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

pandas.Series.xs

Series.**xs**(*self*, *key*, *axis=0*, *level=None*, *drop_level: bool = True*)

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

DataFrame.loc Access a group of rows and columns by label(s) or a boolean array.

DataFrame.iloc Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see *MultiIndex 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
```

3.3.2 Attributes

Axes

Series.index	The index (axis labels) of the Series.

Series.array	The ExtensionArray of the data backing this Series or Index.
Series.values	Return Series as ndarray or ndarray-like depending on the dtype.
Series.dtype	Return the dtype object of the underlying data.
Series.shape	Return a tuple of the shape of the underlying data.
Series.nbytes	Return the number of bytes in the underlying data.
Series.ndim	Number of dimensions of the underlying data, by definition 1.
Series.size	Return the number of elements in the underlying data.
Series.T	Return the transpose, which is by definition self.
Series.memory_usage(self[, index, deep])	Return the memory usage of the Series.
Series.hasnans	Return if I have any nans; enables various perf speedups.
Series.empty	Indicator whether DataFrame is empty.
Series.dtypes	Return the dtype object of the underlying data.
Series.name	

pandas.Series.empty

property Series.**empty**

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If DataFrame is empty, return True, if not return False.

See also:

Series.dropna
DataFrame.dropna

Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

pandas.Series.name

property Series.**name**

3.3.3 Conversion

Series.astype(self, dtype, copy, errors)	Cast a pandas object to a specified dtype `dtype`.
Series.convert_dtypes(self, infer_objects, ...)	Convert columns to best possible dtypes using dtypes supporting `pd.NA`.
Series.infer_objects(self)	Attempt to infer better dtypes for object columns.
Series.copy(self, deep)	Make a copy of this object's indices and data.
Series.bool(self)	Return the bool of a single element PandasObject.
Series.to_numpy(self[, dtype, copy, na_value])	A NumPy ndarray representing the values in this Series or Index.
Series.to_period(self[, freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
Series.to_timestamp(self[, freq, how, copy])	Cast to DatetimeIndex of Timestamps, at *beginning* of period.
Series.to_list(self)	Return a list of the values.
Series.__array__(self[, dtype])	Return the values as a NumPy array.

pandas.Series.__array__

Series.**__array__**(*self*, *dtype=None*) → numpy.ndarray

Return the values as a NumPy array.

Users should not call this directly. Rather, it is invoked by `numpy.array()` and `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 `numpy.ndarary` with the specified *dtype*.

See also:

array Create a new array from data.
Series.array Zero-copy view to the array backing the Series.
Series.to_numpy Series method for similar behavior.

Examples

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

For timezone-aware data, the timezones may be retained with `dtype='object'`

```
>>> tzser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> np.asarray(tzser, dtype="object")
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', 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 discarded with `dtype='datetime64[ns]'`

```
>>> np.asarray(tzser, dtype="datetime64[ns]")
array(['1999-12-31T23:00:00.000000000', ...],
      dtype='datetime64[ns]')
```

3.3.4 Indexing, iteration

Series.get(self, key[, default])	Get item from object for given key (ex: DataFrame column).
Series.at	Access a single value for a row/column label pair.
Series.iat	Access a single value for a row/column pair by integer position.
Series.loc	Access a group of rows and columns by label(s) or a boolean array.
Series.iloc	Purely integer-location based indexing for selection by position.
Series.__iter__(self)	Return an iterator of the values.

continues on next page

Table 33 – continued from previous page

Series.items(self)	Lazily iterate over (index, value) tuples.
Series.iteritems(self)	Lazily iterate over (index, value) tuples.
Series.keys(self)	Return alias for index.
Series.pop(self, item)	Return item and drop from frame.
Series.item(self)	Return the first element of the underlying data as a python scalar.
Series.xs(self, key[, axis, level])	Return cross-section from the Series/DataFrame.

pandas.Series.__iter__

Series.**__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)

Returns

iterator

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the *indexing documentation*.

3.3.5 Binary operator functions

Series.add(self, other[, level, fill_value, ...])	Return Addition of series and other, element-wise (binary operator *add*).
Series.sub(self, other[, level, fill_value, ...])	Return Subtraction of series and other, element-wise (binary operator *sub*).
Series.mul(self, other[, level, fill_value, ...])	Return Multiplication of series and other, element-wise (binary operator *mul*).
Series.div(self, other[, level, fill_value, ...])	Return Floating division of series and other, element-wise (binary operator *truediv*).
Series.truediv(self, other[, level, ...])	Return Floating division of series and other, element-wise (binary operator *truediv*).
Series.floordiv(self, other[, level, ...])	Return Integer division of series and other, element-wise (binary operator *floordiv*).
Series.mod(self, other[, level, fill_value, ...])	Return Modulo of series and other, element-wise (binary operator *mod*).
Series.pow(self, other[, level, fill_value, ...])	Return Exponential power of series and other, element-wise (binary operator *pow*).
Series.radd(self, other[, level, ...])	Return Addition of series and other, element-wise (binary operator *radd*).
Series.rsub(self, other[, level, ...])	Return Subtraction of series and other, element-wise (binary operator *rsub*).
Series.rmul(self, other[, level, ...])	Return Multiplication of series and other, element-wise (binary operator *rmul*).
Series.rdiv(self, other[, level, ...])	Return Floating division of series and other, element-wise (binary operator *rtruediv*).
Series.rtruediv(self, other[, level, ...])	Return Floating division of series and other, element-wise (binary operator *rtruediv*).

continues on next page

Table 34 – continued from previous page

Series.rfloordiv(self, other[, level, ...])	Return Integer division of series and other, element-wise (binary operator *rfloordiv*).
Series.rmod(self, other[, level, ...])	Return Modulo of series and other, element-wise (binary operator *rmod*).
Series.rpow(self, other[, level, ...])	Return Exponential power of series and other, element-wise (binary operator *rpow*).
Series.combine(self, other, func[, fill_value])	Combine the Series with a Series or scalar according to *func*.
Series.combine_first(self, other)	Combine Series values, choosing the calling Series's values first.
Series.round(self[, decimals])	Round each value in a Series to the given number of decimals.
Series.lt(self, other[, level, fill_value, axis])	Return Less than of series and other, element-wise (binary operator *lt*).
Series.gt(self, other[, level, fill_value, axis])	Return Greater than of series and other, element-wise (binary operator *gt*).
Series.le(self, other[, level, fill_value, axis])	Return Less than or equal to of series and other, element-wise (binary operator *le*).
Series.ge(self, other[, level, fill_value, axis])	Return Greater than or equal to of series and other, element-wise (binary operator *ge*).
Series.ne(self, other[, level, fill_value, axis])	Return Not equal to of series and other, element-wise (binary operator *ne*).
Series.eq(self, other[, level, fill_value, axis])	Return Equal to of series and other, element-wise (binary operator *eq*).
Series.product(self[, axis, skipna, level, ...])	Return the product of the values for the requested axis.
Series.dot(self, other)	Compute the dot product between the Series and the columns of other.

3.3.6 Function application, groupby & window

Series.apply(self, func[, convert_dtype, args])	Invoke function on values of Series.
Series.agg(self, func[, axis])	Aggregate using one or more operations over the specified axis.
Series.aggregate(self, func[, axis])	Aggregate using one or more operations over the specified axis.
Series.transform(self, func[, axis])	Call `func` on self producing a Series with transformed values.
Series.map(self, arg[, na_action])	Map values of Series according to input correspondence.
Series.groupby(self[, by, axis, level])	Group Series using a mapper or by a Series of columns.
Series.rolling(self, window[, min_periods, ...])	Provide rolling window calculations.
Series.expanding(self[, min_periods, ...])	Provide expanding transformations.
Series.ewm(self[, com, span, halflife, ...])	Provide exponential weighted functions.
Series.pipe(self, func, *args, **kwargs)	Apply func(self, *args, **kwargs).

3.3.7 Computations / descriptive stats

Series.abs(self)	Return a Series/DataFrame with absolute numeric value of each element.
Series.all(self[, axis, bool_only, skipna, ...])	Return whether all elements are True, potentially over an axis.
Series.any(self[, axis, bool_only, skipna, ...])	Return whether any element is True, potentially over an axis.
Series.autocorr(self[, lag])	Compute the lag-N autocorrelation.
Series.between(self, left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
Series.clip(self[, lower, upper, axis])	Trim values at input threshold(s).
Series.corr(self, other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values.
Series.count(self[, level])	Return number of non-NA/null observations in the Series.
Series.cov(self, other[, min_periods])	Compute covariance with Series, excluding missing values.
Series.cummax(self[, axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
Series.cummin(self[, axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
Series.cumprod(self[, axis, skipna])	Return cumulative product over a DataFrame or Series axis.
Series.cumsum(self[, axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
Series.describe(self[, percentiles, ...])	Generate descriptive statistics.
Series.diff(self[, periods])	First discrete difference of element.
Series.factorize(self[, sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
Series.kurt(self[, axis, skipna, level, ...])	Return unbiased kurtosis over requested axis.
Series.mad(self[, axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
Series.max(self[, axis, skipna, level, ...])	Return the maximum of the values for the requested axis.
Series.mean(self[, axis, skipna, level, ...])	Return the mean of the values for the requested axis.
Series.median(self[, axis, skipna, level, ...])	Return the median of the values for the requested axis.
Series.min(self[, axis, skipna, level, ...])	Return the minimum of the values for the requested axis.
Series.mode(self[, dropna])	Return the mode(s) of the dataset.
Series.nlargest(self[, n, keep])	Return the largest *n* elements.
Series.nsmallest(self[, n, keep])	Return the smallest *n* elements.
Series.pct_change(self[, periods, ...])	Percentage change between the current and a prior element.
Series.prod(self[, axis, skipna, level, ...])	Return the product of the values for the requested axis.
Series.quantile(self[, q, interpolation])	Return value at the given quantile.
Series.rank(self[, axis])	Compute numerical data ranks (1 through n) along axis.
Series.sem(self[, axis, skipna, level, ...])	Return unbiased standard error of the mean over requested axis.
Series.skew(self[, axis, skipna, level, ...])	Return unbiased skew over requested axis.
Series.std(self[, axis, skipna, level, ...])	Return sample standard deviation over requested axis.
Series.sum(self[, axis, skipna, level, ...])	Return the sum of the values for the requested axis.
Series.var(self[, axis, skipna, level, ...])	Return unbiased variance over requested axis.

continues on next page

Table 36 – continued from previous page

Series.kurtosis(self[, axis, skipna, level, ...])	Return unbiased kurtosis over requested axis.
Series.unique(self)	Return unique values of Series object.
Series.nunique(self[, dropna])	Return number of unique elements in the object.
Series.is_unique	Return boolean if values in the object are unique.
Series.is_monotonic	Return boolean if values in the object are monotonic_increasing.
Series.is_monotonic_increasing	Return boolean if values in the object are monotonic_increasing.
Series.is_monotonic_decreasing	Return boolean if values in the object are monotonic_decreasing.
Series.value_counts(self[, normalize, sort, ...])	Return a Series containing counts of unique values.

3.3.8 Reindexing / selection / label manipulation

Series.align(self, other[, join, axis, ...])	Align two objects on their axes with the specified join method.
Series.drop(self[, labels, axis, index, ...])	Return Series with specified index labels removed.
Series.droplevel(self, level[, axis])	Return DataFrame with requested index / column level(s) removed.
Series.drop_duplicates(self[, keep, inplace])	Return Series with duplicate values removed.
Series.duplicated(self[, keep])	Indicate duplicate Series values.
Series.equals(self, other)	Test whether two objects contain the same elements.
Series.first(self, offset)	Method to subset initial periods of time series data based on a date offset.
Series.head(self, n)	Return the first *n* rows.
Series.idxmax(self[, axis, skipna])	Return the row label of the maximum value.
Series.idxmin(self[, axis, skipna])	Return the row label of the minimum value.
Series.isin(self, values)	Check whether *values* are contained in Series.
Series.last(self, offset)	Method to subset final periods of time series data based on a date offset.
Series.reindex(self[, index])	Conform Series to new index with optional filling logic.
Series.reindex_like(self, other, method, ...)	Return an object with matching indices as other object.
Series.rename(self[, index, axis, copy, ...])	Alter Series index labels or name.
Series.rename_axis(self[, mapper, index, ...])	Set the name of the axis for the index or columns.
Series.reset_index(self[, level, drop, ...])	Generate a new DataFrame or Series with the index reset.
Series.sample(self[, n, frac, replace, ...])	Return a random sample of items from an axis of object.
Series.set_axis(self, labels[, axis, inplace])	Assign desired index to given axis.
Series.take(self, indices[, axis, is_copy])	Return the elements in the given *positional* indices along an axis.
Series.tail(self, n)	Return the last *n* rows.
Series.truncate(self[, before, after, axis])	Truncate a Series or DataFrame before and after some index value.
Series.where(self, cond[, other, inplace, ...])	Replace values where the condition is False.
Series.mask(self, cond[, other, inplace, ...])	Replace values where the condition is True.
Series.add_prefix(self, prefix)	Prefix labels with string *prefix*.
Series.add_suffix(self, suffix)	Suffix labels with string *suffix*.

continues on next page

Table 37 – continued from previous page

Series.filter(self[, items, axis])	Subset the dataframe rows or columns according to the specified index labels.

3.3.9 Missing data handling

Series.isna(self)	Detect missing values.
Series.notna(self)	Detect existing (non-missing) values.
Series.dropna(self[, axis, inplace, how])	Return a new Series with missing values removed.
Series.fillna(self[, value, method, axis, ...])	Fill NA/NaN values using the specified method.
Series.interpolate(self[, method, axis, ...])	Interpolate values according to different methods.

3.3.10 Reshaping, sorting

Series.argsort(self[, axis, kind, order])	Override ndarray.argsort.
Series.argmin(self[, axis, skipna])	Return a ndarray of the minimum argument indexer.
Series.argmax(self[, axis, skipna])	Return an ndarray of the maximum argument indexer.
Series.reorder_levels(self, order)	Rearrange index levels using input order.
Series.sort_values(self[, axis, ascending, ...])	Sort by the values.
Series.sort_index(self[, axis, level, ...])	Sort Series by index labels.
Series.swaplevel(self[, i, j, copy])	Swap levels i and j in a MultiIndex.
Series.unstack(self[, level, fill_value])	Unstack, a.k.a.
Series.explode(self)	Transform each element of a list-like to a row, replicating the index values.
Series.searchsorted(self, value[, side, sorter])	Find indices where elements should be inserted to maintain order.
Series.ravel(self[, order])	Return the flattened underlying data as an ndarray.
Series.repeat(self, repeats[, axis])	Repeat elements of a Series.
Series.squeeze(self[, axis])	Squeeze 1 dimensional axis objects into scalars.
Series.view(self[, dtype])	Create a new view of the Series.

3.3.11 Combining / joining / merging

Series.append(self, to_append[, ...])	Concatenate two or more Series.
Series.replace(self[, to_replace, value, ...])	Replace values given in *to_replace* with *value*.
Series.update(self, other)	Modify Series in place using non-NA values from passed Series.

3.3.12 Time series-related

Series.asfreq(self, freq[, method, fill_value])	Convert TimeSeries to specified frequency.
Series.asof(self, where[, subset])	Return the last row(s) without any NaNs before *where*.
Series.shift(self[, periods, freq, axis, ...])	Shift index by desired number of periods with an optional time *freq*.
Series.first_valid_index(self)	Return index for first non-NA/null value.
Series.last_valid_index(self)	Return index for last non-NA/null value.
Series.resample(self, rule[, axis, loffset, ...])	Resample time-series data.
Series.tz_convert(self, tz[, axis, level])	Convert tz-aware axis to target time zone.
Series.tz_localize(self, tz[, axis, level, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
Series.at_time(self, time, asof[, axis])	Select values at particular time of day (e.g.
Series.between_time(self, start_time, ...[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
Series.tshift(self, periods[, freq, axis])	Shift the time index, using the index's frequency if available.
Series.slice_shift(self, periods[, axis])	Equivalent to *shift* without copying data.

3.3.13 Accessors

Pandas provides dtype-specific methods under various accessors. These are separate namespaces within *Series* that only apply to specific data types.

Data Type	Accessor
Datetime, Timedelta, Period	*dt*
String	*str*
Categorical	*cat*
Sparse	*sparse*

Datetimelike properties

`Series.dt` can be used to access the values of the series as datetimelike and return several properties. These can be accessed like `Series.dt.<property>`.

Datetime properties

Series.dt.date	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
Series.dt.time	Returns numpy array of datetime.time.
Series.dt.timetz	Returns numpy array of datetime.time also containing timezone information.
Series.dt.year	The year of the datetime.
Series.dt.month	The month as January=1, December=12.
Series.dt.day	The month as January=1, December=12.
Series.dt.hour	The hours of the datetime.

continues on next page

Table 42 – continued from previous page

Series.dt.minute	The minutes of the datetime.
Series.dt.second	The seconds of the datetime.
Series.dt.microsecond	The microseconds of the datetime.
Series.dt.nanosecond	The nanoseconds of the datetime.
Series.dt.week	The week ordinal of the year.
Series.dt.weekofyear	The week ordinal of the year.
Series.dt.dayofweek	The day of the week with Monday=0, Sunday=6.
Series.dt.weekday	The day of the week with Monday=0, Sunday=6.
Series.dt.dayofyear	The ordinal day of the year.
Series.dt.quarter	The quarter of the date.
Series.dt.is_month_start	Indicates whether the date is the first day of the month.
Series.dt.is_month_end	Indicates whether the date is the last day of the month.
Series.dt.is_quarter_start	Indicator for whether the date is the first day of a quarter.
Series.dt.is_quarter_end	Indicator for whether the date is the last day of a quarter.
Series.dt.is_year_start	Indicate whether the date is the first day of a year.
Series.dt.is_year_end	Indicate whether the date is the last day of the year.
Series.dt.is_leap_year	Boolean indicator if the date belongs to a leap year.
Series.dt.daysinmonth	The number of days in the month.
Series.dt.days_in_month	The number of days in the month.
Series.dt.tz	Return timezone, if any.
Series.dt.freq	Return the frequency object for this PeriodArray.

pandas.Series.dt.date

Series.dt.**date**
Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).

pandas.Series.dt.time

Series.dt.**time**
Returns numpy array of datetime.time. The time part of the Timestamps.

pandas.Series.dt.timetz

Series.dt.**timetz**
Returns numpy array of datetime.time also containing timezone information. The time part of the Timestamps.

pandas.Series.dt.year

Series.dt.**year**
The year of the datetime.

pandas.Series.dt.month

Series.dt.**month**
The month as January=1, December=12.

pandas.Series.dt.day

Series.dt.**day**
The month as January=1, December=12.

pandas.Series.dt.hour

Series.dt.**hour**
The hours of the datetime.

pandas.Series.dt.minute

Series.dt.**minute**
The minutes of the datetime.

pandas.Series.dt.second

Series.dt.**second**
The seconds of the datetime.

pandas.Series.dt.microsecond

Series.dt.**microsecond**
The microseconds of the datetime.

pandas.Series.dt.nanosecond

Series.dt.**nanosecond**
The nanoseconds of the datetime.

pandas.Series.dt.week

Series.dt.**week**
The week ordinal of the year.

pandas.Series.dt.weekofyear

Series.dt.**weekofyear**
The week ordinal of the year.

pandas.Series.dt.dayofweek

Series.dt.**dayofweek**
The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.
Series.dt.weekday Alias.
Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.Series.dt.weekday

Series.dt.**weekday**
The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.
Series.dt.weekday Alias.
Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.Series.dt.dayofyear

Series.dt.**dayofyear**
The ordinal day of the year.

pandas.Series.dt.quarter

Series.dt.**quarter**
The quarter of the date.

pandas.Series.dt.is_month_start

Series.dt.**is_month_start**
Indicates whether the date is the first day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.
is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
```

(continues on next page)

(continued from previous page)

```
dtype: bool
>>> s.dt.is_month_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.Series.dt.is_month_end

Series.dt.**is_month_end**

Indicates whether the date is the last day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.
is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2     True
dtype: bool
>>> s.dt.is_month_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.Series.dt.is_quarter_start

Series.dt.**is_quarter_start**

Indicator for whether the date is the first day of a quarter.

Returns

is_quarter_start [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.
is_quarter_end Similar property for indicating the quarter start.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                   periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
       dates  quarter  is_quarter_start
0 2017-03-30        1             False
1 2017-03-31        1             False
2 2017-04-01        2              True
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

pandas.Series.dt.is_quarter_end

Series.dt.**is_quarter_end**

Indicator for whether the date is the last day of a quarter.

Returns

is_quarter_end [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.
is_quarter_start Similar property indicating the quarter start.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                    periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
       dates  quarter    is_quarter_end
0 2017-03-30        1             False
1 2017-03-31        1              True
2 2017-04-01        2             False
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

pandas.Series.dt.is_year_start

Series.dt.**is_year_start**

Indicate whether the date is the first day of a year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_end Similar property indicating the last day of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2    True
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
```

(continues on next page)

(continued from previous page)

```
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

pandas.Series.dt.is_year_end

Series.dt.**is_year_end**

Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

pandas.Series.dt.is_leap_year

Series.dt.**is_leap_year**

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

Returns

Series or ndarray Booleans indicating if dates belong to a leap year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```
>>> dates = pd.Series(idx)
>>> dates_series
0   2012-12-31
1   2013-12-31
2   2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool
```

pandas.Series.dt.daysinmonth

Series.dt.**daysinmonth**

The number of days in the month.

pandas.Series.dt.days_in_month

Series.dt.**days_in_month**

The number of days in the month.

pandas.Series.dt.tz

Series.dt.**tz**

Return timezone, if any.

Returns

datetime.tzinfo, pytz.tzinfo.BaseTZInfo, dateutil.tz.tz.tzfile, or None Returns None when the array is tz-naive.

pandas.Series.dt.freq

Series.dt.**freq**

Datetime methods

Series.dt.to_period(self, *args, **kwargs)	Cast to PeriodArray/Index at a particular frequency.
Series.dt.to_pydatetime(self)	Return the data as an array of native Python datetime objects.
Series.dt.tz_localize(self, *args, **kwargs)	Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.
Series.dt.tz_convert(self, *args, **kwargs)	Convert tz-aware Datetime Array/Index from one time zone to another.
Series.dt.normalize(self, *args, **kwargs)	Convert times to midnight.
Series.dt.strftime(self, *args, **kwargs)	Convert to Index using specified date_format.
Series.dt.round(self, *args, **kwargs)	Perform round operation on the data to the specified *freq*.
Series.dt.floor(self, *args, **kwargs)	Perform floor operation on the data to the specified *freq*.
Series.dt.ceil(self, *args, **kwargs)	Perform ceil operation on the data to the specified *freq*.
Series.dt.month_name(self, *args, **kwargs)	Return the month names of the DateTimeIndex with specified locale.
Series.dt.day_name(self, *args, **kwargs)	Return the day names of the DateTimeIndex with specified locale.

pandas.Series.dt.to_period

Series.dt.**to_period**(*self*, **args*, ***kwargs*)

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

Parameters

freq [str or Offset, optional] One of pandas' *offset strings* or an Offset object. Will be inferred by default.

Returns

PeriodArray/Index

Raises

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

See also:

PeriodIndex Immutable ndarray holding ordinal values.
DatetimeIndex.to_pydatetime Return DatetimeIndex as object.

Examples

```
>>> df = pd.DataFrame({"y": [1, 2, 3]},
...                   index=pd.to_datetime(["2000-03-31 00:00:00",
...                                         "2000-05-31 00:00:00",
...                                         "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

pandas.Series.dt.to_pydatetime

Series.dt.**to_pydatetime**(*self*)

Return the data as an array of native Python datetime objects.

Timezone information is retained if present.

Warning: Python's datetime uses microsecond resolution, which is lower than pandas (nanosecond). The values are truncated.

Returns

numpy.ndarray Object dtype array containing native Python datetime objects.

See also:

datetime.datetime Standard library value for a datetime.

Examples

```
>>> s = pd.Series(pd.date_range('20180310', periods=2))
>>> s
0   2018-03-10
1   2018-03-11
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 11, 0, 0)], dtype=object)
```

pandas' nanosecond precision is truncated to microseconds.

```
>>> s = pd.Series(pd.date_range('20180310', periods=2, freq='ns'))
>>> s
0   2018-03-10 00:00:00.000000000
1   2018-03-10 00:00:00.000000001
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 10, 0, 0)], dtype=object)
```

pandas.Series.dt.tz_localize

Series.dt.**tz_localize**(*self*, **args*, ***kwargs*)

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

ambiguous ['infer', 'NaT', bool array, 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 signifies 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 ['shift_forward', 'shift_backward, 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- '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.

New in version 0.24.0.

Returns

Same type as self Array/Index converted to the specified time zone.

Raises

TypeError If the Datetime Array/Index is tz-aware and tz is not None.

See also:

DatetimeIndex.tz_convert Convert tz-aware DatetimeIndex from one time zone to another.

Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
               '2018-03-02 09:00:00-05:00',
               '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the `tz=None`, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.to_datetime(pd.Series(['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.dt.tz_localize('CET', ambiguous='infer')
0   2018-10-28 01:30:00+02:00
1   2018-10-28 02:00:00+02:00
2   2018-10-28 02:30:00+02:00
3   2018-10-28 02:00:00+01:00
4   2018-10-28 02:30:00+01:00
5   2018-10-28 03:00:00+01:00
6   2018-10-28 03:30:00+01:00
dtype: datetime64[ns, CET]
```

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.to_datetime(pd.Series(['2018-10-28 01:20:00',
...                               '2018-10-28 02:36:00',
...                               '2018-10-28 03:46:00']))
>>> s.dt.tz_localize('CET', ambiguous=np.array([True, True, False]))
```

(continues on next page)

(continued from previous page)

```
0   2015-03-29 03:00:00+02:00
1   2015-03-29 03:30:00+02:00
dtype: datetime64[ns, Europe/Warsaw]
```

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.to_datetime(pd.Series(['2015-03-29 02:30:00',
...                               '2015-03-29 03:30:00']))
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
0   2015-03-29 03:00:00+02:00
1   2015-03-29 03:30:00+02:00
dtype: datetime64[ns, 'Europe/Warsaw']
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
0   2015-03-29 01:59:59.999999999+01:00
1   2015-03-29 03:30:00+02:00
dtype: datetime64[ns, 'Europe/Warsaw']
>>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
0   2015-03-29 03:30:00+02:00
1   2015-03-29 03:30:00+02:00
dtype: datetime64[ns, 'Europe/Warsaw']
```

pandas.Series.dt.tz_convert

`Series.dt.`**`tz_convert`**(*self*, **args*, ***kwargs*)

Convert tz-aware Datetime Array/Index from one time zone to another.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A *tz* of None will convert to UTC and remove the timezone information.

Returns

Array or Index

Raises

TypeError If Datetime Array/Index is tz-naive.

See also:

DatetimeIndex.tz A timezone that has a variable offset from UTC.

DatetimeIndex.tz_localize Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.

Examples

With the *tz* parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.date_range(start='2014-08-01 09:00',
...                     freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
```

(continues on next page)