Part 2 — Lecture 3

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
Norwegian School of Economics (NHH)

October 16, 2024

See GitHub repository for notebooks and data:

`https://github.com/richardfoltyn/TECH2-H24`

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-H24/main/data'
```

1.1.1 Working with entire DataFrames

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

[2]:
```
import pandas as pd

# Read in Titanic passenger data, set PassengerId column as index
df = pd.read_csv(f'{DATA_PATH}/titanic.csv', 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 Working on 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 example, we can easily group passengers by class using `groupby()`:

[4]:
```
import pandas as pd

# Import Titanic data set, set PassengerId column as index
df = pd.read_csv(f'{DATA_PATH}/titanic.csv', 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

[5]:
```
groups.mean(numeric_only=True)
```

[5]:
```
        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:

[6]:
```
groups['Fare'].sum()
```

[6]:
```
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()`: averages within each group
- `sum()`: sum values within each group
- `std()`, `var()`: within-group standard deviation and variance
- `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

[7]:
```
groups.size()       # return number of elements in each group
```

[7]:
```
Pclass
1    216
2    184
3    491
dtype: int64
```

Example: Return first observation of each group

[8]:
```
groups[['Survived', 'Age', 'Sex', 'Fare']].first()      # return first observation in each
 ↪group
```

[8]:
```
        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()`:

[9]:
```
# Calculate group means in needlessly complicated way
groups["Age"].agg("mean")

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

[9]:
```
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:

[10]:
```
import numpy as np

groups['Age'].agg(lambda x: np.sum(x >= 40))
```

[10]:
```
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:

[11]:
```
groups['Age'].agg(['mean', 'median'])
```

[11]:
```
            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:

[12]:
```
groups['Age'].agg([np.mean, np.median])
```

[12]:
```
            mean  median
Pclass
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:

[13]:
```
groups.agg(
    average_age=('Age', 'mean'),
    max_fare=('Fare', 'max')
)
```

[13]:
```
        average_age  max_fare
Pclass
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:

1. 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.
2. Compute the number of passengers, the average age and the fraction of women by embarkation port in a single `agg()` operation. This one is more challenging and probably requires use of lambda expressions.

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:

[14]:
```
df['Avg_Fare'] = df.groupby('Pclass')[['Fare']].transform('mean')

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

[14]:
```
             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:

[15]:
```
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)
```

[15]:
```
             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.

2 Working with time series data

In economics and finance, we frequently work with time series data, i.e., observations that are associated with a particular point in time (time stamp) or a time period. pandas offers comprehensive support for such data, in particular if the time stamp or time period is used as the index of a `Series` or `DataFrame`. This section presents a few of the most important concepts, see the official documentation for a comprehensive guide.

To illustrate, let's construct a set of daily data for the first three months of 2024, i.e., the period 2024-01-01 to 2024-03-31 using the `date_range()` function (we use the data format `YYYY-MM-DD` in this section, but pandas also supports other date formats).

[16]:
```
import pandas as pd
import numpy as  np

# Create sequence of dates from 2024-01-01 to 2024-03-31
# at daily frequency
index = pd.date_range(start='2024-01-01', end='2024-03-31', freq='D')

# Use date range as index for Series with some artificial data
data = pd.Series(np.arange(len(index)), index=index)

# Print first 5 observations
data.head(5)
```

[16]:
```
2024-01-01    0
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
Freq: D, dtype: int64
```

2.1 Indexing with date/time indices

pandas implements several convenient ways to select observations associated with a particular date or a set of dates. For example, if we want to select one specific date, we can pass it as a string to `.loc[]`:

[17]:
```
# Select single observation by date
data.loc['2024-01-01']
```

[17]:
```
0
```

It is also possible to select a time period by passing a start and end point (where the end point is included, as usual with label-based indexing in pandas):

[18]:
```
# Select first 5 days
data.loc['2024-01-01':'2024-01-05']
```

[18]:
```
2024-01-01    0
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
Freq: D, dtype: int64
```

A particularly useful way to index time periods is a to pass a partial index. For example, if we want to select all observations from January 2024, we could use the range `'2024-01-01':'2024-01-31'`, but it is much easier to specify the partial index `'2024-01'` instead which includes all observations from January.

[19]:
```
# Select all observations from January 2024
data.loc['2024-01']
```

[19]:
```
2024-01-01     0
2024-01-02     1
2024-01-03     2
2024-01-04     3
2024-01-05     4
2024-01-06     5
2024-01-07     6
2024-01-08     7
2024-01-09     8
2024-01-10     9
2024-01-11    10
2024-01-12    11
2024-01-13    12
2024-01-14    13
2024-01-15    14
2024-01-16    15
2024-01-17    16
2024-01-18    17
2024-01-19    18
2024-01-20    19
2024-01-21    20
2024-01-22    21
2024-01-23    22
2024-01-24    23
2024-01-25    24
2024-01-26    25
2024-01-27    26
2024-01-28    27
2024-01-29    28
2024-01-30    29
2024-01-31    30
Freq: D, dtype: int64
```

2.2 Lags, differences, and other useful transformations

When working with time series data, we often need to create lags or leads of a variable (e.g., if we want to include lagged values in a regression model). In pandas, this is done using `shift()` which shifts the index by the desired number of periods (default: 1). For example, invoking `shift(1)` creates lagged observations of each column in the `DataFrame`:

[20]:
```
# Lag observations by 1 period
data.shift(1).head(5)
```

[20]:
```
2024-01-01    NaN
2024-01-02    0.0
2024-01-03    1.0
2024-01-04    2.0
2024-01-05    3.0
Freq: D, dtype: float64
```

Note that the first observation is now missing since there is no preceding observation which could have provided the lagged value.

Another useful method is `diff()` which computes the difference between adjacent observations (the period over which the difference is taken can be passed as a parameter).

[21]:
```
# Compute 1-period difference
data.diff().head(5)
```

[21]:
```
2024-01-01    NaN
2024-01-02    1.0
2024-01-03    1.0
2024-01-04    1.0
2024-01-05    1.0
Freq: D, dtype: float64
```

Note that `diff()` is identical to manually computing the difference with the lagged value like this:

```
data - data.shift()
```

Additionally, we can use `pct_change()` which computes the percentage change (the relative difference) over a given number of periods (default: 1).

[22]:
```
# Compute percentage change vs. previous period
data.pct_change().head(5)
```

[22]:
```
2024-01-01         NaN
2024-01-02         inf
2024-01-03    1.000000
2024-01-04    0.500000
2024-01-05    0.333333
Freq: D, dtype: float64
```

Again, this is just a convenience method that is a short-cut for manually computing the percentage change:

```
(data - data.shift()) / data.shift()
```

2.3 Resampling and aggregation

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.).

For example, if we want to aggregate our 3 months of artificial 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()`:

[23]:
```
# Resample to monthly frequency, aggregate to mean of daily observations
# within each month
data.resample('ME').mean()
```

[23]:
```
2024-01-31    15.0
2024-02-29    45.0
2024-03-31    75.0
Freq: ME, dtype: float64
```

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):

[24]:
```
# Return last observation of each week
data.resample('W').last()
```

[24]:
```
2024-01-07     6
2024-01-14    13
2024-01-21    20
2024-01-28    27
2024-02-04    34
2024-02-11    41
```

```
2024-02-18    48
2024-02-25    55
2024-03-03    62
2024-03-10    69
2024-03-17    76
2024-03-24    83
2024-03-31    90
Freq: W-SUN, dtype: int64
```