User Guide > Merge,...

Merge, join, concatenate and compare

pandas provides various methods for combining and comparing `Series` or `DataFrame`.

- `concat()`: Merge multiple `Series` or `DataFrame` objects along a shared index or column
- `DataFrame.join()`: Merge multiple `DataFrame` objects along the columns
- `DataFrame.combine_first()`: Update missing values with non-missing values in the same location
- `merge()`: Combine two `Series` or `DataFrame` objects with SQL-style joining
- `merge_ordered()`: Combine two `Series` or `DataFrame` objects along an ordered axis
- `merge_asof()`: Combine two `Series` or `DataFrame` objects by near instead of exact matching keys
- `Series.compare()` and `DataFrame.compare()`: Show differences in values between two `Series` or `DataFrame` objects

`concat()`

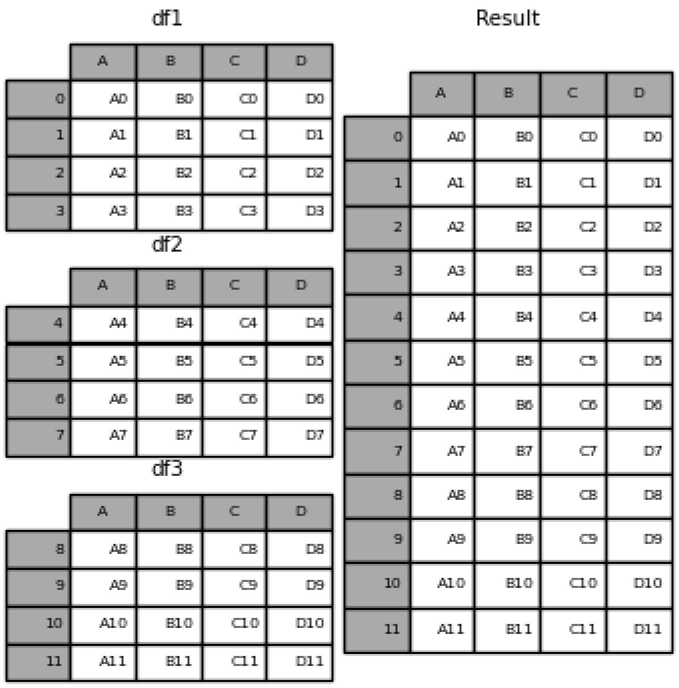
The `concat()` function concatenates an arbitrary amount of `Series` or `DataFrame` objects along an axis while performing optional set logic (union or intersection) of the indexes on the other axes. Like `numpy.concatenate`, `concat()` takes a list or dict of homogeneously-typed objects and concatenates them.

```
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],
   ...: )
   ...:
```

Skip to main content

```
   ...:     {
   ...:         "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)

In [6]: result
Out[6]:
      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
```

Skip to main content



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

`concat()` makes a full copy of the data, and iteratively reusing `concat()` can create unnecessary copies. Collect all `DataFrame` or `Series` objects in a list before using `concat()`.

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

Note

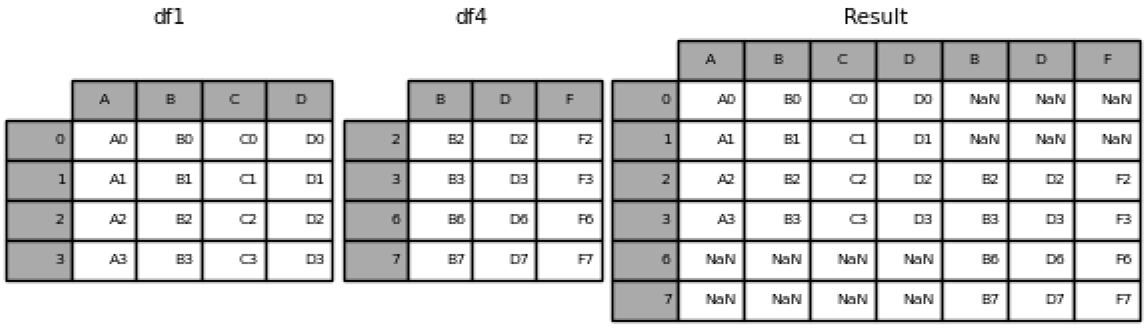
When concatenating `DataFrame` with named axes, pandas will attempt to preserve these index/column names whenever possible. In the case where all inputs share a common name, this name will be assigned to the result. When the input names do not all agree, the result will be unnamed. The same is true for `MultiIndex`, but the logic is applied separately on a level-by-level basis.

Joining logic of the resulting axis

Skip to main content

`join='outer'` takes the union of all axis values

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

In [8]: result = pd.concat([df1, df4], axis=1)

In [9]: result
Out[9]:
     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
```



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

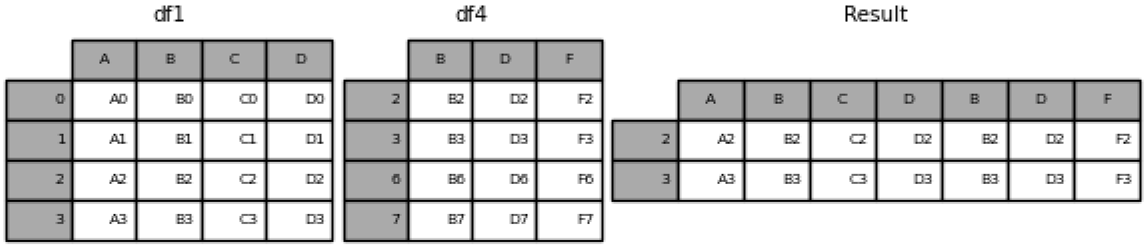
Result

	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

`join='inner'` takes the intersection of the axis values

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

In [11]: result
Out[11]:
    A   B   C   D   B   D   F
2  A2  B2  C2  D2  B2  D2  F2
3  A3  B3  C3  D3  B3  D3  F3
```

Skip to main content



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

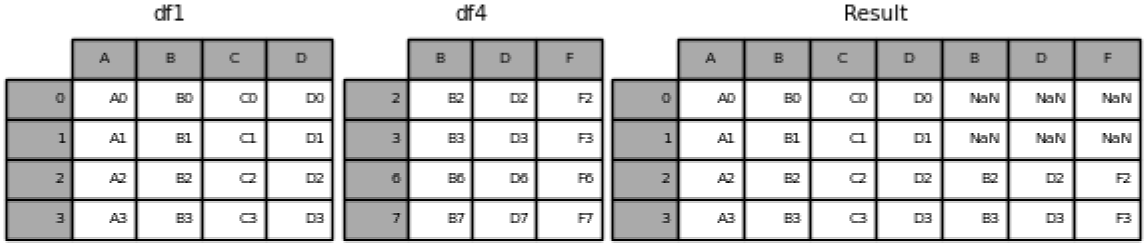
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

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

To perform an effective “left” join using the *exact index* from the original `DataFrame`, result can be reindexed.

```
In [12]: result = pd.concat([df1, df4], axis=1).reindex(df1.index)

In [13]: result
Out[13]: 
    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
```



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

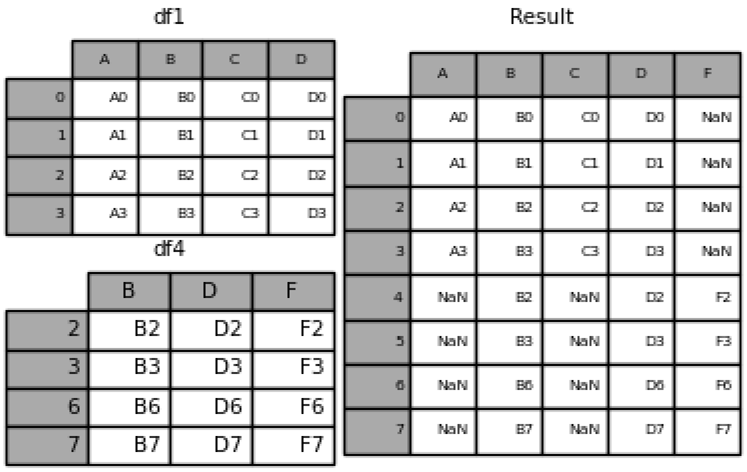
Ignoring indexes on the concatenation axis

For `DataFrame` objects which don’t have a meaningful index, the `ignore_index` ignores overlapping indexes.

```
In [14]: result = pd.concat([df1, df4], ignore_index=True, sort=False)

In [15]: result
Out[15]: 
     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
```

Skip to main content

```
6  NaN  B6  NaN  D6   F6
7  NaN  B7  NaN  D7   F7
```



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

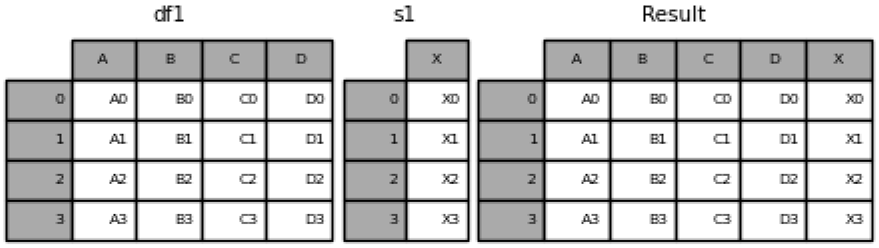
	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Concatenating `Series` and `DataFrame` together

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 [16]: s1 = pd.Series(["X0", "X1", "X2", "X3"], name="X")

In [17]: result = pd.concat([df1, s1], axis=1)

In [18]: result
Out[18]: 
    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
```

Skip to main content



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

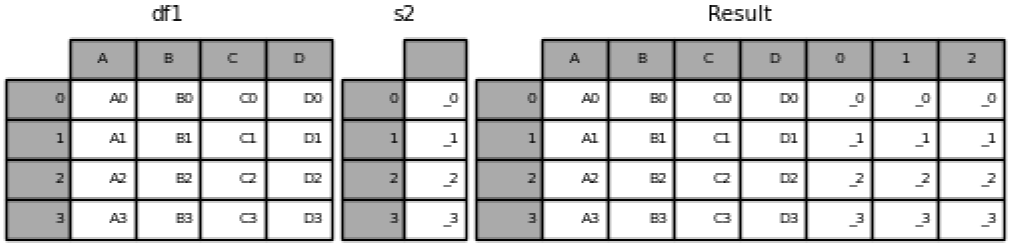
	X
0	X0
1	X1
2	X2
3	X3

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

Unnamed `Series` will be numbered consecutively.

```
In [19]: s2 = pd.Series(["_0", "_1", "_2", "_3"])

In [20]: result = pd.concat([df1, s2, s2, s2], axis=1)

In [21]: result
Out[21]:
    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
```



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

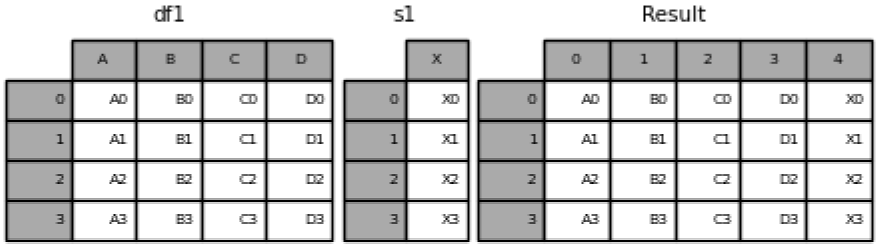
Result

	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

`ignore_index=True` will drop all name references.

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

In [23]: result
Out[23]:
    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
```

Skip to main content



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

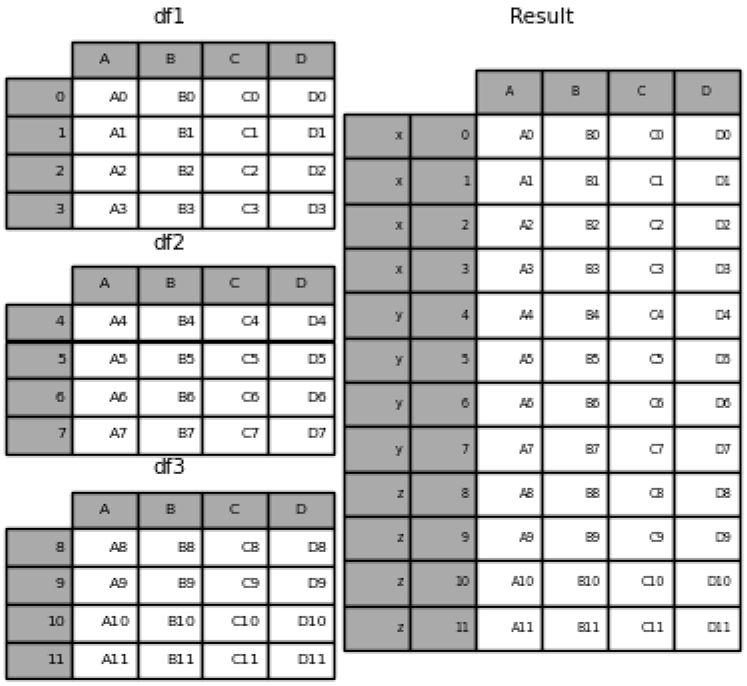
	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

Resulting `keys`

The `keys` argument adds another axis level to the resulting index or column (creating a `MultiIndex`) associate specific keys with each original `DataFrame`.

```
In [24]: result = pd.concat(frames, keys=["x", "y", "z"])

In [25]: result
Out[25]:
        A    B    C    D
x 0    A0   B0   C0   D0
  1    A1   B1   C1   D1
  2    A2   B2   C2   D2
  3    A3   B3   C3   D3
y 4    A4   B4   C4   D4
  5    A5   B5   C5   D5
  6    A6   B6   C6   D6
  7    A7   B7   C7   D7
z 8    A8   B8   C8   D8
  9    A9   B9   C9   D9
  10  A10  B10  C10  D10
  11  A11  B11  C11  D11

In [26]: result.loc["y"]
Out[26]:
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

Skip to main content



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

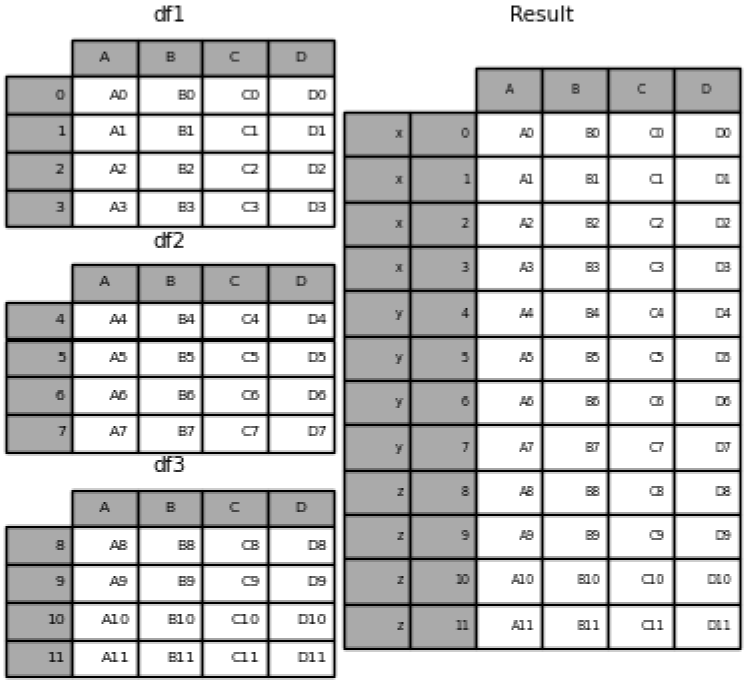
The `keys` argument cane override the column names when creating a new `DataFrame` based on existing `Series`.

```
In [27]: s3 = pd.Series([0, 1, 2, 3], name="foo")

In [28]: s4 = pd.Series([0, 1, 2, 3])

In [29]: s5 = pd.Series([0, 1, 4, 5])

In [30]: pd.concat([s3, s4, s5], axis=1)
Out[30]: 
   foo  0  1
0    0  0  0
1    1  1  1
2    2  2  4
3    3  3  5

In [31]: pd.concat([s3, s4, s5], axis=1, keys=["red", "blue", "yellow"])
Out[31]: 
   red  blue  yellow
0    0     0       0
1    1     1       1
2    2     2       4
3    3     3       5
```

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

Skip to main content

```
In [32]: pieces = {"x": df1, "y": df2, "z": df3}

In [33]: result = pd.concat(pieces)

In [34]: result
Out[34]: 
        A    B    C    D
x 0    A0   B0   C0   D0
  1    A1   B1   C1   D1
  2    A2   B2   C2   D2
  3    A3   B3   C3   D3
y 4    A4   B4   C4   D4
  5    A5   B5   C5   D5
  6    A6   B6   C6   D6
  7    A7   B7   C7   D7
z 8    A8   B8   C8   D8
  9    A9   B9   C9   D9
  10  A10  B10  C10  D10
  11  A11  B11  C11  D11
```



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

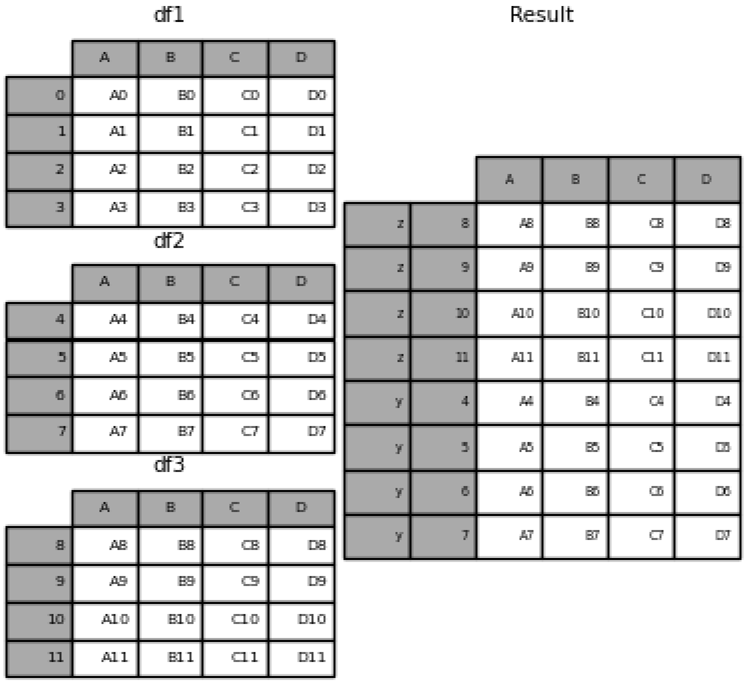
Result

		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 [35]: result = pd.concat(pieces, keys=["z", "y"])

In [36]: result
Out[36]: 
        A    B    C    D
```

Skip to main content

```
  10  A10  B10  C10  D10
  11  A11  B11  C11  D11
y 4    A4   B4   C4   D4
  5    A5   B5   C5   D5
  6    A6   B6   C6   D6
  7    A7   B7   C7   D7
```



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:

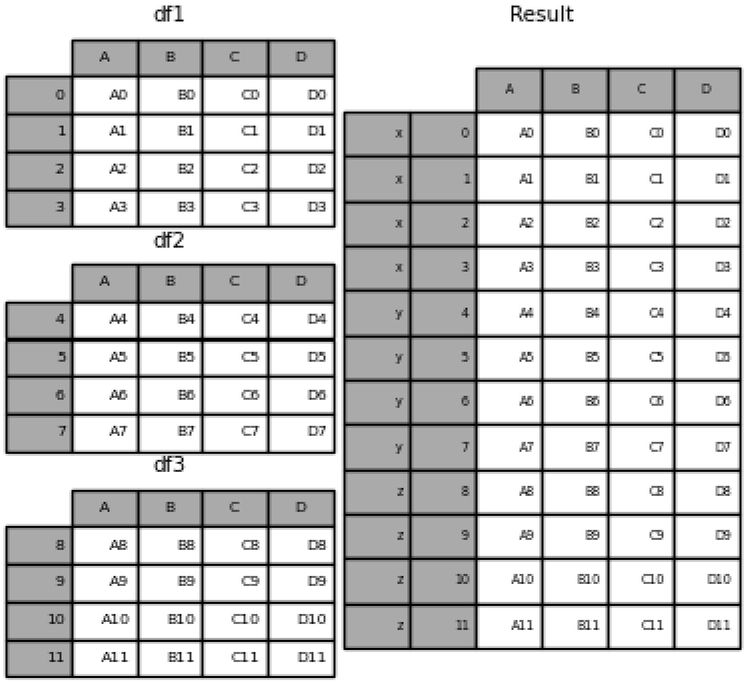
```
In [37]: result.index.levels
Out[37]: FrozenList([['z', 'y'], [4, 5, 6, 7, 8, 9, 10, 11]])
```

`levels` argument allows specifying resulting levels associated with the `keys`

```
In [38]: result = pd.concat(
   ....:     pieces, keys=["x", "y", "z"], levels=[["z", "y", "x", "w"]], names=["
   ....: )
   ....:

In [39]: result
Out[39]:
                A    B    C    D
```

Skip to main content

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



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 [40]: result.index.levels
Out[40]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
```

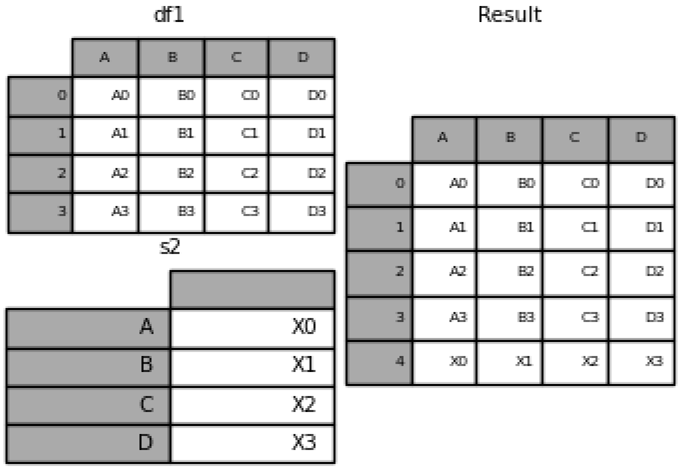
Appending rows to a `DataFrame`

If you have a `Series` that you want to append as a single row to a `DataFrame`, you can convert the row into a `DataFrame` and use `concat()`

```
In [41]: s2 = pd.Series(["X0", "X1", "X2", "X3"], index=["A", "B", "C", "D"])
```

Skip to main content

```
In [42]: result = pd.concat([df1, s2.to_frame().T], ignore_index=True)

In [43]: result
Out[43]: 
    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
```



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

`merge()`

`merge()` performs join operations similar to relational databases like SQL. Users who are familiar with SQL but new to pandas can reference a comparison with SQL.

Merge types

`merge()` implements common SQL style joining operations.

- **one-to-one**: joining two `DataFrame` objects on their indexes which must contain unique values.
- **many-to-one**: joining a unique index to one or more columns in a different `DataFrame`.
- **many-to-many** : joining columns on columns.

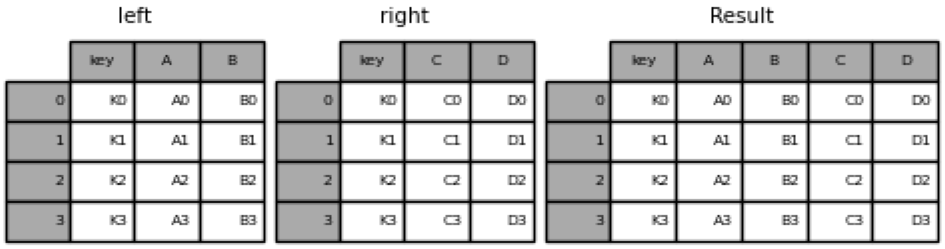
Skip to main content

Note

When joining columns on columns, potentially a many-to-many join, any indexes on the passed `DataFrame` objects **will be discarded**.

For a **many-to-many** join, if a key combination appears more than once in both tables, the `DataFrame` will have the **Cartesian product** of the associated data.

```
In [44]: left = pd.DataFrame(
   ....:     {
   ....:         "key": ["K0", "K1", "K2", "K3"],
   ....:         "A": ["A0", "A1", "A2", "A3"],
   ....:         "B": ["B0", "B1", "B2", "B3"],
   ....:     }
   ....: )
   ....:

In [45]: right = pd.DataFrame(
   ....:     {
   ....:         "key": ["K0", "K1", "K2", "K3"],
   ....:         "C": ["C0", "C1", "C2", "C3"],
   ....:         "D": ["D0", "D1", "D2", "D3"],
   ....:     }
   ....: )
   ....:

In [46]: result = pd.merge(left, right, on="key")

In [47]: result
Out[47]:
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```



left

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

right

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

Result

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

The `how` argument to `merge()` specifies which keys are included in the resulting table. If a key

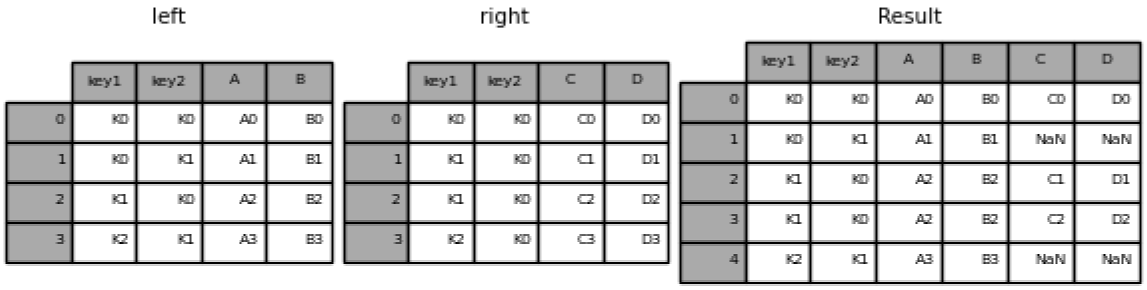
Skip to main content

`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
`cross`	`CROSS JOIN`	Create the cartesian product of rows of both frames

```
In [48]: left = pd.DataFrame(
   ....:     {
   ....:         "key1": ["K0", "K0", "K1", "K2"],
   ....:         "key2": ["K0", "K1", "K0", "K1"],
   ....:         "A": ["A0", "A1", "A2", "A3"],
   ....:         "B": ["B0", "B1", "B2", "B3"],
   ....:     }
   ....: )
   ....:

In [49]: right = pd.DataFrame(
   ....:     {
   ....:         "key1": ["K0", "K1", "K1", "K2"],
   ....:         "key2": ["K0", "K0", "K0", "K0"],
   ....:         "C": ["C0", "C1", "C2", "C3"],
   ....:         "D": ["D0", "D1", "D2", "D3"],
   ....:     }
   ....: )
   ....:

In [50]: result = pd.merge(left, right, how="left", on=["key1", "key2"])

In [51]: result
Out[51]:
  key1 key2   A   B    C    D
0   K0   K0  A0  B0   C0   D0
1   K0   K1  A1  B1  NaN  NaN
2   K1   K0  A2  B2   C1   D1
3   K1   K0  A2  B2   C2   D2
4   K2   K1  A3  B3  NaN  NaN
```

Skip to main content



left

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

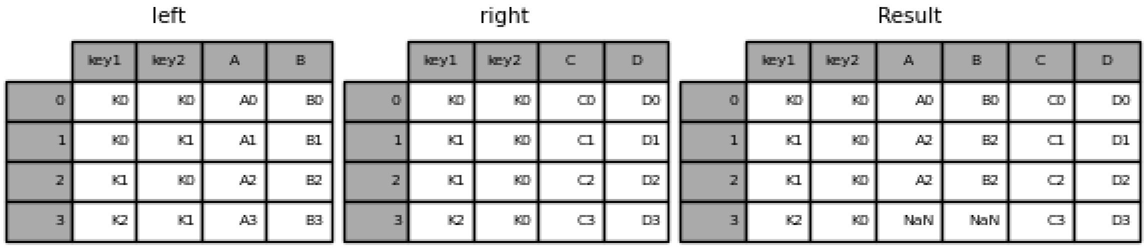
right

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

Result

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

```
In [52]: result = pd.merge(left, right, how="right", on=["key1", "key2"])

In [53]: result
Out[53]:
  key1 key2    A    B   C   D
0   K0   K0   A0   B0  C0  D0
1   K1   K0   A2   B2  C1  D1
2   K1   K0   A2   B2  C2  D2
3   K2   K0  NaN  NaN  C3  D3
```



left

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

right

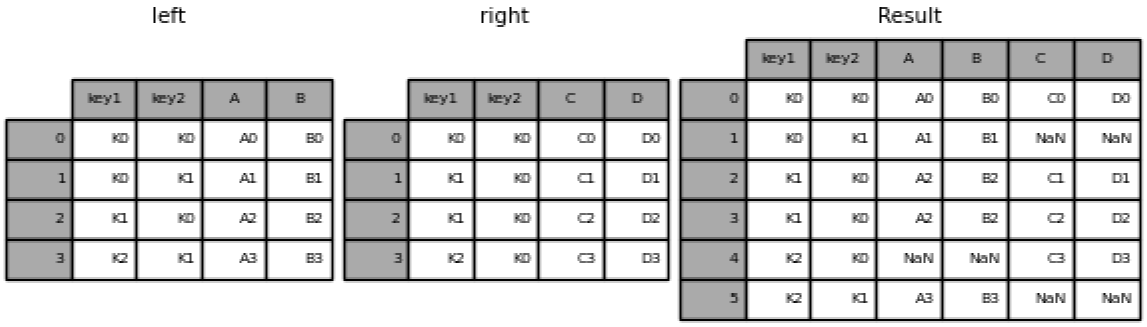
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

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

```
In [54]: result = pd.merge(left, right, how="outer", on=["key1", "key2"])

In [55]: result
Out[55]:
  key1 key2    A    B    C    D
0   K0   K0   A0   B0   C0   D0
1   K0   K1   A1   B1  NaN  NaN
2   K1   K0   A2   B2   C1   D1
3   K1   K0   A2   B2   C2   D2
4   K2   K0  NaN  NaN   C3   D3
5   K2   K1   A3   B3  NaN  NaN
```

Skip to main content



left

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

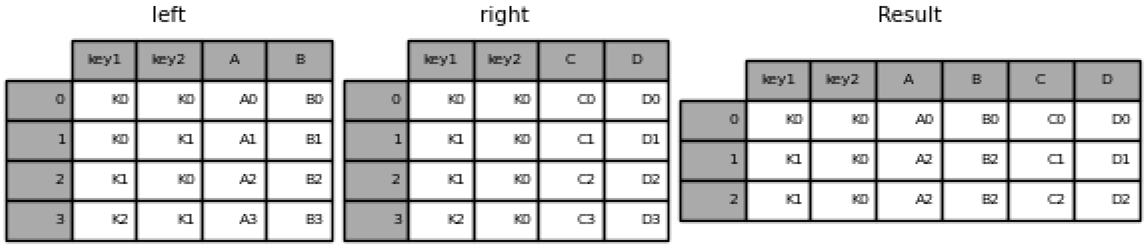
right

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

Result

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

```
In [56]: result = pd.merge(left, right, how="inner", on=["key1", "key2"])

In [57]: result
Out[57]: 
  key1 key2   A   B   C   D
0   K0   K0  A0  B0  C0  D0
1   K1   K0  A2  B2  C1  D1
2   K1   K0  A2  B2  C2  D2
```



left

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

right

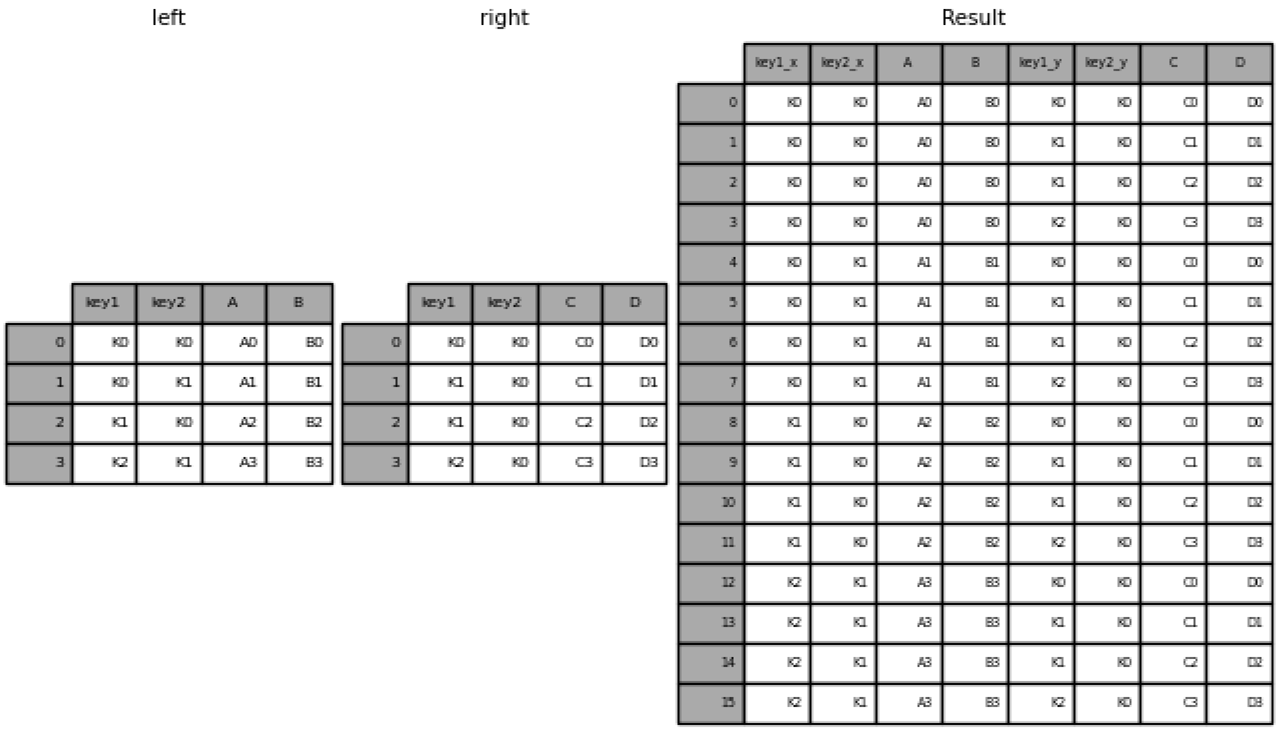
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

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

```
In [58]: result = pd.merge(left, right, how="cross")

In [59]: result
Out[59]: 
   key1_x key2_x   A   B key1_y key2_y   C   D
0      K0     K0  A0  B0     K0     K0  C0  D0
1      K0     K0  A0  B0     K1     K0  C1  D1
2      K0     K0  A0  B0     K1     K0  C2  D2
3      K0     K0  A0  B0     K2     K0  C3  D3
4      K0     K1  A1  B1     K0     K0  C0  D0
..    ...    ...  ..  ..    ...    ...  ..  ..
11     K1     K0  A2  B2     K2     K0  C3  D3
12     K2     K1  A3  B3     K0     K0  C0  D0
13     K2     K1  A3  B3     K1     K0  C1  D1
14     K2     K1  A3  B3     K1     K0  C2  D2
15     K2     K1  A3  B3     K2     K0  C3  D3

[16 rows x 8 columns]
```

Skip to main content



left

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

right

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

Result

	key1_x	key2_x	A	B	key1_y	key2_y	C	D
0	K0	K0	A0	B0	K0	K0	C0	D0
1	K0	K0	A0	B0	K1	K0	C1	D1
2	K0	K0	A0	B0	K1	K0	C2	D2
3	K0	K0	A0	B0	K2	K0	C3	D3
4	K0	K1	A1	B1	K0	K0	C0	D0
5	K0	K1	A1	B1	K1	K0	C1	D1
6	K0	K1	A1	B1	K1	K0	C2	D2
7	K0	K1	A1	B1	K2	K0	C3	D3
8	K1	K0	A2	B2	K0	K0	C0	D0
9	K1	K0	A2	B2	K1	K0	C1	D1
10	K1	K0	A2	B2	K1	K0	C2	D2
11	K1	K0	A2	B2	K2	K0	C3	D3
12	K2	K1	A3	B3	K0	K0	C0	D0
13	K2	K1	A3	B3	K1	K0	C1	D1
14	K2	K1	A3	B3	K1	K0	C2	D2
15	K2	K1	A3	B3	K2	K0	C3	D3

You can `Series` and a `DataFrame` with a `MultiIndex` if the names of the `MultiIndex` correspond to the columns from the `DataFrame`. Transform the `Series` to a `DataFrame` using `Series.reset_index()` before merging

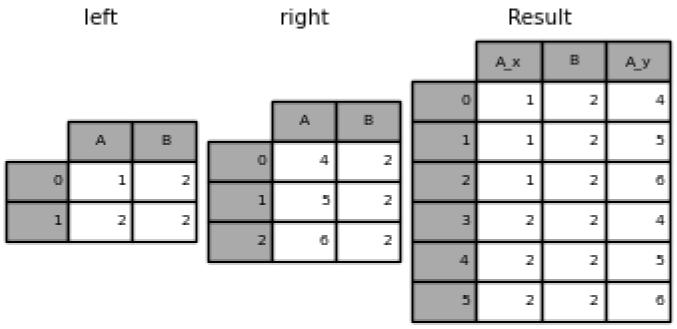
```
In [60]: df = pd.DataFrame({"Let": ["A", "B", "C"], "Num": [1, 2, 3]})

In [61]: df
Out[61]: 
  Let  Num
0   A    1
1   B    2
2   C    3

In [62]: ser = pd.Series(
   ....:     ["a", "b", "c", "d", "e", "f"],
   ....:     index=pd.MultiIndex.from_arrays(
   ....:         [["A", "B", "C"] * 2, [1, 2, 3, 4, 5, 6]], names=["Let", "Num"]
   ....:     ),
   ....: )
   ....: 

In [63]: ser
Out[63]: 
```

Skip to main content

```
B    2      b
C    3      c
A    4      d
B    5      e
C    6      f
dtype: object

In [64]: pd.merge(df, ser.reset_index(), on=["Let", "Num"])
Out[64]: 
  Let  Num  0
0   A    1  a
1   B    2  b
2   C    3  c
```

Performing an outer join with duplicate join keys in `DataFrame`

```
In [65]: left = pd.DataFrame({"A": [1, 2], "B": [2, 2]})

In [66]: right = pd.DataFrame({"A": [4, 5, 6], "B": [2, 2, 2]})

In [67]: result = pd.merge(left, right, on="B", how="outer")

In [68]: result
Out[68]: 
   A_x  B  A_y
0    1  2    4
1    1  2    5
2    1  2    6
3    2  2    4
4    2  2    5
5    2  2    6
```



left

	A	B
0	1	2
1	2	2

right

	A	B
0	4	2
1	5	2
2	6	2

Result

	A_x	B	A_y
0	1	2	4
1	1	2	5
2	1	2	6
3	2	2	4
4	2	2	5
5	2	2	6

Skip to main content

Warning

Merging on duplicate keys significantly increase the dimensions of the result and can cause a memory overflow.

Merge key uniqueness

The `validate` argument checks whether the uniqueness of merge keys. Key uniqueness is checked before merge operations and can protect against memory overflows and unexpected key duplication.

```
In [69]: left = pd.DataFrame({"A": [1, 2], "B": [1, 2]})

In [70]: right = pd.DataFrame({"A": [4, 5, 6], "B": [2, 2, 2]})

In [71]: result = pd.merge(left, right, on="B", how="outer", validate="one_to_one"
---------------------------------------------------------------------------
MergeError                                Traceback (most recent call last)
Cell In[71], line 1
----> 1 result = pd.merge(left, right, on="B", how="outer", validate="one_to_one")

File ~/work/pandas/pandas/pandas/core/reshape/merge.py:170, in merge(left, right,
    155     return _cross_merge(
    156         left_df,
    157         right_df,
   (...)
    167         copy=copy,
    168     )
    169 else:
--> 170     op = _MergeOperation(
    171         left_df,
    172         right_df,
    173         how=how,
    174         on=on,
    175         left_on=left_on,
    176         right_on=right_on,
    177         left_index=left_index,
    178         right_index=right_index,
    179         sort=sort,
    180         suffixes=suffixes,
    181         indicator=indicator,
    182         validate=validate,
    183     )
    184     return op.get_result(copy=copy)

File ~/work/pandas/pandas/pandas/core/reshape/merge.py:813, in _MergeOperation.__i
    809 # If argument passed to validate,
```

Skip to main content

```
    811 # are in fact unique.
    812 if validate is not None:
--> 813     self._validate_validate_kwd(validate)

File ~/work/pandas/pandas/pandas/core/reshape/merge.py:1658, in _MergeOperation._v
   1654         raise MergeError(
   1655             "Merge keys are not unique in left dataset; not a one-to-one m
   1656         )
   1657     if not right_unique:
-> 1658         raise MergeError(
   1659             "Merge keys are not unique in right dataset; not a one-to-one
   1660         )
   1662 elif validate in ["one_to_many", "1:m"]:
   1663     if not left_unique:

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 [72]: pd.merge(left, right, on="B", how="outer", validate="one_to_many")
Out[72]:
   A_x  B  A_y
0    1  1  NaN
1    2  2  4.0
2    2  2  5.0
3    2  2  6.0
```

Merge result 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`

Skip to main content

```
In [73]: df1 = pd.DataFrame({"col1": [0, 1], "col_left": ["a", "b"]})

In [74]: df2 = pd.DataFrame({"col1": [1, 2, 2], "col_right": [2, 2, 2]})

In [75]: pd.merge(df1, df2, on="col1", how="outer", indicator=True)
Out[75]:
   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
```

A string argument to `indicator` will use the value as the name for the indicator column.

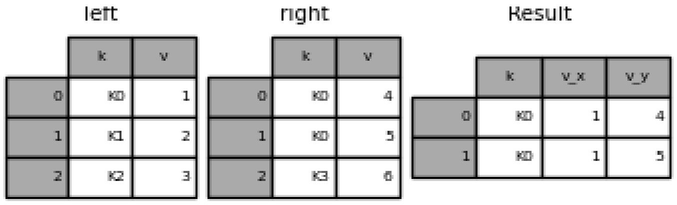
```
In [76]: pd.merge(df1, df2, on="col1", how="outer", indicator="indicator_column")
Out[76]:
   col1 col_left  col_right indicator_column
0     0        a        NaN        left_only
1     1        b        2.0             both
2     2      NaN        2.0       right_only
3     2      NaN        2.0       right_only
```

Overlapping value columns

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

```
In [77]: left = pd.DataFrame({"k": ["K0", "K1", "K2"], "v": [1, 2, 3]})

In [78]: right = pd.DataFrame({"k": ["K0", "K0", "K3"], "v": [4, 5, 6]})

In [79]: result = pd.merge(left, right, on="k")

In [80]: result
Out[80]:
    k  v_x  v_y
0  K0    1    4
1  K0    1    5
```

Skip to main content



left

	k	v
0	K0	1
1	K1	2
2	K2	3

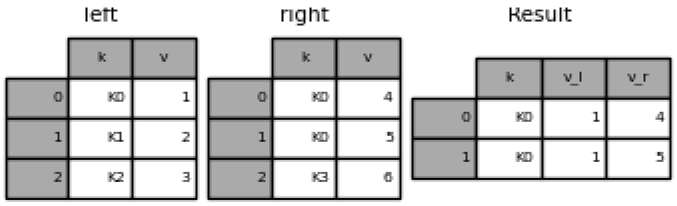
right

	k	v
0	K0	4
1	K0	5
2	K3	6

Result

	k	v_x	v_y
0	K0	1	4
1	K0	1	5

```
In [81]: result = pd.merge(left, right, on="k", suffixes=("_l", "_r"))

In [82]: result
Out[82]: 
    k  v_l  v_r
0  K0    1    4
1  K0    1    5
```



left

	k	v
0	K0	1
1	K1	2
2	K2	3

right

	k	v
0	K0	4
1	K0	5
2	K3	6

Result

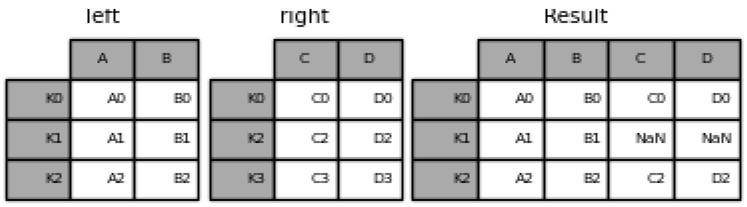
	k	v_l	v_r
0	K0	1	4
1	K0	1	5

DataFrame.join()

DataFrame.join() combines the columns of multiple, potentially differently-indexed DataFrame into a single result DataFrame.

```
In [83]: left = pd.DataFrame(
   ....:     {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"]}, index=["K0", "K1"
   ....: )
   ....: 

In [84]: right = pd.DataFrame(
   ....:     {"C": ["C0", "C2", "C3"], "D": ["D0", "D2", "D3"]}, index=["K0", "K2"
   ....: )
   ....: 

In [85]: result = left.join(right)

In [86]: result
Out[86]: 
     A   B    C    D
K0  A0  B0   C0   D0
K1  A1  B1  NaN  NaN
```

Skip to main content



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

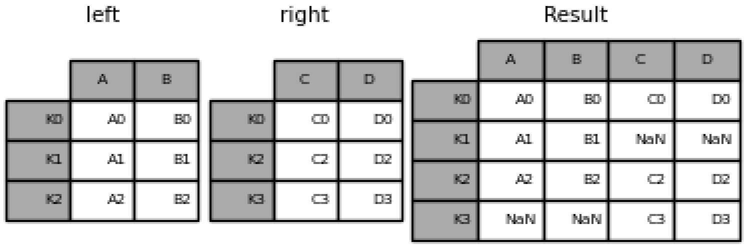
right

	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 [87]: result = left.join(right, how="outer")

In [88]: result
Out[88]: 
      A    B    C    D
K0   A0   B0   C0   D0
K1   A1   B1  NaN  NaN
K2   A2   B2   C2   D2
K3  NaN  NaN   C3   D3
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

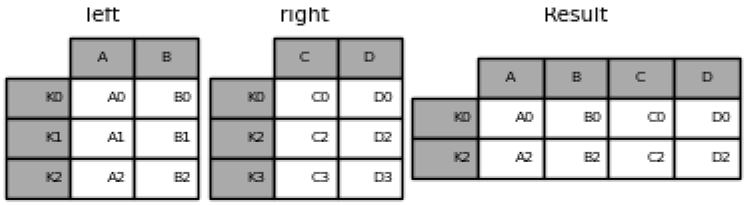
right

	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 [89]: result = left.join(right, how="inner")

In [90]: result
Out[90]: 
     A   B   C   D
K0  A0  B0  C0  D0
K2  A2  B2  C2  D2
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

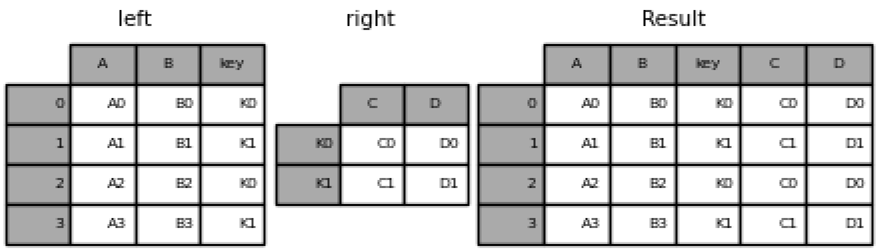
Result

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

Skip to main content

`DataFrame.join()` takes an optional `on` argument which may be a column or multiple column names that the passed `DataFrame` is to be aligned.

```
In [91]: left = pd.DataFrame(
   ....:     {
   ....:         "A": ["A0", "A1", "A2", "A3"],
   ....:         "B": ["B0", "B1", "B2", "B3"],
   ....:         "key": ["K0", "K1", "K0", "K1"],
   ....:     }
   ....: )
   ....:

In [92]: right = pd.DataFrame({"C": ["C0", "C1"], "D": ["D0", "D1"]}, index=["K0",

In [93]: result = left.join(right, on="key")

In [94]: result
Out[94]:
    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
```



left

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

right

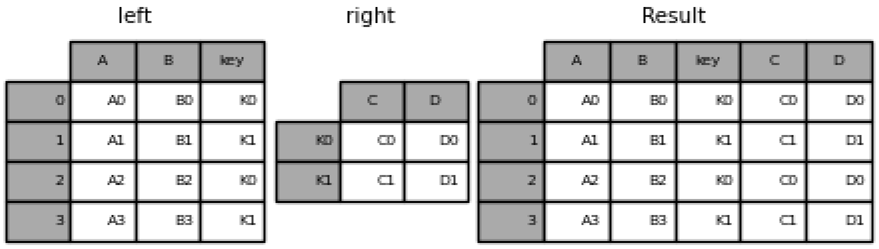
	C	D
K0	C0	D0
K1	C1	D1

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 [95]: result = pd.merge(
   ....:     left, right, left_on="key", right_index=True, how="left", sort=False
   ....: )
   ....:

In [96]: result
Out[96]:
    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
```

Skip to main content



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

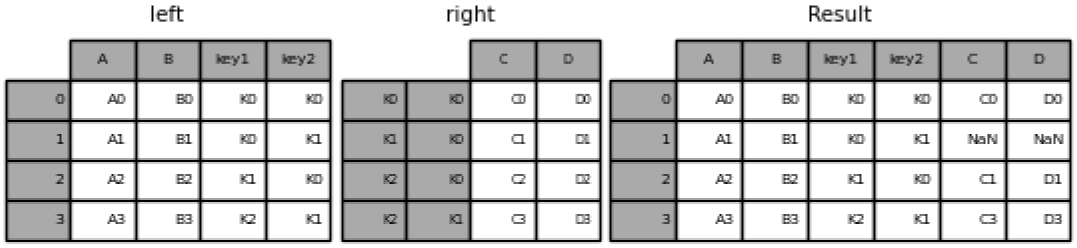
Result

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

In [98]: index = pd.MultiIndex.from_tuples(
   ....:     [("K0", "K0"), ("K1", "K0"), ("K2", "K0"), ("K2", "K1")]
   ....: )
   ....:

In [99]: right = pd.DataFrame(
   ....:     {"C": ["C0", "C1", "C2", "C3"], "D": ["D0", "D1", "D2", "D3"]}, index
   ....: )
   ....:

In [100]: result = left.join(right, on=["key1", "key2"])

In [101]: result
Out[101]:
    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
```

Skip to main content



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

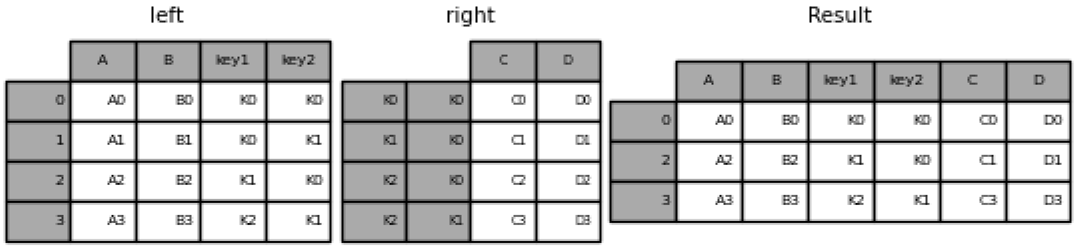
		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

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 which uses only the keys found in the calling `DataFrame`. Other join types can be specified with `how`.

```
In [102]: result = left.join(right, on=["key1", "key2"], how="inner")

In [103]: result
Out[103]: 
    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
```



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

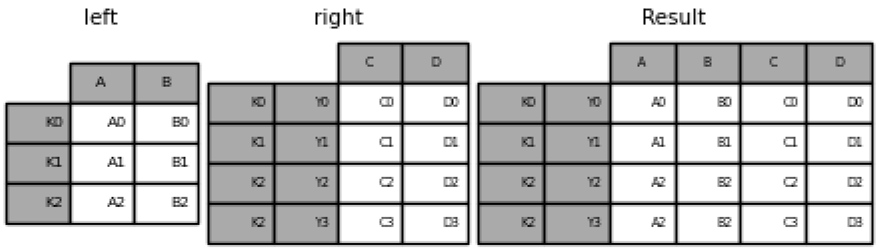
Joining a single Index to a MultiIndex

You can join a `DataFrame` with a `Index` to a `DataFrame` with a `MultiIndex` on a level. The `name` of the `Index` with match the level name of the `MultiIndex`.

```
In [104]: left = pd.DataFrame(
   .....:     {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"]},
   .....:     index=pd.Index(["K0", "K1", "K2"], name="key"),
   .....: )
   .....: 

In [105]: index = pd.MultiIndex.from_tuples(
   .....:     [("K0", "Y0"), ("K1", "Y1"), ("K2", "Y2"), ("K2", "Y3")],
   .....:     names=["key", "Y"],
   .....: )
```

Skip to main content

```
In [106]: right = pd.DataFrame(
   .....:     {"C": ["C0", "C1", "C2", "C3"], "D": ["D0", "D1", "D2", "D3"]},
   .....:     index=index,
   .....: )
   .....:

In [107]: result = left.join(right, how="inner")

In [108]: result
Out[108]:
          A   B   C   D
key Y
K0  Y0  A0  B0  C0  D0
K1  Y1  A1  B1  C1  D1
K2  Y2  A2  B2  C2  D2
    Y3  A2  B2  C3  D3
```



left

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

right

		C	D
K0	Y0	C0	D0
K1	Y1	C1	D1
K2	Y2	C2	D2
K2	Y3	C3	D3

Result

		A	B	C	D
K0	Y0	A0	B0	C0	D0
K1	Y1	A1	B1	C1	D1
K2	Y2	A2	B2	C2	D2
K2	Y3	A2	B2	C3	D3

Joining with two `MultiIndex`

The `MultiIndex` of the input argument must be completely used in the join and is a subset of the indices in the left argument.

```
In [109]: leftindex = pd.MultiIndex.from_product(
   .....:     [list("abc"), list("xy"), [1, 2]], names=["abc", "xy", "num"]
   .....: )
   .....:

In [110]: left = pd.DataFrame({"v1": range(12)}, index=leftindex)

In [111]: left
Out[111]:
            v1
abc xy num
a   x  1     0
       2     1
    y  1     2
       2     3
```

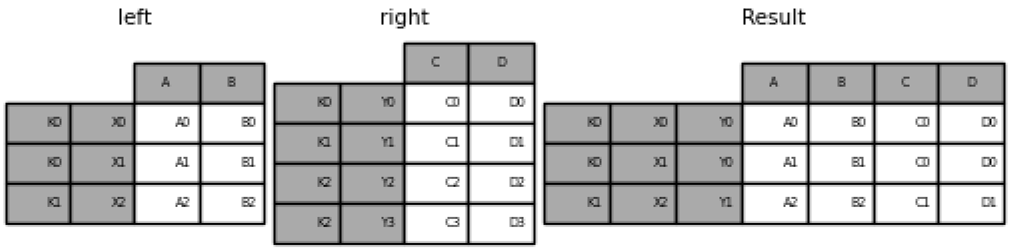
Skip to main content

```
       2     5
    y  1     6
       2     7
c   x  1     8
       2     9
    y  1    10
       2    11

In [112]: rightindex = pd.MultiIndex.from_product(
   .....:     [list("abc"), list("xy")], names=["abc", "xy"]
   .....: )
   .....:

In [113]: right = pd.DataFrame({"v2": [100 * i for i in range(1, 7)]}, index=right

In [114]: right
Out[114]: 
         v2
abc xy     
a   x   100
    y   200
b   x   300
    y   400
c   x   500
    y   600

In [115]: left.join(right, on=["abc", "xy"], how="inner")
Out[115]: 
            v1   v2
abc xy num         
a   x  1     0  100
       2     1  100
    y  1     2  200
       2     3  200
b   x  1     4  300
       2     5  300
    y  1     6  400
       2     7  400
c   x  1     8  500
       2     9  500
    y  1    10  600
       2    11  600
```

```
In [116]: leftindex = pd.MultiIndex.from_tuples(
   .....:     [("K0", "X0"), ("K0", "X1"), ("K1", "X2")], names=["key", "X"]
   .....: )
   .....:

In [117]: left = pd.DataFrame(
   .....:     {"A": ["A0", "A1", "A2"], "B": ["B0", "B1", "B2"]}, index=leftindex
   .....: )
   .....:
```

Skip to main content

```
   .....:     [("K0", "Y0"), ("K1", "Y1"), ("K2", "Y2"), ("K2", "Y3")], names=["ke
   .....: )
   .....: 

In [119]: right = pd.DataFrame(
   .....:     {"C": ["C0", "C1", "C2", "C3"], "D": ["D0", "D1", "D2", "D3"]}, inde
   .....: )
   .....: 

In [120]: result = pd.merge(
   .....:     left.reset_index(), right.reset_index(), on=["key"], how="inner"
   .....: ).set_index(["key", "X", "Y"])
   .....: 

In [121]: result
Out[121]: 
           A   B   C   D
key X  Y                
K0  X0 Y0  A0  B0  C0  D0
    X1 Y0  A1  B1  C0  D0
K1  X2 Y1  A2  B2  C1  D1
```



left

		A	B
K0	X0	A0	B0
K0	X1	A1	B1
K1	X2	A2	B2

right

		C	D
K0	Y0	C0	D0
K1	Y1	C1	D1
K2	Y2	C2	D2
K2	Y3	C3	D3

Result

			A	B	C	D
K0	X0	Y0	A0	B0	C0	D0
K0	X1	Y0	A1	B1	C0	D0
K1	X2	Y1	A2	B2	C1	D1

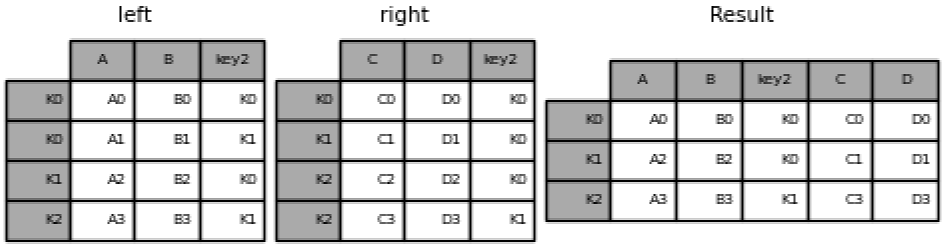
Merging on a combination of columns and index levels

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 [122]: left_index = pd.Index(["K0", "K0", "K1", "K2"], name="key1")

In [123]: left = pd.DataFrame(
   .....:     {
   .....:         "A": ["A0", "A1", "A2", "A3"],
   .....:         "B": ["B0", "B1", "B2", "B3"],
   .....:         "key2": ["K0", "K1", "K0", "K1"],
   .....:     },
```

Skip to main content

```
   .....:

In [124]: right_index = pd.Index(["K0", "K1", "K2", "K2"], name="key1")

In [125]: right = pd.DataFrame(
   .....:     {
   .....:         "C": ["C0", "C1", "C2", "C3"],
   .....:         "D": ["D0", "D1", "D2", "D3"],
   .....:         "key2": ["K0", "K0", "K0", "K1"],
   .....:     },
   .....:     index=right_index,
   .....: )
   .....:

In [126]: result = left.merge(right, on=["key1", "key2"])

In [127]: result
Out[127]:
       A   B key2   C   D
key1
K0    A0  B0   K0  C0  D0
K1    A2  B2   K0  C1  D1
K2    A3  B3   K1  C3  D3
```



left

	A	B	key2
K0	A0	B0	K0
K0	A1	B1	K1
K1	A2	B2	K0
K2	A3	B3	K1

right

	C	D	key2
K0	C0	D0	K0
K1	C1	D1	K0
K2	C2	D2	K0
K2	C3	D3	K1

Result

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

Note

When `DataFrame` are joined on a string that matches an index level in both arguments, the index level is preserved as an index level in the resulting `DataFrame`.

Note

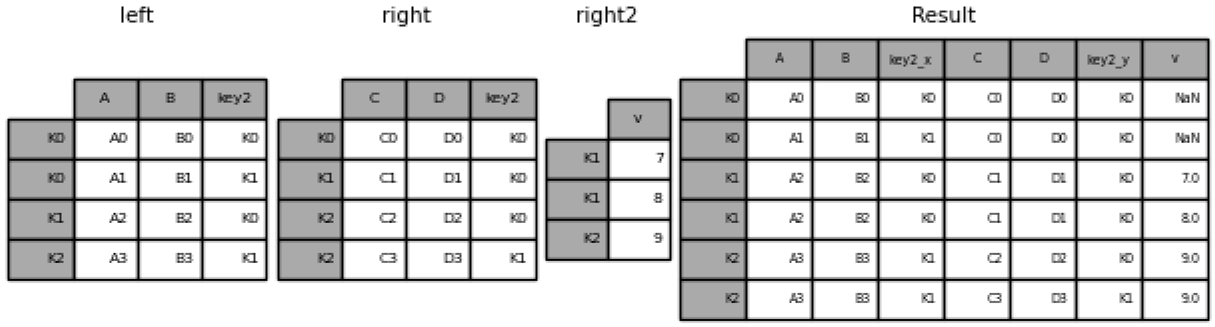
When `DataFrame` are joined using only some of the levels of a `MultiIndex`, the extra levels will be dropped from the resulting join. To preserve those levels, use `DataFrame.reset_index()` on those level names to move those levels to columns prior to the join.

Skip to main content

Joining multiple `DataFrame`

A list or tuple of `` :class:`DataFrame` `` can also be passed to `join()` to join them together on their indexes.

```
In [128]: right2 = pd.DataFrame({"v": [7, 8, 9]}, index=["K1", "K1", "K2"])

In [129]: result = left.join([right, right2])
```



left

	A	B	key2
K0	A0	B0	K0
K0	A1	B1	K1
K1	A2	B2	K0
K2	A3	B3	K1

right

	C	D	key2
K0	C0	D0	K0
K1	C1	D1	K0
K2	C2	D2	K0
K2	C3	D3	K1

right2

	v
K1	7
K1	8
K2	9

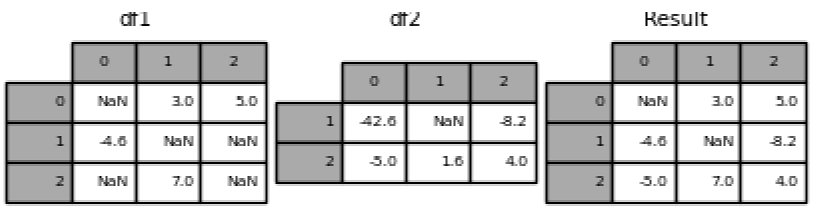
Result

	A	B	key2_x	C	D	key2_y	v
K0	A0	B0	K0	C0	D0	K0	NaN
K0	A1	B1	K1	C0	D0	K0	NaN
K1	A2	B2	K0	C1	D1	K0	7.0
K1	A2	B2	K0	C1	D1	K0	8.0
K2	A3	B3	K1	C2	D2	K0	9.0
K2	A3	B3	K1	C3	D3	K1	9.0

`DataFrame.combine_first()`

`DataFrame.combine_first()` update missing values from one `DataFrame` with the non-missing values in another `DataFrame` in the corresponding location.

```
In [130]: df1 = pd.DataFrame(
   .....:     [[np.nan, 3.0, 5.0], [-4.6, np.nan, np.nan], [np.nan, 7.0, np.nan]]
   .....: )
   .....:

In [131]: df2 = pd.DataFrame([[-42.6, np.nan, -8.2], [-5.0, 1.6, 4]], index=[1, 2]

In [132]: result = df1.combine_first(df2)

In [133]: result
Out[133]:
     0    1    2
0  NaN  3.0  5.0
1 -4.6  NaN -8.2
2 -5.0  7.0  4.0
```



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

`merge_ordered()`

`merge_ordered()` combines order data such as numeric or time series data with optional filling of missing data with `fill_method`.

```
In [134]: left = pd.DataFrame(
   .....:     {"k": ["K0", "K1", "K1", "K2"], "lv": [1, 2, 3, 4], "s": ["a", "b",
   .....: )
   .....:

In [135]: right = pd.DataFrame({"k": ["K1", "K2", "K4"], "rv": [1, 2, 3]})

In [136]: pd.merge_ordered(left, right, fill_method="ffill", left_by="s")
Out[136]:
     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
```

`merge_asof()`

`merge_asof()` is similar to an ordered left-join except that mactches are on the nearest key rather than equal keys. For each row in the `left` `DataFrame`, the last row in the `right` `DataFrame` are selected where the `on` key is less than the left's key. Both `DataFrame` must be

Optionally an `merge_asof()` can perform a group-wise merge by matching the `by` key in addition to the nearest match on the `on` key.

```
In [137]: 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 [138]: 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
   .....:         "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.0
   .....:         "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.0
   .....:     },
   .....:     columns=["time", "ticker", "bid", "ask"],
   .....: )
   .....:

In [139]: trades
Out[139]:
                     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
```

Skip to main content

```
Out[140]:
                     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

In [141]: pd.merge_asof(trades, quotes, on="time", by="ticker")
Out[141]:
                     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
```

`merge_asof()` within `2ms` between the quote time and the trade time.

```
In [142]: pd.merge_asof(trades, quotes, on="time", by="ticker", tolerance=pd.Timed
Out[142]:
                     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
```

`merge_asof()` within `10ms` between the quote time and the trade time and exclude exact matches on time. Note that though we exclude the exact matches (of the quotes), prior quotes **do** propagate to that point in time.

```
In [143]: pd.merge_asof(
   .....:     trades,
   .....:     quotes,
   .....:     on="time",
   .....:     by="ticker",
   .....:     tolerance=pd.Timedelta("10ms"),
   .....:     allow_exact_matches=False,
   .....: )
   .....:
Out[143]:
                     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
```

Skip to main content

```
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
```

compare()

The `Series.compare()` and `DataFrame.compare()` methods allow you to compare two `DataFrame` or `Series`, respectively, and summarize their differences.

```
In [144]: df = pd.DataFrame(
   .....:     {
   .....:         "col1": ["a", "a", "b", "b", "a"],
   .....:         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
   .....:         "col3": [1.0, 2.0, 3.0, 4.0, 5.0],
   .....:     },
   .....:     columns=["col1", "col2", "col3"],
   .....: )
   .....:

In [145]: df
Out[145]:
  col1  col2  col3
0    a   1.0   1.0
1    a   2.0   2.0
2    b   3.0   3.0
3    b   NaN   4.0
4    a   5.0   5.0

In [146]: df2 = df.copy()

In [147]: df2.loc[0, "col1"] = "c"

In [148]: df2.loc[2, "col3"] = 4.0

In [149]: df2
Out[149]:
  col1  col2  col3
0    c   1.0   1.0
```