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.Interval.left

Interval.**left**
Left bound for the interval.

pandas.Interval.length

Interval.**length**
Return the length of the Interval.

pandas.Interval.mid

Interval.**mid**
Return the midpoint of the Interval.

pandas.Interval.open_left

Interval.**open_left**
Check if the interval is open on the left side.

For the meaning of *closed* and *open* see *Interval*.

Returns

bool True if the Interval is closed on the left-side.

pandas.Interval.open_right

Interval.**open_right**
Check if the interval is open on the right side.

For the meaning of *closed* and *open* see *Interval*.

Returns

bool True if the Interval is closed on the left-side.

pandas.Interval.right

Interval.**right**
Right bound for the interval.

Methods

overlaps()	Check whether two Interval objects overlap.

pandas.Interval.overlaps

Interval.**overlaps**()
Check whether two Interval objects overlap.

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 [Interval] Interval to check against for an overlap.

Returns

bool True if the two intervals overlap.

See also:

IntervalArray.overlaps The corresponding method for IntervalArray.

IntervalIndex.overlaps The corresponding method for IntervalIndex.

Examples

```
>>> i1 = pd.Interval(0, 2)
>>> i2 = pd.Interval(1, 3)
>>> i1.overlaps(i2)
True
>>> i3 = pd.Interval(4, 5)
>>> i1.overlaps(i3)
False
```

Intervals that share closed endpoints overlap:

```
>>> i4 = pd.Interval(0, 1, closed='both')
>>> i5 = pd.Interval(1, 2, closed='both')
>>> i4.overlaps(i5)
True
```

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

```
>>> i6 = pd.Interval(1, 2, closed='neither')
>>> i4.overlaps(i6)
False
```

Properties

Interval.closed	Whether the interval is closed on the left-side, right-side, both or neither.
Interval.closed_left	Check if the interval is closed on the left side.
Interval.closed_right	Check if the interval is closed on the right side.
Interval.is_empty	Indicates if an interval is empty, meaning it contains no points.
Interval.left	Left bound for the interval.
Interval.length	Return the length of the Interval.
Interval.mid	Return the midpoint of the Interval.
Interval.open_left	Check if the interval is open on the left side.
Interval.open_right	Check if the interval is open on the right side.
Interval.overlaps()	Check whether two Interval objects overlap.
Interval.right	Right bound for the interval.

A collection of intervals may be stored in an *arrays.IntervalArray*.

arrays.IntervalArray(data[, closed, dtype, ...])	Pandas array for interval data that are closed on the same side.

pandas.arrays.IntervalArray

class pandas.arrays.**IntervalArray**(*data*, *closed=None*, *dtype=None*, *copy=False*, *verify_integrity=True*)

Pandas array for interval data that are closed on the same side.

New in version 0.24.0.

Parameters

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

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.

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

See also:

Index The base pandas Index type.
Interval A bounded slice-like interval; the elements of an IntervalArray.
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 `IntervalArray` can be constructed directly from an array-like of `Interval` objects:

```
>>> pd.arrays.IntervalArray([pd.Interval(0, 1), pd.Interval(1, 5)])
<IntervalArray>
[(0, 1], (1, 5]]
Length: 2, closed: right, dtype: interval[int64]
```

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

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.

continues on next page

Table 107 – continued from previous page

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.

pandas.arrays.IntervalArray.left

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

pandas.arrays.IntervalArray.right

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

pandas.arrays.IntervalArray.closed

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

pandas.arrays.IntervalArray.mid

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

pandas.arrays.IntervalArray.length

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

pandas.arrays.IntervalArray.is_empty

IntervalArray.**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.arrays.IntervalArray.is_non_overlapping_monotonic

property IntervalArray.**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.

Methods

from_arrays(left, right[, closed, copy, dtype])	Construct from two arrays defining the left and right bounds.
from_tuples(data[, closed, copy, dtype])	Construct an IntervalArray from an array-like of tuples.
from_breaks(breaks[, closed, copy, dtype])	Construct an IntervalArray from an array of splits.
contains(self, other)	Check elementwise if the Intervals contain the value.
overlaps(self, other)	Check elementwise if an Interval overlaps the values in the IntervalArray.

continues on next page

Table 108 – continued from previous page

set_closed(self, closed)	Return an IntervalArray identical to the current one, but closed on the specified side.
to_tuples(self[, na_tuple])	Return an ndarray of tuples of the form (left, right).

pandas.arrays.IntervalArray.from_arrays

classmethod IntervalArray.**from_arrays**(*left*, *right*, *closed='right'*, *copy=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

IntervalArray

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.

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

IntervalArray.from_tuples Construct an IntervalArray 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.

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

pandas.arrays.IntervalArray.from_tuples

classmethod IntervalArray.**from_tuples**(*data*, *closed='right'*, *copy=False*, *dtype=None*)

Construct an IntervalArray 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

IntervalArray

See also:

interval_range Function to create a fixed frequency IntervalIndex.

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

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

Examples

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

pandas.arrays.IntervalArray.from_breaks

classmethod IntervalArray.**from_breaks**(*breaks*, *closed='right'*, *copy=False*, *dtype=None*)

Construct an IntervalArray 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

IntervalArray

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_arrays Construct from a left and right array.

IntervalArray.from_tuples Construct from a sequence of tuples.

Examples

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

pandas.arrays.IntervalArray.contains

IntervalArray.**contains**(*self*, *other*)

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.arrays.IntervalArray.overlaps

IntervalArray.**overlaps**(*self*, *other*)

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.arrays.IntervalArray.set_closed

IntervalArray.**set_closed**(*self*, *closed*)

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.arrays.IntervalArray.to_tuples

IntervalArray.**to_tuples**(*self*, *na_tuple=True*)

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

IntervalDtype([subtype])	An ExtensionDtype for Interval data.

pandas.IntervalDtype

class pandas.**IntervalDtype**(*subtype=None*)

An ExtensionDtype for Interval data.

This is not an actual numpy dtype, but a duck type.

Parameters

subtype [str, np.dtype] The dtype of the Interval bounds.

Examples

```
>>> pd.IntervalDtype(subtype='int64')
interval[int64]
```

Attributes

subtype	The dtype of the Interval bounds.

pandas.IntervalDtype.subtype

property IntervalDtype.**subtype**
The dtype of the Interval bounds.

Methods

None	

3.5.7 Nullable integer

numpy.ndarray cannot natively represent integer-data with missing values. Pandas provides this through *arrays.IntegerArray*.

arrays.IntegerArray(values, mask[, copy])	Array of integer (optional missing) values.

pandas.arrays.IntegerArray

class pandas.arrays.**IntegerArray**(*values*, *mask*, *copy=False*)
Array of integer (optional missing) values.

New in version 0.24.0.

Changed in version 1.0.0: Now uses pandas.NA as the missing value rather than numpy.nan.

> **Warning:** IntegerArray is currently experimental, and its API or internal implementation may change without warning.

We represent an IntegerArray with 2 numpy arrays:
- data: contains a numpy integer array of the appropriate dtype
- mask: a boolean array holding a mask on the data, True is missing

To construct an IntegerArray from generic array-like input, use *pandas.array()* with one of the integer dtypes (see examples).

See *Nullable integer data type* for more.

Parameters

values [numpy.ndarray] A 1-d integer-dtype array.

mask [numpy.ndarray] A 1-d boolean-dtype array indicating missing values.

copy [bool, default False] Whether to copy the *values* and *mask*.

Returns

IntegerArray

Examples

Create an IntegerArray with *pandas.array()*.

```
>>> int_array = pd.array([1, None, 3], dtype=pd.Int32Dtype())
>>> int_array
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: Int32
```

String aliases for the dtypes are also available. They are capitalized.

```
>>> pd.array([1, None, 3], dtype='Int32')
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: Int32
```

```
>>> pd.array([1, None, 3], dtype='UInt16')
<IntegerArray>
[1, <NA>, 3]
Length: 3, dtype: UInt16
```

Attributes

None	

Methods

None	

Int8Dtype()	An ExtensionDtype for int8 integer data.
Int16Dtype()	An ExtensionDtype for int16 integer data.
Int32Dtype()	An ExtensionDtype for int32 integer data.
Int64Dtype()	An ExtensionDtype for int64 integer data.
UInt8Dtype()	An ExtensionDtype for uint8 integer data.
UInt16Dtype()	An ExtensionDtype for uint16 integer data.
UInt32Dtype()	An ExtensionDtype for uint32 integer data.
UInt64Dtype()	An ExtensionDtype for uint64 integer data.

pandas.Int8Dtype

class pandas.**Int8Dtype**
An ExtensionDtype for int8 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None	

Methods

None	

pandas.Int16Dtype

class pandas.**Int16Dtype**
An ExtensionDtype for int16 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None	

Methods

None	

pandas.Int32Dtype

class pandas.**Int32Dtype**
An ExtensionDtype for int32 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None

Methods

None

pandas.Int64Dtype

class pandas.**Int64Dtype**

An ExtensionDtype for int64 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None

Methods

None

pandas.UInt8Dtype

class pandas.**UInt8Dtype**

An ExtensionDtype for uint8 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None

Methods

None

pandas.UInt16Dtype

class pandas.**UInt16Dtype**
An ExtensionDtype for uint16 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None	

Methods

None	

pandas.UInt32Dtype

class pandas.**UInt32Dtype**
An ExtensionDtype for uint32 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None	

Methods

None	

pandas.UInt64Dtype

class pandas.**UInt64Dtype**
An ExtensionDtype for uint64 integer data.

Changed in version 1.0.0: Now uses `pandas.NA` as its missing value, rather than `numpy.nan`.

Attributes

None	

Methods

None	

3.5.8 Categorical data

Pandas defines a custom data type for representing data that can take only a limited, fixed set of values. The dtype of a `Categorical` can be described by a `pandas.api.types.CategoricalDtype`.

CategoricalDtype([categories])	Type for categorical data with the categories and orderedness.

pandas.CategoricalDtype

class pandas.**CategoricalDtype**(*categories=None*, *ordered: Optional[bool] = False*)

Type for categorical data with the categories and orderedness.

Changed in version 0.21.0.

Parameters

categories [sequence, optional] Must be unique, and must not contain any nulls.

ordered [bool or None, default False] Whether or not this categorical is treated as a ordered categorical. None can be used to maintain the ordered value of existing categoricals when used in operations that combine categoricals, e.g. astype, and will resolve to False if there is no existing ordered to maintain.

See also:

Categorical

Notes

This class is useful for specifying the type of a `Categorical` independent of the values. See *CategoricalDtype* for more.

Examples

```
>>> t = pd.CategoricalDtype(categories=['b', 'a'], ordered=True)
>>> pd.Series(['a', 'b', 'a', 'c'], dtype=t)
0      a
1      b
2      a
3    NaN
dtype: category
Categories (2, object): [b < a]
```

Attributes

categories	An Index containing the unique categories allowed.
ordered	Whether the categories have an ordered relationship.

pandas.CategoricalDtype.categories

property CategoricalDtype.**categories**
An Index containing the unique categories allowed.

pandas.CategoricalDtype.ordered

property CategoricalDtype.**ordered**
Whether the categories have an ordered relationship.

Methods

None	

CategoricalDtype.categories	An Index containing the unique categories allowed.
CategoricalDtype.ordered	Whether the categories have an ordered relationship.

Categorical data can be stored in a *pandas.Categorical*

Categorical(values[, categories, ordered, ...])	Represent a categorical variable in classic R / S-plus fashion.

pandas.Categorical

class pandas.**Categorical**(*values*, *categories=None*, *ordered=None*, *dtype=None*, *fastpath=False*)
Represent a categorical variable in classic R / S-plus fashion.

Categoricals can only take on only a limited, and usually fixed, number of possible values (*categories*). In contrast to statistical categorical variables, a *Categorical* might have an order, but numerical operations (additions, divisions, ...) are not possible.

All values of the *Categorical* are either in *categories* or *np.nan*. Assigning values outside of *categories* will raise a *ValueError*. Order is defined by the order of the *categories*, not lexical order of the values.

Parameters

values [list-like] The values of the categorical. If categories are given, values not in categories will be replaced with NaN.

categories [Index-like (unique), optional] The unique categories for this categorical. If not given, the categories are assumed to be the unique values of *values* (sorted, if possible, otherwise in the order in which they appear).

ordered [bool, default False] Whether or not this categorical is treated as a ordered categorical. If True, the resulting categorical will be ordered. An ordered categorical respects,

when sorted, the order of its *categories* attribute (which in turn is the *categories* argument, if provided).

dtype [CategoricalDtype] An instance of `CategoricalDtype` to use for this categorical.

New in version 0.21.0.

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:

CategoricalDtype Type for categorical data.
CategoricalIndex An Index with an underlying `Categorical`.

Notes

See the user guide for more.

Examples

```
>>> pd.Categorical([1, 2, 3, 1, 2, 3])
[1, 2, 3, 1, 2, 3]
Categories (3, int64): [1, 2, 3]
```

```
>>> pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
[a, b, c, a, b, c]
Categories (3, object): [a, b, c]
```

Ordered *Categoricals* can be sorted according to the custom order of the categories and can have a min and max value.

```
>>> c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'], ordered=True,
...                    categories=['c', 'b', 'a'])
>>> c
[a, b, c, a, b, c]
Categories (3, object): [c < b < a]
>>> c.min()
'c'
```

Attributes

categories	The categories of this categorical.
codes	The category codes of this categorical.
ordered	Whether the categories have an ordered relationship.
dtype	The `CategoricalDtype` for this instance.

pandas.Categorical.categories

property Categorical.**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.Categorical.codes

property Categorical.**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.Categorical.ordered

property Categorical.**ordered**
Whether the categories have an ordered relationship.

pandas.Categorical.dtype

property Categorical.**dtype**
The CategoricalDtype for this instance.

Methods

from_codes(codes[, categories, ordered, dtype])	Make a Categorical type from codes and categories or dtype.
__array__(self[, dtype])	The numpy array interface.

pandas.Categorical.from_codes

classmethod Categorical.**from_codes**(*codes*, *categories=None*, *ordered=None*, *dtype=None*)

Make a Categorical type from codes and categories or dtype.

This constructor is useful if you already have codes and categories/dtype and so do not need the (computation intensive) factorization step, which is usually done on the constructor.

If your data does not follow this convention, please use the normal constructor.

Parameters

codes [array-like of int] An integer array, where each integer points to a category in categories or dtype.categories, or else is -1 for NaN.

categories [index-like, optional] The categories for the categorical. Items need to be unique. If the categories are not given here, then they must be provided in *dtype*.

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 0.24.0: When *dtype* is provided, neither *categories* nor *ordered* should be provided.

Returns

Categorical

Examples

```
>>> dtype = pd.CategoricalDtype(['a', 'b'], ordered=True)
>>> pd.Categorical.from_codes(codes=[0, 1, 0, 1], dtype=dtype)
[a, b, a, b]
Categories (2, object): [a < b]
```

pandas.Categorical.__array__

Categorical.**__array__**(*self*, *dtype=None*) → numpy.ndarray

The numpy array interface.

Returns

numpy.array A numpy array of either the specified dtype or, if dtype==None (default), the same dtype as categorical.categories.dtype.

The alternative *Categorical.from_codes()* constructor can be used when you have the categories and integer codes already:

Categorical.from_codes(codes[, categories, ...])	Make a Categorical type from codes and categories or dtype.

The dtype information is available on the `Categorical`

Categorical.dtype	The `CategoricalDtype` for this instance.
Categorical.categories	The categories of this categorical.
Categorical.ordered	Whether the categories have an ordered relationship.
Categorical.codes	The category codes of this categorical.

`np.asarray(categorical)` works by implementing the array interface. Be aware, that this converts the Categorical back to a NumPy array, so categories and order information is not preserved!

Categorical.__array__(self[, dtype])	The numpy array interface.

A `Categorical` can be stored in a `Series` or `DataFrame`. To create a Series of dtype `category`, use `cat = s.astype(dtype)` or `Series(..., dtype=dtype)` where `dtype` is either

- the string `'category'`
- an instance of `CategoricalDtype`.

If the Series is of dtype `CategoricalDtype`, `Series.cat` can be used to change the categorical data. See *Categorical accessor* for more.

3.5.9 Sparse data

Data where a single value is repeated many times (e.g. `0` or `NaN`) may be stored efficiently as a *arrays.SparseArray*.

arrays.SparseArray(data[, sparse_index, ...])	An ExtensionArray for storing sparse data.

pandas.arrays.SparseArray

class pandas.arrays.**SparseArray**(*data*, *sparse_index=None*, *index=None*, *fill_value=None*, *kind='integer'*, *dtype=None*, *copy=False*)

An ExtensionArray for storing sparse data.

Changed in version 0.24.0: Implements the ExtensionArray interface.

Parameters

data [array-like] A dense array of values to store in the SparseArray. This may contain *fill_value*.

sparse_index [SparseIndex, optional]

index [Index]

fill_value [scalar, optional] Elements in *data* that are *fill_value* are not stored in the SparseArray. For memory savings, this should be the most common value in *data*. By default, *fill_value* depends on the dtype of *data*:

data.dtype	na_value
float	`np.nan`
int	`0`
bool	False
datetime64	`pd.NaT`
timedelta64	`pd.NaT`

The fill value is potentially specified in three ways. In order of precedence, these are

1. The *fill_value* argument
2. `dtype.fill_value` if *fill_value* is None and *dtype* is a `SparseDtype`
3. `data.dtype.fill_value` if *fill_value* is None and *dtype* is not a `SparseDtype` and *data* is a `SparseArray`.

kind [{'integer', 'block'}, default 'integer'] The type of storage for sparse locations.

- 'block': Stores a *block* and *block_length* for each contiguous *span* of sparse values. This is best when sparse data tends to be clumped together, with large regions of `fill-value` values between sparse values.
- 'integer': uses an integer to store the location of each sparse value.

dtype [np.dtype or SparseDtype, optional] The dtype to use for the SparseArray. For numpy dtypes, this determines the dtype of `self.sp_values`. For SparseDtype, this determines `self.sp_values` and `self.fill_value`.

copy [bool, default False] Whether to explicitly copy the incoming *data* array.

Attributes

None	

Methods

None	

SparseDtype(dtype, numpy.dtype, ...)	Dtype for data stored in `SparseArray`.

pandas.SparseDtype

class pandas.**SparseDtype**(*dtype: Union[str, numpy.dtype, ExtensionDtype] = <class 'numpy.float64'>*, *fill_value: Any = None*)

Dtype for data stored in `SparseArray`.

This dtype implements the pandas ExtensionDtype interface.

New in version 0.24.0.

Parameters

dtype [str, ExtensionDtype, numpy.dtype, type, default numpy.float64] The dtype of the underlying array storing the non-fill value values.

fill_value [scalar, optional] The scalar value not stored in the SparseArray. By default, this depends on *dtype*.

dtype	na_value
float	`np.nan`
int	`0`
bool	`False`
datetime64	`pd.NaT`
timedelta64	`pd.NaT`

The default value may be overridden by specifying a *fill_value*.

Attributes

None	

Methods

None	

The `Series.sparse` accessor may be used to access sparse-specific attributes and methods if the *`Series`* contains sparse values. See *Sparse accessor* for more.

3.5.10 Text data

When working with text data, where each valid element is a string or missing, we recommend using *`StringDtype`* (with the alias `"string"`).

`arrays.StringArray`(values[, copy])	Extension array for string data.

pandas.arrays.StringArray

class pandas.arrays.**StringArray**(*values*, *copy=False*)

Extension array for string data.

New in version 1.0.0.

> **Warning:** StringArray is considered experimental. The implementation and parts of the API may change without warning.

Parameters

values [array-like] The array of data.

> **Warning:** Currently, this expects an object-dtype ndarray where the elements are Python strings or `pandas.NA`. This may change without warning in the future.

> Use *pandas.array()* with `dtype="string"` for a stable way of creating a *StringArray* from any sequence.

copy [bool, default False] Whether to copy the array of data.

See also:

array The recommended function for creating a StringArray.
Series.str The string methods are available on Series backed by a StringArray.

Notes

StringArray returns a BooleanArray for comparison methods.

Examples

```
>>> pd.array(['This is', 'some text', None, 'data.'], dtype="string")
<StringArray>
['This is', 'some text', <NA>, 'data.']
Length: 4, dtype: string
```

Unlike `object` dtype arrays, `StringArray` doesn't allow non-string values.

```
>>> pd.array(['1', 1], dtype="string")
Traceback (most recent call last):
...
ValueError: StringArray requires an object-dtype ndarray of strings.
```

For comparison methods, this returns a `pandas.BooleanArray`

```
>>> pd.array(["a", None, "c"], dtype="string") == "a"
<BooleanArray>
[True, <NA>, False]
Length: 3, dtype: boolean
```

Attributes

None	

Methods

None	

StringDtype()	Extension dtype for string data.