

Data Cleaning

Python basics

Kunal Khurana

2024-02-23

Table of contents

Handing missing data	2
Data Transformation	3
Extension data types	3
String Manipulation	3
Categorical Data	3
pandas.isna - other methods	4
filling in missing data	7
imputations with fillna()	343
Data Transformation	345
Removing duplicates	345
Transforming data using a Function or mapping	346
Replacing values	348
Renaming Axis indexes	349
Discretization and Binning	350
Detecting and Filtering Outliers	353
Permutation and Random Sampling	355
Computing Indicator/Dummay Variables	356
Extension data types	358
String manipulation	360
Regular expressions	361
String functions in pandas	363
Categorical data	364
Categorical Extension type in pandas	365
Categorical data	367
Computations with categoricals	368
using groupby for summary statistics	369
Better performance with categoricals	369
Categorical Methods	370
Creating dummy variables for modelling	372

Handing missing data

- Filtering

- Filling

Data Transformation

- Removing duplicates
- Transforming data using function or Mapping
- Replacing values
- Renaming Axis Indexes
- Discretization and Binning
- Detecting and Filtering Outliers
- Permutation and Random Sampling
- Computing Indicator/Dummy variables

Extension data types

String Manipulation

- Regular expressions
- String functions in Pandas

Categorical Data

- Background
- types
- computations with categoricals
- Better performance with categoricals
- Categorical methods
- Creating dummy variables for modeling

```
import pandas as pd
import numpy as np
```

```
float_data = pd.Series([1.2, -3.5, np.nan, 0])
float_data
```

```
0    1.2
1   -3.5
2    NaN
3    0.0
dtype: float64
```

pandas.isna - other methods

```
# checking for nan values with booleans
float_data.isna()
```

```
0    False
1    False
2     True
3    False
dtype: bool
```

```
### filtering out missing data
float_data.dropna()
```

```
0    1.2
1   -3.5
3    0.0
dtype: float64
```

```
### or with notna()
float_data[float_data.notna()]
```

```
0    1.2
1   -3.5
3    0.0
dtype: float64
```

```
data = pd.DataFrame([[1., 6.5, 3., 4],
                     [1., np.nan, np.nan, 4],
                     [3, 4, 22, np.nan],
                     [np.nan, 434, 33, 1]])
```

```
data
```

	0	1	2	3
0	1.0	6.5	3.0	4.0
1	1.0	NaN	NaN	4.0
2	3.0	4.0	22.0	NaN
3	NaN	434.0	33.0	1.0

```
data.dropna()
```

	0	1	2	3
0	1.0	6.5	3.0	4.0

```
# how = 'all' will drop all rows taht are all NA  
data.dropna(how='all')
```

	0	1	2	3
0	1.0	6.5	3.0	4.0
1	1.0	NaN	NaN	4.0
2	3.0	4.0	22.0	NaN
3	NaN	434.0	33.0	1.0

```
# dropping columns by how= all
```

```
data[4] = np.nan  
data
```

	0	1	2	3	4
0	1.0	6.5	3.0	4.0	NaN
1	1.0	NaN	NaN	4.0	NaN
2	3.0	4.0	22.0	NaN	NaN
3	NaN	434.0	33.0	1.0	NaN

```
data.dropna(axis = "columns", how="all")
```

	0	1	2	3
0	1.0	6.5	3.0	4.0
1	1.0	NaN	NaN	4.0
2	3.0	4.0	22.0	NaN
3	NaN	434.0	33.0	1.0

```
df = pd.DataFrame(np.random.standard_normal((7, 3)))
```

```
df
```

	0	1	2
0	-1.716945	1.430864	0.198477
1	2.815565	-0.425012	-1.749359
2	-0.073905	-0.618281	0.826025
3	1.122968	2.883936	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
# make null- first four rows of second column
```

```
df.iloc[:4, 1] = np.nan
```

```
df
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
df.dropna()
```

	0	1	2
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
# to learn more about this method  
help(df.dropna())
```

```
#ça marche pas
```

```
df.dropna(thresh=2)
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

filling in missing data

```
df.fillna(0)
```

	0	1	2
0	-1.716945	0.000000	0.198477
1	2.815565	0.000000	-1.749359
2	-0.073905	0.000000	0.826025
3	1.122968	0.000000	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
# using different fillvalue for each column
```

```
df.fillna({1:0.5})
```

	0	1	2
0	-1.716945	0.500000	0.198477
1	2.815565	0.500000	-1.749359
2	-0.073905	0.500000	0.826025
3	1.122968	0.500000	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
df
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
# same interpolation using fillna()
```

```
df.fillna(method = "ffill").astype(float)
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841


```
! python --version
```

Python 3.9.13

```
help(df)
```

```
boxplot = df.boxplot()
```

```
import matplotlib as plt
```

```
%plt.inline.boxplot.show()
```

```
help(pd.Series)
```

Help on class Series in module pandas.core.series:

```
class Series(pandas.core.base.IndexOpsMixin, pandas.core.generic.NDFrame)
| Series(data=None, index=None, dtype: 'Dtype | None' = None, name=None, copy: 'bool | None' = True)
|
| One-dimensional ndarray with axis labels (including time series).
|
| Labels need not be unique but must be a hashable type. The object
| supports both integer- and label-based indexing and provides a host of
| methods for performing operations involving the index. Statistical
| methods from ndarray have been overridden to automatically exclude
| missing data (currently represented as NaN).
|
| Operations between Series (+, -, /, \*, \*\*) align values based on their
| associated index values-- they need not be the same length. The result
| index will be the sorted union of the two indexes.
|
| Parameters
| -----
| data : array-like, Iterable, dict, or scalar value
|     Contains data stored in Series. If data is a dict, argument order is
|     maintained.
| index : array-like or Index (1d)
|     Values must be hashable and have the same length as `data`.
|     Non-unique index values are allowed. Will default to
```

```

|     RangeIndex (0, 1, 2, ..., n) if not provided. If data is dict-like
|     and index is None, then the keys in the data are used as the index. If the
|     index is not None, the resulting Series is reindexed with the index values.
| dtype : str, numpy.dtype, or ExtensionDtype, optional
|     Data type for the output Series. If not specified, this will be
|     inferred from `data`.
|     See the :ref:`user guide <basics.dtypes>` for more usages.
| name : Hashable, default None
|     The name to give to the Series.
| copy : bool, default False
|     Copy input data. Only affects Series or 1d ndarray input. See examples.
|
| Notes
| -----
| Please reference the :ref:`User Guide <basics.series>` for more information.
|
| Examples
| -----
| Constructing Series from a dictionary with an Index specified
|
| >>> d = {'a': 1, 'b': 2, 'c': 3}
| >>> ser = pd.Series(data=d, index=['a', 'b', 'c'])
| >>> ser
| a    1
| b    2
| c    3
| dtype: int64
|
| The keys of the dictionary match with the Index values, hence the Index
| values have no effect.
|
| >>> d = {'a': 1, 'b': 2, 'c': 3}
| >>> ser = pd.Series(data=d, index=['x', 'y', 'z'])
| >>> ser
| x    NaN
| y    NaN
| z    NaN
| dtype: float64
|
| Note that the Index is first build with the keys from the dictionary.
| After this the Series is reindexed with the given Index values, hence we
| get all NaN as a result.
|

```

| Constructing Series from a list with `copy=False`.

```
| >>> r = [1, 2]
| >>> ser = pd.Series(r, copy=False)
| >>> ser.iloc[0] = 999
| >>> r
| [1, 2]
| >>> ser
| 0    999
| 1     2
| dtype: int64
```

| Due to input data type the Series has a `copy` of the original data even though `copy=False`, so the data is unchanged.

| Constructing Series from a 1d ndarray with `copy=False`.

```
| >>> r = np.array([1, 2])
| >>> ser = pd.Series(r, copy=False)
| >>> ser.iloc[0] = 999
| >>> r
| array([999,  2])
| >>> ser
| 0    999
| 1     2
| dtype: int64
```

| Due to input data type the Series has a `view` on the original data, so the data is changed as well.

```
| Method resolution order:
|   Series
|   pandas.core.base.IndexOpsMixin
|   pandas.core.arraylike.OpsMixin
|   pandas.core.generic.NDFrame
|   pandas.core.base.PandasObject
|   pandas.core.accessor.DirNamesMixin
|   pandas.core.indexing.IndexingMixin
|   builtins.object
```

| Methods defined here:

```

|
|  __array__(self, dtype: 'npt.DTypeLike | None' = None) -> 'np.ndarray'
|      Return the values as a NumPy array.
|
|
|  Users should not call this directly. Rather, it is invoked by
|  :func:`numpy.array` and :func:`numpy.asarray`.
|
|
|  Parameters
|  -----
|
|  dtype : str or numpy.dtype, optional
|          The dtype to use for the resulting NumPy array. By default,
|          the dtype is inferred from the data.
|
|
|  Returns
|  -----
|
|  numpy.ndarray
|          The values in the series converted to a :class:`numpy.ndarray`
|          with the specified `dtype`.
|
|
|  See Also
|  -----
|
|  array : Create a new array from data.
|  Series.array : Zero-copy view to the array backing the Series.
|  Series.to_numpy : Series method for similar behavior.
|
|
|  Examples
|  -----
|
|  >>> ser = pd.Series([1, 2, 3])
|  >>> np.asarray(ser)
|  array([1, 2, 3])
|
|
|  For timezone-aware data, the timezones may be retained with
|  ``dtype='object'``
|
|
|  >>> tzser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
|  >>> np.asarray(tzser, dtype="object")
|  array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
|         Timestamp('2000-01-02 00:00:00+0100', tz='CET')],
|         dtype=object)
|
|
|  Or the values may be localized to UTC and the tzinfo discarded with
|  ``dtype='datetime64[ns]'``
|
|

```

```

|     >>> np.asarray(tzser, dtype="datetime64[ns]") # doctest: +ELLIPSIS
|     array(['1999-12-31T23:00:00.000000000', ...],
|           dtype='datetime64[ns]')
|
|     __float__(self)
|
|     __getitem__(self, key)
|
|     __init__(self, data=None, index=None, dtype: 'Dtype | None' = None, name=None, copy: 'bo
|           Initialize self. See help(type(self)) for accurate signature.
|
|     __int__(self)
|
|     __len__(self) -> 'int'
|           Return the length of the Series.
|
|     __matmul__(self, other)
|           Matrix multiplication using binary `@` operator in Python>=3.5.
|
|     __repr__(self) -> 'str'
|           Return a string representation for a particular Series.
|
|     __rmatmul__(self, other)
|           Matrix multiplication using binary `@` operator in Python>=3.5.
|
|     __setitem__(self, key, value) -> 'None'
|
|     add(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|           Return Addition of series and other, element-wise (binary operator `add`).
|
|           Equivalent to ``series + other``, but with support to substitute a fill_value for
|           missing data in either one of the inputs.
|
|           Parameters
|           -----
|           other : Series or scalar value
|           level : int or name
|                   Broadcast across a level, matching Index values on the
|                   passed MultiIndex level.
|           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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
| Returns
| -----
| Series
|     The result of the operation.
|
| See Also
| -----
| Series.radd : Reverse of the Addition operator, see
|     `Python documentation
|     <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|     for more details.
|
| 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.add(b, fill_value=0)
| a    2.0
| b    1.0
| c    1.0
| d    1.0
| e    NaN
| dtype: float64
|
| agg = aggregate(self, func=None, axis: 'Axis' = 0, *args, **kwargs)
|
| aggregate(self, func=None, axis: 'Axis' = 0, *args, **kwargs)

```

Aggregate using one or more operations over the specified axis.

Parameters

`func` : function, str, list or dict

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[[np.sum, 'mean']]`
- dict of axis labels -> functions, function names or list of such.

`axis` : {0 or 'index'}

Unused. Parameter needed for compatibility with DataFrame.

`*args`

Positional arguments to pass to `func`.

`**kwargs`

Keyword arguments to pass to `func`.

Returns

scalar, Series or DataFrame

The return can be:

- * scalar : when Series.agg is called with single function
- * Series : when DataFrame.agg is called with a single function
- * DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

See Also

`Series.apply` : Invoke function on a Series.

`Series.transform` : Transform function producing a Series with like indexes.

Notes

`agg` is an alias for `aggregate`. Use the alias.

Functions that mutate the passed object can produce unexpected

behavior or errors and are not supported. See :ref:`gotchas.udf-mutation` for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
```

```
>>> s
```

```
0    1
```

```
1    2
```

```
2    3
```

```
3    4
```

```
dtype: int64
```

```
>>> s.agg('min')
```

```
1
```

```
>>> s.agg(['min', 'max'])
```

```
min    1
```

```
max    4
```

```
dtype: int64
```

`align(self, other: 'Series', join: 'AlignJoin' = 'outer', axis: 'Axis | None' = None, level: 'int | None' = None)`
Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

Parameters

`other` : DataFrame or Series

`join` : {'outer', 'inner', 'left', 'right'}, default 'outer'

`axis` : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None).

`level` : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level.

`copy` : bool, default True

Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

`fill_value` : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value.


```

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.

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.
fill_axis : {0 or 'index'}, default 0
        Filling axis, method and limit.
broadcast_axis : {0 or 'index'}, default None
        Broadcast values along this axis, if aligning two objects of
        different dimensions.

Returns
-----
tuple of (Series, type of other)
        Aligned objects.

Examples
-----
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1, 2]
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   D  B  E  A
1  1  2  3  4
2  6  7  8  9
>>> other
   A   B   C   D
2  10  20  30  40
3  60  70  80  90
4  600 700 800 900

```

```

|
| Align on columns:
|
| >>> left, right = df.align(other, join="outer", axis=1)
| >>> left
|      A  B  C  D  E
| 1  4  2 NaN 1  3
| 2  9  7 NaN 6  8
| >>> right
|      A    B    C    D    E
| 2  10   20   30   40 NaN
| 3  60   70   80   90 NaN
| 4 600  700  800  900 NaN
|
| We can also align on the index:
|
| >>> left, right = df.align(other, join="outer", axis=0)
| >>> left
|      D    B    E    A
| 1  1.0  2.0  3.0  4.0
| 2  6.0  7.0  8.0  9.0
| 3  NaN  NaN  NaN  NaN
| 4  NaN  NaN  NaN  NaN
| >>> right
|      A    B    C    D
| 1   NaN  NaN  NaN  NaN
| 2  10.0  20.0  30.0  40.0
| 3  60.0  70.0  80.0  90.0
| 4 600.0 700.0 800.0 900.0
|
| Finally, the default `axis=None` will align on both index and columns:
|
| >>> left, right = df.align(other, join="outer", axis=None)
| >>> left
|      A  B  C  D  E
| 1  4.0  2.0 NaN 1.0 3.0
| 2  9.0  7.0 NaN 6.0 8.0
| 3  NaN  NaN NaN NaN NaN
| 4  NaN  NaN NaN NaN NaN
| >>> right
|      A    B    C    D    E
| 1   NaN  NaN  NaN  NaN NaN
| 2  10.0  20.0  30.0  40.0 NaN

```

```

|      3  60.0  70.0  80.0  90.0 NaN
|      4 600.0 700.0 800.0 900.0 NaN
|
| all(self, axis: 'Axis' = 0, bool_only=None, skipna: 'bool_t' = True, **kwargs)
|     Return whether all elements are True, potentially over an axis.
|
| Returns True unless there at least one element within a series or
| along a Dataframe axis that is False or equivalent (e.g. zero or
| empty).
|
| Parameters
| -----
| axis : {0 or 'index', 1 or 'columns', None}, default 0
|       Indicate which axis or axes should be reduced. For `Series` this parameter
|       is unused and defaults to 0.
|
|       * 0 / 'index' : reduce the index, return a Series whose index is the
|         original column labels.
|       * 1 / 'columns' : reduce the columns, return a Series whose index is the
|         original index.
|       * None : reduce all axes, return a scalar.
|
| bool_only : bool, default None
|           Include only boolean columns. If None, will attempt to use everything,
|           then use only boolean data. Not implemented for Series.
| skipna : bool, default True
|         Exclude NA/null values. If the entire row/column is NA and skipna is
|         True, then the result will be True, as for an empty row/column.
|         If skipna is False, then NA are treated as True, because these are not
|         equal to zero.
| **kwargs : any, default None
|           Additional keywords have no effect but might be accepted for
|           compatibility with NumPy.
|
| Returns
| -----
| scalar or Series
|       If level is specified, then, Series is returned; otherwise, scalar
|       is returned.
|
| See Also
| -----
| Series.all : Return True if all elements are True.

```

DataFrame.any : Return True if one (or more) elements are True.

Examples

****Series****

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

****DataFrames****

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1  col2
0  True  True
1  True False
```

Default behaviour checks if values in each column all return True.

```
>>> df.all()
col1    True
col2   False
dtype: bool
```

Specify ``axis='columns'`` to check if values in each row all return True.

```
>>> df.all(axis='columns')
0    True
1   False
dtype: bool
```

Or ``axis=None`` for whether every value is True.

```

|     >>> df.all(axis=None)
|     False
|
|
| any(self, *, axis: 'Axis' = 0, bool_only=None, skipna: 'bool_t' = True, **kwargs)
|     Return whether any element is True, potentially over an axis.
|
|     Returns False unless there is at least one element within a series or
|     along a Dataframe axis that is True or equivalent (e.g. non-zero or
|     non-empty).
|
|     Parameters
|     -----
|
|     axis : {0 or 'index', 1 or 'columns', None}, default 0
|         Indicate which axis or axes should be reduced. For `Series` this parameter
|         is unused and defaults to 0.
|
|         * 0 / 'index' : reduce the index, return a Series whose index is the
|           original column labels.
|         * 1 / 'columns' : reduce the columns, return a Series whose index is the
|           original index.
|         * None : reduce all axes, return a scalar.
|
|     bool_only : bool, default None
|         Include only boolean columns. If None, will attempt to use everything,
|         then use only boolean data. Not implemented for Series.
|     skipna : bool, default True
|         Exclude NA/null values. If the entire row/column is NA and skipna is
|         True, then the result will be False, as for an empty row/column.
|         If skipna is False, then NA are treated as True, because these are not
|         equal to zero.
|     **kwargs : any, default None
|         Additional keywords have no effect but might be accepted for
|         compatibility with NumPy.
|
|     Returns
|     -----
|
|     scalar or Series
|         If level is specified, then, Series is returned; otherwise, scalar
|         is returned.
|
|     See Also
|     -----
|
|     numpy.any : Numpy version of this method.

```

Series.any : Return whether any element is True.
Series.all : Return whether all elements are True.
DataFrame.any : Return whether any element is True over requested axis.
DataFrame.all : Return whether all elements are True over requested axis.

Examples

----- **Series**

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0

>>> df.any()
A     True
B     True
C     False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
```


preserved for some extension array dtypes, such as Categorical.

args : tuple

Positional arguments passed to func after the series value.

**kwargs

Additional keyword arguments passed to func.

Returns

Series or DataFrame

If func returns a Series object the result will be a DataFrame.

See Also

Series.map: For element-wise operations.

Series.agg: Only perform aggregating type operations.

Series.transform: Only perform transforming type operations.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See :ref:`gotchas.udf-mutation` for more details.

Examples

Create a series with typical summer temperatures for each city.

```
>>> s = pd.Series([20, 21, 12],
...               index=['London', 'New York', 'Helsinki'])
>>> s
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to ``apply``.

```
>>> def square(x):
...     return x ** 2
>>> s.apply(square)
London      400
New York    441
```



```
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> s.apply(lambda x: x ** 2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x - custom_value
```

```
>>> s.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply()`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x += kwargs[month]
...     return x
```

```
>>> s.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> s.apply(np.log)
London      2.995732
```

```

|     New York      3.044522
|     Helsinki     2.484907
|     dtype: float64
|
| argsort(self, axis: 'Axis' = 0, kind: 'SortKind' = 'quicksort', order: 'None' = None) ->
|     Return the integer indices that would sort the Series values.
|
|     Override ndarray.argsort. Argsorts the value, omitting NA/null values,
|     and places the result in the same locations as the non-NA values.
|
|     Parameters
|     -----
|     axis : {0 or 'index'}
|           Unused. Parameter needed for compatibility with DataFrame.
|     kind : {'mergesort', 'quicksort', 'heapsort', 'stable'}, default 'quicksort'
|           Choice of sorting algorithm. See :func:`numpy.sort` for more
|           information. 'mergesort' and 'stable' are the only stable algorithms.
|     order : None
|           Has no effect but is accepted for compatibility with numpy.
|
|     Returns
|     -----
|     Series[np.intp]
|           Positions of values within the sort order with -1 indicating
|           nan values.
|
|     See Also
|     -----
|     numpy.ndarray.argsort : Returns the indices that would sort this array.
|
| asfreq(self, freq: 'Frequency', method: 'FillnaOptions | None' = None, how: 'str | None'
|     Convert time series to specified frequency.
|
|     Returns the original data conformed to a new index with the specified
|     frequency.
|
|     If the index of this Series is a :class:`~pandas.PeriodIndex`, the new index
|     is the result of transforming the original index with
|     :meth:`~PeriodIndex.asfreq <pandas.PeriodIndex.asfreq>` (so the original index
|     will map one-to-one to the new index).
|
|     Otherwise, the new index will be equivalent to ``pd.date_range(start, end,
|     freq=freq)`` where ``start`` and ``end`` are, respectively, the first and

```

last entries in the original index (see `:func:`pandas.date_range``). The values corresponding to any timesteps in the new index which were not present in the original index will be null (``NaN``), unless a method for filling such unknowns is provided (see the ``method`` parameter below).

The `:meth:`resample`` method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

Parameters

`freq` : DateOffset or str
Frequency DateOffset or string.

`method` : {'backfill'/'bfill', 'pad'/'ffill'}, default None
Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- * 'pad' / 'ffill': propagate last valid observation forward to next valid
- * 'backfill' / 'bfill': use NEXT valid observation to fill.

`how` : {'start', 'end'}, default end
For PeriodIndex only (see `PeriodIndex.asfreq`).

`normalize` : bool, default False
Whether to reset output index to midnight.

`fill_value` : scalar, optional
Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns

Series
Series object reindexed to the specified frequency.

See Also

`reindex` : Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see ``this link``
<https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-alias>

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
```

```
          s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00   NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
          s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30   NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a ``fill value``.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
          s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a ``method``.

```
>>> df.asfreq(freq='30S', method='bfill')
          s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
```

```

|      2000-01-01 00:01:00    NaN
|      2000-01-01 00:01:30    2.0
|      2000-01-01 00:02:00    2.0
|      2000-01-01 00:02:30    3.0
|      2000-01-01 00:03:00    3.0
|
| autocorr(self, lag: 'int' = 1) -> 'float'
|     Compute the lag-N autocorrelation.
|
|     This method computes the Pearson correlation between
|     the Series and its shifted self.
|
|     Parameters
|     -----
|     lag : int, default 1
|           Number of lags to apply before performing autocorrelation.
|
|     Returns
|     -----
|     float
|           The Pearson correlation between self and self.shift(lag).
|
|     See Also
|     -----
|     Series.corr : Compute the correlation between two Series.
|     Series.shift : Shift index by desired number of periods.
|     DataFrame.corr : Compute pairwise correlation of columns.
|     DataFrame.corrwith : Compute pairwise correlation between rows or
|           columns of two DataFrame objects.
|
|     Notes
|     -----
|     If the Pearson correlation is not well defined return 'NaN'.
|
|     Examples
|     -----
|     >>> s = pd.Series([0.25, 0.5, 0.2, -0.05])
|     >>> s.autocorr() # doctest: +ELLIPSIS
|     0.10355...
|     >>> s.autocorr(lag=2) # doctest: +ELLIPSIS
|     -0.99999...
|
|     If the Pearson correlation is not well defined, then 'NaN' is returned.

```

```

|
| >>> s = pd.Series([1, 0, 0, 0])
| >>> s.autocorr()
| nan
|
| between(self, left, right, inclusive: "Literal['both', 'neither', 'left', 'right']" = 'b
| Return boolean Series equivalent to left <= series <= right.
|
| This function returns a boolean vector containing `True` wherever the
| corresponding Series element is between the boundary values `left` and
| `right`. NA values are treated as `False`.
|
| Parameters
| -----
| left : scalar or list-like
|     Left boundary.
| right : scalar or list-like
|     Right boundary.
| inclusive : {"both", "neither", "left", "right"}
|     Include boundaries. Whether to set each bound as closed or open.
|
| .. versionchanged:: 1.3.0
|
| Returns
| -----
| Series
|     Series representing whether each element is between left and
|     right (inclusive).
|
| See Also
| -----
| Series.gt : Greater than of series and other.
| Series.lt : Less than of series and other.
|
| Notes
| -----
| This function is equivalent to `` (left <= ser) & (ser <= right) ``
|
| Examples
| -----
| >>> s = pd.Series([2, 0, 4, 8, np.nan])
|
| Boundary values are included by default:

```

```

|
| >>> s.between(1, 4)
| 0    True
| 1    False
| 2    True
| 3    False
| 4    False
| dtype: bool
|
| With `inclusive` set to ``"neither"`` boundary values are excluded:
|
| >>> s.between(1, 4, inclusive="neither")
| 0    True
| 1    False
| 2    False
| 3    False
| 4    False
| dtype: bool
|
| `left` and `right` can be any scalar value:
|
| >>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
| >>> s.between('Anna', 'Daniel')
| 0    False
| 1    True
| 2    True
| 3    False
| dtype: bool
|
| bfill(self, *, axis: 'None | Axis' = None, inplace: 'bool' = False, limit: 'None | int' =
|     Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.
|
| Returns
| -----
| Series/DataFrame or None
|     Object with missing values filled or None if ``inplace=True``.
|
| clip(self: 'Series', lower=None, upper=None, *, axis: 'Axis | None' = None, inplace: 'bo
|     Trim values at input threshold(s).
|
| Assigns values outside boundary to boundary values. Thresholds
| can be singular values or array like, and in the latter case
| the clipping is performed element-wise in the specified axis.

```

Parameters

`lower` : float or array-like, default None
Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g. `NA`) will not clip the value.

`upper` : float or array-like, default None
Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g. `NA`) will not clip the value.

`axis` : {{0 or 'index', 1 or 'columns', None}}, default None
Align object with lower and upper along the given axis.
For `Series` this parameter is unused and defaults to `None`.

`inplace` : bool, default False
Whether to perform the operation in place on the data.

`*args, **kwargs`
Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame or None
Same type as calling object with the values outside the clip boundaries replaced or None if `inplace=True`.

See Also

`Series.clip` : Trim values at input threshold in series.
`DataFrame.clip` : Trim values at input threshold in dataframe.
`numpy.clip` : Clip (limit) the values in an array.

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```


Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

Clips using specific lower threshold per column element, with missing values:

```
>>> t = pd.Series([2, -4, np.NaN, 6, 3])
>>> t
0    2.0
1   -4.0
2    NaN
3    6.0
4    3.0
dtype: float64
```

```
>>> df.clip(t, axis=0)
   col_0  col_1
```

```

0      9      2
1     -3     -4
2      0      6
3      6      8
4      5      3

```

```

combine(self, other: 'Series | Hashable', func: 'Callable[[Hashable, Hashable], Hashable]')
Combine the Series with a Series or scalar according to `func`.

```

```

Combine the Series and `other` using `func` to perform elementwise
selection for combined Series.
`fill_value` is assumed when value is missing at some index
from one of the two objects being combined.

```

Parameters

```

-----

```

```

other : Series or scalar

```

```

    The value(s) to be combined with the `Series`.

```

```

func : function

```

```

    Function that takes two scalars as inputs and returns an element.

```

```

fill_value : scalar, optional

```

```

    The value to assume when an index is missing from
    one Series or the other. The default specifies to use the
    appropriate NaN value for the underlying dtype of the Series.

```

Returns

```

-----

```

```

Series

```

```

    The result of combining the Series with the other object.

```

See Also

```

-----

```

```

Series.combine_first : Combine Series values, choosing the calling
Series' values first.

```

Examples

```

-----

```

```

Consider 2 Datasets ``s1`` and ``s2`` containing
highest clocked speeds of different birds.

```

```

>>> s1 = pd.Series({'falcon': 330.0, 'eagle': 160.0})
>>> s1
falcon    330.0

```

```

eagle      160.0
dtype: float64
>>> s2 = pd.Series({'falcon': 345.0, 'eagle': 200.0, 'duck': 30.0})
>>> s2
falcon     345.0
eagle      200.0
duck        30.0
dtype: float64

```

Now, to combine the two datasets and view the highest speeds of the birds across the two datasets

```

>>> s1.combine(s2, max)
duck       NaN
eagle      200.0
falcon     345.0
dtype: float64

```

In the previous example, the resulting value for duck is missing, because the maximum of a NaN and a float is a NaN. So, in the example, we set ``fill_value=0``, so the maximum value returned will be the value from some dataset.

```

>>> s1.combine(s2, max, fill_value=0)
duck       30.0
eagle      200.0
falcon     345.0
dtype: float64

```

`combine_first(self, other) -> 'Series'`

Update null elements with value in the same location in 'other'.

Combine two Series objects by filling null values in one Series with non-null values from the other Series. Result index will be the union of the two indexes.

Parameters

`other` : Series

The value(s) to be used for filling null values.

Returns

Series

The result of combining the provided Series with the other object.

See Also

`Series.combine` : Perform element-wise operation on two Series using a given function.

Examples

```
>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4, 5])
>>> s1.combine_first(s2)
0    1.0
1    4.0
2    5.0
dtype: float64
```

Null values still persist if the location of that null value does not exist in `other`

```
>>> s1 = pd.Series({'falcon': np.nan, 'eagle': 160.0})
>>> s2 = pd.Series({'eagle': 200.0, 'duck': 30.0})
>>> s1.combine_first(s2)
duck      30.0
eagle     160.0
falcon      NaN
dtype: float64
```

```
compare(self, other: 'Series', align_axis: 'Axis' = 1, keep_shape: 'bool' = False, keep_
```

Compare to another Series and show the differences.

```
.. versionadded:: 1.1.0
```

Parameters

`other` : Series

Object to compare with.

`align_axis` : {0 or 'index', 1 or 'columns'}, default 1

Determine which axis to align the comparison on.

* 0, or 'index' : Resulting differences are stacked vertically

with rows drawn alternately from self and other.
* 1, or 'columns' : Resulting differences are aligned horizontally
with columns drawn alternately from self and other.

keep_shape : bool, default False
If true, all rows and columns are kept.
Otherwise, only the ones with different values are kept.

keep_equal : bool, default False
If true, the result keeps values that are equal.
Otherwise, equal values are shown as NaNs.

result_names : tuple, default ('self', 'other')
Set the dataframes names in the comparison.

.. versionadded:: 1.5.0

Returns

Series or DataFrame

If axis is 0 or 'index' the result will be a Series.
The resulting index will be a MultiIndex with 'self' and 'other'
stacked alternately at the inner level.

If axis is 1 or 'columns' the result will be a DataFrame.
It will have two columns namely 'self' and 'other'.

See Also

DataFrame.compare : Compare with another DataFrame and show differences.

Notes

Matching NaNs will not appear as a difference.

Examples

```
>>> s1 = pd.Series(["a", "b", "c", "d", "e"])
>>> s2 = pd.Series(["a", "a", "c", "b", "e"])
```

Align the differences on columns

```
>>> s1.compare(s2)
```

```

|         self other
|    1     b     a
|    3     d     b
|
| Stack the differences on indices
|
| >>> s1.compare(s2, align_axis=0)
|    1 self     b
|       other   a
|    3 self     d
|       other   b
|    dtype: object
|
| Keep all original rows
|
| >>> s1.compare(s2, keep_shape=True)
|         self other
|    0 NaN     NaN
|    1  b      a
|    2 NaN     NaN
|    3  d      b
|    4 NaN     NaN
|
| Keep all original rows and also all original values
|
| >>> s1.compare(s2, keep_shape=True, keep_equal=True)
|         self other
|    0  a      a
|    1  b      a
|    2  c      c
|    3  d      b
|    4  e      e
|
| corr(self, other: 'Series', method: 'CorrelationMethod' = 'pearson', min_periods: 'int'
|     Compute correlation with `other` Series, excluding missing values.
|
| The two `Series` objects are not required to be the same length and will be
| aligned internally before the correlation function is applied.
|
| Parameters
| -----
| other : Series
|         Series with which to compute the correlation.

```

```
method : {'pearson', 'kendall', 'spearman'} or callable
        Method used to compute correlation:

- pearson : Standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- callable: Callable with input two 1d ndarrays and returning a float.
```

```
.. warning::
    Note that the returned matrix from corr will have 1 along the
    diagonals and will be symmetric regardless of the callable's
    behavior.
```

```
min_periods : int, optional
    Minimum number of observations needed to have a valid result.
```

```
Returns
-----
float
    Correlation with other.
```

```
See Also
-----
DataFrame.corr : Compute pairwise correlation between columns.
DataFrame.corrwith : Compute pairwise correlation with another
    DataFrame or Series.
```

```
Notes
-----
Pearson, Kendall and Spearman correlation are currently computed using pairwise comp

* `Pearson correlation coefficient <https://en.wikipedia.org/wiki/Pearson\_correlation>
* `Kendall rank correlation coefficient <https://en.wikipedia.org/wiki/Kendall\_rank\_>
* `Spearman's rank correlation coefficient <https://en.wikipedia.org/wiki/Spearman%2>
```

```
Examples
-----
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> s1 = pd.Series([.2, .0, .6, .2])
>>> s2 = pd.Series([.3, .6, .0, .1])
>>> s1.corr(s2, method=histogram_intersection)
0.3
```

```

|
| count(self)
|     Return number of non-NA/null observations in the Series.
|
| Returns
| -----
| int or Series (if level specified)
|     Number of non-null values in the Series.
|
| See Also
| -----
| DataFrame.count : Count non-NA cells for each column or row.
|
| Examples
| -----
| >>> s = pd.Series([0.0, 1.0, np.nan])
| >>> s.count()
| 2
|
| cov(self, other: 'Series', min_periods: 'int | None' = None, ddof: 'int | None' = 1) ->
|     Compute covariance with Series, excluding missing values.
|
| The two `Series` objects are not required to be the same length and
| will be aligned internally before the covariance is calculated.
|
| Parameters
| -----
| other : Series
|     Series with which to compute the covariance.
| min_periods : int, optional
|     Minimum number of observations needed to have a valid result.
| ddof : int, default 1
|     Delta degrees of freedom. The divisor used in calculations
|     is ``N - ddof``, where ``N`` represents the number of elements.
|
| .. versionadded:: 1.1.0
|
| Returns
| -----
| float
|     Covariance between Series and other normalized by N-1
|     (unbiased estimator).
|
|

```


See Also

DataFrame.cov : Compute pairwise covariance of columns.

Examples

>>> s1 = pd.Series([0.90010907, 0.13484424, 0.62036035])
>>> s2 = pd.Series([0.12528585, 0.26962463, 0.51111198])
>>> s1.cov(s2)
-0.01685762652715874

cummax(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, *args, **kwargs)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

For `Series` this parameter is unused and defaults to 0.

skipna : bool, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs

Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

scalar or Series

Return cumulative maximum of scalar or Series.

See Also

core.window.expanding.Expanding.max : Similar functionality but ignores ``NaN`` values.

Series.max : Return the maximum over Series axis.

Series.cummax : Return cumulative maximum over Series axis.

Series.cummin : Return cumulative minimum over Series axis.

Series.cumsum : Return cumulative sum over Series axis.

Series.cumprod : Return cumulative product over Series axis.

Examples

****Series****

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
```

```
>>> s
```

```
0    2.0
```

```
1    NaN
```

```
2    5.0
```

```
3   -1.0
```

```
4    0.0
```

```
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
```

```
0    2.0
```

```
1    NaN
```

```
2    5.0
```

```
3    5.0
```

```
4    5.0
```

```
dtype: float64
```

To include NA values in the operation, use ``skipna=False``

```
>>> s.cummax(skipna=False)
```

```
0    2.0
```

```
1    NaN
```

```
2    NaN
```

```
3    NaN
```

```
4    NaN
```

```
dtype: float64
```

****DataFrame****

```
>>> df = pd.DataFrame([[2.0, 1.0],  
...                    [3.0, np.nan],  
...                    [1.0, 0.0]],  
...                   columns=list('AB'))
```

```
>>> df
```

```
   A    B
```

```
| 0 2.0 1.0
| 1 3.0 NaN
| 2 1.0 0.0
```

| By default, iterates over rows and finds the maximum
| in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```
| >>> df.cummax()
|      A    B
| 0 2.0  1.0
| 1 3.0  NaN
| 2 3.0  1.0
```

| To iterate over columns and find the maximum in each row,
| use ``axis=1``

```
| >>> df.cummax(axis=1)
|      A    B
| 0 2.0  2.0
| 1 3.0  NaN
| 2 1.0  1.0
```

```
| cummin(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, *args, **kwargs)
```

| Return cumulative minimum over a DataFrame or Series axis.

| Returns a DataFrame or Series of the same size containing the cumulative
| minimum.

| Parameters

| -----
| axis : {0 or 'index', 1 or 'columns'}, default 0

| The index or the name of the axis. 0 is equivalent to None or 'index'.

| For `Series` this parameter is unused and defaults to 0.

| skipna : bool, default True

| Exclude NA/null values. If an entire row/column is NA, the result
| will be NA.

| *args, **kwargs

| Additional keywords have no effect but might be accepted for
| compatibility with NumPy.

| Returns

| -----
| scalar or Series

Return cumulative minimum of scalar or Series.

See Also

core.window.expanding.Expanding.min : Similar functionality
but ignores ``NaN`` values.
Series.min : Return the minimum over
Series axis.
Series.cummax : Return cumulative maximum over Series axis.
Series.cummin : Return cumulative minimum over Series axis.
Series.cumsum : Return cumulative sum over Series axis.
Series.cumprod : Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use ``skipna=False``

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
```

```
4 NaN
dtype: float64
```

```
**DataFrame**
```

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
```

```
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```
>>> df.cummin()
   A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use ``axis=1``

```
>>> df.cummin(axis=1)
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

```
cumprod(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, *args, **kwargs)
Return cumulative product over a DataFrame or Series axis.
```

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

For `Series` this parameter is unused and defaults to 0.

`skipna` : bool, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA.

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

Returns

scalar or Series

Return cumulative product of scalar or Series.

See Also

`core.window.expanding.Expanding.prod` : Similar functionality but ignores `NaN` values.

`Series.prod` : Return the product over Series axis.

`Series.cummax` : Return cumulative maximum over Series axis.

`Series.cummin` : Return cumulative minimum over Series axis.

`Series.cumsum` : Return cumulative sum over Series axis.

`Series.cumprod` : Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
```

```
>>> s
```

```
0    2.0
```

```
1    NaN
```

```
2    5.0
```

```
3   -1.0
```

```
4    0.0
```

```
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
```

```
0    2.0
```

```
1    NaN
```

```
2   10.0
```

```
| 3 -10.0
| 4 -0.0
| dtype: float64
```

To include NA values in the operation, use ``skipna=False``

```
| >>> s.cumprod(skipna=False)
| 0 2.0
| 1 NaN
| 2 NaN
| 3 NaN
| 4 NaN
| dtype: float64
```

****DataFrame****

```
| >>> df = pd.DataFrame([[2.0, 1.0],
| ...                    [3.0, np.nan],
| ...                    [1.0, 0.0]],
| ...                    columns=list('AB'))
| >>> df
|      A  B
| 0 2.0 1.0
| 1 3.0 NaN
| 2 1.0 0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```
| >>> df.cumprod()
|      A  B
| 0 2.0 1.0
| 1 6.0 NaN
| 2 6.0 0.0
```

To iterate over columns and find the product in each row, use ``axis=1``

```
| >>> df.cumprod(axis=1)
|      A  B
| 0 2.0 2.0
| 1 3.0 NaN
| 2 1.0 0.0
```

```

| cumsum(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, *args, **kwargs)
|     Return cumulative sum over a DataFrame or Series axis.
|
| Returns a DataFrame or Series of the same size containing the cumulative
| sum.
|
| Parameters
| -----
| axis : {0 or 'index', 1 or 'columns'}, default 0
|         The index or the name of the axis. 0 is equivalent to None or 'index'.
|         For `Series` this parameter is unused and defaults to 0.
| skipna : bool, default True
|         Exclude NA/null values. If an entire row/column is NA, the result
|         will be NA.
| *args, **kwargs
|         Additional keywords have no effect but might be accepted for
|         compatibility with NumPy.
|
| Returns
| -----
| scalar or Series
|     Return cumulative sum of scalar or Series.
|
| See Also
| -----
| core.window.expanding.Expanding.sum : Similar functionality
|     but ignores ``NaN`` values.
| Series.sum : Return the sum over
|     Series axis.
| Series.cummax : Return cumulative maximum over Series axis.
| Series.cummin : Return cumulative minimum over Series axis.
| Series.cumsum : Return cumulative sum over Series axis.
| Series.cumprod : Return cumulative product over Series axis.
|
| Examples
| -----
| **Series**
|
| >>> s = pd.Series([2, np.nan, 5, -1, 0])
| >>> s
| 0    2.0
| 1    NaN

```



```
| 2    5.0
| 3   -1.0
| 4    0.0
| dtype: float64
```

| By default, NA values are ignored.

```
| >>> s.cumsum()
| 0    2.0
| 1   NaN
| 2    7.0
| 3    6.0
| 4    6.0
| dtype: float64
```

| To include NA values in the operation, use ``skipna=False``

```
| >>> s.cumsum(skipna=False)
| 0    2.0
| 1   NaN
| 2   NaN
| 3   NaN
| 4   NaN
| dtype: float64
```

| ****DataFrame****

```
| >>> df = pd.DataFrame([[2.0, 1.0],
| ...                   [3.0, np.nan],
| ...                   [1.0, 0.0]],
| ...                   columns=list('AB'))
| >>> df
|      A    B
| 0  2.0  1.0
| 1  3.0  NaN
| 2  1.0  0.0
```

| By default, iterates over rows and finds the sum
| in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```
| >>> df.cumsum()
|      A    B
| 0  2.0  1.0
```

```
1 5.0 NaN
2 6.0 1.0
```

To iterate over columns and find the sum in each row,
use ``axis=1``

```
>>> df.cumsum(axis=1)
      A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

`diff(self, periods: 'int' = 1) -> 'Series'`

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the Series (default is element in previous row).

Parameters

periods : int, default 1
Periods to shift for calculating difference, accepts negative values.

Returns

Series
First differences of the Series.

See Also

Series.pct_change: Percent change over given number of periods.
Series.shift: Shift index by desired number of periods with an optional time freq.
DataFrame.diff: First discrete difference of object.

Notes

For boolean dtypes, this uses `:meth:`operator.xor`` rather than `:meth:`operator.sub``.
The result is calculated according to current dtype in Series, however dtype of the result is always float64.

Examples

Difference with previous row

```
>>> s = pd.Series([1, 1, 2, 3, 5, 8])
>>> s.diff()
0    NaN
1    0.0
2    1.0
3    1.0
4    2.0
5    3.0
dtype: float64
```

Difference with 3rd previous row

```
>>> s.diff(periods=3)
0    NaN
1    NaN
2    NaN
3    2.0
4    4.0
5    6.0
dtype: float64
```

Difference with following row

```
>>> s.diff(periods=-1)
0    0.0
1   -1.0
2   -1.0
3   -2.0
4   -3.0
5    NaN
dtype: float64
```

Overflow in input dtype

```
>>> s = pd.Series([1, 0], dtype=np.uint8)
>>> s.diff()
0    NaN
1   255.0
```

```

dtype: float64
|
|
| div = truediv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|
|
| divide = truediv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|
|
| divmod(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Integer division and modulo of series and other, element-wise (binary operator)
|
|     Equivalent to ``divmod(series, other)`` , but with support to substitute a fill_value
|     missing data in either one of the inputs.
|
|
| Parameters
| -----
|
| other : Series or scalar value
| level : int or name
|         Broadcast across a level, matching Index values on the
|         passed MultiIndex level.
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
|
| Returns
| -----
|
| 2-Tuple of Series
|     The result of the operation.
|
|
| See Also
| -----
|
| Series.rdivmod : Reverse of the Integer division and modulo operator, see
|     `Python documentation
|     <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|     for more details.
|
|
| 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.divmod(b, fill_value=0)
|     (a    1.0
|      b    NaN
|      c    NaN
|      d    0.0
|      e    NaN
|      dtype: float64,
|      a    0.0
|      b    NaN
|      c    NaN
|      d    0.0
|      e    NaN
|      dtype: float64)

```

```

| dot(self, other: 'AnyArrayLike') -> 'Series | np.ndarray'
|     Compute the dot product between the Series and the columns of other.
|
|     This method computes the dot product between the Series and another
|     one, or the Series and each columns of a DataFrame, or the Series and
|     each columns of an array.
|
|     It can also be called using `self @ other` in Python >= 3.5.
|
|     Parameters
|     -----
|     other : Series, DataFrame or array-like
|         The other object to compute the dot product with its columns.
|
|     Returns
|     -----
|     scalar, Series or numpy.ndarray
|         Return the dot product of the Series and other if other is a

```

Series, the Series of the dot product of Series and each rows of other if other is a DataFrame or a numpy.ndarray between the Series and each columns of the numpy array.

See Also

DataFrame.dot: Compute the matrix product with the DataFrame.
Series.mul: Multiplication of series and other, element-wise.

Notes

The Series and other has to share the same index if other is a Series or a DataFrame.

Examples

>>> s = pd.Series([0, 1, 2, 3])
>>> other = pd.Series([-1, 2, -3, 4])
>>> s.dot(other)
8
>>> s @ other
8
>>> df = pd.DataFrame([[0, 1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(df)
0 24
1 14
dtype: int64
>>> arr = np.array([[0, 1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(arr)
array([24, 14])

drop(self, labels: 'IndexLabel' = None, *, axis: 'Axis' = 0, index: 'IndexLabel' = None,
Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels.
When using a multi-index, labels on different levels can be removed
by specifying the level.

Parameters

labels : single label or list-like
Index labels to drop.
axis : {0 or 'index'}

```

|         Unused. Parameter needed for compatibility with DataFrame.
| index : single label or list-like
|         Redundant for application on Series, but 'index' can be used instead
|         of 'labels'.
| columns : single label or list-like
|         No change is made to the Series; use 'index' or 'labels' instead.
| level : int or level name, optional
|         For MultiIndex, level for which the labels will be removed.
| inplace : bool, default False
|         If True, do operation inplace and return None.
| errors : {'ignore', 'raise'}, default 'raise'
|         If 'ignore', suppress error and only existing labels are dropped.
|
| Returns
| -----
| Series or None
|         Series with specified index labels removed or None if ``inplace=True``.
|
| Raises
| -----
| KeyError
|         If none of the labels are found in the index.
|
| See Also
| -----
| Series.reindex : Return only specified index labels of Series.
| Series.dropna : Return series without null values.
| Series.drop_duplicates : Return Series with duplicate values removed.
| DataFrame.drop : Drop specified labels from rows or columns.
|
| Examples
| -----
| >>> s = pd.Series(data=np.arange(3), index=['A', 'B', 'C'])
| >>> s
| A    0
| B    1
| C    2
| dtype: int64
|
| Drop labels B en C
|
| >>> s.drop(labels=['B', 'C'])
| A    0

```

```

dtype: int64
|
| Drop 2nd level label in MultiIndex Series
|
| >>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
| ...                               ['speed', 'weight', 'length']],
| ...                               codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
| ...                               [0, 1, 2, 0, 1, 2, 0, 1, 2]])
| >>> s = pd.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
| ...               index=midx)
| >>> s
| lama    speed    45.0
|         weight   200.0
|         length   1.2
| cow     speed    30.0
|         weight   250.0
|         length   1.5
| falcon  speed    320.0
|         weight   1.0
|         length   0.3
| dtype: float64
|
| >>> s.drop(labels='weight', level=1)
| lama    speed    45.0
|         length   1.2
| cow     speed    30.0
|         length   1.5
| falcon  speed    320.0
|         length   0.3
| dtype: float64
|
| drop_duplicates(self, *, keep: 'DropKeep' = 'first', inplace: 'bool' = False, ignore_index: 'bool' = False)
| Return Series with duplicate values removed.
|
| Parameters
| -----
| keep : {'first', 'last', ``False``}, default 'first'
| Method to handle dropping duplicates:
|
| - 'first' : Drop duplicates except for the first occurrence.
| - 'last'  : Drop duplicates except for the last occurrence.
| - ``False`` : Drop all duplicates.
|

```



```
inplace : bool, default ``False``
    If ``True``, performs operation inplace and returns None.

ignore_index : bool, default ``False``
    If ``True``, the resulting axis will be labeled 0, 1, ..., n - 1.

.. versionadded:: 2.0.0
```

Returns

Series or None

Series with duplicates dropped or None if ``inplace=True``.

See Also

`Index.drop_duplicates` : Equivalent method on `Index`.

`DataFrame.drop_duplicates` : Equivalent method on `DataFrame`.

`Series.duplicated` : Related method on `Series`, indicating duplicate Series values.

`Series.unique` : Return unique values as an array.

Examples

Generate a Series with duplicated entries.

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
```

```
>>> s
```

```
0    lama
1     cow
2    lama
3  beetle
4    lama
5   hippo
```

```
Name: animal, dtype: object
```

With the 'keep' parameter, the selection behaviour of duplicated values can be changed. The value 'first' keeps the first occurrence for each set of duplicated entries. The default value of keep is 'first'.

```
>>> s.drop_duplicates()
```

```
0    lama
1     cow
```

```
| 3 beetle  
| 5 hippo  
| Name: animal, dtype: object  
|
```

The value 'last' for parameter 'keep' keeps the last occurrence for each set of duplicated entries.

```
| >>> s.drop_duplicates(keep='last')  
| 1 cow  
| 3 beetle  
| 4 lama  
| 5 hippo  
| Name: animal, dtype: object  
|
```

The value ``False`` for parameter 'keep' discards all sets of duplicated entries.

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

```
| dropna(self, *, axis: 'Axis' = 0, inplace: 'bool' = False, how: 'AnyAll | None' = None, )  
| Return a new Series with missing values removed.  
|
```

See the :ref:`User Guide <missing_data>` for more on which values are considered missing, and how to work with missing data.

Parameters

```
| -----  
| axis : {0 or 'index'}  
| Unused. Parameter needed for compatibility with DataFrame.  
| inplace : bool, default False  
| If True, do operation inplace and return None.  
| how : str, optional  
| Not in use. Kept for compatibility.  
| ignore_index : bool, default ``False``  
| If ``True``, the resulting axis will be labeled 0, 1, ..., n - 1.  
|
```

```
| .. versionadded:: 2.0.0  
|
```

Returns

```
| -----  
| Series or None  
|     Series with NA entries dropped from it or None if ``inplace=True``.
```

```
| 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
```

```
| 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

duplicated(self, keep: 'DropKeep' = 'first') -> 'Series'
    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[bool]
    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
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
```

```
eq(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.

Parameters

```

| other : Series or scalar value
| level : int or name
|     Broadcast across a level, matching Index values on the
|     passed MultiIndex level.
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|     Unused. Parameter needed for compatibility with DataFrame.
|
| Returns
| -----
| Series
|     The result of the operation.
|
| 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.eq(b, fill_value=0)
| a    True
| b    False
| c    False
| d    False
| e    False
| dtype: bool
|
| explode(self, ignore_index: 'bool' = False) -> 'Series'
|     Transform each element of a list-like to a row.

```

Parameters

`ignore_index` : bool, default False

If True, the resulting index will be labeled 0, 1, ..., n - 1.

.. versionadded:: 1.1.0

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, sets, Series, and `np.ndarray`. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a `np.nan` for that row. In addition, the ordering of elements in the output will be non-deterministic when exploding sets.

Reference :ref:`the user guide <reshaping.explode>` for more examples.

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
|
| ffill(self, *, axis: 'None | Axis' = None, inplace: 'bool' = False, limit: 'None | int' =
|     None)
|     Synonym for :meth:`DataFrame.fillna` with ``method='ffill'``.
|
|     Returns
|     -----
|     Series/DataFrame or None
|         Object with missing values filled or None if ``inplace=True``.
|
| fillna(self, value: 'Hashable | Mapping | Series | DataFrame' = None, *, method: 'Fillna
|     Method')
|     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', 'ffill', None}, default None
|         Method to use for filling holes in reindexed Series:
|
|         * 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. For `Series`
|         this parameter is unused and defaults to 0.
|     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, np.nan],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list("ABCD"))
```

```
>>> df
```

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	NaN	1.0
2	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
```

	A	B	C	D
0	0.0	2.0	0.0	0.0
1	3.0	4.0	0.0	1.0
2	0.0	0.0	0.0	0.0
3	0.0	3.0	0.0	4.0

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

```
>>> df.fillna(method="ffill")
```

```
   A    B    C    D
0 NaN  2.0 NaN  0.0
1  3.0  4.0 NaN  1.0
2  3.0  4.0 NaN  1.0
3  3.0  3.0 NaN  4.0
```

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  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
```

```
   A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  3.0
3  NaN  3.0  NaN  4.0
```

When filling using a DataFrame, replacement happens along the same column names and same indices

```
>>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
>>> df.fillna(df2)
```

```
   A    B    C    D
0  0.0  2.0  0.0  0.0
1  3.0  4.0  0.0  1.0
2  0.0  0.0  0.0  NaN
3  0.0  3.0  0.0  4.0
```

Note that column D is not affected since it is not present in df2.

```

| floordiv(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.rfloordiv : Reverse of the Integer division operator, see
|         `Python documentation
|         <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|         for more details.
|
|     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    inf
|     c    inf
|     d    0.0
|     e    NaN
|     dtype: float64
|
|     ge(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Greater than or equal to of series and other, element-wise (binary operator >=)
|
|     Equivalent to ``series >= other``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     Examples
|     -----
|     >>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
|     >>> a
|     a    1.0

```

```

|     b    1.0
|     c    1.0
|     d    NaN
|     e    1.0
|     dtype: float64
|     >>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
|     >>> b
|     a    0.0
|     b    1.0
|     c    2.0
|     d    NaN
|     f    1.0
|     dtype: float64
|     >>> a.ge(b, fill_value=0)
|     a     True
|     b     True
|     c    False
|     d    False
|     e     True
|     f    False
|     dtype: bool

```

```

| groupby(self, by=None, axis: 'Axis' = 0, level: 'IndexLabel' = None, as_index: 'bool' = True)
| 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, `pd.Grouper` or list of such

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 a list or ndarray of length equal to the selected axis is passed (see the `groupby` user guide https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#splitting), the values are used as-is to 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). For `Series` this parameter
    is unused and defaults to 0.
level : int, level name, or sequence of such, default None
    If the axis is a MultiIndex (hierarchical), group by a particular
    level or levels. Do not specify both ``by`` and ``level``.
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.

.. versionchanged:: 2.0.0

    Specifying ``sort=False`` with an ordered categorical grouper will no
    longer sort the values.

group_keys : bool, default True
    When calling apply and the ``by`` argument produces a like-indexed
    (i.e. :ref:`a transform <groupby.transform>`) result, add group keys to
    index to identify pieces. By default group keys are not included
    when the result's index (and column) labels match the inputs, and
    are included otherwise.

.. versionchanged:: 1.5.0

    Warns that ``group_keys`` will no longer be ignored when the
    result from ``apply`` is a like-indexed Series or DataFrame.
    Specify ``group_keys`` explicitly to include the group keys or
    not.

.. versionchanged:: 2.0.0

    ``group_keys`` now defaults to ``True``.

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.
dropna : bool, default True

```

If True, and if group keys contain NA values, NA values together with row/column will be dropped.
If False, NA values will also be treated as the key in groups.

.. versionadded:: 1.1.0

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

<<https://pandas.pydata.org/pandas-docs/stable/groupby.html>>`__ for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and 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
Wild       185.0
Name: Max Speed, dtype: float64
```

We can also choose to include `NA` in group keys or not by defining `dropna` parameter, the default setting is `True`.

```
>>> ser = pd.Series([1, 2, 3, 3], index=["a", 'a', 'b', np.nan])
>>> ser.groupby(level=0).sum()
a     3
b     3
dtype: int64

>>> ser.groupby(level=0, dropna=False).sum()
a     3
b     3
```



```

|     NaN 3
|     dtype: int64
|
|     >>> arrays = ['Falcon', 'Falcon', 'Parrot', 'Parrot']
|     >>> ser = pd.Series([390., 350., 30., 20.], index=arrays, name="Max Speed")
|     >>> ser.groupby(["a", "b", "a", np.nan]).mean()
|     a    210.0
|     b    350.0
|     Name: Max Speed, dtype: float64
|
|     >>> ser.groupby(["a", "b", "a", np.nan], dropna=False).mean()
|     a    210.0
|     b    350.0
|     NaN   20.0
|     Name: Max Speed, dtype: float64
|
| gt(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     Examples
|     -----

```

```

| >>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
| >>> a
| a    1.0
| b    1.0
| c    1.0
| d    NaN
| e    1.0
| dtype: float64
| >>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
| >>> b
| a    0.0
| b    1.0
| c    2.0
| d    NaN
| f    1.0
| dtype: float64
| >>> a.gt(b, fill_value=0)
| a    True
| b    False
| c    False
| d    False
| e    True
| f    False
| dtype: bool

```

```

| hist = hist_series(self, by=None, ax=None, grid: 'bool' = True, xlabelsize: 'int | None'
| 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``.
|
|     legend : bool, default False
|         Whether to show the legend.
|
|
|     .. versionadded:: 1.1.0
|
|
|     **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.
|
|
| idxmax(self, axis: 'Axis' = 0, skipna: 'bool' = True, *args, **kwargs) -> 'Hashable'
|     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 : {0 or 'index'}
|     Unused. Parameter needed for compatibility with DataFrame.
|
| 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
```

`idxmin(self, axis: 'Axis' = 0, skipna: 'bool' = True, *args, **kwargs) -> 'Hashable'`
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` : {0 or 'index'}

Unused. Parameter needed for compatibility with DataFrame.

`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.argmax`. This method returns the label of the minimum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax`.

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
```

```
info(self, verbose: 'bool | None' = None, buf: 'IO[str] | None' = None, max_cols: 'int | None' = None)
Print a concise summary of a Series.
```

This method prints information about a Series including the index dtype, non-null values and memory usage.

```
.. versionadded:: 1.4.0
```

Parameters

`verbose` : bool, optional

Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

`buf` : writable buffer, defaults to `sys.stdout`

Where to send the output. By default, the output is printed to `sys.stdout`. Pass a writable buffer if you need to further process the output.

memory_usage : bool, str, optional
 Specifies whether total memory usage of the Series elements (including the index) should be displayed. By default, this follows the ``pandas.options.display.memory_usage`` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources. See the :ref:`Frequently Asked Questions <df-memory-usage>` for more details.

show_counts : bool, optional
 Whether to show the non-null counts. By default, this is shown only if the DataFrame is smaller than ``pandas.options.display.max_info_rows`` and ``pandas.options.display.max_info_columns``. A value of True always shows the counts, and False never shows the counts.

Returns

None

This method prints a summary of a Series and returns None.

See Also

Series.describe: Generate descriptive statistics of Series.
 Series.memory_usage: Memory usage of Series.

Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> s = pd.Series(text_values, index=int_values)
>>> s.info()
<class 'pandas.core.series.Series'>
Index: 5 entries, 1 to 5
Series name: None
Non-Null Count  Dtype
```

```
| -----  -----  
| 5 non-null      object  
| dtypes: object(1)  
| memory usage: 80.0+ bytes  
|
```

| Prints a summary excluding information about its values:

```
| >>> s.info(verbose=False)  
| <class 'pandas.core.series.Series'>  
| Index: 5 entries, 1 to 5  
| dtypes: object(1)  
| memory usage: 80.0+ bytes  
|
```

| Pipe output of Series.info to buffer instead of sys.stdout, get
| buffer content and writes to a text file:

```
| >>> import io  
| >>> buffer = io.StringIO()  
| >>> s.info(buf=buffer)  
| >>> s = buffer.getvalue()  
| >>> with open("df_info.txt", "w",  
| ...         encoding="utf-8") as f: # doctest: +SKIP  
| ...     f.write(s)  
| 260  
|
```

| The `memory_usage` parameter allows deep introspection mode, specially
| useful for big Series and fine-tune memory optimization:

```
| >>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)  
| >>> s = pd.Series(np.random.choice(['a', 'b', 'c'], 10 ** 6))  
| >>> s.info()  
| <class 'pandas.core.series.Series'>  
| RangeIndex: 1000000 entries, 0 to 999999  
| Series name: None  
| Non-Null Count      Dtype  
| -----  -----  
| 1000000 non-null  object  
| dtypes: object(1)  
| memory usage: 7.6+ MB  
|
```

```
| >>> s.info(memory_usage='deep')  
| <class 'pandas.core.series.Series'>  
| RangeIndex: 1000000 entries, 0 to 999999  
|
```



```

Series name: None
Non-Null Count    Dtype
-----
1000000 non-null object
dtypes: object(1)
memory usage: 55.3 MB

interpolate(self: 'Series', method: 'str' = 'linear', *, axis: 'Axis' = 0, limit: 'int')
Fill NaN values using an interpolation method.

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',
      'barycentric', 'polynomial': Passed to
      `scipy.interpolate.interp1d`, whereas 'spline' is passed to
      `scipy.interpolate.UnivariateSpline`. 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)``. Note that,
      `slinear` method in Pandas refers to the Scipy first order `spline`
      instead of Pandas first order `spline`.
    * 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima',
      'cubicspline': 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. For `Series` this parameter is unused

```

```

    and defaults to 0.
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'}, Optional
    Consecutive NaNs will be filled in this direction.

    If limit is specified:
        * If 'method' is 'pad' or 'ffill', 'limit_direction' must be 'forward'.
        * If 'method' is 'backfill' or 'bfill', 'limit_direction' must be
          'backwards'.

    If 'limit' is not specified:
        * If 'method' is 'backfill' or 'bfill', the default is 'backward'
        * else the default is 'forward'

    .. versionchanged:: 1.1.0
        raises ValueError if `limit_direction` is 'forward' or 'both' and
        method is 'backfill' or 'bfill'.
        raises ValueError if `limit_direction` is 'backward' or 'both' and
        method is 'pad' or 'ffill'.

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).

downcast : optional, 'infer' or None, defaults to None
    Downcast dtypes if possible.
`**kwargs` : optional
    Keyword arguments to pass on to the interpolating function.

Returns
-----
Series or DataFrame or None
    Returns the same object type as the caller, interpolated at
    some or all ``NaN`` values or None if ``inplace=True``.

```

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`
<<https://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-interpolation>>

Examples

Filling in ``NaN`` in a :class:`~pandas.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
```

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
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
```

`isin(self, values) -> 'Series'`

Whether elements in Series are contained in `values`.

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
```

To invert the boolean values, use the ``~`` operator:

```
>>> ~s.isin(['cow', 'lama'])
0    False
1    False
2    False
3     True
4    False
5     True
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
|
| Strings and integers are distinct and are therefore not comparable:
|
| >>> pd.Series([1]).isin(['1'])
| 0    False
| dtype: bool
| >>> pd.Series([1.1]).isin(['1.1'])
| 0    False
| dtype: bool
|
| isna(self) -> 'Series'
| Detect missing values.
|
| Return a boolean same-sized object indicating if the values are NA.
| NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
| values.
| Everything else gets mapped to False values. Characters such as empty
| strings ``''`` or :attr:`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 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(dict(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
|
| isnull(self) -> 'Series'
| Series.isnull is an alias for Series.isna.
|
| Detect missing values.
|
| Return a boolean same-sized object indicating if the values are NA.
| NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
| values.

```


Everything else gets mapped to False values. Characters such as empty strings ``''`` or `:attr:`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 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(dict(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
|
| items(self) -> 'Iterable[tuple[Hashable, Any]]'
|     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
|
| keys(self) -> 'Index'
|     Return alias for index.
|
|     Returns
|     -----
|     Index
|         Index of the Series.

```

```

| kurt(self, axis: 'Axis | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t' = Fa
|     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.
|     For `Series` this parameter is unused and defaults to 0.
|
|     For DataFrames, specifying ``axis=None`` will apply the aggregation
|     across both axes.
|
|     .. versionadded:: 2.0.0
|
| skipna : bool, default True
|     Exclude NA/null values when computing the result.
| numeric_only : bool, default False
|     Include only float, int, boolean columns. Not implemented for Series.
|
| **kwargs
|     Additional keyword arguments to be passed to the function.
|
| Returns
| -----
| scalar or scalar
|
| kurtosis = kurt(self, axis: 'Axis | None' = 0, skipna: 'bool_t' = True, numeric_only: 'b
|
| le(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.
|
| Parameters
| -----
| other : Series or scalar value
| level : int or name
|     Broadcast across a level, matching Index values on the

```

```

|         passed MultiIndex level.
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
| Returns
| -----
| Series
|     The result of the operation.
|
| Examples
| -----
| >>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
| >>> a
| a    1.0
| b    1.0
| c    1.0
| d    NaN
| e    1.0
| dtype: float64
| >>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
| >>> b
| a    0.0
| b    1.0
| c    2.0
| d    NaN
| f    1.0
| dtype: float64
| >>> a.le(b, fill_value=0)
| a    False
| b     True
| c     True
| d    False
| e    False
| f     True
| dtype: bool
|
| lt(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.

Parameters

other : Series or scalar value

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

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 of filling (at that location) will be missing.

axis : {0 or 'index'}

Unused. Parameter needed for compatibility with DataFrame.

Returns

Series

The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
```

```
>>> a
```

```
a    1.0
```

```
b    1.0
```

```
c    1.0
```

```
d    NaN
```

```
e    1.0
```

```
dtype: float64
```

```
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
```

```
>>> b
```

```
a    0.0
```

```
b    1.0
```

```
c    2.0
```

```
d    NaN
```

```
f    1.0
```

```
dtype: float64
```

```
>>> a.lt(b, fill_value=0)
```

```
a    False
```

```

|     b    False
|     c    True
|     d    False
|     e    False
|     f    True
|     dtype: bool
|
| map(self, arg: 'Callable | Mapping | Series', na_action: "Literal['ignore'] | None" = None)
|     Map values of Series according to an input mapping or function.
|
|     Used for substituting each value in a Series with another value,
|     that may be derived from a function, a ``dict`` or
|     a :class:`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

```

| `mask(self, cond, other=<no_default>, *, inplace: 'bool' = False, axis: 'Axis | None' = None)`
| 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).
    If not specified, entries will be filled with the corresponding
    NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension
    dtypes).
inplace : bool, default False
    Whether to perform the operation in place on the data.
axis : int, default None
    Alignment axis if needed. For `Series` this parameter is
    unused and defaults to 0.
level : int, default None
    Alignment level if needed.

```

Returns

Same type as caller or None if ``inplace=True``.

See Also

```
:func:`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. If the axis of ``other`` does not align with axis of ``cond`` Series/DataFrame, the misaligned index positions will be filled with True.

The signature for :func:`DataFrame.where` differs from :func:`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 [:ref:`indexing <indexing.where_mask>`](#).

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

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 = pd.Series(range(5))
```

```
>>> t = pd.Series([True, False])
```

```
>>> s.where(t, 99)
```

```
0     0
```

```
1    99
```

```
2    99
```

```
3    99
```

```
4    99
```

```
dtype: int64
```

```
>>> s.mask(t, 99)
```

```
0    99
```

```
1     1
```

```
2    99
```

```
3    99
```

```
4    99
```

```
dtype: int64
```

```

|     >>> s.where(s > 1, 10)
|     0    10
|     1    10
|     2     2
|     3     3
|     4     4
|     dtype: int64
|     >>> s.mask(s > 1, 10)
|     0     0
|     1     1
|     2    10
|     3    10
|     4    10
|     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
|
| max(self, axis: 'AxisInt | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t' = True)
|     Return the maximum of the values over the requested axis.
|
|     If you want the *index* of the maximum, use ``idxmax``. This is the equivalent of the
|
| Parameters
| -----
| axis : {index (0)}
|     Axis for the function to be applied on.
|     For `Series` this parameter is unused and defaults to 0.
|
|     For DataFrames, specifying ``axis=None`` will apply the aggregation
|     across both axes.
|
|     .. versionadded:: 2.0.0
|
| skipna : bool, default True
|     Exclude NA/null values when computing the result.
| numeric_only : bool, default False
|     Include only float, int, boolean columns. Not implemented for Series.
|
| **kwargs
|     Additional keyword arguments to be passed to the function.
|
| Returns
| -----
| scalar or scalar
|
| 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
```

```
mean(self, axis: 'AxisInt | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t' =
Return the mean of the values over the requested axis.
```

Parameters

axis : {index (0)}

Axis for the function to be applied on.

For `Series` this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

.. versionadded:: 2.0.0

skipna : bool, default True

Exclude NA/null values when computing the result.

numeric_only : bool, default False

Include only float, int, boolean columns. Not implemented for Series.

****kwargs**

Additional keyword arguments to be passed to the function.

Returns

```

|     scalar or scalar
|
| median(self, axis: 'AxisInt | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t'
|     Return the median of the values over the requested axis.
|
| Parameters
| -----
| axis : {index (0)}
|     Axis for the function to be applied on.
|     For `Series` this parameter is unused and defaults to 0.
|
|     For DataFrames, specifying ``axis=None`` will apply the aggregation
|     across both axes.
|
|     .. versionadded:: 2.0.0
|
| skipna : bool, default True
|     Exclude NA/null values when computing the result.
| numeric_only : bool, default False
|     Include only float, int, boolean columns. Not implemented for Series.
|
| **kwargs
|     Additional keyword arguments to be passed to the function.
|
| Returns
| -----
| scalar or scalar
|
| memory_usage(self, index: 'bool' = True, deep: 'bool' = False) -> 'int'
|     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)
244
```

```
min(self, axis: 'AxisInt | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t' = True)
```

Return the minimum of the values over the requested axis.

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

Parameters

`axis` : {index (0)}

Axis for the function to be applied on.

For ``Series`` this parameter is unused and defaults to 0.

For DataFrames, specifying ``axis=None`` will apply the aggregation across both axes.

.. versionadded:: 2.0.0

skipna : bool, default True

Exclude NA/null values when computing the result.

numeric_only : bool, default False

Include only float, int, boolean columns. Not implemented for Series.

**kwargs

Additional keyword arguments to be passed to the function.

Returns

scalar or scalar

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
|
| mod(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|           Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.rmod : Reverse of the Modulo operator, see
|         `Python documentation
|         <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|         for more details.
|
|     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

```

```

| mode(self, dropna: 'bool' = True) -> 'Series'

```

```

|     Return the mode(s) of the Series.

```

```

|     The mode is the value that appears most often. There can be multiple modes.

```

```

|     Always returns Series even if only one value is returned.

```

```

|     Parameters

```

```

|     -----

```

```

|     dropna : bool, default True

```

```

|         Don't consider counts of NaN/NaT.

```

```

|     Returns

```

```

|     -----

```

```

|     Series

```

```

|         Modes of the Series in sorted order.

```

```

| mul(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.

```

```

|     Parameters

```

```

-----
other : Series or scalar value
level : int or name
        Broadcast across a level, matching Index values on the
        passed MultiIndex level.
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 of filling (at that location) will be missing.
axis : {0 or 'index'}
        Unused. Parameter needed for compatibility with DataFrame.

```

Returns

```
-----
```

Series

The result of the operation.

See Also

```
-----
```

Series.rmul : Reverse of the Multiplication operator, see
`Python documentation

<<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>>`
for more details.

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
|     d    0.0
|     e    NaN
|     dtype: float64
|
| multiply = mul(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|
| ne(self, other, level=None, fill_value=None, axis: '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 either one of the inputs.
|
| Parameters
| -----
| other : Series or scalar value
| level : int or name
|         Broadcast across a level, matching Index values on the
|         passed MultiIndex level.
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
| Returns
| -----
| Series
|     The result of the operation.
|
| 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.ne(b, fill_value=0)
|     a    False
|     b     True
|     c     True
|     d     True
|     e     True
|     dtype: bool
|
|     nlargest(self, n: 'int' = 5, keep: "Literal['first', 'last', 'all']" = 'first') -> 'Series'
|     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, "Montserrat": 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
Montserrat     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

notna(self) -> 'Series'
    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 :attr:`numpy.inf` are not considered NA values
    (unless you set ``pandas.options.mode.use_inf_as_na = True``).
    NA values, such as None or :attr:`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(dict(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
```

| notnull(self) -> 'Series'

| Series.notnull is an alias for Series.notna.

| 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 `:attr:`numpy.inf`` are not considered NA values (unless you set ```pandas.options.mode.use_inf_as_na = True```). NA values, such as None or `:attr:`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(dict(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
|
| nsmallest(self, n: 'int' = 5, keep: 'str' = 'first') -> 'Series'
| 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.
|

```

Notes

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

Examples

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

The `n`` smallest elements where ``.n=5`` by default.

```
>>> s.nsmallest()
Montserrat     5200
Nauru          11300
Tuvalu         11300
Anguilla       11300
Iceland        337000
dtype: int64
```

The `n`` smallest elements where ``.n=3``. Default `keep`` value is `'first'` so Nauru and Tuvalu will be kept.

```
>>> s.nsmallest(3)
Montserrat     5200
Nauru          11300
Tuvalu         11300
```

```

dtype: int64

The `n` smallest elements where ``n=3`` and keeping the last
duplicates. Anguilla and Tuvalu will be kept since they are the last
with value 11300 based on the index order.

>>> s.nsmallest(3, keep='last')
Montserrat    5200
Anguilla      11300
Tuvalu        11300
dtype: int64

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

>>> s.nsmallest(3, keep='all')
Montserrat    5200
Nauru         11300
Tuvalu        11300
Anguilla      11300
dtype: int64

```

pop(self, item: 'Hashable') -> 'Any'

Return item and drops from series. Raise KeyError if not found.

Parameters

item : label

 Index of the element that needs to be removed.

Returns

Value that is popped from series.

Examples

```

>>> ser = pd.Series([1,2,3])

>>> ser.pop(0)
1

>>> ser
1    2

```

```

|     2     3
|     dtype: int64
|
| pow(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Exponential power of series and other, element-wise (binary operator `pow`).
|
|     Equivalent to ``series ** other``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.rpow : Reverse of the Exponential power operator, see
|         `Python documentation
|         <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|         for more details.
|
|     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.pow(b, fill_value=0)
| a    1.0
| b    1.0
| c    1.0
| d    0.0
| e    NaN
| dtype: float64

```

```

| prod(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, numeric_only: 'bool_t' =
| Return the product of the values over the requested axis.

```

Parameters

axis : {index (0)}

Axis for the function to be applied on.

For `Series` this parameter is unused and defaults to 0.

For DataFrames, specifying ``axis=None`` will apply the aggregation across both axes.

.. versionadded:: 2.0.0

skipna : bool, default True

Exclude NA/null values when computing the result.

numeric_only : bool, default False

Include only float, int, boolean columns. Not implemented for Series.

min_count : int, default 0

The required number of valid values to perform the operation. If fewer than

``min_count`` non-NA values are present the result will be NA.

**kwargs

Additional keyword arguments to be passed to the function.

Returns

scalar or scalar

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

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([], dtype="float64").prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([], dtype="float64").prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

```
product = prod(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, numeric_only:
```

```
quantile(self, q: 'float | Sequence[float] | AnyArrayLike' = 0.5, interpolation: 'Quanti
Return value at the given quantile.
```

Parameters

`q` : float or array-like, default 0.5 (50% quantile)
The quantile(s) to compute, which can lie in range: $0 \leq q \leq 1$.
`interpolation` : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
This optional parameter specifies the interpolation method to use,
when the desired quantile lies between two data points `i` and `j`:

- * linear: `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- * lower: `i`.
- * higher: `j`.
- * nearest: `i` or `j` whichever is nearest.
- * midpoint: $(i + j) / 2$.

Returns

float or Series

If `q` is an array, a Series will be returned where the index is `q` and the values are the quantiles, otherwise a float will be returned.

See Also

`core.window.Rolling.quantile` : Calculate the rolling quantile.

`numpy.percentile` : Returns the q-th percentile(s) of the array elements.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25    1.75
0.50    2.50
0.75    3.25
dtype: float64
```

`radd(self, other, level=None, fill_value=None, axis: 'Axis' = 0)`

Return Addition of series and other, element-wise (binary operator `radd`).

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

Parameters

```

| -----
| other : Series or scalar value
| level : int or name
|         Broadcast across a level, matching Index values on the
|         passed MultiIndex level.
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.

```

Returns

```
| -----
```

Series

The result of the operation.

See Also

```
| -----
```

Series.add : Element-wise Addition, see

``Python documentation`

`<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>``

for more details.

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.add(b, fill_value=0)
```

```
| a    2.0
```



```

|     b    1.0
|     c    1.0
|     d    1.0
|     e    NaN
|     dtype: float64
|
| ravel(self, order: 'str' = 'C') -> 'ArrayLike'
|     Return the flattened underlying data as an ndarray or ExtensionArray.
|
|     Returns
|     -----
|     numpy.ndarray or ExtensionArray
|         Flattened data of the Series.
|
|     See Also
|     -----
|     numpy.ndarray.ravel : Return a flattened array.
|
| rdiv = rtruediv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|
| rdivmod(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Integer division and modulo of series and other, element-wise (binary operator)
|
|     Equivalent to ``other divmod series``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|         Broadcast across a level, matching Index values on the
|         passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     2-Tuple of Series

```

The result of the operation.

See Also

Series.divmod : Element-wise Integer division and modulo, see
`Python documentation
<<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>>`
for more details.

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

reindex(self, index=None, *, axis: 'Axis | None' = None, method: 'str | None' = None, copy=None)
Conform Series to new index with optional filling logic.

Places NA/NaN in locations having no value in the previous index. A new object

is produced unless the new index is equivalent to the current one and ``copy=False``.

Parameters

index : array-like, optional

New labels for the index. Preferably an Index object to avoid duplicating data.

axis : int or str, optional

Unused.

method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}

Method to use for filling holes in reindexed DataFrame.

Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- * None (default): don't fill gaps

- * pad / ffill: Propagate last valid observation forward to next valid.

- * backfill / bfill: Use next valid observation to fill gap.

- * nearest: Use nearest valid observations to fill gap.

copy : bool, default True

Return a new object, even if the passed indexes are the same.

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit : int, default None

Maximum number of consecutive elements to forward or backward fill.

tolerance : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation ``abs(index[indexer] - target) <= tolerance``.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

Series with changed index.

See Also

DataFrame.set_index : Set row labels.

DataFrame.reset_index : Remove row labels or move them to new columns.

DataFrame.reindex_like : Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

```
* ``(index=index_labels, columns=column_labels, ...)``
```

```
* ``(labels, axis={'index', 'columns'}, ...)``
```

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                   'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                   index=index)
>>> df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
           http_status  response_time
Safari              404.0           0.07
```

Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
      http_status  response_time
Safari           404            0.07
Iceweasel         0             0.00
Comodo Dragon     0             0.00
IE10              404            0.08
Chrome            200            0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
      http_status  response_time
Safari           404            0.07
Iceweasel      missing          missing
Comodo Dragon   missing          missing
IE10            404            0.08
Chrome          200            0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
      http_status  user_agent
Firefox          200         NaN
Chrome           200         NaN
Safari           404         NaN
IE10              404         NaN
Konqueror        301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
      http_status  user_agent
Firefox          200         NaN
Chrome           200         NaN
Safari           404         NaN
```

IE10	404	NaN
Konqueror	301	NaN

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                     index=date_index)
```

```
>>> df2
           prices
2010-01-01  100.0
2010-01-02  101.0
2010-01-03    NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
```

```
           prices
2009-12-29    NaN
2009-12-30    NaN
2009-12-31    NaN
2010-01-01  100.0
2010-01-02  101.0
2010-01-03    NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
2010-01-07    NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`.

If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN`

values, pass ``bfill`` as an argument to the ``method`` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
```

```
           prices
2009-12-29  100.0
2009-12-30  100.0
2009-12-31  100.0
2010-01-01  100.0
2010-01-02  101.0
2010-01-03   NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
2010-01-07   NaN
```

Please note that the ``NaN`` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the ``NaN`` values present in the original dataframe, use the ``fillna()`` method.

See the :ref:`user guide <basics.reindexing>` for more.

```
rename(self, index: 'Renamer | Hashable | None' = None, *, axis: 'Axis | None' = None, c
    Alter Series index labels or name.
```

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Alternatively, change ``Series.name`` with a scalar value.

See the :ref:`user guide <basics.rename>` for more.

Parameters

index : scalar, hashable sequence, dict-like or function optional

Functions or dict-like are transformations to apply to the index.

Scalar or hashable sequence-like will alter the ``Series.name`` attribute.

axis : {0 or 'index'}

```

|         Unused. Parameter needed for compatibility with DataFrame.
| copy : bool, default True
|         Also copy underlying data.
| inplace : bool, default False
|         Whether to return a new Series. If True the value of copy is ignored.
| level : int or level name, default None
|         In case of MultiIndex, only rename labels in the specified level.
| errors : {'ignore', 'raise'}, default 'ignore'
|         If 'raise', raise `KeyError` when a `dict-like mapper` or
|         `index` contains labels that are not present in the index being transformed.
|         If 'ignore', existing keys will be renamed and extra keys will be ignored.
|
| Returns
| -----
| Series or None
|     Series with index labels or name altered or None if ``inplace=True``.
|
| See Also
| -----
| DataFrame.rename : Corresponding DataFrame method.
| Series.rename_axis : Set the name of the axis.
|
| Examples
| -----
| >>> s = pd.Series([1, 2, 3])
| >>> s
| 0    1
| 1    2
| 2    3
| dtype: int64
| >>> s.rename("my_name") # scalar, changes Series.name
| 0    1
| 1    2
| 2    3
| Name: my_name, dtype: int64
| >>> s.rename(lambda x: x ** 2) # function, changes labels
| 0    1
| 1    2
| 4    3
| dtype: int64
| >>> s.rename({1: 3, 2: 5}) # mapping, changes labels
| 0    1
| 3    2

```


5 3

dtype: int64

rename_axis(self: 'Series', mapper: 'IndexLabel | lib.NoDefault' = <no_default>, *, index=None)
Set the name of the axis for the index or columns.

Parameters

mapper : scalar, list-like, optional

Value to set the axis name attribute.

index, columns : scalar, list-like, dict-like or function, optional

A scalar, list-like, dict-like or functions transformations to apply to that axis' values.

Note that the ``columns`` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

Use either ``mapper`` and ``axis`` to specify the axis to target with ``mapper``, or ``index`` and/or ``columns``.

axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to rename. For `Series` this parameter is unused and defaults to 0.

copy : bool, default None

Also copy underlying data.

inplace : bool, default False

Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None

The same type as the caller or None if ``inplace=True``.

See Also

Series.rename : Alter Series index labels or name.

DataFrame.rename : Alter DataFrame index labels or name.

Index.rename : Set new names on index.

Notes

``DataFrame.rename_axis`` supports two calling conventions

```
| * ``(index=index_mapper, columns=columns_mapper, ...)``  
| * ``(mapper, axis={'index', 'columns'}, ...)``  
|
```

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter ``copy`` is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

****Series****

```
| >>> s = pd.Series(["dog", "cat", "monkey"])  
| >>> s  
| 0      dog  
| 1      cat  
| 2  monkey  
| dtype: object  
| >>> s.rename_axis("animal")  
| animal  
| 0  dog  
| 1  cat  
| 2  monkey  
| dtype: object
```

****DataFrame****

```
| >>> df = pd.DataFrame({"num_legs": [4, 4, 2],  
| ...                    "num_arms": [0, 0, 2]},  
| ...                    ["dog", "cat", "monkey"])  
| >>> df  
|          num_legs  num_arms  
| dog              4         0  
| cat              4         0  
| monkey          2         2  
| >>> df = df.rename_axis("animal")
```

```

|     >>> df
|
|           num_legs  num_arms
| animal
| dog                4        0
| cat                4        0
| monkey            2        2
| >>> df = df.rename_axis("limbs", axis="columns")
| >>> df
| limbs  num_legs  num_arms
| animal
| dog                4        0
| cat                4        0
| monkey            2        2
|
| **MultiIndex**
|
| >>> df.index = pd.MultiIndex.from_product(['mammal'],
| ...                                     ['dog', 'cat', 'monkey']],
| ...                                     names=['type', 'name'])
| >>> df
| limbs          num_legs  num_arms
| type  name
| mammal dog                4        0
|         cat                4        0
|         monkey            2        2
|
| >>> df.rename_axis(index={'type': 'class'})
| limbs          num_legs  num_arms
| class  name
| mammal dog                4        0
|         cat                4        0
|         monkey            2        2
|
| >>> df.rename_axis(columns=str.upper)
| LIMBS          num_legs  num_arms
| type  name
| mammal dog                4        0
|         cat                4        0
|         monkey            2        2
|
| reorder_levels(self, order: 'Sequence[Level]') -> 'Series'
|   Rearrange index levels using input order.
|

```

```

|     May not drop or duplicate levels.
|
|     Parameters
|     -----
|     order : list of int representing new level order
|             Reference level by number or key.
|
|     Returns
|     -----
|     type of caller (new object)
|
| repeat(self, repeats: 'int | Sequence[int]', axis: 'None' = None) -> 'Series'
|     Repeat elements of a Series.
|
|     Returns a new Series where each element of the current Series
|     is repeated consecutively a given number of times.
|
|     Parameters
|     -----
|     repeats : int or array of ints
|             The number of repetitions for each element. This should be a
|             non-negative integer. Repeating 0 times will return an empty
|             Series.
|     axis : None
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         Newly created Series with repeated elements.
|
|     See Also
|     -----
|     Index.repeat : Equivalent function for Index.
|     numpy.repeat : Similar method for :class:`numpy.ndarray`.
|
|     Examples
|     -----
|     >>> s = pd.Series(['a', 'b', 'c'])
|     >>> s
|     0    a
|     1    b
|     2    c

```

```

| dtype: object
| >>> s.repeat(2)
| 0    a
| 0    a
| 1    b
| 1    b
| 2    c
| 2    c
| dtype: object
| >>> s.repeat([1, 2, 3])
| 0    a
| 1    b
| 1    b
| 2    c
| 2    c
| 2    c
| dtype: object
|
| replace(self, to_replace=None, value=<no_default>, *, inplace: 'bool' = False, limit: 'int' = None)
| Replace values given in `to_replace` with `value`.
|
| Values of the Series are replaced with other values dynamically.
|
| This differs from updating with ``.loc`` or ``.iloc``, which require
| you to specify a location to update with some value.
|
| Parameters
| -----
| to_replace : str, regex, list, dict, Series, int, float, or None
|             How to find the values that will be replaced.
|
| * numeric, str or regex:
|
|   - numeric: numeric values equal to `to_replace` will be
|     replaced with `value`
|   - str: string exactly matching `to_replace` will be replaced
|     with `value`
|   - regex: regexs matching `to_replace` will be replaced with
|     `value`
|
| * list of str, regex, or numeric:
|
|   - First, if `to_replace` and `value` are both lists, they

```

- `**must**` be the same length.
- Second, if `regex=True` then all of the strings in `**both**` lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for `value`` since there are only a few possible substitution regexes you can use.
- `str`, `regex` and `numeric` rules apply as above.

`* dict:`

- Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way, the optional `value`` parameter should not be given.
- For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in `value``. The `value`` parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The optional `value`` parameter should not be specified to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) `**cannot**` be regular expressions.

`* None:`

- This means that the `regex`` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If `value`` is also `None` then this `**must**` be a nested dictionary or Series.

See the examples section for examples of each of these.

`value`` : scalar, dict, list, str, regex, default None
 Value to replace any values matching `to_replace`` with.
 For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such

objects are also allowed.

`inplace` : bool, default False
If True, performs operation inplace and returns None.

`limit` : int, default None
Maximum size gap to forward or backward fill.

`regex` : bool or same types as ``to_replace``, default False
Whether to interpret ``to_replace`` and/or ``value`` as regular expressions. If this is ```True``` then ``to_replace`` *must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case ``to_replace`` must be ```None```.

`method` : {'pad', 'ffill', 'bfill'}
The method to use when for replacement, when ``to_replace`` is a scalar, list or tuple and ``value`` is ```None```.

Returns

Series

Object after replacement.

Raises

AssertionError

* If ``regex`` is not a ```bool``` and ``to_replace`` is not ```None```.

TypeError

* If ``to_replace`` is not a scalar, array-like, ```dict```, or ```None```
* If ``to_replace`` is a ```dict``` and ``value`` is not a ```list```, ```dict```, ```ndarray```, or ```Series```
* If ``to_replace`` is ```None``` and ``regex`` is not compilable into a regular expression or is a list, dict, ndarray, or Series.
* When replacing multiple ```bool``` or ```datetime64``` objects and the arguments to ``to_replace`` does not match the type of the value being replaced

ValueError

* If a ```list``` or an ```ndarray``` is passed to ``to_replace`` and ``value`` but they are not the same length.

See Also

```
| Series.fillna : Fill NA values.
| Series.where : Replace values based on boolean condition.
| Series.str.replace : Simple string replacement.
```

| Notes

| -----

- | * Regex substitution is performed under the hood with ``re.sub``. The rules for substitution for ``re.sub`` are the same.
- | * Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- | * This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- | * When dict is used as the `to_replace` value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

| Examples

| -----

| ****Scalar `to_replace` and `value`****

```
| >>> s = pd.Series([1, 2, 3, 4, 5])
```

```
| >>> s.replace(1, 5)
```

```
| 0    5
| 1    2
| 2    3
| 3    4
| 4    5
```

```
| dtype: int64
```

```
| >>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
```

```
| ...           'B': [5, 6, 7, 8, 9],
```

```
| ...           'C': ['a', 'b', 'c', 'd', 'e']})
```

```
| >>> df.replace(0, 5)
```

```
|      A  B  C
| 0  5  5  a
| 1  1  6  b
| 2  2  7  c
| 3  3  8  d
| 4  4  9  e
```



```

|
|
| **List-like `to_replace`**
|
| >>> df.replace([0, 1, 2, 3], 4)
|
|      A B C
| 0  4  5  a
| 1  4  6  b
| 2  4  7  c
| 3  4  8  d
| 4  4  9  e
|
|
| >>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
|
|      A B C
| 0  4  5  a
| 1  3  6  b
| 2  2  7  c
| 3  1  8  d
| 4  4  9  e
|
|
| >>> s.replace([1, 2], method='bfill')
|
| 0    3
| 1    3
| 2    3
| 3    4
| 4    5
|
| dtype: int64
|
|
| **dict-like `to_replace`**
|
| >>> df.replace({0: 10, 1: 100})
|
|      A B C
| 0  10  5  a
| 1 100  6  b
| 2   2  7  c
| 3   3  8  d
| 4   4  9  e
|
|
| >>> df.replace({'A': 0, 'B': 5}, 100)
|
|      A  B C
| 0 100 100 a
| 1   1   6 b
| 2   2   7 c
| 3   3   8 d

```

```

| 4 4 9 e
|
| >>> df.replace({'A': {0: 100, 4: 400}})
|      A B C
| 0 100 5 a
| 1 1 6 b
| 2 2 7 c
| 3 3 8 d
| 4 400 9 e
|
| **Regular expression `to_replace`**
|
| >>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
| ...                    'B': ['abc', 'bar', 'xyz']})
| >>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
|      A B
| 0 new abc
| 1 foo new
| 2 bait xyz
|
| >>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
|      A B
| 0 new abc
| 1 foo bar
| 2 bait xyz
|
| >>> df.replace(regex=r'^ba.$', value='new')
|      A B
| 0 new abc
| 1 foo new
| 2 bait xyz
|
| >>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
|      A B
| 0 new abc
| 1 xyz new
| 2 bait xyz
|
| >>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
|      A B
| 0 new abc
| 1 new new
| 2 bait xyz

```

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the `value` parameter.

`s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1     None
2     None
3        b
4     None
dtype: object
```

When `value` is not explicitly passed and `to_replace` is a scalar, list or tuple, `replace` uses the `method` parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case.

```
>>> s.replace('a')
0      10
1      10
2      10
3        b
4        b
dtype: object
```

On the other hand, if `None` is explicitly passed for `value`, it will be respected:

```
>>> s.replace('a', None)
0      10
1     None
2     None
3        b
4     None
dtype: object
```

```

|
|     .. versionchanged:: 1.4.0
|         Previously the explicit ``None`` was silently ignored.
|
| resample(self, rule, axis: 'Axis' = 0, closed: 'str | None' = None, label: 'str | None' = None,
|         Resample time-series data.
|
| Convenience method for frequency conversion and resampling of time series.
| The object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`,
| or `TimedeltaIndex`), or the caller must pass the label of a datetime-like
| series/index to the `on`/`level` keyword parameter.
|
| Parameters
| -----
| rule : DateOffset, Timedelta or str
|     The offset string or object representing target conversion.
| axis : {0 or 'index', 1 or 'columns'}, default 0
|     Which axis to use for up- or down-sampling. For `Series` this parameter
|     is unused and defaults to 0. Must be
|     `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
| closed : {'right', 'left'}, default None
|     Which side of bin interval is closed. The default is 'left'
|     for all frequency offsets except for 'M', 'A', 'Q', 'BM',
|     'BA', 'BQ', and 'W' which all have a default of 'right'.
| label : {'right', 'left'}, default None
|     Which bin edge label to label bucket with. The default is 'left'
|     for all frequency offsets except for 'M', 'A', 'Q', 'BM',
|     'BA', 'BQ', and 'W' which all have a default of 'right'.
| convention : {'start', 'end', 's', 'e'}, default 'start'
|     For `PeriodIndex` only, controls whether to use the start or
|     end of `rule`.
| kind : {'timestamp', 'period'}, optional, default None
|     Pass 'timestamp' to convert the resulting index to a
|     `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`.
|     By default the input representation is retained.
|
| on : str, optional
|     For a DataFrame, column to use instead of index for resampling.
|     Column must be datetime-like.
| level : str or int, optional
|     For a MultiIndex, level (name or number) to use for
|     resampling. `level` must be datetime-like.
| origin : Timestamp or str, default 'start_day'

```

The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index.

If string, must be one of the following:

- 'epoch': `origin` is 1970-01-01
- 'start': `origin` is the first value of the timeseries
- 'start_day': `origin` is the first day at midnight of the timeseries

.. versionadded:: 1.1.0

- 'end': `origin` is the last value of the timeseries
- 'end_day': `origin` is the ceiling midnight of the last day

.. versionadded:: 1.3.0

offset : Timedelta or str, default is None

An offset timedelta added to the origin.

.. versionadded:: 1.1.0

group_keys : bool, default False

Whether to include the group keys in the result index when using ``.apply()`` on the resampled object.

.. versionadded:: 1.5.0

Not specifying ``group_keys`` will retain values-dependent behavior from pandas 1.4 and earlier (see :ref:`pandas 1.5.0 Release notes <whatsnew_150.enhancements.resample_group_keys>` for examples).

.. versionchanged:: 2.0.0

``group_keys`` now defaults to ``False``.

Returns

pandas.core.Resampler

:class:`~pandas.core.Resampler` object.

See Also

Series.resample : Resample a Series.

DataFrame.resample : Resample a DataFrame.

groupby : Group Series by mapping, function, label, or list of labels.
asfreq : Reindex a Series with the given frequency without grouping.

Notes

See the `user guide

<https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling>
for more.

To learn more about the offset strings, please see `this link

<https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
```

```
>>> series = pd.Series(range(9), index=index)
```

```
>>> series
```

```
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
```

```
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values
of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
```

```
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
```

```
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each
bin using the right edge instead of the left. Please note that the
value in the bucket used as the label is not included in the bucket,
which it labels. For example, in the original series the

bucket ``2000-01-01 00:03:00`` contains the value 3, but the summed value in the resampled bucket with the label ``2000-01-01 00:03:00`` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00    3
2000-01-01 00:06:00   12
2000-01-01 00:09:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]    # Select first 5 rows
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the ``NaN`` values using the ``ffill`` method.

```
>>> series.resample('30S').ffill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

```

| Upsample the series into 30 second bins and fill the
| ``NaN`` values using the ``bfill`` method.
|
| >>> series.resample('30S').bfill()[0:5]
| 2000-01-01 00:00:00    0
| 2000-01-01 00:00:30    1
| 2000-01-01 00:01:00    1
| 2000-01-01 00:01:30    2
| 2000-01-01 00:02:00    2
| Freq: 30S, dtype: int64
|
| Pass a custom function via ``apply``
|
| >>> def custom_resampler(arraylike):
| ...     return np.sum(arraylike) + 5
| ...
| >>> series.resample('3T').apply(custom_resampler)
| 2000-01-01 00:00:00     8
| 2000-01-01 00:03:00    17
| 2000-01-01 00:06:00    26
| Freq: 3T, dtype: int64
|
| For a Series with a PeriodIndex, the keyword `convention` can be
| used to control whether to use the start or end of `rule`.
|
| Resample a year by quarter using 'start' `convention`. Values are
| assigned to the first quarter of the period.
|
| >>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
| ...                                           freq='A',
| ...                                           periods=2))
| >>> s
| 2012    1
| 2013    2
| Freq: A-DEC, dtype: int64
| >>> s.resample('Q', convention='start').asfreq()
| 2012Q1    1.0
| 2012Q2    NaN
| 2012Q3    NaN
| 2012Q4    NaN
| 2013Q1    2.0
| 2013Q2    NaN
| 2013Q3    NaN

```



```

| 2013Q4    NaN
| Freq: Q-DEC, dtype: float64
|
| Resample quarters by month using 'end' `convention`. Values are
| assigned to the last month of the period.
|
| >>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
| ...                                     freq='Q',
| ...                                     periods=4))
| >>> q
| 2018Q1    1
| 2018Q2    2
| 2018Q3    3
| 2018Q4    4
| Freq: Q-DEC, dtype: int64
| >>> q.resample('M', convention='end').asfreq()
| 2018-03    1.0
| 2018-04    NaN
| 2018-05    NaN
| 2018-06    2.0
| 2018-07    NaN
| 2018-08    NaN
| 2018-09    3.0
| 2018-10    NaN
| 2018-11    NaN
| 2018-12    4.0
| Freq: M, dtype: float64
|
| For DataFrame objects, the keyword `on` can be used to specify the
| column instead of the index for resampling.
|
| >>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
| ...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
| >>> df = pd.DataFrame(d)
| >>> df['week_starting'] = pd.date_range('01/01/2018',
| ...                                     periods=8,
| ...                                     freq='W')
| >>> df
|    price  volume week_starting
| 0     10     50   2018-01-07
| 1     11     60   2018-01-14
| 2      9     40   2018-01-21
| 3     13    100   2018-01-28

```

```

| 4      14      50      2018-02-04
| 5      18     100      2018-02-11
| 6      17      40      2018-02-18
| 7      19      50      2018-02-25
| >>> df.resample('M', on='week_starting').mean()
|           price  volume
| week_starting
| 2018-01-31     10.75    62.5
| 2018-02-28     17.00    60.0

```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on which level the resampling needs to take place.

```

| >>> days = pd.date_range('1/1/2000', periods=4, freq='D')
| >>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
| ...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
| >>> df2 = pd.DataFrame(
| ...     d2,
| ...     index=pd.MultiIndex.from_product(
| ...         [days, ['morning', 'afternoon']]
| ...     )
| ... )
| >>> df2
|
|           price  volume
| 2000-01-01 morning      10      50
|           afternoon     11      60
| 2000-01-02 morning      9      40
|           afternoon     13     100
| 2000-01-03 morning     14      50
|           afternoon     18     100
| 2000-01-04 morning     17      40
|           afternoon     19      50
| >>> df2.resample('D', level=0).sum()
|           price  volume
| 2000-01-01     21     110
| 2000-01-02     22     140
| 2000-01-03     32     150
| 2000-01-04     36      90

```

If you want to adjust the start of the bins based on a fixed timestamp:

```

| >>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
| >>> rng = pd.date_range(start, end, freq='7min')

```

```

|     >>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
|     >>> ts
|     2000-10-01 23:30:00    0
|     2000-10-01 23:37:00    3
|     2000-10-01 23:44:00    6
|     2000-10-01 23:51:00    9
|     2000-10-01 23:58:00   12
|     2000-10-02 00:05:00   15
|     2000-10-02 00:12:00   18
|     2000-10-02 00:19:00   21
|     2000-10-02 00:26:00   24
|     Freq: 7T, dtype: int64
|
|     >>> ts.resample('17min').sum()
|     2000-10-01 23:14:00    0
|     2000-10-01 23:31:00    9
|     2000-10-01 23:48:00   21
|     2000-10-02 00:05:00   54
|     2000-10-02 00:22:00   24
|     Freq: 17T, dtype: int64
|
|     >>> ts.resample('17min', origin='epoch').sum()
|     2000-10-01 23:18:00    0
|     2000-10-01 23:35:00   18
|     2000-10-01 23:52:00   27
|     2000-10-02 00:09:00   39
|     2000-10-02 00:26:00   24
|     Freq: 17T, dtype: int64
|
|     >>> ts.resample('17min', origin='2000-01-01').sum()
|     2000-10-01 23:24:00    3
|     2000-10-01 23:41:00   15
|     2000-10-01 23:58:00   45
|     2000-10-02 00:15:00   45
|     Freq: 17T, dtype: int64
|
|     If you want to adjust the start of the bins with an `offset` Timedelta, the two
|     following lines are equivalent:
|
|     >>> ts.resample('17min', origin='start').sum()
|     2000-10-01 23:30:00    9
|     2000-10-01 23:47:00   21
|     2000-10-02 00:04:00   54

```

```
| 2000-10-02 00:21:00    24
| Freq: 17T, dtype: int64
|
```

```
| >>> ts.resample('17min', offset='23h30min').sum()
| 2000-10-01 23:30:00     9
| 2000-10-01 23:47:00    21
| 2000-10-02 00:04:00    54
| 2000-10-02 00:21:00    24
| Freq: 17T, dtype: int64
|
```

| If you want to take the largest Timestamp as the end of the bins:

```
| >>> ts.resample('17min', origin='end').sum()
| 2000-10-01 23:35:00     0
| 2000-10-01 23:52:00    18
| 2000-10-02 00:09:00    27
| 2000-10-02 00:26:00    63
| Freq: 17T, dtype: int64
|
```

| In contrast with the ``start_day``, you can use ``end_day`` to take the ceiling
| midnight of the largest Timestamp as the end of the bins and drop the bins
| not containing data:

```
| >>> ts.resample('17min', origin='end_day').sum()
| 2000-10-01 23:38:00     3
| 2000-10-01 23:55:00    15
| 2000-10-02 00:12:00    45
| 2000-10-02 00:29:00    45
| Freq: 17T, dtype: int64
|
```

```
| reset_index(self, level: 'IndexLabel' = None, *, drop: 'bool' = False, name: 'Level' = <
| Generate a new DataFrame or Series with the index reset.
```

| This is useful when the index needs to be treated as a column, or
| when the index is meaningless and needs to be reset to the default
| before another operation.

| Parameters

```
| -----
| level : int, str, tuple, or list, default optional
```

| For a Series with a MultiIndex, only remove the specified levels
| from the index. Removes all levels by default.

```
| drop : bool, default False
```

```
|         Just reset the index, without inserting it as a column in
|         the new DataFrame.
| name : object, optional
|         The name to use for the column containing the original Series
|         values. Uses ``self.name`` by default. This argument is ignored
|         when `drop` is True.
| inplace : bool, default False
|         Modify the Series in place (do not create a new object).
| allow_duplicates : bool, default False
|         Allow duplicate column labels to be created.
```

```
| .. versionadded:: 1.5.0
```

Returns

```
| -----
```

Series or DataFrame or None

When `drop` is False (the default), a DataFrame is returned.

The newly created columns will come first in the DataFrame, followed by the original Series values.

When `drop` is True, a `Series` is returned.

In either case, if ``inplace=True``, no value is returned.

See Also

```
| -----
```

DataFrame.reset_index: Analogous function for DataFrame.

Examples

```
| -----
```

```
| >>> s = pd.Series([1, 2, 3, 4], name='foo',
| ...               index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
| >>> s.reset_index()
```

```
|   idx  foo
| 0   a    1
| 1   b    2
| 2   c    3
| 3   d    4
```

To specify the name of the new column use `name`.

```
| >>> s.reset_index(name='values')
```

```

|     idx  values
|  0   a      1
|  1   b      2
|  2   c      3
|  3   d      4

```

To generate a new Series with the default set `drop` to True.

```

| >>> s.reset_index(drop=True)
|  0   1
|  1   2
|  2   3
|  3   4
| Name: foo, dtype: int64

```

The `level` parameter is interesting for Series with a multi-level index.

```

| >>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
| ...           np.array(['one', 'two', 'one', 'two'])]
| >>> s2 = pd.Series(
| ...     range(4), name='foo',
| ...     index=pd.MultiIndex.from_arrays(arrays,
| ...                                     names=['a', 'b']))

```

To remove a specific level from the Index, use `level`.

```

| >>> s2.reset_index(level='a')
|      a  foo
|  b
| one bar   0
| two bar   1
| one baz   2
| two baz   3

```

If `level` is not set, all levels are removed from the Index.

```

| >>> s2.reset_index()
|      a  b  foo
|  0 bar one   0
|  1 bar two   1
|  2 baz one   2
|  3 baz two   3

```

```

|
| rfloordiv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Integer division of series and other, element-wise (binary operator `rfloordi
|
|     Equivalent to ``other // series``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.floordiv : Element-wise Integer division, see
|         `Python documentation
|         <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|         for more details.
|
|     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    inf
|     c    inf
|     d    0.0
|     e    NaN
|     dtype: float64
|
| rmod(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Modulo of series and other, element-wise (binary operator `rmod`).
|
|     Equivalent to ``other % series``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.mod : Element-wise Modulo, see
|         `Python documentation

```


<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types> for more details.

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
```

`rmul(self, other, level=None, fill_value=None, axis: 'Axis' = 0)`

Return Multiplication of series and other, element-wise (binary operator ``rmul``).

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

Parameters

`other` : Series or scalar value

`level` : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

`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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|         Unused. Parameter needed for compatibility with DataFrame.
|
| Returns
| -----
| Series
|     The result of the operation.
|
| See Also
| -----
| Series.mul : Element-wise Multiplication, see
|     `Python documentation
|     <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|     for more details.
|
| 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
| d    0.0
| e    NaN
| dtype: float64
|
| round(self, decimals: 'int' = 0, *args, **kwargs) -> 'Series'
|     Round each value in a Series to the given number of decimals.
|

```

Parameters

`decimals` : int, default 0

Number of decimal places to round to. If `decimals` is negative, it specifies the number of positions to the left of the decimal point.

`*args`, `**kwargs`

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

Returns

Series

Rounded values of the Series.

See Also

`numpy.around` : Round values of an `np.array`.

`DataFrame.round` : Round values of a `DataFrame`.

Examples

```
>>> s = pd.Series([0.1, 1.3, 2.7])
```

```
>>> s.round()
```

```
0    0.0
```

```
1    1.0
```

```
2    3.0
```

```
dtype: float64
```

```
rpow(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
```

Return Exponential power of series and other, element-wise (binary operator ``rpow``).

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

Parameters

`other` : Series or scalar value

`level` : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

`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 of filling (at that location) will be missing.
axis : {0 or 'index'}
Unused. Parameter needed for compatibility with DataFrame.

Returns

Series

The result of the operation.

See Also

Series.pow : Element-wise Exponential power, see

`Python documentation`

`<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>` for more details.

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.pow(b, fill_value=0)
```

```
a    1.0
```

```
b    1.0
```

```
c    1.0
```

```
d    0.0
```

```
e    NaN
```

```
dtype: float64
```

```
rsub(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
```

Return Subtraction of series and other, element-wise (binary operator ``rsub``).

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

Parameters

other : Series or scalar value

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

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 of filling (at that location) will be missing.

axis : {0 or 'index'}

Unused. Parameter needed for compatibility with DataFrame.

Returns

Series

The result of the operation.

See Also

Series.sub : Element-wise Subtraction, see

Python documentation

<<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>>`_` for more details.

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.subtract(b, fill_value=0)
|     a    0.0
|     b    1.0
|     c    1.0
|     d   -1.0
|     e    NaN
|     dtype: float64
|
| rtruediv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|     Return Floating division of series and other, element-wise (binary operator `rtruediv`
|
|     Equivalent to ``other / series``, but with support to substitute a fill_value for
|     missing data in either one of the inputs.
|
|     Parameters
|     -----
|     other : Series or scalar value
|     level : int or name
|             Broadcast across a level, matching Index values on the
|             passed MultiIndex level.
|     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 of filling (at that location) will be missing.
|     axis : {0 or 'index'}
|             Unused. Parameter needed for compatibility with DataFrame.
|
|     Returns
|     -----
|     Series
|         The result of the operation.
|
|     See Also
|     -----
|     Series.truediv : Element-wise Floating division, see
|         `Python documentation
|         <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|         for more details.
|

```

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.divide(b, fill_value=0)
```

```
a    1.0
```

```
b    inf
```

```
c    inf
```

```
d    0.0
```

```
e    NaN
```

```
dtype: float64
```

```
searchsorted(self, value: 'NumpyValueArrayLike | ExtensionArray', side: "Literal['left',  
Find indices where elements should be inserted to maintain order.
```

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

.. note::

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

Parameters

value : array-like or scalar

Values to insert into `self`.

side : {'left', 'right'}, optional

If 'left', the index of the first suitable location found is given.

If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).

sorter : 1-D array-like, optional

Optional array of integer indices that sort `self` into ascending order. They are typically the result of ``np.argsort``.

Returns

int or array of int

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

See Also

sort_values : Sort by the values along either axis.

numpy.searchsorted : Similar method from NumPy.

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> ser = pd.Series([1, 2, 3])
```

```
>>> ser
```

```
0    1
```

```
1    2
```

```
2    3
```

```
dtype: int64
```

```
>>> ser.searchsorted(4)
```

```
3
```

```
>>> ser.searchsorted([0, 4])
```

```
array([0, 3])
```

```
>>> ser.searchsorted([1, 3], side='left')
```

```
array([0, 2])
```

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

```
array([1, 3])
```

```
>>> ser = pd.Series(pd.to_datetime(['3/11/2000', '3/12/2000', '3/13/2000']))
```



```

|     >>> ser
|     0    2000-03-11
|     1    2000-03-12
|     2    2000-03-13
|     dtype: datetime64[ns]
|
|     >>> ser.searchsorted('3/14/2000')
|     3
|
|     >>> ser = pd.Categorical(
|     ...     ['apple', 'bread', 'bread', 'cheese', 'milk'], ordered=True
|     ... )
|     >>> ser
|     ['apple', 'bread', 'bread', 'cheese', 'milk']
|     Categories (4, object): ['apple' < 'bread' < 'cheese' < 'milk']
|
|     >>> ser.searchsorted('bread')
|     1
|
|     >>> ser.searchsorted(['bread'], side='right')
|     array([3])

```

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

```

|     >>> ser = pd.Series([2, 1, 3])
|     >>> ser
|     0    2
|     1    1
|     2    3
|     dtype: int64
|
|     >>> ser.searchsorted(1) # doctest: +SKIP
|     0 # wrong result, correct would be 1

```

sem(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, ddof: 'int' = 1, numeric_only: bool = True)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis : {index (0)}

```

|         For `Series` this parameter is unused and defaults to 0.
| skipna : bool, default True
|         Exclude NA/null values. If an entire row/column is NA, the result
|         will be NA.
| ddof : int, default 1
|         Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
|         where N represents the number of elements.
| numeric_only : bool, default False
|         Include only float, int, boolean columns. Not implemented for Series.
|
| Returns
| -----
| scalar or Series (if level specified)
|
| set_axis(self, labels, *, axis: 'Axis' = 0, copy: 'bool | None' = None) -> 'Series'
| Assign desired index to given axis.
|
| Indexes for row labels can be changed by assigning
| a list-like or Index.
|
| Parameters
| -----
| labels : list-like, Index
|         The values for the new index.
|
| axis : {0 or 'index'}, default 0
|         The axis to update. The value 0 identifies the rows. For `Series`
|         this parameter is unused and defaults to 0.
|
| copy : bool, default True
|         Whether to make a copy of the underlying data.
|
| .. versionadded:: 1.5.0
|
| Returns
| -----
| Series
|     An object of type Series.
|
| See Also
| -----
| Series.rename_axis : Alter the name of the index.

```

Examples

```
>>> s = pd.Series([1, 2, 3])
```

```
>>> s
```

```
0    1
```

```
1    2
```

```
2    3
```

```
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0)
```

```
a    1
```

```
b    2
```

```
c    3
```

```
dtype: int64
```

```
shift(self, periods: 'int' = 1, freq=None, axis: 'Axis' = 0, fill_value: 'Hashable' = None)
```

Shift index by desired number of periods with an optional time `freq`.

When `freq` is not passed, shift the index without realigning the data.

If `freq` is passed (in this case, the index must be date or datetime,

or it will raise a `NotImplementedError`), the index will be

increased using the periods and the `freq`. `freq` can be inferred

when specified as "infer" as long as either freq or inferred_freq

attribute is set in the index.

Parameters

periods : int

Number of periods to shift. Can be positive or negative.

freq : DateOffset, tseries.offsets, timedelta, or str, optional

Offset to use from the tseries module or time rule (e.g. 'EOM').

If `freq` is specified then the index values are shifted but the

data is not realigned. That is, use `freq` if you would like to

extend the index when shifting and preserve the original data.

If `freq` is specified as "infer" then it will be inferred from

the freq or inferred_freq attributes of the index. If neither of

those attributes exist, a ValueError is thrown.

axis : {0 or 'index', 1 or 'columns', None}, default None

Shift direction. For `Series` this parameter is unused and defaults to 0.

fill_value : object, optional

The scalar value to use for newly introduced missing values.

the default depends on the dtype of `self`.

For numeric data, ``np.nan`` is used.

For datetime, timedelta, or period data, etc. :attr:`NaT` is used.
For extension dtypes, ``self.dtype.na_value`` is used.

.. versionchanged:: 1.1.0

Returns

Series

Copy of input object, shifted.

See Also

Index.shift : Shift values of Index.

DatetimeIndex.shift : Shift values of DatetimeIndex.

PeriodIndex.shift : Shift values of PeriodIndex.

Examples

```
>>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
...                    "Col2": [13, 23, 18, 33, 48],
...                    "Col3": [17, 27, 22, 37, 52]},
...                    index=pd.date_range("2020-01-01", "2020-01-05"))
```

```
>>> df
```

	Col1	Col2	Col3
2020-01-01	10	13	17
2020-01-02	20	23	27
2020-01-03	15	18	22
2020-01-04	30	33	37
2020-01-05	45	48	52

```
>>> df.shift(periods=3)
```

	Col1	Col2	Col3
2020-01-01	NaN	NaN	NaN
2020-01-02	NaN	NaN	NaN
2020-01-03	NaN	NaN	NaN
2020-01-04	10.0	13.0	17.0
2020-01-05	20.0	23.0	27.0

```
>>> df.shift(periods=1, axis="columns")
```

	Col1	Col2	Col3
2020-01-01	NaN	10	13
2020-01-02	NaN	20	23
2020-01-03	NaN	15	18

```

|      2020-01-04    NaN    30    33
|      2020-01-05    NaN    45    48
|
|      >>> df.shift( periods=3, fill_value=0)
|              Col1  Col2  Col3
|      2020-01-01     0     0     0
|      2020-01-02     0     0     0
|      2020-01-03     0     0     0
|      2020-01-04    10    13    17
|      2020-01-05    20    23    27

```

```

|      >>> df.shift( periods=3, freq="D")
|              Col1  Col2  Col3
|      2020-01-04    10    13    17
|      2020-01-05    20    23    27
|      2020-01-06    15    18    22
|      2020-01-07    30    33    37
|      2020-01-08    45    48    52

```

```

|      >>> df.shift( periods=3, freq="infer")
|              Col1  Col2  Col3
|      2020-01-04    10    13    17
|      2020-01-05    20    23    27
|      2020-01-06    15    18    22
|      2020-01-07    30    33    37
|      2020-01-08    45    48    52

```

```

| skew(self, axis: 'AxisInt | None' = 0, skipna: 'bool_t' = True, numeric_only: 'bool_t' =
|     Return unbiased skew over requested axis.

```

```

|     Normalized by N-1.

```

```

|     Parameters

```

```

|     -----
|     axis : {index (0)}

```

```

|         Axis for the function to be applied on.

```

```

|         For `Series` this parameter is unused and defaults to 0.

```

```

|         For DataFrames, specifying ``axis=None`` will apply the aggregation
|         across both axes.

```

```

|     .. versionadded:: 2.0.0

```

```

| skipna : bool, default True
|     Exclude NA/null values when computing the result.
| numeric_only : bool, default False
|     Include only float, int, boolean columns. Not implemented for Series.
|
| **kwargs
|     Additional keyword arguments to be passed to the function.
|
| Returns
| -----
| scalar or scalar
|
| sort_index(self, *, axis: 'Axis' = 0, level: 'IndexLabel' = None, ascending: 'bool | Sequence' = True, inplace: bool = False, kind: str = 'quicksort',
|             na_position: str = 'last', sort_remaining: bool = True, ignore_index: bool = False, key: Callable = None)
|     Sort Series by index labels.
|
| Returns a new Series sorted by label if `inplace` argument is
| ``False``, otherwise updates the original series and returns None.
|
| Parameters
| -----
| axis : {0 or 'index'}
|     Unused. Parameter needed for compatibility with DataFrame.
| level : int, optional
|     If not None, sort on values in specified index level(s).
| ascending : bool or list-like of bools, default True
|     Sort ascending vs. descending. When the index is a MultiIndex the
|     sort direction can be controlled for each level individually.
| inplace : bool, default False
|     If True, perform operation in-place.
| kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'
|     Choice of sorting algorithm. See also :func:`numpy.sort` for more
|     information. 'mergesort' and 'stable' are the only stable algorithms. For
|     DataFrames, this option is only applied when sorting on a single
|     column or label.
| na_position : {'first', 'last'}, default 'last'
|     If 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.
|     Not implemented for MultiIndex.
| sort_remaining : bool, default True
|     If True and sorting by level and index is multilevel, sort by other
|     levels too (in order) after sorting by specified level.
| ignore_index : bool, default False
|     If True, the resulting axis will be labeled 0, 1, ..., n - 1.
| key : callable, optional

```

If not None, apply the key function to the index values before sorting. This is similar to the `key` argument in the builtin `:meth:`sorted`` function, with the notable difference that this `key` function should be **vectorized**. It should expect an ``Index`` and return an ``Index`` of the same shape.

.. versionadded:: 1.1.0

Returns

Series or None

The original Series sorted by the labels or None if ``inplace=True``.

See Also

`DataFrame.sort_index`: Sort DataFrame by the index.

`DataFrame.sort_values`: Sort DataFrame by the value.

`Series.sort_values` : Sort Series by the value.

Examples

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, 4])
```

```
>>> s.sort_index()
```

```
1    c
```

```
2    b
```

```
3    a
```

```
4    d
```

```
dtype: object
```

Sort Descending

```
>>> s.sort_index(ascending=False)
```

```
4    d
```

```
3    a
```

```
2    b
```

```
1    c
```

```
dtype: object
```

By default NaNs are put at the end, but use ``na_position`` to place them at the beginning

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, np.nan])
```

```
>>> s.sort_index(na_position='first')
```

```
NaN    d
 1.0   c
 2.0   b
 3.0   a
dtype: object
```

Specify index level to sort

```
>>> arrays = [np.array(['qux', 'qux', 'foo', 'foo',
...                      'baz', 'baz', 'bar', 'bar']),
...           np.array(['two', 'one', 'two', 'one',
...                      'two', 'one', 'two', 'one'])]
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
>>> s.sort_index(level=1)
bar one    8
baz one    6
foo one    4
qux one    2
bar two    7
baz two    5
foo two    3
qux two    1
dtype: int64
```

Does not sort by remaining levels when sorting by levels

```
>>> s.sort_index(level=1, sort_remaining=False)
qux one    2
foo one    4
baz one    6
bar one    8
qux two    1
foo two    3
baz two    5
bar two    7
dtype: int64
```

Apply a key function before sorting

```
>>> s = pd.Series([1, 2, 3, 4], index=['A', 'b', 'C', 'd'])
>>> s.sort_index(key=lambda x : x.str.lower())
A    1
b    2
```



```

|      C      3
|      d      4
|      dtype: int64
|
|  sort_values(self, *, axis: 'Axis' = 0, ascending: 'bool | int | Sequence[bool] | Sequence
|      Sort by the values.
|
|      Sort a Series in ascending or descending order by some
|      criterion.
|
|      Parameters
|      -----
|
|      axis : {0 or 'index'}
|          Unused. Parameter needed for compatibility with DataFrame.
|      ascending : bool or list of bools, default True
|          If True, sort values in ascending order, otherwise descending.
|      inplace : bool, default False
|          If True, perform operation in-place.
|      kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'
|          Choice of sorting algorithm. See also :func:`numpy.sort` for more
|          information. 'mergesort' and 'stable' are the only stable algorithms.
|      na_position : {'first' or 'last'}, default 'last'
|          Argument 'first' puts NaNs at the beginning, 'last' puts NaNs at
|          the end.
|      ignore_index : bool, default False
|          If True, the resulting axis will be labeled 0, 1, ..., n - 1.
|      key : callable, optional
|          If not None, apply the key function to the series values
|          before sorting. This is similar to the `key` argument in the
|          builtin :meth:`sorted` function, with the notable difference that
|          this `key` function should be *vectorized*. It should expect a
|          ``Series`` and return an array-like.
|
|      .. versionadded:: 1.1.0
|
|      Returns
|      -----
|
|      Series or None
|          Series ordered by values or None if ``inplace=True``.
|
|      See Also
|      -----
|
|      Series.sort_index : Sort by the Series indices.

```

```
DataFrame.sort_values : Sort DataFrame by the values along either axis.  
DataFrame.sort_index : Sort DataFrame by indices.
```

Examples

```
-----  
>>> s = pd.Series([np.nan, 1, 3, 10, 5])  
>>> s  
0      NaN  
1      1.0  
2      3.0  
3     10.0  
4      5.0  
dtype: float64
```

Sort values ascending order (default behaviour)

```
>>> s.sort_values(ascending=True)  
1      1.0  
2      3.0  
4      5.0  
3     10.0  
0      NaN  
dtype: float64
```

Sort values descending order

```
>>> s.sort_values(ascending=False)  
3     10.0  
4      5.0  
2      3.0  
1      1.0  
0      NaN  
dtype: float64
```

Sort values putting NAs first

```
>>> s.sort_values(na_position='first')  
0      NaN  
1      1.0  
2      3.0  
4      5.0  
3     10.0  
dtype: float64
```

|
| Sort a series of strings
|

```
>>> s = pd.Series(['z', 'b', 'd', 'a', 'c'])  
>>> s  
0    z  
1    b  
2    d  
3    a  
4    c  
dtype: object
```

```
>>> s.sort_values()  
3    a  
1    b  
4    c  
2    d  
0    z  
dtype: object
```

| Sort using a key function. Your `key` function will be
| given the ``Series`` of values and should return an array-like.
|

```
>>> s = pd.Series(['a', 'B', 'c', 'D', 'e'])  
>>> s.sort_values()  
1    B  
3    D  
0    a  
2    c  
4    e  
dtype: object  
>>> s.sort_values(key=lambda x: x.str.lower())  
0    a  
1    B  
2    c  
3    D  
4    e  
dtype: object
```

| NumPy ufuncs work well here. For example, we can
| sort by the ``sin`` of the value
|

```
>>> s = pd.Series([-4, -2, 0, 2, 4])
```

```

|     >>> s.sort_values(key=np.sin)
|     1  -2
|     4   4
|     2   0
|     0  -4
|     3   2
|     dtype: int64
|
|     More complicated user-defined functions can be used,
|     as long as they expect a Series and return an array-like
|
|     >>> s.sort_values(key=lambda x: (np.tan(x.cumsum()))))
|     0  -4
|     3   2
|     4   4
|     1  -2
|     2   0
|     dtype: int64
|
|     std(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, ddof: 'int' = 1, numeric_
|     Return sample standard deviation over requested axis.
|
|     Normalized by N-1 by default. This can be changed using the ddof argument.
|
|     Parameters
|     -----
|     axis : {index (0)}
|           For `Series` this parameter is unused and defaults to 0.
|     skipna : bool, default True
|           Exclude NA/null values. If an entire row/column is NA, the result
|           will be NA.
|     ddof : int, default 1
|           Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
|           where N represents the number of elements.
|     numeric_only : bool, default False
|           Include only float, int, boolean columns. Not implemented for Series.
|
|     Returns
|     -----
|     scalar or Series (if level specified)
|
|     Notes
|     -----

```

To have the same behaviour as `numpy.std`, use `ddof=0` (instead of the default `ddof=1`)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                    ).set_index('person_id')
```

```
>>> df
      age  height
person_id
0        21    1.61
1        25    1.87
2        62    1.49
3        43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age          18.786076
height       0.237417
dtype: float64
```

Alternatively, `ddof=0` can be set to normalize by N instead of $N-1$:

```
>>> df.std(ddof=0)
age          16.269219
height       0.205609
dtype: float64
```

`sub(self, other, level=None, fill_value=None, axis: 'Axis' = 0)`

Return Subtraction of series and other, element-wise (binary operator `sub`).

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

Parameters

`other` : Series or scalar value

`level` : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

```
| 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 of filling (at that location) will be missing.
| axis : {0 or 'index'}
|     Unused. Parameter needed for compatibility with DataFrame.
```

Returns

```
| -----
```

Series

```
|     The result of the operation.
```

See Also

```
| -----
```

```
| Series.rsub : Reverse of the Subtraction operator, see
```

```
|     `Python documentation
```

```
|     <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>`_
|     for more details.
```

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.subtract(b, fill_value=0)
```

```
| a    0.0
```

```
| b    1.0
```

```
| c    1.0
```

```
| d   -1.0
```

```
| e    NaN
```

```
| dtype: float64
```

```

subtract = sub(self, other, level=None, fill_value=None, axis: 'Axis' = 0)

sum(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, numeric_only: 'bool_t' = 1
    Return the sum of the values over the requested axis.

    This is equivalent to the method ``numpy.sum``.

Parameters
-----
axis : {index (0)}
    Axis for the function to be applied on.
    For `Series` this parameter is unused and defaults to 0.

    For DataFrames, specifying ``axis=None`` will apply the aggregation
    across both axes.

    .. versionadded:: 2.0.0

skipna : bool, default True
    Exclude NA/null values when computing the result.
numeric_only : bool, default False
    Include only float, int, boolean columns. Not implemented for Series.

min_count : int, default 0
    The required number of valid values to perform the operation. If fewer than
    ``min_count`` non-NA values are present the result will be NA.
**kwargs
    Additional keyword arguments to be passed to the function.

Returns
-----
scalar or scalar

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.sum()
14
```

By default, the sum of an empty or all-NA Series is ``0``.

```
>>> pd.Series([], dtype="float64").sum() # min_count=0 is the default
0.0
```

This can be controlled with the ``min_count`` parameter. For example, if you'd like the sum of an empty series to be NaN, pass ``min_count=1``.

```
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

Thanks to the ``skipna`` parameter, ``min_count`` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

```
swaplevel(self, i: 'Level' = -2, j: 'Level' = -1, copy: 'bool | None' = None) -> 'Series'
```


Swap levels *i* and *j* in a `:class:`MultiIndex``.

Default is to swap the two innermost levels of the index.

Parameters

i, *j* : int or str

Levels of the indices to be swapped. Can pass level name as string.

copy : bool, default True

Whether to copy underlying data.

Returns

Series

Series with levels swapped in MultiIndex.

Examples

```
>>> s = pd.Series(
...     ["A", "B", "A", "C"],
...     index=[
...         ["Final exam", "Final exam", "Coursework", "Coursework"],
...         ["History", "Geography", "History", "Geography"],
...         ["January", "February", "March", "April"],
...     ],
... )
```

```
>>> s
Final exam History January A
           Geography February B
Coursework History March A
           Geography April C
dtype: object
```

In the following example, we will swap the levels of the indices.

Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour.

By not supplying any arguments for *i* and *j*, we swap the last and second to last indices.

```
>>> s.swaplevel()
```

```
Final exam January History A
           February Geography B
Coursework March History A
```

```
April      Geography      C
dtype: object
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```
>>> s.swaplevel(0)
January      History      Final exam      A
February     Geography     Final exam      B
March        History      Coursework      A
April        Geography     Coursework      C
dtype: object
```

We can also define explicitly which indices we want to swap by supplying values for both *i* and *j*. Here, we for example swap the first and second indices.

```
>>> s.swaplevel(0, 1)
History      Final exam      January          A
Geography    Final exam      February         B
History      Coursework      March            A
Geography    Coursework      April            C
dtype: object
```

```
take(self, indices, axis: 'Axis' = 0, **kwargs) -> 'Series'
Return the elements in the given *positional* indices along an axis.
```

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices : array-like

An array of ints indicating which positions to take.

axis : {0 or 'index', 1 or 'columns', None}, default 0

The axis on which to select elements. ``0`` means that we are selecting rows, ``1`` means that we are selecting columns.

For `Series` this parameter is unused and defaults to 0.

****kwargs**

For compatibility with `:meth:`numpy.take``. Has no effect on the output.

Returns

same type as caller

An array-like containing the elements taken from the object.

See Also

DataFrame.loc : Select a subset of a DataFrame by labels.

DataFrame.iloc : Select a subset of a DataFrame by positions.

numpy.take : Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
```

```
>>> df
   name  class  max_speed
0 falcon  bird    389.0
2 parrot  bird     24.0
3  lion  mammal    80.5
1 monkey  mammal     NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
   name  class  max_speed
0 falcon  bird    389.0
1 monkey  mammal     NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
   class  max_speed
0  bird    389.0
2  bird    24.0
```

```

3 mammal      80.5
1 mammal      NaN

```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```

>>> df.take([-1, -2])
      name  class  max_speed
1  monkey  mammal      NaN
3    lion  mammal      80.5

```

`to_dict(self, into: 'type[dict]' = <class 'dict'>) -> 'dict'`
 Convert Series to {label -> value} dict or dict-like object.

Parameters

`into` : class, default dict
 The collections.abc.Mapping subclass to use as the return object. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

Returns

collections.abc.Mapping
 Key-value representation of Series.

Examples

```

>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_dict()
{0: 1, 1: 2, 2: 3, 3: 4}
>>> from collections import OrderedDict, defaultdict
>>> s.to_dict(OrderedDict)
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> dd = defaultdict(list)
>>> s.to_dict(dd)
defaultdict(<class 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})

```

`to_frame(self, name: 'Hashable' = <no_default>) -> 'DataFrame'`
 Convert Series to DataFrame.

Parameters

```
| -----  
| name : object, optional  
|     The passed name should substitute for the series name (if it has  
|     one).
```

```
| Returns  
| -----
```

```
| DataFrame  
|     DataFrame representation of Series.
```

```
| Examples  
| -----
```

```
| >>> s = pd.Series(["a", "b", "c"],  
| ...               name="vals")  
| >>> s.to_frame()  
|      vals  
| 0      a  
| 1      b  
| 2      c
```

```
| to_markdown(self, buf: 'IO[str] | None' = None, mode: 'str' = 'wt', index: 'bool' = True  
|     Print Series in Markdown-friendly format.
```

```
| Parameters  
| -----
```

```
| buf : str, Path or StringIO-like, optional, default None  
|     Buffer to write to. If None, the output is returned as a string.  
| mode : str, optional  
|     Mode in which file is opened, "wt" by default.  
| index : bool, optional, default True  
|     Add index (row) labels.
```

```
| .. versionadded:: 1.1.0
```

```
| storage_options : dict, optional  
|     Extra options that make sense for a particular storage connection, e.g.  
|     host, port, username, password, etc. For HTTP(S) URLs the key-value pairs  
|     are forwarded to ``urllib.request.Request`` as header options. For other  
|     URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are  
|     forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more  
|     details, and for more examples on storage options refer `here  
|     <https://pandas.pydata.org/docs/user\_guide/io.html?  
|     highlight=storage\_options#reading-writing-remote-files>`_.
```

.. versionadded:: 1.2.0

****kwargs**

These parameters will be passed to ``tabulate``

[`<https://pypi.org/p/`tabulate`>`](https://pypi.org/p/)

Returns

str

Series in Markdown-friendly format.

Notes

Requires the ``tabulate`` <https://pypi.org/project/tabulate> package.

Examples

```
>>> s = pd.Series(["elk", "pig", "dog", "quetzal"], name="animal")
```

```
>>> print(s.to_markdown())
```

```
|   | animal |
|---:|:-----|
| 0 | elk     |
| 1 | pig     |
| 2 | dog     |
| 3 | quetzal |
```

Output markdown with a tabulate option.

```
>>> print(s.to_markdown(tablefmt="grid"))
```

```
+----+-----+
|   | animal |
+----+-----+
| 0 | elk     |
+----+-----+
| 1 | pig     |
+----+-----+
| 2 | dog     |
+----+-----+
| 3 | quetzal |
+----+-----+
```

`to_period(self, freq: 'str | None' = None, copy: 'bool | None' = None) -> 'Series'`

Convert Series from DatetimeIndex to PeriodIndex.

```

Parameters
-----
freq : str, default None
    Frequency associated with the PeriodIndex.
copy : bool, default True
    Whether or not to return a copy.

Returns
-----
Series
    Series with index converted to PeriodIndex.

Examples
-----
>>> idx = pd.DatetimeIndex(['2023', '2024', '2025'])
>>> s = pd.Series([1, 2, 3], index=idx)
>>> s = s.to_period()
>>> s
2023    1
2024    2
2025    3
Freq: A-DEC, dtype: int64

Viewing the index

>>> s.index
PeriodIndex(['2023', '2024', '2025'], dtype='period[A-DEC]')

to_string(self, buf: 'FilePath | WriteBuffer[str] | None' = None, na_rep: 'str' = 'NaN',
    Render a string representation of the Series.

Parameters
-----
buf : StringIO-like, optional
    Buffer to write to.
na_rep : str, optional
    String representation of NaN to use, default 'NaN'.
float_format : one-parameter function, optional
    Formatter function to apply to columns' elements if they are
    floats, default None.
header : bool, default True
    Add the Series header (index name).
index : bool, optional

```

```

|         Add index (row) labels, default True.
| length : bool, default False
|         Add the Series length.
| dtype : bool, default False
|         Add the Series dtype.
| name : bool, default False
|         Add the Series name if not None.
| max_rows : int, optional
|         Maximum number of rows to show before truncating. If None, show
|         all.
| min_rows : int, optional
|         The number of rows to display in a truncated repr (when number
|         of rows is above `max_rows`).
|
| Returns
| -----
| str or None
|         String representation of Series if ``buf=None``, otherwise None.
|
| to_timestamp(self, freq=None, how: "Literal['s', 'e', 'start', 'end']" = 'start', copy:
|         Cast to DatetimeIndex of Timestamps, at *beginning* of period.
|
| Parameters
| -----
| freq : str, default frequency of PeriodIndex
|         Desired frequency.
| how : {'s', 'e', 'start', 'end'}
|         Convention for converting period to timestamp; start of period
|         vs. end.
| copy : bool, default True
|         Whether or not to return a copy.
|
| Returns
| -----
| Series with DatetimeIndex
|
| Examples
| -----
| >>> idx = pd.PeriodIndex(['2023', '2024', '2025'], freq='Y')
| >>> s1 = pd.Series([1, 2, 3], index=idx)
| >>> s1
| 2023    1
| 2024    2

```



```

| 2025    3
| Freq: A-DEC, dtype: int64
|
| The resulting frequency of the Timestamps is `YearBegin`
|
| >>> s1 = s1.to_timestamp()
| >>> s1
| 2023-01-01    1
| 2024-01-01    2
| 2025-01-01    3
| Freq: AS-JAN, dtype: int64
|
| Using `freq` which is the offset that the Timestamps will have
|
| >>> s2 = pd.Series([1, 2, 3], index=idx)
| >>> s2 = s2.to_timestamp(freq='M')
| >>> s2
| 2023-01-31    1
| 2024-01-31    2
| 2025-01-31    3
| Freq: A-JAN, dtype: int64
|
| transform(self, func: 'AggFuncType', axis: 'Axis' = 0, *args, **kwargs) -> 'DataFrame | S
| Call ``func`` on self producing a Series with the same axis shape as self.
|
| Parameters
| -----
| func : function, str, list-like or dict-like
|       Function to use for transforming the data. If a function, must either
|       work when passed a Series or when passed to Series.apply. If func
|       is both list-like and dict-like, dict-like behavior takes precedence.
|
|       Accepted combinations are:
|
|         - function
|         - string function name
|         - list-like of functions and/or function names, e.g. ``[np.exp, 'sqrt']``
|         - dict-like of axis labels -> functions, function names or list-like of such.
| axis : {0 or 'index'}
|       Unused. Parameter needed for compatibility with DataFrame.
| *args
|       Positional arguments to pass to `func`.
| **kwargs

```

Keyword arguments to pass to `func`.

Returns

Series

A Series that must have the same length as self.

Raises

ValueError : If the returned Series has a different length than self.

See Also

Series.agg : Only perform aggregating type operations.

Series.apply : Invoke function on a Series.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See :ref:`gotchas.udf-mutation` for more details.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
```

```
>>> df
```

```
   A  B
0  0  1
1  1  2
2  2  3
```

```
>>> df.transform(lambda x: x + 1)
```

```
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting Series must have the same length as the input Series, it is possible to provide several input functions:

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

```
>>> s
```

```
0    0
1    1
```

```

|     2     2
| dtype: int64
| >>> s.transform([np.sqrt, np.exp])
|           sqrt      exp
| 0  0.000000  1.000000
| 1  1.000000  2.718282
| 2  1.414214  7.389056
|
| You can call transform on a GroupBy object:
|
| >>> df = pd.DataFrame({
| ...     "Date": [
| ...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
| ...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
| ...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
| ... })
| >>> df
|           Date  Data
| 0  2015-05-08     5
| 1  2015-05-07     8
| 2  2015-05-06     6
| 3  2015-05-05     1
| 4  2015-05-08    50
| 5  2015-05-07   100
| 6  2015-05-06    60
| 7  2015-05-05   120
| >>> df.groupby('Date')['Data'].transform('sum')
| 0     55
| 1    108
| 2     66
| 3    121
| 4     55
| 5    108
| 6     66
| 7    121
| Name: Data, dtype: int64
|
| >>> df = pd.DataFrame({
| ...     "c": [1, 1, 1, 2, 2, 2, 2],
| ...     "type": ["m", "n", "o", "m", "m", "n", "n"]
| ... })
| >>> df
|           c type

```

```

|   0 1  m
|   1 1  n
|   2 1  o
|   3 2  m
|   4 2  m
|   5 2  n
|   6 2  n
|   >>> df['size'] = df.groupby('c')['type'].transform(len)
|   >>> df
|      c type size
|   0 1  m    3
|   1 1  n    3
|   2 1  o    3
|   3 2  m    4
|   4 2  m    4
|   5 2  n    4
|   6 2  n    4

```

```

|   truediv(self, other, level=None, fill_value=None, axis: 'Axis' = 0)
|       Return Floating division of series and other, element-wise (binary operator `truediv`
|
|       Equivalent to ``series / other``, but with support to substitute a fill_value for
|       missing data in either one of the inputs.
|
|       Parameters
|       -----
|       other : Series or scalar value
|       level : int or name
|           Broadcast across a level, matching Index values on the
|           passed MultiIndex level.
|       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 of filling (at that location) will be missing.
|       axis : {0 or 'index'}
|           Unused. Parameter needed for compatibility with DataFrame.
|
|       Returns
|       -----
|       Series
|           The result of the operation.

```

See Also

Series.rtruediv : Reverse of the Floating division operator, see
`Python documentation
<<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>>`
for more details.

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.divide(b, fill_value=0)
a 1.0
b inf
c inf
d 0.0
e NaN
dtype: float64

unique(self) -> 'ArrayLike'

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique,
therefore does NOT sort.

Returns

ndarray or ExtensionArray

The unique values returned as a NumPy array. See Notes.

See Also

```
-----
Series.drop_duplicates : Return Series with duplicate values removed.
unique : Top-level unique method for any 1-d array-like object.
Index.unique : Return Index with unique values from an Index object.
```

Notes

```
-----
Returns the unique values as a NumPy array. In case of an
extension-array backed Series, a new
:class:`~api.extensions.ExtensionArray` of that type with just
the unique values is returned. This includes
```

- * Categorical
- * Period
- * Datetime with Timezone
- * Datetime without Timezone
- * Timedelta
- * Interval
- * Sparse
- * IntegerNA

See Examples section.

Examples

```
-----
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])

>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00']
Length: 1, dtype: datetime64[ns]

>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...           for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]
```

An Categorical will return categories in the order of appearance and with the same dtype.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
```

```

|     ['b', 'a', 'c']
|     Categories (3, object): ['a', 'b', 'c']
|     >>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
|     ...                                     ordered=True)).unique()
|     ['b', 'a', 'c']
|     Categories (3, object): ['a' < 'b' < 'c']
|
| unstack(self, level: 'IndexLabel' = -1, fill_value: 'Hashable' = None) -> 'DataFrame'
|     Unstack, also known as pivot, Series with MultiIndex to produce DataFrame.
|
|     Parameters
|     -----
|     level : int, str, or list of these, default last level
|             Level(s) to unstack, can pass level name.
|     fill_value : scalar value, default None
|             Value to use when replacing NaN values.
|
|     Returns
|     -----
|     DataFrame
|         Unstacked Series.
|
|     Notes
|     -----
|     Reference :ref:`the user guide <reshaping.stacking>` for more examples.
|
|     Examples
|     -----
|     >>> s = pd.Series([1, 2, 3, 4],
|     ...               index=pd.MultiIndex.from_product([['one', 'two'],
|     ...               ['a', 'b']]))
|     >>> s
|     one  a    1
|         b    2
|     two  a    3
|         b    4
|     dtype: int64
|
|     >>> s.unstack(level=-1)
|         a  b
|     one  1  2
|     two  3  4
|

```

```

|     >>> s.unstack(level=0)
|         one  two
|     a     1   3
|     b     2   4
|
| update(self, other: 'Series | Sequence | Mapping') -> 'None'
|     Modify Series in place using values from passed Series.
|
|     Uses non-NA values from passed Series to make updates. Aligns
|     on index.
|
|     Parameters
|     -----
|     other : Series, or object coercible into Series
|
|     Examples
|     -----
|     >>> s = pd.Series([1, 2, 3])
|     >>> s.update(pd.Series([4, 5, 6]))
|     >>> s
|     0     4
|     1     5
|     2     6
|     dtype: int64
|
|     >>> s = pd.Series(['a', 'b', 'c'])
|     >>> s.update(pd.Series(['d', 'e'], index=[0, 2]))
|     >>> s
|     0     d
|     1     b
|     2     e
|     dtype: object
|
|     >>> s = pd.Series([1, 2, 3])
|     >>> s.update(pd.Series([4, 5, 6, 7, 8]))
|     >>> s
|     0     4
|     1     5
|     2     6
|     dtype: int64
|
|     If ``other`` contains NaNs the corresponding values are not updated
|     in the original Series.

```



```

|
| >>> s = pd.Series([1, 2, 3])
| >>> s.update(pd.Series([4, np.nan, 6]))
| >>> s
| 0    4
| 1    2
| 2    6
| dtype: int64
|
| ``other`` can also be a non-Series object type
| that is coercible into a Series
|
| >>> s = pd.Series([1, 2, 3])
| >>> s.update([4, np.nan, 6])
| >>> s
| 0    4
| 1    2
| 2    6
| dtype: int64
|
| >>> s = pd.Series([1, 2, 3])
| >>> s.update({1: 9})
| >>> s
| 0    1
| 1    9
| 2    3
| dtype: int64
|
| var(self, axis: 'Axis | None' = None, skipna: 'bool_t' = True, ddof: 'int' = 1, numeric_
|   Return unbiased variance over requested axis.
|
|   Normalized by N-1 by default. This can be changed using the ddof argument.
|
|   Parameters
|   -----
|   axis : {index (0)}
|       For `Series` this parameter is unused and defaults to 0.
|   skipna : bool, default True
|       Exclude NA/null values. If an entire row/column is NA, the result
|       will be NA.
|   ddof : int, default 1
|       Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
|       where N represents the number of elements.

```

numeric_only : bool, default False
Include only float, int, boolean columns. Not implemented for Series.

Returns

scalar or Series (if level specified)

Examples

>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
... 'age': [21, 25, 62, 43],
... 'height': [1.61, 1.87, 1.49, 2.01]}
...).set_index('person_id')

>>> df

	age	height
person_id		
0	21	1.61
1	25	1.87
2	62	1.49
3	43	2.01

>>> df.var()
age 352.916667
height 0.056367
dtype: float64

Alternatively, ``ddof=0`` can be set to normalize by N instead of N-1:

>>> df.var(ddof=0)
age 264.687500
height 0.042275
dtype: float64

view(self, dtype: 'Dtype | None' = None) -> 'Series'
Create a new view of the Series.

This function will return a new Series with a view of the same underlying values in memory, optionally reinterpreted with a new data type. The new data type must preserve the same size in bytes as to not cause index misalignment.

Parameters

dtype : data type
Data type object or one of their string representations.

Returns

Series

A new Series object as a view of the same data in memory.

See Also

numpy.ndarray.view : Equivalent numpy function to create a new view of the same data in memory.

Notes

Series are instantiated with ``dtype=float64`` by default. While ``numpy.ndarray.view()`` will return a view with the same data type as the original array, ``Series.view()`` (without specified dtype) will try using ``float64`` and may fail if the original data type size in bytes is not the same.

Examples

```
>>> s = pd.Series([-2, -1, 0, 1, 2], dtype='int8')
>>> s
0    -2
1    -1
2     0
3     1
4     2
dtype: int8
```

The 8 bit signed integer representation of `-1` is `0b11111111`, but the same bytes represent 255 if read as an 8 bit unsigned integer:

```
>>> us = s.view('uint8')
>>> us
0    254
1    255
2     0
3     1
4     2
dtype: uint8
```

The views share the same underlying values:

```
>>> us[0] = 128
>>> s
0   -128
1     -1
2      0
3      1
4      2
dtype: int8
```

```
where(self, cond, other=<no_default>, *, inplace: 'bool' = False, axis: 'Axis | None' = None)
    Replace values where the condition is False.
```

Parameters

```
cond : bool Series/DataFrame, array-like, or callable
    Where `cond` is True, keep the original value. Where
    False, 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 False 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).
    If not specified, entries will be filled with the corresponding
    NULL value (`np.nan` for numpy dtypes, `pd.NA` for extension
    dtypes).
inplace : bool, default False
    Whether to perform the operation in place on the data.
axis : int, default None
    Alignment axis if needed. For `Series` this parameter is
    unused and defaults to 0.
level : int, default None
    Alignment level if needed.
```

Returns

```
Same type as caller or None if ``inplace=True``.
```

See Also

:func:`DataFrame.mask` : Return an object of same shape as
self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if ``cond`` is ``True`` the element is used; otherwise the corresponding element from the DataFrame ``other`` is used. If the axis of ``other`` does not align with axis of ``cond`` Series/DataFrame, the misaligned index positions will be filled with False.

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

For further details and examples see the ``where`` documentation in :ref:`indexing <indexing.where_mask>`.

The dtype of the object takes precedence. The fill value is casted to the object's dtype, if this can be done losslessly.

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 = pd.Series(range(5))
|     >>> t = pd.Series([True, False])
|     >>> s.where(t, 99)
|     0     0
|     1    99
|     2    99
|     3    99
|     4    99
|     dtype: int64
|     >>> s.mask(t, 99)
|     0    99
|     1     1
|     2    99
|     3    99
|     4    99
|     dtype: int64
|
|     >>> s.where(s > 1, 10)
|     0    10
|     1    10
|     2     2
|     3     3
|     4     4
|     dtype: int64
|     >>> s.mask(s > 1, 10)
|     0     0
|     1     1
|     2    10
|     3    10
|     4    10
|     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

```

Readonly properties defined here:

`array`

The ExtensionArray of the data backing this Series or Index.

Returns

ExtensionArray

An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around `:class:`numpy.ndarray``.

```.array``` differs ```.values``` which may require converting the data to a different form.

See Also

-----

`Index.to_numpy` : Similar method that always returns a NumPy array.

`Series.to_numpy` : Similar method that always returns a NumPy array.

Notes

-----

This table lays out the different array types for each extension dtype within pandas.

dtype	array type
category	Categorical
period	PeriodArray
interval	IntervalArray
IntegerNA	IntegerArray
string	StringArray
boolean	BooleanArray
datetime64[ns, tz]	DatetimeArray

For any 3rd-party extension types, the array type will be an ExtensionArray.

For all remaining dtypes ``.array`` will be a :class:`arrays.NumpyExtensionArray` wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use :meth:`Series.to\_numpy` instead.

#### Examples

-----

For regular NumPy types like int, and float, a PandasArray is returned.

```
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
['a', 'b', 'a']
Categories (2, object): ['a', 'b']
```

#### axes

Return a list of the row axis labels.



```

| dtype
| Return the dtype object of the underlying data.
|
| Examples
| -----
| >>> s = pd.Series([1, 2, 3])
| >>> s.dtype
| dtype('int64')
|
| dtypes
| Return the dtype object of the underlying data.
|
| Examples
| -----
| >>> s = pd.Series([1, 2, 3])
| >>> s.dtypes
| dtype('int64')
|
| hasnans
| Return True if there are any NaNs.
|
| Enables various performance speedups.
|
| Returns
| -----
| bool
|
| values
| Return Series as ndarray or ndarray-like depending on the dtype.
|
| .. warning::
|
| We recommend using :attr:`Series.array` or
| :meth:`Series.to_numpy`, depending on whether you need
| a reference to the underlying data or a NumPy array.
|
| Returns
| -----
| numpy.ndarray or ndarray-like
|
| See Also
| -----

```

Series.array : Reference to the underlying data.  
Series.to\_numpy : A NumPy array representing the underlying data.

#### Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
['a', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
... tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
 '2013-01-02T05:00:00.000000000',
 '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

---

Data descriptors defined here:

#### index

The index (axis labels) of the Series.

#### name

Return the name of the Series.

The name of a Series becomes its index or column name if it is used to form a DataFrame. It is also used whenever displaying the Series using the interpreter.

#### Returns

label (hashable object)

The name of the Series, also the column name if part of a DataFrame.

#### See Also

Series.rename : Sets the Series name when given a scalar input.  
Index.name : Corresponding Index property.

### Examples

The Series name can be set initially when calling the constructor.

```
>>> s = pd.Series([1, 2, 3], dtype=np.int64, name='Numbers')
```

```
>>> s
```

```
0 1
```

```
1 2
```

```
2 3
```

```
Name: Numbers, dtype: int64
```

```
>>> s.name = "Integers"
```

```
>>> s
```

```
0 1
```

```
1 2
```

```
2 3
```

```
Name: Integers, dtype: int64
```

The name of a Series within a DataFrame is its column name.

```
>>> df = pd.DataFrame([[1, 2], [3, 4], [5, 6]],
... columns=["Odd Numbers", "Even Numbers"])
```

```
>>> df
```

```
 Odd Numbers Even Numbers
```

```
0 1 2
```

```
1 3 4
```

```
2 5 6
```

```
>>> df["Even Numbers"].name
```

```
'Even Numbers'
```

---

Data and other attributes defined here:

```
__annotations__ = {'_AXIS_ORDERS': "list[Literal['index', 'columns']]"}...
```

```
cat = <class 'pandas.core.arrays.categorical.CategoricalAccessor'>
Accessor object for categorical properties of the Series values.
```

### Parameters

data : Series or CategoricalIndex

## Examples

-----

```
>>> s = pd.Series(list("abbccc")).astype("category")
```

```
>>> s
```

```
0 a
1 b
2 b
3 c
4 c
5 c
```

```
dtype: category
```

```
Categories (3, object): ['a', 'b', 'c']
```

```
>>> s.cat.categories
```

```
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> s.cat.rename_categories(list("cba"))
```

```
0 c
1 b
2 b
3 a
4 a
5 a
```

```
dtype: category
```

```
Categories (3, object): ['c', 'b', 'a']
```

```
>>> s.cat.reorder_categories(list("cba"))
```

```
0 a
1 b
2 b
3 c
4 c
5 c
```

```
dtype: category
```

```
Categories (3, object): ['c', 'b', 'a']
```

```
>>> s.cat.add_categories(["d", "e"])
```

```
0 a
1 b
2 b
3 c
4 c
```

```

| 5 c
| dtype: category
| Categories (5, object): ['a', 'b', 'c', 'd', 'e']
|
| >>> s.cat.remove_categories(["a", "c"])
| 0 NaN
| 1 b
| 2 b
| 3 NaN
| 4 NaN
| 5 NaN
| dtype: category
| Categories (1, object): ['b']
|
| >>> s1 = s.cat.add_categories(["d", "e"])
| >>> s1.cat.remove_unused_categories()
| 0 a
| 1 b
| 2 b
| 3 c
| 4 c
| 5 c
| dtype: category
| Categories (3, object): ['a', 'b', 'c']
|
| >>> s.cat.set_categories(list("abcde"))
| 0 a
| 1 b
| 2 b
| 3 c
| 4 c
| 5 c
| dtype: category
| Categories (5, object): ['a', 'b', 'c', 'd', 'e']
|
| >>> s.cat.as_ordered()
| 0 a
| 1 b
| 2 b
| 3 c
| 4 c
| 5 c
| dtype: category

```

```

| Categories (3, object): ['a' < 'b' < 'c']
|
| >>> s.cat.as_unordered()
| 0 a
| 1 b
| 2 b
| 3 c
| 4 c
| 5 c
| dtype: category
| Categories (3, object): ['a', 'b', 'c']
|
|
| dt = <class 'pandas.core.indexes.accessors.CombinedDatetimelikePropert...
|
| plot = <class 'pandas.plotting._core.PlotAccessor'>
| Make plots of Series or DataFrame.
|
| Uses the backend specified by the
| option ``plotting.backend``. By default, matplotlib is used.
|
| Parameters
| -----
| data : Series or DataFrame
| The object for which the method is called.
| x : label or position, default None
| Only used if data is a DataFrame.
| y : label, position or list of label, positions, default None
| Allows plotting of one column versus another. Only used if data is a
| DataFrame.
| kind : str
| The kind of plot to produce:
|
| - 'line' : line plot (default)
| - 'bar' : vertical bar plot
| - 'barh' : horizontal bar plot
| - 'hist' : histogram
| - 'box' : boxplot
| - 'kde' : Kernel Density Estimation plot
| - 'density' : same as 'kde'
| - 'area' : area plot
| - 'pie' : pie plot
| - 'scatter' : scatter plot (DataFrame only)

```

```

| - 'hexbin' : hexbin plot (DataFrame only)
| ax : matplotlib axes object, default None
| An axes of the current figure.
| subplots : bool or sequence of iterables, default False
| Whether to group columns into subplots:
|
| - ``False`` : No subplots will be used
| - ``True`` : Make separate subplots for each column.
| - sequence of iterables of column labels: Create a subplot for each
| group of columns. For example ``(['a', 'c'), ('b', 'd')]`` will
| create 2 subplots: one with columns 'a' and 'c', and one
| with columns 'b' and 'd'. Remaining columns that aren't specified
| will be plotted in additional subplots (one per column).
|
| .. versionadded:: 1.5.0
|
| sharex : bool, default True if ax is None else False
| In case ``subplots=True``, share x axis and set some x axis labels
| to invisible; defaults to True if ax is None otherwise False if
| an ax is passed in; Be aware, that passing in both an ax and
| ``sharex=True`` will alter all x axis labels for all axis in a figure.
| sharey : bool, default False
| In case ``subplots=True``, share y axis and set some y axis labels to invisible.
| layout : tuple, optional
| (rows, columns) for the layout of subplots.
| figsize : a tuple (width, height) in inches
| Size of a figure object.
| use_index : bool, default True
| Use index as ticks for x axis.
| title : str or list
| Title to use for the plot. If a string is passed, print the string
| at the top of the figure. If a list is passed and ``subplots`` is
| True, print each item in the list above the corresponding subplot.
| grid : bool, default None (matlab style default)
| Axis grid lines.
| legend : bool or {'reverse'}
| Place legend on axis subplots.
| style : list or dict
| The matplotlib line style per column.
| logx : bool or 'sym', default False
| Use log scaling or symlog scaling on x axis.
|
| logy : bool or 'sym' default False

```

```

| Use log scaling or symlog scaling on y axis.
|
| loglog : bool or 'sym', default False
| Use log scaling or symlog scaling on both x and y axes.
|
| xticks : sequence
| Values to use for the xticks.
| yticks : sequence
| Values to use for the yticks.
| xlim : 2-tuple/list
| Set the x limits of the current axes.
| ylim : 2-tuple/list
| Set the y limits of the current axes.
| xlabel : label, optional
| Name to use for the xlabel on x-axis. Default uses index name as xlabel, or the
| x-column name for planar plots.
|
| .. versionadded:: 1.1.0
|
| .. versionchanged:: 1.2.0
|
| Now applicable to planar plots (`scatter`, `hexbin`).
|
| .. versionchanged:: 2.0.0
|
| Now applicable to histograms.
|
| ylabel : label, optional
| Name to use for the ylabel on y-axis. Default will show no ylabel, or the
| y-column name for planar plots.
|
| .. versionadded:: 1.1.0
|
| .. versionchanged:: 1.2.0
|
| Now applicable to planar plots (`scatter`, `hexbin`).
|
| .. versionchanged:: 2.0.0
|
| Now applicable to histograms.
|
| rot : float, default None
| Rotation for ticks (xticks for vertical, yticks for horizontal

```



```

plots).
fontsize : float, default None
 Font size for xticks and yticks.
colormap : str or matplotlib colormap object, default None
 Colormap to select colors from. If string, load colormap with that
 name from matplotlib.
colorbar : bool, optional
 If True, plot colorbar (only relevant for 'scatter' and 'hexbin'
 plots).
position : float
 Specify relative alignments for bar plot layout.
 From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5
 (center).
table : bool, Series or DataFrame, default False
 If True, draw a table using the data in the DataFrame and the data
 will be transposed to meet matplotlib's default layout.
 If a Series or DataFrame is passed, use passed data to draw a
 table.
yerr : DataFrame, Series, array-like, dict and str
 See :ref:`Plotting with Error Bars <visualization.errorbars>` for
 detail.
xerr : DataFrame, Series, array-like, dict and str
 Equivalent to yerr.
stacked : bool, default False in line and bar plots, and True in area plot
 If True, create stacked plot.
secondary_y : bool or sequence, default False
 Whether to plot on the secondary y-axis if a list/tuple, which
 columns to plot on secondary y-axis.
mark_right : bool, default True
 When using a secondary_y axis, automatically mark the column
 labels with "(right)" in the legend.
include_bool : bool, default is False
 If True, boolean values can be plotted.
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``.
**kwargs
 Options to pass to matplotlib plotting method.

```

Returns

-----

:class:`matplotlib.axes.Axes` or `numpy.ndarray` of them  
If the backend is not the default matplotlib one, the return value will be the object returned by the backend.

#### Notes

-----

- See matplotlib documentation online for more on this subject
- If `kind` = 'bar' or 'barh', you can specify relative alignments for bar plot layout by `position` keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

`sparse = <class 'pandas.core.arrays.sparse.accessor.SparseAccessor'>`  
Accessor for SparseSparse from other sparse matrix data types.

`str = <class 'pandas.core.strings.accessor.StringMethods'>`  
Vectorized string functions for Series and Index.

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

#### Examples

-----

```
>>> s = pd.Series(["A_Str_Series"])
>>> s
0 A_Str_Series
dtype: object
```

```
>>> s.str.split("_")
0 [A, Str, Series]
dtype: object
```

```
>>> s.str.replace("_", "")
0 AStrSeries
dtype: object
```

-----  
Methods inherited from `pandas.core.base.IndexOpsMixin`:

```

| __iter__(self) -> 'Iterator'
| Return an iterator of the values.
|
| These are each a scalar type, which is a Python scalar
| (for str, int, float) or a pandas scalar
| (for Timestamp/Timedelta/Interval/Period)
|
| Returns
| -----
| iterator
|
| argmax(self, axis: 'AxisInt | None' = None, skipna: 'bool' = True, *args, **kwargs) -> 'int'
| Return int position of the largest value in the Series.
|
| If the maximum is achieved in multiple locations,
| the first row position is returned.
|
| Parameters
| -----
| axis : {None}
| Unused. Parameter needed for compatibility with DataFrame.
| skipna : bool, default True
| Exclude NA/null values when showing the result.
| *args, **kwargs
| Additional arguments and keywords for compatibility with NumPy.
|
| Returns
| -----
| int
| Row position of the maximum value.
|
| See Also
| -----
| Series.argmax : Return position of the maximum value.
| Series.argmin : Return position of the minimum value.
| numpy.ndarray.argmax : Equivalent method for numpy arrays.
| Series.idxmax : Return index label of the maximum values.
| Series.idxmin : Return index label of the minimum values.
|
| Examples
| -----
| Consider dataset containing cereal calories

```

```

| >>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
| ... 'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
| >>> s
| Corn Flakes 100.0
| Almond Delight 110.0
| Cinnamon Toast Crunch 120.0
| Cocoa Puff 110.0
| dtype: float64
|
| >>> s.argmax()
| 2
| >>> s.argmin()
| 0

```

The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

```

| argmin(self, axis: 'AxisInt | None' = None, skipna: 'bool' = True, *args, **kwargs) -> 'int'
| Return int position of the smallest value in the Series.

```

If the minimum is achieved in multiple locations, the first row position is returned.

#### Parameters

```

| -----
| axis : {None}
| Unused. Parameter needed for compatibility with DataFrame.
| skipna : bool, default True
| Exclude NA/null values when showing the result.
| *args, **kwargs
| Additional arguments and keywords for compatibility with NumPy.

```

#### Returns

```

| -----
| int
| Row position of the minimum value.

```

#### See Also

```

| -----
| Series.argmin : Return position of the minimum value.
| Series.argmax : Return position of the maximum value.
| numpy.ndarray.argmin : Equivalent method for numpy arrays.

```

Series.idxmax : Return index label of the maximum values.  
Series.idxmin : Return index label of the minimum values.

#### Examples

-----  
Consider dataset containing cereal calories

```
>>> s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
... 'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes 100.0
Almond Delight 110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff 110.0
dtype: float64

>>> s.argmax()
2
>>> s.argmin()
0
```

The maximum cereal calories is the third element and the minimum cereal calories is the first element, since series is zero-indexed.

factorize(self, sort: 'bool' = False, use\_na\_sentinel: 'bool' = True) -> 'tuple[npt.NDArray, ...]  
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 :func:`pandas.factorize`, and as a method :meth:`Series.factorize` and :meth:`Index.factorize`.

#### Parameters

-----  
sort : bool, default False

Sort `uniques` and shuffle `codes` to maintain the relationship.

use\_na\_sentinel : bool, default True

If True, the sentinel -1 will be used for NaN values. If False, NaN values will be encoded as non-negative integers and will not drop the NaN from the uniques of the values.

.. versionadded:: 1.5.0

#### 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.

#### Notes

Reference :ref:`the user guide <reshaping.factorize>` for more examples.

#### Examples

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

When `use_na_sentinel=True` (the default), missing values are indicated in the `codes` with the sentinel value `-1` and missing values are not 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')
```

If NaN is in the values, and we want to include NaN in the uniques of the values, it can be achieved by setting `use_na_sentinel=False`.

```

| >>> values = np.array([1, 2, 1, np.nan])
| >>> codes, uniques = pd.factorize(values) # default: use_na_sentinel=True
| >>> codes
| array([0, 1, 0, -1])
| >>> uniques
| array([1., 2.])
|
| >>> codes, uniques = pd.factorize(values, use_na_sentinel=False)
| >>> codes
| array([0, 1, 0, 2])
| >>> uniques
| array([1., 2., nan])
|
| 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.
|
| nunique(self, dropna: 'bool' = True) -> 'int'
| Return number of unique elements in the object.
|
| Excludes NA values by default.
|
| Parameters
| -----
| dropna : bool, default True
| Don't include NaN in the count.
|
| Returns
| -----
| int
|
| See Also
| -----
| DataFrame.nunique: Method nunique for DataFrame.

```



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

#### Examples

```
>>> s = pd.Series([1, 3, 5, 7, 7])
```

```
>>> s
```

```
0 1
```

```
1 3
```

```
2 5
```

```
3 7
```

```
4 7
```

```
dtype: int64
```

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

```
4
```

```
to_list = tolist(self)
```

```
to_numpy(self, dtype: 'npt.DTypeLike | None' = None, copy: 'bool' = False, na_value: 'ob
```

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

#### Parameters

dtype : str or numpy.dtype, optional

The dtype to pass to :meth:`numpy.asarray`.

copy : bool, default False

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

na\_value : Any, optional

The value to use for missing values. The default value depends on `dtype` and the type of the array.

\*\*kwargs

Additional keywords passed through to the ``to\_numpy`` method of the underlying array (for extension arrays).

#### Returns

numpy.ndarray

#### See Also

Series.array : Get the actual data stored within.  
Index.array : Get the actual data stored within.  
DataFrame.to\_numpy : Similar method for DataFrame.

#### Notes

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

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

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

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

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

#### Examples

-----  
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))  
>>> ser.to\_numpy()  
array(['a', 'b', 'a'], dtype=object)

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

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

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

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

`tolist(self)`

Return a list of the values.

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

Returns

-----

list

See Also

-----

`numpy.ndarray.tolist` : Return the array as an `a.ndim`-levels deep nested list of Python scalars.

`transpose(self: '_T', *args, **kwargs) -> '_T'`

Return the transpose, which is by definition self.

Returns

-----

`%(klass)s`

`value_counts(self, normalize: 'bool' = False, sort: 'bool' = True, ascending: 'bool' = F`

Return a Series containing counts of unique values.

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

#### Parameters

-----  
normalize : bool, default False  
    If True then the object returned will contain the relative frequencies of the unique values.  
sort : bool, default True  
    Sort by frequencies.  
ascending : bool, default False  
    Sort in ascending order.  
bins : int, optional  
    Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data.  
dropna : bool, default True  
    Don't include counts of NaN.

#### Returns

-----  
Series

#### See Also

-----  
Series.count: Number of non-NA elements in a Series.  
DataFrame.count: Number of non-NA elements in a DataFrame.  
DataFrame.value\_counts: Equivalent method on DataFrames.

#### Examples

-----  
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])  
>>> index.value\_counts()  
3.0 2  
1.0 1  
2.0 1  
4.0 1  
Name: count, dtype: int64

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

```

|
| >>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
| >>> s.value_counts(normalize=True)
| 3.0 0.4
| 1.0 0.2
| 2.0 0.2
| 4.0 0.2
| Name: proportion, dtype: float64
|
| **bins**
|
| Bins can be useful for going from a continuous variable to a
| categorical variable; instead of counting unique
| apparitions of values, divide the index in the specified
| number of half-open bins.
|
| >>> s.value_counts(bins=3)
| (0.996, 2.0] 2
| (2.0, 3.0] 2
| (3.0, 4.0] 1
| Name: count, dtype: int64
|
| **dropna**
|
| With `dropna` set to `False` we can also see NaN index values.
|
| >>> s.value_counts(dropna=False)
| 3.0 2
| 1.0 1
| 2.0 1
| 4.0 1
| NaN 1
| Name: count, dtype: int64
|
| -----
| Readonly properties inherited from pandas.core.base.IndexOpsMixin:
|
| T
| Return the transpose, which is by definition self.
|
| empty
|
| is_monotonic_decreasing

```

```

| Return boolean if values in the object are monotonically decreasing.
|
| Returns
| -----
| bool
|
| is_monotonic_increasing
| Return boolean if values in the object are monotonically increasing.
|
| Returns
| -----
| bool
|
| is_unique
| Return boolean if values in the object are unique.
|
| Returns
| -----
| bool
|
| nbytes
| Return the number of bytes in the underlying data.
|
| ndim
| Number of dimensions of the underlying data, by definition 1.
|
| shape
| Return a tuple of the shape of the underlying data.
|
| Examples
| -----
| >>> s = pd.Series([1, 2, 3])
| >>> s.shape
| (3,)
|
| size
| Return the number of elements in the underlying data.
|
| -----
| Data and other attributes inherited from pandas.core.base.IndexOpsMixin:
|
| __array_priority__ = 1000
|

```

```

Methods inherited from pandas.core.arraylike.OpsMixin:

__add__(self, other)
 Get Addition of DataFrame and other, column-wise.

 Equivalent to ``DataFrame.add(other)``.

Parameters

other : scalar, sequence, Series, dict or DataFrame
 Object to be added to the DataFrame.

Returns

DataFrame
 The result of adding ``other`` to DataFrame.

See Also

DataFrame.add : Add a DataFrame and another object, with option for index-
 or column-oriented addition.

Examples

>>> df = pd.DataFrame({'height': [1.5, 2.6], 'weight': [500, 800]},
... index=['elk', 'moose'])
>>> df
 height weight
elk 1.5 500
moose 2.6 800

Adding a scalar affects all rows and columns.

>>> df[['height', 'weight']] + 1.5
 height weight
elk 3.0 501.5
moose 4.1 801.5

Each element of a list is added to a column of the DataFrame, in order.

>>> df[['height', 'weight']] + [0.5, 1.5]
 height weight

```

```

| elk 2.0 501.5
| moose 3.1 801.5
|

```

Keys of a dictionary are aligned to the DataFrame, based on column names; each value in the dictionary is added to the corresponding column.

```

| >>> df[['height', 'weight']] + {'height': 0.5, 'weight': 1.5}
| height weight
| elk 2.0 501.5
| moose 3.1 801.5
|

```

When `other` is a `:class:`Series``, the index of `other` is aligned with the columns of the DataFrame.

```

| >>> s1 = pd.Series([0.5, 1.5], index=['weight', 'height'])
| >>> df[['height', 'weight']] + s1
| height weight
| elk 3.0 500.5
| moose 4.1 800.5
|

```

Even when the index of `other` is the same as the index of the DataFrame, the `:class:`Series`` will not be reoriented. If index-wise alignment is desired, `:meth:`DataFrame.add`` should be used with ``axis='index'``.

```

| >>> s2 = pd.Series([0.5, 1.5], index=['elk', 'moose'])
| >>> df[['height', 'weight']] + s2
| elk height moose weight
| elk NaN NaN NaN NaN
| moose NaN NaN NaN NaN
|

```

```

| >>> df[['height', 'weight']].add(s2, axis='index')
| height weight
| elk 2.0 500.5
| moose 4.1 801.5
|

```

When `other` is a `:class:`DataFrame``, both columns names and the index are aligned.

```

| >>> other = pd.DataFrame({'height': [0.2, 0.4, 0.6]},
| ... index=['elk', 'moose', 'deer'])
| >>> df[['height', 'weight']] + other
| height weight
| deer NaN NaN
|

```



```

| elk 1.7 NaN
| moose 3.0 NaN
|
| __and__(self, other)
|
| __divmod__(self, other)
|
| __eq__(self, other)
| Return self==value.
|
| __floordiv__(self, other)
|
| __ge__(self, other)
| Return self>=value.
|
| __gt__(self, other)
| Return self>value.
|
| __le__(self, other)
| Return self<=value.
|
| __lt__(self, other)
| Return self<value.
|
| __mod__(self, other)
|
| __mul__(self, other)
|
| __ne__(self, other)
| Return self!=value.
|
| __or__(self, other)
|
| __pow__(self, other)
|
| __radd__(self, other)
|
| __rand__(self, other)
|
| __rdivmod__(self, other)
|
| __rfloordiv__(self, other)
|

```

```

| __rmod__(self, other)
|
| __rmul__(self, other)
|
| __ror__(self, other)
|
| __rpow__(self, other)
|
| __rsub__(self, other)
|
| __rtruediv__(self, other)
|
| __rxor__(self, other)
|
| __sub__(self, other)
|
| __truediv__(self, other)
|
| __xor__(self, other)
|
|-----
| Data descriptors inherited from pandas.core.arraylike.OpsMixin:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)
|
|-----
| Data and other attributes inherited from pandas.core.arraylike.OpsMixin:
|
| __hash__ = None
|
|-----
| Methods inherited from pandas.core.generic.NDFrame:
|
| __abs__(self: 'NDFrameT') -> 'NDFrameT'
|
| __array_ufunc__(self, ufunc: 'np.ufunc', method: 'str', *inputs: 'Any', **kwargs: 'Any')
|
| __bool__ = __nonzero__(self) -> 'NoReturn'
|

```

```

| __contains__(self, key) -> 'bool_t'
| True if the key is in the info axis
|
| __copy__(self: 'NDFrameT', deep: 'bool_t' = True) -> 'NDFrameT'
|
| __deepcopy__(self: 'NDFrameT', memo=None) -> 'NDFrameT'
| Parameters
| -----
| memo, default None
| Standard signature. Unused
|
| __delitem__(self, key) -> 'None'
| Delete item
|
| __finalize__(self: 'NDFrameT', other, method: 'str | None' = None, **kwargs) -> 'NDFrameT'
| Propagate metadata from other to self.
|
| Parameters
| -----
| other : the object from which to get the attributes that we are going
| to propagate
| method : str, optional
| A passed method name providing context on where ``__finalize__``
| was called.
|
| .. warning::
|
| The value passed as `method` are not currently considered
| stable across pandas releases.
|
| __getattr__(self, name: 'str')
| After regular attribute access, try looking up the name
| This allows simpler access to columns for interactive use.
|
| __getstate__(self) -> 'dict[str, Any]'
|
| __iadd__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __iand__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __ifloordiv__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __imod__(self: 'NDFrameT', other) -> 'NDFrameT'

```

```

|
| __imul__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __invert__(self: 'NDFrameT') -> 'NDFrameT'
|
| __ior__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __ipow__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __isub__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __itruediv__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __ixor__(self: 'NDFrameT', other) -> 'NDFrameT'
|
| __neg__(self: 'NDFrameT') -> 'NDFrameT'
|
| __nonzero__(self) -> 'NoReturn'
|
| __pos__(self: 'NDFrameT') -> 'NDFrameT'
|
| __round__(self: 'NDFrameT', decimals: 'int' = 0) -> 'NDFrameT'
|
| __setattr__(self, name: 'str', value) -> 'None'
| After regular attribute access, try setting the name
| This allows simpler access to columns for interactive use.
|
| __setstate__(self, state) -> 'None'
|
| abs(self: 'NDFrameT') -> 'NDFrameT'
| Return a Series/DataFrame with absolute numeric value of each element.
|
| This function only applies to elements that are all numeric.
|
| Returns
| -----
| abs
| Series/DataFrame containing the absolute value of each element.
|
| See Also
| -----
| numpy.absolute : Calculate the absolute value element-wise.
|

```

## Notes

-----

For ``complex`` inputs, ``1.2 + 1j``, the absolute value is  
:math:`\sqrt{a^2 + b^2}`.

## Examples

-----

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0 1.10
1 2.00
2 3.33
3 4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0 1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0 1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from  
`StackOverflow <<https://stackoverflow.com/a/17758115>>`\_\_).

```
>>> df = pd.DataFrame({
... 'a': [4, 5, 6, 7],
... 'b': [10, 20, 30, 40],
... 'c': [100, 50, -30, -50]
... })
>>> df
 a b c
0 4 10 100
1 5 20 50
```

```

| 2 6 30 -30
| 3 7 40 -50
| >>> df.loc[(df.c - 43).abs().argsort()]
| a b c
| 1 5 20 50
| 0 4 10 100
| 2 6 30 -30
| 3 7 40 -50
|
| add_prefix(self: 'NDFrameT', prefix: 'str', axis: 'Axis | None' = None) -> 'NDFrameT'
| Prefix labels with string `prefix`.
|
| For Series, the row labels are prefixed.
| For DataFrame, the column labels are prefixed.
|
| Parameters
| -----
| prefix : str
| The string to add before each label.
| axis : {{0 or 'index', 1 or 'columns', None}}, default None
| Axis to add prefix on
|
| .. versionadded:: 2.0.0
|
| Returns
| -----
| Series or DataFrame
| New Series or DataFrame with updated labels.
|
| See Also
| -----
| Series.add_suffix: Suffix row labels with string `suffix`.
| DataFrame.add_suffix: Suffix column labels with string `suffix`.
|
| Examples
| -----
| >>> s = pd.Series([1, 2, 3, 4])
| >>> s
| 0 1
| 1 2
| 2 3
| 3 4
| dtype: int64

```

```

|
|
| >>> s.add_prefix('item_')
| item_0 1
| item_1 2
| item_2 3
| item_3 4
| dtype: int64
|
|
| >>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
| >>> df
| A B
| 0 1 3
| 1 2 4
| 2 3 5
| 3 4 6
|
|
| >>> df.add_prefix('col_')
| col_A col_B
| 0 1 3
| 1 2 4
| 2 3 5
| 3 4 6
|
|
| add_suffix(self: 'NDFrameT', suffix: 'str', axis: 'Axis | None' = None) -> 'NDFrameT'
| Suffix labels with string `suffix`.
|
| For Series, the row labels are suffixed.
| For DataFrame, the column labels are suffixed.
|
| Parameters
| -----
| suffix : str
| The string to add after each label.
| axis : {{0 or 'index', 1 or 'columns', None}}, default None
| Axis to add suffix on
|
| .. versionadded:: 2.0.0
|
| Returns
| -----
| Series or DataFrame
| New Series or DataFrame with updated labels.
|
|

```

See Also

-----

Series.add\_prefix: Prefix row labels with string `prefix`.

DataFrame.add\_prefix: Prefix column labels with string `prefix`.

Examples

-----

```
>>> s = pd.Series([1, 2, 3, 4])
```

```
>>> s
```

```
0 1
```

```
1 2
```

```
2 3
```

```
3 4
```

```
dtype: int64
```

```
>>> s.add_suffix('_item')
```

```
0_item 1
```

```
1_item 2
```

```
2_item 3
```

```
3_item 4
```

```
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
```

```
>>> df
```

```
 A B
```

```
0 1 3
```

```
1 2 4
```

```
2 3 5
```

```
3 4 6
```

```
>>> df.add_suffix('_col')
```

```
 A_col B_col
```

```
0 1 3
```

```
1 2 4
```

```
2 3 5
```

```
3 4 6
```

```
asof(self, where, subset=None)
```

Return the last row(s) without any NaNs before `where`.

The last row (for each element in `where`, if list) without any NaN is taken.

In case of a `:class:`~pandas.DataFrame``, the last row without NaN



considering only the subset of columns (if not `None`)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

#### Parameters

-----  
where : date or array-like of dates

Date(s) before which the last row(s) are returned.

subset : str or array-like of str, default `None`

For DataFrame, if not `None`, only use these columns to check for NaNs.

#### Returns

-----  
scalar, Series, or DataFrame

The return can be:

- \* scalar : when `self` is a Series and `where` is a scalar
- \* Series: when `self` is a Series and `where` is an array-like, or when `self` is a DataFrame and `where` is a scalar
- \* DataFrame : when `self` is a DataFrame and `where` is an array-like

Return scalar, Series, or DataFrame.

#### See Also

-----  
merge\_asof : Perform an asof merge. Similar to left join.

#### Notes

-----  
Dates are assumed to be sorted. Raises if this is not the case.

#### Examples

-----  
A Series and a scalar `where`.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
```

```
>>> s
```

```
10 1.0
```

```
20 2.0
```

```
30 NaN
40 4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence `where`, a Series is returned. The first value is NaN, because the first element of `where` is before the first index value.

```
>>> s.asof([5, 20])
5 NaN
20 2.0
dtype: float64
```

Missing values are not considered. The following is ``2.0``, not NaN, even though NaN is at the index location for ``30``.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
... 'b': [None, None, None, None, 500]},
... index=pd.DatetimeIndex(['2018-02-27 09:01:00',
... '2018-02-27 09:02:00',
... '2018-02-27 09:03:00',
... '2018-02-27 09:04:00',
... '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
... '2018-02-27 09:04:30']))
 a b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
... '2018-02-27 09:04:30']),
... subset=['a'])
 a b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

```

| 2018-02-27 09:03:30 30 NaN
| 2018-02-27 09:04:30 40 NaN
|
| astype(self: 'NDFrameT', dtype, copy: 'bool_t | None' = None, errors: 'IgnoreRaise' = 'r
| Cast a pandas object to a specified dtype ``dtype``.
|
| Parameters
| -----
| dtype : str, data type, Series or Mapping of column name -> data type
| Use a str, numpy.dtype, pandas.ExtensionDtype or Python type to
| cast entire pandas object to the same type. Alternatively, use a
| mapping, e.g. {col: dtype, ...}, where col is a column label and dtype is
| a numpy.dtype or Python type to cast one or more of the DataFrame's
| columns to column-specific types.
| copy : bool, default True
| Return a copy when ``copy=True`` (be very careful setting
| ``copy=False`` as changes to values then may propagate to other
| pandas objects).
| errors : {'raise', 'ignore'}, default 'raise'
| Control raising of exceptions on invalid data for provided dtype.
|
| - ``raise`` : allow exceptions to be raised
| - ``ignore`` : suppress exceptions. On error return original object.
|
| Returns
| -----
| same type as caller
|
| See Also
| -----
| to_datetime : Convert argument to datetime.
| to_timedelta : Convert argument to timedelta.
| to_numeric : Convert argument to a numeric type.
| numpy.ndarray.astype : Cast a numpy array to a specified type.
|
| Notes
| -----
| .. versionchanged:: 2.0.0
|
| Using ``astype`` to convert from timezone-naive dtype to
| timezone-aware dtype will raise an exception.
| Use :meth:`Series.dt.tz_localize` instead.

```

## Examples

-----

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1 int64
col2 int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1 int32
col2 int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1 int32
col2 int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0 1
1 2
dtype: int32
>>> ser.astype('int64')
0 1
1 2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0 1
1 2
dtype: category
```

```

| Categories (2, int32): [1, 2]
|
| Convert to ordered categorical type with custom ordering:
|
| >>> from pandas.api.types import CategoricalDtype
| >>> cat_dtype = CategoricalDtype(
| ... categories=[2, 1], ordered=True)
| >>> ser.astype(cat_dtype)
| 0 1
| 1 2
| dtype: category
| Categories (2, int64): [2 < 1]
|
| Create a series of dates:
|
| >>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
| >>> ser_date
| 0 2020-01-01
| 1 2020-01-02
| 2 2020-01-03
| dtype: datetime64[ns]
|
| at_time(self: 'NDFrameT', time, asof: 'bool_t' = False, axis: 'Axis | None' = None) -> 'I
| Select values at particular time of day (e.g., 9:30AM).
|
| Parameters
| -----
| time : datetime.time or str
| The values to select.
| axis : {0 or 'index', 1 or 'columns'}, default 0
| For `Series` this parameter is unused and defaults to 0.
|
| Returns
| -----
| Series or DataFrame
|
| Raises
| -----
| TypeError
| If the index is not a :class:`DatetimeIndex`
|
| See Also
| -----

```

between\_time : Select values between particular times of the day.  
first : Select initial periods of time series based on a date offset.  
last : Select final periods of time series based on a date offset.  
DatetimeIndex.indexer\_at\_time : Get just the index locations for  
values at particular time of the day.

#### Examples

-----

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
```

```
 A
2018-04-09 00:00:00 1
2018-04-09 12:00:00 2
2018-04-10 00:00:00 3
2018-04-10 12:00:00 4
```

```
>>> ts.at_time('12:00')
```

```
 A
2018-04-09 12:00:00 2
2018-04-10 12:00:00 4
```

backfill(self: 'NDFrameT', \*, axis: 'None | Axis' = None, inplace: 'bool\_t' = False, limit: 'int' = None)  
Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.

.. deprecated:: 2.0

Series/DataFrame.backfill is deprecated. Use Series/DataFrame.bfill instead.

#### Returns

-----

Series/DataFrame or None

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

between\_time(self: 'NDFrameT', start\_time, end\_time, inclusive: 'IntervalClosedType' = 'both')  
Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting ``start\_time`` to be later than ``end\_time``,  
you can get the times that are *not* between the two times.

#### Parameters

-----

start\_time : datetime.time or str

```

Initial time as a time filter limit.
end_time : datetime.time or str
 End time as a time filter limit.
inclusive : {"both", "neither", "left", "right"}, default "both"
 Include boundaries; whether to set each bound as closed or open.
axis : {0 or 'index', 1 or 'columns'}, default 0
 Determine range time on index or columns value.
 For `Series` this parameter is unused and defaults to 0.

Returns

Series or DataFrame
 Data from the original object filtered to the specified dates range.

Raises

TypeError
 If the index is not a :class:`DatetimeIndex`

See Also

at_time : Select values at a particular time of the day.
first : Select initial periods of time series based on a date offset.
last : Select final periods of time series based on a date offset.
DatetimeIndex.indexer_between_time : Get just the index locations for
 values between particular times of the day.

Examples

>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
 A
2018-04-09 00:00:00 1
2018-04-10 00:20:00 2
2018-04-11 00:40:00 3
2018-04-12 01:00:00 4

>>> ts.between_time('0:15', '0:45')
 A
2018-04-10 00:20:00 2
2018-04-11 00:40:00 3

```

```

| You get the times that are *not* between two times by setting
| ``start_time`` later than ``end_time``:
|
| >>> ts.between_time('0:45', '0:15')
|
| A
| 2018-04-09 00:00:00 1
| 2018-04-12 01:00:00 4
|
| bool(self) -> 'bool_t'
| Return the bool of a single element Series or DataFrame.
|
| This must be a boolean scalar value, either True or False. It will raise a
| ValueError if the Series or DataFrame does not have exactly 1 element, or that
| element is not boolean (integer values 0 and 1 will also raise an exception).
|
| Returns
| -----
| bool
| The value in the Series or DataFrame.
|
| See Also
| -----
| Series.astype : Change the data type of a Series, including to boolean.
| DataFrame.astype : Change the data type of a DataFrame, including to boolean.
| numpy.bool_ : NumPy boolean data type, used by pandas for boolean values.
|
| Examples
| -----
| The method will only work for single element objects with a boolean value:
|
| >>> pd.Series([True]).bool()
| True
| >>> pd.Series([False]).bool()
| False
|
| >>> pd.DataFrame({'col': [True]}).bool()
| True
| >>> pd.DataFrame({'col': [False]}).bool()
| False
|
| convert_dtypes(self: 'NDFrameT', infer_objects: 'bool_t' = True, convert_string: 'bool_t'
| Convert columns to the best possible dtypes using dtypes supporting ``pd.NA``.
|

```



## Parameters

-----

`infer_objects` : bool, default True

Whether object dtypes should be converted to the best possible types.

`convert_string` : bool, default True

Whether object dtypes should be converted to `StringDtype()`.

`convert_integer` : bool, default True

Whether, if possible, conversion can be done to integer extension types.

`convert_boolean` : bool, defaults True

Whether object dtypes should be converted to `BooleanDtypes()`.

`convert_floating` : bool, defaults True

Whether, if possible, conversion can be done to floating extension types.

If `convert_integer` is also True, preference will be give to integer

dtypes if the floats can be faithfully casted to integers.

.. versionadded:: 1.2.0

`dtype_backend` : {"numpy\_nullable", "pyarrow"}, default "numpy\_nullable"

Which dtype\_backend to use, e.g. whether a DataFrame should use nullable dtypes for all dtypes that have a nullable

implementation when "numpy\_nullable" is set, pyarrow is used for all

dtypes if "pyarrow" is set.

The dtype\_backends are still experimental.

.. versionadded:: 2.0

## Returns

-----

Series or DataFrame

Copy of input object with new dtype.

## See Also

-----

`infer_objects` : Infer dtypes of objects.

`to_datetime` : Convert argument to datetime.

`to_timedelta` : Convert argument to timedelta.

`to_numeric` : Convert argument to a numeric type.

## Notes

-----

By default, `convert_dtypes` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support `pd.NA`. By using the options `convert_string`, `convert_integer`, `convert_boolean` and

``convert\_floating``, it is possible to turn off individual conversions to ``StringDtype``, the integer extension types, ``BooleanDtype`` or floating extension types, respectively.

For object-dtyped columns, if ``infer\_objects`` is ``True``, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to ``StringDtype``, ``BooleanDtype`` or an appropriate integer or floating extension type, otherwise leave as ``object``.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

.. versionchanged:: 1.2

Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support ``pd.NA``, the results of this method will change to support those new dtypes.

#### Examples

-----

```
>>> df = pd.DataFrame(
... {
... "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
... "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
... "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
... "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
... "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
... "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
... }
...)
```

Start with a DataFrame with default dtypes.

```
>>> df
 a b c d e f
0 1 x True h 10.0 NaN
1 2 y False i NaN 100.5
2 3 z NaN NaN 20.0 200.0
```

```

>>> df.dtypes
a int32
b object
c object
d object
e float64
f float64
dtype: object

```

Convert the DataFrame to use best possible dtypes.

```

>>> dfn = df.convert_dtypes()
>>> dfn
 a b c d e f
0 1 x True h 10 <NA>
1 2 y False i <NA> 100.5
2 3 z <NA> <NA> 20 200.0

```

```

>>> dfn.dtypes
a Int32
b string[python]
c boolean
d string[python]
e Int64
f Float64
dtype: object

```

Start with a Series of strings and missing data represented by ``np.nan``.

```

>>> s = pd.Series(["a", "b", np.nan])
>>> s
0 a
1 b
2 NaN
dtype: object

```

Obtain a Series with dtype ``StringDtype``.

```

>>> s.convert_dtypes()
0 a
1 b
2 <NA>
dtype: string

```

```

| copy(self: 'NDFrameT', deep: 'bool_t | None' = True) -> 'NDFrameT'
| Make a copy of this object's indices and data.
|
| When ``deep=True`` (default), a new object will be created with a
| copy of the calling object's data and indices. Modifications to
| the data or indices of the copy will not be reflected in the
| original object (see notes below).
|
| When ``deep=False``, a new object will be created without copying
| the calling object's data or index (only references to the data
| and index are copied). Any changes to the data of the original
| will be reflected in the shallow copy (and vice versa).
|
| Parameters
| -----
| deep : bool, default True
| Make a deep copy, including a copy of the data and the indices.
| With ``deep=False`` neither the indices nor the data are copied.
|
| Returns
| -----
| Series or DataFrame
| Object type matches caller.
|
| Notes
| -----
| When ``deep=True``, data is copied but actual Python objects
| will not be copied recursively, only the reference to the object.
| This is in contrast to copy.deepcopy in the Standard Library,
| which recursively copies object data (see examples below).
|
| While Index objects are copied when deep=True, the underlying
| numpy array is not copied for performance reasons. Since Index is
| immutable, the underlying data can be safely shared and a copy
| is not needed.
|
| Since pandas is not thread safe, see the
| :ref:`gotchas <gotchas.thread-safety>` when copying in a threading
| environment.
|
| Examples
| -----

```

```

| >>> s = pd.Series([1, 2], index=["a", "b"])
| >>> s
| a 1
| b 2
| dtype: int64
|
| >>> s_copy = s.copy()
| >>> s_copy
| a 1
| b 2
| dtype: int64
|
| **Shallow copy versus default (deep) copy:**
|
| >>> s = pd.Series([1, 2], index=["a", "b"])
| >>> deep = s.copy()
| >>> shallow = s.copy(deep=False)
|
| Shallow copy shares data and index with original.
|
| >>> s is shallow
| False
| >>> s.values is shallow.values and s.index is shallow.index
| True
|
| Deep copy has own copy of data and index.
|
| >>> s is deep
| False
| >>> s.values is deep.values or s.index is deep.index
| False
|
| Updates to the data shared by shallow copy and original is reflected
| in both; deep copy remains unchanged.
|
| >>> s[0] = 3
| >>> shallow[1] = 4
| >>> s
| a 3
| b 4
| dtype: int64
| >>> shallow
| a 3

```

```
| b 4
| dtype: int64
| >>> deep
| a 1
| b 2
| dtype: int64
|
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
| >>> s = pd.Series([[1, 2], [3, 4]])
| >>> deep = s.copy()
| >>> s[0][0] = 10
| >>> s
| 0 [10, 2]
| 1 [3, 4]
| dtype: object
| >>> deep
| 0 [10, 2]
| 1 [3, 4]
| dtype: object
|
```

```
| describe(self: 'NDFrameT', percentiles=None, include=None, exclude=None) -> 'NDFrameT'
| Generate descriptive statistics.
|
```

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding ``NaN`` values.

Analyzes both numeric and object series, as well as ``DataFrame`` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

-----

percentiles : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is ``[.25, .5, .75]``, which returns the 25th, 50th, and 75th percentiles.

include : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for ``Series``. Here are the options:

- 'all' : All columns of the input will be included in the output.
  - A list-like of dtypes : Limits the results to the provided data types.  
To limit the result to numeric types submit ``numpy.number``. To limit it instead to object columns submit the ``numpy.object`` data type. Strings can also be used in the style of ``select\_dtypes`` (e.g. ``df.describe(include=['O'])``). To select pandas categorical columns, use ``'category'``
  - None (default) : The result will include all numeric columns.
- exclude : list-like of dtypes or None (default), optional,  
A black list of data types to omit from the result. Ignored for ``Series``. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit ``numpy.number``. To exclude object columns submit the data type ``numpy.object``. Strings can also be used in the style of ``select\_dtypes`` (e.g. ``df.describe(exclude=['O'])``). To exclude pandas categorical columns, use ``'category'``
- None (default) : The result will exclude nothing.

#### Returns

-----

#### Series or DataFrame

Summary statistics of the Series or Dataframe provided.

#### See Also

-----

DataFrame.count: Count number of non-NA/null observations.

DataFrame.max: Maximum of the values in the object.

DataFrame.min: Minimum of the values in the object.

DataFrame.mean: Mean of the values.

DataFrame.std: Standard deviation of the observations.

DataFrame.select\_dtypes: Subset of a DataFrame including/excluding columns based on their dtype.

#### Notes

-----

For numeric data, the result's index will include ``count``,

``mean``, ``std``, ``min``, ``max`` as well as lower, ``50`` and upper percentiles. By default the lower percentile is ``25`` and the upper percentile is ``75``. The ``50`` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include ``count``, ``unique``, ``top``, and ``freq``. The ``top`` is the most common value. The ``freq`` is the most common value's frequency. Timestamps also include the ``first`` and ``last`` items.

If multiple object values have the highest count, then the ``count`` and ``top`` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a ``DataFrame``, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If ``include='all'`` is provided as an option, the result will include a union of attributes of each type.

The ``include`` and ``exclude`` parameters can be used to limit which columns in a ``DataFrame`` are analyzed for the output. The parameters are ignored when analyzing a ``Series``.

#### Examples

-----  
Describing a numeric ``Series``.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
dtype: float64
```

Describing a categorical ``Series``.



```

| >>> s = pd.Series(['a', 'a', 'b', 'c'])
| >>> s.describe()
| count 4
| unique 3
| top a
| freq 2
| dtype: object
|
| Describing a timestamp ``Series``.
|
| >>> s = pd.Series([
| ... np.datetime64("2000-01-01"),
| ... np.datetime64("2010-01-01"),
| ... np.datetime64("2010-01-01")
| ...])
| >>> s.describe()
| count 3
| mean 2006-09-01 08:00:00
| min 2000-01-01 00:00:00
| 25% 2004-12-31 12:00:00
| 50% 2010-01-01 00:00:00
| 75% 2010-01-01 00:00:00
| max 2010-01-01 00:00:00
| dtype: object
|
| Describing a ``DataFrame``. By default only numeric fields
| are returned.
|
| >>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
| ... 'numeric': [1, 2, 3],
| ... 'object': ['a', 'b', 'c']
| ... })
| >>> df.describe()
|
| numeric
| count 3.0
| mean 2.0
| std 1.0
| min 1.0
| 25% 1.5
| 50% 2.0
| 75% 2.5
| max 3.0
|

```

Describing all columns of a ``DataFrame`` regardless of data type.

```
>>> df.describe(include='all') # doctest: +SKIP
```

	categorical	numeric	object
count	3	3.0	3
unique	3	NaN	3
top	f	NaN	a
freq	1	NaN	1
mean	NaN	2.0	NaN
std	NaN	1.0	NaN
min	NaN	1.0	NaN
25%	NaN	1.5	NaN
50%	NaN	2.0	NaN
75%	NaN	2.5	NaN
max	NaN	3.0	NaN

Describing a column from a ``DataFrame`` by accessing it as an attribute.

```
>>> df.numeric.describe()
```

count	3.0
mean	2.0
std	1.0
min	1.0
25%	1.5
50%	2.0
75%	2.5
max	3.0

Name: numeric, dtype: float64

Including only numeric columns in a ``DataFrame`` description.

```
>>> df.describe(include=[np.number])
```

	numeric
count	3.0
mean	2.0
std	1.0
min	1.0
25%	1.5
50%	2.0
75%	2.5
max	3.0

Including only string columns in a ``DataFrame`` description.

```
>>> df.describe(include=[object]) # doctest: +SKIP
```

```
 object
count 3
unique 3
top a
freq 1
```

Including only categorical columns from a ``DataFrame`` description.

```
>>> df.describe(include=['category'])
```

```
 categorical
count 3
unique 3
top d
freq 1
```

Excluding numeric columns from a ``DataFrame`` description.

```
>>> df.describe(exclude=[np.number]) # doctest: +SKIP
```

```
 categorical object
count 3 3
unique 3 3
top f a
freq 1 1
```

Excluding object columns from a ``DataFrame`` description.

```
>>> df.describe(exclude=[object]) # doctest: +SKIP
```

```
 categorical numeric
count 3 3.0
unique 3 NaN
top f NaN
freq 1 NaN
mean NaN 2.0
std NaN 1.0
min NaN 1.0
25% NaN 1.5
50% NaN 2.0
75% NaN 2.5
max NaN 3.0
```

```

| droplevel(self: 'NDFrameT', level: 'IndexLabel', axis: 'Axis' = 0) -> 'NDFrameT'
| Return Series/DataFrame with requested index / column level(s) removed.
|
| 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
| Axis along which the level(s) is removed:
|
| * 0 or 'index': remove level(s) in column.
| * 1 or 'columns': remove level(s) in row.
|
| For `Series` this parameter is unused and defaults to 0.
|
| Returns
| -----
|
| Series/DataFrame
| Series/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
| b
| 2 3 4
| 6 7 8
| 10 11 12
|
| >>> df.droplevel('level_2', axis=1)
| level_1 c d
| a b
| 1 2 3 4
| 5 6 7 8
| 9 10 11 12
|
| equals(self, other: 'object') -> 'bool_t'
| 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 row/column index do not need to have the same type, as long
| as the values are considered equal. Corresponding columns must be of
| 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.

#### 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
|
| ewm(self, com: 'float | None' = None, span: 'float | None' = None, halflife: 'float | Timedelta' = None, times: int = 1, min_periods: int = 0)
| Provide exponentially weighted (EW) calculations.
|
| Exactly one of ``com``, ``span``, ``halflife``, or ``alpha`` must be
| provided if ``times`` is not provided. If ``times`` is provided,
| ``halflife`` and one of ``com``, ``span`` or ``alpha`` may be provided.
|
| 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, str, timedelta, optional
| Specify decay in terms of half-life
|
|
$$\alpha = 1 - \exp(-\ln(2) / halflife)$$
, for
| $halflife > 0$.
|
| If ``times`` is specified, a timedelta convertible unit over which an
| observation decays to half its value. Only applicable to ``mean()``,
| and halflife value will not apply to the other functions.
|
| .. versionadded:: 1.1.0
|
| 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 ``np.nan``.

`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).

- When ``adjust=True`` (default), the EW function is calculated using weights  $w_i = (1 - \alpha)^i$ . For example, the EW moving average of the series  $[x_0, x_1, \dots, x_t]$  would be:

```
.. math::
 y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + \dots + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + \dots + (1 - \alpha)^t}
```

- When ``adjust=False``, the exponentially weighted function is calculated recursively:

```
.. math::
 \begin{split}
 y_0 &= x_0 \\
 y_t &= (1 - \alpha) y_{t-1} + \alpha x_t,
 \end{split}
```

`ignore_na` : bool, default False

Ignore missing values when calculating weights.

- When ``ignore\_na=False`` (default), weights are based on absolute positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $[x_0, \text{None}, x_2]$  are  $(1 - \alpha)^2$  and  $1$  if ``adjust=True``, and  $(1 - \alpha)^2$  and  $\alpha$  if ``adjust=False``.

- When ``ignore\_na=True``, weights are based on relative positions. For example, the weights of  $x_0$  and  $x_2$  used in calculating the final weighted average of  $[x_0, \text{None}, x_2]$  are  $1 - \alpha$  and  $1$  if ``adjust=True``, and  $1 - \alpha$  and  $\alpha$  if ``adjust=False``.

`axis` : {0, 1}, default 0

If ``0`` or ``'index'``, calculate across the rows.

If ``1`` or ``'columns'``, calculate across the columns.



For `Series` this parameter is unused and defaults to 0.

times : np.ndarray, Series, default None

.. versionadded:: 1.1.0

Only applicable to `mean()`.

Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

If 1-D array like, a sequence with the same shape as the observations.

method : str {'single', 'table'}, default 'single'

.. versionadded:: 1.4.0

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

Only applicable to `mean()`

Returns

-----

`ExponentialMovingWindow` subclass

See Also

-----

rolling : Provides rolling window calculations.  
expanding : Provides expanding transformations.

Notes

-----

See [:ref:Windowing Operations <window.exponentially\\_weighted>](#) for further usage details and examples.

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
| >>> df.ewm(alpha=2 / 3).mean()
| B
| 0 0.000000
| 1 0.750000
| 2 1.615385
| 3 1.615385
| 4 3.670213
|
| **adjust**
|
| >>> df.ewm(com=0.5, adjust=True).mean()
| B
| 0 0.000000
| 1 0.750000
| 2 1.615385
| 3 1.615385
| 4 3.670213
| >>> df.ewm(com=0.5, adjust=False).mean()
| B
| 0 0.000000
| 1 0.666667
| 2 1.555556
| 3 1.555556
| 4 3.650794
|
| **ignore_na**
|
| >>> df.ewm(com=0.5, ignore_na=True).mean()
| B
| 0 0.000000

```

```

| 1 0.750000
| 2 1.615385
| 3 1.615385
| 4 3.225000
| >>> df.ewm(com=0.5, ignore_na=False).mean()
|
| B
| 0 0.000000
| 1 0.750000
| 2 1.615385
| 3 1.615385
| 4 3.670213
|
| **times**
|
| Exponentially weighted mean with weights calculated with a timedelta ``halflife``
| relative to ``times``.
|
| >>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-17']
| >>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
|
| B
| 0 0.000000
| 1 0.585786
| 2 1.523889
| 3 1.523889
| 4 3.233686
|
| expanding(self, min_periods: 'int' = 1, axis: 'Axis' = 0, method: 'str' = 'single') -> 'I
| Provide expanding window calculations.
|
| Parameters
| -----
| min_periods : int, default 1
| Minimum number of observations in window required to have a value;
| otherwise, result is ``np.nan``.
|
| axis : int or str, default 0
| If ``0`` or ``'index'``, roll across the rows.
|
| If ``1`` or ``'columns'``, roll across the columns.
|
| For `Series` this parameter is unused and defaults to 0.
|
| method : str {'single', 'table'}, default 'single'

```

Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

This argument is only implemented when specifying `engine='numba'` in the method call.

`.. versionadded:: 1.3.0`

#### Returns

-----

`'Expanding'` subclass

#### See Also

-----

`rolling` : Provides rolling window calculations.

`ewm` : Provides exponential weighted functions.

#### Notes

-----

See `:ref:`Windowing Operations <window.expanding>`` for further usage details and examples.

#### 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
```

**`**min_periods**`**

Expanding sum with 1 vs 3 observations needed to calculate a value.

```
>>> df.expanding(1).sum()
```

```
 B
0 0.0
1 1.0
2 3.0
3 3.0
```

```

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

```

`filter(self: 'NDFrameT', items=None, like: 'str | None' = None, regex: 'str | None' = None)`  
 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, 'columns' for

DataFrame. For `Series` this parameter is unused and defaults to `None`.

#### Returns

-----

same type as input object

#### See Also

-----

`DataFrame.loc` : Access a group of rows and columns

by label(s) or a boolean array.

#### 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'])
>>> df
```

```
 one two three
mouse 1 2 3
rabbit 4 5 6
```

```
>>> # 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
```

```
first(self: 'NDFrameT', offset) -> 'NDFrameT'
```

Select initial periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function can select the first few rows based on a date offset.

## Parameters

```

offset : str, DateOffset or dateutil.relativedelta
```

The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

## Returns

```

```

```

Series or DataFrame
 A subset of the caller.

Raises

TypeError
 If the index is not a :class:`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 calendar days were returned, not the first
3 days observed in the dataset, and therefore data for 2018-04-13 was
not returned.

first_valid_index(self) -> 'Hashable | None'
 Return index for first non-NA value or None, if no non-NA value is found.

Returns

type of index

```

```

Notes

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

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

same type as items contained in object

Examples

>>> df = pd.DataFrame(
... [
... [24.3, 75.7, "high"],
... [31, 87.8, "high"],
... [22, 71.6, "medium"],
... [35, 95, "medium"],
...],
... columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
... index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
...)

>>> df
 temp_celsius temp_fahrenheit windspeed
2014-02-12 24.3 75.7 high
2014-02-13 31.0 87.8 high
2014-02-14 22.0 71.6 medium
2014-02-15 35.0 95.0 medium

>>> df.get(["temp_celsius", "windspeed"])
 temp_celsius windspeed
2014-02-12 24.3 high
2014-02-13 31.0 high
2014-02-14 22.0 medium

```



```

2014-02-15 35.0 medium
|
|
| >>> ser = df['windspeed']
| >>> ser.get('2014-02-13')
| 'high'
|
| If the key isn't found, the default value will be used.
|
| >>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
| 'default_value'
|
| >>> ser.get('2014-02-10', '[unknown]')
| '[unknown]'
|
head(self: 'NDFrameT', n: 'int' = 5) -> 'NDFrameT'
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]``.
|
| If n is larger than the number of rows, this function returns all rows.
|
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
|
| infer_objects(self: 'NDFrameT', copy: 'bool_t | None' = None) -> 'NDFrameT'

```

```

| 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.
|
| Parameters
| -----
| copy : bool, default True
| Whether to make a copy for non-object or non-inferable columns
| or Series.
|
| Returns
| -----
| 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
|
| last(self: 'NDFrameT', offset) -> 'NDFrameT'

```

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

For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

#### Parameters

offset : str, DateOffset, dateutil.relativedelta

The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the last 3 days.

#### Returns

Series or DataFrame

A subset of the caller.

#### Raises

TypeError

If the index is not a :class:`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 calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

`last_valid_index(self)` -> 'Hashable | None'

Return index for last non-NA value or None, if no non-NA value is found.

Returns

-----

type of index

Notes

-----

If all elements are non-NA/null, returns None.

Also returns None for empty Series/DataFrame.

`pad(self: 'NDFrameT', *, axis: 'None | Axis' = None, inplace: 'bool_t' = False, limit: 'int' = None)`

Synonym for `:meth:`DataFrame.fillna`` with ```method='ffill'```.

.. deprecated:: 2.0

Series/DataFrame.pad is deprecated. Use Series/DataFrame.ffill instead.

Returns

-----

Series/DataFrame or None

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

`pct_change(self: 'NDFrameT', periods: 'int' = 1, fill_method: "Literal['backfill', 'bfill', 'pad', 'ffill', None]" = 'pad')`

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

-----

`periods` : int, default 1

Periods to shift for forming percent change.

`fill_method` : {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'

How to handle NAs **before** computing percent changes.

```
limit : int, default None
 The number of consecutive NAs to fill before stopping.
freq : DateOffset, timedelta, or str, optional
 Increment to use from time series API (e.g. 'M' or BDay()).
**kwargs
 Additional keyword arguments are passed into
 `DataFrame.shift` or `Series.shift`.
```

#### Returns

-----

Series or DataFrame

The same type as the calling object.

#### See Also

-----

Series.diff : Compute the difference of two elements in a Series.

DataFrame.diff : Compute the difference of two elements in a DataFrame.

Series.shift : Shift the index by some number of periods.

DataFrame.shift : Shift the index by some number of periods.

#### Examples

-----

**\*\*Series\*\***

```
>>> s = pd.Series([90, 91, 85])
```

```
>>> s
```

```
0 90
```

```
1 91
```

```
2 85
```

```
dtype: int64
```

```
>>> s.pct_change()
```

```
0 NaN
```

```
1 0.011111
```

```
2 -0.065934
```

```
dtype: float64
```

```
>>> s.pct_change(periods=2)
```

```
0 NaN
```

```
1 NaN
```

```
2 -0.055556
```

```
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0 90.0
1 91.0
2 NaN
3 85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0 NaN
1 0.011111
2 0.000000
3 -0.065934
dtype: float64
```

**\*\*DataFrame\*\***

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
... 'FR': [4.0405, 4.0963, 4.3149],
... 'GR': [1.7246, 1.7482, 1.8519],
... 'IT': [804.74, 810.01, 860.13]},
... index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
```

	FR	GR	IT
1980-01-01	4.0405	1.7246	804.74
1980-02-01	4.0963	1.7482	810.01
1980-03-01	4.3149	1.8519	860.13

```
>>> df.pct_change()
 FR GR IT
1980-01-01 NaN NaN NaN
1980-02-01 0.013810 0.013684 0.006549
1980-03-01 0.053365 0.059318 0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```

| >>> df = pd.DataFrame({
| ... '2016': [1769950, 30586265],
| ... '2015': [1500923, 40912316],
| ... '2014': [1371819, 41403351]},
| ... index=['GOOG', 'APPL'])
| >>> df
|
| 2016 2015 2014
| GOOG 1769950 1500923 1371819
| APPL 30586265 40912316 41403351
|
| >>> df.pct_change(axis='columns', periods=-1)
| 2016 2015 2014
| GOOG 0.179241 0.094112 NaN
| APPL -0.252395 -0.011860 NaN
|
| pipe(self, func: 'Callable[..., T] | tuple[Callable[..., T], str]', *args, **kwargs) ->
| Apply chainable functions that expect Series or DataFrames.
|
| Parameters
| -----
| func : function
| Function to apply to the Series/DataFrame.
| ``args``, and ``kwargs`` are passed into ``func``.
| Alternatively a ``(callable, data_keyword)`` tuple where
| ``data_keyword`` is a string indicating the keyword of
| ``callable`` that expects the Series/DataFrame.
| args : iterable, optional
| Positional arguments passed into ``func``.
| kwargs : mapping, optional
| A dictionary of keyword arguments passed into ``func``.
|
| Returns
| -----
| the return type of ``func``.
|
| See Also
| -----
| DataFrame.apply : Apply a function along input axis of DataFrame.
| DataFrame.applymap : Apply a function elementwise on a whole DataFrame.
| Series.map : Apply a mapping correspondence on a
| :class:`~pandas.Series`.
|
| Notes

```



-----  
Use ``.pipe`` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> func(g(h(df), arg1=a), arg2=b, arg3=c) # doctest: +SKIP
```

You can write

```
>>> (df.pipe(h)
... .pipe(g, arg1=a)
... .pipe(func, arg2=b, arg3=c)
...) # doctest: +SKIP
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `func`` takes its data as `arg2``:

```
>>> (df.pipe(h)
... .pipe(g, arg1=a)
... .pipe((func, 'arg2'), arg1=a, arg3=c)
...) # doctest: +SKIP
```

```
rank(self: 'NDFrameT', axis: 'Axis' = 0, method: 'str' = 'average', numeric_only: 'bool_1
Compute numerical data ranks (1 through n) along axis.
```

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

-----  
axis : {0 or 'index', 1 or 'columns'}, default 0  
Index to direct ranking.

For `Series`` this parameter is unused and defaults to 0.

method : {'average', 'min', 'max', 'first', 'dense'}, default 'average'

How to rank the group of records that have the same value (i.e. ties):

- \* average: average rank of the group
- \* min: lowest rank in the group
- \* max: highest rank in the group
- \* first: ranks assigned in order they appear in the array
- \* dense: like 'min', but rank always increases by 1 between groups.

numeric\_only : bool, default False

For DataFrame objects, rank only numeric columns if set to True.

.. versionchanged:: 2.0.0

The default value of ``numeric\_only`` is now ``False``.

na\_option : {'keep', 'top', 'bottom'}, default 'keep'

How to rank NaN values:

- \* keep: assign NaN rank to NaN values
- \* top: assign lowest rank to NaN values
- \* bottom: assign highest rank to NaN values

ascending : bool, default True

Whether or not the elements should be ranked in ascending order.

pct : bool, default False

Whether or not to display the returned rankings in percentile form.

Returns

-----

same type as caller

Return a Series or DataFrame with data ranks as values.

See Also

-----

core.groupby.DataFrameGroupBy.rank : Rank of values within each group.

core.groupby.SeriesGroupBy.rank : Rank of values within each group.

Examples

-----

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
... 'spider', 'snake'],
... 'Number_legs': [4, 2, 4, 8, np.nan]})
```

```
>>> df
 Animal Number_legs
0 cat 4.0
1 penguin 2.0
2 dog 4.0
3 spider 8.0
4 snake NaN
```

Ties are assigned the mean of the ranks (by default) for the group.

```

| >>> s = pd.Series(range(5), index=list("abcde"))
| >>> s["d"] = s["b"]
| >>> s.rank()
| a 1.0
| b 2.5
| c 4.0
| d 2.5
| e 5.0
| dtype: float64

```

The following example shows how the method behaves with the above parameters:

- | \* default\_rank: this is the default behaviour obtained without using any parameter.
- | \* max\_rank: setting ``method = 'max'`` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- | \* NA\_bottom: choosing ``na\_option = 'bottom'``, if there are records with NaN values they are placed at the bottom of the ranking.
- | \* pct\_rank: when setting ``pct = True``, the ranking is expressed as percentile rank.

```

| >>> df['default_rank'] = df['Number_legs'].rank()
| >>> df['max_rank'] = df['Number_legs'].rank(method='max')
| >>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
| >>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
| >>> df
|
| Animal Number_legs default_rank max_rank NA_bottom pct_rank
| 0 cat 4.0 2.5 3.0 2.5 0.625
| 1 penguin 2.0 1.0 1.0 1.0 0.250
| 2 dog 4.0 2.5 3.0 2.5 0.625
| 3 spider 8.0 4.0 4.0 4.0 1.000
| 4 snake NaN NaN NaN 5.0 NaN

```

```

| reindex_like(self: 'NDFrameT', other, method: "Literal['backfill', 'bfill', 'pad', 'ffill']")
| Return an object with matching indices as other object.

```

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

## Parameters

-----

`other` : Object of the same data type  
Its row and column indices are used to define the new indices of this object.

`method` : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}  
Method to use for filling holes in reindexed DataFrame.  
Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- \* None (default): don't fill gaps
- \* pad / ffill: propagate last valid observation forward to next valid
- \* backfill / bfill: use next valid observation to fill gap
- \* nearest: use nearest valid observations to fill gap.

`copy` : bool, default True  
Return a new object, even if the passed indexes are the same.

`limit` : int, default None  
Maximum number of consecutive labels to fill for inexact matches.

`tolerance` : optional  
Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

## Returns

-----

Series or DataFrame  
Same type as caller, but with changed indices on each axis.

## See Also

-----

`DataFrame.set_index` : Set row labels.  
`DataFrame.reset_index` : Remove row labels or move them to new columns.  
`DataFrame.reindex` : Change to new indices or expand indices.

## Notes

```

| -----
| Same as calling
| ``.reindex(index=other.index, columns=other.columns,...)``.
|
| Examples
| -----
| >>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
| ... [31, 87.8, 'high'],
| ... [22, 71.6, 'medium'],
| ... [35, 95, 'medium']],
| ... columns=['temp_celsius', 'temp_fahrenheit',
| ... 'windspeed'],
| ... index=pd.date_range(start='2014-02-12',
| ... end='2014-02-15', freq='D'))
|
| >>> df1
|
| temp_celsius temp_fahrenheit windspeed
| 2014-02-12 24.3 75.7 high
| 2014-02-13 31.0 87.8 high
| 2014-02-14 22.0 71.6 medium
| 2014-02-15 35.0 95.0 medium
|
| >>> df2 = pd.DataFrame([[28, 'low'],
| ... [30, 'low'],
| ... [35.1, 'medium']],
| ... columns=['temp_celsius', 'windspeed'],
| ... index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
| ... '2014-02-15']))
|
| >>> df2
|
| temp_celsius windspeed
| 2014-02-12 28.0 low
| 2014-02-13 30.0 low
| 2014-02-15 35.1 medium
|
| >>> df2.reindex_like(df1)
|
| temp_celsius temp_fahrenheit windspeed
| 2014-02-12 28.0 NaN low
| 2014-02-13 30.0 NaN low
| 2014-02-14 NaN NaN NaN
| 2014-02-15 35.1 NaN medium
|
| rolling(self, window: 'int | dt.timedelta | str | BaseOffset | BaseIndexer', min_periods

```

Provide rolling window calculations.

#### Parameters

`window` : int, timedelta, str, offset, or BaseIndexer subclass

Size of the moving window.

If an integer, the fixed number of observations used for each window.

If a timedelta, str, or offset, the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes.

To learn more about the offsets & frequency strings, please see [this link](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-) <[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html#offset-](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-)

If a BaseIndexer subclass, the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely `min_periods`, `center`, `closed` and `step` will be passed to `get_window_bounds`.

`min_periods` : int, default None

Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.

For a window that is specified by an offset, `min_periods` will default to 1.

For a window that is specified by an integer, `min_periods` will default to the size of the window.

`center` : bool, default False

If False, set the window labels as the right edge of the window index.

If True, set the window labels as the center of the window index.

`win_type` : str, default None

If `None`, all points are evenly weighted.

If a string, it must be a valid `scipy.signal` window function

<<https://docs.scipy.org/doc/scipy/reference/signal.windows.html#module-scipy.sig>

Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match

the keywords specified in the Scipy window type method signature.

on : str, optional

For a DataFrame, a column label or Index level on which to calculate the rolling window, rather than the DataFrame's index.

Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

axis : int or str, default 0

If ``0`` or ``'index'``, roll across the rows.

If ``1`` or ``'columns'``, roll across the columns.

For `Series` this parameter is unused and defaults to 0.

closed : str, default None

If ``'right'``, the first point in the window is excluded from calculations.

If ``'left'``, the last point in the window is excluded from calculations.

If ``'both'``, the no points in the window are excluded from calculations.

If ``'neither'``, the first and last points in the window are excluded from calculations.

Default ``None`` (``'right'``).

.. versionchanged:: 1.2.0

The closed parameter with fixed windows is now supported.

step : int, default None

.. versionadded:: 1.5.0

Evaluate the window at every ``step`` result, equivalent to slicing as ``[:, :step]``. ``window`` must be an integer. Using a step argument other than None or 1 will produce a result with a different shape than the input.

method : str {'single', 'table'}, default 'single'

.. versionadded:: 1.3.0

Execute the rolling operation per single column or row (``'single'``)  
or over the entire object (``'table'``).

This argument is only implemented when specifying ``engine='numba'``  
in the method call.

#### Returns

-----

``Window`` subclass if a ``win\_type`` is passed

``Rolling`` subclass if ``win\_type`` is not passed

#### See Also

-----

expanding : Provides expanding transformations.

ewm : Provides exponential weighted functions.

#### Notes

-----

See :ref:`Windowing Operations <window.generic>` for further usage details  
and examples.

#### 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
```

```
window
```

Rolling sum with a window length of 2 observations.

```
>>> df.rolling(2).sum()
```

```
 B
0 NaN
1 1.0
2 3.0
```



```
3 NaN
4 NaN
```

Rolling sum with a window span of 2 seconds.

```
>>> df_time = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
... index = [pd.Timestamp('20130101 09:00:00'),
... pd.Timestamp('20130101 09:00:02'),
... pd.Timestamp('20130101 09:00:03'),
... pd.Timestamp('20130101 09:00:05'),
... pd.Timestamp('20130101 09:00:06')])
```

```
>>> df_time
 B
2013-01-01 09:00:00 0.0
2013-01-01 09:00:02 1.0
2013-01-01 09:00:03 2.0
2013-01-01 09:00:05 NaN
2013-01-01 09:00:06 4.0
```

```
>>> df_time.rolling('2s').sum()
 B
2013-01-01 09:00:00 0.0
2013-01-01 09:00:02 1.0
2013-01-01 09:00:03 3.0
2013-01-01 09:00:05 NaN
2013-01-01 09:00:06 4.0
```

Rolling sum with forward looking windows with 2 observations.

```
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
 B
0 1.0
1 3.0
2 2.0
3 4.0
4 4.0
```

**\*\*min\_periods\*\***

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```
|>>> df.rolling(2, min_periods=1).sum()
```

```
| B
| 0 0.0
| 1 1.0
| 2 3.0
| 3 2.0
| 4 4.0
```

```
|
| **center**
```

```
|
| Rolling sum with the result assigned to the center of the window index.
```

```
|>>> df.rolling(3, min_periods=1, center=True).sum()
```

```
| B
| 0 1.0
| 1 3.0
| 2 3.0
| 3 6.0
| 4 4.0
```

```
|>>> df.rolling(3, min_periods=1, center=False).sum()
```

```
| B
| 0 0.0
| 1 1.0
| 2 3.0
| 3 3.0
| 4 6.0
```

```
|
| **step**
```

```
|
| Rolling sum with a window length of 2 observations, minimum of 1 observation to
| calculate a value, and a step of 2.
```

```
|>>> df.rolling(2, min_periods=1, step=2).sum()
```

```
| B
| 0 0.0
| 2 3.0
| 4 4.0
```

```
|
| **win_type**
```

```
|
| Rolling sum with a window length of 2, using the Scipy ``'gaussian'``
```

| window type. ``std`` is required in the aggregation function.

| >>> df.rolling(2, win\_type='gaussian').sum(std=3)

|           B  
| 0       NaN  
| 1   0.986207  
| 2   2.958621  
| 3       NaN  
| 4       NaN

| \*\*on\*\*

| Rolling sum with a window length of 2 days.

| >>> df = pd.DataFrame({  
| ...     'A': [pd.to\_datetime('2020-01-01'),  
|             pd.to\_datetime('2020-01-01'),  
|             pd.to\_datetime('2020-01-02')],  
| ...     'B': [1, 2, 3], },  
| ...     index=pd.date\_range('2020', periods=3))

| >>> df  
|                    A   B  
| 2020-01-01 2020-01-01   1  
| 2020-01-02 2020-01-01   2  
| 2020-01-03 2020-01-02   3

| >>> df.rolling('2D', on='A').sum()

|                    A    B  
| 2020-01-01 2020-01-01  1.0  
| 2020-01-02 2020-01-01  3.0  
| 2020-01-03 2020-01-02  6.0

| sample(self: 'NDFrameT', n: 'int | None' = None, frac: 'float | None' = None, replace: 'bool' = True)

|     Return a random sample of items from an axis of object.

|     You can use `random\_state` for reproducibility.

|     Parameters

|     -----

|     n : int, optional

|         Number of items from axis to return. Cannot be used with `frac`.

|         Default = 1 if `frac` = None.

```

| frac : float, optional
| Fraction of axis items to return. Cannot be used with `n`.
| replace : bool, default False
| Allow or disallow sampling of the same row more than once.
| weights : str or ndarray-like, optional
| Default 'None' results in equal probability weighting.
| If passed a Series, will align with target object on index. Index
| values in weights not found in sampled object will be ignored and
| index values in sampled object not in weights will be assigned
| weights of zero.
| If called on a DataFrame, will accept the name of a column
| when axis = 0.
| Unless weights are a Series, weights must be same length as axis
| being sampled.
| If weights do not sum to 1, they will be normalized to sum to 1.
| Missing values in the weights column will be treated as zero.
| Infinite values not allowed.
| random_state : int, array-like, BitGenerator, np.random.RandomState, np.random.Generator
| If int, array-like, or BitGenerator, seed for random number generator.
| If np.random.RandomState or np.random.Generator, use as given.
|
| .. versionchanged:: 1.1.0
|
| array-like and BitGenerator object now passed to np.random.RandomState()
| as seed
|
| .. versionchanged:: 1.4.0
|
| np.random.Generator objects now accepted
|
| axis : {0 or 'index', 1 or 'columns', None}, default None
| Axis to sample. Accepts axis number or name. Default is stat axis
| for given data type. For `Series` this parameter is unused and defaults to `None`
| ignore_index : bool, default False
| If True, the resulting index will be labeled 0, 1, ..., n - 1.
|
| .. versionadded:: 1.3.0
|
| Returns
| -----
| Series or DataFrame
| A new object of same type as caller containing `n` items randomly
| sampled from the caller object.

```

## See Also

-----  
DataFrameGroupBy.sample: Generates random samples from each group of a DataFrame object.  
SeriesGroupBy.sample: Generates random samples from each group of a Series object.  
numpy.random.choice: Generates a random sample from a given 1-D numpy array.

## Notes

-----  
If `frac > 1`, `replacement` should be set to `True`.

## Examples

-----  
>>> df = pd.DataFrame({'num\_legs': [2, 4, 8, 0],  
... 'num\_wings': [2, 0, 0, 0],  
... 'num\_specimen\_seen': [10, 2, 1, 8]},  
... index=['falcon', 'dog', 'spider', 'fish'])  
>>> df

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
dog	4	0	2
spider	8	0	1
fish	0	0	8

Extract 3 random elements from the `Series` `df['num_legs']`:  
Note that we use `random_state` to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish 0
spider 8
falcon 2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
 num_legs num_wings num_specimen_seen
dog 4 0 2
fish 0 0 8
```

An upsample sample of the ``DataFrame`` with replacement:  
Note that `replace` parameter has to be `True` for `frac` parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
```

	num_legs	num_wings	num_specimen_seen
dog	4	0	2
fish	0	0	8
falcon	2	2	10
falcon	2	2	10
fish	0	0	8
dog	4	0	2
fish	0	0	8
dog	4	0	2

Using a DataFrame column as weights. Rows with larger value in the  
`num\_specimen\_seen` column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
```

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
fish	0	0	8

```
set_flags(self: 'NDFrameT', *, copy: 'bool_t' = False, allows_duplicate_labels: 'bool_t')
Return a new object with updated flags.
```

Parameters

-----

copy : bool, default False

Specify if a copy of the object should be made.

allows\_duplicate\_labels : bool, optional

Whether the returned object allows duplicate labels.

Returns

-----

Series or DataFrame

The same type as the caller.

See Also

-----

DataFrame.attrs : Global metadata applying to this dataset.

DataFrame.flags : Global flags applying to this object.

## Notes

-----

This method returns a new object that's a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `:attr: `DataFrame.attrs``.

## Examples

-----

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

```
squeeze(self, axis: 'Axis | None' = None)
```

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call ``squeeze`` to ensure you have a Series.

## Parameters

-----

`axis` : {0 or 'index', 1 or 'columns', None}, default None

A specific axis to squeeze. By default, all length-1 axes are squeezed. For ``Series`` this parameter is unused and defaults to ``None``.

## Returns

-----

DataFrame, Series, or scalar

The projection after squeezing ``axis`` or all the axes.

|  
| See Also  
|

| -----  
| Series.iloc : Integer-location based indexing for selecting scalars.  
| DataFrame.iloc : Integer-location based indexing for selecting Series.  
| Series.to\_frame : Inverse of DataFrame.squeeze for a  
|       single-column DataFrame.  
|

| Examples  
|

| -----  
| >>> primes = pd.Series([2, 3, 5, 7])  
|

| Slicing might produce a Series with a single value:  
|

| >>> even\_primes = primes[primes % 2 == 0]  
| >>> even\_primes  
| 0    2  
| dtype: int64  
|

| >>> even\_primes.squeeze()  
| 2  
|

| Squeezing objects with more than one value in every axis does nothing:  
|

| >>> odd\_primes = primes[primes % 2 == 1]  
| >>> odd\_primes  
| 1    3  
| 2    5  
| 3    7  
| dtype: int64  
|

| >>> odd\_primes.squeeze()  
| 1    3  
| 2    5  
| 3    7  
| dtype: int64  
|

| Squeezing is even more effective when used with DataFrames.  
|

| >>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])  
| >>> df  
|     a   b  
| 0   1   2  
|



```
1 3 4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
 a
0 1
1 3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0 1
1 3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
 a
0 1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a 1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

```
swapaxes(self: 'NDFrameT', axis1: 'Axis', axis2: 'Axis', copy: 'bool_t | None' = None) ->
Interchange axes and swap values axes appropriately.
```

Returns

-----

same as input

```

| tail(self: 'NDFrameT', n: 'int' = 5) -> 'NDFrameT'
| Return the last `n` rows.
|
| This function returns last `n` rows from the object based on
| position. It is useful for quickly verifying data, for example,
| after sorting or appending rows.
|
| For negative values of `n`, this function returns all rows except
| the first `|n|` rows, equivalent to ``df[|n|:]``.
|
| If n is larger than the number of rows, this function returns all rows.
|
| Parameters
| -----
| n : int, default 5
| Number of rows to select.
|
| Returns
| -----
| type of caller
| The last `n` rows of the caller object.
|
| See Also
| -----
| DataFrame.head : The first `n` rows of the caller object.
|
| 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 last 5 lines

```
>>> df.tail()
 animal
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

Viewing the last `n` lines (three in this case)

```
>>> df.tail(3)
 animal
6 shark
7 whale
8 zebra
```

For negative values of `n`

```
>>> df.tail(-3)
 animal
3 lion
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

```
to_clipboard(self, excel: 'bool_t' = True, sep: 'str | None' = None, **kwargs) -> 'None'
Copy object to the system clipboard.
```

Write a text representation of object to the system clipboard.  
This can be pasted into Excel, for example.

Parameters

-----

excel : bool, default True

Produce output in a csv format for easy pasting into excel.

- True, use the provided separator for csv pasting.

- False, write a string representation of the object to the clipboard.

sep : str, default ``'\t'``

Field delimiter.

\*\*kwargs

These parameters will be passed to DataFrame.to\_csv.

See Also

-----  
DataFrame.to\_csv : Write a DataFrame to a comma-separated values  
(csv) file.

read\_clipboard : Read text from clipboard and pass to read\_csv.

Notes

-----  
Requirements for your platform.

- Linux : `xclip`, or `xsel` (with `PyQt4` modules)
- Windows : none
- macOS : none

This method uses the processes developed for the package `pyperclip`. A solution to render any output string format is given in the examples.

Examples

-----  
Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
```

```
>>> df.to_clipboard(sep=',') # doctest: +SKIP
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword `index` and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False) # doctest: +SKIP
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

Using the original `pyperclip` package for any string output format.

```
.. code-block:: python
```

```
import pyperclip
html = df.style.to_html()
pyperclip.copy(html)
```

```
to_csv(self, path_or_buf: 'FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None' = None)
Write object to a comma-separated values (csv) file.
```

Parameters

-----

`path_or_buf` : str, path object, file-like object, or None, default None  
String, path object (implementing `os.PathLike[str]`), or file-like object implementing a `write()` function. If None, the result is returned as a string. If a non-binary file object is passed, it should be opened with `newline=''`, disabling universal newlines. If a binary file object is passed, `mode` might need to contain a `'b'`.

.. versionchanged:: 1.2.0

Support for binary file objects was introduced.

`sep` : str, default ','

String of length 1. Field delimiter for the output file.

`na_rep` : str, default ''

Missing data representation.

`float_format` : str, Callable, default None

Format string for floating point numbers. If a Callable is given, it takes precedence over other numeric formatting parameters, like `decimal`.

`columns` : sequence, optional

Columns to write.

`header` : bool or list of str, default True

Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

`index` : bool, default True

Write row names (index).

`index_label` : str or sequence, or False, default None

Column label for index column(s) if desired. If None is given, and `header` and `index` are True, then the index names are used. A sequence should be given if the object uses `MultiIndex`. If False do not print fields for index names. Use `index_label=False`

```

| for easier importing in R.
| mode : str, default 'w'
| Python write mode. The available write modes are the same as
| :py:func:`open`.
| encoding : str, optional
| A string representing the encoding to use in the output file,
| defaults to 'utf-8'. `encoding` is not supported if `path_or_buf`
| is a non-binary file object.
| compression : str or dict, default 'infer'
| For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' is
| path-like, then detect compression from the following extensions: '.gz',
| '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2'
| (otherwise no compression).
| Set to ``None`` for no compression.
| Can also be a dict with key ``'method'`` set
| to one of {``'zip'``, ``'gzip'``, ``'bz2'``, ``'zstd'``, ``'tar'``} and other
| key-value pairs are forwarded to
| ``zipfile.ZipFile``, ``gzip.GzipFile``,
| ``bz2.BZ2File``, ``zstandard.ZstdCompressor`` or
| ``tarfile.TarFile``, respectively.
| As an example, the following could be passed for faster compression and to create
| a reproducible gzip archive:
| ``compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}``.
|
| .. versionadded:: 1.5.0
| Added support for `.tar` files.
|
| .. versionchanged:: 1.0.0
|
| May now be a dict with key 'method' as compression mode
| and other entries as additional compression options if
| compression mode is 'zip'.
|
| .. versionchanged:: 1.1.0
|
| Passing compression options as keys in dict is
| supported for compression modes 'gzip', 'bz2', 'zstd', and 'zip'.
|
| .. versionchanged:: 1.2.0
|
| Compression is supported for binary file objects.
|
| .. versionchanged:: 1.2.0

```

Previous versions forwarded dict entries for 'gzip' to  
`gzip.open` instead of `gzip.GzipFile` which prevented  
setting `mtime`.

quoting : optional constant from csv module

Defaults to csv.QUOTE\_MINIMAL. If you have set a `float\_format`  
then floats are converted to strings and thus csv.QUOTE\_NONNUMERIC  
will treat them as non-numeric.

quotechar : str, default '\"'

String of length 1. Character used to quote fields.

lineterminator : str, optional

The newline character or character sequence to use in the output  
file. Defaults to `os.linesep`, which depends on the OS in which  
this method is called (`\n` for linux, `\r\n` for Windows, i.e.).

.. versionchanged:: 1.5.0

Previously was `line_terminator`, changed for consistency with  
`read_csv` and the standard library 'csv' module.

chunksize : int or None

Rows to write at a time.

date\_format : str, default None

Format string for datetime objects.

doublequote : bool, default True

Control quoting of `quotechar` inside a field.

escapechar : str, default None

String of length 1. Character used to escape `sep` and `quotechar`  
when appropriate.

decimal : str, default '.'

Character recognized as decimal separator. E.g. use ',' for  
European data.

errors : str, default 'strict'

Specifies how encoding and decoding errors are to be handled.  
See the errors argument for `:func:`open`` for a full list  
of options.

.. versionadded:: 1.1.0

storage\_options : dict, optional

Extra options that make sense for a particular storage connection, e.g.  
host, port, username, password, etc. For HTTP(S) URLs the key-value pairs

are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer [here](https://pandas.pydata.org/docs/user_guide/io.html?highlight=storage_options#reading-writing-remote-files)

```
.. versionadded:: 1.2.0
```

#### Returns

-----

None or str

If `path_or_buf` is None, returns the resulting csv format as a string. Otherwise returns None.

#### See Also

-----

`read_csv` : Load a CSV file into a DataFrame.

`to_excel` : Write DataFrame to an Excel file.

#### Examples

-----

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
... 'mask': ['red', 'purple'],
... 'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

Create 'out.zip' containing 'out.csv'

```
>>> compression_opts = dict(method='zip',
... archive_name='out.csv') # doctest: +SKIP
>>> df.to_csv('out.zip', index=False,
... compression=compression_opts) # doctest: +SKIP
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path # doctest: +SKIP
>>> filepath = Path('folder/subfolder/out.csv') # doctest: +SKIP
>>> filepath.parent.mkdir(parents=True, exist_ok=True) # doctest: +SKIP
>>> df.to_csv(filepath) # doctest: +SKIP
```



```

| >>> import os # doctest: +SKIP
| >>> os.makedirs('folder/subfolder', exist_ok=True) # doctest: +SKIP
| >>> df.to_csv('folder/subfolder/out.csv') # doctest: +SKIP
|
| to_excel(self, excel_writer, sheet_name: 'str' = 'Sheet1', na_rep: 'str' = '', float_for
| Write object to an Excel sheet.
|
| To write a single object to an Excel .xlsx file it is only necessary to
| specify a target file name. To write to multiple sheets it is necessary to
| create an `ExcelWriter` object with a target file name, and specify a sheet
| in the file to write to.
|
| Multiple sheets may be written to by specifying unique `sheet_name`.
| With all data written to the file it is necessary to save the changes.
| Note that creating an `ExcelWriter` object with a file name that already
| exists will result in the contents of the existing file being erased.
|
| Parameters
| -----
| excel_writer : path-like, file-like, or ExcelWriter object
| File path or existing ExcelWriter.
| sheet_name : str, default 'Sheet1'
| Name of sheet which will contain DataFrame.
| na_rep : str, default ''
| Missing data representation.
| float_format : str, optional
| Format string for floating point numbers. For example
| ``float_format="%.2f"`` will format 0.1234 to 0.12.
| columns : sequence or list of str, optional
| Columns to write.
| header : bool or list of str, default True
| Write out the column names. If a list of string is given it is
| assumed to be aliases for the column names.
| index : bool, default True
| Write row names (index).
| index_label : str or sequence, optional
| Column label for index column(s) if desired. If not specified, and
| `header` and `index` are True, then the index names are used. A
| sequence should be given if the DataFrame uses MultiIndex.
| startrow : int, default 0
| Upper left cell row to dump data frame.
| startcol : int, default 0
| Upper left cell column to dump data frame.

```

```

engine : str, optional
 Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this
 via the options ``io.excel.xlsx.writer`` or
 ``io.excel.xlsm.writer``.

merge_cells : bool, default True
 Write MultiIndex and Hierarchical Rows as merged cells.
inf_rep : str, default 'inf'
 Representation for infinity (there is no native representation for
 infinity in Excel).
freeze_panes : tuple of int (length 2), optional
 Specifies the one-based bottommost row and rightmost column that
 is to be frozen.
storage_options : dict, optional
 Extra options that make sense for a particular storage connection, e.g.
 host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 are forwarded to ``urllib.request.Request`` as header options. For other
 URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are
 forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more
 details, and for more examples on storage options refer `here
 <https://pandas.pydata.org/docs/user_guide/io.html?
 highlight=storage_options#reading-writing-remote-files>`_.

```

```
.. versionadded:: 1.2.0
```

#### See Also

```

to_csv : Write DataFrame to a comma-separated values (csv) file.
ExcelWriter : Class for writing DataFrame objects into excel sheets.
read_excel : Read an Excel file into a pandas DataFrame.
read_csv : Read a comma-separated values (csv) file into DataFrame.
io.formats.style.Styler.to_excel : Add styles to Excel sheet.

```

#### Notes

```

For compatibility with :meth:`~DataFrame.to_csv`,
to_excel serializes lists and dicts to strings before writing.

```

Once a workbook has been saved it is not possible to write further data without rewriting the whole workbook.

#### Examples

```

```

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
... index=['row 1', 'row 2'],
... columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx") # doctest: +SKIP
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
... sheet_name='Sheet_name_1') # doctest: +SKIP
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer: # doctest: +SKIP
... df1.to_excel(writer, sheet_name='Sheet_name_1')
... df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
... mode='a') as writer: # doctest: +SKIP
... df.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the `engine` keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter') # doctest: +SKIP
```

`to_hdf(self, path_or_buf: 'FilePath | HDFStore', key: 'str', mode: 'str' = 'a', compleve`

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

.. warning::

One can store a subclass of ``DataFrame`` or ``Series`` to HDF5, but the type of the subclass is lost upon storing.

For more information see the :ref:`user guide <io.hdf5>`.

#### Parameters

-----

path\_or\_buf : str or pandas.HDFStore

File path or HDFStore object.

key : str

Identifier for the group in the store.

mode : {'a', 'w', 'r+'}, default 'a'

Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

complevel : {0-9}, default None

Specifies a compression level for data.

A value of 0 or None disables compression.

complib : {'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'

Specifies the compression library to be used.

As of v0.20.2 these additional compressors for Blosc are supported

(default if no compressor specified: 'blosc:blosclz'):

{'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}.

Specifying a compression library which is not available issues a ValueError.

append : bool, default False

For Table formats, append the input data to the existing.

format : {'fixed', 'table', None}, default 'fixed'

Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

```

| - If None, pd.get_option('io.hdf.default_format') is checked,
| followed by fallback to "fixed".
| index : bool, default True
| Write DataFrame index as a column.
| min_itemsize : dict or int, optional
| Map column names to minimum string sizes for columns.
| nan_rep : Any, optional
| How to represent null values as str.
| Not allowed with append=True.
| dropna : bool, default False, optional
| Remove missing values.
| data_columns : list of columns or True, optional
| List of columns to create as indexed data columns for on-disk
| queries, or True to use all columns. By default only the axes
| of the object are indexed. See
| :ref:`Query via data columns<io.hdf5-query-data-columns>`. for
| more information.
| Applicable only to format='table'.
| errors : str, default 'strict'
| Specifies how encoding and decoding errors are to be handled.
| See the errors argument for :func:`open` for a full list
| of options.
| encoding : str, default "UTF-8"

```

See Also

```

| -----
| read_hdf : Read from HDF file.
| DataFrame.to_orc : Write a DataFrame to the binary orc format.
| DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
| DataFrame.to_sql : Write to a SQL table.
| DataFrame.to_feather : Write out feather-format for DataFrames.
| DataFrame.to_csv : Write out to a csv file.

```

Examples

```

| -----
| >>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
| ... index=['a', 'b', 'c']) # doctest: +SKIP
| >>> df.to_hdf('data.h5', key='df', mode='w') # doctest: +SKIP

```

We can add another object to the same file:

```

| >>> s = pd.Series([1, 2, 3, 4]) # doctest: +SKIP
| >>> s.to_hdf('data.h5', key='s') # doctest: +SKIP

```

```

|
| Reading from HDF file:
|
| >>> pd.read_hdf('data.h5', 'df') # doctest: +SKIP
| A B
| a 1 4
| b 2 5
| c 3 6
| >>> pd.read_hdf('data.h5', 's') # doctest: +SKIP
| 0 1
| 1 2
| 2 3
| 3 4
| dtype: int64
|
| to_json(self, path_or_buf: 'FilePath | WriteBuffer[bytes] | WriteBuffer[str] | None' = None)
| Convert the object to a JSON string.
|
| Note NaN's and None will be converted to null and datetime objects
| will be converted to UNIX timestamps.
|
| Parameters
| -----
| path_or_buf : str, path object, file-like object, or None, default None
| String, path object (implementing os.PathLike[str]), or file-like
| object implementing a write() function. If None, the result is
| returned as a string.
| orient : str
| Indication of expected JSON string format.
|
| * Series:
|
| - default is 'index'
| - allowed values are: {'split', 'records', 'index', 'table'}.
|
| * DataFrame:
|
| - default is 'columns'
| - allowed values are: {'split', 'records', 'index', 'columns',
| 'values', 'table'}.
|
| * The format of the JSON string:
|

```

- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}
- 'columns' : dict like {column -> {index -> value}}
- 'values' : just the values array
- 'table' : dict like {'schema': {schema}, 'data': {data}}

Describing the data, where data component is like ``orient='records'``.

date\_format : {None, 'epoch', 'iso'}  
 Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the `orient`. For ``orient='table'``, the default is 'iso'. For all other orients, the default is 'epoch'.

double\_precision : int, default 10  
 The number of decimal places to use when encoding floating point values.

force\_ascii : bool, default True  
 Force encoded string to be ASCII.

date\_unit : str, default 'ms' (milliseconds)  
 The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default\_handler : callable, default None  
 Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines : bool, default False  
 If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.

compression : str or dict, default 'infer'  
 For on-the-fly compression of the output data. If 'infer' and 'path\_or\_buf' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression).  
 Set to ``None`` for no compression.  
 Can also be a dict with key ``'method'`` set to one of {``'zip'``, ``'gzip'``, ``'bz2'``, ``'zstd'``, ``'tar'``} and other key-value pairs are forwarded to  
 ``zipfile.ZipFile``, ``gzip.GzipFile``,  
 ``bz2.BZ2File``, ``zstandard.ZstdCompressor`` or

``tarfile.TarFile``, respectively.  
As an example, the following could be passed for faster compression and to create a reproducible gzip archive:

```
``compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}```.
```

```
.. versionadded:: 1.5.0
```

```
 Added support for ``.tar`` files.
```

```
.. versionchanged:: 1.4.0 Zstandard support.
```

index : bool, default True

Whether to include the index values in the JSON string. Not including the index (``index=False``) is only supported when orient is 'split' or 'table'.

indent : int, optional

Length of whitespace used to indent each record.

storage\_options : dict, optional

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to ``urllib.request.Request`` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more details, and for more examples on storage options refer [here](https://pandas.pydata.org/docs/user_guide/io.html?highlight=storage_options#reading-writing-remote-files) <[https://pandas.pydata.org/docs/user\\_guide/io.html?highlight=storage\\_options#reading-writing-remote-files](https://pandas.pydata.org/docs/user_guide/io.html?highlight=storage_options#reading-writing-remote-files)>`\_.

```
.. versionadded:: 1.2.0
```

mode : str, default 'w' (writing)

Specify the IO mode for output when supplying a path\_or\_buf. Accepted args are 'w' (writing) and 'a' (append) only. mode='a' is only supported when lines is True and orient is 'records'.

Returns

-----

None or str

If path\_or\_buf is None, returns the resulting json format as a string. Otherwise returns None.

See Also

-----

read\_json : Convert a JSON string to pandas object.



## Notes

-----

The behavior of ``indent=0`` varies from the stdlib, which does not indent the output but does insert newlines. Currently, ``indent=0`` and the default ``indent=None`` are equivalent in pandas, though this may change in a future release.

``orient='table'`` contains a 'pandas\_version' field under 'schema'. This stores the version of `pandas` used in the latest revision of the schema.

## Examples

-----

```
>>> from json import loads, dumps
>>> df = pd.DataFrame(
... [["a", "b"], ["c", "d"]],
... index=["row 1", "row 2"],
... columns=["col 1", "col 2"],
...)

>>> result = df.to_json(orient="split")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
{
 "columns": [
 "col 1",
 "col 2"
],
 "index": [
 "row 1",
 "row 2"
],
 "data": [
 [
 "a",
 "b"
],
 [
 "c",
 "d"
]
]
}
```

```
}
|
```

Encoding/decoding a Dataframe using ``'records'`` formatted JSON.  
Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
[
| {
| "col 1": "a",
| "col 2": "b"
| },
| {
| "col 1": "c",
| "col 2": "d"
| }
|]
```

Encoding/decoding a Dataframe using ``'index'`` formatted JSON:

```
>>> result = df.to_json(orient="index")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
{
| "row 1": {
| "col 1": "a",
| "col 2": "b"
| },
| "row 2": {
| "col 1": "c",
| "col 2": "d"
| }
| }
```

Encoding/decoding a Dataframe using ``'columns'`` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
{
| "col 1": {
| "row 1": "a",
```

```

 "row 2": "c"
 },
 "col 2": {
 "row 1": "b",
 "row 2": "d"
 }
}

```

Encoding/decoding a Dataframe using ``'values'`` formatted JSON:

```

>>> result = df.to_json(orient="values")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
[
 [
 "a",
 "b"
],
 [
 "c",
 "d"
]
]

```

Encoding with Table Schema:

```

>>> result = df.to_json(orient="table")
>>> parsed = loads(result)
>>> dumps(parsed, indent=4) # doctest: +SKIP
{
 "schema": {
 "fields": [
 {
 "name": "index",
 "type": "string"
 },
 {
 "name": "col 1",
 "type": "string"
 },
 {
 "name": "col 2",
 "type": "string"
 }
]
 }
}

```

```

 }
],
 "primaryKey": [
 "index"
],
 "pandas_version": "1.4.0"
},
"data": [
 {
 "index": "row 1",
 "col 1": "a",
 "col 2": "b"
 },
 {
 "index": "row 2",
 "col 1": "c",
 "col 2": "d"
 }
]
}

```

`to_latex(self, buf: 'FilePath | WriteBuffer[str] | None' = None, columns: 'Sequence[Hashable]')`  
 Render object to a LaTeX tabular, longtable, or nested table.

Requires ```\usepackage{booktabs}```. The output can be copy/pasted into a main LaTeX document or read from an external file with ```\input{table.tex}```.

`.. versionchanged:: 1.2.0`  
 Added position argument, changed meaning of caption argument.

`.. versionchanged:: 2.0.0`  
 Refactored to use the Styler implementation via jinja2 templating.

#### Parameters

-----

`buf` : str, Path or StringIO-like, optional, default None  
 Buffer to write to. If None, the output is returned as a string.

`columns` : list of label, optional  
 The subset of columns to write. Writes all columns by default.

`header` : bool or list of str, default True  
 Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

```

| index : bool, default True
| Write row names (index).
| na_rep : str, default 'NaN'
| Missing data representation.
| formatters : list of functions or dict of {{str: function}}, optional
| Formatter functions to apply to columns' elements by position or
| name. The result of each function must be a unicode string.
| List must be of length equal to the number of columns.
| float_format : one-parameter function or str, optional, default None
| Formatter for floating point numbers. For example
| ``float_format="%.2f"`` and ``float_format="{:0.2f}"`` will
| both result in 0.1234 being formatted as 0.12.
| sparsify : bool, optional
| Set to False for a DataFrame with a hierarchical index to print
| every multiindex key at each row. By default, the value will be
| read from the config module.
| index_names : bool, default True
| Prints the names of the indexes.
| bold_rows : bool, default False
| Make the row labels bold in the output.
| column_format : str, optional
| The columns format as specified in `LaTeX table format
| <https://en.wikibooks.org/wiki/LaTeX/Tables>`__ e.g. 'rcl' for 3
| columns. By default, 'l' will be used for all columns except
| columns of numbers, which default to 'r'.
| longtable : bool, optional
| Use a longtable environment instead of tabular. Requires
| adding a \usepackage{longtable} to your LaTeX preamble.
| By default, the value will be read from the pandas config
| module, and set to `True` if the option ``styler.latex.environment`` is
| "longtable".
|
| .. versionchanged:: 2.0.0
| The pandas option affecting this argument has changed.
| escape : bool, optional
| By default, the value will be read from the pandas config
| module and set to `True` if the option ``styler.format.escape`` is
| "latex". When set to False prevents from escaping latex special
| characters in column names.
|
| .. versionchanged:: 2.0.0
| The pandas option affecting this argument has changed, as has the
| default value to `False`.

```

```

| encoding : str, optional
| A string representing the encoding to use in the output file,
| defaults to 'utf-8'.
| decimal : str, default '.'
| Character recognized as decimal separator, e.g. ',' in Europe.
| multicolumn : bool, default True
| Use \multicolumn to enhance MultiIndex columns.
| The default will be read from the config module, and is set
| as the option ``styler.sparse.columns``.
|
| .. versionchanged:: 2.0.0
| The pandas option affecting this argument has changed.
| multicolumn_format : str, default 'r'
| The alignment for multicolumns, similar to `column_format`
| The default will be read from the config module, and is set as the option
| ``styler.latex.multicol_align``.
|
| .. versionchanged:: 2.0.0
| The pandas option affecting this argument has changed, as has the
| default value to "r".
| multirow : bool, default True
| Use \multirow to enhance MultiIndex rows. Requires adding a
| \usepackage{{multirow}} to your LaTeX preamble. Will print
| centered labels (instead of top-aligned) across the contained
| rows, separating groups via clines. The default will be read
| from the pandas config module, and is set as the option
| ``styler.sparse.index``.
|
| .. versionchanged:: 2.0.0
| The pandas option affecting this argument has changed, as has the
| default value to `True`.
| caption : str or tuple, optional
| Tuple (full_caption, short_caption),
| which results in ``\caption[short_caption]{{full_caption}}``;
| if a single string is passed, no short caption will be set.
|
| .. versionchanged:: 1.2.0
| Optionally allow caption to be a tuple ``(full_caption, short_caption)``.
|
| label : str, optional
| The LaTeX label to be placed inside ``\label{{}}`` in the output.
| This is used with ``\ref{{}}`` in the main ``.tex`` file.

```

position : str, optional  
The LaTeX positional argument for tables, to be placed after  
```\begin{table}``` in the output.

.. versionadded:: 1.2.0

Returns

str or None

If buf is None, returns the result as a string. Otherwise returns None.

See Also

`io.formats.style.Styler.to_latex` : Render a DataFrame to LaTeX
with conditional formatting.

`DataFrame.to_string` : Render a DataFrame to a console-friendly
tabular output.

`DataFrame.to_html` : Render a DataFrame as an HTML table.

Notes

As of v2.0.0 this method has changed to use the Styler implementation as
part of `:meth:`.Styler.to_latex`` via ```jinja2``` templating. This means
that ```jinja2``` is a requirement, and needs to be installed, for this method
to function. It is advised that users switch to using Styler, since that
implementation is more frequently updated and contains much more
flexibility with the output.

Examples

Convert a general DataFrame to LaTeX with formatting:

```
>>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                          age=[26, 45],
...                          height=[181.23, 177.65]))
>>> print(df.to_latex(index=False,
...                    formatters={"name": str.upper},
...                    float_format="{:.1f}".format,
... )) # doctest: +SKIP
\begin{tabular}{lrr}
\toprule
name & age & height \\
\midrule
```

```

RAPHAEL & 26 & 181.2 \\
DONATELLO & 45 & 177.7 \\
\bottomrule
\end{tabular}

```

```

to_pickle(self, path: 'FilePath | WriteBuffer[bytes]', compression: 'CompressionOptions')
    Pickle (serialize) object to file.

```

Parameters

`path` : str, path object, or file-like object

String, path object (implementing `os.PathLike[str]`), or file-like object implementing a binary `write()` function. File path where the pickled object will be stored.

`compression` : str or dict, default 'infer'

For on-the-fly compression of the output data. If 'infer' and 'path' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression).

Set to `None` for no compression.

Can also be a dict with key `'method'` set

to one of `{'zip', 'gzip', 'bz2', 'zstd', 'tar'}` and other key-value pairs are forwarded to

`zipfile.ZipFile`, `gzip.GzipFile`,

`bz2.BZ2File`, `zstandard.ZstdCompressor` or

`tarfile.TarFile`, respectively.

As an example, the following could be passed for faster compression and to create a reproducible gzip archive:

```

compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}

```

.. versionadded:: 1.5.0

Added support for `.tar` files.

`protocol` : int

Int which indicates which protocol should be used by the pickler, default `HIGHEST_PROTOCOL` (see [1]_ paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to `HIGHEST_PROTOCOL`.

.. [1] <https://docs.python.org/3/library/pickle.html>.

`storage_options` : dict, optional

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs

are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer [here](https://pandas.pydata.org/docs/user_guide/io.html?highlight=storage_options#reading-writing-remote-files)

`.. versionadded:: 1.2.0`

See Also

`read_pickle` : Load pickled pandas object (or any object) from file.
`DataFrame.to_hdf` : Write DataFrame to an HDF5 file.
`DataFrame.to_sql` : Write DataFrame to a SQL database.
`DataFrame.to_parquet` : Write a DataFrame to the binary parquet format.

Examples

`>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)}) # doctest: +SKIP`
`>>> original_df # doctest: +SKIP`

	foo	bar
0	0	5
1	1	6
2	2	7
3	3	8
4	4	9

`>>> original_df.to_pickle("./dummy.pkl") # doctest: +SKIP`

`>>> unpickled_df = pd.read_pickle("./dummy.pkl") # doctest: +SKIP`
`>>> unpickled_df # doctest: +SKIP`

	foo	bar
0	0	5
1	1	6
2	2	7
3	3	8
4	4	9

`to_sql(self, name: 'str', con, schema: 'str | None' = None, if_exists: "Literal['fail', 'replace', 'append', 'ignore']" = 'fail', chunksize: int = None, chunkmethod: str = 'parquet', write_index: bool = True, dtype: dict[str, str] = None, index_label: str = None, **kwargs)`
Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name : str

Name of SQL table.

con : sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection

Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable. See [here](https://docs.sqlalchemy.org/en/20/core/c) <https://docs.sqlalchemy.org/en/20/core/c

If passing a sqlalchemy.engine.Connection which is already in a transaction, the transaction will not be committed. If passing a sqlite3.Connection, it will not be possible to roll back the record insertion.

schema : str, optional

Specify the schema (if database flavor supports this). If None, use default schema.

if_exists : {'fail', 'replace', 'append'}, default 'fail'

How to behave if the table already exists.

* fail: Raise a ValueError.

* replace: Drop the table before inserting new values.

* append: Insert new values to the existing table.

index : bool, default True

Write DataFrame index as a column. Uses `index_label` as the column name in the table.

index_label : str or sequence, default None

Column label for index column(s). If None is given (default) and `index` is True, then the index names are used.

A sequence should be given if the DataFrame uses MultiIndex.

chunksize : int, optional

Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype : dict or scalar, optional

Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method : {None, 'multi', callable}, optional

Controls the SQL insertion clause used:

* None : Uses standard SQL `INSERT` clause (one per row).

* 'multi': Pass multiple values in a single `INSERT` clause.

* callable with signature ``(pd_table, conn, keys, data_iter)``.

Details and a sample callable implementation can be found in the section [:ref:`insert method <io.sql.method>](#).

Returns

None or int

Number of rows affected by `to_sql`. None is returned if the callable passed into `method` does not return an integer number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of `sqlite3.Cursor` or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the `sqlite3` [sqlite3 <https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.rowcount>](https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.rowcount) or SQLAlchemy [SQLAlchemy <https://docs.sqlalchemy.org/en/20/core/connections.html#sqlalchemy>](https://docs.sqlalchemy.org/en/20/core/connections.html#sqlalchemy).

.. versionadded:: 1.4.0

Raises

ValueError

When the table already exists and `if_exists` is 'fail' (the default).

See Also

`read_sql` : Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

References

- .. [1] <https://docs.sqlalchemy.org>
- .. [2] <https://www.python.org/dev/peps/pep-0249/>

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
```

```
   name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
3
```

```
>>> from sqlalchemy import text
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An `sqlalchemy.engine.Connection` can also be passed to `con`:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql('users', con=connection, if_exists='append')
2
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql('users', con=engine, if_exists='append')
2
>>> with engine.connect() as conn:
...     conn.execute(text("SELECT * FROM users")).fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql('users', con=engine, if_exists='replace',
```

```

|         ...         index_label='id')
| 2
| >>> with engine.connect() as conn:
| ...     conn.execute(text("SELECT * FROM users")).fetchall()
| [(0, 'User 6'), (1, 'User 7')]
|
| Specify the dtype (especially useful for integers with missing values).
| Notice that while pandas is forced to store the data as floating point,
| the database supports nullable integers. When fetching the data with
| Python, we get back integer scalars.
|
| >>> df = pd.DataFrame({"A": [1, None, 2]})
| >>> df
|      A
| 0  1.0
| 1  NaN
| 2  2.0
|
| >>> from sqlalchemy.types import Integer
| >>> df.to_sql('integers', con=engine, index=False,
| ...         dtype={"A": Integer()})
| 3
|
| >>> with engine.connect() as conn:
| ...     conn.execute(text("SELECT * FROM integers")).fetchall()
| [(1,), (None,), (2,)]
|
| to_xarray(self)
|     Return an xarray object from the pandas object.
|
| Returns
| -----
| xarray.DataArray or xarray.Dataset
|     Data in the pandas structure converted to Dataset if the object is
|     a DataFrame, or a DataArray if the object is a Series.
|
| See Also
| -----
| DataFrame.to_hdf : Write DataFrame to an HDF5 file.
| DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
|
| Notes
| -----

```

See the `xarray docs <<https://xarray.pydata.org/en/stable/>>`__

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                    columns=['name', 'class', 'max_speed',
...                             'num_legs'])
>>> df
   name  class  max_speed  num_legs
0  falcon  bird     389.0         2
1  parrot  bird      24.0         2
2   lion  mammal     80.5         4
3  monkey  mammal      NaN         4

>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 4)
Coordinates:
  * index    (index) int64 0 1 2 3
Data variables:
  name      (index) object 'falcon' 'parrot' 'lion' 'monkey'
  class     (index) object 'bird' 'bird' 'mammal' 'mammal'
  max_speed (index) float64 389.0 24.0 80.5 nan
  num_legs  (index) int64 2 2 4 4

>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. , 24. , 80.5,  nan])
Coordinates:
  * index    (index) int64 0 1 2 3

>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                          '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                         'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])

>>> df_multiindex
```

```

|           speed
| date      animal
| 2018-01-01 falcon   350
|           parrot   18
| 2018-01-02 falcon   361
|           parrot   15
|
| >>> df_multiindex.to_xarray()
| <xarray.Dataset>
| Dimensions:  (date: 2, animal: 2)
| Coordinates:
|   * date      (date) datetime64[ns] 2018-01-01 2018-01-02
|   * animal    (animal) object 'falcon' 'parrot'
| Data variables:
|   speed      (date, animal) int64 350 18 361 15
|
| truncate(self: 'NDFrameT', before=None, after=None, axis: 'Axis | None' = None, copy: 'b
|   Truncate a Series or DataFrame before and after some index value.
|
| This is a useful shorthand for boolean indexing based on index
| values above or below certain thresholds.
|
| Parameters
| -----
| before : date, str, int
|         Truncate all rows before this index value.
| after  : date, str, int
|         Truncate all rows after this index value.
| axis   : {0 or 'index', 1 or 'columns'}, optional
|         Axis to truncate. Truncates the index (rows) by default.
|         For `Series` this parameter is unused and defaults to 0.
| copy   : bool, default is True,
|         Return a copy of the truncated section.
|
| Returns
| -----
| type of caller
|         The truncated Series or DataFrame.
|
| See Also
| -----
| DataFrame.loc : Select a subset of a DataFrame by label.
| DataFrame.iloc : Select a subset of a DataFrame by position.

```

Notes

If the index being truncated contains only datetime values, `before` and `after` may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
```

```
>>> df
   A B C
1  a f k
2  b g l
3  c h m
4  d i n
5  e j o
```

```
>>> df.truncate(before=2, after=4)
```

```
   A B C
2  b g l
3  c h m
4  d i n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
```

```
   A B
1  a f
2  b g
3  c h
4  d i
5  e j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
```

```
2    b
3    c
4    d
```



```
Name: A, dtype: object
```

The index values in ``truncate`` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
```

```
          A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...              after=pd.Timestamp('2016-01-10')).tail()
```

```
          A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify `before` and `after` as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
```

```
          A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that ``truncate`` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
```

```
          A
2016-01-10 23:59:55  1
```

```

|      2016-01-10 23:59:56  1
|      2016-01-10 23:59:57  1
|      2016-01-10 23:59:58  1
|      2016-01-10 23:59:59  1
|
| tz_convert(self: 'NDFrameT', tz, axis: 'Axis' = 0, level=None, copy: 'bool_t | None' = None)
|     Convert tz-aware axis to target time zone.
|
| Parameters
| -----
| tz : str or tzinfo object or None
|     Target time zone. Passing ``None`` will convert to
|     UTC and remove the timezone information.
| axis : {0 or 'index', 1 or 'columns'}, default 0
|     The axis to convert
| level : int, str, default None
|     If axis is a MultiIndex, convert a specific level. Otherwise
|     must be None.
| copy : bool, default True
|     Also make a copy of the underlying data.
|
| Returns
| -----
| Series/DataFrame
|     Object with time zone converted axis.
|
| Raises
| -----
| TypeError
|     If the axis is tz-naive.
|
| Examples
| -----
| Change to another time zone:
|
| >>> s = pd.Series(
| ...     [1],
| ...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']),
| ... )
| >>> s.tz_convert('Asia/Shanghai')
| 2018-09-15 07:30:00+08:00    1
| dtype: int64
|

```

```

| Pass None to convert to UTC and get a tz-naive index:
|
| >>> s = pd.Series([1],
| ...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))
| >>> s.tz_convert(None)
| 2018-09-14 23:30:00    1
| dtype: int64
|
| tz_localize(self: 'NDFrameT', tz, axis: 'Axis' = 0, level=None, copy: 'bool_t | None' = True)
| Localize tz-naive index of a Series or DataFrame to target time zone.
|
| This operation localizes the Index. To localize the values in a
| timezone-naive Series, use :meth:`Series.dt.tz_localize`.
|
| Parameters
| -----
| tz : str or tzinfo or None
|     Time zone to localize. Passing ``None`` will remove the
|     time zone information and preserve local time.
| axis : {0 or 'index', 1 or 'columns'}, default 0
|     The axis to localize
| level : int, str, default None
|     If axis is a MultiIndex, localize a specific level. Otherwise
|     must be None.
| copy : bool, default True
|     Also make a copy of the underlying data.
| ambiguous : 'infer', bool-ndarray, 'NaT', default 'raise'
|     When clocks moved backward due to DST, ambiguous times may arise.
|     For example in Central European Time (UTC+01), when going from
|     03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at
|     00:30:00 UTC and at 01:30:00 UTC. In such a situation, the
|     `ambiguous` parameter dictates how ambiguous times should be
|     handled.
|
|     - 'infer' will attempt to infer fall dst-transition hours based on
|       order
|     - bool-ndarray where True signifies a DST time, False designates
|       a non-DST time (note that this flag is only applicable for
|       ambiguous times)
|     - 'NaT' will return NaT where there are ambiguous times
|     - 'raise' will raise an AmbiguousTimeError if there are ambiguous
|       times.
| nonexistent : str, default 'raise'

```

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

Returns

Series/DataFrame

Same type as the input.

Raises

TypeError

If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series(  
...     [1],  
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00']),  
... )  
>>> s.tz_localize('CET')  
2018-09-15 01:30:00+02:00    1  
dtype: int64
```

Pass None to convert to tz-naive index and preserve local time:

```
>>> s = pd.Series([1],  
...     index=pd.DatetimeIndex(['2018-09-15 01:30:00+02:00']))  
>>> s.tz_localize(None)  
2018-09-15 01:30:00    1  
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas

can infer the DST time:

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
...                                       '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or `'shift_forward'` or `'shift_backward'`.

```
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                       '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00    0
2015-03-29 03:30:00+02:00    1
```

```

| dtype: int64
| >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
| 2015-03-29 01:59:59.999999999+01:00    0
| 2015-03-29 03:30:00+02:00          1
| dtype: int64
| >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
| 2015-03-29 03:30:00+02:00    0
| 2015-03-29 03:30:00+02:00    1
| dtype: int64
|
| xs(self: 'NDFrameT', key: 'IndexLabel', axis: 'Axis' = 0, level: 'IndexLabel' = None, drop_level: bool = True)
| Return cross-section from the Series/DataFrame.
|
| This method takes a `key` argument to select data at a particular
| level of a MultiIndex.
|
| Parameters
| -----
| key : label or tuple of label
|       Label contained in the index, or partially in a MultiIndex.
| axis : {0 or 'index', 1 or 'columns'}, default 0
|       Axis to retrieve cross-section on.
| level : object, defaults to first n levels (n=1 or len(key))
|       In case of a key partially contained in a MultiIndex, indicate
|       which levels are used. Levels can be referred by label or position.
| drop_level : bool, default True
|       If False, returns object with same levels as self.
|
| Returns
| -----
| Series or DataFrame
|       Cross-section from the original Series or DataFrame
|       corresponding to the selected index levels.
|
| See Also
| -----
| DataFrame.loc : Access a group of rows and columns
|                 by label(s) or a boolean array.
| DataFrame.iloc : Purely integer-location based indexing
|                 for selection by position.
|
| Notes
| -----

```

`xs` can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels.

It is a superset of `xs` functionality, see

:ref:`MultiIndex Slicers <advanced.mi_slicers>`.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
```

			num_legs	num_wings
class	animal	locomotion		
mammal	cat	walks	4	0
	dog	walks	4	0
	bat	flies	2	2
bird	penguin	walks	2	2

Get values at specified index

```
>>> df.xs('mammal')
           num_legs  num_wings
animal locomotion
cat     walks      4          0
dog     walks      4          0
bat     flies      2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog', 'walks'))
num_legs      4
num_wings      0
Name: (mammal, dog, walks), dtype: int64
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
```

```

                num_legs  num_wings
class locomotion
mammal walks                4         0

```

Get values at several indexes and levels

```

>>> df.xs(('bird', 'walks'),
...       level=[0, 'locomotion'])
                num_legs  num_wings
animal
penguin                2         2

```

Get values at specified column and axis

```

>>> df.xs('num_wings', axis=1)
class  animal  locomotion
mammal  cat    walks      0
         dog    walks      0
         bat    flies      2
bird    penguin walks      2
Name: num_wings, dtype: int64

```

Readonly properties inherited from pandas.core.generic.NDFrame:

flags

Get the properties associated with this pandas object.

The available flags are

* :attr:`Flags.allows_duplicate_labels`

See Also

Flags : Flags that apply to pandas objects.

DataFrame.attrs : Global metadata applying to this dataset.

Notes

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in :attr:`DataFrame.attrs`.


```

|   Examples
|   -----
|   >>> df = pd.DataFrame({"A": [1, 2]})
|   >>> df.flags
|   <Flags(allows_duplicate_labels=True)>
|
|   Flags can be get or set using ``.``
|
|   >>> df.flags.allows_duplicate_labels
|   True
|   >>> df.flags.allows_duplicate_labels = False
|
|   Or by slicing with a key
|
|   >>> df.flags["allows_duplicate_labels"]
|   False
|   >>> df.flags["allows_duplicate_labels"] = True
|
|   -----
|   Data descriptors inherited from pandas.core.generic.NDFrame:
|
|   attrs
|       Dictionary of global attributes of this dataset.
|
|       .. warning::
|
|           attrs is experimental and may change without warning.
|
|       See Also
|       -----
|       DataFrame.flags : Global flags applying to this object.
|
|   -----
|   Methods inherited from pandas.core.base.PandasObject:
|
|   __sizeof__(self) -> 'int'
|       Generates the total memory usage for an object that returns
|       either a value or Series of values
|
|   -----
|   Methods inherited from pandas.core.accessor.DirNamesMixin:
|
|   __dir__(self) -> 'list[str]'

```

Provide method name lookup and completion.

Notes

Only provide 'public' methods.

Readonly properties inherited from `pandas.core.indexing.IndexingMixin`:

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a `DataFrame` or `Series`.

Raises

`KeyError`

* If getting a value and 'label' does not exist in a `DataFrame` or `Series`.

`ValueError`

* If row/column label pair is not a tuple or if any label from the pair is not a scalar for `DataFrame`.
* If label is list-like (**excluding** `NamedTuple`) for `Series`.

See Also

`DataFrame.at` : Access a single value for a row/column pair by label.

`DataFrame.iat` : Access a single value for a row/column pair by integer position.

`DataFrame.loc` : Access a group of rows and columns by label(s).

`DataFrame.iloc` : Access a group of rows and columns by integer position(s).

`Series.at` : Access a single value by label.

`Series.iat` : Access a single value by integer position.

`Series.loc` : Access a group of rows by label(s).

`Series.iloc` : Access a group of rows by integer position(s).

Notes

See [:ref:`Fast scalar value getting and setting <indexing.basics.get_value>`](#) for more details.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    index=[4, 5, 6], columns=['A', 'B', 'C'])
```

```
>>> df
```

```
   A  B  C
4  0  2  3
5  0  4  1
6 10 20 30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

iat

Access a single value for a row/column pair by integer position.

Similar to ``iloc``, in that both provide integer-based lookups. Use ``iat`` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError

When integer position is out of bounds.

See Also

`DataFrame.at` : Access a single value for a row/column label pair.

`DataFrame.loc` : Access a group of rows and columns by label(s).

DataFrame.iloc : Access a group of rows and columns by integer position(s).

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

iloc

Purely integer-location based indexing for selection by position.

`df.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

- A tuple of row and column indexes. The tuple elements consist of one of the above inputs, e.g. `((0, 1))`.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except `*slice*` indexers which allow out-of-bounds indexing (this conforms with python/numpy `*slice*` semantics).

See more at [:ref:`Selection by Position <indexing.integer>`](#).

See Also

`DataFrame.iat` : Fast integer location scalar accessor.
`DataFrame.loc` : Purely label-location based indexer for selection by label.
`Series.iloc` : Purely integer-location based indexing for selection by position.

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
```

	a	b	c	d
0	1	2	3	4
1	100	200	300	400
2	1000	2000	3000	4000

****Indexing just the rows****

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
```

a	1
b	2
c	3
d	4

Name: 0, dtype: int64

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>

>>> df.iloc[[0, 1]]
   a  b  c  d
0  1  2  3  4
1 100 200 300 400
```

With a `slice` object.

```
>>> df.iloc[:3]
   a  b  c  d
0  1  2  3  4
1 100 200 300 400
2 1000 2000 3000 4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
   a  b  c  d
0  1  2  3  4
2 1000 2000 3000 4000
```

With a callable, useful in method chains. The `x` passed to the ``lambda`` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
   a  b  c  d
0  1  2  3  4
2 1000 2000 3000 4000
```

****Indexing both axes****

You can mix the indexer types for the index and columns. Use ``:``` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b      d
0      2      4
2 2000 4000
```

With `slice` objects.

```
>>> df.iloc[1:3, 0:3]
      a      b      c
1  100  200  300
2 1000 2000 3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a      c
0      1      3
1  100  300
2 1000 3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a      c
0      1      3
1  100  300
2 1000 3000
```

loc

Access a group of rows and columns by label(s) or a boolean array.

`df.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `df.loc['5']` or `df.loc['a']`, (note that `df.loc[5]` is

```

|         interpreted as a *label* of the index, and **never** as an
|         integer position along the index).
|     - A list or array of labels, e.g. ``['a', 'b', 'c']``.
|     - A slice object with labels, e.g. ``'a':'f'``.
|
|     .. warning:: Note that contrary to usual python slices, **both** the
|         start and the stop are included
|
|     - A boolean array of the same length as the axis being sliced,
|       e.g. ``[True, False, True]``.
|     - An alignable boolean Series. The index of the key will be aligned before
|       masking.
|     - An alignable Index. The Index of the returned selection will be the input.
|     - A ``callable`` function with one argument (the calling Series or
|       DataFrame) and that returns valid output for indexing (one of the above)
|
| See more at :ref:`Selection by Label <indexing.label>`.
|
| Raises
| -----
| KeyError
|     If any items are not found.
| IndexingError
|     If an indexed key is passed and its index is unalignable to the frame index.
|
| See Also
| -----
| DataFrame.at : Access a single value for a row/column label pair.
| DataFrame.iloc : Access group of rows and columns by integer position(s).
| DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the
|     Series/DataFrame.
| Series.loc : Access group of values using labels.
|
| Examples
| -----
| **Getting values**
|
| >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
| ...     index=['cobra', 'viper', 'sidewinder'],
| ...     columns=['max_speed', 'shield'])
| >>> df
|
|           max_speed  shield
| cobra                1      2

```



```
viper          4      5
sidewinder     7      8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed     4
shield        5
Name: viper, dtype: int64
```

List of labels. Note using ``[[]]`` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
      max_speed  shield
viper          4      5
sidewinder     7      8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra      1
viper      4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
      max_speed  shield
sidewinder     7      8
```

Alignable boolean Series:

```
>>> df.loc[pd.Series([False, True, False],
...                  index=['viper', 'sidewinder', 'cobra'])]
      max_speed  shield
sidewinder     7      8
```

```

| Index (same behavior as ``df.reindex``)
|
| >>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
|           max_speed  shield
| foo
| cobra           1      2
| viper           4      5
|
| Conditional that returns a boolean Series
|
| >>> df.loc[df['shield'] > 6]
|           max_speed  shield
| sidewinder       7      8
|
| Conditional that returns a boolean Series with column labels specified
|
| >>> df.loc[df['shield'] > 6, ['max_speed']]
|           max_speed
| sidewinder       7
|
| Callable that returns a boolean Series
|
| >>> df.loc[lambda df: df['shield'] == 8]
|           max_speed  shield
| sidewinder       7      8
|
| **Setting values**
|
| Set value for all items matching the list of labels
|
| >>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
| >>> df
|           max_speed  shield
| cobra           1      2
| viper           4     50
| sidewinder       7     50
|
| Set value for an entire row
|
| >>> df.loc['cobra'] = 10
| >>> df
|           max_speed  shield
| cobra           10     10

```

```
viper          4      50
sidewinder     7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
```

```
      max_speed  shield
cobra          30     10
viper          30     50
sidewinder     30     50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
```

```
      max_speed  shield
cobra          30     10
viper           0      0
sidewinder      0      0
```

****Getting values on a DataFrame with an index that has integer labels****

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
```

```
      max_speed  shield
7             1      2
8             4      5
9             7      8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
      max_speed  shield
7             1      2
8             4      5
9             7      8
```

****Getting values with a MultiIndex****

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [  
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),  
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),  
...     ('viper', 'mark ii'), ('viper', 'mark iii')  
... ]  
>>> index = pd.MultiIndex.from_tuples(tuples)  
>>> values = [[12, 2], [0, 4], [10, 20],  
...           [1, 4], [7, 1], [16, 36]]  
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)  
>>> df
```

		max_speed	shield
cobra	mark i	12	2
	mark ii	0	4
sidewinder	mark i	10	20
	mark ii	1	4
viper	mark ii	7	1
	mark iii	16	36

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']  
           max_speed  shield  
mark i           12      2  
mark ii          0      4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]  
max_speed    0  
shield       4  
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']  
max_speed    12  
shield       2  
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using ``[[]]`` returns a DataFrame.

```
>>> df.loc[(['cobra', 'mark ii'])]
           max_speed  shield
cobra mark ii         0      4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
           max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
           mark iii      16     36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):(('viper', 'mark ii'))]
           max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
```

Please see the `:ref:`user guide<advanced.advanced_hierarchical>`` for more details and explanations of advanced indexing.

df

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025

	0	1	2
3	1.122968	NaN	0.932057
4	-0.037637	1.100561	-0.328430
5	-0.077328	-1.032715	0.157982
6	0.363370	1.845914	-0.172841

```
df.iloc[2:, 1]= np.nan #all rows from third onwards, second column
df
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	NaN	-0.328430
5	-0.077328	NaN	0.157982
6	0.363370	NaN	-0.172841

```
df.iloc[4:, 2] = np.nan
df
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	NaN	NaN
5	-0.077328	NaN	NaN
6	0.363370	NaN	NaN

```
df.fillna(method = "ffill")
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025

	0	1	2
3	1.122968	NaN	0.932057
4	-0.037637	NaN	0.932057
5	-0.077328	NaN	0.932057
6	0.363370	NaN	0.932057

```
df.fillna({1:0.5})
```

	0	1	2
0	-1.716945	0.5	0.198477
1	2.815565	0.5	-1.749359
2	-0.073905	0.5	0.826025
3	1.122968	0.5	0.932057
4	-0.037637	0.5	NaN
5	-0.077328	0.5	NaN
6	0.363370	0.5	NaN

```
df.fillna(method='ffill', limit=2) # 2 values of third column gets filled
```

	0	1	2
0	-1.716945	NaN	0.198477
1	2.815565	NaN	-1.749359
2	-0.073905	NaN	0.826025
3	1.122968	NaN	0.932057
4	-0.037637	NaN	0.932057
5	-0.077328	NaN	0.932057
6	0.363370	NaN	NaN

imputations with fillna()

- function arguments (value, method, axis, limit)

```
# imputations with fillna
data = pd.Series([1., np.nan, 3.5, np.nan, 7])

data
```

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

```
data.fillna(data.mean())
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```


Data Transformation

Removing duplicates

```
data = pd.DataFrame({"k1": ['one', 'two']* 3 + ['two'],  
                    "k2": [1, 22, 1, 22, 3, 22, 3,]})
```

```
data
```

	k1	k2
0	one	1
1	two	22
2	one	1
3	two	22
4	one	3
5	two	22
6	two	3

```
data.duplicated()
```

```
0    False  
1    False  
2     True  
3     True  
4    False  
5     True  
6    False  
dtype: bool
```

```
data.drop_duplicates()
```

	k1	k2
0	one	1
1	two	22
4	one	3
6	two	3

```
# removing duplicates based on column
data['k3'] = range(7) # adding column
```

```
data
```

	k1	k2	k3
0	one	1	0
1	two	22	1
2	one	1	2
3	two	22	3
4	one	3	4
5	two	22	5
6	two	3	6

```
data.drop_duplicates(subset=["k1"])
```

	k1	k2	k3
0	one	1	0
1	two	22	1

Transforming data using a Function or mapping

```
data = pd.DataFrame({'food': ['poisson', 'boeuf', 'mouton', 'bacon', 'poulet'],
                    'quantité - ounces' : [4, 5, 3, 4, 2]})
```

```
data
```

	food	quantité - ounces
0	poisson	4
1	boeuf	5
2	mouton	3
3	bacon	4
4	poulet	2

```
# writing animal names in english alongside
meat_of_animal = {'poisson': 'fish',
                  'boeuf' : 'beef',
                  'mouton' : 'sheep',
                  'bacon': 'pig',
                  'poulet': 'chicken'}

data['english'] = data['food'].map(meat_of_animal)
```

```
data
```

	food	quantité - ounces	english
0	poisson	4	fish
1	boeuf	5	beef
2	mouton	3	sheep
3	bacon	4	pig
4	poulet	2	chicken

```
# creating a function for the same and using map()

def animal_english (x):
    return meat_of_animal[x]

data['food'].map(animal_english)
```

```
0      fish
1      beef
2      sheep
3       pig
4     chicken
Name: food, dtype: object
```

Replacing values

```
data
```

	food	quantité - ounces	english
0	poisson	4	fish
1	boeuf	5	beef
2	mouton	3	sheep
3	bacon	4	pig
4	poulet	2	chicken

```
data2 = pd.Series([1., - 323, -.32 , 4.]
```

```
data2
```

```
0    1.00
1   -323.00
2    -0.32
3     4.00
dtype: float64
```

```
data2.replace(-323, np.nan)
```

```
0    1.00
1     NaN
2    -0.32
3     4.00
dtype: float64
```

```
# replacing multiple values
data2.replace([-323, -.32], np.nan)
```

```
0    1.0
1     NaN
2     NaN
3     4.0
dtype: float64
```

```
# using different replacement values for different substitutes
data2.replace([-323, -.32], [np.nan, 0])

0    1.0
1    NaN
2    0.0
3    4.0
dtype: float64
```

```
# argument passed can also be a dictionary
data2.replace({-323: np.nan, -.32:55555})

0         1.0
1         NaN
2    55555.0
3         4.0
dtype: float64
```

Renaming Axis indexes

data

	food	quantité - ounces	english
0	poisson	4	fish
1	boeuf	5	beef
2	mouton	3	sheep
3	bacon	4	pig
4	poulet	2	chicken

```
# transforming first four letters of column
def transform(x):
    return x[:5].upper()

data.columns.map(transform)

Index(['FOOD', 'QUANT', 'ENGLI'], dtype='object')
```

```
# changing the titles of the DataFrame
data.columns = data.columns.map(transform)
```

```
data
```

	FOOD	QUANT	ENGLI
0	poisson	4	fish
1	boeuf	5	beef
2	mouton	3	sheep
3	bacon	4	pig
4	poulet	2	chicken

```
# tranformed version of dataset without modifying the original
```

```
data.rename(columns = str.lower)
```

	food	quant	engli
0	poisson	4	fish
1	boeuf	5	beef
2	mouton	3	sheep
3	bacon	4	pig
4	poulet	2	chicken

Discretization and Binning

- pandas.cut
- pandas.value_counts

```
ages = [21, 33, 12, 33, 32, 21, 44, 55]
```

```
bins = [12, 30, 40, 50, 60, 70, 100]
```

```
age_categories = pd.cut(ages, bins)
```

```
age_categories
```

```
[(12.0, 30.0], (30.0, 40.0], NaN, (30.0, 40.0], (30.0, 40.0], (12.0, 30.0], (40.0, 50.0], (50.0, 60.0], (60.0, 70.0], (70.0, 100.0]]
Categories (6, interval[int64, right]): [(12, 30] < (30, 40] < (40, 50] < (50, 60] < (60, 70] < (70, 100]]
```

```
pd.value_counts(age_categories)
```

```
(30, 40]      3
(12, 30]      2
(40, 50]      1
(50, 60]      1
(60, 70]      0
(70, 100]     0
Name: count, dtype: int64
```

```
help(pd.value_counts)
```

Help on function value_counts in module pandas.core.algorithms:

```
value_counts(values, sort: 'bool' = True, ascending: 'bool' = False, normalize: 'bool' = False)
    Compute a histogram of the counts of non-null values.
```

Parameters

```
values : ndarray (1-d)
sort : bool, default True
    Sort by values
ascending : bool, default False
    Sort in ascending order
normalize: bool, default False
    If True then compute a relative histogram
bins : integer, optional
    Rather than count values, group them into half-open bins,
    convenience for pd.cut, only works with numeric data
dropna : bool, default True
    Don't include counts of NaN
```

Returns

Series

```

# parnethesis is towards open side

# changing side

pd.cut(ages, bins, right= False)

[[12, 30), [30, 40), [12, 30), [30, 40), [30, 40), [12, 30), [40, 50), [50, 60)]
Categories (6, interval[int64, left]): [[12, 30) < [30, 40) < [40, 50) < [50, 60) < [60, 70)

# changing labels

group_names = ['ados','youth', 'youngsters', 'midaged', 'senior', 'retired', ]

pd.cut(ages, bins, labels = group_names)

['ados', 'youth', NaN, 'youth', 'youth', 'ados', 'youngsters', 'midaged']
Categories (6, object): ['ados' < 'youth' < 'youngsters' < 'midaged' < 'senior' < 'retired']

data3 = np.random.uniform (size = 20)

categories = pd.cut(data3, 4, precision=2) # 4 is the number of bins

# precision limits decmical point to two decimal places

pd.value_counts(categories)

(0.011, 0.25]    8
(0.49, 0.73]    6
(0.73, 0.97]    5
(0.25, 0.49]    1
Name: count, dtype: int64

# for equal sized bins
categories_1 = pd.qcut(data3, 4, precision = 2)

pd.value_counts(categories_1)

```



```
(0.002, 0.14]    5
(0.14, 0.51]    5
(0.51, 0.69]    5
(0.69, 0.97]    5
Name: count, dtype: int64
```

Detecting and Filtering Outliers

```
data4 = pd.DataFrame(np.random.standard_normal((1000, 4)))

data4.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.016561	-0.031729	0.027078	0.064505
std	1.007664	1.047383	1.001156	0.993706
min	-3.592826	-3.066364	-3.149016	-3.644190
25%	-0.627244	-0.749349	-0.603733	-0.592056
50%	0.046336	-0.056226	0.010209	0.030974
75%	0.686210	0.658083	0.737684	0.765335
max	3.262253	3.918947	2.981393	3.029627

```
# find values of columns exceeding 3 in absolute value

col = data4[2]

col[col.abs() > 3]
```

```
614    -3.149016
701    -3.093409
Name: 2, dtype: float64
```

```
# selecting all the columns

data4[(data4.abs() > 3).any(axis = 'columns')]
```

	0	1	2	3
78	-3.591639	-0.652692	2.002021	-0.233262
105	1.620532	3.918947	-0.014565	0.483555
178	-0.642515	-3.020820	-1.899087	-0.244327
184	-3.592826	-1.979726	0.371343	1.022697
240	-0.331648	3.687370	-0.459598	0.994123
267	1.236309	-3.066364	-0.132627	0.551233
274	-1.350742	3.505783	0.908839	-0.219144
306	-0.419746	3.068152	-0.085927	1.446228
346	0.174638	3.035741	1.034076	1.208338
414	1.224208	3.060689	0.439984	-1.323200
433	1.493799	-0.528079	-0.241609	3.029627
543	3.262253	-0.713190	0.583197	1.791658
614	0.535075	-1.450999	-3.149016	1.604812
617	0.240876	-0.628409	0.151670	-3.455415
701	-0.618249	0.618510	-3.093409	1.118688
817	-0.376220	-0.653628	-1.408788	-3.644190

```
# code to cap values outside the interval -3 to 3
```

```
data4[data4.abs() > 3] = np.sign(data4) * 3
```

```
data4.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.017483	-0.033918	0.027320	0.065575
std	1.002943	1.039751	1.000406	0.989902
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.627244	-0.749349	-0.603733	-0.592056
50%	0.046336	-0.056226	0.010209	0.030974
75%	0.686210	0.658083	0.737684	0.765335
max	3.000000	3.000000	2.981393	3.000000

```
# np.sign(data4) produces 1 and -1 values
```

```
np.sign(data4).head()
```

	0	1	2	3
0	-1.0	1.0	-1.0	1.0
1	-1.0	-1.0	-1.0	1.0
2	1.0	1.0	-1.0	-1.0
3	-1.0	1.0	-1.0	-1.0
4	-1.0	-1.0	1.0	-1.0

Permutation and Random Sampling

```
df2 = pd.DataFrame (np.arange(5*7).reshape(5,7))
df2
```

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
sampler = np.random.permutation(5)
sampler
```

```
array([4, 1, 2, 0, 3])
```

```
# take function or 'iloc' based indexing
df2.take(sampler)
```

	0	1	2	3	4	5	6
4	28	29	30	31	32	33	34
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27

```
df2.iloc[sampler]
```

	0	1	2	3	4	5	6
4	28	29	30	31	32	33	34
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27

```
# selecting random subset without replacement
```

```
df2.sample(n=3)
```

	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
4	28	29	30	31	32	33	34
3	21	22	23	24	25	26	27

```
# with replacement
```

```
df2.sample(n =4, replace = True)
```

	0	1	2	3	4	5	6
4	28	29	30	31	32	33	34
4	28	29	30	31	32	33	34
2	14	15	16	17	18	19	20
1	7	8	9	10	11	12	13

Computing Indicator/Dummy Variables

- `pandas.get_dummies` function
- used for statistical modelling or machine learning applications
- `DataFrame.join()` method
- combine `get_dummies()` with `pandas.cut()`

```
df_dict = pd.DataFrame({'key': ['a', 'b', 'c', 'd', 'e'],
                        'data1': range(5)})
```

```
df_dict
```

	key	data1
0	a	0
1	b	1
2	c	2
3	d	3
4	e	4

```
pd.get_dummies(df_dict['key'])
```

	a	b	c	d	e
0	True	False	False	False	False
1	False	True	False	False	False
2	False	False	True	False	False
3	False	False	False	True	False
4	False	False	False	False	True

```
dummies = pd.get_dummies(df_dict['key'], prefix = 'key')
```

```
df_with_dummy = df_dict[['data1']].join(dummies)
```

```
df_with_dummy
```

	data1	key_a	key_b	key_c	key_d	key_e
0	0	True	False	False	False	False
1	1	False	True	False	False	False
2	2	False	False	True	False	False
3	3	False	False	False	True	False
4	4	False	False	False	False	True

```
# combine pd.get_dummies() with pd.cut()
```

```

np.random.seed(123)

values = np.random.uniform(size = 10)

values

array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
       0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752])

bins = [0, 0.2, 0.4, 0.6, 0.8, 1.0]

pd.get_dummies(pd.cut(values, bins))

```

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	False	False	False	True	False
1	False	True	False	False	False
2	False	True	False	False	False
3	False	False	True	False	False
4	False	False	False	True	False
5	False	False	True	False	False
6	False	False	False	False	True
7	False	False	False	True	False
8	False	False	True	False	False
9	False	True	False	False	False

Extension data types

```

s = pd.Series([1, 2, 4, None])

s

0    1.0
1    2.0
2    4.0
3    NaN
dtype: float64

```

```
s.dtype
```

```
dtype('float64')
```

```
s.isna()
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

```
s_int = pd.Series(['one', 'two', 'three', None, 'four'],
                  dtype = pd.StringDtype())
```

```
s_int
```

```
0     one
1     two
2    three
3    <NA>
4     four
dtype: string
```

```
df3 = pd.DataFrame({'A': [1,2, None, 4],
                    'B': ['one', 'two', 'three', None],
                    'C': [False, True, None, True]})
```

```
df3
```

	A	B	C
0	1.0	one	False
1	2.0	two	True
2	NaN	three	None
3	4.0	None	True

```
# changing to their respective categories
df3['A'] = df3['A'].astype('Int64')
df3['B'] = df3['B'].astype('string')
df3['C'] = df3['C'].astype('boolean')
```

```
df3
```

	A	B	C
0	1	one	False
1	2	two	True
2	<NA>	three	<NA>
3	4	<NA>	True

String manipulation

```
val = 'a, b, kuch v'

val.split(',')
```

```
['a', ' b', ' kuch v']
```

```
# removing white space
remove = [x.strip() for x in val.split(',')]

remove
```

```
['a', 'b', 'kuch v']
```

```
# two colon concatenation
first, second, third = remove
first + '::' + second + '::' + third
```

```
'a::b::kuch v'
```



```
'::'.join(val)
'::'.join(remove)
```

```
'a::b::kuch v'
```

```
# in keyword
'kuch v' in val
```

True

```
val.index(',')
```

1

```
val.find(':')
```

```
# remark- index doesnt give an error if the value is not found
```

-1

Regular expressions

- re saves CPU cycles

```
import re
```

```
text = 'bla bla bla ta ta ta bar\t baz aja \t nai fer'
```

```
text
```

```
'bla bla bla ta ta ta bar\t baz aja \t nai fer'
```

```
re.split(r"\s+", text)
```

```
['bla', 'bla', 'bla', 'ta', 'ta', 'ta', 'bar', 'baz', 'aja', 'nai', 'fer']
```

```
# doing it with re
withre = re.compile(r"\s+")

withre.split(text)
```

```
['bla', 'bla', 'bla', 'ta', 'ta', 'ta', 'bar', 'baz', 'aja', 'nai', 'fer']
```

```
# finding pattern

withre.findall(text)
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', '\t ', ' ', ' \t ', ' ']
```

```
text2 = """
Kunal Khuranasoilpau@gmail.com
Sonakshi mehra.43@gmail.com
Karan gill007@outlook.ca
Smriti cuti3_43@ourkut.ca
"""
```

```
pattern = r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
```

```
# using re.IGNORECASE
final_text = re.compile(pattern, flags=re.IGNORECASE)

final_text.findall(text2)
```

```
['Khuranasoilpau@gmail.com',
'mehra.43@gmail.com',
'gill007@outlook.ca',
'cuti3_43@ourkut.ca']
```

```
print(final_text.sub('REDACTED', text2))
```

```
Kunal REDACTED
Sonakshi REDACTED
Karan REDACTED
Smriti REDACTED
```

String functions in pandas

- convert dictionary to Series with pandas
- [pandas string methods](#)

```
data = {'Kunal': 'Khuranasoilpau@gmail.com', 'Robin': 'aryan_robin@yahoo.com',  
        'Deepika': 'padukone.deepi@outlook.ca', 'Ranbir' : "singh.cool7@yahoo.com",  
        'kabir': np.nan}
```

```
data = pd.Series(data)  
data
```

```
Kunal      Khuranasoilpau@gmail.com  
Robin      aryan_robin@yahoo.com  
Deepika    padukone.deepi@outlook.ca  
Ranbir     singh.cool7@yahoo.com  
kabir      NaN  
dtype: object
```

```
data.str.findall(pattern, flags=re.IGNORECASE)
```

```
Kunal      [Khuranasoilpau@gmail.com]  
Robin      [aryan_robin@yahoo.com]  
Deepika    [padukone.deepi@outlook.ca]  
Ranbir     [singh.cool7@yahoo.com]  
kabir      NaN  
dtype: object
```

```
# slice  
data.str[:7]
```

```
Kunal      Khurana  
Robin      aryan_r  
Deepika    padukon  
Ranbir     singh.c  
kabir      NaN  
dtype: object
```

Categorical data

```
values = pd.Series(['apple', 'orange', 'apple', 'mango']* 2)
```

```
values
```

```
0    apple
1    orange
2    apple
3    mango
4    apple
5    orange
6    apple
7    mango
dtype: object
```

```
pd.unique(values)
```

```
array(['apple', 'orange', 'mango'], dtype=object)
```

```
values2 = pd.Series([0,1,0, 0] * 2)
```

```
dim = pd.Series(['apple', 'orange'])
```

```
values2
```

```
0    0
1    1
2    0
3    0
4    0
5    1
6    0
7    0
dtype: int64
```

```
dim
```

```
0    apple
1    orange
dtype: object
```

```
# take method to restore original set of strings
dim.take(values2)
```

```
0    apple
1    orange
0    apple
0    apple
0    apple
1    orange
0    apple
0    apple
dtype: object
```

Categorical Extension type in pandas

```
fruits = ['apple', 'orange', 'apple', 'papaya'] * 2

N = len(fruits)

rng = np.random.default_rng(seed = 123)

df = pd.DataFrame({'fruit': fruits,
                   'basket_id': np.arange(N),
                   'count': rng.integers(3, 15, size = N),
                   'weight': rng.uniform(0, 4, size = N)},
                  columns = ['basket_id', 'fruit', 'count', 'weight'])
```

df

	basket_id	fruit	count	weight
0	0	apple	9	0.694527
1	1	orange	3	1.250969
2	2	apple	7	0.057898
3	3	papaya	9	0.130208

	basket_id	fruit	count	weight
4	4	apple	14	1.986807
5	5	orange	5	1.873250
6	6	apple	10	0.510761
7	7	papaya	12	1.030250

```
# converting df to categorical
fruit_cat = df['fruit'].astype('category')
fruit_cat
```

```
0    apple
1    orange
2    apple
3    papaya
4    apple
5    orange
6    apple
7    papaya
```

```
Name: fruit, dtype: category
Categories (3, object): ['apple', 'orange', 'papaya']
```

```
c = fruit_cat.array
```

```
type(c)
```

```
pandas.core.arrays.categorical.Categorical
```

```
c.categories
```

```
Index(['apple', 'orange', 'papaya'], dtype='object')
```

```
c.codes
```

```
array([0, 1, 0, 2, 0, 1, 0, 2], dtype=int8)
```

```
# how to get mapping between code and categories
dict(enumerate(c.categories))

{0: 'apple', 1: 'orange', 2: 'papaya'}

pd.unique(values)

array([0, 1], dtype=int64)
```

Categorical data

```
values3 = pd.Series(['apple', 'orange', 'apple',
                    'apple'] * 2)

values3

0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
dtype: object

pd.unique(values3)

array(['apple', 'orange'], dtype=object)

pd.value_counts(values3)

apple    6
orange   2
Name: count, dtype: int64
```

```
categories = ['foo', 'bar', 'baz']

codes = [0, 1, 2, 0, 2, 0, 1]

my_cats2 = pd.Categorical.from_codes(codes, categories)

my_cats2
```

```
['foo', 'bar', 'baz', 'foo', 'baz', 'foo', 'bar']
Categories (3, object): ['foo', 'bar', 'baz']
```

```
ordered_cat = pd.Categorical.from_codes(codes, categories,
                                         ordered = True)

ordered_cat
```

```
['foo', 'bar', 'baz', 'foo', 'baz', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

Computations with categoricals

```
rng = np.random.default_rng(seed = 123)

draws = rng.standard_normal(1000)

draws[:5]
```

```
array([-0.98912135, -0.36778665,  1.28792526,  0.19397442,  0.9202309 ])
```

```
bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

bins
```

```
['Q1', 'Q2', 'Q4', 'Q3', 'Q4', ..., 'Q1', 'Q3', 'Q3', 'Q1', 'Q3']
Length: 1000
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```


using groupby for summary statistics

```
bins = pd.Series(bins, name= 'quartile')

results = (pd.Series(draws)
           .groupby(bins)
           .agg(['count', 'min', 'max'])
           .reset_index())

results
```

	quartile	count	min	max
0	Q1	250	-3.298281	-0.626241
1	Q2	250	-0.622043	0.040753
2	Q3	250	0.043084	0.736086
3	Q4	250	0.738013	3.058244

Better performance with categoricals

```
N = 10_000_000

labels = pd.Series(['foo', 'bar', 'baz', 'qux']) * (N // 4)

# convert labels to categoricals
categories = labels.astype('category')

# memory use
labels.memory_usage(deep = True)
```

30000356

```
categories.memory_usage(deep = True)
```

30000532

Categorical Methods

- [list of categorical methods](#)

```
s = pd.Series(['a', 'b', 'c', 'd'] * 2)
```

```
cat_s = s.astype('category')
```

```
cat_s
```

```
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']
```

```
# using special accessor attribute cat
```

```
cat_s.cat.codes
```

```
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8
```

```
actual_categories = ['a', 'b', 'c', 'd', 'e']
```

```
cat_s2 = cat_s.cat.set_categories(actual_categories)
```

```
cat_s2
```

```
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
```

```
dtype: category
```

```
Categories (5, object): ['a', 'b', 'c', 'd', 'e']
```

```
cat_s.value_counts()
```

```
a    2
b    2
c    2
d    2
```

```
Name: count, dtype: int64
```

```
cat_s2.value_counts()
```

```
a    2
b    2
c    2
d    2
e    0
```

```
Name: count, dtype: int64
```

```
cat_s3 = cat_s[cat_s.isin(['a', 'b'])]
```

```
cat_s3
```

```
0    a
1    b
4    a
```

```
5    b
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']
```

```
# removing unused categories
cat_s3.cat.remove_unused_categories()
```

```
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): ['a', 'b']
```

Creating dummy variables for modelling

```
cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2,
                  dtype= 'category')
```

```
pd.get_dummies(cat_s)
```

	a	b	c	d
0	True	False	False	False
1	False	True	False	False
2	False	False	True	False
3	False	False	False	True
4	True	False	False	False
5	False	True	False	False
6	False	False	True	False
7	False	False	False	True