Financial Data Analysis with Python

Instructor: Luping Yu

Mar 1, 2022

Lecture 02. Data Structure

We'll start with Python's workhorse data structures: lists, dicts, and sets. Then, we'll look at the mechanics of Python file objects and interacting with your local hard drive.

Python的数据结构

- Numeric types (数值类型): The primary Python types for numbers are int and float.

In [ ]:

```
a = 2          # int, 整型
b = 4.8        # float, 浮点型
```

In [ ]:

```
# 数值计算
a + b    # 加
a - b    # 减
a * b    # 乘
b / a    # 除
b ** a   # 幂
b % a    # 余
```

- String (字符串): Many people use Python for its powerful and flexible built-in string processing capabilities.

In [ ]:

```
var = 'Hello, XMU School of Management'  # 单引号双引号都可
```

In [ ]:

```
# 切片：按索引取部分内容，索引从0开始，从左至右
var[0]        # 'H'
var[-1]       # 't'
var[0:5]      # 'Hello'
var[-5:]      # 'ement'
var[0:10:2]   # 'Hlo M'
```

In [ ]:

```
# 常用的字符串操作
len(var)                                  # 字符串长度
var.replace('Management','Economics')     # 字符串替换
var.split()                               # 字符串分隔（默认空格）
var.split(',')                            # 字符串分隔（指定字符）
' '.join([var, 'Finance'])                # 字符串连接（空格字符）
'-'.join([var, 'Finance', 'Accounting'])  # 字符串连接（指定字符）
var.upper()                               # 全转大写
var.lower()                               # 全转小写
'1'.zfill(6)                              # 指定长度。如长度不够，前面补0（常用情
```

- Boolean 布尔型: The two boolean values in Python are written as **True** and **False**.

In [ ]:

```python
a = 0
b = 1
c = 2

# 布尔运算
a == b
a > b
c > b
a != b
not a == b
(a > b) and (c > b)
(a > b) or (c > b)
(a < b) and (c > b)
```

- List 列表: Lists are variable-length and their contents can be modified in-place. You can define them using square brackets [ ].

In [ ]:

```python
x = []
x = [1, 2, 3, 4, 5]
x = ['a', 'b', 'c']
x = [1, 'a', True, [2, 3, 4], None]
```

In [ ]:

```python
# 列表和字符串一样支持切片访问，可以将字符串的一个字符当成列表中的一个元素
a = [1, 5, 4, 2, 3]
len(a)
max(a)
min(a)
sum(a)
a.index(3)        # 指定元素的位置
a.count(3)        # 统计元素个数
sorted(a)         # 元素排序
a.append(6)       # 增加一个元素
a.extend([7, 8])  # 与其他列表合并
a.insert(1, 'a')  # 在指定索引位置插入元素，索引从0开始
a.pop()           # 删除指定索引（未指定索引时，删除最后一个元素）
a.remove('a')     # 删除指定元素
```

In [ ]:

```python
# 迭代元素
a = [1, 5, 4, 2, 3]
for i in a:
    print(i)
```

In [ ]:

```python
# 列表生成器
[i for i in range(5)]

# 自定义结果
['第' + str(i) for i in range(5)]

# 条件筛选
[i for i in range(5) if i > 2]

# 拆开字符，过滤空格，全变成大写
[i.upper() for i in 'Hello XMU' if i != ' ']
```

- Dict 字典: A more common name for it is hash map or associative array. It is a flexibly sized collection of key-value pairs, where key and value are Python objects. One approach for creating one is to use curly braces {} and colons to separate keys and values.

In [ ]:
```
d = {}
d = dict()
d = {'a': 1, 'b': 2, 'c': 3}

# 以下方法均可定义字典
{'name': 'Tom', 'age': 18, 'height': 180}
d = dict(name='Tom', age=18, height=180)
d = dict([('name', 'Tom'), ('age', 18), ('height', 180)])
```

In [ ]:
```
# 访问字典的方法
d['name']               # 查询值
d['age'] = 20           # 改变值
d['gender'] = 'female'  # 增加属性
```

In [ ]:
```
# 常用的字典操作方法
d.pop('name')
d.clear()

d.keys()
d.values()
d.items()
d.copy()

max(d)
min(d)
len(d)
str(d)
sorted(d)
```

- set 集合: A set is an unordered collection of **unique** elements. You can think of them like dicts, but keys only, no values.

In [ ]:
```
s = {}
s = set()
s = {1, 2, 3, 4, 5}
s = {[1, 2, 3, 4, 5]}  # 使用列表定义
s = {1, 2, 2, 2}       # 去重
```

In [ ]:
```
# 集合没有顺序，没有索引，无法指定位置访问
s = {'a', 'b', 'c'}

'a' in s
s.add(2)
s.update([1, 3, 4])
s.remove('a')
s.discard('d')
s.clear()
```

Pandas的数据结构

Throughout the rest of the book, I use the following import convention for pandas:

```
In [5]: import pandas as pd
```

To get started with pandas, you will need to get comfortable with its two workhorse data structures: **Series** and **DataFrame**.

Series

A Series is a one-dimensional array-like object containing a sequence of values and an associated array of data labels, called its **index**. The simplest Series is formed from only an array of data.

```
In [6]: obj = pd.Series([4, 7, -5, 3])
        obj

Out[6]: 0    4
        1    7
        2   -5
        3    3
        dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

```
In [7]: obj.values

Out[7]: array([ 4,  7, -5,  3])
```

```
In [8]: obj.index

Out[8]: RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [9]: obj2 = pd.Series([4, 7, 5, 3], index = ['d', 'b', 'a', 'c'])
        obj2

Out[9]: d    4
        b    7
        a    5
        c    3
        dtype: int64
```

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [10]: obj2['a']

Out[10]: 5
```

In [11]:
```
obj2[['c', 'a', 'd']]
```

Out[11]:
```
c    3
a    5
d    4
dtype: int64
```

Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers.

We can also using functions or operations:

In [12]:
```
obj2[obj2 > 0]
```

Out[12]:
```
d    4
b    7
a    5
c    3
dtype: int64
```

In [13]:
```
obj2 * 2
```

Out[13]:
```
d     8
b    14
a    10
c     6
dtype: int64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

In [14]:
```
'b' in obj2
```

Out[14]:
```
True
```

In [15]:
```
'e' in obj2
```

Out[15]:
```
False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

In [16]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
obj3
```

Out[16]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

When you are only passing a dict, the index in the resulting Series will have the dict’s keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

In [17]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

Out[17]:

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as **NaN** (not a number), which is considered in pandas to mark missing or NA values. Since 'Utah' was not included in states, it is excluded from the resulting object.

I will use the terms “missing” or “NA” interchangeably to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

In [18]:

```
pd.isnull(obj4)
```

Out[18]:

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [19]:

```
pd.notnull(obj4)
```

Out[19]:

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

In [20]:

```
obj3 + obj4
```

Out[20]:

```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Missing data and data alignment features will be addressed in more detail later.

Both the Series object itself and its index have a **name** attribute, which integrates with other key areas of pandas functionality:

In [21]:

```
obj4.name = 'population'
obj4.index.name = 'state'
obj4
```

Out[21]:

```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

A Series’s index can be altered in-place by assignment:

In [22]:

```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
obj
```

Out[22]:

```
Bob       4
Steve     7
Jeff     -5
Ryan      3
dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more **two-dimensional** blocks rather than a list, dict, or some other collection of one-dimensional arrays.

In [23]:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

In [24]:

```
frame
```

Out[24]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

For large DataFrames, the head method selects only the first five rows:

In [25]:

```
frame.head()
```

Out[25]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

In [26]:

```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[26]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

In [27]:

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                      index=['one', 'two', 'three', 'four','five', 'six'])
frame2
```

Out[27]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

In [28]:

```
frame2['state']
```

Out[28]:

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

In [29]:

```
frame2.year
```

Out[29]:

```
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

Note that the returned Series have the same index as the DataFrame, and their name attribute has been appropriately set.

Rows can also be retrieved by position or name with the special loc attribute:

In [30]:
```
frame2.loc['three']
```

Out[30]:
```
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three, dtype: object
```

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

In [31]:
```
frame2['debt'] = 16.5
frame2
```

Out[31]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

In [32]:
```
frame2['debt'] = [0, 1, 2, 3, 4, 5]
frame2
```

Out[32]:

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4
six	2003	Nevada	3.2	5

When you are assigning lists or arrays to a column, **the value's length must match the length of the DataFrame**. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

In [33]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame2['debt'] = val
frame2
```

Out[33]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

Assigning a column that doesn’t exist will create a new column. The del keyword will delete columns as with a dict.

As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

In [34]:

```
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

Out[34]:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

In [35]:

```
del frame2['eastern']
frame2.columns
```

Out[35]:

```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Another common form of data is a nested dict of dicts:

In [36]:

```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [37]:

```
frame3 = pd.DataFrame(pop)
frame3
```

Out[37]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

You can transpose the DataFrame (swap rows and columns):

In [38]:

```
frame.T
```

Out[38]:

	0	1	2	3	4	5
state	Ohio	Ohio	Ohio	Nevada	Nevada	Nevada
year	2000	2001	2002	2001	2002	2003
pop	1.5	1.7	3.6	2.4	2.9	3.2

Series与DataFrame的基本功能

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame.

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. The drop method will return a **new object** with the indicated value or values deleted from an axis:

In [39]:

```
obj = pd.Series([0, 1, 2, 3, 4], index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[39]:

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

In [40]:

```
new_obj = obj.drop('c')
new_obj
```

Out[40]:

```
a    0
b    1
d    3
e    4
dtype: int64
```

In [41]:

```
obj.drop(['d', 'c'])
```

Out[41]:

```
a    0
b    1
e    4
dtype: int64
```

In [42]:

```
obj
```

Out[42]:

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object in-place without returning a new object:

In [43]:

```
obj.drop('d', inplace=True)
```

```
obj
```

Out[43]:

```
a    0
b    1
c    2
e    4
dtype: int64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

In [44]:

```
data = pd.DataFrame([[0, 1, 2, 3],[4, 5, 6, 7],[8, 9, 10, 11],[12, 13, 14, 1
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[44]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

In [45]:

```
data.drop(['Colorado', 'Ohio'])
```

Out[45]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

You can drop values from the columns by passing axis=1 or axis='columns':

In [46]:

```
data.drop('two', axis=1)
```

Out[46]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [47]:

```
data.drop(['two', 'four'], axis='columns')
```

Out[47]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Selection and Filtering

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

In [48]:

```
data = pd.DataFrame([[0, 1, 2, 3],[4, 5, 6, 7],[8, 9, 10, 11],[12, 13, 14, 1
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[48]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [49]:

```
data['two']
```

Out[49]:

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

In [50]:

```
data[['three', 'one']]
```

Out[50]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

In [51]:

```
data[:2]
```

Out[51]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [52]:

```
data[data['three'] > 5]
```

Out[52]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

The row selection syntax data [:2] is provided as a convenience. Passing a single element or a list to the [ ] operator selects columns.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

In [53]:

```
data < 5
```

Out[53]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [54]:

```
data[data < 5] = 0
data
```

Out[54]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

For DataFrame label-indexing on the rows, I introduce the special indexing operators **loc** and **iloc**. They enable you to select a subset of the rows and columns from a DataFrame using either axis labels (loc) or integers (iloc).

As a preliminary example, let's select a single row and multiple columns by label:

In [55]:

```
data.loc['Colorado', ['two', 'three']]
```

Out[55]:

```
two      5
three    6
Name: Colorado, dtype: int64
```

We'll then perform some similar selections with integers using iloc:

In [56]:

```
data.iloc[2, [3, 0, 1]]
```

Out[56]:

```
four    11
one      8
two      9
Name: Utah, dtype: int64
```

Both indexing functions work with slices in addition to single labels or lists of labels:

In [57]:

```
data.loc[:'Utah', 'two']
```

Out[57]:

```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64
```

In [58]:

```
data.iloc[:, :3][data['three'] > 5]
```

Out[58]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

In [59]:

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

Out[59]:

```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [60]:

```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
s2
```

Out[60]:

```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [61]:

```
s1 + s2
```

Out[61]:

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don’t overlap. Missing values will then propagate in further arithmetic computations.

In [62]:

```
df1 = pd.DataFrame([[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                   columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df1
```

Out[62]:

	b	c	d
Ohio	0	1	2
Texas	3	4	5
Colorado	6	7	8

In [63]:

```
df2 = pd.DataFrame([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]],
                       columns=list('bde'),
                       index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df2
```

Out[63]:

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

In [64]:

```
df1 + df2
```

Out[64]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Since the 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result.

Sorting and Ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

In [65]:

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj
```

Out[65]:

```
d    0
a    1
b    2
c    3
dtype: int64
```

In [66]:

```
obj.sort_index()
```

Out[66]:

```
a    1
b    2
c    3
d    0
dtype: int64
```

With a DataFrame, you can sort by index on either axis:

In [67]:

```
frame = pd.DataFrame([[4, 5, 6, 7], [0, 1, 2, 3]],
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
frame
```

Out[67]:

	d	a	b	c
three	4	5	6	7
one	0	1	2	3

In [68]:

```
frame.sort_index()
```

Out[68]:

	d	a	b	c
one	0	1	2	3
three	4	5	6	7

In [69]:

```
frame.sort_index(axis=1)
```

Out[69]:

	a	b	c	d
three	5	6	7	4
one	1	2	3	0

The data is sorted in ascending order by default, but can be sorted in descending order, too:

In [70]:

```
frame.sort_index(axis=1, ascending=False)
```

Out[70]:

	d	c	b	a
three	4	7	6	5
one	0	3	2	1

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the by option of sort_values:

In [71]:

```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[71]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [72]:

```
frame.sort_values(by='b')
```

Out[72]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

In [73]:

```
frame.sort_values(by=['a', 'b'])
```

Out[73]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Axis Indexes with Duplicate Labels

Up until now all of the examples we've looked at have had **unique** axis labels (index values). While many pandas functions (like reindex) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

In [74]:

```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
obj
```

Out[74]:

```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

The index's is_unique property can tell you whether its labels are unique or not:

In [75]:

```
obj.index.is_unique
```

Out[75]:

```
False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

In [76]:

```
obj['a']
```

Out[76]:

```
a    0
a    1
dtype: int64
```

In [77]:

```
 obj['c']
```

Out[77]:

```
4
```