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.

Early Pandas versions provided `Panel` and `Panel4D` objects that could be thought of as 3D or 4D analogs to the 2D `DataFrame`, but they were somewhat **clunky to use in practice.**

A far more common pattern for handling higher-dimensional data 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 chapter, 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 data.

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

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.**

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 [ ]:
```
index = [('California', 2010), ('California', 2020),
         ('New York', 2010), ('New York', 2020),
         ('Texas', 2010), ('Texas', 2020)]
populations = [37253956, 39538223,
               19378102, 20201249,
               25145561, 29145505]
pop = pd.Series(populations, index=index)
pop
```

```
Out[ ]: (California, 2010)    37253956
        (California, 2020)    39538223
        (New York, 2010)      19378102
        (New York, 2020)      20201249
        (Texas, 2010)         25145561
        (Texas, 2020)         29145505
        dtype: int64
```

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

```
In [ ]: pop[('California', 2020):('Texas', 2010)]
```

```
Out[ ]: (California, 2020)    39538223
        (New York, 2010)      19378102
        (New York, 2020)      20201249
        (Texas, 2010)         25145561
        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 [ ]:
```
pop[[i for i in pop.index if i[1] == 2010]]
```

Out[ ]:
```
(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.

The Better Way: The 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 types of operations we wish to have.

We can create **a multi-index** from the tuples as follows:

In [ ]:
```
index = pd.MultiIndex.from_tuples(index)
```

The `MultiIndex` **represents 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 reindex our series with this `MultiIndex`, we see the **hierarchical representation of the data:**

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

Out[ ]:
```
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
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 2020, we can use the Pandas slicing notation:

In [ ]:
```
pop[:, 2020]
```

Out[ ]:
```
California    39538223
New York      20201249
Texas         29145505
dtype: int64
```

The result is a **singly indexed Series** 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.

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

```python
pop_df
```

Out[ ]:

	2010	2020
California	37253956	39538223
New York	19378102	20201249
Texas	25145561	29145505

Naturally, the `stack` method provides the **opposite operation:**

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

Out[ ]:
```
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
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 manipulate two-dimensional data within a one-dimensional `Series`,

we can **also use it to manipulate 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 [ ]:
```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9284094, 8898092,
                                   4318033, 4181528,
                                   6879014, 7432474]})
pop_df
```

Out[ ]:

		total	under18
California	**2010**	37253956	9284094
	2020	39538223	8898092
New York	**2010**	19378102	4318033
	2020	20201249	4181528
Texas	**2010**	25145561	6879014
	2020	29145505	7432474

In addition, **all the ufuncs and other functionality work** with hierarchical indices as well.

Here we **compute the fraction of people under 18 by year,** given the above data

In [ ]:

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

Out[ ]:

	2010	2020
California	0.249211	0.225050
New York	0.222831	0.206994
Texas	0.273568	0.255013

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

Out[ ]:

		data1	data2
a	**1**	0.748464	0.561409
	2	0.379199	0.622461
b	**1**	0.701679	0.687932
	2	0.436200	0.950664

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 [ ]:

```python
data = {('California', 2010): 37253956,
        ('California', 2020): 39538223,
        ('New York', 2010): 19378102,
        ('New York', 2020): 20201249,
        ('Texas', 2010): 25145561,
        ('Texas', 2020): 29145505}
pd.Series(data)
```

Out[ ]:

```
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Nevertheless, it is **sometimes useful to explicitly create a `MultiIndex`;**

we'll look at a couple of **methods** for doing this next.

Explicit MultiIndex Constructors

For more flexibility in how the index is constructed, you can **instead use the constructor methods** available in the `pd.MultiIndex` class.

For example, as we did before, you can construct a `MultiIndex` from a simple list of arrays giving the index values within each level:

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

```
Out[ ]: MultiIndex([('a', 1),
                    ('a', 2),
                    ('b', 1),
                    ('b', 2)],
                   )
```

Or you can **construct it from a list of tuples** giving the multiple index values of each point:

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

```
Out[ ]: MultiIndex([('a', 1),
                    ('a', 2),
                    ('b', 1),
                    ('b', 2)],
                   )
```

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

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

```
Out[ ]: MultiIndex([('a', 1),
                    ('a', 2),
                    ('b', 1),
                    ('b', 2)],
                   )
```

Similarly, you can construct a `MultiIndex` **directly using its internal encoding by passing**

`levels` (a list of lists containing available index values for each level) and

`codes` (a list of lists that reference these labels):

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

Out[ ]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

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`.

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 previously discussed `MultiIndex` constructors,

or by **setting** the `names` attribute of the index after the fact:

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

Out[ ]:
```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
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 [ ]:
```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue']
                                     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[ ]:

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	30.0	38.0	56.0	38.3	45.0	35.8
	2	47.0	37.1	27.0	36.0	37.0	36.4
2014	1	51.0	35.9	24.0	36.7	32.0	36.2
	2	49.0	36.3	48.0	39.2	31.0	35.7

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 [ ]:
```
health_data['Guido']
```

Out[ ]:

	type	HR	Temp
year	**visit**		
2013	**1**	56.0	38.3
	2	27.0	36.0
2014	**1**	24.0	36.7
	2	48.0	39.2

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` objects.

Multiply Indexed Series

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

In [ ]:
```
pop
```

Out[ ]:
```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

We can access single elements by indexing with multiple terms:

In [ ]:
```python
pop['California', 2010]
```

Out[ ]:
```
37253956
```

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 [ ]:
```python
pop['California']
```

Out[ ]:
```
year
2010    37253956
2020    39538223
dtype: int64
```

Partial slicing is available as well, as long as the `MultiIndex` is sorted.

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

Out[ ]:
```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
dtype: int64
```

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

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

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

Other types of indexing and selection work as well; **for example,** selection based on Boolean masks:

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

```
Out[ ]: state       year
        California  2010    37253956
                    2020    39538223
        Texas       2010    25145561
                    2020    29145505
        dtype: int64
```

Selection based on fancy indexing also works:

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

```
Out[ ]: state       year
        California  2010    37253956
                    2020    39538223
        Texas       2010    25145561
                    2020    29145505
        dtype: int64
```

Multiply Indexed DataFrames

A **multiply indexed** `DataFrame` behaves in a similar manner.

Consider our toy medical `DataFrame` from before:

In [ ]:
```
health_data
```

Out[ ]:

subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	**1**	30.0	38.0	56.0	38.3	45.0	35.8
	2	47.0	37.1	27.0	36.0	37.0	36.4
2014	**1**	51.0	35.9	24.0	36.7	32.0	36.2
	2	49.0	36.3	48.0	39.2	31.0	35.7

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 [ ]:
```python
health_data['Guido', 'HR']
```

Out[ ]:
```
year  visit
2013  1        56.0
      2        27.0
2014  1        24.0
      2        48.0
Name: (Guido, HR), dtype: float64
```

Also, as with the single-index case, we can use the `loc`, `iloc`, and `ix` indexers. **For example:**

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

Out[ ]:

subject		Bob	
	type	HR	Temp
year	visit		
2013	1	30.0	38.0
	2	47.0	37.1

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 [ ]:

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

Out[ ]:

```
year  visit
2013  1        30.0
      2        47.0
2014  1        51.0
      2        49.0
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 [ ]:

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

```
  File "/var/folders/xc/sptt9bk14s34rgxt7453p03r0000gp/T/ipykernel_86488/3311942670.py", 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 [ ]:
```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

Out[ ]:

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	**1**	30.0	56.0	45.0
2014	**1**	51.0	24.0	32.0

As you can see, there are **many ways to interact** with data in multiply indexed `Series` and `DataFrame`s

Rearranging Multi-Indexes

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

Many of the `MultiIndex` **slicing operations will fail if the index is not sorted.**

Let's take a closer look.

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

In [ ]:
```
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[ ]:
```
char  int
a     1      0.280341
      2      0.097290
c     1      0.206217
      2      0.431771
b     1      0.100183
      2      0.015851
dtype: float64
```

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

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

```
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, such as the `sort_index` and `sortlevel` methods of the `DataFrame`.

We'll use the simplest, `sort_index`, here:

In [ ]:

```
data = data.sort_index()
data
```

Out[ ]:

```
char  int
a     1      0.280341
      2      0.097290
b     1      0.100183
      2      0.015851
c     1      0.206217
      2      0.431771
dtype: float64
```

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

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

Out[ ]:
```
char  int
a     1       0.280341
      2       0.097290
b     1       0.100183
      2       0.015851
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 [ ]:
```
pop.unstack(level=0)
```

Out[ ]:

state	California	New York	Texas
year			
2010	37253956	19378102	25145561
2020	39538223	20201249	29145505

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

Out[ ]:

year	2010	2020
state		
California	37253956	39538223
New York	19378102	20201249
Texas	25145561	29145505

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

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

Out[ ]:
```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
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 `state` and `year` columns **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 [ ]:
```
pop_flat = pop.reset_index(name='population')
pop_flat
```

Out[ ]:

	state	year	population
0	California	2010	37253956
1	California	2020	39538223
2	New York	2010	19378102
3	New York	2020	20201249
4	Texas	2010	25145561
5	Texas	2020	29145505

A common pattern is 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 [ ]:

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

Out[ ]:

state	year	population
California	2010	37253956
	2020	39538223
New York	2010	19378102
	2020	20201249
Texas	2010	25145561
	2020	29145505

In practice, this type of reindexing is one of the **more useful patterns** when exploring real-world datasets.