Lecture 02. Data Structure

Instructor: Luping Yu

Mar 5, 2024

We'll start with Python data structures such as **lists**, **dicts**, and **sets**. Then, we'll delve into the mechanics of Pandas objects, including **series** and **dataframe**.

Python Language Basics

- `Numeric types` : The primary Python types for numbers are **int** and **float**.

In [ ]:
```
a = 2    # int
b = 4.8  # float
```

- `String` : Many people use Python for its powerful and flexible built-in string processing capabilities.

In [ ]:
```
var = 'Hello, XMU School of Management'  # Either single quotes ' or double quotes "
```

In [ ]:
```
# Common string operations
var[:5]
len(var)
var.replace('Management','Economics')
var.split()
var.split(',')
' '.join([var, 'Finance'])
var.upper()
var.lower()
'1'.zfill(6)
```

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

In [ ]:
```
# Boolean operations
a == b
a > b
a < b
not a == b # a != b
(a > b) and (c > b) # (a > b) & (c > b)
(a > b) or (c > b) # (a > b) | (c > b)
```

- `List` : Lists are variable-length and their contents can be modified in-place. You can define them using square brackets `[ ]`
  - `List` supports **slicing** just like `String` , a single character of a string can be treated as an element of a list.

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

In [ ]:
```
# Common list operations
a = [1, 5, 4, 2, 3]
len(a)
max(a)
min(a)
sum(a)
```

```
a.count(3)
sorted(a)
a.append(6)
a.extend([7, 8])
a.insert(1, 'a')
a.pop()
a.remove('a')
```

In [ ]:
```
# Iterate over a list
a = [1, 5, 4, 2, 3]
for i in a:
    print(i * 2)
```

In [ ]:
```
# List comprehensions
[i for i in range(5)]

# Customize output
['第' + str(i) for i in range(5)]

# Filter
[i for i in range(5) if i > 2]

# Split the string, filter out spaces, and convert all characters to uppercase
[i.upper() for i in 'Hello XMU' if i != ' ']
```

- `set` : A set is an unordered collection of **unique** elements. Sets have no order and no way to access elements by position

In [ ]:
```
# The following methods can be used to define a set
s = {1, 2, 3, 4, 5}
s = set([1, 2, 3, 4, 5])

# Unique elements
s = {1, 2, 2, 2}
```

- `Dict` : A more common name for it is **associative array**. It is a flexibly sized collection of **key-value** pairs. You can define them using curly braces `{ }`

In [ ]:
```
# The following methods can be used to define a dictionary
d = {'name': 'Tom', 'age': 18, 'height': 180}
d = dict(name='Tom', age=18, height=180)
d = dict([('name', 'Tom'), ('age', 18), ('height', 180)])
```

In [ ]:
```
# Ways to access a Python dictionary
d['name']
d['age'] = 20
d['gender'] = 'female'

# Common dict operations
d.keys()
d.values()
d.items()
```

Pandas Basics

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

In [1]:
```
import pandas as pd
```

To get started with pandas, you will need to get comfortable with two 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 [2]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[2]:
```
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 [3]:
```
obj.values
```

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

In [4]:
```
obj.index
```

Out[4]:
```
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 [5]:
```
obj2 = pd.Series([81, 77, 85, 59], index = ['amy', 'bob', 'chris', 'david'])
obj2
```

Out[5]:
```
amy      81
bob      77
chris    85
david    59
dtype: int64
```

You can use **labels** in the index when selecting single values or a set of values:

In [6]:
```
obj2['amy']
```

Out[6]:
```
81
```

In [7]:
```
obj2[['chris', 'amy', 'david']]
```

Out[7]:
```
chris    85
amy      81
david    59
dtype: int64
```

Here `['chris', 'amy', 'david']` is interpreted as a list of indices, even though it contains strings instead of integers.

We can also using functions or operations:

In [8]:
```
obj2[obj2 > 60]
```

Out[8]:
```
amy      81
bob      77
chris    85
dtype: int64
```

In [9]:
```
obj2 * 2
```

Out[9]:
```
amy      162
bob      154
chris    170
david    118
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.

In [10]:
```
'bob' in obj2
```

Out[10]:
```
True
```

In [11]:
```
'emma' in obj2
```

Out[11]:
```
False
```

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

In [12]:
```
sdata = {'Fujian': 53110, 'Sichuan': 56750, 'Shanghai': 44653, 'Guangdong': 129119}
obj3 = pd.Series(sdata)
```

```python
obj3
```

Out[12]:

```
Fujian        53110
Sichuan       56750
Shanghai      44653
Guangdong    129119
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 [13]:

```python
obj4 = pd.Series(sdata, index=['Guangdong', 'Sichuan', 'Fujian','Beijing'])
obj4
```

Out[13]:

```
Guangdong    129119.0
Sichuan       56750.0
Fujian        53110.0
Beijing           NaN
dtype: float64
```

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

The `isnull` and `notnull` functions in pandas should be used to detect missing data:

In [14]:

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

Out[14]:

```
Guangdong    False
Sichuan      False
Fujian       False
Beijing       True
dtype: bool
```

In [15]:

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

Out[15]:
```
Guangdong     True
Sichuan       True
Fujian        True
Beijing      False
dtype: bool
```

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

In [16]:
```
obj3 + obj4
```

Out[16]:
```
Beijing           NaN
Fujian       106220.0
Guangdong    258238.0
Shanghai          NaN
Sichuan      113500.0
dtype: float64
```

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

In [17]:
```
obj4.name = 'gdp'
obj4.index.name = 'province'
obj4
```

Out[17]:
```
province
Guangdong    129119.0
Sichuan       56750.0
Fujian        53110.0
Beijing           NaN
Name: gdp, dtype: float64
```

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

In [18]:
```
obj4.index = ['A', 'B', 'C', 'D']
obj4
```

Out[18]:
```
A    129119.0
B     56750.0
C     53110.0
D         NaN
Name: gdp, dtype: float64
```

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 [19]:
```
data = {'firm': ['Tencent', 'Tencent', 'Tencent', 'Xiaomi', 'Xiaomi', 'Xiaomi'],
        'year': [2019, 2020, 2021, 2020, 2021, 2022],
        'revenue': [54.5, 70.4, 86.6, 36.0, 50.8, 45.4]}
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 [20]:
```
frame
```

Out[20]:

	firm	year	revenue
0	Tencent	2019	54.5
1	Tencent	2020	70.4
2	Tencent	2021	86.6
3	Xiaomi	2020	36.0
4	Xiaomi	2021	50.8
5	Xiaomi	2022	45.4

For large DataFrames, the `.head()` method selects only the first five rows:

In [21]:
```
frame.head()
```

Out[21]:

	firm	year	revenue
0	Tencent	2019	54.5
1	Tencent	2020	70.4
2	Tencent	2021	86.6
3	Xiaomi	2020	36.0
4	Xiaomi	2021	50.8

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

In [22]:
```
pd.DataFrame(data, columns=['year', 'revenue', 'firm'])
```

Out[22]:

	year	revenue	firm
0	2019	54.5	Tencent
1	2020	70.4	Tencent
2	2021	86.6	Tencent
3	2020	36.0	Xiaomi
4	2021	50.8	Xiaomi
5	2022	45.4	Xiaomi

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

In [23]:
```
frame2 = pd.DataFrame(data, columns=['year', 'firm', 'revenue', 'roa'],
                      index=['one', 'two', 'three', 'four','five', 'six'])
frame2
```

Out[23]:

	year	firm	revenue	roa
one	2019	Tencent	54.5	NaN
two	2020	Tencent	70.4	NaN
three	2021	Tencent	86.6	NaN
four	2020	Xiaomi	36.0	NaN
five	2021	Xiaomi	50.8	NaN
six	2022	Xiaomi	45.4	NaN

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

In [24]:
```
frame2['firm']
```

Out[24]:
```
one      Tencent
two      Tencent
three    Tencent
four      Xiaomi
five      Xiaomi
six       Xiaomi
Name: firm, dtype: object
```

In [25]:
```
frame2.year
```

Out[25]:
```
one      2019
two      2020
three    2021
four     2020
five     2021
six      2022
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 [26]:
```
frame2.loc['three']
```

Out[26]:
```
year          2021
firm       Tencent
revenue       86.6
roa            NaN
Name: three, dtype: object
```

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

In [27]:
```
frame2['roa'] = 10
frame2
```

Out[27]:

	year	firm	revenue	roa
one	2019	Tencent	54.5	10
two	2020	Tencent	70.4	10
three	2021	Tencent	86.6	10
four	2020	Xiaomi	36.0	10
five	2021	Xiaomi	50.8	10
six	2022	Xiaomi	45.4	10

In [28]:

```
frame2['roa'] = [11.4, 14.0, 15.5, 7.1, 9.3, 4.3]
frame2
```

Out[28]:

	year	firm	revenue	roa
one	2019	Tencent	54.5	11.4
two	2020	Tencent	70.4	14.0
three	2021	Tencent	86.6	15.5
four	2020	Xiaomi	36.0	7.1
five	2021	Xiaomi	50.8	9.3
six	2022	Xiaomi	45.4	4.3

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 [29]:

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

Out[29]:

	year	firm	revenue	roa
one	2019	Tencent	54.5	NaN
two	2020	Tencent	70.4	-1.2
three	2021	Tencent	86.6	NaN
four	2020	Xiaomi	36.0	-1.5
five	2021	Xiaomi	50.8	-1.7
six	2022	Xiaomi	45.4	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 'Tencent':

In [30]:

```
frame2['video_game_company'] = (frame2['firm'] == 'Tencent')
frame2
```

Out[30]:

	year	firm	revenue	roa	video_game_company
one	2019	Tencent	54.5	NaN	True
two	2020	Tencent	70.4	-1.2	True
three	2021	Tencent	86.6	NaN	True
four	2020	Xiaomi	36.0	-1.5	False
five	2021	Xiaomi	50.8	-1.7	False
six	2022	Xiaomi	45.4	NaN	False

In [31]:

```
del frame2['video_game_company']
frame2
```

Out[31]:

	year	firm	revenue	roa
one	2019	Tencent	54.5	NaN
two	2020	Tencent	70.4	-1.2
three	2021	Tencent	86.6	NaN
four	2020	Xiaomi	36.0	-1.5
five	2021	Xiaomi	50.8	-1.7
six	2022	Xiaomi	45.4	NaN

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

In [32]:
```
revenue = {'Tencent': {2020: 70.4, 2021: 86.6},
           'Xiaomi': {2020: 36.0, 2021: 50.8, 2022: 45.4}}
```

In [33]:
```
frame3 = pd.DataFrame(revenue)
frame3
```

Out[33]:

	Tencent	Xiaomi
2020	70.4	36.0
2021	86.6	50.8
2022	NaN	45.4

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

In [34]:
```
frame3.T
```

Out[34]:

	2020	2021	2022
Tencent	70.4	86.6	NaN
Xiaomi	36.0	50.8	45.4

Essential functionalities of Series and DataFrame

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 [35]:
```
obj = pd.Series([0, 1, 2, 3, 4], index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[35]:
```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

In [36]:
```
new_obj = obj.drop('c')
new_obj
```

Out[36]:
```
a    0
b    1
d    3
e    4
dtype: int64
```

In [37]:
```
obj.drop(['d', 'c'])
```

Out[37]:
```
a    0
b    1
e    4
dtype: int64
```

In [38]:
```
obj
```

Out[38]:
```
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 [39]:
```
obj.drop('d', inplace=True)
obj
```

Out[39]:
```
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 [40]:
```
data = pd.DataFrame([[0, 1, 2, 3],[4, 5, 6, 7],[8, 9, 10, 11],[12, 13, 14, 15]],
                    index=['Tencent', 'Xiaomi', 'ByteDance', 'miHoYo'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[40]:

	one	two	three	four
Tencent	0	1	2	3
Xiaomi	4	5	6	7
ByteDance	8	9	10	11
miHoYo	12	13	14	15

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

In [41]:
```
data.drop(['Xiaomi', 'ByteDance'])
```

Out[41]:

	one	two	three	four
Tencent	0	1	2	3
miHoYo	12	13	14	15

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

In [42]:
```
data.drop('two', axis=1)
```

Out[42]:

	one	three	four
Tencent	0	2	3
Xiaomi	4	6	7
ByteDance	8	10	11
miHoYo	12	14	15

Selection and Filtering

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

In [43]:
```
data
```

Out[43]:

	one	two	three	four
Tencent	0	1	2	3
Xiaomi	4	5	6	7
ByteDance	8	9	10	11
miHoYo	12	13	14	15

In [44]:
```
data['two']
```

Out[44]:
```
Tencent       1
Xiaomi        5
ByteDance     9
miHoYo       13
Name: two, dtype: int64
```

In [45]:
```
data[['three', 'one']]
```

Out[45]:

	three	one
Tencent	2	0
Xiaomi	6	4
ByteDance	10	8
miHoYo	14	12

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

In [46]:
```
data[:2]
```

Out[46]:

	one	two	three	four
Tencent	0	1	2	3
Xiaomi	4	5	6	7

In [47]:
```python
data[data['one'] > 7]
```

Out[47]:

	one	two	three	four
ByteDance	8	9	10	11
miHoYo	12	13	14	15

Passing 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 [50]:
```python
data[data < 10] = 0
data
```

Out[50]:

	one	two	three	four
Tencent	0	0	0	0
Xiaomi	0	0	0	0
ByteDance	0	0	10	11
miHoYo	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 [51]:
```python
data = pd.DataFrame([[0, 1, 2, 3],[4, 5, 6, 7],[8, 9, 10, 11],[12, 13, 14, 15]],
                    index=['Tencent', 'Xiaomi', 'ByteDance', 'miHoYo'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[51]:

	one	two	three	four
Tencent	0	1	2	3
Xiaomi	4	5	6	7
ByteDance	8	9	10	11
miHoYo	12	13	14	15

In [52]:
```
data.loc['ByteDance', ['two', 'three']]
```

Out[52]:
```
two       9
three    10
Name: ByteDance, dtype: int64
```

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

In [53]:
```
data.iloc[2, [3, 0, 1]]
```

Out[53]:
```
four    11
one      8
two      9
Name: ByteDance, dtype: int64
```

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

In [54]:
```
data.loc['Xiaomi']
```

Out[54]:
```
one      4
two      5
three    6
four     7
Name: Xiaomi, dtype: int64
```

In [55]:
```
data.iloc[:, :3][data['three'] > 2]
```

Out[55]:

	one	two	three
Xiaomi	4	5	6
ByteDance	8	9	10
miHoYo	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 [56]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

Out[56]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [57]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
s2
```

Out[57]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [58]:
```
s1 + s2
```

Out[58]:
```
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.

In [59]:
```
df1 = pd.DataFrame([[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                   columns=list('bcd'),
                   index=['Tencent', 'Xiaomi', 'ByteDance'])
df1
```

Out[59]:

	b	c	d
Tencent	0	1	2
Xiaomi	3	4	5
ByteDance	6	7	8

In [60]:
```
df2 = pd.DataFrame([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]],
                   columns=list('bde'),
                   index=['miHoYo', 'ByteDance', 'Tencent', 'Alibaba'])
df2
```

Out[60]:

	b	d	e
miHoYo	0	1	2
ByteDance	3	4	5
Tencent	6	7	8
Alibaba	9	10	11

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

In [61]:
```
df1 + df2
```

Out[61]:

	b	c	d	e
Alibaba	NaN	NaN	NaN	NaN
ByteDance	9.0	NaN	12.0	NaN
Tencent	6.0	NaN	9.0	NaN
Xiaomi	NaN	NaN	NaN	NaN
miHoYo	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 by row or column index, use the `.sort_index()` method, which returns a new, sorted object:

In [62]:
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj
```

Out[62]:
```
d    0
a    1
b    2
c    3
dtype: int64
```

In [63]:
```
obj.sort_index()
```

Out[63]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

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

In [64]:
```
frame = pd.DataFrame([[8, 9, 10, 11], [0, 1, 2, 3], [4, 5, 6, 7]],
                     index=['three','one','two'],
                     columns=['d', 'a', 'b', 'c'])
frame
```

Out[64]:

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

In [65]:
```
frame.sort_index()
```

Out[65]:

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

In [66]:
```
frame.sort_index(axis=1)
```

Out[66]:

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

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

In [67]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[67]:

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

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 [68]:
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[68]:

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

In [69]:
```
frame.sort_values(by='b')
```

Out[69]:

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

In [70]:
```
frame.sort_values(by=['a', 'b'])
```

Out[70]:

	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 require that the labels be unique, it's not mandatory. Let's consider a small `Series` with duplicate indices:

In [71]:
```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
obj
```

Out[71]:
```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

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 [72]:
```
obj['a']
```

Out[72]:
```
a    0
a    1
dtype: int64
```

In [73]:
```
obj['c']
```

Out[73]: 4