From a dictionary of Series objects. As we saw before, a `DataFrame` can be constructed from a dictionary of `Series` objects as well:

```
In[26]: pd.DataFrame({'population': population,
                      'area': area})
```

```
Out[26]:             area      population
         California  423967    38332521
         Florida     170312    19552860
         Illinois    149995    12882135
         New York    141297    19651127
         Texas       695662    26448193
```

From a two-dimensional NumPy array. Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
In[27]: pd.DataFrame(np.random.rand(3, 2),
                     columns=['foo', 'bar'],
                     index=['a', 'b', 'c'])
```

```
Out[27]:    foo       bar
         a  0.865257  0.213169
         b  0.442759  0.108267
         c  0.047110  0.905718
```

From a NumPy structured array. We covered structured arrays in "Structured Data: NumPy's Structured Arrays" on page 92. A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

```
In[28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
        A
```

```
Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])
```

```
In[29]: pd.DataFrame(A)
```

```
Out[29]:    A  B
         0  0  0.0
         1  0  0.0
         2  0  0.0
```

The Pandas Index Object

We have seen here that both the `Series` and `DataFrame` objects contain an explicit *index* that lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multiset, as `Index` objects may contain repeated values). Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an `Index` from a list of integers:

```
In[30]: ind = pd.Index([2, 3, 5, 7, 11])
        ind

Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The `Index` object in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
In[31]: ind[1]

Out[31]: 3

In[32]: ind[::2]

Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

`Index` objects also have many of the attributes familiar from NumPy arrays:

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)

5 (5,) 1 int64
```

One difference between `Index` objects and NumPy arrays is that indices are immutable—that is, they cannot be modified via the normal means:

```
In[34]: ind[1] = 0

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0


/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py ...
   1243
   1244     def __setitem__(self, key, value):
-> 1245         raise TypeError("Index does not support mutable operations")
   1246
   1247     def __getitem__(self, key):


TypeError: Index does not support mutable operations
```

This immutability makes it safer to share indices between multiple `DataFrame`s and arrays, without the potential for side effects from inadvertent index modification.

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The `Index` object follows many of

the conventions used by Python's built-in `set` data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
In[35]: indA = pd.Index([1, 3, 5, 7, 9])
        indB = pd.Index([2, 3, 5, 7, 11])

In[36]: indA & indB  # intersection

Out[36]: Int64Index([3, 5, 7], dtype='int64')

In[37]: indA | indB  # union

Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In[38]: indA ^ indB  # symmetric difference

Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods—for example, `indA.intersection(indB)`.

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-dimensional `DataFrame` object.

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.

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
```