pandas_multiindexing

January 3, 2017

1 Pandas MultiIndexing

This document will go into using dataframes with multi-indexes to accomplish common tasks. There are many ways to do each of the things described in this doc, but the examples given are the best way that I have found to do them.

```
In [5]: import numpy as np
        import pandas as pd
        import itertools
```

1.1 Regular Indexes

Indexes are straightforward, so they will be used as a starting point for understindg multiindexes.

1.1.1 Construct Regular Indexes

```
In [6]: # construct from a list or iterable
        ind_cols = pd.Index(['a','b', 'c'])
        print('List-based Index:', ind_cols)
        ind_rows = pd.Index(range(4))
        print('Range-based Index:', ind_rows)
```

```
List-based Index: Index(['a', 'b', 'c'], dtype='object')
Range-based Index: Int64Index([0, 1, 2, 3], dtype='int64')
```

1.1.2 Use Indexes to Build a DataFrame

Next, we'll build a dataframe that uses these indices for index and column dimensions. These are the two default dimensions, and are sufficient for non-complex tabular data.

```
In [7]: df = pd.DataFrame(index=ind_rows,columns=ind_cols)
        print(df)
```

```
a    b    c
0  NaN  NaN  NaN
1  NaN  NaN  NaN
2  NaN  NaN  NaN
3  NaN  NaN  NaN
```

```
In [8]: # and this time with numbers
        df = pd.DataFrame(np.random.uniform(0,1,size=(4,3)),index=ind_rows,columns=ind_cols)
        print(df)
```

```
a         b         c
0  0.905112  0.665538  0.817139
1  0.088187  0.332614  0.058465
2  0.267938  0.621476  0.967733
3  0.885783  0.258796  0.858933
```

1.1.3 Column Indexing

Column selection can be done using df[col_name] and row selection using a logical index like df[logical_index] where logical_index is the result of an operation like df[col_name] > 4.0.

```
In [9]: df[’b’]
```

```
Out[9]: 0    0.665538
        1    0.332614
        2    0.621476
        3    0.258796
        Name: b, dtype: float64
```

1.1.4 Logical Indexing

Logical comparisons can be done to index into a dataframe. Compound logic can be done with the numpy functions np.logical_not, np.logical_and, np.logical_or, etc. Note that this method of indexing is quite slow because it must compare every element and then use that to index the dataframe.

```
In [10]: df[df[’a’] > 0.2]
```

```
Out[10]:          a         b         c
         0  0.905112  0.665538  0.817139
         2  0.267938  0.621476  0.967733
         3  0.885783  0.258796  0.858933
```

```
In [11]: df[np.logical_and(df[’a’] > 0.1, df[’b’] > 0.1)]
```

```
Out[11]:          a         b         c
         0  0.905112  0.665538  0.817139
         2  0.267938  0.621476  0.967733
         3  0.885783  0.258796  0.858933
```

1.1.5 .loc[] Indexing

The best or most common way of indexing uses the .loc method of a dataframe, but it can also be the most challenging. Dimensions (like index and columns) are always separated by commas in the loc[] brackets.

```
In [12]: df.loc[1,’a’]
```

```
Out[12]: 0.088187275308875712
```

Slicing can also used on each dimension.

```
In [13]: df.loc[1:,’a’:’b’]
```

```
Out[13]:          a         b
         1  0.088187  0.332614
         2  0.267938  0.621476
         3  0.885783  0.258796
```

When accessing dimensions that use Simple Indexes, providing a tuple will allow one to explicitly select multiple columns. Be careful, because this has a different meaning with multi-indexes.

```
In [14]: df.loc[1:,(’a’,’c’)]
```

```
Out[14]:          a         c
         1  0.088187  0.058465
         2  0.267938  0.967733
         3  0.885783  0.858933
```

1.2 The MultiIndex

There are a couple different ways to construct and use a multi-index. First thing to note is that a pandas MultiIndex object can be used instead of a regular Index on any dimension (like index, columns). First, there are a few ways to construct them.

1.2.1 Construct a MultiIndex

```
In [15]: nodes = range(4)
         attr = (’in’, ’out’)
         indlist = list(itertools.product(nodes,attr))
         mi = pd.MultiIndex.from_tuples(indlist, names=[’number’,’direction’])
         print(mi)
```

```
MultiIndex(levels=[[0, 1, 2, 3], [’in’, ’out’]],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=[’number’, ’direction’])
```

or, equivalently:

```
In [16]: nodes = range(4)
         attr = (’in’, ’out’)
         mi = pd.MultiIndex.from_product([nodes,attr], names=[’number’,’direction’])
         print(mi)
```

```
MultiIndex(levels=[[0, 1, 2, 3], [’in’, ’out’]],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=[’number’, ’direction’])
```

A MultiIndex itself is an efficient structure that is a little hard to view with a simple print. One of the easiest ways to observe it might be in list form. You’ll have to imagine the heararchy part this way. It’s also convenient to see the heirarchy later in the DataFrame view.

```
In [17]: print(list(mi))
```

```
[(0, ’in’), (0, ’out’), (1, ’in’), (1, ’out’), (2, ’in’), (2, ’out’), (3, ’in’), (3, ’out’)]
```

You can access these items directly to get all possible labels at each level.

```
In [18]: mi.levels[1]
```

```
Out[18]: Index([’in’, ’out’], dtype=’object’, name=’direction’)
```

And you can also use the method “get_level_values” to acess different pseudo-columns of the index.

```
In [19]: print(mi.get_level_values(1))
         print(mi.get_level_values(’direction’))
```

```
Index([’in’, ’out’, ’in’, ’out’, ’in’, ’out’, ’in’, ’out’], dtype=’object’, name=’direction’)
Index([’in’, ’out’, ’in’, ’out’, ’in’, ’out’, ’in’, ’out’], dtype=’object’, name=’direction’)
```

You can also reorder the index levels if needed.

```
In [20]: mi = mi.reorder_levels((1,0))
         print(mi)
         mi = mi.reorder_levels((1,0))
         print(mi)
```

```
MultiIndex(levels=[[’in’, ’out’], [0, 1, 2, 3]],
           labels=[[0, 1, 0, 1, 0, 1, 0, 1], [0, 0, 1, 1, 2, 2, 3, 3]],
           names=[’direction’, ’number’])
MultiIndex(levels=[[0, 1, 2, 3], [’in’, ’out’]],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=[’number’, ’direction’])
```

1.2.2 Construct a DataFrame Using a MultiIndex

Now the MultiIndex will be used to construct a dataframe. Without data values it will look like this:

```
In [21]: cols = ['attr_a','attr_b','attr_c']
         df = pd.DataFrame(index=mi,columns=cols)
         print(df)
```

```
attr_a attr_b attr_c
number direction
0      in             NaN    NaN    NaN
       out            NaN    NaN    NaN
1      in             NaN    NaN    NaN
       out            NaN    NaN    NaN
2      in             NaN    NaN    NaN
       out            NaN    NaN    NaN
3      in             NaN    NaN    NaN
       out            NaN    NaN    NaN
```

and with data values, like this:

```
In [22]: cols = ['attr_a','attr_b','attr_c']
         dat = np.random.uniform(0,1,(len(mi),len(cols)))
         df = pd.DataFrame(dat,index=mi,columns=cols)
         print(df)
```

```
attr_a    attr_b    attr_c
number direction
0      in         0.598200  0.968099  0.541835
       out        0.624987  0.032465  0.525873
1      in         0.694152  0.981712  0.069959
       out        0.501086  0.900349  0.317123
2      in         0.911546  0.733099  0.912633
       out        0.438137  0.922308  0.043530
3      in         0.040312  0.247064  0.932242
       out        0.467241  0.908447  0.461814
```

Although in this case it has already been done, in some cases you may need to sort the axis dimensions to perform partial indexing.

```
In [23]: df.sort_index(axis='index',inplace=True)
```

1.2.3 .loc[] Indexing With MultiIndex

Just as with simple index dataframes, .loc[] for dataframes with multiindices should separate dimensions by commas. In dimensions that use the MultiIndex, provide a complete tuple to get or assign a specific value.

```
In [24]: df.loc[(1,'in'),'attr_a']
```

```
Out[24]: 0.69415240507530573
```

And ignore all but the first index level by providng a scalar.

```
In [25]: df.loc[1,'attr_a']
```

```
Out[25]: direction
         in     0.694152
         out    0.501086
         Name: attr_a, dtype: float64
```

In this example, you can consider the tuple (0,‘in’) to be a single element of the MultiIndex. Because of that, you can slice values like the following.

```
In [26]: df.loc[(0,’in’):(1,’out’),’attr_a’]
```

```
Out[26]: number  direction
         0       in           0.598200
                 out          0.624987
         1       in           0.694152
                 out          0.501086
         Name: attr_a, dtype: float64
```

1.2.4 .loc[] With Partial Index

You can also provide a partial index leveling in order.

```
In [27]: df.loc[(0,),’attr_a’]
```

```
Out[27]: direction
         in     0.598200
         out    0.624987
         Name: attr_a, dtype: float64
```

Also include slices in tuples. Because it is in the tuple you need to use the ‘slice’ function instead of the ‘:’ operator.

```
In [28]: df.loc[(slice(0,1),),’attr_a’]
```

```
Out[28]: number  direction
         0       in           0.598200
                 out          0.624987
         1       in           0.694152
                 out          0.501086
         Name: attr_a, dtype: float64
```

Or use another tuple to get specific values at a specific level.

```
In [29]: df.loc[((0,3),),’attr_a’]
```

```
Out[29]: number  direction
         0       in           0.598200
                 out          0.624987
         3       in           0.040312
                 out          0.467241
         Name: attr_a, dtype: float64
```

Incomplete indexing can be made more robust by including slicing in specific levels according to the lexical sorting. Note that ‘slice(None)’ can be used instead of the ‘all’ slice operator ‘:’.

```
In [30]: df.loc[(slice(None),’out’),’attr_a’]
```

```
Out[30]: number  direction
         0       out          0.624987
         1       out          0.501086
         2       out          0.438137
         3       out          0.467241
         Name: attr_a, dtype: float64
```

1.3 Looping Through DataFrame With MultiIndex Dimension

Looping is one of the most simple things you may want to do with your MultiIndex and DataFrame. You can always get your multiindex back out of the dataframe directly. For example we will switch to a thre-dimensional indexing scheme.

```
In [49]: cols = ['skill', 'experience']
         mi = pd.MultiIndex.from_product([range(3),('in','out'),('a','b')])
         data = np.random.uniform(0,1,size=(12,len(cols)))
         df = pd.DataFrame(data,index=mi,columns=cols)
         df = df[df['skill'] > 0.5]
         print(df)
```

```
skill  experience
0 in   a  0.959982    0.748411
       b  0.800790    0.368096
  out  a  0.837333    0.838389
       b  0.849932    0.503325
1 out  a  0.538083    0.615077
2 in   a  0.906411    0.328551
       b  0.924629    0.034630
  out  a  0.703278    0.578899
       b  0.773594    0.030768
```

```
In [71]: def mdf(mi,match):
             ''' Returns the list of children of the ordered match
             set given by match.'''
             matchfilt = filter(lambda x: x[:len(match)] == match,mi)
             return set([x[len(match)] for x in matchfilt])

         for i in mdf(df.index,()):
             print(df.loc[(i,slice(None),slice(None))])
             for j in mdf(df.index,(i,)):
                 print(df.loc[(i,j,slice(None))])
                 for k in mdf(df.index,(i,j)):
                     print(df.loc[(i,j,k)])
```

```
skill  experience
0 in   a  0.959982    0.748411
       b  0.800790    0.368096
  out  a  0.837333    0.838389
       b  0.849932    0.503325
            skill  experience
0 out  a  0.837333    0.838389
       b  0.849932    0.503325
skill         0.837333
experience    0.838389
Name: (0, out, a), dtype: float64
skill         0.849932
experience    0.503325
Name: (0, out, b), dtype: float64
           skill  experience
0 in a  0.959982    0.748411
     b  0.800790    0.368096
skill         0.959982
experience    0.748411
```

```
Name: (0, in, a), dtype: float64
skill         0.800790
experience    0.368096
Name: (0, in, b), dtype: float64
            skill  experience
1 out a  0.538083    0.615077
            skill  experience
1 out a  0.538083    0.615077
skill         0.538083
experience    0.615077
Name: (1, out, a), dtype: float64
            skill  experience
2 in  a  0.906411    0.328551
      b  0.924629    0.034630
  out a  0.703278    0.578899
      b  0.773594    0.030768
            skill  experience
2 out a  0.703278    0.578899
      b  0.773594    0.030768
skill         0.703278
experience    0.578899
Name: (2, out, a), dtype: float64
skill         0.773594
experience    0.030768
Name: (2, out, b), dtype: float64
           skill  experience
2 in a  0.906411    0.328551
     b  0.924629    0.034630
skill         0.906411
experience    0.328551
Name: (2, in, a), dtype: float64
skill         0.924629
experience    0.034630
Name: (2, in, b), dtype: float64
```