Lecture 10: Aggregation and merging

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

March 18, 2025

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/FIE463-V25

Contents

1 Grouping and aggregation with pandas

1.1 Aggregation and reduction

Similar to NumPy, pandas supports data aggregation and reduction functions such as computing sums or averages. By *"aggregation"* or *"reduction"* we mean that the result of a computation has a lower dimension than the original data: for example, the mean reduces a series of observations (1 dimension) into a scalar value (0 dimensions).

Unlike NumPy, these operations can be applied to subsets of the data which have been grouped according to some criterion.

Such operations are often referred to as *split-apply-combine* (see the official user guide) as they involve these three steps:

1. *Split* data into groups based on some criteria;
2. *Apply* some function to each group separately; and
3. *Combine* the results into a single `DataFrame` or `Series`.

See also the pandas cheat sheet for an illustration of such operations.

We first set the path pointing to the folder which contains the data files used in this lecture. You may need to adapt it to your own environment.

[1]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/FIE463-V25/main/data'
```

1.1.1 Aggregations of whole Series or DataFrames

The simplest way to perform data reduction is to invoke the desired function on the entire `DataFrame`.

[2]:
```
import pandas as pd

# Path to Titanic passenger data CSV file
file = f'{DATA_PATH}/titanic.csv'

# Read in Titanic passenger data, set PassenderId column as index
df = pd.read_csv(file, index_col='PassengerId')

# Compute mean of all numerical columns
df.mean(numeric_only=True)
```

[2]:
```
Survived     0.383838
Pclass       2.308642
Age         29.699118
Fare        32.204208
dtype: float64
```

Methods such as `mean()` are by default applied column-wise to each column. The `numeric_only=True` argument is used to discard all non-numeric columns (depending on the version of pandas, `mean()` will issue a warning if there are non-numerical columns in the `DataFrame`).

One big advantage over NumPy is that missing values (represented by `np.nan`) are automatically ignored:

[3]:
```
# mean() automatically drops missing observations
mean_pandas = df['Age'].mean()

# Compare this to the NumPy variant:
import numpy as np

# Returns NaN since some ages are missing (coded as NaN)
mean_numpy = np.mean(df['Age'].to_numpy())

print(f'Mean using Pandas: {mean_pandas}')
print(f'Mean using NumPy:  {mean_numpy}')
```

```
Mean using Pandas: 29.69911764705882
Mean using NumPy:  nan
```

As we have seen previously, NumPy implements an additional set of aggregation functions which drop NaNs, for example `np.nanmean()`.

1.1.2 Aggregations of subsets of data (grouping)

Applying aggregation functions to the entire `DataFrame` is similar to what we can do with NumPy. The added flexibility of pandas becomes obvious once we want to apply these functions to subsets of data, i.e., groups which we can define based on values or index labels.

For the remainder of this section, we use a 10% sample of the Survey of Consumer Finances (SCF) for the years 1989-2022. The SCF is a triennial cross-sectional survey of U.S. households which includes information on balance sheets, pensions, income, and demographic characteristics.

We load the SCF data as follows:

[4]:
```
# Path to SCF data file
file = f'{DATA_PATH}/SCF/SCF_10pct.csv'

# Read in SCF data, set id column as index
df = pd.read_csv(file, index_col='id')
```

This dataset contains the following mostly self-explanatory columns:

[5]:
```
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5817 entries, 1 to 5817
Data columns (total 20 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        5817 non-null   int64
 1   year       5817 non-null   int64
 2   female     5817 non-null   int64
 3   married    5817 non-null   int64
 4   educ       5817 non-null   int64
 5   empl       5817 non-null   int64
 6   income     5817 non-null   float64
 7   rent       5817 non-null   float64
 8   equity     5817 non-null   float64
 9   finassets  5817 non-null   float64
 10  liqassets  5817 non-null   float64
 11  houses     5817 non-null   float64
 12  business   5817 non-null   float64
 13  vehicles   5817 non-null   float64
 14  assets     5817 non-null   float64
 15  mortages   5817 non-null   float64
 16  debt       5817 non-null   float64
 17  networth   5817 non-null   float64
 18  owner      5817 non-null   int64
 19  weight     5817 non-null   float64
dtypes: float64(13), int64(7)
memory usage: 954.4 KB
```

As a first example, we group the SCF sample by employment status (column `empl`) using `groupby()`:

[6]:
```
# Group observations by employment status (1 = working for someone else,
# 2 = self-employed, 3 = retired/disabled, 4 = not in labor force)
groups = df.groupby(['empl'])
```

Here `groups` is a special pandas objects which can subsequently be used to process group-specific data. To compute the group-wise averages, we simply run

[7]:
```
# Compute group-specific means for ALL columns
groups.mean()
```

[7]:
```
            age         year    female   married      educ        income  \
empl
1     43.795960  2007.520202  0.239394  0.637374  2.917508  4.260747e+05
2     54.712018  2006.367347  0.075586  0.817838  3.359788  2.410422e+06
3     68.233807  2006.969194  0.347551  0.515798  2.614534  5.113619e+05
4     39.782946  2006.116279  0.445736  0.414729  2.600775  2.663478e+05

            rent        equity     finassets      liqassets        houses  \
empl
1     397.726214  1.438976e+06  2.040738e+06  119333.585657  4.130635e+05
2     261.911157  5.603988e+06  8.996746e+06  820293.861980  1.632493e+06
3     246.440889  3.353023e+06  5.441704e+06  360725.330016  5.758679e+05
4     526.246509  4.780461e+05  9.799629e+05  126127.593798  2.415235e+05

          business       vehicles        assets       mortages           debt  \
empl
1     1.060121e+06  106391.366968  3.900751e+06  123503.647677  223526.750640
2     2.029157e+07  210922.228408  3.510614e+07  297781.022902  759135.329403
3     1.823703e+06   67000.125811  9.187975e+06   47285.060585   88804.125355
4     5.517233e+05   21814.942094  2.059549e+06   32220.589147   55874.311240

          networth     owner        weight
empl
1     3.677224e+06  0.617845  26727.697271
2     3.434700e+07  0.876039  11787.453134
3     9.099171e+06  0.733017  26611.096044
4     2.003675e+06  0.360465  27140.389862
```

Groups support column indexing: if we want to only compute the average age by employment status, we can do this as follows:

[8]:
```
groups['age'].mean()
```

[8]:
```
empl
1    43.795960
2    54.712018
3    68.233807
4    39.782946
Name: age, dtype: float64
```

Built-in aggregations

There are numerous routines to aggregate grouped data, for example:

- `mean()`: averages within each group
- `sum()`: sum values within each group
- `std()`, `var()`: within-group standard deviation and variance
- `median()`: compute median within each group

- `quantile()`: compute quantiles within each group
- `size()`: number of observations in each group
- `count()`: number of non-missing observations in each group
- `first()`, `last()`: first and last elements in each group
- `min()`, `max()`: minimum and maximum elements within a group

See the official documentation for a complete list.

Example: Number of elements within each group

```
[9]: groups.size()       # return number of elements in each group
```

```
[9]: empl
     1    2970
     2    1323
     3    1266
     4     258
     dtype: int64
```

Note that `size()` and `count()` are two different functions. The former returns the group sizes (and the return value is a `Series`), whereas `count()` returns the number of non-missing observations for *each* column.

Example: Return first observation of each group

```
[10]: groups[['age', 'female', 'married']].first()      # return first observation in each group
```

```
[10]:       age  female  married
      empl
      1      40       0        1
      2      34       0        1
      3      64       0        1
      4      22       1        0
```

Your turn. Use the SCF data set to perform the following aggregations:

1. Compute the average net worth (`networth`) by marital status (`married`).
2. Compute the median value of the primary residence (`houses`) by education (`educ`).
3. Compute the home ownership rate (`owner`) by marital status (`married`) and the sex of the household head (`female`).

Writing custom aggregations

We can create custom aggregation routines by calling `agg()` (short-hand for `aggregate()`) on the grouped object. Such functions operate on one column at a time, so it is only possible to use observations from that column for computations.

For example, we can alternatively call the built-in aggregation functions we just covered via `agg()`:

```
[11]: # Calculate group means in needlessly complicated way
      groups['age'].agg('mean')

      # More direct approach:
      # df.groupby('empl')['age'].mean()
```

```
[11]: empl
      1    43.795960
      2    54.712018
      3    68.233807
      4    39.782946
      Name: age, dtype: float64
```

On the other hand, we *have to* use `agg()` if there is no built-in function to perform the desired aggregation. To illustrate, imagine that we want to count the number of household heads aged 60+ by employment status. There is no built-in function to achieve this, so we need to use `agg()` combined with a custom function to perform the desired aggregation:

[12]:
```
import numpy as np

# Count number of individuals age 60+ in each group
groups['age'].agg(lambda x: np.sum(x >= 60))
```

[12]:
```
empl
1     374
2     459
3    1017
4      17
Name: age, dtype: int64
```

Note that we called `agg()` only on the column `age`, otherwise the function would be applied to every column separately, which is not what we want.

Applying multiple functions at once

It is possible to apply multiple functions in a single call by passing a list of functions. These can be passed as strings or as callables (functions).

Example: Applying multiple functions to a ***single*** *column*

To compute the mean and median age by employment status, we proceed as follows:

[13]:
```
groups['age'].agg(['mean', 'median'])
```

[13]:
```
           mean  median
empl
1     43.795960    43.0
2     54.712018    55.0
3     68.233807    69.0
4     39.782946    39.0
```

Note that we could have also specified these function by passing references to the corresponding NumPy functions instead:

[14]:
```
df.groupby('empl')['age'].agg([np.mean, np.median])
```

```
/tmp/ipykernel_2361350/2161376742.py:1: FutureWarning: The provided callable
<function mean at 0x7f78cb717060> is currently using SeriesGroupBy.mean. In a
future version of pandas, the provided callable will be used directly. To keep
current behavior pass the string "mean" instead.
  df.groupby('empl')['age'].agg([np.mean, np.median])
/tmp/ipykernel_2361350/2161376742.py:1: FutureWarning: The provided callable
<function median at 0x7f78ca95de40> is currently using SeriesGroupBy.median. In
a future version of pandas, the provided callable will be used directly. To keep
current behavior pass the string "median" instead.
  df.groupby('empl')['age'].agg([np.mean, np.median])
```

[14]:
```
           mean  median
empl
1     43.795960    43.0
2     54.712018    55.0
3     68.233807    69.0
4     39.782946    39.0
```

The following more advanced syntax allows us to create new column names using existing columns and some operation:

```
groups.agg(
    new_column_name1=('column_name1', 'operation1'),
    new_column_name2=('column_name2', 'operation2'),
    ...
)
```

This is called "named aggregation" as the keywords determine the output column *names*.

Example: Applying multiple functions to ***multiple*** *columns*

The following code computes the median age, the smallest net worth, and the share of female household heads by employment status in a single operation:

[15]:
```
# Compute multiple statistics for multiple columns
groups.agg(
    median_age=('age', 'median'),
    min_networth=('networth', 'min'),
    share_female=('female', 'mean')
)
```

[15]:
```
      median_age  min_networth  share_female
empl
1           43.0    -4069933.6      0.239394
2           55.0    -2712279.9      0.075586
3           69.0     -388230.5      0.347551
4           39.0    -1034938.5      0.445736
```

Finally, the most flexible aggregation method is `apply()` which calls a given function, passing the *entire* group-specific subset of data (including all columns) as an argument. You need to use `apply()` if data from more than one column is required to compute a statistic of interest.

Your turn. Use the SCF data set to perform the following aggregations:

1. Compute the minimum, maximum and average age (`age`) by marital status and sex (`married` and `female`) in a single `agg()` operation.
2. Compute the number of observations, the home ownership rate (`owner`), and median net worth (`networth`) by education level (`educ`) in a single `agg()` operation.

1.2 Transformations

In the previous section, we combined grouping and reduction, i.e., data at the group level was reduced to a single statistic such as the mean. Alternatively, we can combine grouping with the `transform()` function which assigns the result of a computation to each observation within a group and consequently leaves the number of observations unchanged.

For example, for *each* observation we could compute the home ownership rate by eduction as follows:

[16]:
```
df['frac_owners'] = df.groupby('educ')['owner'].transform('mean')

# Print relevant columns
df[['educ', 'owner', 'frac_owners']].head(10)
```

[16]:
```
    educ  owner  frac_owners
id
1      2      1     0.610607
2      2      1     0.610607
3      2      0     0.610607
4      2      1     0.610607
5      1      0     0.524887
```

```
6       1       1      0.524887
7       3       1      0.611952
8       1       0      0.524887
9       1       0      0.524887
10      3       0      0.611952
```

As you can see, instead of collapsing the `DataFrame` to only 4 observations (one for each education level), the number of observations remains the same, and the home ownership rate is constant within each education level.

When would we want to use `transform()` instead of aggregation? Such use cases arise whenever we want to perform computations that include the individual value as well as an aggregate statistic.

Example: Deviation from median net worth

Assume that we want to compute how much each household's net worth differs from the median net worth in their respective education group. We could compute this using `transform()` as follows:

[17]:
```
# Compute difference of HH's net worth from median net worth in same education group
df['nw_diff'] = df['networth'] - df.groupby('educ')['networth'].transform('median')

# Print relevant columns
df[['educ', 'networth', 'nw_diff']].head(10)
```

[17]:
```
    educ    networth     nw_diff
id
1      2    234478.0    136351.9
2      2     61513.0    -36613.1
3      2      -622.5    -98748.6
4      2    587924.1    489798.0
5      1       691.7    -27449.4
6      1     62481.3     34340.2
7      3  11124446.8  10993179.8
8      1     -1475.6    -29616.7
9      1         0.0    -28141.1
10     3       322.8   -130944.2
```

Your turn. Use the SCF data set to answer the following questions:

1. Compute how much a household pays more in rent (`rent`) than the average household with the same marital (`married`) and employment status (`empl`). Restrict your analysis to households who do not own their residence (`owner = 0`).

1.3 Resampling and aggregation

We introduced support for time series data in pandas in the previous lecture. This basically comes down to specifying an index which is a date or time stamp and supports operations such as computing leads, lags, and differences over time.

Another useful feature of the time series support in pandas is *resampling* which is used to group observations by time period and apply some aggregation function. This can be accomplished using the `resample()` method which in its simplest form takes a string argument that describes how observations should be grouped (`'YE'` for aggregation to years, `'QE'` for quarters, `'ME'` for months, `'W'` for weeks, etc.).

To illustrate, we load the daily data on the value of the NASDAQ at close:

[18]:
```
# Path to NASDAQ data file
file = f'{DATA_PATH}/stockmarket/NASDAQ.csv'
```

```
# Read in NASDAQ data, set Date column as index
df = pd.read_csv(file, index_col='Date', parse_dates=True)

# Keep observations after 2024
df = df.loc['2024':]

# Print first few rows
df.head()
```

[18]:
```
             NASDAQ
Date
2024-01-02  14765.9
2024-01-03  14592.2
2024-01-04  14510.3
2024-01-05  14524.1
2024-01-08  14843.8
```

For example, if we want to aggregate this daily data to monthly frequency, we would use `resample('ME')`. This returns an object which is very similar to the one returned by `groupby()` we studied previously, and we can call various aggregation methods such as `mean()`:

[19]:
```
# Resample to monthly frequency, aggregate to mean of daily observations
# within each month
df.resample('ME').mean()
```

[19]:
```
                  NASDAQ
Date
2024-01-31  15081.390476
2024-02-29  15808.935000
2024-03-31  16216.295000
2024-04-30  15950.868182
2024-05-31  16536.322727
2024-06-30  17495.900000
2024-07-31  17963.281818
2024-08-31  17268.263636
2024-09-30  17599.235000
2024-10-31  18316.413043
2024-11-30  18961.345000
2024-12-31  19755.730000
```

Similarly, we can use `resample('W')` to resample to weekly frequency. Below, we combine this with the aggregator `last()` to return the last observation of each week (weeks by default start on Sundays):

[20]:
```
# Return last observation of each week, print first 10 rows
df.resample('W').last().head(10)
```

[20]:
```
             NASDAQ
Date
2024-01-07  14524.1
2024-01-14  14972.8
2024-01-21  15311.0
2024-01-28  15455.4
2024-02-04  15629.0
2024-02-11  15990.7
2024-02-18  15775.7
2024-02-25  15996.8
2024-03-03  16274.9
2024-03-10  16085.1
```

Your turn. Use the daily NASDAQ data for 2024 and compute the percentage change from the first to the last trading day within each month.

2 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 section:

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.

2.1 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.

2.1.1 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

[21]:
```
import pandas as pd

# Create first series of 3 observations
a = pd.Series(['A1', 'A2', 'A3'])
a
```

[21]:
```
0    A1
1    A2
2    A3
dtype: object
```

[22]:
```
# 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
```

[22]:
```
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:

[23]:
```
# 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
```

[23]:
```
0    A1
1    A2
2    A3
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. We can rectify this with the `reset_index()` method hat we encountered in previous units:

[24]:
```
# Reset index to get rid of duplicates
s = s.reset_index(drop=True)
s
```

[24]:
```
0    A1
1    A2
2    A3
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).

[25]:
```
s = pd.concat((a, b), axis=1)
s
```

[25]:
```
     0   1
0   A1  B0
1   A2  B1
2   A3  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:

[26]:
```
s = pd.concat((a, b), axis=1, keys=['Variable1', 'Variable2'])
s
```

[26]:
```
  Variable1 Variable2
0        A1        B0
1        A2        B1
2        A3        B2
3       NaN        B3
4       NaN        B4
```

Your turn.

1. Create a new Series with observations `['C1', 'C2']`.
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'`.

2.1.2 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.

[27]:
```
import numpy as np

# Create 2 x 2 array of string data
data_a = np.array(('A1', 'A2', 'A3', 'A4')).reshape((2, 2))

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

[27]:
```
    0   1
0  A1  A2
1  A3  A4
```

[28]:
```
# 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
```

[28]:
```
    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)`:

[29]:
```
# Concatenate along the column axis
df = pd.concat((df_a, df_b), axis=1)
df
```

[29]:
```
    0   1   0   1   2
0  A1  A2  B0  B1  B2
1  A3  A4  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, e.g., as follows:

[30]:
```
# Reset column index to 0, 1, 2,...
df.columns = np.arange(len(df.columns))
df
```

[30]:
```
    0   1   2   3   4
0  A1  A2  B0  B1  B2
1  A3  A4  B3  B4  B5
```

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

[31]:
```
# Concatenate along column axis, add additional column index level [A, B]
df = pd.concat((df_a, df_b), axis=1, keys=['A', 'B'])
df
```

[31]:
```
    A       B
    0   1   0   1   2
0  A1  A2  B0  B1  B2
1  A3  A4  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.

[32]:
```
s = pd.Series(['C1', 'C2'])
s
```

[32]:
```
0    C1
1    C2
dtype: object
```

[33]:
```
# Concatenate DataFrame and Series
pd.concat((df_a, s), axis=1)
```

[33]:
```
    0   1   0
0  A1  A2  C1
1  A3  A4  C2
```

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

[34]:
```
#  Concatenate 2x2 DataFrame and 3x2 DataFrame (note the transpose!)
df = pd.concat((df_a, df_b.T), axis=0)
df
```

[34]:
```
    0   1
0  A1  A2
1  A3  A4
0  B0  B3
1  B1  B4
2  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`:

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

```
[35]:     0   1    2
      0  A1  A2  NaN
      1  A3  A4  NaN
      0  B0  B1   B2
      1  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.

2.2 Merging and joining data sets

2.2.1 Types of merges

While concatenation simply appends a block 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.

2.2.2 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, e.g.,

   ```
   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, e.g.,

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

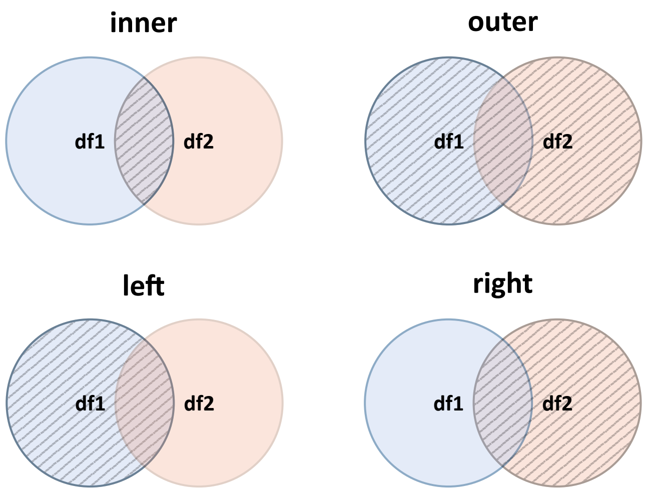
Both ways are equivalent and can be used interchangeably.

2.2.3 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.



2.2.4 Merging with `merge()`

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.

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

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

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

```
df_b
```

[37]:
```
   key value_b
0    1      B1
1    2      B2
```

Using `pd.merge()`

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()` or `df.merge()`

Example: one-to-one merges

[38]:
```
# Merge A and B on the identifier 'key' using an inner join
pd.merge(df_a, df_b, on='key', how='inner')
```

[38]:
```
   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:

[39]:
```
# Merge A and B on default key using default inner join
pd.merge(df_a, df_b)
```

[39]:
```
   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:

[40]:
```
# Merge A and B using outer join (keep union of observations)
pd.merge(df_a, df_b, on='key', how='outer')
```

[40]:
```
   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:

[41]:
```
# Merge A and B on the identifier 'key', keep left identifiers
pd.merge(df_a, df_b, on='key', how='left')
```

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

[42]:
```
# Merge A and B on the identifier 'key', keep right identifiers
pd.merge(df_a, df_b, on='key', how='right')
```

[42]:
```
   key value_a value_b
0    1      A1      B1
1    2     NaN      B2
```

Using `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()`:

[43]:
```
# Use DataFrame method to merge, keep only left identifiers
df_a.merge(df_b, on='key', how='left')
```

[43]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

[44]:
```
# 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')
```

[44]:
```
   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:

[45]:
```
# 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']`:

[46]:
```
# Invoking merge() with default on argument has unintended consequences
df_a.merge(df_b)
```

[46]:
```
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:

[47]:
```
# Merge DataFrames with overlapping column 'value'
df_a.merge(df_b, on='key')
```

[47]:
```
   key value_x value_y
0    1      A1      B1
```

[48]:
```
df_a.merge(df_b, on='key', suffixes=('_left', '_right'))
```

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

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 (`how='left'`). How many observations does the resulting DataFrame have?

3. Merge the CPI with the GDP time series with `merge()` using an inner join (`how='inner'`). How many observations does the resulting DataFrame have, and why is this different from the previous case?

2.2.5 Joining 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()`.

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

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

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

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

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

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

[52]:
```
# Join with explicit inner join
df_a.join(df_b, how='inner')
```

[52]:
```
  value_a value_b
1      A1      B1
```

[53]:
```
# Perform an outer join
df_a.join(df_b, how='outer')
```

[53]:
```
  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.

3 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:

[54]:
```
# 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']})
```

[55]:
```
# Perform outer merge, keep union of keys
pd.merge(df_a, df_b, on='key', how='outer')
```

[55]:
```
   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.

3.1 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 are missing, as is the case above.

Missing values can be dropped by either

1. Using `dropna()` or selecting a subset of observations with a boolean operation such as `notna()`.
2. 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.

[56]:
```
# Merge with outer join, thus creating missing values
df = pd.merge(df_a, df_b, on='key', how='outer')
```

[57]:
```
# Drop any row which contains at least one missing value
df.dropna()
```

[57]:
```
   key value_a value_b
1    1      A1      B1
```

[58]:
```
# Drop rows which contain missing values in column 'value_a', ignore missing
# values in 'value_b'
df.dropna(subset='value_a')
```

[58]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

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):

[59]:
```
# Merge using inner join, drop keys not present in both DataFrames
pd.merge(df_a, df_b, on='key', how='inner')
```

[59]:
```
   key value_a value_b
0    1      A1      B1
```

3.2 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:

[60]:
```
df = pd.merge(df_a, df_b, on='key', how='outer')
df
```

[60]:
```
   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.

[61]:
```
# Replace ALL missing values with 'Some value'
df.fillna('Some value')
```

[61]:
```
   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.

[62]:
```
# Use different replacement values for columns 'value_a' and 'value_b'
df.fillna({'value_a': 'Missing A', 'value_b': 'Missing B'})
```

[62]:
```
   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:

[63]:
```
# Forward-fill missing values from previous observation
df.ffill()
```

[63]:
```
   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`:

[64]:
```
df.bfill()
```

[64]:
```
   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):

[65]:
```
s = pd.Series([1.0, 2.0, 3.0, np.nan, 5.0])
s
```

[65]:
```
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).

[66]:
```python
# Interpolate missing values using linear interpolation
s.interpolate(method='linear')
```

[66]:
```
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.