pandas.DataFrame.to_timestamp

DataFrame.**to_timestamp**(*self*, *freq=None*, *how='start'*, *axis=0*, *copy=True*) → 'DataFrame'

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

freq [str, default frequency of PeriodIndex] Desired frequency.

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default).

copy [bool, default True] If False then underlying input data is not copied.

Returns

DataFrame with DatetimeIndex

pandas.DataFrame.to_xarray

DataFrame.**to_xarray**(*self*)

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
```

```
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

pandas.DataFrame.transform

DataFrame.**transform**(*self*, *func*, *axis=0*, **args*, ***kwargs*) → 'DataFrame'

Call `func` on self producing a DataFrame with transformed values.

Produced DataFrame will have same axis length as self.

Parameters

func [function, str, list or dict] Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

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

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame A DataFrame that must have the same length as self.

Raises

ValueError [If the returned DataFrame has a different length than self.]

See also:

DataFrame.agg Only perform aggregating type operations.

DataFrame.apply Invoke function on a DataFrame.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

pandas.DataFrame.transpose

DataFrame.**transpose**(*self*, **args*, *copy: bool = False*) → 'DataFrame'

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

Parameters

***args** [tuple, optional] Accepted for compatibility with NumPy.

copy [bool, default False] Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame The transposed DataFrame.

See also:

numpy.transpose Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
```

(continues on next page)

(continued from previous page)

```
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

pandas.DataFrame.truediv

DataFrame.**truediv**(*self*, *other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

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

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

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.truncate

DataFrame.**truncate**(*self: ~ FrameOrSeries*, *before=None*, *after=None*, *axis=None*, *copy: bool = True*) → ~FrameOrSeries

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before [date, str, int] Truncate all rows before this index value.

after [date, str, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [bool, default is True,] Return a copy of the truncated section.

Returns

type of caller The truncated Series or DataFrame.

See also:

DataFrame.loc Select a subset of a DataFrame by label.

DataFrame.iloc Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                   index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...              after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

pandas.DataFrame.tshift

DataFrame.**tshift**(*self: ~ FrameOrSeries*, *periods: int = 1*, *freq=None*, *axis=0*) → ~FrameOrSeries

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative.

freq [DateOffset, timedelta, or str, default None] Increment to use from the tseries module or time rule expressed as a string (e.g. 'EOM').

axis [{0 or 'index', 1 or 'columns', None}, default 0] Corresponds to the axis that contains the Index.

Returns

shifted [Series/DataFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

pandas.DataFrame.tz_convert

DataFrame.**tz_convert**(*self: ~ FrameOrSeries*, *tz*, *axis=0*, *level=None*, *copy: bool = True*) → ~FrameOrSeries

Convert tz-aware axis to target time zone.

Parameters

tz [str or tzinfo object]

axis [the axis to convert]

level [int, str, default None] If axis is a MultiIndex, convert a specific level. Otherwise must be None.

copy [bool, default True] Also make a copy of the underlying data.

Returns

%(klass)s Object with time zone converted axis.

Raises

TypeError If the axis is tz-naive.

pandas.DataFrame.tz_localize

DataFrame.**tz_localize**(*self: ~ FrameOrSeries*, *tz*, *axis=0*, *level=None*, *copy: bool = True*, *ambiguous='raise'*, *nonexistent: str = 'raise'*) → ~FrameOrSeries

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use *Series.dt.tz_localize()*.

Parameters

tz [str or tzinfo]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None.

copy [bool, default True] Also make a copy of the underlying data.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

nonexistent [str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

Series or DataFrame Same type as the input.

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series([1],
...               index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

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

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
...                                       '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.Series(range(2), ... index=pd.DatetimeIndex(['2015-03-29 02:30:00', ... '2015-03-29 03:30:00'])) >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 2015-03-29 03:00:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 2015-03-29 01:59:59.999999999+01:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 2015-03-29 03:30:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64

pandas.DataFrame.unstack

DataFrame.**unstack**(*self*, *level=- 1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

The level involved will automatically get sorted.

Parameters

level [int, str, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name.

fill_value [int, str or dict] Replace NaN with this value if the unstack produces missing values.

Returns

Series or DataFrame

See also:

DataFrame.pivot Pivot a table based on column values.

DataFrame.stack Pivot a level of the column labels (inverse operation from *unstack*).

Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

pandas.DataFrame.update

DataFrame.**update**(*self*, *other*, *join='left'*, *overwrite=True*, *filter_func=None*, *errors='ignore'*) → None

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

Parameters

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> bool 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors [{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Changed in version 0.24.0: Changed from *raise_conflict=False|True* to *errors='ignore'|'raise'*.

Returns

None [method directly changes calling object]

Raises

ValueError

- When *errors='raise'* and there's overlapping non-NA data.
- When *errors* is not either *'ignore'* or *'raise'*

NotImplementedError

- If *join != 'left'*

See also:

`dict.update` Similar method for dictionaries.

DataFrame.merge For column(s)-on-columns(s) operations.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

pandas.DataFrame.var

DataFrame.**var**(*self*, *axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased variance over requested axis.

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

Parameters

axis [{index (0), columns (1)}]

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 Series.

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

Series or DataFrame (if level specified)

pandas.DataFrame.where

DataFrame.**where**(*self*, *cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*)

Replace values where the condition is False.

Parameters

cond [bool Series/DataFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other [scalar, Series/DataFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it).

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

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default 'raise'] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- 'raise' : allow exceptions to be raised.
- 'ignore' : suppress exceptions. On error return original object.

try_cast [bool, default False] Try to cast the result back to the input type (if possible).

Returns

Same type as caller

See also:

DataFrame.mask() Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for *`DataFrame.where()`* differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> 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.DataFrame.xs

DataFrame.**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
```

3.4.2 Attributes and underlying data

Axes

DataFrame.index	The index (row labels) of the DataFrame.
DataFrame.columns	The column labels of the DataFrame.

DataFrame.dtypes	Return the dtypes in the DataFrame.
DataFrame.select_dtypes(self[, include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
DataFrame.values	Return a Numpy representation of the DataFrame.
DataFrame.axes	Return a list representing the axes of the DataFrame.

continues on next page

Table 61 – continued from previous page

DataFrame.ndim	Return an int representing the number of axes / array dimensions.
DataFrame.size	Return an int representing the number of elements in this object.
DataFrame.shape	Return a tuple representing the dimensionality of the DataFrame.
DataFrame.memory_usage(self[, index, deep])	Return the memory usage of each column in bytes.
DataFrame.empty	Indicator whether DataFrame is empty.

3.4.3 Conversion

DataFrame.astype(self, dtype, copy, errors)	Cast a pandas object to a specified dtype `dtype`.
DataFrame.convert_dtypes(self, ...)	Convert columns to best possible dtypes using dtypes supporting `pd.NA`.
DataFrame.infer_objects(self)	Attempt to infer better dtypes for object columns.
DataFrame.copy(self, deep)	Make a copy of this object's indices and data.
DataFrame.isna(self)	Detect missing values.
DataFrame.notna(self)	Detect existing (non-missing) values.
DataFrame.bool(self)	Return the bool of a single element PandasObject.

3.4.4 Indexing, iteration

DataFrame.head(self, n)	Return the first *n* rows.
DataFrame.at	Access a single value for a row/column label pair.
DataFrame.iat	Access a single value for a row/column pair by integer position.
DataFrame.loc	Access a group of rows and columns by label(s) or a boolean array.
DataFrame.iloc	Purely integer-location based indexing for selection by position.
DataFrame.insert(self, loc, column, value[, ...])	Insert column into DataFrame at specified location.
DataFrame.__iter__(self)	Iterate over info axis.
DataFrame.items(self)	Iterate over (column name, Series) pairs.
DataFrame.iteritems(self)	Iterate over (column name, Series) pairs.
DataFrame.keys(self)	Get the 'info axis' (see Indexing for more).
DataFrame.iterrows(self)	Iterate over DataFrame rows as (index, Series) pairs.
DataFrame.itertuples(self[, index, name])	Iterate over DataFrame rows as namedtuples.
DataFrame.lookup(self, row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
DataFrame.pop(self, item)	Return item and drop from frame.
DataFrame.tail(self, n)	Return the last *n* rows.
DataFrame.xs(self, key[, axis, level])	Return cross-section from the Series/DataFrame.
DataFrame.get(self, key[, default])	Get item from object for given key (ex: DataFrame column).
DataFrame.isin(self, values)	Whether each element in the DataFrame is contained in values.
DataFrame.where(self, cond[, other, ...])	Replace values where the condition is False.
DataFrame.mask(self, cond[, other, inplace, ...])	Replace values where the condition is True.

continues on next page

Table 63 – continued from previous page

DataFrame.query(self, expr[, inplace])	Query the columns of a DataFrame with a boolean expression.

pandas.DataFrame.__iter__

DataFrame.**__iter__**(*self*)

Iterate over info axis.

Returns

iterator Info axis as iterator.

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

3.4.5 Binary operator functions

DataFrame.add(self, other[, axis, level, ...])	Get Addition of dataframe and other, element-wise (binary operator *add*).
DataFrame.sub(self, other[, axis, level, ...])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
DataFrame.mul(self, other[, axis, level, ...])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).
DataFrame.div(self, other[, axis, level, ...])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
DataFrame.truediv(self, other[, axis, ...])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
DataFrame.floordiv(self, other[, axis, ...])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
DataFrame.mod(self, other[, axis, level, ...])	Get Modulo of dataframe and other, element-wise (binary operator *mod*).
DataFrame.pow(self, other[, axis, level, ...])	Get Exponential power of dataframe and other, element-wise (binary operator *pow*).
DataFrame.dot(self, other)	Compute the matrix multiplication between the DataFrame and other.
DataFrame.radd(self, other[, axis, level, ...])	Get Addition of dataframe and other, element-wise (binary operator *radd*).
DataFrame.rsub(self, other[, axis, level, ...])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
DataFrame.rmul(self, other[, axis, level, ...])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).
DataFrame.rdiv(self, other[, axis, level, ...])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
DataFrame.rtruediv(self, other[, axis, ...])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
DataFrame.rfloordiv(self, other[, axis, ...])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
DataFrame.rmod(self, other[, axis, level, ...])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
DataFrame.rpow(self, other[, axis, level, ...])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).

continues on next page

Table 64 – continued from previous page

DataFrame.lt(self, other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
DataFrame.gt(self, other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).
DataFrame.le(self, other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).
DataFrame.ge(self, other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
DataFrame.ne(self, other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator *ne*).
DataFrame.eq(self, other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
DataFrame.combine(self, other, func[, ...])	Perform column-wise combine with another DataFrame.
DataFrame.combine_first(self, other)	Update null elements with value in the same location in *other*.

3.4.6 Function application, GroupBy & window

DataFrame.apply(self, func[, axis, raw, ...])	Apply a function along an axis of the DataFrame.
DataFrame.applymap(self, func)	Apply a function to a Dataframe elementwise.
DataFrame.pipe(self, func, *args, **kwargs)	Apply func(self, *args, **kwargs).
DataFrame.agg(self, func[, axis])	Aggregate using one or more operations over the specified axis.
DataFrame.aggregate(self, func[, axis])	Aggregate using one or more operations over the specified axis.
DataFrame.transform(self, func[, axis])	Call `func` on self producing a DataFrame with transformed values.
DataFrame.groupby(self[, by, axis, level])	Group DataFrame using a mapper or by a Series of columns.
DataFrame.rolling(self, window[, ...])	Provide rolling window calculations.
DataFrame.expanding(self[, min_periods, ...])	Provide expanding transformations.
DataFrame.ewm(self[, com, span, halflife, ...])	Provide exponential weighted functions.

3.4.7 Computations / descriptive stats

DataFrame.abs(self)	Return a Series/DataFrame with absolute numeric value of each element.
DataFrame.all(self[, axis, bool_only, ...])	Return whether all elements are True, potentially over an axis.
DataFrame.any(self[, axis, bool_only, ...])	Return whether any element is True, potentially over an axis.
DataFrame.clip(self[, lower, upper, axis])	Trim values at input threshold(s).
DataFrame.corr(self[, method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values.
DataFrame.corrwith(self, other[, axis, ...])	Compute pairwise correlation.
DataFrame.count(self[, axis, level, ...])	Count non-NA cells for each column or row.

continues on next page

Table 66 – continued from previous page

DataFrame.cov(self[, min_periods])	Compute pairwise covariance of columns, excluding NA/null values.
DataFrame.cummax(self[, axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
DataFrame.cummin(self[, axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
DataFrame.cumprod(self[, axis, skipna])	Return cumulative product over a DataFrame or Series axis.
DataFrame.cumsum(self[, axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
DataFrame.describe(self[, percentiles, ...])	Generate descriptive statistics.
DataFrame.diff(self[, periods, axis])	First discrete difference of element.
DataFrame.eval(self, expr[, inplace])	Evaluate a string describing operations on DataFrame columns.
DataFrame.kurt(self[, axis, skipna, level, ...])	Return unbiased kurtosis over requested axis.
DataFrame.kurtosis(self[, axis, skipna, ...])	Return unbiased kurtosis over requested axis.
DataFrame.mad(self[, axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
DataFrame.max(self[, axis, skipna, level, ...])	Return the maximum of the values for the requested axis.
DataFrame.mean(self[, axis, skipna, level, ...])	Return the mean of the values for the requested axis.
DataFrame.median(self[, axis, skipna, ...])	Return the median of the values for the requested axis.
DataFrame.min(self[, axis, skipna, level, ...])	Return the minimum of the values for the requested axis.
DataFrame.mode(self[, axis, numeric_only, ...])	Get the mode(s) of each element along the selected axis.
DataFrame.pct_change(self[, periods, ...])	Percentage change between the current and a prior element.
DataFrame.prod(self[, axis, skipna, level, ...])	Return the product of the values for the requested axis.
DataFrame.product(self[, axis, skipna, ...])	Return the product of the values for the requested axis.
DataFrame.quantile(self[, q, axis, ...])	Return values at the given quantile over requested axis.
DataFrame.rank(self[, axis])	Compute numerical data ranks (1 through n) along axis.
DataFrame.round(self[, decimals])	Round a DataFrame to a variable number of decimal places.
DataFrame.sem(self[, axis, skipna, level, ...])	Return unbiased standard error of the mean over requested axis.
DataFrame.skew(self[, axis, skipna, level, ...])	Return unbiased skew over requested axis.
DataFrame.sum(self[, axis, skipna, level, ...])	Return the sum of the values for the requested axis.
DataFrame.std(self[, axis, skipna, level, ...])	Return sample standard deviation over requested axis.
DataFrame.var(self[, axis, skipna, level, ...])	Return unbiased variance over requested axis.
DataFrame.nunique(self[, axis, dropna])	Count distinct observations over requested axis.

3.4.8 Reindexing / selection / label manipulation

DataFrame.add_prefix(self, prefix)	Prefix labels with string *prefix*.
DataFrame.add_suffix(self, suffix)	Suffix labels with string *suffix*.
DataFrame.align(self, other[, join, axis, ...])	Align two objects on their axes with the specified join method.
DataFrame.at_time(self, time, asof[, axis])	Select values at particular time of day (e.g.
DataFrame.between_time(self, start_time, ...)	Select values between particular times of the day (e.g., 9:00-9:30 AM).
DataFrame.drop(self[, labels, axis, index, ...])	Drop specified labels from rows or columns.

continues on next page

Table 67 – continued from previous page

DataFrame.drop_duplicates(self, subset, ...)	Return DataFrame with duplicate rows removed.
DataFrame.duplicated(self, subset, ...)	Return boolean Series denoting duplicate rows.
DataFrame.equals(self, other)	Test whether two objects contain the same elements.
DataFrame.filter(self[, items, axis])	Subset the dataframe rows or columns according to the specified index labels.
DataFrame.first(self, offset)	Method to subset initial periods of time series data based on a date offset.
DataFrame.head(self, n)	Return the first *n* rows.
DataFrame.idxmax(self[, axis, skipna])	Return index of first occurrence of maximum over requested axis.
DataFrame.idxmin(self[, axis, skipna])	Return index of first occurrence of minimum over requested axis.
DataFrame.last(self, offset)	Method to subset final periods of time series data based on a date offset.
DataFrame.reindex(self[, labels, index, ...])	Conform DataFrame to new index with optional filling logic.
DataFrame.reindex_like(self, other, method, ...)	Return an object with matching indices as other object.
DataFrame.rename(self[, mapper, index, ...])	Alter axes labels.
DataFrame.rename_axis(self[, mapper, index, ...])	Set the name of the axis for the index or columns.
DataFrame.reset_index(self, level, ...)	Reset the index, or a level of it.
DataFrame.sample(self[, n, frac, replace, ...])	Return a random sample of items from an axis of object.
DataFrame.set_axis(self, labels[, axis, inplace])	Assign desired index to given axis.
DataFrame.set_index(self, keys[, drop, ...])	Set the DataFrame index using existing columns.
DataFrame.tail(self, n)	Return the last *n* rows.
DataFrame.take(self, indices[, axis])	Return the elements in the given *positional* indices along an axis.
DataFrame.truncate(self[, before, after, axis])	Truncate a Series or DataFrame before and after some index value.

3.4.9 Missing data handling

DataFrame.dropna(self[, axis, how, thresh, ...])	Remove missing values.
DataFrame.fillna(self[, value, method, ...])	Fill NA/NaN values using the specified method.
DataFrame.replace(self[, to_replace, value, ...])	Replace values given in *to_replace* with *value*.
DataFrame.interpolate(self[, method, axis, ...])	Interpolate values according to different methods.

3.4.10 Reshaping, sorting, transposing

DataFrame.droplevel(self, level[, axis])	Return DataFrame with requested index / column level(s) removed.
DataFrame.pivot(self[, index, columns, values])	Return reshaped DataFrame organized by given index / column values.
DataFrame.pivot_table(self[, values, index, ...])	Create a spreadsheet-style pivot table as a DataFrame.

continues on next page