Introducing Pandas Objects By Mrittika Megaraj

Pandas objects, at their most fundamental level, may be seen as augmented versions of NumPy structured arrays, with rows and columns identifiable by labels rather than plain integer indices. Pandas, as we shall see throughout this chapter, offers a plethora of valuable tools, methods, and functionality on top of the fundamental data structures, but almost everything that follows will need an awareness of what these structures are. Now, then, before we proceed any farther, 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:

```
import numpy as np
import pandas as pd
```

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:

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

```
data.values
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.

```
data.index
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:

```
data[1]
0.5
data[1:3]
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.

`Series` as generalized NumPy array

So far, it may seem that the Series object may be replaced with a standard NumPy array of a single dimension. The index makes all the difference: in a Numpy Array, the integer index is defined implicitly and is used to access the values, but in a Pandas Series, the index is declared explicitly and is used to access 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:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And the item access works as expected:

```
data['b']
0.5
```

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

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
data[5]
0.5
```

Series as specialized dictionary

This makes a Pandas Series similar to a customized version of Python's dictionary. A dictionary is a data structure that associates each key with a list of values, whereas a series is a data structure that associates each key with a list of values that have been predefined by the user. Pandas Series, according to its type information, is substantially more efficient than Python dictionaries for some operations, much as NumPy arrays are thanks to the type-specific built code underneath them.

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

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

```
population['California']
38332521
```

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

```
population['California':'Illinois']
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.

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:

```
pd.Series([2, 4, 6])
0    2
1    4
2    6
dtype: int64
```

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

```
pd.Series(5, index=[100, 200, 300])
100    5
200    5
300    5
dtype: int64
```

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

```
pd.Series({2:'a', 1:'b', 3:'c'})
1    b
2    a
3    c
dtype: object
```

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

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

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

The Pandas DataFrame Object

DataFrame is Pandas' next primary structure. The DataFrame is similar to the Series object covered before in that it may be seen as either a generalization of a NumPy array or a specialization of a Python dictionary. We're going to examine all of these points of view immediately.

DataFrame as a generalized NumPy array

DataFrames are similar to two-dimensional arrays with adjustable row indices and column names, whereas Series are similar to one-dimensional arrays with flexible indices. As a two-dimensional array is a series of aligned one-dimensional columns, so too is a DataFrame a sequence of aligned Series objects. We say that two things are "aligned" if they have the same index in this context.

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

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

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

	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:

```
states.index
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:

```
states.columns
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.

DataFrame as specialized dictionary

A DataFrame may be conceptualized in the same way, as a more specialized form of a dictionary. DataFrames are similar to dictionaries, except that instead of associating a key with a value, they associate a column name with a Series of column data. When we query for the property "area," for instance, we get back the Series object that contains the "areas" we saw:

```
states['area']
```

```
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-dimesnional 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 `DataFrame`s as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful.

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`:

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

	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:

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

	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:

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

	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:

```
pd.DataFrame({'population': population,
              'area': area})
```

	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:

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

	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

From a NumPy structured array

Structured arrays were discussed in detail in Structured Data: Structured Arrays in NumPy. As with other structured arrays, a Pandas DataFrame may be created from an existing one.

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('A', '<i8'), ('B', '<f8')])
pd.DataFrame(A)
```

	A	B
0	0	0.0
1	0	0.0

	A	B
2	0	0.0

The Pandas Index Object

As we've seen, both Series and DataFrame objects include an explicit index that may be used for data reference and manipulation. The Index object has a unique structure that may be interpreted in two different ways: as an immutable array or a sorted set (technically a multi-set, as Index objects may contain repeated values). The effects of these perspectives on the possible actions with Index items are intriguing. Let's create an Index as a basic example using a list of numbers.:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

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:

```
ind[1]
3
ind[::2]
Int64Index([2, 5, 11], dtype='int64')
```

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

```
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:

```
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__(self, key, value)
   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 intended to simplify tasks that rely on set arithmetic, such as joining many datasets together. The Index object conforms to many of the same rules as Python's built-in set data structure, allowing for the familiar computation of unions, intersections, differences, and other combinations:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
indA & indB  # intersection
Int64Index([3, 5, 7], dtype='int64')
indA | indB  # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
indA ^ indB  # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

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

Operating on Data in Pandas

The ability to do rapid element-wise operations, both with standard arithmetic (adding, subtracting, multiplying, and dividing) and with more advanced operations, is a crucial part of NumPy (trigonometric functions, exponential and logarithmic functions, etc.). Most of Pandas' features are carried over from NumPy; the ufuncs we presented in Computation on NumPy Arrays: Universal Functions are particularly important here.

But Pandas has a few neat extras: for unary operations like negation and trigonometric functions, the output of these ufuncs will keep the index and column labels intact; and for binary operations like addition and multiplication, Pandas will automatically align the indices of the objects being passed to the ufunc. Pandas makes activities like preserving data context and merging data from several sources, which may be prone to mistakes when working with raw NumPy arrays, almost error-proof. It will also become clear that there are well-defined operations between Series structures of one dimension and DataFrame structures of two dimensions.

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:

```
import pandas as pd
import numpy as np
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
0    6
1    3
2    7
3    4
dtype: int64
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	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:*

```
np.exp(ser)
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

```
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

UFuncs: Index Alignment

For binary operations on two `Series` or `DataFrame` objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we'll see in some of the examples that follow.

Index alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
```

Let's see what happens when we divide these to compute the population density:

```
population / area
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which could be determined using standard Python set arithmetic on these indices:

```
area.index | population.index
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked with `NaN`, or "Not a Number," which is how Pandas marks missing data This index matching is implemented this way for any of Python's built-in arithmetic expressions; any missing values are filled in with NaN by default:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

If using NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling `A.add(B)` is equivalent to calling `A + B`, but allows optional explicit specification of the fill value for any elements in `A` or `B` that might be missing:

```
A.add(B, fill_value=0)
0    2.0
1    5.0
2    9.0
3    5.0
dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on `DataFrame`s:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=list('AB'))
A
```

	A	B
0	1	11
1	5	1

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

```
A + B
```

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries. Here we'll fill with the mean of all values in `A` (computed by first stacking the rows of `A`):

```
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Ufuncs: Operations Between DataFrame and Series

Index and column alignment are also preserved during operations between DataFrame and Series. The similarities between working with a NumPy two-dimensional array and a one-dimensional Series are obvious. Take the case of determining the difference between a two-dimensional array and a single row:

```
A = rng.randint(10, size=(3, 4))
A
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])
A - A[0]
array([[ 0,  0,  0,  0],
       [-1, -2,  2,  4],
       [ 3, -7,  1,  4]])
```

According to NumPy's broadcasting rules, subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the `axis` keyword:

```
df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

Note that these `DataFrame`/`Series` operations, like the operations discussed above, will automatically align indices between the two elements:

```
halfrow = df.iloc[0, ::2]
halfrow
Q    3
S    2
Name: 0, dtype: int64
df - halfrow
```

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	-1.0	NaN	2.0	NaN
2	3.0	NaN	1.0	NaN

In contrast to working with heterogeneous and/or misaligned data in raw NumPy arrays, operations on data in Pandas will always maintain the data context thanks to the preservation and alignment of indices and columns.