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 lab, 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 [2]:
```
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 [3]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Out[3]:
```
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 [4]:
```
data.values
```

Out[4]:
```
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 [5]:
```
data.index
```

Out[5]:
```
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 [6]:
```
data[1]
```

Out[6]:
```
0.5
```

In [7]:
```
data[1:3]
```

Out[7]:
```
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, 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 [8]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[8]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And the item access works as expected:

In [9]:
```
data['b']
```

Out[9]:
```
0.5
```

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

In [10]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

Out[10]:
```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

In [11]:
```
data[5]
```

Out[11]:
```
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 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 [12]:
```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[12]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
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 [13]:
```
population['California']
```

Out[13]:
```
38332521
```

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

In [14]:
```python
population['California':'Illinois']
```

Out[14]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

We'll discuss some of the quirks of Pandas indexing and slicing in [Data Indexing and Selection (03.02-Data-Indexing-and-Selection.ipynb)](03.02-Data-Indexing-and-Selection.ipynb).

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:

```python
>>> 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 [15]:
```python
pd.Series([2, 4, 6])
```

Out[15]:
```
0    2
1    4
2    6
dtype: int64
```

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

In [16]:
```python
pd.Series(5, index=[100, 200, 300])
```

Out[16]:
```
100    5
200    5
300    5
dtype: int64
```

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

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

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

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

In [18]:
```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

Out[18]:
```
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

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 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 [19]:
```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

Out[19]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
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 [20]:
```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Out[20]:

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

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

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

Out[21]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='obje
ct')
```

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

In [22]:
```
states.columns
```

Out[22]:
```
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 [23]:
```
states['area']
```

Out[23]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
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. We'll explore more flexible means of indexing `DataFrame` s in [Data Indexing and Selection (03.02-Data-Indexing-and-Selection.ipynb)](03.02-Data-Indexing-and-Selection.ipynb).

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 [24]:
```
pd.DataFrame(population, columns=['population'])
```

Out[24]:

	population
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

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 [25]:
```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

Out[25]:

	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 [26]:
```python
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[26]:

	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 [27]:
```python
pd.DataFrame({'population': population,
              'area': area})
```

Out[27]:

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

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 [28]:
```python
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Out[28]:

	foo	bar
a	0.858219	0.939572
b	0.770332	0.423583
c	0.920911	0.218402

From a NumPy structured array

We covered structured arrays in [Structured Data: NumPy's Structured Arrays (02.09-Structured-Data-NumPy.ipynb)](02.09-Structured-Data-NumPy.ipynb). A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

In [29]:
```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
```

Out[29]:
```
array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])
```

In [30]:
```
pd.DataFrame(A)
```

Out[30]:

	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 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 [31]:
```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[31]:
```
Index([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 [32]:
```
ind[1]
```

Out[32]:
```
3
```

In [33]:
```
ind[::2]
```

Out[33]:
```
Index([2, 5, 11], dtype='int64')
```

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

In [34]:
```
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 [35]:
```
ind[1] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 ind[1] = 0

File ~\anaconda3\Lib\site-packages\pandas\core\indexes\base.py:5157, in Inde
x.__setitem__(self, key, value)
   5155 @final
   5156 def __setitem__(self, key, value):
-> 5157     raise TypeError("Index does not support mutable operations")

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 [36]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [45]:
```
indA & indB  # intersection
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_6456\76773435.py:1: FutureWarning:
Index.__and__ operating as a set operation is deprecated, in the future this
will be a logical operation matching Series.__and__.  Use index.intersection
(other) instead.
  indA & indB  # intersection
```

Out[45]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [46]:
```
indA | indB  # union
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_6456\2015131817.py:1: FutureWarnin
g: Index.__or__ operating as a set operation is deprecated, in the future thi
s will be a logical operation matching Series.__or__.  Use index.union(other)
instead.
  indA | indB  # union
```

Out[46]:
```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

In [47]:
```
indA ^ indB  # symmetric difference
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_6456\1234474844.py:1: FutureWarnin
g: Index.__xor__ operating as a set operation is deprecated, in the future th
is will be a logical operation matching Series.__xor__.  Use index.symmetric_
difference(other) instead.
  indA ^ indB  # symmetric difference
```

Out[47]:
```
Int64Index([1, 2, 9, 11], dtype='int64')
```

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