(continued from previous page)

```
5      hippo
Name: animal, dtype: object
```

The value `False` for parameter 'keep' discards all sets of duplicated entries. Setting the value of 'inplace' to `True` performs the operation inplace and returns `None`.

```
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s
1        cow
3     beetle
5      hippo
Name: animal, dtype: object
```

pandas.Series.droplevel

`Series.droplevel(self: ~ FrameOrSeries, level, axis=0) → ~FrameOrSeries`

Return DataFrame with requested index / column level(s) removed.

New in version 0.24.0.

Parameters

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

axis [{0 or 'index', 1 or 'columns'}, default 0]

Returns

DataFrame DataFrame with requested index / column level(s) removed.

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...    ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
```

(continues on next page)

(continued from previous page)

```
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

pandas.Series.dropna

`Series.dropna(self, axis=0, inplace=False, how=None)`

Return a new Series with missing values removed.

See the *User Guide* for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or 'index'}, default 0] There is only one axis to drop values from.

inplace [bool, default False] If True, do operation inplace and return None.

how [str, optional] Not in use. Kept for compatibility.

Returns

Series Series with NA entries dropped from it.

See also:

Series.isna Indicate missing values.

Series.notna Indicate existing (non-missing) values.

Series.fillna Replace missing values.

DataFrame.dropna Drop rows or columns which contain NA values.

Index.dropna Drop missing indices.

Examples

```
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2    NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Keep the Series with valid entries in the same variable.

```
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```

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

```
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0       NaN
1         2
2       NaT
3
4      None
5    I stay
dtype: object
>>> ser.dropna()
1         2
3
5    I stay
dtype: object
```

pandas.Series.dt

`Series.dt()`
Accessor object for datetimelike properties of the Series values.

Examples

```
>>> s.dt.hour
>>> s.dt.second
>>> s.dt.quarter
```

Returns a Series indexed like the original Series. Raises TypeError if the Series does not contain datetime-like values.

pandas.Series.duplicated

`Series.duplicated(self, keep='first')`

Indicate duplicate Series values.

Duplicated values are indicated as `True` values in the resulting Series. Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.

Parameters

keep [{'first', 'last', False}, default 'first'] Method to handle dropping duplicates:

- 'first' : Mark duplicates as `True` except for the first occurrence.
- 'last' : Mark duplicates as `True` except for the last occurrence.
- `False` : Mark all duplicates as `True`.

Returns

Series Series indicating whether each value has occurred in the preceding values.

See also:

Index.duplicated Equivalent method on pandas.Index.

DataFrame.duplicated Equivalent method on pandas.DataFrame.

Series.drop_duplicates Remove duplicate values from Series.

Examples

By default, for each set of duplicated values, the first occurrence is set on False and all others on True:

```
>>> animals = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> animals.duplicated()
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

which is equivalent to

```
>>> animals.duplicated(keep='first')
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```
>>> animals.duplicated(keep='last')
0     True
1    False
2     True
3    False
```

(continues on next page)

(continued from previous page)

```
4    False
dtype: bool
```

By setting keep on `False`, all duplicates are True:

```
>>> animals.duplicated(keep=False)
0     True
1    False
2     True
3    False
4     True
dtype: bool
```

pandas.Series.eq

`Series.eq(self, other, level=None, fill_value=None, axis=0)`

Return Equal to of series and other, element-wise (binary operator *eq*).

Equivalent to `series == other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

See also:

`Series.None`

pandas.Series.equals

`Series.equals(self, other)`

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal. The column headers do not need to have the same type, but the elements within the columns must be the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

testing.assert_series_equal Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

testing.assert_frame_equal Like assert_series_equal, but targets DataFrames.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Notes

This function requires that the elements have the same dtype as their respective elements in the other Series or DataFrame. However, the column labels do not need to have the same type, as long as they are still considered equal.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

pandas.Series.ewm

`Series.ewm`(*self*, *com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)

Provide exponential weighted functions.

Parameters

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com)$, for $com \geq 0$.

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1)$, for $span \geq 1$.

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife), \text{for} halflife > 0$.

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).

ignore_na [bool, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. The value 0 identifies the rows, and 1 identifies the columns.

Returns

DataFrame A Window sub-classed for the particular operation.

See also:

rolling Provides rolling window calculations.

expanding Provides expanding transformations.

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), …, 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at https://pandas.pydata.org/pandas-docs/stable/user_guide/computation.html#exponentially-weighted-windows

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

pandas.Series.expanding

`Series.`**`expanding`**(*self*, *min_periods=1*, *center=False*, *axis=0*)

Provide expanding transformations.

Parameters

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [bool, default False] Set the labels at the center of the window.

axis [int or str, default 0]

Returns

a Window sub-classed for the particular operation

See also:

`rolling` Provides rolling window calculations.

`ewm` Provides exponential weighted functions.

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

pandas.Series.explode

Series.**explode**(*self*) → 'Series'

Transform each element of a list-like to a row, replicating the index values.

New in version 0.25.0.

Returns

Series Exploded lists to rows; index will be duplicated for these rows.

See also:

Series.str.split Split string values on specified separator.

Series.unstack Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.

DataFrame.melt Unpivot a DataFrame from wide format to long format.

DataFrame.explode Explode a DataFrame from list-like columns to long format.

Notes

This routine will explode list-likes including lists, tuples, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged. Empty list-likes will result in a np.nan for that row.

Examples

```
>>> s = pd.Series([[1, 2, 3], 'foo', [], [3, 4]])
>>> s
0    [1, 2, 3]
1          foo
2           []
3       [3, 4]
dtype: object
```

```
>>> s.explode()
0      1
0      2
0      3
1    foo
2    NaN
3      3
3      4
dtype: object
```

pandas.Series.factorize

`Series.factorize(self, sort=False, na_sentinel=- 1)`

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

Parameters

sort [bool, default False] Sort *uniques* and shuffle *codes* to maintain the relationship.

na_sentinel [int, default -1] Value to mark "not found".

Returns

codes [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(codes)` will have the same values as *values*.

uniques [ndarray, Index, or Categorical] The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

See also:

`cut` Discretize continuous-valued array.

`unique` Find the unique value in an array.

Examples

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> codes
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *codes* will be shuffled so that the relationship is the maintained.

```
>>> codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> codes
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *codes* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> codes, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> codes
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> codes, uniques = pd.factorize(cat)
>>> codes
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

pandas.Series.ffill

`Series.ffill`(*self: ~ FrameOrSeries*, *axis=None*, *inplace: bool = False*, *limit=None*, *downcast=None*) → Union[~FrameOrSeries, NoneType]

Synonym for `DataFrame.fillna()` with `method='ffill'`.

Returns

%(klass)s or None Object with missing values filled or None if `inplace=True`.

pandas.Series.fillna

`Series.fillna(self, value=None, method=None, axis=None, inplace=False, limit=None, downcast=None) → Union[ForwardRef('Series'), NoneType]`

Fill NA/NaN values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.

axis [{0 or 'index'}] Axis along which to fill missing values.

inplace [bool, default False] If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns

Series or None Object with missing values filled or None if `inplace=True`.

See also:

interpolate Fill NaN values using interpolation.

reindex Conform object to new index.

asfreq Convert TimeSeries to specified frequency.

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

pandas.Series.filter

`Series.filter`(*self: ~ FrameOrSeries*, *items=None*, *like: Union[str, NoneType] = None*, *regex: Union[str, NoneType] = None*, *axis=None*) → ~FrameOrSeries

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items [list-like] Keep labels from axis which are in items.

like [str] Keep labels from axis for which "like in label == True".

regex [str (regular expression)] Keep labels from axis for which re.search(regex, label) == True.

axis [{0 or 'index', 1 or 'columns', None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

same type as input object

See also:

DataFrame.loc

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

pandas.Series.first

`Series.`**`first`**(*self: ~ FrameOrSeries*, *offset*) → ~FrameOrSeries

Method to subset initial periods of time series data based on a date offset.

Parameters

offset [str, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

last Select final periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

pandas.Series.first_valid_index

Series.**first_valid_index**(*self*)

Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

pandas.Series.floordiv

Series.**floordiv**(*self*, *other*, *level=None*, *fill_value=None*, *axis=0*)

Return Integer division of series and other, element-wise (binary operator *floordiv*).

Equivalent to `series // other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

See also:

Series.rfloordiv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.floordiv(b, fill_value=0)
a    1.0
b    NaN
c    NaN
d    0.0
e    NaN
dtype: float64
```

pandas.Series.ge

`Series.ge`(*self*, *other*, *level=None*, *fill_value=None*, *axis=0*)

Return Greater than or equal to of series and other, element-wise (binary operator *ge*).

Equivalent to `series >= other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

See also:

Series.None

pandas.Series.get

`Series.get(self, key, default=None)`

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

pandas.Series.groupby

`Series.groupby(self, by=None, axis=0, level=None, as_index: bool = True, sort: bool = True, group_keys: bool = True, squeeze: bool = False, observed: bool = False) → 'groupby_generic.SeriesGroupBy'`

Group Series using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis [{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1).

level [int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

as_index [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output.

sort [bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

group_keys [bool, default True] When calling apply, add group keys to index to identify pieces.

squeeze [bool, default False] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

observed [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

Returns

SeriesGroupBy Returns a groupby object that contains information about the groups.

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

```
>>> ser = pd.Series([390., 350., 30., 20.],
...                 index=['Falcon', 'Falcon', 'Parrot', 'Parrot'], name="Max␣
↪Speed")
>>> ser
Falcon    390.0
Falcon    350.0
Parrot     30.0
Parrot     20.0
Name: Max Speed, dtype: float64
>>> ser.groupby(["a", "b", "a", "b"]).mean()
a    210.0
b    185.0
Name: Max Speed, dtype: float64
>>> ser.groupby(level=0).mean()
Falcon    370.0
Parrot     25.0
Name: Max Speed, dtype: float64
>>> ser.groupby(ser > 100).mean()
Max Speed
False     25.0
True     370.0
Name: Max Speed, dtype: float64
```

Grouping by Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> ser = pd.Series([390., 350., 30., 20.], index=index, name="Max Speed")
>>> ser
Animal  Type
Falcon  Captive    390.0
        Wild       350.0
Parrot  Captive     30.0
        Wild        20.0
Name: Max Speed, dtype: float64
>>> ser.groupby(level=0).mean()
Animal
Falcon    370.0
Parrot     25.0
Name: Max Speed, dtype: float64
>>> ser.groupby(level="Type").mean()
Type
Captive    210.0
```

(continues on next page)

(continued from previous page)

```
Wild        185.0
Name: Max Speed, dtype: float64
```

pandas.Series.gt

`Series.gt`(*self*, *other*, *level=None*, *fill_value=None*, *axis=0*)

Return Greater than of series and other, element-wise (binary operator *gt*).

Equivalent to `series > other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

Series The result of the operation.

See also:

`Series.None`

pandas.Series.head

`Series.head`(*self: ~ FrameOrSeries*, *n: int = 5*) → ~FrameOrSeries

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of *n*, this function returns all rows except the last *n* rows, equivalent to `df[:-n]`.

Parameters

n [int, default 5] Number of rows to select.

Returns

same type as caller The first *n* rows of the caller object.

See also:

`DataFrame.tail` Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of *n*

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

pandas.Series.hist

`Series.hist`(*self*, *by=None*, *ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *figsize=None*, *bins=10*, *backend=None*, ***kwargs*)

Draw histogram of the input series using matplotlib.

Parameters

by [object, optional] If passed, then used to form histograms for separate groups.

ax [matplotlib axis object] If not passed, uses gca().

grid [bool, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels.

figsize [tuple, default None] Figure size in inches by default.

bins [int or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

backend [str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

New in version 1.0.0.

****kwargs** To be passed to the actual plotting function.

Returns

matplotlib.AxesSubplot A histogram plot.

See also:

`matplotlib.axes.Axes.hist` Plot a histogram using matplotlib.

pandas.Series.idxmax

`Series.`**`idxmax`**(*self*, *axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

Parameters

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

skipna [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

***args, **kwargs** Additional arguments and keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Index Label of the maximum value.

Raises

ValueError If the Series is empty.

See also:

`numpy.argmax` Return indices of the maximum values along the given axis.

`DataFrame.idxmax` Return index of first occurrence of maximum over requested axis.

Series.idxmin Return index *label* of the first occurrence of minimum of values.

Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

pandas.Series.idxmin

`Series.`**`idxmin`**(*self*, *axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

Parameters

axis [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

skipna [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

***args, **kwargs** Additional arguments and keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Index Label of the minimum value.

Raises

ValueError If the Series is empty.

See also:

numpy.argmin Return indices of the minimum values along the given axis.

DataFrame.idxmin Return index of first occurrence of minimum over requested axis.

Series.idxmax Return index *label* of the first occurrence of maximum of values.

Notes

This method is the Series version of `ndarray.argmin`. This method returns the label of the minimum, while `ndarray.argmin` returns the position. To get the position, use `series.values.argmin()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...               index=['A', 'B', 'C', 'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmin(skipna=False)
nan
```

pandas.Series.infer_objects

`Series.`**`infer_objects`**(*self: ~ FrameOrSeries*) → ~FrameOrSeries

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

convert_dtypes Convert argument to best possible dtype.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

pandas.Series.interpolate

`Series.interpolate`(*self*, *method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. These methods use the numerical values of the index. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=5)`.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.23.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all `NaN` values.

See also:

`fillna` Fill missing values using different methods.

`scipy.interpolate.Akima1DInterpolator` Piecewise cubic polynomials (Akima interpolator).

`scipy.interpolate.BPoly.from_derivatives` Piecewise polynomial in the Bernstein basis.

`scipy.interpolate.interp1d` Interpolate a 1-D function.

`scipy.interpolate.KroghInterpolator` Interpolate polynomial (Krogh interpolator).

`scipy.interpolate.PchipInterpolator` PCHIP 1-d monotonic cubic interpolation.

`scipy.interpolate.CubicSpline` Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in `NaN` in a *`Series`* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```