This notebook was adapted from the [Python Data Science Handbook (http://shop.oreilly.com/product/0636920034919.do)](http://shop.oreilly.com/product/0636920034919.do) by Jake VanderPlas; the content is available [on GitHub (https://github.com/jakevdp/PythonDataScienceHandbook)](https://github.com/jakevdp/PythonDataScienceHandbook).

Data Manipulation with Pandas

Pandas is a 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.

Installing Pandas

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]:

```
'0.22.0'
```

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

In [2]:

```
import pandas as pd
```

Introducing Pandas Objects

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
```

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.  ])
```

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
```

`Series` as generalized NumPy array

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
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 [13]:

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

Out[13]:

```
38332521
```

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

In [14]:

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

Out[14]:

```
California    38332521
Florida       19552860
Illinois      12882135
dtype: int64
```

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 [15]:

```
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]:

```
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]:

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

Out[17]:

```
1    b
2    a
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
```

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.

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,
area = pd.Series(area_dict)
area
```

Out[19]:

```
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 [20]:

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

Out[20]:

	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 [21]:

```
states.index
```

Out[21]:

```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='obj
ect')
```

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

In [22]:

```
states.columns
```

Out[22]:

```
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

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
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`:

In [24]:

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

Out[24]:

	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 [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]:

```
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]:

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

Out[27]:

	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 [28]:

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

Out[28]:

	foo	bar
a	0.273574	0.433513
b	0.545063	0.020087
c	0.989773	0.487844

From a NumPy structured array

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]:

```
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 [32]:

```
ind[1]
```

Out[32]:

```
3
```

In [33]:

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

Out[33]:

```
Int64Index([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)
<ipython-input-35-906a9fa1424c> in <module>()
----> 1 ind[1] = 0

D:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(se
lf, key, value)
   1722
   1723     def __setitem__(self, key, value):
-> 1724         raise TypeError("Index does not support mutable operations")
   1725
   1726     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

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 [37]:

```
indA & indB  # intersection
```

Out[37]:

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

In [38]:

```
indA | indB  # union
```

Out[38]:

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

In [39]:

```
indA ^ indB  # symmetric difference
```

Out[39]:

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

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

Data Indexing and Selection

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 [40]:

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

Out[40]:

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

In [41]:

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

Out[41]:

```
0.5
```

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

In [42]:

```
'a' in data
```

Out[42]:

```
True
```

In [43]:

```
data.keys()
```

Out[43]:

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

In [44]:

```
list(data.items())
```

Out[44]:

```
[('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 [45]:

```
data['e'] = 1.25
data
```

Out[45]:

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

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 [46]:

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

Out[46]:

```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [47]:

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

Out[47]:

```
a    0.25
b    0.50
dtype: float64
```

In [48]:

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

Out[48]:

```
b    0.50
c    0.75
dtype: float64
```

In [49]:

```
# fancy indexing
data[['a', 'e']]
```

Out[49]:

```
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. 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 [50]:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[50]:

```
1    a
3    b
5    c
dtype: object
```

In [51]:

```
# explicit index when indexing
data[1]
```

Out[51]:

```
'a'
```

In [52]:

```
# implicit index when slicing
data[1:3]
```

Out[52]:

```
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 [53]:

```
data.loc[1]
```

Out[53]:

```
'a'
```

In [54]:

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

Out[54]:

```
1    a
3    b
dtype: object
```

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

In [55]:

```
data.iloc[1]
```

Out[55]:

```
'b'
```

In [56]:

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

Out[56]:

```
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.

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 [57]:

```
area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170
pop = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York': 19651127, 'Florida'
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[57]:

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

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

In [58]:

```
data['area']
```

Out[58]:

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

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

In [59]:

```
data.area
```

Out[59]:

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

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

In [60]:

```
data.area is data['area']
```

Out[60]:

```
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 [61]:

```
data.pop is data['pop']
```

Out[61]:

```
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 [62]:

```
data['density'] = data['pop'] / data['area']
data
```

Out[62]:

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

This shows a preview of the straightforward syntax of element-by-element arithmetic between Series objects.

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 [63]:

```
data.values
```

Out[63]:

```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+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 [64]:

```
data.T
```

Out[64]:

	California	Florida	Illinois	New York	Texas
area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+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 [65]:

```
data.values[0]
```

Out[65]:

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

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

In [66]:

```
data['area']
```

Out[66]:

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

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` 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 [67]:

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

Out[67]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

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 [68]:

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

Out[68]:

	area	pop
California	423967	38332521
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 [69]:

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

Out[69]:

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

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 [70]:

```
data.iloc[0, 2] = 90
data
```

Out[70]:

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

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 [71]:

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

Out[71]:

	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 [72]:

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

Out[72]:

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

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

In [73]:

```
data[data.density > 100]
```

Out[73]:

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

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.

Operating on Data in Pandas

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy.

Review:

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the numpy.ufunc class. Many of the built-in functions are implemented in compiled C code. The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other 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:

In [74]:

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

In [75]:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
```

Out[75]:

```
0    6
1    3
2    7
3    4
dtype: int32
```

In [76]:

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

Out[76]:

	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 [77]:

```
np.exp(ser)
```

Out[77]:

```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

In [78]:

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

Out[78]:

	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*:

In [79]:

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

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

In [80]:

```
population / area
```

Out[80]:

```
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:

In [81]:

```
area.index | population.index
```

Out[81]:

```
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 [82]:

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

Out[82]:

```
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 [83]:

```
A.add(B, fill_value=0)
```

Out[83]:

```
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 [84]:

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

Out[84]:

	A	B
0	1	11
1	5	1

In [85]:

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

Out[85]:

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

In [86]:

```
A + B
```

Out[86]:

	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 [87]:

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

Out[87]:

	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()`

Python Operator	Pandas Method(s)
%	mod()
**	pow()

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 [88]:

```
A = rng.randint(10, size=(3, 4))
A
```

Out[88]:

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

In [89]:

```
A - A[0]
```

Out[89]:

```
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 [90]:

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

Out[90]:

	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:

In [91]:

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

Out[91]:

	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 [92]:

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

Out[92]:

```
Q    3
S    2
Name: 0, dtype: int32
```

In [93]:

```
df - halfrow
```

Out[93]:

	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.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

We will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. We will refer to missing data in general as *null*, *NaN*, or *NA* values.

Missing Data in Pandas

There are a number of schemes that have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point `NaN` value, and the Python `None` object.

`None` : Pythonic missing data

The first sentinel value used by Pandas is `None`, a Python singleton object that is often used for missing data in Python code. Because it is a Python object, `None` cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type `'object'` (i.e., arrays of Python objects):

In [94]:

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

In [95]:

```
vals1 = np.array([1, None, 3, 4])
vals1
```

Out[95]:

```
array([1, None, 3, 4], dtype=object)
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error:

In [96]:

```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-30a3fc8c6726> in <module>()
----> 1 vals1.sum()

D:\Anaconda3\lib\site-packages\numpy\core\_methods.py in _sum(a, axis, dtyp
e, out, keepdims)
     30
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

This reflects the fact that addition between an integer and `None` is undefined.

`NaN`: Missing numerical data

The other missing data representation, `NaN` (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

In [97]:

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

Out[97]:

```
dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that `NaN` is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`:

In [98]:

```
1 + np.nan
```

Out[98]:

```
nan
```

In [99]:

```
0 *  np.nan
```

Out[99]:

```
nan
```

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

In [100]:

```
vals2.sum(), vals2.min(), vals2.max()
```

Out[100]:

```
(nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

In [101]:

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Out[101]:

```
(8.0, 1.0, 4.0)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None in Pandas

`NaN` and `None` both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

In [102]:

```
pd.Series([1, np.nan, 2, None])
```

Out[102]:

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

In [103]:

```
x = pd.Series(range(2), dtype=int)
x
```

Out[103]:

```
0    0
1    1
dtype: int32
```

In [104]:

```
x[0] = None
x
```

Out[104]:

```
0    NaN
1    1.0
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. (Be aware that there is a proposal to add a native integer NA to Pandas in the future; as of this writing, it has not been included).

While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works quite well in practice and in my experience only rarely causes issues.

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()` : Generate a boolean mask indicating missing values
- `notnull()` : Opposite of `isnull()`
- `dropna()` : Return a filtered version of the data
- `fillna()` : Return a copy of the data with missing values filled or imputed

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()` . Either one will return a Boolean mask over the data. For example:

In [105]:

```
data = pd.Series([1, np.nan, 'hello', None])
```

In [106]:

```
data.isnull()
```

Out[106]:

```
0    False
1     True
2    False
3     True
dtype: bool
```

As already mentioned, Boolean masks can be used directly as a `Series` or `DataFrame` index:

In [107]:

```
data[data.notnull()]
```

Out[107]:

```
0        1
2    hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar Boolean results for `DataFrame` s.

Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a `Series` , the result is straightforward:

In [108]:

```
data.dropna()
```

Out[108]:

```
0        1
2    hello
dtype: object
```

For a `DataFrame` , there are more options. Consider the following `DataFrame` :

In [109]:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[109]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

We cannot drop single values from a `DataFrame`; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a `DataFrame`.

By default, `dropna()` will drop all rows in which *any* null value is present:

In [110]:

```
df.dropna()
```

Out[110]:

	0	1	2
1	2.0	3.0	5

Alternatively, you can drop NA values along a different axis; `axis=1` drops all columns containing a null value:

In [111]:

```
df.dropna(axis='columns')
```

Out[111]:

	2
0	2
1	5
2	6

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

In [112]:

```
df[3] = np.nan
df
```

Out[112]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [113]:

```
df.dropna(axis='columns', how='all')
```

Out[113]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

In [114]:

```
df.dropna(axis='rows', thresh=3)
```

Out[114]:

	0	1	2	3
1	2.0	3.0	5	NaN

Here the first and last row have been dropped, because they contain only two non-null values.

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following `Series` :

In [115]:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

Out[115]:

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

We can fill NA entries with a single value, such as zero:

In [116]:

```
data.fillna(0)
```

Out[116]:

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

In [117]:

```
# forward-fill
data.fillna(method='ffill')
```

Out[117]:

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

In [118]:

```
# back-fill
data.fillna(method='bfill')
```

Out[118]:

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

For `DataFrame`s, the options are similar, but we can also specify an `axis` along which the fills take place:

In [119]:

```
df
```

Out[119]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [120]:

```
df.fillna(method='ffill', axis=1)
```

Out[120]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Notice that if a previous value is not available during a forward fill, the NA value remains.

Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. `Series` and `DataFrame`s are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

We will take a look at simple concatenation of `Series` and `DataFrame`s with the `pd.concat` function; later we'll dive into more sophisticated in-memory merges and joins implemented in Pandas.

In [121]:

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

For convenience, we'll define this function which creates a `DataFrame` of a particular form that will be useful below:

In [122]:

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

Out[122]:

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class that allows us to display multiple `DataFrame` s side by side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

In [123]:

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Recall: Concatenation of NumPy Arrays

Concatenation of `Series` and `DataFrame` objects is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function. Recall that with it, you can combine the contents of two or more arrays into a single array:

In [124]:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

Out[124]:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an axis keyword that allows you to specify the axis along which the result will be concatenated:

In [125]:

```
x = [[1, 2],
     [3, 4]]
np.concatenate([x, x], axis=1)
```

Out[125]:

```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Simple Concatenation with `pd.concat`

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

In [126]:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Out[126]:

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

It also works to concatenate higher-dimensional objects, such as `DataFrame`s:

In [127]:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

Out[127]:

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

pd.concat([df1, df2])

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

By default, the concatenation takes place row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

In [128]:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis=1)")
```

Out[128]:

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

pd.concat([df3, df4], axis=1)

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

In [129]:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
```

Out[129]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y])

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

Notice the repeated indices in the result. While this is valid within `DataFrame`s, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the repeats as an error

If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

In [130]:

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: [0, 1]
```

Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting `Series`:

In [131]:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

Out[131]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y], ignore_index=True)

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Adding MultiIndex keys

Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

In [132]:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

Out[132]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y], keys=['x', 'y'])

		A	B
x	**0**	A0	B0
	1	A1	B1
y	**0**	A2	B2
	1	A3	B3

Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating `DataFrame` s with shared column names. In practice, data from different sources might have different sets of column names, and `pd.concat` offers several options in this case. Consider the concatenation of the following two `DataFrame` s, which have some (but not all!) columns in common:

In [133]:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

Out[133]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6])

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the `join` and `join_axes` parameters of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'` :

In [134]:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

Out[134]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], join='inner')

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Another option is to directly specify the index of the remaininig colums using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

In [135]:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
```

Out[135]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], join_axes=[df5.columns])

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data.

The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call `df1.append(df2)`:

In [136]:

```
display('df1', 'df2', 'df1.append(df2)')
```

Out[136]:

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

df1.append(df2)

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the `append()` method in Pandas does not modify the original object–instead it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index *and* data buffer. Thus, if you plan to do

multiple `append` operations, it is generally better to build a list of `DataFrame` s and pass them all at once to the `concat()` function.

Combining Datasets: Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

For convenience, we will start by redefining the `display()` functionality from the previous section:

In [137]:

```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Categories of Joins

The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-one joins

Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in the previous section. As a concrete example, consider the following two `DataFrames` which contain information on several employees in a company:

In [138]:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

Out[138]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

In [139]:

```
df3 = pd.merge(df1, df2)
df3
```

Out[139]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

The `pd.merge()` function recognizes that each `DataFrame` has an "employee" column, and automatically joins using this column as a key. The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the "employee" column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see the `left_index` and `right_index` keywords, discussed momentarily).

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

In [140]:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

Out[140]:

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

The resulting `DataFrame` has an aditional column with the "supervisor" information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a `DataFrame` showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

In [141]:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

Out[141]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

Specification of the Merge Key

We've already seen the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

In [142]:

```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

Out[142]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

This option works only if both the left and right `DataFrame` s have the specified column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

In [143]:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
```

Out[143]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column that we can drop if desired–for example, by using the `drop()` method of `DataFrame`s:

In [144]:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

Out[144]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

In [145]:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

Out[145]:

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

employee	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

In [146]:

```
display('df1a', 'df2a',
        "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

Out[146]:

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

employee	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

employee	group	hire_date
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

For convenience, `DataFrame`s implement the `join()` method, which performs a merge that defaults to joining on indices:

In [147]:

```
display('df1a', 'df2a', 'df1a.join(df2a)')
```

Out[147]:

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

employee	hire_date
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

df1a.join(df2a)

employee	group	hire_date
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

In [148]:

```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
```

Out[148]:

df1a

employee	group
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1a, df3, left_index=True, right_on='name')

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

In [149]:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

Out[149]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7)

	name	food	drink
0	Mary	bread	wine

Here we have merged two datasets that have only a single "name" entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to `"inner"`:

In [150]:

```
pd.merge(df6, df7, how='inner')
```

Out[150]:

	name	food	drink
0	Mary	bread	wine

Other options for the `how` keyword are `'outer'`, `'left'`, and `'right'`. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

In [151]:

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

Out[151]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

In [152]:

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
```

Out[152]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='left')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding join types.

Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input `DataFrame` s have conflicting column names. Consider this example:

In [153]:

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

Out[153]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name")

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

In [154]:

```
display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
```

Out[154]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset. We will explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

For convenience, we'll use the same `display` magic function that we've seen in previous sections:

In [155]:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Planets Data

Here we will use the Planets dataset, available via the Seaborn package (http://seaborn.pydata.org/). It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple Seaborn command:

In [156]:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

Out[156]:

```
(1035, 6)
```

In [157]:

```
planets.head()
```

Out[157]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

This has some details on the 1,000+ extrasolar planets discovered up to 2014.

Simple Aggregation in Pandas

Earlier, we explored some of the data aggregations available for NumPy arrays. As with a one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value:

In [158]:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

Out[158]:

```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

In [159]:

```
ser.sum()
```

Out[159]:

```
2.811925491708157
```

In [160]:

```
ser.mean()
```

Out[160]:

```
0.5623850983416314
```

For a `DataFrame`, by default the aggregates return results within each column:

In [161]:

```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

Out[161]:

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

In [162]:

```
df.mean()
```

Out[162]:

```
A    0.477888
B    0.443420
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate within each row:

In [163]:

```
df.mean(axis='columns')
```

Out[163]:

```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas `Series` and `DataFrame` s include all of the common aggregates mentioned; in addition, there is a convenience method `describe()` that computes several common aggregates for each column and returns the result. Let's use this on the Planets data, for now dropping rows with missing values:

In [164]:

```
planets.dropna().describe()
```

Out[164]:

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

This can be a useful way to begin understanding the overall properties of a dataset. For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all known expolanets were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

The following table summarizes some other built-in Pandas aggregations:

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the `groupby` operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

What the `groupby` accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. Rather, the `GroupBy` can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way. The power of the `GroupBy` is that it abstracts away these steps: the user need not think about *how* the computation is done under the hood, but rather thinks about the *operation as a whole*.

As a concrete example, we'll start by creating the input `DataFrame`:

In [165]:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

Out[165]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The most basic split-apply-combine operation can be computed with the `groupby()` method of `DataFrame`s, passing the name of the desired key column:

In [166]:

```
df.groupby('key')
```

Out[166]:

```
<pandas.core.groupby.DataFrameGroupBy object at 0x0000020DF1E136D8>
```

Notice that what is returned is not a set of `DataFrame`s, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the `DataFrame`, which is poised to dig into the groups but does no actual computation until the aggregation is applied. This "lazy evaluation" approach means that common aggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate apply/combine steps to produce the desired result:

In [167]:

```
df.groupby('key').sum()
```

Out[167]:

	data
key	
A	3
B	5
C	7

The `sum()` method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid `DataFrame` operation, as we will see in the following discussion.

The GroupBy object

The `GroupBy` object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of `DataFrame` s, and it does the difficult things under the hood. Let's see some examples using the Planets data.

Column indexing

The `GroupBy` object supports column indexing in the same way as the `DataFrame` , and returns a modified `GroupBy` object. For example:

In [168]:

```
planets.groupby('method')
```

Out[168]:

```
<pandas.core.groupby.DataFrameGroupBy object at 0x0000020DF1E13A90>
```

In [169]:

```
planets.groupby('method')['orbital_period']
```

Out[169]:

```
<pandas.core.groupby.SeriesGroupBy object at 0x0000020DF1E38B70>
```

Here we've selected a particular `Series` group from the original `DataFrame` group by reference to its column name. As with the `GroupBy` object, no computation is done until we call some aggregate on the object:

In [170]:

```
planets.groupby('method')['orbital_period'].median()
```

Out[170]:

```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that each method is sensitive to.

Iteration over groups

The `GroupBy` object supports direct iteration over the groups, returning each group as a `Series` or `DataFrame` :

In [171]:

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

This can be useful for doing certain things manually, though it is often much faster to use the built-in `apply` functionality, which we will discuss momentarily.

Aggregate, filter, transform, apply

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, `GroupBy` objects have `aggregate()`, `filter()`, `transform()`, and `apply()` methods that efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the following subsections, we'll use this `DataFrame`:

In [172]:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

Out[172]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation

We're now familiar with `GroupBy` aggregations with `sum()`, `median()`, and the like, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once. Here is a quick example combining all these:

In [173]:

```
df.groupby('key').aggregate(['min', np.median, max])
```

Out[173]:

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Another useful pattern is to pass a dictionary mapping column names to operations to be applied on that column:

In [174]:

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

Out[174]:

	data1	data2
key		
A	0	5
B	1	7
C	2	9

Filtering

A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

In [175]:

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()", "df.groupby('key').filter(filter_func)")
```

Out[175]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').std()

	data1	data2
key		
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

df.groupby('key').filter(filter_func)

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

The filter function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

Transformation

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

In [176]:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

Out[176]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

The apply() method

The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object (e.g., `DataFrame`, `Series`) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` that normalizes the first column by the sum of the second:

In [177]:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")
```

Out[177]:

`df`

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

`df.groupby('key').apply(norm_by_data2)`

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

`apply()` within a `GroupBy` is quite flexible: the only criterion is that the function takes a `DataFrame` and returns a Pandas object or scalar; what you do in the middle is up to you!

Specifying the split key

In the simple examples presented before, we split the `DataFrame` on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here.

A list, array, series, or index providing the grouping keys

The key can be any series or list with a length matching that of the `DataFrame`. For example:

In [178]:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
```

Out[178]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(L).sum()

	data1	data2
0	7	17
1	4	3
2	4	7

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from before:

In [179]:

```
display('df', "df.groupby(df['key']).sum()")
```

Out[179]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(df['key']).sum()

	data1	data2
key		
A	3	8
B	5	7
C	7	12

A dictionary or series mapping index to group

Another method is to provide a dictionary that maps index values to the group keys:

In [180]:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

Out[180]:

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(mapping).sum()

	data1	data2
consonant	12	19
vowel	3	8

Any Python function

Similar to mapping, you can pass any Python function that will input the index value and output the group:

In [181]:

```
display('df2', 'df2.groupby(str.lower).mean()')
```

Out[181]:

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(str.lower).mean()

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

A list of valid keys

Further, any of the preceding key choices can be combined to group on a multi-index:

In [182]:

```
df2.groupby([str.lower, mapping]).mean()
```

Out[182]:

		data1	data2
a	**vowel**	1.5	4.0
b	**consonant**	2.5	3.5
c	**consonant**	3.5	6.0

Grouping example

As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

In [183]:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

Out[183]:

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!

In [ ]:

```

```