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 [2]: 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 [3]: # 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 [4]: 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 [5]: # 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  0.171932  0.200883  0.541344
1  0.682860  0.660650  0.136091
2  0.789951  0.014180  0.541751
3  0.479402  0.677582  0.529187
```

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 [6]: df['b']
```

```
Out[6]: 0    0.200883
        1    0.660650
        2    0.014180
        3    0.677582
        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 [7]: df[df['a'] > 0.2]
```

```
Out[7]:           a         b         c
        1  0.682860  0.660650  0.136091
        2  0.789951  0.014180  0.541751
        3  0.479402  0.677582  0.529187
```

```
In [8]: df[np.logical_and(df['a'] > 0.1, df['b'] > 0.1)]
```

```
Out[8]:           a         b         c
        0  0.171932  0.200883  0.541344
        1  0.682860  0.660650  0.136091
        3  0.479402  0.677582  0.529187
```

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 [9]: df.loc[1,'a']
```

```
Out[9]: 0.68285950843423981
```

Slicing can also used on each dimension.

```
In [10]: df.loc[1:,'a':'b']
```

```
Out[10]:           a         b
         1  0.682860  0.660650
         2  0.789951  0.014180
         3  0.479402  0.677582
```

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 [11]: df.loc[1:,('a','c')]
```

```
Out[11]:           a         c
         1  0.682860  0.136091
         2  0.789951  0.541751
         3  0.479402  0.529187
```

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 [12]: 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 [13]: 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 hearachy part this way. It's also convenient to see the heirarchy later in the DataFrame view.

```
In [14]: 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 [15]: mi.levels[1]
```

```
Out[15]: 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 [16]: 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 [17]: 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 [18]: 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 [19]: 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         0.729400  0.817141  0.269680
       out        0.441074  0.535002  0.618083
1      in         0.158032  0.746562  0.820763
       out        0.342587  0.236849  0.871743
2      in         0.087112  0.874758  0.029679
       out        0.877390  0.399852  0.593797
3      in         0.561688  0.929694  0.723619
       out        0.102683  0.927524  0.145657
```

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 [20]: 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 [21]: df.loc[(1,'in'),'attr_a']
```

```
Out[21]: 0.15803238425049915
```

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

```
In [22]: df.loc[1,'attr_a']
```

```
Out[22]: direction
         in     0.158032
         out    0.342587
         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 [23]: df.loc[(0,'in'):(1,'out'),'attr_a']
```

```
Out[23]: number  direction
         0       in           0.729400
                 out          0.441074
         1       in           0.158032
                 out          0.342587
         Name: attr_a, dtype: float64
```

1.2.4 .loc[] With Partial Index

You can also provide a partial index leveling in order.

```
In [24]: df.loc[(0,),'attr_a']
```

```
Out[24]: direction
         in     0.729400
         out    0.441074
         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 [25]: df.loc[(slice(0,1),),'attr_a']
```

```
Out[25]: number  direction
         0       in           0.729400
                 out          0.441074
         1       in           0.158032
                 out          0.342587
         Name: attr_a, dtype: float64
```

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

```
In [26]: df.loc[((0,3),),'attr_a']
```

```
Out[26]: number  direction
         0       in           0.729400
                 out          0.441074
         3       in           0.561688
                 out          0.102683
         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 [27]: df.loc[(slice(None),'out'),'attr_a']
```

```
Out[27]: number  direction
         0       out          0.441074
         1       out          0.342587
         2       out          0.877390
         3       out          0.102683
         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 thredimensional indexing scheme.

```
In [28]: 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)
         print(df)
```

```
skill  experience
0 in  a  0.345953    0.527715
      b  0.336080    0.866092
  out a  0.395736    0.780968
      b  0.798062    0.843355
1 in  a  0.483975    0.812950
      b  0.147582    0.136542
  out a  0.771971    0.352005
      b  0.849107    0.037425
2 in  a  0.224733    0.692692
      b  0.907861    0.606785
  out a  0.554473    0.302768
      b  0.683281    0.182192
```

```
In [31]: for i in df.index.levels[0]:
             print(df.loc[(i,slice(None),slice(None))])
             for j in df.index.levels[1]:
                 print(df.loc[(i,j,slice(None))])
                 for k in df.index.levels[2]:
                     print('%d,%s,%s'%(i,j,k))
                     print(df.loc[(i,j,k)])
                     print('\r\n')
```

```
skill  experience
0 in  a  0.345953    0.527715
      b  0.336080    0.866092
  out a  0.395736    0.780968
      b  0.798062    0.843355
          skill  experience
0 in a  0.345953    0.527715
     b  0.336080    0.866092
0,in,a
skill         0.345953
experience    0.527715
Name: (0, in, a), dtype: float64


0,in,b
skill         0.336080
experience    0.866092
Name: (0, in, b), dtype: float64


            skill  experience
```

```
0 out a  0.395736    0.780968
      b  0.798062    0.843355
0,out,a
skill         0.395736
experience    0.780968
Name: (0, out, a), dtype: float64


0,out,b
skill         0.798062
experience    0.843355
Name: (0, out, b), dtype: float64


            skill  experience
1 in  a  0.483975    0.812950
      b  0.147582    0.136542
  out a  0.771971    0.352005
      b  0.849107    0.037425
           skill  experience
1 in a  0.483975    0.812950
     b  0.147582    0.136542
1,in,a
skill         0.483975
experience    0.812950
Name: (1, in, a), dtype: float64


1,in,b
skill         0.147582
experience    0.136542
Name: (1, in, b), dtype: float64


            skill  experience
1 out a  0.771971    0.352005
      b  0.849107    0.037425
1,out,a
skill         0.771971
experience    0.352005
Name: (1, out, a), dtype: float64


1,out,b
skill         0.849107
experience    0.037425
Name: (1, out, b), dtype: float64


            skill  experience
2 in  a  0.224733    0.692692
      b  0.907861    0.606785
  out a  0.554473    0.302768
      b  0.683281    0.182192
```

```
           skill  experience
2 in a  0.224733    0.692692
     b  0.907861    0.606785
2,in,a
skill         0.224733
experience    0.692692
Name: (2, in, a), dtype: float64


2,in,b
skill         0.907861
experience    0.606785
Name: (2, in, b), dtype: float64


            skill  experience
2 out a  0.554473    0.302768
      b  0.683281    0.182192
2,out,a
skill         0.554473
experience    0.302768
Name: (2, out, a), dtype: float64


2,out,b
skill         0.683281
experience    0.182192
Name: (2, out, b), dtype: float64
```

1.3.1 Build Tree From MultiIndex

```
In [30]: def get_mitree(mi):
             return mitree_r(mi,0)

         def get_mitree_r(mi,lev):
             if len(mi) == 1:
                 return mi[0]
             else:
                 children = list()
                 numlev = len(mi[0]) # total number of levels
                 for val in set(map(lambda x: x[lev],mi)):
                     mislice = list(filter(lambda x: x[lev]==val, mi))
                     key = mislice[0][:lev] + ((slice(None),)*(numlev-lev))
                     val = get_mitree_r(mislice,lev+1)
                     children.append(val)
                 return children


         mitree = get_mitree(mi)
         #import pprint; pprint.pprint(mitree)
         for num in mitree:
             for loc in num:
                 print(loc)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-30-78d61070e45f> in <module>()
 16
 17
---> 18 mitree = get_mitree(mi)
 19 #import pprint; pprint.pprint(mitree)
 20 for num in mitree:


<ipython-input-30-78d61070e45f> in get_mitree(mi)
  1 def get_mitree(mi):
----> 2     return mitree_r(mi,0)
  3
  4 def get_mitree_r(mi,lev):
  5     if len(mi) == 1:


NameError: name 'mitree_r' is not defined
```