(continued from previous page)

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

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

In [105]: s.sort_index(level=0)
Out[105]:
bar  one    1.266143
     two    0.408204
baz  one    1.063327
     two   -0.863838
foo  one   -2.213588
     two   -0.251905
qux  one    0.206053
     two    0.299368
dtype: float64

In [106]: s.sort_index(level=1)
Out[106]:
bar  one    1.266143
baz  one    1.063327
foo  one   -2.213588
qux  one    0.206053
bar  two    0.408204
baz  two   -0.863838
foo  two   -0.251905
qux  two    0.299368
dtype: float64
```

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

```
In [107]: s.index.set_names(['L1', 'L2'], inplace=True)

In [108]: s.sort_index(level='L1')
Out[108]:
L1   L2
bar  one    1.266143
     two    0.408204
baz  one    1.063327
     two   -0.863838
foo  one   -2.213588
     two   -0.251905
```

(continues on next page)

(continued from previous page)

```
qux  one    0.206053
     two    0.299368
dtype: float64

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

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

```
In [110]: df.T.sort_index(level=1, axis=1)
Out[110]:
        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 [111]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
   .....:                     'joe': ['x', 'x', 'z', 'y'],
   .....:                     'jolie': np.random.rand(4)})
   .....:

In [112]: dfm = dfm.set_index(['jim', 'joe'])

In [113]: dfm
Out[113]:
             jolie
jim joe
0   x    0.490671
    x    0.120248
1   z    0.537020
    y    0.110968
```

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

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

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

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

The *is_lexsorted()* method on a `MultiIndex` shows if the index is sorted, and the `lexsort_depth` property returns the sort depth:

```
In [114]: dfm.index.is_lexsorted()
Out[114]: False

In [115]: dfm.index.lexsort_depth
Out[115]: 1
```

```
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_lexsorted()
Out[118]: True

In [119]: dfm.index.lexsort_depth
Out[119]: 2
```

And now selection works as expected.

```
In [120]: dfm.loc[(0, 'y'):(1, 'z')]
Out[120]:
             jolie
jim joe
1   y    0.110968
    z    0.537020
```

2.3.4 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 [121]: index = pd.Index(np.random.randint(0, 1000, 10))

In [122]: index
Out[122]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64
↪')

In [123]: positions = [0, 9, 3]

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

(continues on next page)

(continued from previous page)

```
In [125]: index.take(positions)
Out[125]: Int64Index([214, 329, 567], dtype='int64')

In [126]: ser = pd.Series(np.random.randn(10))

In [127]: ser.iloc[positions]
Out[127]:
0   -0.179666
9    1.824375
3    0.392149
dtype: float64

In [128]: ser.take(positions)
Out[128]:
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 [129]: frm = pd.DataFrame(np.random.randn(5, 3))

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

In [133]: arr.take([False, False, True, True])
Out[133]: array([-1.1935, -1.1935,  0.6775,  0.6775])

In [134]: arr[[0, 1]]
Out[134]: array([-1.1935,  0.6775])

In [135]: ser = pd.Series(np.random.randn(10))

In [136]: ser.take([False, False, True, True])
Out[136]:
0    0.233141
0    0.233141
```

(continues on next page)

(continued from previous page)

```
1   -0.223540
1   -0.223540
dtype: float64

In [137]: ser.iloc[[0, 1]]
Out[137]:
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 [138]: arr = np.random.randn(10000, 5)

In [139]: indexer = np.arange(10000)

In [140]: random.shuffle(indexer)

In [141]: %timeit arr[indexer]
   .....: %timeit arr.take(indexer, axis=0)
   .....:
238 us +- 3.95 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
63.2 us +- 2.03 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

```
In [142]: ser = pd.Series(arr[:, 0])

In [143]: %timeit ser.iloc[indexer]
   .....: %timeit ser.take(indexer)
   .....:
200 us +- 15 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
169 us +- 2.1 us per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

2.3.5 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 [144]: from pandas.api.types import CategoricalDtype

In [145]: df = pd.DataFrame({'A': np.arange(6),
   .....:                    'B': list('aabbca')})
   .....:
```

(continues on next page)

(continued from previous page)

```
In [146]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))

In [147]: df
Out[147]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [148]: df.dtypes
Out[148]:
A       int64
B    category
dtype: object

In [149]: df['B'].cat.categories
Out[149]: Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a `CategoricalIndex`.

```
In [150]: df2 = df.set_index('B')

In [151]: df2.index
Out[151]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
↪ ordered=False, name='B', dtype='category')
```

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 [152]: df2.loc['a']
Out[152]:
   A
B
a  0
a  1
a  5
```

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

```
In [153]: df2.loc['a'].index
Out[153]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
↪ name='B', dtype='category')
```

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 [154]: df2.sort_index()
Out[154]:
   A
B
c  4
a  0
a  1
```

(continues on next page)

(continued from previous page)

```
a  5
b  2
b  3
```

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

```
In [155]: df2.groupby(level=0).sum()
Out[155]:
   A
B
c  4
a  6
b  5

In [156]: df2.groupby(level=0).sum().index
Out[156]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False,
 ↪ name='B', dtype='category')
```

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 [157]: df3 = pd.DataFrame({'A': np.arange(3),
   .....:                      'B': pd.Series(list('abc')).astype('category')})
   .....:

In [158]: df3 = df3.set_index('B')

In [159]: df3
Out[159]:
   A
B
a  0
b  1
c  2
```

```
In [160]: df3.reindex(['a', 'e'])
Out[160]:
     A
B
a  0.0
e  NaN

In [161]: df3.reindex(['a', 'e']).index
Out[161]: Index(['a', 'e'], dtype='object', name='B')

In [162]: df3.reindex(pd.Categorical(['a', 'e'], categories=list('abe')))
Out[162]:
     A
B
a  0.0
e  NaN

In [163]: df3.reindex(pd.Categorical(['a', 'e'], categories=list('abe'))).index
Out[163]: CategoricalIndex(['a', 'e'], categories=['a', 'b', 'e'], ordered=False,␣
 ↪name='B', dtype='category')
```

(continues on next page)

(continued from previous page)

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

```
In [164]: df4 = pd.DataFrame({'A': np.arange(2),
   .....:                     'B': list('ba')})
   .....:

In [165]: df4['B'] = df4['B'].astype(CategoricalDtype(list('ab')))

In [166]: df4 = df4.set_index('B')

In [167]: df4.index
Out[167]: CategoricalIndex(['b', 'a'], categories=['a', 'b'], ordered=False, name='B
↪', dtype='category')

In [168]: df5 = pd.DataFrame({'A': np.arange(2),
   .....:                     'B': list('bc')})
   .....:

In [169]: df5['B'] = df5['B'].astype(CategoricalDtype(list('bc')))

In [170]: df5 = df5.set_index('B')

In [171]: df5.index
Out[171]: CategoricalIndex(['b', 'c'], categories=['b', 'c'], ordered=False, name='B
↪', dtype='category')
```

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

Int64Index and RangeIndex

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

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

In [173]: indexf
Out[173]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [174]: sf = pd.Series(range(5), index=indexf)
```

(continues on next page)

(continued from previous page)

```
In [175]: sf
Out[175]:
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 [176]: sf[3]
Out[176]: 2

In [177]: sf[3.0]
Out[177]: 2

In [178]: sf.loc[3]
Out[178]: 2

In [179]: sf.loc[3.0]
Out[179]: 2
```

The only positional indexing is via `iloc`.

```
In [180]: sf.iloc[3]
Out[180]: 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 [181]: sf[2:4]
Out[181]:
2.0    1
3.0    2
dtype: int64

In [182]: sf.loc[2:4]
Out[182]:
2.0    1
3.0    2
dtype: int64

In [183]: sf.iloc[2:4]
Out[183]:
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```
In [184]: sf[2.1:4.6]
Out[184]:
```

(continues on next page)

(continued from previous page)

```
3.0    2
4.5    3
dtype: int64

In [185]: sf.loc[2.1:4.6]
Out[185]:
3.0    2
4.5    3
dtype: int64
```

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

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

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type␣
↪(Int64Index)
```

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

In [187]: dfir
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
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 [188]: dfir[0:1000.4]
Out[188]:
                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
```

(continues on next page)

(continued from previous page)

```
In [189]: dfir.loc[0:1001, 'A']
Out[189]:
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 [190]: dfir.loc[1000.4]
Out[190]:
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 [191]: dfir[0:1000]
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
```

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

```
In [192]: dfir.iloc[0:5]
Out[192]:
                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 [193]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
   .....:                   index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
   .....:

In [194]: df
Out[194]:
         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 [195]: df.loc[2]
Out[195]:
A    2
Name: (1, 2], dtype: int64

In [196]: df.loc[[2, 3]]
Out[196]:
         A
(1, 2]   2
(2, 3]   3
```

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

```
In [197]: df.loc[2.5]
Out[197]:
A    3
Name: (2, 3], dtype: int64

In [198]: df.loc[[2.5, 3.5]]
Out[198]:
         A
(2, 3]   3
(3, 4]   4
```

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

```
In [199]: df.loc[pd.Interval(1, 2)]
Out[199]:
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 [200]: idxr = df.index.overlaps(pd.Interval(0.5, 2.5))

In [201]: idxr
Out[201]: array([ True,  True,  True, False])

In [202]: df[idxr]
Out[202]:
        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 [203]: c = pd.cut(range(4), bins=2)

In [204]: c
Out[204]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]

In [205]: c.categories
Out[205]:
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]],
              closed='right',
              dtype='interval[float64]')
```

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 [206]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[206]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
```

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 [207]: pd.interval_range(start=0, end=5)
Out[207]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right',
              dtype='interval[int64]')
```

(continues on next page)

(continued from previous page)

```
In [208]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4)
Out[208]:
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03], (2017-01-03, 2017-
↪01-04], (2017-01-04, 2017-01-05]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [209]: pd.interval_range(end=pd.Timedelta('3 days'), periods=3)
Out[209]:
IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:00:00, 2 days 00:00:00],
↪ (2 days 00:00:00, 3 days 00:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

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

```
In [210]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[210]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]],
              closed='right',
              dtype='interval[float64]')

In [211]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4, freq='W')
Out[211]:
IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-
↪01-22], (2017-01-22, 2017-01-29]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [212]: pd.interval_range(start=pd.Timedelta('0 days'), periods=3, freq='9H')
Out[212]:
IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days 18:00:00],
↪ (0 days 18:00:00, 1 days 03:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

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

```
In [213]: pd.interval_range(start=0, end=4, closed='both')
Out[213]:
IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]],
              closed='both',
              dtype='interval[int64]')

In [214]: pd.interval_range(start=0, end=4, closed='neither')
Out[214]:
IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)],
              closed='neither',
              dtype='interval[int64]')
```

New in version 0.23.0.

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

```
In [215]: pd.interval_range(start=0, end=6, periods=4)
Out[215]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
              closed='right',
              dtype='interval[float64]')

In [216]: pd.interval_range(pd.Timestamp('2018-01-01'),
   .....:                   pd.Timestamp('2018-02-28'), periods=3)
   .....:
Out[216]:
IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-02-08 16:
↪00:00], (2018-02-08 16:00:00, 2018-02-28]],
              closed='right',
              dtype='interval[datetime64[ns]]')
```

2.3.6 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 [217]: s = pd.Series(range(5))

In [218]: s[-1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-218-76c3dce40054> in <module>
----> 1 s[-1]

/pandas-release/pandas/pandas/core/series.py in __getitem__(self, key)
    869         key = com.apply_if_callable(key, self)
    870         try:
--> 871             result = self.index.get_value(self, key)
    872
    873             if not is_scalar(result):

/pandas-release/pandas/pandas/core/indexes/base.py in get_value(self, series, key)
   4403         k = self._convert_scalar_indexer(k, kind="getitem")
   4404         try:
-> 4405             return self._engine.get_value(s, k, tz=getattr(series.dtype, "tz",
↪ None))
   4406         except KeyError as e1:
   4407             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_
↪value()

/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_
↪value()

/pandas-release/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_
↪loc()
```

(continues on next page)

(continued from previous page)

```
/pandas-release/pandas/pandas/_libs/hashtable_class_helper.pxi in pandas._libs.
↪hashtable.Int64HashTable.get_item()

/pandas-release/pandas/pandas/_libs/hashtable_class_helper.pxi in pandas._libs.
↪hashtable.Int64HashTable.get_item()

KeyError: -1

In [219]: df = pd.DataFrame(np.random.randn(5, 4))

In [220]: df
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

In [221]: df.loc[-2:]
Out[221]:
          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 [222]: df = pd.DataFrame(index=[2, 3, 3, 4, 5], columns=['data'],␣
↪data=list(range(5)))

In [223]: df.index.is_monotonic_increasing
Out[223]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [224]: df.loc[0:4, :]
Out[224]:
   data
2     0
3     1
3     2
4     3

# slice is are outside the index, so empty DataFrame is returned
In [225]: df.loc[13:15, :]
```

(continues on next page)

(continued from previous page)

```
Out[225]:
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 [226]: df = pd.DataFrame(index=[2, 3, 1, 4, 3, 5],
   .....:                   columns=['data'], data=list(range(6)))
   .....:

In [227]: df.index.is_monotonic_increasing
Out[227]: False

# OK because 2 and 4 are in the index
In [228]: df.loc[2:4, :]
Out[228]:
   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 [229]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])

In [230]: weakly_monotonic
Out[230]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [231]: weakly_monotonic.is_monotonic_increasing
Out[231]: True

In [232]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_unique
Out[232]: 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 [233]: s = pd.Series(np.random.randn(6), index=list('abcdef'))

In [234]: s
Out[234]:
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 [235]: s[2:5]
Out[235]:
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 [236]: s.loc['c':'e']
Out[236]:
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 [237]: series1 = pd.Series([1, 2, 3])

In [238]: series1.dtype
Out[238]: dtype('int64')

In [239]: res = series1.reindex([0, 4])
```

(continues on next page)

(continued from previous page)

```
In [240]: res.dtype
Out[240]: dtype('float64')

In [241]: res
Out[241]:
0    1.0
4    NaN
dtype: float64
```

```
In [242]: series2 = pd.Series([True])

In [243]: series2.dtype
Out[243]: dtype('bool')

In [244]: res = series2.reindex_like(series1)

In [245]: res.dtype
Out[245]: dtype('O')

In [246]: res
Out[246]:
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 this old issue for a more detailed discussion.

2.4 Merge, join, and concatenate

pandas provides various facilities for easily combining together Series or DataFrame with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

2.4.1 Concatenating objects

The *concat()* function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say "if any" because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```
In [1]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ...:                     'B': ['B0', 'B1', 'B2', 'B3'],
   ...:                     'C': ['C0', 'C1', 'C2', 'C3'],
   ...:                     'D': ['D0', 'D1', 'D2', 'D3']},
   ...:                    index=[0, 1, 2, 3])
   ...:

In [2]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
   ...:                     'B': ['B4', 'B5', 'B6', 'B7'],
```

(continues on next page)

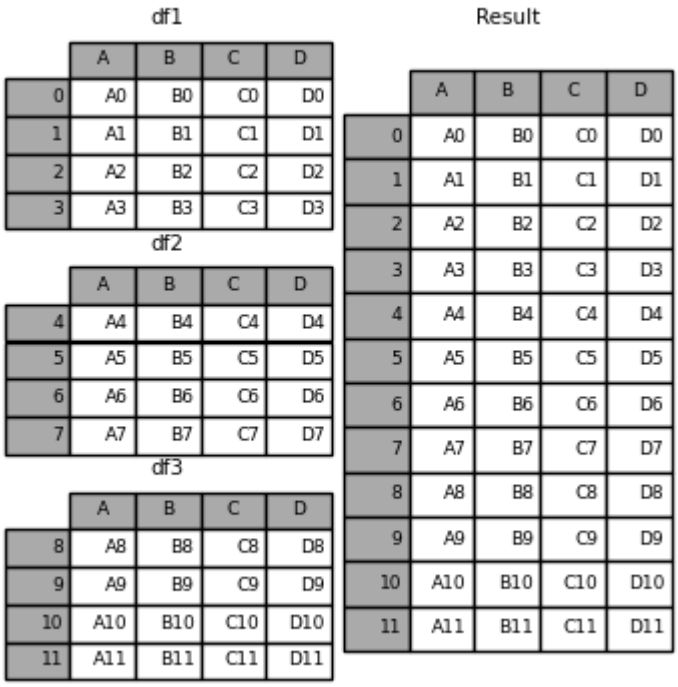
(continued from previous page)

```
   ...:                      'C': ['C4', 'C5', 'C6', 'C7'],
   ...:                      'D': ['D4', 'D5', 'D6', 'D7']},
   ...:                     index=[4, 5, 6, 7])
   ...:

In [3]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
   ...:                      'B': ['B8', 'B9', 'B10', 'B11'],
   ...:                      'C': ['C8', 'C9', 'C10', 'C11'],
   ...:                      'D': ['D8', 'D9', 'D10', 'D11']},
   ...:                     index=[8, 9, 10, 11])
   ...:

In [4]: frames = [df1, df2, df3]

In [5]: result = pd.concat(frames)
```



Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of "what to do with the other axes":

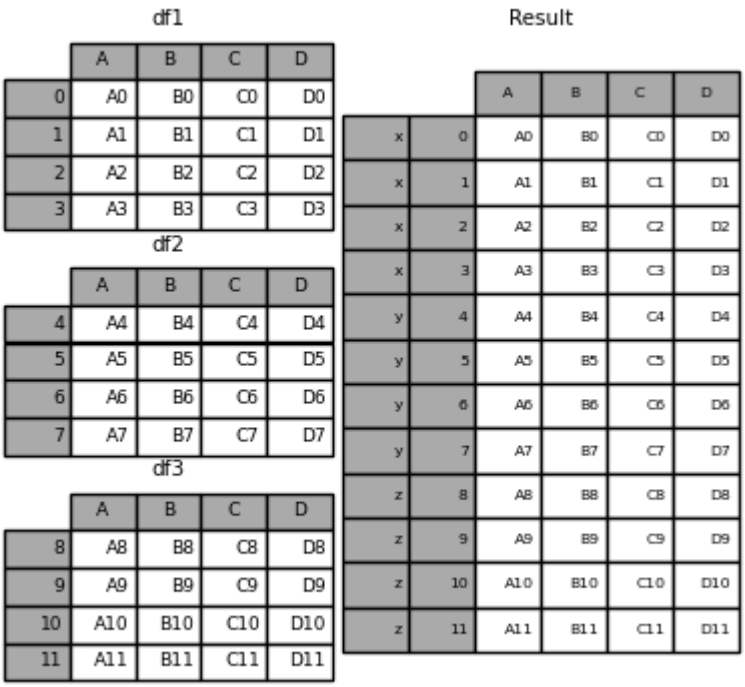
```
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
          levels=None, names=None, verify_integrity=False, copy=True)
```

- `objs` : a sequence or mapping of Series or DataFrame objects. If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.
- `axis` : {0, 1, ...}, default 0. The axis to concatenate along.

- `join` : {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- `ignore_index` : boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, . . . , n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.
- `keys` : sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.
- `levels` : list of sequences, default None. Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.
- `names` : list, default None. Names for the levels in the resulting hierarchical index.
- `verify_integrity` : boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.
- `copy` : boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments don't make much sense. Let's revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the `keys` argument:

```
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```



As you can see (if you've read the rest of the documentation), the resulting object's index has a *hierarchical index*. This means that we can now select out each chunk by key:

```
In [7]: result.loc['y']
Out[7]:
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

Note: It is worth noting that *`concat()`* (and therefore `append()`) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following two ways:

- Take the union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.

Here is an example of each of these methods. First, the default `join='outer'` behavior:

```
In [8]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
   ...:                     'D': ['D2', 'D3', 'D6', 'D7'],
   ...:                     'F': ['F2', 'F3', 'F6', 'F7']},
   ...:                    index=[2, 3, 6, 7])
   ...:

In [9]: result = pd.concat([df1, df4], axis=1, sort=False)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

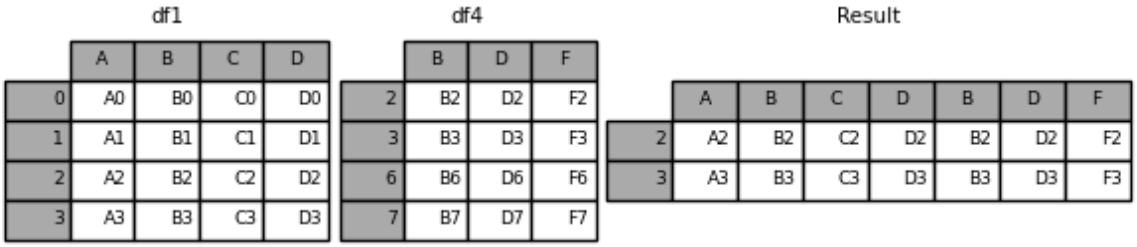
	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Warning: Changed in version 0.23.0.

The default behavior with `join='outer'` is to sort the other axis (columns in this case). In a future version of pandas, the default will be to not sort. We specified `sort=False` to opt in to the new behavior now.

Here is the same thing with `join='inner'`:

```
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```

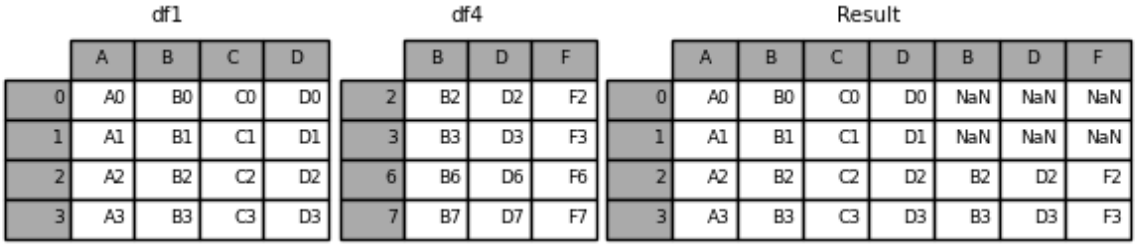


Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

```
In [11]: result = pd.concat([df1, df4], axis=1).reindex(df1.index)
```

Similarly, we could index before the concatenation:

```
In [12]: pd.concat([df1, df4.reindex(df1.index)], axis=1)
Out[12]:
    A   B   C   D    B    D    F
0  A0  B0  C0  D0  NaN  NaN  NaN
1  A1  B1  C1  D1  NaN  NaN  NaN
2  A2  B2  C2  D2   B2   D2   F2
3  A3  B3  C3  D3   B3   D3   F3
```



Concatenating using `append`

A useful shortcut to *`concat()`* are the *`append()`* instance methods on `Series` and `DataFrame`. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

```
In [13]: result = df1.append(df2)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In the case of `DataFrame`, the indexes must be disjoint but the columns do not need to be:

```
In [14]: result = df1.append(df4, sort=False)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

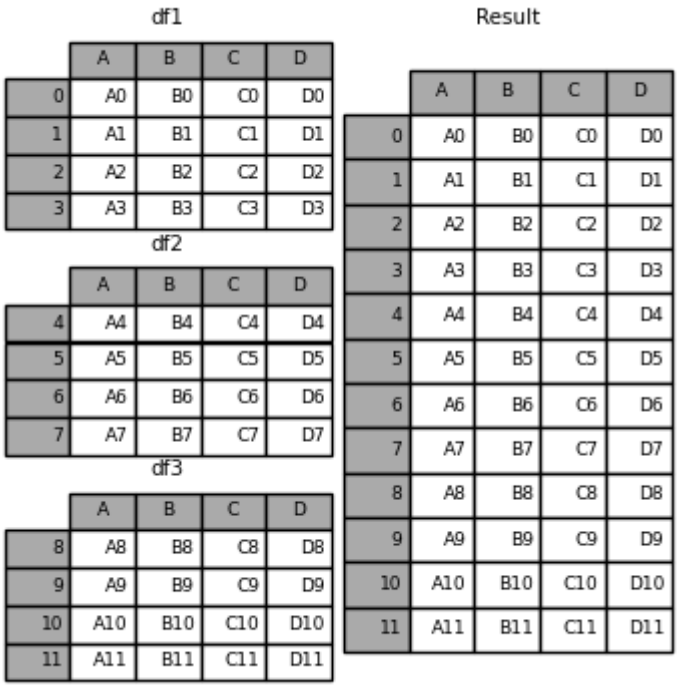
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

`append` may take multiple objects to concatenate:

```
In [15]: result = df1.append([df2, df3])
```



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Note: Unlike the `append()` method, which appends to the original list and returns `None`, *`append()`* here **does not** modify `df1` and returns its copy with `df2` appended.

Ignoring indexes on the concatenation axis

For `DataFrame` objects which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the `ignore_index` argument:

```
In [16]: result = pd.concat([df1, df4], ignore_index=True, sort=False)
```