# Series

**The first main data type we will learn about for pandas is the Series data type. Let's import Pandas and explore the Series object.**

**A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.**

**Let's explore this concept through some examples:**

## Imports

In [12]:

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

## Creating a Series from Python Objects

In [13]:

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

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

class Series(pandas.core.base.IndexOpsMixin, pandas.core.generic.NDFrame)
 |  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.
 |
 |      .. versionchanged :: 0.23.0
 |         If data is a dict, argument order is maintained for Python 3.6
 |         and later.
 |
 |  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 both a dict and index
 |      sequence are used, the index will override the keys found in the
```

```
|      dict.
|  dtype : str, numpy.dtype, or ExtensionDtype, optional
|      dtype for the output Series. If not specified, this will be
|      inferred from `data`.
|      See the :ref:`user guide <basics.dtypes>` for more usages.
|  copy : bool, default False
|      Copy input data.
|
|  Method resolution order:
|      Series
|      pandas.core.base.IndexOpsMixin
|      pandas.core.generic.NDFrame
|      pandas.core.base.PandasObject
|      pandas.core.base.StringMixin
|      pandas.core.accessor.DirNamesMixin
|      pandas.core.base.SelectionMixin
|      builtins.object
|
|  Methods defined here:
|
|  __add__(left, right)
|
|  __and__(self, other)
|
|  __array__(self, dtype=None)
|      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.ndarary`
|          with the specified `dtype`.
|
|      See Also
|      --------
|      pandas.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', freq='D'),
|             Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
|            dtype=object)
|
|      Or the values may be localized to UTC and the tzinfo discared with
|      ``dtype='datetime64[ns]'``
|
|      >>> np.asarray(tzser, dtype="datetime64[ns]")  # doctest: +ELLIPSIS
|      array(['1999-12-31T23:00:00.000000000', ...],
|            dtype='datetime64[ns]')
|
|  __array_prepare__(self, result, context=None)
|      Gets called prior to a ufunc.
```

```
 |
 |  __array_wrap__(self, result, context=None)
 |      Gets called after a ufunc.
 |
 |  __div__ = __truediv__(left, right)
 |
 |  __divmod__(left, right)
 |
 |  __eq__(self, other, axis=None)
 |
 |  __float__(self)
 |
 |  __floordiv__(left, right)
 |
 |  __ge__(self, other, axis=None)
 |
 |  __getitem__(self, key)
 |
 |  __gt__(self, other, axis=None)
 |
 |  __iadd__(self, other)
 |
 |  __iand__(self, other)
 |
 |  __ifloordiv__(self, other)
 |
 |  __imod__(self, other)
 |
 |  __imul__(self, other)
 |
 |  __init__(self, data=None, index=None, dtype=None, name=None, copy=False, fastpath=Fal
se)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __int__(self)
 |
 |  __ior__(self, other)
 |
 |  __ipow__(self, other)
 |
 |  __isub__(self, other)
 |
 |  __itruediv__(self, other)
 |
 |  __ixor__(self, other)
 |
 |  __le__(self, other, axis=None)
 |
 |  __len__(self)
 |      Return the length of the Series.
 |
 |  __long__ = __int__(self)
 |
 |  __lt__(self, other, axis=None)
 |
 |  __matmul__(self, other)
 |      Matrix multiplication using binary `@` operator in Python>=3.5.
 |
 |  __mod__(left, right)
 |
 |  __mul__(left, right)
 |
 |  __ne__(self, other, axis=None)
 |
 |  __or__(self, other)
 |
 |  __pow__(left, right)
 |
 |  __radd__(left, right)
 |
 |  __rand__(self, other)
 |
```

```
|  __rdiv__ = __rtruediv__(left, right)
|
|  __rdivmod__(left, right)
|
|  __rfloordiv__(left, right)
|
|  __rmatmul__(self, other)
|      Matrix multiplication using binary `@` operator in Python>=3.5.
|
|  __rmod__(left, right)
|
|  __rmul__(left, right)
|
|  __ror__(self, other)
|
|  __rpow__(left, right)
|
|  __rsub__(left, right)
|
|  __rtruediv__(left, right)
|
|  __rxor__(self, other)
|
|  __setitem__(self, key, value)
|
|  __sub__(left, right)
|
|  __truediv__(left, right)
|
|  __unicode__(self)
|      Return a string representation for a particular DataFrame.
|
|      Invoked by unicode(df) in py2 only. Yields a Unicode String in both
|      py2/py3.
|
|  __xor__(self, other)
|
|  add(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.radd
|
|      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, axis=0, *args, **kwargs)
|
|  aggregate(self, func, axis=0, *args, **kwargs)
|      Aggregate using one or more operations over the specified axis.
|
|      .. versionadded:: 0.20.0
|
|      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'}
|              Parameter needed for compatibility with DataFrame.
|      *args
|          Positional arguments to pass to `func`.
|      **kwargs
|          Keyword arguments to pass to `func`.
|
|      Returns
|      -------
|      DataFrame, Series or scalar
|          if DataFrame.agg is called with a single function, returns a Series
|          if DataFrame.agg is called with several functions, returns a DataFrame
|          if Series.agg is called with single function, returns a scalar
|          if Series.agg is called with several functions, returns a Series
|
|
|      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.
|
|      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, join='outer', axis=None, level=None, copy=True, fill_value=None, m
ethod=None, limit=None, fill_axis=0, broadcast_axis=None)
 |      Align two objects on their axes with the
 |      specified join method 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 : boolean, 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
 |      -------
 |      (left, right) : (Series, type of other)
 |          Aligned objects
 |
 |  all(self, axis=0, bool_only=None, skipna=True, level=None, **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.
 |
 |          * 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.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      **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([]).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 column-wise values all return True.
|
|      >>> df.all()
|      col1     True
|      col2    False
|      dtype: bool
|
|      Specify ``axis='columns'`` to check if row-wise values 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=0, bool_only=None, skipna=True, level=None, **kwargs)
|      Return whether any element is True, potentially over an axis.
|
|      Returns False unless there 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.
|
|           * 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.
|       level : int or level name, default None
|           If the axis is a MultiIndex (hierarchical), count along a
|           particular level, collapsing into a scalar.
|       **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([]).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
|             A  B
|      0   True  1
|      1  False  2
|
|      >>> df.any(axis='columns')
|      0    True
|      1    True
|      dtype: bool
|
|      >>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
|      >>> df
|             A  B
|      0   True  1
|      1  False  0
|
|      >>> df.any(axis='columns')
|      0    True
|      1    False
|      dtype: bool
|
|      Aggregating over the entire DataFrame with ``axis=None``.
|
|      >>> df.any(axis=None)
|      True
|
|      `any` for an empty DataFrame is an empty Series.
|
|      >>> pd.DataFrame([]).any()
|      Series([], dtype: bool)
|
|  append(self, to_append, ignore_index=False, verify_integrity=False)
|      Concatenate two or more Series.
|
|      Parameters
|      ----------
|      to_append : Series or list/tuple of Series
|      ignore_index : boolean, default False
|          If True, do not use the index labels.
|
|          .. versionadded:: 0.19.0
|
|      verify_integrity : boolean, default False
|          If True, raise Exception on creating index with duplicates
|
|      Returns
|      -------
|      appended : Series
|
|      See Also
|      --------
|      concat : General function to concatenate DataFrame, Series
|          or Panel objects.
|
|      Notes
|      -----
|      Iteratively appending to a Series can be more computationally intensive
|      than a single concatenate. A better solution is to append values to a
|      list and then concatenate the list with the original Series all at
|      once.
|
|      Examples
|      --------
|      >>> s1 = pd.Series([1, 2, 3])
|      >>> s2 = pd.Series([4, 5, 6])
```

```
|      >>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
|      >>> s1.append(s2)
|      0    1
|      1    2
|      2    3
|      0    4
|      1    5
|      2    6
|      dtype: int64
|
|      >>> s1.append(s3)
|      0    1
|      1    2
|      2    3
|      3    4
|      4    5
|      5    6
|      dtype: int64
|
|      With `ignore_index` set to True:
|
|      >>> s1.append(s2, ignore_index=True)
|      0    1
|      1    2
|      2    3
|      3    4
|      4    5
|      5    6
|      dtype: int64
|
|      With `verify_integrity` set to True:
|
|      >>> s1.append(s2, verify_integrity=True)
|      Traceback (most recent call last):
|      ...
|      ValueError: Indexes have overlapping values: [0, 1, 2]
|
|  apply(self, func, convert_dtype=True, args=(), **kwds)
|      Invoke function on values of Series.
|
|      Can be ufunc (a NumPy function that applies to the entire Series)
|      or a Python function that only works on single values.
|
|      Parameters
|      ----------
|      func : function
|          Python function or NumPy ufunc to apply.
|      convert_dtype : bool, default True
|          Try to find better dtype for elementwise function results. If
|          False, leave as dtype=object.
|      args : tuple
|          Positional arguments passed to func after the series value.
|      **kwds
|          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.
|
|      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
|
|  argmax = idxmax(self, axis=0, skipna=True, *args, **kwargs)
|      Return the row label of the maximum value.
|
|      .. deprecated:: 0.21.0
|
|      The current behaviour of 'Series.argmax' is deprecated, use 'idxmax'
|      instead.
|      The behavior of 'argmax' will be corrected to return the positional
|      maximum in the future. For now, use 'series.values.argmax' or
|      'np.argmax(np.array(values))' to get the position of the maximum
|      row.
```

```
|
|      If multiple values equal the maximum, the first row label with that
|      value is returned.
|
|      Parameters
|      ----------
|      skipna : boolean, default True
|          Exclude NA/null values. If the entire Series is NA, the result
|          will be NA.
|      axis : int, default 0
|          For compatibility with DataFrame.idxmax. Redundant for application
|          on Series.
|      *args, **kwargs
|          Additional keywords have no effect but might be accepted
|          for compatibility with NumPy.
|
|      Returns
|      -------
|      idxmax : Index of maximum of values.
|
|      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
|
|  argmin = idxmin(self, axis=0, skipna=True, *args, **kwargs)
|      Return the row label of the minimum value.
|
|      .. deprecated:: 0.21.0
|
|      The current behaviour of 'Series.argmin' is deprecated, use 'idxmin'
|      instead.
|      The behavior of 'argmin' will be corrected to return the positional
|      minimum in the future. For now, use 'series.values.argmin' or
|      'np.argmin(np.array(values))' to get the position of the minimum
|      row.
```

```
|
|      If multiple values equal the minimum, the first row label with that
|      value is returned.
|
|      Parameters
|      ----------
|      skipna : boolean, default True
|          Exclude NA/null values. If the entire Series is NA, the result
|          will be NA.
|      axis : int, default 0
|          For compatibility with DataFrame.idxmin. Redundant for application
|          on Series.
|      *args, **kwargs
|          Additional keywords have no effect but might be accepted
|          for compatibility with NumPy.
|
|      Returns
|      -------
|      idxmin : Index of minimum of values.
|
|      Raises
|      ------
|      ValueError
|          If the Series is empty.
|
|      See Also
|      --------
|      numpy.argmin : Return indices of the minimum values
|          along the given axis.
|      DataFrame.idxmin : Return index of first occurrence of minimum
|          over requested axis.
|      Series.idxmax : Return index *label* of the first occurrence
|          of maximum of values.
|
|      Notes
|      -----
|      This method is the Series version of ``ndarray.argmin``. This method
|      returns the label of the minimum, while ``ndarray.argmin`` returns
|      the position. To get the position, use ``series.values.argmin()``.
|
|      Examples
|      --------
|      >>> s = pd.Series(data=[1, None, 4, 1],
|      ...                index=['A' ,'B' ,'C' ,'D'])
|      >>> s
|      A    1.0
|      B    NaN
|      C    4.0
|      D    1.0
|      dtype: float64
|
|      >>> s.idxmin()
|      'A'
|
|      If `skipna` is False and there is an NA value in the data,
|      the function returns ``nan``.
|
|      >>> s.idxmin(skipna=False)
|      nan
|
|  argsort(self, axis=0, kind='quicksort', order=None)
|      Overrides ndarray.argsort. Argsorts the value, omitting NA/null values,
|      and places the result in the same locations as the non-NA values.
|
|      Parameters
|      ----------
|      axis : int
|          Has no effect but is accepted for compatibility with numpy.
|      kind : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'
|          Choice of sorting algorithm. See np.sort for more
|          information. 'mergesort' is the only stable algorithm
|      order : None
```

```
|          Has no effect but is accepted for compatibility with numpy.
|
|      Returns
|      -------
|      argsorted : Series, with -1 indicated where nan values are present
|
|      See Also
|      --------
|      numpy.ndarray.argsort
|
|  autocorr(self, lag=1)
|      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=True)
|      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
|          Left boundary.
|      right : scalar
|          Right boundary.
|      inclusive : bool, default True
|          Include boundaries.
|
|      Returns
|      -------
|      Series
|          Each element will be a boolean.
|
```

```
|
|      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 ``False`` boundary values are excluded:
|
|      >>> s.between(1, 4, inclusive=False)
|      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
|
|  combine(self, other, func, fill_value=None)
|      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)
|      Combine Series values, choosing the calling Series's values first.
|
|      Parameters
|      ----------
|      other : Series
|          The value(s) to be combined with the `Series`.
|
|      Returns
|      -------
|      Series
|          The result of combining the Series with the other object.
|
|      See Also
|      --------
|      Series.combine : Perform elementwise operation on two Series
|          using a given function.
|
|      Notes
|      -----
|      Result index will be the union of the two indexes.
|
|      Examples
|      --------
|      >>> s1 = pd.Series([1, np.nan])
|      >>> s2 = pd.Series([3, 4])
|      >>> s1.combine_first(s2)
|      0    1.0
|      1    4.0
|      dtype: float64
|
|  compound(self, axis=None, skipna=None, level=None)
|      Return the compound percentage of the values for the requested axis.
|
```

```
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      compounded : scalar or Series (if level specified)
|
|  compress(self, condition, *args, **kwargs)
|      Return selected slices of an array along given axis as a Series.
|
|      .. deprecated:: 0.24.0
|
|      See Also
|      --------
|      numpy.ndarray.compress
|
|  corr(self, other, method='pearson', min_periods=None)
|      Compute correlation with `other` Series, excluding missing values.
|
|      Parameters
|      ----------
|      other : Series
|      method : {'pearson', 'kendall', 'spearman'} or callable
|          * pearson : standard correlation coefficient
|          * kendall : Kendall Tau correlation coefficient
|          * spearman : Spearman rank correlation
|          * callable: callable with input two 1d ndarray
|              and returning a float
|              .. versionadded:: 0.24.0
|
|      min_periods : int, optional
|          Minimum number of observations needed to have a valid result
|
|      Returns
|      -------
|      correlation : float
|
|      Examples
|      --------
|      >>> histogram_intersection = lambda a, b: np.minimum(a, b
|      ... ).sum().round(decimals=1)
|      >>> s1 = pd.Series([.2, .0, .6, .2])
|      >>> s2 = pd.Series([.3, .6, .0, .1])
|      >>> s1.corr(s2, method=histogram_intersection)
|      0.3
|
|  count(self, level=None)
|      Return number of non-NA/null observations in the Series.
|
|      Parameters
|      ----------
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a smaller Series
|
|      Returns
|      -------
|      nobs : int or Series (if level specified)
|
|  cov(self, other, min_periods=None)
```

```
|      Compute covariance with Series, excluding missing values.
|
|      Parameters
|      ----------
|      other : Series
|      min_periods : int, optional
|          Minimum number of observations needed to have a valid result
|
|      Returns
|      -------
|      covariance : float
|
|      Normalized by N-1 (unbiased estimator).
|
|  cummax(self, axis=None, skipna=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'.
|      skipna : boolean, 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
|      -------
|      cummax : scalar or Series
|
|      See Also
|      --------
|      core.window.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=None, skipna=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'.
|      skipna : boolean, 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
|      -------
|      cummin : scalar or Series
|
|      See Also
|      --------
|      core.window.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=None, skipna=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'.
|      skipna : boolean, 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
    -------
    cumprod : scalar or Series

    See Also
    --------
    core.window.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=None, skipna=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'.
|      skipna : boolean, 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
|      -------
|      cumsum : scalar or Series
|
|      See Also
|      --------
|      core.window.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=1)
|      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
|      -------
|      diffed : 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.
|
|      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
 |
 |  div = truediv(self, other, level=None, fill_value=None, axis=0)
 |
 |  divide = truediv(self, other, level=None, fill_value=None, axis=0)
 |
 |  divmod(self, other, level=None, fill_value=None, axis=0)
 |      Integer division and modulo of series and other, element-wise (binary operator `d
ivmod`).
 |
 |      Equivalent to ``series divmod other``, but with support to substitute a fill_valu
e for
 |      missing data in one of the inputs.
 |
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.rdivmod
 |
 |      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
|
|  dot(self, other)
|      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=None, axis=0, index=None, columns=None, level=None, inplace=False,
errors='raise')
|      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, default 0
|          Redundant for application on Series.
|      index, columns : None
|          Redundant for application on Series, but index can be used instead
|          of labels.
|
```

```
|          .. versionadded:: 0.21.0
|      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
|      -------
|      dropped : pandas.Series
|
|      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='first', inplace=False)
|      Return Series with duplicate values removed.
|
|      Parameters
```

```
|      ----------
|      keep : {'first', 'last', ``False``}, default 'first'
|          - 'first' : Drop duplicates except for the first occurrence.
|          - 'last' : Drop duplicates except for the last occurrence.
|          - ``False`` : Drop all duplicates.
|      inplace : boolean, default ``False``
|          If ``True``, performs operation inplace and returns None.
|
|      Returns
|      -------
|      deduplicated : Series
|
|      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.
|
|      Examples
|      --------
|      Generate an 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. Setting the value of 'inplace' to ``True`` performs
|      the operation inplace and returns ``None``.
|
|      >>> s.drop_duplicates(keep=False, inplace=True)
|      >>> s
|      1       cow
|      3    beetle
|      5     hippo
|      Name: animal, dtype: object
|
|  dropna(self, axis=0, inplace=False, **kwargs)
|      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'}, default 0
|          There is only one axis to drop values from.
|      inplace : bool, default False
|          If True, do operation inplace and return None.
|      **kwargs
|          Not in use.
|
|      Returns
|      -------
|      Series
|          Series with NA entries dropped from it.
|
|      See Also
|      --------
|      Series.isna: Indicate missing values.
|      Series.notna : Indicate existing (non-missing) values.
|      Series.fillna : Replace missing values.
|      DataFrame.dropna : Drop rows or columns which contain NA values.
|      Index.dropna : Drop missing indices.
|
|      Examples
|      --------
|      >>> ser = pd.Series([1., 2., np.nan])
|      >>> ser
|      0    1.0
|      1    2.0
|      2    NaN
|      dtype: float64
|
|      Drop NA values from a Series.
|
|      >>> ser.dropna()
|      0    1.0
|      1    2.0
|      dtype: float64
|
|      Keep the Series with valid entries in the same variable.
|
|      >>> ser.dropna(inplace=True)
|      >>> ser
|      0    1.0
|      1    2.0
|      dtype: float64
|
|      Empty strings are not considered NA values. ``None`` is considered an
|      NA value.
|
|      >>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
|      >>> ser
|      0       NaN
|      1         2
|      2       NaT
|      3
|      4      None
|      5    I stay
|      dtype: object
|      >>> ser.dropna()
|      1         2
|      3
|      5    I stay
|      dtype: object
|
|  duplicated(self, keep='first')
|      Indicate duplicate Series values.
|
|      Duplicated values are indicated as ``True`` values in the resulting
|      Series. Either all duplicates, all except the first or all except the
|      last occurrence of duplicates can be indicated.
|
|      Parameters
```

```
|      ----------
|      keep : {'first', 'last', False}, default 'first'
|          - '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
|      -------
|      pandas.core.series.Series
|
|      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=0)
|      Equal to of series and other, element-wise (binary operator `eq`).
|
|      Equivalent to ``series == other``, but with support to substitute a fill_value fo
r
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
```

```
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.None
|
|      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
|
|  ewm(self, com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True,
ignore_na=False, axis=0)
|      Provides exponential weighted functions.
|
|      .. versionadded:: 0.18.0
|
|      Parameters
|      ----------
|      com : float, optional
|          Specify decay in terms of center of mass,
|          :math:`\alpha = 1 / (1 + com),\text{ for } com \geq 0`
|      span : float, optional
|          Specify decay in terms of span,
|          :math:`\alpha = 2 / (span + 1),\text{ for } span \geq 1`
|      halflife : float, optional
|          Specify decay in terms of half-life,
|          :math:`\alpha = 1 - exp(log(0.5) / halflife),\text{ for } halflife > 0`
|      alpha : float, optional
|          Specify smoothing factor :math:`\alpha` directly,
|          :math:`0 < \alpha \leq 1`
|
|          .. versionadded:: 0.18.0
|
|      min_periods : int, default 0
|          Minimum number of observations in window required to have a value
|          (otherwise result is NA).
|      adjust : bool, default True
|          Divide by decaying adjustment factor in beginning periods to account
|          for imbalance in relative weightings (viewing EWMA as a moving average)
|      ignore_na : bool, default False
```

```
 |          Ignore missing values when calculating weights;
 |          specify True to reproduce pre-0.15.0 behavior
 |
 |      Returns
 |      -------
 |      a Window sub-classed for the particular operation
 |
 |      See Also
 |      --------
 |      rolling : Provides rolling window calculations.
 |      expanding : Provides expanding transformations.
 |
 |      Notes
 |      -----
 |      Exactly one of center of mass, span, half-life, and alpha must be provided.
 |      Allowed values and relationship between the parameters are specified in the
 |      parameter descriptions above; see the link at the end of this section for
 |      a detailed explanation.
 |
 |      When adjust is True (default), weighted averages are calculated using
 |      weights (1-alpha)**(n-1), (1-alpha)**(n-2), ..., 1-alpha, 1.
 |
 |      When adjust is False, weighted averages are calculated recursively as:
 |         weighted_average[0] = arg[0];
 |         weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].
 |
 |      When ignore_na is False (default), weights are based on absolute positions.
 |      For example, the weights of x and y used in calculating the final weighted
 |      average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and
 |      (1-alpha)**2 and alpha (if adjust is False).
 |
 |      When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based
 |      on relative positions. For example, the weights of x and y used in
 |      calculating the final weighted average of [x, None, y] are 1-alpha and 1
 |      (if adjust is True), and 1-alpha and alpha (if adjust is False).
 |
 |      More details can be found at
 |      http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weight
ed-windows
 |
 |      Examples
 |      --------
 |
 |      >>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
 |           B
 |      0  0.0
 |      1  1.0
 |      2  2.0
 |      3  NaN
 |      4  4.0
 |
 |      >>> df.ewm(com=0.5).mean()
 |                B
 |      0  0.000000
 |      1  0.750000
 |      2  1.615385
 |      3  1.615385
 |      4  3.670213
 |
 |  expanding(self, min_periods=1, center=False, axis=0)
 |      Provides expanding transformations.
 |
 |      .. versionadded:: 0.18.0
 |
 |      Parameters
 |      ----------
 |      min_periods : int, default 1
 |          Minimum number of observations in window required to have a value
 |          (otherwise result is NA).
 |      center : bool, default False
 |          Set the labels at the center of the window.
 |      axis : int or str, default 0
```

```
 |
 |      Returns
 |      -------
 |      a Window sub-classed for the particular operation
 |
 |      See Also
 |      --------
 |      rolling : Provides rolling window calculations.
 |      ewm : Provides exponential weighted functions.
 |
 |      Notes
 |      -----
 |      By default, the result is set to the right edge of the window. This can be
 |      changed to the center of the window by setting ``center=True``.
 |
 |      Examples
 |      --------
 |
 |      >>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
 |           B
 |      0  0.0
 |      1  1.0
 |      2  2.0
 |      3  NaN
 |      4  4.0
 |
 |      >>> df.expanding(2).sum()
 |           B
 |      0  NaN
 |      1  1.0
 |      2  3.0
 |      3  3.0
 |      4  7.0
 |
 |  fillna(self, value=None, method=None, axis=None, inplace=False, limit=None, downcast=
None, **kwargs)
 |      Fill NA/NaN values using the specified method.
 |
 |      Parameters
 |      ----------
 |      value : scalar, dict, Series, or DataFrame
 |          Value to use to fill holes (e.g. 0), alternately a
 |          dict/Series/DataFrame of values specifying which value to use for
 |          each index (for a Series) or column (for a DataFrame). (values not
 |          in the dict/Series/DataFrame will not be filled). This value cannot
 |          be a list.
 |      method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None
 |          Method to use for filling holes in reindexed Series
 |          pad / ffill: propagate last valid observation forward to next valid
 |          backfill / bfill: use NEXT valid observation to fill gap
 |      axis : {0 or 'index'}
 |      inplace : boolean, 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
 |      -------
 |      filled : Series
 |
 |      See Also
```

```
|      --------
|      interpolate : Fill NaN values using interpolation.
|      reindex, asfreq
|
|      Examples
|      --------
|      >>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
|      ...                    [3, 4, np.nan, 1],
|      ...                    [np.nan, np.nan, np.nan, 5],
|      ...                    [np.nan, 3, np.nan, 4]],
|      ...                    columns=list('ABCD'))
|      >>> df
|          A    B   C  D
|      0  NaN  2.0 NaN  0
|      1  3.0  4.0 NaN  1
|      2  NaN  NaN NaN  5
|      3  NaN  3.0 NaN  4
|
|      Replace all NaN elements with 0s.
|
|      >>> df.fillna(0)
|          A   B   C   D
|      0   0.0 2.0 0.0 0
|      1   3.0 4.0 0.0 1
|      2   0.0 0.0 0.0 5
|      3   0.0 3.0 0.0 4
|
|      We can also propagate non-null values forward or backward.
|
|      >>> df.fillna(method='ffill')
|          A   B   C   D
|      0   NaN 2.0 NaN 0
|      1   3.0 4.0 NaN 1
|      2   3.0 4.0 NaN 5
|      3   3.0 3.0 NaN 4
|
|      Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1,
|      2, and 3 respectively.
|
|      >>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
|      >>> df.fillna(value=values)
|          A   B   C   D
|      0   0.0 2.0 2.0 0
|      1   3.0 4.0 2.0 1
|      2   0.0 1.0 2.0 5
|      3   0.0 3.0 2.0 4
|
|      Only replace the first NaN element.
|
|      >>> df.fillna(value=values, limit=1)
|          A   B   C   D
|      0   0.0 2.0 2.0 0
|      1   3.0 4.0 NaN 1
|      2   NaN 1.0 NaN 5
|      3   NaN 3.0 NaN 4
|
|  floordiv(self, other, level=None, fill_value=None, axis=0)
|      Integer division of series and other, element-wise (binary operator `floordiv`).
|
|      Equivalent to ``series // other``, but with support to substitute a fill_value fo
r
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
```

```
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.rfloordiv
 |
 |      Examples
 |      --------
 |      >>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
 |      >>> a
 |      a    1.0
 |      b    1.0
 |      c    1.0
 |      d    NaN
 |      dtype: float64
 |      >>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
 |      >>> b
 |      a    1.0
 |      b    NaN
 |      d    1.0
 |      e    NaN
 |      dtype: float64
 |      >>> a.add(b, fill_value=0)
 |      a    2.0
 |      b    1.0
 |      c    1.0
 |      d    1.0
 |      e    NaN
 |      dtype: float64
 |
 |  ge(self, other, level=None, fill_value=None, axis=0)
 |      Greater than or equal to of series and other, element-wise (binary operator `ge`)
.
 |
 |      Equivalent to ``series >= other``, but with support to substitute a fill_value fo
r
 |      missing data in one of the inputs.
 |
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.None
 |
 |      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
|
|  get_value(self, label, takeable=False)
|      Quickly retrieve single value at passed index label.
|
|      .. deprecated:: 0.21.0
|          Please use .at[] or .iat[] accessors.
|
|      Parameters
|      ----------
|      label : object
|      takeable : interpret the index as indexers, default False
|
|      Returns
|      -------
|      value : scalar value
|
|  get_values(self)
|      Same as values (but handles sparseness conversions); is a view.
|
|  gt(self, other, level=None, fill_value=None, axis=0)
|      Greater than of series and other, element-wise (binary operator `gt`).
|
|      Equivalent to ``series > other``, but with support to substitute a fill_value for
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.None
|
|      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
|
|  hist = hist_series(self, by=None, ax=None, grid=True, xlabelsize=None, xrot=None, yla
belsize=None, yrot=None, figsize=None, bins=10, **kwds)
|      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 : boolean, 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 : integer 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.
|      bins : integer, default 10
|          Number of histogram bins to be used
|      `**kwds` : keywords
|          To be passed to the actual plotting function
|
|      See Also
|      --------
|      matplotlib.axes.Axes.hist : Plot a histogram using matplotlib.
|
|  idxmax(self, axis=0, skipna=True, *args, **kwargs)
|      Return the row label of the maximum value.
|
|      If multiple values equal the maximum, the first row label with that
|      value is returned.
|
|      Parameters
|      ----------
|      skipna : boolean, default True
|          Exclude NA/null values. If the entire Series is NA, the result
|          will be NA.
|      axis : int, default 0
|          For compatibility with DataFrame.idxmax. Redundant for application
|          on Series.
|      *args, **kwargs
|          Additional keywords have no effect but might be accepted
|          for compatibility with NumPy.
|
|      Returns
|      -------
|      idxmax : Index of maximum of values.
|
|      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=0, skipna=True, *args, **kwargs)
|      Return the row label of the minimum value.
|
|      If multiple values equal the minimum, the first row label with that
|      value is returned.
|
|      Parameters
|      ----------
|      skipna : boolean, default True
|          Exclude NA/null values. If the entire Series is NA, the result
|          will be NA.
|      axis : int, default 0
|          For compatibility with DataFrame.idxmin. Redundant for application
|          on Series.
|      *args, **kwargs
|          Additional keywords have no effect but might be accepted
|          for compatibility with NumPy.
|
|      Returns
|      -------
|      idxmin : Index of minimum of values.
|
|      Raises
|      ------
|      ValueError
|          If the Series is empty.
|
|      See Also
|      --------
|      numpy.argmin : Return indices of the minimum values
|          along the given axis.
|      DataFrame.idxmin : Return index of first occurrence of minimum
|          over requested axis.
|      Series.idxmax : Return index *label* of the first occurrence
```

```
|          of maximum of values.
|
|      Notes
|      -----
|      This method is the Series version of ``ndarray.argmin``. This method
|      returns the label of the minimum, while ``ndarray.argmin`` returns
|      the position. To get the position, use ``series.values.argmin()``.
|
|      Examples
|      --------
|      >>> s = pd.Series(data=[1, None, 4, 1],
|      ...               index=['A' ,'B' ,'C' ,'D'])
|      >>> s
|      A    1.0
|      B    NaN
|      C    4.0
|      D    1.0
|      dtype: float64
|
|      >>> s.idxmin()
|      'A'
|
|      If `skipna` is False and there is an NA value in the data,
|      the function returns ``nan``.
|
|      >>> s.idxmin(skipna=False)
|      nan
|
|  isin(self, values)
|      Check whether `values` are contained in Series.
|
|      Return a boolean Series showing whether each element in the Series
|      matches an element in the passed sequence of `values` exactly.
|
|      Parameters
|      ----------
|      values : set or list-like
|          The sequence of values to test. Passing in a single string will
|          raise a ``TypeError``. Instead, turn a single string into a
|          list of one element.
|
|          .. versionadded:: 0.18.1
|
|            Support for values as a set.
|
|      Returns
|      -------
|      isin : Series (bool dtype)
|
|      Raises
|      ------
|      TypeError
|        * If `values` is a string
|
|      See Also
|      --------
|      DataFrame.isin : Equivalent method on DataFrame.
|
|      Examples
|      --------
|      >>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
|      ...                'hippo'], name='animal')
|      >>> s.isin(['cow', 'lama'])
|      0     True
|      1     True
|      2     True
|      3    False
|      4     True
|      5    False
|      Name: animal, dtype: bool
|
|      Passing a single string as ``s.isin('lama')`` will raise an error. Use
```

```
|      a list of one element instead:
|
|      >>> s.isin(['lama'])
|      0     True
|      1    False
|      2     True
|      3    False
|      4     True
|      5    False
|      Name: animal, dtype: bool
|
|  isna(self)
|      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 not an NA value.
|
|      See Also
|      --------
|      Series.isnull : Alias of isna.
|      Series.notna : Boolean inverse of isna.
|      Series.dropna : Omit axes labels with missing values.
|      isna : Top-level isna.
|
|      Examples
|      --------
|      Show which entries in a DataFrame are NA.
|
|      >>> df = pd.DataFrame({'age': [5, 6, np.NaN],
|      ...                   'born': [pd.NaT, pd.Timestamp('1939-05-27'),
|      ...                            pd.Timestamp('1940-04-25')],
|      ...                   'name': ['Alfred', 'Batman', ''],
|      ...                   'toy': [None, 'Batmobile', 'Joker']})
|      >>> df
|         age       born    name        toy
|      0  5.0        NaT  Alfred       None
|      1  6.0 1939-05-27  Batman  Batmobile
|      2  NaN 1940-04-25              Joker
|
|      >>> df.isna()
|           age   born   name    toy
|      0  False   True  False   True
|      1  False  False  False  False
|      2   True  False  False  False
|
|      Show which entries in a Series are NA.
|
|      >>> ser = pd.Series([5, 6, np.NaN])
|      >>> ser
|      0    5.0
|      1    6.0
|      2    NaN
|      dtype: float64
|
|      >>> ser.isna()
|      0    False
|      1    False
|      2     True
|      dtype: bool
|
|  isnull(self)
|      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 not an NA value.
|
|      See Also
|      --------
|      Series.isnull : Alias of isna.
|      Series.notna : Boolean inverse of isna.
|      Series.dropna : Omit axes labels with missing values.
|      isna : Top-level isna.
|
|      Examples
|      --------
|      Show which entries in a DataFrame are NA.
|
|      >>> df = pd.DataFrame({'age': [5, 6, np.NaN],
|      ...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
|      ...                             pd.Timestamp('1940-04-25')],
|      ...                    'name': ['Alfred', 'Batman', ''],
|      ...                    'toy': [None, 'Batmobile', 'Joker']})
|      >>> df
|         age       born    name        toy
|      0  5.0        NaT  Alfred       None
|      1  6.0 1939-05-27  Batman  Batmobile
|      2  NaN 1940-04-25              Joker
|
|      >>> df.isna()
|           age   born   name    toy
|      0  False   True  False   True
|      1  False  False  False  False
|      2   True  False  False  False
|
|      Show which entries in a Series are NA.
|
|      >>> ser = pd.Series([5, 6, np.NaN])
|      >>> ser
|      0    5.0
|      1    6.0
|      2    NaN
|      dtype: float64
|
|      >>> ser.isna()
|      0    False
|      1    False
|      2     True
|      dtype: bool
|
|  items = iteritems(self)
|
|  iteritems(self)
|      Lazily iterate over (index, value) tuples.
|
|  keys(self)
|      Alias for index.
|
|  kurt(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
|      Return unbiased kurtosis over requested axis using Fisher's definition of
|      kurtosis (kurtosis of normal == 0.0). Normalized by N-1.
|
|      Parameters
|      ----------
|      axis : {index (0)}
```

```
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a scalar.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |
 |      Returns
 |      -------
 |      kurt : scalar or Series (if level specified)
 |
 |  kurtosis = kurt(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs
)
 |
 |  le(self, other, level=None, fill_value=None, axis=0)
 |      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 fo
r
 |      missing data in one of the inputs.
 |
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.None
 |
 |      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
 |
 |  lt(self, other, level=None, fill_value=None, axis=0)
 |      Less than of series and other, element-wise (binary operator `lt`).
```

```
|
|      Equivalent to ``series < other``, but with support to substitute a fill_value for
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.None
|
|      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
|
|  mad(self, axis=None, skipna=None, level=None)
|      Return the mean absolute deviation of the values for the requested axis.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      mad : scalar or Series (if level specified)
|
|  map(self, arg, na_action=None)
|      Map values of Series according to input correspondence.
```

```
|
|      Used for substituting each value in a Series with another value,
|      that may be derived from a function, a ``dict`` or
|      a :class:`Series`.
|
|      Parameters
|      ----------
|      arg : function, dict, 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
|
```

```
|  max(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
|      Return the maximum of the values for the requested axis.
|
|                  If you want the *index* of the maximum, use ``idxmax``. This is
|                  the equivalent of the ``numpy.ndarray`` method ``argmax``.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      max : scalar or Series (if level specified)
|
|      See Also
|      --------
|      Series.sum : Return the sum.
|      Series.min : Return the minimum.
|      Series.max : Return the maximum.
|      Series.idxmin : Return the index of the minimum.
|      Series.idxmax : Return the index of the maximum.
|      DataFrame.min : Return the sum over the requested axis.
|      DataFrame.min : Return the minimum over the requested axis.
|      DataFrame.max : Return the maximum over the requested axis.
|      DataFrame.idxmin : Return the index of the minimum over the requested axis.
|      DataFrame.idxmax : Return the index of the maximum over the requested axis.
|
|      Examples
|      --------
|
|      >>> idx = pd.MultiIndex.from_arrays([
|      ...     ['warm', 'warm', 'cold', 'cold'],
|      ...     ['dog', 'falcon', 'fish', 'spider']],
|      ...     names=['blooded', 'animal'])
|      >>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
|      >>> s
|      blooded  animal
|      warm     dog       4
|               falcon    2
|      cold     fish      0
|               spider    8
|      Name: legs, dtype: int64
|
|      >>> s.max()
|      8
|
|      Max using level names, as well as indices.
|
|      >>> s.max(level='blooded')
|      blooded
|      warm    4
|      cold    8
|      Name: legs, dtype: int64
|
|      >>> s.max(level=0)
|      blooded
|      warm    4
|      cold    8
|      Name: legs, dtype: int64
|
|  mean(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
```

```
|      Return the mean of the values for the requested axis.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      mean : scalar or Series (if level specified)
|
|  median(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
|      Return the median of the values for the requested axis.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      median : scalar or Series (if level specified)
|
|  memory_usage(self, index=True, deep=False)
|      Return the memory usage of the Series.
|
|      The memory usage can optionally include the contribution of
|      the index and of elements of `object` dtype.
|
|      Parameters
|      ----------
|      index : bool, default True
|          Specifies whether to include the memory usage of the Series index.
|      deep : bool, default False
|          If True, introspect the data deeply by interrogating
|          `object` dtypes for system-level memory consumption, and include
|          it in the returned value.
|
|      Returns
|      -------
|      int
|          Bytes of memory consumed.
|
|      See Also
|      --------
|      numpy.ndarray.nbytes : Total bytes consumed by the elements of the
|          array.
|      DataFrame.memory_usage : Bytes consumed by a DataFrame.
|
|      Examples
|      --------
|      >>> s = pd.Series(range(3))
```

```
|      >>> s.memory_usage()
|      104
|
|      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()
|      96
|      >>> s.memory_usage(deep=True)
|      212
|
|  min(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
|      Return the minimum of the values for the requested axis.
|
|                  If you want the *index* of the minimum, use ``idxmin``. This is
|                  the equivalent of the ``numpy.ndarray`` method ``argmin``.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      min : scalar or Series (if level specified)
|
|      See Also
|      --------
|      Series.sum : Return the sum.
|      Series.min : Return the minimum.
|      Series.max : Return the maximum.
|      Series.idxmin : Return the index of the minimum.
|      Series.idxmax : Return the index of the maximum.
|      DataFrame.min : Return the sum over the requested axis.
|      DataFrame.min : Return the minimum over the requested axis.
|      DataFrame.max : Return the maximum over the requested axis.
|      DataFrame.idxmin : Return the index of the minimum over the requested axis.
|      DataFrame.idxmax : Return the index of the maximum over the requested axis.
|
|      Examples
|      --------
|
|      >>> idx = pd.MultiIndex.from_arrays([
|      ...     ['warm', 'warm', 'cold', 'cold'],
|      ...     ['dog', 'falcon', 'fish', 'spider']],
|      ...     names=['blooded', 'animal'])
|      >>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
|      >>> s
|      blooded  animal
|      warm     dog       4
|               falcon    2
|      cold     fish      0
|               spider    8
|      Name: legs, dtype: int64
```

```
|
|      >>> s.min()
|      0
|
|      Min using level names, as well as indices.
|
|      >>> s.min(level='blooded')
|      blooded
|      warm    2
|      cold    0
|      Name: legs, dtype: int64
|
|      >>> s.min(level=0)
|      blooded
|      warm    2
|      cold    0
|      Name: legs, dtype: int64
|
|  mod(self, other, level=None, fill_value=None, axis=0)
|      Modulo of series and other, element-wise (binary operator `mod`).
|
|      Equivalent to ``series % other``, but with support to substitute a fill_value for
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.rmod
|
|      Examples
|      --------
|      >>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
|      >>> a
|      a    1.0
|      b    1.0
|      c    1.0
|      d    NaN
|      dtype: float64
|      >>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
|      >>> b
|      a    1.0
|      b    NaN
|      d    1.0
|      e    NaN
|      dtype: float64
|      >>> a.add(b, fill_value=0)
|      a    2.0
|      b    1.0
|      c    1.0
|      d    1.0
|      e    NaN
|      dtype: float64
|
|  mode(self, dropna=True)
|      Return the mode(s) of the dataset.
|
|      Always returns Series even if only one value is returned.
```

```
|
|      Parameters
|      ----------
|      dropna : boolean, default True
|          Don't consider counts of NaN/NaT.
|
|          .. versionadded:: 0.24.0
|
|      Returns
|      -------
|      modes : Series (sorted)
|
|  mul(self, other, level=None, fill_value=None, axis=0)
|      Multiplication of series and other, element-wise (binary operator `mul`).
|
|      Equivalent to ``series * other``, but with support to substitute a fill_value for
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.rmul
|
|      Examples
|      --------
|      >>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
|      >>> a
|      a    1.0
|      b    1.0
|      c    1.0
|      d    NaN
|      dtype: float64
|      >>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
|      >>> b
|      a    1.0
|      b    NaN
|      d    1.0
|      e    NaN
|      dtype: float64
|      >>> a.add(b, fill_value=0)
|      a    2.0
|      b    1.0
|      c    1.0
|      d    1.0
|      e    NaN
|      dtype: float64
|
|  multiply = mul(self, other, level=None, fill_value=None, axis=0)
|
|  ne(self, other, level=None, fill_value=None, axis=0)
|      Not equal to of series and other, element-wise (binary operator `ne`).
|
|      Equivalent to ``series != other``, but with support to substitute a fill_value fo
r
|      missing data in one of the inputs.
|
|      Parameters
```

```
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.None
|
|      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
|
|  nlargest(self, n=5, keep='first')
|      Return the largest `n` elements.
|
|      Parameters
|      ----------
|      n : int, default 5
|          Return this many descending sorted values.
|      keep : {'first', 'last', 'all'}, default 'first'
|          When there are duplicate values that cannot all fit in a
|          Series of `n` elements:
|
|          - ``first`` : take the first occurrences based on the index order
|          - ``last`` : take the last occurrences based on the index order
|          - ``all`` : keep all occurrences. This can result in a Series of
|              size larger than `n`.
|
|      Returns
|      -------
|      Series
|          The `n` largest values in the Series, sorted in decreasing order.
|
|      See Also
|      --------
|      Series.nsmallest: Get the `n` smallest elements.
|      Series.sort_values: Sort Series by values.
|      Series.head: Return the first `n` rows.
|
|      Notes
|      -----
```

```
|      Faster than ``.sort_values(ascending=False).head(n)`` for small `n`
|      relative to the size of the ``Series`` object.
|
|      Examples
|      --------
|      >>> countries_population = {"Italy": 59000000, "France": 65000000,
|      ...                        "Malta": 434000, "Maldives": 434000,
|      ...                        "Brunei": 434000, "Iceland": 337000,
|      ...                        "Nauru": 11300, "Tuvalu": 11300,
|      ...                        "Anguilla": 11300, "Monserat": 5200}
|      >>> s = pd.Series(countries_population)
|      >>> s
|      Italy       59000000
|      France      65000000
|      Malta         434000
|      Maldives      434000
|      Brunei        434000
|      Iceland       337000
|      Nauru          11300
|      Tuvalu         11300
|      Anguilla       11300
|      Monserat        5200
|      dtype: int64
|
|      The `n` largest elements where ``n=5`` by default.
|
|      >>> s.nlargest()
|      France      65000000
|      Italy       59000000
|      Malta         434000
|      Maldives      434000
|      Brunei        434000
|      dtype: int64
|
|      The `n` largest elements where ``n=3``. Default `keep` value is 'first'
|      so Malta will be kept.
|
|      >>> s.nlargest(3)
|      France    65000000
|      Italy     59000000
|      Malta       434000
|      dtype: int64
|
|      The `n` largest elements where ``n=3`` and keeping the last duplicates.
|      Brunei will be kept since it is the last with value 434000 based on
|      the index order.
|
|      >>> s.nlargest(3, keep='last')
|      France      65000000
|      Italy       59000000
|      Brunei        434000
|      dtype: int64
|
|      The `n` largest elements where ``n=3`` with all duplicates kept. Note
|      that the returned Series has five elements due to the three duplicates.
|
|      >>> s.nlargest(3, keep='all')
|      France      65000000
|      Italy       59000000
|      Malta         434000
|      Maldives      434000
|      Brunei        434000
|      dtype: int64
|
|  nonzero(self)
|      Return the *integer* indices of the elements that are non-zero.
|
|      .. deprecated:: 0.24.0
|         Please use .to_numpy().nonzero() as a replacement.
|
|      This method is equivalent to calling `numpy.nonzero` on the
|      series data. For compatibility with NumPy, the return value is
```

```
|      the same (a tuple with an array of indices for each dimension),
|      but it will always be a one-item tuple because series only have
|      one dimension.
|
|      See Also
|      --------
|      numpy.nonzero
|
|      Examples
|      --------
|      >>> s = pd.Series([0, 3, 0, 4])
|      >>> s.nonzero()
|      (array([1, 3]),)
|      >>> s.iloc[s.nonzero()[0]]
|      1    3
|      3    4
|      dtype: int64
|
|      >>> s = pd.Series([0, 3, 0, 4], index=['a', 'b', 'c', 'd'])
|      # same return although index of s is different
|      >>> s.nonzero()
|      (array([1, 3]),)
|      >>> s.iloc[s.nonzero()[0]]
|      b    3
|      d    4
|      dtype: int64
|
|  notna(self)
|      Detect existing (non-missing) values.
|
|      Return a boolean same-sized object indicating if the values are not NA.
|      Non-missing values get mapped to True. Characters such as empty
|      strings ``''`` or :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({'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)
|      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({'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=5, keep='first')
|      Return the smallest `n` elements.
|
|      Parameters
|      ----------
|      n : int, default 5
|          Return this many ascending sorted values.
|      keep : {'first', 'last', 'all'}, default 'first'
|          When there are duplicate values that cannot all fit in a
|          Series of `n` elements:
|
|          - ``first`` : take the first occurrences based on the index order
|          - ``last`` : take the last occurrences based on the index order
|          - ``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, "Monserat": 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
|      Monserat        5200
|      dtype: int64
|
|      The `n` largest elements where ``n=5`` by default.
|
|      >>> s.nsmallest()
|      Monserat      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)
|      Monserat     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')
 |      Monserat     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')
 |      Monserat     5200
 |      Nauru       11300
 |      Tuvalu      11300
 |      Anguilla    11300
 |      dtype: int64
 |
 |  pow(self, other, level=None, fill_value=None, axis=0)
 |      Exponential power of series and other, element-wise (binary operator `pow`).
 |
 |      Equivalent to ``series ** other``, but with support to substitute a fill_value fo
r
 |      missing data in one of the inputs.
 |
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.rpow
 |
 |      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
 |
```

```
 |  prod(self, axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwar
gs)
 |      Return the product of the values for the requested axis.
 |
 |      Parameters
 |      ----------
 |      axis : {index (0)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a scalar.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      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.
 |
 |          .. versionadded :: 0.22.0
 |
 |             Added with the default being 0. This means the sum of an all-NA
 |             or empty Series is 0, and the product of an all-NA or empty
 |             Series is 1.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |
 |      Returns
 |      -------
 |      prod : scalar or Series (if level specified)
 |
 |      Examples
 |      --------
 |      By default, the product of an empty or all-NA Series is ``1``
 |
 |      >>> pd.Series([]).prod()
 |      1.0
 |
 |      This can be controlled with the ``min_count`` parameter
 |
 |      >>> pd.Series([]).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=None, skipna=None, level=None, numeric_only=None, min_count
=0, **kwargs)
 |
 |  ptp(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |      Returns the difference between the maximum value and the
 |                  minimum value in the object. This is the equivalent of the
 |                  ``numpy.ndarray`` method ``ptp``.
 |
 |      .. deprecated:: 0.24.0
 |                      Use numpy.ptp instead
 |
 |      Parameters
 |      ----------
 |      axis : {index (0)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
```

```
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      ptp : scalar or Series (if level specified)
|
|  put(self, *args, **kwargs)
|      Applies the `put` method to its `values` attribute if it has one.
|
|      See Also
|      --------
|      numpy.ndarray.put
|
|  quantile(self, q=0.5, interpolation='linear')
|      Return value at the given quantile.
|
|      Parameters
|      ----------
|      q : float or array-like, default 0.5 (50% quantile)
|          0 <= q <= 1, the quantile(s) to compute
|      interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
|          .. versionadded:: 0.18.0
|
|          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
|      -------
|      quantile : float or Series
|          if ``q`` is an array, a Series will be returned where the
|          index is ``q`` and the values are the quantiles.
|
|      See Also
|      --------
|      core.window.Rolling.quantile
|      numpy.percentile
|
|      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=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
```

```
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.add
 |
 |      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='C')
 |      Return the flattened underlying data as an ndarray.
 |
 |      See Also
 |      --------
 |      numpy.ndarray.ravel
 |
 |  rdiv = rtruediv(self, other, level=None, fill_value=None, axis=0)
 |
 |  rdivmod(self, other, level=None, fill_value=None, axis=0)
 |      Integer division and modulo of series and other, element-wise (binary operator `r
divmod`).
 |
 |      Equivalent to ``other divmod series``, but with support to substitute a fill_valu
e for
 |      missing data in one of the inputs.
 |
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
```

```
|      --------
|      Series.divmod
|
|      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
|
|  reindex(self, index=None, **kwargs)
|      Conform Series to new index with optional filling logic, placing
|      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 / index to conform to, should be specified using
|          keywords. Preferably an Index object to avoid duplicating data
|
|      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 most
|          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.
|
```

```
|          .. versionadded:: 0.21.0 (list-like tolerance)
|
|      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.
|
|  reindex_axis(self, labels, axis=0, **kwargs)
|      Conform Series to new index with optional filling logic.
|
|      .. deprecated:: 0.21.0
|          Use ``Series.reindex`` instead.
|
|  rename(self, index=None, **kwargs)
|      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
|          dict-like or functions are transformations to apply to
|          the index.
|          Scalar or hashable sequence-like will alter the ``Series.name``
|          attribute.
|      copy : bool, default True
|          Also copy underlying data
|      inplace : bool, default False
|          Whether to return a new Series. If True then value of copy is
|          ignored.
|      level : int or level name, default None
|          In case of a MultiIndex, only rename labels in the specified
|          level.
|
|      Returns
|      -------
|      renamed : Series (new object)
|
|      See Also
|      --------
|      Series.rename_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
|
|  reorder_levels(self, order)
|      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, axis=None)
|      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
|          Must be ``None``. Has no effect but is accepted for compatibility
|          with numpy.
|
|      Returns
|      -------
|      repeated_series : 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=None, inplace=False, limit=None, regex=False, me
```

```
thod='pad')
 |      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 `value`
 |                parameter should be `None`.
 |              - 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 `value`
 |                parameter should be ``None`` 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, in place. Note: this will modify any
 |          other views on this object (e.g. a column from a DataFrame).
 |          Returns the caller if this is True.
```

```
|       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', `None`}
|           The method to use when for replacement, when `to_replace` is a
|           scalar, list or tuple and `value` is ``None``.
|
|           .. versionchanged:: 0.23.0
|               Added to DataFrame.
|
|       Returns
|       -------
|       Series
|           Object after replacement.
|
|       Raises
|       ------
|       AssertionError
|           * If `regex` is not a ``bool`` and `to_replace` is not
|             ``None``.
|       TypeError
|           * 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([0, 1, 2, 3, 4])
|       >>> s.replace(0, 5)
|       0    5
|       1    1
|       2    2
|       3    3
|       4    4
|       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    0
|       1    3
|       2    3
|       3    3
|       4    4
|       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
|
|      Note that when replacing multiple ``bool`` or ``datetime64`` objects,
|      the data types in the `to_replace` parameter must match the data
|      type of the value being replaced:
|
|      >>> df = pd.DataFrame({'A': [True, False, True],
|      ...                    'B': [False, True, False]})
|      >>> df.replace({'a string': 'new value', True: False})  # raises
|      Traceback (most recent call last):
|          ...
|      TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
|
|      This raises a ``TypeError`` because one of the ``dict`` keys is not of
|      the correct type for replacement.
|
|      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=None`` 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.
|      The command ``s.replace('a', None)`` is actually equivalent to
|      ``s.replace(to_replace='a', value=None, method='pad')``:
|
|      >>> s.replace('a', None)
|      0    10
|      1    10
|      2    10
|      3     b
|      4     b
```

```
|      dtype: object
|
|  reset_index(self, level=None, drop=False, name=None, inplace=False)
|      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).
|
|      Returns
|      -------
|      Series or DataFrame
|          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
|
|      To update the Series in place, without generating a new one
|      set `inplace` to True. Note that it also requires ``drop=True``.
|
```

```
|      >>> s.reset_index(inplace=True, drop=True)
|      >>> s
|      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=0)
|      Integer division of series and other, element-wise (binary operator `rfloordiv`).
|
|      Equivalent to ``other // series``, but with support to substitute a fill_value fo
r
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.floordiv
|
|      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
|
|  rmod(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.mod
|
|      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
|
|  rmul(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
```

```
 |      Parameters
 |      ----------
 |      other : Series or scalar value
 |      fill_value : None or float value, default None (NaN)
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful Series alignment, with this value before computation.
 |          If data in both corresponding Series locations is missing
 |          the result will be missing
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level
 |
 |      Returns
 |      -------
 |      result : Series
 |
 |      See Also
 |      --------
 |      Series.mul
 |
 |      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
 |
 |  rolling(self, window, min_periods=None, center=False, win_type=None, on=None, axis=0,
closed=None)
 |      Provides rolling window calculations.
 |
 |      .. versionadded:: 0.18.0
 |
 |      Parameters
 |      ----------
 |      window : int, or offset
 |          Size of the moving window. This is the number of observations used for
 |          calculating the statistic. Each window will be a fixed size.
 |
 |          If its an offset then this will be 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. This is
 |          new in 0.19.0
 |      min_periods : int, default None
 |          Minimum number of observations in window required to have a value
 |          (otherwise result is NA). For a window that is specified by an offset,
 |          `min_periods` will default to 1. Otherwise, `min_periods` will default
 |          to the size of the window.
 |      center : bool, default False
 |          Set the labels at the center of the window.
 |      win_type : str, default None
 |          Provide a window type. If ``None``, all points are evenly weighted.
 |          See the notes below for further information.
 |      on : str, optional
 |          For a DataFrame, column on which to calculate
```

```
|          the rolling window, rather than the index
|      axis : int or str, default 0
|      closed : str, default None
|          Make the interval closed on the 'right', 'left', 'both' or
|          'neither' endpoints.
|          For offset-based windows, it defaults to 'right'.
|          For fixed windows, defaults to 'both'. Remaining cases not implemented
|          for fixed windows.
|
|          .. versionadded:: 0.20.0
|
|      Returns
|      -------
|      a Window or Rolling sub-classed for the particular operation
|
|      See Also
|      --------
|      expanding : Provides expanding transformations.
|      ewm : Provides exponential weighted functions.
|
|      Notes
|      -----
|      By default, the result is set to the right edge of the window. This can be
|      changed to the center of the window by setting ``center=True``.
|
|      To learn more about the offsets & frequency strings, please see `this link
|      <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>`__.
|
|      The recognized win_types are:
|
|      * ``boxcar``
|      * ``triang``
|      * ``blackman``
|      * ``hamming``
|      * ``bartlett``
|      * ``parzen``
|      * ``bohman``
|      * ``blackmanharris``
|      * ``nuttall``
|      * ``barthann``
|      * ``kaiser`` (needs beta)
|      * ``gaussian`` (needs std)
|      * ``general_gaussian`` (needs power, width)
|      * ``slepian`` (needs width).
|
|      If ``win_type=None`` all points are evenly weighted. To learn more about
|      different window types see `scipy.signal window functions
|      <https://docs.scipy.org/doc/scipy/reference/signal.html#window-functions>`__.
|
|      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
|
|      Rolling sum with a window length of 2, using the 'triang'
|      window type.
|
|      >>> df.rolling(2, win_type='triang').sum()
|           B
|      0  NaN
|      1  1.0
|      2  2.5
|      3  NaN
|      4  NaN
|
```

```
|      Rolling sum with a window length of 2, min_periods defaults
|      to the window length.
|
|      >>> df.rolling(2).sum()
|           B
|      0  NaN
|      1  1.0
|      2  3.0
|      3  NaN
|      4  NaN
|
|      Same as above, but explicitly set the min_periods
|
|      >>> df.rolling(2, min_periods=1).sum()
|           B
|      0  0.0
|      1  1.0
|      2  3.0
|      3  2.0
|      4  4.0
|
|      A ragged (meaning not-a-regular frequency), time-indexed DataFrame
|
|      >>> df = 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
|                             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
|
|      Contrasting to an integer rolling window, this will roll a variable
|      length window corresponding to the time period.
|      The default for min_periods is 1.
|
|      >>> df.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
|
|  round(self, decimals=0, *args, **kwargs)
|      Round each value in a Series to the given number of decimals.
|
|      Parameters
|      ----------
|      decimals : int
|          Number of decimal places to round to (default: 0).
|          If decimals is negative, it specifies the number of
|          positions to the left of the decimal point.
|
|      Returns
|      -------
|      Series object
|
|      See Also
|      --------
|      numpy.around
|      DataFrame.round
|
|  rpow(self, other, level=None, fill_value=None, axis=0)
|      Exponential power of series and other, element-wise (binary operator `rpow`).
|
```

```
|      Equivalent to ``other ** series``, but with support to substitute a fill_value fo
r
|      missing data in one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.pow
|
|      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
|
|  rsub(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|
```

```
|      Series.sub
|
|      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
|
|  rtruediv(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.truediv
|
|      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
|
|  searchsorted(self, value, side='left', sorter=None)
|      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.
|
|      Parameters
|      ----------
|      value : array_like
|          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`.
|
|          .. versionchanged :: 0.24.0
|              If `value` is a scalar, an int is now always returned.
|              Previously, scalar inputs returned an 1-item array for
|              :class:`Series` and :class:`Categorical`.
|
|      See Also
|      --------
|      numpy.searchsorted
|
|      Notes
|      -----
|      Binary search is used to find the required insertion points.
|
|      Examples
|      --------
|
|      >>> x = pd.Series([1, 2, 3])
|      >>> x
|      0    1
|      1    2
|      2    3
|      dtype: int64
|
|      >>> x.searchsorted(4)
|      3
|
|      >>> x.searchsorted([0, 4])
|      array([0, 3])
|
|      >>> x.searchsorted([1, 3], side='left')
|      array([0, 2])
|
|      >>> x.searchsorted([1, 3], side='right')
|      array([1, 3])
|
|      >>> x = pd.Categorical(['apple', 'bread', 'bread',
|                              'cheese', 'milk'], ordered=True)
|      [apple, bread, bread, cheese, milk]
|      Categories (4, object): [apple < bread < cheese < milk]
|
|      >>> x.searchsorted('bread')
|      1
|
```

```
|      >>> x.searchsorted(['bread'], side='right')
|      array([3])
|
|  sem(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)
|      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)}
|      skipna : boolean, default True
|          Exclude NA/null values. If an entire row/column is NA, the result
|          will be NA
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar
|      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 : boolean, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|
|      Returns
|      -------
|      sem : scalar or Series (if level specified)
|
|  set_value(self, label, value, takeable=False)
|      Quickly set single value at passed label.
|
|      .. deprecated:: 0.21.0
|          Please use .at[] or .iat[] accessors.
|
|      If label is not contained, a new object is created with the label
|      placed at the end of the result index.
|
|      Parameters
|      ----------
|      label : object
|          Partial indexing with MultiIndex not allowed
|      value : object
|          Scalar value
|      takeable : interpret the index as indexers, default False
|
|      Returns
|      -------
|      series : Series
|          If label is contained, will be reference to calling Series,
|          otherwise a new object
|
|  shift(self, periods=1, freq=None, axis=0, fill_value=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`.
|
|      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.
|      axis : {0 or 'index', 1 or 'columns', None}, default None
|          Shift direction.
|      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:: 0.24.0
|
|      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.
|      tshift : Shift the time index, using the index's frequency if
|          available.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'Col1': [10, 20, 15, 30, 45],
|      ...                    'Col2': [13, 23, 18, 33, 48],
|      ...                    'Col3': [17, 27, 22, 37, 52]})
|
|      >>> df.shift(periods=3)
|         Col1  Col2  Col3
|      0   NaN   NaN   NaN
|      1   NaN   NaN   NaN
|      2   NaN   NaN   NaN
|      3  10.0  13.0  17.0
|      4  20.0  23.0  27.0
|
|      >>> df.shift(periods=1, axis='columns')
|         Col1  Col2  Col3
|      0   NaN  10.0  13.0
|      1   NaN  20.0  23.0
|      2   NaN  15.0  18.0
|      3   NaN  30.0  33.0
|      4   NaN  45.0  48.0
|
|      >>> df.shift(periods=3, fill_value=0)
|         Col1  Col2  Col3
|      0     0     0     0
|      1     0     0     0
|      2     0     0     0
|      3    10    13    17
|      4    20    23    27
|
|  skew(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
|      Return unbiased skew over requested axis
|      Normalized by N-1.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      skew : scalar or Series (if level specified)
```

```
 |
 |  sort_index(self, axis=0, level=None, ascending=True, inplace=False, kind='quicksort',
na_position='last', sort_remaining=True)
 |      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 : int, default 0
 |          Axis to direct sorting. This can only be 0 for Series.
 |      level : int, optional
 |          If not None, sort on values in specified index level(s).
 |      ascending : bool, default true
 |          Sort ascending vs. descending.
 |      inplace : bool, default False
 |          If True, perform operation in-place.
 |      kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
 |          Choice of sorting algorithm. See also :func:`numpy.sort` for more
 |          information.  'mergesort' is the only stable algorithm. 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.
 |
 |      Returns
 |      -------
 |      pandas.Series
 |          The original Series sorted by the labels
 |
 |      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
 |
 |      Sort Inplace
 |
 |      >>> s.sort_index(inplace=True)
 |      >>> s
 |      1    c
 |      2    b
 |      3    a
 |      4    d
 |      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
 |
 |  sort_values(self, axis=0, ascending=True, inplace=False, kind='quicksort', na_positio
n='last')
 |      Sort by the values.
 |
 |      Sort a Series in ascending or descending order by some
 |      criterion.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index'}, default 0
 |          Axis to direct sorting. The value 'index' is accepted for
 |          compatibility with DataFrame.sort_values.
 |      ascending : bool, default True
 |          If True, sort values in ascending order, otherwise descending.
 |      inplace : bool, default False
 |          If True, perform operation in-place.
 |      kind : {'quicksort', 'mergesort' or 'heapsort'}, default 'quicksort'
 |          Choice of sorting algorithm. See also :func:`numpy.sort` for more
 |          information. 'mergesort' is the only stable  algorithm.
 |      na_position : {'first' or 'last'}, default 'last'
 |          Argument 'first' puts NaNs at the beginning, 'last' puts NaNs at
 |          the end.
 |
 |      Returns
 |      -------
 |      Series
 |          Series ordered by values.
 |
 |      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 inplace
|
|       >>> s.sort_values(ascending=False, inplace=True)
|       >>> s
|       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
```

```
|
|  std(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)
|      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)}
|      skipna : boolean, default True
|          Exclude NA/null values. If an entire row/column is NA, the result
|          will be NA
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar
|      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 : boolean, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|
|      Returns
|      -------
|      std : scalar or Series (if level specified)
|
|  sub(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|      result : Series
|
|      See Also
|      --------
|      Series.rsub
|
|      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
|
|  subtract = sub(self, other, level=None, fill_value=None, axis=0)
|
|  sum(self, axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwarg
s)
|      Return the sum of the values for the requested axis.
|
|                  This is equivalent to the method ``numpy.sum``.
|
|      Parameters
|      ----------
|      axis : {index (0)}
|          Axis for the function to be applied on.
|      skipna : bool, default True
|          Exclude NA/null values when computing the result.
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar.
|      numeric_only : bool, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|      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.
|
|          .. versionadded :: 0.22.0
|
|             Added with the default being 0. This means the sum of an all-NA
|             or empty Series is 0, and the product of an all-NA or empty
|             Series is 1.
|      **kwargs
|          Additional keyword arguments to be passed to the function.
|
|      Returns
|      -------
|      sum : scalar or Series (if level specified)
|
|      See Also
|      --------
|      Series.sum : Return the sum.
|      Series.min : Return the minimum.
|      Series.max : Return the maximum.
|      Series.idxmin : Return the index of the minimum.
|      Series.idxmax : Return the index of the maximum.
|      DataFrame.min : 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
|
```

```
|      Sum using level names, as well as indices.
|
|      >>> s.sum(level='blooded')
|      blooded
|      warm    6
|      cold    8
|      Name: legs, dtype: int64
|
|      >>> s.sum(level=0)
|      blooded
|      warm    6
|      cold    8
|      Name: legs, dtype: int64
|
|      By default, the sum of an empty or all-NA Series is ``0``.
|
|      >>> pd.Series([]).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([]).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=-2, j=-1, copy=True)
|      Swap levels i and j in a MultiIndex.
|
|      Parameters
|      ----------
|      i, j : int, string (can be mixed)
|          Level of index to be swapped. Can pass level name as string.
|
|      Returns
|      -------
|      swapped : Series
|
|      .. versionchanged:: 0.18.1
|
|         The indexes ``i`` and ``j`` are now optional, and default to
|         the two innermost levels of the index.
|
|  to_csv(self, *args, **kwargs)
|      Write object to a comma-separated values (csv) file.
|
|      .. versionchanged:: 0.24.0
|          The order of arguments for Series was changed.
|
|      Parameters
|      ----------
|      path_or_buf : str or file handle, default None
|          File path or object, if None is provided the result is returned as
|          a string.  If a file object is passed it should be opened with
|          `newline=''`, disabling universal newlines.
|
|          .. versionchanged:: 0.24.0
|
|             Was previously named "path" for Series.
|
|      sep : str, default ','
|          String of length 1. Field delimiter for the output file.
|      na_rep : str, default ''
|          Missing data representation.
```

```
|       float_format : str, default None
|           Format string for floating point numbers.
|       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.
|
|           .. versionchanged:: 0.24.0
|
|              Previously defaulted to False for Series.
|
|       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
|           Python write mode, default 'w'.
|       encoding : str, optional
|           A string representing the encoding to use in the output file,
|           defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.
|       compression : str, default 'infer'
|           Compression mode among the following possible values: {'infer',
|           'gzip', 'bz2', 'zip', 'xz', None}. If 'infer' and `path_or_buf`
|           is path-like, then detect compression from the following
|           extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no
|           compression).
|
|           .. versionchanged:: 0.24.0
|
|              'infer' option added and set to default.
|
|       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.
|       line_terminator : string, 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:: 0.24.0
|       chunksize : int or None
|           Rows to write at a time.
|       tupleize_cols : bool, default False
|           Write MultiIndex columns as a list of tuples (if True) or in
|           the new, expanded format, where each MultiIndex column is a row
|           in the CSV (if False).
|
|           .. deprecated:: 0.21.0
|              This argument will be removed and will always write each row
|              of the multi-index as a separate row in the CSV file.
|       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.
|
|       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 : Load an Excel file into a DataFrame.
|
|      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'
|
|  to_dict(self, into=<class 'dict'>)
|      Convert Series to {label -> value} dict or dict-like object.
|
|      Parameters
|      ----------
|      into : class, default dict
|          The collections.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.
|
|          .. versionadded:: 0.21.0
|
|      Returns
|      -------
|      value_dict : collections.Mapping
|
|      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(<type 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})
|
|  to_frame(self, name=None)
|      Convert Series to DataFrame.
|
|      Parameters
|      ----------
|      name : object, default None
|          The passed name should substitute for the series name (if it has
|          one).
|
|      Returns
|      -------
|      data_frame : DataFrame
|
|  to_period(self, freq=None, copy=True)
|      Convert Series from DatetimeIndex to PeriodIndex with desired
|      frequency (inferred from index if not passed).
|
|      Parameters
|      ----------
|      freq : string, default
|
|      Returns
|      -------
|      ts : Series with PeriodIndex
|
|  to_sparse(self, kind='block', fill_value=None)
|      Convert Series to SparseSeries.
```

```
 |
 |      Parameters
 |      ----------
 |      kind : {'block', 'integer'}
 |      fill_value : float, defaults to NaN (missing)
 |
 |      Returns
 |      -------
 |      sp : SparseSeries
 |
 |  to_string(self, buf=None, na_rep='NaN', float_format=None, header=True, index=True, l
ength=False, dtype=False, name=False, max_rows=None)
 |      Render a string representation of the Series.
 |
 |      Parameters
 |      ----------
 |      buf : StringIO-like, optional
 |          buffer to write to
 |      na_rep : string, 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 : boolean, default True
 |          Add the Series header (index name)
 |      index : bool, optional
 |          Add index (row) labels, default True
 |      length : boolean, default False
 |          Add the Series length
 |      dtype : boolean, default False
 |          Add the Series dtype
 |      name : boolean, 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.
 |
 |      Returns
 |      -------
 |      formatted : string (if not buffer passed)
 |
 |  to_timestamp(self, freq=None, how='start', copy=True)
 |      Cast to datetimeindex of timestamps, at *beginning* of period.
 |
 |      Parameters
 |      ----------
 |      freq : string, default frequency of PeriodIndex
 |          Desired frequency
 |      how : {'s', 'e', 'start', 'end'}
 |          Convention for converting period to timestamp; start of period
 |          vs. end
 |
 |      Returns
 |      -------
 |      ts : Series with DatetimeIndex
 |
 |  transform(self, func, axis=0, *args, **kwargs)
 |      Call ``func`` on self producing a Series with transformed values
 |      and that has the same axis length as self.
 |
 |      .. versionadded:: 0.20.0
 |
 |      Parameters
 |      ----------
 |      func : function, str, list or dict
 |          Function to use for transforming 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.exp. 'sqrt']``
|          - dict of axis labels -> functions, function names or list of such.
|      axis : {0 or 'index'}
|          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.
|
|      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
|
|  truediv(self, other, level=None, fill_value=None, axis=0)
|      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 one of the inputs.
|
|      Parameters
|      ----------
|      other : Series or scalar value
|      fill_value : None or float value, default None (NaN)
|          Fill existing missing (NaN) values, and any new element needed for
|          successful Series alignment, with this value before computation.
|          If data in both corresponding Series locations is missing
|          the result will be missing
|      level : int or name
|          Broadcast across a level, matching Index values on the
|          passed MultiIndex level
|
|      Returns
|      -------
|
```

```
|      result : Series
|
|      See Also
|      --------
|      Series.rtruediv
|
|      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
|
|  unique(self)
|      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. 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
|          * Interval
|          * Sparse
|          * IntegerNA
|
|      See Also
|      --------
|      unique : Top-level unique method for any 1-d array-like object.
|      Index.unique : Return Index with unique values from an Index object.
|
|      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()
|      array(['2016-01-01T00:00:00.000000000'], 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 unordered Categorical will return categories in the order of
|      appearance.
```

```
|
|      >>> pd.Series(pd.Categorical(list('baabc'))).unique()
|      [b, a, c]
|      Categories (3, object): [b, a, c]
|
|      An ordered Categorical preserves the category ordering.
|
|      >>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
|      ...                           ordered=True)).unique()
|      [b, a, c]
|      Categories (3, object): [a < b < c]
|
|  unstack(self, level=-1, fill_value=None)
|      Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.
|      The level involved will automatically get sorted.
|
|      Parameters
|      ----------
|      level : int, string, or list of these, default last level
|          Level(s) to unstack, can pass level name
|      fill_value : replace NaN with this value if the unstack produces
|          missing values
|
|          .. versionadded:: 0.18.0
|
|      Returns
|      -------
|      unstacked : DataFrame
|
|      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)
|      Modify Series in place using non-NA values from passed
|      Series. Aligns on index.
|
|      Parameters
|      ----------
|      other : 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
|
|  valid(self, inplace=False, **kwargs)
|      Return Series without null values.
|
|      .. deprecated:: 0.23.0
|          Use :meth:`Series.dropna` instead.
|
|  var(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)
|      Return unbiased variance over requested axis.
|
|      Normalized by N-1 by default. This can be changed using the ddof argument
|
|      Parameters
|      ----------
|      axis : {index (0)}
|      skipna : boolean, default True
|          Exclude NA/null values. If an entire row/column is NA, the result
|          will be NA
|      level : int or level name, default None
|          If the axis is a MultiIndex (hierarchical), count along a
|          particular level, collapsing into a scalar
|      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 : boolean, default None
|          Include only float, int, boolean columns. If None, will attempt to use
|          everything, then use only numeric data. Not implemented for Series.
|
|      Returns
|      -------
|      var : scalar or Series (if level specified)
|
|  view(self, dtype=None)
|      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
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_array(arr, index=None, name=None, dtype=None, copy=False, fastpath=False) from b
uiltins.type
 |      Construct Series from array.
 |
 |      .. deprecated :: 0.23.0
 |          Use pd.Series(..) constructor instead.
 |
 |  from_csv(path, sep=',', parse_dates=True, header=None, index_col=0, encoding=None, in
fer_datetime_format=False) from builtins.type
 |      Read CSV file.
 |
 |      .. deprecated:: 0.21.0
 |          Use :func:`pandas.read_csv` instead.
 |
 |      It is preferable to use the more powerful :func:`pandas.read_csv`
 |      for most general purposes, but ``from_csv`` makes for an easy
 |      roundtrip to and from a file (the exact counterpart of
 |      ``to_csv``), especially with a time Series.
 |
 |      This method only differs from :func:`pandas.read_csv` in some defaults:
 |
 |      - `index_col` is ``0`` instead of ``None`` (take first column as index
```

```
|        by default)
|      - `header` is ``None`` instead of ``0`` (the first row is not used as
|        the column names)
|      - `parse_dates` is ``True`` instead of ``False`` (try parsing the index
|        as datetime by default)
|
|      With :func:`pandas.read_csv`, the option ``squeeze=True`` can be used
|      to return a Series like ``from_csv``.
|
|      Parameters
|      ----------
|      path : string file path or file handle / StringIO
|      sep : string, default ','
|          Field delimiter
|      parse_dates : boolean, default True
|          Parse dates. Different default from read_table
|      header : int, default None
|          Row to use as header (skip prior rows)
|      index_col : int or sequence, default 0
|          Column to use for index. If a sequence is given, a MultiIndex
|          is used. Different default from read_table
|      encoding : string, optional
|          a string representing the encoding to use if the contents are
|          non-ascii, for python versions prior to 3
|      infer_datetime_format : boolean, default False
|          If True and `parse_dates` is True for a column, try to infer the
|          datetime format based on the first datetime string. If the format
|          can be inferred, there often will be a large parsing speed-up.
|
|      Returns
|      -------
|      y : Series
|
|      See Also
|      --------
|      read_csv
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  asobject
|      Return object Series which contains boxed values.
|
|      .. deprecated :: 0.23.0
|
|         Use ``astype(object)`` instead.
|
|      *this is an internal non-public method*
|
|  axes
|      Return a list of the row axis labels.
|
|  dtype
|      Return the dtype object of the underlying data.
|
|  dtypes
|      Return the dtype object of the underlying data.
|
|  ftype
|      Return if the data is sparse|dense.
|
|  ftypes
|      Return if the data is sparse|dense.
|
|  hasnans
|      Return if I have any nans; enables various perf speedups.
|
|  imag
|      Return imag value of vector.
|
|  index
|      The index (axis labels) of the Series.
```

```
|
|  name
|      Return name of the Series.
|
|  real
|      Return the real value of vector.
|
|  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
|      -------
|      arr : 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 and other attributes defined here:
|
|  cat = <class 'pandas.core.arrays.categorical.CategoricalAccessor'>
|      Accessor object for categorical properties of the Series values.
|
|      Be aware that assigning to `categories` is a inplace operation, while all
|      methods return new categorical data per default (but can be called with
|      `inplace=True`).
|
|      Parameters
|      ----------
|      data : Series or CategoricalIndex
|
|      Examples
|      --------
|      >>> s.cat.categories
|      >>> s.cat.categories = list('abc')
|      >>> s.cat.rename_categories(list('cab'))
|      >>> s.cat.reorder_categories(list('cab'))
|      >>> s.cat.add_categories(['d','e'])
|      >>> s.cat.remove_categories(['d'])
|      >>> s.cat.remove_unused_categories()
|      >>> s.cat.set_categories(list('abcde'))
|      >>> s.cat.as_ordered()
|      >>> s.cat.as_unordered()
|
```

```
|  dt = <class 'pandas.core.indexes.accessors.CombinedDatetimelikePropert...
|      Accessor object for datetimelike properties of the Series values.
|
|      Examples
|      --------
|      >>> s.dt.hour
|      >>> s.dt.second
|      >>> s.dt.quarter
|
|      Returns a Series indexed like the original Series.
|      Raises TypeError if the Series does not contain datetimelike values.
|
|  plot = <class 'pandas.plotting._core.SeriesPlotMethods'>
|      Series plotting accessor and method.
|
|      Examples
|      --------
|      >>> s.plot.line()
|      >>> s.plot.bar()
|      >>> s.plot.hist()
|
|      Plotting methods can also be accessed by calling the accessor as a method
|      with the ``kind`` argument:
|      ``s.plot(kind='line')`` is equivalent to ``s.plot.line()``
|
|  sparse = <class 'pandas.core.arrays.sparse.SparseAccessor'>
|      Accessor for SparseSparse from other sparse matrix data types.
|
|  str = <class 'pandas.core.strings.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.str.split('_')
|      >>> s.str.replace('_', '')
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.base.IndexOpsMixin:
|
|  __iter__(self)
|      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)
|
|  factorize(self, sort=False, na_sentinel=-1)
|      Encode the object as an enumerated type or categorical variable.
|
|      This method is useful for obtaining a numeric representation of an
|      array when all that matters is identifying distinct values. `factorize`
|      is available as both a top-level function :func:`pandas.factorize`,
|      and as a method :meth:`Series.factorize` and :meth:`Index.factorize`.
|
|      Parameters
|      ----------
|      sort : boolean, default False
|          Sort `uniques` and shuffle `labels` to maintain the
|          relationship.
|
|      na_sentinel : int, default -1
|          Value to mark "not found".
|
|      Returns
|      -------
|      labels : ndarray
|          An integer ndarray that's an indexer into `uniques`.
|          ``uniques.take(labels)`` will have the same values as `values`.
|      uniques : ndarray, Index, or Categorical
|          The unique valid values. When `values` is Categorical, `uniques`
```

```
|          is a Categorical. When `values` is some other pandas object, an
|          `Index` is returned. Otherwise, a 1-D ndarray is returned.
|
|          .. note ::
|
|             Even if there's a missing value in `values`, `uniques` will
|             *not* contain an entry for it.
|
|      See Also
|      --------
|      cut : Discretize continuous-valued array.
|      unique : Find the unique value in an array.
|
|      Examples
|      --------
|      These examples all show factorize as a top-level method like
|      ``pd.factorize(values)``. The results are identical for methods like
|      :meth:`Series.factorize`.
|
|      >>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
|      >>> labels
|      array([0, 0, 1, 2, 0])
|      >>> uniques
|      array(['b', 'a', 'c'], dtype=object)
|
|      With ``sort=True``, the `uniques` will be sorted, and `labels` will be
|      shuffled so that the relationship is the maintained.
|
|      >>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
|      >>> labels
|      array([1, 1, 0, 2, 1])
|      >>> uniques
|      array(['a', 'b', 'c'], dtype=object)
|
|      Missing values are indicated in `labels` with `na_sentinel`
|      (``-1`` by default). Note that missing values are never
|      included in `uniques`.
|
|      >>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
|      >>> labels
|      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'])
|      >>> labels, uniques = pd.factorize(cat)
|      >>> labels
|      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'])
|      >>> labels, uniques = pd.factorize(cat)
|      >>> labels
|      array([0, 0, 1])
|      >>> uniques
|      Index(['a', 'c'], dtype='object')
|
|  item(self)
|      Return the first element of the underlying data as a python scalar.
|
```

```
|  nunique(self, dropna=True)
|      Return number of unique elements in the object.
|
|      Excludes NA values by default.
|
|      Parameters
|      ----------
|      dropna : boolean, default True
|          Don't include NaN in the count.
|
|      Returns
|      -------
|      nunique : int
|
|  to_list = 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)
|
|      See Also
|      --------
|      numpy.ndarray.tolist
|
|  to_numpy(self, dtype=None, copy=False)
|      A NumPy ndarray representing the values in this Series or Index.
|
|      .. versionadded:: 0.24.0
|
|
|      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 a 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.
|
|      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', freq='D'),
 |             Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
 |            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)
 |
 |      See Also
 |      --------
 |      numpy.ndarray.tolist
 |
 |  transpose(self, *args, **kwargs)
 |      Return the transpose, which is by definition self.
 |
 |  value_counts(self, normalize=False, sort=True, ascending=False, bins=None, dropna=Tru
e)
 |      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 : boolean, default False
 |          If True then the object returned will contain the relative
 |          frequencies of the unique values.
 |      sort : boolean, default True
 |          Sort by values.
 |      ascending : boolean, default False
 |          Sort in ascending order.
 |      bins : integer, optional
 |          Rather than count values, group them into half-open bins,
 |          a convenience for ``pd.cut``, only works with numeric data.
 |      dropna : boolean, default True
```

```
|          Don't include counts of NaN.
|
|      Returns
|      -------
|      counts : Series
|
|      See Also
|      --------
|      Series.count: Number of non-NA elements in a Series.
|      DataFrame.count: Number of non-NA elements in a DataFrame.
|
|      Examples
|      --------
|      >>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
|      >>> index.value_counts()
|      3.0    2
|      4.0    1
|      2.0    1
|      1.0    1
|      dtype: int64
|
|      With `normalize` set to `True`, returns the relative frequency by
|      dividing all values by the sum of values.
|
|      >>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
|      >>> s.value_counts(normalize=True)
|      3.0    0.4
|      4.0    0.2
|      2.0    0.2
|      1.0    0.2
|      dtype: float64
|
|      **bins**
|
|      Bins can be useful for going from a continuous variable to a
|      categorical variable; instead of counting unique
|      apparitions of values, divide the index in the specified
|      number of half-open bins.
|
|      >>> s.value_counts(bins=3)
|      (2.0, 3.0]      2
|      (0.996, 2.0]    2
|      (3.0, 4.0]      1
|      dtype: int64
|
|      **dropna**
|
|      With `dropna` set to `False` we can also see NaN index values.
|
|      >>> s.value_counts(dropna=False)
|      3.0    2
|      NaN    1
|      4.0    1
|      2.0    1
|      1.0    1
|      dtype: int64
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pandas.core.base.IndexOpsMixin:
|
|  T
|      Return the transpose, which is by definition self.
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  array
|      The ExtensionArray of the data backing this Series or Index.
|
```

```
|      .. versionadded:: 0.24.0
|
|      Returns
|      -------
|      array : 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
|      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]
|
|  base
|      Return the base object if the memory of the underlying data is shared.
|
|  data
|      Return the data pointer of the underlying data.
|
|  empty
|
|  flags
|      Return the ndarray.flags for the underlying data.
|
|  is_monotonic
|      Return boolean if values in the object are
```

```
|      monotonic_increasing.
|
|      .. versionadded:: 0.19.0
|
|      Returns
|      -------
|      is_monotonic : boolean
|
|  is_monotonic_decreasing
|      Return boolean if values in the object are
|      monotonic_decreasing.
|
|      .. versionadded:: 0.19.0
|
|      Returns
|      -------
|      is_monotonic_decreasing : boolean
|
|  is_monotonic_increasing
|      Return boolean if values in the object are
|      monotonic_increasing.
|
|      .. versionadded:: 0.19.0
|
|      Returns
|      -------
|      is_monotonic : boolean
|
|  is_unique
|      Return boolean if values in the object are unique.
|
|      Returns
|      -------
|      is_unique : boolean
|
|  itemsize
|      Return the size of the dtype of the item of the underlying data.
|
|  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.
|
|  size
|      Return the number of elements in the underlying data.
|
|  strides
|      Return the strides of the underlying data.
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from pandas.core.base.IndexOpsMixin:
|
|  __array_priority__ = 1000
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.generic.NDFrame:
|
|  __abs__(self)
|
|  __bool__ = __nonzero__(self)
|
|  __contains__(self, key)
|      True if the key is in the info axis
|
|  __copy__(self, deep=True)
|
|  __deepcopy__(self, memo=None)
|      Parameters
```

```
|      ----------
|      memo, default None
|          Standard signature. Unused
|
|  __delitem__(self, key)
|      Delete item
|
|  __finalize__(self, other, method=None, **kwargs)
|      Propagate metadata from other to self.
|
|      Parameters
|      ----------
|      other : the object from which to get the attributes that we are going
|          to propagate
|      method : optional, a passed method name ; possibly to take different
|          types of propagation actions based on this
|
|  __getattr__(self, name)
|      After regular attribute access, try looking up the name
|      This allows simpler access to columns for interactive use.
|
|  __getstate__(self)
|
|  __hash__(self)
|      Return hash(self).
|
|  __invert__(self)
|
|  __neg__(self)
|
|  __nonzero__(self)
|
|  __pos__(self)
|
|  __round__(self, decimals=0)
|
|  __setattr__(self, name, value)
|      After regular attribute access, try setting the name
|      This allows simpler access to columns for interactive use.
|
|  __setstate__(self, state)
|
|  abs(self)
|      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, prefix)
|      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.
|
|      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, suffix)
|      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.
|
|      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
|
|  as_blocks(self, copy=True)
|      Convert the frame to a dict of dtype -> Constructor Types that each has
```

```
|      a homogeneous dtype.
|
|      .. deprecated:: 0.21.0
|
|      NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in
|            as_matrix)
|
|      Parameters
|      ----------
|      copy : boolean, default True
|
|      Returns
|      -------
|      values : a dict of dtype -> Constructor Types
|
|  as_matrix(self, columns=None)
|      Convert the frame to its Numpy-array representation.
|
|      .. deprecated:: 0.23.0
|          Use :meth:`DataFrame.values` instead.
|
|      Parameters
|      ----------
|      columns : list, optional, default:None
|          If None, return all columns, otherwise, returns specified columns.
|
|      Returns
|      -------
|      values : ndarray
|          If the caller is heterogeneous and contains booleans or objects,
|          the result will be of dtype=object. See Notes.
|
|      See Also
|      --------
|      DataFrame.values
|
|      Notes
|      -----
|      Return is NOT a Numpy-matrix, rather, a Numpy-array.
|
|      The dtype will be a lower-common-denominator dtype (implicit
|      upcasting); that is to say if the dtypes (even of numeric types)
|      are mixed, the one that accommodates all will be chosen. Use this
|      with care if you are not dealing with the blocks.
|
|      e.g. If the dtypes are float16 and float32, dtype will be upcast to
|      float32.  If dtypes are int32 and uint8, dtype will be upcase to
|      int32. By numpy.find_common_type convention, mixing int64 and uint64
|      will result in a float64 dtype.
|
|      This method is provided for backwards compatibility. Generally,
|      it is recommended to use '.values'.
|
|  asfreq(self, freq, method=None, how=None, normalize=False, fill_value=None)
|      Convert TimeSeries to specified frequency.
|
|      Optionally provide filling method to pad/backfill missing values.
|
|      Returns the original data conformed to a new index with the specified
|      frequency. ``resample`` is more appropriate if an operation, such as
|      summarization, is necessary to represent the data at the new frequency.
|
|      Parameters
|      ----------
|      freq : DateOffset object, 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).
|
|          .. versionadded:: 0.20.0
|
|      Returns
|      -------
|      converted : same type as caller
|
|      See Also
|      --------
|      reindex
|
|      Notes
|      -----
|      To learn more about the frequency strings, please see `this link
|      <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>`__.
|
|      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
|
|  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`)
|
|      .. versionadded:: 0.19.0 For DataFrame
|
|      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
|
|         * 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
|
|      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   30.0 NaN
|      2018-02-27 09:04:30   40.0 NaN
|
|  astype(self, dtype, copy=True, errors='raise', **kwargs)
|      Cast a pandas object to a specified dtype ``dtype``.
|
|      Parameters
|      ----------
|      dtype : data type, or dict of column name -> data type
|          Use a numpy.dtype or Python type to cast entire pandas object to
|          the same type. Alternatively, use {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
|
|          .. versionadded:: 0.20.0
|
|      kwargs : keyword arguments to pass on to the constructor
|
|      Returns
|      -------
|      casted : 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.
|
|      Examples
|      --------
|      >>> 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, int64): [1, 2]
|
|      Convert to ordered categorical type with custom ordering:
|
|      >>> cat_dtype = pd.api.types.CategoricalDtype(
|      ...                     categories=[2, 1], ordered=True)
|      >>> ser.astype(cat_dtype)
|      0    1
|      1    2
|      dtype: category
|      Categories (2, int64): [2 < 1]
|
|      Note that using ``copy=False`` and changing data on a new
|      pandas object may propagate changes:
|
|      >>> s1 = pd.Series([1,2])
|      >>> s2 = s1.astype('int64', copy=False)
|      >>> s2[0] = 10
|      >>> s1  # note that s1[0] has changed too
|      0    10
|      1     2
|      dtype: int64
|
|  at_time(self, time, asof=False, axis=None)
|      Select values at particular time of day (e.g. 9:30AM).
|
|      Parameters
|      ----------
|      time : datetime.time or string
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|
|          .. versionadded:: 0.24.0
|
|      Returns
|      -------
|      values_at_time : same type as caller
|
|      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
|
|  between_time(self, start_time, end_time, include_start=True, include_end=True, axis=N
one)
```

```
|      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 string
|      end_time : datetime.time or string
|      include_start : boolean, default True
|      include_end : boolean, default True
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|
|          .. versionadded:: 0.24.0
|
|      Returns
|      -------
|      values_between_time : same type as caller
|
|      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
|
|  bfill(self, axis=None, inplace=False, limit=None, downcast=None)
|      Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.
|
|  bool(self)
|      Return the bool of a single element PandasObject.
|
|      This must be a boolean scalar value, either True or False.  Raise a
|      ValueError if the PandasObject does not have exactly 1 element, or that
|      element is not boolean
|
|  clip(self, lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)
|      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.
|      upper : float or array_like, default None
|          Maximum threshold value. All values above this
|          threshold will be set to it.
|      axis : int or string axis name, optional
|          Align object with lower and upper along the given axis.
|      inplace : boolean, default False
|          Whether to perform the operation in place on the data.
|
|          .. versionadded:: 0.21.0
|      *args, **kwargs
|          Additional keywords have no effect but might be accepted
|          for compatibility with numpy.
|
|      Returns
|      -------
|      Series or DataFrame
|          Same type as calling object with the values outside the
|          clip boundaries replaced
|
|      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
|
|  clip_lower(self, threshold, axis=None, inplace=False)
|      Trim values below a given threshold.
|
|      .. deprecated:: 0.24.0
|          Use clip(lower=threshold) instead.
|
```

```
|       Elements below the `threshold` will be changed to match the
|       `threshold` value(s). Threshold can be a single value or an array,
|       in the latter case it performs the truncation element-wise.
|
|       Parameters
|       ----------
|       threshold : numeric or array-like
|           Minimum value allowed. All values below threshold will be set to
|           this value.
|
|           * float : every value is compared to `threshold`.
|           * array-like : The shape of `threshold` should match the object
|             it's compared to. When `self` is a Series, `threshold` should be
|             the length. When `self` is a DataFrame, `threshold` should 2-D
|             and the same shape as `self` for ``axis=None``, or 1-D and the
|             same length as the axis being compared.
|
|       axis : {0 or 'index', 1 or 'columns'}, default 0
|           Align `self` with `threshold` along the given axis.
|
|       inplace : boolean, default False
|           Whether to perform the operation in place on the data.
|
|           .. versionadded:: 0.21.0
|
|       Returns
|       -------
|       Series or DataFrame
|           Original data with values trimmed.
|
|       See Also
|       --------
|       Series.clip : General purpose method to trim Series values to given
|           threshold(s).
|       DataFrame.clip : General purpose method to trim DataFrame values to
|           given threshold(s).
|
|       Examples
|       --------
|
|       Series single threshold clipping:
|
|       >>> s = pd.Series([5, 6, 7, 8, 9])
|       >>> s.clip(lower=8)
|       0    8
|       1    8
|       2    8
|       3    8
|       4    9
|       dtype: int64
|
|       Series clipping element-wise using an array of thresholds. `threshold`
|       should be the same length as the Series.
|
|       >>> elemwise_thresholds = [4, 8, 7, 2, 5]
|       >>> s.clip(lower=elemwise_thresholds)
|       0    5
|       1    8
|       2    7
|       3    8
|       4    9
|       dtype: int64
|
|       DataFrames can be compared to a scalar.
|
|       >>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
|       >>> df
|          A  B
|       0  1  2
|       1  3  4
|       2  5  6
|
```

```
|      >>> df.clip(lower=3)
|         A  B
|      0  3  3
|      1  3  4
|      2  5  6
|
|      Or to an array of values. By default, `threshold` should be the same
|      shape as the DataFrame.
|
|      >>> df.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))
|         A  B
|      0  3  4
|      1  3  4
|      2  6  6
|
|      Control how `threshold` is broadcast with `axis`. In this case
|      `threshold` should be the same length as the axis specified by
|      `axis`.
|
|      >>> df.clip(lower=[3, 3, 5], axis='index')
|         A  B
|      0  3  3
|      1  3  4
|      2  5  6
|
|      >>> df.clip(lower=[4, 5], axis='columns')
|         A  B
|      0  4  5
|      1  4  5
|      2  5  6
|
|  clip_upper(self, threshold, axis=None, inplace=False)
|      Trim values above a given threshold.
|
|      .. deprecated:: 0.24.0
|          Use clip(upper=threshold) instead.
|
|      Elements above the `threshold` will be changed to match the
|      `threshold` value(s). Threshold can be a single value or an array,
|      in the latter case it performs the truncation element-wise.
|
|      Parameters
|      ----------
|      threshold : numeric or array-like
|          Maximum value allowed. All values above threshold will be set to
|          this value.
|
|          * float : every value is compared to `threshold`.
|          * array-like : The shape of `threshold` should match the object
|            it's compared to. When `self` is a Series, `threshold` should be
|            the length. When `self` is a DataFrame, `threshold` should 2-D
|            and the same shape as `self` for ``axis=None``, or 1-D and the
|            same length as the axis being compared.
|
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|          Align object with `threshold` along the given axis.
|      inplace : boolean, default False
|          Whether to perform the operation in place on the data.
|
|          .. versionadded:: 0.21.0
|
|      Returns
|      -------
|      Series or DataFrame
|          Original data with values trimmed.
|
|      See Also
|      --------
|      Series.clip : General purpose method to trim Series values to given
|          threshold(s).
|      DataFrame.clip : General purpose method to trim DataFrame values to
|          given threshold(s).
```

```
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series([1, 2, 3, 4, 5])
 |      >>> s
 |      0    1
 |      1    2
 |      2    3
 |      3    4
 |      4    5
 |      dtype: int64
 |
 |      >>> s.clip(upper=3)
 |      0    1
 |      1    2
 |      2    3
 |      3    3
 |      4    3
 |      dtype: int64
 |
 |      >>> elemwise_thresholds = [5, 4, 3, 2, 1]
 |      >>> elemwise_thresholds
 |      [5, 4, 3, 2, 1]
 |
 |      >>> s.clip(upper=elemwise_thresholds)
 |      0    1
 |      1    2
 |      2    3
 |      3    2
 |      4    1
 |      dtype: int64
 |
 |  convert_objects(self, convert_dates=True, convert_numeric=False, convert_timedeltas=T
rue, copy=True)
 |      Attempt to infer better dtype for object columns.
 |
 |      .. deprecated:: 0.21.0
 |
 |      Parameters
 |      ----------
 |      convert_dates : boolean, default True
 |          If True, convert to date where possible. If 'coerce', force
 |          conversion, with unconvertible values becoming NaT.
 |      convert_numeric : boolean, default False
 |          If True, attempt to coerce to numbers (including strings), with
 |          unconvertible values becoming NaN.
 |      convert_timedeltas : boolean, default True
 |          If True, convert to timedelta where possible. If 'coerce', force
 |          conversion, with unconvertible values becoming NaT.
 |      copy : boolean, default True
 |          If True, return a copy even if no copy is necessary (e.g. no
 |          conversion was done). Note: This is meant for internal use, and
 |          should not be confused with inplace.
 |
 |      Returns
 |      -------
 |      converted : same as input object
 |
 |      See Also
 |      --------
 |      to_datetime : Convert argument to datetime.
 |      to_timedelta : Convert argument to timedelta.
 |      to_numeric : Convert argument to numeric type.
 |
 |  copy(self, deep=True)
 |      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
|       -------
|       copy : Series, DataFrame or Panel
|           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.
|
|       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, percentiles=None, include=None, exclude=None)
|      Generate descriptive statistics 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(include=['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 obersvations.
|       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
|       unique                    2
|       top       2010-01-01 00:00:00
|       freq                      2
|       first     2000-01-01 00:00:00
|       last      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')
|               categorical  numeric object
|       count            3      3.0      3
|       unique           3      NaN      3
|       top              f      NaN      c
|       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=[np.object])
```

```
|             object
|      count       3
|      unique      3
|      top         c
|      freq        1
|
|      Including only categorical columns from a ``DataFrame`` description.
|
|      >>> df.describe(include=['category'])
|             categorical
|      count             3
|      unique            3
|      top               f
|      freq              1
|
|      Excluding numeric columns from a ``DataFrame`` description.
|
|      >>> df.describe(exclude=[np.number])
|             categorical object
|      count             3      3
|      unique            3      3
|      top               f      c
|      freq              1      1
|
|      Excluding object columns from a ``DataFrame`` description.
|
|      >>> df.describe(exclude=[np.object])
|             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, level, axis=0)
|      Return DataFrame with requested index / column level(s) removed.
|
|      .. versionadded:: 0.24.0
|
|      Parameters
|      ----------
|      level : int, str, or list-like
|          If a string is given, must be the name of a level
|          If list-like, elements must be names or positional indexes
|          of levels.
|
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|
|      Returns
|      -------
|      DataFrame.droplevel()
|
|      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('level2', axis=1)
|      level_1   c   d
|      a b
|      1 2      3   4
|      5 6      7   8
|      9 10    11  12
|
|  equals(self, other)
|      Test whether two objects contain the same elements.
|
|      This function allows two Series or DataFrames to be compared against
|      each other to see if they have the same shape and elements. NaNs in
|      the same location are considered equal. The column headers do not
|      need to have the same type, but the elements within the columns must
|      be the same dtype.
|
|      Parameters
|      ----------
|      other : Series or DataFrame
|          The other Series or DataFrame to be compared with the first.
|
|      Returns
|      -------
|      bool
|          True if all elements are the same in both objects, False
|          otherwise.
|
|      See Also
|      --------
|      Series.eq : Compare two Series objects of the same length
|          and return a Series where each element is True if the element
|          in each Series is equal, False otherwise.
|      DataFrame.eq : Compare two DataFrame objects of the same shape and
|          return a DataFrame where each element is True if the respective
|          element in each DataFrame is equal, False otherwise.
|      assert_series_equal : Return True if left and right Series are equal,
|          False otherwise.
|      assert_frame_equal : Return True if left and right DataFrames are
|          equal, False otherwise.
|      numpy.array_equal : Return True if two arrays have the same shape
|          and elements, False otherwise.
|
|      Notes
|      -----
|      This function requires that the elements have the same dtype as their
|      respective elements in the other Series or DataFrame. However, the
|      column labels do not need to have the same type, as long as they are
|      still considered equal.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({1: [10], 2: [20]})
|      >>> df
|          1   2
|      0  10  20
|
|      DataFrames df and exactly_equal have the same types and values for
|      their elements and column labels, which will return True.
```

```
|      >>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
|      >>> exactly_equal
|          1   2
|      0  10  20
|      >>> df.equals(exactly_equal)
|      True
|
|      DataFrames df and different_column_type have the same element
|      types and values, but have different types for the column labels,
|      which will still return True.
|
|      >>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
|      >>> different_column_type
|         1.0  2.0
|      0   10   20
|      >>> df.equals(different_column_type)
|      True
|
|      DataFrames df and different_data_type have different types for the
|      same values for their elements, and will return False even though
|      their column labels are the same values and types.
|
|      >>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
|      >>> different_data_type
|            1     2
|      0  10.0  20.0
|      >>> df.equals(different_data_type)
|      False
|
|  ffill(self, axis=None, inplace=False, limit=None, downcast=None)
|      Synonym for :meth:`DataFrame.fillna` with ``method='ffill'``.
|
|  filter(self, items=None, like=None, regex=None, axis=None)
|      Subset rows or columns of dataframe according to labels in
|      the specified index.
|
|      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
|          List of axis to restrict to (must not all be present).
|      like : string
|          Keep axis where "arg in col == True".
|      regex : string (regular expression)
|          Keep axis with re.search(regex, col) == True.
|      axis : int or string axis name
|          The axis to filter on.  By default this is the info axis,
|          'index' for Series, 'columns' for DataFrame.
|
|      Returns
|      -------
|      same type as input object
|
|      See Also
|      --------
|      DataFrame.loc
|
|      Notes
|      -----
|      The ``items``, ``like``, and ``regex`` parameters are
|      enforced to be mutually exclusive.
|
|      ``axis`` defaults to the info axis that is used when indexing
|      with ``[]``.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame(np.array(([1,2,3], [4,5,6])),
|      ...                   index=['mouse', 'rabbit'],
```

```
|      ...                   columns=['one', 'two', 'three'])
|
|      >>> # select columns by name
|      >>> df.filter(items=['one', 'three'])
|                one  three
|      mouse     1      3
|      rabbit    4      6
|
|      >>> # select columns by regular expression
|      >>> df.filter(regex='e$', axis=1)
|                one  three
|      mouse     1      3
|      rabbit    4      6
|
|      >>> # select rows containing 'bbi'
|      >>> df.filter(like='bbi', axis=0)
|                one  two  three
|      rabbit    4    5      6
|
|  first(self, offset)
|      Convenience method for subsetting initial periods of time series data
|      based on a date offset.
|
|      Parameters
|      ----------
|      offset : string, DateOffset, dateutil.relativedelta
|
|      Returns
|      -------
|      subset : same type as 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 calender 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)
|      Return index for first non-NA/null value.
|
|      Returns
|      --------
|      scalar : type of index
|
|      Notes
```

```
|      --------
|      If all elements are non-NA/null, returns None.
|      Also returns None for empty NDFrame.
|
|  get(self, key, default=None)
|      Get item from object for given key (DataFrame column, Panel slice,
|      etc.). Returns default value if not found.
|
|      Parameters
|      ----------
|      key : object
|
|      Returns
|      -------
|      value : same type as items contained in object
|
|  get_dtype_counts(self)
|      Return counts of unique dtypes in this object.
|
|      Returns
|      -------
|      dtype : Series
|          Series with the count of columns with each dtype.
|
|      See Also
|      --------
|      dtypes : Return the dtypes in this object.
|
|      Examples
|      --------
|      >>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
|      >>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
|      >>> df
|        str  int  float
|      0   a    1    1.0
|      1   b    2    2.0
|      2   c    3    3.0
|
|      >>> df.get_dtype_counts()
|      float64    1
|      int64      1
|      object     1
|      dtype: int64
|
|  get_ftype_counts(self)
|      Return counts of unique ftypes in this object.
|
|      .. deprecated:: 0.23.0
|
|      This is useful for SparseDataFrame or for DataFrames containing
|      sparse arrays.
|
|      Returns
|      -------
|      dtype : Series
|          Series with the count of columns with each type and
|          sparsity (dense/sparse)
|
|      See Also
|      --------
|      ftypes : Return ftypes (indication of sparse/dense and dtype) in
|          this object.
|
|      Examples
|      --------
|      >>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
|      >>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
|      >>> df
|        str  int  float
|      0   a    1    1.0
|      1   b    2    2.0
|      2   c    3    3.0
```

```
 |
 |      >>> df.get_ftype_counts()  # doctest: +SKIP
 |      float64:dense    1
 |      int64:dense      1
 |      object:dense     1
 |      dtype: int64
 |
 |  groupby(self, by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True,
squeeze=False, observed=False, **kwargs)
 |      Group DataFrame or Series using a mapper or by a Series of columns.
 |
 |      A groupby operation involves some combination of splitting the
 |      object, applying a function, and combining the results. This can be
 |      used to group large amounts of data and compute operations on these
 |      groups.
 |
 |      Parameters
 |      ----------
 |      by : mapping, function, label, or list of labels
 |          Used to determine the groups for the groupby.
 |          If ``by`` is a function, it's called on each value of the object's
 |          index. If a dict or Series is passed, the Series or dict VALUES
 |          will be used to determine the groups (the Series' values are first
 |          aligned; see ``.align()`` method). If an ndarray is passed, the
 |          values are used as-is determine the groups. A label or list of
 |          labels may be passed to group by the columns in ``self``. Notice
 |          that a tuple is interpreted a (single) key.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Split along rows (0) or columns (1).
 |      level : int, level name, or sequence of such, default None
 |          If the axis is a MultiIndex (hierarchical), group by a particular
 |          level or levels.
 |      as_index : bool, default True
 |          For aggregated output, return object with group labels as the
 |          index. Only relevant for DataFrame input. as_index=False is
 |          effectively "SQL-style" grouped output.
 |      sort : bool, default True
 |          Sort group keys. Get better performance by turning this off.
 |          Note this does not influence the order of observations within each
 |          group. Groupby preserves the order of rows within each group.
 |      group_keys : bool, default True
 |          When calling apply, add group keys to index to identify pieces.
 |      squeeze : bool, default False
 |          Reduce the dimensionality of the return type if possible,
 |          otherwise return a consistent type.
 |      observed : bool, default False
 |          This only applies if any of the groupers are Categoricals.
 |          If True: only show observed values for categorical groupers.
 |          If False: show all values for categorical groupers.
 |
 |          .. versionadded:: 0.23.0
 |
 |      **kwargs
 |          Optional, only accepts keyword argument 'mutated' and is passed
 |          to groupby.
 |
 |      Returns
 |      -------
 |      DataFrameGroupBy or SeriesGroupBy
 |          Depends on the calling object and returns 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
 |      <http://pandas.pydata.org/pandas-docs/stable/groupby.html>`_ for more.
 |
```

```
|      Examples
|      --------
|      >>> df = pd.DataFrame({'Animal' : ['Falcon', 'Falcon',
|      ...                                 'Parrot', 'Parrot'],
|      ...                    'Max Speed' : [380., 370., 24., 26.]})
|      >>> df
|         Animal  Max Speed
|      0  Falcon      380.0
|      1  Falcon      370.0
|      2  Parrot       24.0
|      3  Parrot       26.0
|      >>> df.groupby(['Animal']).mean()
|              Max Speed
|      Animal
|      Falcon      375.0
|      Parrot       25.0
|
|      **Hierarchical Indexes**
|
|      We can groupby different levels of a hierarchical index
|      using the `level` parameter:
|
|      >>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
|      ...           ['Capitve', 'Wild', 'Capitve', 'Wild']]
|      >>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
|      >>> df = pd.DataFrame({'Max Speed' : [390., 350., 30., 20.]},
|      ...                    index=index)
|      >>> df
|                      Max Speed
|      Animal Type
|      Falcon Capitve      390.0
|             Wild         350.0
|      Parrot Capitve       30.0
|             Wild          20.0
|      >>> df.groupby(level=0).mean()
|              Max Speed
|      Animal
|      Falcon      370.0
|      Parrot       25.0
|      >>> df.groupby(level=1).mean()
|               Max Speed
|      Type
|      Capitve      210.0
|      Wild         185.0
|
|  head(self, n=5)
|      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.
|
|      Parameters
|      ----------
|      n : int, default 5
|          Number of rows to select.
|
|      Returns
|      -------
|      obj_head : 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
|
|  infer_objects(self)
|      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.
|
|      .. versionadded:: 0.21.0
|
|      Returns
|      -------
|      converted : same type as input object
|
|      See Also
|      --------
|      to_datetime : Convert argument to datetime.
|      to_timedelta : Convert argument to timedelta.
|      to_numeric : Convert argument to numeric type.
|
|      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
|
|  interpolate(self, method='linear', axis=0, limit=None, inplace=False, limit_direction
='forward', limit_area=None, downcast=None, **kwargs)
|      Interpolate values according to different methods.
|
|      Please note that only ``method='linear'`` is supported for
|      DataFrame/Series with a MultiIndex.
```

```
|
|       Parameters
|       ----------
|       method : str, default 'linear'
|           Interpolation technique to use. One of:
|
|           * 'linear': Ignore the index and treat the values as equally
|             spaced. This is the only method supported on MultiIndexes.
|           * 'time': Works on daily and higher resolution data to interpolate
|             given length of interval.
|           * 'index', 'values': use the actual numerical values of the index.
|           * 'pad': Fill in NaNs using existing values.
|           * 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline',
|             'barycentric', 'polynomial': Passed to
|             `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline'
|             require that you also specify an `order` (int),
|             e.g. ``df.interpolate(method='polynomial', order=4)``.
|             These use the numerical values of the index.
|           * 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima':
|             Wrappers around the SciPy interpolation methods of similar
|             names. See `Notes`.
|           * 'from_derivatives': Refers to
|             `scipy.interpolate.BPoly.from_derivatives` which
|             replaces 'piecewise_polynomial' interpolation method in
|             scipy 0.18.
|
|           .. versionadded:: 0.18.1
|
|              Added support for the 'akima' method.
|              Added interpolate method 'from_derivatives' which replaces
|              'piecewise_polynomial' in SciPy 0.18; backwards-compatible with
|              SciPy < 0.18
|
|       axis : {0 or 'index', 1 or 'columns', None}, default None
|           Axis to interpolate along.
|       limit : int, optional
|           Maximum number of consecutive NaNs to fill. Must be greater than
|           0.
|       inplace : bool, default False
|           Update the data in place if possible.
|       limit_direction : {'forward', 'backward', 'both'}, default 'forward'
|           If limit is specified, consecutive NaNs will be filled in this
|           direction.
|       limit_area : {`None`, 'inside', 'outside'}, default None
|           If limit is specified, consecutive NaNs will be filled with this
|           restriction.
|
|           * ``None``: No fill restriction.
|           * 'inside': Only fill NaNs surrounded by valid values
|             (interpolate).
|           * 'outside': Only fill NaNs outside valid values (extrapolate).
|
|           .. versionadded:: 0.21.0
|
|       downcast : optional, 'infer' or None, defaults to None
|           Downcast dtypes if possible.
|       **kwargs
|           Keyword arguments to pass on to the interpolating function.
|
|       Returns
|       -------
|       Series or DataFrame
|           Returns the same object type as the caller, interpolated at
|           some or all ``NaN`` values
|
|       See Also
|       --------
|       fillna : Fill missing values using different methods.
|       scipy.interpolate.Akima1DInterpolator : Piecewise cubic polynomials
|           (Akima interpolator).
|       scipy.interpolate.BPoly.from_derivatives : Piecewise polynomial in the
|           Bernstein basis.
```

```
|      scipy.interpolate.interp1d : Interpolate a 1-D function.
|      scipy.interpolate.KroghInterpolator : Interpolate polynomial (Krogh
|          interpolator).
|      scipy.interpolate.PchipInterpolator : PCHIP 1-d monotonic cubic
|          interpolation.
|      scipy.interpolate.CubicSpline : Cubic spline data interpolator.
|
|      Notes
|      -----
|      The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima'
|      methods are wrappers around the respective SciPy implementations of
|      similar names. These use the actual numerical values of the index.
|      For more information on their behavior, see the
|      `SciPy documentation
|      <http://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-interpolat
ion>`__
|      and `SciPy tutorial
|      <http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html>`__.
|
|      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 befofe 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
|
|  last(self, offset)
|      Convenience method for subsetting final periods of time series data
|      based on a date offset.
|
|      Parameters
|      ----------
|      offset : string, DateOffset, dateutil.relativedelta
|
|      Returns
|      -------
|      subset : same type as 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 calender 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)
 |      Return index for last non-NA/null value.
 |
 |      Returns
 |      --------
 |      scalar : type of index
 |
 |      Notes
 |      --------
 |      If all elements are non-NA/null, returns None.
 |      Also returns None for empty NDFrame.
 |
 |  mask(self, cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try
_cast=False, raise_on_error=None)
 |      Replace values where the condition is True.
 |
 |      Parameters
 |      ----------
 |      cond : boolean NDFrame, 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 NDFrame and
 |          should return boolean NDFrame or array. The callable must
 |          not change input NDFrame (though pandas doesn't check it).
 |
 |          .. versionadded:: 0.18.1
 |              A callable can be used as cond.
 |
 |      other : scalar, NDFrame, or callable
 |          Entries where `cond` is True are replaced with
 |          corresponding value from `other`.
 |          If other is callable, it is computed on the NDFrame and
 |          should return scalar or NDFrame. The callable must not
 |          change input NDFrame (though pandas doesn't check it).
 |
 |          .. versionadded:: 0.18.1
 |              A callable can be used as other.
 |
 |      inplace : boolean, default False
 |          Whether to perform the operation in place on the data.
 |      axis : int, default None
 |          Alignment axis if needed.
 |      level : int, default None
 |          Alignment level if needed.
 |      errors : str, {'raise', 'ignore'}, default `raise`
 |          Note that currently this parameter won't affect
 |          the results and will always coerce to a suitable dtype.
 |
 |          - `raise` : allow exceptions to be raised.
 |          - `ignore` : suppress exceptions. On error return original object.
 |
 |      try_cast : boolean, default False
 |          Try to cast the result back to the input type (if possible).
 |      raise_on_error : boolean, default True
 |          Whether to raise on invalid data types (e.g. trying to where on
 |          strings).
 |
```

```
|          .. deprecated:: 0.21.0
|
|             Use `errors`.
|
|      Returns
|      -------
|      wh : same type as caller
|
|      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.
|
|      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>`.
|
|      Examples
|      --------
|      >>> s = pd.Series(range(5))
|      >>> s.where(s > 0)
|      0    NaN
|      1    1.0
|      2    2.0
|      3    3.0
|      4    4.0
|      dtype: float64
|
|      >>> s.mask(s > 0)
|      0    0.0
|      1    NaN
|      2    NaN
|      3    NaN
|      4    NaN
|      dtype: float64
|
|      >>> s.where(s > 1, 10)
|      0    10
|      1    10
|      2    2
|      3    3
|      4    4
|      dtype: int64
|
|      >>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
|      >>> 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
|
|  pct_change(self, periods=1, fill_method='pad', limit=None, freq=None, **kwargs)
|      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 : str, 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 offset alias string, 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
|      -------
|      chg : 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')
|            2016      2015      2014
|      GOOG   NaN -0.151997 -0.086016
|      APPL   NaN  0.337604  0.012002
|
|  pipe(self, func, *args, **kwargs)
|      Apply func(self, \*args, \*\*kwargs).
|
|      Parameters
|      ----------
|      func : function
|          function to apply to the NDFrame.
|          ``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 NDFrame.
|      args : iterable, optional
|          positional arguments passed into ``func``.
|      kwargs : mapping, optional
|          a dictionary of keyword arguments passed into ``func``.
|
|      Returns
|      -------
|      object : the return type of ``func``.
|
|      See Also
|      --------
|      DataFrame.apply
|      DataFrame.applymap
```

```
|      Series.map
|
|      Notes
|      -----
|
|      Use ``.pipe`` when chaining together functions that expect
|      Series, DataFrames or GroupBy objects. Instead of writing
|
|      >>> f(g(h(df), arg1=a), arg2=b, arg3=c)
|
|      You can write
|
|      >>> (df.pipe(h)
|      ...    .pipe(g, arg1=a)
|      ...    .pipe(f, arg2=b, arg3=c)
|      ... )
|
|      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 ``f`` takes its data as ``arg2``:
|
|      >>> (df.pipe(h)
|      ...    .pipe(g, arg1=a)
|      ...    .pipe((f, 'arg2'), arg1=a, arg3=c)
|      ...  )
|
|  pop(self, item)
|      Return item and drop from frame. Raise KeyError if not found.
|
|      Parameters
|      ----------
|      item : str
|          Column label to be popped
|
|      Returns
|      -------
|      popped : Series
|
|      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'))
|      >>> df
|           name   class  max_speed
|      0  falcon    bird      389.0
|      1  parrot    bird       24.0
|      2    lion  mammal       80.5
|      3  monkey  mammal        NaN
|
|      >>> df.pop('class')
|      0      bird
|      1      bird
|      2    mammal
|      3    mammal
|      Name: class, dtype: object
|
|      >>> df
|           name  max_speed
|      0  falcon      389.0
|      1  parrot       24.0
|      2    lion       80.5
|      3  monkey        NaN
|
|  rank(self, axis=0, method='average', numeric_only=None, na_option='keep', ascending=T
rue, pct=False)
|      Compute numerical data ranks (1 through n) along axis. 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
|      method : {'average', 'min', 'max', 'first', 'dense'}
|          * average: average rank of group
|          * min: lowest rank in group
|          * max: highest rank in group
|          * first: ranks assigned in order they appear in the array
|          * dense: like 'min', but rank always increases by 1 between groups
|      numeric_only : boolean, default None
|          Include only float, int, boolean data. Valid only for DataFrame or
|          Panel objects
|      na_option : {'keep', 'top', 'bottom'}
|          * keep: leave NA values where they are
|          * top: smallest rank if ascending
|          * bottom: smallest rank if descending
|      ascending : boolean, default True
|          False for ranks by high (1) to low (N)
|      pct : boolean, default False
|          Computes percentage rank of data
|
|      Returns
|      -------
|      ranks : same type as caller
|
|  reindex_like(self, other, method=None, copy=True, limit=None, tolerance=None)
|      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 most
|          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.
|
|          .. versionadded:: 0.21.0 (list-like tolerance)
|
|      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
|
|  rename_axis(self, mapper=None, index=None, columns=None, axis=None, copy=True, inplac
e=False)
|      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.
|
|          Use either ``mapper`` and ``axis`` to
|          specify the axis to target with ``mapper``, or ``index``
|          and/or ``columns``.
|
|          .. versionchanged:: 0.24.0
|
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|          The axis to rename.
|      copy : bool, default True
|          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` is 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
|       -----
|       Prior to version 0.21.0, ``rename_axis`` could also be used to change
|       the axis *labels* by passing a mapping or scalar. This behavior is
|       deprecated and will be removed in a future version. Use ``rename``
|       instead.
|
|       ``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
|       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
|
|  resample(self, rule, how=None, axis=0, fill_method=None, closed=None, label=None, con
vention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)
|      Resample time-series data.
|
|      Convenience method for frequency conversion and resampling of time
|      series. Object must have a datetime-like index (`DatetimeIndex`,
|      `PeriodIndex`, or `TimedeltaIndex`), or pass datetime-like values
|      to the `on` or `level` keyword.
|
|      Parameters
|      ----------
|      rule : str
|          The offset string or object representing target conversion.
|      how : str
|          Method for down/re-sampling, default to 'mean' for downsampling.
|
|          .. deprecated:: 0.18.0
|             The new syntax is ``.resample(...).mean()``, or
|             ``.resample(...).apply(<func>)``
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|          Which axis to use for up- or down-sampling. For `Series` this
|          will default to 0, i.e. along the rows. Must be
|          `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
|      fill_method : str, default None
|          Filling method for upsampling.
|
|          .. deprecated:: 0.18.0
|             The new syntax is ``.resample(...).<func>()``,
|             e.g. ``.resample(...).pad()``
|      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.
|       loffset : timedelta, default None
|           Adjust the resampled time labels.
|       limit : int, default None
|           Maximum size gap when reindexing with `fill_method`.
|
|           .. deprecated:: 0.18.0
|       base : int, default 0
|           For frequencies that evenly subdivide 1 day, the "origin" of the
|           aggregated intervals. For example, for '5min' frequency, base could
|           range from 0 through 4. Defaults to 0.
|       on : str, optional
|           For a DataFrame, column to use instead of index for resampling.
|           Column must be datetime-like.
|
|           .. versionadded:: 0.19.0
|
|       level : str or int, optional
|           For a MultiIndex, level (name or number) to use for
|           resampling. `level` must be datetime-like.
|
|           .. versionadded:: 0.19.0
|
|       Returns
|       -------
|       Resampler object
|
|       See Also
|       --------
|       groupby : Group by mapping, function, label, or list of labels.
|       Series.resample : Resample a Series.
|       DataFrame.resample: Resample a DataFrame.
|
|       Notes
|       -----
|       See the `user guide
|       <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#resampling>`_
|       for more.
|
|       To learn more about the offset strings, please see `this link
|       <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>`__.
|
|       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 ``pad`` method.
|
|      >>> series.resample('30S').pad()[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(array_like):
|      ...     return np.sum(array_like) + 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 = dict({'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 = dict({'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
 |
 |  sample(self, n=None, frac=None, replace=False, weights=None, random_state=None, axis=
None)
 |      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
 |          Sample with or without replacement.
 |      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 or numpy.random.RandomState, optional
 |          Seed for the random number generator (if int), or numpy RandomState
 |          object.
 |      axis : int or string, optional
 |          Axis to sample. Accepts axis number or name. Default is stat axis
 |          for given data type (0 for Series and DataFrames, 1 for Panels).
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          A new object of same type as caller containing `n` items randomly
 |          sampled from the caller object.
 |
 |      See Also
 |      --------
 |      numpy.random.choice: Generates a random sample from a given 1-D numpy
```

```
|          array.
|
|      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
|
|      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
|
|  select(self, crit, axis=0)
|      Return data corresponding to axis labels matching criteria.
|
|      .. deprecated:: 0.21.0
|          Use df.loc[df.index.map(crit)] to select via labels
|
|      Parameters
|      ----------
|      crit : function
|          To be called on each index (label). Should return True or False
|      axis : int
|
|      Returns
|      -------
|      selection : same type as caller
|
|  set_axis(self, labels, axis=0, inplace=None)
|      Assign desired index to given axis.
|
|      Indexes for column or row labels can be changed by assigning
|      a list-like or Index.
|
|      .. versionchanged:: 0.21.0
|
|         The signature is now `labels` and `axis`, consistent with
|         the rest of pandas API. Previously, the `axis` and `labels`
|         arguments were respectively the first and second positional
|         arguments.
|
|      Parameters
|      ----------
|      labels : list-like, Index
```

```
|          The values for the new index.
|
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|          The axis to update. The value 0 identifies the rows, and 1
|          identifies the columns.
|
|      inplace : boolean, default None
|          Whether to return a new %(klass)s instance.
|
|          .. warning::
|
|             ``inplace=None`` currently falls back to to True, but in a
|             future version, will default to False. Use inplace=True
|             explicitly rather than relying on the default.
|
|      Returns
|      -------
|      renamed : %(klass)s or None
|          An object of same type as caller if inplace=False, None otherwise.
|
|      See Also
|      --------
|      DataFrame.rename_axis : Alter the name of the index or columns.
|
|      Examples
|      --------
|      **Series**
|
|      >>> s = pd.Series([1, 2, 3])
|      >>> s
|      0    1
|      1    2
|      2    3
|      dtype: int64
|
|      >>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
|      a    1
|      b    2
|      c    3
|      dtype: int64
|
|      The original object is not modified.
|
|      >>> s
|      0    1
|      1    2
|      2    3
|      dtype: int64
|
|      **DataFrame**
|
|      >>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
|
|      Change the row labels.
|
|      >>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
|         A  B
|      a  1  4
|      b  2  5
|      c  3  6
|
|      Change the column labels.
|
|      >>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
|         I  II
|      0  1   4
|      1  2   5
|      2  3   6
|
|      Now, update the labels inplace.
|
|      >>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
```

```
|      >>> df
|         i  ii
|      0  1   4
|      1  2   5
|      2  3   6
|
|  slice_shift(self, periods=1, axis=0)
|      Equivalent to `shift` without copying data. The shifted data will
|      not include the dropped periods and the shifted axis will be smaller
|      than the original.
|
|      Parameters
|      ----------
|      periods : int
|          Number of periods to move, can be positive or negative
|
|      Returns
|      -------
|      shifted : same type as caller
|
|      Notes
|      -----
|      While the `slice_shift` is faster than `shift`, you may pay for it
|      later during alignment.
|
|  squeeze(self, axis=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.
|
|          .. versionadded:: 0.20.0
|
|      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 wil project directly into a scalar:
|
|      >>> df_0a.squeeze()
|      1
|
|  swapaxes(self, axis1, axis2, copy=True)
|      Interchange axes and swap values axes appropriately.
|
|      Returns
|      -------
|      y : same as input
|
|  tail(self, n=5)
|      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.
|
|      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
|
|  take(self, indices, axis=0, convert=None, is_copy=True, **kwargs)
|      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.
|      convert : bool, default True
|          Whether to convert negative indices into positive ones.
|          For example, ``-1`` would map to the ``len(axis) - 1``.
|          The conversions are similar to the behavior of indexing a
|          regular Python list.
|
|          .. deprecated:: 0.21.0
|             In the future, negative indices will always be converted.
|
|      is_copy : bool, default True
|          Whether to return a copy of the original object or not.
```

```
|      **kwargs
|          For compatibility with :meth:`numpy.take`. Has no effect on the
|          output.
|
|      Returns
|      -------
|      taken : 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_clipboard(self, excel=True, sep=None, **kwargs)
|      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
|          - True, use the provided separator, writing in a csv format for
|            allowing easy pasting into excel.
|          - 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_table.
 |
 |      Notes
 |      -----
 |      Requirements for your platform.
 |
 |        - Linux : `xclip`, or `xsel` (with `gtk` or `PyQt4` modules)
 |        - Windows : none
 |        - OS X : none
 |
 |      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=',')
 |      ... # Wrote the following to the system clipboard:
 |      ... # ,A,B,C
 |      ... # 0,1,2,3
 |      ... # 1,4,5,6
 |
 |      We can omit the the index by passing the keyword `index` and setting
 |      it to false.
 |
 |      >>> df.to_clipboard(sep=',', index=False)
 |      ... # Wrote the following to the system clipboard:
 |      ... # A,B,C
 |      ... # 1,2,3
 |      ... # 4,5,6
 |
 |  to_dense(self)
 |      Return dense representation of NDFrame (as opposed to sparse).
 |
 |  to_excel(self, excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, colum
ns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None,
merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None)
 |      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 : str 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``, ``io.excel.xls.writer``, and
|          ``io.excel.xlsm.writer``.
|      merge_cells : bool, default True
|          Write MultiIndex and Hierarchical Rows as merged cells.
|      encoding : str, optional
|          Encoding of the resulting excel file. Only necessary for xlwt,
|          other writers support unicode natively.
|      inf_rep : str, default 'inf'
|          Representation for infinity (there is no native representation for
|          infinity in Excel).
|      verbose : bool, default True
|          Display more information in the error logs.
|      freeze_panes : tuple of int (length 2), optional
|          Specifies the one-based bottommost row and rightmost column that
|          is to be frozen.
|
|          .. versionadded:: 0.20.0.
|
|      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.
|
|      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 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')
|
|      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, key, **kwargs)
|      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.
|
|      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.
|      format : {'fixed', 'table'}, 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.
|      append : bool, default False
|          For Table formats, append the input data to the existing.
|      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:`io.hdf5-query-data-columns`.
|          Applicable only to format='table'.
|      complevel : {0-9}, optional
|          Specifies a compression level for data.
|          A value of 0 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.
|      fletcher32 : bool, default False
|          If applying compression use the fletcher32 checksum.
|      dropna : bool, default False
|          If true, ALL nan rows will not be written to store.
|      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.
|
|      See Also
|      --------
|      DataFrame.read_hdf : Read from HDF file.
|      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'])
 |      >>> df.to_hdf('data.h5', key='df', mode='w')
 |
 |      We can add another object to the same file:
 |
 |      >>> s = pd.Series([1, 2, 3, 4])
 |      >>> s.to_hdf('data.h5', key='s')
 |
 |      Reading from HDF file:
 |
 |      >>> pd.read_hdf('data.h5', 'df')
 |      A  B
 |      a  1  4
 |      b  2  5
 |      c  3  6
 |      >>> pd.read_hdf('data.h5', 's')
 |      0    1
 |      1    2
 |      2    3
 |      3    4
 |      dtype: int64
 |
 |      Deleting file with data:
 |
 |      >>> import os
 |      >>> os.remove('data.h5')
 |
 |  to_json(self, path_or_buf=None, orient=None, date_format=None, double_precision=10, f
orce_ascii=True, date_unit='ms', default_handler=None, lines=False, compression='infer',
index=True)
 |      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 : string or file handle, optional
 |          File path or object. If not specified, the result is returned as
 |          a string.
 |      orient : string
 |          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, and the data component is
 |              like ``orient='records'``.
 |
 |              .. versionchanged:: 0.20.0
```

```
|
|      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 : string, 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.
|
|          .. versionadded:: 0.19.0
|
|      compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}
|
|          A string representing the compression to use in the output file,
|          only used when the first argument is a filename. By default, the
|          compression is inferred from the filename.
|
|          .. versionadded:: 0.21.0
|          .. versionchanged:: 0.24.0
|             'infer' option added and set to default
|      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'.
|
|          .. versionadded:: 0.23.0
|
|      See Also
|      --------
|      read_json
|
|      Examples
|      --------
|
|      >>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
|      ...                   index=['row 1', 'row 2'],
|      ...                   columns=['col 1', 'col 2'])
|      >>> df.to_json(orient='split')
|      '{"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.
|
|      >>> df.to_json(orient='records')
|      '[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
|
|      Encoding/decoding a Dataframe using ``'index'`` formatted JSON:
|
|      >>> df.to_json(orient='index')
|      '{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
|
|      Encoding/decoding a Dataframe using ``'columns'`` formatted JSON:
|
|      >>> df.to_json(orient='columns')
|      '{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

```
 |
 |      Encoding/decoding a Dataframe using ``'values'`` formatted JSON:
 |
 |      >>> df.to_json(orient='values')
 |      '[["a","b"],["c","d"]]'
 |
 |      Encoding with Table Schema
 |
 |      >>> df.to_json(orient='table')
 |      '{"schema": {"fields": [{"name": "index", "type": "string"},
 |                              {"name": "col 1", "type": "string"},
 |                              {"name": "col 2", "type": "string"}],
 |                    "primaryKey": "index",
 |                    "pandas_version": "0.20.0"},
 |        "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
 |                 {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
 |
 |  to_latex(self, buf=None, columns=None, col_space=None, header=True, index=True, na_re
p='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=F
alse, column_format=None, longtable=None, escape=None, encoding=None, decimal='.', multic
olumn=None, multicolumn_format=None, multirow=None)
 |      Render an object to a LaTeX tabular environment table.
 |
 |      Render an object to a tabular environment table. You can splice
 |      this into a LaTeX document. Requires \usepackage{booktabs}.
 |
 |      .. versionchanged:: 0.20.2
 |         Added to Series
 |
 |      Parameters
 |      ----------
 |      buf : file descriptor or 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.
 |      col_space : int, optional
 |          The minimum width of each column.
 |      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 : str, optional
 |          Format string for floating point numbers.
 |      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
 |          By default, the value will be read from the pandas config
 |          module. Use a longtable environment instead of tabular. Requires
 |          adding a \usepackage{longtable} to your LaTeX preamble.
 |      escape : bool, optional
 |          By default, the value will be read from the pandas config
 |          module. When set to False prevents from escaping latex special
 |          characters in column names.
 |      encoding : str, optional
```

```
 |          A string representing the encoding to use in the output file,
 |          defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.
 |      decimal : str, default '.'
 |          Character recognized as decimal separator, e.g. ',' in Europe.
 |          .. versionadded:: 0.18.0
 |      multicolumn : bool, default True
 |          Use \multicolumn to enhance MultiIndex columns.
 |          The default will be read from the config module.
 |          .. versionadded:: 0.20.0
 |      multicolumn_format : str, default 'l'
 |          The alignment for multicolumns, similar to `column_format`
 |          The default will be read from the config module.
 |          .. versionadded:: 0.20.0
 |      multirow : bool, default False
 |          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.
 |          .. versionadded:: 0.20.0
 |
 |      Returns
 |      -------
 |      str or None
 |          If buf is None, returns the resulting LateX format as a
 |          string. Otherwise returns None.
 |
 |      See Also
 |      --------
 |      DataFrame.to_string : Render a DataFrame to a console-friendly
 |          tabular output.
 |      DataFrame.to_html : Render a DataFrame as an HTML table.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
 |      ...                    'mask': ['red', 'purple'],
 |      ...                    'weapon': ['sai', 'bo staff']})
 |      >>> df.to_latex(index=False) # doctest: +NORMALIZE_WHITESPACE
 |      '\\begin{tabular}{lll}\n\\toprule\n      name &    mask &    weapon
 |      \\\\\n\\midrule\n   Raphael &     red &       sai \\\\\n Donatello &
 |       purple &  bo staff \\\\\n\\bottomrule\n\\end{tabular}\n'
 |
 |  to_msgpack(self, path_or_buf=None, encoding='utf-8', **kwargs)
 |      Serialize object to input file path using msgpack format.
 |
 |      THIS IS AN EXPERIMENTAL LIBRARY and the storage format
 |      may not be stable until a future release.
 |
 |      Parameters
 |      ----------
 |      path : string File path, buffer-like, or None
 |          if None, return generated string
 |      append : bool whether to append to an existing msgpack
 |          (default is False)
 |      compress : type of compressor (zlib or blosc), default to None (no
 |          compression)
 |
 |  to_pickle(self, path, compression='infer', protocol=4)
 |      Pickle (serialize) object to file.
 |
 |      Parameters
 |      ----------
 |      path : str
 |          File path where the pickled object will be stored.
 |      compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None},         default 'infe
r'
 |          A string representing the compression to use in the output file. By
 |          default, infers from the file extension in specified path.
 |
 |          .. versionadded:: 0.20.0
 |      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 for this parameter depend on the version of Python. For
 |          Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a
 |          valid value. For Python >= 3.4, 4 is a valid value. 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
 |          .. versionadded:: 0.21.0
 |
 |      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)})
 |      >>> original_df
 |         foo  bar
 |      0    0    5
 |      1    1    6
 |      2    2    7
 |      3    3    8
 |      4    4    9
 |      >>> original_df.to_pickle("./dummy.pkl")
 |
 |      >>> unpickled_df = pd.read_pickle("./dummy.pkl")
 |      >>> unpickled_df
 |         foo  bar
 |      0    0    5
 |      1    1    6
 |      2    2    7
 |      3    3    8
 |      4    4    9
 |
 |      >>> import os
 |      >>> os.remove("./dummy.pkl")
 |
 |  to_sql(self, name, con, schema=None, if_exists='fail', index=True, index_label=None,
chunksize=None, dtype=None, method=None)
 |      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 : string
 |          Name of SQL table.
 |      con : sqlalchemy.engine.Engine or sqlite3.Connection
 |          Using SQLAlchemy makes it possible to use any DB supported by that
 |          library. Legacy support is provided for sqlite3.Connection objects.
 |      schema : string, 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 : string 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
|          Rows will be written in batches of this size at a time. By default,
|          all rows will be written at once.
|      dtype : dict, optional
|          Specifying the datatype for columns. The keys should be the column
|          names and the values should be the SQLAlchemy types or strings for
|          the sqlite3 legacy mode.
|      method : {None, 'multi', callable}, default None
|          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>`.
|
|          .. versionadded:: 0.24.0
|
|      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.
|
|      .. versionadded:: 0.24.0
|
|      References
|      ----------
|      .. [1] http://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)
|      >>> engine.execute("SELECT * FROM users").fetchall()
|      [(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
|
|      >>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
|      >>> df1.to_sql('users', con=engine, if_exists='append')
|      >>> engine.execute("SELECT * FROM users").fetchall()
|      [(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
|       (0, 'User 4'), (1, 'User 5')]
|
|      Overwrite the table with just ``df1``.
```

```
|
|      >>> df1.to_sql('users', con=engine, if_exists='replace',
|      ...            index_label='id')
|      >>> engine.execute("SELECT * FROM users").fetchall()
|      [(0, 'User 4'), (1, 'User 5')]
|
|      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()})
|
|      >>> engine.execute("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 <http://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]}).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:  (animal: 2, date: 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, before=None, after=None, axis=None, copy=True)
|      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, string, int
|          Truncate all rows before this index value.
|      after : date, string, 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.
|      copy : boolean, 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
|
```

```
 |  tshift(self, periods=1, freq=None, axis=0)
 |      Shift the time index, using the index's frequency if available.
 |
 |      Parameters
 |      ----------
 |      periods : int
 |          Number of periods to move, can be positive or negative
 |      freq : DateOffset, timedelta, or time rule string, default None
 |          Increment to use from the tseries module or time rule (e.g. 'EOM')
 |      axis : int or basestring
 |          Corresponds to the axis that contains the Index
 |
 |      Returns
 |      -------
 |      shifted : NDFrame
 |
 |      Notes
 |      -----
 |      If freq is not specified then tries to use the freq or inferred_freq
 |      attributes of the index. If neither of those attributes exist, a
 |      ValueError is thrown
 |
 |  tz_convert(self, tz, axis=0, level=None, copy=True)
 |      Convert tz-aware axis to target time zone.
 |
 |      Parameters
 |      ----------
 |      tz : string or pytz.timezone object
 |      axis : the axis to convert
 |      level : int, str, default None
 |          If axis ia a MultiIndex, convert a specific level. Otherwise
 |          must be None
 |      copy : boolean, default True
 |          Also make a copy of the underlying data
 |
 |      Returns
 |      -------
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the axis is tz-naive.
 |
 |  tz_localize(self, tz, axis=0, level=None, copy=True, ambiguous='raise', nonexistent='
raise')
 |      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 : string or pytz.timezone object
 |      axis : the axis to localize
 |      level : int, str, default None
 |          If axis ia a MultiIndex, localize a specific level. Otherwise
 |          must be None
 |      copy : boolean, 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 valuse 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
|
|          .. versionadded:: 0.24.0
|
|      Returns
|      -------
|      Series or 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
|
|      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 backwards with a timedelta object or `'shift_forward'`
|      or `'shift_backwards'`.
|      >>> 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
|
|  where(self, cond, other=nan, inplace=False, axis=None, level=None, errors='raise', tr
y_cast=False, raise_on_error=None)
|      Replace values where the condition is False.
|
|      Parameters
|      ----------
|      cond : boolean NDFrame, 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 NDFrame and
|          should return boolean NDFrame or array. The callable must
|          not change input NDFrame (though pandas doesn't check it).
|
|          .. versionadded:: 0.18.1
|              A callable can be used as cond.
|
|      other : scalar, NDFrame, or callable
|          Entries where `cond` is False are replaced with
|          corresponding value from `other`.
|          If other is callable, it is computed on the NDFrame and
|          should return scalar or NDFrame. The callable must not
|          change input NDFrame (though pandas doesn't check it).
|
|          .. versionadded:: 0.18.1
|              A callable can be used as other.
|
|      inplace : boolean, default False
|          Whether to perform the operation in place on the data.
|      axis : int, default None
|          Alignment axis if needed.
|      level : int, default None
|          Alignment level if needed.
|      errors : str, {'raise', 'ignore'}, default `raise`
|          Note that currently this parameter won't affect
|          the results and will always coerce to a suitable dtype.
|
|          - `raise` : allow exceptions to be raised.
|          - `ignore` : suppress exceptions. On error return original object.
|
|      try_cast : boolean, default False
|          Try to cast the result back to the input type (if possible).
|      raise_on_error : boolean, default True
|          Whether to raise on invalid data types (e.g. trying to where on
|          strings).
|
|          .. deprecated:: 0.21.0
|
|             Use `errors`.
|
|      Returns
|      -------
|      wh : same type as caller
```

```
|
|      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.
|
|      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>`.
|
|      Examples
|      --------
|      >>> s = pd.Series(range(5))
|      >>> s.where(s > 0)
|      0    NaN
|      1    1.0
|      2    2.0
|      3    3.0
|      4    4.0
|      dtype: float64
|
|      >>> s.mask(s > 0)
|      0    0.0
|      1    NaN
|      2    NaN
|      3    NaN
|      4    NaN
|      dtype: float64
|
|      >>> s.where(s > 1, 10)
|      0    10
|      1    10
|      2    2
|      3    3
|      4    4
|      dtype: int64
|
|      >>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
|      >>> 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
|
|  xs(self, key, axis=0, level=None, drop_level=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'))
|                   num_legs  num_wings
|       locomotion
|       walks              4          0
|
```

```
|      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
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pandas.core.generic.NDFrame:
|
|  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
|          When label does not exist in DataFrame
|
|      See Also
|      --------
|      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).
|      Series.at : Access a single value using a label.
|
|      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
```

```
|
|  blocks
|      Internal property, property synonym for as_blocks().
|
|      .. deprecated:: 0.21.0
|
|  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.
|
|      ``.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, DataFrame
|        or Panel) 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.
|
|      ``.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
|
|  is_copy
|      Return the copy.
|
|  ix
|      A primarily label-location based indexer, with integer position
|      fallback.
|
|      Warning: Starting in 0.20.0, the .ix indexer is deprecated, in
|      favor of the more strict .iloc and .loc indexers.
|
|      ``.ix[]`` supports mixed integer and label based access. It is
|      primarily label based, but will fall back to integer positional
|      access unless the corresponding axis is of integer type.
|
|      ``.ix`` is the most general indexer and will support any of the
|      inputs in ``.loc`` and ``.iloc``. ``.ix`` also supports floating
|      point label schemes. ``.ix`` is exceptionally useful when dealing
|      with mixed positional and label based hierarchical indexes.
|
|      However, when an axis is integer based, ONLY label based access
|      and not positional access is supported. Thus, in such cases, it's
|      usually better to be explicit and use ``.iloc`` or ``.loc``.
|
|      See more at :ref:`Advanced Indexing <advanced>`.
|
|  loc
|      Access a group of rows and columns by label(s) or a boolean array.
|
|      ``.loc[]`` is primarily label based, but may also be used with a
|      boolean array.
|
|      Allowed inputs are:
```

```
|
|       - A single label, e.g. ``5`` or ``'a'``, (note that ``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]``.
|       - A ``callable`` function with one argument (the calling Series, DataFrame
|         or Panel) and that returns valid output for indexing (one of the above)
|
|       See more at :ref:`Selection by Label <indexing.label>`
|
|       Raises
|       ------
|       KeyError:
|           when any items are not found
|
|       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
|
|      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
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from pandas.core.generic.NDFrame:
|
|  timetuple = None
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.base.PandasObject:
|
|  __sizeof__(self)
|      Generates the total memory usage for an object that returns
|      either a value or Series of values
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.base.StringMixin:
|
|  __bytes__(self)
|      Return a string representation for a particular object.
|
|      Invoked by bytes(obj) in py3 only.
|      Yields a bytestring in both py2/py3.
|
|  __repr__(self)
|      Return a string representation for a particular object.
|
|      Yields Bytestring in Py2, Unicode String in py3.
|
|  __str__(self)
|      Return a string representation for a particular Object
|
|      Invoked by str(df) in both py2/py3.
|      Yields Bytestring in Py2, Unicode String in py3.
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.accessor.DirNamesMixin:
|
|  __dir__(self)
|      Provide method name lookup and completion
|      Only provide 'public' methods
```

## Index and Data Lists

### We can create a Series from Python lists (also from NumPy arrays)

In [14]:

```
myindex = ['USA','Canada','Mexico']
```

In [15]:

```
mydata = [1776,1867,1821]
```

In [16]:

```
myser = pd.Series(data=mydata)
```

In [17]:

In [17]:

```
myser
```

Out[17]:

```
0    1776
1    1867
2    1821
dtype: int64
```

In [18]:

```
pd.Series(data=mydata,index=myindex)
```

Out[18]:

```
USA       1776
Canada    1867
Mexico    1821
dtype: int64
```

In [23]:

```
ran_data = np.random.randint(0,100,4)
```

In [24]:

```
ran_data
```

Out[24]:

```
array([39, 35, 37, 23])
```

In [26]:

```
names = ['Andrew','Bobo','Claire','David']
```

In [27]:

```
ages = pd.Series(ran_data,names)
```

In [28]:

```
ages
```

Out[28]:

```
Andrew    39
Bobo      35
Claire    37
David     23
dtype: int32
```

## From a Dictionary

In [29]:

```
ages = {'Sammy':5,'Frank':10,'Spike':7}
```

In [30]:

```
ages
```

Out[30]:

```
{'Frank': 10, 'Sammy': 5, 'Spike': 7}
```

In [31]:

```
pd.Series(ages)
```

Out[31]:

```
Sammy     5
Frank    10
Spike     7
dtype: int64
```

# Key Ideas of a Series

## Named Index

In [32]:

```
# Imaginary Sales Data for 1st and 2nd Quarters for Global Company
q1 = {'Japan': 80, 'China': 450, 'India': 200, 'USA': 250}
q2 = {'Brazil': 100,'China': 500, 'India': 210,'USA': 260}
```

In [33]:

```
# Convert into Pandas Series
sales_Q1 = pd.Series(q1)
sales_Q2 = pd.Series(q2)
```

In [34]:

```
sales_Q1
```

Out[34]:

```
Japan     80
China    450
India    200
USA      250
dtype: int64
```

In [35]:

```
# Call values based on Named Index
sales_Q1['Japan']
```

Out[35]:

```
80
```

In [36]:

```
# Integer Based Location information also retained!
sales_Q1[0]
```

Out[36]:

```
80
```

**Be careful with potential errors!**

In [37]:

```
# Wrong Name
# sales_Q1['France']
```

In [38]:

```
# Accidental Extra Space
# sales_Q1['USA ']
```

In [39]:

```
# Capitalization Mistake
```

```
# sales_Q1['usa']
```

## Operations

In [40]:

```
# Grab just the index keys
sales_Q1.keys()
```

Out[40]:

```
Index(['Japan', 'China', 'India', 'USA'], dtype='object')
```

In [41]:

```
# Can Perform Operations Broadcasted across entire Series
sales_Q1 * 2
```

Out[41]:

```
Japan    160
China    900
India    400
USA      500
dtype: int64
```

In [42]:

```
sales_Q2 / 100
```

Out[42]:

```
Brazil    1.0
China     5.0
India     2.1
USA       2.6
dtype: float64
```

## Between Series

In [43]:

```
# Notice how Pandas informs you of mismatch with NaN
sales_Q1 + sales_Q2
```

Out[43]:

```
Brazil      NaN
China     950.0
India     410.0
Japan       NaN
USA       510.0
dtype: float64
```

In [44]:

```
# You can fill these with any value you want
sales_Q1.add(sales_Q2,fill_value=0)
```

Out[44]:

```
Brazil    100.0
China     950.0
India     410.0
Japan      80.0
USA       510.0
dtype: float64
```

**That is all we need to know about Series, up next, DataFrames!**