(continued from previous page)

```
In [300]: ind
Out[300]: Int64Index([1, 2, 3], dtype='int64', name='bob')
```

`set_names`, `set_levels`, and `set_codes` also take an optional `level` argument

```
In [301]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first
↪', 'second'])

In [302]: index
Out[302]:
MultiIndex([(0, 'one'),
            (0, 'two'),
            (1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['first', 'second'])

In [303]: index.levels[1]
Out[303]: Index(['one', 'two'], dtype='object', name='second')

In [304]: index.set_levels(["a", "b"], level=1)
Out[304]:
MultiIndex([(0, 'a'),
            (0, 'b'),
            (1, 'a'),
            (1, 'b'),
            (2, 'a'),
            (2, 'b')],
           names=['first', 'second'])
```

Set operations on Index objects

The two main operations are `union (|)` and `intersection (&)`. These can be directly called as instance methods or used via overloaded operators. Difference is provided via the `.difference()` method.

```
In [305]: a = pd.Index(['c', 'b', 'a'])

In [306]: b = pd.Index(['c', 'e', 'd'])

In [307]: a | b
Out[307]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [308]: a & b
Out[308]: Index(['c'], dtype='object')

In [309]: a.difference(b)
Out[309]: Index(['a', 'b'], dtype='object')
```

Also available is the `symmetric_difference (^)` operation, which returns elements that appear in either `idx1` or `idx2`, but not in both. This is equivalent to the Index created by `idx1.difference(idx2).union(idx2.difference(idx1))`, with duplicates dropped.

```
In [310]: idx1 = pd.Index([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
In [311]: idx2 = pd.Index([2, 3, 4, 5])

In [312]: idx1.symmetric_difference(idx2)
Out[312]: Int64Index([1, 5], dtype='int64')

In [313]: idx1 ^ idx2
Out[313]: Int64Index([1, 5], dtype='int64')
```

Note: The resulting index from a set operation will be sorted in ascending order.

When performing *`Index.union()`* between indexes with different dtypes, the indexes must be cast to a common dtype. Typically, though not always, this is object dtype. The exception is when performing a union between integer and float data. In this case, the integer values are converted to float

```
In [314]: idx1 = pd.Index([0, 1, 2])

In [315]: idx2 = pd.Index([0.5, 1.5])

In [316]: idx1 | idx2
Out[316]: Float64Index([0.0, 0.5, 1.0, 1.5, 2.0], dtype='float64')
```

Missing values

Important: Even though `Index` can hold missing values (`NaN`), it should be avoided if you do not want any unexpected results. For example, some operations exclude missing values implicitly.

`Index.fillna` fills missing values with specified scalar value.

```
In [317]: idx1 = pd.Index([1, np.nan, 3, 4])

In [318]: idx1
Out[318]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')

In [319]: idx1.fillna(2)
Out[319]: Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')

In [320]: idx2 = pd.DatetimeIndex([pd.Timestamp('2011-01-01'),
   .....:                          pd.NaT,
   .....:                          pd.Timestamp('2011-01-03')])
   .....:

In [321]: idx2
Out[321]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'], dtype='datetime64[ns]',␣
↪freq=None)

In [322]: idx2.fillna(pd.Timestamp('2011-01-02'))
Out[322]: DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], dtype=
↪'datetime64[ns]', freq=None)
```

2.2.21 Set / reset index

Occasionally you will load or create a data set into a DataFrame and want to add an index after you've already done so. There are a couple of different ways.

Set an index

DataFrame has a *set_index()* method which takes a column name (for a regular `Index`) or a list of column names (for a `MultiIndex`). To create a new, re-indexed DataFrame:

```
In [323]: data
Out[323]:
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0

In [324]: indexed1 = data.set_index('c')

In [325]: indexed1
Out[325]:
     a    b    d
c
z  bar  one  1.0
y  bar  two  2.0
x  foo  one  3.0
w  foo  two  4.0

In [326]: indexed2 = data.set_index(['a', 'b'])

In [327]: indexed2
Out[327]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [328]: frame = data.set_index('c', drop=False)

In [329]: frame = frame.set_index(['a', 'b'], append=True)

In [330]: frame
Out[330]:
           c    d
c a   b
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0
```

Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):

```
In [331]: data.set_index('c', drop=False)
Out[331]:
     a    b  c    d
c
z  bar  one  z  1.0
y  bar  two  y  2.0
x  foo  one  x  3.0
w  foo  two  w  4.0

In [332]: data.set_index(['a', 'b'], inplace=True)

In [333]: data
Out[333]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

Reset the index

As a convenience, there is a new function on DataFrame called `reset_index()` which transfers the index values into the DataFrame's columns and sets a simple integer index. This is the inverse operation of `set_index()`.

```
In [334]: data
Out[334]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0

In [335]: data.reset_index()
Out[335]:
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0
```

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the `names` attribute.

You can use the `level` keyword to remove only a portion of the index:

```
In [336]: frame
Out[336]:
           c    d
c a   b
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0
```

(continues on next page)

(continued from previous page)

```
In [337]: frame.reset_index(level=1)
Out[337]:
         a  c    d
c b
z one  bar  z  1.0
y two  bar  y  2.0
x one  foo  x  3.0
w two  foo  w  4.0
```

`reset_index` takes an optional parameter `drop` which if true simply discards the index, instead of putting index values in the DataFrame's columns.

Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
data.index = index
```

2.2.22 Returning a view versus a copy

When setting values in a pandas object, care must be taken to avoid what is called `chained indexing`. Here is an example.

```
In [338]: dfmi = pd.DataFrame([list('abcd'),
   .....:                      list('efgh'),
   .....:                      list('ijkl'),
   .....:                      list('mnop')],
   .....:                     columns=pd.MultiIndex.from_product([['one', 'two'],
   .....:                                                         ['first', 'second
↪']]))
   .....:

In [339]: dfmi
Out[339]:
    one          two
  first second first second
0     a      b     c      d
1     e      f     g      h
2     i      j     k      l
3     m      n     o      p
```

Compare these two access methods:

```
In [340]: dfmi['one']['second']
Out[340]:
0    b
1    f
2    j
3    n
Name: second, dtype: object
```

```
In [341]: dfmi.loc[:, ('one', 'second')]
Out[341]:
```

(continues on next page)

(continued from previous page)

```
0    b
1    f
2    j
3    n
Name: (one, second), dtype: object
```

These both yield the same results, so which should you use? It is instructive to understand the order of operations on these and why method 2 (`.loc`) is much preferred over method 1 (chained `[]`).

`dfmi['one']` selects the first level of the columns and returns a DataFrame that is singly-indexed. Then another Python operation `dfmi_with_one['second']` selects the series indexed by `'second'`. This is indicated by the variable `dfmi_with_one` because pandas sees these operations as separate events. e.g. separate calls to `__getitem__`, so it has to treat them as linear operations, they happen one after another.

Contrast this to `df.loc[:,('one','second')]` which passes a nested tuple of `(slice(None),('one', 'second'))` to a single call to `__getitem__`. This allows pandas to deal with this as a single entity. Furthermore this order of operations *can* be significantly faster, and allows one to index *both* axes if so desired.

Why does assignment fail when using chained indexing?

The problem in the previous section is just a performance issue. What's up with the `SettingWithCopy` warning? We don't **usually** throw warnings around when you do something that might cost a few extra milliseconds!

But it turns out that assigning to the product of chained indexing has inherently unpredictable results. To see this, think about how the Python interpreter executes this code:

```
dfmi.loc[:, ('one', 'second')] = value
# becomes
dfmi.loc.__setitem__((slice(None), ('one', 'second')), value)
```

But this code is handled differently:

```
dfmi['one']['second'] = value
# becomes
dfmi.__getitem__('one').__setitem__('second', value)
```

See that `__getitem__` in there? Outside of simple cases, it's very hard to predict whether it will return a view or a copy (it depends on the memory layout of the array, about which pandas makes no guarantees), and therefore whether the `__setitem__` will modify `dfmi` or a temporary object that gets thrown out immediately afterward. **That's** what `SettingWithCopy` is warning you about!

Note: You may be wondering whether we should be concerned about the `loc` property in the first example. But `dfmi.loc` is guaranteed to be `dfmi` itself with modified indexing behavior, so `dfmi.loc.__getitem__` / `dfmi.loc.__setitem__` operate on `dfmi` directly. Of course, `dfmi.loc.__getitem__(idx)` may be a view or a copy of `dfmi`.

Sometimes a `SettingWithCopy` warning will arise at times when there's no obvious chained indexing going on. **These** are the bugs that `SettingWithCopy` is designed to catch! Pandas is probably trying to warn you that you've done this:

```
def do_something(df):
    foo = df[['bar', 'baz']]  # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...
```

(continues on next page)

(continued from previous page)

```
    # We don't know whether this will modify df or not!
    foo['quux'] = value
    return foo
```

Yikes!

Evaluation order matters

When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.

Pandas has the `SettingWithCopyWarning` because assigning to a copy of a slice is frequently not intentional, but a mistake caused by chained indexing returning a copy where a slice was expected.

If you would like pandas to be more or less trusting about assignment to a chained indexing expression, you can set the *option* `mode.chained_assignment` to one of these values:

- `'warn'`, the default, means a `SettingWithCopyWarning` is printed.
- `'raise'` means pandas will raise a `SettingWithCopyException` you have to deal with.
- `None` will suppress the warnings entirely.

```
In [342]: dfb = pd.DataFrame({'a': ['one', 'one', 'two',
   .....:                           'three', 'two', 'one', 'six'],
   .....:                     'c': np.arange(7)})
   .....:

# This will show the SettingWithCopyWarning
# but the frame values will be set
In [343]: dfb['c'][dfb['a'].str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```
>>> pd.set_option('mode.chained_assignment','warn')
>>> dfb[dfb['a'].str.startswith('o')]['c'] = 42
Traceback (most recent call last)
     ...
SettingWithCopyWarning:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

A chained assignment can also crop up in setting in a mixed dtype frame.

Note: These setting rules apply to all of `.loc`/`.iloc`.

This is the correct access method:

```
In [344]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [345]: dfc.loc[0, 'A'] = 11

In [346]: dfc
Out[346]:
     A  B
```

(continues on next page)

(continued from previous page)

```
0   11  1
1  bbb  2
2  ccc  3
```

This *can* work at times, but it is not guaranteed to, and therefore should be avoided:

```
In [347]: dfc = dfc.copy()

In [348]: dfc['A'][0] = 111

In [349]: dfc
Out[349]:
     A  B
0  111  1
1  bbb  2
2  ccc  3
```

This will **not** work at all, and so should be avoided:

```
>>> pd.set_option('mode.chained_assignment','raise')
>>> dfc.loc[0]['A'] = 1111
Traceback (most recent call last)
     ...
SettingWithCopyException:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

> **Warning:** The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.

2.3 MultiIndex / advanced indexing

This section covers *indexing with a MultiIndex* and *other 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.

2.3.1 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.

Changed in version 0.24.0: `MultiIndex.labels` has been renamed to `MultiIndex.codes` and `MultiIndex.set_labels` to `MultiIndex.set_codes`.

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()`), a crossed set of iterables (using `MultiIndex.from_product()`), or a `DataFrame` (using `MultiIndex.from_frame()`). 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'),
 ('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([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')],
           names=['first', 'second'])
```

(continues on next page)

(continued from previous page)

```
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()* method:

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

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

You can also construct a MultiIndex from a DataFrame directly, using the method *MultiIndex.from_frame()*. This is a complementary method to *MultiIndex.to_frame()*.

New in version 0.24.0.

```
In [10]: df = pd.DataFrame([['bar', 'one'], ['bar', 'two'],
   ....:                    ['foo', 'one'], ['foo', 'two']],
   ....:                   columns=['first', 'second'])
   ....: 

In [11]: pd.MultiIndex.from_frame(df)
Out[11]: 
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('foo', 'one'),
            ('foo', 'two')],
           names=['first', 'second'])
```

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

```
In [12]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
   ....:           np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
   ....: 
```

(continues on next page)

(continued from previous page)

```
In [13]: s = pd.Series(np.random.randn(8), index=arrays)

In [14]: s
Out[14]:
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

In [15]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)

In [16]: df
Out[16]:
                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 [17]: df.index.names
Out[17]: 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 [18]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'],
↪columns=index)

In [19]: df
Out[19]:
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 [20]: pd.DataFrame(np.random.randn(6, 6), index=index[:6], columns=index[:6])
Out[20]:
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
```

(continues on next page)

(continued from previous page)

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

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

```
In [22]: pd.Series(np.random.randn(8), index=tuples)
Out[22]:
(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 [23]: index.get_level_values(0)
Out[23]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], dtype='object
↪', name='first')

In [24]: index.get_level_values('second')
Out[24]: Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'], dtype='object
↪', name='second')
```

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 [25]: df['bar']
Out[25]:
second        one       two
A        0.895717  0.805244
B        0.410835  0.813850
C       -1.413681  1.607920
```

(continues on next page)

(continued from previous page)

```
In [26]: df['bar', 'one']
Out[26]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64

In [27]: df['bar']['one']
Out[27]:
A    0.895717
B    0.410835
C   -1.413681
Name: one, dtype: float64

In [28]: s['qux']
Out[28]:
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 `MultiIndex` keeps all the defined levels of an index, even if they are not actually used. When slicing an index, you may notice this. For example:

```
In [29]: df.columns.levels  # original MultiIndex
Out[29]: FrozenList([['bar', 'baz', 'foo', 'qux'], ['one', 'two']])

In [30]: df[['foo','qux']].columns.levels  # sliced
Out[30]: FrozenList([['bar', 'baz', 'foo', 'qux'], ['one', 'two']])
```

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 `get_level_values()` method.

```
In [31]: df[['foo', 'qux']].columns.to_numpy()
Out[31]:
array([('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')],
      dtype=object)

# for a specific level
In [32]: df[['foo', 'qux']].columns.get_level_values(0)
Out[32]: 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.

```
In [33]: new_mi = df[['foo', 'qux']].columns.remove_unused_levels()

In [34]: new_mi.levels
Out[34]: FrozenList([['foo', 'qux'], ['one', 'two']])
```

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 [35]: s + s[:-2]
Out[35]:
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 [36]: s + s[::2]
Out[36]:
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
```

The *reindex()* method of `Series`/`DataFrames` can be called with another `MultiIndex`, or even a list or array of tuples:

```
In [37]: s.reindex(index[:3])
Out[37]:
first  second
bar    one      -0.861849
       two      -2.104569
baz    one      -0.494929
dtype: float64

In [38]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
Out[38]:
foo  two   -0.706771
bar  one   -0.861849
qux  one   -1.039575
baz  one   -0.494929
dtype: float64
```

2.3.2 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:

```
In [39]: df = df.T

In [40]: df
Out[40]:
                     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 [41]: df.loc[('bar', 'two')]
Out[41]:
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 [42]: df.loc[('bar', 'two'), 'A']
Out[42]: 0.8052440253863785
```

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 [43]: df.loc['baz':'foo']
Out[43]:
                     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 [44]: df.loc[('baz', 'two'):('qux', 'one')]
Out[44]:
```

(continues on next page)

(continued from previous page)

```
                     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

In [45]: df.loc[('baz', 'two'):'foo']
Out[45]:
                     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 [46]: df.loc[[('bar', 'two'), ('qux', 'one')]]
Out[46]:
                     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 [47]: s = pd.Series([1, 2, 3, 4, 5, 6],
   ....:               index=pd.MultiIndex.from_product([["A", "B"], ["c", "d", "e
↪"]]))
   ....:

In [48]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
Out[48]:
A  c    1
B  d    5
dtype: int64

In [49]: s.loc[(["A", "B"], ["c", "d"])]  # tuple of lists
Out[49]:
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.

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'), ...), :]             # noqa: E999
```

You should **not** do this:

```
df.loc[(slice('A1', 'A3'), ...)]                # noqa: E999
```

```
In [50]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:

In [51]: miindex = pd.MultiIndex.from_product([mklbl('A', 4),
   ....:                                       mklbl('B', 2),
   ....:                                       mklbl('C', 4),
   ....:                                       mklbl('D', 2)])
   ....:

In [52]: micolumns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'),
   ....:                                        ('b', 'foo'), ('b', 'bah')],
   ....:                                       names=['lvl0', 'lvl1'])
   ....:

In [53]: dfmi = pd.DataFrame(np.arange(len(miindex) * len(micolumns))
   ....:                       .reshape((len(miindex), len(micolumns))),
   ....:                     index=miindex,
   ....:                     columns=micolumns).sort_index().sort_index(axis=1)
   ....:

In [54]: dfmi
Out[54]:
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
...          ...  ...  ...  ...
A3 B1 C1 D1  237  236  239  238
      C2 D0  241  240  243  242
         D1  245  244  247  246
```

(continues on next page)

(continued from previous page)

```
      C3 D0   249  248  251  250
         D1   253  252  255  254

[64 rows x 4 columns]
```

Basic MultiIndex slicing using slices, lists, and labels.

```
In [55]: dfmi.loc[(slice('A1', 'A3'), slice(None), ['C1', 'C3']), :]
Out[55]:
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
...          ...  ...  ...  ...
A3 B0 C3 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 [56]: idx = pd.IndexSlice

In [57]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[57]:
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
...          ...  ...
A3 B0 C3 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 [58]: dfmi.loc['A1', (slice(None), 'foo')]
Out[58]:
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
      D1   76   78
```

(continues on next page)

(continued from previous page)

```
   C2 D0   80   82
...       ...  ...
B1 C1 D1  108  110
   C2 D0  112  114
      D1  116  118
   C3 D0  120  122
      D1  124  126

[16 rows x 2 columns]

In [59]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[59]: 
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
...          ...  ...
A3 B0 C3 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 [60]: mask = dfmi[('a', 'foo')] > 200

In [61]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[61]: 
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 [62]: dfmi.loc(axis=0)[:, :, ['C1', 'C3']]
Out[62]: 
lvl0           a         b     
lvl1         bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
         D1   13   12   15   14
      C3 D0   25   24   27   26
         D1   29   28   31   30
   B1 C1 D0   41   40   43   42
...          ...  ...  ...  ...
A3 B0 C3 D1  221  220  223  222
```

(continues on next page)

(continued from previous page)

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

In [64]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

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  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0   17   16   19   18
...          ...  ...  ...  ...
A3 B1 C1 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 [66]: df2 = dfmi.copy()

In [67]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [68]: df2
Out[68]:
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
...             ...     ...     ...     ...
A3 B1 C1 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 [69]: df
Out[69]:
                     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 [70]: df.xs('one', level='second')
Out[70]:
              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 [71]: df.loc[(slice(None), 'one'), :]
Out[71]:
                     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 [72]: df = df.T

In [73]: df.xs('one', level='second', axis=1)
Out[73]:
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 [74]: df.loc[:, (slice(None), 'one')]
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
```

`xs` also allows selection with multiple keys.

```
In [75]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out[75]:
first         bar
second        one
A        0.895717
B        0.410835
C       -1.413681
```

```
# using the slicers
In [76]: df.loc[:, ('bar', 'one')]
Out[76]:
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 [77]: df.xs('one', level='second', axis=1, drop_level=False)
Out[77]:
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 [78]: df.xs('one', level='second', axis=1, drop_level=True)
Out[78]:
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
```

Advanced reindexing and alignment

Using the parameter level in the *reindex()* and *align()* methods of pandas objects is useful to broadcast values across a level. For instance:

```
In [79]: midx = pd.MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
   ....:                      codes=[[1, 1, 0, 0], [1, 0, 1, 0]])
   ....:

In [80]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)

In [81]: df
Out[81]:
               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 [82]: df2 = df.mean(level=0)
```

(continues on next page)

(continued from previous page)

```
In [83]: df2
Out[83]:
             0         1
one   1.060074 -0.109716
zero  1.271532  0.713416

In [84]: df2.reindex(df.index, level=0)
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

# aligning
In [85]: df_aligned, df2_aligned = df.align(df2, level=0)

In [86]: df_aligned
Out[86]:
               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 [87]: df2_aligned
Out[87]:
               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`

The *`swaplevel()`* method can switch the order of two levels:

```
In [88]: df[:5]
Out[88]:
               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 [89]: df[:5].swaplevel(0, 1, axis=0)
Out[89]:
               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()* method generalizes the `swaplevel` method, allowing you to permute the hierarchical index levels in one step:

```
In [90]: df[:5].reorder_levels([1, 0], axis=0)
Out[90]:
                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
```

Renaming names of an `Index` or `MultiIndex`

The *rename()* method is used to rename the labels of a `MultiIndex`, and is typically used to rename the columns of a `DataFrame`. The `columns` argument of `rename` allows a dictionary to be specified that includes only the columns you wish to rename.

```
In [91]: df.rename(columns={0: "col0", 1: "col1"})
Out[91]:
            col0      col1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520
```

This method can also be used to rename specific labels of the main index of the `DataFrame`.

```
In [92]: df.rename(index={"one": "two", "y": "z"})
Out[92]:
                0         1
two  z  1.519970 -0.493662
     x  0.600178  0.274230
zero z  0.132885 -0.023688
     x  2.410179  1.450520
```

The *rename_axis()* method is used to rename the name of a `Index` or `MultiIndex`. In particular, the names of the levels of a `MultiIndex` can be specified, which is useful if `reset_index()` is later used to move the values from the `MultiIndex` to a column.

```
In [93]: df.rename_axis(index=['abc', 'def'])
Out[93]:
                  0         1
abc  def
one  y     1.519970 -0.493662
     x     0.600178  0.274230
zero y     0.132885 -0.023688
     x     2.410179  1.450520
```

Note that the columns of a `DataFrame` are an index, so that using `rename_axis` with the `columns` argument will change the name of that index.

```
In [94]: df.rename_axis(columns="Cols").columns
Out[94]: RangeIndex(start=0, stop=2, step=1, name='Cols')
```

Both `rename` and `rename_axis` support specifying a dictionary, `Series` or a mapping function to map labels/names to new values.

When working with an `Index` object directly, rather than via a `DataFrame`, *`Index.set_names()`* can be used to change the names.

```
In [95]: mi = pd.MultiIndex.from_product([[1, 2], ['a', 'b']], names=['x', 'y'])

In [96]: mi.names
Out[96]: FrozenList(['x', 'y'])

In [97]: mi2 = mi.rename("new name", level=0)

In [98]: mi2
Out[98]:
MultiIndex([(1, 'a'),
            (1, 'b'),
            (2, 'a'),
            (2, 'b')],
           names=['new name', 'y'])
```

You cannot set the names of the MultiIndex via a level.

```
In [99]: mi.levels[0].name = "name via level"
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-99-35d32a9a5218> in <module>
----> 1 mi.levels[0].name = "name via level"

/pandas-release/pandas/pandas/core/indexes/base.py in name(self, value)
   1189             # Used in MultiIndex.levels to avoid silently ignoring name␣
↪updates.
   1190             raise RuntimeError(
-> 1191                 "Cannot set name on a level of a MultiIndex. Use "
   1192                 "'MultiIndex.set_names' instead."
   1193             )

RuntimeError: Cannot set name on a level of a MultiIndex. Use 'MultiIndex.set_names'␣
↪instead.
```

Use *`Index.set_names()`* instead.

2.3.3 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 [100]: import random

In [101]: random.shuffle(tuples)

In [102]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))

In [103]: s
Out[103]:
qux  one    0.206053
foo  two   -0.251905
```

(continues on next page)