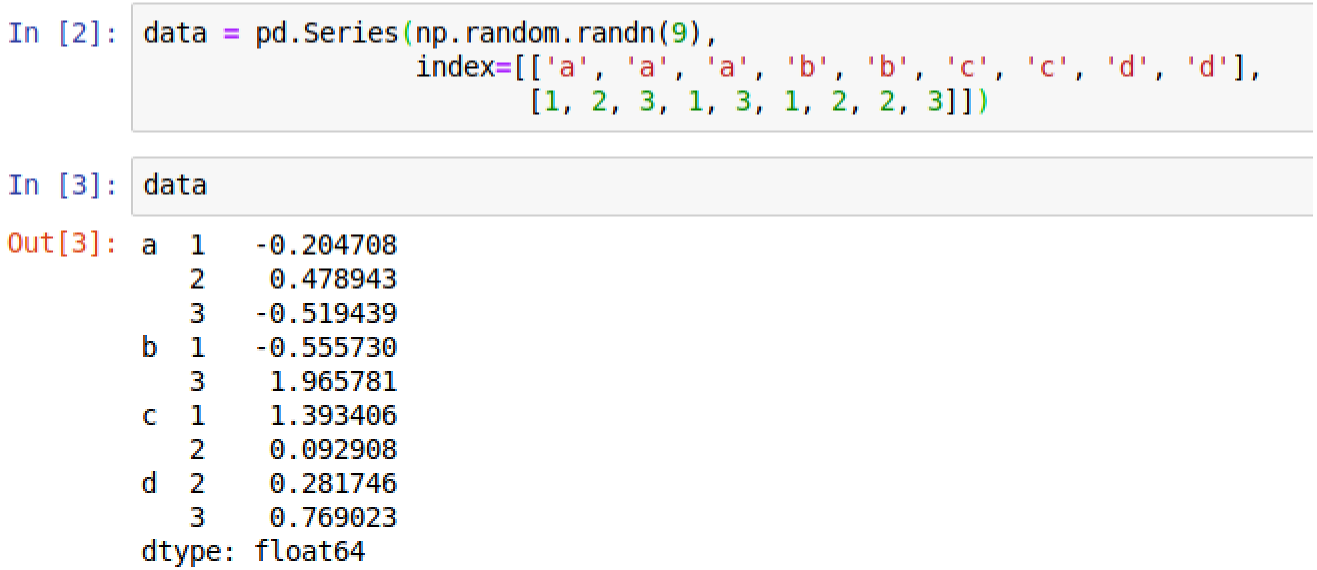
Data Wrangling: Join, Combine, and Reshape

Part 1

Hierarchical Indexing

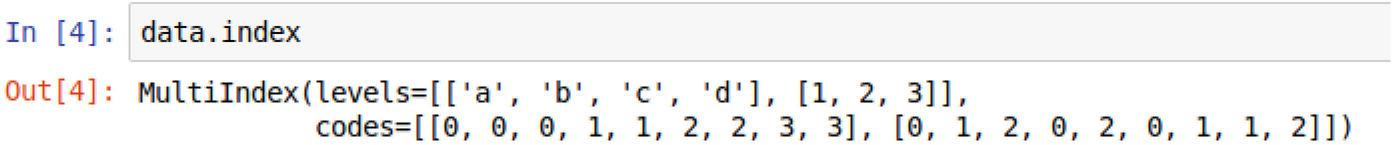
Hierarchical Indexing

- *Hierarchical indexing* is an important feature of pandas that enables you to have multiple (two or more) index *levels* on an axis.



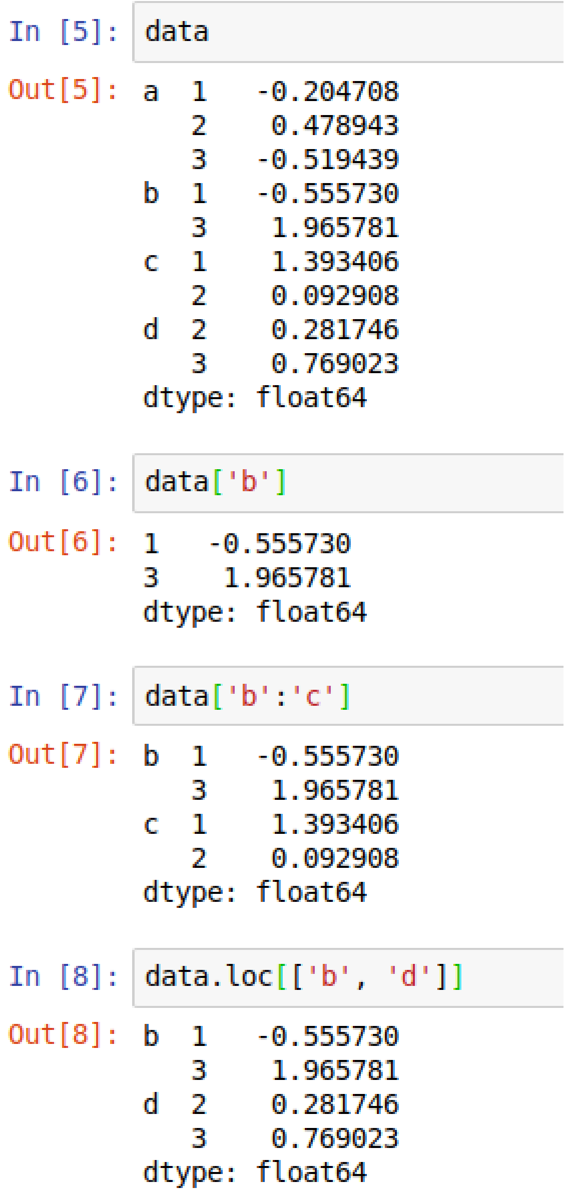
```
In [2]: data = pd.Series(np.random.randn(9),
                         index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
                                [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [3]: data

Out[3]: a  1   -0.204708
           2    0.478943
           3   -0.519439
        b  1   -0.555730
           3    1.965781
        c  1    1.393406
           2    0.092908
        d  2    0.281746
           3    0.769023
        dtype: float64
```

- What you’re seeing is a prettified view of a Series with a MultiIndex as its index.
- The “gaps” in the index display mean “use the label directly above”:



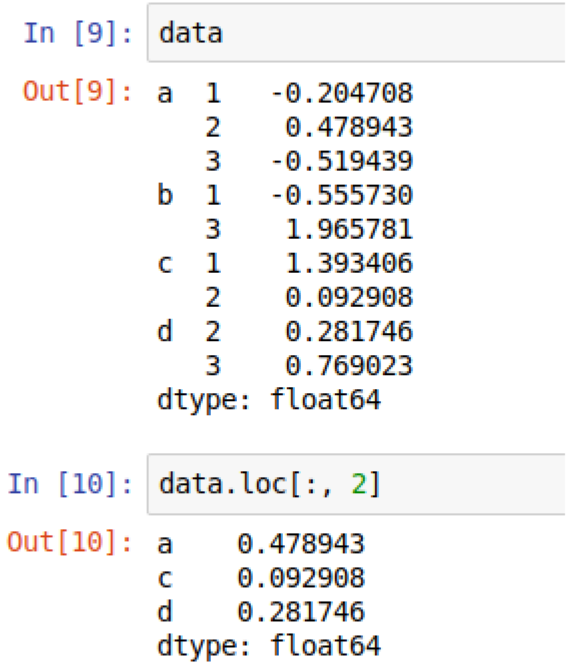
```
In [4]: data.index

Out[4]: MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
                   codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

- With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:



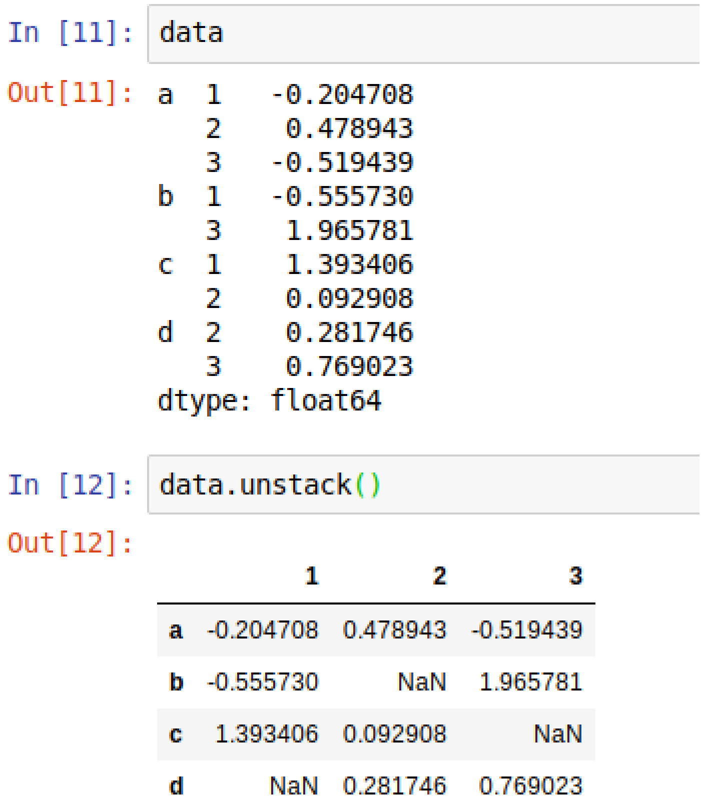
```
In [5]:  data

Out[5]:  a  1   -0.204708
            2    0.478943
            3   -0.519439
         b  1   -0.555730
            3    1.965781
         c  1    1.393406
            2    0.092908
         d  2    0.281746
            3    0.769023
         dtype: float64

In [6]:  data['b']

Out[6]:  1   -0.555730
         3    1.965781
         dtype: float64

In [7]:  data['b':'c']

Out[7]:  b  1   -0.555730
            3    1.965781
         c  1    1.393406
            2    0.092908
         dtype: float64

In [8]:  data.loc[['b', 'd']]

Out[8]:  b  1   -0.555730
            3    1.965781
         d  2    0.281746
            3    0.769023
         dtype: float64
```

- Selection is even possible from an “inner” level:



```
In [9]: data

Out[9]: a  1   -0.204708
           2    0.478943
           3   -0.519439
        b  1   -0.555730
           3    1.965781
        c  1    1.393406
           2    0.092908
        d  2    0.281746
           3    0.769023
        dtype: float64

In [10]: data.loc[:, 2]

Out[10]: a    0.478943
         c    0.092908
         d    0.281746
         dtype: float64
```

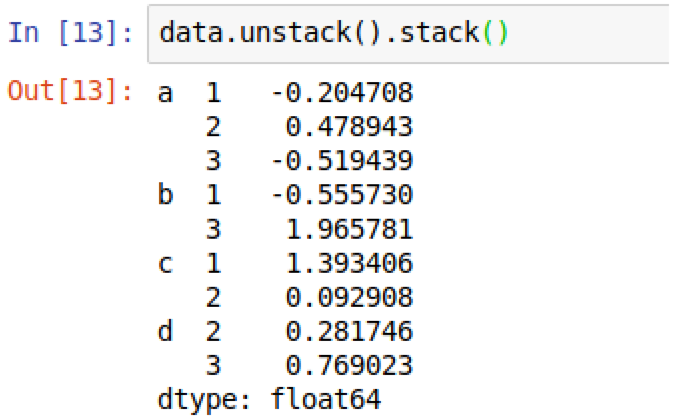
- Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table.
- For example, you could rearrange the data into a DataFrame using its `unstack` method:



```
In [11]: data

Out[11]: a  1   -0.204708
            2    0.478943
            3   -0.519439
         b  1   -0.555730
            3    1.965781
         c  1    1.393406
            2    0.092908
         d  2    0.281746
            3    0.769023
         dtype: float64

In [12]: data.unstack()

Out[12]:
```

	1	2	3
a	-0.204708	0.478943	-0.519439
b	-0.555730	NaN	1.965781
c	1.393406	0.092908	NaN
d	NaN	0.281746	0.769023

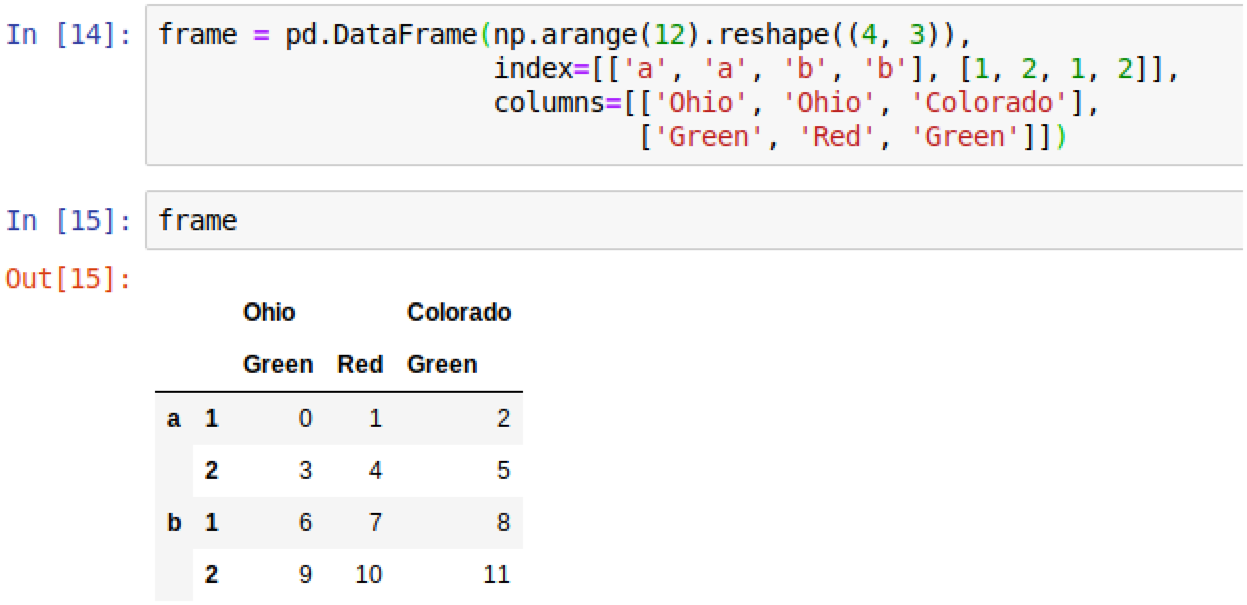
- The inverse operation of `unstack` is `stack`:



```
In [13]: data.unstack().stack()
```

```
Out[13]: a  1   -0.204708
            2    0.478943
            3   -0.519439
         b  1   -0.555730
            3    1.965781
         c  1    1.393406
            2    0.092908
         d  2    0.281746
            3    0.769023
         dtype: float64
```

- With a DataFrame, either axis can have a hierarchical index:



In [14]:

```
frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
                     index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                     columns=[['Ohio', 'Ohio', 'Colorado'],
                              ['Green', 'Red', 'Green']])
```

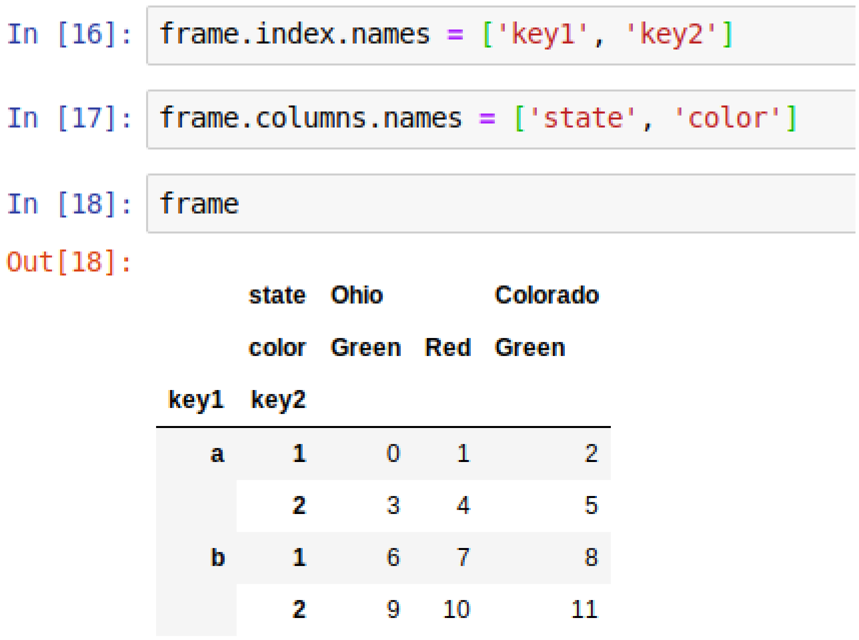
In [15]:

```
frame
```

Out[15]:

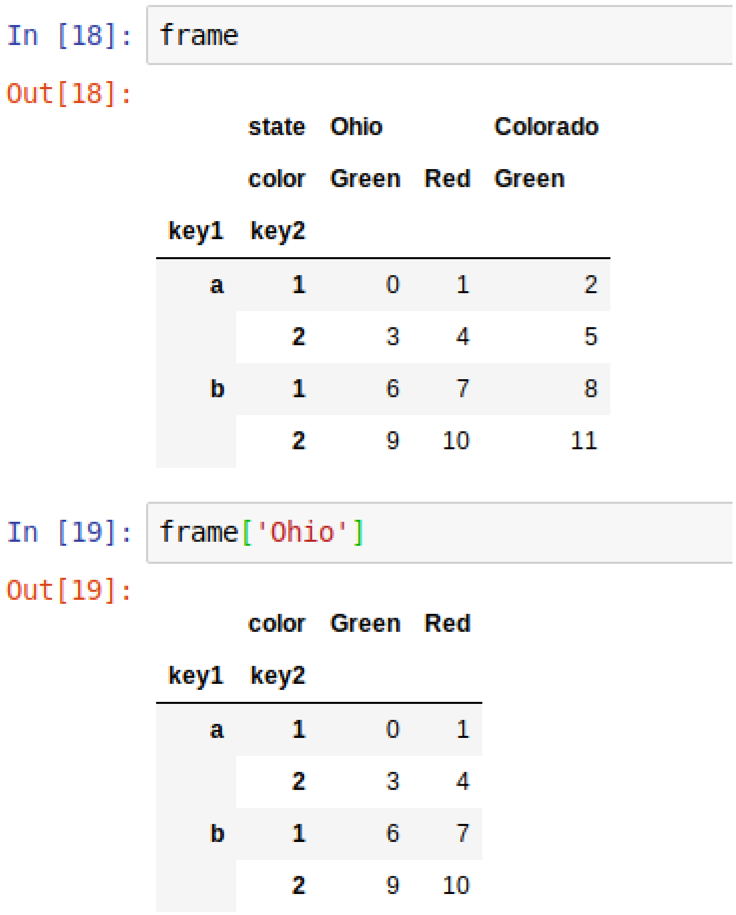
		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

- The hierarchical levels can have names (as strings or any Python objects).
- If so, these will show up in the console output:



```
In [16]: frame.index.names = ['key1', 'key2']

In [17]: frame.columns.names = ['state', 'color']

In [18]: frame

Out[18]:
             state  Ohio        Colorado
             color  Green  Red  Green
       key1  key2
          a     1       0    1         2
                2       3    4         5
          b     1       6    7         8
                2       9   10        11
```

- With partial column indexing you can similarly select groups of columns:



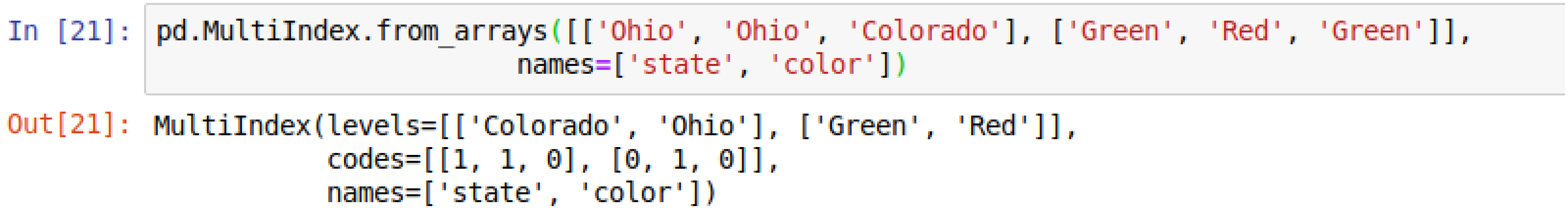
```
In [18]: frame
```

```
Out[18]:
              state  Ohio        Colorado
              color  Green  Red  Green
         key1 key2
         a    1          0    1         2
              2          3    4         5
         b    1          6    7         8
              2          9   10        11
```

```
In [19]: frame['Ohio']
```

```
Out[19]:
              color  Green  Red
         key1 key2
         a    1          0    1
              2          3    4
         b    1          6    7
              2          9   10
```

- A `MultiIndex` can be created by itself and then reused; the columns in the preceding DataFrame with level names could be created like this:

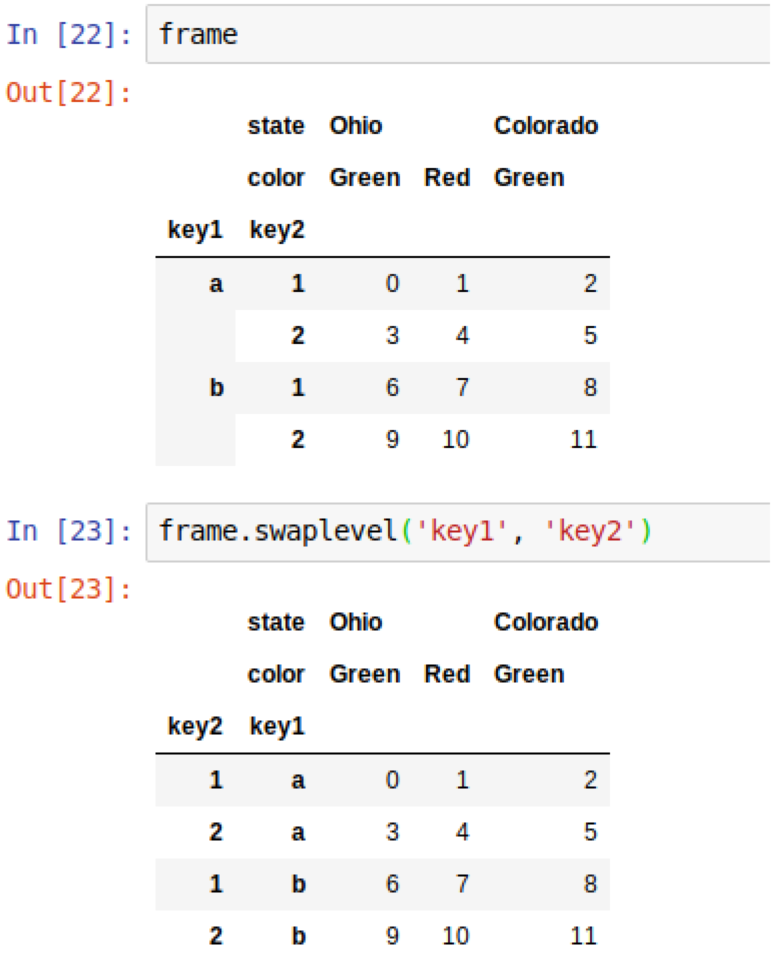


```
In [21]: pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                                   names=['state', 'color'])

Out[21]: MultiIndex(levels=[['Colorado', 'Ohio'], ['Green', 'Red']],
                    codes=[[1, 1, 0], [0, 1, 0]],
                    names=['state', 'color'])
```

Reordering and Sorting Levels

- At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level.

- The `swaplevel` takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):



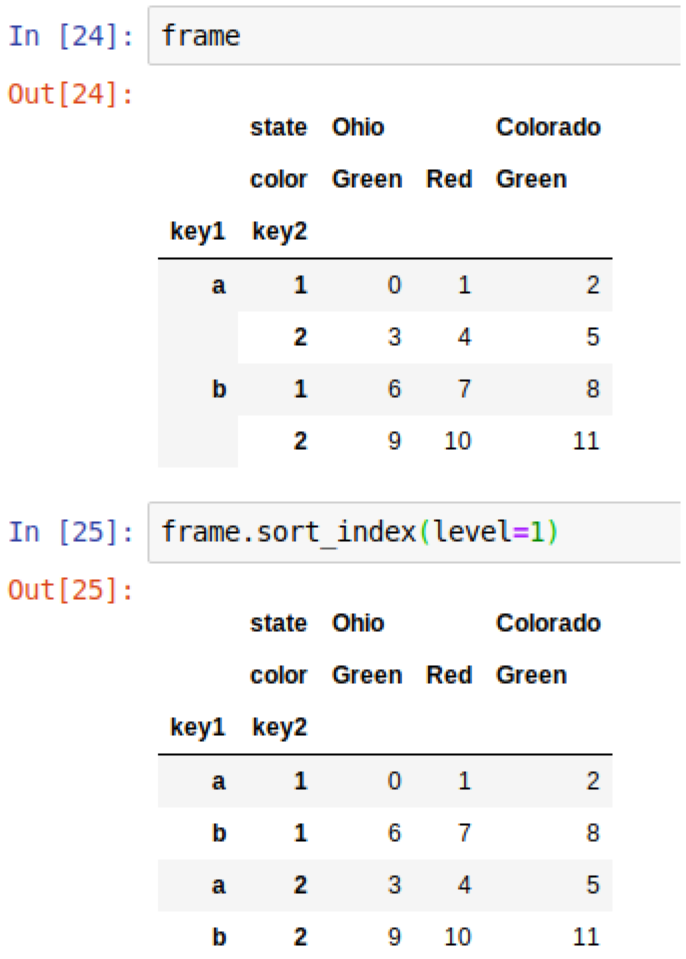
```
In [22]: frame
```

```
Out[22]:
            state  Ohio         Colorado
            color  Green  Red   Green
key1  key2
   a     1             0    1          2
         2             3    4          5
   b     1             6    7          8
         2             9   10         11
```

```
In [23]: frame.swaplevel('key1', 'key2')
```

```
Out[23]:
            state  Ohio         Colorado
            color  Green  Red   Green
key2  key1
   1     a             0    1          2
   2     a             3    4          5
   1     b             6    7          8
   2     b             9   10         11
```

- `sort_index`, on the other hand, sorts the data using only the values in a single level.



In [24]:
```
frame
```

Out[24]:

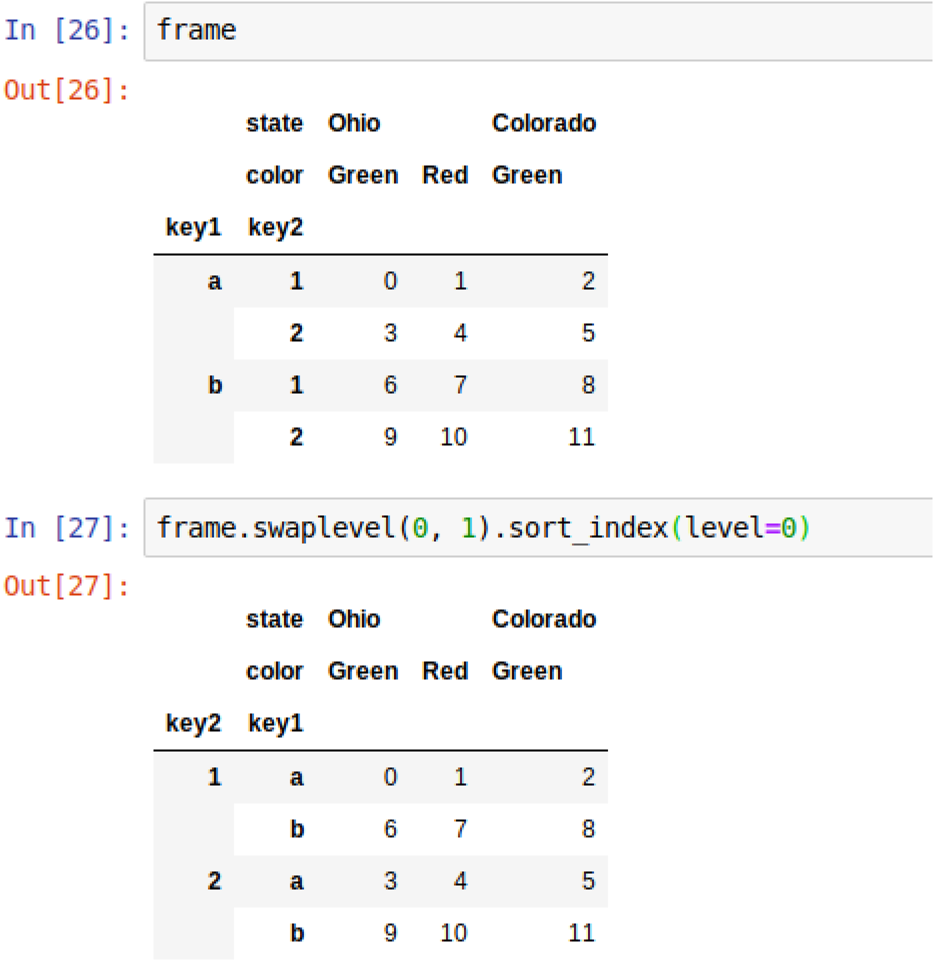
	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [25]:
```
frame.sort_index(level=1)
```

Out[25]:

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

- When swapping levels, it’s not uncommon to also use `sort_index` so that the result is lexicographically sorted by the indicated level:



```
In [26]: frame
```

Out[26]:

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

```
In [27]: frame.swaplevel(0, 1).sort_index(level=0)
```

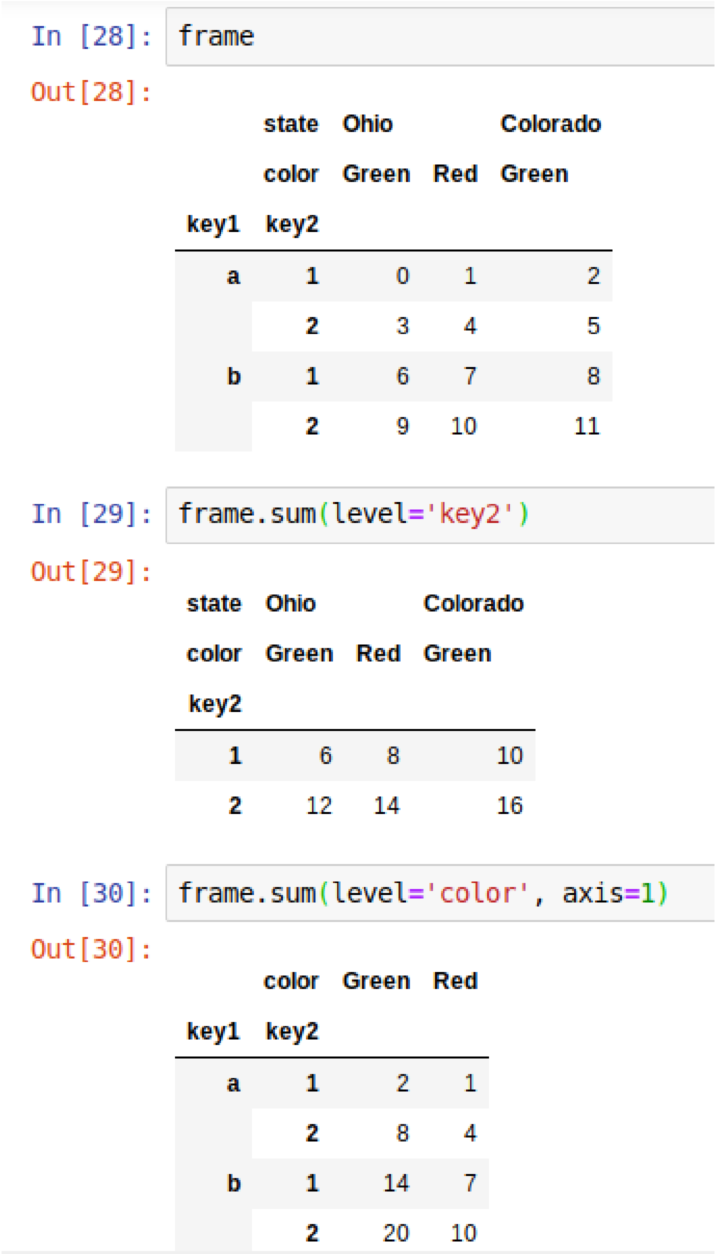
Out[27]:

	state	Ohio		Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

Summary Statistics by Level

- Many descriptive and summary statistics on DataFrame and Series have a `level` option in which you can specify the level you want to aggregate by on a particular axis.

- Consider the above DataFrame; we can aggregate by level on either the rows or columns like so:



In [28]:
```
frame
```

Out[28]:

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [29]:
```
frame.sum(level='key2')
```

Out[29]:

state	Ohio		Colorado
color	Green	Red	Green
key2			
1	6	8	10
2	12	14	16

In [30]:
```
frame.sum(level='color', axis=1)
```

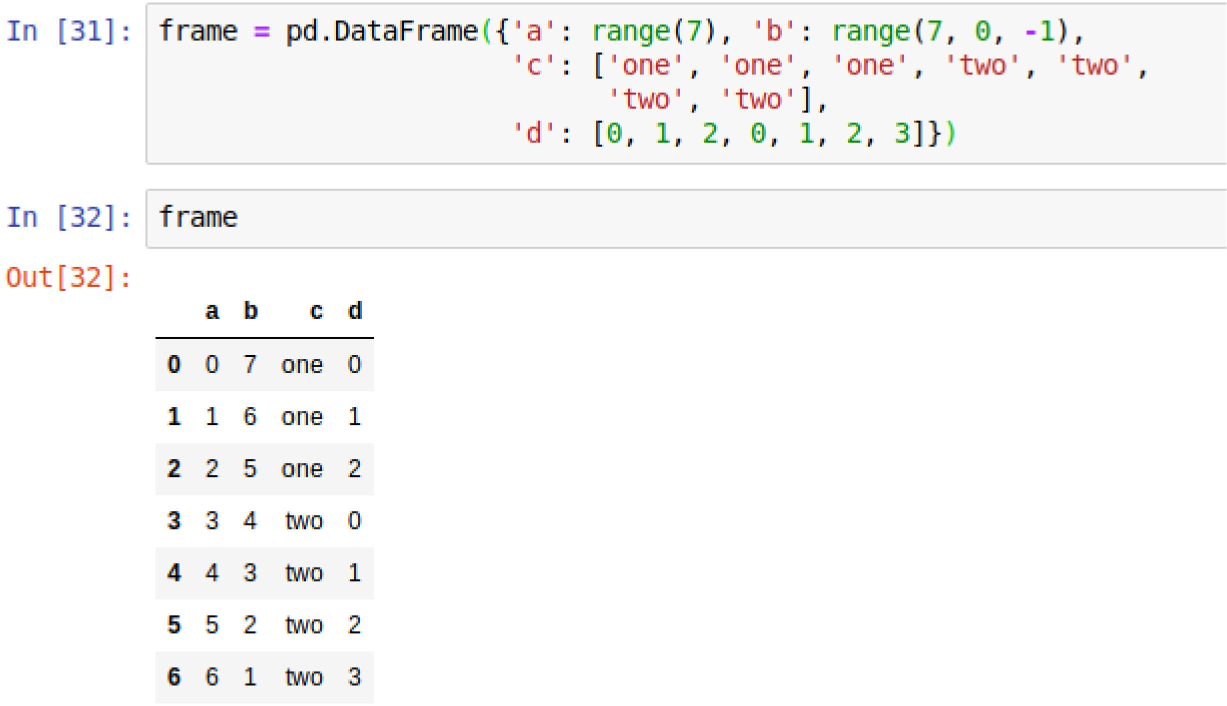
Out[30]:

	color	Green	Red
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

Indexing with a DataFrame’s columns

- It’s not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame’s columns.

- Here’s an example DataFrame:



In [31]:

```
frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
                      'c': ['one', 'one', 'one', 'two', 'two',
                            'two', 'two'],
                      'd': [0, 1, 2, 0, 1, 2, 3]})
```

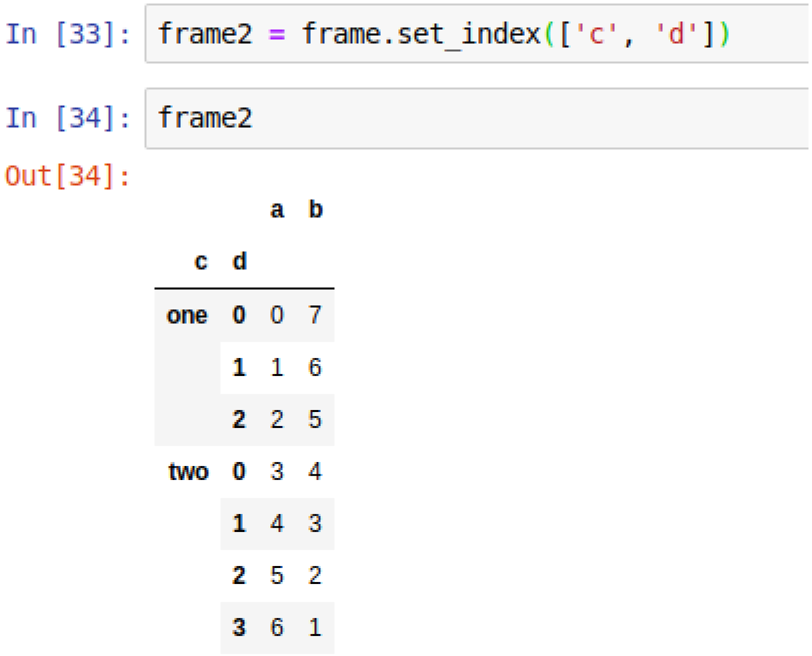
In [32]:

```
frame
```

Out[32]:

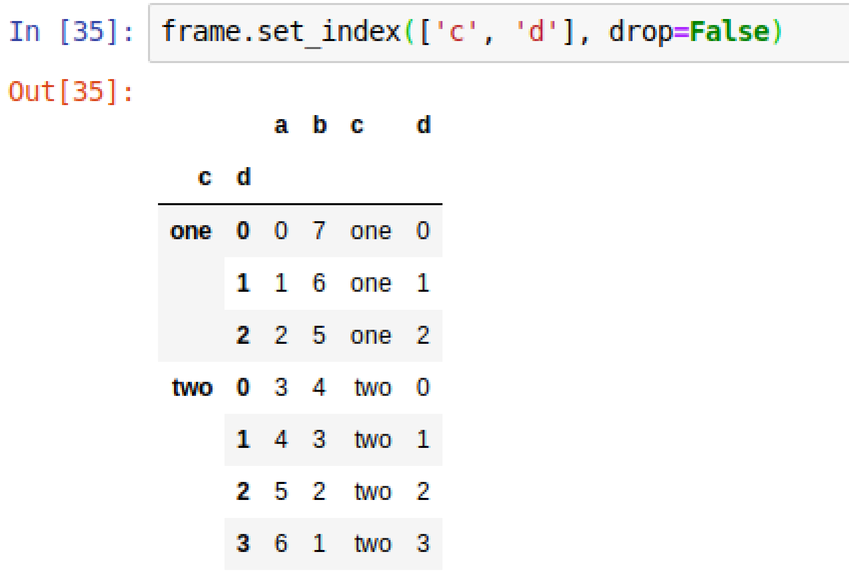
	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

- DataFrame’s `set_index` function will create a new DataFrame using one or more of its columns as the index:



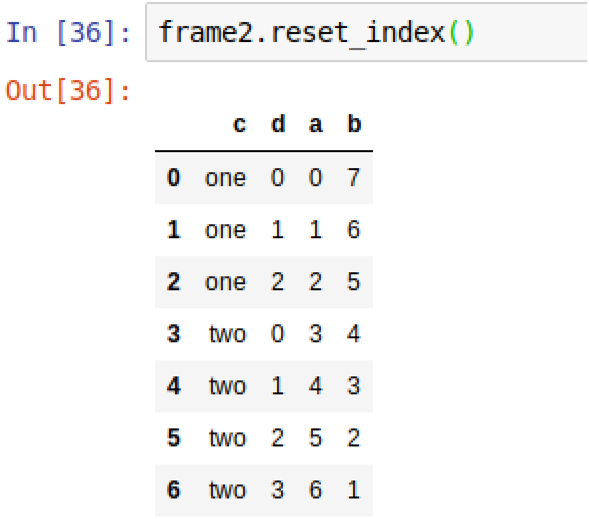
```
In [33]: frame2 = frame.set_index(['c', 'd'])

In [34]: frame2

Out[34]:
             a  b
    c    d
    one  0   0  7
         1   1  6
         2   2  5
    two  0   3  4
         1   4  3
         2   5  2
         3   6  1
```

- By default the columns are removed from the DataFrame, though you can leave them in:



```
In [35]: frame.set_index(['c', 'd'], drop=False)

Out[35]:
                a  b  c    d
        c   d
        one 0   0  7  one  0
            1   1  6  one  1
            2   2  5  one  2
        two 0   3  4  two  0
            1   4  3  two  1
            2   5  2  two  2
            3   6  1  two  3
```

- `reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:



In [36]:

```
frame2.reset_index()
```

Out[36]:

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1