(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.DataFrame.nsmallest

`DataFrame.`**`nsmallest`**(*self*, *n*, *columns*, *keep='first'*) → 'DataFrame'

Return the first *n* rows ordered by *columns* in ascending order.

Return the first *n* rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant.

Parameters

n [int] Number of items to retrieve.

columns [list or str] Column name or names to order by.

keep [{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : take the first occurrence.
- `last` : take the last occurrence.
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

New in version 0.24.0.

Returns

DataFrame

See also:

DataFrame.nlargest Return the first *n* rows ordered by *columns* in descending order.

DataFrame.sort_values Sort DataFrame by the values.

DataFrame.head Return the first *n* rows without re-ordering.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column "a".

```
>>> df.nsmallest(3, 'population')
          population  GDP alpha-2
Nauru          11300  182      NR
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nsmallest(3, 'population', keep='last')
          population  GDP alpha-2
Anguilla       11300  311      AI
Tuvalu         11300   38      TV
Nauru          11300  182      NR
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nsmallest(3, 'population', keep='all')
          population  GDP alpha-2
Nauru          11300  182      NR
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Nauru          11300  182      NR
Anguilla       11300  311      AI
```

pandas.DataFrame.nunique

DataFrame.**nunique**(*self*, *axis=0*, *dropna=True*) → pandas.core.series.Series

Count distinct observations over requested axis.

Return Series with number of distinct observations. Can ignore NaN values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

dropna [bool, default True] Don't include NaN in the counts.

Returns

Series

See also:

Series.nunique Method nunique for Series.

DataFrame.count Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

pandas.DataFrame.pct_change

DataFrame.**pct_change**(*self: ~ FrameOrSeries*, *periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*) → ~FrameOrSeries

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or str, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

chg [Series or DataFrame] The same type as the calling object.

See also:

`Series.diff` Compute the difference of two elements in a Series.

`DataFrame.diff` Compute the difference of two elements in a DataFrame.

`Series.shift` Shift the index by some number of periods.

`DataFrame.shift` Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pandas.DataFrame.pipe

`DataFrame.pipe(self, func, *args, **kwargs)`

Apply func(self, *args, **kwargs).

Parameters

func [function] Function to apply to the Series/DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the Series/DataFrame.

args [iterable, optional] Positional arguments passed into `func`.

kwargs [mapping, optional] A dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

DataFrame.apply

DataFrame.applymap

Series.map

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.pivot

`DataFrame.pivot(self, index=None, columns=None, values=None) → 'DataFrame'`

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the *User Guide* for more on reshaping.

Parameters

index [str or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [str or object] Column to use to make new frame's columns.

values [str, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

Returns

DataFrame Returns reshaped DataFrame.

Raises

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

`DataFrame.pivot_table` Generalization of pivot that can handle duplicate values for one index/column pair.

`DataFrame.unstack` Pivot based on the index values instead of a column.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pandas.DataFrame.pivot_table

`DataFrame.pivot_table`(*self*, *values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*, *observed=False*) → 'DataFrame'

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values [column to aggregate, optional]

index [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions.

fill_value [scalar, default None] Value to replace missing values with.

margins [bool, default False] Add all row / columns (e.g. for subtotal / grand totals).

dropna [bool, default True] Do not include columns whose entries are all NaN.

margins_name [str, default 'All'] Name of the row / column that will contain the totals when margins is True.

observed [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Changed in version 0.25.0.

Returns

DataFrame An Excel style pivot table.

See also:

DataFrame.pivot Pivot without aggregation that can handle non-numeric data.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum, fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': np.mean})
>>> table
                D         E
A   C
bar large  5.500000  7.500000
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                D    E
             mean  max      mean  min
A   C
bar large  5.500000  9.0  7.500000  6.0
    small  5.500000  9.0  8.500000  8.0
foo large  2.000000  5.0  4.500000  4.0
    small  2.333333  6.0  4.333333  2.0
```

pandas.DataFrame.plot

`DataFrame.plot(self, *args, **kwargs)`

Make plots of Series or DataFrame.

Uses the backend specified by the option `plotting.backend`. By default, matplotlib is used.

Parameters

data [Series or DataFrame] The object for which the method is called.

x [label or position, default None] Only used if data is a DataFrame.

y [label, position or list of label, positions, default None] Allows plotting of one column versus another. Only used if data is a DataFrame.

kind [str] The kind of plot to produce:

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot
- 'scatter' : scatter plot
- 'hexbin' : hexbin plot.

figsize [a tuple (width, height) in inches]

use_index [bool, default True] Use index as ticks for x axis.

title [str or list] Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and *subplots* is True, print each item in the list above the corresponding subplot.

grid [bool, default None (matlab style default)] Axis grid lines.

legend [bool or {'reverse'}] Place legend on axis subplots.

style [list or dict] The matplotlib line style per column.

logx [bool or 'sym', default False] Use log scaling or symlog scaling on x axis. .. versionchanged:: 0.25.0

logy [bool or 'sym' default False] Use log scaling or symlog scaling on y axis. .. versionchanged:: 0.25.0

loglog [bool or 'sym', default False] Use log scaling or symlog scaling on both x and y axes. .. versionchanged:: 0.25.0

xticks [sequence] Values to use for the xticks.

yticks [sequence] Values to use for the yticks.

xlim [2-tuple/list]

ylim [2-tuple/list]

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots).

fontsize [int, default None] Font size for xticks and yticks.

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

colorbar [bool, optional] If True, plot colorbar (only relevant for 'scatter' and 'hexbin' plots).

position [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center).

table [bool, Series or DataFrame, default False] If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib's default layout. If a Series or DataFrame is passed, use passed data to draw a table.

yerr [DataFrame, Series, array-like, dict and str] See *Plotting with Error Bars* for detail.

xerr [DataFrame, Series, array-like, dict and str] Equivalent to yerr.

mark_right [bool, default True] When using a secondary_y axis, automatically mark the column labels with "(right)" in the legend.

include_bool [bool, default is False] If True, boolean values can be plotted.

backend [str, default None] Backend to use instead of the backend specified in the option `plotting.backend`. For instance, 'matplotlib'. Alternatively, to specify the `plotting.backend` for the whole session, set `pd.options.plotting.backend`.

New in version 1.0.0.

****kwargs** Options to pass to matplotlib plotting method.

Returns

`matplotlib.axes.Axes` or numpy.ndarray of them If the backend is not the default matplotlib one, the return value will be the object returned by the backend.

Notes

- See matplotlib documentation online for more on this subject
- If *kind* = 'bar' or 'barh', you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

pandas.DataFrame.pop

DataFrame.**pop**(*self: ~ FrameOrSeries*, *item*) → ~FrameOrSeries

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Label of column to be popped.

Returns

Series

Examples

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

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pandas.DataFrame.pow

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

Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

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

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.prod

`DataFrame.prod`(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.DataFrame.product

`DataFrame.`**`product`**(*self*, *axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

Series or DataFrame (if level specified)

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.DataFrame.quantile

`DataFrame.`**`quantile`**(*self*, *q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis.

Parameters

q [float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis [{0, 1, 'index', 'columns'} (default 0)] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only [bool, default True] If False, the quantile of datetime and timedelta data will be computed as well.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

Returns

Series or DataFrame

If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.

If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

See also:

core.window.Rolling.quantile Rolling quantile.

numpy.percentile Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.DataFrame.query

`DataFrame.query`(*self*, *expr*, *inplace=False*, ***kwargs*)

Query the columns of a DataFrame with a boolean expression.

Parameters

expr [str] The query string to evaluate.

You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

You can refer to column names that contain spaces or operators by surrounding them in backticks. This way you can also escape names that start with a digit, or those that are a Python keyword. Basically when it is not valid Python identifier. See notes down for more details.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be `` `a a` + b ``.

New in version 0.25.0: Backtick quoting introduced.

New in version 1.0.0: Expanding functionality of backtick quoting for more than only spaces.

inplace [bool] Whether the query should modify the data in place or return a modified copy.

****kwargs** See the documentation for `eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

DataFrame DataFrame resulting from the provided query expression.

See also:

`eval` Evaluate a string describing operations on DataFrame columns.

`DataFrame.eval` Evaluate a string describing operations on DataFrame columns.

Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

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

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, ``` `it's` > `that's` ``` will raise an error, as it forms a quoted string (``` 's > `that' ```) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in `pandas.core.computation.parsing`.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1)})
>>> df
   A   B  C C
0  1  10   10
1  2   8    9
2  3   6    8
3  4   4    7
4  5   2    6
>>> df.query('A > B')
   A  B  C C
4  5  2    6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A  B  C C
4  5  2    6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A   B  C C
0  1  10   10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A   B  C C
0  1  10   10
```

pandas.DataFrame.radd

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

Get Addition of dataframe and other, element-wise (binary operator *radd*).

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

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.rank

`DataFrame.rank`(*self: ~ FrameOrSeries, axis=0, method: str = 'average', numeric_only: Union[bool, NoneType] = None, na_option: str = 'keep', ascending: bool = True, pct: bool = False*) → ~FrameOrSeries

Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] Index to direct ranking.

method [{'average', 'min', 'max', 'first', 'dense'}, default 'average'] How to rank the group of records that have the same value (i.e. ties):

- average: average rank of the group
- min: lowest rank in the group
- max: highest rank in the group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups.

numeric_only [bool, optional] For DataFrame objects, rank only numeric columns if set to True.

na_option [{'keep', 'top', 'bottom'}, default 'keep'] How to rank NaN values:

- keep: assign NaN rank to NaN values
- top: assign smallest rank to NaN values if ascending
- bottom: assign highest rank to NaN values if ascending.

ascending [bool, default True] Whether or not the elements should be ranked in ascending order.

pct [bool, default False] Whether or not to display the returned rankings in percentile form.

Returns

same type as caller Return a Series or DataFrame with data ranks as values.

See also:

core.groupby.GroupBy.rank Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```