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

In [21]:

data.area **is** data['area']

Out[21]:

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 [22]:

data.pop **is** data['pop']

Out[22]:

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).

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() |

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 [6]:

1 + np.nan

Out[6]:

nan

In [7]:

0 \* np.nan

Out[7]:

nan

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:

## **Operating on Null Values**

* 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

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:

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

### he Better Way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndex type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

In [5]:

index = pd.MultiIndex.from\_tuples(index)

index

Out[5]:

MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],

labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])

Notice that the MultiIndex contains multiple levels of indexing–in this case, the state names and the years, as well as multiple labels for each data point which encode these levels.

If we re-index our series with this MultiIndex, we see the hierarchical representation of the data:

### MultiIndex as extra dimension

You might notice something else here: we could easily have stored the same data using a simple DataFrame with index and column labels. In fact, Pandas is built with this equivalence in mind. The unstack() method will quickly convert a multiply indexed Series into a conventionally indexed DataFrame: