Part 2 – Lecture 5: Concatenation and merging

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
NHH Norwegian School of Economics

October 22, 2025

Contents

1 Concatenating and merging data

More often than not, data sets come from various sources and need to be concatenated (the process of appending observations or variables) or merged as part of data pre-processing. Pandas offers several routines to accomplish such tasks which we study in this unit:

1. `pd.concat()` allows us to combine multiple Series or DataFrames by appending observations (rows) or columns.
2. `pd.merge()` allows us to match observations from one Series or DataFrame with observations from another Series or DataFrame and combine these into a *merged* DataFrame.

You can also consult the official user guide and the pandas cheat sheet for more information.

1.1 Concatenation

Concatenation with `pd.concat()` is used to combine multiple data sets along the row or column axes. This function can be called with both `Series` and `DataFrame` arguments, as we illustrate below.

1.1.1 Concatenating Series

We begin with the simplest case of combining two `Series` to obtain a new `Series` which contains observations from both.

Example: Concatenating two Series along the row axis

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

```
# Create first series of 3 observations
a = pd.Series(['A0', 'A1', 'A2'])
a
```

```
[1]: 0    A0
     1    A1
     2    A2
     dtype: object
```

```
[2]: # Data for second series (5 observations)
     data_b = [f'B{i}' for i in range(5)]

     # Create second series
     b = pd.Series(data_b)
     b
```

```
[2]: 0    B0
     1    B1
     2    B2
     3    B3
     4    B4
     dtype: object
```

To concatenate a and b along the first dimension, we call `pd.concat()` as follows:

```
[3]: # Call concat() with the default value for axis, which is axis=0
     s = pd.concat((a, b))

     # Alternatively, make explicit that we are concatenating along the row axis
     # s = pd.concat((a, b), axis=0)
     s
```

```
[3]: 0    A0
     1    A1
     2    A2
     0    B0
     1    B1
     2    B2
     3    B3
     4    B4
     dtype: object
```

As you can see, `pd.concat()` also concatenates the index, which has the undesirable effect that the index values are no longer unique. As a rule, you never want to operate with non-unique indices in pandas as this can cause problems in some scenarios which are hard to diagnose. We can rectify this with the `reset_index()` method that we encountered in previous units:

```
[4]: # Reset index to get rid of duplicates
     s = s.reset_index(drop=True)
     s
```

```
[4]: 0    A0
     1    A1
     2    A2
     3    B0
     4    B1
     5    B2
     6    B3
     7    B4
     dtype: object
```

Alternatively, we can pass the `ignore_index=True` argument to `pd.concat()` and the return value will use the default index:

```
[5]: # Concatenate two series, use the default index for the return value
     pd.concat((a, b), ignore_index=True)
```

```
[5]: 0    A0
     1    A1
     2    A2
     3    B0
     4    B1
     5    B2
     6    B3
     7    B4
     dtype: object
```

Example: Concatenating along the column axis

It is also possible to concatenate `Series` along the column dimension by specifying `axis=1`. We would usually use this only for `Series` of equal length, as the result otherwise contains `NaN` values if the Series have different indices (e.g., because they differ in the number of observations).

```
[6]: s = pd.concat((a, b), axis=1)
     s
```

```
[6]:      0   1
     0   A0  B0
     1   A1  B1
     2   A2  B2
     3  NaN  B3
     4  NaN  B4
```

If the `Series` in question have no names, pandas assigns the values `0`, `1`, ... as column names. This can be avoided by explicitly passing the desired column names using the `keys` argument:

```
[7]: s = pd.concat((a, b), axis=1, keys=['Variable1', 'Variable2'])
     s
```

```
[7]:   Variable1 Variable2
     0        A0        B0
     1        A1        B1
     2        A2        B2
     3       NaN        B3
     4       NaN        B4
```

Your turn.

1. Create a new Series with observations ['C0', 'C1'].
2. Using the previously created Series `a` and `b`, concatenate all three objects along the row axis and create a new (unique) index.
3. Repeat the previous step, but now concatenate along the column axis. Assign the column names `'Column1'`, `'Column2'`, and `'Column3'`.

1.1.2 Concatenating DataFrames

Concatenating DataFrames works exactly the same way as for Series.

Concatenating along the column axis

Example: Concatenating two DataFrames along the column axis

In this example, we create two DataFrames with two and three columns, respectively.

```
[8]: import numpy as np

     # Create 2 x 2 array of string data
     data_a = np.array([f'A{i}' for i in range(4)]).reshape((2, 2))

     df_a = pd.DataFrame(data_a)
     df_a
```

```
[8]:     0   1
     0  A0  A1
     1  A2  A3
```

```
[9]: # Create 2 x 3 array of string data
     data_b = np.array([f'B{i}' for i in range(6)]).reshape((2, 3))

     df_b = pd.DataFrame(data_b)
     df_b
```

```
[9]:     0   1   2
     0  B0  B1  B2
     1  B3  B4  B5
```

To create a new DataFrame which contains the columns from both `df_a` and `df_b`, we use `pd.concat(..., axis=1)`:

```
[10]: # Concatenate along the column axis
      df = pd.concat((df_a, df_b), axis=1)
      df
```

```
[10]:     0   1   0   1   2
      0  A0  A1  B0  B1  B2
      1  A2  A3  B3  B4  B5
```

As before, the resulting `DataFrame` can have non-unique column names which is undesirable. There is no `reset_index()` method for columns, but we can easily create unique column names as follows:

```
[11]: # Reset column index to 0, 1, 2,...
      df.columns = np.arange(len(df.columns))
      df
```

```
[11]:     0   1   2   3   4
      0  A0  A1  B0  B1  B2
      1  A2  A3  B3  B4  B5
```

Alternatively, we can pass the `ignore_index=True` argument to `pd.concat()` as we did earlier to have pandas create default column names.

```
[12]: # Concatenate columns, return DataFrame with default column names
      pd.concat((df_a, df_b), axis=1, ignore_index=True)
```

```
[12]:     0   1   2   3   4
      0  A0  A1  B0  B1  B2
      1  A2  A3  B3  B4  B5
```

It is also possible to add an additional level of column names to the resulting `DataFrame` by specifying the keys argument:

```
[13]: # Concatenate along column axis, add additional column index level [A, B]
      df = pd.concat((df_a, df_b), axis=1, keys=['A', 'B'])
      df
```

[13]:
```
    A       B
    0   1   0   1   2
0  A0  A1  B0  B1  B2
1  A2  A3  B3  B4  B5
```

The new `DataFrame` then has a so-called hierarchical column index.

Example: Concatenating a DataFrame and a Series

One can also concatenate DataFrames and Series object along the column axis. In that case, the `Series` is automatically converted to a `DataFrame` using the default column name.

[14]:
```
s = pd.Series(['C0', 'C1'])
s
```

[14]:
```
0    C0
1    C1
dtype: object
```

[15]:
```
# Concatenate DataFrame and Series
pd.concat((df_a, s), axis=1, ignore_index=True)
```

[15]:
```
    0   1   2
0  A0  A1  C0
1  A2  A3  C1
```

Concatenating along the row axis

We usually concatenate DataFrames along the row axis if we have observations on the same variables scattered across multiple data sets. Appending DataFrames with different columns will usually create `NaN` values and hence is often not useful.

Example: Concatenating rows with identical columns

[16]:
```
#  Concatenate 2x2 DataFrame and 3x2 DataFrame (note the transpose!)
df = pd.concat((df_a, df_b.T), axis=0, ignore_index=True)
df
```

[16]:
```
    0   1
0  A0  A1
1  A2  A3
2  B0  B3
3  B1  B4
4  B2  B5
```

Example: Concatenating rows with different columns

The DataFrames `df_a` and `df_b` have a different number of columns, so the resulting `DataFrame` will contain `NaN` for all observations of column `2` that were originally in `df_a`:

[17]:
```
# Concatenate DataFrame rows with different numbers of columns
df = pd.concat((df_a, df_b), axis=0, ignore_index=True)
df
```

[17]:
```
    0   1    2
0  A0  A1  NaN
1  A2  A3  NaN
2  B0  B1   B2
3  B3  B4   B5
```

Your turn. Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `FRED_monthly_1950.csv` and `FRED_monthly_1960.csv` into two different DataFrames. The files contain monthly macroeconomic time series for the 1950s and 1960s, respectively.

   Hint: Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Concatenate these DataFrames along the row dimension to get a total of 240 observations.

3. Set the column `DATE` as index for the newly created DataFrame.

1.2 Merging and joining data sets

1.2.1 Types of merges

While concatenation simply appends blocks of rows or columns from multiple data sets, merging allows for more fine-grained control over how data should be combined. The most common scenarios in empirical work are:

1. *one-to-one*: The observations in data sets `A` and `B` have a unique identifier (*"key"*), and each observation in `A` is matched with at most one observation in `B`. For example, we could have data on individuals from multiple sources, and each of these data sets identifies individuals by their social security number. Each observation in one data set corresponds to exactly one observation in the other data set.

2. *many-to-one*: Data set `A` contains unique identifiers but these can correspond to multiple observations in data set `B`. For example, we could have data at the ZIP-code (neighborhood) level in data set `A` and data on individuals in data set `B`. ZIP-codes are a unique identifier in `A`, but many individuals can live in the same neighboorhood, so each observation in `A` can reasonably be matched with many different observations in `B`.

3. *many-to-many*: Identifying keys are not unique in either data set, and the resulting data set is a Cartesian product of all possible key combinations from both data sets. This situation should usually be *avoided* as it tends to have surprising results and can potentially consume large amounts of memory.

1.2.2 Implementation in pandas

Merging in pandas can be performed in two different ways:

1. `pd.merge()` is a function that takes as argument the *two* DataFrames to be merged:

   ```
   result = pd.merge(df_A, df_B)
   ```

2. `df.merge()` is a method of a specific `DataFrame` object, and takes as an argument the other `DataFrame` to be merged:

   ```
   result = df_A.merge(df_B)
   ```

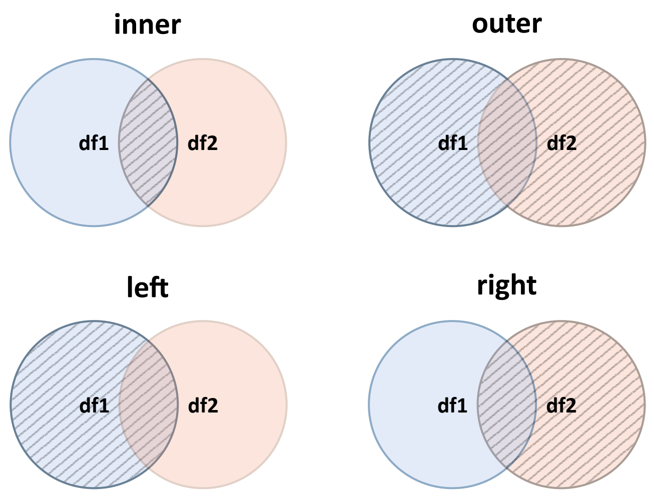
Both ways are equivalent and can be used interchangeably.

1.2.3 Controlling the resulting data set

Irrespective of whether we perform a *one-to-one* or a *many-to-one* merge, we frequently face the situation that some observations are present in one data set but not the other. We therefore need to control which subset of the data we want to retain in the final data set. This is accomplished using the `how` argument passed to `merge()`. There are several possible merge methods which were originally introduced in SQL, a data processing language for relational databases (see also the official user guide):

1. `how='inner'` performs a so-called *inner join*: the merged data contains only the *intersection* of keys that are present in *both* data sets.
2. `how='outer'` performs an *outer join*: the merged data contains the *union* of keys present in either of the data sets. Rows which are not present in both data sets will contain missing values.
3. `how='left'` performs a *left join*: all identifiers from the *left* data set are present in the merge result, but rows that are only present in the *right* data set are dropped.
4. `how='right'` performs a *right join*: all identifiers from the *right* data set are present in the merge result, but rows that are only present in the *left* data set are dropped.

The following figure illustrates these concepts graphically using Venn diagrams. Each circle represents the keys present in the left (`df1`) or right (`df2`) DataFrames. The merge method controls which subset of keys is retained in the merge result.



1.2.4 Merging with `pd.merge()`

One-to-one merges

We first create two data sets `A` and `B` used to demonstrate various merge methods. We use the column `key` as the identifier on which to perform merges.

[18]:
```
# Create first DataFrame with 2 rows
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})
df_a
```

[18]:
```
   key value_a
0    0      A0
1    1      A1
```

[19]:
```
# Create second DataFrame with 2 rows
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
df_b
```

[19]:
```
   key value_b
0    1      B1
1    2      B2
```

When merging two DataFrames, in most cases we need to specify the columns (or index levels) on which the merge should be performed. We do this using the argument `on` when calling `pd.merge()`.

[20]:
```
# Merge A and B on the identifier 'key' using an inner join
pd.merge(df_a, df_b, on='key', how='inner')
```

[20]:
```
   key value_a value_b
0    1      A1      B1
```

Note that in this case we could leave the `on` argument unspecified, as then `pd.merge()` by default merges on the intersection of columns present in both DataFrames (which in this case is just the column `key`). However, for clarity it is advisable to always specify `on` explicitly.

Moreover, `pd.merge()` performs an inner join by default, so we could have called the function as follows to get the same result:

[21]:
```
# Merge A and B on default key using default inner join
pd.merge(df_a, df_b)
```

[21]:
```
   key value_a value_b
0    1      A1      B1
```

Since we are performing an inner join, the merged data set contains only a single column corresponding to the identifier `1`, the only one present on both DataFrames.

If we want to retain all observations, we achieve this using an outer join:

[22]:
```
# Merge A and B using outer join (keep union of observations)
pd.merge(df_a, df_b, on='key', how='outer')
```

[22]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2     NaN      B2
```

Since the keys `0` and `2` are not present in both DataFrames, the corresponding columns contain missing values.

We can also only retain the keys present in the left (i.e., the first argument) or the right (i.e., the second argument) DataFrame:

[23]:
```
# Merge A and B on the identifier 'key', keep left identifiers
pd.merge(df_a, df_b, on='key', how='left')
```

[23]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

[24]:
```
# Merge A and B on the identifier 'key', keep right identifiers
pd.merge(df_a, df_b, on='key', how='right')
```

[24]:
```
   key value_a value_b
0    1      A1      B1
1    2     NaN      B2
```

Your turn. Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively.

   Hint: Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Merge the CPI and the GDP time series with `pd.merge()` without specifying the `how=` argument.
3. Merge the CPI and the GDP time series with `merge()` using a left join (`how='left'`).
4. Merge the CPI and the GDP time series with `merge()` using an inner join (`how='inner'`).

How do the results differ depending on the value of the `how` argument?

Many-to-one merges

To illustrate many-to-one merges, we create a DataFrame `A` which has non-unique values in the column `keys`. We want to merge this with another DataFrame `B` with unique values in the column `keys`. This operation is therefore a many-to-one merge.

[25]:
```
# Create first DataFrame with 4 rows, non-unique key
df_a = pd.DataFrame({'key': [0, 1, 0, 1], 'value_a': ['A0', 'A1', 'A2', 'A3']})
df_a
```

[25]:
```
   key value_a
0    0      A0
1    1      A1
2    0      A2
3    1      A3
```

[26]:
```
# Create second DataFrame with 2 rows, unique key
df_b = pd.DataFrame({'key': [0, 1], 'value_b': ['B0', 'B1']})
df_b
```

[26]:
```
   key value_b
0    0      B0
1    1      B1
```

[27]:
```
# Merge A and B on the identifier 'key', keep left identifiers
pd.merge(df_a, df_b, on='key')
```

[27]:
```
   key value_a value_b
0    0      A0      B0
1    1      A1      B1
2    0      A2      B0
3    1      A3      B1
```

As you can see, in the resulting DataFrame the values from `B` are matched with multiple rows of `A`.

1.2.5 Merging with `DataFrame.merge()`

As mentioned above, there is an alternative but equivalent way to merge DataFrames using the method `df.merge()`. In this context, the *left* `DataFrame` is the one on which `merge()` is being invoked, while the *right* `DataFrame` is the argument passed to `merge()`.

To illustrate, we recreate the DataFrames we used for the one-to-one merge examples above:

[28]:
```
# Create first DataFrame with 2 rows
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})

# Create second DataFrame with 2 rows
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
```

We now use the `merge()` method invoked on DataFrame `A` to merge it with DataFrame `B`:

[29]:
```
# Use DataFrame method to merge, keep only left identifiers
df_a.merge(df_b, on='key', how='left')
```

[29]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

[30]:
```
# Now df_a is the right DataFrame, set of final identifiers is the same as
# in the example above!
df_b.merge(df_a, on='key', how='right')
```

[30]:
```
   key value_b value_a
0    0     NaN      A0
1    1      B1      A1
```

Example: Merging with overlapping column names

Sometimes both DataFrames contain the same column names. If these columns are not used as keys in the merge operation, pandas automatically renames these columns in the resulting `DataFrame` to avoid naming clashes.

To illustrate, we rename the value columns to `'value'` in both DataFrames and then perform the merge:

[31]:
```
# Rename columns to common name 'value'
df_a = df_a.rename(columns={'value_a': 'value'})
df_b = df_b.rename(columns={'value_b': 'value'})
```

Note that once we have identical column names `['key', 'value']` in both DataFrames, we *must* specify the `on` argument to `merge()` as otherwise pandas by default merges on the intersection on column names in both DataFrames, i.e., in this case it merges on `['key', 'value']`:

[32]:
```
# Invoking merge() with default on argument has unintended consequences
df_a.merge(df_b)
```

[32]:
```
Empty DataFrame
Columns: [key, value]
Index: []
```

The merge result is empty because we are performing an *inner join* (the default), and there are no overlapping rows that have the same values for both `key` and `value` columns. We therefore need to explicitly specify `on='key'` to get the desired result:

[33]:
```
# Merge DataFrames with overlapping column 'value'
df_a.merge(df_b, on='key')
```

[33]:
```
   key value_x value_y
0    1      A1      B1
```

As you can see, pandas automatically appends the suffixes `'_x'` and `'_y'` to the column from the left and right DataFrames, respectively. we can change this default behavior by explicitly specifying the suffixes to be appended:

[34]:
```
df_a.merge(df_b, on='key', suffixes=('_left', '_right'))
```

[34]:
```
   key value_left value_right
0    1         A1          B1
```

Your turn. Repeat the previous exercise, but use the `DataFrame.merge()` method to perform the merge.

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively.

   Hint: Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Merge the DataFrame containing the CPI with the DataFrame containing GDP without specifying the `how=` argument.
3. Merge the DataFrame containing the CPI with the DataFrame containing GDP using a *left* join.
4. Merge the DataFrame containing the CPI with the DataFrame containing GDP using an *inner* join.

How do the results differ depending on the value of the `how` argument?

1.2.6 Merging with `join()`

The `DataFrame` method `join()` is a convenience wrapper around `pd.merge()` with the following subtle differences:

1. `join()` can be called *only* directly on the `DataFrame` object, i.e., `df.join()`, while for merge we have both the `pd.merge()` and the `df.merge()` variants.
2. `join()` always operates on the *index* of the other `DataFrame`, whereas `merge()` is more flexible and can operate on either the index or on columns.
3. `join()` by default performs a `left` join, whereas `merge()` performs an `inner` join.

As a rule of thumb, you should use `join()` if you want to join DataFrames which have a similar index.

Example: joining DataFrames

We first create two DataFrames to be joined. This time, we explicitly set an index for each of them which will be used to perform the `join()`.

[35]:
```python
# Create first DataFrame with 2 rows
df_a = pd.DataFrame(['A0', 'A1'], columns=['value_a'], index=[0, 1])
df_a
```

[35]:
```
  value_a
0      A0
1      A1
```

[36]:
```python
# Create second DataFrame with 2 rows
df_b = pd.DataFrame(['B1', 'B2'], columns=['value_b'], index=[1, 2])
df_b
```

[36]:
```
  value_b
1      B1
2      B2
```

[37]:
```python
# Perform left join (the default option)
df_a.join(df_b)
```

[37]:
```
  value_a value_b
0      A0     NaN
```

```
1      A1      B1
```

[38]:
```
# Join with explicit inner join
df_a.join(df_b, how='inner')
```

[38]:
```
  value_a value_b
1      A1      B1
```

[39]:
```
# Perform an outer join
df_a.join(df_b, how='outer')
```

[39]:
```
  value_a value_b
0      A0     NaN
1      A1      B1
2     NaN      B2
```

Your turn. Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively.

   Hint: Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Set the `DATE` column as the index for each of the two DataFrames.
3. Merge the CPI with the GDP time series with `join()`. Do this with both a left and an inner join.

2 Dealing with missing values

We already encountered missing values in earlier lectures. These are particularly likely to arise when merging or concatenating data if individual DataFrames lack some observations.

To illustrate, recall the example from above:

[40]:
```
# Create two DataFrames with partially overlapping keys
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
```

[41]:
```
# Perform outer merge, keep union of keys
pd.merge(df_a, df_b, on='key', how='outer')
```

[41]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2     NaN      B2
```

Since they keys in DataFrames `df_a` and `df_b` were only partially overlapping, the resulting DataFrame has missing values by construction. In what follows, we explore strategies on how to handle these missing data.

2.1 Dropping missing values

One strategy is to drop missing values outright, even though we might lose information that could be useful to perform data analysis if only some but not all columns contain missing values, as is the case above.

Missing values can be dropped by either

1. Using `dropna()`, potentially restricted to a subset of columns.
2. Selecting a subset of observations to keep with a boolean operation such as `notna()` or `isna()`.
3. Avoiding the missing values in the first place, e.g., by using `merge(..., how='inner')`.

Example: Dropping missing values

Consider the merged `DataFrame` from above. We can drop rows with missing values with `dropna()`, which by default drops all rows with *any* missing values. Alternatively, we can specify only a subset of columns to be checked for missing values.

[42]:
```
# Merge with outer join, thus creating missing values
df = pd.merge(df_a, df_b, on='key', how='outer')
```

[43]:
```
# Drop any row which contains at least one missing value
df.dropna()
```

[43]:
```
   key value_a value_b
1    1      A1      B1
```

[44]:
```
# Drop rows which contain missing values in column 'value_a', ignore missing
# values in 'value_b'
df.dropna(subset='value_a')
```

[44]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

Your turn. Instead of using `dropna()`, drop the missing observations using either `notna()` or `isna()`.

1. Drop all rows with any missing observations.
2. Drop all rows with missing observations in column value_a.

Example: Avoiding missing values in the first place

Of course the missing values in the example above arose only because we specified `how='outer'`. Merging with `how='inner'` drops keys which are not present in both DataFrames right away, avoiding the issue of missing values (unless these are already present in the original DataFrames):

[45]:
```
# Merge using inner join, drop keys not present in both DataFrames
pd.merge(df_a, df_b, on='key', how='inner')
```

[45]:
```
   key value_a value_b
0    1      A1      B1
```

2.2 Filling missing values

Instead of dropping data, we can impute missing values in various ways:

1. `fillna()` can be used to replace missing data with user-specified values.
2. `ffill()` and `bfill()` can be used to fill missing values forward or backward from adjacent non-missing observations.
3. `interpolate()` supports various interpolation methods such as linear interpolation based on non-missing values.

Example: Replacing missing values with `fillna()`

Consider the merged `DataFrame` we have created above:

```
[46]: df = pd.merge(df_a, df_b, on='key', how='outer')
      df
```

```
[46]:    key value_a value_b
      0    0      A0     NaN
      1    1      A1      B1
      2    2     NaN      B2
```

We can use `fillna()` to replace missing values with some constant.

```
[47]: # Replace ALL missing values with 'Some value'
      df.fillna('Some value')
```

```
[47]:    key     value_a     value_b
      0    0          A0  Some value
      1    1          A1          B1
      2    2  Some value          B2
```

This might not be what you want as the provided non-missing value is imposed on *all* columns. It is therefore possible to specify a different value for each column using a dictionary as an argument.

```
[48]: # Use different replacement values for columns 'value_a' and 'value_b'
      df.fillna({'value_a': 'Missing A', 'value_b': 'Missing B'})
```

```
[48]:    key    value_a    value_b
      0    0         A0  Missing B
      1    1         A1         B1
      2    2  Missing A         B2
```

Example: forward- or backward-filling missing values

Another common imputation method is to use the previous (*"forward"*) or next (*"backward"*) non-missing value as replacement for missing data.

Continuing with the `DataFrame` from the previous example, we can apply these methods as follows:

```
[49]: # Forward-fill missing values from previous observation
      df.ffill()
```

```
[49]:    key value_a value_b
      0    0      A0     NaN
      1    1      A1      B1
      2    2      A1      B2
```

This inserts the value `'A1'` in the 3rd row of column `value_a`, but does not do anything about the missing value in column `value_b` since there is no preceding non-missing value.

Conversely, `bfill()` does the opposite and backfills the missing value in column `value_b`:

```
[50]: df.bfill()
```

```
[50]:    key value_a value_b
      0    0      A0      B1
      1    1      A1      B1
      2    2     NaN      B2
```

Example: linear interpolation

Consider the following `Series` with numerical data (interpolation only makes sense for numerical data, not strings):

[51]:
```
s = pd.Series([1.0, 2.0, 3.0, np.nan, 5.0])
s
```

[51]:
```
0    1.0
1    2.0
2    3.0
3    NaN
4    5.0
dtype: float64
```

We can interpolate the missing data using `interpolate()`, for example by using linear interpolation (check the documentation for many other interpolation methods).

[52]:
```
# Interpolate missing values using linear interpolation
s.interpolate(method='linear')
```

[52]:
```
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
dtype: float64
```

Your turn. Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively.

   Hint: Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Merge the CPI with the GDP time series with `merge()` using a left join. This creates missing values in the GDP column.

3. Impute the missing GDP values using `interpolate()` and replace the missing values in column GDP.

3 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [go to section] links to the relevant section in this notebook.

Concatenating (appending) data

- `pd.concat()` — append Series or DataFrames along the row or column dimension [go to section]
- `reset_index()` — reset (potentially non-unique) index after concatenation [go to section]

Merging data

- `pd.merge()` — merge two Series or DataFrames along the column dimension based some keys [go to section]
- `DataFrame.merge()` — merge a given (left) DataFrame with another (right) Series or DataFrame based on some keys [go to section]

- `DataFrame.join()` — merge a given (left) DataFrame with another (right) Series or DataFrame based on the index [go to section]

Dealing with missing values

- `notna()` — check whether (one or more) columns have non-missing values [go to section]
- `isna()` — check whether (one or more) columns have missing values [go to section]
- `dropna()` — drop rows with missing observations [go to section]
- `fillna()` — replace missing values with a given value [go to section]
- `ffill()` — fill missing values by propagating the last valid observation [go to section]
- `bfill()` — fill missing values by using the next valid observation to fill the gap [go to section]
- `interpolate()` — interpolate missing values from adjacent non-missing ones [go to section]