MultiIndex / Advanced Indexing

This section covers indexing with a `MultiIndex` and more advanced indexing features.

See the Indexing and Selecting Data for general indexing documentation.

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See Returning a View versus Copy.

See the cookbook for some advanced strategies.

Hierarchical indexing (MultiIndex)

Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like Series (1d) and DataFrame (2d).

In this section, we will show what exactly we mean by "hierarchical" indexing and how it integrates with all of the pandas indexing functionality described above and in prior sections. Later, when discussing group by and pivoting and reshaping data, we'll show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the cookbook for some advanced strategies.

Creating a MultiIndex (hierarchical index) object

The `MultiIndex` object is the hierarchical analogue of the standard `Index` object which typically stores the axis labels in pandas objects. You can think of `MultiIndex` as an array of tuples where each tuple is unique. A `MultiIndex` can be created from a list of arrays (using `MultiIndex.from_arrays`), an array of tuples (using `MultiIndex.from_tuples`), or a crossed set of iterables (using `MultiIndex.from_product`). The `Index` constructor will attempt to return a `MultiIndex` when it is passed a list of tuples. The following examples demonstrate different ways to initialize MultiIndexes.

```
In [1]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ...:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ...:

In [2]: tuples = list(zip(*arrays))

In [3]: tuples
Out[3]:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
```

Scroll To Top

```
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [5]: index
Out[5]: 
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [6]: s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]: 
first  second
bar    one       0.469112
       two      -0.282863
baz    one      -1.509059
       two      -1.135632
foo    one       1.212112
       two      -0.173215
qux    one       0.119209
       two      -1.044236
dtype: float64
```

When you want every pairing of the elements in two iterables, it can be easier to use the `MultiIndex.from_product` function:

```
In [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])
Out[9]: 
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])
```

As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

```
In [10]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
   ....:           np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
   ....: 

In [11]: s = pd.Series(np.random.randn(8), index=arrays)

In [12]: s
Out[12]: 
bar  one   -0.861849
     two   -2.104569
baz  one   -0.494929
     two    1.071804
foo  one    0.721555
     two   -0.706771
qux  one   -1.039575
     two    0.271860
dtype: float64
```

Scroll To Top

```
In [13]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)

In [14]: df
Out[14]:
                0         1         2         3
bar one -0.424972  0.567020  0.276232 -1.087401
    two -0.673690  0.113648 -1.478427  0.524988
baz one  0.404705  0.577046 -1.715002 -1.039268
    two -0.370647 -1.157892 -1.344312  0.844885
foo one  1.075770 -0.109050  1.643563 -1.469388
    two  0.357021 -0.674600 -1.776904 -0.968914
qux one -1.294524  0.413738  0.276662 -0.472035
    two -0.013960 -0.362543 -0.006154 -0.923061
```

All of the `MultiIndex` constructors accept a `names` argument which stores string names for the levels themselves. If no names are provided, `None` will be assigned:

```
In [15]: df.index.names
Out[15]: FrozenList([None, None])
```

This index can back any axis of a pandas object, and the number of **levels** of the index is up to you:

```
In [16]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'], columns=index)

In [17]: df
Out[17]:
first        bar                 baz                 foo                 qux          
second       one       two       one       two       one       two       one       two
A       0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309 -1.170299 -0.226169
B       0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737
C      -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747

In [18]: pd.DataFrame(np.random.randn(6, 6), index=index[:6], columns=index[:6])
Out[18]:
first              bar                 baz                 foo          
second             one       two       one       two       one       two
first second                                                            
bar   one    -0.410001 -0.078638  0.545952 -1.219217 -1.226825  0.769804
      two    -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734
baz   one     0.959726 -1.110336 -0.619976  0.149748 -0.732339  0.687738
      two     0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849
foo   one    -0.954208  1.462696 -1.743161 -0.826591 -0.345352  1.314232
      two     0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441
```

We've "sparsified" the higher levels of the indexes to make the console output a bit easier on the eyes. Note that how the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_options()`:

```
In [19]: with pd.option_context('display.multi_sparse', False):
   ....:     df
   ....:
```

Scroll To Top

It's worth keeping in mind that there's nothing preventing you from using tuples as atomic labels on an axis:

```
In [20]: pd.Series(np.random.randn(8), index=tuples)
Out[20]:
(bar, one)   -1.236269
(bar, two)    0.896171
(baz, one)   -0.487602
(baz, two)   -0.082240
(foo, one)   -2.182937
(foo, two)    0.380396
(qux, one)    0.084844
(qux, two)    0.432390
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

```
In [21]: index.get_level_values(0)
Out[21]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], dtype='object'

In [22]: index.get_level_values('second')
Out[22]: Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'], dtype='object'
```

Basic indexing on axis with MultiIndex

One of the important features of hierarchical indexing is that you can select data by a “partial” label identifying a subgroup in the data. **Partial** selection “drops” levels of the hierarchical index in the result in a completely analogous way to selecting a column in a regular DataFrame:

```
In [23]: df['bar']
Out[23]:
second       one       two
A       0.895717  0.805244
B       0.410835  0.813850
C      -1.413681  1.607920

In [24]: df['bar', 'one']
Out[24]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64

In [25]: df['bar']['one']
Out[25]:
A    0.895717
B    0.410835
```

Scroll To Top

```
C   -1.413681
Name: one, dtype: float64

In [26]: s['qux']
Out[26]:
one   -1.039575
two    0.271860
dtype: float64
```

See Cross-section with hierarchical index for how to select on a deeper level.

Defined Levels

The repr of a `MultiIndex` shows all the defined levels of an index, even if the they are not actually used. When slicing an index, you may notice this. For example:

```
In [27]: df.columns  # original MultiIndex
Out[27]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [28]: df[['foo','qux']].columns  # sliced
Out[28]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           labels=[[2, 2, 3, 3], [0, 1, 0, 1]],
           names=['first', 'second'])
```

This is done to avoid a recomputation of the levels in order to make slicing highly performant. If you want to see only the used levels, you can use the **`MultiIndex.get_level_values()`** method.

```
In [29]: df[['foo','qux']].columns.values
Out[29]: array([('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')], dtype=

# for a specific level
In [30]: df[['foo','qux']].columns.get_level_values(0)
Out[30]: Index(['foo', 'foo', 'qux', 'qux'], dtype='object', name='first')
```

To reconstruct the `MultiIndex` with only the used levels, the `remove_unused_levels` method may be used.

New in version 0.20.0.

```
In [31]: df[['foo','qux']].columns.remove_unused_levels()
Out[31]:
MultiIndex(levels=[['foo', 'qux'], ['one', 'two']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['first', 'second'])
```

Scroll To Top

Data alignment and using `reindex`

Operations between differently-indexed objects having `MultiIndex` on the axes will work as you expect; data alignment will work the same as an Index of tuples:

```
In [32]: s + s[:-2]
Out[32]:
bar  one   -1.723698
     two   -4.209138
baz  one   -0.989859
     two    2.143608
foo  one    1.443110
     two   -1.413542
qux  one         NaN
     two         NaN
dtype: float64

In [33]: s + s[::2]
Out[33]:
bar  one   -1.723698
     two         NaN
baz  one   -0.989859
     two         NaN
foo  one    1.443110
     two         NaN
qux  one   -2.079150
     two         NaN
dtype: float64
```

`reindex` can be called with another `MultiIndex`, or even a list or array of tuples:

```
In [34]: s.reindex(index[:3])
Out[34]:
first  second
bar    one      -0.861849
       two      -2.104569
baz    one      -0.494929
dtype: float64

In [35]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
Out[35]:
foo  two   -0.706771
bar  one   -0.861849
qux  one   -1.039575
baz  one   -0.494929
dtype: float64
```

Advanced indexing with hierarchical index

Syntactically integrating `MultiIndex` in advanced indexing with `.loc` is a bit challenging, but we've made every effort to do so. In general, MultiIndex keys take the form of tuples. For example, the following works as you would expect:

Scroll To Top

```
In [36]: df = df.T

In [37]: df
Out[37]:
```

```
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
      two     0.805244  0.813850  1.607920
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466
      two    -0.226169 -1.436737 -2.006747

In [38]: df.loc[('bar', 'two'),]
Out[38]:
A    0.805244
B    0.813850
C    1.607920
Name: (bar, two), dtype: float64
```

Note that `df.loc['bar', 'two']` would also work in this example, but this shorthand notation can lead to ambiguity in general.

If you also want to index a specific column with `.loc`, you must use a tuple like this:

```
In [39]: df.loc[('bar', 'two'), 'A']
Out[39]: 0.80524402538637851
```

You don't have to specify all levels of the `MultiIndex` by passing only the first elements of the tuple. For example, you can use "partial" indexing to get all elements with `bar` in the first level as follows:

df.loc['bar']

This is a shortcut for the slightly more verbose notation `df.loc[('bar',),]` (equivalent to `df.loc['bar',]` in this example).

"Partial" slicing also works quite nicely.

```
In [40]: df.loc['baz':'foo']
Out[40]:
                     A         B         C
first second
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
```

You can slice with a 'range' of values, by providing a slice of tuples.

```
In [41]: df.loc[('baz', 'two'):('qux', 'one')]
Out[41]:
                     A         B         C
first second
baz   two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466
```

Scroll To Top

```
In [42]: df.loc[('baz', 'two'):'foo']
Out[42]:
                     A         B         C
first second
baz   two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
```

Passing a list of labels or tuples works similar to reindexing:

```
In [43]: df.loc[[('bar', 'two'), ('qux', 'one')]]
Out[43]:
                     A         B         C
first second
bar   two     0.805244  0.813850  1.607920
qux   one    -1.170299  1.130127  0.974466
```

Note: It is important to note that tuples and lists are not treated identically in pandas when it comes to indexing. Whereas a tuple is interpreted as one multi-level key, a list is used to specify several keys. Or in other words, tuples go horizontally (traversing levels), lists go vertically (scanning levels).

Importantly, a list of tuples indexes several complete `MultiIndex` keys, whereas a tuple of lists refer to several values within a level:

```
In [44]: s = pd.Series([1, 2, 3, 4, 5, 6],
   ....:               index=pd.MultiIndex.from_product([["A", "B"], ["c", "d", "e"]]))
   ....:

In [45]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
Out[45]:
A  c    1
B  d    5
dtype: int64

In [46]: s.loc[(["A", "B"], ["c", "d"])]  # tuple of lists
Out[46]:
A  c    1
   d    2
B  c    4
   d    5
dtype: int64
```

Using slicers

You can slice a `MultiIndex` by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see Selection by Label, including slices, lists of labels, labels, and boolean indexers.

Scroll To Top

You can use `slice(None)` to select all the contents of that level. You do not need to specify all the deeper levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

Warning: You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing both axes, rather than into say the `MultiIndex` for the rows.

You should do this:

```
df.loc[(slice('A1','A3'),.....), :]
```

You should **not** do this:

```
df.loc[(slice('A1','A3'),.....)]
```

```
In [47]: def mklbl(prefix,n):
   ....:     return ["%s%s" % (prefix,i)  for i in range(n)]
   ....:

In [48]: miindex = pd.MultiIndex.from_product([mklbl('A',4),
   ....:                                       mklbl('B',2),
   ....:                                       mklbl('C',4),
   ....:                                       mklbl('D',2)])
   ....:

In [49]: micolumns = pd.MultiIndex.from_tuples([('a','foo'),('a','bar'),
   ....:                                        ('b','foo'),('b','bah')],
   ....:                                       names=['lvl0', 'lvl1'])
   ....:

In [50]: dfmi = pd.DataFrame(np.arange(len(miindex)*len(micolumns)).reshape((len(miinde
   ....:                     index=miindex,
   ....:                     columns=micolumns).sort_index().sort_index(axis=1)
   ....:

In [51]: dfmi
Out[51]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0    9    8   11   10
         D1   13   12   15   14
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0   25   24   27   26
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  233  232  235  234
         D1  237  236  239  238
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  249  248  251  250
         D1  253  252  255  254

[64 rows x 4 columns]
```

Scroll To Top

Basic multi-index slicing using slices, lists, and labels.

```
In [52]: dfmi.loc[(slice('A1','A3'), slice(None), ['C1', 'C3']), :]
Out[52]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A1 B0 C1 D0   73   72   75   74
         D1   77   76   79   78
      C3 D0   89   88   91   90
         D1   93   92   95   94
   B1 C1 D0  105  104  107  106
         D1  109  108  111  110
      C3 D0  121  120  123  122
...          ...  ...  ...  ...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
      C3 D0  249  248  251  250
         D1  253  252  255  254

[24 rows x 4 columns]
```

You can use **`pandas.IndexSlice`** to facilitate a more natural syntax using `:`, rather than using `slice(None)`.

```
In [53]: idx = pd.IndexSlice

In [54]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[54]:
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
...          ...  ...
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[32 rows x 2 columns]
```

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```
In [55]: dfmi.loc['A1', (slice(None), 'foo')]
Out[55]:
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
```

Scroll To Top

```
        D1   76   78
     C2 D0   80   82
        D1   84   86
     C3 D0   88   90
...         ...  ...
B1  C0 D1  100  102
    C1 D0  104  106
       D1  108  110
    C2 D0  112  114
       D1  116  118
    C3 D0  120  122
       D1  124  126

[16 rows x 2 columns]

In [56]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[56]:
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
...          ...  ...
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[32 rows x 2 columns]
```

Using a boolean indexer you can provide selection related to the values.

```
In [57]: mask = dfmi[('a', 'foo')] > 200

In [58]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[58]:
lvl0           a    b
lvl1         foo  foo
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```
In [59]: dfmi.loc(axis=0)[:, :, ['C1', 'C3']]
Out[59]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
```

Scroll To Top

```
          D1   13   12   15   14
       C3 D0   25   24   27   26
          D1   29   28   31   30
    B1 C1 D0   41   40   43   42
          D1   45   44   47   46
       C3 D0   57   56   59   58
...           ...  ...  ...  ...
A3 B0 C1 D1   205  204  207  206
       C3 D0  217  216  219  218
          D1  221  220  223  222
    B1 C1 D0  233  232  235  234
          D1  237  236  239  238
       C3 D0  249  248  251  250
          D1  253  252  255  254

[32 rows x 4 columns]
```

Furthermore you can set the values using the following methods.

```
In [60]: df2 = dfmi.copy()

In [61]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

In [62]: df2
Out[62]: 
lvl0           a         b     
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0  -10  -10  -10  -10
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10

[64 rows x 4 columns]
```

You can use a right-hand-side of an alignable object as well.

```
In [63]: df2 = dfmi.copy()

In [64]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [65]: df2
Out[65]: 
lvl0              a               b       
lvl1            bar    foo      bah    foo
A0 B0 C0 D0       1      0        3      2
         D1       5      4        7      6
      C1 D0    9000   8000    11000  10000
         D1   13000  12000    15000  14000
      C2 D0      17     16       19     18
```

Scroll To Top

```
            D1      21      20      23      22
         C3 D0   25000   24000   27000   26000
...             ...     ...     ...     ...
A3 B1 C0 D1     229     228     231     230
         C1 D0  233000  232000  235000  234000
            D1  237000  236000  239000  238000
         C2 D0     241     240     243     242
            D1     245     244     247     246
         C3 D0  249000  248000  251000  250000
            D1  253000  252000  255000  254000

[64 rows x 4 columns]
```

Cross-section

The `xs` method of `DataFrame` additionally takes a level argument to make selecting data at a particular level of a MultiIndex easier.

```
In [66]: df
Out[66]:
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
      two     0.805244  0.813850  1.607920
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466
      two    -0.226169 -1.436737 -2.006747

In [67]: df.xs('one', level='second')
Out[67]:
              A         B         C
first
bar    0.895717  0.410835 -1.413681
baz   -1.206412  0.132003  1.024180
foo    1.431256 -0.076467  0.875906
qux   -1.170299  1.130127  0.974466
```

```
# using the slicers
In [68]: df.loc[(slice(None),'one'),:]
Out[68]:
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
baz   one    -1.206412  0.132003  1.024180
foo   one     1.431256 -0.076467  0.875906
qux   one    -1.170299  1.130127  0.974466
```

You can also select on the columns with `xs()`, by providing the axis argument.

```
In [69]: df = df.T

In [70]: df.xs('one', level='second', axis=1)
Out[70]:
```

Scroll To Top

```
first        bar       baz       foo       qux
A       0.895717 -1.206412  1.431256 -1.170299
B       0.410835  0.132003 -0.076467  1.130127
C      -1.413681  1.024180  0.875906  0.974466
```

```
# using the slicers
In [71]: df.loc[:,(slice(None),'one')]
Out[71]:
first        bar       baz       foo       qux
second       one       one       one       one
A       0.895717 -1.206412  1.431256 -1.170299
B       0.410835  0.132003 -0.076467  1.130127
C      -1.413681  1.024180  0.875906  0.974466
```

`xs()` also allows selection with multiple keys.

```
In [72]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out[72]:
first        bar
second       one
A       0.895717
B       0.410835
C      -1.413681
```

```
# using the slicers
In [73]: df.loc[:,('bar','one')]
Out[73]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64
```

You can pass `drop_level=False` to `xs()` to retain the level that was selected.

```
In [74]: df.xs('one', level='second', axis=1, drop_level=False)
Out[74]:
first        bar       baz       foo       qux
second       one       one       one       one
A       0.895717 -1.206412  1.431256 -1.170299
B       0.410835  0.132003 -0.076467  1.130127
C      -1.413681  1.024180  0.875906  0.974466
```

Compare the above with the result using `drop_level=True` (the default value).

```
In [75]: df.xs('one', level='second', axis=1, drop_level=True)
Out[75]:
first        bar       baz       foo       qux
A       0.895717 -1.206412  1.431256 -1.170299
B       0.410835  0.132003 -0.076467  1.130127
C      -1.413681  1.024180  0.875906  0.974466
```

Scroll To Top

Advanced reindexing and alignment

The parameter `level` has been added to the `reindex` and `align` methods of pandas objects. This is useful to broadcast values across a level. For instance:

```
In [76]: midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']],
   ....:                      labels=[[1,1,0,0],[1,0,1,0]])
   ....:

In [77]: df = pd.DataFrame(np.random.randn(4,2), index=midx)

In [78]: df
Out[78]:
               0         1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [79]: df2 = df.mean(level=0)

In [80]: df2
Out[80]:
             0         1
one   1.060074 -0.109716
zero  1.271532  0.713416

In [81]: df2.reindex(df.index, level=0)
Out[81]:
               0         1
one  y  1.060074 -0.109716
     x  1.060074 -0.109716
zero y  1.271532  0.713416
     x  1.271532  0.713416

# aligning
In [82]: df_aligned, df2_aligned = df.align(df2, level=0)

In [83]: df_aligned
Out[83]:
               0         1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [84]: df2_aligned
Out[84]:
               0         1
one  y  1.060074 -0.109716
     x  1.060074 -0.109716
zero y  1.271532  0.713416
     x  1.271532  0.713416
```

Swapping levels with `swaplevel()`

Scroll To Top

The `swaplevel` function can switch the order of two levels:

```
In [85]: df[:5]
Out[85]:
               0         1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [86]: df[:5].swaplevel(0, 1, axis=0)
Out[86]:
               0         1
y one   1.519970 -0.493662
x one   0.600178  0.274230
y zero  0.132885 -0.023688
x zero  2.410179  1.450520
```

Reordering levels with `reorder_levels()`

The `reorder_levels` function generalizes the `swaplevel` function, allowing you to permute the hierarchical index levels in one step:

```
In [87]: df[:5].reorder_levels([1,0], axis=0)
Out[87]:
               0         1
y one   1.519970 -0.493662
x one   0.600178  0.274230
y zero  0.132885 -0.023688
x zero  2.410179  1.450520
```

Sorting a `MultiIndex`

For MultiIndex-ed objects to be indexed and sliced effectively, they need to be sorted. As with any index, you can use `sort_index`.

```
In [88]: import random; random.shuffle(tuples)

In [89]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))

In [90]: s
Out[90]:
baz  one    0.206053
foo  two   -0.251905
     one   -2.213588
baz  two    1.063327
qux  two    1.266143
bar  two    0.299368
     one   -0.863838
qux  one    0.408204
dtype: float64

In [91]: s.sort_index()
Out[91]:
bar  one   -0.863838
     two    0.299368
baz  one    0.206053
```

Scroll To Top

```
     two    1.063327
foo  one   -2.213588
     two   -0.251905
qux  one    0.408204
     two    1.266143
dtype: float64

In [92]: s.sort_index(level=0)
Out[92]:
bar  one   -0.863838
     two    0.299368
baz  one    0.206053
     two    1.063327
foo  one   -2.213588
     two   -0.251905
qux  one    0.408204
     two    1.266143
dtype: float64

In [93]: s.sort_index(level=1)
Out[93]:
bar  one   -0.863838
baz  one    0.206053
foo  one   -2.213588
qux  one    0.408204
bar  two    0.299368
baz  two    1.063327
foo  two   -0.251905
qux  two    1.266143
dtype: float64
```

You may also pass a level name to `sort_index` if the MultiIndex levels are named.

```
In [94]: s.index.set_names(['L1', 'L2'], inplace=True)

In [95]: s.sort_index(level='L1')
Out[95]:
L1   L2
bar  one   -0.863838
     two    0.299368
baz  one    0.206053
     two    1.063327
foo  one   -2.213588
     two   -0.251905
qux  one    0.408204
     two    1.266143
dtype: float64

In [96]: s.sort_index(level='L2')
Out[96]:
L1   L2
bar  one   -0.863838
baz  one    0.206053
foo  one   -2.213588
qux  one    0.408204
bar  two    0.299368
baz  two    1.063327
foo  two   -0.251905
qux  two    1.266143
dtype: float64
```

Scroll To Top

On higher dimensional objects, you can sort any of the other axes by level if they have a `MultiIndex`:

```
In [97]: df.T.sort_index(level=1, axis=1)
Out[97]:
        one      zero       one      zero
          x         x         y         y
0  0.600178  2.410179  1.519970  0.132885
1  0.274230  1.450520 -0.493662 -0.023688
```

Indexing will work even if the data are not sorted, but will be rather inefficient (and show a `PerformanceWarning`). It will also return a copy of the data rather than a view:

```
In [98]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
   ....:                     'joe': ['x', 'x', 'z', 'y'],
   ....:                     'jolie': np.random.rand(4)})
   ....:

In [99]: dfm = dfm.set_index(['jim', 'joe'])

In [100]: dfm
Out[100]:
            jolie
jim joe
0   x    0.490671
    x    0.120248
1   z    0.537020
    y    0.110968
```

```
In [4]: dfm.loc[(1, 'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.

Out[4]:
           jolie
jim joe
1   z    0.64094
```

Furthermore if you try to index something that is not fully lexsorted, this can raise:

```
In [5]: dfm.loc[(0,'y'):(1, 'z')]
UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'
```

The `is_lexsorted()` method on an `Index` show if the index is sorted, and the `lexsort_depth` property returns the sort depth:

```
In [101]: dfm.index.is_lexsorted()
Out[101]: False

In [102]: dfm.index.lexsort_depth
Out[102]: 1
```

Scroll To Top

```
In [103]: dfm = dfm.sort_index()
```

```
In [104]: dfm
Out[104]:
             jolie
jim joe
0   x    0.490671
    x    0.120248
1   y    0.110968
    z    0.537020

In [105]: dfm.index.is_lexsorted()
Out[105]: True

In [106]: dfm.index.lexsort_depth
Out[106]: 2
```

And now selection works as expected.

```
In [107]: dfm.loc[(0,'y'):(1, 'z')]
Out[107]:
             jolie
jim joe
1   y    0.110968
    z    0.537020
```

Take Methods

Similar to NumPy ndarrays, pandas Index, Series, and DataFrame also provides the `take` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [108]: index = pd.Index(np.random.randint(0, 1000, 10))

In [109]: index
Out[109]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64')

In [110]: positions = [0, 9, 3]

In [111]: index[positions]
Out[111]: Int64Index([214, 329, 567], dtype='int64')

In [112]: index.take(positions)
Out[112]: Int64Index([214, 329, 567], dtype='int64')

In [113]: ser = pd.Series(np.random.randn(10))

In [114]: ser.iloc[positions]
Out[114]:
0   -0.179666
9    1.824375
3    0.392149
dtype: float64

In [115]: ser.take(positions)
Out[115]:
0   -0.179666
9    1.824375
```

Scroll To Top

```
3    0.392149
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [116]: frm = pd.DataFrame(np.random.randn(5, 3))

In [117]: frm.take([1, 4, 3])
Out[117]: 
          0         1         2
1 -1.237881  0.106854 -1.276829
4  0.629675 -1.425966  1.857704
3  0.979542 -1.633678  0.615855

In [118]: frm.take([0, 2], axis=1)
Out[118]: 
          0         2
0  0.595974  0.601544
1 -1.237881 -1.276829
2 -0.767101  1.499591
3  0.979542  0.615855
4  0.629675  1.857704
```

It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```
In [119]: arr = np.random.randn(10)

In [120]: arr.take([False, False, True, True])
Out[120]: array([-1.1935, -1.1935,  0.6775,  0.6775])

In [121]: arr[[0, 1]]
Out[121]: array([-1.1935,  0.6775])

In [122]: ser = pd.Series(np.random.randn(10))

In [123]: ser.take([False, False, True, True])
Out[123]: 
0    0.233141
0    0.233141
1   -0.223540
1   -0.223540
dtype: float64

In [124]: ser.iloc[[0, 1]]
Out[124]: 
0    0.233141
1   -0.223540
dtype: float64
```

Finally, as a small note on performance, because the `take` method handles a narrower range of inputs, it can offer performance that is a good deal faster than fancy indexing.

Scroll To Top

Index Types

We have discussed `MultiIndex` in the previous sections pretty extensively. `DatetimeIndex` and `PeriodIndex` are shown here, and information about TimedeltaIndex` is found here.

In the following sub-sections we will highlight some other index types.

CategoricalIndex

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements.

```
In [125]: from pandas.api.types import CategoricalDtype

In [126]: df = pd.DataFrame({'A': np.arange(6),
   .....:                    'B': list('aabbca')})
   .....:

In [127]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))

In [128]: df
Out[128]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [129]: df.dtypes
Out[129]:
A       int64
B    category
dtype: object

In [130]: df.B.cat.categories
Out[130]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a `CategoricalIndex`.

```
In [131]: df2 = df.set_index('B')

In [132]: df2.index
Out[132]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
```

Indexing with `__getitem__/.iloc/.loc` works similarly to an `Index` with duplicates. The indexers **must** be in the category or the operation will raise a `KeyError`.

```
In [133]: df2.loc['a']
Out[133]:
   A
B
a  0
```

Scroll To Top

```
a   1
a   5
```

The `CategoricalIndex` is **preserved** after indexing:

```
In [134]: df2.loc['a'].index
Out[134]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
```

Sorting the index will sort by the order of the categories (recall that we created the index with `CategoricalDtype(list('cab'))`, so the sorted order is `cab`).

```
In [135]: df2.sort_index()
Out[135]: 
   A
B   
c  4
a  0
a  1
a  5
b  2
b  3
```

Groupby operations on the index will preserve the index nature as well.

```
In [136]: df2.groupby(level=0).sum()
Out[136]: 
   A
B   
c  4
a  6
b  5

In [137]: df2.groupby(level=0).sum().index
Out[137]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False,
```

Reindexing operations will return a resulting index based on the type of the passed indexer. Passing a list will return a plain-old `Index`; indexing with a `Categorical` will return a `CategoricalIndex`, indexed according to the categories of the **passed** `Categorical` dtype. This allows one to arbitrarily index these even with values **not** in the categories, similarly to how you can reindex **any** pandas index.

```
In [138]: df2.reindex(['a','e'])
Out[138]: 
     A
B     
a  0.0
a  1.0
a  5.0
e  NaN

In [139]: df2.reindex(['a','e']).index
Out[139]: Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [140]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde')))
```

Scroll To Top

```
Out[140]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [141]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde'))).index
Out[141]: CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c', 'd', 'e'],
```

Warning: Reshaping and Comparison operations on a `CategoricalIndex` must have the same categories or a `TypeError` will be raised.

```
In [9]: df3 = pd.DataFrame({'A' : np.arange(6),
                            'B' : pd.Series(list('aabbca')).astype('category')})

In [11]: df3 = df3.set_index('B')

In [11]: df3.index
Out[11]: CategoricalIndex([u'a', u'a', u'b', u'b', u'c', u'a'], categories=[u'a', u'b

In [12]: pd.concat([df2, df3]
TypeError: categories must match existing categories when appending
```

Int64Index and RangeIndex

Warning: Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see here.

`Int64Index` is a fundamental basic index in pandas. This is an Immutable array implementing an ordered, sliceable set. Prior to 0.18.0, the `Int64Index` would provide the default index for all `NDFrame` objects.

`RangeIndex` is a sub-class of `Int64Index` added in version 0.18.0, now providing the default index for all `NDFrame` objects. `RangeIndex` is an optimized version of `Int64Index` that can represent a monotonic ordered set. These are analogous to Python range types.

Float64Index

By default a `Float64Index` will be automatically created when passing floating, or mixed-integer-floating values in index creation. This enables a pure label-based slicing paradigm that makes `[],ix,loc` for scalar indexing and slicing work exactly the same.

```
In [142]: indexf = pd.Index([1.5, 2, 3, 4.5, 5])

In [143]: indexf
Out[143]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [144]: sf = pd.Series(range(5), index=indexf)
```

Scroll To Top

```
In [145]: sf
Out[145]:
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
dtype: int64
```

Scalar selection for `[],.loc` will always be label based. An integer will match an equal float index (e.g. `3` is equivalent to `3.0`).

```
In [146]: sf[3]
Out[146]: 2

In [147]: sf[3.0]
Out[147]: 2

In [148]: sf.loc[3]
Out[148]: 2

In [149]: sf.loc[3.0]
Out[149]: 2
```

The only positional indexing is via `iloc`.

```
In [150]: sf.iloc[3]
Out[150]: 3
```

A scalar index that is not found will raise a `KeyError`. Slicing is primarily on the values of the index when using `[],ix,loc`, and **always** positional when using `iloc`. The exception is when the slice is boolean, in which case it will always be positional.

```
In [151]: sf[2:4]
Out[151]:
2.0    1
3.0    2
dtype: int64

In [152]: sf.loc[2:4]
Out[152]:
2.0    1
3.0    2
dtype: int64

In [153]: sf.iloc[2:4]
Out[153]:
3.0    2
4.5    3
dtype: int64
```

Scroll To Top

In float indexes, slicing using floats is allowed.

```
In [154]: sf[2.1:4.6]
Out[154]:
3.0    2
4.5    3
dtype: int64

In [155]: sf.loc[2.1:4.6]
Out[155]:
3.0    2
4.5    3
dtype: int64
```

In non-float indexes, slicing using floats will raise a `TypeError`.

```
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type (Int64Inde
```

Warning: Using a scalar float indexer for `.iloc` has been removed in 0.18.0, so the following will raise a `TypeError`:

```
In [3]: pd.Series(range(5)).iloc[3.0]
TypeError: cannot do positional indexing on <class 'pandas.indexes.range.RangeIndex'>
```

Here is a typical use-case for using this type of indexing. Imagine that you have a somewhat irregular timedelta-like indexing scheme, but the data is recorded as floats. This could for example be millisecond offsets.

```
In [156]: dfir = pd.concat([pd.DataFrame(np.random.randn(5,2),
   .....:                                index=np.arange(5) * 250.0,
   .....:                                columns=list('AB')),
   .....:                   pd.DataFrame(np.random.randn(6,2),
   .....:                                index=np.arange(4,10) * 250.1,
   .....:                                columns=list('AB'))])
   .....:

In [157]: dfir
Out[157]:
               A         B
0.0     0.997289 -1.693316
250.0  -0.179129 -1.598062
500.0   0.936914  0.912560
750.0  -1.003401  1.632781
1000.0 -0.724626  0.178219
1000.4  0.310610 -0.108002
1250.5 -0.974226 -1.147708
1500.6 -2.281374  0.760010
1750.7 -0.742532  1.533318
2000.8  2.495362 -0.432771
2250.9 -0.068954  0.043520
```

Scroll To Top

Selection operations then will always work on a value basis, for all selection operators.

```
In [158]: dfir[0:1000.4]
Out[158]:
                A         B
0.0      0.997289 -1.693316
250.0   -0.179129 -1.598062
500.0    0.936914  0.912560
750.0   -1.003401  1.632781
1000.0  -0.724626  0.178219
1000.4   0.310610 -0.108002

In [159]: dfir.loc[0:1001,'A']
Out[159]:
0.0       0.997289
250.0    -0.179129
500.0     0.936914
750.0    -1.003401
1000.0   -0.724626
1000.4    0.310610
Name: A, dtype: float64

In [160]: dfir.loc[1000.4]
Out[160]:
A    0.310610
B   -0.108002
Name: 1000.4, dtype: float64
```

You could retrieve the first 1 second (1000 ms) of data as such:

```
In [161]: dfir[0:1000]
Out[161]:
                A         B
0.0      0.997289 -1.693316
250.0   -0.179129 -1.598062
500.0    0.936914  0.912560
750.0   -1.003401  1.632781
1000.0  -0.724626  0.178219
```

If you need integer based selection, you should use `iloc`:

```
In [162]: dfir.iloc[0:5]
Out[162]:
                A         B
0.0      0.997289 -1.693316
250.0   -0.179129 -1.598062
500.0    0.936914  0.912560
750.0   -1.003401  1.632781
1000.0  -0.724626  0.178219
```

IntervalIndex

Scroll To Top

New in version 0.20.0.

`IntervalIndex` together with its own dtype, `interval` as well as the **`Interval`** scalar type, allow first-class support in pandas for interval notation.

The `IntervalIndex` allows some unique indexing and is also used as a return type for the categories in **`cut()`** and **`qcut()`**.

> **Warning:** These indexing behaviors are provisional and may change in a future version of pandas.

An `IntervalIndex` can be used in `Series` and in `DataFrame` as the index.

```
In [163]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
   .....:                   index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
   .....:

In [164]: df
Out[164]:
        A
(0, 1]  1
(1, 2]  2
(2, 3]  3
(3, 4]  4
```

Label based indexing via `.loc` along the edges of an interval works as you would expect, selecting that particular interval.

```
In [165]: df.loc[2]
Out[165]:
A    2
Name: (1, 2], dtype: int64

In [166]: df.loc[[2, 3]]
Out[166]:
        A
(1, 2]  2
(2, 3]  3
```

If you select a label contained within an interval, this will also select the interval.

```
In [167]: df.loc[2.5]
Out[167]:
A    3
Name: (2, 3], dtype: int64

In [168]: df.loc[[2.5, 3.5]]
Out[168]:
        A
(2, 3]  3
(3, 4]  4
```

`Interval` and `IntervalIndex` are used by `cut` and `qcut`:

Scroll To Top

```
In [169]: c = pd.cut(range(4), bins=2)

In [170]: c
Out[170]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]

In [171]: c.categories
Out[171]:
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]]
              closed='right',
              dtype='interval[float64]')
```

Furthermore, `IntervalIndex` allows one to bin other data with these same bins, with `NaN` representing a missing value similar to other dtypes.

```
In [172]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[172]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
```

Generating Ranges of Intervals

If we need intervals on a regular frequency, we can use the **`interval_range()`** function to create an `IntervalIndex` using various combinations of `start`, `end`, and `periods`. The default frequency for `interval_range` is a 1 for numeric intervals, and calendar day for datetime-like intervals:

```
In [173]: pd.interval_range(start=0, end=5)
Out[173]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]
              closed='right',
              dtype='interval[int64]')

In [174]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4)
Out[174]:
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03], (2017-01-03, 2017-01
              closed='right',
              dtype='interval[datetime64[ns]]')

In [175]: pd.interval_range(end=pd.Timedelta('3 days'), periods=3)
Out[175]:
IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:00:00, 2 days 00:00:00],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

The `freq` parameter can used to specify non-default frequencies, and can utilize a variety of frequency aliases with datetime-like intervals:

```
In [176]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[176]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]]
              closed='right',
              dtype='interval[float64]')
```

Scroll To Top

```
In [177]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4, freq='W')
Out[177]:
IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-01
              closed='right',
              dtype='interval[datetime64[ns]]')

In [178]: pd.interval_range(start=pd.Timedelta('0 days'), periods=3, freq='9H')
Out[178]:
IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days 18:00:00],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

Additionally, the `closed` parameter can be used to specify which side(s) the intervals are closed on. Intervals are closed on the right side by default.

```
In [179]: pd.interval_range(start=0, end=4, closed='both')
Out[179]:
IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]]
              closed='both',
              dtype='interval[int64]')

In [180]: pd.interval_range(start=0, end=4, closed='neither')
Out[180]:
IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)]
              closed='neither',
              dtype='interval[int64]')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced intervals from `start` to `end` inclusively, with `periods` number of elements in the resulting `IntervalIndex`:

```
In [181]: pd.interval_range(start=0, end=6, periods=4)
Out[181]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]]
              closed='right',
              dtype='interval[float64]')

In [182]: pd.interval_range(pd.Timestamp('2018-01-01'), pd.Timestamp('2018-02-28'), per
Out[182]:
IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-02-08 16:0
              closed='right',
              dtype='interval[datetime64[ns]]')
```

Miscellaneous indexing FAQ

Integer indexing

Label-based indexing with integer axis labels is a thorny topic. It has been discussed heavily on mailing lists and among various members of the scientific Python community. In pandas, our general viewpoint is

Scroll To Top

that labels matter more than integer locations. Therefore, with an integer axis index only label-based indexing is possible with the standard tools like `.loc`. The following code will generate exceptions:

```
s = pd.Series(range(5))
s[-1]
df = pd.DataFrame(np.random.randn(5, 4))
df
df.loc[-2:]
```

This deliberate decision was made to prevent ambiguities and subtle bugs (many users reported finding bugs when the API change was made to stop “falling back” on position-based indexing).

Non-monotonic indexes require exact matches

If the index of a `Series` or `DataFrame` is monotonically increasing or decreasing, then the bounds of a label-based slice can be outside the range of the index, much like slice indexing a normal Python `list`. Monotonicity of an index can be tested with the `is_monotonic_increasing` and `is_monotonic_decreasing` attributes.

```
In [183]: df = pd.DataFrame(index=[2,3,3,4,5], columns=['data'], data=list(range(5)))

In [184]: df.index.is_monotonic_increasing
Out[184]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [185]: df.loc[0:4, :]
Out[185]:
   data
2     0
3     1
3     2
4     3

# slice is are outside the index, so empty DataFrame is returned
In [186]: df.loc[13:15, :]
Out[186]:
Empty DataFrame
Columns: [data]
Index: []
```

On the other hand, if the index is not monotonic, then both slice bounds must be unique members of the index.

```
In [187]: df = pd.DataFrame(index=[2,3,1,4,3,5], columns=['data'], data=list(range(6)))

In [188]: df.index.is_monotonic_increasing
Out[188]: False

# OK because 2 and 4 are in the index
In [189]: df.loc[2:4, :]
Out[189]:
   data
2     0
3     1
```

Scroll To Top

```
1      2
4      3
```

```
# 0 is not in the index
In [9]: df.loc[0:4, :]
KeyError: 0

# 3 is not a unique label
In [11]: df.loc[2:3, :]
KeyError: 'Cannot get right slice bound for non-unique label: 3'
```

`Index.is_monotonic_increasing()` and **`Index.is_monotonic_decreasing()`** only check that an index is weakly monotonic. To check for strict montonicity, you can combine one of those with **`Index.is_unique()`**

```
In [190]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])

In [191]: weakly_monotonic
Out[191]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [192]: weakly_monotonic.is_monotonic_increasing
Out[192]: True

In [193]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_unique
Out[193]: False
```

Endpoints are inclusive

Compared with standard Python sequence slicing in which the slice endpoint is not inclusive, label-based slicing in pandas **is inclusive**. The primary reason for this is that it is often not possible to easily determine the “successor” or next element after a particular label in an index. For example, consider the following Series:

```
In [194]: s = pd.Series(np.random.randn(6), index=list('abcdef'))

In [195]: s
Out[195]:
a    0.112246
b    0.871721
c   -0.816064
d   -0.784880
e    1.030659
f    0.187483
dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be accomplished as such:

```
In [196]: s[2:5]
Out[196]:
c   -0.816064
d   -0.784880
e    1.030659
dtype: float64
```

Scroll To Top

However, if you only had `c` and `e`, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.loc['c':'e'+1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design to make label-based slicing include both endpoints:

```
In [197]: s.loc['c':'e']
Out[197]:
c   -0.816064
d   -0.784880
e    1.030659
dtype: float64
```

This is most definitely a “practicality beats purity” sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

Indexing potentially changes underlying Series dtype

The different indexing operation can potentially change the dtype of a `Series`.

```
In [198]: series1 = pd.Series([1, 2, 3])

In [199]: series1.dtype
Out[199]: dtype('int64')

In [200]: res = series1.reindex([0, 4])

In [201]: res.dtype
Out[201]: dtype('float64')

In [202]: res
Out[202]:
0    1.0
4    NaN
dtype: float64
```

```
In [203]: series2 = pd.Series([True])

In [204]: series2.dtype
Out[204]: dtype('bool')

In [205]: res = series2.reindex_like(series1)

In [206]: res.dtype
Out[206]: dtype('O')

In [207]: res
Out[207]:
0    True
1     NaN
```

Scroll To Top

```
2      NaN
dtype: object
```

This is because the (re)indexing operations above silently inserts `NaNs` and the `dtype` changes accordingly. This can cause some issues when using `numpy ufuncs` such as `numpy.logical_and`.

See the this old issue for a more detailed discussion.

Scroll To Top