $i$  到标签  $j$  的 sub-DataFrame
- 情况 4 用 `iloc` 加 `i:j` 来获取从行  $i+1$  到行  $j$  的 sub-DataFrame

个人建议，只用 `loc` 和 `iloc`。情况 1 和 2 的 `df[]` 很容易混淆中括号 `[]` 里的到底是切片 `index` 还是 `columns`。

### 情况 1

```
1 df[ 1:4 ]
```

	行业	价格	交易量	雇员
代号				
<b>JD</b>	电商	25.95	27113291	175336
<b>AAPL</b>	科技	172.97	18913154	100000
<b>MS</b>	金融	41.79	10132145	60348

用 `[1:4]` 获取第 2 到 4 行的 sub-DataFrame。

### 情况 2

```
1 df[ 'GS':'WMT' ]
```

	行业	价格	交易量	雇员
代号				
<b>GS</b>	金融	196.00	2626634	36600
<b>WMT</b>	零售	99.55	8086946	2200000

用 `['GS':'WMT']` 获取 标签 从 'GS' 到 'WMT' 的 sub-DataFrame。  
(WMT = Walmart = 沃尔玛)

### 情况 3

```
1 df.loc[ 'MS':'GS', : ]
```

	行业	价格	交易量	雇员
代号				
MS	金融	41.79	10132145	60348
GS	金融	196.00	2626634	36600

用 `loc` 获取标签从 'MS' 到 'GS' 的 sub-DataFrame。注意 'MS':'GS' 要按着 index 里面元素的顺序，要不然会返回一个空的 DataFrame，比如：

```
1 df.loc[ 'MS':'JD', : ]
```

	行业	价格	交易量	雇员
代号				

#### 情况 4

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

	行业	价格	交易量	雇员
代号				
JD	电商	25.95	27113291	175336
AAPL	科技	172.97	18913154	100000

用 `iloc` 获取第 2 到 3 行的 sub-DataFrame。

切片多个 index 的总结图：

iloc		0	1	2	3
	loc	行业	价格	交易量	雇员
0	BABA	电商	176.92	16175610	101550
1	JD	电商	25.95	27113291	175336
2	AAPL	科技	172.97	18913154	100000
3	MS	金融	41.79	10132145	60348

4	GS	金融	196.00	2626634	36600
5	WMT	零售	99.55	8086946	2200000

df[1:4]		行业	价格	交易量	雇员	df.loc['MS:GS',:]			
JD	电商	25.95	27113291	175336	MS	金融	41.79	10132145	60348
AAPL	科技	172.97	18913154	100000	GS	金融	196.00	2626634	36600
MS	金融	41.79	10132145	60348					
df['GS':'WMT']		行业	价格	交易量	雇员	df.iloc[1:3,:]			
GS	金融	196.00	2626634	36600	JD	电商	25.95	27113291	175336
WMT	零售	99.55	8086946	2200000	AAPL	科技	172.97	18913154	100000

### 3.4 切片 index 和 columns

切片多个 index 和 columns 会返回一个 sub-DataFrame，有以下两种情况。情况 1 基于标签 `loc`，情况 2 基于位置 `iloc`。

- 情况 1 - `df.loc['idx_i':'idx_j', 'attr_k':'attr_l']`
- 情况 2 - `df.iloc[i:j, k:l]`

清清楚楚，明明白白，用 `loc` 和 `iloc`。

#### 情况 1

```
1 df.loc[ 'GS':'WMT', '价格': ]
```

	价格	交易量	雇员
代号			
GS	196.00	2626634	36600
WMT	99.55	8086946	2200000

用 `loc` 获取行标签从 'GS' 到 'WMT'，列标签从 '价格' 到最后的 sub-DataFrame。

#### 情况 2

```
1 df.iloc[ :2, 1:3 ]
```

	价格	交易量
代号		
BABA	176.92	16175610
JD	25.95	27113291

用 `iloc` 获取第 1 到 2 行，第 1 到 2 列的 sub-DataFrame。

切片 index 和 columns 的总结图：

iat		0	1	2	3
	at	行业	价格	交易量	雇员
0	BABA	电商	176.92	16175610	101550
1	JD	电商	25.95	27113291	175336
2	AAPL	科技	172.97	18913154	100000
3	MS	金融	41.79	10132145	60348
4	GS	金融	196.00	2626634	36600
5	WMT	零售	99.55	8086946	2200000

`df.loc['GS:WMT', '价格':]`

	价格	交易量	雇员
GS	196.00	2626634	36600
WMT	99.55	8086946	2200000

`df.iloc[:2, 1:3]`

	价格	交易量
BABA	176.92	16175610
JD	25.95	27113291

### 3.5 高级索引

高级索引 (advanced indexing) 可以用 **布尔索引** (boolean indexing) 和 **调用函数** (callable function) 来实现，两种方法都返回一组“正确”的索引，而且可以和 `loc`, `iloc`, `[]` 一起套用，具体形式有以下常见几种：

- `df.loc[布尔索引, :]`
- `df.iloc[布尔索引, :]`
- `df[布尔索引]`
- `df.loc[调用函数, :]`
- `df.iloc[调用函数, :]`
- `df[调用函数]`

还有以下罕见几种：

- `df.loc[:, 布尔索引]`
- `df.iloc[:, 布尔索引]`
- `df.loc[:, 调用函数]`
- `df.iloc[:, 调用函数]`

读者可以想一想为什么第一组形式「常见」而第二组形式「罕见」呢？(Hint: 看看两组里冒号：在不同位置，再想想 DataFrame 每一行和每一列中数据的特点)

## 布尔索引

在【[数组计算之 NumPy \(上\)](#)】提过，布尔索引就是用一个由布尔类型值组成的数组来选择元素的方法。

当我们要过滤掉雇员小于 100,000 人的公司，我们可以用 `loc` 加上布尔索引。

```
1 print( df.雇员 >= 100000 )
2 df.loc[ df.雇员 >= 100000, : ]
```

代号  
BABA True  
JD True  
AAPL True  
MS False  
GS False  
WMT True  
Name: 雇员, dtype: bool

	行业	价格	交易量	雇员
代号				
BABA	电商	176.92	16175610	101550

JD	电商	25.95	27113291	175336
AAPL	科技	172.97	18913154	100000
WMT	零售	99.55	8086946	2200000

一种更简便的表达形式是用 `df[]`，但是我个人不喜欢 `[]`，总觉得会引起「到底在切片 index 还是 columns」的歧义。

```
1 df[ df.雇员 >= 100000 ]
```

	行业	价格	交易量	雇员
代号				
BABA	电商	176.92	16175610	101550
JD	电商	25.95	27113291	175336
AAPL	科技	172.97	18913154	100000
WMT	零售	99.55	8086946	2200000

现在来看一个「罕见」例子，假如我们想找到所有值为整数型的 columns

```
1 print( df.dtypes == 'int64' )
2 df.loc[ :, df.dtypes == 'int64' ]
```

```
行业      False
价格      False
交易量     True
雇员      True
dtype: bool
```

	交易量	雇员
代号		
BABA	16175610	101550
JD	27113291	175336
AAPL	18913154	100000
MS	10132145	60348
GS	2626634	36600
WMT	8086946	2200000

## 调用函数

调用函数是只能有一个参数 (DataFrame, Series) 并返回一组索引的函数。因为调用函数定义在 `loc`, `iloc`, `[]` 里面，因此它就像在【Python 入门篇 (下)】提过的匿名函

数。

当我们要找出交易量大于平均交易量的所有公司，我们可以用 `loc` 加上匿名函数 (这里 `x` 代表 `df`)。

```
1 df.loc[ lambda x: x.交易量 > x.交易量.mean() , : ]
```

	行业	价格	交易量	雇员
代号				
<b>BABA</b>	电商	176.92	16175610	101550
<b>JD</b>	电商	25.95	27113291	175336
<b>AAPL</b>	科技	172.97	18913154	100000

在上面基础上再加一个条件 -- 价格要在 100 之上 (这里 `x` 还是代表 `df`)

```
1 df.loc[ lambda x: (x.交易量 > x.交易量.mean())
2           & (x.价格 > 100), : ]
```

	行业	价格	交易量	雇员
代号				
<b>BABA</b>	电商	176.92	16175610	101550
<b>AAPL</b>	科技	172.97	18913154	100000

最后来看看价格大于 100 的股票 (注意这里 `x` 代表 `df.价格`)

```
1 df.价格.loc[ lambda x: x > 100 ]
```

```
代号
BABA 176.92
AAPL 172.97
GS 196.00
Name: 价格, dtype: float64
```

### 3.6 多层索引

在 Panel 那节已经提到过，多层索引可以将「低维数据」升维到「高维数据」，此外，

多层索引还可以。。。

## 多层索引 Series

首先定义一个 Series，注意它的 index 是一个二维列表，列表第一行 dates 作为第一层索引，第二行 codes 作为第二层索引。

```
1 price = [190, 32, 196, 192, 200, 189, 31, 30, 199]
2 dates = ['2019-04-01']*3 + ['2019-04-02']*2
3     +['2019-04-03']*2 + ['2019-04-04']*2
4 codes = ['BABA', 'JD', 'GS', 'BABA', 'GS', 'BABA', 'JD', 'JD', 'GS']
5
6 data = pd.Series( price,
7                   index=[ dates, codes ])
8 data
```

```
2019-04-01 BABA 190
                JD 32
                GS 196
2019-04-02 BABA 192
                GS 200
2019-04-03 BABA 189
                JD 31
2019-04-04 JD 30
                GS 199
dtype: int64
```

这个 Series 存储了四天里若干股票的价格，2019-04-01 储存了阿里巴巴、京东和高盛的股价，2019-04-04 只储存了京东和高盛的股价。试想，如果不用多层索引的 Series，我们需要用一个 DataFrame 来存储在这样的数据，把 index 设置成 dates，把 columns 设置成 codes。

让我们看看 Series 的多层 index 是如何表示的

```
1 data.index
```

```
MultiIndex(levels=[[ '2019-04-01', '2019-04-02', '2019-04-03', '2019-04-04'],
                  ['BABA', 'GS', 'JD']],
             labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3],
```

```
[0, 2, 1, 0, 1, 0, 2, 2, 1]])
```

输出是一个 MultiIndex 的对象，里面有 levels 和 labels 二类信息。

### 知识点

索引既然分多层，那么肯定分「内层」和「外层」把，levels 就是描述层的先后的。levels 是一个二维列表，每一行只存储着「唯一」的索引信息：

- dates 是第一层索引，有 4 个「唯一」元素
- codes 是第二层索引，有 3 个「唯一」元素

但是 data 里面有九行啊，4 个 dates 和 3 个 codes 怎么能描述这九行信息呢？这就需要 labels 了。labels 也是一个二维列表：

- 第一行储存 dates 每个元素在 data 里的位置索引
- 第二行储存 codes 每个元素在 data 里的位置索引

用 `[]` 加第一层索引可以获取第一层信息。

```
1 data['2019-04-02']
```

```
BABA 192
GS    200
dtype: int64
```

同理，用 `loc` 加第一层索引也可以切片获取第一层信息。

```
1 data.loc['2019-04-02':'2019-04-04']
```

```
2019-04-02 BABA 192
              GS 200
2019-04-03 BABA 189
              JD 31
2019-04-04 JD 30
              GS 199
dtype: int64
```

此外，切片还可以在不同层上进行，下面 `loc` 中的冒号`:`表示第一层所有元素，'GS' 表示第二层标签为 'GS'。

```
1 data.loc[ :, 'GS' ]
```

```
2019-04-01 196  
2019-04-02 200  
2019-04-04 199  
dtype: int64
```

## 多层索引 DataFrame

Series 只有 index，上面刚介绍完多层 index，DataFrame 有 index 和 columns，它们可以设置成多层吗？下面代码用 MultiIndex 函数创建「多层 index」`midx` 和「多层columns」`mcol`。

`midx` 和 `mcol` 都是对象，各种都有 `levels`, `labels`, `names` 等性质。

```
1 data = [ ['电商', 101550, 176.92, 16175610],  
2     ['电商', 175336, 25.95, 27113291],  
3     ['金融', 60348, 41.79, 10132145],  
4     ['金融', 36600, 196.00, 2626634] ]  
5  
6 midx = pd.MultiIndex(  
7     levels=[[ '中国', '美国'],  
8             ['BABA', 'JD', 'GS', 'MS']],  
9     labels=[[0,0,1,1],[0,1,2,3]],  
10    names=[ '地区', '代号'])  
11  
12 mcol = pd.MultiIndex(  
13     levels=[[ '公司数据', '交易数据'],  
14             ['行业', '雇员', '价格', '交易量']],  
15     labels=[[0,0,1,1],[0,1,2,3]],  
16     names=[ '概括', '细分'])  
17  
18 df = pd.DataFrame(data, index=midx, columns=mcol)  
19 df
```

	概括	公司数据		交易数据	
	细分	行业	雇员	价格	交易量
地区	代号				
中国	BABA	电商	101550	176.92	16175610
	JD	电商	175336	25.95	27113291
美国	GS	金融	60348	41.79	10132145
	MS	金融	36600	196.00	2626634

这个 DataFrame 的 index 和 columns 都有两层，严格来说是个四维数据。下面看看如何进行「多层索引」的操作吧。

在第一层 columns 的‘公司数据’和第二层 columns 的‘行业’做索引，得到一个含两层 index 的 Series。

```
1 # 1st level-1 column, 2nd level-2 column
2 df['公司数据','行业']
```

```
地区 代号
中国 BABA 电商
        JD    电商
美国 GS   金融
        MS   金融
Name: (公司数据, 行业), dtype: object
```

在第一层 index 的‘中国’做切片，得到一个含两层 columns 的 DataFrame。

```
1 df.loc['中国'].loc['BABA':'JD']
```

	概括	公司数据		交易数据	
	细分	行业	雇员	价格	交易量
	代号				
中国	BABA	电商	101550	176.92	16175610
	JD	电商	175336	25.95	27113291

## 调位 level

如果你不喜欢 index level 的顺序，可用 `swaplevel` 将它们调位。

```
1 df.swaplevel('地区', '代号')
```

	概括	公司数据		交易数据	
	细分	行业	雇员	价格	交易量
代号	地区				
BABA	中国	电商	101550	176.92	16175610
JD	中国	电商	175336	25.95	27113291
GS	美国	金融	60348	41.79	10132145
MS	美国	金融	36600	196.00	2626634

如果你不喜欢 columns level 的顺序，也可用 `swaplevel` 将它们调位。

```
1 df.columns = df.columns.swaplevel(0,1)
2 df
```

	细分	行业	雇员	价格	交易量
	概括	公司数据	公司数据	交易数据	交易数据
地区	代号				
中国	BABA	电商	101550	176.92	16175610
	JD	电商	175336	25.95	27113291
美国	GS	金融	60348	41.79	10132145
	MS	金融	36600	196.00	2626634

## 重设 index

有时候，一个 DataFrame 的一个或者多个 columns 适合做 index，这时可用 `set_index` 将它们设置为 index，如果要将 index 还原成 columns，那么用 `reset_index`。

看下面这个例子。

```
1 data = {'地区': ['中国', '中国', '美国', '美国'],
2         '代号': ['BABA', 'JD', 'MS', 'GS'],
3         '行业': ['电商', '电商', '金融', '金融'],
4         '价格': [176.92, 25.95, 41.79, 196.00],
5         '交易量': [16175610, 27113291, 10132145, 2626634],
6         '雇员': [101550, 175336, 60348, 36600] }
```

```
7 df = pd.DataFrame( data )
8 df
```

	地区	代号	行业	价格	交易量	雇员
0	中国	BABA	电商	176.92	16175610	101550
1	中国	JD	电商	25.95	27113291	175336
2	美国	MS	金融	41.79	10132145	60348
3	美国	GS	金融	196.00	2626634	36600

将「地区」和「代号」设置为第一层 index 和第二层 index。

```
1 df2 = df.set_index( ['地区','代号'] )
2 df2
```

		行业	价格	交易量	雇员
地区	代号				
中国	BABA	电商	176.92	16175610	101550
	JD	电商	25.95	27113291	175336
美国	MS	金融	41.79	10132145	60348
	GS	金融	196.00	2626634	36600

将所有 index 变成 columns。

```
1 df2.reset_index()
```

	地区	代号	行业	价格	交易量	雇员
0	中国	BABA	电商	176.92	16175610	101550
1	中国	JD	电商	25.95	27113291	175336
2	美国	MS	金融	41.79	10132145	60348
3	美国	GS	金融	196.00	2626634	36600



## 4 总结 <

Pandas 里面的数据结构是多维数据表，细化为一维的 Series，二维的 DataFrame，

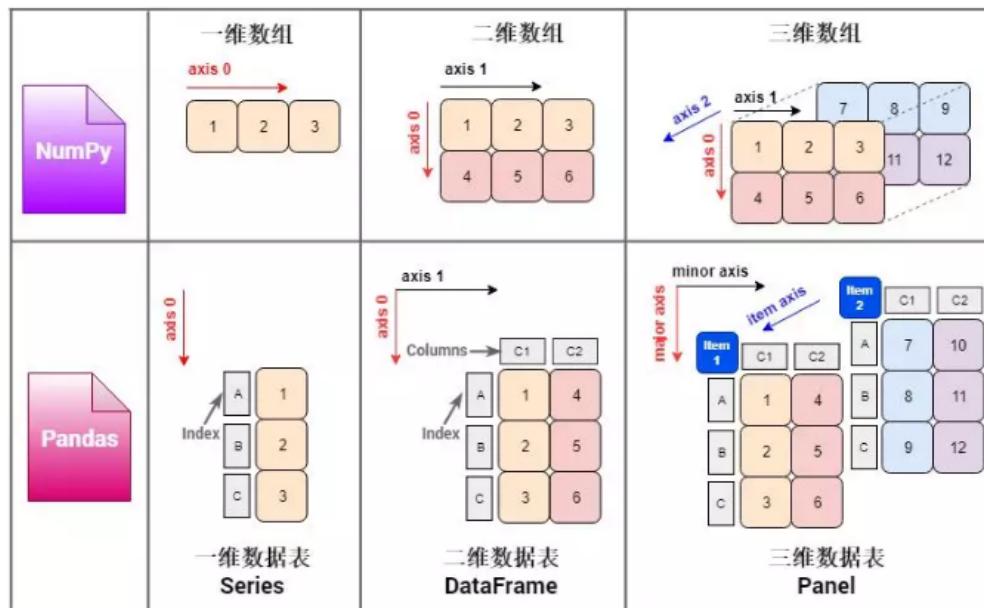
三维的 Panel。

多维数据表 = 多维数组 + 描述

其中

- Series = 1darray + index
- DataFrame = 2darray + index + columns
- Panel = 3darray + index + columns + item

pd 多维数据表和 np 多维数组之间的类比关系如下图所示。



【创建数据表】 创建 Series, DataFrame, Panel 用下面语句

- `pd.Series(x, index=idx)`
- `pd.DataFrame(x, index=idx, columns=col)`
- `pd.Panel(x, item=item, major_axis=n1, minor_axis=n2)`

DataFrame 由多个 Series 组成，Panel 有多个 DataFrame 组成。Series 非常类似于一维的 DataFrame，Panel 未来会被废掉，因此学 Pandas 把注意力放在 DataFrame 上即可。

**【索引和切片数据表】**在索引或切片 DataFrame，有很多种方法。最好记的而不易出错的是用基于位置的 `at` 和 `loc`，和基于标签的 `iat` 和 `iloc`，具体来说，索引用 `at` 和 `iat`，切片用 `loc` 和 `iloc`。带 `i` 的基于位置，不带 `i` 的基于标签。

用 `MultiIndex` 可以创建多层索引的对象，获取 DataFrame `df` 的信息可用

- `df.loc[1st].loc[2nd]`
- `df.loc[1st].iloc[2nd]`
- `df.iloc[1st].loc[2nd]`
- `df.iloc[1st].iloc[2nd]`

要调换 `level` 可用

- `df.index.swaplevel(0, 1)`
- `df.columns.swaplevel(0, 1)`

要设置和重设 `index` 可用

- `df.set_index( columns )`
- `df.reset_index`

---

下篇讨论 Pandas 系列的后三节，分别是

- 「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」

Stay Tuned!