

Chapter 2

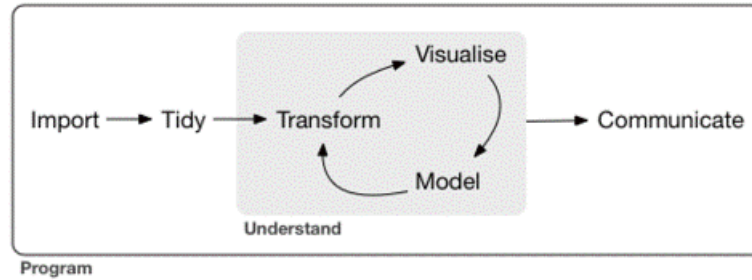
Data preparation

Prof. Dr. K. Verbarg

Outline

Preparation overview

How to get started



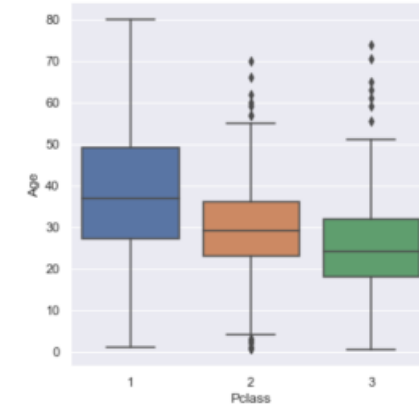
2 Data preparation

© Prof. Dr. K. Verburg

3

Range of values

Check the data quality and do some simple plotting
for every column separately



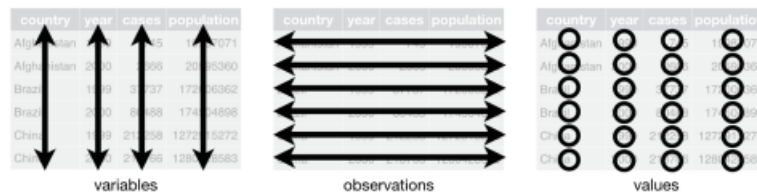
2 Data preparation

© Prof. Dr. K. Verburg

10

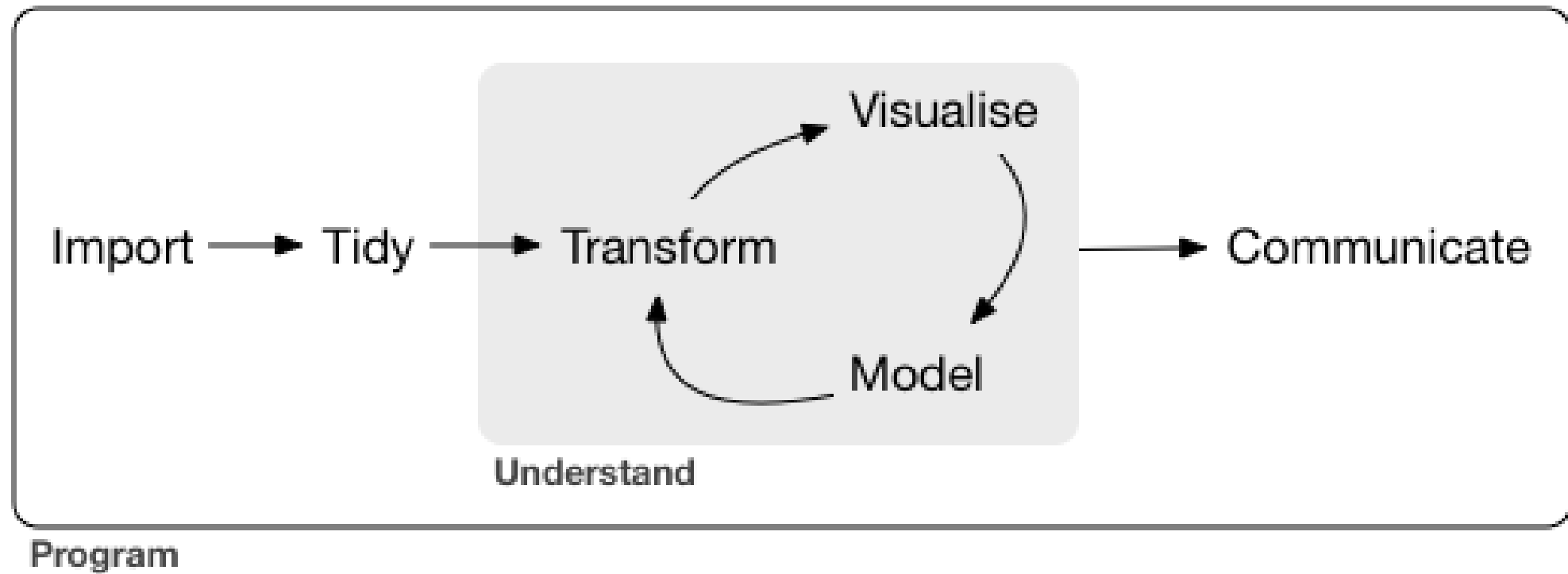
Tidy data

Reshaping data to ease analysis



Preparation overview

How to get started

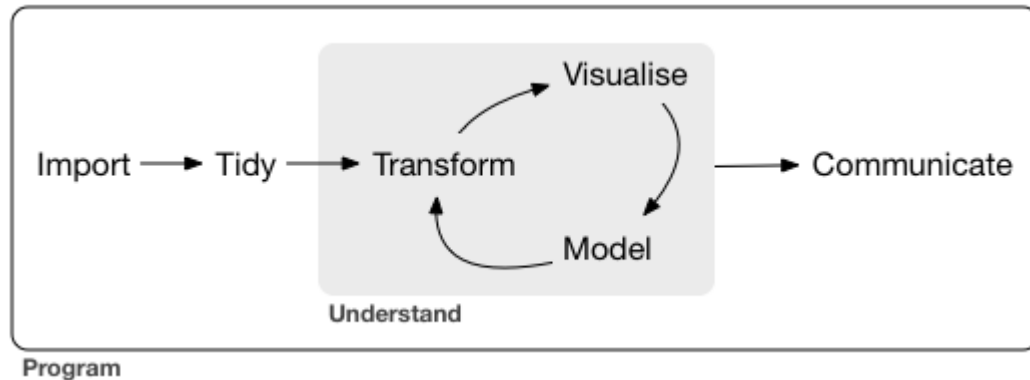


Data preparation overview

We now know the basic tools to manipulate data.

To do data analysis, we need to do the following steps according to the "**R for data science**" book

<https://r4ds.had.co.nz/>



Import is just the first technical step.

Tidying and **transforming** is what we discuss in this chapter. It is about preparing / munging / wrangling / cleansing data, so as to do standard reporting (**visualize**) or **modelling** (statistical models, machine learning) afterwards.

Communicate again is a more technical step of documenting your work and results with Latex or the like.

So, what are the basic tasks to do tidying and transforming?

- Get a general impression of the data (see this section).
- Look at every column and check the range of values. We hereby want to check the data quality.
- Reshape the data into a standard format (tidy data)
- Do additional transformations with respect to the intended analysis (we already know how to do that):
 - filter for observations of interest
 - create new variables = compute calculated columns
 - compute summary statistics

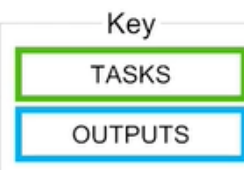
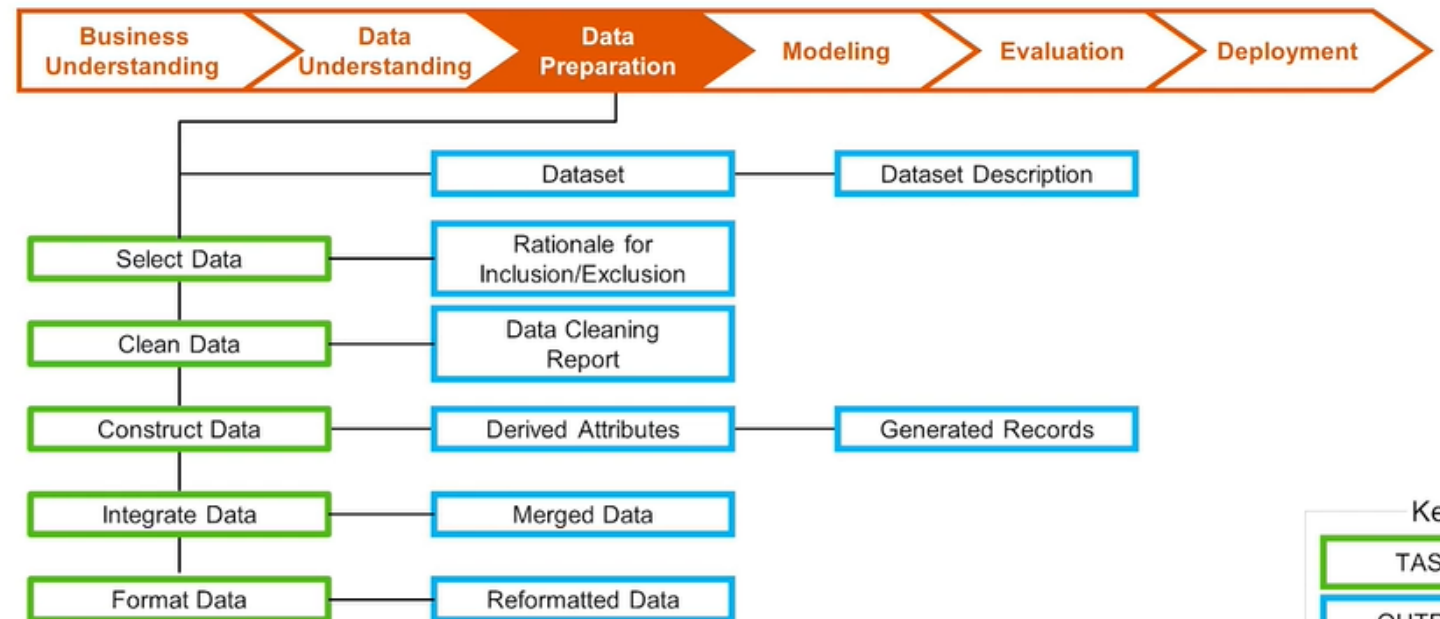
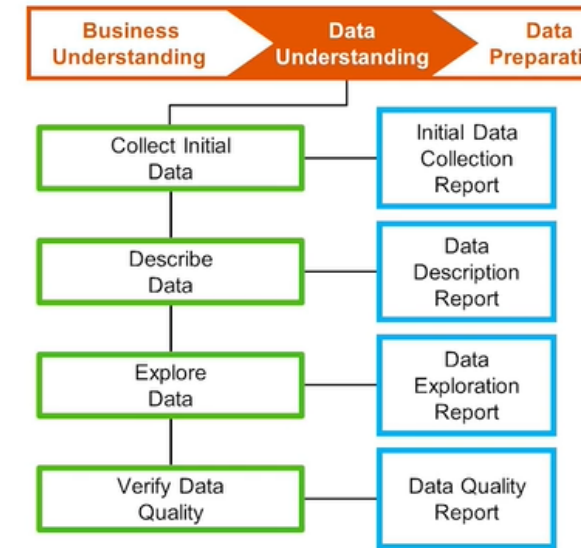
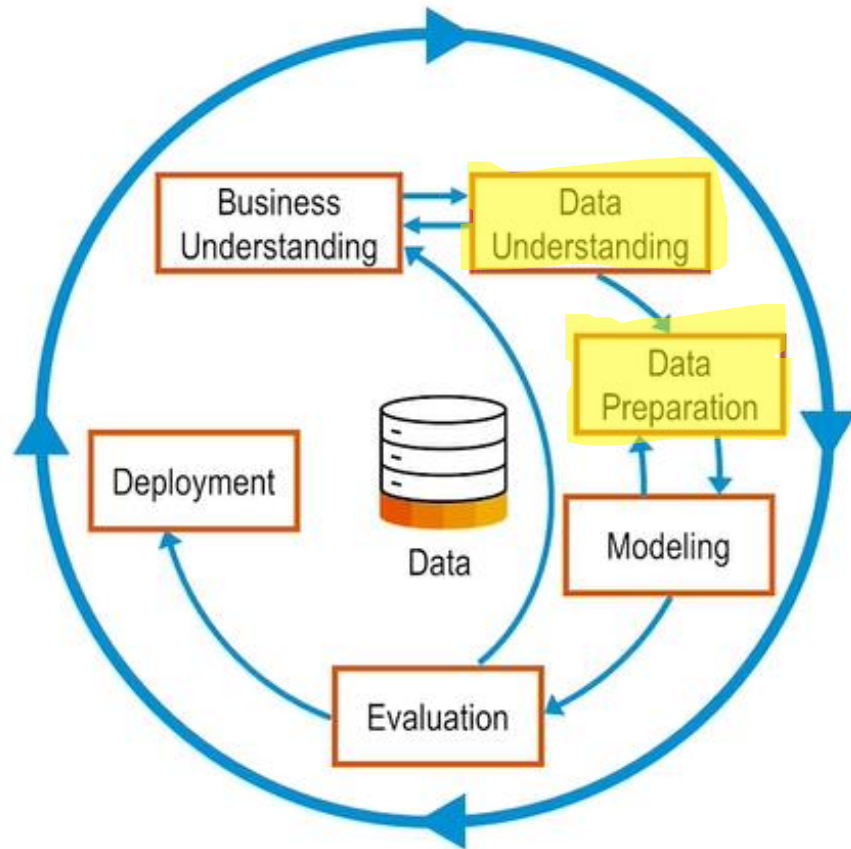
It is said that data preparation really is the major part of the work and it is vital for validity of the results.

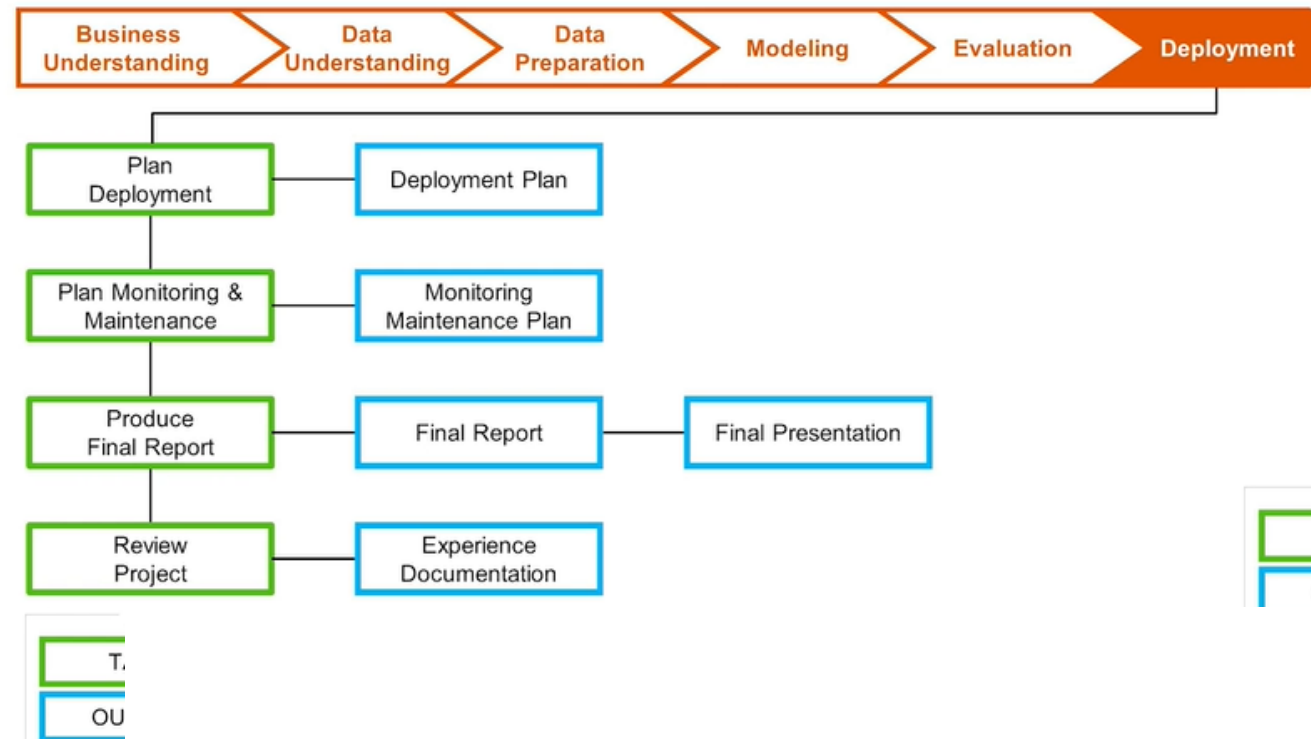
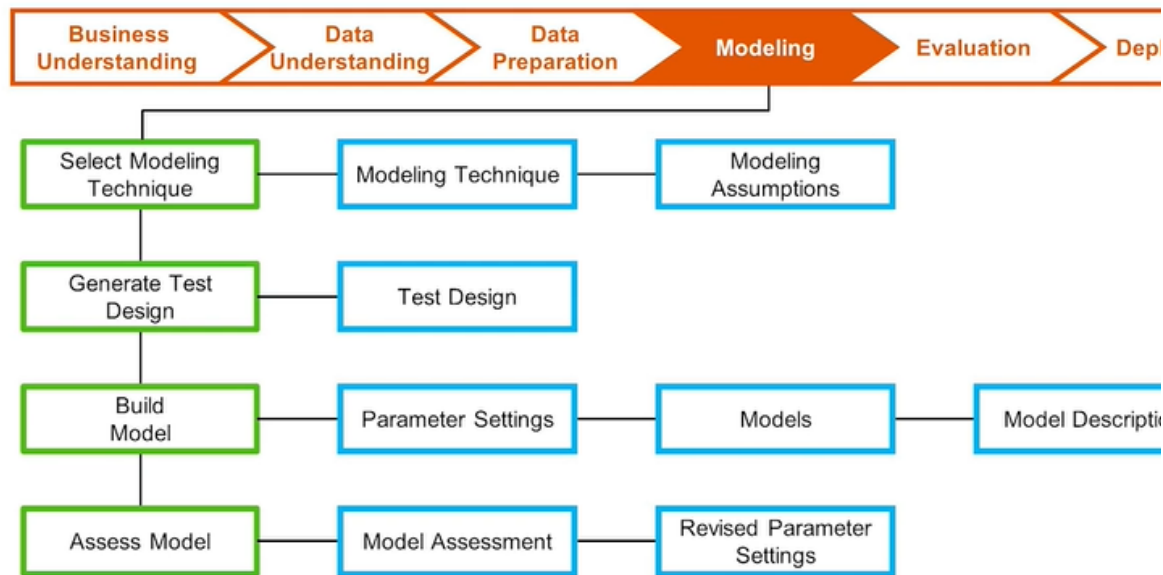
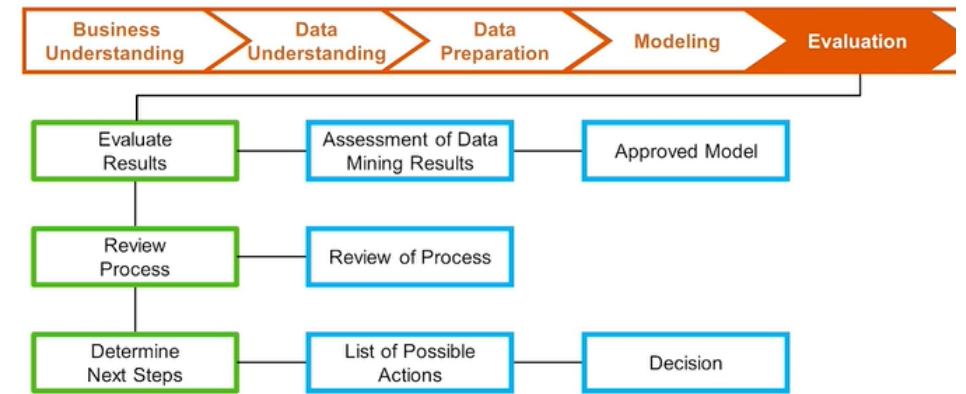
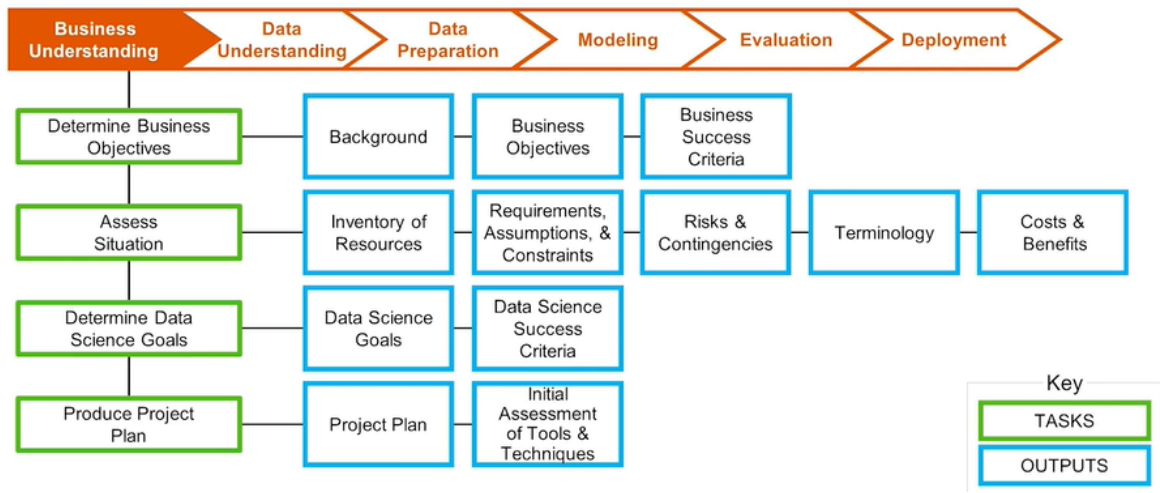
A New York Times article reported that data scientists spend from **50% to 80%** of their time mired in the more mundane task of collecting and preparing unruly digital data before it can be explored for useful nuggets.

For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights.
New York Times. STEVE LOHR.
AUG. 17, 2014

CRISP-DM

The **cross-industry standard process for data mining (CRISP-DM)** specifies a comparable methodology.





Data Understanding Phase – Overview

Phase 2.1: Collect Initial Data

- **Task**

- Acquire the data (or access to the data) listed in the project resources.
- This initial collection includes data loading into the data exploration tool and data integration if multiple data sources are acquired.

- **Output – Initial Data Collection Report**

- List the following:
 - The dataset (or datasets) acquired
 - The dataset locations
 - The methods used to acquire the datasets
 - Any problems encountered
- Record problems encountered and any solutions

Data Understanding Phase – Overview

Phase 2.2: Describe Data

- **Task**

- Examine the “gross” or “surface” properties of the acquired data and report on the results.

- **Output – Data Description Report**

- Describe the data that has been acquired, including:
 - The format of the data
 - The quantity of data, e.g. the number of records and fields in each table
 - The identities of the fields
 - Any other surface features of the data that have been discovered

Data Understanding Phase – Overview

Phase 2.3: Explore Data

- **Task**

- This task tackles the data science questions, which can be addressed using querying, visualization, and reporting.

- **Output – Data Exploration Report**

- Describe results of this task, including:
 - First findings or initial hypothesis and their impact on the remainder of the project
 - If appropriate, include graphs and plots

Data Understanding Phase – Overview

Phase 2.4: Verify Data Quality

- **Task**

- Examine the quality of the data, addressing questions such as:
 - Is the data complete?
 - Is it correct or does it contain errors?
 - Are there missing values in the data?

- **Output – Data Quality Report**

- List the results of the data quality verification
- If quality problems exist, list possible solutions

Look at the DataFrame

To get started, take a look at your data using:

- `df.columns`, `df.index`
- `df.dtypes`
- `df.shape`
- `df.info()`
- `df.head()`
- `df.sample(10)`
- `df.describe()`

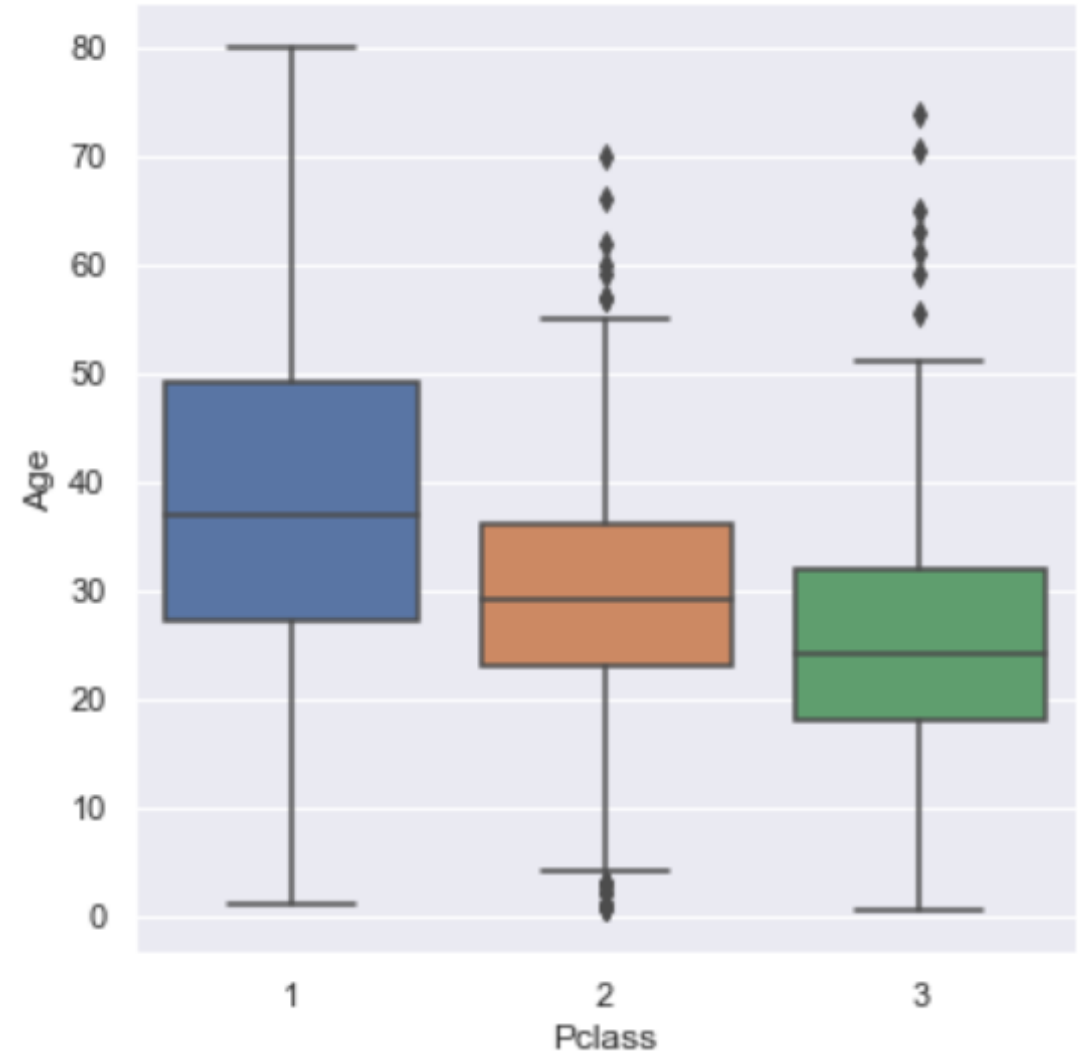
```
titanic[["Age", "Fare"]].describe()
```

	Age	Fare
count	714.000000	891.000000
mean	29.699118	31.489295
std	14.526497	43.973014
min	0.420000	0.000000
25%	20.125000	7.910400
50%	28.000000	14.454200
75%	38.000000	31.000000
max	80.000000	300.000000

- Is that really the (complete and correct) data you are expecting?
- Are the **data types** fitting?
In case of mixed text and numbers, we might need the "object" data type.
We already discussed converting data types and Null-handling when that fails (Chapter 1, "Using pd.NA").
- Do you have a clear understanding of the **contents** (the semantics) of your columns?

Range of values

Check the data quality and do some simple plotting
for every column separately



Data type

As already discussed in Chapter 1, we want the optimal data type for each column.

- We would avoid "object".
- Dates and times will be discussed in the next chapter.
- "category" may be more efficient in terms of space (we do not use it here – it needs additional explanation).

Pandas dtype	Usage
object	Text (Python <code>str</code>) or mixed values (e.g. numeric and non-numeric) – usually less efficient, since calculations are done on Python level
Int64	Integer numbers (not "int64")
Float64	Floating point numbers (not ("float64"))
<u>boolean</u>	True / False values (not "bool")
string	This is not chosen by default ;- (object is preferred)
datetime64[ns]	Date and time with unit nanoseconds
<u>timedelta[ns]</u>	Differences between two datetimes
category	Finite list of text values

Data format

Looking at the content of a column, we want to have it formatted uniformly and as expected.

Examples:

- **Translation of coded values:** “1” (*meaning male*) → “M”, “2” → “F”
- **Standardization of date and number formats**
2014-01-01 01.01.2014 01/01/14
 01. Januar 2014 01JAN14
2050,2 2.050,20 2,050.20 (*amount*)
K12014002 K1/002.2015 (*fiscal period*)
AB23123 0000000000AB23123 (*material number*)
- Standardization of strings into **capital letters**
MAleR → MAIER
- filtering **illegal characters**
Eva Czépuřnikš → Eva Czepurnik
Material number ABŘ123š → AB123
- Converting **units of measure:** G → KG
- Convert attributes **into bins:**
age classes, quantiles

What can we do?

- Mapping / modifying values using vector functions (next slide)
- Dates see next Chapter
- String vector operations (subsequent slide)
- Convert attributes into bins (subsequent slide)

Repetition: Find out the range of values

```
set(titanic["Embarked"])
```

```
{'C', 'Q', 'S', nan}
```

Mapping values

Let's start with `map()`. It works on Series (=columns). It is an elementwise mapping (substitution of values) defined by a dictionary (or a Series or a function).

```
titanic["Sex"].map({"male" : "M", "female" : "F"})
```

```
0      M
1      F
2      F
```

```
titanic["Sex"].map(lambda x: f'I am {x}.')
```

```
0      I am male.
1      I am female.
2      I am female.
```

Note that `map()` is a similar method on DataFrames working on all cells elementwise (hence on all columns).

For more complex mappings / computations, we can use `apply()` instead. On a DataFrame it works row-wise (with `axis='columns'`) and applies a function to (potentially multiple) columns of each row.

Remember that in contrast to DataFrames, using `apply()` on groupings aggregates each $n \times m$ group into the same or a smaller shape (this is confusing and was already discussed in Chapter 1).

To use these mapping methods in a chained command, we would have to embrace them with `assign()`.

Vector operations

We just discussed element-wise mapping of values. In some situations, we have vectorized methods available. These are then to be preferred (easier, better performance).

We already discussed those in Chapter 1:

- Extended projection

```
df["death"] / df["cases"] * 100
```

```
0      NaN
1    0.000000
2   12.727273
3   12.121212
dtype: float64
```

- Boolean vectors with `.loc[]`

```
tips.loc[tips["tip"] > 7]
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
170	50.81	10.00	Male	Yes	Sat	Dinner	3
212	48.33	9.00	Male	No	Sat	Dinner	4

For string manipulation, Python provides a bunch of methods.

```
'Weird sTRinG'.lower()
```

```
'weird string'
```

Instead applying such scalar methods to DataFrames using `.map()`, pandas provides **vectorized string methods**. These work on Series (hence we need a projection on one column first). To access the values, we use the str-attribute.

```
titanic["Name"].str.lower()
```

```
0      braund, mr. owen harris
1  cumings, mrs. john bradley (florence briggs th...
2      heikkinen, miss. laina
3  futrelle, mrs. jacques heath (lily may peel)
4      allen, mr. william henry
```

String vector operations

To find available string operations, check the pandas documentation under [pandas.Series.str](#).

This is an incomplete overview.

We start with simple methods which usually originate from a Python scalar pendant (example: `str.lower`).

How is Null-handling different for vector methods compared to element-wise `.map()`?

<code>len</code>	Length of string (or tuple or list)
<code>lower, upper, title, capitalize, swapcase</code>	Modify the case
<code>isalnum, isalpha, isdigit, isspace, istitle, islower, isupper, isnumeric, isdecimal, startswith, endswith</code>	Returns True / False based on the checked condition
<code>strip, lstrip, rstrip</code>	Remove trailing whitespace
<code>pad</code>	Pad with fill character
<code>translate</code>	Map characters based on mapping table
<code>find, rfind, index, rindex, slice</code>	Find substring
<code>split</code>	Split string at separator (<code>expand=True</code> makes df-columns instead of list)
<code>join</code> <code>cat</code>	Concatenate strings in list Concatenate strings of two Series
<code>repeat</code>	Duplicates values: <code>Series.str.repeat(3)</code> is equivalent to <code>x * 3</code>
<code>wrap</code>	Wrap long texts into lines with maximal length

String vector operations with regular expressions

The Python `re`-module provides methods for regular expressions.

First, define a regular expression (a pattern):

```
import re

text = ' foo bar      bar\tbaz  \n\t\nabc'
regex = re.compile(r'\s+') # one or more whitespace chars
```

Then, use it to analyse or modify the text:

```
regex.split(text)
```

```
['', 'foo', 'bar', 'bar', 'baz', 'abc']
```

```
regex.findall(text)
```

```
[' ', ' ', ' ', ' ', '\t', ' ', '\n\t\n']
```

```
print(regex.match(text))
```

```
<re.Match object; span=(0, 2), match=' '>
```

```
regex.sub('*', text)
```

```
'*foo*bar*bar*baz*abc'
```

contains	Applies <code>re.search()</code> and returns Boolean (True/False/NA) (match pattern at first occurrence in string)
match, fullmatch	Applies <code>re.match()</code> and returns Boolean (match pattern at beginning of string)
extract, extractall	Applies <code>re.match()</code> and returns DataFrame with a column for each matched group
findall	Applies <code>re.findall()</code>
replace	Replace pattern with other string
count	Counts occurrences of pattern
split, rsplit	Split string at separator given by pattern <ul style="list-style-type: none">access the <i>i</i>-th element of the split with <code>.str.split(...).str.get(i)</code>or, split strings into separate columns with <code>expand=True</code>

```
t["Name"].str.extract(r'(\w+),')
```

```
0
0 Braund
1 Cumings
2 Heikkinen
```


Convert attributes into bins

We want to categorize values.

$$\text{Age class} = \begin{cases} \text{'child'} \\ \text{'teenager'} \\ \text{'adult'} \\ \text{'elder'} \\ \text{pd.NA} \end{cases}, \text{ if Age} \in \begin{cases} [0,12] \\]12,18] \\]18,65] \\]65,100] \\ \text{else} \end{cases}$$

```
bins = [0, 12, 18, 65, 100]
labels = ['child', 'teenager', 'adult', 'elder']
titanic["Age class"] = pd.cut(titanic["Age"], bins, labels=labels)
titanic[["Age", "Age class"]]
```

	Age	Age class
0	22.0	adult
1	38.0	adult

```
groups = titanic.groupby("Age class")
groups.size()
```

```
Age class
child      69
teenager   70
adult     567
elder       8
```

Note that the dtype of "Age class" is `category`.

We can also do bins based on quantiles with `qcut()`.

To do more complex categories, we might have to use `apply()` with a custom function instead.

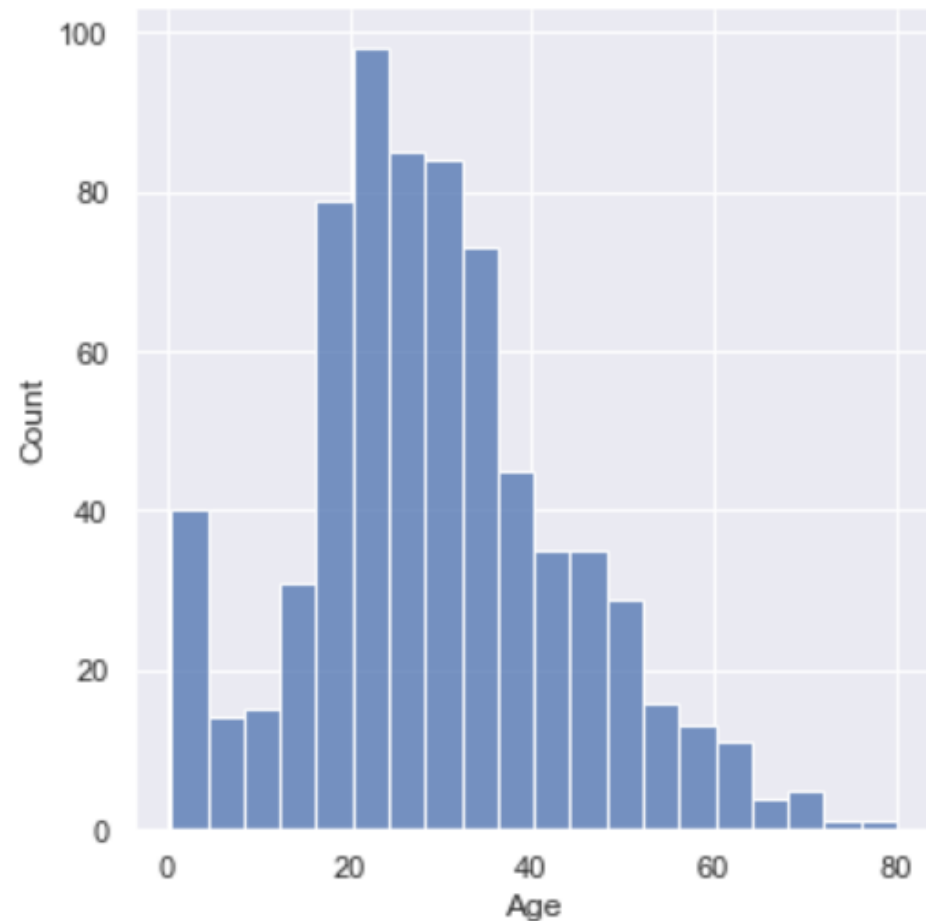
Instead of displaying distributions over bins in a tabular format, we also might want to do some graphics.

Plotting with seaborn

```
import seaborn as sns
```

```
sns.displot(data=titanic, x="Age")
```

```
<seaborn.axisgrid.FacetGrid at 0x1480cc16bb0>
```



The library for plotting in pandas is **matplotlib**.

pandas has plotting methods for DataFrames which simply invoke related matplotlib functions (`DataFrame.plot`, `DataFrame.hist`, etc.).

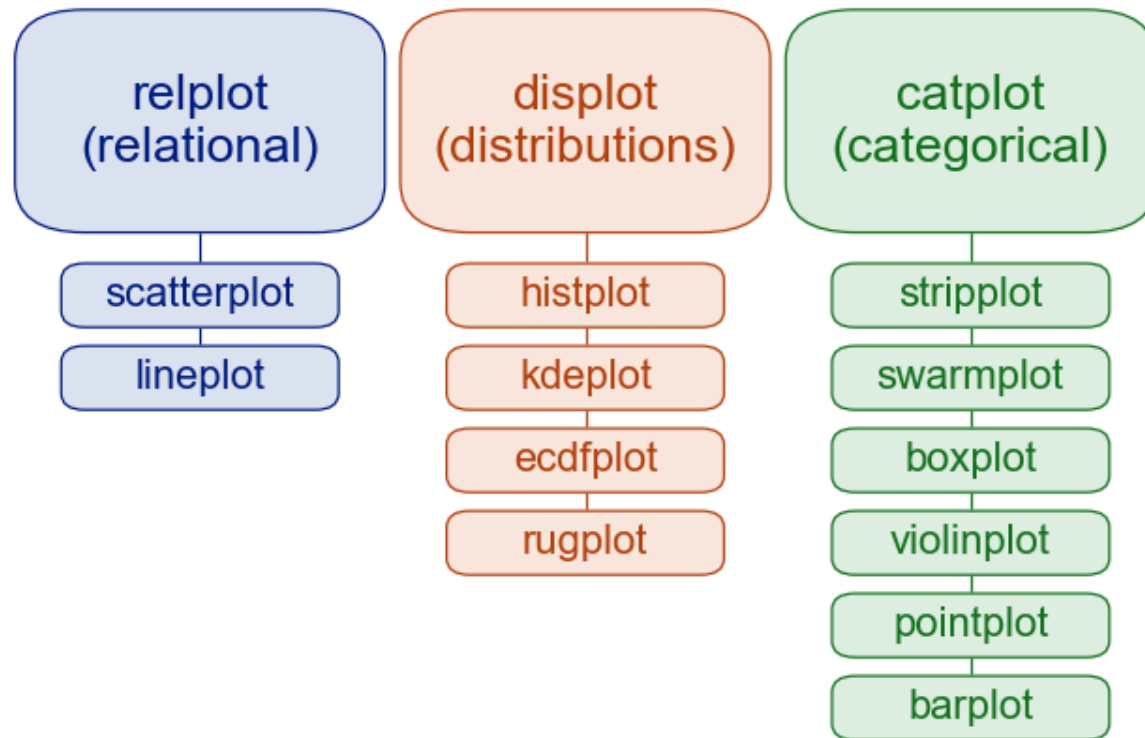
We go for **seaborn** here. It seems to be easier to use, it has a more modern appearance and has some build-in statistics. seaborn understands pandas DataFrames and GroupBy-objects.

seaborn is also based on matplotlib. Hence, it might be (rarely) necessary to use basic matplotlib commands to achieve what is not available with the simple seaborn interface. Also, you might have to check the matplotlib documentation to understand the related seaborn command.

Seaborn function overview

It is advisable to use "figure-level" functions only (seaborn takes care of the entire figure, not just the plot inside the axes). They serve as a wrapper around the various kinds of plots. We have the three top-level functions (`relplot`, `displot`, `catplot`), plus `jointplot` and `pairplot`.

For a first overview and documentation, see <https://seaborn.pydata.org/introduction.html>

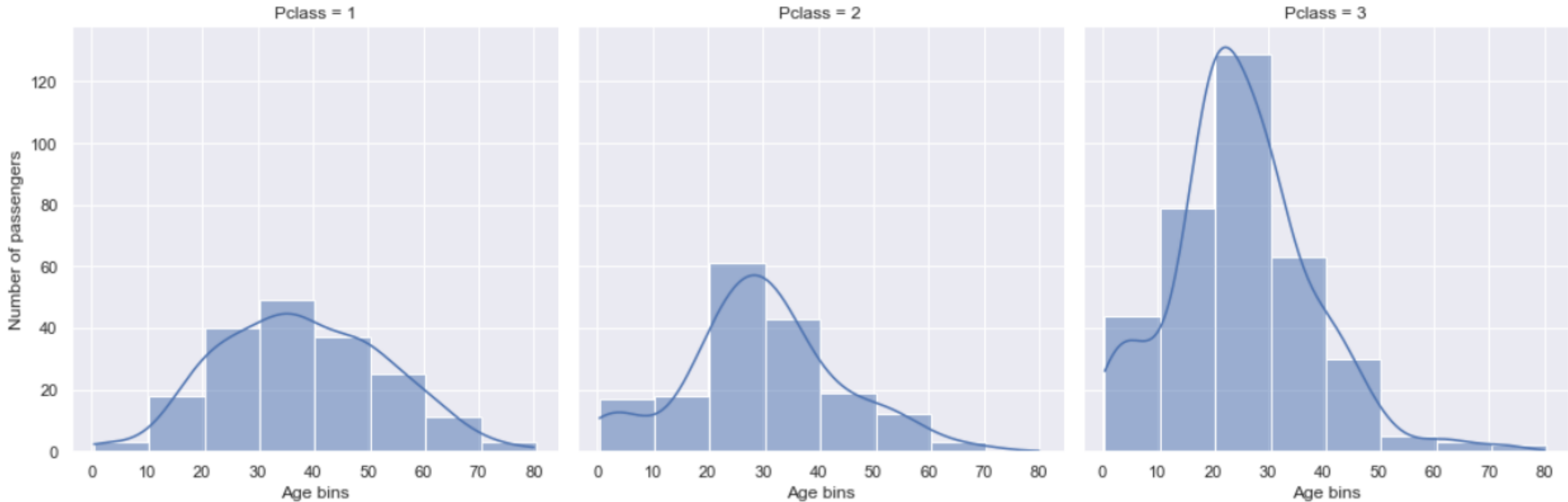


https://seaborn.pydata.org/tutorial/function_overview.html

Some plotting options

Where can we find documentation on the options?

```
g = sns.displot(data=titanic, x="Age", col="Pclass", binwidth=10, kde=True)
g.set_axis_labels("Age bins", "Number of passengers")
g.savefig("Sample figure.png", format='png', dpi=150)
```



Bar plot

Manual analysis:

What are the chances to survive by Age and Sex?

We can compute it manually...

```
(titanic.groupby(["Age class", "Sex"])\n  ["Survived"].mean()\n)
```

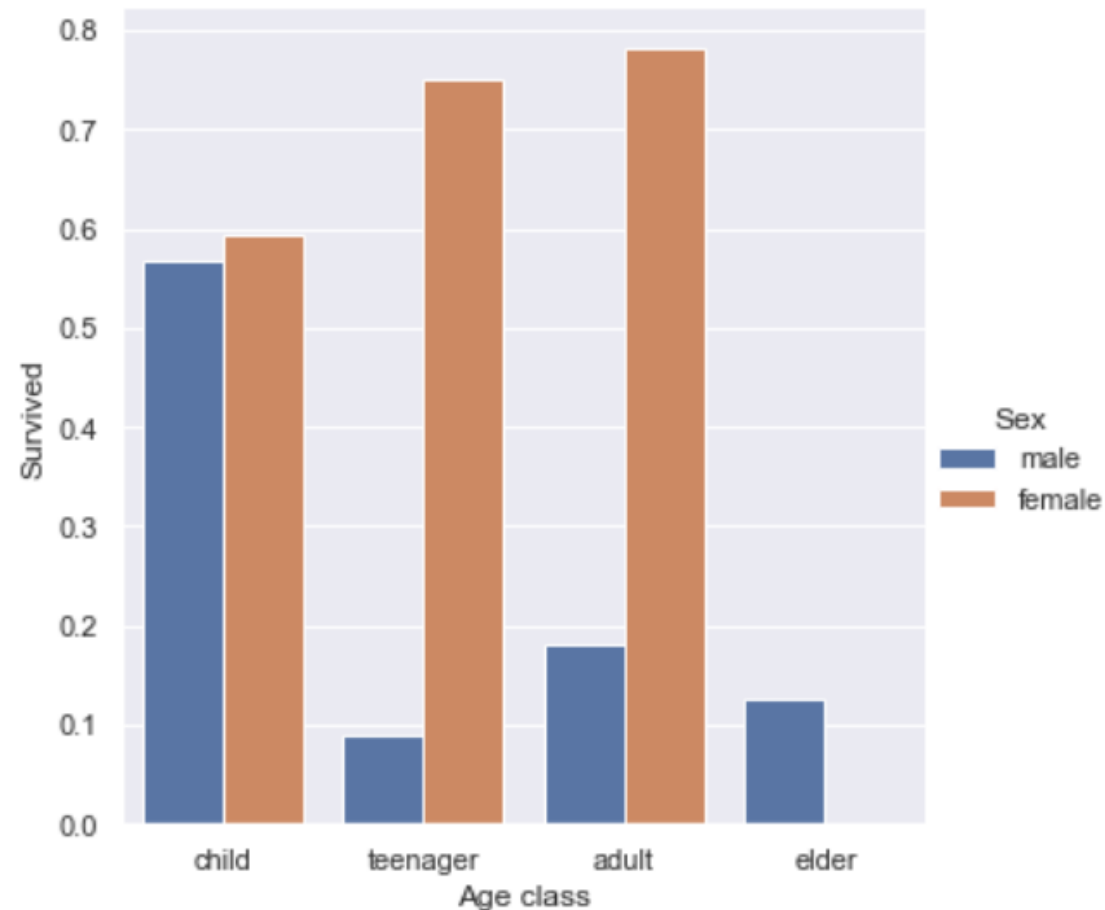
Age class	Sex	
child	female	0.593750
	male	0.567568
teenager	female	0.750000
	male	0.088235
adult	female	0.782383
	male	0.181818
elder	female	NaN
	male	0.125000

Name: Survived, dtype: float64

... or use a bar plot (the age class for x-values already has bins defined – otherwise use `displot`)

```
sns.catplot(kind="bar", data=titanic, x="Age class", y="Survived", hue="Sex", ci=None)
```

<seaborn.axisgrid.FacetGrid at 0x14812e97340>



Line plot

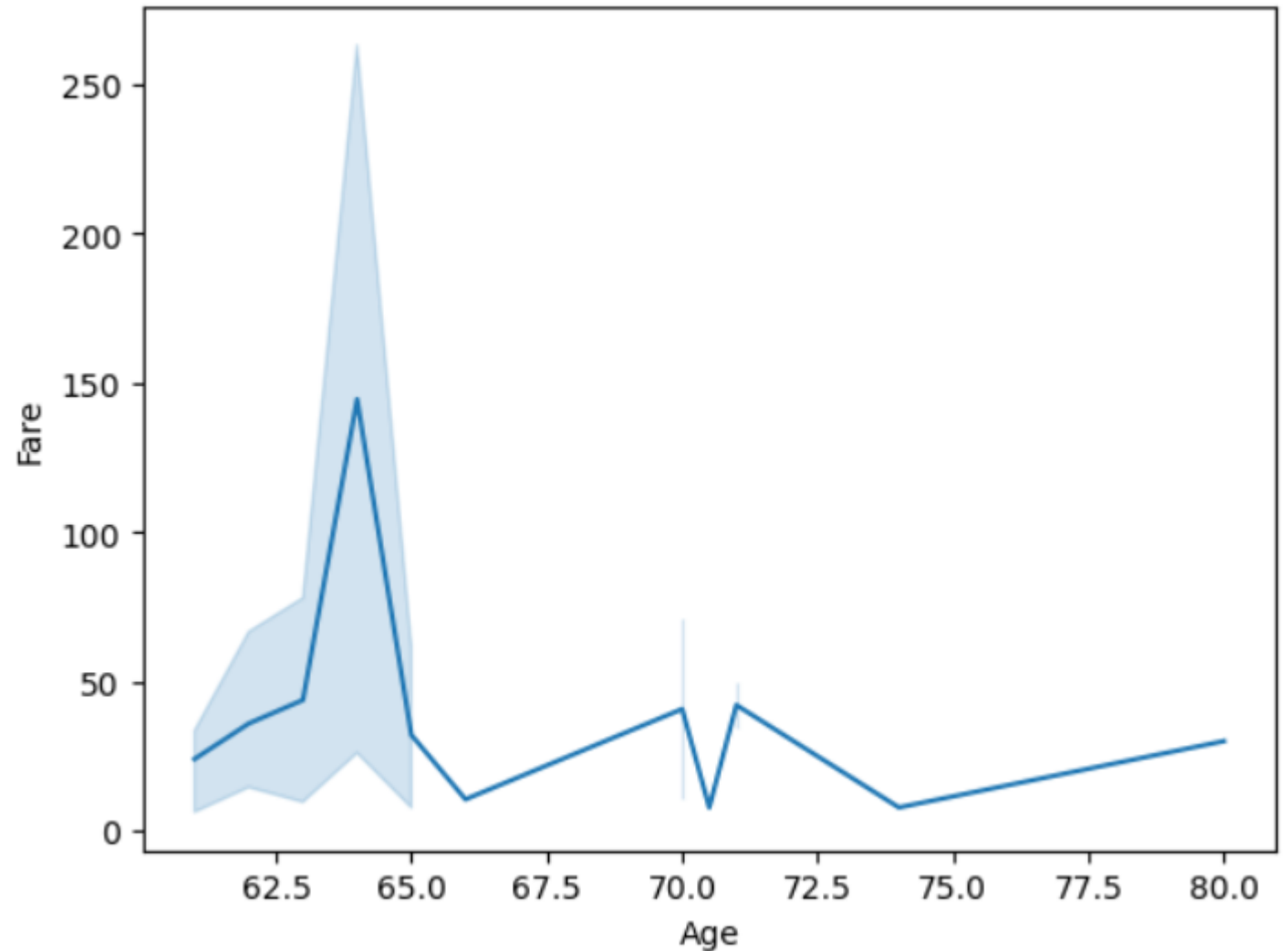
`lineplot` also aggregates common x-values. The blue line is the standard estimator 'mean'.

The blue shaded area depicts the range of y-values. `errorbar=None` would remove it.

Format the line appearance using e.g.:

```
, linestyle='dashed'  
, linewidth=3  
, color='grey'
```

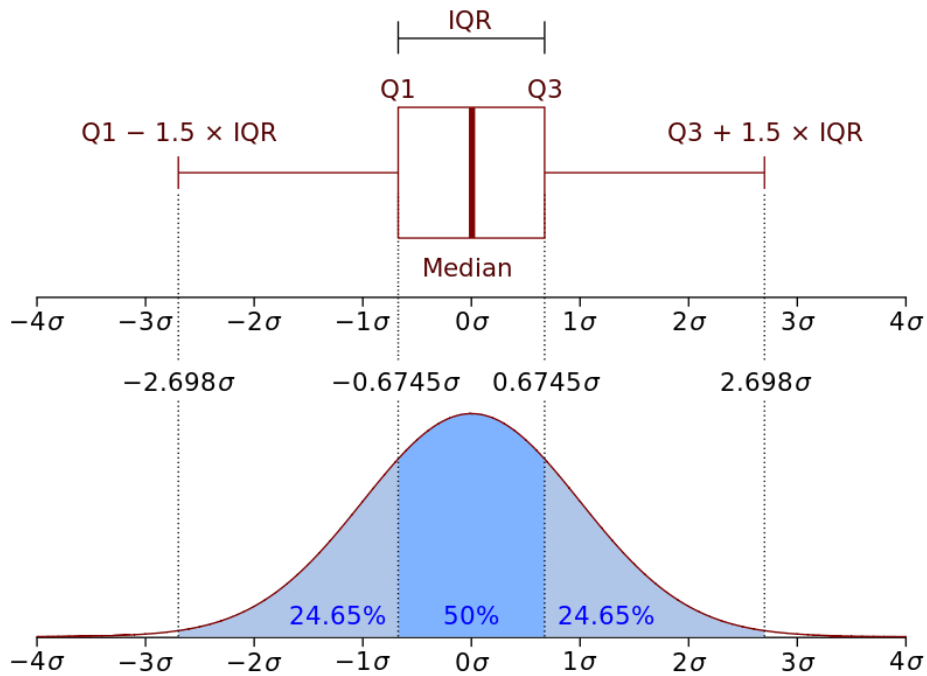
```
sns.lineplot(data=titanic.loc[titanic["Age"] > 60], x="Age", y="Fare", estimator='mean')
```



Boxplot

Another option to visualize the distribution is the standard box plot.

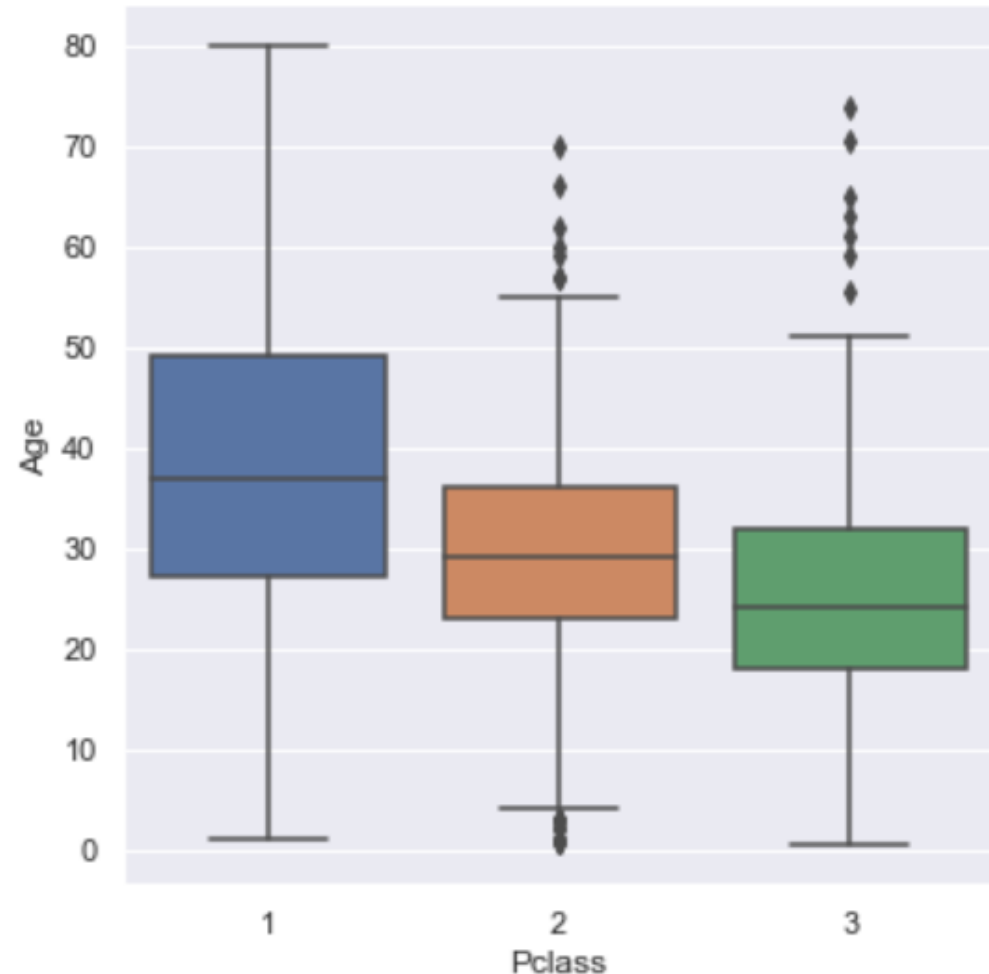
- Lower and upper quartile displayed as a box (50 % of the data points)
- Median is a line in the box
- Whiskers extend to 1.5 IQR (interquartile range)
- Show outliers outside whiskers as points



© Wikipedia, boxplot

```
sns.catplot(kind="box", data=titanic, y="Age", x="Pclass")
```

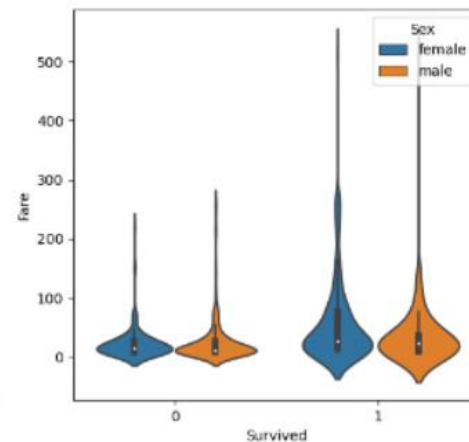
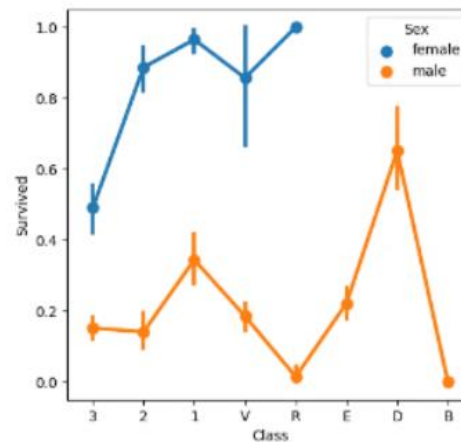
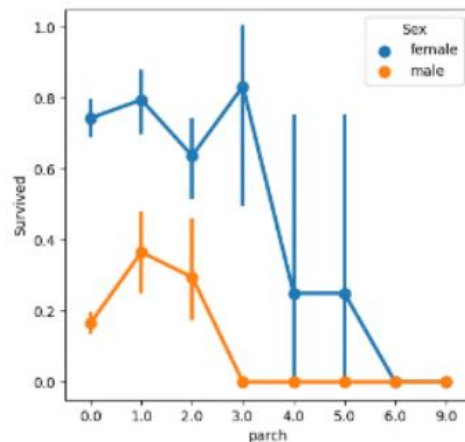
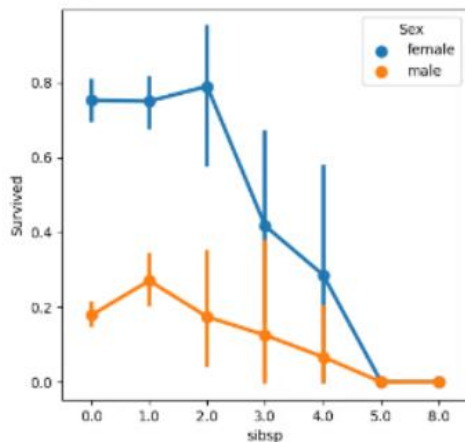
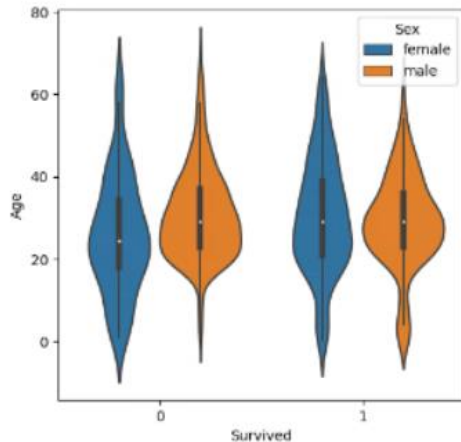
<seaborn.axisgrid.FacetGrid at 0x14812ea0130>



Multiple plots

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, axs = plt.subplots(ncols=5, figsize=(30,5))
sns.violinplot(x="Survived", y="Age", hue="Sex", data=titanic, ax=axs[0])
sns.pointplot(x="sibsp", y="Survived", hue="Sex", data=titanic, ax=axs[1])
sns.pointplot(x="parch", y="Survived", hue="Sex", data=titanic, ax=axs[2])
sns.pointplot(x="Class", y="Survived", hue="Sex", data=titanic, ax=axs[3])
sns.violinplot(x="Survived", y="Fare", hue="Sex", data=titanic, ax=axs[4])
```

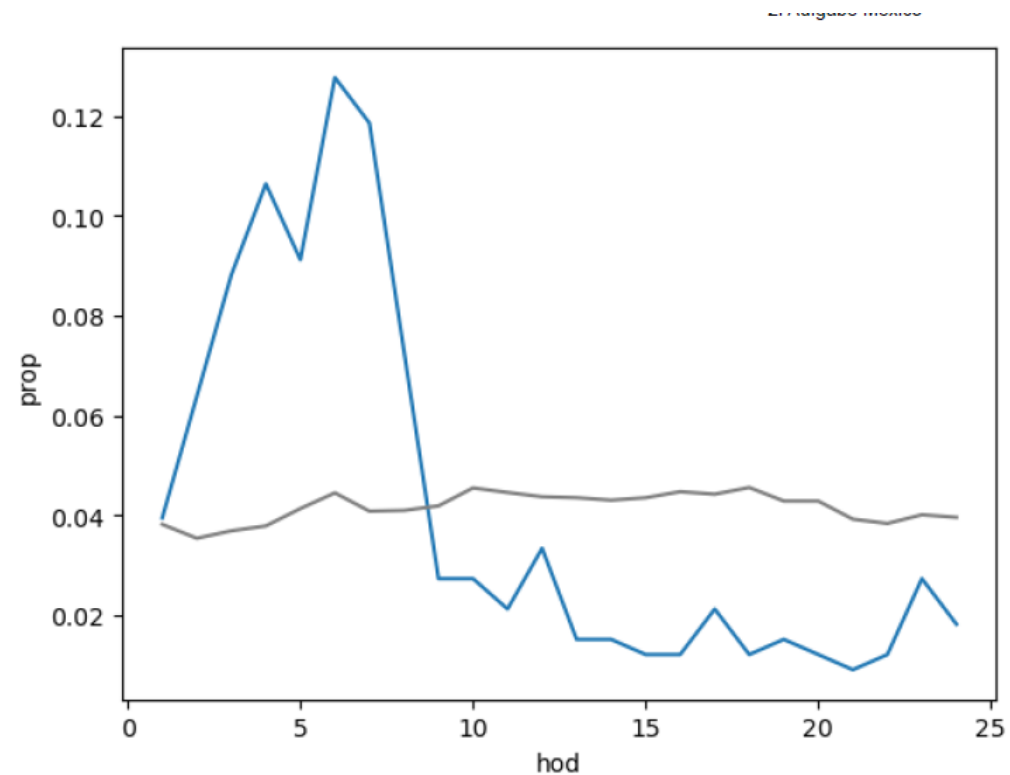


Multiple plots in one figure

```
# Create figure and axes
fig, ax = plt.subplots()

# Plot data from hod_unusual data frame
sns.lineplot(x='hod', y='prop', data=infants, ax=ax)

# Plot data from overall data frame
sns.lineplot(x='hod', y='prop_all', data=overall, color='grey', ax=ax)
```



Verify the range of values

So far, we checked:

- Data types
- Data format (and we computed new columns from existing data like bins)

We already looked at distributions of the range of values (graphically as well as in tabular format).

What is left to analyse the range of values of each column?

- A little bit more on the distribution of values
- Check for Null values
- Check for duplicates
- Plausibility checks

See the next slides in this section...

More on the distribution of values

The method `value_counts()` computes the set of unique values and the frequency counts for each of them (in descending order).

```
titanic["Sex"].value_counts()
```

```
male      577  
female    314
```

This allows to easily compute a histogram (instead of plotting with `sns.displot()`) and to check values.

Repetition: How can the same be achieved with `groupby()`?

Lacking values

First, we need to **detect** Null values.

They may already be marked as such and can then be normalized to `pd.NA` using `df.fillna(value=pd.NA)` (when the datatype already supports `pd.NA`).

To find columns containing Nulls:

```
titanic.info()
```

```
titanic.isna().any()
```

Sometimes, data contains some weird default values instead of Nulls, like `age = 999` or `age = -1`.

These have to be detected looking at the range of values as discussed.

Then, we have to **deal with** Nulls.

In many cases, Nulls are perfectly suitable to do analysis, as long as we are aware of them.

If not, we have the options to:

- Replace Null values with other values
- Completely remove lines containing unwanted Nulls

Duplicates

Best to avoid duplicates is to have a (non-artificial) key.

What about the titanic dataset?

A column can be checked for uniqueness with this attribute

```
titanic["Cabin"].is_unique
```

False

```
titanic["Name"].is_unique
```

True

Or, you may extract the list (a Series) of unique values:

```
titanic["Sex"].unique()
```

```
array(['male', 'female'], dtype=object)
```

Remove duplicates, using `df.drop_duplicates()`

In case of spelling errors, checking for uniqueness is not enough.

name	sex	birth
Tom Quax	M	01.04.1970
Quax Tom	M	11.04.1970
Quax, Tom	M	01.04.1970
tom quaxx	M	01.04.1970

What is the birthdate? Is this one person?

To detect such fuzzy duplicates, we could use **soundex** or the **edit-distance**. There is no vectorized command available in pandas. There are (outdated) libraries in python, but nothing in anaconda.

Still, swapping first and last names is an issue.

Eventually removing such duplicates is a manual process. After candidates are identified by an algorithm, you would need to check them based on further research.

Data profiling

Besides standard checks, you may want to manually investigate the range of values for **plausibility** with respect to the expected meaning of columns.

Obviously, there is no ultimate list of tasks. It rather depends on what you think is reasonable.

You will find further ideas and scripts searching for the buzzword "data profiling".

Exercise: Investigate
`titanic["Cabin"]`

Examples:

- *Manual investigation of sorted values*
- *Truncated fields*
- *Umlauts broken*
- *Invalid values (sex = Z, birthdate = 0333-02-31)*
- *Spelling errors*
- *Pattern matching for the format of phone numbers*
- *Check addresses for correctness*
- *Alignment (country = Stralsund)*
- *Investigate relationships between attributes to detect conflicts (city = Stralsund, post code = 89077)*
- *Compute aggregate reports and statistics to prove contents are reasonable and complete (distribution of sex over birth years)*

Summary / Checklist

It is absolutely mandatory to **investigate** each column in terms of:

- Data type
- Range of values
 - Data format
 - Plausibility
 - Distribution
 - Nulls
 - Duplicates
 - Completeness (e.g. with respect to time range)
- Plausibility with respect to other data

Document your adaptations made and the quantities involved (number of removed lines, of Nulls replaced). This allows to judge the quality of your data and the significance of a subsequent analysis.

Compute derived values as needed.

Tidy data

Reshaping data to ease analysis

country	year	cases	population
Afghanistan	1999	37745	19987071
Afghanistan	2000	4666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	216766	1280428583

variables

country	year	cases	population
Afghanistan	1999	37745	19987071
Afghanistan	2000	4666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	216766	1280428583

observations

country	year	cases	population
Afghanistan	99	745	987071
Afghanistan	00	666	595360
Brazil	99	737	006362
Brazil	00	488	504898
China	99	2258	2915272
China	00	6766	428583

values

Idea

This data is in **tidy** format

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

and we can easily analyse or plot it:

```
SELECT country, cases/population AS incidence FROM df WHERE year=2000;
```

```
(  
  df.loc[df["year"] == 2000]  
  .assign(incidence = df["cases"] / df["population"] * 1_000_000)  
  [["country", "incidence"]]  
)
```

✓ 0.9s

	country	incidence
1	Afghanistan	129.446633
3	Brazil	461.236337
5	China	166.948788

This is the same data in **non-tidy** format

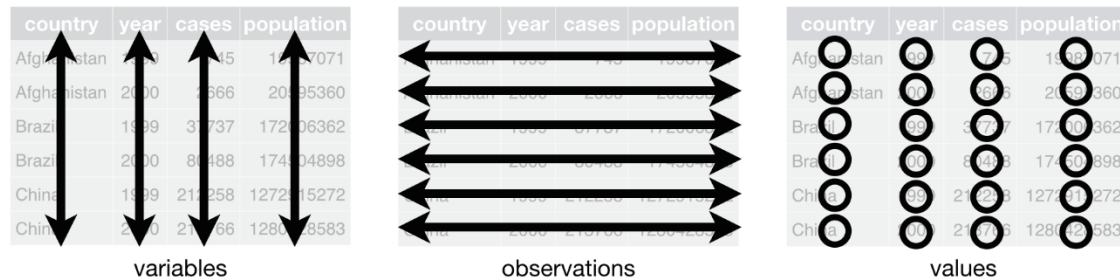
	country	year	variable	value
0	Afghanistan	1999	cases	745
1	Afghanistan	2000	cases	2666
2	Brazil	1999	cases	37737
3	Brazil	2000	cases	80488
4	China	1999	cases	212258
5	China	2000	cases	213766
6	Afghanistan	1999	population	19987071
7	Afghanistan	2000	population	20595360
8	Brazil	1999	population	172006362
9	Brazil	2000	population	174504898
10	China	1999	population	1272915272
11	China	2000	population	1280428583

cases:			population:		
year	1999	2000	year	1999	2000
country			country		
Afghanistan	745	2666	Afghanistan	19987071	20595360
Brazil	37737	80488	Brazil	172006362	174504898
China	212258	213766	China	1272915272	1280428583

Definition tidy data

A dataset is **tidy**, if:

1. Each type of observational unit forms a table.
2. Each observation forms a row.
3. Each variable must have its own column.
Sort variables: fixed variables (known in advance) first, followed by measured variables.
4. Each value must have its own cell.



Advantages of tidy data:

- It's easier to learn the commands and work with data structured in a consistent way.
- Leverage vectorized operations with good performance
- Standardize the interface to further processing

How do these rules compare to

- decomposition rules in relational databases?
- the dimensional model for data warehouses?

Synonyms for tidy data:

- **long-form data** (as opposed to wide-form data—or, a crosstab where both, columns and rows, contain levels of a dimension)
- **key figure model** or measure-based model (as opposed to account model with two column, key figure name and value)

References:

Book "R for Data Science", Chapter 12

<https://r4ds.had.co.nz/tidy-data.html#fig:tidy-structure>

Hadley Wickham, "Tidy Data", Journal of Statistical Software, August 2014, volume 59, Issue 10.

<http://dx.doi.org/10.18637/jss.v059.i10>

Messy data

We will now look at some cases of messy (= not tidy) data:

But before doing that,

we need two more commands to reshape our DataFrame.

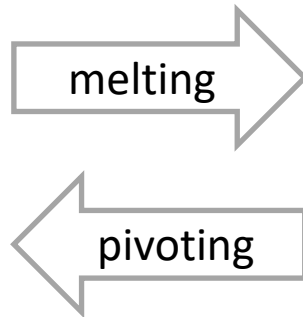


- 1) Column headers contain values, not variable names
- 2) Variable names in the data cells
- 3) Multiple values stored in one column (a violation of atomicity)
- 4) Multiple levels of observations in one table
- 5) One type of observation spread across multiple tables

... and combinations of these cases.

Conversion between long and wide format

df			
row	a	b	c
A	1	4	7
B	2	5	8
C	3	6	9



melt		
row	column	value
A	a	1
B	a	2
C	a	3
A	b	4
B	b	5
C	b	6
A	c	7
B	c	8
C	c	9

Melting is keeping a set of columns (here: "row") and the other columns are converted into a new variable (here: "column"). The values are preserved in variable "value".

Pivoting is the opposite conversion. Again, we keep a set of columns (here: "row"), we add a horizontal drill-down in columns (here from variable "column") and the data in the DataFrame comes from another variable (here: "value").

- `pivot()` creates a (Multi-)index we might want to reset.
- `pivot()` cannot handle duplicate values for one index/column pair, e.g. when we omit columns of DataFrame `melt`. In this case, an aggregation of the values is necessary and we need to use `pivot_table()` instead. There, we have to specify an aggregation function (sum, mean, min,...) and the values have to be numeric.

```
melt=df.melt(id_vars="row", var_name="column")
```

```
melt.pivot(index="row", columns="column", values="value").reset_index()
```

Which one (df or melt) is tidy?

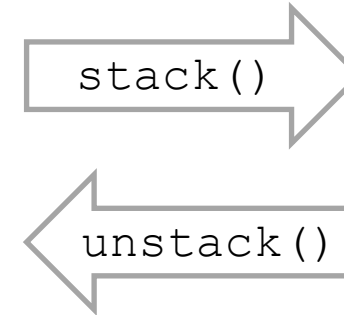
stack() / unstack()

When working with MultiIndexes, we can achieve the same with `stack()` (for melting) and `unstack()` (for pivoting).

Stacking a DataFrame means moving the *innermost column* index to become the *innermost row index*.

When working with tidy data, there should be no need for these functions.

	a	b	c
row			
A	1	4	7
B	2	5	8
C	3	6	9



	row	
A	a	1
	b	4
	c	7
B	a	2
	b	5
	c	8
C	a	3
	b	6
	c	9

Output looks different, since this is only a Series.

1) Column headers contain values, not labels

Raw data (pew.csv)

religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\$40-50k	\$50-75k	\$75-100k	\$100-150k	>150k	Don't know/refused
Agnostic	27	34	60	81	76	137	122	109	84	96
Atheist	12	27	37	52	35	70	73	59	74	76
Buddhist	27	21	30	34	33	58	62	39	53	54
Catholic	418	617	732	670	638	1116	949	792	633	1489
Don't know/refused	15	14	15	11	10	35	21	17	18	116
Evangelical Prot	575	869	1064	982	881	1486	949	723	414	1529
Hindu	1	9	7	9	11	34	47	48	54	37
Historically Black Prot	228	244	236	238	197	223	131	81	78	339
Jehovah's Witness	20	27	24	24	21	30	15	11	6	37
Jewish	19	19	25	25	30	95	69	87	151	162

Tidy

religion	income	count
Agnostic	<\$10k	27
Atheist	<\$10k	12
Buddhist	<\$10k	27
Catholic	<\$10k	418
Don't know/refused	<\$10k	15
...

```
pew.melt(id_vars="religion", var_name="income", value_name="count")
```

We have three variables: religion, income, number of people in the survey.

We have to melt everything except religion.
We rename the new columns (income and count).

2) Variable names in the data cells

Account model

This is just the opposite of case 1)

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Tidy

We have two key figures (cases and population).

	key	cases	population
country	year		
Afghanistan	1999	745	19987071
	2000	2666	20595360
Brazil	1999	37737	172006362
	2000	80488	174504898
China	1999	212258	1272915272
	2000	213766	1280428583

```
account.pivot(index=["country", "year"], columns="key", values="value")
```

Is it a variable name or an observation?

Sometimes it is hard to decide what the variables are. It may depend on the intended analysis.

Which of the following two designs is correct?

name	phone	number
Lola	private	0745 19987071
Lola	work	0266 20595360
Lucy	private	03737 172006362
Loreen	work	08048 17454898
Lucy	work	02158 72915272
Lucy	mobile	021366 80428583

name	mobile phone	private phone	work phone
Lola	NaN	0745 19987071	0266 20595360
Loreen	NaN	NaN	08048 17454898
Lucy	021366 80428583	03737 172006362	02158 72915272

This design allows combination of variables to calculate new variables.

Example: preferred phone = mobile phone when present, private phone otherwise.

*Example: incidence = cases / population * 10.000*

Comparing rows: This design allows comparing various phone numbers (e.g. one number for multiple people suggests fraud).

Extend values: We may be able to work with yet unknown additional phone types (fax, pager, car,...).


3a) Multiple values with fixed schema stored in one column

Consider cases per population given in one column
("rate" has the fixed schema of "cases/population"):

country	year	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

We can use string vector operations to remedy this:

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583



```
df[["cases", "population"]] = df["rate"].str.split('/', expand=True)
```

We may have to modify the dtype from object/string to numeric afterwards!

3b) Multiple indicator values in one column

There is a nice special case, when the column contains a list of indicators like

A = born in America

B = born in United Kingdom

C = likes cheese

D = likes spam

info	name
B C D	Graham Chapman
B D	John Cleese
A C	Terry Gilliam
B D	Eric Idle
B C	Terry Jones
B C D	Michael Palin

`df["info"].str.split('|', expand=True)`

0	1	2
B	C	D
B	D	None
A	C	None
B	D	None
B	C	None
B	C	D

We like to have a column for each indicator:

```
info = df["info"].str.get_dummies('|')  
df.merge(info, how="inner", left_index=True, right_index=True)
```

name	A	B	C	D
Graham Chapman	0	1	1	1
John Cleese	0	1	0	1
Terry Gilliam	1	0	1	0
Eric Idle	0	1	0	1
Terry Jones	0	1	1	0
Michael Palin	0	1	1	1

3c) A list of values in one column (explode vertically)

In this situation, the column containing a list models a 1:n relationship, e.g. student applied for a course.

First, we need to transform the string with commas into a real list:

	matrikel	name	course
0	123	freak	ERP, DB1, Prog
1	42	doe	DB2
2	888	lazy	
3	999	crazy	<NA>

```
students["course_list"] = students["course"].str.split(', ')
```

	matrikel	name	course	course_list
0	123	freak	ERP, DB1, Prog	[ERP, DB1, Prog]
1	42	doe	DB2	[DB2]
2	888	lazy		[]
3	999	crazy	<NA>	<NA>

Then, we can explode the list into a separate row for each entry in the list:

```
(
    students.explode("course_list")
    .drop(columns="course")
    .rename(columns={"course_list" : "course"})
)
```

```
.dropna()
.groupby(["matrikel", "name"]).agg(course=("course", ', '.join))
```

	matrikel	name	course
0	123	freak	ERP
0	123	freak	DB1
0	123	freak	Prog
1	42	doe	DB2
2	888	lazy	
3	999	crazy	<NA>

3d) explode a list horizontally

Alternatively, we can "explode" the list horizontally,
i.e. make a new column for each list element

	matrikel	name	course	course_list
0	123	freak	ERP, DB1, Prog	[ERP, DB1, Prog]
1	42	doe	DB2	[DB2]
2	888	lazy		[]
3	999	crazy	<NA>	<NA>

```
students.merge(  
    students["course_list"].apply(pd.Series).fillna(pd.NA)  
    , left_index=True, right_index=True  
)
```

	matrikel	name	course	course_list	0	1	2
0	123	freak	ERP, DB1, Prog	[ERP, DB1, Prog]	ERP	DB1	Prog
1	42	doe	DB2	[DB2]	DB2	<NA>	<NA>
2	888	lazy		[]	<NA>	<NA>	<NA>
3	999	crazy	<NA>	<NA>	<NA>	<NA>	<NA>

4) Multiple levels of observations in one table

In a given analysis, there may be multiple levels of observations. For example, in a trial of new allergy medication we might have three observational types: demographic data collected from each person (age, sex, race), medical data collected from each person on each day (number of sneezes, redness of eyes), and meteorological data collected on each day (temperature, pollen count).

Source: Hadley Wickham

In this case we would decompose the table into smaller bits as we know it from RDBMS theory.

For the sake of analysis though, we might later join back all observation types together into one big table. But take care to aggregate measures on their respective level of observation only.

How would the decomposition look like in the above mentioned example?

How to **decompose**:

- Do a projection to the necessary columns of the observation
- What is the key?
- `drop_duplicates()`

How to **de-normalize** (inverse operation):

- join using the key columns

4) the other way round: referential integrity

It might also be the case that a single table per observation is already given in the first place. This seems to be desirable, but:

For the sake of analysis, is a join into one big table (de-normalize) possible without problems?

Or: do we have referential integrity between the tables?

Example: The billboard contains a rating for a song "flowers in the rain", but this song does not exist in the songs-table. Then, we don't know the length of the song (pd.NA).

Standard database approach: Do an outer join and look for such dangling tuples (we can look for `pd.NA` or use `indicator=True`)

song	artist
1	franky
2	jimmy

song	week	rating
1	23	wow
2	22	oh no
2	24	well done son
42	22	super
1	22	outstanding

```
songs.merge(ratings, how="outer", on="song",  
            validate='one_to_many', indicator=True)
```



	song	artist	week	rating	_merge
0	1	franky	23	wow	both
1	1	franky	22	outstanding	both
2	2	jimmy	22	oh no	both
3	2	jimmy	24	well done son	both
4	42	NaN	22	super	right_only

5) One type of observation spread across multiple tables (vertically)

Pooh, we skip this one!

Basically the problem is that we might have a lot of files having the same format and we want to concatenate them into one big DataFrame.

Approach:

- Make a loop over all files
 - Read file into DataFrame a
 - Concatenate a to the final DataFrame s

```
import pandas as pd
import glob

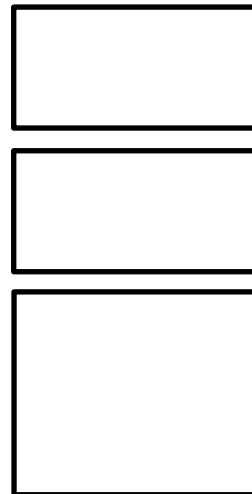
path = r'C:\DR0\DCL_rawdata_files' # use your path
all_files = glob.glob(path + "/*.csv")

li = []

for filename in all_files:
    df = pd.read_csv(filename, index_col=None, header=0)
    li.append(df)

frame = pd.concat(li, axis=0, ignore_index=True)
```

Source: stackoverflow

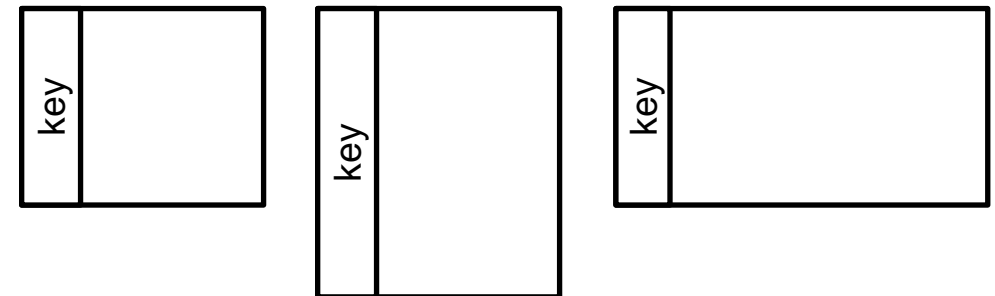


A more complicated situation is when the dataset structure changes over time. This requires to tidy each dataset individually first.

What if after we prepared our data and finished the analysis, new additional data comes into play. We like to incorporate this new data into the analysis. How can we do this delta-management?

The same, but horizontally:

Join the df, but make sure to use a common key.



Method chaining

Step by step

If we do a sequence of operations on a DataFrame, we create many intermediate variables.

```
url = "https://raw.githubusercontent.com/pandas-dev/pandas/master/doc/data/titanic.csv"
df = pd.read_csv(url)
df = df.convert_dtypes()
df.rename(columns={"Pclass" : "Klasse", "Age" : "Alter"}, inplace=True)
df = df.drop(columns="PassengerId")
df = df.set_index(keys=["Cabin", "Name"])
df
```

		Survived	Klasse	Sex	Alter	SibSp	Parch
Cabin	Name						
NaN	Braund, Mr. Owen Harris	0	3	male	22.0	1	0
C85	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	1	1	female	38.0	1	0
NaN	Heikkinen, Miss. Laina	1	3	female	26.0	0	0
C123	Futrelle, Mrs. Jacques Heath (Lily May Peel)	1	1	female	35.0	1	0

Make a chain or pipeline

To avoid this, we can chain all operations in one big step.

- We need to be able to write each operation as a function.
- We use brackets to be able to put each function into a separate line.

```
df = (
    pd.read_csv(url)
    .convert_dtypes()
    .rename(columns={"Pclass" : "Klasse", "Age" : "Alter"})
    .drop(columns="PassengerId")
    .set_index(keys=["Cabin", "Name"])
)
```


Method chaining

To be able to do chaining, we need to use a function in each step which will not modify the original DataFrame (do not use `inplace=True`). This is straightforward for all DataFrame methods we know, like `.rename()`, `.drop()`, `.assign()`, `.convert_dtypes()`, `.astype()`, `.dropna()`, `.sort_values()`,...

What about `.loc[]`?

Remember what we already explained in Chapter 1 ("`loc[]` with callable function"):

```
tips.loc[(tips["tip"] > 7)].loc[lambda df: ~(df["smoker"] == 'Yes')]
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
212	48.33	9.00	Male	No	Sat	Dinner	4

What about other functions?

Often, we will not have a fitting DataFrame method. In this case we can define a function like

```
def my_function(df, arg1)
    df = df.copy() # do not modify df!
    ... do something with df...
    return df
```

Then, we can use this function with `.pipe()`:

```
df = df.pipe(my_function, arg1)
```

A nice example can be found here

<https://tomaugspurger.github.io/method-chaining>

Method chaining

The chained command is not necessarily more elegant
(perhaps we could do better)

```
s["course"] = s["course"].str.split(',')  
s2 = s.explode("course")  
s2
```

✓ 0.1s

	matrikel	name	course
0	123	freak	ERP
0	123	freak	DB1
0	123	freak	Prog
1	42	doe	DB2
2	888	lazy	
3	999	crazy	<NA>

```
s3 = (  
    s["course"].str.split(',')  
    .explode()  
    .to_frame()  
    .merge(s, left_index=True, right_index=True)  
    .drop(columns="course_y")  
    .rename({"course_x": "course"}, axis='columns')  
)  
s3
```

✓ 0.5s

	course	matrikel	name
0	ERP	123	freak
0	DB1	123	freak
0	Prog	123	freak
1	DB2	42	doe
2		888	lazy
3	<NA>	999	crazy

Disclaimer: What we did not discuss

Analytic / Window functions

`expanding()` and `rolling()` can be used on a `DataFrame`. This resembles the windowing-clause for SQL analytic functions (see Chapter 3).

Pandas categorical data type

(well at least we a little bit with the `cut()` function)

Catch Exceptions, e.g. do a `df.loc[5]` but the value 5 is not present in the index. This will throw a `KeyError`.

`eval()` / `query()`

These are alternative ways to query data from a `DataFrame`. The syntax may sometimes easier to read (more SQL-like) and possibly they might need less space.