Data Indexing and Selection

In **Part 2,** we looked in detail at **methods and tools to access, set, and modify values in NumPy arrays.**

These included:

- indexing (e.g., `arr[2, 1]`),
- slicing (e.g., `arr[:, 1:5]`),
- masking (e.g., `arr[arr > 0]`),
- fancy indexing (e.g., `arr[0, [1, 5]]`),
- and combinations thereof (e.g., `arr[:, [1, 5]]`).

Here we'll look at **similar means of accessing and modifying values** in Pandas `Series` **and** `DataFrame` **objects.**

If you have used the **NumPy patterns,** the corresponding **patterns in Pandas** will feel very **familiar,** though there are a few **quirks** to be aware of.

We'll **start with** the simple case of the **one-dimensional** `Series` **object,** and **then** move on to the **more complicated two-dimensional** `DataFrame` **object.**

Data Selection in Series

As you saw in the previous chapter, a `Series` **object acts**

- in many ways like a **one-dimensional NumPy array,**
- and in many ways like a **standard Python dictionary.**

If you **keep these two overlapping analogies in mind,** it will **help you understand the patterns of data indexing and selection** in these arrays.

Series as Dictionary

Like a dictionary, the `Series` object provides a **mapping from a collection of keys to a collection of values:**

In [1]:
```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[1]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [ ]:
```
data['b']
```

Out[ ]:
```
0.5
```

We can also **use dictionary-like Python expressions and methods** to examine the **keys/indices and values:**

In [ ]:
```
'a' in data
```

Out[ ]:
```
True
```

In [ ]:
```
data.keys()
```

Out[ ]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [2]:
```
data.items()
```

Out[2]:
```
<zip at 0x7e102f66c0c0>
```

In [ ]:
```
list(data.items())
```

Out[ ]:
```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can also be **modified** with a **dictionary-like syntax.**

Just as you can **extend a dictionary** by **assigning to a new key,**

you can **extend a `Series`** by **assigning to a new index value:**

In [ ]:
```
data['e'] = 1.25
data
```

Out[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

This **easy mutability** of the objects is a **convenient feature:**

under the hood, **Pandas is making decisions** about **memory layout** and **data copying** that might need to take place,

and the **user** generally **does not need to worry** about these issues.

Series as One-Dimensional Array

A `Series` **builds on this dictionary-like interface,**

and provides **array-style item selection** via the same basic mechanisms **as NumPy arrays** — that is, **slices, masking, and fancy indexing.**

Examples of these are as follows:

In [ ]:
```
# slicing by explicit index
data['a':'c']
```

Out[ ]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [ ]:
```
# slicing by implicit integer index
data[0:2]
```

Out[ ]:
```
a    0.25
b    0.50
dtype: float64
```

In [ ]:
```python
# masking
data[(data > 0.3) & (data < 0.8)]
```

Out[ ]:
```
b    0.50
c    0.75
dtype: float64
```

In [ ]:
```python
# fancy indexing
data[['a', 'e']]
```

Out[ ]:
```
a    0.25
e    1.25
dtype: float64
```

Of these, **slicing may be the source of the most confusion.**

Notice that when **slicing with an explicit index** (e.g., `data['a':'c']`), the **final index is included in the slice,**

while when **slicing with an implicit index** (e.g., `data[0:2]`), the **final index is excluded from the slice.**

Indexers: loc and iloc

If your `Series` has an **explicit integer index,** an **indexing operation** such as `data[1]` **will use the explicit indices,**

while a **slicing operation like `data[1:3]`** will use the **implicit Python-style indices:**

In [3]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[3]:
```
1    a
3    b
5    c
dtype: object
```

In [ ]:
```python
# explicit index when indexing
data[1]
```

Out[ ]:
```
'a'
```

In [ ]:
```python
# implicit index when slicing
data[1:3]
```

Out[ ]:
```
3    b
5    c
dtype: object
```

Because of this **potential confusion** in the case of **integer indexes,**

Pandas provides some **special indexer attributes** that **explicitly expose certain indexing schemes.**

These are **not functional methods,** but ***attributes*** that **expose a particular slicing interface** to the data in the `Series`.

First, the `loc` **attribute allows indexing and slicing** that **always references the explicit index:**

In [5]:
```
data
```

Out[5]:
```
1    a
3    b
5    c
dtype: object
```

In [ ]:
```
data.loc[1]
```

Out[ ]:
```
'a'
```

In [ ]:
```
data.loc[1:3]
```

Out[ ]:
```
1    a
3    b
dtype: object
```

The `iloc` **attribute** allows **indexing and slicing** that **always references the implicit** Python-style index:

In [6]:
```
data
```

Out[6]:
```
1    a
3    b
5    c
dtype: object
```

In [ ]:
```
data.iloc[1]
```

Out[ ]:
```
'b'
```

In [ ]:
```
data.iloc[1:3]
```

Out[ ]:
```
3    b
5    c
dtype: object
```

One guiding **principle** of Python code is that **"explicit is better than implicit."**

The **explicit nature of** `loc` **and** `iloc` makes them helpful in maintaining **clean and readable code;**

especially in the case of integer indexes, using them consistently can prevent **subtle bugs due to the mixed indexing/slicing convention.**

Data Selection in DataFrames

Recall that a *`DataFrame`* *acts* in many ways like a **two-dimensional or structured array,** and in other ways like a **dictionary of** `Series` structures sharing the same index.

These **analogies can be helpful** to keep in mind as we explore **data selection** within this structure.

DataFrame as Dictionary

The **first analogy** we will consider is the `DataFrame` **as a dictionary of related** `Series` **objects.**

Let's return to our **example of areas and populations of states:**

In [7]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'Florida': 170312, 'New York': 141297,
                  'Pennsylvania': 119280})
pop = pd.Series({'California': 39538223, 'Texas': 29145505,
                 'Florida': 21538187, 'New York': 20201249,
                 'Pennsylvania': 13002700})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[7]:

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187
New York	141297	20201249
Pennsylvania	119280	13002700

The **individual** `Series` that make up the **columns** of the `DataFrame` can be **accessed via dictionary-style indexing** of the **column name:**

In [ ]:

```
data['area']
```

```
Out[ ]: California      423967
        Texas           695662
        Florida         170312
        New York        141297
        Pennsylvania    119280
        Name: area, dtype: int64
```

Equivalently, we can use **attribute-style access** with **column names** that are **strings:**

```
In [ ]: data.area
```

```
Out[ ]: California      423967
        Texas           695662
        Florida         170312
        New York        141297
        Pennsylvania    119280
        Name: area, dtype: int64
```

Though this is a **useful shorthand,** keep in mind that it **does not work for all cases!**

For example, if the **column names are not strings,** or if the **column names conflict with methods of the `DataFrame`,** this attribute-style access is not possible.

For example, the `DataFrame` **has a `pop` method,** so `data.pop` will point to this rather than the `pop` **column:**

In [ ]:
```
data.pop is data["pop"]
```

Out[ ]:
```
False
```

In particular, you should **avoid** the temptation to try **column assignment via attribute**s (i.e., use `data['pop'] = z` rather than `data.pop = z`).

Like with the `Series` objects discussed earlier, this **dictionary-style syntax** can also be used to **modify** the object, in this case **adding a new column:**

In [8]:
```
data['density'] = data['pop'] / data['area']
data
```

Out[8]:

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

This **shows a preview** of the **straightforward** syntax of **element-by-element arithmetic** between `Series` **objects;**

we'll dig into this further in Operating on Data in Pandas.

DataFrame as Two-Dimensional Array

As mentioned previously, we can also **view the `DataFrame` as an enhanced two-dimensional array.**

We can examine the **raw underlying data array** using the `values` **attribute:**

In [ ]:
```
data.values
```

Out[ ]:
```
array([[4.23967000e+05, 3.95382230e+07, 9.32577842e+01],
       [6.95662000e+05, 2.91455050e+07, 4.18960717e+01],
       [1.70312000e+05, 2.15381870e+07, 1.26463121e+02],
       [1.41297000e+05, 2.02012490e+07, 1.42970120e+02],
       [1.19280000e+05, 1.30027000e+07, 1.09009893e+02]])
```

With this picture in mind, many familiar **array-like operations can be done on the `DataFrame` itself.**

For example, we can **transpose** the full `DataFrame` to swap rows and columns:

In [ ]:
```
data.T
```

Out[ ]:

	California	Texas	Florida	New York
area	4.239670e+05	6.956620e+05	1.703120e+05	1.412970e+05
pop	3.953822e+07	2.914550e+07	2.153819e+07	2.020125e+07
density	9.325778e+01	4.189607e+01	1.264631e+02	1.429701e+02

When it comes to **indexing of a** `DataFrame` **object,** however, it is clear that the **dictionary-style indexing** of columns **precludes** our **ability to simply treat it as a NumPy array.**

In particular, passing a single **index** to **an array** accesses a **row:**

In [ ]:
```
data.values[0]
```

Out[ ]:
```
array([4.23967000e+05, 3.95382230e+07, 9.32577842e+01])
```

and passing a single **"index"** to a `DataFrame` accesses a **column:**

In [ ]:
```
data['area']
```

Out[ ]:
```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
Name: area, dtype: int64
```

Thus, for **array-style indexing,** we need **another convention.**

Here Pandas again uses the `loc` **and** `iloc` **indexers** mentioned earlier.

Using the `iloc` indexer, we can **index the underlying array** as if it were a simple **NumPy array** (using the implicit Python-style index),

but the `DataFrame` **index and column labels** are **maintained in the result:**

In [9]:
```
data
```

Out[9]:

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [ ]:
```
data.iloc[:3, :2]
```

Out[ ]:

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187

Similarly, using the `loc` indexer we can **index the underlying data** in an **array-like style** but using the **explicit index and column names:**

In [10]:
```
data
```

Out[10]:

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [ ]:

```
data.loc[:'Florida', :'pop']
```

Out[ ]:

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187

Any of the familiar **NumPy-style data access patterns** can be used **within these indexers.**

For example, in the `loc` indexer we can combine masking and fancy indexing as follows:

In [11]:
```
data
```

Out[11]:

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [ ]:

```
data.loc[data.density > 120, ['pop', 'density']]
```

Out[ ]:

	pop	density
Florida	21538187	126.463121
New York	20201249	142.970120

Any of these **indexing conventions** may also be used to **set or modify values;**

this is done in the **standard way** that you might be accustomed to from working with **NumPy:**

In [12]:

```
data
```

Out[12]:

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [13]:

```
data.iloc[0, 2] = 90
data
```

Out[13]:

	area	pop	density
California	423967	39538223	90.000000
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

For **fluency in Pandas** data manipulation, **spend some time**

- with a simple `DataFrame` and exploring the types of
- indexing,
- slicing,
- masking,
- and fancy indexing

that are allowed by these various indexing approaches.

Additional Indexing Conventions

There are a **couple of extra indexing conventions** that might **seem at odds with the preceding discussion,**

but nevertheless **can be useful in practice.**

First, while **indexing refers to columns, slicing refers to rows:**

In [14]:
```
data
```

Out[14]:

	area	pop	density
California	423967	39538223	90.000000
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [ ]:

```
data['Florida':'New York']
```

Out[ ]:

	area	pop	density
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120

Such **slices** can also **refer to rows by number rather than by index:**

In [15]:
```
data
```

Out[15]:

	area	pop	density
California	423967	39538223	90.000000
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [ ]:
```
data[1:3]
```

Out[ ]:

	area	pop	density
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121

Similarly, **direct masking** operations are **interpreted row-wise** rather than column-wise:

In [16]:

```
data
```

Out[16]:

	area	pop	density
California	423967	39538223	90.000000
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

In [17]:

```
data.density > 120
```

Out[17]:

```
California      False
Texas           False
Florida          True
New York         True
Pennsylvania    False
Name: density, dtype: bool
```

In [ ]:
```
data[data.density > 120]
```

Out[ ]:

	area	pop	density
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120

These **two conventions** are **syntactically similar** to those on a **NumPy array,**

and while they **may not precisely fit the mold of the Pandas conventions,** they are **included due to their practical utility.**