03.02-Data-Indexing-and-Selection

June 27, 2018

This notebook contains an excerpt from the [Python Data Science Handbook](http://shop.oreilly.com/product/0636920034919.do) by Jake VanderPlas; the content is available [on GitHub](https://github.com/jakevdp/PythonDataScienceHandbook).

< Introducing Pandas Objects | Contents | Operating on Data in Pandas >

1 Data Indexing and Selection

In Chapter 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-dimesnional `DataFrame` object.

1.1 Data Selection in Series

As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

1.1.1 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 [2]: data['b']
```

```
Out[2]: 0.5
```

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

```
In [3]: 'a' in data
```

```
Out[3]: True
```

```
In [4]: data.keys()
```

```
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [5]: list(data.items())
```

```
Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can even 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 [6]: data['e'] = 1.25
        data
```

```
Out[6]: 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; the user generally does not need to worry about these issues.

1.1.2 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 [7]: # slicing by explicit index
        data['a':'c']
```

```
Out[7]: a    0.25
        b    0.50
        c    0.75
        dtype: float64
```

```
In [8]: # slicing by implicit integer index
        data[0:2]
```

```
Out[8]: a    0.25
        b    0.50
        dtype: float64
```

```
In [9]: # masking
        data[(data > 0.3) & (data < 0.8)]
```

```
Out[9]: b    0.50
        c    0.75
        dtype: float64
```

```
In [10]: # fancy indexing
         data[['a', 'e']]
```

```
Out[10]: a    0.25
         e    1.25
         dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

1.1.3 Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, 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 index.

```
In [11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
```

```
Out[11]: 1    a
         3    b
         5    c
         dtype: object
```

```
In [12]: # explicit index when indexing
         data[1]
```

```
Out[12]: 'a'
```

```
In [13]: # implicit index when slicing
         data[1:3]
```

```
Out[13]: 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 [14]: data.loc[1]

Out[14]: 'a'

In [15]: data.loc[1:3]

Out[15]: 1    a
         3    b
         dtype: object
```

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

```
In [16]: data.iloc[1]

Out[16]: 'b'

In [17]: data.iloc[1:3]

Out[17]: 3    b
         5    c
         dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for `Series` objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of `DataFrame` objects, which we will discuss in a moment.

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

1.2 Data Selection in DataFrame

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.

1.2.1 DataFrame as a 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 [18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                           'New York': 141297, 'Florida': 170312,
                           'Illinois': 149995})
         pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                          'New York': 19651127, 'Florida': 19552860,
                          'Illinois': 12882135})
         data = pd.DataFrame({'area':area, 'pop':pop})
         data
```

```
Out[18]:               area       pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
         New York    141297  19651127
         Texas       695662  26448193
```

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

```
In [19]: data['area']
```

```
Out[19]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

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

```
In [20]: data.area
```

```
Out[20]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

```
In [21]: data.area is data['area']
```

```
Out[21]: True
```

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 [22]: data.pop is data['pop']
```

```
Out[22]: False
```

In particular, you should avoid the temptation to try column assignment via attribute (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 [23]: data['density'] = data['pop'] / data['area']
         data
```

```
Out[23]:               area       pop     density
         California  423967  38332521   90.413926
         Florida     170312  19552860  114.806121
         Illinois    149995  12882135   85.883763
         New York    141297  19651127  139.076746
         Texas       695662  26448193   38.018740
```

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.

1.2.2 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 [24]: data.values
```

```
Out[24]: array([[  4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
                [  1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
                [  1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
                [  1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
                [  6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

```
In [25]: data.T
```

```
Out[25]:           California       Florida      Illinois      New York         Texas
         area     4.239670e+05  1.703120e+05  1.499950e+05  1.412970e+05  6.956620e+05
         pop      3.833252e+07  1.955286e+07  1.288214e+07  1.965113e+07  2.644819e+07
         density  9.041393e+01  1.148061e+02  8.588376e+01  1.390767e+02  3.801874e+01
```

When it comes to indexing of `DataFrame` objects, 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 [26]: data.values[0]
```

```
Out[26]: array([  4.23967000e+05,   3.83325210e+07,   9.04139261e+01])
```

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

```
In [27]: data['area']

Out[27]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

```
In [28]: data.iloc[:3, :2]

Out[28]:               area       pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
```

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 [29]: data.loc[:'Illinois', :'pop']

Out[29]:               area       pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
```

The `ix` indexer allows a hybrid of these two approaches:

```
In [30]: data.ix[:3, :'pop']

Out[30]:               area       pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
```

Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects.

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 in the following:

```
In [31]: data.loc[data.density > 100, ['pop', 'density']]
```

```
Out[31]:                 pop     density
         Florida   19552860  114.806121
         New York  19651127  139.076746
```

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

```
Out[32]:               area       pop     density
         California  423967  38332521   90.000000
         Florida     170312  19552860  114.806121
         Illinois    149995  12882135   85.883763
         New York    141297  19651127  139.076746
         Texas       695662  26448193   38.018740
```

To build up your fluency in Pandas data manipulation, I suggest spending 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.

1.2.3 Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

```
In [33]: data['Florida':'Illinois']
```

```
Out[33]:             area       pop     density
         Florida   170312  19552860  114.806121
         Illinois  149995  12882135   85.883763
```

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

```
In [34]: data[1:3]
```

```
Out[34]:             area       pop     density
         Florida   170312  19552860  114.806121
         Illinois  149995  12882135   85.883763
```

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

```
In [35]: data[data.density > 100]
```

```
Out[35]:             area       pop     density
         Florida   170312  19552860  114.806121
         New York  141297  19651127  139.076746
```

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

< Introducing Pandas Objects | Contents | Operating on Data in Pandas >