Lecture 7: Pandas

Data Science, DST, UIC

In the previous lectures, we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. In this section, we will look in detail at the data structures provided by the **Pandas** library.

- Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a `DataFrame`.
- `DataFrame`'s are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.
- As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's `ndarray` data structure provides essential features for the type of **clean, well-organized data** typically seen in **numerical computing** tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.), which is an important piece of analyzing the less structured data available in many forms in the world around us.

Installing and Using Pandas

Installation of Pandas on your system requires NumPy to be installed, and if building the library from source, requires the appropriate tools to compile the C and Cython sources on which Pandas is built. Details on this installation can be found in the Pandas documentation (http://pandas.pydata.org/). If you installed Anaconda, you already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
In [1]: import pandas
        pandas.__version__
```

```
Out[1]: '1.2.2'
```

Just as we generally import NumPy under the alias `np`, we will import Pandas under the alias `pd`:

```
In [2]: import pandas as pd
```

This import convention will be used throughout the remainder of our notes.

1 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**.

- 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 [3]:
```
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 [4]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

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

Out[5]:
```
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 [6]:
```
data.index
```

Out[6]:
```
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 [7]:
```
data[1]
```

Out[7]:
```
0.5
```

In [8]:
```
data[1:3] # slicing using implicit index
```

Out[8]:
```
1    0.50
2    0.75
dtype: float64
```

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 [9]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[9]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And the item access works as expected:

In [10]:
```
print(data['a']) # explicit index
print(data[0]) # implicit index
```

```
0.25
0.25
```

In [11]:
```
data.index
```

Out[11]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

The dtype object comes from NumPy, it describes the type of element in a ndarray. Every element in a ndarray must has the same size in byte. For int64 and float64, they are 8 bytes. But for strings, the length of the string is not fixed. So instead of save the bytes of strings in the ndarray directly, Pandas use object ndarray, which save pointers to objects, because of this the dtype of this kind ndarray is object.

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

In [12]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
data
```

Out[12]:
```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

In [13]:
```
print(data[5])
# print(data[0]) #invalid key, no implicit positional index if your index is integer
```

```
0.5
```

In [14]:
```
data.index
```

Out[14]:
```
Int64Index([2, 5, 3, 7], dtype='int64')
```

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 [15]:
```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[15]:
```
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 [16]:
```
population['California']
```

Out[16]:
```
38332521
```

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

In [17]:
```
population['California':'Florida']
```

Out[17]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
dtype: int64
```

In [18]:
```
population[0:3]
```

Out[18]:
```
California    38332521
Texas         26448193
New York      19651127
dtype: int64
```

In [19]:
```
population.index
```

Out[19]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

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

```
Out[20]: 0    2
         1    4
         2    6
         dtype: int64
```

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

```
In [21]: pd.Series(5, index=[100,200,300])
```

```
Out[21]: 100    5
         200    5
         300    5
         dtype: int64
```

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

```
In [22]: pd.Series({2:'a', 1:'b', 3:'c'})
```

```
Out[22]: 2    a
         1    b
         3    c
         dtype: object
```

The index can be explicitly set if a different result is preferred:

```
In [23]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

```
Out[23]: 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.

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 [24]:
```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

Out[24]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

In [25]:
```
population
```

Out[25]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
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 [26]:
```
states = pd.DataFrame({'population': population,
                       'area': area})  # a dictionary of series objects
states
```

Out[26]:

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

In [27]:
```
states.values
```

Out[27]:
```
array([[38332521,   423967],
       [26448193,   695662],
       [19651127,   141297],
       [19552860,   170312],
       [12882135,   149995]])
```

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

```
In [28]: states.index
Out[28]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

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

```
In [29]: states.columns
Out[29]: 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.

- 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 [30]: states['area']
Out[30]: California    423967
         Texas         695662
         New York      141297
         Florida       170312
         Illinois      149995
         Name: area, dtype: int64
```

You can get the specific column as a Series by following code:

```
In [31]: states[states.columns[0]]
Out[31]: California    38332521
         Texas         26448193
         New York      19651127
         Florida       19552860
         Illinois      12882135
         Name: population, dtype: int64
```

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 [32]:
```
pd.DataFrame(population)
```

Out[32]:

	0
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [33]:
```
pd.DataFrame([population,area], index=['population','area']).transpose()
```

Out[33]:

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

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 [34]:
```
data = [{'a': i, 'b': 2 * i} for i in range (3)]
pd.DataFrame(data)
```

Out[34]:

	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 [35]:
```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[35]:

	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 [36]:
```
pd.DataFrame({'population': population,
              'area': area})
```

Out[36]:

	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 [37]:
```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Out[37]:

	foo	bar
a	0.831420	0.921539
b	0.181863	0.174391
c	0.490238	0.355947

In [38]:
```
pd.DataFrame(np.random.rand(4))
```

Out[38]:

	0
0	0.137227
1	0.864736
2	0.730310
3	0.585341

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).

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

In [39]:
```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[39]:
```
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 [40]: ind[1]
Out[40]: 3

In [41]: ind[::2]
Out[41]: Int64Index([2, 5, 11], dtype='int64')
```

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

```
In [42]: 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 [43]: #ind[1] = 0
```

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 via index methods:

```
In [44]: indA = pd.Index([1, 3, 5, 7, 9])
         indB = pd.Index([2, 3, 5, 7, 11])

In [45]: indA.intersection(indB)  # intersection
Out[45]: Int64Index([3, 5, 7], dtype='int64')

In [46]: indA.union(indB)  # union
Out[46]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [47]: indA.symmetric_difference(indB)  # symmetric difference
Out[47]: Int64Index([1, 2, 9, 11], dtype='int64')
```

2 Data Indexing and Selection

In the previous topic, 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.

We'll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimesnional `DataFrame` object.

2.1 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 [48]: import pandas as pd
         data = pd.Series([0.25, 0.5, 0.75, 1.0],
                          index=['a', 'b', 'c', 'd'])
         data
```

```
Out[48]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         dtype: float64
```

```
In [49]: data['b']
```

```
Out[49]: 0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
In [50]: 'a' in data
```

```
Out[50]: True
```

```
In [51]: data.keys()
```

```
Out[51]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [52]: list(data.items())
```

```
Out[52]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a `Series` by assigning to a new index value:

```
In [53]: data['e'] = 1.25
         data
```

```
Out[53]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         e    1.25
         dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A `Series` builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

```
In [54]: # slicing by explicit index
         data['a':'c']
```

```
Out[54]: a    0.25
         b    0.50
         c    0.75
         dtype: float64
```

```
In [55]: # slicing by implicit integer index
         data[0:2]
```

```
Out[55]: a    0.25
         b    0.50
         dtype: float64
```

```
In [56]: # masking
         data[(data > 0.3) & (data < 0.8)]
```

```
Out[56]: b    0.50
         c    0.75
         dtype: float64
```

```
In [57]: # fancy indexing
         data[['a', 'e']]
```

```
Out[57]: a    0.25
         e    1.25
         dtype: float64
```

Notice that when slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

Indexers: loc and iloc

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
In [58]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
Out[58]: 1    a
         3    b
         5    c
         dtype: object

In [59]: # explicit index when indexing
         data[1]
Out[59]: 'a'

In [60]: # implicit index when slicing
         data[1:3]
Out[60]: 3    b
         5    c
         dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series`.

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

```
In [61]: data.loc[1]
Out[61]: 'a'

In [62]: data.loc[1:3] # explicit slicing end include
Out[62]: 1    a
         3    b
         dtype: object
```

The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

```
In [63]: data.iloc[1]
Out[63]: 'b'

In [64]: data.iloc[1:3] # implicit slicing, end exclude
Out[64]: 3    b
         5    c
         dtype: object
```

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, it is recommended using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

2.2 Data Selection in DataFrame

Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let's return to our example of areas and populations of states:

In [65]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[65]:

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

The individual `Series` that make up the columns of the `DataFrame` can be accessed via dictionary-style indexing of the column name:

In [66]:
```
data['area']
```

Out[66]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

In [67]:
```
data.area
```

Out[67]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

In [68]:
```
data.area is data['area']
```

Out[68]:
```
True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame`, this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the `"pop"` column:

In [69]:
```
data.pop is data['pop']
```

Out[69]:
```
False
```

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use `data['pop'] = z` rather than `data.pop = z`).

Like with the `Series` objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

In [70]:
```
data['density'] = data['pop'] / data['area']
data
```

Out[70]:

	area	pop	density
California	423967	38332521	90.413926
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

In [71]:
```
data.values
```

Out[71]:
```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

In [72]:
```
data
```

Out[72]:

	area	pop	density
California	423967	38332521	90.413926
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

In [73]:
```
data.T
```

Out[73]:

	California	Texas	New York	Florida	Illinois
area	4.239670e+05	6.956620e+05	1.412970e+05	1.703120e+05	1.499950e+05
pop	3.833252e+07	2.644819e+07	1.965113e+07	1.955286e+07	1.288214e+07
density	9.041393e+01	3.801874e+01	1.390767e+02	1.148061e+02	8.588376e+01

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

In [74]:
```
data.values[0]
```

Out[74]:
```
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

and **passing a single "index" to a `DataFrame` accesses a column**:

In [75]:
```
data['area']
```

Out[75]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc` and `iloc` indexers mentioned earlier.

Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

In [76]:
```
data.iloc[:3, :2]
```

Out[76]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

In [77]:
```
data.loc[:'Illinois', :'pop']
```

Out[77]:

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

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

In [78]:
```
data.loc[data.density > 100, ['pop', 'density']]
```

Out[78]:

	pop	density
New York	19651127	139.076746
Florida	19552860	114.806121

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

In [79]:
```
data.iloc[0, 2] = 90
data
```

Out[79]:

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice.

- First, **while *indexing* refers to columns, *slicing* refers to rows**:

In [80]:
```
data['Florida':'Illinois']
```

Out[80]:

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Such slices can also refer to rows by number rather than by index:

In [81]:
```
data[1:3] # slicing with positional row index
```

Out[81]:

	area	pop	density
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746

- Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

In [82]:
```
data[data.density > 100]
```

Out[82]:

	area	pop	density
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

3 Operating on Data in Pandas

- One of the essential pieces of NumPy is the ability to perform quick element-wise operations,e.g.,
  - with basic arithmetic (addition, subtraction, multiplication, etc.)
  - with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.).
- Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in numpy computation section are key to this.
- Pandas includes a couple useful twists:
  - for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output
  - for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc.

This means that keeping the context of data and combining data from different sources–both potentially error-prone tasks with raw NumPy arrays–become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional `Series` structures and two-dimensional `DataFrame` structures.

3.1 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:

In [83]:
```
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
```

Out[83]:
```
0    6
1    3
2    7
3    4
dtype: int64
```

In [84]:
```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

Out[84]:

	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:*

In [85]:
```
np.exp?
np.exp(ser)
```

Out[85]:
```
0      403.428793
1       20.085537
2     1096.633158
3       54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

In [86]:
```
np.sin(df * np.pi / 4)
```

Out[86]:

	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

Any of the ufuncs discussed in the Computation on NumPy Arrays section can be used in a similar manner.

3.2 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*:

In [87]:
```
# The name argument allows you to give a name to a Series object, i.e. to the column.
# So that when you'll put that in a DataFrame, the column will be named according to the name pa
rameter.

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:

In [88]:
```
dfarea = pd.DataFrame(area)
dfpopulation = pd.DataFrame(population)
dfarea
```

Out[88]:

	area
Alaska	1723337
Texas	695662
California	423967

In [89]:
```
dfpopulation
```

Out[89]:

	population
California	38332521
Texas	26448193
New York	19651127

In [90]:
```
df_result = pd.DataFrame(population / area, columns=['Density'])
df_result
```

Out[90]:

	Density
Alaska	NaN
California	90.413926
New York	NaN
Texas	38.018740

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:

In [91]:
```
area.index.union(population.index)
```

Out[91]:
```
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:

In [92]:
```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

Out[92]:
```
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:

In [93]:
```
A.add(B, fill_value=0)
```

Out[93]:
```
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:

In [94]:
```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=list('AB'))
A
```

Out[94]:

	A	B
0	1	11
1	5	1

In [95]:
```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

Out[95]:

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

In [96]:
```
A + B
```

Out[96]:

	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`):

In [97]:
```
# stack method will stack the prescribed level(s) from columns to index.
A.stack()
```

Out[97]:
```
0  A     1
   B    11
1  A     5
   B     1
dtype: int64
```

In [98]:
```
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

Out[98]:

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

In [99]:
```
A.stack?
```

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()

3.3 Ufuncs: Operations Between DataFrame and Series

When performing operations between a `DataFrame` and a `Series`, the index and column alignment is similarly maintained. Operations between a `DataFrame` and a `Series` are similar to operations between a two-dimensional and one-dimensional NumPy array. Consider one common operation, where we find the difference of a two-dimensional array and one of its rows:

In [100]:
```
A = rng.randint(10, size=(3, 4))
A
```

Out[100]:
```
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])
```

In [101]:
```
A - A[0] # broadcasting
```

Out[101]:
```
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:

In [102]:
```
df = pd.DataFrame(A, columns=list('QRST'))
df
```

Out[102]:

	Q	R	S	T
0	3	8	2	4
1	2	6	4	8
2	6	1	3	8

In [103]:
```
df - df.iloc[0]
```

Out[103]:

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

In [104]:
```
df['R']
```

Out[104]:
```
0    8
1    6
2    1
Name: R, dtype: int64
```

In [105]:
```
df - df['R']
```

Out[105]:

	Q	R	S	T	0	1	2
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

In [106]:
```
# Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1)
df.subtract(df['R'], axis=0)
```

Out[106]:

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

In [107]:
```
df.subtract(df['R'],axis='index')
```

Out[107]:

	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:

In [108]:
```
halfrow = df.iloc[0, ::2]
halfrow
```

Out[108]:
```
Q    3
S    2
Name: 0, dtype: int64
```

In [109]:
```
print(df)
df - halfrow # align column lables to perform row-wise subtraction
#Q: 3, R: NaN, S:2, T: NaN
```

```
   Q  R  S  T
0  3  8  2  4
1  2  6  4  8
2  6  1  3  8
```

Out[109]:

	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

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when working with heterogeneous and/or misaligned data in raw NumPy arrays.