			df1					Res	sult		
_		Α	В	С	D		Α	В	С	D	F
	0	A0	В	ω c	0 D0	0	A0	BO	ω	D0	NaN
Γ	1	A1	В	1 0	1 D1		_	_	_	_	-
Ì	2	A2	В	2 0	2 D2	1	A1	B1	C1	D1	NaN
Ì	3	АЗ	В	3 C	3 D3	2	A2	B2	(2	D2	NaN
•			df4			3	A3	В3	СЗ	D3	NaN
_		В		D	F	4	NaN	B2	NaN	D2	F2
L	2	2	B2	D2	F2	5	NaN	В3	NaN	D3	F3
	3	3	B3	D3	F3	6	NaN	B6	NaN	D6	F6
Γ	6	5	B6	D6	F6		_			_	
Ì	7	7	B7	D7	F7	7	NaN	B7	NaN	D7	F7

This is also a valid argument to <code>DataFrame.append():</code>

```
In [17]: result = df1.append(df4, ignore_index=True, sort=False)
```

			df]	l					Res	sult		
		Α	В		С	D		Α	В	С	D	F
	0	A0	E	30	α	D0		40	DO.		- PO	NI-NI
ı	1	Al	E	31	CI	. D1	0	A0	B0	ω	D0	NaN
ı	2	A2	E	32	C2	D2	1	A1	B1	C1	D1	NaN
ı	3	A3	E	33	C3	D3	2	A2	B2	(2	D2	NaN
١			df4	1			3	A3	В3	СЗ	D3	NaN
		В)	F	4	NaN	B2	NaN	D2	F2
	2	2	B2		D2	F2	5	NaN	В3	NaN	D3	F3
	3	3	B3		D3	F3	6	NaN	B6	NaN	D6	F6
ı	6	5	B6		D6	F6					-	\vdash
ı	7	7	B7		D7	F7	7	NaN	B7	NaN	D7	F7

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.

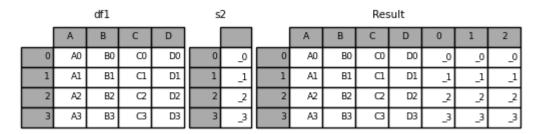
```
In [18]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')
In [19]: result = pd.concat([df1, s1], axis=1)
```

		df1			S	1			Res	ult		
	Α	В	С	D		Х		Α	В	С	D	Х
0	A0	B0	α	D0	0	X0	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	Х2	2	A2	B2	C2	D2	Х2
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.

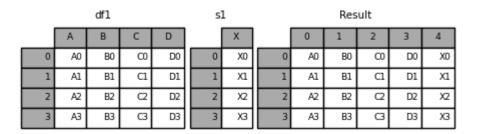
If unnamed Series are passed they will be numbered consecutively.

```
In [20]: s2 = pd.Series(['_0', '_1', '_2', '_3'])
In [21]: result = pd.concat([df1, s2, s2, s2], axis=1)
```



Passing ignore_index=True will drop all name references.

```
In [22]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```



438

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

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

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

```
In [27]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
Out[27]:
    red blue yellow
0     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 [28]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

		df1					Res	sult		
	Α	В	С	D						_
0	A0	B0	α	D0			А	В	U	D
1	A1	B1	Cl	D1	×	0	AD	BO	8	DO
2	A2	B2	В	D2	×	1	A1	B1	а	D1
3	A3	В3	СЗ	D3	×	2	A2	B2	Q	D2
		df2			×	3	A3	B3	з	D3
	Α	В	U	D	^	3		- 53		\vdash
4	A4	B4	C4	D4	У	4	A4	B4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	C	D5
6	Аб	B6	C6	D6	У	6	Aß	B6	C6	D6
7	A7	B7	C7	D7	У	7	A7	B7	a	D7
		df3			z	8	AB	BB	СВ	D8
	Α	В	U	D			\vdash			$\vdash \vdash$
8	A8	B8	C8	DB	z	9	A9	B9	9	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 [29]: pieces = {'x': df1, 'y': df2, 'z': df3}
In [30]: result = pd.concat(pieces)
```

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	α	D0			А	В	U	D
1	A1	B1	C1	D1	×	0	AD	В0	8	DO
2	A2	B2	C2	D2	×	1	A1	B1	а	D1
3	A3	В3	СЗ	D3	×	2	A2	B2	a	D2
		df2								-
	Α	В	С	D	×	3	А3	B3	В	D3
4	A4	B4	C4	D4	У	4	A4	B4	C4	D4
5	A5	B5	C5	D5	У	5	A5	B5	G	D5
6	Аб	B6	C 6	D6	У	6	Aß	Bő	8	D6
7	A7	B7	C7	D7	У	7	A7	B7	a	D7
		df3					40			
	Α	В	С	D	z	8	AB	BB	СВ	D8
8	A8	B8	C8	DB	z	9	AĐ	B9	9	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 [31]: result = pd.concat(pieces, keys=['z', 'y'])
```

		dfl					Res	sult		
	Α	В	С	D						
0	A0	B0	0	D0						
1	A1	B1	Cl	D1						
2	A2	B2	C2	D2			А	В	С	D
3	A3	В3	СЗ	D3	z	8	AB	B8	СВ	D8
		df2				_		_	-	\vdash
	Α	В	С	D	z	9	A9	B9	В	D9
4	A4	B4	C4	D4	z	10	A10	B10	G 0	D10
5	A5	B5	C5	D5	z	11	A11	B11	а1	D11
6	Аб	B6	C6	D6	У	4	A4	В4	C4	D4
7	A7	B7	C7	D7	У	5	A5	B5	0	D5
		df3								\vdash
	Α	В	С	D	У	6	Aß	B6	C6	D6
8	A8	B8	C8	DB	У	7	A7	B7	ō	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 [32]: result.index.levels
Out[32]: 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:

		df1					Res	sult		
	Α	В	С	D						
0	A0	B0	0	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	B3	СЗ	D3	×	2	A2	B2	a	D2
		df2								
	Α	В	С	D	×	3	А3	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	O5	D6	У	6	Aß	B6	C6	D6
7	A7	B7	C7	D7	У	7	A7	В7	a	D7
		df3								
	Α	В	С	D	z	8	AB	B8	СВ	D8
8	A8	B8	C8	D8	z	9	A9	B9	В	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						

```
In [34]: result.index.levels
Out[34]: 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

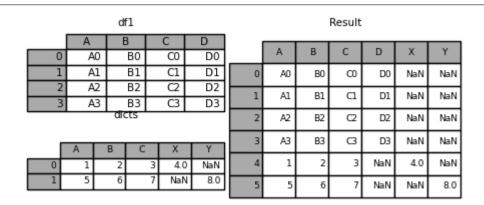
While not especially efficient (since a new object must be created), you can append a single row to a DataFrame by passing a Series or dict to append, which returns a new DataFrame as above.

```
In [35]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])
In [36]: result = df1.append(s2, ignore_index=True)
```

			df1					Result		
		Α	В	С	D					
	0	A0	BO	α	D0					
	1	A1	B1	C1	D1		Α	В	С	D
	2	A2	B2	C2	D2	0	A0	BO	ω	D0
	3	A3	В3	СЗ	D3	1	A1	B1	C1	D1
			s2			2	A2	B2	C2	D2
							~	DZ.		D2
			А		X0	3	A3	В3	В	D3
Ì			В		X1	4	X0	X1	X2	ХЗ
l			С		X2					
			D		ХЗ					

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:



2.4.2 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 base::merge.data.frame in R). The reason for this is careful algorithmic design and the 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, <code>merge()</code>, as the entry point for all standard database join operations between <code>DataFrame</code> or named <code>Series</code> 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 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 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 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:

	le	ft			rig	ht				Res	sult		
	key	Α	В		key	С	D		key	Α	В	С	D
0	K0	A0	В0	0	K0	ω	D0	0	K0	A0	B0	ω	D0
1	K1	A1	B1	1	K1	Cl	D1	1	K1	A1	B1	Cl	D1
2	K2	A2	B2	2	K2	C2	D2	2	K2	A2	B2	C2	D2
3	КЗ	A3	В3	3	КЗ	СЗ	D3	3	КЗ	A3	В3	C3	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.

(continues on next page)

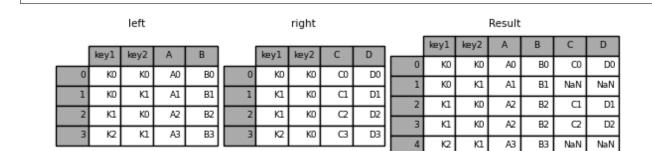
(continued from previous page)

		left					right						Result			
	keyl	key2	Α	В		keyl	key2	С	D		keyl	kev2	Α	В	C	D
0	KO	KO	A0	B0	0	K0	K0	α	D0		-					
1	K0	K1	A1	B1	1	К1	K0	C1	D1	0	K0	K0	A0	В0	0	D0
2	К1	KO	A2	B2	2	К1	КО	(2	D2	1	K1	K0	A2	B2	Cl	D1
				$\vdash \vdash$		_	-		-	2	K1	K0	A2	B2	C2	D2
3	K2	K1	A3	B3	3	K2	K0	СЗ	D3							

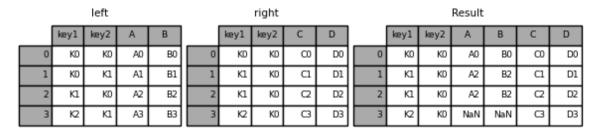
The how argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination **does not appear** in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

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

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



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



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

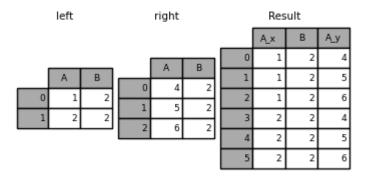
left right Result D keyl key2 В С key1 key2 key1 key2 D K0 K0 D0 1 NaN A0 В0 0 K0 $^{\circ}$ D0 K0 K1 A1 В1 NaN K0 K0 K0 K0 K1 A1 В1 K1 K0 C1 D1 2 K1 K0 A2 В2 C1 D1 3 K1 A2 K1 C2 D2 K0 B2 D2 K0 NaN K2 K1 A3 В3 K2 C3 D3 4 K2 K1 АЗ В3 NaN 5 K2 ΚO NaN NaN C3 D3

In [48]: result = pd.merge(left, right, how='inner', on=['key1', 'key2'])

left right Result key1 key2 keyl key2 D key1 key2 Α В С Α0 В0 α D0 0 D0 K0 K0 D1 K0 K1 A1 В1 K1 K0 C1 D1 K0 B2 K1 K1 ΚO A2 B2 K1 K0 C2 D2 K1 KO A2 D2 K2 K1 АЗ ВЗ K2 C3 D3

Here is another example with duplicate join keys in DataFrames:

```
In [49]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})
In [50]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})
In [51]: 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 validate 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 [52]: left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})
In [53]: 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 DataFrame but wants to ensure there are no duplicates in the left DataFrame, one can use the validate='one_to_many' argument instead, which will not raise an exception.

```
In [54]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[54]:
    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 [55]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})
In [56]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})
In [57]: pd.merge(df1, df2, on='coll', how='outer', indicator=True)
Out [57]:
                                _merge
  col1 col_left col_right
                           left_only
0
     0
          a NaN
     1
              b
                       2.0
                                both
1
     2
                       2.0 right_only
2
            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 [58]: pd.merge(df1, df2, on='col1', how='outer', indicator='indicator_column')
Out [58]:
  col1 col_left col_right indicator_column
     0 a NaN
                          left_only
1
     1
             b
                      2.0
                                    both
2
     2
           NaN
                      2.0
                               right_only
3
                      2.0
           NaN
                               right_only
```

Merge dtypes

Merging will preserve the dtype of the join keys.

```
In [59]: left = pd.DataFrame({'key': [1], 'v1': [10]})
In [60]: left
Out[60]:
    key v1
0    1 10

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

In [62]: right
Out[62]:
    key v1
0    1 20
1    2 30
```

We are able to preserve the join keys:

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

In [64]: pd.merge(left, right, how='outer').dtypes
Out[64]:
key int64
v1 int64
dtype: object
```

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

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

The left frame.

```
In [67]: from pandas.api.types import CategoricalDtype
In [68]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))
In [69]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))
In [70]: left = pd.DataFrame({'X': X,
                              'Y': np.random.choice(['one', 'two', 'three'],
  . . . . :
                                                    size=(10,))})
  . . . . :
  . . . . :
In [71]: left
Out [71]:
           Y
    X
0 bar
         one
  foo
        one
  foo three
  bar
       three
  foo
        one
  bar
         one
6 bar three
  bar three
8 bar three
9 foo three
In [72]: left.dtypes
```

(continues on next page)

(continued from previous page)

```
Out[72]:

X category

Y object
dtype: object
```

The right frame.

```
In [73]: right = pd.DataFrame({'X': pd.Series(['foo', 'bar'],
                                               dtype=CategoricalDtype(['foo', 'bar'])),
                              'Z': [1, 2]})
  . . . . :
   . . . . :
In [74]: right
Out [74]:
   X Z
0 foo 1
1 bar 2
In [75]: right.dtypes
Out [75]:
X
   category
Ζ
     int64
dtype: object
```

The merged result:

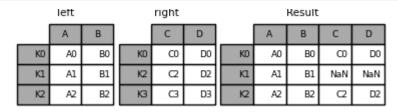
```
In [76]: result = pd.merge(left, right, how='outer')
In [77]: result
Out [77]:
         Y Z
   X
0 bar
       one 2
1 bar three 2
       one 2
2 bar
3 bar three 2
4 bar three 2
5 bar three 2
6
  foo
       one 1
7
  foo three 1
       one 1
 foo
  foo three 1
In [78]: result.dtypes
Out [78]:
    category
X
     object
Υ
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 the categories' dtype.

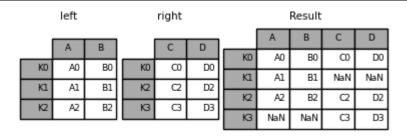
Note: Merging on category dtypes that are the same can be quite performant compared to object dtype merging.

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:

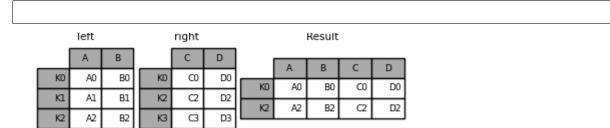


```
In [82]: result = left.join(right, how='outer')
```

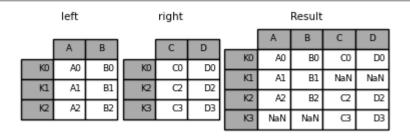


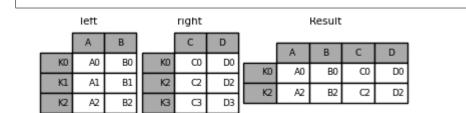
The same as above, but with how='inner'.

```
In [83]: 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:





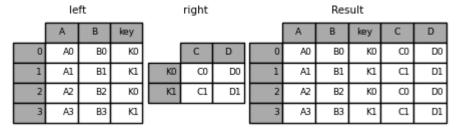
Joining key columns on an index

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				Result					
	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	00	D0	1	A1	B1	K1	C1	D1	
2	A2	B2	KO	K1	C1	D1	2	A2	B2	K0	ω	D0	
3	A3	В3	K1				3	A3	В3	K1	Cl	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 [93]: result = left.join(right, on=['key1', 'key2'])
```

		left			right					Result						
	Α	В	keyl	key2			С	D		Α	В	keyl	key2	С	D	
0	A0	BO	K0	K0	KD	KD	В	D0	0	A0	BO	K0	K0	ω	D0	
1	A1	B1	K0	K1	K1	KD	а	D1	1	A1	B1	K0	K1	NaN	NaN	
2	A2	B2	K1	K0	K2	KD	Q	D2	2	A2	B2	K1	K0	C1	D1	
3	A3	В3	K2	K1	K2	K1	В	D3	3	A3	В3	K2	K1	СЗ	D3	

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

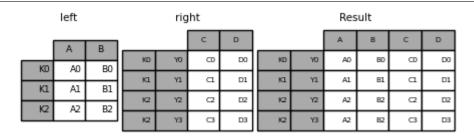
			left			right					Result						
		Α	В	keyl	key2			С	D	ا ا	А	В	keyl	key2	С	D	
	0	A0	B0	K0	KO	KD	KD	В	D0	0	A0	BO	KO	K0	0	D0	
	1	A1	B1	K0	K1	K1	KD	а	D1	- 0				-	-	-	
1	2	A2	B2	K1	K0	K2	KD	a	D2		A2	B2	K1	K0	C1	D1	
ł	3	A3	В3	K2	K1	K2	кі	З	D3	3	A3	В3	K2	K1	C3	D3	

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.

```
In [95]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                                'B': ['B0', 'B1', 'B2']},
   . . . . :
                                index=pd.Index(['K0', 'K1', 'K2'], name='key'))
   . . . . :
   . . . . :
In [96]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                                              ('K2', 'Y2'), ('K2', 'Y3')],
                                               names=['key', 'Y'])
   . . . . :
In [97]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                                 'D': ['D0', 'D1', 'D2', 'D3']},
   . . . . :
                                 index=index)
   . . . . :
In [98]: result = left.join(right, how='inner')
```



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

	left			right					Result							
	Α	В			С	D].			А	В	С	D			
10	_		KD	YO	8	D0	11	KD	YO	AD	B0	8	D0			
K	A0	B0	ка	Y1	а	D1	11	кі	Y1	Al	B1	а	D1			
K	A1	B1			_		11			- 12						
К	A2	B2	K2	Y2	(2	D2	II	K2	Υ2	A2	B2	Q	D2			
N2		K2	Y3	В	D3	$\ $	K2	Y3	A2	B2	В	D3				

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 [100]: leftindex = pd.MultiIndex.from_product([list('abc'), list('xy'), [1, 2]],
                                                   names=['abc', 'xy', 'num'])
   . . . . . :
In [101]: left = pd.DataFrame({'v1': range(12)}, index=leftindex)
In [102]: left
Out[102]:
            v1
abc xy num
  x 1
       2
             1
      1
             2
             3
       2
      1
             4
   X
       2
             5
      1
             6
   У
             7
       2
      1
             8
   Х
       2
             9
            10
   y 1
       2
            11
In [103]: rightindex = pd.MultiIndex.from_product([list('abc'), list('xy')],
                                                    names=['abc', 'xy'])
   . . . . . :
   . . . . . :
In [104]: right = pd.DataFrame({'v2': [100 * i for i in range(1, 7)]},_
→index=rightindex)
In [105]: right
Out[105]:
         v2
abc xy
  X
        100
   У
        200
        300
   Х
        400
   У
        500
   Х
        600
```

(continues on next page)

(continued from previous page)

```
In [106]: left.join(right, on=['abc', 'xy'], how='inner')
Out[106]:
          v1 v2
abc xy num
          0 100
a x 1
      2
          1 100
   У
     1
           2 200
      2
          3 200
          4 300
  x 1
          5 300
      2
   y 1
          6 400
     2
          7 400
   x 1
          8 500
     2
          9 500
   y 1
         10 600
          11 600
     2
```

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

```
In [107]: leftindex = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
                                                     ('K1', 'X2')],
                                                    names=['key', 'X'])
   . . . . . :
   . . . . . :
In [108]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                                 'B': ['B0', 'B1', 'B2']},
   . . . . . :
   . . . . . :
                                index=leftindex)
   . . . . . :
In [109]: rightindex = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                                                      ('K2', 'Y2'), ('K2', 'Y3')],
                                                     names=['key', 'Y'])
   . . . . :
   . . . . . :
In [110]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                                   'D': ['D0', 'D1', 'D2', 'D3']},
                                 index=rightindex)
   . . . . . :
   . . . . . :
In [111]: result = pd.merge(left.reset_index(), right.reset_index(),
                              on=['key'], how='inner').set_index(['key', 'X', 'Y'])
   . . . . . :
```

		le	ft			rig	ht		Result							
			А	В			С	D				А	В	-	D	
				D	KD	YO	0	D0				٤	a	J		
-[KD	XD	AD	BO					KD	XD	YO	AD	BO	8	DO	
ı					K1	Y1	a	D1								
- 1	KD	X1	A1	B1					KD	X1	YO	A1	B1	CO	D0	
ı					K2	Y2	(2	D2								
- [K1	X2	A2	B2					K1	X2	Y1	A2	B2	а	D1	
ı					K2	Y3	СЗ	D3		į		!		1		
					100	1	1									

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 [112]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')
In [113]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                                 'B': ['B0', 'B1', 'B2', 'B3'],
                                 'key2': ['K0', 'K1', 'K0', 'K1']},
   . . . . . :
                                index=left_index)
   . . . . :
   . . . . . :
In [114]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')
In [115]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                                   'D': ['D0', 'D1', 'D2', 'D3'],
   . . . . . :
                                   'key2': ['K0', 'K0', 'K0', 'K1']},
   . . . . . :
                                 index=right_index)
   . . . . . :
   . . . . . :
In [116]: result = left.merge(right, on=['key1', 'key2'])
```

		le	eft			rig	ht		Result						
		Α	В	key2		С	D	key2	ا ا	Α	В	key2	С	D	
	KO	A0	В0	K0	KO	α	D0	KO	VO.						
Ì	KO	A1	B1	K1	K1	Cl	D1	K0	KO	A0	B0	K0	00	D0	
ł	K1	A2	B2	KΩ	K2	(2	D2	KO	K1	A2	B2	K0	C1	D1	
ł		A3	B3	K1		_	-	-	K2	A3	В3	K1	СЗ	D3	
l	K2	A3	B3	KT.	K2 C3 I	D3	D3 K1								

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.

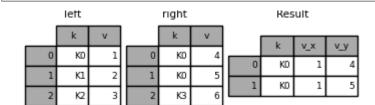
Note: When DataFrames are merged using only some of the levels of a *MultiIndex*, the extra levels will be dropped from the resulting merge. In order to preserve those levels, use reset_index on those level names to move those levels to columns prior to doing the merge.

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.

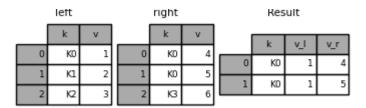
Overlapping value columns

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 [117]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})
In [118]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})
In [119]: result = pd.merge(left, right, on='k')
```

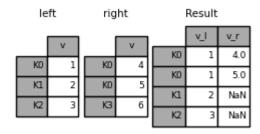


```
In [120]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```



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

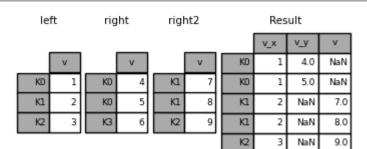
```
In [121]: left = left.set_index('k')
In [122]: right = right.set_index('k')
In [123]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```



Joining multiple DataFrames

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

```
In [124]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])
In [125]: result = left.join([right, right2])
```

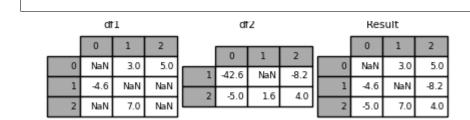


Merging together values within Series or DataFrame columns

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:

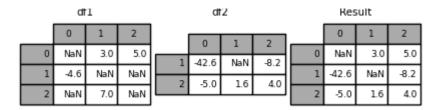
For this, use the <code>combine_first()</code> method:

```
In [128]: result = df1.combine_first(df2)
```



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 in place:

```
In [129]: df1.update(df2)
```



2.4.3 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 [130]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
                              'lv': [1, 2, 3, 4],
  . . . . . :
                              's': ['a', 'b', 'c', 'd']})
  . . . . . :
   . . . . . :
In [131]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
                              'rv': [1, 2, 3]})
   . . . . . :
In [132]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')
Out[132]:
    k lv s rv
   K0 1.0 a NaN
0
   K1 1.0 a 1.0
1
2
   K2 1.0 a 2.0
3
   K4 1.0 a 3.0
   K1 2.0 b 1.0
4
5
   K2 2.0 b 2.0
   K4 2.0 b 3.0
6
7
       3.0 c 1.0
   K1
8
       3.0 c 2.0
   K2
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

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.

(continues on next page)