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:

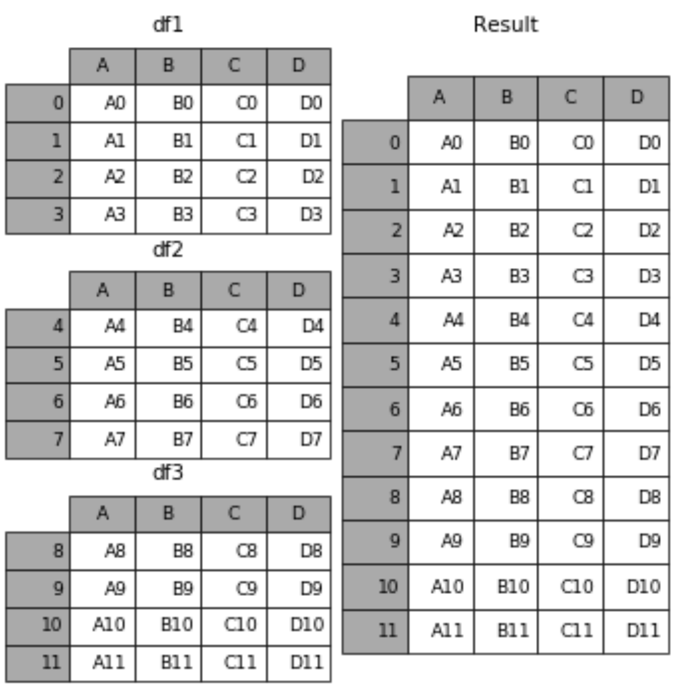
```
In [1]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ...:                     'B': ['B0', 'B1', 'B2', 'B3'],
   ...:                     'C': ['C0', 'C1', 'C2', 'C3'],
   ...:                     'D': ['D0', 'D1', 'D2', 'D3']},
   ...:                     index=[0, 1, 2, 3])
   ...:

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

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

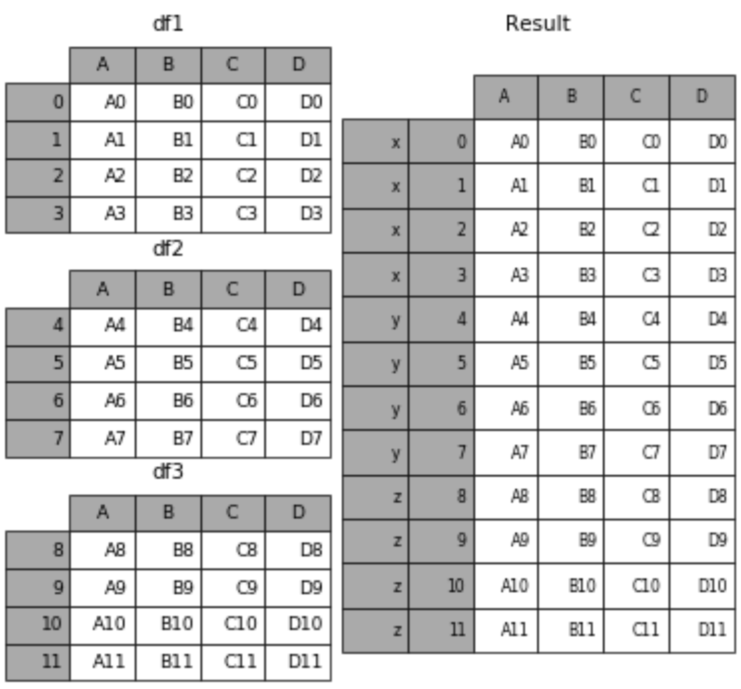
Like its sibling function on ndarrays, numpy.concatenate, pandas.concat takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of “what to do with the other axes”:

```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

- 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 and example many of these arguments don't make much sense. Let's take 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

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	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 do stuff like select out each chunk by key:

```
In [7]: result.loc['y']
Out[7]: 
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

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

Note: It is worth noting however, 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 (or Panels or…), for example, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in three ways:

- Take the (sorted) union of them all, join='outer'. This is the default option as it results in zero information loss.
- Take the intersection, join='inner'.
- Use a specific index (in the case of DataFrame) or indexes (in the case of Panel or future higher dimensional objects), i.e. the join_axes argument

Here is a example of each of these methods. First, the default join='outer' behavior:

```
In [8]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
   ...:                     'D': ['D2', 'D3', 'D6', 'D7'],
   ...:                     'F': ['F2', 'F3', 'F6', 'F7']},
   ...:                    index=[2, 3, 6, 7])
   ...:

In [9]: result = pd.concat([df1, df4], axis=1)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Note that the row indexes have been unioned and sorted. Here is the same thing with join='inner':

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

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

```
In [11]: result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Concatenating using append

A useful shortcut to concat are the append instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along axis=0, namely the index:

```
In [12]: result = df1.append(df2)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

append may take multiple objects to concatenate:

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Note: Unlike *list.append* method, which appends to the original list and returns nothing, append here **does not** modify df1 and returns its copy with df2 appended.

Ignoring indexes on the concatenation axis

For DataFrames 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)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

This is also a valid argument to DataFrame.append:

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Concatenating with mixed ndims

You can concatenate a mix of Series and DataFrames. The Series will be transformed to DataFrames 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

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

Result

	A	B	C	D	X
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

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)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

0	_0
1	_1
2	_2
3	_3

Result

	A	B	C	D	0	1	2
0	A0	B0	C0	D0	_0	_0	_0
1	A1	B1	C1	D1	_1	_1	_1
2	A2	B2	C2	D2	_2	_2	_2
3	A3	B3	C3	D3	_3	_3	_3

Passing ignore_index=True will drop all name references.

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

Result

	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

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 inherits 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  5
```

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       0
1    1     1       1
2    2     2       4
3    3     3       5
```

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

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

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

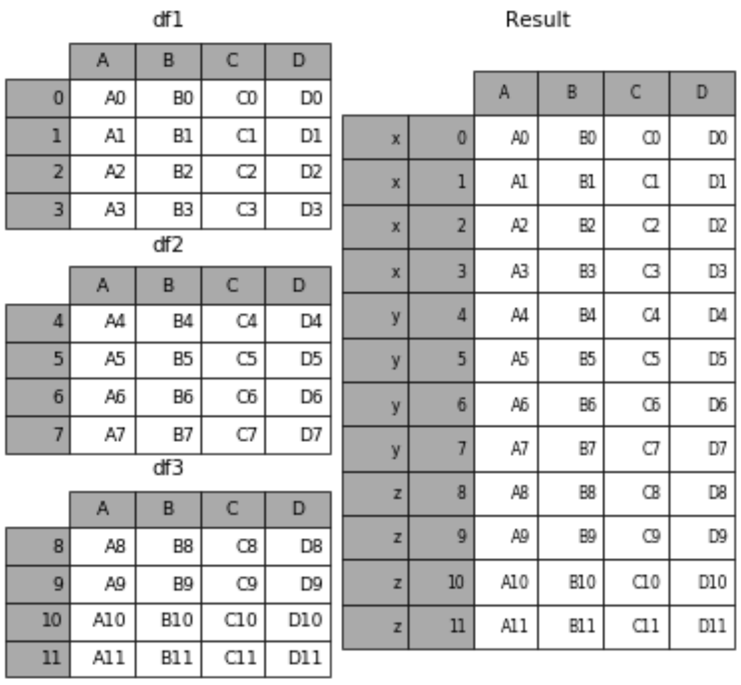
Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	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

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

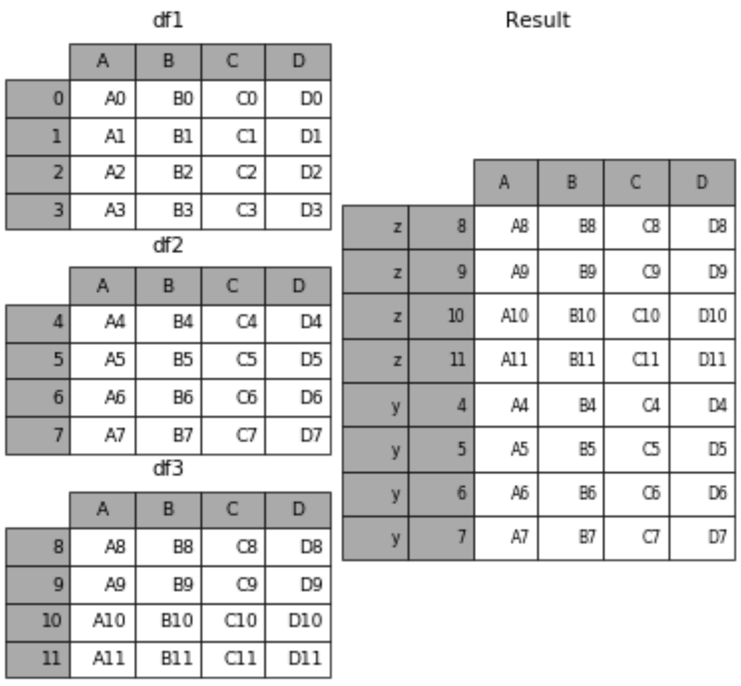
df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

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



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7

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 levels argument:

```
In [32]: result = pd.concat(pieces, keys=['x', 'y', 'z'],
   ....:                    levels=[['z', 'y', 'x', 'w']],
   ....:                    names=['group_key'])
   ....:
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	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]])
```

Yes, this is fairly esoteric, but 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)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

A	X0
B	X1
C	X2
D	X3

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	X0	X1	X2	X3

You should use ignore_index 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:

```
In [36]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
   ....:          {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]
   ....:

In [37]: result = df1.append(dicts, ignore_index=True)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

dicts

	A	B	C	X	Y
0	1	2	3	4.0	NaN
1	5	6	7	NaN	8.0

Result

	A	B	C	D	X	Y
0	A0	B0	C0	D0	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN
4	1	2	3	NaN	4.0	NaN
5	5	6	7	NaN	NaN	8.0

Database-style DataFrame 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 base::merge.data.frame in R). The reason for this is careful algorithmic design and internal layout of the data in DataFrame.

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 objects:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False)
```

- left: A DataFrame object
- right: Another DataFrame object
- on: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and left_index and right_index are False, the intersection of the columns in the DataFrames will be inferred to be the join keys
- left_on: Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- right_on: Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- left_index: If True, use the index (row labels) from the left DataFrame as its join key(s). In the case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame
- right_index: Same usage as left_index for the right DataFrame
- 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 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, right_only for observations whose merge key only appears in 'right' DataFrame, and both if the observation's merge key is found in both.

New in version 0.17.0.

The return type will be the same as left. If left is a DataFrame 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, with the calling DataFrame being implicitly considered the left object in the join.

The related DataFrame.join method, uses merge 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 DataFrame.join 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 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 DataFrame 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:

```
In [38]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:

In [39]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [40]: result = pd.merge(left, right, on='key')
```

left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

right

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Here is a more complicated example with multiple join keys:

```
In [41]: left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:

In [42]: right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
   ....:                       'key2': ['K0', 'K0', 'K0', 'K0'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [43]: result = pd.merge(left, right, on=['key1', 'key2'])
```

left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2

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

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN

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

left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2
3	NaN	NaN	K2	K0	C3	D3

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

left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

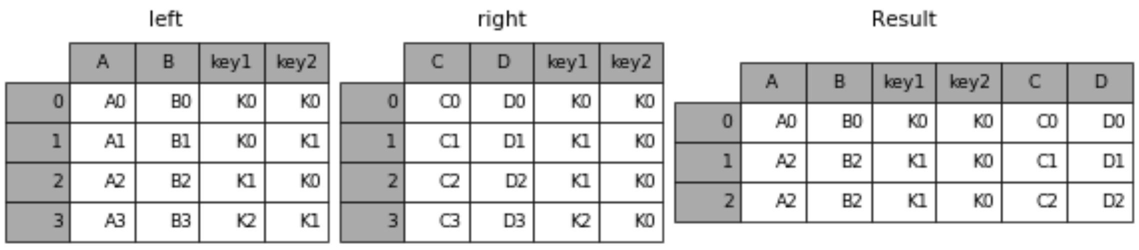
right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN
5	NaN	NaN	K2	K0	C3	D3

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

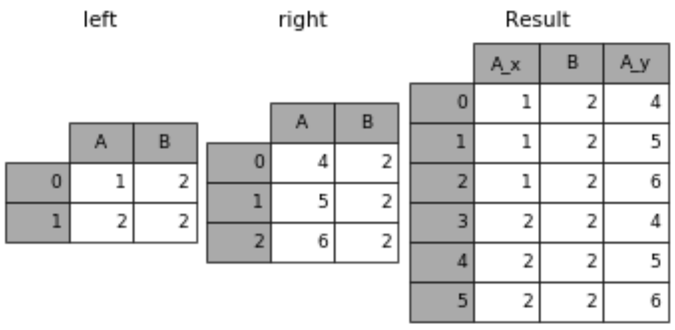


Here is another example with duplicate join keys in DataFrames:

```
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, may result in memory overflow. It is the user' s responsibility to manage duplicate values in keys before joining large DataFrames.

The merge indicator

New in version 0.17.0.

merge now 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 [51]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left':['a', 'b']})

In [52]: df2 = pd.DataFrame({'col1': [1, 2, 2],'col_right':[2, 2, 2]})

In [53]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[53]: 
   col1 col_left  col_right      _merge
0     0        a        NaN   left_only
1     1        b        2.0        both
2     2      NaN        2.0  right_only
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 [54]: pd.merge(df1, df2, on='col1', how='outer', indicator='indicator_column')
Out[54]: 
   col1 col_left  col_right indicator_column
0     0        a        NaN        left_only
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 [55]: left = pd.DataFrame({'key': [1], 'v1': [10]})

In [56]: left
Out[56]: 
   key  v1
0    1  10

In [57]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [58]: right
Out[58]: 
   key  v1
0    1  20
1    2  30
```

We are able to preserve the join keys

```
In [59]: pd.merge(left, right, how='outer')
Out[59]: 
   key  v1
0    1  10
1    1  20
2    2  30

In [60]: pd.merge(left, right, how='outer').dtypes
Out[60]: 
key    int64
v1     int64
dtype: object
```

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

```
In [61]: pd.merge(left, right, how='outer', on='key')
Out[61]: 
   key  v1_x  v1_y
0    1  10.0    20
1    2   NaN    30

In [62]: pd.merge(left, right, how='outer', on='key').dtypes
Out[62]: 
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 [63]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [64]: X = X.astype('category', categories=['foo', 'bar'])

In [65]: left = pd.DataFrame({'X': X,
   ....:                      'Y': np.random.choice(['one', 'two', 'three'], size=(10,))})
   ....:

In [66]: left
Out[66]: 
     X      Y
0  bar    one
1  foo    one
2  foo  three
3  bar  three
4  foo    one
5  bar    one
6  bar  three
7  bar  three
8  bar  three
9  foo  three

In [67]: left.dtypes
Out[67]: 
X    category
Y      object
dtype: object
```

The right frame.

```
In [68]: right = pd.DataFrame({'X': pd.Series(['foo', 'bar']).astype('category', categories=['foo', 'bar']),
   ....:                       'Z': [1, 2]})
   ....:

In [69]: right
Out[69]: 
     X  Z
0  foo  1
1  bar  2

In [70]: right.dtypes
Out[70]: 
X    category
Z       int64
dtype: object
```

The merged result

```
In [71]: result = pd.merge(left, right, how='outer')

In [72]: result
Out[72]: 
     X      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 [73]: result.dtypes
Out[73]: 
X    category
Y      object
Z       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 [74]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                      index=['K0', 'K1', 'K2'])
   ....:

In [75]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D2', 'D3']},
   ....:                       index=['K0', 'K2', 'K3'])
   ....:

In [76]: result = left.join(right)
```

left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

```
In [77]: result = left.join(right, how='outer')
```

left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

```
In [78]: result = left.join(right, how='inner')
```

left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

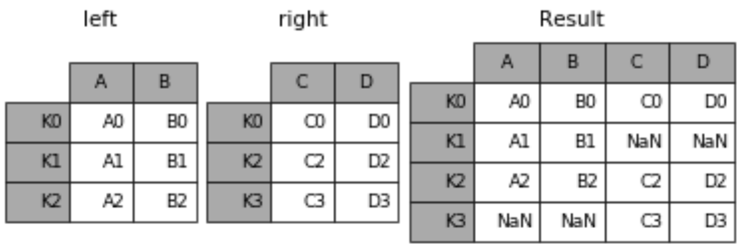
	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

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 [79]: result = pd.merge(left, right, left_index=True, right_index=True, how='outer')
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

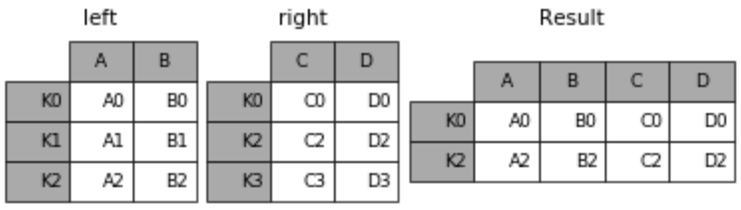
right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

```
In [80]: result = pd.merge(left, right, left_index=True, right_index=True, how='inner');
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

Result

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

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:

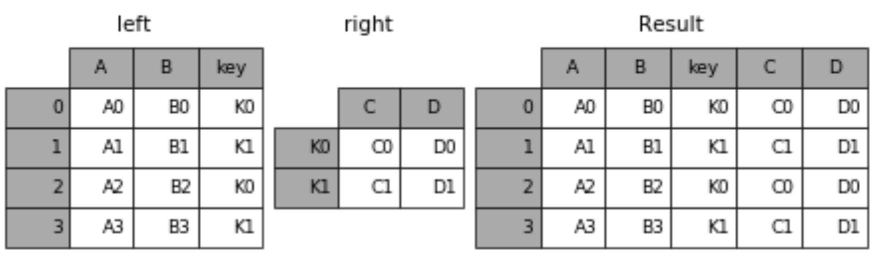
```
left.join(right, on=key_or_keys)
pd.merge(left, right, left_on=key_or_keys, right_index=True,
     how='left', sort=False)
```

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:

```
In [81]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [82]: right = pd.DataFrame({'C': ['C0', 'C1'],
   ....:                       'D': ['D0', 'D1']},
   ....:                       index=['K0', 'K1'])
   ....:

In [83]: result = left.join(right, on='key')
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

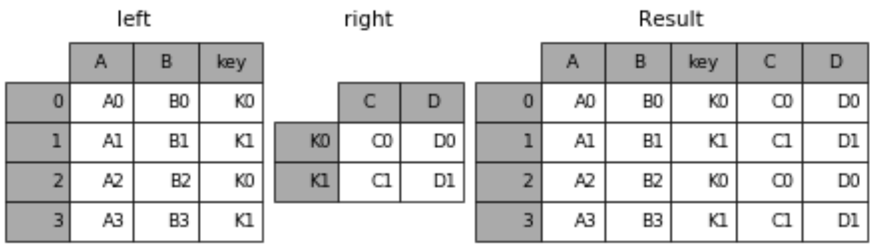
right

	C	D
K0	C0	D0
K1	C1	D1

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

```
In [84]: result = pd.merge(left, right, left_on='key', right_index=True,
   ....:                   how='left', sort=False);
   ....:
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

right

	C	D
K0	C0	D0
K1	C1	D1

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

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

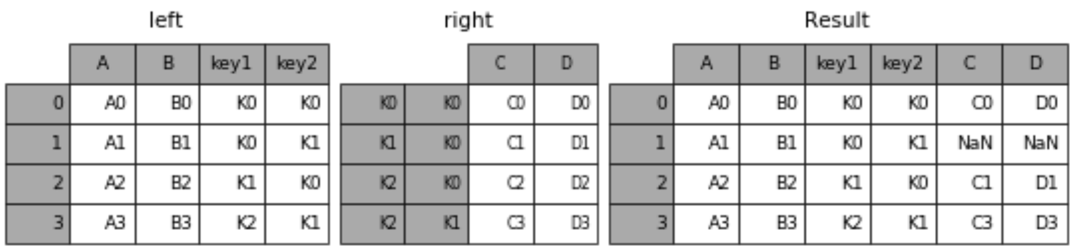
```
In [85]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [86]: index = pd.MultiIndex.from_tuples([('K0', 'K0'), ('K1', 'K0'),
   ....:                                   ('K2', 'K0'), ('K2', 'K1')])
   ....:

In [87]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                      index=index)
   ....:
```

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

```
In [88]: result = left.join(right, on=['key1', 'key2'])
```



left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

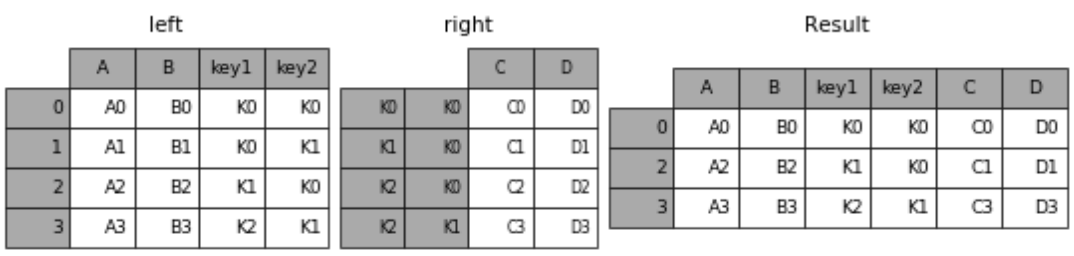
		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A3	B3	K2	K1	C3	D3

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

```
In [89]: result = left.join(right, on=['key1', 'key2'], how='inner')
```



left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
2	A2	B2	K1	K0	C1	D1
3	A3	B3	K2	K1	C3	D3

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

Joining a single Index to a Multi-index

New in version 0.14.0.

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

```
In [90]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                      index=pd.Index(['K0', 'K1', 'K2'], name='key'))
   ....:

In [91]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   ....:                                   ('K2', 'Y2'), ('K2', 'Y3')],
   ....:                                    names=['key', 'Y'])
   ....:

In [92]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                       index=index)
   ....:

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

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

```
In [94]: result = pd.merge(left.reset_index(), right.reset_index(),
   ....:       on=['key'], how='inner').set_index(['key','Y'])
   ....:
```

Joining with two multi-indexes

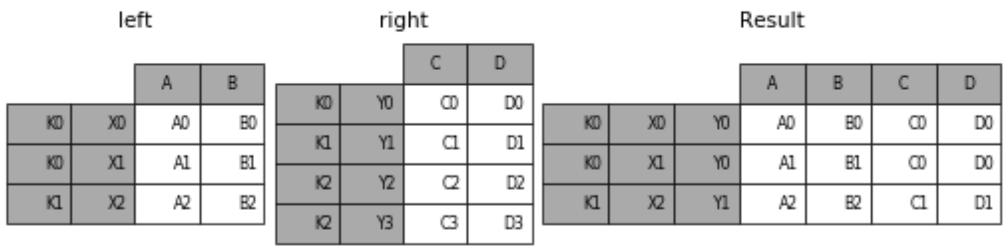
This is not Implemented via join at-the-moment, however it can be done using the following.

```
In [95]: index = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
   ....:                                    ('K1', 'X2')],
   ....:                                     names=['key', 'X'])
   ....:

In [96]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
```

```
   ....:                     index=index)
   ....:

In [97]: result = pd.merge(left.reset_index(), right.reset_index(),
   ....:                   on=['key'], how='inner').set_index(['key','X','Y'])
   ....:
```



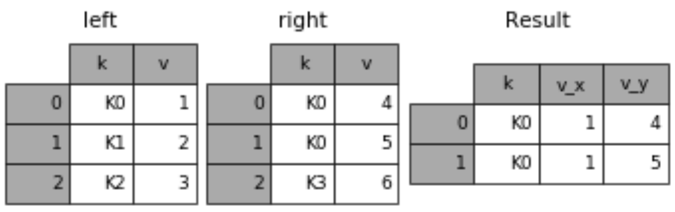
Overlapping value columns

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:

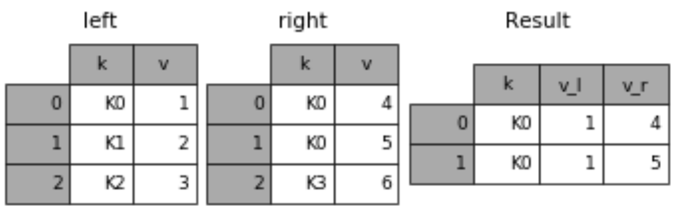
```
In [98]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})

In [99]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})

In [100]: result = pd.merge(left, right, on='k')
```




```
In [101]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```



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

```
In [102]: left = left.set_index('k')

In [103]: right = right.set_index('k')

In [104]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```

left

	v
K0	1
K1	2
K2	3

right

	v
K0	4
K0	5
K3	6

Result

	v_l	v_r
K0	1	4.0
K0	1	5.0
K1	2	NaN
K2	3	NaN

Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to DataFrame.join to join them together on their indexes. The same is true for Panel.join.

```
In [105]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])

In [106]: result = left.join([right, right2])
```

left

	v
K0	1
K1	2
K2	3

right

	v
K0	4
K0	5
K3	6

right2

	v
K1	7
K1	8
K2	9

Result

	v_x	v_y	v
K0	1.0	4.0	NaN
K0	1.0	5.0	NaN
K1	2.0	NaN	7.0
K1	2.0	NaN	8.0
K2	3.0	NaN	9.0
K3	NaN	6.0	NaN

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:

```
In [107]: df1 = pd.DataFrame([[np.nan, 3., 5.], [-4.6, np.nan, np.nan],
   .....:                     [np.nan, 7., np.nan]])
   .....:

In [108]: df2 = pd.DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4]],
   .....:                     index=[1, 2])
   .....:
```

For this, use the combine_first method:

```
In [109]: result = df1.combine_first(df2)
```

df1

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	NaN
2	NaN	7.0	NaN

df2

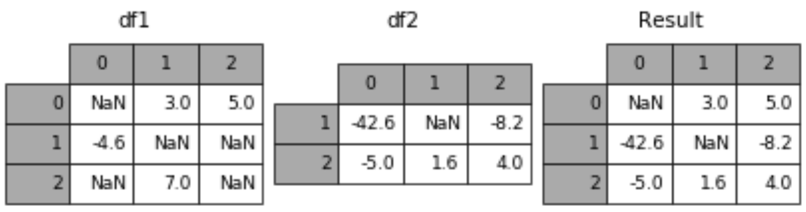
	0	1	2
1	-42.6	NaN	-8.2
2	-5.0	1.6	4.0

Result

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	-8.2
2	-5.0	7.0	4.0

Note that this method only takes values from the right DataFrame if they are missing in the left DataFrame. A related method, update, alters non-NA values inplace:

```
In [110]: df1.update(df2)
```



df1

	0	1	2
0	NaN	3.0	5.0
1	-4.6	NaN	NaN
2	NaN	7.0	NaN

df2

	0	1	2
1	-42.6	NaN	-8.2
2	-5.0	1.6	4.0

Result

	0	1	2
0	NaN	3.0	5.0
1	-42.6	NaN	-8.2
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 [111]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
   .....:                      'lv': [1, 2, 3, 4],
   .....:                      's': ['a', 'b', 'c', 'd']})
   .....:

In [112]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
   .....:                       'rv': [1, 2, 3]})
   .....:

In [113]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')
Out[113]:
     k   lv  s   rv
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.

```
In [114]: trades = pd.DataFrame({
   .....:     'time': pd.to_datetime(['20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.038',
   .....:                             '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 [115]: 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 [116]: trades
Out[116]: 
                     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

In [117]: quotes
Out[117]: 
                     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
```

By default we are taking the asof of the quotes.

```
In [118]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker')
   .....: 
Out[118]: 
                     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 betwen the quote time and the trade time.

```
In [119]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('2ms'))
   .....: 
Out[119]: 
                     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 betwen 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 propogate to that point in time.

```
In [120]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('10ms'),
   .....:               allow_exact_matches=False)
   .....:
Out[120]:
                     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
4 2016-05-25 13:30:00.048   AAPL   98.00       100    NaN    NaN
```