```
Out[4]: 0     403.428793
        1      20.085537
        2    1096.633158
        3      54.598150
        dtype: float64
```

Or, for a slightly more complex calculation:

```
In[5]: np.sin(df * np.pi / 4)
```

```
Out[5]:           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 "Computation on NumPy Arrays: Universal Functions" on page 50 can be used in a similar manner.

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 you are 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[6]: 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[7]: population / area
```

```
Out[7]: 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 we could determine using standard Python set arithmetic on these indices:

```
In[8]: area.index | population.index
```

```
Out[8]: 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 (see further discussion of missing data in "Handling Missing Data" on page 119). This index matching is imple-

mented this way for any of Python's built-in arithmetic expressions; any missing values are filled in with NaN by default:

```
In[9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])
       B = pd.Series([1, 3, 5], index=[1, 2, 3])
       A + B

Out[9]: 0    NaN
        1    5.0
        2    9.0
        3    NaN
        dtype: float64
```

If using NaN values is not the desired behavior, we can modify the fill value 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[10]: A.add(B, fill_value=0)

Out[10]: 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 you are performing operations on `DataFrame`s:

```
In[11]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                         columns=list('AB'))
        A

Out[11]:    A   B
         0  1  11
         1  5   1

In[12]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                         columns=list('BAC'))
        B

Out[12]:    B  A  C
         0  4  0  9
         1  5  8  0
         2  9  2  6

In[13]: A + B

Out[13]:       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` (which we compute by first stacking the rows of `A`):

```
In[14]: fill = A.stack().mean()
        A.add(B, fill_value=fill)

Out[14]:      A     B     C
          0   1.0  15.0  13.5
          1  13.0   6.0   4.5
          2   6.5  13.5  10.5
```

Table 3-1 lists Python operators and their equivalent Pandas object methods.

Table 3-1. Mapping between Python operators and Pandas methods

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

Ufuncs: Operations Between DataFrame and Series

When you are 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[15]: A = rng.randint(10, size=(3, 4))
        A

Out[15]: array([[3, 8, 2, 4],
                [2, 6, 4, 8],
                [6, 1, 3, 8]])

In[16]: A - A[0]

Out[16]: array([[ 0,  0,  0,  0],
                [-1, -2,  2,  4],
                [ 3, -7,  1,  4]])
```