Part 2 – Lecture 4: Grouping and aggregation

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
NHH Norwegian School of Economics

October 15, 2025

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/TECH2-H25/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`. We illustrate this using the Titanic dataset which we have encountered in the previous lectures.

We first load the data and tabulate the columns present in the `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(f'{DATA_PATH}/titanic.csv', index_col='PassengerId')

# Tabulate columns and number of observations
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 891 entries, 1 to 891
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Survived  891 non-null    int64  
 1   Pclass    891 non-null    int64  
 2   Name      891 non-null    object 
 3   Sex       891 non-null    object 
 4   Age       714 non-null    float64
 5   Ticket    891 non-null    object 
 6   Fare      891 non-null    float64
 7   Cabin     204 non-null    object 
 8   Embarked  889 non-null    object 
dtypes: float64(2), int64(2), object(5)
memory usage: 69.6+ KB
```

We can now apply the `mean()` method to all numerical columns to compute the average for each column:

[3]:
```
# Compute mean of all numerical columns
df.mean(numeric_only=True)
```

[3]:
```
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, as the following code demonstrates:

[4]:
```
import numpy as np

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

# np.mean() 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:.1f}')
```

```
print(f'Mean using NumPy:  {mean_numpy:.1f}')
```

```
Mean using Pandas: 29.7
Mean using NumPy:  nan
```

For this reason, 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 define based on values or index labels.

For example, we can easily group passengers by class using `groupby()`:

[5]:
```
import pandas as pd

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

# Group observations by accommodation class (first, second, third)
groups = df.groupby(['Pclass'])
```

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

[6]:
```
groups.mean(numeric_only=True)
```

[6]:

	Survived	Age	Fare
Pclass			
1	0.629630	38.233441	84.154687
2	0.472826	29.877630	20.662183
3	0.242363	25.140620	13.675550

Groups support column indexing: if we want to only compute the total fare paid by passengers in each class, we can do this as follows:

[7]:
```
groups['Fare'].sum()
```

[7]:
```
Pclass
1    18177.4125
2     3801.8417
3     6714.6951
Name: Fare, dtype: float64
```

Built-in aggregations

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

- `mean()`: compute average 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

[8]:
```
groups.size()        # return number of elements in each group
```

[8]:
```
Pclass
1    216
2    184
3    491
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

[9]:
```
groups[['Survived', 'Age', 'Sex', 'Fare']].first()      # return first observation in each
  ↪group
```

[9]:
```
        Survived   Age     Sex     Fare
Pclass
1              1  38.0  female  71.2833
2              1  14.0  female  30.0708
3              0  22.0    male   7.2500
```

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

1. Compute the average survival rate by sex (stored in the Sex column).
2. Count the number of passengers aged 50+. Compute the average survival rate by sex for this group.
3. Count the number of passengers below the age of 20 by class and sex. Compute the average survival rate for this group by class and sex.

Writing custom aggregations

We can create custom aggregation routines by calling `agg()` (short-hand for `aggregate()`) on the grouped object. These 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()`:

[10]:
```
# Calculate group means in needlessly complicated way
groups["Age"].agg("mean")

# More direct approach:
# groups["age"].mean()
```

[10]:
```
Pclass
1    38.233441
2    29.877630
```

```
3    25.140620
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 passengers aged 40+ in each class. 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:

[11]:
```python
import numpy as np

# Apply a custom aggregation using a lambda expression
groups['Age'].agg(lambda x: np.sum(x >= 40))
```

[11]:
```
Pclass
1    81
2    37
3    45
Name: Age, dtype: int64
```

In this example, we use a `lambda` expression to define the custom aggregation function in place. This is a shorthand notation which is equivalent to defining a custom function first:

[12]:
```python
# Define a custom aggregation function
def fcn(x):
    return np.sum(x >= 40)

# Apply the custom aggregation function to the 'Age' column
groups['Age'].agg(fcn)
```

[12]:
```
Pclass
1    81
2    37
3    45
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 passenger age by class, we proceed as follows:

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

[13]:

	mean	median
Pclass		
1	38.233441	37.0
2	29.877630	29.0
3	25.140620	24.0

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

[14]:
```python
groups['Age'].agg([np.mean, np.median])
```

```
/tmp/ipykernel_30596/3990418967.py:1: FutureWarning: The provided callable
<function mean at 0x7f2f3c1ce980> 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.
  groups['Age'].agg([np.mean, np.median])
/tmp/ipykernel_30596/3990418967.py:1: FutureWarning: The provided callable
<function median at 0x7f2f23f09940> 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.
  groups['Age'].agg([np.mean, np.median])
```

[14]:

Pclass	mean	median
1	38.233441	37.0
2	29.877630	29.0
3	25.140620	24.0

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 average age and the highest fare in a single aggregation:

[15]:
```
groups.agg(
    average_age=('Age', 'mean'),
    max_fare=('Fare', 'max')
)
```

[15]:

Pclass	average_age	max_fare
1	38.233441	512.3292
2	29.877630	73.5000
3	25.140620	69.5500

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 Titanic data set to perform the following aggregations:

- Compute the minimum, maximum and average age by embarkation port (stored in the column `Embarked`) in a single `agg()` operation. Note that there are several ways to solve this problem.
- Compute the number of passengers, the average age and the fraction of women by embarkation port in a single `agg()` operation.

  Hint: To compute the fraction of women, you can either use a `lambda` expressions, or you first create a numerical indicator variable for females (as we did in the workshop).

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 average fare by class as follows:

[16]:
```
df['Avg_Fare'] = df.groupby('Pclass')[['Fare']].transform('mean')

# Print results for each institution
df[['Pclass', 'Fare', 'Avg_Fare']].head(10)
```

[16]:
```
             Pclass     Fare   Avg_Fare
PassengerId
1                 3   7.2500  13.675550
2                 1  71.2833  84.154687
3                 3   7.9250  13.675550
4                 1  53.1000  84.154687
5                 3   8.0500  13.675550
6                 3   8.4583  13.675550
7                 1  51.8625  84.154687
8                 3  21.0750  13.675550
9                 3  11.1333  13.675550
10                2  30.0708  20.662183
```

As you can see, instead of collapsing the `DataFrame` to only 3 observations (one for each class), the number of observations remains the same, and the average fare is constant within each class.

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 average fare

Assume that we want to compute how much each passenger's fare differed from the average fare in their respective class. We could compute this using `transform()` as follows:

[17]:
```
import numpy as np

# Compute difference of passenger's fare and avg. fare paid within class
df['Fare_Diff'] = df.groupby('Pclass')['Fare'].transform(lambda x: x - np.mean(x))

# Print relevant columns
df[['Pclass', 'Fare', 'Fare_Diff']].head(10)
```

[17]:
```
             Pclass     Fare  Fare_Diff
PassengerId
1                 3   7.2500  -6.425550
2                 1  71.2833 -12.871387
3                 3   7.9250  -5.750550
4                 1  53.1000 -31.054687
5                 3   8.0500  -5.625550
6                 3   8.4583  -5.217250
7                 1  51.8625 -32.292187
8                 3  21.0750   7.399450
9                 3  11.1333  -2.542250
10                2  30.0708   9.408617
```

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

1. Compute the excess fare paid by each passenger relative to the minimum fare by embarkation port and class, i.e., compute $Fare - \min(Fare)$ by port and class.

1.3 Resampling and aggregation

We discussed how to handle time series data in pandas in the previous lecture. This basically comes down to specifying an index which is a date or time stamp. Such and index allows us to easily perform 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 a data set that contains daily observations on the value of the NASDAQ stock market index 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.