Introducing Pandas Objects

At a 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 take a look at 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
```

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

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

The `Series` **combines a sequence of values with an explicit sequence of indices,** which we can **access** with the `values` and `index` attributes.

The `values` **are simply a familiar NumPy array:**

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

Out[ ]:
```
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 [ ]:
```
data.index
```

Out[ ]:
```
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 [ ]:
```
data[1]
```

Out[ ]:
```
0.5
```

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

Out[ ]:
```
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

From what we've seen so far, the `Series` object **may appear** to be basically **interchangeable** with a **one-dimensional NumPy array.**

The **essential difference** is that 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.**

So, if we wish, we can use **strings as an index:**

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

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

And the **item access works as expected:**

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

Out[ ]:
```
0.5
```

We can even use **noncontiguous** or **nonsequential indices:**

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

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

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

Out[ ]:
```
0.5
```

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 that **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 **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.**

Here the **five most populous US states** according to the 2020 census:

In [ ]:

```
population_dict = {'California': 39538223, 'Texas': 29145505,
                   'Florida': 21538187, 'New York': 20201249,
                   'Pennsylvania': 13002700}
population = pd.Series(population_dict)
population
```

Out[ ]:
```
California      39538223
Texas           29145505
Florida         21538187
New York        20201249
Pennsylvania    13002700
dtype: int64
```

From here, **typical dictionary-style item access can be performed:**

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

Out[ ]:
```
39538223
```

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

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

```
Out[ ]: California    39538223
        Texas         29145505
        Florida       21538187
        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:

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

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

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

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

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

Or it can be a **dictionary,** in which case `index` **defaults to the dictionary keys:**

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

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

In **each case,** the **index can be explicitly set to control the order** or the **subset of keys** used:

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

```
Out[ ]: 1    b
        2    a
        dtype: object
```

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.

DataFrame as Generalized NumPy Array

If a `Series` **is an analog of a one-dimensional array with explicit indices,** a `DataFrame` **is an analog of a two-dimensional array with explicit row and column indices.**

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 square kilometers):

In [ ]:

```
area_dict = {'California': 423967, 'Texas': 695662, 'Florida'
             'New York': 141297, 'Pennsylvania': 119280}
area = pd.Series(area_dict)
area
```

Out[ ]:

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
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 [ ]:

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

Out[ ]:

	population	area
California	39538223	423967
Texas	29145505	695662
Florida	21538187	170312
New York	20201249	141297
Pennsylvania	13002700	119280

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

In [ ]:
```
states.index
```

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

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

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

Out[ ]:
```
Index(['population', 'area'], 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

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

Out[ ]:
```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
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** `DataFrame`**s** 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** `DataFrame` **s** in **Data Indexing and Selection.**

Constructing DataFrame Objects

A Pandas `DataFrame` **can be constructed in a variety of ways.**

Here we'll explore **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 [ ]:
```
pd.DataFrame(population, columns=['population'])
```

Out[ ]:

	population
California	39538223
Texas	29145505
Florida	21538187
New York	20201249
Pennsylvania	13002700

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

Out[2]:
```
[{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'b': 4}]
```

In [3]:
```
pd.DataFrame(data)
```

Out[3]:

	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` **values** (i.e., "Not a Number"; see **Handling Missing Data**):

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

Out[ ]:

	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 [ ]:

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

Out[ ]:

	population	area
California	39538223	423967
Texas	29145505	695662
Florida	21538187	170312
New York	20201249	141297
Pennsylvania	13002700	119280

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 [4]:
```
np.random.rand(3, 2)
```

Out[4]:
```
array([[0.89150021, 0.52792497],
       [0.55672672, 0.19928865],
       [0.61379592, 0.72308653]])
```

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

Out[ ]:

	foo	bar
a	0.471098	0.317396
b	0.614766	0.305971
c	0.533596	0.512377

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

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

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

Out[ ]:

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

The Pandas Index Object

As you've seen, the `Series` **and** `DataFrame` **objects both 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 terms of the **operations available on `Index` objects.**

As a simple **example,** let's **construct an `Index` from a list of integers:**

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

Out[ ]:
```
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:**

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

Out[ ]: 3

In [ ]:
```
ind[::2]
```

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

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

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

```
5 (5,) 1 int64
```

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/xc/sptt9bk14s34rgxt7453p03r0000gp/T/ipykernel_83282/393126374.py in <module>
----> 1 ind[1] = 0

~/.local/share/virtualenvs/python-data-science-handbook-2e-u_kwqDTB/lib/python3.9/site-packages/pandas/core/indexes/base.py in __setitem__(self, key, value)
   4583     @final
   4584     def __setitem__(self, key, value):
-> 4585         raise TypeError("Index does not support mutable operations")
   4586
   4587     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 [ ]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [ ]:
```
indA.intersection(indB)
```

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

In [ ]:
```
indA.union(indB)
```

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

In [ ]:
```
indA.symmetric_difference(indB)
```

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