



merge

Merging DataFrames with pandas

Here, you'll learn all about merging pandas DataFrames. You'll explore different techniques for merging, and learn about left joins, right joins, inner joins, and outer joins, as well as when to use which. You'll also learn about ordered merging, which is useful when you want to merge DataFrames whose columns have natural orderings, like date-time columns.

```
In [1]: import pandas as pd
```

[DataSets tutorial](#)

Merging company DataFrames

Suppose your company has operations in several different cities under several different managers. The DataFrames `revenue` and `managers` contain partial information related to the company. That is, the rows of the `city` columns don't quite match in `revenue` and `managers` (the Mendocino branch has no revenue yet since it just opened and the manager of Springfield branch recently left the company).

```
In [9]: revenue = pd.read_csv("revenue.csv")
managers = pd.read_csv("managers.csv")
print(revenue)
print(managers)
```

	city	revenue
0	Austin	100
1	Denver	83
2	Springfield	4

	city	manager
0	Austin	Charlers
1	Denver	Joel
2	Mendocino	Brett

The DataFrames have been printed in the IPython Shell. If you were to run the command `combined = pd.merge(revenue, managers, on='city')`, how many rows would `combined` have?

```
In [15]: combined = pd.merge(revenue, managers, on="city")
print(combined)
```

	city	revenue	manager
0	Austin	100	Charlers
1	Denver	83	Joel

Correct! Since the default strategy for `pd.merge()` is an *inner join*, `combined` will have 2 rows.

The `merge` command is the key learning objective of this tutorial. The merging operation at its simplest takes a left DataFrame (the first argument), a right DataFrame (the second argument), and then a merge column name, or a column to merge "on". In the output/result, rows from the left and right DataFrames are matched up where there are common values of the merge column specified by "on".

An *inner merge*, (or *inner join*) keeps only the common values in both the left and right DataFrames for the result.

Merging on a specific column

You expect your company to grow and, eventually, to operate in cities with the same name on different states. As such, you decide that every branch should have a numerical branch identifier. Thus, you add a `branch_id` column to both DataFrames. Moreover, new cities have been added to both the `revenue` and `managers` DataFrames as well:

```
In [18]: revenue = pd.read_csv("revenue_branch_id.csv")
managers = pd.read_csv("managers_branch_id.csv")
print(revenue)
print(managers)
```

	branch_id	city	revenue
0	10	Austin	100
1	20	Denver	83
2	30	Springfield	4
3	47	Mendocino	200

	branch_id	city	manager
0	10	Austin	Charlers
1	20	Denver	Joel
2	47	Mendocino	Brett
3	31	Springfield	Sally

Using `pd.merge()`, merge the DataFrames `revenue` and `managers` on the 'city' column of each

```
In [19]: merge_by_city = pd.merge(revenue, managers, on="city")
print(merge_by_city)
```

```

branch_id_x      city  revenue branch_id_y   manager
0            10    Austin     100          10  Charlers
1            20    Denver      83          20    Joel
2            30 Springfield      4          31  Sally
3           47 Mendocino     200          47  Brett

```

Merge the DataFrames `revenue` and `managers` on the `'branch_id'` column of each.

```
In [20]: merge_by_id = pd.merge(revenue, managers, on="branch_id")
print(merge_by_id)

branch_id      city_x  revenue   city_y   manager
0            10    Austin     100    Austin  Charlers
1            20    Denver      83    Denver    Joel
2           47 Mendocino     200  Mendocino  Brett
```

Well done! Notice that when you merge on `'city'`, the resulting DataFrame has a peculiar result: In row 2, the city Springfield has two different branch IDs. This is because there are actually two different cities named Springfield - one in the State of Illinois, and the other in Missouri. The `revenue` DataFrame has the one from Illinois, and the `managers` DataFrame has the one from Missouri. Consequently, when you merge on `'branch_id'`, both of these get dropped from the merged DataFrame.

Merging on columns with non-matching labels

We continue working with the `revenue` & `managers` DataFrames from before. This time, someone has changed the field name `'city'` to `'branch'` in the `managers` table. Now, when you attempt to merge DataFrames, an exception is thrown:

```
In [27]: revenue = pd.read_csv("revenue_branch_id_2.csv")
managers = pd.read_csv("managers_branch_id_2.csv")
print(revenue)
print(managers)

branch_id      city  revenue
0            10    Austin     100
1            20    Denver      83
2            30 Springfield      4
3           47 Mendocino     200
branch_id  branch  manager
0            10    Austin  Charlers
1            20    Denver    Joel
2           47 Mendocino  Brett
3           31 Springfield  Sally
```



```
pd.merge(revenue, managers, on='city') Traceback (most recent call last): ... <text deleted> ... pd.merge(revenue, managers, on='city') ... <text deleted> ... KeyError: 'city'
```

Given this, it will take a bit more work for you to join or merge on the city/branch name. You have to specify the `left_on` and `right_on` parameters in the call to `pd.merge()`.

```
In [26]: combined = pd.merge(revenue, managers, left_on="city", right_on="branch")
print(combined)

branch_id_x      city  revenue state_x  branch_id_y   branch \
0            10    Austin     100        TX          10    Austin
1            20    Denver      83        CO          20    Denver
2            30 Springfield      4        IL          31 Springfield
3           47 Mendocino     200        CA          47 Mendocino

manager state_y
0  Charlers    TX
1    Joel      CO
2   Sally      MO
3   Brett      CA
```

Great work! It is **important** to pay attention to **how columns are named** in different DataFrames.

Merging on multiple columns

Another strategy to disambiguate cities with identical names is to add information on the states in which the cities are located. To this end, you add a column called `state` to both DataFrames from the preceding exercises.

Our goal in this exercise is to use `pd.merge()` to merge DataFrames using multiple columns (using `'branch_id'`, `'city'`, and `'state'` in this case).

```
In [29]: revenue = pd.read_csv("revenue_branch_id.csv")
managers = pd.read_csv("managers_branch_id.csv")

# Add 'state' column to revenue
revenue["state"] = ["TX", "CO", "IL", "CA"]
# Add 'state' column to managers
managers["state"] = ["TX", "CO", "CA", "MO"]

print(revenue)
print(managers)

branch_id      city  revenue state
0            10    Austin     100    TX
1            20    Denver      83    CO
2            30 Springfield      4    IL
3           47 Mendocino     200    CA
branch_id      city  manager state
0            10    Austin  Charlers    TX
1            20    Denver    Joel      CO
2           47 Mendocino  Brett      CA
3           31 Springfield  Sally      MO
```



```
In [31]: # Merge revenue & managers on 'branch_id', 'city', & 'state'
combined = pd.merge(revenue, managers, on=["branch_id", "city", "state"])
print(combined)

branch_id      city  revenue state  manager
0            10    Austin     100    TX  Charlers
1            20    Denver      83    CO    Joel
2           47 Mendocino     200    CA    Brett
```

Excellent work!

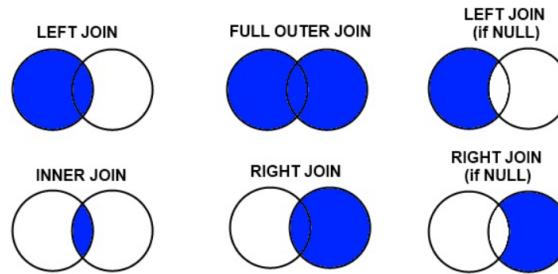
Other Merge Types

There are three different types of merges available in Pandas. These merge types are common across most database and data-orientated languages (SQL, R, SAS) and are typically referred to as "joins". If you don't know them, learn them now.

- **Inner Merge / Inner join** – The default Pandas behaviour, only keep rows where the merge "on" value exists in both the left and right dataframes.
- **Left Merge / Left outer join** – (aka left merge or left join) Keep every row in the left dataframe. Where there are missing values of the "on" variable in the right dataframe, add empty / NaN values in the result.
- **Right Merge / Right outer join** – (aka right merge or right join) Keep every row in the right dataframe. Where there are missing values of the "on" variable in the left column, add empty / NaN values in the result.
- **Outer Merge / Full outer join** – A full outer join returns all the rows from the left dataframe, all the rows from the right dataframe, and matches up rows where possible, with NaNs elsewhere.

The `merge` type to use is specified using the `how` parameter in the `merge` command, taking values `left`, `right`, `inner` (default), or `outer`.

Venn diagrams are commonly used to exemplify the different merge and join types.



Left & right merging on multiple columns

We now have, in addition to the `revenue` and `managers`, a DataFrame `sales` that summarizes units sold from specific branches (identified by `city` and `state` but not `branch_id`).

By merging `revenue` and `sales` with a `right` merge, we can identify the missing `revenue` values. Here, we don't need to specify `left_on` or `right_on` because the columns to merge on have matching labels.

```
In [32]: managers = pd.read_csv("managers_branch_id_2.csv")
managers["state"] = ["TX", "CO", "CA", "MO"]
sales = pd.read_csv("sales.csv")
print(sales)
```

	city	state	units
0	Mendocino	CA	1
1	Denver	CO	4
2	Austin	TX	2
3	Springfield	MO	5
4	Springfield	IL	1

```
In [33]: revenue_and_sales = pd.merge(revenue, sales, how="right", on=["city", "state"])
print(revenue_and_sales)
```

	branch_id	city	revenue	state	units
0	10.0	Austin	100.0	TX	2
1	20.0	Denver	83.0	CO	4
2	30.0	Springfield	4.0	IL	1
3	47.0	Mendocino	200.0	CA	1
4	Nan	Springfield	Nan	MO	5

By merging `sales` and `managers` with a `left` merge, we can identify the missing manager. Here, the columns to merge on have conflicting labels, so we must specify `left_on` and `right_on`. In both cases, we're looking to figure out how to connect the fields in rows containing `Springfield`.

```
In [34]: sales_and_managers = pd.merge(
    sales, managers, how="left", left_on=["city", "state"], right_on=["branch", "state"]
)
print(sales_and_managers)
```

	city	state	units	branch_id	branch	manager
0	Mendocino	CA	1	47.0	Mendocino	Brett
1	Denver	CO	4	20.0	Denver	Joel
2	Austin	TX	2	10.0	Austin	Charlers
3	Springfield	MO	5	31.0	Springfield	Sally
4	Springfield	IL	1	Nan	Nan	Nan

Well done! This is a good way to retain both entries of `Springfield`.

Merging DataFrames with outer join

The merged DataFrames contain enough information to construct a DataFrame with 5 rows with all known information correctly aligned and each branch listed only once. We will try to merge the merged DataFrames on all matching keys (which computes an inner join by default). We can compare the result to an outer join and also to an outer join with restricted subset of columns as keys.

Merge `sales_and_managers` with `revenue_and_sales`

```
In [35]: merge_default = pd.merge(sales_and_managers, revenue_and_sales)
print(merge_default)
```

```
   city state  units  branch_id    branch  manager  revenue
0  Mendocino    CA      1       47.0  Mendocino    Brett   200.0
1    Denver     CO      4       20.0    Denver    Joel    83.0
2    Austin     TX      2       10.0    Austin  Charlors   100.0
```

Merge sales_and_managers with revenue_and_sales using how='outer'

```
In [36]: merge_outer = pd.merge(sales_and_managers, revenue_and_sales, how="outer")
print(merge_outer)
```

```
   city state  units  branch_id    branch  manager  revenue
0  Mendocino    CA      1       47.0  Mendocino    Brett   200.0
1    Denver     CO      4       20.0    Denver    Joel    83.0
2    Austin     TX      2       10.0    Austin  Charlors   100.0
3  Springfield    MO      5      31.0  Springfield   Sally    NaN
4  Springfield    IL      1      NaN    Springfield   NaN    NaN
5  Springfield    IL      1      30.0    Springfield   NaN    4.0
6  Springfield    MO      5      NaN    Springfield   NaN    NaN
```

Merge sales_and_managers with revenue_and_sales only on ['city', 'state'] using an outerjoin.

```
In [37]: merge_outer_on = pd.merge(
    sales_and_managers, revenue_and_sales, how="outer", on=["city", "state"]
)
print(merge_outer_on)
```

```
   city state  units_x  branch_id_x    branch  manager  \
0  Mendocino    CA        1       47.0  Mendocino    Brett
1    Denver     CO        4       20.0    Denver    Joel
2    Austin     TX        2       10.0    Austin  Charlors
3  Springfield    MO        5      31.0  Springfield   Sally
4  Springfield    IL        1      NaN    Springfield   NaN

   branch_id_y  revenue  units_y
0       47.0   200.0        1
1       20.0    83.0        4
2       10.0   100.0        2
3       NaN     NaN        5
4       30.0     4.0        1
```

Fantastic work! Notice how the default merge drops the Springfield rows, while the default outer merge includes them twice.

Ordered merges

Using merge_ordered()

This exercise uses DataFrames austin and houston that contain weather data from the cities Austin and Houston respectively.

Weather conditions were recorded on separate days and we need to merge these two DataFrames together such that the dates are ordered. To do this, we'll use pd.merge_ordered(). Note the order of the rows before and after merging.

```
In [39]: austin = pd.read_csv("austin.csv")
houston = pd.read_csv("houston.csv")
print(austin)
print(houston)
```

```
   date ratings
0  2016-01-01  Cloudy
1  2016-02-08  Cloudy
2  2016-01-17  Sunny
   date ratings
0  2016-01-04  Rainy
1  2016-01-01  Cloudy
2  2016-03-01  Sunny
```

Perform an ordered merge on austin and houston using pd.merge_ordered()

```
In [40]: tx_weather = pd.merge_ordered(austin, houston)
print(tx_weather)
```

```
   date ratings
0  2016-01-01  Cloudy
1  2016-01-04  Rainy
2  2016-01-17  Sunny
3  2016-02-08  Cloudy
4  2016-03-01  Sunny
```

Perform another ordered merge on austin and houston. This time, specify the keyword arguments on='date' and suffixes=['_aus', '_hus'] so that the rows can be distinguished.

```
In [41]: tx_weather_suff = pd.merge_ordered(
    austin, houston, on="date", suffixes=['_aus', '_hus']
)
print(tx_weather_suff)
```

```
   date ratings_aus ratings_hus
0  2016-01-01  Cloudy    Cloudy
1  2016-01-04    NaN    Rainy
2  2016-01-17  Sunny    NaN
3  2016-02-08  Cloudy    NaN
4  2016-03-01    NaN    Sunny
```

Perform a third ordered merge on austin and houston. This time, in addition to the on and suffixes parameters, specify the keyword argument fill_method='ffill' to use forward-filling to replace NaN entries with the most recent non-null entry

```
In [42]: tx_weather_ffill = pd.merge_ordered(
    austin, houston, on="date", suffixes=['_aus', '_hus'], fill_method="ffill"
)
print(tx_weather_ffill)
```

```
   date ratings_aus ratings_hus
0  2016-01-01  Cloudy    Cloudy
```

```
1 2016-01-04     Cloudy     Rainy
2 2016-01-17     Sunny      Rainy
3 2016-02-08     Cloudy     Rainy
4 2016-03-01     Cloudy     Sunny
```

Well done! Notice how after using a fill method, there are no more `NaN` entries.

Conclusion

Hurray! You have come to the end of the tutorial. In this tutorial, you learned to merge DataFrames using the `merge()` function of pandas library. Towards the end, you also practiced the special function `merge_ordered()`.

This tutorial used the following sources to help write it:

- [DataCamp](#)
- [Official reference documentation from Pandas](#)
- [Tutorial from Shane Lynn](#)
- [Tutorial from Manish Pathak](#)
- [Cheat SheetPandas](#)

This website does not host notebooks, it only renders
notebooks available on other websites.

Delivered by **Fasty**, Rendered by OVHcloud
nbviewer GitHub [repository](#).

nbviewer version: 90c61cc
nbconvert version: 5.6.1
Rendered a few seconds ago