Big Data and Automated Content Analysis (12EC)
Week 3: »Data Wrangling and Exploratory Analysis«
Wednesday

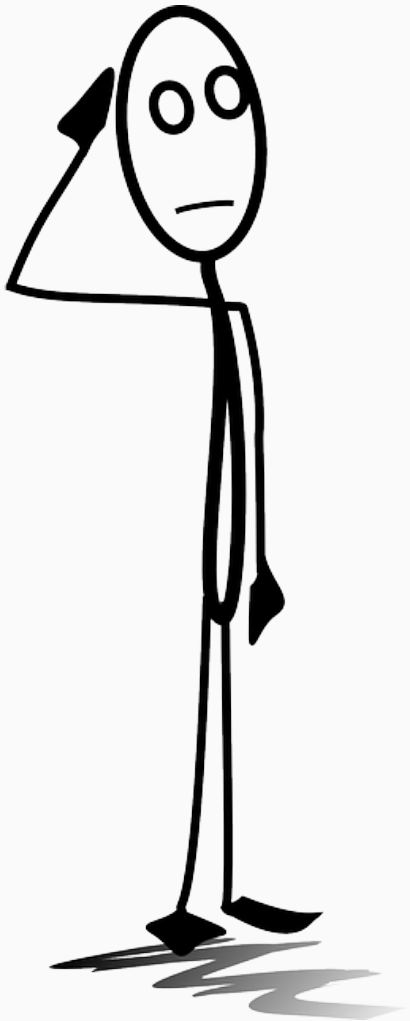
Damian Trilling
d.c.trilling@uva.nl, @damian0604

February 22, 2022

UvA RM Communication Science

Today

- Working with pandas
  - Basics of pandas
  - Selecting and recoding data
  - Joining and Merging
  - Grouping and aggregating
  - From wide to long
- Basics of plotting in Python
- Next steps



Everything clear from last week?

Main points from last week

I assume that by now, everybody knows:

- how to read and write different types of files
- how to work with both nested and tabular data
- how to deal with data of an unknown structure (such as responses from an unfamiliar JSON API)
- . . . and of course all the basics: data types, for loops, conditions, etc.

This week, we will look into tabular, relatively traditional data. But remember what we said last week about different data structures – we will work with them a lot as well in the upcoming weeks!

Working with pandas

Working with pandas

Basics of pandas

The idea behind pandas

- Central concepts: dataframe (2D) and series (1D)
- *Vectorized* operations: apply a function to a whole column or row
- Resembles working with R/tidyverse (but: object orientation instead of pipes)

Most central ressource: McKinney (2012), a book about pandas by the creator of pandas. Updated 3rd edition (2022) freely available online: https://wesmckinney.com/book/

The idea behind pandas

- Central concepts: dataframe (2D) and series (1D)
- *Vectorized* operations: apply a function to a whole column or row
- Resembles working with R/tidyverse (but: object orientation instead of pipes)

Most central ressource: McKinney (2012), a book about pandas by the creator of pandas. Updated 3rd edition (2022) freely available online: https://wesmckinney.com/book/

Pandas in the Python ecosystem

- *The* package that unites everything that fits in a dataframe or in a one-dimensional (time) series
- Operates, on the background, with "basic" packages such as `numpy` (stats) or `matplotlib` (plotting)

```
1  import pandas as pd
2  df = pd.DataFrame({"A":[2,3,3,4,3,5], "B":[10,8,9,7,5,5]})
3  df.corr()
```

```
1  import numpy as np
2  A = [2,3,3,4,3,5]
3  B = [10,8,9,7,5,5]
4  np.corrcoef(A,B)
```

$\Rightarrow$ The dataframe method `.corr()` does the same as we could achieve by feeding lists into the numpy function `corrcoef()`.

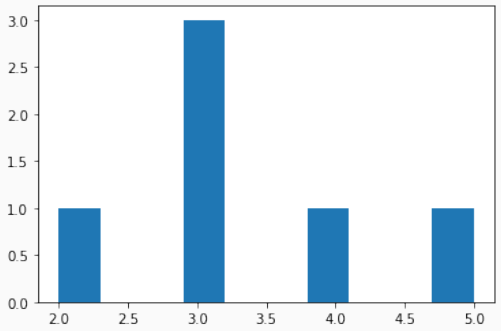
Pandas in the Python ecosystem

Similarily, you can plot directly from pandas...

```
1  df["A"].hist()
```

...or achieve the same by using matplotlib and a list:

```
1  import matplotlib.pyplot as plt
2  plt.hist(A)
```



Pandas simply makes these things easier, and “to pandas or not to pandas” is partly a matter of personal preferences. But its really strong point is data wrangling. You really don’t want to do *that* without pandas.

Central concepts

index *Both* columns and rows have labels – these are called an index. You can refer to columns and/or rows either by their number or by their label.

axis `axis=0`: row-wise, `axis=1`: column-wise

dtype columns have a data type (e.g., `int64`). The generic "fallback" option (e.g., mixed types in one column) is simply called `object`. Not to be confused with...

object-orientation Dataframes (and columns) are objects and hence have methods. These methods (typically) return new objects, which have new methods, etc.

Central concepts

index *Both* columns and rows have labels – these are called an index. You can refer to columns and/or rows either by their number or by their label.

axis `axis=0`: row-wise, `axis=1`: column-wise

dtype columns have a data type (e.g., `int64`). The generic "fallback" option (e.g., mixed types in one column) is simply called `object`. Not to be confused with...

object-orientation Dataframes (and columns) are objects and hence have methods. These methods (typically) return new objects, which have new methods, etc.

Central concepts

index *Both* columns and rows have labels – these are called an index. You can refer to columns and/or rows either by their number or by their label.

axis `axis=0`: row-wise, `axis=1`: column-wise

dtype columns have a data type (e.g., `int64`). The generic "fallback" option (e.g., mixed types in one column) is simply called `object`. Not to be confused with...

object-orientation Dataframes (and columns) are objects and hence have methods. These methods (typically) return new objects, which have new methods, etc.

Central concepts

index *Both* columns and rows have labels – these are called an index. You can refer to columns and/or rows either by their number or by their label.

axis `axis=0`: row-wise, `axis=1`: column-wise

dtype columns have a data type (e.g., `int64`). The generic "fallback" option (e.g., mixed types in one column) is simply called `object`. Not to be confused with...

object-orientation Dataframes (and columns) are objects and hence have methods. These methods (typically) return new objects, which have new methods, etc.

Central concepts

You can use build-in methods...

```
df['radio'].mean()    # returns 3.33
```

...or apply a function:

```
import numpy as np

df['age'].apply(np.log)
# if we want to assign the result to a new column, we could do:
df['log(age)'] = df['age'].apply(np.log)
```

Getting to know your data

```
import pandas as pd

df = pd.read_csv("https://cssbook.net/d/media.csv")
df
```

	gender	age	education	radio	newspaper	tv	internet
0	1	71	4.0	5	6	5	0
1	1	40	2.0	6	0	0	0
2	1	41	2.0	4	3	7	3
3	0	65	5.0	0	0	5	0
4	0	39	2.0	0	1	7	7
...	...	...	...	...	...	...	...
2076	0	49	5.0	3	6	6	0
2077	0	51	4.0	7	7	5	5
2078	1	31	6.0	3	5	5	6

Getting to know your data

```
# descriptive statistics of all numerical columns
df.describe()
```

	gender	age	education	radio	newspaper	tv	internet
count	2081.000000	2081.000000	2065.000000	2081.000000	2081.000000	2081.000000	2081.000000
mean	0.481499	46.073522	4.272639	3.333974	3.111004	4.167227	2.684286
std	0.499778	18.267401	1.661451	2.699082	2.853082	2.517039	2.786262
min	0.000000	13.000000	1.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	31.000000	3.000000	0.000000	0.000000	2.000000	0.000000
50%	0.000000	46.000000	4.000000	4.000000	3.000000	5.000000	2.000000
75%	1.000000	61.000000	6.000000	6.000000	6.000000	7.000000	5.000000
max	1.000000	95.000000	7.000000	7.000000	7.000000	7.000000	7.000000

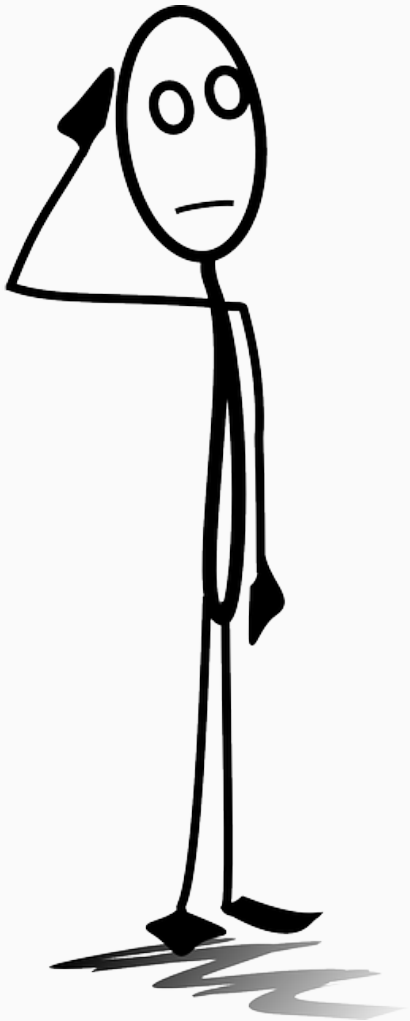
Getting to know your data

```
# frequency counts (e.g., for nominal variables)
df['gender'].value_counts()
```

```
0    1079
1    1002
Name: gender, dtype: int64
```

```
# we can also have a list of columns
df[['radio','newspaper','tv','internet']].describe()
```

	radio	newspaper	tv	internet
count	2081.000000	2081.000000	2081.000000	2081.000000
mean	3.333974	3.111004	4.167227	2.684286
std	2.699082	2.853082	2.517039	2.786262
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	2.000000	0.000000



Any questions so far?

Sidenote: standard statistics

Not the topic of today, but for those interested: Statistical modelling in pandas is a bit "R-like" if you use `statsmodels`:

```
1 import statsmodels.formula.api as smf
2
3 mymodel = smf.ols(formula='internet ~ age + gender +  education',
  ↪  data=df).fit()
4 mymodel.summary()
```

Working with pandas

Selecting and recoding data

Recoding values

There are multiple ways of recoding values: Via the `.replace()` method or via a function.

```
valuemap = {1:"geen", 7: "WO", 4:"HBO"}
df['education_recoded'] = df['education'].replace(valuemap)  # what's not in the map is kept as-is
```

```
def recode_edu(x):
    if x==0:
        return "no"
    elif 1<=x<=4:
        return "low"
    elif x>4:
        return "high"

df['education_recoded'] = df['education'].apply(recode_edu)
```

Subsetting and slicing

```
1  df[['gender', 'age']]           # refers to two columns
2  df.loc[:, ["gender", "age"]]    # the *values* in them
3  df.iloc[:, [3, 4]]              # use column number instead of label
4
5  df.loc[5:10, ["gender", "age"]]  # the values in rows 5--10 in them
6
7  df[df.loc[:, 'gender']==0]      # select all rows with females
```

So what's the difference between the approach in line 1 and the `.loc`/`.iloc` approach?

Subsetting and slicing

```
1  df[['gender', 'age']]           # refers to two columns
2  df.loc[:, ["gender", "age"]]    # the *values* in them
3  df.iloc[:, [3, 4]]              # use column number instead of label
4
5  df.loc[5:10, ["gender", "age"]]  # the values in rows 5--10 in them
6
7  df[df.loc[:, 'gender']==0]      # select all rows with females
```

So what's the difference between the approach in line 1 and the `.loc`/`.iloc` approach?

Subsetting and slicing

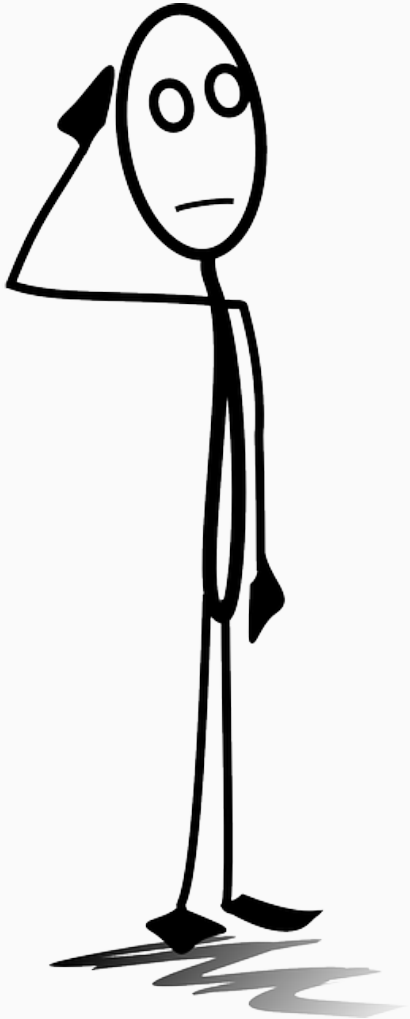
The difference is between creating a copy and referring to *exactly* the same object itself:

```
# we are a mad dictator and forbid women to listen to the radio:
# this often works, but is dangerous
df[df['gender']==0]['radio']=0
```

```
<ipython-input-89-f44d1c735422>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/par
g-a-view-versus-a-copy
  df[df['gender']==0]['radio']=0
```

```
# this is the better way of doing it
df[df.loc[:,'gender']==0]['radio']=0
```



Do you understand why there is two times `df` *in this expression:* `df[df.loc[:, 'gender']==0]`*? Hint: What does the inner part* (`df.loc[:,'gender']==0`) *return? (You may guess!)*

Subsetting and slicing

A note on hard-coding "magic numbers

- Hard-coding "magic numbers" like `687` or `(0, 5)` should be avoided. Always *calculate* them from your data.
- This is a good argument for using `.loc` over `.iloc`
- If you *really* cannot avoid this, define such things a constant at the beginning of your script:)

```
1  INVALID_ROWS = [33, 42, 18]
2  SPECIAL_ROWS = [120, 111, 230]
3
4  # and then, for example, things like:
5  df.drop(INVALID_ROWS)
6  newdf = df.iloc[SPECIAL_ROWS, :]
```

Working with pandas

Joining and Merging

Joining and Merging

Typical scenario

- You have two datasets that share one column
- For instance, data from www.cbs.nl: one with economic indicators, one with social indicators
- You want to make one dataframe

```
economie = pd.read_csv('82800ENG_UntypedDataSet_15112018_205454.csv', delimiter=';')
economie.head()
```

	ID	EconomicSectorsSIC2008	Regions	Periods	GDPVolumeChanges_1
0	132	T001081	PV20	1996JJ00	9.3
1	133	T001081	PV20	1997JJ00	-2.0
2	134	T001081	PV20	1998JJ00	-0.9
3	135	T001081	PV20	1999JJ00	-0.7
4	136	T001081	PV20	2000JJ00	1.5

```
population = pd.read_csv('37259eng_UntypedDataSet_15112018_204553.csv', delimiter=';')
population.head()
```

	ID	Sex	Regions	Periods	LiveBornChildrenRatio_3
0	290	T001038	PV20	1960JJ00	18.6
1	291	T001038	PV20	1961JJ00	18.9
2	292	T001038	PV20	1962JJ00	18.9
3	293	T001038	PV20	1963JJ00	19.5
4	294	T001038	PV20	1964JJ00	19.6

What do you think: How could/should a joined table look like?

First clean up...

```
# remove unnecessary columns
economie.drop('ID',axis=1,inplace=True)
population.drop('ID',axis=1,inplace=True)
# remove differentiation by sex
population = population[population['Sex']=='T001038']
population.drop('Sex',axis=1,inplace = True)
# keep only rows of economie dataframe that contain the total economic activity
economie = economie[economie['EconomicSectorsSIC2008']=='T001081   ']
economie.drop('EconomicSectorsSIC2008', axis=1, inplace=True)
```

```
# remove those evil spaces at the end of the names of the provinces
population['Regions'] = population['Regions'].map(lambda x: x.strip())
economie['Regions'] = economie['Regions'].map(lambda x: x.strip())
```

```
population.merge(economie, on=['Periods','Regions'], how='inner')
```

	Regions	Periods	LiveBornChildrenRatio_3	GDPVolumeChanges_1
0	PV20	1996JJ00	11.0	9.3
1	PV20	1997JJ00	11.4	-2.0
2	PV20	1998JJ00	11.6	-0.9
3	PV20	1999JJ00	11.6	-0.7
4	PV20	2000JJ00	11.5	1.5
5	PV20	2001JJ00	11.7	3.9
6	PV20	2002JJ00	11.4	2.1

Then merge

On what do you want to merge/join?

Standard behavior of.join(): on the row index (i.e., the row number, unless you changed it to sth else like a date)

```
1 df3 = df1.join(df2)
```

But that's only meaningful if the indices of df1 and df2 mean the same. Therefore you can also join on a column if both dfs have it:

```
1 df3 = df1.merge(df2, on='Regions')
```

`.merge()` is the more powerful tool, `.join()` is a bit easier when joining ion indices.

On what do you want to merge/join?

Standard behavior of.join(): on the row index (i.e., the row number, unless you changed it to sth else like a date)

```
1 df3 = df1.join(df2)
```

But that's only meaningful if the indices of df1 and df2 mean the same. Therefore you can also join on a column if both dfs have it:

```
1 df3 = df1.merge(df2, on='Regions')
```

`.merge()` is the more powerful tool, `.join()` is a bit easier when joining ion indices.

On what do you want to merge/join?

Standard behavior of.join(): on the row index (i.e., the row number, unless you changed it to sth else like a date)

```
1 df3 = df1.join(df2)
```

But that's only meaningful if the indices of df1 and df2 mean the same. Therefore you can also join on a column if both dfs have it:

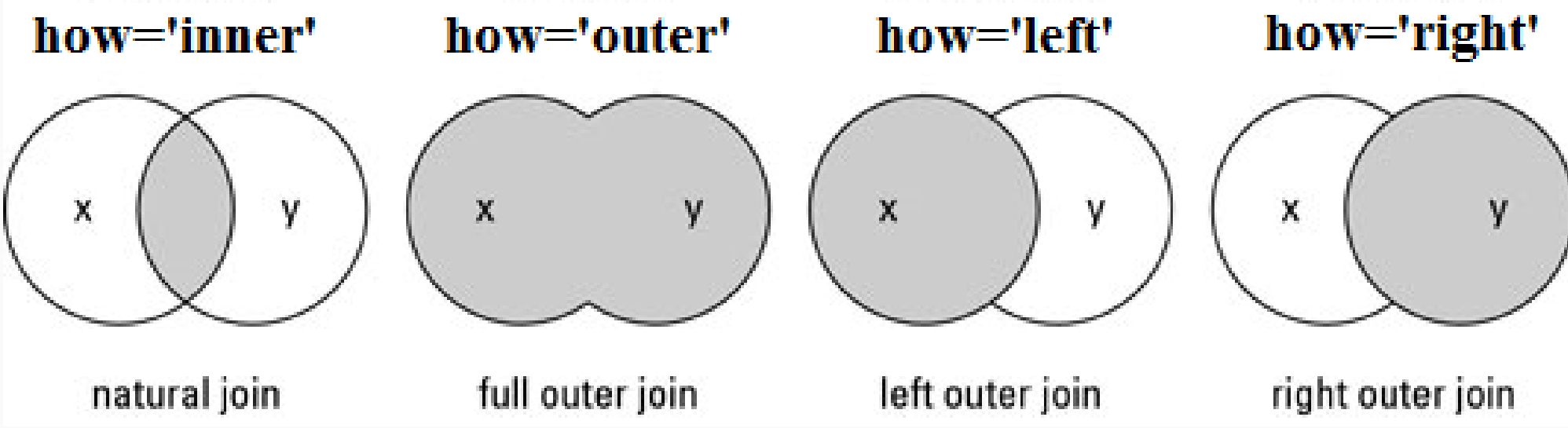
```
1 df3 = df1.merge(df2, on='Regions')
```

`.merge()` is the more powerful tool, `.join()` is a bit easier when joining ion indices.

Inner, Outer, Left, and Right

Main question: What do you want to do with keys that exist only in one of the dataframes?

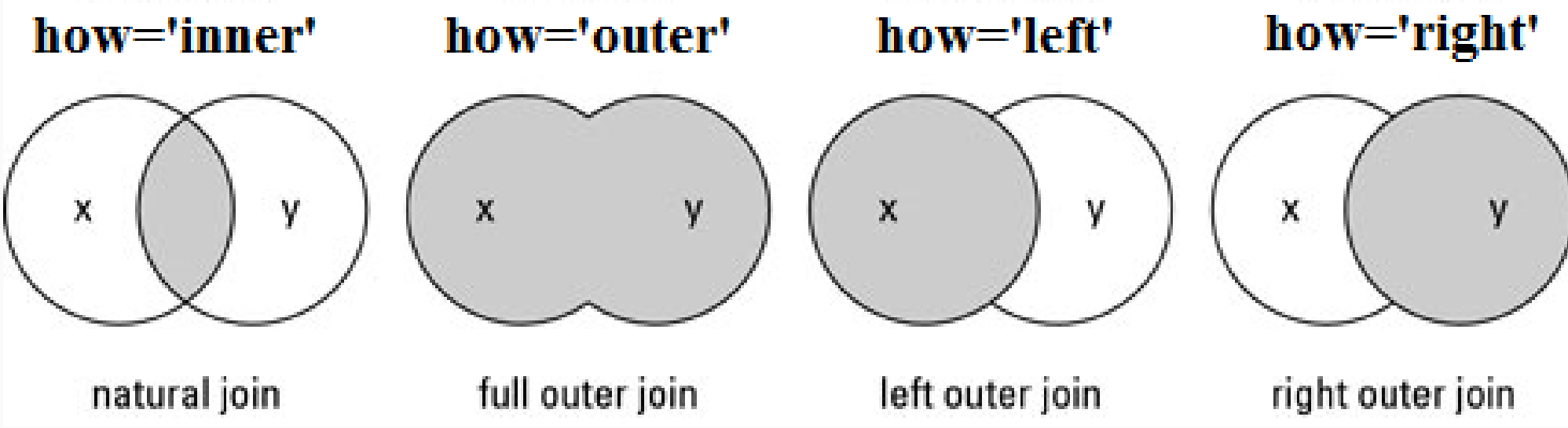
```
df3 = df1.join(df2, how='xxx')
```



Inner, Outer, Left, and Right

Main question: What do you want to do with keys that exist only in one of the dataframes?

```
df3 = df1.join(df2, how='xxx')
```



Working with pandas

Grouping and aggregating

An example

- Suppose you have two dataframes, both containing information on something per region per year.
- You want to merge (join) the two, however, in one of them, the information is also split up by age groups. You don't want that.
- How do you bring these rows back to one row? With `.agg()`!

.agg()

- Very useful after a `.groupby()`
- Takes a function as argument:

```
df2 = df.groupby('region').agg(sum)
```

- Or multiple functions:

```
df2 = df.groupby('region').agg([sum, np.mean])
```

- → yes, you could do `.describe()`, but `.agg()` is more flexible

.agg()

- Very useful after a `.groupby()`
- Takes a function as argument:

```
df2 = df.groupby('region').agg(sum)
```

- Or multiple functions:

```
df2 = df.groupby('region').agg([sum, np.mean])
```

- → yes, you could do `.describe()`, but `.agg()` is more flexible

.agg()

- Very useful after a `.groupby()`
- Takes a function as argument:

```
df2 = df.groupby('region').agg(sum)
```

- Or multiple functions:

```
df2 = df.groupby('region').agg([sum, np.mean])
```

- → yes, you could do `.describe()`, but `.agg()` is more flexible

.agg()

- Very useful after a `.groupby()`
- Takes a function as argument:

```
df2 = df.groupby('region').agg(sum)
```

- Or multiple functions:

```
df2 = df.groupby('region').agg([sum, np.mean])
```

- $\rightarrow$ yes, you could do `.describe()`, but `.agg()` is more flexible

An example

```
1  import numpy as np
2
3  # get all descriptive statistics...
4  df.groupby("gender")['internet'].describe()
5
6  # ... or select specific aggregation functions
7  df.groupby("gender")['internet'].agg([np.mean, np.std])
```

Working with pandas

From wide to long

Wide vs long datasets

R-users know the long dataset also as "tidy" data

```
df_wide.head(5)
```

	Year	U.S.	Japan	Germany	France	U.K.	Italy	Canada	Australia	Spain
0	1970	3.42	2.99	2.25	3.10	3.06	2.39	2.47	3.30	NaN
1	1971	3.41	3.28	2.20	3.04	3.28	2.45	2.52	3.38	NaN
2	1972	3.49	3.73	2.22	3.07	3.54	2.58	2.51	3.44	NaN
3	1973	3.39	4.04	2.18	3.05	3.40	2.53	2.46	3.47	NaN
4	1974	3.21	3.96	2.20	3.03	3.37	2.82	2.39	3.48	NaN

```
df_long.head(5)
```

	Year	country	capital
0	1970	U.S.	3.42
1	1971	U.S.	3.41
2	1972	U.S.	3.49
3	1973	U.S.	3.39
4	1974	U.S.	3.21

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

The problem with wide data

- The information that "U.S.", "Germany" etc. belong together (but not "Year") is nowhere encoded
- Hence, we cannot refer to the "country" for plotting
- The information what the cell entries mean is nowhere encoded (it's the private capital)
- (We could solve that by renaming the columns to "capital_US", "capital_DE" etc. (but that's not very elegant)

Wide data have an advantage, though: It's easier to interpret/read at a glance *for humans*.

Converting from wide to long

We need to specify

- "identifier variables"that remain untouched (`id_vars`)
- how the new variable in which the column names are to be placed is to be called (`var_name`)
- how the new variable in which the cell entries are to be placed is to be called (`value_name`)

Converting from wide to long

```
url="https://cssbook.net/d/private_capital.csv"
df_wide = pd.read_csv(url)
df_long = df_wide.melt(id_vars="Year",
                       var_name="country",
                       value_name="capital")
```

Opposite direction:

```
# note that there is just one value per country per year, so
# min, max, mean, sum... all lead to the same result in this case
# unstack flattens the hierarchical index (try without to see!)
df_wide =
↪  df_long.groupby(['Year','country'])['capital'].agg(min).unstack()
```

Basics of plotting in Python

Main libraries

matplotlib The workhorse and underlying basic of most of the following packages

seaborn More fancy; compromise between Pythonic syntax and ggplot2-like approach

plotnine An attempt to implement R ggplot2 in Python

plotly Fancy plots, support interactivity

bokeh Powerful interactive visualizations

You really should know matplotlib and seaborn (because they are so common), the rest is optional. For your final project, use whatever you want.

Main libraries

matplotlib The workhorse and underlying basic of most of the following packages

seaborn More fancy; compromise between Pythonic syntax and ggplot2-like approach

plotnine An attempt to implement R ggplot2 in Python

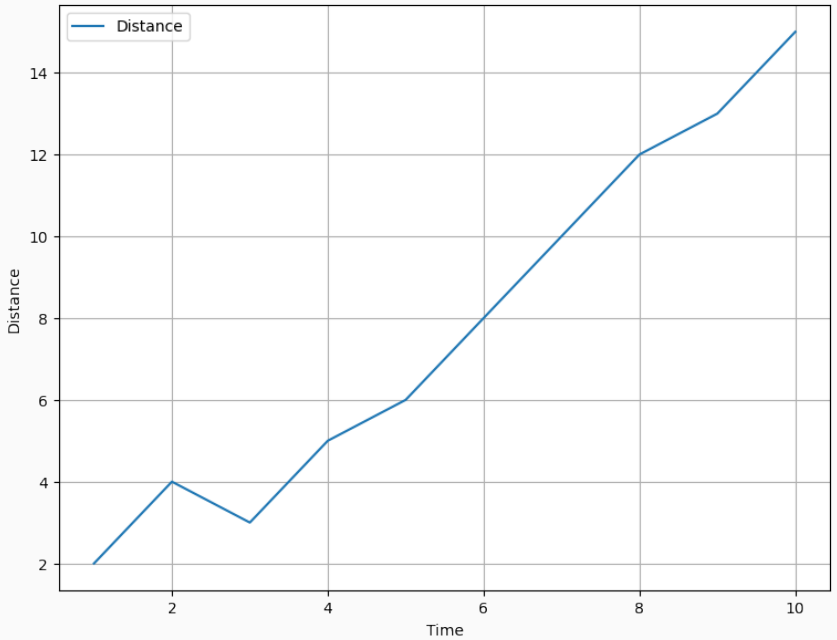
plotly Fancy plots, support interactivity

bokeh Powerful interactive visualizations

You really should know matplotlib and seaborn (because they are so common), the rest is optional. For your final project, use whatever you want.

Two interfaces to Matplotlib

Two ways to produce this graph:



inspired by https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

Matplotlib: Pyplot (MATLAB inspired interface)

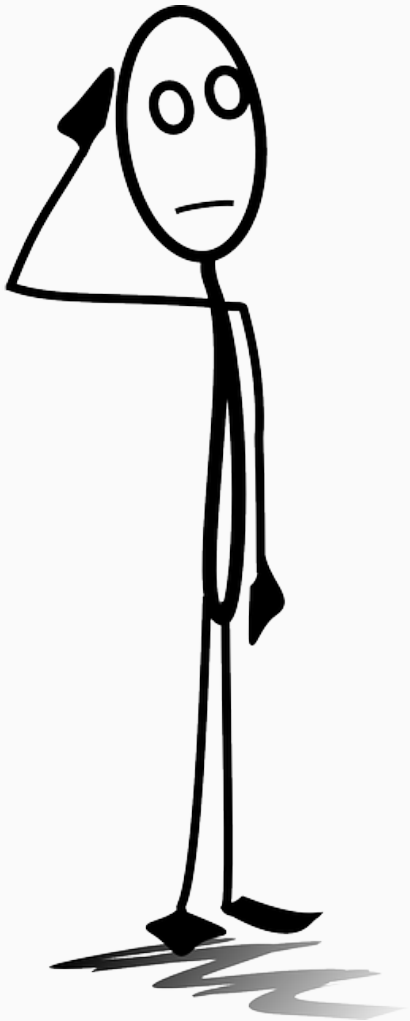
(1) Building up the plot line-by-line

```
1  import matplotlib.pyplot as plt
2
3  x = [1,2,3,4,5,6,7,8,9,10]
4  y = [2,4,3,5,6,8,10,12,13,15]
5
6  plt.figure(figsize=(9,7), dpi=100)
7  plt.plot(x,y)
8  plt.xlabel("Time")
9  plt.ylabel("Distance")
10 plt.legend(["Distance"])
11 plt.grid(True)
```

Matplotlib: Object-oriented interface

(2) Creating and modifying objects

```
1  fig, ax = plt.subplots()      # <-- this is the important difference!!
2
3  ax.set_ylabel("Distance")
4  ax.set_xlabel("Time")
5  ax.plot(x, y, "blue")
6  ax.xaxis.grid()
7  ax.yaxis.grid()
8
9  fig.set_size_inches(9,7)
10 fig.set_dpi(100)
```



Can you think of pros and cons?

Matplotlib: Two interfaces

- You will find examples using both styles
- Pyplot: can be shorter/quicker/easier
- OO: because you get named objects, you can modify them/refer to them/etc:

"Also, the pyplot approach doesn't really scale when we are required to make multiple plots or when we have to make intricate plots that require a lot of customisation. However, internally matplotlib has an Object-Oriented interface that can be accessed just as easily, which allows to reuse objects."

https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

Matplotlib: Two interfaces

- You will find examples using both styles
- Pyplot: can be shorter/quicker/easier
- OO: because you get named objects, you can modify them/refer to them/etc:

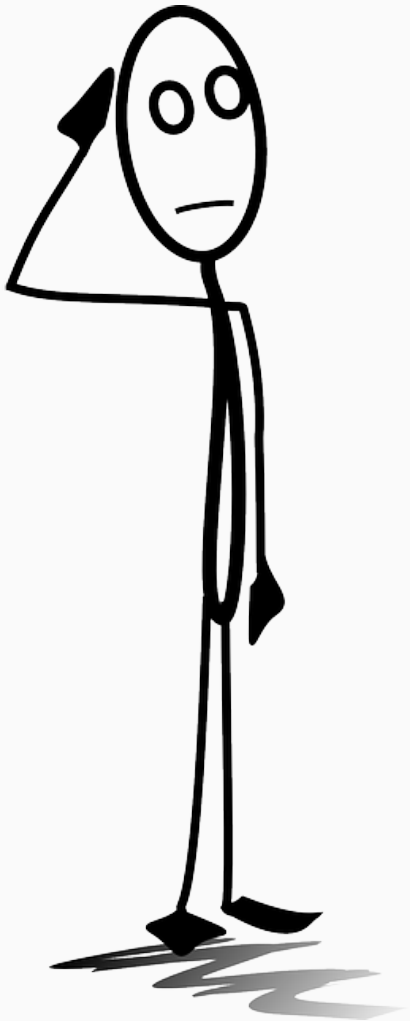
"Also, the pyplot approach doesn't really scale when we are required to make multiple plots or when we have to make intricate plots that require a lot of customisation. However, internally matplotlib has an Object-Oriented interface that can be accessed just as easily, which allows to reuse objects."

https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

Pandas and Plotting

- As we have seen, we can directly plot from lists
- But if we have a dataframe *anyway*, we can:
  - use the build-in `.plot()` method to have pandas handle the (default: matplotlib) backend; or
  - use seaborn directly on the dataframe

Let's look at https://github.com/uvacw/teaching-bdaca/blob/main/modules/basics/visualization.ipynb for a demonstration.



Any questions?

Next steps

Make sure you understood all of today's concepts.

Re-read the chapters.

I prepared exercises to work on *during* the Friday meeting (alone or in teams):

https://github.com/uvacw/teaching-bdaca/blob/main/12ec-course/week03/exercises/

References

McKinney, W. (2012). *Python for data analysis*. Sebastopol, CA, O'Reilly.