

# 03.01-Introducing-Pandas-Objects

June 27, 2018

*This notebook contains an excerpt from the [Python Data Science Handbook](#) by Jake VanderPlas; the content is available [on GitHub](#).*

*The text is released under the [CC-BY-NC-ND license](#), and code is released under the [MIT license](#). If you find this content useful, please consider supporting the work by [buying the book](#)!*

*< [Data Manipulation with Pandas](#) | [Contents](#) | [Data Indexing and Selection](#) >*

## 1 Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's introduce these three fundamental Pandas data structures: the Series, DataFrame, and Index.

We will start our code sessions with the standard NumPy and Pandas imports:

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

### 1.1 The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
Out[2]: 0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

As we see in the output, the Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes. The values are simply a familiar NumPy array:

```
In [3]: data.values
```

```
Out [3]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

The index is an array-like object of type `pd.Index`, which we'll discuss in more detail momentarily.

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

```
Out [4]: RangeIndex(start=0, stop=4, step=1)
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
In [5]: data[1]
```

```
Out [5]: 0.5
```

```
In [6]: data[1:3]
```

```
Out [6]: 1    0.50
         2    0.75
         dtype: float64
```

As we will see, though, the Pandas Series is much more general and flexible than the one-dimensional NumPy array that it emulates.

### 1.1.1 Series as generalized NumPy array

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
In [7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
        data
```

```
Out [7]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         dtype: float64
```

And the item access works as expected:

```
In [8]: data['b']
```

```
Out [8]: 0.5
```

We can even use non-contiguous or non-sequential indices:

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=[2, 5, 3, 7])
```

```
data
```

```
Out[9]: 2    0.25
        5    0.50
        3    0.75
        7    1.00
        dtype: float64
```

```
In [10]: data[5]
```

```
Out[10]: 0.5
```

### 1.1.2 Series as specialized dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary:

```
In [11]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135}
        population = pd.Series(population_dict)
        population
```

```
Out[11]: California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
         dtype: int64
```

By default, a Series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```
In [12]: population['California']
```

```
Out[12]: 38332521
```

Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

```
In [13]: population['California':'Illinois']
```

```
Out[13]: California    38332521  
Florida      19552860  
Illinois     12882135  
dtype: int64
```

We'll discuss some of the quirks of Pandas indexing and slicing in [Data Indexing and Selection](#).

### 1.1.3 Constructing Series objects

We've already seen a few ways of constructing a Pandas Series from scratch; all of them are some version of the following:

```
>>> pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities.

For example, data can be a list or NumPy array, in which case index defaults to an integer sequence:

```
In [14]: pd.Series([2, 4, 6])
```

```
Out[14]: 0    2  
         1    4  
         2    6  
dtype: int64
```

data can be a scalar, which is repeated to fill the specified index:

```
In [15]: pd.Series(5, index=[100, 200, 300])
```

```
Out[15]: 100    5  
         200    5  
         300    5  
dtype: int64
```

data can be a dictionary, in which index defaults to the sorted dictionary keys:

```
In [16]: pd.Series({'2': 'a', 1: 'b', 3: 'c'})
```

```
Out[16]: 1    b  
         2    a  
         3    c  
dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
In [17]: pd.Series({'2': 'a', 1: 'b', 3: 'c'}, index=[3, 2])
```

```
Out[17]: 3    c  
         2    a  
dtype: object
```

Notice that in this case, the Series is populated only with the explicitly identified keys.

## 1.2 The Pandas DataFrame Object

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` object discussed in the previous section, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

### 1.2.1 DataFrame as a generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by "aligned" we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states discussed in the previous section:

```
In [18]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                    'Florida': 170312, 'Illinois': 149995}
         area = pd.Series(area_dict)
         area

Out[18]: California    423967
         Florida      170312
         Illinois     149995
         New York     141297
         Texas        695662
         dtype: int64
```

Now that we have this along with the population `Series` from before, we can use a dictionary to construct a single two-dimensional object containing this information:

```
In [19]: states = pd.DataFrame({'population': population,
                              'area': area})
         states

Out[19]:
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

```
In [20]: states.index

Out[20]: Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the `DataFrame` has a `columns` attribute, which is an `Index` object holding the column labels:

```
In [21]: states.columns
```

```
Out[21]: Index(['area', 'population'], dtype='object')
```

Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

### 1.2.2 DataFrame as specialized dictionary

Similarly, we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier:

```
In [22]: states['area']
```

```
Out[22]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimensional NumPy array, `data[0]` will return the first *row*. For a DataFrame, `data['col0']` will return the first *column*. Because of this, it is probably better to think about DataFrames as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing DataFrames in [Data Indexing and Selection](#).

### 1.2.3 Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

**From a single Series object** A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series:

```
In [23]: pd.DataFrame(population, columns=['population'])
```

```
Out[23]:      population
California  38332521
Florida    19552860
Illinois   12882135
New York   19651127
Texas      26448193
```

**From a list of dicts** Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data:

```
In [24]: data = [{'a': i, 'b': 2 * i}
                  for i in range(3)]
          pd.DataFrame(data)
```

```
Out [24]:
```

	a	b
0	0	0
1	1	2
2	2	4

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values:

```
In [25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
Out [25]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

**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](#). 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

### 1.3 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 multi-set, 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')
```

#### 1.3.1 Index as immutable array

The `Index` 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 in __setitem__
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 DataFrames and arrays, without the potential for side effects from inadvertent index modification.

### 1.3.2 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 Manipulation with Pandas](#) | [Contents](#) | [Data Indexing and Selection](#) >