Filling in `NaN` in a Series by padding, but filling at most two consecutive `NaN` at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
```

(continues on next page)

(continued from previous page)

```
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

pandas.Series.isin

Series.**isin**(*self*, *values*)

Check whether *values* are contained in Series.

Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

Parameters

values [set or list-like] The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a list of one element.

Returns

Series Series of booleans indicating if each element is in values.

Raises

TypeError

- If *values* is a string

See also:

DataFrame.isin Equivalent method on DataFrame.

Examples

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...                'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0     True
1     True
2     True
3    False
4     True
5    False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['lama'])
0     True
1    False
2     True
3    False
4     True
5    False
Name: animal, dtype: bool
```

pandas.Series.isna

Series.**isna**(*self*)

Detect missing values.

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

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull Alias of isna.

Series.notna Boolean inverse of isna.

Series.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Series.isnull

Series.**isnull**(*self*)

Detect missing values.

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

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull Alias of isna.

Series.notna Boolean inverse of isna.

Series.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
```

(continues on next page)

(continued from previous page)

```
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Series.item

Series.**item**(*self*)

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

Returns

scalar The first element of %(klass)s.

Raises

ValueError If the data is not length-1.

pandas.Series.items

Series.**items**(*self*)

Lazily iterate over (index, value) tuples.

This method returns an iterable tuple (index, value). This is convenient if you want to create a lazy iterator.

Returns

iterable Iterable of tuples containing the (index, value) pairs from a Series.

See also:

DataFrame.items Iterate over (column name, Series) pairs.

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

Examples

```
>>> s = pd.Series(['A', 'B', 'C'])
>>> for index, value in s.items():
...     print(f"Index : {index}, Value : {value}")
Index : 0, Value : A
Index : 1, Value : B
Index : 2, Value : C
```

pandas.Series.iteritems

Series.**iteritems**(*self*)

Lazily iterate over (index, value) tuples.

This method returns an iterable tuple (index, value). This is convenient if you want to create a lazy iterator.

Returns

iterable Iterable of tuples containing the (index, value) pairs from a Series.

See also:

DataFrame.items Iterate over (column name, Series) pairs.

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

Examples

```
>>> s = pd.Series(['A', 'B', 'C'])
>>> for index, value in s.items():
...     print(f"Index : {index}, Value : {value}")
Index : 0, Value : A
Index : 1, Value : B
Index : 2, Value : C
```

pandas.Series.keys

Series.**keys**(*self*)

Return alias for index.

Returns

Index Index of the Series.

pandas.Series.kurt

Series.**kurt**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.kurtosis

Series.**kurtosis**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis.

Kurtosis obtained using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.last

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

Method to subset final 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:

first Select initial 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 last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

pandas.Series.last_valid_index

Series.**last_valid_index**(*self*)

Return index for last 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.le

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

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

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.lt

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

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

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.mad

Series.**mad**(*self*, *axis=None*, *skipna=None*, *level=None*)

Return the mean absolute deviation of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.map

Series.**map**(*self*, *arg*, *na_action=None*)

Map values of Series according to input correspondence.

Used for substituting each value in a Series with another value, that may be derived from a function, a `dict` or a *Series*.

Parameters

arg [function, collections.abc.Mapping subclass or Series] Mapping correspondence.

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

Returns

Series Same index as caller.

See also:

Series.apply For applying more complex functions on a Series.

DataFrame.apply Apply a function row-/column-wise.

DataFrame.applymap Apply a function elementwise on a whole DataFrame.

Notes

When `arg` is a dictionary, values in Series that are not in the dictionary (as keys) are converted to `NaN`. However, if the dictionary is a `dict` subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than `NaN`.

Examples

```
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])
>>> s
0      cat
1      dog
2      NaN
3   rabbit
dtype: object
```

`map` accepts a `dict` or a `Series`. Values that are not found in the `dict` are converted to `NaN`, unless the dict has a default value (e.g. `defaultdict`):

```
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0   kitten
1    puppy
2      NaN
3      NaN
dtype: object
```

It also accepts a function:

```
>>> s.map('I am a {}'.format)
0       I am a cat
1       I am a dog
2       I am a nan
3    I am a rabbit
dtype: object
```

To avoid applying the function to missing values (and keep them as `NaN`) `na_action='ignore'` can be used:

```
>>> s.map('I am a {}'.format, na_action='ignore')
0     I am a cat
1     I am a dog
2            NaN
3  I am a rabbit
dtype: object
```

pandas.Series.mask

`Series.`**`mask`**(*self*, *cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*)

Replace values where the condition is True.

Parameters

cond [bool Series/DataFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other [scalar, Series/DataFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it).

inplace [bool, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default 'raise'] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- 'raise' : allow exceptions to be raised.
- 'ignore' : suppress exceptions. On error return original object.

try_cast [bool, default False] Try to cast the result back to the input type (if possible).

Returns

Same type as caller

See also:

`DataFrame.where()` Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for *`DataFrame.where()`* differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
```

(continues on next page)

(continued from previous page)

```
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.Series.max

Series.**max**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the maximum of the values for the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.sum Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

Max using level names, as well as indices.

```
>>> s.max(level='blooded')
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

```
>>> s.max(level=0)
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

pandas.Series.mean

Series.**mean**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.median

Series.**median**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

pandas.Series.memory_usage

Series.**memory_usage**(*self*, *index=True*, *deep=False*)

Return the memory usage of the Series.

The memory usage can optionally include the contribution of the index and of elements of *object* dtype.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the Series index.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned value.

Returns

int Bytes of memory consumed.

See also:

`numpy.ndarray.nbytes` Total bytes consumed by the elements of the array.

`DataFrame.memory_usage` Bytes consumed by a DataFrame.

Examples

```
>>> s = pd.Series(range(3))
>>> s.memory_usage()
152
```

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```
>>> s.memory_usage(index=False)
24
```

The memory footprint of *object* values is ignored by default:

```
>>> s = pd.Series(["a", "b"])
>>> s.values
array(['a', 'b'], dtype=object)
>>> s.memory_usage()
144
>>> s.memory_usage(deep=True)
260
```

pandas.Series.min

`Series.`**`min`**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the minimum of the values for the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

scalar or Series (if level specified)

See also:

`Series.sum` Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.sum Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

Min using level names, as well as indices.

```
>>> s.min(level='blooded')
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

```
>>> s.min(level=0)
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

pandas.Series.mod

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

Return Modulo of series and other, element-wise (binary operator *mod*).

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.rmod

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.mod(b, fill_value=0)
a    0.0
b    NaN
c    NaN
d    0.0
e    NaN
dtype: float64
```

pandas.Series.mode

Series.**mode**(*self*, *dropna=True*)

Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

Parameters

dropna [bool, default True] Don't consider counts of NaN/NaT.

New in version 0.24.0.

Returns

Series Modes of the Series in sorted order.

pandas.Series.mul

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

Return Multiplication of series and other, element-wise (binary operator *mul*).

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.rmul

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
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> a.multiply(b, fill_value=0)
a    1.0
b    0.0
c    0.0
d    0.0
e    NaN
dtype: float64
```

pandas.Series.multiply

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

Return Multiplication of series and other, element-wise (binary operator *mul*).

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.rmul

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.multiply(b, fill_value=0)
a    1.0
b    0.0
c    0.0
```

(continues on next page)

(continued from previous page)

```
d    0.0
e    NaN
dtype: float64
```

pandas.Series.ne

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

Return Not equal to of series and other, element-wise (binary operator *ne*).

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.nlargest

Series.**nlargest**(*self*, *n=5*, *keep='first'*)

Return the largest *n* elements.

Parameters

n [int, default 5] Return this many descending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- **first** [return the first *n* occurrences in order] of appearance.
- **last** [return the last *n* occurrences in reverse] order of appearance.
- **all** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* largest values in the Series, sorted in decreasing order.

See also:

Series.nsmallest Get the *n* smallest elements.

Series.sort_values Sort Series by values.

Series.head Return the first *n* rows.

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Malta": 434000, "Maldives": 434000,
...                         "Brunei": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Malta         434000
Maldives      434000
Brunei        434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Monserat        5200
dtype: int64
```

The *n* largest elements where `n=5` by default.

```
>>> s.nlargest()
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3`. Default *keep* value is 'first' so Malta will be kept.

```
>>> s.nlargest(3)
France    65000000
Italy     59000000
Malta       434000
dtype: int64
```

The *n* largest elements where `n=3` and keeping the last duplicates. Brunei will be kept since it is the last with value 434000 based on the index order.

```
>>> s.nlargest(3, keep='last')
France      65000000
Italy       59000000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3` with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```
>>> s.nlargest(3, keep='all')
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

pandas.Series.notna

Series.**notna**(*self*)

Detect existing (non-missing) values.

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

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.notnull Alias of notna.

Series.isna Boolean inverse of notna.

Series.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Series.notnull

Series.**notnull**(*self*)

Detect existing (non-missing) values.

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

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.notnull Alias of notna.

Series.isna Boolean inverse of notna.

Series.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
```

(continues on next page)

(continued from previous page)

```
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Series.nsmallest

Series.**nsmallest**(*self*, *n=5*, *keep='first'*)

Return the smallest *n* elements.

Parameters

n [int, default 5] Return this many ascending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- **first** [return the first *n* occurrences in order] of appearance.
- **last** [return the last *n* occurrences in reverse] order of appearance.
- **all** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* smallest values in the Series, sorted in increasing order.

See also:

Series.nlargest Get the *n* largest elements.

Series.sort_values Sort Series by values.

Series.head Return the first *n* rows.