numpy.random.randint

random.randint(*low*, *high=None*, *size=None*, *dtype=int*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

> **Note**
>
> New code should use the `integers` method of a `Generator` instance instead; please see the Quick start.

Parameters:

low : *int or array-like of ints*

Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

high : *int or array-like of ints, optional*

If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

size : *int or tuple of ints, optional*

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

dtype : *dtype, optional*

Desired dtype of the result. Byteorder must be native. The default value is long.

> *New in version 1.11.0.*

Warning

This function defaults to the C-long dtype, which is 32bit on windows and otherwise 64bit on 64bit platforms (and 32bit on 32bit ones). Since NumPy 2.0, NumPy's default integer is 32bit on 32bit platforms and 64bit on 64bit platforms. Which corresponds to *np.intp*. (*dtype=int* is not the same as in most NumPy functions.)

Returns:

out : *int or ndarray of ints*

`size`-shaped array of random integers from the appropriate distribution, or a single such random int if `size` not provided.

See also

`random_integers`
similar to `randint`, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

`random.Generator.integers`
which should be used for new code.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

numpy.dot

numpy.dot(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using `matmul` or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to `multiply` and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

It uses an optimized BLAS library when possible (see `numpy.linalg`).

Parameters:

a : *array_like*

First argument.

b : *array_like*

Second argument.

out : *ndarray, optional*

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns:

output : *ndarray*

Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

Raises:

ValueError

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

See also

`vdot`
Complex-conjugating dot product.

`tensordot`
Sum products over arbitrary axes.

`einsum`
Einstein summation convention.

`matmul`
'@' operator as method with out parameter.

`linalg.multi_dot`
Chained dot product.

Examples

```
>>> import numpy as np
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it is the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

numpy.transpose

numpy.transpose(*a*, *axes=None*) [source]

Returns an array with axes transposed.

For a 1-D array, this returns an unchanged view of the original array, as a transposed vector is simply the same vector. To convert a 1-D array into a 2-D column vector, an additional dimension must be added, e.g., `np.atleast_2d(a).T` achieves this, as does `a[:, np.newaxis]`. For a 2-D array, this is the standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided, then `transpose(a).shape == a.shape[::-1]`.

Parameters:

a : *array_like*

Input array.

axes : *tuple or list of ints, optional*

If specified, it must be a tuple or list which contains a permutation of [0,1,...,N-1] where N is the number of axes of *a*. The *i*'th axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, defaults to `range(a.ndim)[::-1]`, which reverses the order of the axes.

Returns:

p : *ndarray*

a with its axes permuted. A view is returned whenever possible.

See also

`ndarray.transpose`
Equivalent method.

`moveaxis`
Move axes of an array to new positions.

`argsort`
Return the indices that would sort an array.

Notes

Use `transpose(a, argsort(axes))` to invert the transposition of tensors when using the *axes* keyword argument.

Examples

```
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> np.transpose(a)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> np.transpose(a)
array([1, 2, 3, 4])
```

```
>>> a = np.ones((1, 2, 3))
>>> np.transpose(a, (1, 0, 2)).shape
(2, 1, 3)
```

```
>>> a = np.ones((2, 3, 4, 5))
>>> np.transpose(a).shape
(5, 4, 3, 2)
```

pandas.DataFrame.apply

DataFrame.apply(*func*, *axis=0*, *raw=False*, *result_type=None*, *args=()*, *by_row='compat'*, *engine='python'*, *engine_kwargs=None*, ***kwargs*) [source]

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters:

func : *function*

Function to apply to each column or row.

axis : *{0 or 'index', 1 or 'columns'}, default 0*

Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

raw : *bool, default False*

Determines if row or column is passed as a Series or ndarray object:

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

result_type : *{'expand', 'reduce', 'broadcast', None}, default None*

These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function:

Skip to main content

returns a Series these are expanded to columns.

args : *tuple*

Positional arguments to pass to *func* in addition to the array/series.

by_row : *False or "compat", default "compat"*

Only has an effect when `func` is a listlike or dictlike of funcs and the func isn't a string. If "compat", will if possible first translate the func into pandas methods (e.g. `Series().apply(np.sum)` will be translated to `Series().sum()`). If that doesn't work, will try call to apply again with `by_row=True` and if that fails, will call apply again with `by_row=False` (backward compatible). If False, the funcs will be passed the whole Series at once.

> ***Added in version 2.1.0.***

engine : *{'python', 'numba'}, default 'python'*

Choose between the python (default) engine or the numba engine in apply.
The numba engine will attempt to JIT compile the passed function, which may result in speedups for large DataFrames. It also supports the following engine_kwargs :

- nopython (compile the function in nopython mode)
- nogil (release the GIL inside the JIT compiled function)
- parallel (try to apply the function in parallel over the DataFrame)

  Note: Due to limitations within numba/how pandas interfaces with numba, you should only use this if raw=True

Note: The numba compiler only supports a subset of valid Python/numpy operations.
Please read more about the supported python features and supported numpy features in numba to learn what you can or cannot use in the passed function.

> ***Added in version 2.2.0.***

engine_kwargs : *dict*

Pass keyword arguments to the engine. This is currently only used by the numba engine, see the documentation for the engine argument for more information.

****kwargs**

Additional keyword arguments to pass as keywords arguments to *func*

Skip to main content

Returns:

Series or DataFrame

Result of applying `func` along the given axis of the DataFrame.

See also

`DataFrame.map`
For elementwise operations.

`DataFrame.aggregate`
Only perform aggregating type operations.

`DataFrame.transform`
Only perform transforming type operations.

Notes

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

Examples

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

Skip to main content

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing `result_type='expand'` will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

Previous
pandas.DataFrame.combine_first

Next
pandas.DataFrame.map

Skip to main content

pandas.DataFrame.astype

`DataFrame.astype(dtype, copy=None, errors='raise')` [source]

Cast a pandas object to a specified dtype `dtype`.

Parameters:

dtype : *str, data type, Series or Mapping of column name -> data type*

Use a str, numpy.dtype, pandas.ExtensionDtype or Python type to cast entire pandas object to the same type. Alternatively, use a mapping, e.g. {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy : *bool, default True*

Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

> **Note**
>
> The *copy* keyword will change behavior in pandas 3.0. Copy-on-Write will be enabled by default, which means that all methods with a *copy* keyword will use a lazy copy mechanism to defer the copy and ignore the *copy* keyword. The *copy* keyword will be removed in a future version of pandas.
>
> You can already get the future behavior and improvements through enabling copy on write `pd.options.mode.copy_on_write = True`

errors : *{'raise', 'ignore'}, default 'raise'*

Control raising of exceptions on invalid data for provided dtype.

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

Returns:

same type as caller

Skip to main content

See also

`to_datetime`
: Convert argument to datetime.

`to_timedelta`
: Convert argument to timedelta.

`to_numeric`
: Convert argument to a numeric type.

`numpy.ndarray.astype`
: Cast a numpy array to a specified type.

Notes

Changed in version 2.0.0: Using `astype` to convert from timezone-naive dtype to timezone-aware dtype will raise an exception. Use `Series.dt.tz_localize()` instead.

Examples

Create a DataFrame:

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



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

Cast col1 to int32 using a dictionary:

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

Skip to main content

Create a series:

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

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int32): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

Previous
pandas.DataFrame.set_flags

Next
pandas.DataFrame.convert_dtypes

Skip to main content

Created using Sphinx 8.0.2.

pandas.DataFrame

class pandas.DataFrame(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=None*) [source]

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters:

data : *ndarray (structured or homogeneous), Iterable, dict, or DataFrame*

Dict can contain Series, arrays, constants, dataclass or list-like objects. If data is a dict, column order follows insertion-order. If a dict contains Series which have an index defined, it is aligned by its index. This alignment also occurs if data is a Series or a DataFrame itself. Alignment is done on Series/DataFrame inputs.
If data is a list of dicts, column order follows insertion-order.

index : *Index or array-like*

Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided.

columns : *Index or array-like*

Column labels to use for resulting frame when data does not have them, defaulting to RangeIndex(0, 1, 2, …, n). If data contains column labels, will perform column selection instead.

dtype : *dtype, default None*

Data type to force. Only a single dtype is allowed. If None, infer.

copy : *bool or None, default None*

Copy data from inputs. For dict data, the default of None behaves like `copy=True`. For DataFrame or 2d ndarray input, the default of None behaves like `copy=False`. If data is a dict containing one or more Series (possibly of different dtypes), `copy=False` will ensure that these inputs are not copied.

Skip to main content

Changed in version 1.3.0.

See also

`DataFrame.from_records`
: Constructor from tuples, also record arrays.

`DataFrame.from_dict`
: From dicts of Series, arrays, or dicts.

`read_csv`
: Read a comma-separated values (csv) file into DataFrame.

`read_table`
: Read general delimited file into DataFrame.

`read_clipboard`
: Read text from clipboard into DataFrame.

Notes

Please reference the User Guide for more information.

Examples

Constructing DataFrame from a dictionary.

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

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
```

Skip to main content

```
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from a dictionary including Series:

```
>>> d = {'col1': [0, 1, 2, 3], 'col2': pd.Series([2, 3], index=[2, 3])}
>>> pd.DataFrame(data=d, index=[0, 1, 2, 3])
   col1  col2
0     0   NaN
1     1   NaN
2     2   2.0
3     3   3.0
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Constructing DataFrame from a numpy ndarray that has labeled columns:

```
>>> data = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)],
...                 dtype=[("a", "i4"), ("b", "i4"), ("c", "i4")])
>>> df3 = pd.DataFrame(data, columns=['c', 'a'])
...
>>> df3
   c  a
0  3  1
1  6  4
2  9  7
```

Constructing DataFrame from dataclass:

```
>>> from dataclasses import make_dataclass
>>> Point = make_dataclass("Point", [("x", int), ("y", int)])
>>> pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
   x  y
0  0  0
1  0  3
2  2  3
```

Constructing DataFrame from Series/DataFrame:

Skip to main content

```
>>> ser = pd.Series([1, 2, 3], index=["a", "b", "c"])
>>> df = pd.DataFrame(data=ser, index=["a", "c"])
>>> df
   0
a  1
c  3
```

```
>>> df1 = pd.DataFrame([1, 2, 3], index=["a", "b", "c"], columns=["x"])
>>> df2 = pd.DataFrame(data=df1, index=["a", "c"])
>>> df2
   x
a  1
c  3
```

Attributes

`T`	The transpose of the DataFrame.
`at`	Access a single value for a row/column label pair.
`attrs`	Dictionary of global attributes of this dataset.
`axes`	Return a list representing the axes of the DataFrame.
`columns`	The column labels of the DataFrame.
`dtypes`	Return the dtypes in the DataFrame.
`empty`	Indicator whether Series/DataFrame is empty.
`flags`	Get the properties associated with this pandas object.
`iat`	Access a single value for a row/column pair by integer position.
`iloc`	(DEPRECATED) Purely integer-location based indexing for selection by position.
`index`	The index (row labels) of the DataFrame.
`loc`	Access a group of rows and columns by label(s) or a boolean array.
`ndim`	Return an int representing the number of axes / array dimensions.
`shape`	Return a tuple representing the dimensionality of the DataFrame.
`size`	Return an int representing the number of elements in this object.

Skip to main content

`style`	Returns a Styler object.
`values`	Return a Numpy representation of the DataFrame.

Methods

`abs`()	Return a Series/DataFrame with absolute numeric value of each element.
`add`(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *add*).
`add_prefix`(prefix[, axis])	Prefix labels with string *prefix*.
`add_suffix`(suffix[, axis])	Suffix labels with string *suffix*.
`agg`([func, axis])	Aggregate using one or more operations over the specified axis.
`aggregate`([func, axis])	Aggregate using one or more operations over the specified axis.
`align`(other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
`all`([axis, bool_only, skipna])	Return whether all elements are True, potentially over an axis.
`any`(*[, axis, bool_only, skipna])	Return whether any element is True, potentially over an axis.
`apply`(func[, axis, raw, result_type, args, ...])	Apply a function along an axis of the DataFrame.
`applymap`(func[, na_action])	(DEPRECATED) Apply a function to a Dataframe elementwise.
`asfreq`(freq[, method, how, normalize, ...])	Convert time series to specified frequency.
`asof`(where[, subset])	Return the last row(s) without any

Skip to main content

`assign`(**kwargs)	Assign new columns to a DataFrame.
`astype`(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
`at_time`(time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
`backfill`(*[, axis, inplace, limit, downcast])	(DEPRECATED) Fill NA/NaN values by using the next valid observation to fill the gap.
`between_time`(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
`bfill`(*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by using the next valid observation to fill the gap.
`bool`()	(DEPRECATED) Return the bool of a single element Series or DataFrame.
`boxplot`([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame columns.
`clip`([lower, upper, axis, inplace])	Trim values at input threshold(s).
`combine`(other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame.
`combine_first`(other)	Update null elements with value in the same location in *other*.
`compare`(other[, align_axis, keep_shape, ...])	Compare to another DataFrame and show the differences.
`convert_dtypes`([infer_objects, ...])	Convert columns to the best possible dtypes using dtypes supporting `pd.NA`.
`copy`([deep])	Make a copy of this object's indices and data.
`corr`([method, min_periods, numeric_only])	Compute pairwise correlation of

Skip to main content

`corrwith` (other[, axis, drop, method, ...])	Compute pairwise correlation.
`count` ([axis, numeric_only])	Count non-NA cells for each column or row.
`cov` ([min_periods, ddof, numeric_only])	Compute pairwise covariance of columns, excluding NA/null values.
`cummax` ([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
`cummin` ([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
`cumprod` ([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
`cumsum` ([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
`describe` ([percentiles, include, exclude])	Generate descriptive statistics.
`diff` ([periods, axis])	First discrete difference of element.
`div` (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
`divide` (other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
`dot` (other)	Compute the matrix multiplication between the DataFrame and other.
`drop` ([labels, axis, index, columns, level, ...])	Drop specified labels from rows or columns.
`drop_duplicates` ([subset, keep, inplace, ...])	Return DataFrame with duplicate rows removed.
`droplevel` (level[, axis])	Return Series/DataFrame with requested index / column level(s)

Skip to main content

`dropna`(*[, axis, how, thresh, subset, ...])	Remove missing values.
`duplicated`([subset, keep])	Return boolean Series denoting duplicate rows.
`eq`(other[, axis, level])	Get Equal to of dataframe and other, element-wise (binary operator *eq*).
`equals`(other)	Test whether two objects contain the same elements.
`eval`(expr, *[, inplace])	Evaluate a string describing operations on DataFrame columns.
`ewm`([com, span, halflife, alpha, ...])	Provide exponentially weighted (EW) calculations.
`expanding`([min_periods, axis, method])	Provide expanding window calculations.
`explode`(column[, ignore_index])	Transform each element of a list-like to a row, replicating index values.
`ffill`(*[, axis, inplace, limit, limit_area, ...])	Fill NA/NaN values by propagating the last valid observation to next valid.
`fillna`([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
`filter`([items, like, regex, axis])	Subset the dataframe rows or columns according to the specified index labels.
`first`(offset)	(DEPRECATED) Select initial periods of time series data based on a date offset.
`first_valid_index`()	Return index for first non-NA value or None, if no non-NA value is found.

Skip to main content

`floordiv`(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).
`from_dict`(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
`from_records`(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
`ge`(other[, axis, level])	Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
`get`(key[, default])	Get item from object for given key (ex: DataFrame column).
`groupby`([by, axis, level, as_index, sort, ...])	Group DataFrame using a mapper or by a Series of columns.
`gt`(other[, axis, level])	Get Greater than of dataframe and other, element-wise (binary operator *gt*).
`head`([n])	Return the first *n* rows.
`hist`([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's columns.
`idxmax`([axis, skipna, numeric_only])	Return index of first occurrence of maximum over requested axis.
`idxmin`([axis, skipna, numeric_only])	Return index of first occurrence of minimum over requested axis.
`infer_objects`([copy])	Attempt to infer better dtypes for object columns.
`info`([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
`insert`(loc, column, value[, allow_duplicates])	Insert column into DataFrame at

Skip to main content

`interpolate`([method, axis, limit, inplace, ...])	Fill NaN values using an interpolation method.
`isetitem`(loc, value)	Set the given value in the column with position *loc*.
`isin`(values)	Whether each element in the DataFrame is contained in values.
`isna`()	Detect missing values.
`isnull`()	DataFrame.isnull is an alias for DataFrame.isna.
`items`()	Iterate over (column name, Series) pairs.
`iterrows`()	Iterate over DataFrame rows as (index, Series) pairs.
`itertuples`([index, name])	Iterate over DataFrame rows as namedtuples.
`join`(other[, on, how, lsuffix, rsuffix, ...])	Join columns of another DataFrame.
`keys`()	Get the 'info axis' (see Indexing for more).
`kurt`([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
`kurtosis`([axis, skipna, numeric_only])	Return unbiased kurtosis over requested axis.
`last`(offset)	(DEPRECATED) Select final periods of time series data based on a date offset.
`last_valid_index`()	Return index for last non-NA value or None, if no non-NA value is found.
`le`(other[, axis, level])	Get Less than or equal to of dataframe and other, element-wise

Skip to main content

`lt`(other[, axis, level])	Get Less than of dataframe and other, element-wise (binary operator *lt*).
`map`(func[, na_action])	Apply a function to a Dataframe elementwise.
`mask`(cond[, other, inplace, axis, level])	Replace values where the condition is True.
`max`([axis, skipna, numeric_only])	Return the maximum of the values over the requested axis.
`mean`([axis, skipna, numeric_only])	Return the mean of the values over the requested axis.
`median`([axis, skipna, numeric_only])	Return the median of the values over the requested axis.
`melt`([id_vars, value_vars, var_name, ...])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
`memory_usage`([index, deep])	Return the memory usage of each column in bytes.
`merge`(right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
`min`([axis, skipna, numeric_only])	Return the minimum of the values over the requested axis.
`mod`(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *mod*).
`mode`([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
`mul`(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Skip to main content

	mul).
`ne`(other[, axis, level])	Get Not equal to of dataframe and other, element-wise (binary operator *ne*).
`nlargest`(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
`notna`()	Detect existing (non-missing) values.
`notnull`()	DataFrame.notnull is an alias for DataFrame.notna.
`nsmallest`(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
`nunique`([axis, dropna])	Count number of distinct elements in specified axis.
`pad`(*[, axis, inplace, limit, downcast])	(DEPRECATED) Fill NA/NaN values by propagating the last valid observation to next valid.
`pct_change`([periods, fill_method, limit, freq])	Fractional change between the current and a prior element.
`pipe`(func, *args, **kwargs)	Apply chainable functions that expect Series or DataFrames.
`pivot`(*, columns[, index, values])	Return reshaped DataFrame organized by given index / column values.
`pivot_table`([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
`pop`(item)	Return item and drop from frame.
`pow`(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

Skip to main content

`prod`([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
`product`([axis, skipna, numeric_only, min_count])	Return the product of the values over the requested axis.
`quantile`([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
`query`(expr, *[, inplace])	Query the columns of a DataFrame with a boolean expression.
`radd`(other[, axis, level, fill_value])	Get Addition of dataframe and other, element-wise (binary operator *radd*).
`rank`([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
`rdiv`(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
`reindex`([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic.
`reindex_like`(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
`rename`([mapper, index, columns, axis, copy, ...])	Rename columns or index labels.
`rename_axis`([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
`reorder_levels`(order[, axis])	Rearrange index levels using input order.
`replace`([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
`resample`(rule[, axis, closed, label, ...])	Resample time-series data.
`reset_index`([level, drop, inplace, ...])	Reset the index, or a level of it.

Skip to main content

`rfloordiv`(other[, axis, level, fill_value])	Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
`rmod`(other[, axis, level, fill_value])	Get Modulo of dataframe and other, element-wise (binary operator *rmod*).
`rmul`(other[, axis, level, fill_value])	Get Multiplication of dataframe and other, element-wise (binary operator *rmul*).
`rolling`(window[, min_periods, center, ...])	Provide rolling window calculations.
`round`([decimals])	Round a DataFrame to a variable number of decimal places.
`rpow`(other[, axis, level, fill_value])	Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).
`rsub`(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).
`rtruediv`(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
`sample`([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
`select_dtypes`([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
`sem`([axis, skipna, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
`set_axis`(labels, *[, axis, copy])	Assign desired index to given axis.
`set_flags`(*[, copy, allows_duplicate_labels])	Return a new object with updated flags.

Skip to main content

`set_index`(keys, *[, drop, append, inplace, ...])	Set the DataFrame index using existing columns.
`shift`([periods, freq, axis, fill_value, suffix])	Shift index by desired number of periods with an optional time *freq*.
`skew`([axis, skipna, numeric_only])	Return unbiased skew over requested axis.
`sort_index`(*[, axis, level, ascending, ...])	Sort object by labels (along an axis).
`sort_values`(by, *[, axis, ascending, ...])	Sort by the values along either axis.
`squeeze`([axis])	Squeeze 1 dimensional axis objects into scalars.
`stack`([level, dropna, sort, future_stack])	Stack the prescribed level(s) from columns to index.
`std`([axis, skipna, ddof, numeric_only])	Return sample standard deviation over requested axis.
`sub`(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
`subtract`(other[, axis, level, fill_value])	Get Subtraction of dataframe and other, element-wise (binary operator *sub*).
`sum`([axis, skipna, numeric_only, min_count])	Return the sum of the values over the requested axis.
`swapaxes`(axis1, axis2[, copy])	(DEPRECATED) Interchange axes and swap values axes appropriately.
`swaplevel`([i, j, axis])	Swap levels i and j in a `MultiIndex`.
`tail`([n])	Return the last *n* rows.
`take`(indices[, axis])	Return the elements in the given *positional* indices along an axis.

Skip to main content

`to_csv`([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
`to_dict`([orient, into, index])	Convert the DataFrame to a dictionary.
`to_excel`(excel_writer, *[, sheet_name, ...])	Write object to an Excel sheet.
`to_feather`(path, **kwargs)	Write a DataFrame to the binary Feather format.
`to_gbq`(destination_table, *[, project_id, ...])	(DEPRECATED) Write a DataFrame to a Google BigQuery table.
`to_hdf`(path_or_buf, *, key[, mode, ...])	Write the contained data to an HDF5 file using HDFStore.
`to_html`([buf, columns, col_space, header, ...])	Render a DataFrame as an HTML table.
`to_json`([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
`to_latex`([buf, columns, header, index, ...])	Render object to a LaTeX tabular, longtable, or nested table.
`to_markdown`([buf, mode, index, storage_options])	Print DataFrame in Markdown-friendly format.
`to_numpy`([dtype, copy, na_value])	Convert the DataFrame to a NumPy array.
`to_orc`([path, engine, index, engine_kwargs])	Write a DataFrame to the ORC format.
`to_parquet`([path, engine, compression, ...])	Write a DataFrame to the binary parquet format.
`to_period`([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex.
`to_pickle`(path, *[, compression, protocol, ...])	Pickle (serialize) object to file.
`to_records`([index, column_dtypes, index_dtypes])	Convert DataFrame to a NumPy

Skip to main content

`to_sql`(name, con, *[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
`to_stata`(path, *[, convert_dates, ...])	Export DataFrame object to Stata dta format.
`to_string`([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
`to_timestamp`([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
`to_xarray`()	Return an xarray object from the pandas object.
`to_xml`([path_or_buffer, index, root_name, ...])	Render a DataFrame to an XML document.
`transform`(func[, axis])	Call `func` on self producing a DataFrame with the same axis shape as self.
`transpose`(*args[, copy])	Transpose index and columns.
`truediv`(other[, axis, level, fill_value])	Get Floating division of dataframe and other, element-wise (binary operator *truediv*).
`truncate`([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
`tz_convert`(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
`tz_localize`(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
`unstack`([level, fill_value, sort])	Pivot a level of the (necessarily hierarchical) index labels.
`update`(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.

Skip to main content

`value_counts`([subset, normalize, sort, ...])	Return a Series containing the

pandas.DataFrame.groupby

DataFrame.groupby(*by=None, axis=<no_default>, level=None, as_index=True, sort=True, group_keys=True, observed=<no_default>, dropna=True*) [source]

Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters:

by : *mapping, function, label, pd.Grouper or list of such*

Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If a list or ndarray of length equal to the selected axis is passed (see the groupby user guide), the values are used as-is to determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis : *{0 or 'index', 1 or 'columns'}, default 0*

Split along rows (0) or columns (1). For *Series* this parameter is unused and defaults to 0.

> ***Deprecated since version 2.1.0:*** Will be removed and behave like axis=0 in a future version. For `axis=1`, do `frame.T.groupby(...)` instead.

level : *int, level name, or sequence of such, default None*

If the axis is a MultiIndex (hierarchical), group by a particular level or levels. Do not specify both `by` and `level`.

as_index : *bool, default True*

Return object with group labels as the index. Only relevant for DataFrame input.

Skip to main content

effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

sort : *bool, default True*

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group. If False, the groups will appear in the same order as they did in the original DataFrame. This argument has no effect on filtrations (see the filtrations in the user guide), such as `head()`, `tail()`, `nth()` and in transformations (see the transformations in the user guide).

> ***Changed in version 2.0.0:*** Specifying `sort=False` with an ordered categorical grouper will no longer sort the values.

group_keys : *bool, default True*

When calling apply and the `by` argument produces a like-indexed (i.e. a transform) result, add group keys to index to identify pieces. By default group keys are not included when the result's index (and column) labels match the inputs, and are included otherwise.

> ***Changed in version 1.5.0:*** Warns that `group_keys` will no longer be ignored when the result from `apply` is a like-indexed Series or DataFrame. Specify `group_keys` explicitly to include the group keys or not.

> ***Changed in version 2.0.0:*** `group_keys` now defaults to `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.

> ***Deprecated since version 2.1.0:*** The default value will change to True in a future version of pandas.

dropna : *bool, default True*

If True, and if group keys contain NA values, NA values together with row/column

Skip to main content

Returns:

pandas.api.typing.DataFrameGroupBy

Returns a groupby object that contains information about the groups.

See also

`resample`

Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more detailed usage and examples, including splitting an object into groups, iterating through groups, selecting a group, aggregation, and more.

Examples

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
...                   index=index)
>>> df
                Max Speed
Animal Type
Falcon Captive      390.0
       Wild         350.0
```

Skip to main content

```
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level="Type").mean()
         Max Speed
Type
Captive      210.0
Wild         185.0
```

We can also choose to include NA in group keys or not by setting *dropna* parameter, the default setting is *True*.

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by=["b"]).sum()
    a   c
b
1.0 2   3
2.0 2   5
```

```
>>> df.groupby(by=["b"], dropna=False).sum()
    a   c
b
1.0 2   3
2.0 2   5
NaN 1   4
```

```
>>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by="a").sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
```

```
>>> df.groupby(by="a", dropna=False).sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
NaN 12.3   33.0
```

Skip to main content

When using `.apply()`, use `group_keys` to include or exclude the group keys. The `group_keys` argument defaults to `True` (include).

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df.groupby("Animal", group_keys=True)[['Max Speed']].apply(lambda x: x)
          Max Speed
Animal
Falcon 0      380.0
       1      370.0
Parrot 2       24.0
       3       26.0
```

```
>>> df.groupby("Animal", group_keys=False)[['Max Speed']].apply(lambda x: x)
   Max Speed
0      380.0
1      370.0
2       24.0
3       26.0
```

Previous
pandas.DataFrame.transform

Next
pandas.DataFrame.rolling

pandas.DataFrame.iloc

property DataFrame.**iloc** [source]

Purely integer-location based indexing for selection by position.

> ***Deprecated since version 2.2.0:*** Returning a tuple from a callable is deprecated.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.
- A tuple of row and column indexes. The tuple elements consist of one of the above inputs, e.g. `(0, 1)`.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position.

Skip to main content

See also

`DataFrame.iat`
: Fast integer location scalar accessor.

`DataFrame.loc`
: Purely label-location based indexer for selection by label.

`Series.iloc`
: Purely integer-location based indexing for selection by position.

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000}]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
```

Skip to main content

```
0    1    2    3    4
1  100  200  300  400
```

With a *slice* object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects

Skip to main content

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
```

pandas.DataFrame.mean

`DataFrame.mean(axis=0, skipna=True, numeric_only=False, **kwargs)` [source]

Return the mean of the values over the requested axis.

Parameters:

axis : *{index (0), columns (1)}*

Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

For DataFrames, specifying `axis=None` will apply the aggregation across both axes.

> *Added in version 2.0.0.*

skipna : *bool, default True*

Exclude NA/null values when computing the result.

numeric_only : *bool, default False*

Include only float, int, boolean columns. Not implemented for Series.

****kwargs**

Additional keyword arguments to be passed to the function.

Returns:

Series or scalar

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.mean()
2.0
```

With a DataFrame

Skip to main content

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [2, 3]}, index=['tiger', 'zebra'])
>>> df
       a   b
tiger  1   2
zebra  2   3
>>> df.mean()
a   1.5
b   2.5
dtype: float64
```

Using axis=1

```
>>> df.mean(axis=1)
tiger   1.5
zebra   2.5
dtype: float64
```

In this case, *numeric_only* should be set to *True* to avoid getting an error.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': ['T', 'Z']},
...                   index=['tiger', 'zebra'])
>>> df.mean(numeric_only=True)
a   1.5
dtype: float64
```

Previous
pandas.DataFrame.max

Next
pandas.DataFrame.median

pandas.read_csv

pandas.read_csv(*filepath_or_buffer, *, sep=<no_default>, delimiter=None, header='infer', names=<no_default>, index_col=None, usecols=None, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=<no_default>, skip_blank_lines=True, parse_dates=None, infer_datetime_format=<no_default>, keep_date_col=<no_default>, date_parser=<no_default>, date_format=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, on_bad_lines='error', delim_whitespace=<no_default>, low_memory=True, memory_map=False, float_precision=None, storage_options=None, dtype_backend=<no_default>*) #

Read a comma-separated values (csv) file into DataFrame. [source]

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters:

filepath_or_buffer : *str, path object or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.

sep : *str, default ','*

Character or regex pattern to treat as the delimiter. If `sep=None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator from only

Skip to main content

addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

delimiter : *str, optional*

Alias for `sep`.

header : *int, Sequence of int, 'infer' or None, default 'infer'*

Row number(s) containing column labels and marking the start of the data (zero-indexed). Default behavior is to infer the column names: if no `names` are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly to `names` then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a `MultiIndex` on the columns e.g. `[0, 1, 3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

names : *Sequence of Hashable, optional*

Sequence of column labels to apply. If the file contains a header row, then you should explicitly pass `header=0` to override the column names. Duplicates in this list are not allowed.

index_col : *Hashable, Sequence of Hashable or False, optional*

Column(s) to use as row label(s), denoted either by column labels or column indices. If a sequence of labels or indices is given, `MultiIndex` will be formed for the row labels.

Note: `index_col=False` can be used to force pandas to *not* use the first column as the index, e.g., when you have a malformed file with delimiters at the end of each line.

usecols : *Sequence of Hashable or Callable, optional*

Subset of columns to select, denoted either by column labels or column indices. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). If `names` are given, the document header row(s) are not taken into account. For example, a valid list-like `usecols` parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To

Skip to main content

`pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

dtype : *dtype or dict of {Hashable : dtype}, optional*

Data type(s) to apply to either the whole dataset or individual columns. E.g., `{'a': np.float64, 'b': np.int32, 'c': 'Int64'}` Use `str` or `object` together with suitable `na_values` settings to preserve and not interpret `dtype`. If `converters` are specified, they will be applied INSTEAD of `dtype` conversion.

> ***Added in version 1.5.0:*** Support for `defaultdict` was added. Specify a `defaultdict` as input where the default determines the `dtype` of the columns which are not explicitly listed.

engine : *{'c', 'python', 'pyarrow'}, optional*

Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

> ***Added in version 1.4.0:*** The 'pyarrow' engine was added as an *experimental* engine, and some features are unsupported, or may not work correctly, with this engine.

converters : *dict of {Hashable : Callable}, optional*

Functions for converting values in specified columns. Keys can either be column labels or column indices.

true_values : *list, optional*

Values to consider as `True` in addition to case-insensitive variants of 'True'.

false_values : *list, optional*

Values to consider as `False` in addition to case-insensitive variants of 'False'.

skipinitialspace : *bool, default False*

Skip to main content

skiprows : *int, list of int or Callable, optional*

Line numbers to skip (0-indexed) or number of lines to skip (`int`) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning `True` if the row should be skipped and `False` otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

skipfooter : *int, default 0*

Number of lines at bottom of file to skip (Unsupported with `engine='c'`).

nrows : *int, optional*

Number of rows of file to read. Useful for reading pieces of large files.

na_values : *Hashable, Iterable of Hashable or dict of {Hashable : Iterable}, optional*

Additional strings to recognize as `NA`/`NaN`. If `dict` passed, specific per-column `NA` values. By default the following values are interpreted as `NaN`: " ", "#N/A", "#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN", "-nan", "1.#IND", "1.#QNAN", "<NA>", "N/A", "NA", "NULL", "NaN", "None", "n/a", "nan", "null ".

keep_default_na : *bool, default True*

Whether or not to include the default `NaN` values when parsing the data. Depending on whether `na_values` is passed in, the behavior is as follows:

- If `keep_default_na` is `True`, and `na_values` are specified, `na_values` is appended to the default `NaN` values used for parsing.
- If `keep_default_na` is `True`, and `na_values` are not specified, only the default `NaN` values are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are specified, only the `NaN` values specified `na_values` are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are not specified, no strings will be parsed as `NaN`.

Note that if `na_filter` is passed in as `False`, the `keep_default_na` and `na_values` parameters will be ignored.

na_filter : *bool, default True*

Detect missing value markers (empty strings and the value of `na_values`). In data without any `NA` values, passing `na_filter=False` can improve the performance of reading a large file.

verbose : *bool, default False*

Indicate number of `NA` values placed in non-numeric columns

Skip to main content

> **Deprecated since version 2.2.0.**

skip_blank_lines : *bool, default True*

If `True`, skip over blank lines rather than interpreting as `NaN` values.

parse_dates : *bool, list of Hashable, list of lists or dict of {Hashable : list}, default False*

The behavior is as follows:

- `bool`. If `True` -> try parsing the index. Note: Automatically set to `True` if `date_format` or `date_parser` arguments have been passed.
- `list` of `int` or names. e.g. If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- `list` of `list`. e.g. If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column. Values are joined with a space before parsing.
- `dict`, e.g. `{'foo' : [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. Values are joined with a space before parsing.

If a column or index cannot be represented as an array of `datetime`, say because of an unparsable value or a mixture of timezones, the column or index will be returned unaltered as an `object` data type. For non-standard `datetime` parsing, use `to_datetime()` after `read_csv()`.

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format : *bool, default False*

If `True` and `parse_dates` is enabled, pandas will attempt to infer the format of the `datetime` strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

> ***Deprecated since version 2.0.0:*** A strict version of this argument is now the default, passing it has no effect.

keep_date_col : *bool, default False*

If `True` and `parse_dates` specifies combining multiple columns then keep the original columns.

date_parser : *Callable, optional*

Function to use for converting a sequence of string columns to an array of `datetime` instances. The default uses `dateutil.parser.parser` to do the

Skip to main content

to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

> **Deprecated since version 2.0.0:** Use `date_format` instead, or read in as `object` and then apply `to_datetime()` as-needed.

date_format : *str or dict of column -> format, optional*

Format to use for parsing dates when used in conjunction with `parse_dates`. The strftime to parse time, e.g. `"%d/%m/%Y"`. See strftime documentation for more information on choices, though note that `"%f"` will parse all the way up to nanoseconds. You can also pass:

- **"ISO8601", to parse any ISO8601**

  time string (not necessarily in exactly the same format);

- **"mixed", to infer the format for each element individually. This is risky,**

  and you should probably use it along with *dayfirst*.

> *Added in version 2.0.0.*

dayfirst : *bool, default False*

DD/MM format dates, international and European format.

cache_dates : *bool, default True*

If `True`, use a cache of unique, converted dates to apply the `datetime` conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

iterator : *bool, default False*

Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

chunksize : *int, optional*

Number of lines to read from the file per chunk. Passing a value will cause the function to return a `TextFileReader` object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression : *str or dict, default 'infer'*

Skip to main content

'.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). If using 'zip' or 'tar', the ZIP file must contain only one data file to be read in. Set to `None` for no decompression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdDecompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: `compression={'method': 'zstd', 'dict_data': my_compression_dict}`.

> ***Added in version 1.5.0:*** Added support for *.tar* files.

> ***Changed in version 1.4.0:*** Zstandard support.

thousands : *str (length 1), optional*

Character acting as the thousands separator in numerical values.

decimal : *str (length 1), default '.'*

Character to recognize as decimal point (e.g., use ',' for European data).

lineterminator : *str (length 1), optional*

Character used to denote a line break. Only valid with C parser.

quotechar : *str (length 1), optional*

Character used to denote the start and end of a quoted item. Quoted items can include the `delimiter` and it will be ignored.

quoting : *{0 or csv.QUOTE_MINIMAL, 1 or csv.QUOTE_ALL, 2 or csv.QUOTE_NONNUMERIC, 3 or csv.QUOTE_NONE}, default csv.QUOTE_MINIMAL*

Control field quoting behavior per `csv.QUOTE_*` constants. Default is `csv.QUOTE_MINIMAL` (i.e., 0) which implies that only fields containing special characters are quoted (e.g., characters defined in `quotechar`, `delimiter`, or `lineterminator`.

doublequote : *bool, default True*

When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements INSIDE a field as a single `quotechar` element.

escapechar : *str (length 1), optional*

Skip to main content

comment : *str (length 1), optional*

Character indicating that the remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in `'a,b,c'` being treated as the header.

encoding : *str, optional, default 'utf-8'*

Encoding to use for UTF when reading/writing (ex. `'utf-8'`). List of Python standard encodings .

encoding_errors : *str, optional, default 'strict'*

How encoding errors are treated. List of possible values .

> *Added in version 1.3.0.*

dialect : *str or csv.Dialect, optional*

If provided, this parameter will override values (default or not) for the following parameters: `delimiter`, `doublequote`, `escapechar`, `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

on_bad_lines : *{'error', 'warn', 'skip'} or Callable, default 'error'*

Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are :

- `'error'`, raise an Exception when a bad line is encountered.
- `'warn'`, raise a warning when a bad line is encountered and skip that line.
- `'skip'`, skip bad lines without raising or warning when they are encountered.

> *Added in version 1.3.0.*

Skip to main content

Added in version 1.4.0:

- Callable, function with signature `(bad_line: list[str]) -> list[str] | None` that will process a single bad line. `bad_line` is a list of strings split by the `sep`. If the function returns `None`, the bad line will be ignored. If the function returns a new `list` of strings with more elements than expected, a `ParserWarning` will be emitted while dropping extra elements. Only supported when `engine='python'`

Changed in version 2.2.0:

- Callable, function with signature as described in pyarrow documentation when `engine='pyarrow'`

delim_whitespace : *bool, default False*

Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the `sep` delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

Deprecated since version 2.2.0: Use `sep="\s+"` instead.

low_memory : *bool, default True*

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set `False`, or specify the type with the `dtype` parameter. Note that the entire file is read into a single `DataFrame` regardless, use the `chunksize` or `iterator` parameter to return the data in chunks. (Only valid with C parser).

memory_map : *bool, default False*

If a filepath is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

float_precision : *{'high', 'legacy', 'round_trip'}, optional*

Specifies which converter the C engine should use for floating-point values. The options are `None` or `'high'` for the ordinary converter, `'legacy'` for the original lower precision pandas converter, and `'round_trip'` for the round-trip converter.

storage_options : *dict, optional*

Extra options that make sense for a particular storage connection, e.g. host, port,

Skip to main content

`urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

dtype_backend : *{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'*

Back-end data type applied to the resultant `DataFrame` (still experimental). Behaviour is as follows:

- `"numpy_nullable"`: returns nullable-dtype-backed `DataFrame` (default).
- `"pyarrow"`: returns pyarrow-backed nullable `ArrowDtype` DataFrame.

Added in version 2.0.

Returns:

DataFrame or TextFileReader

A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also

`DataFrame.to_csv`
Write DataFrame to a comma-separated values (csv) file.

`read_table`
Read general delimited file into DataFrame.

`read_fwf`
Read a table of fixed-width formatted lines into DataFrame.

Examples

pandas.DataFrame.shape

property DataFrame.**shape** [source]

Return a tuple representing the dimensionality of the DataFrame.

See also

`ndarray.shape`
Tuple of array dimensions.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

Previous
pandas.DataFrame.size

Next
pandas.DataFrame.memory_usage

pandas.DataFrame.size

property DataFrame.size [source]

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also

`ndarray.size`
Number of elements in the array.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

Previous
pandas.DataFrame.ndim

Next
pandas.DataFrame.shape

pandas.DataFrame.sort_values

DataFrame.sort_values(*by*, ***, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *ignore_index=False*, *key=None*) [source]

Sort by the values along either axis.

Parameters:

by : *str or list of str*

Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

axis : *"{0 or 'index', 1 or 'columns'}", default 0*

Axis to be sorted.

ascending : *bool or list of bool, default True*

Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace : *bool, default False*

If True, perform operation in-place.

kind : *{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'*

Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position : *{'first', 'last'}, default 'last'*

Puts NaNs at the beginning if *first*; *last* puts NaNs at the end.

ignore_index : *bool, default False*

If True, the resulting axis will be labeled 0, 1, …, n - 1.

key : *callable, optional*

Apply the key function to the values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this

Skip to main content

with the same shape as the input. It will be applied to each column in *by* independently.

Returns:

DataFrame or None

DataFrame with sorted values or None if `inplace=True`.

See also

`DataFrame.sort_index`
Sort a DataFrame by the index.

`Series.sort_values`
Similar method for a Series.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort by multiple columns

Skip to main content

```
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
```

Sorting with a key function

```
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Natural sort with the key argument, using the *natsort <https://github.com/SethMMorton/natsort>* package.

```
>>> df = pd.DataFrame({
...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...    "value": [10, 20, 30, 40, 50]
... })
```

Skip to main content

```
0    0hr     10
1  128hr     20
2   72hr     30
3   48hr     40
4   96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...     by="time",
...     key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

Previous
pandas.DataFrame.reorder_levels

Next
pandas.DataFrame.sort_index

pandas.Series.str.contains

`Series.str.contains(pat, case=True, flags=0, na=None, regex=True)` [source]

Test if pattern or regex is contained within a string of a Series or Index.

Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

Parameters:

pat : *str*

Character sequence or regular expression.

case : *bool, default True*

If True, case sensitive.

flags : *int, default 0 (no flags)*

Flags to pass through to the re module, e.g. re.IGNORECASE.

na : *scalar, optional*

Fill value for missing values. The default depends on dtype of the array. For object-dtype, `numpy.nan` is used. For `StringDtype`, `pandas.NA` is used.

regex : *bool, default True*

If True, assumes the pat is a regular expression.

If False, treats the pat as a literal string.

Returns:

Series or Index of boolean values

A Series or Index of boolean values indicating whether the given pattern is contained within the string of each element of the Series or Index.

Skip to main content

See also

`match`
: Analogous, but stricter, relying on re.match instead of re.search.

`Series.str.startswith`
: Test if the start of each string element matches a pattern.

`Series.str.endswith`
: Same as startswith, but tests the end of string.

Examples

Returning a Series of booleans using only a literal pattern.

```
>>> s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.nan])
>>> s1.str.contains('og', regex=False)
0    False
1     True
2    False
3    False
4      NaN
dtype: object
```

Returning an Index of booleans using only a literal pattern.

```
>>> ind = pd.Index(['Mouse', 'dog', 'house and parrot', '23.0', np.nan])
>>> ind.str.contains('23', regex=False)
Index([False, False, False, True, nan], dtype='object')
```

Specifying case sensitivity using *case*.

```
>>> s1.str.contains('oG', case=True, regex=True)
0    False
1    False
2    False
3    False
4      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN* replaces NaN values with *False*. If Series or Index does not contain NaN values the resultant dtype will be *bool*, otherwise, an *object* dtype.

```
>>> s1.str.contains('og', na=False, regex=True)
```

Skip to main content

```
2    False
3    False
4    False
dtype: bool
```

Returning ‘house’ or ‘dog’ when either expression occurs in a string.

```
>>> s1.str.contains('house|dog', regex=True)
0    False
1     True
2     True
3    False
4      NaN
dtype: object
```

Ignoring case sensitivity using *flags* with regex.

```
>>> import re
>>> s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0    False
1    False
2     True
3    False
4      NaN
dtype: object
```

Returning any digit using regular expression.

```
>>> s1.str.contains('\\d', regex=True)
0    False
1    False
2    False
3     True
4      NaN
dtype: object
```

Ensure *pat* is a not a literal pattern when *regex* is set to True. Note in the following example one might expect only *s2[1]* and *s2[3]* to return *True*. However, ‘.0’ as a regex matches any character followed by a 0.

```
>>> s2 = pd.Series(['40', '40.0', '41', '41.0', '35'])
>>> s2.str.contains('.0', regex=True)
0     True
1     True
2    False
3     True
4    False
dtype: bool
```

Skip to main content

pandas.Series.str.replace

`Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=False)` [source]

Replace each occurrence of pattern/regex in the Series/Index.

Equivalent to `str.replace()` or `re.sub()`, depending on the regex value.

Parameters:

pat : *str or compiled regex*

String can be a character sequence or regular expression.

repl : *str or callable*

Replacement string or a callable. The callable is passed the regex match object and must return a replacement string to be used. See `re.sub()`.

n : *int, default -1 (all)*

Number of replacements to make from start.

case : *bool, default None*

Determines if replace is case sensitive:

- If True, case sensitive (the default if *pat* is a string)
- Set to False for case insensitive
- Cannot be set if *pat* is a compiled regex.

flags : *int, default 0 (no flags)*

Regex module flags, e.g. re.IGNORECASE. Cannot be set if *pat* is a compiled regex.

regex : *bool, default False*

Determines if the passed-in pattern is a regular expression:

- If True, assumes the passed-in pattern is a regular expression.
- If False, treats the pattern as a literal string
- Cannot be set to False if *pat* is a compiled regex or *repl* is a callable.

Returns:

Skip to main content

A copy of the object with all matching occurrences of *pat* replaced by *repl*.

Raises:

ValueError

- if *regex* is False and *repl* is a callable or *pat* is a compiled regex
- if *pat* is a compiled regex and *case* or *flags* is set

Notes

When *pat* is a compiled regex, all flags should be included in the compiled regex. Use of *case*, *flags*, or *regex=False* with a compiled regex will raise an error.

Examples

When *pat* is a string and *regex* is True, the given *pat* is compiled as a regex. When *repl* is a string, it replaces matching regex patterns as with `re.sub()`. NaN value(s) in the Series are left as is:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f.', 'ba', regex=True)
0    bao
1    baz
2    NaN
dtype: object
```

When *pat* is a string and *regex* is False, every *pat* is replaced with *repl* as with `str.replace()`:

```
>>> pd.Series(['f.o', 'fuz', np.nan]).str.replace('f.', 'ba', regex=False)
0    bao
1    fuz
2    NaN
dtype: object
```

When *repl* is a callable, it is called on every *pat* using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

To get the idea:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f', repr, regex=True)
0    <re.Match object; span=(0, 1), match='f'>oo
1    <re.Match object; span=(0, 1), match='f'>uz
2                                            NaN
```

Skip to main content

Reverse every lowercase alphabetic word:

```
>>> repl = lambda m: m.group(0)[::-1]
>>> ser = pd.Series(['foo 123', 'bar baz', np.nan])
>>> ser.str.replace(r'[a-z]+', repl, regex=True)
0    oof 123
1    rab zab
2        NaN
dtype: object
```

Using regex groups (extract second group and swap case):

```
>>> pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
>>> repl = lambda m: m.group('two').swapcase()
>>> ser = pd.Series(['One Two Three', 'Foo Bar Baz'])
>>> ser.str.replace(pat, repl, regex=True)
0    tWO
1    bAR
dtype: object
```

Using a compiled regex with flags

```
>>> import re
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar', regex=True)
0    foo
1    bar
2    NaN
dtype: object
```

Previous
pandas.Series.str.repeat

Next
pandas.Series.str.rfind

pandas.DataFrame.sum

```
DataFrame.sum(axis=0, skipna=True, numeric_only=False, min_count=0, **kwargs)
```

[source]

Return the sum of the values over the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters:

axis : *{index (0), columns (1)}*

Axis for the function to be applied on. For *Series* this parameter is unused and defaults to 0.

> **Warning**
>
> The behavior of DataFrame.sum with `axis=None` is deprecated, in a future version this will reduce over both axes and return a scalar To retain the old behavior, pass axis=0 (or do not pass axis).

> ***Added in version 2.0.0.***

skipna : *bool, default True*

Exclude NA/null values when computing the result.

numeric_only : *bool, default False*

Include only float, int, boolean columns. 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.

****kwargs**

Additional keyword arguments to be passed to the function.

Skip to main content

Series or scalar

See also

`Series.sum`
: Return the sum.

`Series.min`
: Return the minimum.

`Series.max`
: Return the maximum.

`Series.idxmin`
: Return the index of the minimum.

`Series.idxmax`
: Return the index of the maximum.

`DataFrame.sum`
: Return the sum over the requested axis.

`DataFrame.min`
: Return the minimum over the requested axis.

`DataFrame.max`
: Return the maximum over the requested axis.

`DataFrame.idxmin`
: Return the index of the minimum over the requested axis.

`DataFrame.idxmax`
: Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
```

Skip to main content

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([], dtype="float64").sum(min_count=1)
nan
```

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

```
>>> pd.Series([np.nan]).sum()
0.0
```

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

Previous
pandas.DataFrame.skew

Next
pandas.DataFrame.std

pandas.DataFrame.query

DataFrame.query(*expr*, *, *inplace=False*, ***kwargs*) [source]

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 are not valid Python variable names by surrounding them in backticks. Thus, column names containing spaces or punctuations (besides underscores) or starting with digits must be surrounded by backticks. (For example, a column named "Area (cm^2)" would be referenced as `` `Area (cm^2)` ``). Column names which are Python keywords (like "list", "for", "import", etc) cannot be used.

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 ``.

inplace : *bool*

Whether to modify the DataFrame rather than creating a new one.

****kwargs**

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

Returns:

DataFrame or None

DataFrame resulting from the provided query expression or None if `inplace=True`.

Skip to main content

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

Skip to main content

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']]
```

Skip to main content

```
   0  1  10  10
```