pandas.Index.to_series

`Index.to_series(self, index=None, name=None)`

Create a Series with both index and values equal to the index keys.

Useful with map for returning an indexer based on an index.

Parameters

index [Index, optional] Index of resulting Series. If None, defaults to original index.

name [str, optional] Dame of resulting Series. If None, defaults to name of original index.

Returns

Series The dtype will be based on the type of the Index values.

pandas.Index.tolist

`Index.tolist(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.transpose

`Index.transpose(self, *args, **kwargs)`

Return the transpose, which is by definition self.

Returns

%(klass)s

pandas.Index.union

`Index.union(self, other, sort=None)`

Form the union of two Index objects.

If the Index objects are incompatible, both Index objects will be cast to dtype('object') first.

Changed in version 0.25.0.

Parameters

other [Index or array-like]

sort [bool or None, default None] Whether to sort the resulting Index.

- None : Sort the result, except when

1. *self* and *other* are equal.
2. *self* or *other* has length 0.
3. Some values in *self* or *other* cannot be compared. A RuntimeWarning is issued in this case.

- 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

union [Index]

Examples

Union matching dtypes

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

Union mismatched dtypes

```
>>> idx1 = pd.Index(['a', 'b', 'c', 'd'])
>>> idx2 = pd.Index([1, 2, 3, 4])
>>> idx1.union(idx2)
Index(['a', 'b', 'c', 'd', 1, 2, 3, 4], dtype='object')
```

pandas.Index.unique

`Index.`**`unique`**`(`*self*, *level=None*`)`

Return unique values in the index. Uniques are returned in order of appearance, this does NOT sort.

Parameters

level [int or str, optional, default None] Only return values from specified level (for MultiIndex).

New in version 0.23.0.

Returns

Index without duplicates

See also:

`unique`

`Series.unique`

pandas.Index.value_counts

Index.**value_counts**(*self*, *normalize=False*, *sort=True*, *ascending=False*, *bins=None*, *dropna=True*)

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters

normalize [bool, default False] If True then the object returned will contain the relative frequencies of the unique values.

sort [bool, default True] Sort by frequencies.

ascending [bool, default False] Sort in ascending order.

bins [int, optional] Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data.

dropna [bool, default True] Don't include counts of NaN.

Returns

Series

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.count Number of non-NA elements in a DataFrame.

Examples

```
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0    2
4.0    1
2.0    1
1.0    1
dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
>>> s.value_counts(normalize=True)
3.0    0.4
4.0    0.2
2.0    0.2
1.0    0.2
dtype: float64
```

bins

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

```
>>> s.value_counts(bins=3)
(2.0, 3.0]      2
(0.996, 2.0]    2
(3.0, 4.0]      1
dtype: int64
```

dropna

With *dropna* set to *False* we can also see NaN index values.

```
>>> s.value_counts(dropna=False)
3.0    2
NaN    1
4.0    1
2.0    1
1.0    1
dtype: int64
```

pandas.Index.where

`Index.`**`where`**(*self*, *cond*, *other=None*)

Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters

cond [bool array-like with the same length as self]

other [scalar, or array-like]

Returns

Index

is_boolean	
is_floating	
is_integer	
is_interval	
is_mixed	
is_numeric	
is_object	
view	

Properties

Index.values	Return an array representing the data in the Index.
Index.is_monotonic	Alias for is_monotonic_increasing.
Index.is_monotonic_increasing	Return if the index is monotonic increasing (only equal or increasing) values.
Index.is_monotonic_decreasing	Return if the index is monotonic decreasing (only equal or decreasing) values.
Index.is_unique	Return if the index has unique values.
Index.has_duplicates	

continues on next page

Table 131 – continued from previous page

Index.hasnans	Return if I have any nans; enables various perf speedups.
Index.dtype	Return the dtype object of the underlying data.
Index.inferred_type	Return a string of the type inferred from the values.
Index.is_all_dates	
Index.shape	Return a tuple of the shape of the underlying data.
Index.name	
Index.names	
Index.nbytes	Return the number of bytes in the underlying data.
Index.ndim	Number of dimensions of the underlying data, by definition 1.
Index.size	Return the number of elements in the underlying data.
Index.empty	
Index.T	Return the transpose, which is by definition self.
Index.memory_usage(self[, deep])	Memory usage of the values.

pandas.Index.has_duplicates

property Index.**has_duplicates**

pandas.Index.is_all_dates

Index.**is_all_dates**

pandas.Index.name

property Index.**name**

pandas.Index.names

property Index.**names**

pandas.Index.empty

property Index.**empty**

Modifying and computations

Index.all(self, *args, **kwargs)	Return whether all elements are True.
Index.any(self, *args, **kwargs)	Return whether any element is True.
Index.argmin(self[, axis, skipna])	Return a ndarray of the minimum argument indexer.
Index.argmax(self[, axis, skipna])	Return an ndarray of the maximum argument indexer.
Index.copy(self[, name, deep, dtype])	Make a copy of this object.
Index.delete(self, loc)	Make new Index with passed location(-s) deleted.
Index.drop(self, labels[, errors])	Make new Index with passed list of labels deleted.

continues on next page

Table 132 – continued from previous page

`Index.drop_duplicates`(self[, keep])	Return Index with duplicate values removed.
`Index.duplicated`(self[, keep])	Indicate duplicate index values.
`Index.equals`(self, other)	Determine if two Index objects contain the same elements.
`Index.factorize`(self[, sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
`Index.identical`(self, other)	Similar to equals, but check that other comparable attributes are also equal.
`Index.insert`(self, loc, item)	Make new Index inserting new item at location.
`Index.is_`(self, other)	More flexible, faster check like `is` but that works through views.
`Index.is_boolean`(self)	
`Index.is_categorical`(self)	Check if the Index holds categorical data.
`Index.is_floating`(self)	
`Index.is_integer`(self)	
`Index.is_interval`(self)	
`Index.is_mixed`(self)	
`Index.is_numeric`(self)	
`Index.is_object`(self)	
`Index.min`(self[, axis, skipna])	Return the minimum value of the Index.
`Index.max`(self[, axis, skipna])	Return the maximum value of the Index.
`Index.reindex`(self, target[, method, level, . . .])	Create index with target's values (move/add/delete values as necessary).
`Index.rename`(self, name[, inplace])	Alter Index or MultiIndex name.
`Index.repeat`(self, repeats[, axis])	Repeat elements of a Index.
`Index.where`(self, cond[, other])	Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.
`Index.take`(self, indices[, axis, . . .])	Return a new Index of the values selected by the indices.
`Index.putmask`(self, mask, value)	Return a new Index of the values set with the mask.
`Index.unique`(self[, level])	Return unique values in the index.
`Index.nunique`(self[, dropna])	Return number of unique elements in the object.
`Index.value_counts`(self[, normalize, sort, . . .])	Return a Series containing counts of unique values.

pandas.Index.is_boolean

`Index.`**`is_boolean`**(*self*) → bool

pandas.Index.is_floating

`Index.is_floating(self) → bool`

pandas.Index.is_integer

`Index.is_integer(self) → bool`

pandas.Index.is_interval

`Index.is_interval(self) → bool`

pandas.Index.is_mixed

`Index.is_mixed(self) → bool`

pandas.Index.is_numeric

`Index.is_numeric(self) → bool`

pandas.Index.is_object

`Index.is_object(self) → bool`

Compatibility with MultiIndex

Index.set_names(self, names[, level, inplace])	Set Index or MultiIndex name.
Index.droplevel(self[, level])	Return index with requested level(s) removed.

Missing values

Index.fillna(self[, value, downcast])	Fill NA/NaN values with the specified value.
Index.dropna(self[, how])	Return Index without NA/NaN values.
Index.isna(self)	Detect missing values.
Index.notna(self)	Detect existing (non-missing) values.

Conversion

Index.astype(self, dtype[, copy])	Create an Index with values cast to dtypes.
Index.item(self)	Return the first element of the underlying data as a python scalar.
Index.map(self, mapper[, na_action])	Map values using input correspondence (a dict, Series, or function).
Index.ravel(self[, order])	Return an ndarray of the flattened values of the underlying data.
Index.to_list(self)	Return a list of the values.
Index.to_native_types(self[, slicer])	Format specified values of *self* and return them.
Index.to_series(self[, index, name])	Create a Series with both index and values equal to the index keys.
Index.to_frame(self[, index, name])	Create a DataFrame with a column containing the Index.
Index.view(self[, cls])	

pandas.Index.view

Index.**view**(*self*, *cls=None*)

Sorting

Index.argsort(self, *args, **kwargs)	Return the integer indices that would sort the index.
Index.searchsorted(self, value[, side, sorter])	Find indices where elements should be inserted to maintain order.
Index.sort_values(self[, return_indexer, . . .])	Return a sorted copy of the index.

Time-specific operations

Index.shift(self[, periods, freq])	Shift index by desired number of time frequency increments.

Combining / joining / set operations

Index.append(self, other)	Append a collection of Index options together.
Index.join(self, other[, how, level, . . .])	Compute join_index and indexers to conform data structures to the new index.
Index.intersection(self, other[, sort])	Form the intersection of two Index objects.
Index.union(self, other[, sort])	Form the union of two Index objects.
Index.difference(self, other[, sort])	Return a new Index with elements from the index that are not in *other*.
Index.symmetric_difference(self, other[, . . .])	Compute the symmetric difference of two Index objects.

Selecting

`Index.asof`(self, label)	Return the label from the index, or, if not present, the previous one.
`Index.asof_locs`(self, where, mask)	Find the locations (indices) of the labels from the index for every entry in the *where* argument.
`Index.get_indexer`(self, target[, method, . . .])	Compute indexer and mask for new index given the current index.
`Index.get_indexer_for`(self, target, **kwargs)	Guaranteed return of an indexer even when non-unique.
`Index.get_indexer_non_unique`(self, target)	Compute indexer and mask for new index given the current index.
`Index.get_level_values`(self, level)	Return an Index of values for requested level.
`Index.get_loc`(self, key[, method, tolerance])	Get integer location, slice or boolean mask for requested label.
`Index.get_slice_bound`(self, label, side, kind)	Calculate slice bound that corresponds to given label.
`Index.get_value`(self, series, key)	Fast lookup of value from 1-dimensional ndarray.
`Index.isin`(self, values[, level])	Return a boolean array where the index values are in *values*.
`Index.slice_indexer`(self[, start, end, . . .])	For an ordered or unique index, compute the slice indexer for input labels and step.
`Index.slice_locs`(self[, start, end, step, kind])	Compute slice locations for input labels.

3.7.2 Numeric Index

`RangeIndex`([start, stop, step, dtype, copy, . . .])	Immutable Index implementing a monotonic integer range.
`Int64Index`([data, dtype, copy, name])	Immutable ndarray implementing an ordered, sliceable set.
`UInt64Index`([data, dtype, copy, name])	Immutable ndarray implementing an ordered, sliceable set.
`Float64Index`([data, dtype, copy, name])	Immutable ndarray implementing an ordered, sliceable set.

pandas.RangeIndex

class pandas.**RangeIndex**(*start=None*, *stop=None*, *step=None*, *dtype=None*, *copy=False*, *name=None*)

Immutable Index implementing a monotonic integer range.

RangeIndex is a memory-saving special case of Int64Index limited to representing monotonic ranges. Using RangeIndex may in some instances improve computing speed.

This is the default index type used by DataFrame and Series when no explicit index is provided by the user.

Parameters

start [int (default: 0), or other RangeIndex instance] If int and “stop” is not given, interpreted as “stop” instead.

stop [int (default: 0)]

step [int (default: 1)]

name [object, optional] Name to be stored in the index.

copy [bool, default False] Unused, accepted for homogeneity with other index types.

See also:

`Index` The base pandas Index type.
`Int64Index` Index of int64 data.

Attributes

`start`	The value of the *start* parameter (`0` if this was not supplied).
`stop`	The value of the *stop* parameter.
`step`	The value of the *step* parameter (`1` if this was not supplied).

pandas.RangeIndex.start

`RangeIndex.`**`start`**

The value of the *start* parameter (`0` if this was not supplied).

pandas.RangeIndex.stop

`RangeIndex.`**`stop`**

The value of the *stop* parameter.

pandas.RangeIndex.step

`RangeIndex.`**`step`**

The value of the *step* parameter (`1` if this was not supplied).

Methods

`from_range`(data[, name, dtype])	Create RangeIndex from a range object.

pandas.RangeIndex.from_range

`classmethod` `RangeIndex.`**`from_range`**(*data*, *name=None*, *dtype=None*)

Create RangeIndex from a range object.

Returns

RangeIndex

pandas.Int64Index

`class pandas.Int64Index(data=None, dtype=None, copy=False, name=None)`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Int64Index is a special case of *Index* with purely integer labels. .

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: int64)]

copy [bool] Make a copy of input ndarray.

name [object] Name to be stored in the index.

See also:

`Index` The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

pandas.UInt64Index

`class pandas.UInt64Index(data=None, dtype=None, copy=False, name=None)`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. UInt64Index is a special case of *Index* with purely unsigned integer labels. .

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: uint64)]

copy [bool] Make a copy of input ndarray.

name [object] Name to be stored in the index.

See also:

`Index` The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

pandas.Float64Index

class pandas.**Float64Index**(*data=None*, *dtype=None*, *copy=False*, *name=None*)

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Float64Index is a special case of *Index* with purely float labels. .

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: float64)]

copy [bool] Make a copy of input ndarray.

name [object] Name to be stored in the index.

See also:

Index The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

RangeIndex.start	The value of the *start* parameter (0 if this was not supplied).
RangeIndex.stop	The value of the *stop* parameter.

continues on next page

Table 143 – continued from previous page

RangeIndex.step	The value of the *step* parameter (1 if this was not supplied).
RangeIndex.from_range(data[, name, dtype])	Create RangeIndex from a range object.

3.7.3 CategoricalIndex

CategoricalIndex([data, categories, …])	Index based on an underlying *Categorical*.

pandas.CategoricalIndex

class pandas.**CategoricalIndex**(*data=None*, *categories=None*, *ordered=None*, *dtype=None*, *copy=False*, *name=None*)

Index based on an underlying *Categorical*.

CategoricalIndex, like Categorical, can only take on a limited, and usually fixed, number of possible values (*categories*). Also, like Categorical, it might have an order, but numerical operations (additions, divisions, …) are not possible.

Parameters

data [array-like (1-dimensional)] The values of the categorical. If *categories* are given, values not in *categories* will be replaced with NaN.

categories [index-like, optional] The categories for the categorical. Items need to be unique. If the categories are not given here (and also not in *dtype*), they will be inferred from the *data*.

ordered [bool, optional] Whether or not this categorical is treated as an ordered categorical. If not given here or in *dtype*, the resulting categorical will be unordered.

dtype [CategoricalDtype or "category", optional] If *CategoricalDtype*, cannot be used together with *categories* or *ordered*.

New in version 0.21.0.

copy [bool, default False] Make a copy of input ndarray.

name [object, optional] Name to be stored in the index.

Raises

ValueError If the categories do not validate.

TypeError If an explicit `ordered=True` is given but no *categories* and the *values* are not sortable.

See also:

Index The base pandas Index type.

Categorical A categorical array.

CategoricalDtype Type for categorical data.

Notes

See the user guide for more.

Examples

```
>>> pd.CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'])
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'], categories=['a', 'b', 'c'],␣
↪ordered=False, dtype='category')  # noqa
```

`CategoricalIndex` can also be instantiated from a `Categorical`:

```
>>> c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
>>> pd.CategoricalIndex(c)
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'], categories=['a', 'b', 'c'],␣
↪ordered=False, dtype='category')  # noqa
```

Ordered `CategoricalIndex` can have a min and max value.

```
>>> ci = pd.CategoricalIndex(['a','b','c','a','b','c'], ordered=True,
...                          categories=['c', 'b', 'a'])
>>> ci
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'], categories=['c', 'b', 'a'],␣
↪ordered=True, dtype='category')  # noqa
>>> ci.min()
'c'
```

Attributes

codes	The category codes of this categorical.
categories	The categories of this categorical.
ordered	Whether the categories have an ordered relationship.

pandas.CategoricalIndex.codes

property `CategoricalIndex.`**codes**

The category codes of this categorical.

Level codes are an array if integer which are the positions of the real values in the categories array.

There is not setter, use the other categorical methods and the normal item setter to change values in the categorical.

pandas.CategoricalIndex.categories

property CategoricalIndex.**categories**

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to *categories* is a inplace operation!

Raises

ValueError If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

rename_categories

reorder_categories

add_categories

remove_categories

remove_unused_categories

set_categories

pandas.CategoricalIndex.ordered

property CategoricalIndex.**ordered**

Whether the categories have an ordered relationship.

Methods

rename_categories(self, *args, **kwargs)	Rename categories.
reorder_categories(self, *args, **kwargs)	Reorder categories as specified in new_categories.
add_categories(self, *args, **kwargs)	Add new categories.
remove_categories(self, *args, **kwargs)	Remove the specified categories.
remove_unused_categories(self, *args, **kwargs)	Remove categories which are not used.
set_categories(self, *args, **kwargs)	Set the categories to the specified new_categories.
as_ordered(self, *args, **kwargs)	Set the Categorical to be ordered.
as_unordered(self, *args, **kwargs)	Set the Categorical to be unordered.
map(self, mapper)	Map values using input correspondence (a dict, Series, or function).

pandas.CategoricalIndex.rename_categories

CategoricalIndex.**rename_categories**(*self*, **args*, ***kwargs*)

Rename categories.

Parameters

new_categories [list-like, dict-like or callable] New categories which will replace old categories.

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.

New in version 0.21.0..

- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

New in version 0.23.0..

inplace [bool, default False] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns

cat [Categorical or None] With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

Raises

ValueError If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

See also:

reorder_categories

add_categories

remove_categories

remove_unused_categories

set_categories

Examples

```
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

pandas.CategoricalIndex.reorder_categories

`CategoricalIndex.`**`reorder_categories`**(*self*, **args*, ***kwargs*)

Reorder categories as specified in new_categories.

new_categories need to include all old categories and no new category items.

Parameters

new_categories [Index-like] The categories in new order.

ordered [bool, optional] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

inplace [bool, default False] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If the new categories do not contain all old category items or any new ones

See also:

`rename_categories`

`add_categories`

`remove_categories`

`remove_unused_categories`

`set_categories`

pandas.CategoricalIndex.add_categories

`CategoricalIndex.`**`add_categories`**(*self*, **args*, ***kwargs*)

Add new categories.

new_categories will be included at the last/highest place in the categories and will be unused directly after this call.

Parameters

new_categories [category or list-like of category] The new categories to be included.

inplace [bool, default False] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical with new categories added or None if inplace.]

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories

reorder_categories

remove_categories

remove_unused_categories

set_categories

pandas.CategoricalIndex.remove_categories

CategoricalIndex.**remove_categories**(*self*, **args*, ***kwargs*)

Remove the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [bool, default False] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical with removed categories or None if inplace.]

Raises

ValueError If the removals are not contained in the categories

See also:

rename_categories

reorder_categories

add_categories

remove_unused_categories

set_categories

pandas.CategoricalIndex.remove_unused_categories

`CategoricalIndex.`**`remove_unused_categories`**(*self*, **args*, ***kwargs*)

Remove categories which are not used.

Parameters

inplace [bool, default False] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Returns

cat [Categorical with unused categories dropped or None if inplace.]

See also:

`rename_categories`

`reorder_categories`

`add_categories`

`remove_categories`

`set_categories`

pandas.CategoricalIndex.set_categories

`CategoricalIndex.`**`set_categories`**(*self*, **args*, ***kwargs*)

Set the categories to the specified new_categories.

new_categories can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes, which does not considers a S1 string equal to a single char python string.

Parameters

new_categories [Index-like] The categories in new order.

ordered [bool, default False] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

rename [bool, default False] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

inplace [bool, default False] Whether or not to reorder the categories in-place or return a copy of this categorical with reordered categories.

Returns

Categorical with reordered categories or None if inplace.

Raises

ValueError If new_categories does not validate as categories

See also:

rename_categories

reorder_categories

add_categories

remove_categories

remove_unused_categories

pandas.CategoricalIndex.as_ordered

CategoricalIndex.**as_ordered**(*self*, **args*, ***kwargs*)

Set the Categorical to be ordered.

Parameters

inplace [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to True.

Returns

Categorical Ordered Categorical.

pandas.CategoricalIndex.as_unordered

CategoricalIndex.**as_unordered**(*self*, **args*, ***kwargs*)

Set the Categorical to be unordered.

Parameters

inplace [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to False.

Returns

Categorical Unordered Categorical.

pandas.CategoricalIndex.map

CategoricalIndex.**map**(*self*, *mapper*)

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

Maps the values (their categories, not the codes) of the index to new categories. If the mapping correspondence is one-to-one the result is a *CategoricalIndex* which has the same order property as the original, otherwise an *Index* is returned.

If a *dict* or *Series* is used any unmapped category is mapped to *NaN*. Note that if this happens an *Index* will be returned.

Parameters

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

Returns

pandas.CategoricalIndex or pandas.Index Mapped index.

See also:

Index.map Apply a mapping correspondence on an Index.

Series.map Apply a mapping correspondence on a Series.

Series.apply Apply more complex functions on a Series.

Examples

```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'])
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                 ordered=False, dtype='category')
>>> idx.map(lambda x: x.upper())
CategoricalIndex(['A', 'B', 'C'], categories=['A', 'B', 'C'],
                 ordered=False, dtype='category')
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'third'})
CategoricalIndex(['first', 'second', 'third'], categories=['first',
                 'second', 'third'], ordered=False, dtype='category')
```

If the mapping is one-to-one the ordering of the categories is preserved:

```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'], ordered=True)
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                 ordered=True, dtype='category')
>>> idx.map({'a': 3, 'b': 2, 'c': 1})
CategoricalIndex([3, 2, 1], categories=[3, 2, 1], ordered=True,
                 dtype='category')
```

If the mapping is not one-to-one an Index is returned:

```
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'first'})
Index(['first', 'second', 'first'], dtype='object')
```

If a *dict* is used, all unmapped categories are mapped to *NaN* and the result is an Index:

```
>>> idx.map({'a': 'first', 'b': 'second'})
Index(['first', 'second', nan], dtype='object')
```

Categorical components

CategoricalIndex.codes	The category codes of this categorical.
CategoricalIndex.categories	The categories of this categorical.
CategoricalIndex.ordered	Whether the categories have an ordered relationship.
CategoricalIndex.rename_categories(self, …)	Rename categories.
CategoricalIndex.reorder_categories(self, …)	Reorder categories as specified in new_categories.
CategoricalIndex.add_categories(self, *args, …)	Add new categories.
CategoricalIndex.remove_categories(self, …)	Remove the specified categories.

continues on next page

Table 147 – continued from previous page

CategoricalIndex.remove_unused_categories(…)	Remove categories which are not used.
CategoricalIndex.set_categories(self, *args, …)	Set the categories to the specified new_categories.
CategoricalIndex.as_ordered(self, *args, …)	Set the Categorical to be ordered.
CategoricalIndex.as_unordered(self, *args, …)	Set the Categorical to be unordered.

Modifying and computations

CategoricalIndex.map(self, mapper)	Map values using input correspondence (a dict, Series, or function).
CategoricalIndex.equals(self, other)	Determine if two CategoricalIndex objects contain the same elements.

pandas.CategoricalIndex.equals

CategoricalIndex.**equals**(*self*, *other*)

Determine if two CategoricalIndex objects contain the same elements.

Returns

bool If two CategoricalIndex objects have equal elements True, otherwise False.

3.7.4 IntervalIndex

IntervalIndex(data[, closed, dtype, name])	Immutable index of intervals that are closed on the same side.

pandas.IntervalIndex

class pandas.**IntervalIndex**(*data*, *closed=None*, *dtype=None*, *copy: bool = False*, *name=None*, *verify_integrity: bool = True*)

Immutable index of intervals that are closed on the same side.

New in version 0.20.0.

Parameters

data [array-like (1-dimensional)] Array-like containing Interval objects from which to build the IntervalIndex.

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

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

New in version 0.23.0.

copy [bool, default False] Copy the input data.

name [object, optional] Name to be stored in the index.

verify_integrity [bool, default True] Verify that the IntervalIndex is valid.

See also:

Index The base pandas Index type.
Interval A bounded slice-like interval; the elements of an IntervalIndex.
interval_range Function to create a fixed frequency IntervalIndex.
cut Bin values into discrete Intervals.
qcut Bin values into equal-sized Intervals based on rank or sample quantiles.

Notes

See the user guide for more.

Examples

A new IntervalIndex is typically constructed using *interval_range()*:

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

It may also be constructed using one of the constructor methods: *IntervalIndex.from_arrays()*, *IntervalIndex.from_breaks()*, and *IntervalIndex.from_tuples()*.

See further examples in the doc strings of interval_range and the mentioned constructor methods.

Attributes

left	Return the left endpoints of each Interval in the IntervalArray as an Index.
right	Return the right endpoints of each Interval in the IntervalArray as an Index.
closed	Whether the intervals are closed on the left-side, right-side, both or neither.
mid	Return the midpoint of each Interval in the IntervalArray as an Index.
length	Return an Index with entries denoting the length of each Interval in the IntervalArray.
is_empty	Indicates if an interval is empty, meaning it contains no points.
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.
is_overlapping	Return True if the IntervalIndex has overlapping intervals, else False.
values	Return the IntervalIndex's data as an IntervalArray.

pandas.IntervalIndex.left

property InternalIndex.**left**

Return the left endpoints of each Interval in the IntervalArray as an Index.

pandas.IntervalIndex.right

property IntervalIndex.**right**

Return the right endpoints of each Interval in the IntervalArray as an Index.

pandas.IntervalIndex.closed

IntervalIndex.**closed**

Whether the intervals are closed on the left-side, right-side, both or neither.

pandas.IntervalIndex.mid

IntervalIndex.**mid**

Return the midpoint of each Interval in the IntervalArray as an Index.

pandas.IntervalIndex.length

property IntervalIndex.**length**

Return an Index with entries denoting the length of each Interval in the IntervalArray.

pandas.IntervalIndex.is_empty

IntervalIndex.**is_empty**

Indicates if an interval is empty, meaning it contains no points.

New in version 0.25.0.

Returns

bool or ndarray A boolean indicating if a scalar *Interval* is empty, or a boolean ndarray positionally indicating if an Interval in an *IntervalArray* or *IntervalIndex* is empty.

Examples

An *Interval* that contains points is not empty:

```
>>> pd.Interval(0, 1, closed='right').is_empty
False
```

An Interval that does not contain any points is empty:

```
>>> pd.Interval(0, 0, closed='right').is_empty
True
>>> pd.Interval(0, 0, closed='left').is_empty
True
>>> pd.Interval(0, 0, closed='neither').is_empty
True
```

An `Interval` that contains a single point is not empty:

```
>>> pd.Interval(0, 0, closed='both').is_empty
False
```

An *`IntervalArray`* or *`IntervalIndex`* returns a boolean `ndarray` positionally indicating if an `Interval` is empty:

```
>>> ivs = [pd.Interval(0, 0, closed='neither'),
...        pd.Interval(1, 2, closed='neither')]
>>> pd.arrays.IntervalArray(ivs).is_empty
array([ True, False])
```

Missing values are not considered empty:

```
>>> ivs = [pd.Interval(0, 0, closed='neither'), np.nan]
>>> pd.IntervalIndex(ivs).is_empty
array([ True, False])
```

pandas.IntervalIndex.is_non_overlapping_monotonic

`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.

pandas.IntervalIndex.is_overlapping

`property` `IntervalIndex.`**`is_overlapping`**

Return True if the IntervalIndex has overlapping intervals, else False.

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.

Returns

bool Boolean indicating if the IntervalIndex has overlapping intervals.

See also:

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

`IntervalIndex.overlaps` Check an IntervalIndex elementwise for overlaps.