pandas.Index.groupby

`Index.groupby(self, values)` → Dict[Hashable, numpy.ndarray]

Group the index labels by a given array of values.

Parameters

values [array] Values used to determine the groups.

Returns

dict {group name -> group labels}

pandas.Index.holds_integer

`Index.holds_integer(self)`

Whether the type is an integer type.

pandas.Index.identical

`Index.identical(self, other)` → bool

Similar to equals, but check that other comparable attributes are also equal.

Returns

bool If two Index objects have equal elements and same type True, otherwise False.

pandas.Index.insert

`Index.insert(self, loc, item)`

Make new Index inserting new item at location.

Follows Python list.append semantics for negative values.

Parameters

loc [int]

item [object]

Returns

new_index [Index]

pandas.Index.intersection

`Index.intersection(self, other, sort=False)`

Form the intersection of two Index objects.

This returns a new Index with elements common to the index and *other*.

Parameters

other [Index or array-like]

sort [False or None, default False] Whether to sort the resulting index.

- False : do not sort the result.

- None : sort the result, except when *self* and *other* are equal or when the values cannot be compared.

New in version 0.24.0.

Changed in version 0.24.1: Changed the default from `True` to `False`, to match the behaviour of 0.23.4 and earlier.

Returns

intersection [Index]

Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.intersection(idx2)
Int64Index([3, 4], dtype='int64')
```

pandas.Index.is_

`Index.is_`(*self*, *other*) → bool

More flexible, faster check like `is` but that works through views.

Note: this is *not* the same as `Index.identical()`, which checks that metadata is also the same.

Parameters

other [object] other object to compare against.

Returns

True if both have same underlying data, False otherwise [bool]

pandas.Index.is_categorical

`Index.is_categorical`(*self*) → bool

Check if the Index holds categorical data.

Returns

boolean True if the Index is categorical.

See also:

CategoricalIndex Index for categorical data.

Examples

```
>>> idx = pd.Index(["Watermelon", "Orange", "Apple",
...                 "Watermelon"]).astype("category")
>>> idx.is_categorical()
True
```

```
>>> idx = pd.Index([1, 3, 5, 7])
>>> idx.is_categorical()
False
```

```
>>> s = pd.Series(["Peter", "Victor", "Elisabeth", "Mar"])
>>> s
0        Peter
1       Victor
2    Elisabeth
3          Mar
dtype: object
>>> s.index.is_categorical()
False
```

pandas.Index.is_type_compatible

`Index.is_type_compatible`(*self*, *kind*) → bool

Whether the index type is compatible with the provided type.

pandas.Index.isin

`Index.isin`(*self*, *values*, *level=None*)

Return a boolean array where the index values are in *values*.

Compute boolean array of whether each index value is found in the passed set of values. The length of the returned boolean array matches the length of the index.

Parameters

values [set or list-like] Sought values.

level [str or int, optional] Name or position of the index level to use (if the index is a *MultiIndex*).

Returns

is_contained [ndarray] NumPy array of boolean values.

See also:

Series.isin Same for Series.

DataFrame.isin Same method for DataFrames.

Notes

In the case of *MultiIndex* you must either specify *values* as a list-like object containing tuples that are the same length as the number of levels, or specify *level*. Otherwise it will raise a `ValueError`.

If *level* is specified:

- if it is the name of one *and only one* index level, use that level;
- otherwise it should be a number indicating level position.

Examples

```
>>> idx = pd.Index([1,2,3])
>>> idx
Int64Index([1, 2, 3], dtype='int64')
```

Check whether each index value in a list of values. >>> idx.isin([1, 4]) array([True, False, False])

```
>>> midx = pd.MultiIndex.from_arrays([[1,2,3],
...                                  ['red', 'blue', 'green']],
...                                  names=('number', 'color'))
>>> midx
MultiIndex(levels=[[1, 2, 3], ['blue', 'green', 'red']],
           codes=[[0, 1, 2], [2, 0, 1]],
           names=['number', 'color'])
```

Check whether the strings in the 'color' level of the MultiIndex are in a list of colors.

```
>>> midx.isin(['red', 'orange', 'yellow'], level='color')
array([ True, False, False])
```

To check across the levels of a MultiIndex, pass a list of tuples:

```
>>> midx.isin([(1, 'red'), (3, 'red')])
array([ True, False, False])
```

For a DatetimeIndex, string values in *values* are converted to Timestamps.

```
>>> dates = ['2000-03-11', '2000-03-12', '2000-03-13']
>>> dti = pd.to_datetime(dates)
>>> dti
DatetimeIndex(['2000-03-11', '2000-03-12', '2000-03-13'],
dtype='datetime64[ns]', freq=None)
```

```
>>> dti.isin(['2000-03-11'])
array([ True, False, False])
```

pandas.Index.isna

`Index.isna(self)`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

numpy.ndarray A boolean array of whether my values are NA.

See also:

`Index.notna` Boolean inverse of isna.

`Index.dropna` Omit entries with missing values.

`isna` Top-level isna.

`Series.isna` Detect missing values in Series object.

Examples

Show which entries in a pandas.Index are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, *NaT* (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                         pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

pandas.Index.isnull

`Index.isnull(self)`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

numpy.ndarray A boolean array of whether my values are NA.

See also:

`Index.notna` Boolean inverse of isna.

`Index.dropna` Omit entries with missing values.

`isna` Top-level isna.

`Series.isna` Detect missing values in Series object.

Examples

Show which entries in a pandas.Index are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, *NaT* (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                         pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

pandas.Index.item

`Index.item(self)`

Return the first element of the underlying data as a python scalar.

Returns

scalar The first element of %(klass)s.

Raises

ValueError If the data is not length-1.

pandas.Index.join

`Index.join(self, other, how='left', level=None, return_indexers=False, sort=False)`

Compute join_index and indexers to conform data structures to the new index.

Parameters

other [Index]

how [{'left', 'right', 'inner', 'outer'}]

level [int or level name, default None]

return_indexers [bool, default False]

sort [bool, default False] Sort the join keys lexicographically in the result Index. If False, the order of the join keys depends on the join type (how keyword).

Returns

join_index, (left_indexer, right_indexer)

pandas.Index.map

`Index.map(self, mapper, na_action=None)`

Map values using input correspondence (a dict, Series, or function).

Parameters

mapper [function, dict, or Series] Mapping correspondence.

na_action [{None, 'ignore'}] If 'ignore', propagate NA values, without passing them to the mapping correspondence.

Returns

applied [Union[Index, MultiIndex], inferred] The output of the mapping function applied to the index. If the function returns a tuple with more than one element a MultiIndex will be returned.

pandas.Index.max

`Index.max`(*self*, *axis=None*, *skipna=True*, **args*, ***kwargs*)
Return the maximum value of the Index.

Parameters

axis [int, optional] For compatibility with NumPy. Only 0 or None are allowed.

skipna [bool, default True]

Returns

scalar Maximum value.

See also:

Index.min Return the minimum value in an Index.

Series.max Return the maximum value in a Series.

DataFrame.max Return the maximum values in a DataFrame.

Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.max()
3
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.max()
('b', 2)
```

pandas.Index.memory_usage

`Index.memory_usage`(*self*, *deep=False*)
Memory usage of the values.

Parameters

deep [bool] Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption.

Returns

bytes used

See also:

`numpy.ndarray.nbytes`

Notes

Memory usage does not include memory consumed by elements that are not components of the array if deep=False or if used on PyPy

pandas.Index.min

`Index.min(self, axis=None, skipna=True, *args, **kwargs)`

Return the minimum value of the Index.

Parameters

axis [{None}] Dummy argument for consistency with Series.

skipna [bool, default True]

Returns

scalar Minimum value.

See also:

Index.max Return the maximum value of the object.

Series.min Return the minimum value in a Series.

DataFrame.min Return the minimum values in a DataFrame.

Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.min()
1
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```

For a MultiIndex, the minimum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.min()
('a', 1)
```

pandas.Index.notna

`Index.notna(self)`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to `False` values.

Returns

numpy.ndarray Boolean array to indicate which entries are not NA.

See also:

Index.notnull Alias of notna.

Index.isna Inverse of notna.

notna Top-level notna.

Examples

Show which entries in an Index are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

pandas.Index.notnull

`Index.notnull(self)`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to `False` values.

Returns

numpy.ndarray Boolean array to indicate which entries are not NA.

See also:

Index.notnull Alias of notna.

Index.isna Inverse of notna.

notna Top-level notna.

Examples

Show which entries in an Index are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

pandas.Index.nunique

`Index.nunique(self, dropna=True)`

Return number of unique elements in the object.

Excludes NA values by default.

Parameters

dropna [bool, default True] Don't include NaN in the count.

Returns

int

See also:

`DataFrame.nunique` Method nunique for DataFrame.

`Series.count` Count non-NA/null observations in the Series.

Examples

```
>>> s = pd.Series([1, 3, 5, 7, 7])
>>> s
0    1
1    3
2    5
3    7
4    7
dtype: int64
```

```
>>> s.nunique()
4
```

pandas.Index.putmask

`Index.putmask(self, mask, value)`

Return a new Index of the values set with the mask.

Returns

Index

See also:

numpy.ndarray.putmask

pandas.Index.ravel

`Index.ravel(self, order='C')`

Return an ndarray of the flattened values of the underlying data.

Returns

numpy.ndarray Flattened array.

See also:

numpy.ndarray.ravel

pandas.Index.reindex

`Index.reindex(self, target, method=None, level=None, limit=None, tolerance=None)`

Create index with target's values (move/add/delete values as necessary).

Parameters

target [an iterable]

Returns

new_index [pd.Index] Resulting index.

indexer [np.ndarray or None] Indices of output values in original index.

pandas.Index.rename

`Index.rename(self, name, inplace=False)`

Alter Index or MultiIndex name.

Able to set new names without level. Defaults to returning new index. Length of names must match number of levels in MultiIndex.

Parameters

name [label or list of labels] Name(s) to set.

inplace [bool, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

Returns

Index The same type as the caller or None if inplace is True.

See also:

Index.set_names Able to set new names partially and by level.

Examples

```
>>> idx = pd.Index(['A', 'C', 'A', 'B'], name='score')
>>> idx.rename('grade')
Index(['A', 'C', 'A', 'B'], dtype='object', name='grade')
```

```
>>> idx = pd.MultiIndex.from_product([['python', 'cobra'],
...                                   [2018, 2019]],
...                                   names=['kind', 'year'])
>>> idx
MultiIndex([('python', 2018),
            ('python', 2019),
            ( 'cobra', 2018),
            ( 'cobra', 2019)],
           names=['kind', 'year'])
>>> idx.rename(['species', 'year'])
MultiIndex([('python', 2018),
            ('python', 2019),
            ( 'cobra', 2018),
            ( 'cobra', 2019)],
           names=['species', 'year'])
>>> idx.rename('species')
Traceback (most recent call last):
TypeError: Must pass list-like as `names`.
```

pandas.Index.repeat

`Index.`**repeat**(*self*, *repeats*, *axis=None*)

Repeat elements of a Index.

Returns a new Index where each element of the current Index is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Index.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_index [Index] Newly created Index with repeated elements.

See also:

Series.repeat Equivalent function for Series.

numpy.repeat Similar method for `numpy.ndarray`.

Examples

```
>>> idx = pd.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.repeat(2)
Index(['a', 'a', 'b', 'b', 'c', 'c'], dtype='object')
>>> idx.repeat([1, 2, 3])
Index(['a', 'b', 'b', 'c', 'c', 'c'], dtype='object')
```

pandas.Index.searchsorted

`Index.`**`searchsorted`**(*self*, *value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Index *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Note: The Index *must* be monotonically sorted, otherwise wrong locations will likely be returned. Pandas does *not* check this for you.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

int or array of int A scalar or array of insertion points with the same shape as *value*.

Changed in version 0.24.0: If *value* is a scalar, an int is now always returned. Previously, scalar inputs returned an 1-item array for `Series` and `Categorical`.

See also:

`sort_values`

`numpy.searchsorted`

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
3
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',
                        'cheese', 'milk'], ordered=True)
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
1
```

```
>>> x.searchsorted(['bread'], side='right')
array([3])
```

If the values are not monotonically sorted, wrong locations may be returned:

```
>>> x = pd.Series([2, 1, 3])
>>> x.searchsorted(1)
0  # wrong result, correct would be 1
```

pandas.Index.set_names

`Index.set_names`(*self*, *names*, *level=None*, *inplace=False*)

Set Index or MultiIndex name.

Able to set new names partially and by level.

Parameters

names [label or list of label] Name(s) to set.

level [int, label or list of int or label, optional] If the index is a MultiIndex, level(s) to set (None for all levels). Otherwise level must be None.

inplace [bool, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

Returns

Index The same type as the caller or None if inplace is True.

See also:

Index.rename Able to set new names without level.

Examples

```
>>> idx = pd.Index([1, 2, 3, 4])
>>> idx
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx.set_names('quarter')
Int64Index([1, 2, 3, 4], dtype='int64', name='quarter')
```

```
>>> idx = pd.MultiIndex.from_product([['python', 'cobra'],
...                                   [2018, 2019]])
>>> idx
MultiIndex([('python', 2018),
            ('python', 2019),
            ( 'cobra', 2018),
            ( 'cobra', 2019)],
           )
>>> idx.set_names(['kind', 'year'], inplace=True)
>>> idx
MultiIndex([('python', 2018),
            ('python', 2019),
            ( 'cobra', 2018),
            ( 'cobra', 2019)],
           names=['kind', 'year'])
>>> idx.set_names('species', level=0)
MultiIndex([('python', 2018),
            ('python', 2019),
            ( 'cobra', 2018),
            ( 'cobra', 2019)],
           names=['species', 'year'])
```

pandas.Index.set_value

`Index.`**`set_value`**(*self*, *arr*, *key*, *value*)

Fast lookup of value from 1-dimensional ndarray.

Deprecated since version 1.0.

Notes

Only use this if you know what you're doing.

pandas.Index.shift

`Index.shift(self, periods=1, freq=None)`

Shift index by desired number of time frequency increments.

This method is for shifting the values of datetime-like indexes by a specified time increment a given number of times.

Parameters

periods [int, default 1] Number of periods (or increments) to shift by, can be positive or negative.

freq [pandas.DateOffset, pandas.Timedelta or str, optional] Frequency increment to shift by. If None, the index is shifted by its own *freq* attribute. Offset aliases are valid strings, e.g., 'D', 'W', 'M' etc.

Returns

pandas.Index Shifted index.

See also:

`Series.shift` Shift values of Series.

Notes

This method is only implemented for datetime-like index classes, i.e., DatetimeIndex, PeriodIndex and TimedeltaIndex.

Examples

Put the first 5 month starts of 2011 into an index.

```
>>> month_starts = pd.date_range('1/1/2011', periods=5, freq='MS')
>>> month_starts
DatetimeIndex(['2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-05-01'],
              dtype='datetime64[ns]', freq='MS')
```

Shift the index by 10 days.

```
>>> month_starts.shift(10, freq='D')
DatetimeIndex(['2011-01-11', '2011-02-11', '2011-03-11', '2011-04-11',
               '2011-05-11'],
              dtype='datetime64[ns]', freq=None)
```

The default value of *freq* is the *freq* attribute of the index, which is 'MS' (month start) in this example.

```
>>> month_starts.shift(10)
DatetimeIndex(['2011-11-01', '2011-12-01', '2012-01-01', '2012-02-01',
               '2012-03-01'],
              dtype='datetime64[ns]', freq='MS')
```

pandas.Index.slice_indexer

Index.**slice_indexer**(*self*, *start=None*, *end=None*, *step=None*, *kind=None*)

For an ordered or unique index, compute the slice indexer for input labels and step.

Parameters

start [label, default None] If None, defaults to the beginning.

end [label, default None] If None, defaults to the end.

step [int, default None]

kind [str, default None]

Returns

indexer [slice]

Raises

KeyError [If key does not exist, or key is not unique and index is] not ordered.

Notes

This function assumes that the data is sorted, so use at your own peril

Examples

This is a method on all index types. For example you can do:

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_indexer(start='b', end='c')
slice(1, 3)
```

```
>>> idx = pd.MultiIndex.from_arrays([list('abcd'), list('efgh')])
>>> idx.slice_indexer(start='b', end=('c', 'g'))
slice(1, 3)
```

pandas.Index.slice_locs

Index.**slice_locs**(*self*, *start=None*, *end=None*, *step=None*, *kind=None*)

Compute slice locations for input labels.

Parameters

start [label, default None] If None, defaults to the beginning.

end [label, default None] If None, defaults to the end.

step [int, defaults None] If None, defaults to 1.

kind [{'ix', 'loc', 'getitem'} or None]

Returns

start, end [int]

See also:

Index.get_loc Get location for a single label.

Notes

This method only works if the index is monotonic or unique.

Examples

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

pandas.Index.sort

Index.**sort**(*self*, **args*, ***kwargs*)

Use sort_values instead.

pandas.Index.sort_values

Index.**sort_values**(*self*, *return_indexer=False*, *ascending=True*)

Return a sorted copy of the index.

Return a sorted copy of the index, and optionally return the indices that sorted the index itself.

Parameters

return_indexer [bool, default False] Should the indices that would sort the index be returned.

ascending [bool, default True] Should the index values be sorted in an ascending order.

Returns

sorted_index [pandas.Index] Sorted copy of the index.

indexer [numpy.ndarray, optional] The indices that the index itself was sorted by.

See also:

Series.sort_values Sort values of a Series.

DataFrame.sort_values Sort values in a DataFrame.

Examples

```
>>> idx = pd.Index([10, 100, 1, 1000])
>>> idx
Int64Index([10, 100, 1, 1000], dtype='int64')
```

Sort values in ascending order (default behavior).

```
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order, and also get the indices *idx* was sorted by.

```
>>> idx.sort_values(ascending=False, return_indexer=True)
(Int64Index([1000, 100, 10, 1], dtype='int64'), array([3, 1, 0, 2]))
```

pandas.Index.sortlevel

`Index.sortlevel`(*self*, *level=None*, *ascending=True*, *sort_remaining=None*)

For internal compatibility with with the Index API.

Sort the Index. This is for compat with MultiIndex

Parameters

ascending [bool, default True] False to sort in descending order

level, sort_remaining are compat parameters

Returns

Index

pandas.Index.str

`Index.str`()

Vectorized string functions for Series and Index. NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

Examples

```
>>> s.str.split('_')
>>> s.str.replace('_', '')
```

pandas.Index.symmetric_difference

`Index.symmetric_difference(self, other, result_name=None, sort=None)`

Compute the symmetric difference of two Index objects.

Parameters

other [Index or array-like]

result_name [str]

sort [False or None, default None] Whether to sort the resulting index. By default, the values are attempted to be sorted, but any TypeError from incomparable elements is caught by pandas.

- None : Attempt to sort the result, but catch any TypeErrors from comparing incomparable elements.
- False : Do not sort the result.

New in version 0.24.0.

Changed in version 0.24.1: Changed the default value from `True` to `None` (without change in behaviour).

Returns

symmetric_difference [Index]

Notes

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the ^ operator:

```
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

pandas.Index.take

`Index.take`(*self, indices, axis=0, allow_fill=True, fill_value=None, **kwargs*)

Return a new Index of the values selected by the indices.

For internal compatibility with numpy arrays.

Parameters

indices [list] Indices to be taken.

axis [int, optional] The axis over which to select values, always 0.

allow_fill [bool, default True]

fill_value [bool, default None] If allow_fill=True and fill_value is not None, indices specified by -1 is regarded as NA. If Index doesn't hold NA, raise ValueError.

Returns

numpy.ndarray Elements of given indices.

See also:

`numpy.ndarray.take`

pandas.Index.to_flat_index

`Index.to_flat_index`(*self*)

Identity method.

New in version 0.24.0.

This is implemented for compatibility with subclass implementations when chaining.

Returns

pd.Index Caller.

See also:

`MultiIndex.to_flat_index` Subclass implementation.

pandas.Index.to_frame

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

Create a DataFrame with a column containing the Index.

New in version 0.24.0.

Parameters

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

name [object, default None] The passed name should substitute for the index name (if it has one).

Returns

DataFrame DataFrame containing the original Index data.

See also:

Index.to_series Convert an Index to a Series.

Series.to_frame Convert Series to DataFrame.

Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
       animal
animal
Ant       Ant
Bear     Bear
Cow       Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
    animal
0   Ant
1  Bear
2   Cow
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(index=False, name='zoo')
    zoo
0   Ant
1  Bear
2   Cow
```

pandas.Index.to_list

Index.**to_list**(*self*)

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

Returns

list

See also:

numpy.ndarray.tolist

pandas.Index.to_native_types

`Index.to_native_types(self, slicer=None, **kwargs)`

Format specified values of *self* and return them.

Parameters

slicer [int, array-like] An indexer into *self* that specifies which values are used in the formatting process.

kwargs [dict] Options for specifying how the values should be formatted. These options include the following:

1) **na_rep** [str] The value that serves as a placeholder for NULL values
2) **quoting** [bool or None] Whether or not there are quoted values in *self*
3) **date_format** [str] The format used to represent date-like values.

Returns

numpy.ndarray Formatted values.

pandas.Index.to_numpy

`Index.to_numpy(self, dtype=None, copy=False, na_value=<object object at 0x7f5374a06320>, **kwargs)`

A NumPy ndarray representing the values in this Series or Index.

New in version 0.24.0.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`.

copy [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value [Any, optional] The value to use for missing values. The default value depends on *dtype* and the type of the array.

New in version 1.0.0.

****kwargs** Additional keywords passed through to the `to_numpy` method of the underlying array (for extension arrays).

New in version 1.0.0.

Returns

numpy.ndarray

See also:

Series.array Get the actual data stored within.

Index.array Get the actual data stored within.

DataFrame.to_numpy Similar method for DataFrame.

Notes

The returned array will be the same up to equality (values equal in *self* will be equal in the returned array; likewise for values that are not equal). When *self* contains an ExtensionArray, the dtype may be different. For example, for a category-dtype Series, `to_numpy()` will return a NumPy array and the categorical dtype will be lost.

For NumPy dtypes, this will be a reference to the actual data stored in this Series or Index (assuming `copy=False`). Modifying the result in place will modify the data stored in the Series or Index (not that we recommend doing that).

For extension types, `to_numpy()` *may* require copying data and coercing the result to a NumPy type (possibly object), which may be expensive. When you need a no-copy reference to the underlying data, `Series.array` should be used instead.

This table lays out the different dtypes and default return types of `to_numpy()` for various dtypes within pandas.

dtype	array type
category[T]	ndarray[T] (same dtype as input)
period	ndarray[object] (Periods)
interval	ndarray[object] (Intervals)
IntegerNA	ndarray[object]
datetime64[ns]	datetime64[ns]
datetime64[ns, tz]	ndarray[object] (Timestamps)

Examples

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

Specify the *dtype* to control how datetime-aware data is represented. Use `dtype=object` to return an ndarray of pandas `Timestamp` objects, each with the correct `tz`.

```
>>> ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> ser.to_numpy(dtype=object)
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or `dtype='datetime64[ns]'` to return an ndarray of native datetime64 values. The values are converted to UTC and the timezone info is dropped.

```
>>> ser.to_numpy(dtype="datetime64[ns]")
...
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00...'],
      dtype='datetime64[ns]')
```