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.

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.

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 to add 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" on page 115.

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, we can do many familiar array-like observations 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
```

```
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.

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.

Operating on Data in Pandas

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in "Computation on NumPy Arrays: Universal Functions" on page 50 are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources—both potentially error-prone tasks with raw NumPy arrays—become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional `Series` structures and two-dimensional `DataFrame` structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas `Series` and `DataFrame` objects. Let's start by defining a simple `Series` and `DataFrame` on which to demonstrate this:

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

In[2]: rng = np.random.RandomState(42)
       ser = pd.Series(rng.randint(0, 10, 4))
       ser

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

In[3]: df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                         columns=['A', 'B', 'C', 'D'])
       df

Out[3]:    A  B  C  D
        0  6  9  2  6
        1  7  4  3  7
        2  7  2  5  4
```

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved:*

```
In[4]: np.exp(ser)
```