two-dimensional data within a one-dimensional Series, we can also use it to represent data of three or more dimensions in a Series or DataFrame. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18); with a MultiIndex this is as easy as adding another column to the DataFrame:

```
In[10]: pop_df = pd.DataFrame({'total': pop,
                               'under18': [9267089, 9284094,
                                           4687374, 4318033,
                                           5906301, 6879014]})
        pop_df

Out[10]:                      total  under18
         California 2000  33871648  9267089
                    2010  37253956  9284094
         New York   2000  18976457  4687374
                    2010  19378102  4318033
         Texas      2000  20851820  5906301
                    2010  25145561  6879014
```

In addition, all the ufuncs and other functionality discussed in "Operating on Data in Pandas" on page 115 work with hierarchical indices as well. Here we compute the fraction of people under 18 by year, given the above data:

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
        f_u18.unstack()

Out[11]:                 2000      2010
         California  0.273594  0.249211
         New York    0.247010  0.222831
         Texas       0.283251  0.273568
```

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor. For example:

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                          index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns=['data1', 'data2'])
        df

Out[12]:         data1     data2
         a 1  0.554233  0.356072
           2  0.925244  0.219474
         b 1  0.441759  0.610054
           2  0.171495  0.886688
```

The work of creating the `MultiIndex` is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
In[13]: data = {('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)
```

```
Out[13]: California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Nevertheless, it is sometimes useful to explicitly create a `MultiIndex`; we'll see a couple of these methods here.

Explicit MultiIndex constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, as we did before, you can construct the `MultiIndex` from a simple list of arrays, giving the index values within each level:

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
```

```
Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples, giving the multiple index values of each point:

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

```
Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
In[16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

```
Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Similarly, you can construct the `MultiIndex` directly using its internal encoding by passing `levels` (a list of lists containing available index values for each level) and `labels` (a list of lists that reference these labels):

```
In[17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[17]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can pass any of these objects as the `index` argument when creating a `Series` or `DataFrame`, or to the `reindex` method of an existing `Series` or `DataFrame`.

MultiIndex level names

Sometimes it is convenient to name the levels of the `MultiIndex`. You can accomplish this by passing the `names` argument to any of the above `MultiIndex` constructors, or by setting the `names` attribute of the index after the fact:

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

With more involved datasets, this can be a useful way to keep track of the meaning of various index values.

MultiIndex for columns

In a `DataFrame`, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
In[19]:
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

```
Out[19]: subject      Bob       Guido         Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

Here we see where the multi-indexing for both rows and columns can come in *very* handy. This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full `DataFrame` containing just that person's information:

```
In[20]: health_data['Guido']
```

```
Out[20]: type          HR  Temp
         year visit
         2013 1      32.0  36.7
              2      50.0  35.0
         2014 1      39.0  37.8
              2      48.0  37.3
```

For complicated records containing multiple labeled measurements across multiple times for many subjects (people, countries, cities, etc.), use of hierarchical rows and columns can be extremely convenient!

Indexing and Slicing a MultiIndex

Indexing and slicing on a `MultiIndex` is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed `Series`, and then multiply indexed `DataFrame`s.

Multiply indexed Series

Consider the multiply indexed `Series` of state populations we saw earlier:

```
In[21]: pop
```

```
Out[21]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

We can access single elements by indexing with multiple terms:

```
In[22]: pop['California', 2000]
```

```
Out[22]: 33871648
```