By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.DataFrame.cumsum

DataFrame.**cumsum**(*self*, *axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [bool, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame

See also:

core.window.Expanding.sum Similar functionality but ignores `NaN` values.

DataFrame.sum Return the sum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
```

(continues on next page)

(continued from previous page)

```
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.DataFrame.describe

DataFrame.**describe**(*self: ~ FrameOrSeries*, *percentiles=None*, *include=None*, *exclude=None*) → ~FrameOrSeries

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

DataFrame.count Count number of non-NA/null observations.

DataFrame.max Maximum of the values in the object.

DataFrame.min Minimum of the values in the object.

DataFrame.mean Mean of the values.

DataFrame.std Standard deviation of the observations.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
```

(continues on next page)

(continued from previous page)

```
... ])
>>> s.describe()
count                        3
unique                       2
top       2010-01-01 00:00:00
freq                         2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

pandas.DataFrame.diff

DataFrame.**diff**(*self*, *periods=1*, *axis=0*) → 'DataFrame'

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

Parameters

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

Returns

DataFrame

See also:

Series.diff First discrete difference for a Series.

DataFrame.pct_change Percent change over given number of periods.

DataFrame.shift Shift index by desired number of periods with an optional time freq.

Notes

For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

pandas.DataFrame.div

DataFrame.**div**(*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']])
```

(continues on next page)

(continued from previous page)

```
>>> 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.divide

DataFrame.**divide**(*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
```

(continues on next page)

(continued from previous page)

```
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.dot

DataFrame.**dot**(*self*, *other*)

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using `self @ other` in Python >= 3.5.

Parameters

other [Series, DataFrame or array-like] The other object to compute the matrix product with.

Returns

Series or DataFrame If other is a Series, return the matrix product between self and other as a Serie. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

See also:

Series.dot Similar method for Series.

Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0    -4
1     5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
    0   1
0   1   4
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other
    0   1
0   1   4
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
    0   1
0   1   4
1   2   2
```

Note how shuffling of the objects does not change the result.

```
>>> s2 = s.reindex([1, 0, 2, 3])
>>> df.dot(s2)
0    -4
1     5
dtype: int64
```

pandas.DataFrame.drop

DataFrame.**drop**(*self*, *labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index [single label or list-like] Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).

New in version 0.21.0.

columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

DataFrame DataFrame without the removed index or column labels.

Raises

KeyError If any of the labels is not found in the selected axis.

See also:

DataFrame.loc Label-location based indexer for selection by label.

DataFrame.dropna Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
```

(continues on next page)

(continued from previous page)

```
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

pandas.DataFrame.drop_duplicates

DataFrame.**drop_duplicates**(*self*, *subset: Union[Hashable, Sequence[Hashable], NoneType] = None*, *keep: Union[str, bool] = 'first'*, *inplace: bool = False*, *ignore_index: bool = False*) → Union[ForwardRef('DataFrame'), NoneType]

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

Parameters

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep [{'first', 'last', False}, default 'first'] Determines which duplicates (if any) to keep. - `first` : Drop duplicates except for the first occurrence. - `last` : Drop duplicates except for the last occurrence. - False : Drop all duplicates.

inplace [bool, default False] Whether to drop duplicates in place or to return a copy.

ignore_index [bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

New in version 1.0.0.

Returns

DataFrame DataFrame with duplicates removed or None if `inplace=True`.

pandas.DataFrame.droplevel

`DataFrame.droplevel(self: ~FrameOrSeries, level, axis=0) → ~FrameOrSeries`

Return DataFrame with requested index / column level(s) removed.

New in version 0.24.0.

Parameters

level [int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis [{0 or 'index', 1 or 'columns'}, default 0]

Returns

DataFrame DataFrame with requested index / column level(s) removed.

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...    ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

pandas.DataFrame.dropna

DataFrame.**dropna**(*self*, *axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the *User Guide* for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Changed in version 1.0.0: Pass tuple or list to drop on multiple axes. Only a single axis is allowed.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

Returns

DataFrame DataFrame with NA entries dropped from it.

See also:

DataFrame.isna Indicate missing values.

DataFrame.notna Indicate existing (non-missing) values.

DataFrame.fillna Replace missing values.

Series.dropna Drop missing values.

Index.dropna Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

pandas.DataFrame.duplicated

DataFrame.**duplicated**(*self*, *subset: Union[Hashable, Sequence[Hashable], NoneType] = None*, *keep: Union[str, bool] = 'first'*) → 'Series'

Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

Parameters

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

keep [{'first', 'last', False}, default 'first'] Determines which duplicates (if any) to mark.

- `first` : Mark duplicates as `True` except for the first occurrence.

- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

Returns

Series

pandas.DataFrame.eq

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

Get Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

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'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

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

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a *Series*, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
```

(continues on next page)

(continued from previous page)

```
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.equals

DataFrame.**equals**(*self*, *other*)

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal. The column headers do not need to have the same type, but the elements within the columns must be the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

testing.assert_series_equal Raises an AssertionError if left and right are not equal. Provides an easy interface to ignore inequality in dtypes, indexes and precision among others.

testing.assert_frame_equal Like assert_series_equal, but targets DataFrames.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Notes

This function requires that the elements have the same dtype as their respective elements in the other Series or DataFrame. However, the column labels do not need to have the same type, as long as they are still considered equal.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

pandas.DataFrame.eval

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

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

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

Returns

ndarray, scalar, or pandas object The result of the evaluation.

See also:

DataFrame.query Evaluates a boolean expression to query the columns of a frame.

DataFrame.assign Can evaluate an expression or function to create new values for a column.

eval Evaluate a Python expression as a string using various backends.