第5章　基本数据操作

Jupyter金融应用从入门到实践

基本数据结构—元组

元组（tuple）是一种使用小括号“()”进行对象定义的高级数据结构。当元组创建之后，其元素不能被修改。元组内部的元素不受类型的限制，整数、小数、字符串等可以同时存在。

1．元组的创建

```
[In]
1.  t = ('上证指数', '000001', '1990-12-19', 100)
2.  type(t)  # 显示数据类型
[Out]
tuple
```

2．元组的访问查询

```
[In]
1.  t[1]
[Out]
'000001'
[In]
2.  type(t[1])  # 因为数据被单引号括住，所以是字符串类型
[Out]
str
```

3．元组的修改

```
[In]
1.  t2 = (1519, 42274.41)
2.  t = t + t2
3.  t
[Out]
('上证指数', '000001', '1990-12-19', 100, 1519, 42274.41)
```

方法和函数	功能
count()	计算某元素出现的次数
index()	查找某元素首次出现的位置
len()	计算数据长度
max()	找出最大值
min()	找出最小值
tuple()	将其他类型的数据转换成元组

基本数据结构—列表

列表（list）是一种比元组更灵活的数据结构。相比于元组，列表内容允许更改。

1．列表的创建

```
[In]
1.  li = ['上证指数', '000001', '1990-12-19', 100]
2.  li

[Out]
['上证指数', '000001', '1990-12-19', 100]
```

2．列表的访问和切片

```
[In]
li[1]

[Out]
'000001'
```

```
[In]
li[0:2]

[Out]
['上证指数', '000001']
```

3．列表的常用函数

方法和函数	功能
list()	将其他数据类型转换为列表类型
reverse()	将整个列表翻转
sort()	对列表进行排序

4．列表的修改

方法和函数	功能
append()	在现有列表后任意添加一个对象
extend()	将其他列表内的元素添加到当前列表
insert()	将元素插入指定位置
remove()	删除第一次出现的元素
pop()	删除指定元素并获取该元素
del()	删除整个对象
clear()	保留列表对象，仅清除内容

基本数据结构—字典

字典（dictionary）是一种按照键值存取的数据类型，它是一种可以变容量的结构。字典与列表的最主要区别是，列表是有序的，而字典是无序的，字典采用键码寻找元素，字典内的键码不允许重复

1．字典的创建

```
[In]
1.  d = {'name':'上证指数',
2.       'code':'000001',
3.       'date':'1990-12-19',
4.       'point':100}
5.  type(d)
[Out]
dict
```

2．字典的访问

```
[In]
d['code']
[Out]
'000001'
```

```
[In]
d.keys()
[Out]
dict_keys(['name', 'code', 'date', 'point'])
```

```
[In]
d.values()
[Out]
dict_values(['上证指数', '000001', '1990-12-19', 100])
```

表 5-4　字典的常用方法和函数

方法和函数	功能
del()	删除整个对象
clear()	保留对象，仅清除内容
copy()	复制当前字典
update()	更新参数中提及的键码，未提及的仍然保留

示例 1：通过直接对新键码赋值来增加新的元素。

```
[In]
1.  d['number of stocks'] = 1518
2.  d
[Out]
{'name': '上证指数',
'code': '000001',
'date': '1990-12-19',
'point': 100,
'number of stocks': 1518}
```

示例 2：修改某个元素的值。

```
[In]
1.  d['number of stocks'] = 1519
2.  d
[Out]
{'name': '上证指数',
'code': '000001',
'date': '1990-12-19',
'point': 100,
'number of stocks': 1519}
```

基本数据结构—集合

集合（set）是数学中常用的概念，是集合论的主要研究对象，现代的集合一般被定义为由一个或多个确定的元素所构成的整体。集合是一个无序的不重复元素序列。

1．集合的创建

```
[In]
1.  s1 = set(['a','b','c','ab','ac','a'])
2.  s1
[Out]
{'a', 'ab', 'ac', 'b', 'c'}
```

示例 1：为两个集合取并集，等同于 s1 | s2。

```
[In]
1.  s2 = set(['ab','ac','bc','c'])
2.  s1.union(s2)
[Out]
{'a', 'ab', 'ac', 'b', 'bc', 'c'}
```

示例 2：为两个集合取交集，等同于 s1 & s2。

```
[In]
    s1.intersection(s2)
[Out]
{'ab', 'ac', 'c'}
```

示例 3：找出在集合 s1 中但不在集合 s2 中的元素，等同于 s1 - s2。

```
[In]
    s1.difference(s2)
[Out]
{'a', 'b'}
```

示例 4：找出只在二集合之一中的元素，等同于 s1 ^ s2。

```
[In]
    s1.symmetric_difference(s2)
[Out]
{'a', 'b', 'bc'}
```

示例 1：在集合中添加元素。

```
[In]
1.  s1.add('d')
2.  s1
[Out]
{'a', 'ab', 'ac', 'b', 'c', 'd'}
```

示例 2：更新集合中的内容。

```
[In]
1.  s1.update('d','e')
2.  s1
[Out]
{'a', 'ab', 'ac', 'b', 'c', 'd', 'e'}
```

示例 3：删除集合中的元素。

```
[In]
1.  s1.remove('d')
2.  s1
[Out]
{'a', 'ab', 'ac', 'b', 'c', 'e'}
```

示例 4：忽略集合中的元素。

```
[In]
1.  s1.discard('e')
2.  s1
[Out]
{'a', 'ab', 'ac', 'b', 'c'}
```

NumPy库

NumPy（全称为Numerical Python）是Python的一个扩展库。它是一个开源的项目，在2005年由特拉维斯·奥利潘特（Travis Oliphant）在Numeri基础上结合Numarray的特点，进一步开发从而发展成NumPy。NumPy运行速度非常快，支持高维度的数组与矩阵运算，并提供大量的数学函数库。

1．ndarray的创建

示例 1：创建一维数组。

```
[In]
1.  import numpy as np
2.  a = np.array([1,2,3])
3.  a
[Out]
array([1, 2, 3])
```

示例 2：创建二维数组。

```
[In]
1.  b = np.array([[1, 2],  [3, 4]])
2.  b
[Out]
array([[1, 2],
       [3, 4]])
```

示例 3：创建元素都是 0 的数组。

```
[In]
1.  a = np.zeros((2,3))
2.  a
[Out]
array([[0., 0., 0.],
       [0., 0., 0.]])
```

示例 4：创建元素都是 1 的数组。

```
[In]
1.  a = np.ones((2,3))
2.  a
[Out]
array([[1., 1., 1.],
       [1., 1., 1.]])
```

示例 5：创建数组，设置元素起点、终点和步长，示例中是起点为 2，终点为 5，步长为 1 的数组，边界值包含左边的起点 2，不包含右边的终点 5。

```
[In]
1.  a = np.arange(2,5,1)
2.  a
[Out]
array([2, 3, 4])
```

示例 6：创建等差数列，示例中为起点为 1，终点为 5，有 5 个数的等差数列。

```
[In]
1.  a = np.linspace(1,5,5)
2.  a
[Out]
array([1., 2., 3., 4., 5.])
```

示例 7：创建等比数列，示例中为起点为 5 的 0 次方，终点为 5 的 3 次方，包含 4 个数的等比数列，若不设置则 base 参数默认为 10。

```
[In]
1.  a = np.logspace(0,3,4,base=5)
2.  a
[Out]
array([  1.,   5.,  25., 125.])
```

示例 8：同时存放字符串和数值，数值被迫转化为字符串类型，比如例子中的数值（100）变成了字符串('100')。

```
[In]
1.  a = np.array(['上证指数','000001','1990-12-19',100])
2.  a
[Out]
array(['上证指数', '000001', '1990-12-19', '100'], dtype='<U10')
```

在ndarray的使用过程中，可以使用中括号“[]”来定位并选择需要的部分，要注意的是多维数组的结构实际上是数组中包含着数组，而定位操作返回的是下级对象，即二维数组定义返回的是一维数组

2. ndarray的访问

示例 1：一维数组定位。

```
[In]
1.  import numpy as np
2.  a=np.array([1,2])
3.  a[1]
[Out]
2
```

示例 2：二维数组一次定位。

```
[In]
1.  b=np.array([[1,2],[3,4]])
2.  b[1]
[Out]
array([3, 4])
```

示例 3：二维数组二次定位。

```
[In]
    b[1][1]
[Out]
4
```

示例 4：二维数组切片。

```
[In]
    b[1:]
[Out]
array([[3, 4]])  #虽然只有一组数但依然
```

3. ndarray的修改和删除

示例 1：对 n 维数组定位后，修改其中的值。

```
[In]
1.  b=np.array([[1,2],[3,4]])
2.  b[1][0] = 5
3.  b
[Out]
array([[1, 2],
       [5, 4]])
```

示例 2：对数组横向拼接，即行数不变，将元素拼接到对应的行之后。

```
[In]
1.  a = np.array([[1,2],[3,4]])
2.  b = np.array([[5,6],[7,8]])
3.  np.vstack((a,b))
[Out]
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

示例 3：对数组纵向拼接，即列数不变，将元素拼接到对应的列元素之中。

```
[In]
1.  np.hstack((a,b))
[Out]
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

示例 4：用一个 n 行列向量和一个 m 列行向量构造出一个 $n \times m$ 矩阵。

```
[In]
1.  a = np.array([[1],[2]])
2.  b = ([[10,20,30]])
3.  a+b
[Out]
array([[11, 21, 31],
       [12, 22, 32]])
```

4．ndarray的常用方法

示例 1：通过“shape”属性查看数组结构。

```
[In]
1.  b = np.array([[1, 2],  [3, 4]])
2.  b.shape
[Out]
(2, 2)
```

示例 2：通过“dtype”查看数组类型。

```
[In]
b.dtype
[Out]
dtype('int32')
```

示例 3：通过“size”属性查看元素个数。

```
[In]
b.size
[Out]
4
```

示例 4：通过“itemsize”属性查看每个元素的大小。

```
[In]
b.itemsize
[Out]
4
```

示例 5：对数组中所有元素求和。

```
[In]
b.sum()
[Out]
10
```

示例 5：对数组中所有元素求和。

```
[In]
b.sum()
[Out]
10
```

示例 6：对数组中所有元素求平均值。

```
[In]
b.mean()
[Out]
2.5
```

示例 7：对数组中所有元素求标准差。

```
[In]
b.std()
[Out]
1.118033988749895
```

示例 8：计算累计和，即自身和之前所有的数字之和

```
[In]
b.cumsum()
[Out]
array([ 1,  3,  6, 10], dtype=int32)
```

示例 9：对数组按列求和。

```
[In]
b.sum(axis=0)
[Out]
array([4, 6])
```

示例 10：对数组按行求和。

```
[In]
b.sum(axis=1)
[Out]
array([3, 7])
```

5．结构化数组

定义了4列名称和类型并填充：

- name列，字符串类型，字符长度等于或少于长度10；
- code列，字符串类型，字符长度等于或少于长度10；
- date列，Datetime类型，精确到天（day）；
- point列，32位int型整数。

示例 1：我们仿照 5.1.1 节关于元组的示例，定义了 4 列名称和类型并填充：

```
[In]
1.  import numpy as np
2.  a = np.array([('上证指数', '000001', '1990-12-19', 100), ('B 股指数', '000003',
'1992-02-21', 100)], dtype=[('name', 'U10'), ('code', 'U10'), ('date', 'datetime64[D]'),
('point', 'i4')])
3.  a

[Out]
array([('上证指数', '000001', '1990-12-19', 100),
('B股指数', '000003', '1992-02-21', 100)],
dtype=[('name', '<U10'), ('code', '<U10'), ('date', '<M8[D]'), ('point', '<i4')])
```

示例 2：在对结构性数组进行操作时，可以参考字典对象的查找方式，采用键码（列名）来查找。

```
[In]
a['name']

[Out]
array(['上证指数', 'B股指数'], dtype='<U10')
```

示例 3：查找 code 为 000001 的对象的名称。

```
[In]
a[a['code']=='000001']['name']

[Out]
array(['上证指数'], dtype='<U10')
```

结构化数组可以看作 ndarray 常规数组的延伸，其中每列的数据类型必须相同；结构化数组为 Python 提供了类似数据库的结构，较复杂但也继承了常规数组的优势。

6．矩阵操作

向量化是机器学习的一个必要过程，通过向量化可以一次对一个复杂对象进行操作，而不用再对该对象的每个元素进行操作，这样能够获得更紧凑的代码以换取更快的执行速度。而NumPy提供了方便的向量（矩阵）化操作

示例 1：按元素位置对应相加，要注意的是，矩阵与矩阵之间的加法要求矩阵的结构相同。

```
[In]
1.  import numpy as np
2.  a = np.random.standard_normal((4,3))  # 随机生成一个符合标准正态分布的 4×3 的矩阵
3.  b = np.random.standard_normal((4,3))
4.  a+b
```

```
[Out]
array([[ 0.29793943,  1.64732   ,  0.82442504],
       [-1.1646719 , -0.71022657,  1.12176551],
       [-0.56051346, -1.16281998,  0.85099824],
       [ 1.38338047,  0.07489906, -0.12774861]])
```

示例 2：对内部所有元素乘以 2 再加上 1。

```
[In]
a*2 +1
```

```
[Out]
array([[-0.28657108,  2.09724565,  1.03899139],
       [-0.61650928, -0.77523953,  1.32721907],
       [ 1.43702134, -0.29336916,  1.78759173],
       [ 1.0735027 ,  1.89784182,  1.28276709]])
```

示例 3：对矩阵进行转置。

```
[In]
b.transpose()
```

```
[Out]
array([[ 0.94122497, -0.35641726, -0.77902413,  1.34662912],
       [ 1.09869717,  0.17739319, -0.5161354 , -0.37402185],
       [ 0.80492935,  0.95815597,  0.45720237, -0.26913216]])
```

Pandas简介

Pandas是基于NumPy的一种工具包，可以把它们认为是增强版的NumPy结构化数组。Pandas纳入了大量的函数库和一些标准的数据模型，为数据分析领域提供了高效而简便的函数和方法。它是Python在数据分析方面成为主流工具的最重要因素之一。Pandas最初由AQR Capital Management于2008年4月作为金融数据分析工具开发出来，并于2009年年底开源，目前由专注于Python数据包开发的PyData开发团队继续开发和维护，属于PyData项目的一部分。

Series数据结构

Series是具有标签的一维数据，与NumPy中的一维array类似，但它的行列不再只是简单的整数索引，还可以使用显示索引，自行定义标签。它与基本的数据结构List也很相似，区别主要在于List中的元素可以是不同的数据类型，而array和Series中则只允许存储相同类型的数据

1．Series 的创建

Series 的创建有些类似字典和 NumPy 的结构化数组，通过 Series()函数来创建，并可以自定义索引，index 若是默认的，那么索引将会从 0 开始编号。若其存放的元素类型不同，则所有元素会都转换为 object 类型。具体操作参考以下示例。

示例 1：用 Series 存放相同类型的对象。

```
[In]
1.  import pandas as pd
2.  ser = pd.Series([1,2,3,4],index =['a','b','c','d'])
3.  ser
```

```
[Out]
a    1
b    2
c    3
d    4
dtype: int64
```

示例 2：用 Series 存放不同类型的对象。

```
[In]
1.  ser2 = pd.Series(['上证指数', '000001', '1990-12-19', 100], index =['name',
'code','date','point'])
2.  ser2
```

```
[Out]
name      上证指数
code      000001
date    1990-12-19
point          100
dtype: object
```

Series数据结构

2．Series 的访问

示例 1：按索引读取一个元素，此时 loc()可以省略。

```
[In]
ser.loc['b']
[Out]
2
```

示例 2：按位置读取一个元素，此时用到 iloc()，不要混淆。

```
[In]
ser.iloc[1]
[Out]
2
```

示例 3：读取多个元素，注意此时使用了两个中括号“[]”。

```
[In]
ser[['a','c']]
[Out]
a    1
c    3
dtype: int64
```

示例 4：切片操作，注意这里的切片同时包含了首尾。

```
[In]
ser.loc['b':'d']
[Out]
b    2
c    3
d    4
dtype: int64
```

3．Series 的增删改操作

示例 1：修改 Series 中某个元素的内容。

```
[In]
1.  ser['b'] = 10
2.  ser
[Out]
a     1
b    10
c     3
d     4
dtype: int64
```

示例 2：添加元素，通过拼接实现效果。

```
[In]
1.  ser3 = pd.Series([11],index =['f'])
2.  ser = ser.append(ser3)
3.  ser
[Out]
a     1
b    10
c     3
d     4
f    11
dtype: int64
```

示例 3：删除 Series 中某个元素。

```
[In]
ser.drop('b')
```

DataFrame数据结构

DataFrame是具有标签的二维数据，可以看作Series的组合，每一列都可以看作一个Series。在Series的基础上，DataFrame的每列数据类型可以不同，既有行索引也有列索引，可以方便地对表类型的数据（例如csv、txt）进行读写操作。

1. DataFrame 的创建

示例 1：直接构建一个 DataFrame。

[In]

```
1.  import pandas as pd
2.  df = pd.DataFrame([['上证指数','000001','1990-12-19',100],['B 股指数', '000003',
'1992-02-21',100]],
3.  index = ['row1','row2'],columns = ['name','code','date','point'])
4.  df
```

[Out]

	name	code	date	point
row1	上证指数	000001	1990-12-19	100
row2	B股指数	000003	1992-02-21	100

示例 2：通过构建字典，此时未定义行的名称，index 则默认为从 0 开始的编号。

[In]

```
1.  d = {'name':['上证指数','B 股指数'],
2.       'code':['000001','000003'],
3.       'date':['1990-12-19','1992-02-21'],
4.       'point':[100,100]}
5.  pd.DataFrame(d)
```

[Out]

	name	code	date	point
0	上证指数	000001	1990-12-19	100
1	B股指数	000003	1992-02-21	100

DataFrame数据结构

2．DataFrame 的访问

示例 1：按索引读取一个元素，此时的 loc()不可以省略。

```
[In]  df.loc['row1','name']
[Out] '上证指数'
```

示例 2：按行读取，此时用到 loc()，同样不可省略，会返回一个 Series。

```
[In]  type(df.loc['row2'])
[Out] pandas.core.series.Series
```

示例 3：按列读取，注意此时不能使用 loc()，同样返回 Series。

```
[In]  df['name']
[Out] row1    上证指数
      row2    B股指数
      Name: name, dtype: object
```

示例 4：也可以读取多列，此时注意需要将读取的列用“[]”包住。

```
[In]  df[['name','code']]
```

[Out]

	name	code
row1	上证指数	000001
row2	B股指数	000003

示例 5：先按列读取，再按行读取可以很容易实现切片操作。

```
[In]  df[['name','code']].loc['row2':]
```

[Out]

	name	code
row2	B股指数	000003

3．DataFrame 的修改

示例 1：先对 DataFrame 定位，再修改对应位置的值。

```
[In]  1.  df.loc['row1','name']='A股指数'
      2.  df
```

[Out]

	name	code	date	point
row1	A股指数	000001	1990-12-19	100
row2	B股指数	000003	1992-02-21	100

示例 2：修改 DataFrame 整行的值，该操作要求新修改的值与原列数对应。

```
[In]  1.  df.loc['row1']=['A股指数','000002','1990-12-19',100]
      2.  df
```

[Out]

	name	code	date	point
row1	A股指数	000002	1990-12-19	100
row2	B股指数	000003	1992-02-21	100

示例 3：修改 DataFrame 一整列的值。

```
[In]  1.  df['point']=101
      2.  df
```

[Out]

	name	code	date	point
row1	A股指数	000002	1990-12-19	101
row2	B股指数	000003	1992-02-21	101

示例 4：修改 DataFrame 一整列的全部元素，该操作要求修改的内容与原行数对应。

```
[In]  1.  df['point']=[0,1]
      2.  df
```

谢谢！