'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`.

Changed in version 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

pandas.Series.resample

Series.**resample**(*self*, *rule*, *axis=0*, *closed: Union[str, NoneType] = None*, *label: Union[str, NoneType] = None*, *convention: str = 'start'*, *kind: Union[str, NoneType] = None*, *loffset=None*, *base: int = 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 [DateOffset, Timedelta or str] The offset string or object representing target conversion.

axis [{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this will default to 0, i.e. along the rows. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

closed [{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind [{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

loffset [timedelta, default None] Adjust the resampled time labels.

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.

level [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

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 for more.

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

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
```

(continues on next page)

(continued from previous page)

```
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
```

pandas.Series.reset_index

`Series.`**`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
```

pandas.Series.rfloordiv

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

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

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

Series The result of the operation.

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

pandas.Series.rmod

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

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

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

Parameters

other [Series or scalar value]

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

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

Returns

Series The result of the operation.

See also:

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

pandas.Series.rmul

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

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

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

Parameters

other [Series or scalar value]

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

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

Returns

Series The result of the operation.

See also:

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

pandas.Series.rolling

Series.**rolling**(*self*, *window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provide rolling window calculations.

Parameters

window [int, offset, or BaseIndexer subclass] 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.

If a BaseIndexer subclass is passed, calculates the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely *min_periods*, *center*, and *closed* will be passed to *get_window_bounds*.

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is 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, a datetime-like column or MultiIndex level on which to calculate the rolling window, rather than the DataFrame's index. Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

axis [int or str, default 0]

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.

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.

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)
- `exponential` (needs tau), center is set to None.

If `win_type=None` all points are evenly weighted. To learn more about different window types see scipy.signal 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  0.5
2  1.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, using the 'gaussian' window type (note how we need to specify std).

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
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'),
```

(continues on next page)

(continued from previous page)

```
...                            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
```

pandas.Series.round

Series.**round**(*self*, *decimals=0*, **args*, ***kwargs*)

Round each value in a Series to the given number of decimals.

Parameters

decimals [int, default 0] Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Series Rounded values of the Series.

See also:

numpy.around Round values of an np.array.

DataFrame.round Round values of a DataFrame.

Examples

```
>>> s = pd.Series([0.1, 1.3, 2.7])
>>> s.round()
0    0.0
1    1.0
2    3.0
dtype: float64
```

pandas.Series.rpow

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

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

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

Parameters

other [Series or scalar value]

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

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

Returns

Series The result of the operation.

See also:

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

pandas.Series.rsub

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

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

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

Parameters

other [Series or scalar value]

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

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

Returns

Series The result of the operation.

See also:

Series.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.subtract(b, fill_value=0)
a    0.0
b    1.0
c    1.0
d   -1.0
e    NaN
dtype: float64
```

pandas.Series.rtruediv

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

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

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

Parameters

other [Series or scalar value]

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

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

Returns

Series The result of the operation.

See also:

Series.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.divide(b, fill_value=0)
a    1.0
b    inf
c    inf
d    0.0
e    NaN
dtype: float64
```

pandas.Series.sample

Series.**sample**(*self: ~ FrameOrSeries*, *n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*) → ~FrameOrSeries

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [bool, default False] Allow or disallow sampling of the same row more than once.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames).

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.

Notes

If *frac* > 1, *replacement* should be set to *True*.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the `DataFrame` with replacement: Note that *replace* parameter has to be *True* for *frac* parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
        num_legs  num_wings  num_specimen_seen
dog            4          0                  2
fish           0          0                  8
falcon         2          2                 10
falcon         2          2                 10
fish           0          0                  8
dog            4          0                  2
fish           0          0                  8
dog            4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

pandas.Series.searchsorted

`Series.`**`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.

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

Parameters

value [array_like] 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*.

Changed in version 0.24.0: If *value* is a scalar, an int is now always returned. Previously, scalar inputs returned an 1-item array for *Series* and *Categorical*.

See also:

sort_values

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])
```

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

```
>>> x = pd.Series([2, 1, 3])
>>> x.searchsorted(1)
0  # wrong result, correct would be 1
```

pandas.Series.sem

Series.**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 [bool, 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 [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.

Returns

scalar or Series (if level specified)

pandas.Series.set_axis

Series.**set_axis**(*self*, *labels*, *axis=0*, *inplace=False*)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 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 [bool, default False] Whether to return a new %(klass)s instance.

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)
a    1
b    2
c    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')
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns')
   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
```

pandas.Series.shift

Series.**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