pandas.Series.add

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

Return Addition of series and other, element-wise (binary operator *add*).

Equivalent to `series + other`, but with support to substitute a fill_value for missing data in 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.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
```

pandas.Series.add_prefix

Series.**add_prefix**(*self: ~ FrameOrSeries*, *prefix: str*) → ~FrameOrSeries

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
```

pandas.Series.add_suffix

Series.**add_suffix**(*self: ~ FrameOrSeries*, *suffix: str*) → ~FrameOrSeries

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
```

pandas.Series.agg

Series.**agg**(*self*, *func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 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

scalar, Series or DataFrame The return can be:

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

Return scalar, Series or DataFrame.

See also:

Series.apply Invoke function on a Series.

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

Notes

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

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
```

pandas.Series.aggregate

Series.**aggregate**(*self*, *func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 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

scalar, Series or DataFrame The return can be:

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

Return scalar, Series or DataFrame.

See also:

Series.apply Invoke function on a Series.

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

Notes

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

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
```

pandas.Series.align

Series.**align**(*self*, *other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

axis [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy [bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series:

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

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

fill_axis [{0 or 'index'}, default 0] Filling axis, method and limit.

broadcast_axis [{0 or 'index'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions.

Returns

(left, right) [(Series, type of other)] Aligned objects.

pandas.Series.all

`Series.`**`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
```

pandas.Series.any

`Series.`**`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)
```

pandas.Series.append

`Series.`**`append`**(*self*, *to_append*, *ignore_index=False*, *verify_integrity=False*)

Concatenate two or more Series.

Parameters

to_append [Series or list/tuple of Series] Series to append with self.

ignore_index [bool, default False] If True, do not use the index labels.

verify_integrity [bool, default False] If True, raise Exception on creating index with duplicates.

Returns

Series Concatenated Series.

See also:

`concat` General function to concatenate DataFrame or Series 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]
```

pandas.Series.apply

`Series.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
```

pandas.Series.argmax

Series.**argmax**(*self*, *axis=None*, *skipna=True*, **args*, ***kwargs*)

Return an ndarray of the maximum argument indexer.

Parameters

axis [{None}] Dummy argument for consistency with Series.

skipna [bool, default True]

Returns

numpy.ndarray Indices of the maximum values.

See also:

numpy.ndarray.argmax

pandas.Series.argmin

Series.**argmin**(*self*, *axis=None*, *skipna=True*, **args*, ***kwargs*)

Return a ndarray of the minimum argument indexer.

Parameters

axis [{None}] Dummy argument for consistency with Series.

skipna [bool, default True]

Returns

numpy.ndarray

See also:

numpy.ndarray.argmin

pandas.Series.argsort

Series.**argsort**(*self*, *axis=0*, *kind='quicksort'*, *order=None*)

Override ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values.

Parameters

axis [{0 or "index"}] 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

Series Positions of values within the sort order with -1 indicating nan values.

See also:

numpy.ndarray.argsort

pandas.Series.asfreq

Series.**asfreq**(*self: ~ FrameOrSeries*, *freq*, *method=None*, *how: Union[str, NoneType] = None*, *normalize: bool = False*, *fill_value=None*) → ~FrameOrSeries

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 or str]

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

- 'pad' / 'ffill': propagate last valid observation forward to next valid

- 'backfill' / 'bfill': use NEXT valid observation to fill.

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

normalize [bool, default False] Whether to reset output index to midnight.

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

Returns

converted [same type as caller]

See also:

reindex

Notes

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

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
```

pandas.Series.asof

`Series.`**`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 *DataFrame*, the last row without NaN considering only the subset of columns (if not *None*)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where [date or array-like of dates] Date(s) before which the last row(s) are returned.

subset [str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame The return can be:

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

Return scalar, Series, or DataFrame.

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                     a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...          subset=['a'])
                       a   b
2018-02-27 09:03:30   30.0 NaN
2018-02-27 09:04:30   40.0 NaN
```

pandas.Series.astype

Series.**astype**(*self: ~ FrameOrSeries*, *dtype*, *copy: bool = True*, *errors: str = 'raise'*) → ~FrameOrSeries

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.

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

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, 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
```

pandas.Series.at_time

`Series.`**`at_time`**(*self: ~ FrameOrSeries*, *time*, *asof: bool = False*, *axis=None*) → ~FrameOrSeries

Select values at particular time of day (e.g. 9:30AM).

Parameters

time [datetime.time or str]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

Series or DataFrame

Raises

TypeError If the index is not a *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
```

pandas.Series.autocorr

Series.**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()
0.10355...
>>> s.autocorr(lag=2)
-0.99999...
```

If the Pearson correlation is not well defined, then 'NaN' is returned.

```
>>> s = pd.Series([1, 0, 0, 0])
>>> s.autocorr()
nan
```

pandas.Series.between

Series.**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 or list-like] Left boundary.

right [scalar or list-like] Right boundary.

inclusive [bool, default True] Include boundaries.

Returns

Series Series representing whether each element is between left and right (inclusive).

See also:

Series.gt Greater than of series and other.

Series.lt Less than of series and other.

Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

Examples

```
>>> s = pd.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

With *inclusive* set to `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
```

pandas.Series.between_time

Series.**between_time**(*self: ~ FrameOrSeries*, *start_time*, *end_time*, *include_start: bool = True*, *include_end: bool = True*, *axis=None*) → ~FrameOrSeries

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time [datetime.time or str]

end_time [datetime.time or str]

include_start [bool, default True]

include_end [bool, default True]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

Series or DataFrame

Raises

TypeError If the index is not a *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
```

pandas.Series.bfill

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

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

Returns

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

pandas.Series.bool

Series.**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

Returns

bool Same single boolean value converted to bool type.

pandas.Series.cat

Series.**cat**()

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()
```

pandas.Series.clip

Series.**clip**(*self: ~ FrameOrSeries*, *lower=None*, *upper=None*, *axis=None*, *inplace: bool = False*, **args*, ***kwargs*) → ~FrameOrSeries

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 str axis name, optional] Align object with lower and upper along the given axis.

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

New in version 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
```