Examples

```
>>> index = pd.IntervalIndex.from_tuples([(0, 2), (1, 3), (4, 5)])
>>> index
IntervalIndex([(0, 2], (1, 3], (4, 5]],
      closed='right',
      dtype='interval[int64]')
>>> index.is_overlapping
True
```

Intervals that share closed endpoints overlap:

```
>>> index = pd.interval_range(0, 3, closed='both')
>>> index
IntervalIndex([[0, 1], [1, 2], [2, 3]],
      closed='both',
      dtype='interval[int64]')
>>> index.is_overlapping
True
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> index = pd.interval_range(0, 3, closed='left')
>>> index
IntervalIndex([[0, 1), [1, 2), [2, 3)],
      closed='left',
      dtype='interval[int64]')
>>> index.is_overlapping
False
```

pandas.IntervalIndex.values

`IntervalIndex.`**`values`**
Return the IntervalIndex's data as an IntervalArray.

Methods

from_arrays(left, right, closed[, name, dtype])	Construct from two arrays defining the left and right bounds.
from_tuples(data, closed[, name, dtype])	Construct an IntervalIndex from an array-like of tuples.
from_breaks(breaks, closed[, name, dtype])	Construct an IntervalIndex from an array of splits.
contains(self, *args, **kwargs)	Check elementwise if the Intervals contain the value.
overlaps(self, *args, **kwargs)	Check elementwise if an Interval overlaps the values in the IntervalArray.
set_closed(self, *args, **kwargs)	Return an IntervalArray identical to the current one, but closed on the specified side.
to_tuples(self, *args, **kwargs)	Return an ndarray of tuples of the form (left, right).

pandas.IntervalIndex.from_arrays

classmethod `IntervalIndex.`**`from_arrays`**(*left*, *right*, *closed: str = 'right'*, *name=None*, *copy: bool = False*, *dtype=None*)

Construct from two arrays defining the left and right bounds.

Parameters

left [array-like (1-dimensional)] Left bounds for each interval.

right [array-like (1-dimensional)] Right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [bool, default False] Copy the data.

dtype [dtype, optional] If None, dtype will be inferred.

New in version 0.23.0.

Returns

IntervalIndex

Raises

ValueError When a value is missing in only one of *left* or *right*. When a value in *left* is greater than the corresponding value in *right*.

See also:

`interval_range` Function to create a fixed frequency IntervalIndex.

`IntervalIndex.from_breaks` Construct an IntervalIndex from an array of splits.

`IntervalIndex.from_tuples` Construct an IntervalIndex from an array-like of tuples.

Notes

Each element of *left* must be less than or equal to the *right* element at the same position. If an element is missing, it must be missing in both *left* and *right*. A TypeError is raised when using an unsupported type for *left* or *right*. At the moment, 'category', 'object', and 'string' subtypes are not supported.

Examples

```
>>> pd.IntervalIndex.from_arrays([0, 1, 2], [1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
              closed='right',
              dtype='interval[int64]')
```

pandas.IntervalIndex.from_tuples

classmethod `IntervalIndex.`**from_tuples**(*data*, *closed: str = 'right'*, *name=None*, *copy: bool = False*, *dtype=None*)

Construct an IntervalIndex from an array-like of tuples.

Parameters

data [array-like (1-dimensional)] Array of tuples.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [bool, default False] By-default copy the data, this is compat only and ignored.

dtype [dtype or None, default None] If None, dtype will be inferred.

New in version 0.23.0.

Returns

IntervalIndex

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalIndex.from_arrays Construct an IntervalIndex from a left and right array.

IntervalIndex.from_breaks Construct an IntervalIndex from an array of splits.

Examples

```
>>> pd.IntervalIndex.from_tuples([(0, 1), (1, 2)])
IntervalIndex([(0, 1], (1, 2]],
               closed='right',
               dtype='interval[int64]')
```

pandas.IntervalIndex.from_breaks

classmethod `IntervalIndex.`**from_breaks**(*breaks*, *closed: str = 'right'*, *name=None*, *copy: bool = False*, *dtype=None*)

Construct an IntervalIndex from an array of splits.

Parameters

breaks [array-like (1-dimensional)] Left and right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [bool, default False] Copy the data.

dtype [dtype or None, default None] If None, dtype will be inferred.

New in version 0.23.0.

Returns

IntervalIndex

See also:

interval_range	Function to create a fixed frequency IntervalIndex.
IntervalIndex.from_arrays	Construct from a left and right array.
IntervalIndex.from_tuples	Construct from a sequence of tuples.

Examples

```
>>> pd.IntervalIndex.from_breaks([0, 1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
              closed='right',
              dtype='interval[int64]')
```

pandas.IntervalIndex.contains

IntervalIndex.**contains**(*self*, **args*, ***kwargs*)

Check elementwise if the Intervals contain the value.

Return a boolean mask whether the value is contained in the Intervals of the IntervalArray.

New in version 0.25.0.

Parameters

other [scalar] The value to check whether it is contained in the Intervals.

Returns

boolean array

See also:

Interval.contains	Check whether Interval object contains value.
IntervalArray.overlaps	Check if an Interval overlaps the values in the IntervalArray.

Examples

```
>>> intervals = pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 3), (2, 4)])
>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]]
Length: 3, closed: right, dtype: interval[int64]
```

```
>>> intervals.contains(0.5)
array([ True, False, False])
```

pandas.IntervalIndex.overlaps

`IntervalIndex.`**`overlaps`**`(`*self*, **args*, ***kwargs*`)`

Check elementwise if an Interval overlaps the values in the IntervalArray.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

New in version 0.24.0.

Parameters

other [IntervalArray] Interval to check against for an overlap.

Returns

ndarray Boolean array positionally indicating where an overlap occurs.

See also:

`Interval.overlaps` Check whether two Interval objects overlap.

Examples

```
>>> data = [(0, 1), (1, 3), (2, 4)]
>>> intervals = pd.arrays.IntervalArray.from_tuples(data)
>>> intervals
<IntervalArray>
[(0, 1], (1, 3], (2, 4]]
Length: 3, closed: right, dtype: interval[int64]
```

```
>>> intervals.overlaps(pd.Interval(0.5, 1.5))
array([ True,  True, False])
```

Intervals that share closed endpoints overlap:

```
>>> intervals.overlaps(pd.Interval(1, 3, closed='left'))
array([ True,  True, True])
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> intervals.overlaps(pd.Interval(1, 2, closed='right'))
array([False,  True, False])
```

pandas.IntervalIndex.set_closed

`IntervalIndex.`**`set_closed`**`(`*self*, **args*, ***kwargs*`)`

Return an IntervalArray identical to the current one, but closed on the specified side.

New in version 0.24.0.

Parameters

closed [{'left', 'right', 'both', 'neither'}] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

new_index [IntervalArray]

Examples

```
>>> index = pd.arrays.IntervalArray.from_breaks(range(4))
>>> index
<IntervalArray>
[(0, 1], (1, 2], (2, 3]]
Length: 3, closed: right, dtype: interval[int64]
>>> index.set_closed('both')
<IntervalArray>
[[0, 1], [1, 2], [2, 3]]
Length: 3, closed: both, dtype: interval[int64]
```

pandas.IntervalIndex.to_tuples

`IntervalIndex.`**`to_tuples`**(*self*, **args*, ***kwargs*)

Return an ndarray of tuples of the form (left, right).

Parameters

na_tuple [boolean, default True] Returns NA as a tuple if True, `(nan, nan)`, or just as the NA value itself if False, `nan`.

New in version 0.23.0.

Returns

tuples: ndarray

IntervalIndex components

IntervalIndex.from_arrays(left, right, closed)	Construct from two arrays defining the left and right bounds.
IntervalIndex.from_tuples(data, closed[, …])	Construct an IntervalIndex from an array-like of tuples.
IntervalIndex.from_breaks(breaks, closed[, …])	Construct an IntervalIndex from an array of splits.
IntervalIndex.left	Return the left endpoints of each Interval in the IntervalArray as an Index.
IntervalIndex.right	Return the right endpoints of each Interval in the IntervalArray as an Index.
IntervalIndex.mid	Return the midpoint of each Interval in the IntervalArray as an Index.
IntervalIndex.closed	Whether the intervals are closed on the left-side, right-side, both or neither.
IntervalIndex.length	Return an Index with entries denoting the length of each Interval in the IntervalArray.
IntervalIndex.values	Return the IntervalIndex's data as an IntervalArray.
IntervalIndex.is_empty	Indicates if an interval is empty, meaning it contains no points.

continues on next page

Table 152 – continued from previous page

IntervalIndex.is_non_overlapping_monotonic	Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False.
IntervalIndex.is_overlapping	Return True if the IntervalIndex has overlapping intervals, else False.
IntervalIndex.get_loc(self, key, method, . . .)	Get integer location, slice or boolean mask for requested label.
IntervalIndex.get_indexer(self, target, . . .)	Compute indexer and mask for new index given the current index.
IntervalIndex.set_closed(self, *args, **kwargs)	Return an IntervalArray identical to the current one, but closed on the specified side.
IntervalIndex.contains(self, *args, **kwargs)	Check elementwise if the Intervals contain the value.
IntervalIndex.overlaps(self, *args, **kwargs)	Check elementwise if an Interval overlaps the values in the IntervalArray.
IntervalIndex.to_tuples(self, *args, **kwargs)	Return an ndarray of tuples of the form (left, right).

pandas.IntervalIndex.get_loc

IntervalIndex.**get_loc**(*self*, *key: Any*, *method: Union[str, NoneType] = None*, *tolerance=None*) → Union[int, slice, numpy.ndarray]

Get integer location, slice or boolean mask for requested label.

Parameters

key [label]

method [{None}, optional]

- default: matches where the label is within an interval only.

Returns

int if unique index, slice if monotonic index, else mask

Examples

```
>>> i1, i2 = pd.Interval(0, 1), pd.Interval(1, 2)
>>> index = pd.IntervalIndex([i1, i2])
>>> index.get_loc(1)
0
```

You can also supply a point inside an interval.

```
>>> index.get_loc(1.5)
1
```

If a label is in several intervals, you get the locations of all the relevant intervals.

```
>>> i3 = pd.Interval(0, 2)
>>> overlapping_index = pd.IntervalIndex([i1, i2, i3])
>>> overlapping_index.get_loc(0.5)
array([ True, False,  True])
```

Only exact matches will be returned if an interval is provided.

```
>>> index.get_loc(pd.Interval(0, 1))
0
```

pandas.IntervalIndex.get_indexer

`IntervalIndex.`**`get_indexer`**(*self*, *target: ~ AnyArrayLike*, *method: Union[str, NoneType] = None*, *limit: Union[int, NoneType] = None*, *tolerance: Union[Any, NoneType] = None*) → numpy.ndarray

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [IntervalIndex or list of Intervals]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit [int, optional] Maximum number of consecutive labels in `target` to match for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Raises

NotImplementedError If any method argument other than the default of None is specified as these are not yet implemented.

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

3.7.5 MultiIndex

MultiIndex([levels, codes, sortorder, . . .])	A multi-level, or hierarchical, index object for pandas objects.

pandas.MultiIndex

class pandas.**MultiIndex**(*levels=None*, *codes=None*, *sortorder=None*, *names=None*, *dtype=None*, *copy=False*, *name=None*, *verify_integrity: bool = True*, *_set_identity: bool = True*)

A multi-level, or hierarchical, index object for pandas objects.

Parameters

levels [sequence of arrays] The unique labels for each level.

codes [sequence of arrays] Integers for each level designating which label at each location.

New in version 0.24.0.

sortorder [optional int] Level of sortedness (must be lexicographically sorted by that level).

names [optional sequence of objects] Names for each of the index levels. (name is accepted for compat).

copy [bool, default False] Copy the meta-data.

verify_integrity [bool, default True] Check that the levels/codes are consistent and valid.

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.
MultiIndex.from_product Create a MultiIndex from the cartesian product of iterables.
MultiIndex.from_tuples Convert list of tuples to a MultiIndex.
MultiIndex.from_frame Make a MultiIndex from a DataFrame.
Index The base pandas Index type.

Notes

See the user guide for more.

Examples

A new `MultiIndex` is typically constructed using one of the helper methods *`MultiIndex.from_arrays()`*, *`MultiIndex.from_product()`* and *`MultiIndex.from_tuples()`*. For example (using `.from_arrays`):

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex([(1,  'red'),
            (1, 'blue'),
            (2,  'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

See further examples for how to construct a MultiIndex in the doc strings of the mentioned helper methods.

Attributes

`names`	Names of levels in MultiIndex.
`nlevels`	Integer number of levels in this MultiIndex.
`levshape`	A tuple with the length of each level.

pandas.MultiIndex.names

`property` `MultiIndex.`**`names`**
Names of levels in MultiIndex.

pandas.MultiIndex.nlevels

`property` `MultiIndex.`**`nlevels`**
Integer number of levels in this MultiIndex.

pandas.MultiIndex.levshape

`property` `MultiIndex.`**`levshape`**
A tuple with the length of each level.

levels	
codes	

Methods

from_arrays(arrays[, sortorder, names])	Convert arrays to MultiIndex.
from_tuples(tuples[, sortorder, names])	Convert list of tuples to MultiIndex.
from_product(iterables[, sortorder, names])	Make a MultiIndex from the cartesian product of multiple iterables.
from_frame(df[, sortorder, names])	Make a MultiIndex from a DataFrame.
set_levels(self, levels[, level, inplace, ...])	Set new levels on MultiIndex.
set_codes(self, codes[, level, inplace, ...])	Set new codes on MultiIndex.
to_frame(self[, index, name])	Create a DataFrame with the levels of the MultiIndex as columns.
to_flat_index(self)	Convert a MultiIndex to an Index of Tuples containing the level values.
is_lexsorted(self)	Return True if the codes are lexicographically sorted.
sortlevel(self[, level, ascending, ...])	Sort MultiIndex at the requested level.
droplevel(self[, level])	Return index with requested level(s) removed.
swaplevel(self[, i, j])	Swap level i with level j.
reorder_levels(self, order)	Rearrange levels using input order.
remove_unused_levels(self)	Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.
get_locs(self, seq)	Get location for a sequence of labels.

pandas.MultiIndex.from_arrays

classmethod MultiIndex.**from_arrays**(*arrays*, *sortorder=None*, *names=<object object at 0x7f5374a06320>*)

Convert arrays to MultiIndex.

Parameters

arrays [list / sequence of array-likes] Each array-like gives one level's value for each data point. len(arrays) is the number of levels.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Returns

MultiIndex

See also:

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex([(1,  'red'),
            (1, 'blue'),
            (2,  'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

pandas.MultiIndex.from_tuples

classmethod MultiIndex.**from_tuples**(*tuples*, *sortorder=None*, *names=None*)

Convert list of tuples to MultiIndex.

Parameters

tuples [list / sequence of tuple-likes] Each tuple is the index of one row/column.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Returns

MultiIndex

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> tuples = [(1, 'red'), (1, 'blue'),
...           (2, 'red'), (2, 'blue')]
>>> pd.MultiIndex.from_tuples(tuples, names=('number', 'color'))
MultiIndex([(1,  'red'),
            (1, 'blue'),
            (2,  'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

pandas.MultiIndex.from_product

classmethod MultiIndex.**from_product**(*iterables*, *sortorder=None*, *names=<object object at 0x7f5374a06320>*)

Make a MultiIndex from the cartesian product of multiple iterables.

Parameters

iterables [list / sequence of iterables] Each iterable has unique labels for each level of the index.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Changed in version 1.0.0: If not explicitly provided, names will be inferred from the elements of iterables if an element has a name attribute

Returns

MultiIndex

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> numbers = [0, 1, 2]
>>> colors = ['green', 'purple']
>>> pd.MultiIndex.from_product([numbers, colors],
...                            names=['number', 'color'])
MultiIndex([(0,  'green'),
            (0, 'purple'),
            (1,  'green'),
            (1, 'purple'),
            (2,  'green'),
            (2, 'purple')],
           names=['number', 'color'])
```

pandas.MultiIndex.from_frame

classmethod MultiIndex.**from_frame**(*df*, *sortorder=None*, *names=None*)

Make a MultiIndex from a DataFrame.

New in version 0.24.0.

Parameters

df [DataFrame] DataFrame to be converted to MultiIndex.

sortorder [int, optional] Level of sortedness (must be lexicographically sorted by that level).

names [list-like, optional] If no names are provided, use the column names, or tuple of column names if the columns is a MultiIndex. If a sequence, overwrite names with the given sequence.

Returns

MultiIndex The MultiIndex representation of the given DataFrame.

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

Examples

```
>>> df = pd.DataFrame([['HI', 'Temp'], ['HI', 'Precip'],
...                    ['NJ', 'Temp'], ['NJ', 'Precip']],
...                   columns=['a', 'b'])
>>> df
      a       b
0    HI    Temp
1    HI  Precip
2    NJ    Temp
3    NJ  Precip
```

```
>>> pd.MultiIndex.from_frame(df)
MultiIndex([('HI',   'Temp'),
            ('HI', 'Precip'),
            ('NJ',   'Temp'),
            ('NJ', 'Precip')],
           names=['a', 'b'])
```

Using explicit names, instead of the column names

```
>>> pd.MultiIndex.from_frame(df, names=['state', 'observation'])
MultiIndex([('HI',   'Temp'),
            ('HI', 'Precip'),
            ('NJ',   'Temp'),
            ('NJ', 'Precip')],
           names=['state', 'observation'])
```

pandas.MultiIndex.set_levels

MultiIndex.**set_levels**(*self*, *levels*, *level=None*, *inplace=False*, *verify_integrity=True*)

Set new levels on MultiIndex. Defaults to returning new index.

Parameters

levels [sequence or list of sequence] New level(s) to apply.

level [int, level name, or sequence of int/level names (default None)] Level(s) to set (None for all levels).

inplace [bool] If True, mutates in place.

verify_integrity [bool, default True] If True, checks that levels and codes are compatible.

Returns

new index (of same type and class. . . etc)

Examples

```
>>> idx = pd.MultiIndex.from_tuples([(1, 'one'), (1, 'two'),
                                    (2, 'one'), (2, 'two'),
                                    (3, 'one'), (3, 'two')],
                                   names=['foo', 'bar'])
>>> idx.set_levels([['a', 'b', 'c'], [1, 2]])
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2),
            ('c', 1),
            ('c', 2)],
           names=['foo', 'bar'])
>>> idx.set_levels(['a', 'b', 'c'], level=0)
MultiIndex([('a', 'one'),
            ('a', 'two'),
            ('b', 'one'),
            ('b', 'two'),
            ('c', 'one'),
            ('c', 'two')],
           names=['foo', 'bar'])
>>> idx.set_levels(['a', 'b'], level='bar')
MultiIndex([(1, 'a'),
            (1, 'b'),
            (2, 'a'),
            (2, 'b'),
            (3, 'a'),
            (3, 'b')],
           names=['foo', 'bar'])
```

If any of the levels passed to `set_levels()` exceeds the existing length, all of the values from that argument will be stored in the MultiIndex levels, though the values will be truncated in the MultiIndex output.

```
>>> idx.set_levels([['a', 'b', 'c'], [1, 2, 3, 4]], level=[0, 1])
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           names=['foo', 'bar'])
>>> idx.set_levels([['a', 'b', 'c'], [1, 2, 3, 4]], level=[0, 1]).levels
FrozenList([['a', 'b', 'c'], [1, 2, 3, 4]])
```

pandas.MultiIndex.set_codes

`MultiIndex.`**`set_codes`**(*self*, *codes*, *level=None*, *inplace=False*, *verify_integrity=True*)

Set new codes on MultiIndex. Defaults to returning new index.

New in version 0.24.0: New name for deprecated method *set_labels*.

Parameters

codes [sequence or list of sequence] New codes to apply.

level [int, level name, or sequence of int/level names (default None)] Level(s) to set (None for all levels).

inplace [bool] If True, mutates in place.

verify_integrity [bool (default True)] If True, checks that levels and codes are compatible.

Returns

new index (of same type and class…etc)

Examples

```
>>> idx = pd.MultiIndex.from_tuples([(1, 'one'),
                                     (1, 'two'),
                                     (2, 'one'),
                                     (2, 'two')],
                                    names=['foo', 'bar'])
>>> idx.set_codes([[1, 0, 1, 0], [0, 0, 1, 1]])
MultiIndex([(2, 'one'),
            (1, 'one'),
            (2, 'two'),
            (1, 'two')],
           names=['foo', 'bar'])
>>> idx.set_codes([1, 0, 1, 0], level=0)
MultiIndex([(2, 'one'),
            (1, 'two'),
            (2, 'one'),
            (1, 'two')],
           names=['foo', 'bar'])
>>> idx.set_codes([0, 0, 1, 1], level='bar')
MultiIndex([(1, 'one'),
            (1, 'one'),
            (2, 'two'),
            (2, 'two')],
           names=['foo', 'bar'])
>>> idx.set_codes([[1, 0, 1, 0], [0, 0, 1, 1]], level=[0, 1])
MultiIndex([(2, 'one'),
            (1, 'one'),
            (2, 'two'),
            (1, 'two')],
           names=['foo', 'bar'])
```

pandas.MultiIndex.to_frame

`MultiIndex.`**`to_frame`**(*self*, *index=True*, *name=None*)

Create a DataFrame with the levels of the MultiIndex as columns.

Column ordering is determined by the DataFrame constructor with data as a dict.

New in version 0.24.0.

Parameters

index [bool, default True] Set the index of the returned DataFrame as the original MultiIndex.

name [list / sequence of strings, optional] The passed names should substitute index level names.

Returns

DataFrame [a DataFrame containing the original MultiIndex data.]

See also:

`DataFrame`

pandas.MultiIndex.to_flat_index

`MultiIndex.`**`to_flat_index`**(*self*)

Convert a MultiIndex to an Index of Tuples containing the level values.

New in version 0.24.0.

Returns

pd.Index Index with the MultiIndex data represented in Tuples.

Notes

This method will simply return the caller if called by anything other than a MultiIndex.

Examples

```
>>> index = pd.MultiIndex.from_product(
...     [['foo', 'bar'], ['baz', 'qux']],
...     names=['a', 'b'])
>>> index.to_flat_index()
Index([('foo', 'baz'), ('foo', 'qux'),
       ('bar', 'baz'), ('bar', 'qux')],
      dtype='object')
```

pandas.MultiIndex.is_lexsorted

`MultiIndex.`**`is_lexsorted`**`(`*self*`)` → bool

Return True if the codes are lexicographically sorted.

Returns

bool

pandas.MultiIndex.sortlevel

`MultiIndex.`**`sortlevel`**`(`*self*, *level=0*, *ascending=True*, *sort_remaining=True*`)`

Sort MultiIndex at the requested level. The result will respect the original ordering of the associated factor at that level.

Parameters

level [list-like, int or str, default 0] If a string is given, must be a name of the level. If list-like must be names or ints of levels.

ascending [bool, default True] False to sort in descending order. Can also be a list to specify a directed ordering.

sort_remaining [sort by the remaining levels after level]

Returns

sorted_index [pd.MultiIndex] Resulting index.

indexer [np.ndarray] Indices of output values in original index.

pandas.MultiIndex.droplevel

`MultiIndex.`**`droplevel`**`(`*self*, *level=0*`)`

Return index with requested level(s) removed.

If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

New in version 0.23.1: (support for non-MultiIndex)

Parameters

level [int, str, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

Returns

Index or MultiIndex

pandas.MultiIndex.swaplevel

`MultiIndex.swaplevel(self, i=- 2, j=- 1)`
Swap level i with level j.

Calling this method does not change the ordering of the values.

Parameters

i [int, str, default -2] First level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

j [int, str, default -1] Second level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

Returns

MultiIndex A new MultiIndex.

See also:

`Series.swaplevel` Swap levels i and j in a MultiIndex.

`Dataframe.swaplevel` Swap levels i and j in a MultiIndex on a particular axis.

Examples

```
>>> mi = pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
...                    codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi
MultiIndex([('a', 'bb'),
            ('a', 'aa'),
            ('b', 'bb'),
            ('b', 'aa')],
           )
>>> mi.swaplevel(0, 1)
MultiIndex([('bb', 'a'),
            ('aa', 'a'),
            ('bb', 'b'),
            ('aa', 'b')],
           )
```

pandas.MultiIndex.reorder_levels

`MultiIndex.reorder_levels(self, order)`
Rearrange levels using input order. May not drop or duplicate levels.

Returns

MultiIndex

pandas.MultiIndex.remove_unused_levels

MultiIndex.**remove_unused_levels**(*self*)

Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.

The resulting MultiIndex will have the same outward appearance, meaning the same .values and ordering. It will also be .equals() to the original.

Returns

MultiIndex

Examples

```
>>> mi = pd.MultiIndex.from_product([range(2), list('ab')])
>>> mi
MultiIndex([(0, 'a'),
            (0, 'b'),
            (1, 'a'),
            (1, 'b')],
           )
```

```
>>> mi[2:]
MultiIndex([(1, 'a'),
            (1, 'b')],
           )
```

The 0 from the first level is not represented and can be removed

```
>>> mi2 = mi[2:].remove_unused_levels()
>>> mi2.levels
FrozenList([[1], ['a', 'b']])
```

pandas.MultiIndex.get_locs

MultiIndex.**get_locs**(*self*, *seq*)

Get location for a sequence of labels.

Parameters

seq [label, slice, list, mask or a sequence of such] You should use one of the above for each level. If a level should not be used, set it to `slice(None)`.

Returns

numpy.ndarray NumPy array of integers suitable for passing to iloc.

See also:

MultiIndex.get_loc Get location for a label or a tuple of labels.

MultiIndex.slice_locs Get slice location given start label(s) and end label(s).

Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
```

```
>>> mi.get_locs('b')
array([1, 2], dtype=int64)
```

```
>>> mi.get_locs([slice(None), ['e', 'f']])
array([1, 2], dtype=int64)
```

```
>>> mi.get_locs([[True, False, True], slice('e', 'f')])
array([2], dtype=int64)
```

IndexSlice	Create an object to more easily perform multi-index slicing.

pandas.IndexSlice

`pandas.`**`IndexSlice = <pandas.core.indexing._IndexSlice object>`**

Create an object to more easily perform multi-index slicing.

See also:

MultiIndex.remove_unused_levels New MultiIndex with no unused levels.

Notes

See *Defined Levels* for further info on slicing a MultiIndex.

Examples

```
>>> midx = pd.MultiIndex.from_product([['A0','A1'], ['B0','B1','B2','B3']])
>>> columns = ['foo', 'bar']
>>> dfmi = pd.DataFrame(np.arange(16).reshape((len(midx), len(columns))),
                        index=midx, columns=columns)
```

Using the default slice command:

```
>>> dfmi.loc[(slice(None), slice('B0', 'B1')), :]
           foo  bar
    A0 B0    0    1
       B1    2    3
    A1 B0    8    9
       B1   10   11
```

Using the IndexSlice class for a more intuitive command:

```
>>> idx = pd.IndexSlice
>>> dfmi.loc[idx[:, 'B0':'B1'], :]
           foo  bar
    A0 B0    0    1
       B1    2    3
```

(continues on next page)

(continued from previous page)

```
A1 B0     8     9
   B1    10    11
```

MultiIndex constructors

MultiIndex.from_arrays(arrays[, sortorder, …])	Convert arrays to MultiIndex.
MultiIndex.from_tuples(tuples[, sortorder, …])	Convert list of tuples to MultiIndex.
MultiIndex.from_product(iterables[, …])	Make a MultiIndex from the cartesian product of multiple iterables.
MultiIndex.from_frame(df[, sortorder, names])	Make a MultiIndex from a DataFrame.

MultiIndex properties

MultiIndex.names	Names of levels in MultiIndex.
MultiIndex.levels	
MultiIndex.codes	
MultiIndex.nlevels	Integer number of levels in this MultiIndex.
MultiIndex.levshape	A tuple with the length of each level.

pandas.MultiIndex.levels

MultiIndex.**levels**

pandas.MultiIndex.codes

property MultiIndex.**codes**

MultiIndex components

MultiIndex.set_levels(self, levels[, level, …])	Set new levels on MultiIndex.
MultiIndex.set_codes(self, codes[, level, …])	Set new codes on MultiIndex.
MultiIndex.to_flat_index(self)	Convert a MultiIndex to an Index of Tuples containing the level values.
MultiIndex.to_frame(self[, index, name])	Create a DataFrame with the levels of the MultiIndex as columns.
MultiIndex.is_lexsorted(self)	Return True if the codes are lexicographically sorted.
MultiIndex.sortlevel(self[, level, …])	Sort MultiIndex at the requested level.
MultiIndex.droplevel(self[, level])	Return index with requested level(s) removed.
MultiIndex.swaplevel(self[, i, j])	Swap level i with level j.
MultiIndex.reorder_levels(self, order)	Rearrange levels using input order.

continues on next page

Table 159 – continued from previous page

`MultiIndex.remove_unused_levels`(self)	Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.

MultiIndex selecting

`MultiIndex.get_loc`(self, key[, method])	Get location for a label or a tuple of labels as an integer, slice or boolean mask.
`MultiIndex.get_locs`(self, seq)	Get location for a sequence of labels.
`MultiIndex.get_loc_level`(self, key[, level])	Get both the location for the requested label(s) and the resulting sliced index.
`MultiIndex.get_indexer`(self, target[, …])	Compute indexer and mask for new index given the current index.
`MultiIndex.get_level_values`(self, level)	Return vector of label values for requested level, equal to the length of the index.

pandas.MultiIndex.get_loc

`MultiIndex.`**`get_loc`**(*self*, *key*, *method=None*)

Get location for a label or a tuple of labels as an integer, slice or boolean mask.

Parameters

key [label or tuple of labels (one for each level)]

method [None]

Returns

loc [int, slice object or boolean mask] If the key is past the lexsort depth, the return may be a boolean mask array, otherwise it is always a slice or int.

See also:

`Index.get_loc` The get_loc method for (single-level) index.
`MultiIndex.slice_locs` Get slice location given start label(s) and end label(s).
`MultiIndex.get_locs` Get location for a label/slice/list/mask or a sequence of such.

Notes

The key cannot be a slice, list of same-level labels, a boolean mask, or a sequence of such. If you want to use those, use `MultiIndex.get_locs()` instead.

Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
```

```
>>> mi.get_loc('b')
slice(1, 3, None)
```

```
>>> mi.get_loc(('b', 'e'))
1
```

pandas.MultiIndex.get_loc_level

`MultiIndex.`**`get_loc_level`**(*self*, *key*, *level=0*, *drop_level: bool = True*)

Get both the location for the requested label(s) and the resulting sliced index.

Parameters

key [label or sequence of labels]

level [int/level name or list thereof, optional]

drop_level [bool, default True] If `False`, the resulting index will not drop any level.

Returns

loc [A 2-tuple where the elements are:] Element 0: int, slice object or boolean array Element 1: The resulting sliced multiindex/index. If the key contains all levels, this will be `None`.

See also:

`MultiIndex.get_loc` Get location for a label or a tuple of labels.
`MultiIndex.get_locs` Get location for a label/slice/list/mask or a sequence of such.

Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')],
...                                names=['A', 'B'])
```

```
>>> mi.get_loc_level('b')
(slice(1, 3, None), Index(['e', 'f'], dtype='object', name='B'))
```

```
>>> mi.get_loc_level('e', level='B')
(array([False,  True, False], dtype=bool),
Index(['b'], dtype='object', name='A'))
```

```
>>> mi.get_loc_level(['b', 'e'])
(1, None)
```

pandas.MultiIndex.get_indexer

`MultiIndex.`**`get_indexer`**(*self*, *target*, *method=None*, *limit=None*, *tolerance=None*)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [MultiIndex or list of tuples]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit [int, optional] Maximum number of consecutive labels in `target` to match for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

pandas.MultiIndex.get_level_values

`MultiIndex.`**`get_level_values`**(*self*, *level*)

Return vector of label values for requested level, equal to the length of the index.

Parameters

level [int or str] `level` is either the integer position of the level in the MultiIndex, or the name of the level.

Returns

values [Index] Values is a level of this MultiIndex converted to a single *`Index`* (or subclass thereof).

Examples

Create a MultiIndex:

```
>>> mi = pd.MultiIndex.from_arrays((list('abc'), list('def')))
>>> mi.names = ['level_1', 'level_2']
```

Get level values by supplying level as either integer or name:

```
>>> mi.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object', name='level_1')
>>> mi.get_level_values('level_2')
Index(['d', 'e', 'f'], dtype='object', name='level_2')
```