```
In[32]: health_data.loc[(:, 1), (:, 'HR')]
```

```
  File "<ipython-input-32-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

You could get around this by building the desired slice explicitly using Python's built-in `slice()` function, but a better way in this context is to use an `IndexSlice` object, which Pandas provides for precisely this situation. For example:

```
In[33]: idx = pd.IndexSlice
        health_data.loc[idx[:, 1], idx[:, 'HR']]
```

```
Out[33]: subject      Bob Guido   Sue
         type          HR    HR    HR
         year visit
         2013 1      31.0  32.0  35.0
         2014 1      30.0  39.0  61.0
```

There are so many ways to interact with data in multiply indexed `Series` and `DataFrames`, and as with many tools in this book the best way to become familiar with them is to try them out!

Rearranging Multi-Indices

One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

Sorted and unsorted indices

Earlier, we briefly mentioned a caveat, but we should emphasize it more here. *Many of the* `MultiIndex` *slicing operations will fail if the index is not sorted.* Let's take a look at this here.

We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

```
In[34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
        data = pd.Series(np.random.rand(6), index=index)
        data.index.names = ['char', 'int']
        data
```

```
Out[34]: char  int
         a     1      0.003001
               2      0.164974
         c     1      0.741650
```

```
           2      0.569264
      b    1      0.001693
           2      0.526226
      dtype: float64
```

If we try to take a partial slice of this index, it will result in an error:

```
In[35]: try:
            data['a':'b']
        except KeyError as e:
            print(type(e))
            print(e)
```

```
<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Although it is not entirely clear from the error message, this is the result of the `MultiIndex` not being sorted. For various reasons, partial slices and other similar operations require the levels in the `MultiIndex` to be in sorted (i.e., lexographical) order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the `DataFrame`. We'll use the simplest, `sort_index()`, here:

```
In[36]: data = data.sort_index()
        data
```

```
Out[36]: char  int
      a     1      0.003001
            2      0.164974
      b     1      0.001693
            2      0.526226
      c     1      0.741650
            2      0.569264
      dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
In[37]: data['a':'b']
```

```
Out[37]: char  int
      a     1      0.003001
            2      0.164974
      b     1      0.001693
            2      0.526226
      dtype: float64
```

Stacking and unstacking indices

As we saw briefly before, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use:

```
In[38]: pop.unstack(level=0)
```

```
Out[38]:  state  California   New York     Texas
          year
          2000     33871648   18976457  20851820
          2010     37253956   19378102  25145561
```

```
In[39]: pop.unstack(level=1)
```

```
Out[39]: year            2000      2010
         state
         California  33871648  37253956
         New York    18976457  19378102
         Texas       20851820  25145561
```

The opposite of `unstack()` is `stack()`, which here can be used to recover the original series:

```
In[40]: pop.unstack().stack()
```

```
Out[40]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Index setting and resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a `DataFrame` with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
In[41]: pop_flat = pop.reset_index(name='population')
        pop_flat
```

```
Out[41]:         state  year  population
         0  California  2000    33871648
         1  California  2010    37253956
         2    New York  2000    18976457
         3    New York  2010    19378102
         4       Texas  2000    20851820
         5       Texas  2010    25145561
```

Often when you are working with data in the real world, the raw input data looks like this and it's useful to build a `MultiIndex` from the column values. This can be done with the `set_index` method of the `DataFrame`, which returns a multiply indexed `DataFrame`:

```
In[42]: pop_flat.set_index(['state', 'year'])

Out[42]:                  population
        state      year
        California 2000    33871648
                   2010    37253956
        New York   2000    18976457
                   2010    19378102
        Texas      2000    20851820
                   2010    25145561
```

In practice, I find this type of reindexing to be one of the more useful patterns when I encounter real-world datasets.

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, and `max()`. For hierarchically indexed data, these can be passed a `level` parameter that controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
In[43]: health_data

Out[43]:  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
```

Perhaps we'd like to average out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

```
In[44]: data_mean = health_data.mean(level='year')
        data_mean

Out[44]: subject   Bob       Guido          Sue
         type       HR  Temp    HR   Temp    HR   Temp
         year
         2013     37.5  38.2  41.0  35.85  32.0  36.95
         2014     38.5  37.6  43.5  37.55  56.0  36.70
```

By further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

```
In[45]: data_mean.mean(axis=1, level='type')

Out[45]: type         HR       Temp
         year
         2013  36.833333  37.000000
         2014  46.000000  37.283333
```