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.

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()`), 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", "seconc

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'])

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()`.

```
In [10]: df = pd.DataFrame(
   ....:     [["bar", "one"], ["bar", "two"], ["foo", "one"], ["foo", "tw
   ....:     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", "
   ....:     np.array(["one", "two", "one", "two", "one", "two", "one", "
   ....: ]
   ....:

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"],

In [19]: df
Out[19]:
first        bar                 baz  ...       foo       qux          
second       one       two       one  ...       two       one       two
A       0.895717  0.805244 -1.206412  ...  1.340309 -1.170299 -0.226169
B       0.410835  0.813850  0.132003  ... -1.187678  1.130127 -1.436737
C      -1.413681  1.607920  1.024180  ... -2.211372  0.974466 -2.006747

[3 rows x 8 columns]

In [20]: pd.DataFrame(np.random.randn(6, 6), index=index[:6], columns=inc
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
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'],

In [24]: index.get_level_values("second")
Out[24]: Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'],
```

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
```

```
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
Out[38]:
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:

```
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:

```
In [43]: df.loc["bar"]
Out[43]: 
               A         B         C
second                              
one     0.895717  0.410835 -1.413681
two     0.805244  0.813850  1.607920
```

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 [44]: df.loc["baz":"foo"]
Out[44]: 
                     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 [45]: df.loc[("baz", "two"):("qux", "one")]
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
qux   one    -1.170299  1.130127  0.974466

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

In [49]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
```

```
In [49]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
Out[49]:
A  c    1
B  d    5
dtype: int64

In [50]: s.loc[(["A", "B"], ["c", "d"])]  # tuple of lists
Out[50]:
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 [51]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....: 

In [52]: miindex = pd.MultiIndex.from_product(
   ....:     [mklbl("A", 4), mklbl("B", 2), mklbl("C", 4), mklbl("D", 2)]
   ....: )
   ....: 

In [53]: micolumns = pd.MultiIndex.from_tuples(
   ....:     [("a", "foo"), ("a", "bar"), ("b", "foo"), ("b", "bah")], na
   ....: )
   ....: 

In [54]: 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 [55]: dfmi
Out[55]: 
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
      C3 D0  249  248  251  250
         D1  253  252  255  254

[64 rows x 4 columns]
```

Basic MultiIndex slicing using slices, lists, and labels.

Basic MultiIndex slicing using slices, lists, and labels.

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

In [58]: dfmi.loc[idx[:, :, ["C1", "C3"]], idx[:, "foo"]]
Out[58]:
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 [59]: dfmi.loc["A1", (slice(None), "foo")]
Out[59]: 
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
      D1   76   78
   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 [60]: dfmi.loc[idx[:, :, ["C1", "C3"]], idx[:, "foo"]]
Out[60]: 
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 [61]: mask = dfmi[("a", "foo")] > 200

In [62]: dfmi.loc[idx[mask, :, ["C1", "C3"]], idx[:, "foo"]]
Out[62]: 
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 [63]: dfmi.loc(axis=0)[:, :, ["C1", "C3"]]
Out[63]: 
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
   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 [64]: df2 = dfmi.copy()

In [65]: df2.loc(axis=0)[:, :, ["C1", "C3"]] = -10

In [66]: df2
Out[66]: 
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 [67]: df2 = dfmi.copy()

In [68]: df2.loc[idx[:, :, ["C1", "C3"]], :] = df2 * 1000

In [69]: df2
Out[69]: 
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 [70]: df
Out[70]: 
                     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 [71]: df.xs("one", level="second")
Out[71]:
              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 [72]: df.loc[(slice(None), "one"), :]
Out[72]:
                     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 [73]: df = df.T

In [74]: df.xs("one", level="second", axis=1)
Out[74]:
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 [75]: df.loc[:, (slice(None), "one")]
Out[75]:
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 [76]: df.xs(("one", "bar"), level=("second", "first"), axis=1)
```

```
In [76]: df.xs(("one", "bar"), level=("second", "first"), axis=1)
Out[76]: 
first        bar
second       one

A       0.895717
B       0.410835
C      -1.413681
```

```
# using the slicers
In [77]: df.loc[:, ("bar", "one")]
Out[77]: 
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 [78]: df.xs("one", level="second", axis=1, drop_level=False)
Out[78]: 
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 [79]: df.xs("one", level="second", axis=1, drop_level=True)
Out[79]: 
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 [80]: midx = pd.MultiIndex(
```

```
In [80]: midx = pd.MultiIndex(
   ....:     levels=[["zero", "one"], ["x", "y"]], codes=[[1, 1, 0, 0], [
   ....: )
   ....:

In [81]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)

In [82]: df
Out[82]:
               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 [83]: df2 = df.groupby(level=0).mean()

In [84]: df2
Out[84]:
             0         1
one   1.060074 -0.109716
zero  1.271532  0.713416

In [85]: df2.reindex(df.index, level=0)
Out[85]:
               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 [86]: df_aligned, df2_aligned = df.align(df2, level=0)

In [87]: df_aligned
Out[87]:
               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 [88]: df2_aligned
Out[88]:
               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 [89]: df[:5]
Out[89]:
               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 [90]: df[:5].swaplevel(0, 1, 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
```

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 [91]: df[:5].reorder_levels([1, 0], axis=0)
Out[91]:
               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 [92]: df.rename(columns={0: "col0", 1: "col1"})
Out[92]: 
            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 [93]: df.rename(index={"one": "two", "y": "z"})
Out[93]: 
               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 [94]: df.rename_axis(index=["abc", "def"])
Out[94]: 
                 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 [95]: df.rename_axis(columns="Cols").columns
Out[95]: 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.

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 [96]: mi = pd.MultiIndex.from_product([[1, 2], ["a", "b"]], names=["x"

In [97]: mi.names
Out[97]: FrozenList(['x', 'y'])

In [98]: mi2 = mi.rename("new name", level=0)

In [99]: mi2
Out[99]:
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 [100]: mi.levels[0].name = "name via level"
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call las
Cell In[100], line 1
----> 1 mi.levels[0].name = "name via level"

File ~/work/pandas/pandas/pandas/core/indexes/base.py:1745, in Index.name
   1741 @name.setter
   1742 def name(self, value: Hashable) -> None:
   1743     if self._no_setting_name:
   1744         # Used in MultiIndex.levels to avoid silently ignoring na
-> 1745         raise RuntimeError(
   1746             "Cannot set name on a level of a MultiIndex. Use "
   1747             "'MultiIndex.set_names' instead."
   1748         )
   1749     maybe_extract_name(value, None, type(self))
   1750     self._name = value

RuntimeError: Cannot set name on a level of a MultiIndex. Use 'MultiIndex
```

Use `Index.set_names()` instead.

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 [101]: import random

In [102]: random.shuffle(tuples)

In [103]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tupl

In [104]: s
Out[104]:
baz  two    0.206053
foo  two   -0.251905
bar  one   -2.213588
qux  two    1.063327
baz  one    1.266143
qux  one    0.299368
foo  one   -0.863838
bar  two    0.408204
dtype: float64

In [105]: s.sort_index()
Out[105]:
bar  one   -2.213588
     two    0.408204
baz  one    1.266143
     two    0.206053
foo  one   -0.863838
     two   -0.251905
qux  one    0.299368
     two    1.063327
dtype: float64

In [106]: s.sort_index(level=0)
Out[106]:
bar  one   -2.213588
     two    0.408204
baz  one    1.266143
     two    0.206053
foo  one   -0.863838
     two   -0.251905
qux  one    0.299368
     two    1.063327
dtype: float64

In [107]: s.sort_index(level=1)
Out[107]:
```

```
bar  one   -2.213588
baz  one    1.266143
foo  one   -0.863838
qux  one    0.299368
bar  two    0.408204
baz  two    0.206053
foo  two   -0.251905
qux  two    1.063327
dtype: float64
```

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

```
In [108]: s.index.set_names(["L1", "L2"], inplace=True)

In [109]: s.sort_index(level="L1")
Out[109]:
L1   L2
bar  one   -2.213588
     two    0.408204
baz  one    1.266143
     two    0.206053
foo  one   -0.863838
     two   -0.251905
qux  one    0.299368
     two    1.063327
dtype: float64

In [110]: s.sort_index(level="L2")
Out[110]:
L1   L2
bar  one   -2.213588
baz  one    1.266143
foo  one   -0.863838
qux  one    0.299368
bar  two    0.408204
baz  two    0.206053
foo  two   -0.251905
qux  two    1.063327
dtype: float64
```

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

```
In [111]: df.T.sort_index(level=1, axis=1)
Out[111]:
        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 [112]: dfm = pd.DataFrame(
   .....:     {"jim": [0, 0, 1, 1], "joe": ["x", "x", "z", "y"], "jolie":
   .....: )
   .....:

In [113]: dfm = dfm.set_index(["jim", "joe"])

In [114]: dfm
Out[114]:
            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 d
```

The **`is_monotonic_increasing()`** method on a `MultiIndex` shows if the index is sorted:

```
In [115]: dfm.index.is_monotonic_increasing
Out[115]: False
```

```
In [116]: dfm = dfm.sort_index()

In [117]: dfm
Out[117]: 
            jolie
jim joe          
0   x    0.490671
    x    0.120248
1   y    0.110968
    z    0.537020

In [118]: dfm.index.is_monotonic_increasing
Out[118]: True
```

And now selection works as expected.

```
In [119]: dfm.loc[(0, "y"):(1, "z")]
Out[119]: 
            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 [120]: index = pd.Index(np.random.randint(0, 1000, 10))

In [121]: index
Out[121]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329],

In [122]: positions = [0, 9, 3]

In [123]: index[positions]
Out[123]: Int64Index([214, 329, 567], dtype='int64')
```

```
In [124]: index.take(positions)
Out[124]: Int64Index([214, 329, 567], dtype='int64')

In [125]: ser = pd.Series(np.random.randn(10))

In [126]: ser.iloc[positions]
Out[126]:
0   -0.179666
9    1.824375
3    0.392149
dtype: float64

In [127]: ser.take(positions)
Out[127]:
0   -0.179666
9    1.824375
3    0.392149
dtype: float64
```

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

```
In [128]: frm = pd.DataFrame(np.random.randn(5, 3))

In [129]: frm.take([1, 4, 3])
Out[129]:
          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 [130]: frm.take([0, 2], axis=1)
Out[130]:
          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 [131]: arr = np.random.randn(10)

In [132]: arr.take([False, False, True, True])
Out[132]: array([-1.1935, -1.1935,  0.6775,  0.6775])
```

```
In [133]: arr[[0, 1]]
Out[133]: array([-1.1935,  0.6775])

In [134]: ser = pd.Series(np.random.randn(10))

In [135]: ser.take([False, False, True, True])
Out[135]: 
0    0.233141
0    0.233141
1   -0.223540
1   -0.223540
dtype: float64

In [136]: ser.iloc[[0, 1]]
Out[136]: 
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.

```
In [137]: arr = np.random.randn(10000, 5)

In [138]: indexer = np.arange(10000)

In [139]: random.shuffle(indexer)

In [140]: %timeit arr[indexer]
   .....: %timeit arr.take(indexer, axis=0)
   .....: 
141 us +- 1.18 us per loop (mean +- std. dev. of 7 runs, 10,000 loops eac
43.6 us +- 1.01 us per loop (mean +- std. dev. of 7 runs, 10,000 loops ea
```

```
In [141]: ser = pd.Series(arr[:, 0])

In [142]: %timeit ser.iloc[indexer]
   .....: %timeit ser.take(indexer)
   .....: 
71.3 us +- 2.24 us per loop (mean +- std. dev. of 7 runs, 10,000 loops ea
63.1 us +- 4.29 us per loop (mean +- std. dev. of 7 runs, 10,000 loops ea
```

Index types

Index types

We have discussed `MultiIndex` in the previous sections pretty extensively. Documentation about `DatetimeIndex` and `PeriodIndex` are shown here, and documentation 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 [143]: from pandas.api.types import CategoricalDtype

In [144]: df = pd.DataFrame({"A": np.arange(6), "B": list("aabbca")})

In [145]: df["B"] = df["B"].astype(CategoricalDtype(list("cab")))

In [146]: df
Out[146]: 
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [147]: df.dtypes
Out[147]: 
A       int64
B    category
dtype: object

In [148]: df["B"].cat.categories
Out[148]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a `CategoricalIndex`.

```
In [149]: df2 = df.set_index("B")


In [150]: df2.index
```

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 [151]: df2.loc["a"]
Out[151]:
   A
B
a  0
a  1
a  5
```

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

```
In [152]: df2.loc["a"].index
Out[152]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], c
```

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 [153]: df2.sort_index()
Out[153]:
   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 [154]: df2.groupby(level=0).sum()
Out[154]:
   A
B
c  4
```

```
a   6
b   5

In [155]: df2.groupby(level=0).sum().index
Out[155]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], c
```

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 [156]: df3 = pd.DataFrame(
   .....:     {"A": np.arange(3), "B": pd.Series(list("abc")).astype("cat
   .....: )
   .....:

In [157]: df3 = df3.set_index("B")

In [158]: df3
Out[158]:
   A
B
a  0
b  1
c  2
```

```
In [159]: df3.reindex(["a", "e"])
Out[159]:
     A
B
a  0.0
e  NaN

In [160]: df3.reindex(["a", "e"]).index
Out[160]: Index(['a', 'e'], dtype='object', name='B')

In [161]: df3.reindex(pd.Categorical(["a", "e"], categories=list("abe")))
Out[161]:
     A
B
a  0.0
e  NaN

In [162]: df3.reindex(pd.Categorical(["a", "e"], categories=list("abe")))
Out[162]: CategoricalIndex(['a', 'e'], categories=['a', 'b', 'e'], ordere
```

```
Out[162]: CategoricalIndex(['a', 'c'], categories=['a', 'b', 'c'], ordere
```

⚠ **Warning**

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

```
In [163]: df4 = pd.DataFrame({"A": np.arange(2), "B": list("ba")})

In [164]: df4["B"] = df4["B"].astype(CategoricalDtype(list("ab")))

In [165]: df4 = df4.set_index("B")

In [166]: df4.index
Out[166]: CategoricalIndex(['b', 'a'], categories=['a', 'b'], orc

In [167]: df5 = pd.DataFrame({"A": np.arange(2), "B": list("bc")})

In [168]: df5["B"] = df5["B"].astype(CategoricalDtype(list("bc")))

In [169]: df5 = df5.set_index("B")

In [170]: df5.index
Out[170]: CategoricalIndex(['b', 'c'], categories=['b', 'c'], orc
```

```
In [1]: pd.concat([df4, df5])
TypeError: categories must match existing categories when appendi
```

Int64Index and RangeIndex

❗ ***Deprecated since version 1.4.0:*** In pandas 2.0, `Index` will become the default index type for numeric types instead of `Int64Index`, `Float64Index` and `UInt64Index` and those index types are therefore deprecated and will be removed in a futire version. `RangeIndex` will not be removed, as it represents an optimized version of an integer index.

`Int64Index` is a fundamental basic index in pandas. This is an immutable array implementing an ordered, sliceable set.

`RangeIndex` is a sub-class of `Int64Index` that provides 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

> **Deprecated since version 1.4.0:** `Index` will become the default index type for numeric types in the future instead of `Int64Index`, `Float64Index` and `UInt64Index` and those index types are therefore deprecated and will be removed in a future version of Pandas. `RangeIndex` will not be removed as it represents an optimized version of an integer index.

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

In [172]: indexf
Out[172]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [173]: sf = pd.Series(range(5), index=indexf)

In [174]: sf
Out[174]:
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 [175]: sf[3]
Out[175]: 2

In [176]: sf[3.0]
Out[176]: 2

In [177]: sf.loc[3]
Out[177]: 2

In [178]: sf.loc[3.0]
Out[178]: 2
```

The only positional indexing is via `iloc`.

```
In [179]: sf.iloc[3]
Out[179]: 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 [180]: sf[2:4]
Out[180]:
2.0    1
3.0    2
dtype: int64

In [181]: sf.loc[2:4]
Out[181]:
2.0    1
3.0    2
dtype: int64

In [182]: sf.iloc[2:4]
Out[182]:
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```
In [183]: sf[2.1:4.6]
Out[183]:
```

```
3.0    2
4.5    3
dtype: int64

In [184]: sf.loc[2.1:4.6]
Out[184]:
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 (I

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

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

In [186]: dfir
Out[186]:
               A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
1000.4 -0.179734  0.993962
```

```
1250.5 -0.212673  0.909872
1500.6 -0.733333 -0.349893
1750.7  0.456434 -0.306735

2000.8  0.553396  0.166221
2250.9 -0.101684 -0.734907
```

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

```
In [187]: dfir[0:1000.4]
Out[187]:
               A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
1000.4 -0.179734  0.993962

In [188]: dfir.loc[0:1001, "A"]
Out[188]:
0.0      -0.435772
250.0    -0.808286
500.0    -1.815703
750.0    -0.243487
1000.0    1.162969
1000.4   -0.179734
Name: A, dtype: float64

In [189]: dfir.loc[1000.4]
Out[189]:
A   -0.179734
B    0.993962
Name: 1000.4, dtype: float64
```

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

```
In [190]: dfir[0:1000]
Out[190]:
               A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
```

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

```
In [191]: dfir.iloc[0:5]

Out[191]:
                A         B
0.0     -0.435772 -1.188928
250.0   -0.808286 -0.284634
500.0   -1.815703  1.347213
750.0   -0.243487  0.514704
1000.0   1.162969 -0.287725
```

IntervalIndex

`IntervalIndex` together with its own dtype, `IntervalDtype` 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()`.

Indexing with an `IntervalIndex`

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

```
In [192]: df = pd.DataFrame(
   .....:     {"A": [1, 2, 3, 4]}, index=pd.IntervalIndex.from_breaks([0,
   .....: )
   .....:

In [193]: df
Out[193]:
        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 [194]: df.loc[2]
```

```
Out[194]:
A    2
Name: (1, 2], dtype: int64


In [195]: df.loc[[2, 3]]
Out[195]:
        A
(1, 2]  2
(2, 3]  3
```

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

```
In [196]: df.loc[2.5]
Out[196]:
A    3
Name: (2, 3], dtype: int64

In [197]: df.loc[[2.5, 3.5]]
Out[197]:
        A
(2, 3]  3
(3, 4]  4
```

Selecting using an `Interval` will only return exact matches (starting from pandas 0.25.0).

```
In [198]: df.loc[pd.Interval(1, 2)]
Out[198]:
A    2
Name: (1, 2], dtype: int64
```

Trying to select an `Interval` that is not exactly contained in the `IntervalIndex` will raise a `KeyError`.

```
In [7]: df.loc[pd.Interval(0.5, 2.5)]
---------------------------------------------------------------------------
KeyError: Interval(0.5, 2.5, closed='right')
```

Selecting all `Intervals` that overlap a given `Interval` can be performed using the `overlaps()` method to create a boolean indexer.

```
In [199]: idxr = df.index.overlaps(pd.Interval(0.5, 2.5))

In [200]: idxr
Out[200]: array([ True,  True,  True, False])

In [201]: df[idxr]
Out[201]:
        A
(0, 1]  1
(1, 2]  2
(2, 3]  3
```

Binning data with `cut` and `qcut`

`cut()` and `qcut()` both return a `Categorical` object, and the bins they create are stored as an `IntervalIndex` in its `.categories` attribute.

```
In [202]: c = pd.cut(range(4), bins=2)

In [203]: c
Out[203]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]

In [204]: c.categories
Out[204]: IntervalIndex([(-0.003, 1.5], (1.5, 3.0]], dtype='interval[floa
```

`cut()` also accepts an `IntervalIndex` for its `bins` argument, which enables a useful pandas idiom. First, We call `cut()` with some data and `bins` set to a fixed number, to generate the bins. Then, we pass the values of `.categories` as the `bins` argument in subsequent calls to `cut()`, supplying new data which will be binned into the same bins.

```
In [205]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[205]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64, right]): [(-0.003, 1.5] < (1.5, 3.0]]
```

Any value which falls outside all bins will be assigned a `NaN` value.

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 [206]: pd.interval_range(start=0, end=5)
Out[206]: IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]], dtype='

In [207]: pd.interval_range(start=pd.Timestamp("2017-01-01"), periods=4)
Out[207]: IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-0

In [208]: pd.interval_range(end=pd.Timedelta("3 days"), periods=3)
Out[208]: IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:0
```

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

```
In [209]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[209]: IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0],

In [210]: pd.interval_range(start=pd.Timestamp("2017-01-01"), periods=4,
Out[210]: IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-1

In [211]: pd.interval_range(start=pd.Timedelta("0 days"), periods=3, freq
Out[211]: IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:0
```

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 [212]: pd.interval_range(start=0, end=4, closed="both")
Out[212]: IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]], dtype='interval

In [213]: pd.interval_range(start=0, end=4, closed="neither")
Out[213]: IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)], dtype='interval
```

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 [214]: pd.interval_range(start=0, end=6, periods=4)
Out[214]: IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],

In [215]: pd.interval_range(pd.Timestamp("2018-01-01"), pd.Timestamp("201
Out[215]: IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 0
```

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 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:

```
In [216]: s = pd.Series(range(5))

In [217]: s[-1]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call las
File ~/work/pandas/pandas/pandas/core/indexes/range.py:391, in RangeIndex
    390 try:
--> 391     return self._range.index(new_key)
    392 except ValueError as err:

ValueError: -1 is not in range

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call las
Cell In[217], line 1
----> 1 s[-1]

File ~/work/pandas/pandas/pandas/core/series.py:981, in Series.__getitem_
    978     return self._values[key]
    980 elif key_is_scalar:
--> 981     return self._get_value(key)
    983 if is_hashable(key):
```

```
    984     # Otherwise index.get_value will raise InvalidIndexError
    985     try:
    986         # For labels that don't resolve as scalars like tuples an

File ~/work/pandas/pandas/pandas/core/series.py:1089, in Series._get_valu
   1086     return self._values[label]
   1088 # Similar to Index.get_value, but we do not fall back to position
-> 1089 loc = self.index.get_loc(label)
   1090 return self.index._get_values_for_loc(self, loc, label)

File ~/work/pandas/pandas/pandas/core/indexes/range.py:393, in RangeIndex
    391         return self._range.index(new_key)
    392     except ValueError as err:
--> 393         raise KeyError(key) from err
    394 self._check_indexing_error(key)
    395 raise KeyError(key)

KeyError: -1

In [218]: df = pd.DataFrame(np.random.randn(5, 4))

In [219]: df
Out[219]:
          0         1         2         3
0 -0.130121 -0.476046  0.759104  0.213379
1 -0.082641  0.448008  0.656420 -1.051443
2  0.594956 -0.151360 -0.069303  1.221431
3 -0.182832  0.791235  0.042745  2.069775
4  1.446552  0.019814 -1.389212 -0.702312

In [220]: df.loc[-2:]
Out[220]:
          0         1         2         3
0 -0.130121 -0.476046  0.759104  0.213379
1 -0.082641  0.448008  0.656420 -1.051443
2  0.594956 -0.151360 -0.069303  1.221431
3 -0.182832  0.791235  0.042745  2.069775
4  1.446552  0.019814 -1.389212 -0.702312
```

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 [221]: df = pd.DataFrame(index=[2, 3, 3, 4, 5], columns=["data"], data

In [222]: df.index.is_monotonic_increasing
Out[222]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [223]: df.loc[0:4, :]
Out[223]:
   data
2     0
3     1
3     2
4     3

# slice is are outside the index, so empty DataFrame is returned
In [224]: df.loc[13:15, :]
Out[224]:
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 [225]: df = pd.DataFrame(index=[2, 3, 1, 4, 3, 5], columns=["data"], d

In [226]: df.index.is_monotonic_increasing
Out[226]: False

# OK because 2 and 4 are in the index
In [227]: df.loc[2:4, :]
Out[227]:
   data
2     0
3     1
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 monotonicity, you can combine one of those with the `is_unique()` attribute.

```
In [228]: weakly_monotonic = pd.Index(["a", "b", "c", "c"])

In [229]: weakly_monotonic
Out[229]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [230]: weakly_monotonic.is_monotonic_increasing
Out[230]: True

In [231]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_
Out[231]: 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 [232]: s = pd.Series(np.random.randn(6), index=list("abcdef"))

In [233]: s
Out[233]:
a    0.301379
b    1.240445
c   -0.846068
d   -0.043312
e   -1.658747
f   -0.819549
dtype: float64
```

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

```
In [234]: s[2:5]
Out[234]:
c    -0.846068
d    -0.043312
e    -1.658747
dtype: float64
```

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 choice to make label-based slicing include both endpoints:

```
In [235]: s.loc["c":"e"]
Out[235]:
c    -0.846068
d    -0.043312
e    -1.658747
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 [236]: series1 = pd.Series([1, 2, 3])
```

```
In [237]: series1.dtype
Out[237]: dtype('int64')

In [238]: res = series1.reindex([0, 4])

In [239]: res.dtype
Out[239]: dtype('float64')

In [240]: res
Out[240]:
0    1.0
4    NaN
dtype: float64
```

```
In [241]: series2 = pd.Series([True])

In [242]: series2.dtype
Out[242]: dtype('bool')

In [243]: res = series2.reindex_like(series1)

In [244]: res.dtype
Out[244]: dtype('O')

In [245]: res
Out[245]:
0    True
1     NaN
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 GH2388 for a more detailed discussion.