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]: df = DataFrame(np.random.randn(10, 4))

In [2]: df
Out[2]:
          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914

# break it into pieces
In [3]: pieces = [df[:3], df[3:7], df[7:]]

In [4]: concatenated = concat(pieces)

In [5]: concatenated
Out[5]:
          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```

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":

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

- `objs`: list or dict 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)
- `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
- `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. If keys passed, specific levels to use for the resulting 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
- `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.

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]: concatenated = concat(pieces, keys=['first', 'second', 'third'])

In [7]: concatenated
Out[7]: 
                 0         1         2         3
first  0  0.469112 -0.282863 -1.509059 -1.135632
       1  1.212112 -0.173215  0.119209 -1.044236
       2 -0.861849 -2.104569 -0.494929  1.071804
second 3  0.721555 -0.706771 -1.039575  0.271860
       4 -0.424972  0.567020  0.276232 -1.087401
       5 -0.673690  0.113648 -1.478427  0.524988
       6  0.404705  0.577046 -1.715002 -1.039268
third  7 -0.370647 -1.157892 -1.344312  0.844885
       8  1.075770 -0.109050  1.643563 -1.469388
       9  0.357021 -0.674600 -1.776904 -0.968914
```

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 [8]: concatenated.ix['second']
Out[8]: 
          0         1         2         3
3  0.721555 -0.706771 -1.039575  0.271860
```

```
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
```

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 signifcant 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 [9]: from pandas.util.testing import rands_array

In [10]: df = DataFrame(np.random.randn(10, 4), columns=['a', 'b', 'c', 'd'],
   ....:                index=rands_array(5, 10))
   ....:

In [11]: df
Out[11]:
              a         b         c         d
YpIua -1.294524  0.413738  0.276662 -0.472035
HpwKq -0.013960 -0.362543 -0.006154 -0.923061
2HQRv  0.895717  0.805244 -1.206412  2.565646
VSDol  1.431256  1.340309 -1.170299 -0.226169
DQeX6  0.410835  0.813850  0.132003 -0.827317
xplCd -0.076467 -1.187678  1.130127 -1.436737
VMkkM -1.413681  1.607920  1.024180  0.569605
vyR6D  0.875906 -2.211372  0.974466 -2.006747
xUE69 -0.410001 -0.078638  0.545952 -1.219217
UoniI -1.226825  0.769804 -1.281247 -0.727707

In [12]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1)
   ....:
Out[12]:
              a         b         c         d
2HQRv  0.895717  0.805244 -1.206412       NaN
```

```
DQeX6  0.410835  0.813850  0.132003 -0.827317
HpwKq -0.013960 -0.362543       NaN       NaN
UoniI       NaN       NaN       NaN -0.727707
VMkkM -1.413681  1.607920  1.024180  0.569605
VSDol  1.431256  1.340309 -1.170299 -0.226169
YpIua -1.294524  0.413738       NaN       NaN
vyR6D       NaN       NaN  0.974466 -2.006747
xUE69       NaN       NaN       NaN -1.219217
xplCd -0.076467 -1.187678  1.130127 -1.436737
```

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

```
In [13]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1, join='inner')
   ....:
Out[13]:
              a         b         c         d
VSDol  1.431256  1.340309 -1.170299 -0.226169
DQeX6  0.410835  0.813850  0.132003 -0.827317
xplCd -0.076467 -1.187678  1.130127 -1.436737
VMkkM -1.413681  1.607920  1.024180  0.569605
```

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

```
In [14]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
   ....:         df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
   ....:
Out[14]:
              a         b         c         d
YpIua -1.294524  0.413738       NaN       NaN
HpwKq -0.013960 -0.362543       NaN       NaN
2HQRv  0.895717  0.805244 -1.206412       NaN
VSDol  1.431256  1.340309 -1.170299 -0.226169
DQeX6  0.410835  0.813850  0.132003 -0.827317
xplCd -0.076467 -1.187678  1.130127 -1.436737
VMkkM -1.413681  1.607920  1.024180  0.569605
vyR6D       NaN       NaN  0.974466 -2.006747
xUE69       NaN       NaN       NaN -1.219217
UoniI       NaN       NaN       NaN -0.727707
```

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 [15]: s = Series(randn(10), index=np.arange(10))

In [16]: s1 = s[:5] # note we're slicing with labels here, so 5 is included

In [17]: s2 = s[6:]

In [18]: s1.append(s2)
Out[18]:
0    0.690579
```

```
1    0.995761
2    2.396780
3    0.014871
4    3.357427
6   -1.236269
7    0.896171
8   -0.487602
9   -0.082240
dtype: float64
```

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

```
In [19]: df = DataFrame(randn(6, 4), index=date_range('1/1/2000', periods=6),
   ....:                columns=['A', 'B', 'C', 'D'])
   ....:

In [20]: df1 = df.ix[:3]

In [21]: df2 = df.ix[3:, :3]

In [22]: df1
Out[22]:
                   A         B         C        D
2000-01-01 -2.182937  0.380396  0.084844  0.43239
2000-01-02  1.519970 -0.493662  0.600178  0.27423
2000-01-03  0.132885 -0.023688  2.410179  1.45052

In [23]: df2
Out[23]:
                   A         B         C
2000-01-04  0.206053 -0.251905 -2.213588
2000-01-05  1.266143  0.299368 -0.863838
2000-01-06 -1.048089 -0.025747 -0.988387

In [24]: df1.append(df2)
Out[24]:
                   A         B         C        D
2000-01-01 -2.182937  0.380396  0.084844  0.43239
2000-01-02  1.519970 -0.493662  0.600178  0.27423
2000-01-03  0.132885 -0.023688  2.410179  1.45052
2000-01-04  0.206053 -0.251905 -2.213588      NaN
2000-01-05  1.266143  0.299368 -0.863838      NaN
2000-01-06 -1.048089 -0.025747 -0.988387      NaN
```

`append` may take multiple objects to concatenate:

```
In [25]: df1 = df.ix[:2]

In [26]: df2 = df.ix[2:4]

In [27]: df3 = df.ix[4:]

In [28]: df1.append([df2,df3])
Out[28]:
                   A         B         C         D
2000-01-01 -2.182937  0.380396  0.084844  0.432390
2000-01-02  1.519970 -0.493662  0.600178  0.274230
2000-01-03  0.132885 -0.023688  2.410179  1.450520
2000-01-04  0.206053 -0.251905 -2.213588  1.063327
2000-01-05  1.266143  0.299368 -0.863838  0.408204
```

```
2000-01-06 -1.048089 -0.025747 -0.988387  0.094055
```

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:

```
In [29]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])

In [30]: df2 = DataFrame(randn(3, 4), columns=['A', 'B', 'C', 'D'])

In [31]: df1
Out[31]:
          A         B         C         D
0  1.262731  1.289997  0.082423 -0.055758
1  0.536580 -0.489682  0.369374 -0.034571
2 -2.484478 -0.281461  0.030711  0.109121
3  1.126203 -0.977349  1.474071 -0.064034
4 -1.282782  0.781836 -1.071357  0.441153
5  2.353925  0.583787  0.221471 -0.744471

In [32]: df2
Out[32]:
          A         B         C         D
0  0.758527  1.729689 -0.964980 -0.845696
1 -1.340896  1.846883 -1.328865  1.682706
2 -1.717693  0.888782  0.228440  0.901805
```

To do this, use the `ignore_index` argument:

```
In [33]: concat([df1, df2], ignore_index=True)
Out[33]:
          A         B         C         D
0  1.262731  1.289997  0.082423 -0.055758
1  0.536580 -0.489682  0.369374 -0.034571
2 -2.484478 -0.281461  0.030711  0.109121
3  1.126203 -0.977349  1.474071 -0.064034
4 -1.282782  0.781836 -1.071357  0.441153
5  2.353925  0.583787  0.221471 -0.744471
6  0.758527  1.729689 -0.964980 -0.845696
7 -1.340896  1.846883 -1.328865  1.682706
8 -1.717693  0.888782  0.228440  0.901805
```

This is also a valid argument to `DataFrame.append`:

```
In [34]: df1.append(df2, ignore_index=True)
Out[34]:
          A         B         C         D
0  1.262731  1.289997  0.082423 -0.055758
1  0.536580 -0.489682  0.369374 -0.034571
```

```
2 -2.484478 -0.281461  0.030711  0.109121
3  1.126203 -0.977349  1.474071 -0.064034
4 -1.282782  0.781836 -1.071357  0.441153
5  2.353925  0.583787  0.221471 -0.744471
6  0.758527  1.729689 -0.964980 -0.845696
7 -1.340896  1.846883 -1.328865  1.682706
8 -1.717693  0.888782  0.228440  0.901805
```

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 [35]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])

In [36]: s1 = Series(randn(6), name='foo')

In [37]: concat([df1, s1],axis=1)
Out[37]: 
          A         B         C         D       foo
0  1.171216  0.520260 -1.197071 -1.066969 -0.496922
1 -0.303421 -0.858447  0.306996 -0.028665  0.306389
2  0.384316  1.574159  1.588931  0.476720 -2.290613
3  0.473424 -0.242861 -0.014805 -0.284319 -1.134623
4  0.650776 -1.461665 -1.137707 -0.891060 -1.561819
5 -0.693921  1.613616  0.464000  0.227371 -0.260838
```

If unnamed Series are passed they will be numbered consecutively.

```
In [38]: s2 = Series(randn(6))

In [39]: concat([df1, s2, s2, s2],axis=1)
Out[39]: 
          A         B         C         D         0         1         2
0  1.171216  0.520260 -1.197071 -1.066969  0.281957  0.281957  0.281957
1 -0.303421 -0.858447  0.306996 -0.028665  1.523962  1.523962  1.523962
2  0.384316  1.574159  1.588931  0.476720 -0.902937 -0.902937 -0.902937
3  0.473424 -0.242861 -0.014805 -0.284319  0.068159  0.068159  0.068159
4  0.650776 -1.461665 -1.137707 -0.891060 -0.057873 -0.057873 -0.057873
5 -0.693921  1.613616  0.464000  0.227371 -0.368204 -0.368204 -0.368204
```

Passing `ignore_index=True` will drop all name references.

```
In [40]: concat([df1, s1],axis=1,ignore_index=True)
Out[40]: 
          0         1         2         3         4
0  1.171216  0.520260 -1.197071 -1.066969 -0.496922
1 -0.303421 -0.858447  0.306996 -0.028665  0.306389
2  0.384316  1.574159  1.588931  0.476720 -2.290613
3  0.473424 -0.242861 -0.014805 -0.284319 -1.134623
4  0.650776 -1.461665 -1.137707 -0.891060 -1.561819
5 -0.693921  1.613616  0.464000  0.227371 -0.260838
```

More concatenating with group keys

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

```
In [41]: df = DataFrame(np.random.randn(10, 4))

In [42]: df
Out[42]:
          0         1         2         3
0 -1.144073  0.861209  0.800193  0.782098
1 -1.069094 -1.099248  0.255269  0.009750
2  0.661084  0.379319 -0.008434  1.952541
3 -1.056652  0.533946 -1.226970  0.040403
4 -0.507516 -0.230096  0.394500 -1.934370
5 -1.652499  1.488753 -0.896484  0.576897
6  1.146000  1.487349  0.604603  2.121453
7  0.597701  0.563700  0.967661 -1.057909
8  1.375020 -0.928797 -0.308853 -0.681087
9  0.377953  0.493672 -2.461467 -1.553902

# break it into pieces
In [43]: pieces = [df.ix[:, [0, 1]], df.ix[:, [2]], df.ix[:, [3]]]

In [44]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'])

In [45]: result
Out[45]:
        one                 two     three
          0         1         2         3
0 -1.144073  0.861209  0.800193  0.782098
1 -1.069094 -1.099248  0.255269  0.009750
2  0.661084  0.379319 -0.008434  1.952541
3 -1.056652  0.533946 -1.226970  0.040403
4 -0.507516 -0.230096  0.394500 -1.934370
5 -1.652499  1.488753 -0.896484  0.576897
6  1.146000  1.487349  0.604603  2.121453
7  0.597701  0.563700  0.967661 -1.057909
8  1.375020 -0.928797 -0.308853 -0.681087
9  0.377953  0.493672 -2.461467 -1.553902
```

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 [46]: pieces = {'one': df.ix[:, [0, 1]],
   ....:           'two': df.ix[:, [2]],
   ....:           'three': df.ix[:, [3]]}
   ....:

In [47]: concat(pieces, axis=1)
Out[47]:
        one               three       two
          0         1         3         2
0 -1.144073  0.861209  0.782098  0.800193
1 -1.069094 -1.099248  0.009750  0.255269
2  0.661084  0.379319  1.952541 -0.008434
3 -1.056652  0.533946  0.040403 -1.226970
4 -0.507516 -0.230096 -1.934370  0.394500
5 -1.652499  1.488753  0.576897 -0.896484
6  1.146000  1.487349  2.121453  0.604603
7  0.597701  0.563700 -1.057909  0.967661
```

```
8  1.375020 -0.928797 -0.681087 -0.308853
9  0.377953  0.493672 -1.553902 -2.461467

In [48]: concat(pieces, keys=['three', 'two'])
Out[48]: 
                2         3
three 0       NaN  0.782098
      1       NaN  0.009750
      2       NaN  1.952541
      3       NaN  0.040403
      4       NaN -1.934370
      5       NaN  0.576897
      6       NaN  2.121453
...           ...       ...
two   3 -1.226970       NaN
      4  0.394500       NaN
      5 -0.896484       NaN
      6  0.604603       NaN
      7  0.967661       NaN
      8 -0.308853       NaN
      9 -2.461467       NaN

[20 rows x 2 columns]
```

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

```
In [49]: result.columns.levels
Out[49]: FrozenList([[u'one', u'two', u'three'], [0, 1, 2, 3]])
```

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

```
In [50]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'],
   ....:                 levels=[['three', 'two', 'one', 'zero']],
   ....:                 names=['group_key'])
   ....: 

In [51]: result
Out[51]: 
group_key       one                 two     three
                  0         1         2         3
0         -1.144073  0.861209  0.800193  0.782098
1         -1.069094 -1.099248  0.255269  0.009750
2          0.661084  0.379319 -0.008434  1.952541
3         -1.056652  0.533946 -1.226970  0.040403
4         -0.507516 -0.230096  0.394500 -1.934370
5         -1.652499  1.488753 -0.896484  0.576897
6          1.146000  1.487349  0.604603  2.121453
7          0.597701  0.563700  0.967661 -1.057909
8          1.375020 -0.928797 -0.308853 -0.681087
9          0.377953  0.493672 -2.461467 -1.553902

In [52]: result.columns.levels
Out[52]: FrozenList([[u'three', u'two', u'one', u'zero'], [0, 1, 2, 3]])
```

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 [53]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [54]: df
Out[54]: 
          A         B         C         D
0  2.015523 -1.833722  1.771740 -0.670027
1  0.049307 -0.521493 -3.201750  0.792716
2  0.146111  1.903247 -0.747169 -0.309038
3  0.393876  1.861468  0.936527  1.255746
4 -2.655452  1.219492  0.062297 -0.110388
5 -1.184357 -0.558081  0.077849  0.629498
6 -1.035260 -0.438229  0.503703  0.413086
7 -1.139050  0.660342  0.464794 -0.309337

In [55]: s = df.xs(3)

In [56]: df.append(s, ignore_index=True)
Out[56]: 
          A         B         C         D
0  2.015523 -1.833722  1.771740 -0.670027
1  0.049307 -0.521493 -3.201750  0.792716
2  0.146111  1.903247 -0.747169 -0.309038
3  0.393876  1.861468  0.936527  1.255746
4 -2.655452  1.219492  0.062297 -0.110388
5 -1.184357 -0.558081  0.077849  0.629498
6 -1.035260 -0.438229  0.503703  0.413086
7 -1.139050  0.660342  0.464794 -0.309337
8  0.393876  1.861468  0.936527  1.255746
```

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 [57]: df = DataFrame(np.random.randn(5, 4),
   ....:                columns=['foo', 'bar', 'baz', 'qux'])
   ....: 

In [58]: dicts = [{'foo': 1, 'bar': 2, 'baz': 3, 'peekaboo': 4},
   ....:          {'foo': 5, 'bar': 6, 'baz': 7, 'peekaboo': 8}]
   ....: 

In [59]: result = df.append(dicts, ignore_index=True)

In [60]: result
Out[60]: 
        bar       baz       foo  peekaboo       qux
0  0.683758 -0.643834 -0.649593       NaN  0.421287
1 -1.290493  0.787872  1.032814       NaN  1.515707
```

```
2 -0.223762  1.397431 -0.276487       NaN  1.503874
3 -0.135950 -0.730327 -0.478905       NaN -0.033277
4 -1.298915 -2.819487  0.281151       NaN -0.851985
5  2.000000  3.000000  1.000000         4       NaN
6  6.000000  7.000000  5.000000         8       NaN
```

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:

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

Here's a description of what each argument is for:

- `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.

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 and index-on-column(s) joins, but *joins on indexes* by default rather than trying to join on common columns (the default behavior for `merge`). If you are joining on index, 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 [61]: left = DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [62]: right = DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [63]: left
Out[63]: 
   key  lval
0  foo     1
1  foo     2
```

```
In [64]: right
Out[64]: 
   key  rval
0  foo     4
1  foo     5

In [65]: merge(left, right, on='key')
Out[65]: 
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Here is a more complicated example with multiple join keys:

```
In [66]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                   'key2': ['one', 'two', 'one'],
   ....:                   'lval': [1, 2, 3]})
   ....: 

In [67]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                    'key2': ['one', 'one', 'one', 'two'],
   ....:                    'rval': [4, 5, 6, 7]})
   ....: 

In [68]: merge(left, right, how='outer')
Out[68]: 
  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  foo  two     2   NaN
3  bar  one     3     6
4  bar  two   NaN     7

In [69]: merge(left, right, how='inner')
Out[69]: 
  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  bar  one     3     6
```

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

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 [70]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [71]: df1 = df.ix[1:, ['A', 'B']]

In [72]: df2 = df.ix[:5, ['C', 'D']]

In [73]: df1
Out[73]:
          A         B
1 -2.277282 -0.390201
2 -1.004168 -1.377627
3  0.162565 -0.067785
4 -2.006481  0.301016
5 -2.400634 -0.280853
6  0.863937  0.252462
7 -2.338595 -0.374279

In [74]: df2
Out[74]:
          C         D
0 -1.537770  0.555759
1  1.207122  0.178690
2  0.499281 -1.405256
3 -1.260006 -1.132896
4  0.059117  1.138469
5  0.025653 -1.386071

In [75]: df1.join(df2)
Out[75]:
          A         B         C         D
1 -2.277282 -0.390201  1.207122  0.178690
2 -1.004168 -1.377627  0.499281 -1.405256
3  0.162565 -0.067785 -1.260006 -1.132896
4 -2.006481  0.301016  0.059117  1.138469
5 -2.400634 -0.280853  0.025653 -1.386071
6  0.863937  0.252462       NaN       NaN
7 -2.338595 -0.374279       NaN       NaN

In [76]: df1.join(df2, how='outer')
Out[76]:
          A         B         C         D
0       NaN       NaN -1.537770  0.555759
1 -2.277282 -0.390201  1.207122  0.178690
2 -1.004168 -1.377627  0.499281 -1.405256
3  0.162565 -0.067785 -1.260006 -1.132896
4 -2.006481  0.301016  0.059117  1.138469
5 -2.400634 -0.280853  0.025653 -1.386071
6  0.863937  0.252462       NaN       NaN
7 -2.338595 -0.374279       NaN       NaN

In [77]: df1.join(df2, how='inner')
Out[77]:
          A         B         C         D
1 -2.277282 -0.390201  1.207122  0.178690
2 -1.004168 -1.377627  0.499281 -1.405256
3  0.162565 -0.067785 -1.260006 -1.132896
4 -2.006481  0.301016  0.059117  1.138469
5 -2.400634 -0.280853  0.025653 -1.386071
```

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 [78]: merge(df1, df2, left_index=True, right_index=True, how='outer')
Out[78]:
          A         B         C         D
0       NaN       NaN -1.537770  0.555759
1 -2.277282 -0.390201  1.207122  0.178690
2 -1.004168 -1.377627  0.499281 -1.405256
3  0.162565 -0.067785 -1.260006 -1.132896
4 -2.006481  0.301016  0.059117  1.138469
5 -2.400634 -0.280853  0.025653 -1.386071
6  0.863937  0.252462       NaN       NaN
7 -2.338595 -0.374279       NaN       NaN
```

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)
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 [79]: df['key'] = ['foo', 'bar'] * 4

In [80]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
   ....:                     columns=['j1', 'j2'])
   ....:

In [81]: df
Out[81]:
          A         B         C         D  key
0 -1.106952 -0.937731 -1.537770  0.555759  foo
1 -2.277282 -0.390201  1.207122  0.178690  bar
2 -1.004168 -1.377627  0.499281 -1.405256  foo
3  0.162565 -0.067785 -1.260006 -1.132896  bar
4 -2.006481  0.301016  0.059117  1.138469  foo
5 -2.400634 -0.280853  0.025653 -1.386071  bar
6  0.863937  0.252462  1.500571  1.053202  foo
7 -2.338595 -0.374279 -2.359958 -1.157886  bar

In [82]: to_join
Out[82]:
           j1        j2
bar -0.551865  1.592673
foo  1.559318  1.562443

In [83]: df.join(to_join, on='key')
Out[83]:
```

```
           A         B         C         D  key        j1        j2
0 -1.106952 -0.937731 -1.537770  0.555759  foo  1.559318  1.562443
1 -2.277282 -0.390201  1.207122  0.178690  bar -0.551865  1.592673
2 -1.004168 -1.377627  0.499281 -1.405256  foo  1.559318  1.562443
3  0.162565 -0.067785 -1.260006 -1.132896  bar -0.551865  1.592673
4 -2.006481  0.301016  0.059117  1.138469  foo  1.559318  1.562443
5 -2.400634 -0.280853  0.025653 -1.386071  bar -0.551865  1.592673
6  0.863937  0.252462  1.500571  1.053202  foo  1.559318  1.562443
7 -2.338595 -0.374279 -2.359958 -1.157886  bar -0.551865  1.592673

In [84]: merge(df, to_join, left_on='key', right_index=True,
   ....:       how='left', sort=False)
   ....:
Out[84]:
           A         B         C         D  key        j1        j2
0 -1.106952 -0.937731 -1.537770  0.555759  foo  1.559318  1.562443
1 -2.277282 -0.390201  1.207122  0.178690  bar -0.551865  1.592673
2 -1.004168 -1.377627  0.499281 -1.405256  foo  1.559318  1.562443
3  0.162565 -0.067785 -1.260006 -1.132896  bar -0.551865  1.592673
4 -2.006481  0.301016  0.059117  1.138469  foo  1.559318  1.562443
5 -2.400634 -0.280853  0.025653 -1.386071  bar -0.551865  1.592673
6  0.863937  0.252462  1.500571  1.053202  foo  1.559318  1.562443
7 -2.338595 -0.374279 -2.359958 -1.157886  bar -0.551865  1.592673
```

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

```
In [85]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   ....:                            ['one', 'two', 'three']],
   ....:                    labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   ....:                            [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   ....:                    names=['first', 'second'])
   ....:

In [86]: to_join = DataFrame(np.random.randn(10, 3), index=index,
   ....:                     columns=['j_one', 'j_two', 'j_three'])
   ....:

# a little relevant example with NAs
In [87]: key1 = ['bar', 'bar', 'bar', 'foo', 'foo', 'baz', 'baz', 'qux',
   ....:         'qux', 'snap']
   ....:

In [88]: key2 = ['two', 'one', 'three', 'one', 'two', 'one', 'two', 'two',
   ....:         'three', 'one']
   ....:

In [89]: data = np.random.randn(len(key1))

In [90]: data = DataFrame({'key1' : key1, 'key2' : key2,
   ....:                   'data' : data})
   ....:

In [91]: data
Out[91]:
       data key1   key2
0 -1.114738  bar    two
1 -0.058216  bar    one
2 -0.486768  bar  three
3  1.685148  foo    one
4  0.112572  foo    two
5 -1.495309  baz    one
6  0.898435  baz    two
```

```
7 -0.148217   qux    two
8 -1.596070   qux  three
9  0.159653  snap    one

In [92]: to_join
Out[92]:
                 j_one     j_two   j_three
first second
foo   one     0.763264  0.162027 -0.902704
      two     1.106010 -0.199234  0.458265
      three   0.491048  0.128594  1.147862
bar   one    -1.256860  0.563637 -2.417312
      two     0.972827  0.041293  1.129659
baz   two     0.086926 -0.445645 -0.217503
      three  -1.420361 -0.015601 -1.150641
qux   one    -0.798334 -0.557697  0.381353
      two     1.337122 -1.531095  1.331458
      three  -0.571329 -0.026671 -1.085663
```

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

```
In [93]: data.join(to_join, on=['key1', 'key2'])
Out[93]:
       data  key1   key2     j_one     j_two   j_three
0 -1.114738   bar    two  0.972827  0.041293  1.129659
1 -0.058216   bar    one -1.256860  0.563637 -2.417312
2 -0.486768   bar  three       NaN       NaN       NaN
3  1.685148   foo    one  0.763264  0.162027 -0.902704
4  0.112572   foo    two  1.106010 -0.199234  0.458265
5 -1.495309   baz    one       NaN       NaN       NaN
6  0.898435   baz    two  0.086926 -0.445645 -0.217503
7 -0.148217   qux    two  1.337122 -1.531095  1.331458
8 -1.596070   qux  three -0.571329 -0.026671 -1.085663
9  0.159653  snap    one       NaN       NaN       NaN
```

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 [94]: data.join(to_join, on=['key1', 'key2'], how='inner')
Out[94]:
       data key1   key2     j_one     j_two   j_three
0 -1.114738  bar    two  0.972827  0.041293  1.129659
1 -0.058216  bar    one -1.256860  0.563637 -2.417312
3  1.685148  foo    one  0.763264  0.162027 -0.902704
4  0.112572  foo    two  1.106010 -0.199234  0.458265
6  0.898435  baz    two  0.086926 -0.445645 -0.217503
7 -0.148217  qux    two  1.337122 -1.531095  1.331458
8 -1.596070  qux  three -0.571329 -0.026671 -1.085663
```

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

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 [95]: left = DataFrame({'key': ['foo', 'foo'], 'value': [1, 2]})

In [96]: right = DataFrame({'key': ['foo', 'foo'], 'value': [4, 5]})

In [97]: merge(left, right, on='key', suffixes=['_left', '_right'])
Out[97]: 
   key  value_left  value_right
0  foo           1            4
1  foo           1            5
2  foo           2            4
3  foo           2            5
```

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

Merging Ordered Data

New in v0.8.0 is the ordered_merge function for combining time series and other ordered data. In particular it has an optional `fill_method` keyword to fill/interpolate missing data:

```
In [98]: A
Out[98]: 
  group key  lvalue
0     a   a       1
1     a   c       2
2     a   e       3
3     b   a       1
4     b   c       2
5     b   e       3

In [99]: B
Out[99]: 
  key  rvalue
0   b       1
1   c       2
2   d       3

In [100]: ordered_merge(A, B, fill_method='ffill', left_by='group')
Out[100]: 
  group key  lvalue  rvalue
0     a   a       1     NaN
1     a   b       1       1
2     a   c       2       2
3     a   d       2       3
4     a   e       3       3
5     b   a       1     NaN
6     b   b       1       1
7     b   c       2       2
8     b   d       2       3
9     b   e       3       3
```

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 [101]: df1 = df.ix[:, ['A', 'B']]

In [102]: df2 = df.ix[:, ['C', 'D']]

In [103]: df3 = df.ix[:, ['key']]

In [104]: df1
Out[104]:
          A         B
0 -1.106952 -0.937731
1 -2.277282 -0.390201
2 -1.004168 -1.377627
3  0.162565 -0.067785
4 -2.006481  0.301016
5 -2.400634 -0.280853
6  0.863937  0.252462
7 -2.338595 -0.374279

In [105]: df1.join([df2, df3])
Out[105]:
          A         B         C         D  key
0 -1.106952 -0.937731 -1.537770  0.555759  foo
1 -2.277282 -0.390201  1.207122  0.178690  bar
2 -1.004168 -1.377627  0.499281 -1.405256  foo
3  0.162565 -0.067785 -1.260006 -1.132896  bar
4 -2.006481  0.301016  0.059117  1.138469  foo
5 -2.400634 -0.280853  0.025653 -1.386071  bar
6  0.863937  0.252462  1.500571  1.053202  foo
7 -2.338595 -0.374279 -2.359958 -1.157886  bar
```

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 [106]: df1 = DataFrame([[nan, 3., 5.], [-4.6, np.nan, nan],
   .....:                  [nan, 7., nan]])
   .....:

In [107]: df2 = DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4]],
   .....:                 index=[1, 2])
   .....:
```

For this, use the `combine_first` method:

```
In [108]: df1.combine_first(df2)
Out[108]:
     0    1    2
0  NaN    3  5.0
1 -4.6  NaN -8.2
2 -5.0    7  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 [109]: df1.update(df2)

In [110]: df1
Out[110]:
      0    1    2
0   NaN  3.0  5.0
1 -42.6  NaN -8.2
2  -5.0  1.6  4.0
```

Merging with Multi-indexes

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 [111]: household = DataFrame(dict(household_id = [1,2,3],
   .....:                            male = [0,1,0],
   .....:                            wealth = [196087.3,316478.7,294750]),
   .....:                       columns = ['household_id','male','wealth']
   .....:                      ).set_index('household_id')
   .....:

In [112]: household
Out[112]:
              male    wealth
household_id
1                0  196087.3
2                1  316478.7
3                0  294750.0

In [113]: portfolio = DataFrame(dict(household_id = [1,2,2,3,3,3,4],
   .....:                            asset_id = ["nl0000301109","nl0000289783","gb00b
   .....:                                        "gb00b03mlx29","lu0197800237","nl000
   .....:                            name = ["ABN Amro","Robeco","Royal Dutch Shell",
   .....:                                    "AAB Eastern Europe Equity Fund","Postba
   .....:                            share = [1.0,0.4,0.6,0.15,0.6,0.25,1.0]),
   .....:                       columns = ['household_id','asset_id','name','share']
   .....:                      ).set_index(['household_id','asset_id'])
   .....:

In [114]: portfolio
Out[114]:
                                        name  share
household_id asset_id
1            nl0000301109           ABN Amro   1.00
2            nl0000289783             Robeco   0.40
             gb00b03mlx29  Royal Dutch Shell   0.60
3            gb00b03mlx29  Royal Dutch Shell   0.15
```

```
             lu0197800237  AAB Eastern Europe Equity Fund   0.60
             nl0000289965          Postbank BioTech Fonds   0.25
4            NaN                                       NaN   1.00

In [115]: household.join(portfolio, how='inner')
Out[115]:
                           male    wealth                            name  \
household_id asset_id
1            nl0000301109     0  196087.3                        ABN Amro
2            nl0000289783     1  316478.7                          Robeco
             gb00b03mlx29     1  316478.7               Royal Dutch Shell
3            gb00b03mlx29     0  294750.0               Royal Dutch Shell
             lu0197800237     0  294750.0  AAB Eastern Europe Equity Fund
             nl0000289965     0  294750.0          Postbank BioTech Fonds

                           share
household_id asset_id
1            nl0000301109   1.00
2            nl0000289783   0.40
             gb00b03mlx29   0.60
3            gb00b03mlx29   0.15
             lu0197800237   0.60
             nl0000289965   0.25
```

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

```
merge(household.reset_index(),
      portfolio.reset_index(),
      on=['household_id'],
      how='inner'
     ).set_index(['household_id','asset_id'])
```

Joining with two multi-indexes

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

```
In [116]: household = DataFrame(dict(household_id = [1,2,2,3,3,3,4],
   .....:                            asset_id = ["nl0000301109","nl0000301109","gb00b
   .....:                                        "gb00b03mlx29","lu0197800237","nl000
   .....:                            share = [1.0,0.4,0.6,0.15,0.6,0.25,1.0]),
   .....:                       columns = ['household_id','asset_id','share']
   .....:                      ).set_index(['household_id','asset_id'])
   .....:

In [117]: household
Out[117]:
                           share
household_id asset_id
1            nl0000301109   1.00
2            nl0000301109   0.40
             gb00b03mlx29   0.60
3            gb00b03mlx29   0.15
             lu0197800237   0.60
             nl0000289965   0.25
4            NaN            1.00

In [118]: log_return = DataFrame(dict(asset_id = ["gb00b03mlx29", "gb00b03mlx29", "gb
```

```
   .....:                                          "lu0197800237", "lu0197800237"],
   .....:                             t = [233, 234, 235, 180, 181],
   .....:                             log_return = [.09604978, -.06524096, .03532373,
   .....:                       ).set_index(["asset_id","t"])
   .....:

In [119]: log_return
Out[119]: 
                  log_return
asset_id     t              
gb00b03mlx29 233    0.096050
             234   -0.065241
             235    0.035324
lu0197800237 180    0.030254
             181    0.036997

In [120]: merge(household.reset_index(),
   .....:       log_return.reset_index(),
   .....:       on=['asset_id'],
   .....:       how='inner'
   .....:      ).set_index(['household_id','asset_id','t'])
   .....:
Out[120]: 
                              share  log_return
household_id asset_id     t                    
2            gb00b03mlx29 233   0.60    0.096050
                          234   0.60   -0.065241
                          235   0.60    0.035324
3            gb00b03mlx29 233   0.15    0.096050
                          234   0.15   -0.065241
                          235   0.15    0.035324
             lu0197800237 180   0.60    0.030254
                          181   0.60    0.036997
```