Merge, join, and concatenate

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

Concatenating objects

The concat() function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say "if any" because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of concat and what it can do, here is a simple example:

```
index=[0, 1, 2, 3])
    ...:
    . . . :
In [2]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                                     'B': ['B4', 'B5', 'B6', 'B7'],
'C': ['C4', 'C5', 'C6', 'C7'],
'D': ['D4', 'D5', 'D6', 'D7']},
    ...:
    . . . :
    . . . :
                                    index=[4, 5, 6, 7])
    . . . :
    . . . :
In [3]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                                   'B': ['B8', 'B9', 'B10', 'B11'],
'C': ['C8', 'C9', 'C10', 'C11'],
'D': ['D8', 'D9', 'D10', 'D11']},
index=[8, 9, 10, 11])
    . . . :
    . . . :
In [4]: frames = [df1, df2, df3]
In [5]: result = pd.concat(frames)
```

		df1					Result		
	Α	В	С	D					
0	A0	В0	8	D0		Α	В	C	D
1	A1	B1	Cl	D1	0	A0	В0	8	D0
2	A2	B2	CZ	D2	1	Al	B1	C1	D1
3	A3	В3	СЗ	D3	2	A2	B2	(2	D2
		df2							
	Α	В	С	D	3	A3	B3	СЗ	D3
4	A4	B4	C4	D4	4	A4	B4	C4	D4
5	A5	B5	C5	D5	5	A5	B5	C5	D5
6	Аб	B6	C6	D6	6	A6	В6	C6	D6
7	A7	B7	C7	D7	7	A7	B7	C7	D7
		df3							
	Α	В	С	D	8	A8	B8	C8	DB
8	A8	B8	C8	DB	9	A9	B9	C9	D9
9	A9	B9	C9	D9	10	A10	B10	C10	D10
10	A10	B10	C10	D10	11	A11	B11	C11	D11
11	A11	B11	C11	D11					

Like its sibling function on ndarrays, numpy.concatenate, pandas.concat takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of "what to do with the other axes":

- objs: a sequence or mapping of Series, DataFrame, or Panel 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, 1, ...}, default 0. The axis to concatenate along.
- join: {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- ignore_index: boolean, default False. If True, do not use the index values on 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.
- join_axes: list of Index objects. Specific indexes to use for the other n 1 axes instead of performing inner/outer set logic.
- keys: sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.
- 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: boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.
- copy: boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments don't make much sense. Let's revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the keys argument:

```
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	α	D0			А	В	С	D
1	A1	B1	C1	D1	×	0	AD	BO	8	D0
2	A2	B2	CZ	D2	×	1	A1	B1	а	D1
3	A3	В3	З	D3	×	2	A2	B2	a	D2
		df2								
	Α	В	С	D	×	3	A3	B3	З	D3
4	A4	B4	C4	D4	У	4	A4	В4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	O	D5
6	Аб	B6	C6	D6	У	0	Аß	Вб	C6	D6
7	A7	B7	C7	D7	У	7	A7	B7	a	D7
		df3				_				-
	Α	В	С	D	z	8	AB	BB	СВ	D8
8	A8	B8	C8	D8	z	9	A9	B9	Ð	D9
9	A9	B9	C9	D9	z	10	A10	B10	П0	D10
10	A10	B10	C10	D10	z	11	A11	B11	G1	D11
11	A11	B11	C11	D11						

As you can see (if you've read the rest of the documentation), the resulting object's index has a hierarchical index. This means that we can now select out each chunk by key:

```
In [7]: result.loc['y']
Out[7]:
      В
         С
             D
   Α
  A4 B4 C4 D4
 A5
5
     B5 C5
            D5
6 A6
     B6 C6
            D6
  Α7
     B7 C7
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

Note: It is worth noting that **concat()** (and therefore **append()**) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following three ways:

- Take the union of them all, <code>join='outer'</code>. This is the default option as it results in zero information loss.
- Take the intersection, join='inner'.
- Use a specific index, as passed to the <code>join_axes</code> argument.

Here is an example of each of these methods. First, the default <code>join='outer'</code> behavior:

		df1				df	4					Res	ult			
										Α	В	С	D	В	D	F
	Α	В	С	D		В	D	F	0	A0	В0	8	D0	NaN	NaN	NaN
0	A0	B0	ω	D0	2	B2	D2	F2	1	A1	B1	C1	D1	NaN	NaN	NaN
1	Al	B1	CI	D1	3	В3	D3	F3	2	A2	B2	CZ	D2	B2	D2	F2
2	A2	B2	Ŋ	D2	6	B6	D6	F6	3	A3	В3	C3	D3	В3	D3	F3
3	A3	В3	СЗ	D3	7	B7	D7	F7	6	NaN	NaN	NaN	NaN	В6	D6	F6
									7	NaN	NaN	NaN	NaN	B7	D7	F7

Warning:

Changed in version 0.23.0.

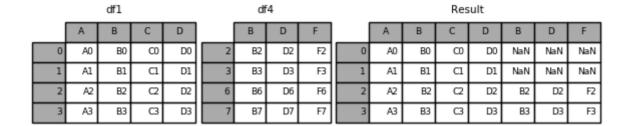
The default behavior with <code>join='outer'</code> is to sort the other axis (columns in this case). In a future version of pandas, the default will be to not sort. We specified <code>sort=False</code> to opt in to the new behavior now.

Here is the same thing with join='inner':

```
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```

		df1				đ	f4					Res	ult			
	Α	В	С	D		В	D	F								
0	A0	BO	œ	D0	2	B2	D2	F2		Α	В	С	D	В	D	F
1	A1	B1	Cl	D1	3	В3	D3	F3	2	A2	B2	C2	D2	B2	D2	F2
2	A2	B2	C2	D2	6	B6	D6	F6	3	A3	В3	C3	D3	В3	D3	F3
3	A3	В3	СЗ	D3	7	B7	D7	F7								

Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:



Concatenating using append

A useful shortcut to **concat()** are the **append()** instance methods on <code>series</code> and <code>DataFrame</code>. These methods actually predated <code>concat</code>. They concatenate along <code>axis=0</code>, namely the index:

			df1					Result		
		Α	В	С	D		Α	В	С	D
	0	A0	B0	œ	D0		40	DO.		D0
Г	1	A1	B1	C1	D1	0	A0	BO	ω	D0
Г	2	A2	B2	C2	D2	1	A1	B1	C1	D1
Г	3	A3	В3	СЗ	D3	2	A2	B2	U	D2
_			df2			3	A3	В3	СЗ	D3
_		Α	В	С	D	4	A4	B4	C4	D4
	4	A4	B4	C4	D4	5	A5	B5	C5	D5
	5	A5	B5	C5	D5	6	A6	B6		De
	6	Аб	B6	C6	D6	0	PID	80	C6	D6
	7	A7	В7	C7	D7	7	A7	B7	C7	D7

In the case of $\mathtt{DataFrame}$, the indexes must be disjoint but the columns do not need to be:

```
In [13]: result = df1.append(df4, sort=False)
```

			df	1					Res	sult		
_		Α	В	3	С	D		Α	В	С	D	F
П	0	A0		B0	α	D0		40	DO.		- Do	
Г	1	Al		В1	C	l D1	0	A0	BO	ω	D0	NaN
h	2	A2		B2	C	2 D2	1	A1	B1	Cl	D1	NaN
r	3	A3		ВЗ	C	3 D3	2	A2	B2	(2	D2	NaN
_	df4					3	A3	В3	СЗ	D3	NaN	
_		В			D	F	2	NaN	B2	NaN	D2	F2
l	2	2	B2		D2	F2	3	NaN	В3	NaN	D3	F3
Г	3	3	В3		D3	F3	-	N-N	ne	NI-NI	De	re.
Ī	(5	B6		D6	F6	6	NaN	B6	NaN	D6	F6
١		-	В7		D7	F7	7	NaN	B7	NaN	D7	F7

append may take multiple objects to concatenate:

```
In [14]: result = df1.append([df2, df3])
```

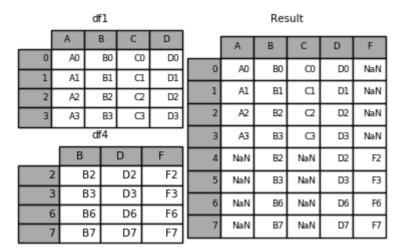
		dfl					Result		
	Α	В	С	D	١.				
0	A0	B0	ω	D0		Α	В	С	D
1	A1	B1	C1	D1	0	A0	B0	ω	D0
2	A2	B2	C2	D2	1	A1	B1	C1	D1
3	A3	В3	СЗ	D3	2	A2	B2	C2	D2
		df2			_				
	Α	В	С	D	3	A3	В3	СЗ	D3
4	A4	B4	C4	D4	4	A4	B4	C4	D4
5	A5	B5	C5	D5	5	A5	B5	C5	D5
6	Аб	B6	C6	D6	6	Аб	B6	C6	D6
7	A7	B7	C7	D7	7	A7	B7	C7	D7
		df3							
-	Α	В	С	D	8	A8	B8	C8	DB
8	A8	B8	C8	DB	9	A9	B9	C9	D9
9	A9	B9	C9	D9	10	A10	B10	C10	D10
10	A10	B10	C10	D10	11	A11	B11	C11	D11
11	A11	B11	C11	D11					

Note: Unlike the append() method, which appends to the original list and returns None, append() here **does not** modify df1 and returns its copy with df2 appended.

Ignoring indexes on the concatenation axis

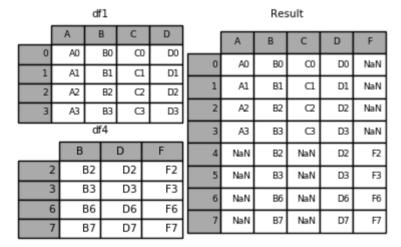
For DataFrame objects which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the ignore_index argument:

```
In [15]: result = pd.concat([df1, df4], ignore_index=True, sort=False)
```



This is also a valid argument to DataFrame.append():

```
In [16]: result = df1.append(df4, ignore_index=True, sort=False)
```



Concatenating with mixed ndims

You can concatenate a mix of series and DataFrame objects. The series will be transformed to DataFrame with the column name as the name of the series.

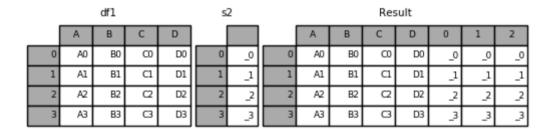
```
In [17]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')
In [18]: result = pd.concat([df1, s1], axis=1)
```

		df1			S	1			Res	ult		
	Α	В	С	D		Х		Α	В	С	D	Х
0	A0	B0	α	D0	0	Х0	0	A0	B0	α	D0	X0
1	A1	B1	CI	D1	1	X1	1	A1	B1	CI	D1	X1
2	A2	B2	C2	D2	2	X2	2	A2	B2	C2	D2	X2
3	A3	В3	СЗ	D3	3	ХЗ	3	A3	В3	СЗ	D3	ХЗ

Note: Since we're concatenating a series to a DataFrame, we could have achieved the same result with **DataFrame.assign()**. To concatenate an arbitrary number of pandas objects (DataFrame or Series), USE concat.

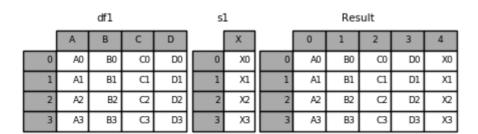
If unnamed series are passed they will be numbered consecutively.

```
In [19]: s2 = pd.Series(['_0', '_1', '_2', '_3'])
In [20]: result = pd.concat([df1, s2, s2, s2], axis=1)
```



Passing ignore index=True will drop all name references.

```
In [21]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```



More concatenating with group keys

A fairly common use of the keys argument is to override the column names when creating a new DataFrame based on existing Series. Notice how the default behaviour consists on letting the resulting DataFrame inherit the parent Series' name, when these existed.

```
In [22]: s3 = pd.Series([0, 1, 2, 3], name='foo')
```

```
In [23]: s4 = pd.Series([0, 1, 2, 3])
In [24]: s5 = pd.Series([0, 1, 4, 5])
In [25]: pd.concat([s3, s4, s5], axis=1)
Out[25]:
  foo 0
          1
0
    0 0
          0
1
    1 1
          1
2
    2 2
          4
3
     3 3
```

Through the keys argument we can override the existing column names.

```
In [26]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
Out[26]:
   red blue yellow
0
     0
         0
1
    1
           1
                   1
2
           2
    2
                   4
3
     3
           3
                   5
```

Let's consider a variation of the very first example presented:

```
In [27]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	α	D0			А	В	С	D
1	Al	B1	C1	D1	×	0	AD	BO	8	D0
2	A2	B2	C2	D2	×	1	A1	B1	а	D1
3	A3	В3	СЗ	D3	×	2	A2	B2	a	D2
		df2								\vdash
	Α	В	С	D	×	3	A3	B3	З	D3
4	A4	B4	C4	D4	У	4	A4	В4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	O	D5
6	Аб	B6	C6	D6	У	6	Aß	B6	C6	D6
7	A7	B7	C7	D7	У	7	A7	B7	a	D7
		df3				_		-	-	\vdash
	Α	В	С	D	z	8	AB	BB	СВ	D8
8	A8	B8	C8	DB	z	9	A9	B9	Ø	D9
9	A9	B9	C9	D9	z	10	A10	B10	Ф.	D10
10	A10	B10	C10	D10	z	11	A11	B11	G1	D11
11	A11	B11	C11	D11						

You can also pass a dict to concat in which case the dict keys will be used for the keys argument (unless other keys are specified):

```
In [28]: pieces = {'x': df1, 'y': df2, 'z': df3}
In [29]: result = pd.concat(pieces)
```

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	α	D0			А	В	С	D
1	A1	B1	C1	D1	×	0	AD	BO	8	D0
2	A2	B2	C	D2	×	1	A1	B1	а	D1
3	A3	В3	З	D3	×	2	A2	B2	Q	D2
		df2								
	Α	В	С	D	×	3	A3	B3	З	D3
4	A4	B4	C4	D4	У	4	A4	В4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	G	D5
6	Аб	B6	O5	D6	У	0	Aß	B6	œ	D6
7	A7	B7	C7	D7	У	7	A7	B7	- 7	D7
		df3								
	Α	В	С	D	z	8	AB	BB	СВ	D8
8	A8	B8	C8	DB	z	9	A9	B9	Ø	D9
9	A9	B9	C9	D9	z	10	A10	B10	G0	D10
10	A10	B10	C10	D10	z	11	A11	B11	СII	D11
11	A11	B11	C11	D11						

```
In [30]: result = pd.concat(pieces, keys=['z', 'y'])
```

		dfl					Res	sult		
	Α	В	С	D						
0	A0	В0	8	D0						
1	A1	B1	CI	D1						
2	A2	B2	C2	D2			А	В	С	D
3	A3	В3	СЗ	D3	z	8	AB	BB	СВ	D8
		df2								
	Α	В	С	D	z	9	A9	B9	C9	D9
4	A4	B4	C4	D4	z	10	A10	B10	G 0	D10
5	A5	B5	C5	D5	z	11	A11	B11	аı	D11
6	Аб	B6	C6	D6	У	4	A4	B4	C4	D4
7	A7	В7	C7	D7	у	5	A5	B5	cs	D5
		df3				_			_	
	Α	В	С	D	У	6	Aß	B6	C6	D6
8	A8	B8	C8	DB	У	7	A7	В7	7	D7
9	A9	B9	C9	D9						
10	A10	B10	C10	D10						
11	A11	B11	C11	D11						

The MultiIndex created has levels that are constructed from the passed keys and the index of the DataFrame pieces:

```
In [31]: result.index.levels
Out[31]: FrozenList([['z', 'y'], [4, 5, 6, 7, 8, 9, 10, 11]])
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the <code>levels</code> argument:

```
df1 Result
```

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	8	D0			Α	В	U	D
1	A1	B1	Cl	D1	×	0	AD	В0	8	D0
2	A2	B2	C2	D2	×	1	A1	B1.	а	D1
3	A3	В3	ß	D3	×	2	A2	B2	Q	D2
		df2							-	
	Α	В	С	D	×	3	A3	B3	З	D3
4	A4	B4	C4	D4	У	4	A4	B4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	O	D5
6	Аб	B6	C 6	D6	У	6	Aß	B6	C6	D6
7	A7	B7	C7	D7	У	7	A7	B7	a	D7
		df3							_	
	Α	В	С	D	z	8	AB	B8	СВ	D8
8	A8	B8	C8	DB	z	9	A9	B9	Ð	D9
9	A9	B9	C9	D9	z	10	A10	B10	ПO	D10
10	A10	B10	C10	D10	z	11	A11	B11	G1	D11
11	A11	B11	C11	D11						

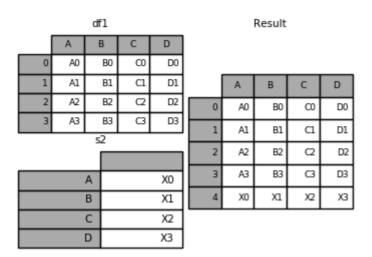
```
In [33]: result.index.levels
Out[33]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
```

This is fairly esoteric, but it is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

Appending rows to a DataFrame

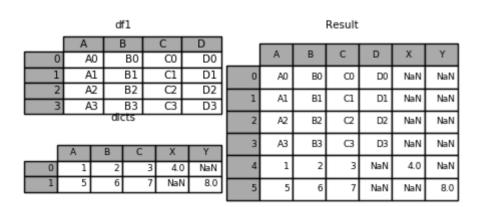
While not especially efficient (since a new object must be created), you can append a single row to a DataFrame by passing a series or dict to append, which returns a new DataFrame as above.

```
In [34]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])
In [35]: result = df1.append(s2, ignore_index=True)
```



You should use <code>ignore_index</code> with this method to instruct DataFrame to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed DataFrame and append or concatenate those objects.

You can also pass a list of dicts or Series:



Database-style DataFrame or named Series joining/merging

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like <code>base::merge.data.frame</code> in R). The reason for this is careful algorithmic design and the internal layout of the data in <code>DataFrame</code>.

See the cookbook for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a comparison with SQL.

pandas provides a single function, merge(), as the entry point for all standard database join operations between DataFrame or named Series objects:

- left: A DataFrame or named Series object.
- right: Another DataFrame or named Series object.
- on: Column or index level names to join on. Must be found in both the left and right DataFrame and/or Series objects. If not passed and left_index and right_index are False, the intersection of the columns in the DataFrames and/or Series will be inferred to be the join keys.
- left_on: Columns or index levels from the left DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.
- right_on: Columns or index levels from the right DataFrame or Series to use as keys. Can either
 be column names, index level names, or arrays with length equal to the length of the DataFrame
 or Series.
- left_index: If True, use the index (row labels) from the left DataFrame or Series as its join key(s). In the case of a DataFrame or Series with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame or Series.
- right index: Same usage as left index for the right DataFrame or Series
- how: One of 'left', 'right', 'outer', 'inner'. Defaults to inner. See below for more detailed description of each method.
- sort: Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve performance substantially in many cases.
- suffixes: A tuple of string suffixes to apply to overlapping columns. Defaults to ('x', 'y').
- copy: Always copy data (default True) from the passed DataFrame or named Series objects, even
 when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but
 this option is provided nonetheless.
- indicator: Add a column to the output DataFrame called _merge with information on the source of each row. _merge is Categorical-type and takes on a value of left_only for observations whose merge key only appears in 'left' DataFrame or Series, right_only for observations whose merge key only appears in 'right' DataFrame or Series, and both if the observation's merge key is found in both.
- validate: string, default None. If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": checks if merge keys are unique in both left and right datasets.
- "one to many" or "1:m": checks if merge keys are unique in left dataset.
- "many_to_one" or "m:1": checks 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.

Note: 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.

The return type will be the same as left. If left is a DataFrame or named series and right is a subclass of DataFrame, the return type will still be DataFrame.

merge is a function in the pandas namespace, and it is also available as a DataFrame instance method merge(), with the calling DataFrame being implicitly considered the left object in the join.

The related <code>join()</code> method, uses <code>merge</code> internally for the index-on-index (by default) and column(s)-on-index join. If you are joining on index only, you may wish to use <code>DataFrame.join</code> to save yourself some typing.

Brief primer on merge methods (relational algebra)

Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (DataFrame objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two DataFrame objects on their indexes (which must contain unique values).
- many-to-one joins: for example when joining an index (unique) to one or more columns in a different DataFrame.
- many-to-many joins: joining columns on columns.

Note: When joining columns on columns (potentially a many-to-many join), any indexes on the passed <code>DataFrame</code> objects will be discarded.

It is worth spending some time understanding the result of the **many-to-many** join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the **Cartesian product** of the associated data. Here is a very basic example with one unique key combination:

	le	ft			rig	ht				Res	sult		
	key	Α	В		key	С	D		key	Α	В	С	D
0	K0	A0	B0	0	K0	ω	D0	0	K0	A0	B0	α	D0
1	K1	A1	B1	1	K1	Cl	D1	1	K1	A1	B1	CI	D1
2	K2	A2	B2	2	K2	C2	D2	2	K2	A2	B2	C2	D2
3	КЗ	A3	В3	3	КЗ	C3	D3	3	КЗ	A3	В3	СЗ	D3

Here is a more complicated example with multiple join keys. Only the keys appearing in left and right are present (the intersection), since how='inner' by default.

			left					right						Result			
		key1	key2	Α	В		key1	key2	С	D		key1	key2	Α	В	С	D
	0	K0	K0	A0	B0	0	K0	K0	α	D0		-	,				
ł	1	КО	К1	A1	B1	1	K1	КО	Cl	D1	0	K0	K0	A0	BO	8	D0
ŀ					\vdash					\vdash	1	K1	K0	A2	B2	C1	D1
ı	2	K1	K0	A2	B2	2	K1	K0	C2	D2	2	К1	KO	A2	B2	C2	D2
ı	3	K2	K1	A3	В3	3	K2	K0	СЗ	D3		KI	NJ	72	DZ	- (2	DZ.

The how argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination **does not appear** in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

Merge method	SQL Join Name	Description
left	LEFT OUTER JOIN	Use keys from left frame only
right	RIGHT OUTER JOIN	Use keys from right frame only
outer	FULL OUTER JOIN	Use union of keys from both frames
inner	INNER JOIN	Use intersection of keys from both frames

```
In [44]: result = pd.merge(left, right, how='left', on=['key1', 'key2'])
```

left key2 Α key1 key1 key2 key2 ΚO KO Α0 \odot D0 В0 D0 KO A0 KO KO α ΚO A1 NaN В1 D1 K0 K1 A1 K1 K0 C1 D1 K1 K0 A2 C1 B2 C2 K1 KO A2 C2 D2 АЗ ВЗ K2 C3 D3 АЗ NaN NaN

```
In [45]: result = pd.merge(left, right, how='right', on=['key1', 'key2'])
```

		left					right						Result			
	key1	key2	Α	В		key1	key2	С	D		key1	key2	Α	В	С	D
0	K0	K0	A0	B0	0	K0	K0	ω	D0	0	K0	K0	A0	B0	ω	D0
1	K0	K1	A1	B1	1	K1	K0	Cl	D1	1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	2	K1	K0	C2	D2	2	K1	K0	A2	B2	(2	D2
3	K2	K1	A3	В3	3	K2	K0	СЗ	D3	3	K2	K0	NaN	NaN	СЗ	D3

```
In [46]: result = pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

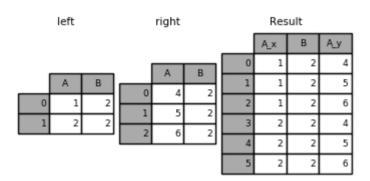
		left					right						Result			
											key1	key2	Α	В	С	D
	key1	key2	Α	В		key1	key2	С	D	0	K0	K0	A0	В0	8	D0
0	KO	KO	A0	В0	0	K0	K0	8	D0	1	K0	K1	A1	B1	NaN	NaN
1	KO	K1	A1	B1	1	K1	K0	Cl	D1	2	K1	K0	A2	B2	Cl	D1
2	K1	KO	A2	B2	2	K1	K0	C2	D2	3	K1	K0	A2	B2	Ŋ	D2
3	K2	K1	A3	В3	3	K2	K0	СЗ	D3	4	K2	K1	A3	В3	NaN	NaN
										5	K2	K0	NaN	NaN	СЗ	D3

```
In [47]: result = pd.merge(left, right, how='inner', on=['key1', 'key2'])
```

		left					right						Result			
	key1	key2	Α	В		key1	key2	С	D		lmv1	low/2	۸	В	C	D
0	K0	K0	A0	В0	0	K0	K0	α	D0		key1	key2	A		-	
1	КО	К1	A1	B1	1	K1	КО	C1	D1	0	K0	K0	A0	В0	00	D0
2	К1	KO	A2	B2	2	K1	KO	(2	D2	1	K1	K0	A2	B2	C1	D1
	-			\vdash		_	-	_	\vdash	2	K1	K0	A2	B2	C2	D2
3	K2	K1	A3	B3	3	K2	K0	C3	D3							

Here is another example with duplicate join keys in DataFrames:

```
In [48]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})
In [49]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})
In [50]: result = pd.merge(left, right, on='B', how='outer')
```



Warning: Joining / merging on duplicate keys can cause a returned frame that is the multiplication of the row dimensions, which may result in memory overflow. It is the user's responsibility to manage duplicate values in keys before joining large DataFrames.

Checking for duplicate keys

New in version 0.21.0.

Users can use the <code>validate</code> argument to automatically check whether there are unexpected duplicates in their merge keys. Key uniqueness is checked before merge operations and so should protect against memory overflows. Checking key uniqueness is also a good way to ensure user data structures are as expected.

In the following example, there are duplicate values of B in the right DataFrame. As this is not a one-to-one merge — as specified in the validate argument — an exception will be raised.

```
In [51]: left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})
In [52]: right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2, 2]})
```

```
In [53]: result = pd.merge(left, right, on='B', how='outer', validate="one_to_one")
...
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right <code>DataFrame</code> but wants to ensure there are no duplicates in the left <code>DataFrame</code>, one can use the <code>validate='one_to_many'</code> argument instead, which will not raise an exception.

```
In [53]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[53]:
    A_x B A_y
0 1 1 NaN
```

```
1 2 2 4.0
2 2 2 5.0
3 2 2 6.0
```

The merge indicator

merge() accepts the argument indicator. If True, a Categorical-type column called _merge will be added to the output object that takes on values:

Observation Origin	_merge value
Merge key only in 'left' frame	left_only
Merge key only in 'right' frame	right_only
Merge key in both frames	both

```
In [54]: df1 = pd.DataFrame({'col1': [0, 1], 'col left': ['a', 'b']})
In [55]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col right': [2, 2, 2]})
In [56]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[56]:
  col1 col_left col_right
                                 merge
0
                            left only
                       NaN
             а
              b
                       2.0
                                  both
1
      1
2
      2
                       2.0 right_only
             NaN
3
      2
             NaN
                       2.0 right_only
```

The indicator argument will also accept string arguments, in which case the indicator function will use the value of the passed string as the name for the indicator column.

```
In [57]: pd.merge(df1, df2, on='coll', how='outer', indicator='indicator column')
Out[57]:
   col1 col_left col_right indicator column
0
                        NaN
                                  left only
              a
1
      1
              b
                        2.0
                                        both
2
      2
             NaN
                        2.0
                                  right only
3
      2
             NaN
                        2.0
                                  right only
```

Merge Dtypes

New in version 0.19.0.

Merging will preserve the dtype of the join keys.

```
In [58]: left = pd.DataFrame({'key': [1], 'v1': [10]})
In [59]: left
Out[59]:
    key v1
0     1     10

In [60]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})
In [61]: right
Out[61]:
    key v1
```

```
0 1 20
1 2 30
```

We are able to preserve the join keys:

```
In [62]: pd.merge(left, right, how='outer')
Out[62]:
   key v1
0
       10
    1
       20
1
     1
     2 30
In [63]: pd.merge(left, right, how='outer').dtypes
Out[63]:
      int64
key
v_1
      int64
dtype: object
```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.

```
In [64]: pd.merge(left, right, how='outer', on='key')
Out[64]:
    key v1_x v1_y
0    1 10.0    20
1    2 NaN    30

In [65]: pd.merge(left, right, how='outer', on='key').dtypes
Out[65]:
key    int64
v1_x    float64
v1_y    int64
dtype: object
```

New in version 0.20.0.

Merging will preserve category dtypes of the mergands. See also the section on categoricals.

The left frame.

```
In [66]: from pandas.api.types import CategoricalDtype
In [67]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))
In [68]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))
In [69]: left = pd.DataFrame({'X': X,
                               'Y': np.random.choice(['one', 'two', 'three'],
                                                     size=(10,))})
   . . . . :
   . . . . :
In [70]: left
Out[70]:
    X
            Υ
0 bar
          one
1 foo
          one
2 foo three
3 bar
       three
  foo
         one
5
  bar
          one
6
  bar
       three
   bar
       three
```

```
8 bar three
9 foo three

In [71]: left.dtypes
Out[71]:
X category
Y object
dtype: object
```

The right frame.

```
In [72]: right = pd.DataFrame({'X': pd.Series(['foo', 'bar'],
                                                dtype=CategoricalDtype(['foo', 'bar'])
                               'Z': [1, 2]})
   • • • • •
   . . . . :
In [73]: right
Out[73]:
     X Z
  foo 1
0
1 bar 2
In [74]: right.dtypes
Out[74]:
X
   category
7.
       int64
dtype: object
```

The merged result:

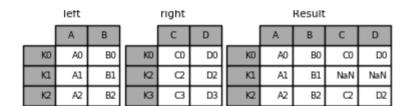
```
In [75]: result = pd.merge(left, right, how='outer')
In [76]: result
Out[76]:
    Х
          Y Z
0
  bar
        one 2
1 bar three 2
2 bar
       one 2
3
  bar three 2
4
  bar three 2
5
  bar three 2
6 foo
       one 1
7 foo three 1
8 foo
       one 1
9 foo three 1
In [77]: result.dtypes
Out[77]:
Χ
   category
Υ
     object
7.
      int64
dtype: object
```

Note: The category dtypes must be *exactly* the same, meaning the same categories and the ordered attribute. Otherwise the result will coerce to object dtype.

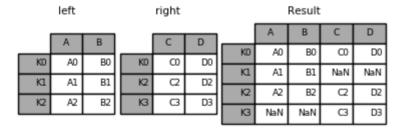
Note: Merging on category dtypes that are the same can be quite performant compared to object dtype merging.

Joining on index

DataFrame.join() is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Here is a very basic example:

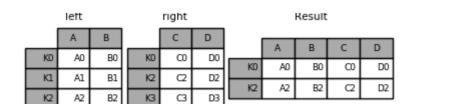


```
In [81]: result = left.join(right, how='outer')
```



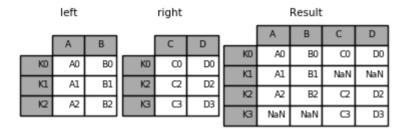
The same as above, but with how='inner'.

```
In [82]: result = left.join(right, how='inner')
```



The data alignment here is on the indexes (row labels). This same behavior can be achieved using merge plus additional arguments instructing it to use the indexes:

```
In [83]: result = pd.merge(left, right, left_index=True, right_index=True, how='oute
```



```
In [84]: result = pd.merge(left, right, left_index=True, right_index=True, how='inne
```

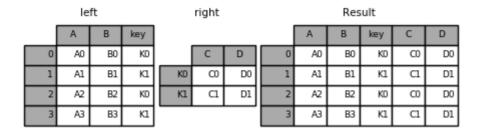
		left			right					Kesult		
		Α	В		С	D		ı	Α	В	С	D
١	KO	A0	B0	KO	α	D0	_ا	_	^	ь	·	D
ı	_	-	\vdash		-		Ш	K0	A0	B0	00	D0
ı	K1	Al	B1	K2	C2	D2	l۲	K2	A2	B2	(2	D2
ı	K2	A2	B2	КЗ	СЗ	D3	IL	1/2	HZ.	DZ	- 42	D2
ı							I –					

Joining key columns on an index

join() takes an optional on argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
pd.merge(left, right, left_on=key_or_keys, right_index=True,
    how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the DataFrame's is already indexed by the join key), using join may be more convenient. Here is a simple example:



	le	ft			right				Res	sult		
	Α	В	key					Α	В	key	С	D
0	A0	BO	KO		С	D	0	A0	B0	K0	ω	D0
1	A1	B1	K1	KO	ω	D0	1	A1	B1	K1	CI	D1
2	A2	B2	K0	Kl	Cl	D1	2	A2	B2	K0	α	D0
3	A3	В3	K1				3	A3	В3	K1	CI	D1

To join on multiple keys, the passed DataFrame must have a MultiIndex:

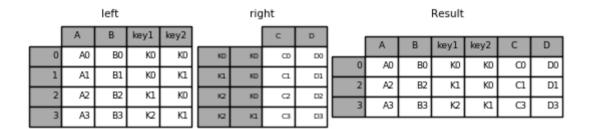
Now this can be joined by passing the two key column names:

```
In [92]: result = left.join(right, on=['key1', 'key2'])
```

		leit				rig	JIIL					Result			
	Α	В	key1	key2			С	D		Α	В	key1	key2	С	D
0	A0	BO	K0	KO	KD	KD	8	D0	0	A0	В0	K0	K0	О	D0
1	A1	B1	K0	K1	KI	KD	Д	D1	1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	K2	KD	Ŋ	D2	2	A2	B2	K1	K0	C1	D1
3	A3	В3	K2	K1	K2	кі	ß	D3	3	A3	В3	K2	K1	СЗ	D3

The default for <code>DataFrame.join</code> is to perform a left join (essentially a "VLOOKUP" operation, for Excel users), which uses only the keys found in the calling <code>DataFrame</code>. Other join types, for example inner join, can be just as easily performed:

```
In [93]: result = left.join(right, on=['key1', 'key2'], how='inner')
```



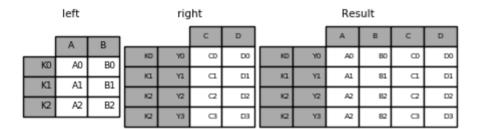
As you can see, this drops any rows where there was no match.

Joining a single Index to a MultiIndex

You can join a singly-indexed DataFrame with a level of a MultiIndexed DataFrame. The level will match on the name of the index of the singly-indexed frame against a level name of the MultiIndexed frame.

	left			rig	ht					Res	ult		
1	Α	В			С	D				А	В	С	D
VO.			KD	YO	8	D0		KD	YO	AD	B0	8	D0
K0	A0	В0	K1	Y1	а	D1		K1	Y1	Al	B1	а	D1
K1	Al	B1	K2	Y2	a	D2	Н	K2	Y2	A2	B2	a	D2
K2	A2	B2	P-2	12		Liz		74	12	A2.	DA.		Liz
K2 A2 B2			K2	Y3	З	D3		K2	Y3	A2	B2	З	D3

This is equivalent but less verbose and more memory efficient / faster than this.



Joining with two MultiIndexes

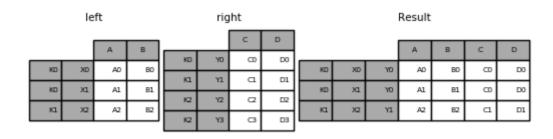
This is supported in a limited way, provided that the index for the right argument is completely used in the join, and is a subset of the indices in the left argument, as in this example:

```
In [99]: leftindex = pd.MultiIndex.from_product([list('abc'), list('xy'), [1, 2]],
                                                   names=['abc', 'xy', 'num'])
   . . . . :
   . . . . :
In [100]: left = pd.DataFrame({'v1': range(12)}, index=leftindex)
In [101]: left
Out[101]:
            v_1
abc xy num
             0
    X
       1
       2
             1
             2
    У
       1
       2
b
    X
       1
       2
             5
    У
       1
             6
       2
             7
    Х
       1
             8
             9
       2
            10
       1
    У
            11
In [102]: rightindex = pd.MultiIndex.from product([list('abc'), list('xy')],
                                                     names=['abc', 'xy'])
   . . . . . :
In [103]: right = pd.DataFrame({'v2': [100 * i for i in range(1, 7)]}, index=rightin
In [104]: right
```

```
Out[104]:
         v2
abc xy
   Х
        100
а
        200
    У
h
        300
   х
        400
    У
        500
С
    х
        600
    У
In [105]: left.join(right, on=['abc', 'xy'], how='inner')
Out[105]:
            v1
                 v2
abc xy num
             0 100
  x 1
       2
             1 100
      1
             2 200
    У
       2
             3 200
             4 300
h
      1
    х
       2
             5 300
             6 400
      1
    У
             7 400
       2
             8 500
С
    х
      1
            9 500
       2
            10 600
    У
      1
       2
            11 600
```

If that condition is not satisfied, a join with two multi-indexes can be done using the following code.

```
names=['key', 'X'])
 . . . . . :
. . . . . :
                 index=leftindex)
 . . . . . :
names=['key', 'Y'])
 . . . . . :
index=rightindex)
 . . . . . :
 . . . . . :
In [110]: result = pd.merge(left.reset index(), right.reset index(),
               on=['key'], how='inner').set_index(['key', 'X', 'Y'])
 . . . . . :
 . . . . . :
```



Merging on a combination of columns and index levels

New in version 0.23.

Strings passed as the on, left_on, and right_on parameters may refer to either column names or index level names. This enables merging DataFrame instances on a combination of index levels and columns without resetting indexes.

```
In [111]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')
In [112]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                                  'B': ['B0', 'B1', 'B2', 'B3'],
                                 'key2': ['K0', 'K1', 'K0', 'K1']},
   . . . . . :
                                index=left index)
   . . . . . :
   . . . . . :
In [113]: right index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')
In [114]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                                   'D': ['D0', 'D1', 'D2', 'D3'],
                                   'key2': ['K0', 'K0', 'K0', 'K1']},
   . . . . . :
                                 index=right index)
   . . . . . :
   . . . . . :
In [115]: result = left.merge(right, on=['key1', 'key2'])
```

		le	ft			rig	ht				Res	sult		
		Α	В	key2		С	D	key2		Α	В	key2	С	D
	KO	A0	В0	K0	KO	ω	D0	K0	V0	- 11		-		
ı	KO	Al	B1	K1	K1	Cl	D1	КО	KO	A0	BO	K0	00	D0
ł	K1	A2	B2	КО	K2	(2	D2	КО	K1	A2	B2	K0	C1	D1
ł	K2	A3	В3	K1	K2	_	D3	K1	K2	A3	В3	K1	СЗ	D3
ı	Ν2	A3	- 63	VT.	N2	L (3)	D3	71	_					

Note: When DataFrames are merged on a string that matches an index level in both frames, the index level is preserved as an index level in the resulting DataFrame.

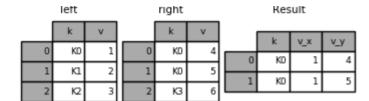
Note: When DataFrames are merged using only some of the levels of a *MultiIndex*, the extra levels will be dropped from the resulting merge. In order to preserve those levels, use reset_index on those level names to move those levels to columns prior to doing the merge.

Note: If a string matches both a column name and an index level name, then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

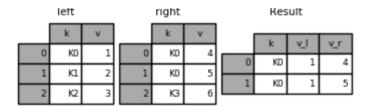
Overlapping value columns

The merge suffixes argument takes a tuple of list of strings to append to overlapping column names in the input DataFrames to disambiguate the result columns:

```
In [116]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})
In [117]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})
In [118]: result = pd.merge(left, right, on='k')
```



```
In [119]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```



DataFrame.join() has lsuffix and rsuffix arguments which behave similarly.

```
In [120]: left = left.set_index('k')
In [121]: right = right.set_index('k')
In [122]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```

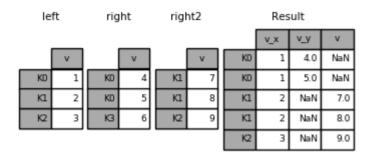
```
left right Result
```

left		rig	ht	Result				
	V				v_l	v_r		
L/D	, ·	140	V 4	KO	1	4.0		
KO	1	K0		KO	1	5.0		
K1	2	K0	5	K1	2	NaN		
K2	3	КЗ	6	K2	3	NaN		
				NZ	3	Nan		

Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to join() to join them together on their indexes.

```
In [123]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])
In [124]: result = left.join([right, right2])
```

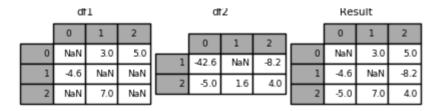


Merging together values within Series or DataFrame columns

Another fairly common situation is to have two like-indexed (or similarly indexed) Series or DataFrame objects and wanting to "patch" values in one object from values for matching indices in the other. Here is an example:

For this, use the **combine_first()** method:

```
In [127]: result = df1.combine_first(df2)
```



Note that this method only takes values from the right <code>DataFrame</code> if they are missing in the left <code>DataFrame</code>. A related method, <code>update()</code>, alters non-NA values in place:

```
In [128]: df1.update(df2)
```

	dt1			dt2				Result			
	0	1	2		0	1	2		0	1	2
0	NaN	3.0	5.0	1	-42.6	NaN	-8.2	0	NaN	3.0	5.0
1	-4.6	NaN	NaN	2	-	-	-	1	-42.6	NaN	-8.2
2	NaN	7.0	NaN		-5.0	1.6	4.0	2	-5.0	1.6	4.0

Timeseries friendly merging

Merging Ordered Data

A merge_ordered() function allows combining time series and other ordered data. In particular it has an optional fill method keyword to fill/interpolate missing data:

```
In [129]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
                            'lv': [1, 2, 3, 4],
                            's': ['a', 'b', 'c', 'd']})
  . . . . . :
In [130]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
                             'rv': [1, 2, 3]})
In [131]: pd.merge ordered(left, right, fill method='ffill', left by='s')
Out[131]:
      lv s
0
  K0 1.0 a NaN
1 K1 1.0 a 1.0
2 K2 1.0 a 2.0
3 K4 1.0 a 3.0
4 K1 2.0 b 1.0
5
  K2 2.0 b 2.0
6
  K4 2.0 b 3.0
7 K1 3.0 c 1.0
8 K2 3.0 c 2.0
9 K4 3.0 c 3.0
10 K1 NaN d 1.0
11 K2 4.0 d 2.0
12 K4 4.0 d 3.0
```

Merging AsOf

New in version 0.19.0.

A merge_asof() is similar to an ordered left-join except that we match on nearest key rather than equal keys. For each row in the left DataFrame, we select the last row in the right DataFrame whose on key is less than the left's key. Both DataFrames must be sorted by the key.

Optionally an asof merge can perform a group-wise merge. This matches the by key equally, in addition to the nearest match on the on key.

For example; we might have trades and quotes and we want to asof merge them.

```
'20160525 13:30:00.048',
   . . . . . :
                                              '20160525 13:30:00.048'
   . . . . . :
                                              '20160525 13:30:00.048']),
    . . . . . :
                 'ticker': ['MSFT', 'MSFT',
   . . . . . :
                              'GOOG', 'GOOG',
                                                'AAPL'],
   . . . . . :
                 'price': [51.95, 51.95,
   . . . . . :
                            720.77, 720.92, 98.00],
   . . . . . :
                 'quantity': [75, 155,
   . . . . . :
                                100, 100, 100]},
   . . . . . :
                columns=['time', 'ticker', 'price', 'quantity'])
   . . . . . :
   . . . . . :
In [133]: quotes = pd.DataFrame({
                'time': pd.to datetime(['20160525 13:30:00.023',
   . . . . . :
                                              '20160525 13:30:00.023',
   . . . . . :
                                              '20160525 13:30:00.030',
   . . . . . :
                                              '20160525 13:30:00.041',
   . . . . . :
                                              '20160525 13:30:00.048',
   . . . . . :
                                              '20160525 13:30:00.049',
   . . . . . :
                                              '20160525 13:30:00.072',
   . . . . . :
                                              '20160525 13:30:00.075']),
   . . . . . :
                'ticker': ['GOOG', 'MSFT', 'MSFT',
   . . . . . :
                              'MSFT', 'GOOG', 'AAPL', 'GOOG',
   . . . . . :
                              'MSFT'],
   . . . . . :
                 'bid': [720.50, 51.95, 51.97, 51.99,
   . . . . . :
                          720.50, 97.99, 720.50, 52.01],
   . . . . . :
                 'ask': [720.93, 51.96, 51.98, 52.00,
   . . . . . :
                          720.93, 98.01, 720.88, 52.03]},
   . . . . . :
                columns=['time', 'ticker', 'bid', 'ask'])
   . . . . . :
    . . . . . :
```

```
In [134]: trades
Out[134]:
                    time ticker
                                  price quantity
                                           75
0 2016-05-25 13:30:00.023 MSFT
                                  51.95
                                  51.95
1 2016-05-25 13:30:00.038
                           MSFT
                                              155
                           GOOG 720.77
2 2016-05-25 13:30:00.048
                                              100
                           GOOG 720.92
3 2016-05-25 13:30:00.048
                                              100
4 2016-05-25 13:30:00.048
                           AAPL
                                  98.00
                                              100
In [135]: quotes
Out[135]:
                    time ticker
                                    bid
0 2016-05-25 13:30:00.023
                                 720.50
                          GOOG
                                        720.93
1 2016-05-25 13:30:00.023
                           MSFT
                                  51.95
                                          51.96
                                 51.97
2 2016-05-25 13:30:00.030
                           MSFT
                                          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
```

By default we are taking the asof of the quotes.

```
In [136]: pd.merge asof(trades, quotes,
                         on='time',
   . . . . . :
                         by='ticker')
   . . . . . :
   . . . . . :
Out[136]:
                      time ticker
                                     price quantity
                                                        bid
                                                                  ask
0 2016-05-25 13:30:00.023
                             MSFT
                                     51.95
                                                        51.95
                                                                51.96
                                                  75
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.

```
In [137]: pd.merge_asof(trades, quotes,
                       on='time',
                       by='ticker',
   . . . . . :
                        tolerance=pd.Timedelta('2ms'))
   . . . . . :
Out[137]:
                     time ticker
                                   price quantity
                                                       bid
                                                               ask
0 2016-05-25 13:30:00.023 MSFT
                                   51.95
                                          75
                                                     51.95
                                                             51.96
                                  51.95
1 2016-05-25 13:30:00.038
                                               155
                           MSFT
                                                      NaN
                                                              NaN
                           GOOG 720.77
2 2016-05-25 13:30:00.048
                                              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. Note that though we exclude the exact matches (of the quotes), prior quotes **do** propagate to that point in time.

```
In [138]: pd.merge asof(trades, quotes,
                        on='time',
                        by='ticker',
   . . . . . :
                        tolerance=pd.Timedelta('10ms'),
   . . . . . :
                        allow exact matches=False)
   . . . . . :
   . . . . . :
Out[138]:
                     time ticker
                                  price quantity
                                                      bid
                                                             ask
0 2016-05-25 13:30:00.023 MSFT
                                 51.95
                                               75
                                                      NaN
                                                             NaN
                                               155 51.97 51.98
1 2016-05-25 13:30:00.038 MSFT 51.95
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
4 2016-05-25 13:30:00.048 AAPL
                                  98.00
                                               100
                                                      NaN
                                                             NaN
```