

## 03.05-Hierarchical-Indexing

June 27, 2018

*This notebook contains an excerpt from the [Python Data Science Handbook](#) by Jake VanderPlas; the content is available [on GitHub](#).*

*The text is released under the [CC-BY-NC-ND license](#), and code is released under the [MIT license](#). If you find this content useful, please consider supporting the work by [buying the book](#)!*

[< Handling Missing Data | Contents | Combining Datasets: Concat and Append >](#)

### 1 Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects, respectively. Often it is useful to go beyond this and store higher-dimensional data—that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects that natively handle three-dimensional and four-dimensional data (see Section 1.6), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

In this section, we'll explore the direct creation of MultiIndex objects, considerations when indexing, slicing, and computing statistics across multiply indexed data, and useful routines for converting between simple and hierarchically indexed representations of your data.

We begin with the standard imports:

```
In [1]: import pandas as pd
import numpy as np
```

#### 1.1 A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, we will consider a series of data where each point has a character and numerical key.

##### 1.1.1 The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
In [2]: index = [('California', 2000), ('California', 2010),
                ('New York', 2000), ('New York', 2010),
```

```

        ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop

```

```

Out[2]: (California, 2000)    33871648
        (California, 2010)    37253956
        (New York, 2000)     18976457
        (New York, 2010)     19378102
        (Texas, 2000)        20851820
        (Texas, 2010)        25145561
        dtype: int64

```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```

In [3]: pop[('California', 2010):('Texas', 2000)]

```

```

Out[3]: (California, 2010)    37253956
        (New York, 2000)     18976457
        (New York, 2010)     19378102
        (Texas, 2000)        20851820
        dtype: int64

```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

```

In [4]: pop[[i for i in pop.index if i[1] == 2010]]

```

```

Out[4]: (California, 2010)    37253956
        (New York, 2010)     19378102
        (Texas, 2010)        25145561
        dtype: int64

```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

### 1.1.2 The 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:

```
In [6]: pop = pop.reindex(index)
        pop

Out[6]: California  2000    33871648
           2010    37253956
           New York  2000    18976457
           2010    19378102
           Texas     2000    20851820
           2010    25145561
           dtype: int64
```

Here the first two columns of the `Series` representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

```
In [7]: pop[:, 2010]

Out[7]: California    37253956
           New York    19378102
           Texas       25145561
           dtype: int64
```

The result is a singly indexed array with just the keys we’re interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We’ll now further discuss this sort of indexing operation on hierarchically indexed data.

### 1.1.3 `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`:

```
In [8]: pop_df = pop.unstack()
        pop_df

Out[8]:
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

Naturally, the `stack()` method provides the opposite operation:

```
In [9]: pop_df.stack()
```

```
Out[9]: California  2000    33871648
          2010    37253956
          New York   2000    18976457
          2010    19378102
          Texas      2000    20851820
          2010    25145561
          dtype: int64
```

Seeing this, you might wonder why would we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to represent 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](#) 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.

## 1.2 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.

### 1.2.1 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]])
```

Any of these objects can be passed as the `index` argument when creating a `Series` or `DataFrame`, or be passed to the `reindex` method of an existing `Series` or `DataFrame`.

### 1.2.2 MultiIndex level names

Sometimes it is convenient to name the levels of the `MultiIndex`. This can be accomplished 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.

### 1.2.3 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!

## 1.3 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 DataFrames.

### 1.3.1 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
```

The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

```
In [23]: pop['California']
```

```
Out[23]: year
2000      33871648
2010      37253956
dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion in Section 1.4.1):

```
In [24]: pop.loc['California':'New York']
```

```
Out[24]: state      year
California 2000      33871648
           2010      37253956
New York   2000      18976457
           2010      19378102
dtype: int64
```

With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

```
In [25]: pop[:, 2000]
```

```
Out[25]: state
California 33871648
New York   18976457
Texas      20851820
dtype: int64
```



Other types of indexing and selection (discussed in [Data Indexing and Selection](#)) work as well; for example, selection based on Boolean masks:

```
In [26]: pop[pop > 22000000]
```

```
Out[26]: state      year
California 2000      33871648
           2010      37253956
Texas      2010      25145561
dtype: int64
```

Selection based on fancy indexing also works:

```
In [27]: pop[['California', 'Texas']]
```

```
Out[27]: state      year
California 2000      33871648
           2010      37253956
Texas      2000      20851820
           2010      25145561
dtype: int64
```

### 1.3.2 Multiply indexed DataFrames

A multiply indexed DataFrame behaves in a similar manner. Consider our toy medical DataFrame from before:

```
In [28]: health_data
```

```
Out[28]: 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
```

Remember that columns are primary in a DataFrame, and the syntax used for multiply indexed Series applies to the columns. For example, we can recover Guido's heart rate data with a simple operation:

```
In [29]: health_data['Guido', 'HR']
```

```
Out[29]: year  visit
2013    1      32.0
         2      50.0
2014    1      39.0
         2      48.0
Name: (Guido, HR), dtype: float64
```

Also, as with the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in [Data Indexing and Selection](#). For example:

```
In [30]: health_data.iloc[:2, :2]
```

```
Out[30]: subject      Bob
         type         HR  Temp
         year visit
         2013 1      31.0  38.7
              2      44.0  37.7
```

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
In [31]: health_data.loc[:, ('Bob', 'HR')]
```

```
Out[31]: year  visit
         2013  1      31.0
              2      44.0
         2014  1      30.0
              2      47.0
         Name: (Bob, HR), dtype: float64
```

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
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!

## 1.4 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.

### 1.4.1 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 lexicographically 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
          a      2      0.164974
          c      1      0.741650
          c      2      0.569264
          b      1      0.001693
          b      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., lexicographical) 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
```

## 1.4.2 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
```

### 1.4.3 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 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 encountering real-world datasets.

## 1.5 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
```

Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. This syntax is actually a short cut to the `GroupBy` functionality, which we will discuss in [Aggregation and Grouping](#). While this is a toy example, many real-world datasets have similar hierarchical structure.

## 1.6 Aside: Panel Data

Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) `Series` and (two-dimensional) `DataFrame` structures. Once you are familiar with indexing and manipulation of data in a `Series` and `DataFrame`, `Panel` and `Panel4D` are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in [Data Indexing and Selection](#) extend readily to these higher-dimensional structures.

We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simpler representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multi-indexing is fundamentally a sparse data representation. As the number of dimensions increases, the dense representation can become very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the `Panel` and `Panel4D` structures, see the references listed in [Further Resources](#).

< [Handling Missing Data](#) | [Contents](#) | [Combining Datasets: Concat and Append](#) >