'tqdm_gui' Use the `tqdm.tqdm_gui()` function to display a progress bar as a graphical dialog box.

Note that his feature requires version 0.12.0 or later of the `pandas-gbq` package. And it requires the `tqdm` package. Slightly different than `pandas-gbq`, here the default is `None`.

New in version 1.0.0.

Returns

df: DataFrame DataFrame representing results of query.

See also:

`pandas_gbq.read_gbq` This function in the pandas-gbq library.

`DataFrame.to_gbq` Write a DataFrame to Google BigQuery.

3.1.15 STATA

`read_stata`(filepath_or_buffer[, ...])	Read Stata file into DataFrame.

pandas.read_stata

`pandas.`**`read_stata`**(*filepath_or_buffer*, *convert_dates=True*, *convert_categoricals=True*, *index_col=None*, *convert_missing=False*, *preserve_dtypes=True*, *columns=None*, *order_categoricals=True*, *chunksize=None*, *iterator=False*)

Read Stata file into DataFrame.

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, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.dta`.

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 handler (e.g. via builtin `open` function) or `StringIO`.

convert_dates [bool, default True] Convert date variables to DataFrame time values.

convert_categoricals [bool, default True] Read value labels and convert columns to Categorical/Factor variables.

index_col [str, optional] Column to set as index.

convert_missing [bool, default False] Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissingValue objects.

preserve_dtypes [bool, default True] Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).

columns [list or None] Columns to retain. Columns will be returned in the given order. None returns all columns.

order_categoricals [bool, default True] Flag indicating whether converted categorical data are ordered.

chunksize [int, default None] Return StataReader object for iterations, returns chunks with given number of lines.

iterator [bool, default False] Return StataReader object.

Returns

DataFrame or StataReader

See also:

io.stata.StataReader Low-level reader for Stata data files.

DataFrame.to_stata Export Stata data files.

Examples

Read a Stata dta file:

```
>>> df = pd.read_stata('filename.dta')
```

Read a Stata dta file in 10,000 line chunks:

```
>>> itr = pd.read_stata('filename.dta', chunksize=10000)
>>> for chunk in itr:
...     do_something(chunk)
```

StataReader.data_label	Return data label of Stata file.
StataReader.value_labels(self)	Return a dict, associating each variable name a dict, associating each value its corresponding label.
StataReader.variable_labels(self)	Return variable labels as a dict, associating each variable name with corresponding label.
StataWriter.write_file(self)	

pandas.io.stata.StataReader.data_label

property StataReader.**data_label**
Return data label of Stata file.

pandas.io.stata.StataReader.value_labels

StataReader.**value_labels**(*self*)
Return a dict, associating each variable name a dict, associating each value its corresponding label.

Returns

dict

pandas.io.stata.StataReader.variable_labels

StataReader.**variable_labels**(*self*)

Return variable labels as a dict, associating each variable name with corresponding label.

Returns

dict

pandas.io.stata.StataWriter.write_file

StataWriter.**write_file**(*self*)

3.2 General functions

3.2.1 Data manipulations

`melt`(frame[, id_vars, value_vars, var_name, . . .])	Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.
`pivot`(data[, index, columns, values])	Return reshaped DataFrame organized by given index / column values.
`pivot_table`(data[, values, index, columns, . . .])	Create a spreadsheet-style pivot table as a DataFrame.
`crosstab`(index, columns[, values, rownames, . . .])	Compute a simple cross tabulation of two (or more) factors.
`cut`(x, bins, right[, labels])	Bin values into discrete intervals.
`qcut`(x, q[, labels])	Quantile-based discretization function.
`merge`(left, right, how[, on, left_on, . . .])	Merge DataFrame or named Series objects with a database-style join.
`merge_ordered`(left, right[, on, left_on, . . .])	Perform merge with optional filling/interpolation.
`merge_asof`(left, right[, on, left_on, . . .])	Perform an asof merge.
`concat`()	Concatenate pandas objects along a particular axis with optional set logic along the other axes.
`get_dummies`(data[, prefix, prefix_sep, . . .])	Convert categorical variable into dummy/indicator variables.
`factorize`(values, sort, na_sentinel, . . .)	Encode the object as an enumerated type or categorical variable.
`unique`(values)	Hash table-based unique.
`wide_to_long`(df, stubnames, i, j, sep, suffix)	Wide panel to long format.

pandas.melt

pandas.**melt**(*frame: pandas.core.frame.DataFrame*, *id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*) → pandas.core.frame.DataFrame

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

Parameters

id_vars [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or str, optional] If columns are a MultiIndex then use this level to melt.

Returns

DataFrame Unpivoted DataFrame.

See also:

DataFrame.melt

pivot_table

DataFrame.pivot

Series.explode

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> pd.melt(df, col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> pd.melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

pandas.pivot

`pandas.pivot`(*data: 'DataFrame'*, *index=None*, *columns=None*, *values=None*) → 'DataFrame'

Return reshaped DataFrame organized by given index / column values.

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

Parameters

data [DataFrame]

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

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

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

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

Returns

DataFrame Returns reshaped DataFrame.

Raises

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

See also:

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

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

Notes

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

Examples

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

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

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

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

A ValueError is raised if there are any duplicates.

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

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

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

pandas.pivot_table

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

Create a spreadsheet-style pivot table as a DataFrame.

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

Parameters

data [DataFrame]

values [column to aggregate, optional]

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

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

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

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

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

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

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

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

Changed in version 0.25.0.

Returns

DataFrame An Excel style pivot table.

See also:

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

Examples

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

This first example aggregates values by taking the sum.

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

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

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

pandas.crosstab

`pandas.crosstab`(*index*, *columns*, *values=None*, *rownames=None*, *colnames=None*, *aggfunc=None*, *margins=False*, *margins_name: str = 'All'*, *dropna: bool = True*, *normalize=False*) → 'DataFrame'

Compute a simple cross tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed.

Parameters

index [array-like, Series, or list of arrays/Series] Values to group by in the rows.

columns [array-like, Series, or list of arrays/Series] Values to group by in the columns.

values [array-like, optional] Array of values to aggregate according to the factors. Requires *aggfunc* be specified.

rownames [sequence, default None] If passed, must match number of row arrays passed.

colnames [sequence, default None] If passed, must match number of column arrays passed.

aggfunc [function, optional] If specified, requires *values* be specified as well.

margins [bool, default False] Add row/column margins (subtotals).

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

New in version 0.21.0.

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

normalize [bool, {'all', 'index', 'columns'}, or {0,1}, default False] Normalize by dividing all values by the sum of values.

- If passed 'all' or *True*, will normalize over all values.
- If passed 'index' will normalize over each row.
- If passed 'columns' will normalize over each column.
- If margins is *True*, will also normalize margin values.

Returns

DataFrame Cross tabulation of the data.

See also:

DataFrame.pivot Reshape data based on column values.

pivot_table Create a pivot table as a DataFrame.

Notes

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

Any input passed containing Categorical data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

In the event that there aren't overlapping indexes an empty DataFrame will be returned.

Examples

```
>>> a = np.array(["foo", "foo", "foo", "foo", "bar", "bar",
...               "bar", "bar", "foo", "foo", "foo"], dtype=object)
>>> b = np.array(["one", "one", "one", "two", "one", "one",
...               "one", "two", "two", "two", "one"], dtype=object)
>>> c = np.array(["dull", "dull", "shiny", "dull", "dull", "shiny",
...               "shiny", "dull", "shiny", "shiny", "shiny"],
...              dtype=object)
>>> pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
b    one        two
c    dull shiny dull shiny
a
bar     1     2    1     0
foo     2     2    1     2
```

Here 'c' and 'f' are not represented in the data and will not be shown in the output because dropna is True by default. Set dropna=False to preserve categories with no data.

```
>>> foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
>>> bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
>>> pd.crosstab(foo, bar)
col_0  d  e
row_0
a      1  0
b      0  1
>>> pd.crosstab(foo, bar, dropna=False)
col_0  d  e  f
row_0
a      1  0  0
b      0  1  0
c      0  0  0
```

pandas.cut

`pandas.cut`(*x*, *bins*, *right: bool = True*, *labels=None*, *retbins: bool = False*, *precision: int = 3*, *include_lowest: bool = False*, *duplicates: str = 'raise'*)

Bin values into discrete intervals.

Use *cut* when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable. For example, *cut* could convert ages to groups of age ranges. Supports binning into an equal number of bins, or a pre-specified array of bins.

Parameters

x [array-like] The input array to be binned. Must be 1-dimensional.

bins [int, sequence of scalars, or IntervalIndex] The criteria to bin by.

- int : Defines the number of equal-width bins in the range of *x*. The range of *x* is extended by .1% on each side to include the minimum and maximum values of *x*.
- sequence of scalars : Defines the bin edges allowing for non-uniform width. No extension of the range of *x* is done.
- IntervalIndex : Defines the exact bins to be used. Note that IntervalIndex for *bins* must be non-overlapping.

right [bool, default True] Indicates whether *bins* includes the rightmost edge or not. If `right == True` (the default), then the *bins* `[1, 2, 3, 4]` indicate (1,2], (2,3], (3,4]. This argument is ignored when *bins* is an IntervalIndex.

labels [array or False, default None] Specifies the labels for the returned bins. Must be the same length as the resulting bins. If False, returns only integer indicators of the bins. This affects the type of the output container (see below). This argument is ignored when *bins* is an IntervalIndex. If True, raises an error.

retbins [bool, default False] Whether to return the bins or not. Useful when bins is provided as a scalar.

precision [int, default 3] The precision at which to store and display the bins labels.

include_lowest [bool, default False] Whether the first interval should be left-inclusive or not.

duplicates [{default 'raise', 'drop'}, optional] If bin edges are not unique, raise ValueError or drop non-uniques.

New in version 0.23.0.

Returns

out [Categorical, Series, or ndarray] An array-like object representing the respective bin for each value of *x*. The type depends on the value of *labels*.

- True (default) : returns a Series for Series *x* or a Categorical for all other inputs. The values stored within are Interval dtype.
- sequence of scalars : returns a Series for Series *x* or a Categorical for all other inputs. The values stored within are whatever the type in the sequence is.
- False : returns an ndarray of integers.

bins [numpy.ndarray or IntervalIndex.] The computed or specified bins. Only returned when *retbins=True*. For scalar or sequence *bins*, this is an ndarray with the computed bins. If set *duplicates=drop*, *bins* will drop non-unique bin. For an IntervalIndex *bins*, this is equal to *bins*.

See also:

qcut Discretize variable into equal-sized buckets based on rank or based on sample quantiles.

Categorical Array type for storing data that come from a fixed set of values.

Series One-dimensional array with axis labels (including time series).

IntervalIndex Immutable Index implementing an ordered, sliceable set.

Notes

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Series or Categorical object.

Examples

Discretize into three equal-sized bins.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
...
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
```

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3, retbins=True)
...
([(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
array([0.994, 3.   , 5.   , 7.   ]))
```

Discovers the same bins, but assign them specific labels. Notice that the returned Categorical's categories are *labels* and is ordered.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]),
...        3, labels=["bad", "medium", "good"])
[bad, good, medium, medium, good, bad]
Categories (3, object): [bad < medium < good]
```

`labels=False` implies you just want the bins back.

```
>>> pd.cut([0, 1, 1, 2], bins=4, labels=False)
array([0, 1, 1, 3])
```

Passing a Series as an input returns a Series with categorical dtype:

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, 3)
...
a    (1.992, 4.667]
b    (1.992, 4.667]
c    (4.667, 7.333]
d     (7.333, 10.0]
e     (7.333, 10.0]
dtype: category
Categories (3, interval[float64]): [(1.992, 4.667] < (4.667, ...
```

Passing a Series as an input returns a Series with mapping value. It is used to map numerically to intervals based on bins.

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, [0, 2, 4, 6, 8, 10], labels=False, retbins=True, right=False)
...
(a    0.0
 b    1.0
 c    2.0
 d    3.0
 e    4.0
 dtype: float64, array([0, 2, 4, 6, 8]))
```

Use *drop* optional when bins is not unique

```
>>> pd.cut(s, [0, 2, 4, 6, 10, 10], labels=False, retbins=True,
...        right=False, duplicates='drop')
...
(a    0.0
 b    1.0
 c    2.0
 d    3.0
 e    3.0
 dtype: float64, array([0, 2, 4, 6, 8]))
```

Passing an IntervalIndex for *bins* results in those categories exactly. Notice that values not covered by the IntervalIndex are set to NaN. 0 is to the left of the first bin (which is closed on the right), and 1.5 falls between two bins.

```
>>> bins = pd.IntervalIndex.from_tuples([(0, 1), (2, 3), (4, 5)])
>>> pd.cut([0, 0.5, 1.5, 2.5, 4.5], bins)
[NaN, (0, 1], NaN, (2, 3], (4, 5]]
Categories (3, interval[int64]): [(0, 1] < (2, 3] < (4, 5]]
```

pandas.qcut

`pandas.qcut` (*x*, *q*, *labels=None*, *retbins: bool = False*, *precision: int = 3*, *duplicates: str = 'raise'*)

Quantile-based discretization function.

Discretize variable into equal-sized buckets based on rank or based on sample quantiles. For example 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data point.

Parameters

x [1d ndarray or Series]

q [int or list-like of int] Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately array of quantiles, e.g. [0, .25, .5, .75, 1.] for quartiles.

labels [array or False, default None] Used as labels for the resulting bins. Must be of the same length as the resulting bins. If False, return only integer indicators of the bins. If True, raises an error.

retbins [bool, optional] Whether to return the (bins, labels) or not. Can be useful if bins is given as a scalar.

precision [int, optional] The precision at which to store and display the bins labels.

duplicates [{default 'raise', 'drop'}, optional] If bin edges are not unique, raise ValueError or drop non-uniques.

Returns

out [Categorical or Series or array of integers if labels is False] The return type (Categorical or Series) depends on the input: a Series of type category if input is a Series else Categorical. Bins are represented as categories when categorical data is returned.

bins [ndarray of floats] Returned only if *retbins* is True.

Notes

Out of bounds values will be NA in the resulting Categorical object

Examples

```
>>> pd.qcut(range(5), 4)
...
[(-0.001, 1.0], (-0.001, 1.0], (1.0, 2.0], (2.0, 3.0], (3.0, 4.0]]
Categories (4, interval[float64]): [(-0.001, 1.0] < (1.0, 2.0] ...
```

```
>>> pd.qcut(range(5), 3, labels=["good", "medium", "bad"])
...
[good, good, medium, bad, bad]
Categories (3, object): [good < medium < bad]
```

```
>>> pd.qcut(range(5), 4, labels=False)
array([0, 0, 1, 2, 3])
```

pandas.merge

`pandas.merge`(*left*, *right*, *how: str = 'inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index: bool = False*, *right_index: bool = False*, *sort: bool = False*, *suffixes='_x', '_y'*, *copy: bool = True*, *indicator: bool = False*, *validate=None*) → 'DataFrame'

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

left [DataFrame]

right [DataFrame or named Series] Object to merge with.

how [{'left', 'right', 'outer', 'inner'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes [tuple of (str, str), default ('_x', '_y')] Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Returns

DataFrame A DataFrame of the two merged objects.

See also:

`merge_ordered` Merge with optional filling/interpolation.

`merge_asof` Merge on nearest keys.

`DataFrame.join` Similar method using indices.

Notes

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0 Support for merging named Series objects was added in version 0.24.0

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo            5
1  foo           1  foo            8
2  foo           5  foo            5
3  foo           5  foo            8
4  bar           2  bar            6
5  baz           3  baz            7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

pandas.merge_ordered

`pandas.merge_ordered`(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *left_by=None*, *right_by=None*, *fill_method=None*, *suffixes='_x', '_y'*, *how: str = 'outer'*) → 'DataFrame'

Perform merge with optional filling/interpolation.

Designed for ordered data like time series data. Optionally perform group-wise merge (see examples).

Parameters

left [DataFrame]

right [DataFrame]

on [label or list] Field names to join on. Must be found in both DataFrames.

left_on [label or list, or array-like] Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns.

right_on [label or list, or array-like] Field names to join on in right DataFrame or vector/list of vectors per left_on docs.

left_by [column name or list of column names] Group left DataFrame by group columns and merge piece by piece with right DataFrame.

right_by [column name or list of column names] Group right DataFrame by group columns and merge piece by piece with left DataFrame.

fill_method [{'ffill', None}, default None] Interpolation method for data.

suffixes [Sequence, default is ("_x", "_y")] A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in *left* and *right* respectively. Pass a value of *None* instead of a string to indicate that the column name from *left* or *right* should be left as-is, with no suffix. At least one of the values must not be None.

Changed in version 0.25.0.

how [{'left', 'right', 'outer', 'inner'}, default 'outer']

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join).

Returns

DataFrame The merged DataFrame output type will the be same as 'left', if it is a subclass of DataFrame.

See also:

`merge`

`merge_asof`

Examples

```
>>> A
    key  lvalue group
0   a       1     a
1   c       2     a
2   e       3     a
3   a       1     b
4   c       2     b
5   e       3     b
```

```
>>> B
    Key  rvalue
0     b       1
1     c       2
2     d       3
```

```
>>> merge_ordered(A, B, fill_method='ffill', left_by='group')
  group key  lvalue  rvalue
0     a   a       1     NaN
1     a   b       1     1.0
2     a   c       2     2.0
3     a   d       2     3.0
4     a   e       3     3.0
5     b   a       1     NaN
6     b   b       1     1.0
7     b   c       2     2.0
8     b   d       2     3.0
9     b   e       3     3.0
```

pandas.merge_asof

pandas.**merge_asof**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *left_index: bool = False*, *right_index: bool = False*, *by=None*, *left_by=None*, *right_by=None*, *suffixes='_x', '_y'*, *tolerance=None*, *allow_exact_matches: bool = True*, *direction: str = 'backward'*) → 'DataFrame'

Perform an asof merge. This is similar to a left-join except that we match on nearest key rather than equal keys.

Both DataFrames must be sorted by the key.

For each row in the left DataFrame:

- A "backward" search selects the last row in the right DataFrame whose 'on' key is less than or equal to the left's key.
- A "forward" search selects the first row in the right DataFrame whose 'on' key is greater than or equal to the left's key.
- A "nearest" search selects the row in the right DataFrame whose 'on' key is closest in absolute distance to the left's key.

The default is "backward" and is compatible in versions below 0.20.0. The direction parameter was added in version 0.20.0 and introduces "forward" and "nearest".

Optionally match on equivalent keys with 'by' before searching with 'on'.

Parameters

left [DataFrame]

right [DataFrame]

on [label] Field name to join on. Must be found in both DataFrames. The data MUST be ordered. Furthermore this must be a numeric column, such as datetimelike, integer, or float. On or left_on/right_on must be given.

left_on [label] Field name to join on in left DataFrame.

right_on [label] Field name to join on in right DataFrame.

left_index [bool] Use the index of the left DataFrame as the join key.

right_index [bool] Use the index of the right DataFrame as the join key.

by [column name or list of column names] Match on these columns before performing merge operation.

left_by [column name] Field names to match on in the left DataFrame.

right_by [column name] Field names to match on in the right DataFrame.

suffixes [2-length sequence (tuple, list, . . .)] Suffix to apply to overlapping column names in the left and right side, respectively.

tolerance [int or Timedelta, optional, default None] Select asof tolerance within this range; must be compatible with the merge index.

allow_exact_matches [bool, default True]

- If True, allow matching with the same 'on' value (i.e. less-than-or-equal-to / greater-than-or-equal-to)
- If False, don't match the same 'on' value (i.e., strictly less-than / strictly greater-than).

direction ['backward' (default), 'forward', or 'nearest'] Whether to search for prior, subsequent, or closest matches.

Returns

merged [DataFrame]

See also:

merge

merge_ordered

Examples

```
>>> left = pd.DataFrame({'a': [1, 5, 10], 'left_val': ['a', 'b', 'c']})
>>> left
    a left_val
0   1        a
1   5        b
2  10        c
```

```
>>> right = pd.DataFrame({'a': [1, 2, 3, 6, 7],
...                       'right_val': [1, 2, 3, 6, 7]})
>>> right
   a  right_val
0  1          1
1  2          2
```

(continues on next page)

(continued from previous page)

```
2  3          3
3  6          6
4  7          7
```

```
>>> pd.merge_asof(left, right, on='a')
    a left_val  right_val
0   1        a          1
1   5        b          3
2  10        c          7
```

```
>>> pd.merge_asof(left, right, on='a', allow_exact_matches=False)
    a left_val  right_val
0   1        a        NaN
1   5        b        3.0
2  10        c        7.0
```

```
>>> pd.merge_asof(left, right, on='a', direction='forward')
    a left_val  right_val
0   1        a        1.0
1   5        b        6.0
2  10        c        NaN
```

```
>>> pd.merge_asof(left, right, on='a', direction='nearest')
    a left_val  right_val
0   1        a          1
1   5        b          6
2  10        c          7
```

We can use indexed DataFrames as well.

```
>>> left = pd.DataFrame({'left_val': ['a', 'b', 'c']}, index=[1, 5, 10])
>>> left
   left_val
1         a
5         b
10        c
```

```
>>> right = pd.DataFrame({'right_val': [1, 2, 3, 6, 7]},
...                      index=[1, 2, 3, 6, 7])
>>> right
   right_val
1          1
2          2
3          3
6          6
7          7
```

```
>>> pd.merge_asof(left, right, left_index=True, right_index=True)
   left_val  right_val
1         a          1
5         b          3
10        c          7
```

Here is a real-world times-series example

```
>>> quotes
                     time ticker     bid     ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT   51.95   51.96
2 2016-05-25 13:30:00.030   MSFT   51.97   51.98
3 2016-05-25 13:30:00.041   MSFT   51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL   97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT   52.01   52.03
```

```
>>> trades
                     time ticker   price  quantity
0 2016-05-25 13:30:00.023   MSFT   51.95        75
1 2016-05-25 13:30:00.038   MSFT   51.95       155
2 2016-05-25 13:30:00.048   GOOG  720.77       100
3 2016-05-25 13:30:00.048   GOOG  720.92       100
4 2016-05-25 13:30:00.048   AAPL   98.00       100
```

By default we are taking the asof of the quotes

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker')
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within 2ms between the quote time and the trade time

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker',
...                       tolerance=pd.Timedelta('2ms'))
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155     NaN     NaN
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. However *prior* data will propagate forward

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker',
...                       tolerance=pd.Timedelta('10ms'),
...                       allow_exact_matches=False)
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75     NaN     NaN
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100     NaN     NaN
3 2016-05-25 13:30:00.048   GOOG  720.92       100     NaN     NaN
```

(continues on next page)

(continued from previous page)

```
4 2016-05-25 13:30:00.048   AAPL   98.00       100      NaN      NaN
```

pandas.concat

`pandas.concat`(*objs: Union[Iterable['DataFrame'], Mapping[Optional[Hashable], 'DataFrame']], axis='0', join: str = "'outer'", ignore_index: bool = 'False', keys='None', levels='None', names='None', verify_integrity: bool = 'False', sort: bool = 'False', copy: bool = 'True'*) → 'DataFrame'

`pandas.concat`(*objs: Union[Iterable[FrameOrSeriesUnion], Mapping[Optional[Hashable], FrameOrSeriesUnion]], axis='0', join: str = "'outer'", ignore_index: bool = 'False', keys='None', levels='None', names='None', verify_integrity: bool = 'False', sort: bool = 'False', copy: bool = 'True'*) → FrameOrSeriesUnion

Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

Parameters

objs [a sequence or mapping of Series or DataFrame objects] If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.

axis [{0/'index', 1/'columns'}, default 0] The axis to concatenate along.

join [{'inner', 'outer'}, default 'outer'] How to handle indexes on other axis (or axes).

ignore_index [bool, default False] If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, …, n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.

keys [sequence, default None] If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level.

levels [list of sequences, default None] Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.

names [list, default None] Names for the levels in the resulting hierarchical index.

verify_integrity [bool, default False] Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.

sort [bool, default False] Sort non-concatenation axis if it is not already aligned when *join* is 'outer'. This has no effect when `join='inner'`, which already preserves the order of the non-concatenation axis.

New in version 0.23.0.

Changed in version 1.0.0: Changed to not sort by default.

copy [bool, default True] If False, do not copy data unnecessarily.

Returns

object, type of objs When concatenating all `Series` along the index (axis=0), a `Series` is returned. When `objs` contains at least one `DataFrame`, a `DataFrame` is returned. When concatenating along the columns (axis=1), a `DataFrame` is returned.

See also:

Series.append Concatenate Series.

DataFrame.append Concatenate DataFrames.

DataFrame.join Join DataFrames using indexes.

DataFrame.merge Merge DataFrames by indexes or columns.

Notes

The keys, levels, and names arguments are all optional.

A walkthrough of how this method fits in with other tools for combining pandas objects can be found here.

Examples

Combine two `Series`.

```
>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```

Clear the existing index and reset it in the result by setting the `ignore_index` option to `True`.

```
>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object
```

Add a hierarchical index at the outermost level of the data with the `keys` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2'])
s1  0    a
    1    b
s2  0    c
    1    d
dtype: object
```

Label the index keys you create with the `names` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2'],
...           names=['Series name', 'Row ID'])
Series name  Row ID
s1           0         a
             1         b
s2           0         c
             1         d
dtype: object
```

Combine two `DataFrame` objects with identical columns.

```
>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                    columns=['letter', 'number'])
>>> df1
  letter  number
0      a       1
1      b       2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                    columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
>>> pd.concat([df1, df2])
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4
```

Combine DataFrame objects with overlapping columns and return everything. Columns outside the intersection will be filled with NaN values.

```
>>> df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                    columns=['letter', 'number', 'animal'])
>>> df3
  letter  number animal
0      c       3    cat
1      d       4    dog
>>> pd.concat([df1, df3], sort=False)
  letter  number animal
0      a       1    NaN
1      b       2    NaN
0      c       3    cat
1      d       4    dog
```

Combine DataFrame objects with overlapping columns and return only those that are shared by passing inner to the join keyword argument.

```
>>> pd.concat([df1, df3], join="inner")
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4
```

Combine DataFrame objects horizontally along the x axis by passing in axis=1.

```
>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                    columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
  letter  number  animal    name
0      a       1    bird   polly
1      b       2  monkey  george
```

Prevent the result from including duplicate index values with the verify_integrity option.

```
>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
   0
a  1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
   0
a  2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
    ...
ValueError: Indexes have overlapping values: ['a']
```

pandas.get_dummies

`pandas.get_dummies`(*data*, *prefix=None*, *prefix_sep='_'*, *dummy_na=False*, *columns=None*, *sparse=False*, *drop_first=False*, *dtype=None*) → 'DataFrame'

Convert categorical variable into dummy/indicator variables.

Parameters

data [array-like, Series, or DataFrame] Data of which to get dummy indicators.

prefix [str, list of str, or dict of str, default None] String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

prefix_sep [str, default '_'] If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

dummy_na [bool, default False] Add a column to indicate NaNs, if False NaNs are ignored.

columns [list-like, default None] Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

sparse [bool, default False] Whether the dummy-encoded columns should be backed by a `SparseArray` (True) or a regular NumPy array (False).

drop_first [bool, default False] Whether to get k-1 dummies out of k categorical levels by removing the first level.

dtype [dtype, default np.uint8] Data type for new columns. Only a single dtype is allowed.

New in version 0.23.0.

Returns

DataFrame Dummy-coded data.

See also:

`Series.str.get_dummies` Convert Series to dummy codes.