Lecture 1a
Visualization: History and Examples

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

René Descartes, La Géométrie, 1637

Introduced Cartesian coordinate system, similar ideas also Nicole Oresme (14th century)

LA

GEOMETRIE.

LIVRE PREMIER.

Des probleſmes qu'on peut conſtruire ſans y employer que des cercles & des lignes droites.

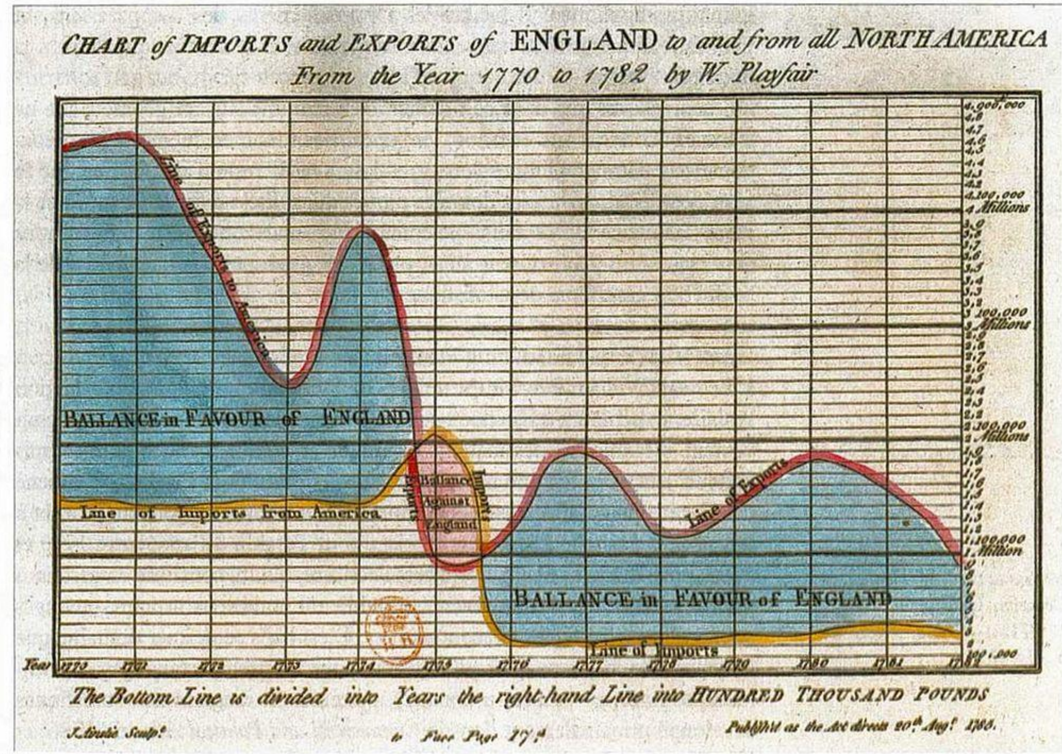
Ous les Probleſmes de Geometrie ſe peuuent facilement reduire a tels termes, qu'il n'eſt beſoin par aprés que de connoiſtre la longeur de quelques lignes droites, pour les conſtruire.

Et comme toute l'Arithmetique n'eſt compoſée, que de quatre ou cinq operations, qui ſont l'Addition, la Souſtraction, la Multiplication, la Diuiſion, & l'Extraction des racines, qu'on peut prendre pour vne eſpece de Diuiſion : Ainſi n'at'on autre choſe a faire en Geometrie touchant les lignes qu'on cherche, pour les preparer a eſtre connuës, que leur en adiouſter d'autres, ou en oſter, Oubien en ayant vne, que ie nommeray l'vnité pour la rapporter d'autant mieux aux nombres, & qui peut ordinairement eſtre priſe a diſcretion, puis en ayant encore deux autres, en trouuer vne quatrieſme, qui ſoit à l'vne de ces deux, comme l'autre eſt a l'vnité, ce qui eſt le meſme que la Multiplication; oubien en trouuer vne quatrieſme, qui ſoit a l'vne de ces deux, comme l'vnité

Commēt le calcul d'Arithmetique ſe rapporte aux operations de Geometrie.

William Playfair, Commercial and Political Atlas, 1786

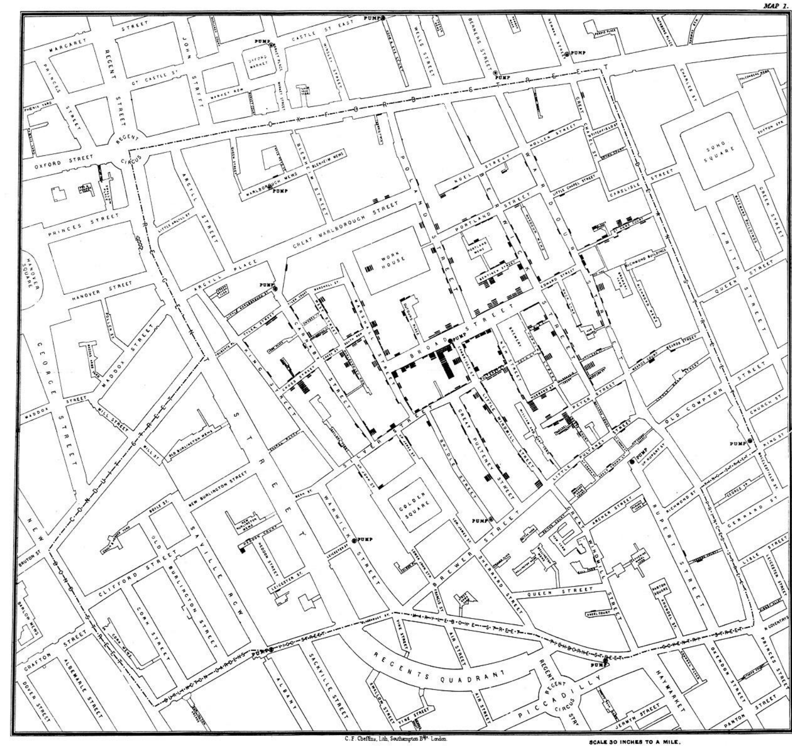
Playfair invented or popularized different types of graphs (line and area graphs, bar graphs, pie charts)



https://commons.wikimedia.org/wiki/File:1786_Playfair_-_Chart_of_import_and_exports_of_England_to_and_from_all_North_America_from_the_year_1770_to_1782.jpg

John Snow, On the Mode of Communication of Cholera, 1854

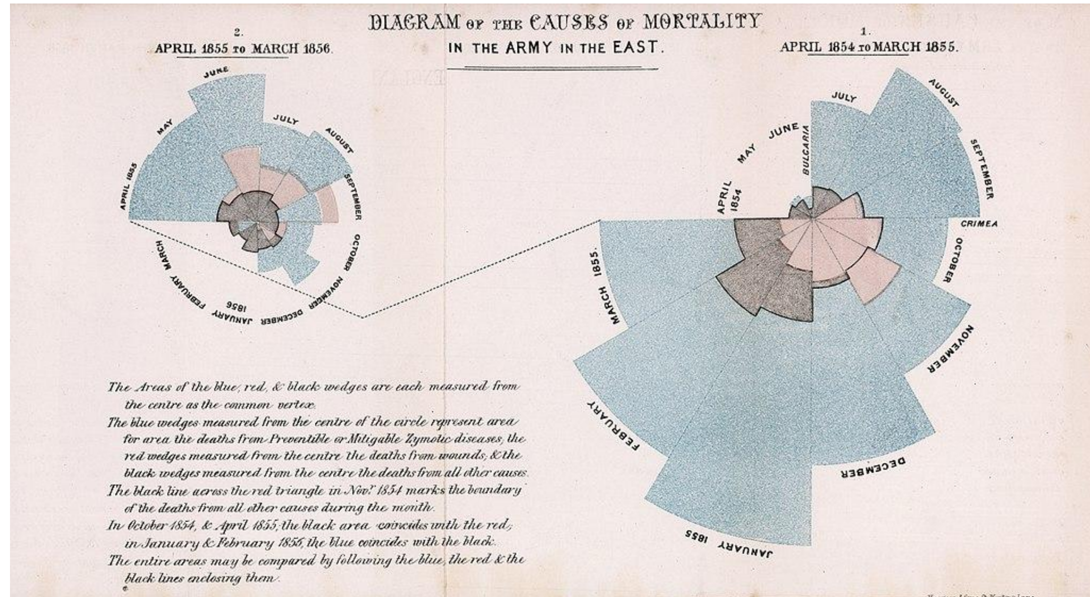
Snow tracked cholera cases in 1854 outbreak and displayed them clustered around a single London well



https://commons.wikimedia.org/wiki/File:Snow-cholera-map-1.jpg

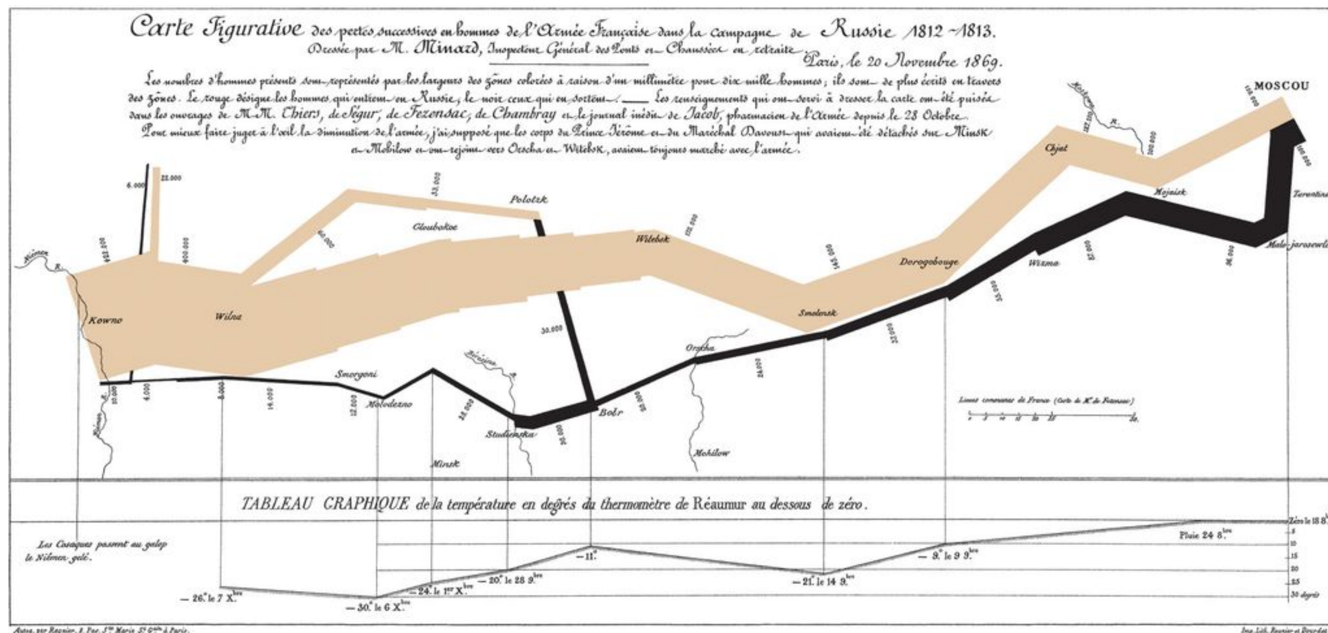
Florence Nightingale, Diagram of the causes of mortality in the army in the East, 1858

Nightingale led a field hospital during Crimean War, reported to the government on poor conditions causing deaths from typhus, cholera etc



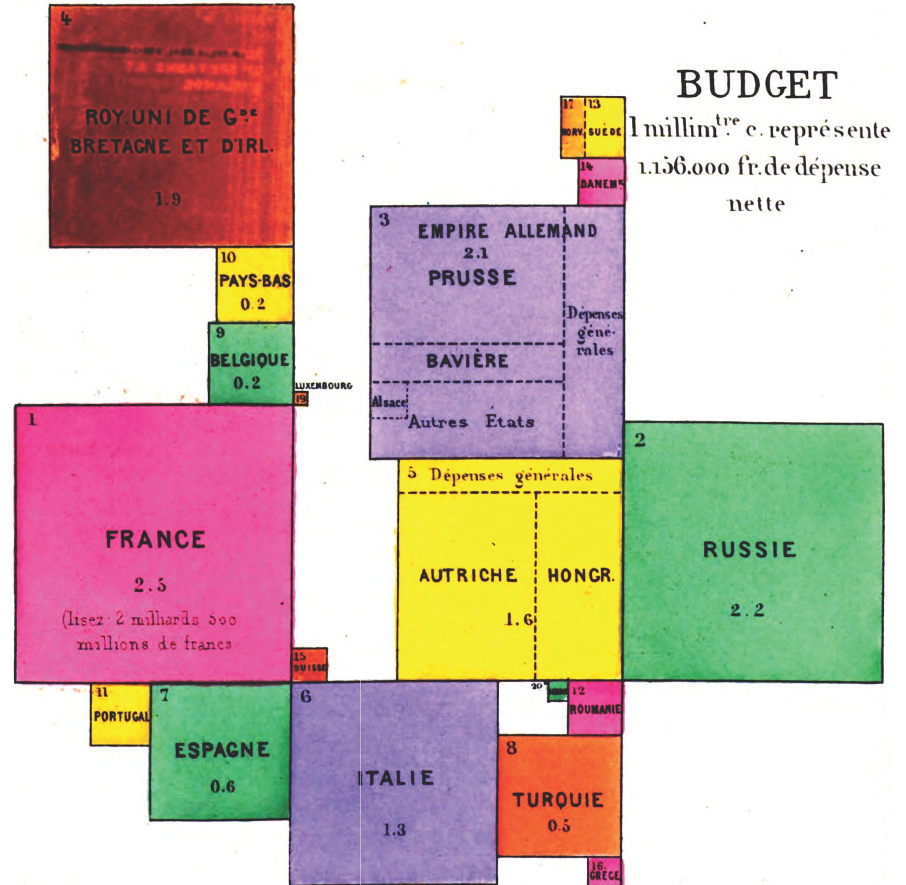
https://commons.wikimedia.org/wiki/File:Nightingale-mortality.jpg

Charles Minard, Figurative Map of the successive losses in men of the French Army in the Russian campaign 1812–1813, 1869



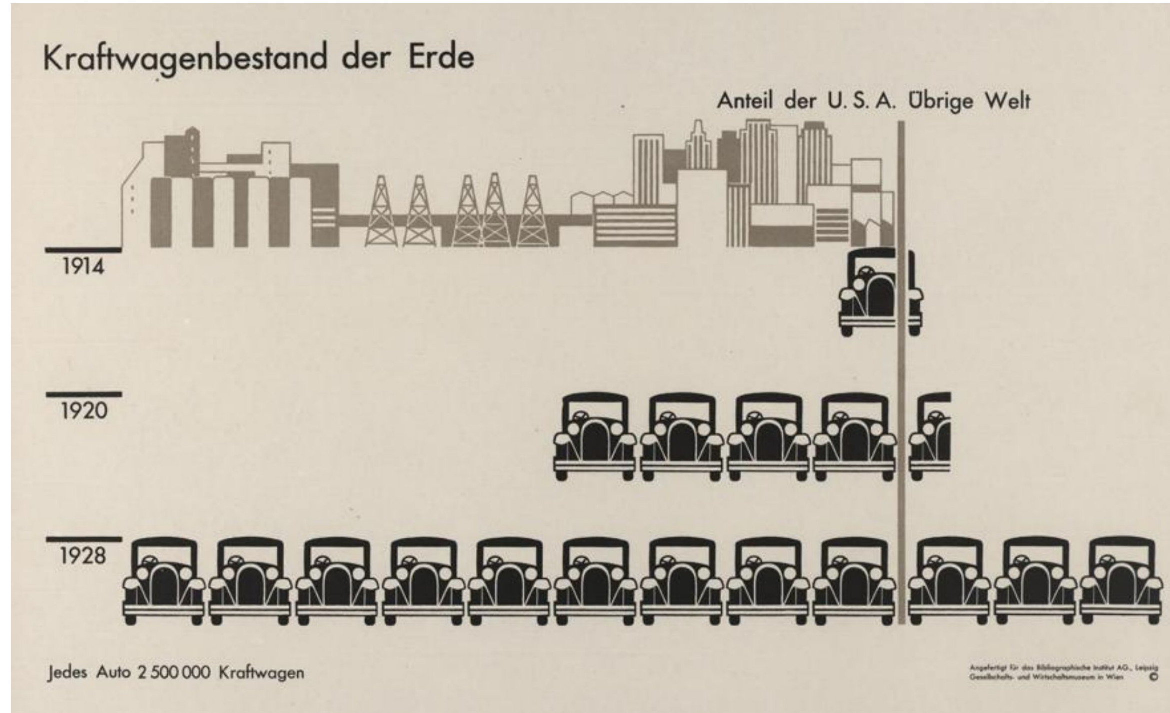
https://commons.wikimedia.org/wiki/File:Minard.png

Pierre Émile Levasseur, 1876



https://commons.wikimedia.org/wiki/File:Levasseur_cartogram.png

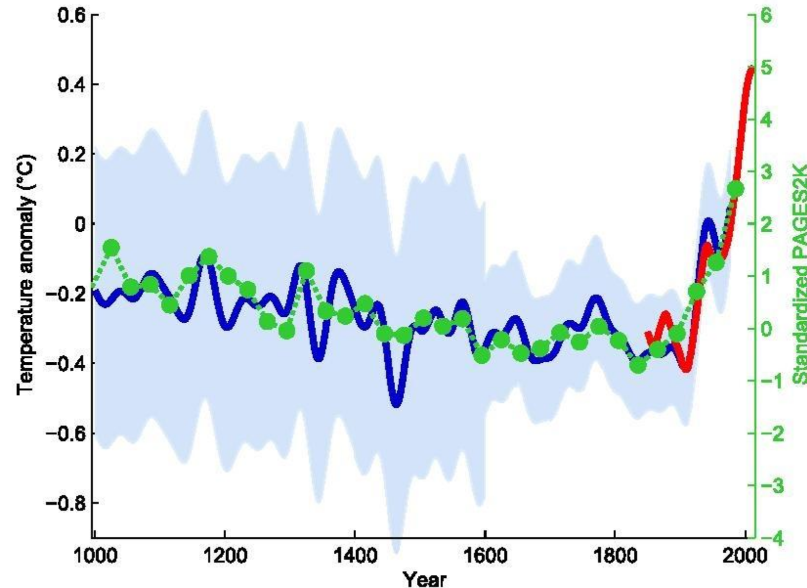
Otto Neurath, Gesellschaft und Wirtschaft, 1930



https://www.digital.wienbibliothek.at/wbrobv/content/pageview/2296000

Mann, Bradley & Hughes: Northern hemisphere temperatures during the past millennium[...] 1999

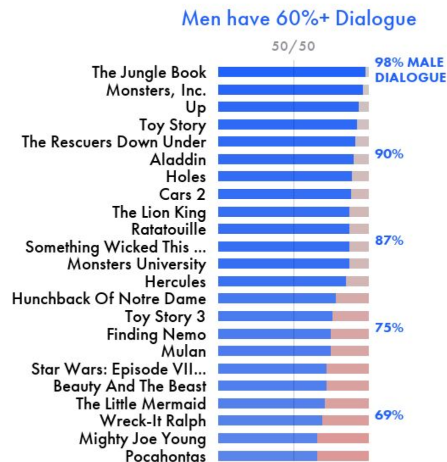
Hockey stick graph, created publicity and controversy, featured also in 2001 Intergovernmental Panel on Climate Change Third Assessment Report

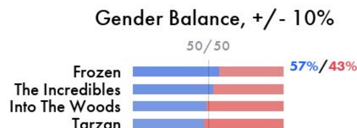


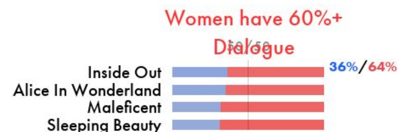
https://en.wikipedia.org/wiki/File:T_comp_61-90.pdf CC BY-SA Klaus Bittermann; original article http://www.geo.umass.edu/faculty/bradley/mann1999.pdf

Anderson & Daniels: Film Dialog from 2,000 Screenplays, Broken Down by Gender and Age, 2016

Screenplay Dialogue, Broken-down by Gender | 2,000 Screenplays: Dialogue Broken-down by Gender | Only High-Grossing Films: Ranked in the Top 2,500 by US Box Office*



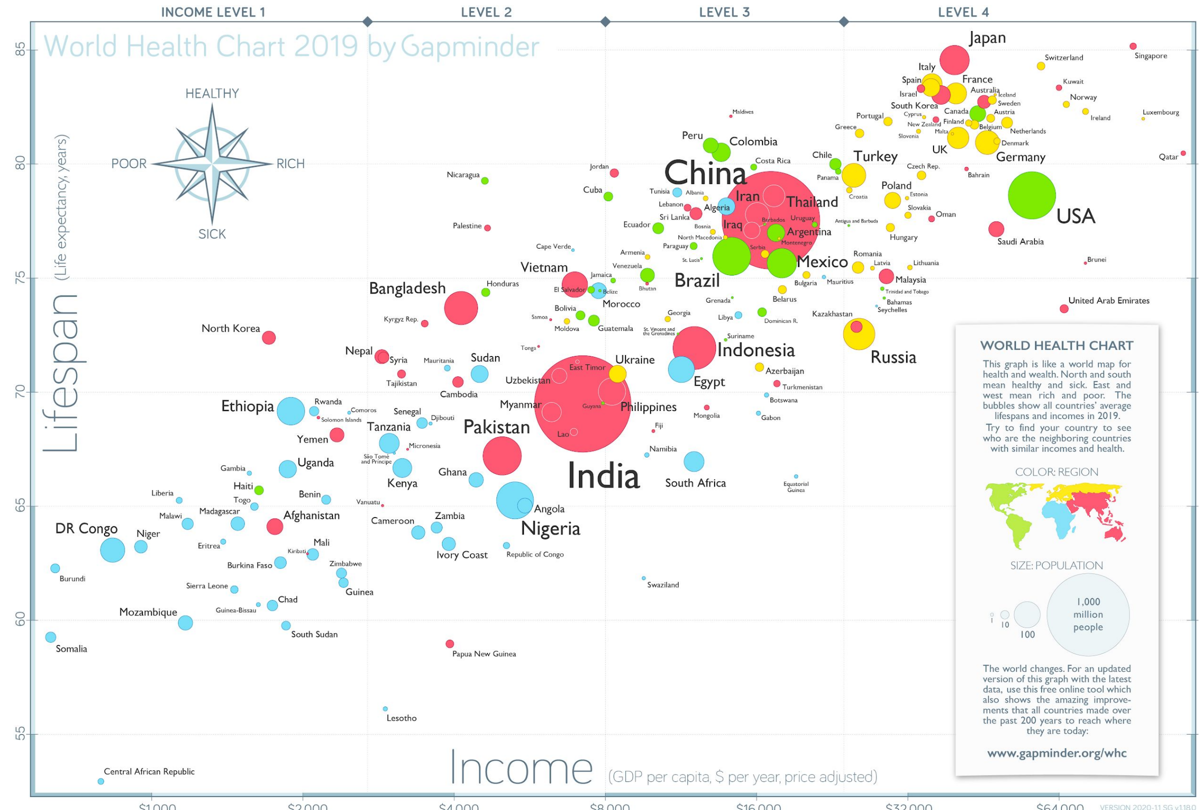




https://pudding.cool/2017/03/film-dialogue/

Gapminder foundation, World Health Chart 2019, 2020

Original version by Hans Rosling in his 2006 TED talk



https://www.gapminder.org/fw/world-health-chart/whc2019/ CC-BY, free license

Two other recent examples

Comparing size of software

https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Deadly pandemics in history

https://www.visualcapitalist.com/history-of-pandemics-deadliest/

Additional resources

- Tableaux.com: Data is beautiful: 10 of the best data visualization examples from history to today https://www.tableau.com/learn/articles/best-beautiful-data-visualization-examples
- Michael Friendly: History of Data Visualization https://friendly.github.io/HistDataVis/
- Hans Rosling The best stats you've ever seen (TED Talk) https://www.ted.com/talks/hans_rosling_the_best_stats_you_ve_ever_seen

1 Lecture 1b: Introduction to Jupyter Notebooks, Google Colab and Matplotlib

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Jupyter Notebook

- Jupyter Notebook is a web-based software for interactive work in Python.
- It is frequently used for data processing and visualization.
- A document called **notebook** consists of **cells**.
- Each cell contains either text or Python code.
- The text may include formatting in Markdown language.
- A cell with Python code can be executed and the results display below, including images.
- This presentation is a Jupyter notebook.
- Notebook files have extension `.ipynb`.

1.2 Google Colab and alternatives

- You can work with Jupyter notebooks on many platforms (both online and installed on your computer).
- In this course, we will primarily use Google Colab.
- Colab stores your notebooks on Google drive, executes them on Google servers, you only need web browser on your computer.
- It integrates well with Google Classroom, which we use as well.
- You are free to use other options as long as the submitted notebooks can be executed in Colab.
- One popular option is VS Code. It requires Jupyter software to be installed as well, see documentation.

1.3 How to use Notebooks

- Notebooks have an intuitive interface with menus, toolbars, context menus (right-click) etc.
- It is useful to learn some keyboard shortcuts.

When you are not editting a cell:

- use Up and Down arrows to move between cells
- use **Enter** (or double-click) to start editing a cell
- use **Esc** to stop editing a cell
- use **Ctrl-Enter** to run a code in a cell, **Shift-Enter** to run the code and move to the next cell

1.3.1 An example of a code cell

- A cell can include imports, function definitions, commands.
- Variables will be visible in other cells.
- The results of print are shown below the cell when executed.
- The last expression is printed below the cell when executed.

[1]:
```
# create variable x with list of numbers 0,1,...,19
x = list(range(20))
# variable y will contain squares of values in x
y = [xval * xval for xval in x]
# print x and y
print(x)
print(y)
# the last value (the first 5 values in y) is also printed automatically
y[0:5]
```

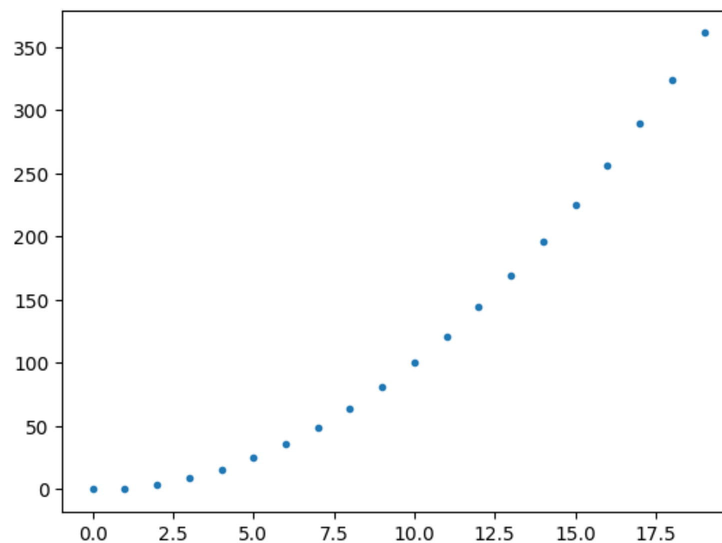
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
324, 361]
```

[1]:
```
[0, 1, 4, 9, 16]
```

1.3.2 Example of a cell with a plot

The plot uses variables x and y from the previous cell to plot the quadratic function.

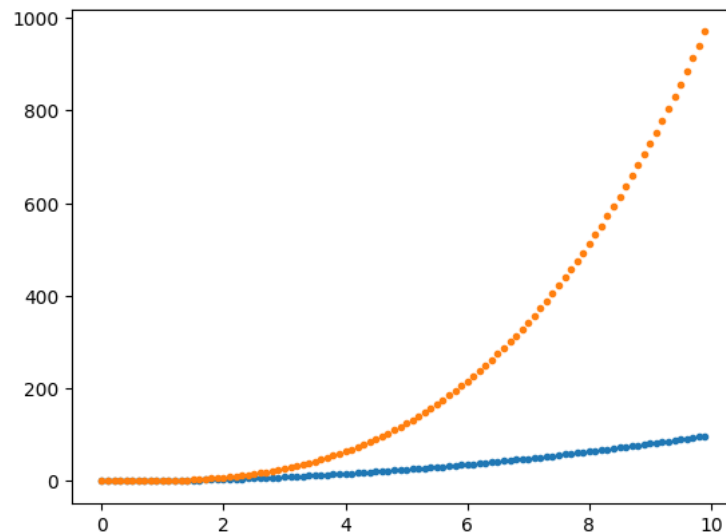
[2]:
```
import matplotlib.pyplot as plt
# create figure with a single plot (axes)
figure, axes = plt.subplots()
# plot x vs y
axes.plot(x, y, '.')
# command pass to suppress unwanted output from plot
pass
```



1.4 Matplotlib library

- Matplotlib is a Python library for creating plots.
- In the code above, axes is the Matplotlib name for a single plot.
- Let us now plot two functions: quadratic and cubic in the same plot.

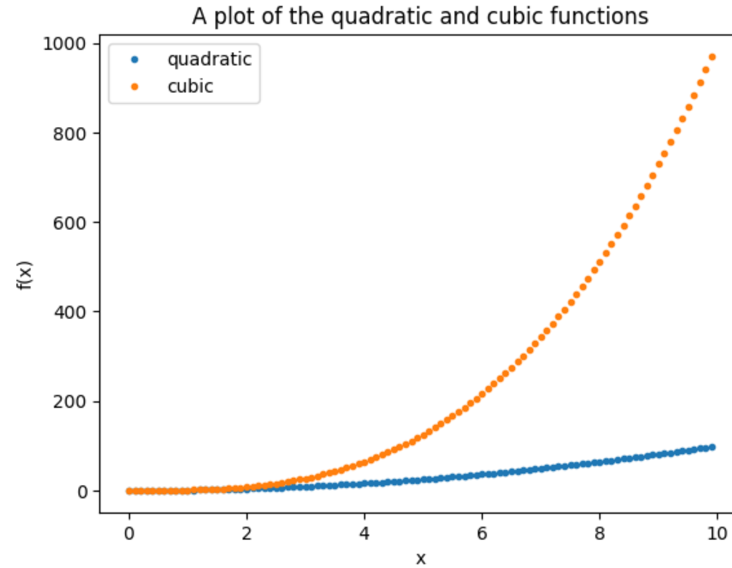
[3]:
```
# x_dense is values from 0 to 10 with step 0.1
x_dense = [val / 10 for val in range(0, 100)]
# values in y2_dense are values from x_dense squared
y2_dense = [xval ** 2 for xval in x_dense]
# values in y3_dense are values from x_dense to the power of 3
y3_dense = [xval ** 3 for xval in x_dense]
# plot the quadratic and cubic function in a single plot
figure, axes = plt.subplots()
axes.plot(x_dense, y2_dense, '.')
axes.plot(x_dense, y3_dense, '.')
pass
```



1.4.1 Setting labels and titles in Matplotlib

[4]:
```
# the same plot as before, but name the two sets of points by label
figure, axes = plt.subplots()
axes.plot(x_dense, y2_dense, '.', label="quadratic")
axes.plot(x_dense, y3_dense, '.', label="cubic")

# add titles for axes (usually use a more descriptive titles)
axes.set_xlabel("x")
axes.set_ylabel("f(x)")
# legend (which plot is which function), uses the labels set in plt.plot
```

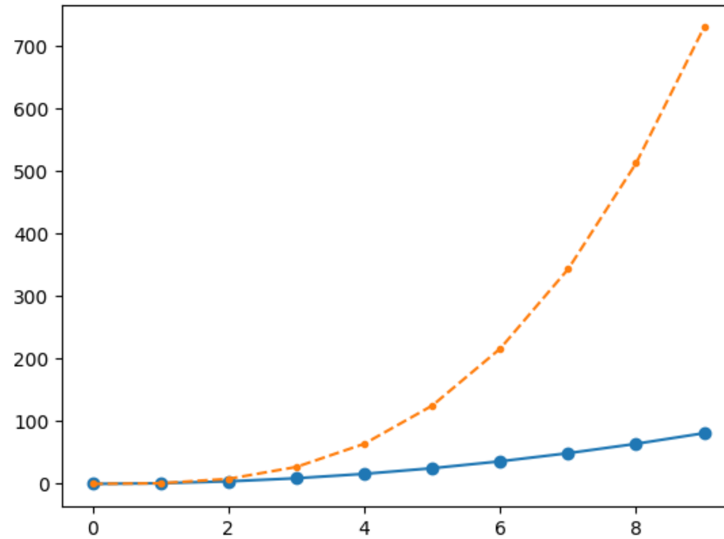
```
axes.legend()
# a title of the whole plot
axes.set_title("A plot of the quadratic and cubic functions")
pass
```



1.4.2 Setting lines, markers and colors

- In the `axes.plot` command, `'.'` represents formatting, in this case a small dot.
- The formatting string has three optional parts: marker, line, color.
- Examples: `'or'` red circle, `'-g'` green solid line, `'--'` dashed line.
- See more in documentation.

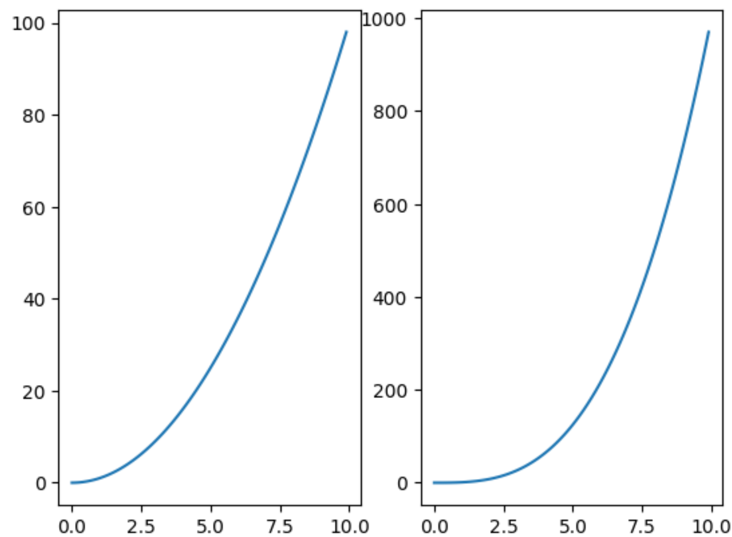
[5]:
```
x_sparse = range(0, 10)
y2_sparse = [xval ** 2 for xval in x_sparse]
y3_sparse = [xval ** 3 for xval in x_sparse]
figure, axes = plt.subplots()
axes.plot(x_sparse, y2_sparse, 'o-')
axes.plot(x_sparse, y3_sparse, '.--')
pass
```



1.4.3 Multiple plots per image

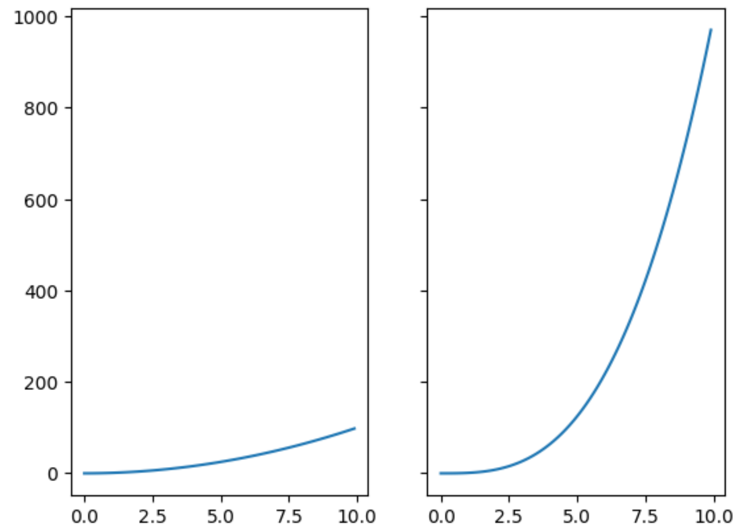
- Function plt.subplots can take as arguments the number of rows and the number of columns and creates multiple subplots per image.
- Why is this figure not an ideal visual comparison of the quadratic and cubic function?

[6]:
```
figure, axes = plt.subplots(1, 2)
axes[0].plot(x_dense, y2_dense)
axes[1].plot(x_dense, y3_dense)
pass
```



- Each plot has a different y-axis, which is not good, because we do not immediately see that the cubic function grows much faster than the quadratic.
- We will fix this in the next plot using `sharey=True` setting (`sharex=True` also exists but here it is not needed).

[7]:
```
# fixing the problem with different y-axis
figure, axes = plt.subplots(1, 2, sharey=True)
axes[0].plot(x_dense, y2_dense)
axes[1].plot(x_dense, y3_dense)
pass
```



1.5 Dangers of notebooks

- Frequent use pattern: the users have several cells finished and executed, they work on the last cell in the notebook, and run it repeatedly until it works correctly. This avoids repeated execution of the top cells, which may be slow.
- Notebooks do not force you to run cells in order from top to bottom. This generates problems if you skip some cells or execute them repeatedly.

Good practice suggestions:

- Do not modify variables introduced in other cells.
- Refactor bigger or repeated parts of code to functions. This also hides local variables from the rest of the notebook and thus prevents clashes.
- Ideally move functions to separate modules but this is harder in Colab.
- Avoid running cells out of order and occasionally restart the kernel (to remove variables) and run all cells (using a menu function).

Below we see an example not obeying the first recommendation; the value printed will depened on how many times we execute the second cell. This can lead to hard-to-find errors in a more complex case.

[8]:
```
value = 0
```

[9]:
```
value += 1
print(value)
```

```
1
```

1.6 Additional resources

- Python Data Science Handbook by Jake VanderPlas, O'Reilly 2016
- Jupyter Notebook documentation
- Google Colab website and introductory video
- Matplotlib tutorials
- I don't like notebooks by Joel Grus (entertaining video explaining some of the pitfalls of notebooks)

1 Lecture 1b: Quick Introduction to New Libraries

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

- We will now briefly discuss several libraries which will be used in the next tutorial.
- We will cover details of these libraries in the coming weeks, this is just a glipse of things to come.

[1]:
```
# importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
```

1.1 Libraries NumPy and Pandas

- Pandas is a Python library for working with tabular data.
- NumPy is a library of efficient multi-dimensional arrays used for numerical computations.

1.1.1 NumPy array and arithmetical operations with arrays

- Function `np.arange` below creates a list of numbers in interval $[1, 3)$ with step 0.5 (generalization of Python `range`).
- It is stored as an object of array class from the Numpy library.

[2]:
```
x = np.arange(1, 3, 0.5)
print('x:', x)
```

```
x: [1.  1.5 2.  2.5]
```

- We can do various arithmetic operations on whole NumPy arrays or apply predefined functions such as `np.exp`.
- Such operations are typically done element-by-element.

[3]:
```
print('x:', x)
print('x+1:', x+1)
print('x*x:', x*x)
print('np.exp(x):', np.exp(x))
```

```
x: [1.  1.5 2.  2.5]
x+1: [2.  2.5 3.  3.5]
x*x: [1.   2.25 4.   6.25]
np.exp(x): [ 2.71828183  4.48168907  7.3890561  12.18249396]
```

1.1.2 Creating Pandas DataFrame

- Below we create an object of Pandas DataFrame class.

- We will cover most Pandas functions used below next week, for now the details are not important.

[4]:
```
def convert_table(x, function_dict):
    """ x is a list (or Numpy array) of values of x,
    function_dict is a dictionary containing function names as keys
    and lists of function values as values. The result will be a Pandas
    DataFrame (table) with each row containing triple x, function, value.
    Zeroes and negative values are masked as missing
    to avoid problems with logarithmic y axis."""

    # check that all functions have the same number of values as x
    for f in function_dict:
        assert(len(function_dict[f])==len(x))

    # create a wide table with each function as one columns
    functions_wide = pd.DataFrame(function_dict, index=x)
    # reformat to long format
    #   where each row is a triple x, function name, function value
    functions = (functions_wide.reset_index()
                 .melt(id_vars='index')
                 .rename(columns={'variable':'function', 'index':'x'}))
    # mask values <= 0 as missing values
    val = functions['value']
    functions['value'] = val.mask(val <= 0, np.nan)
    return functions
```

[5]:
```
functions = convert_table(x, {'quadratic': x * x,'cubic': x * x * x})
```

Let us look at the resulting table `functions`:

- It has three columns named `'x'`, `'function'` and `'value'`.
- Each row is a triple, containing a function name and the values of x and $f(x)$.
- E.g. one of the rows for the cubic function has $x = 2$ and $f(x) = 2^3 = 8$.

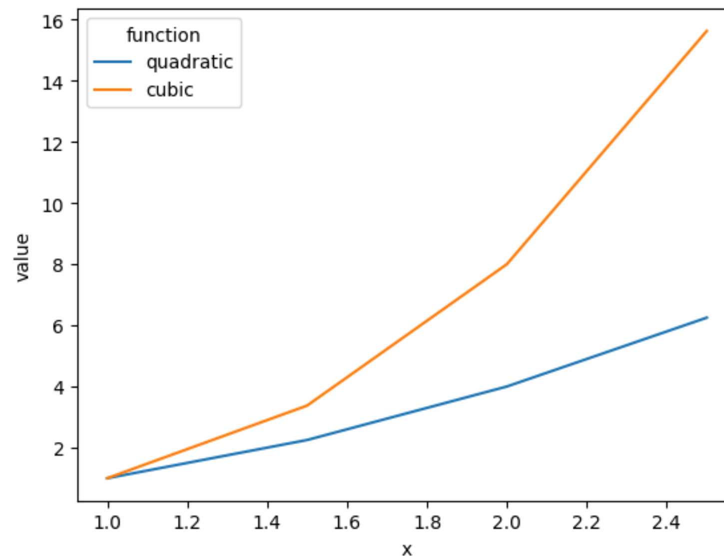
[6]:
```
display(functions)
```

	x	function	value
0	1.0	quadratic	1.000
1	1.5	quadratic	2.250
2	2.0	quadratic	4.000
3	2.5	quadratic	6.250
4	1.0	cubic	1.000
5	1.5	cubic	3.375
6	2.0	cubic	8.000
7	2.5	cubic	15.625

1.2 Displaying Pandas DataFrame using Seaborn and Plotly libraries

- Seaborn library is an extension of Matplotlib.
- It is very convenient for displaying tables.
- In the `sns.lineplot` we first give the table and then specify, which columns should be used as x coordinate, y coordinate and color (`hue`).
- One line will be automaticlaly drawn for each distinct value in the `hue` column and a legend will be added.

[7]:
```
figure, axes = plt.subplots()
sns.lineplot(functions, x='x', y='value', hue='function', ax=axes)
pass
```



- Another popular library is Plotly.
- It provides some additional plot types and all plots are interactive.
- For example, we can also zoom into parts of the plot by selecting a rectangle.
- A menu with additional options appears in the top right corner of the plot.
- A line plot is created similarly as in Seaborn (option `color` is used instead of `hue`).

[8]:
```
figure = px.line(functions, x="x", y="value", color='function')
figure.show()
```

1.3 Interactive plots in Plotly Dash

- Dash library by Plotly allows adding control elements (selectors, sliders, buttons, …).
- It is not preinstalled in Colab, so the next line will install it.

[ ]:
```
! pip install dash
```

- The code below creates an interactive plot in which the user can choose which functions from the list to display.
- The code has many comments so read through it carefully.

[10]:
```
from dash import Dash, dcc, html, Input, Output

# create a list of all functions
all_functions = list(functions['function'].unique())

# create a new dash application app
app = Dash(__name__)

# Create layout of items in application
#   one html <div> item containing text as small headwers (H4),
#   items for individual inputs and a graph at the bottom
# Currently we have two inputs:
#   an input field for entering title text
#   checkboxes for selecting functions
# These elements have identifiers which will be used later in the code
app.layout = html.Div([
    html.H4("Plot title: "),
    # input field for entering title text:
    dcc.Input(
        id='graph-title',
        type='text',
        value='My plot'
    ),
    html.H4("Select functions: "),
    # checkboxes for selecting functions:
    dcc.Checklist(
        id='selected-functions',
        options=all_functions,
        value=['quadratic'],
        inline=True   # place checkboxes horizontally
    ),
    # graph itself
    dcc.Graph(id='graph-content')
])

# @app.callback is a function decorator applied to function update_figure below.
# It defines that this function will be called to update the graph when the␣
↪user makes a change.
#    Input will be the value entered to the input field with id graph-title and
#       the list of functions selected in dcc.Checklist object with id␣
↪'selected-functions'.
#    Output will be the graph created by the function update_figure below,␣
↪which will be used
```

```
#      to update dcc.Graph object with id 'graph-content'
@app.callback(
    Output('graph-content', 'figure'),
    [Input('graph-title', 'value'),
     Input('selected-functions', 'value')
    ]
)
def update_figure(title, selected_functions):
    """ Function for ploting functions listed in list selected_functions
    with plot title given in title"""

    # select a subset of functions table with just those functions in input list
    functions_subset = functions.query('function in @selected_functions')

    # create a plotly line plot using the smaller table in functions_subset
    figure = px.line(
        functions_subset, x="x", y="value", color="function",
        width=800, height=500
    )

    # add title to the plot
    figure.update_layout(title_text=title)

    return figure

# run the whole application
app.run_server(mode='inline')
```

```
<IPython.lib.display.IFrame at 0x7f8f001c8e80>
```

1 Lecture 2: Data processing in Pandas library

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Tabular data

- We will often work with data in the form of tables.
- Columns represent different features / variables (príznaky, atribúty, veličiny, premenné).
- Rows represent different items / data points / observations (countries, people, dates of measurement, …).
- A small example:

Country	Region	Population	Area (km2)	Landlocked
Slovakia	Europe	5450421	49035	yes
Czech Republic	Europe	10649800	78866	yes
Hungary	Europe	9772756	93030	yes
Poland	Europe	38386000	312696	no

1.2 Pandas library

- Pandas is a Python library for data manipulation and analysis.
- It is fast and has many functions for data import and export in various formats.
- Documentation, overview, tutorial

Basic data structures

- `Series`: 1D table, all elements of the same type.
- `DataFrame`: 2D table, elements within each column of the same type.

NumPy library

- NumPy is a library of efficient multi-dimensional arrays used for numerical computations.
- We will mostly use Pandas, but some NumPy functions will be useful.
- Tutorial, reference

[93]:
```
import numpy as np
import pandas as pd
from IPython.display import Markdown
import matplotlib.pyplot as plt
```

1.2.1 Creating Series and DataFrames

- Bellow we show two manual ways of creating a DataFrame containing the small table of countries above.
- The first way gets a Series for each column, the second way gets a dictionary (or a tuple) for each row.
- We will usually read tabular data from files, see an example in the second half this lecture.

```
[94]: countries = pd.Series(['Slovakia', 'Czech Republic', 'Hungary', 'Poland'])
regions = pd.Series(['Europe', 'Europe', 'Europe', 'Europe'])
populations = pd.Series([5450421, 10649800, 9772756, 38386000])
areas = pd.Series([49035, 78866, 93030, 312696])
landlocked = pd.Series([True, True, True, False])
table = pd.DataFrame({'country':countries,
                      'region':regions,
                      'population':populations,
                      'area':areas,
                      'landlocked':landlocked})
# display: special notebook command for pretty-printing data
display(table)
```

	country	region	population	area	landlocked
0	Slovakia	Europe	5450421	49035	True
1	Czech Republic	Europe	10649800	78866	True
2	Hungary	Europe	9772756	93030	True
3	Poland	Europe	38386000	312696	False

```
[95]: list_of_dicts = [
    {'country':'Slovakia', 'region':'Europe', 'population':5450421, 'area':
 ↪49035, 'landlocked':True},
    {'country':'Czech Republic', 'region':'Europe', 'population':10649800,␣
 ↪'area':78866, 'landlocked':True},
    {'country':'Hungary', 'region':'Europe', 'population':9772756, 'area':
 ↪93030, 'landlocked':True},
    {'country':'Poland', 'region':'Europe', 'population':38386000, 'area':
 ↪312696, 'landlocked':False},
]
table_from_list = pd.DataFrame.from_records(list_of_dicts)
display(table_from_list)
```

	country	region	population	area	landlocked
0	Slovakia	Europe	5450421	49035	True
1	Czech Republic	Europe	10649800	78866	True
2	Hungary	Europe	9772756	93030	True
3	Poland	Europe	38386000	312696	False

1.2.2 Accessing elements of Series and DataFrame by position

- Attribute `ndim` is the number of dimensions. E.g. `areas.ndim` is 1, `table.ndim` is 2.
- Attribute `shape` is a tuple holding the size in each dimension. E.g. `areas.shape` is `(4,)`, `table.shape` is `(4,5)`.
- Rows and columns are numbered 0, 1, …
- To access a particular column / row, use `some_series.iloc[row]` or `some_table.iloc[row, column]`.
- Rows and columns in `iloc` can be
  - a single number e.g. `0`,

- a slice (range) e.g. `0:2` or `:` for everything,
  - a list of positions e.g. `[0, 2, 3]`
  - a list of boolean values `[True, False, True, True]`.
- The result is a single element or a Series / DataFrame of a smaller size.

```
[96]: display(Markdown("**table:**"), table)
      display(Markdown("**table.iloc[1, 2]:**"), table.iloc[1, 2])
      display(Markdown("**table.iloc[[0, 2, 3], 0:2]**"), table.iloc[[0, 2, 3],  0:2])
      display(Markdown("**table.iloc[[True, False, True, True], :]**"),
              table.iloc[[True, False, True, True], :])
```

table:

```
          country  region  population    area  landlocked
0        Slovakia  Europe     5450421   49035        True
1  Czech Republic  Europe    10649800   78866        True
2         Hungary  Europe     9772756   93030        True
3          Poland  Europe    38386000  312696       False
```

table.iloc[1, 2]:

```
10649800
```

table.iloc[[0, 2, 3], 0:2]

```
    country  region
0  Slovakia  Europe
2   Hungary  Europe
3    Poland  Europe
```

table.iloc[[True, False, True, True], :]

```
    country  region  population    area  landlocked
0  Slovakia  Europe     5450421   49035        True
2   Hungary  Europe     9772756   93030        True
3    Poland  Europe    38386000  312696       False
```

1.2.3 Views vs. copies

- Accessing parts of tables by `iloc` may return a partial copy or simply a "view".
- If we later modify this result, it is not clear if the original table is modified.
- Direct assignment of new values to a part of the table works: `some_table.iloc[row, column] = new_value` modifies `some_table`.
- To copy a table, use `other_table = some_table.copy(deep=True)`.

```
[97]: table2 = table.copy(deep=True)
      # create a copy of the original table

      table2.iloc[0,0] = 'Slovensko'
      display(table2)
      # table2 now has Slovensko instead of Slovakia
```

```
countries2 = table2.iloc[: , 0]
# countries2 is now a view or a copy of one column of table2
countries2.iloc[2] = 'Maďarsko'
display(table2)
# table2 now can have Hungary or Maďarsko
# we get a warning
```

```
          country  region  population    area  landlocked
0       Slovensko  Europe     5450421   49035        True
1  Czech Republic  Europe    10649800   78866        True
2         Hungary  Europe     9772756   93030        True
3          Poland  Europe    38386000  312696       False
```

```
/tmp/ipykernel_1087193/2667016646.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  countries2.iloc[2] = 'Maďarsko'
```

```
          country  region  population    area  landlocked
0       Slovensko  Europe     5450421   49035        True
1  Czech Republic  Europe    10649800   78866        True
2        Maďarsko  Europe     9772756   93030        True
3          Poland  Europe    38386000  312696       False
```

1.2.4 Inplace operations

- Many operations return a new table.
- If you do not need the original table, you can specify option `inplace=True`.
- The example below sorts a table by a specified column, returning a new table or replacing the old one.

[98]:
```
# copy original table to table2
table2 = table.copy(deep=True)

# table3 is a copy of table2 sorted by population size
table3 = table2.sort_values(by="population")

# display both table2 and table3
display(Markdown("**Original table2:**"), table2)
display(Markdown("**Sorted table3:**"), table3)

# now change table2 to be sorted by name of the country
table2.sort_values(by="country", inplace=True)
display(Markdown("**Sorted table2:**"), table2)
```

Original table2:

```
          country  region  population    area  landlocked
```

```
0        Slovakia  Europe    5450421   49035       True
1  Czech Republic  Europe   10649800   78866       True
2         Hungary  Europe    9772756   93030       True
3          Poland  Europe   38386000  312696      False
```

Sorted table3:

```
          country  region  population    area  landlocked
0        Slovakia  Europe     5450421   49035        True
2         Hungary  Europe     9772756   93030        True
1  Czech Republic  Europe    10649800   78866        True
3          Poland  Europe    38386000  312696       False
```

Sorted table2:

```
          country  region  population    area  landlocked
1  Czech Republic  Europe    10649800   78866        True
2         Hungary  Europe     9772756   93030        True
3          Poland  Europe    38386000  312696       False
0        Slovakia  Europe     5450421   49035        True
```

1.2.5 Indexes

- Rows and columns have both an integer location (0,1,2,...) and an index (name).
- In our table, column names are `'country'`, `'region'` etc.
- We have not named rows, so a default location-based index was constructed.
  - See the sorted tables above—their index labels are kept from the original.
- Indexes can be obtained by attributes `index` and `columns`.
- We can set the country name as an index using `set_index`, the opposite is `reset_index` (in Series, use `set_axis` and `reset_index`).
- Index can be more complex (multiindex), we will see later.

[99]:
```
display(Markdown("**`table.columns` is an object of class `Index`:**"),
        table.columns)
display(Markdown("**`table.columns.values` is an array of column names:**"),
        table.columns.values)

display(Markdown("**`table.index.values` is an array of row names, "
                 + "here equal to location:**"),
        table.index.values)
display(Markdown("**`index` for Series `areas`:**"), areas.index.values)

display(Markdown("**`table` after setting country name as index:**"))
table2 = table.set_index('country')
display(table2)

display(Markdown("**`reset_index` will put the index back as a column:**"))
table3 = table2.reset_index()
display(table3)
```

`table.columns` is an object of class `Index`:

```
Index(['country', 'region', 'population', 'area', 'landlocked'], dtype='object')
```

`table.columns.values` is an array of column names:

```
array(['country', 'region', 'population', 'area', 'landlocked'],
      dtype=object)
```

`table.index.values` is an array of row names, here equal to location:

```
array([0, 1, 2, 3])
```

index for Series `areas`:

```
array([0, 1, 2, 3])
```

table after setting country name as index:

```
                region  population    area  landlocked
country
Slovakia        Europe     5450421   49035        True
Czech Republic  Europe    10649800   78866        True
Hungary         Europe     9772756   93030        True
Poland          Europe    38386000  312696       False
```

`reset_index` will put the index back as a column:

```
          country  region  population    area  landlocked
0        Slovakia  Europe     5450421   49035        True
1  Czech Republic  Europe    10649800   78866        True
2         Hungary  Europe     9772756   93030        True
3          Poland  Europe    38386000  312696       False
```

1.2.6 Accessing elements by index

- Method `some_table.loc[row, column]` is an analog of `iloc`, but using indexes rather than locations.
- You can also use `[]`, and pandas will try to guess whether it is an index or location (but sometimes it may guess wrong, so it is better to use explict `iloc` and `loc`).
- Some examples for `Series`:

[100]:
```
populations2 = populations.set_axis(countries)
display(Markdown("**`populations2` Series with index:**"), populations2)
display(Markdown("**`populations2.loc['Slovakia']`**:"),
        populations2.loc['Slovakia'])
display(Markdown("**`populations2.loc[['Slovakia','Poland']]`**:"),
        populations2.loc[['Slovakia','Poland']])

display(Markdown("**`populations2[1]` and `populations2['Czech Republic']`**:"))
display(populations2[1], populations2['Czech Republic'])
```

`populations2` Series with index:

```
Slovakia           5450421
Czech Republic    10649800
Hungary            9772756
Poland            38386000
dtype: int64
```

populations2.loc['Slovakia']:

```
5450421
```

populations2.loc[['Slovakia','Poland']]:

```
Slovakia     5450421
Poland      38386000
dtype: int64
```

populations2[1] and populations2['Czech Republic']:

```
10649800
```

```
10649800
```

1.2.7 Operations and functions on Series

- Operations such as `+`, `*` can be applied on two Series, causing them to be used on each corresponding pair of elements.
- For example, `populations / areas` will compute population density for each country.
- You can also use a single number (scalar) as an operand, e.g. `populations / 1e6` will get population in millions.
- NumPy also contains functions that can be applied to each element of a series, e.g. `np.log(populations)`.
- Relational operators such as `populations < 10e6` produce Series of boolean values.
  - Those can be then used in `[]` or `loc`.

[101]:
```
# creating two Series with country as index
populations2 = populations.set_axis(countries)
areas2 = areas.set_axis(countries)
display(Markdown("**`populations2 / areas2`:**"), populations2 / areas2)
display(Markdown("**`populations2 / 1e6`:**"), populations2 / 1e6)
display(Markdown("**`populations2 > 10e6`:**"), populations2 > 10e6)
display(Markdown("**`areas2[populations2 > 10e6]`:**"),
        areas2.loc[populations2 > 10e6])
display(Markdown("**`np.log10(populations2)`:**"), np.log10(populations2))
```

populations2 / areas2:

```
Slovakia          111.153686
Czech Republic    135.036644
Hungary           105.049511
Poland            122.758206
dtype: float64
```

populations2 / 1e6:

```
Slovakia           5.450421
Czech Republic    10.649800
Hungary            9.772756
Poland            38.386000
dtype: float64
```

populations2 > 10e6:

```
Slovakia          False
Czech Republic     True
Hungary           False
Poland             True
dtype: bool
```

areas2[populations2 > 10e6]:

```
Czech Republic     78866
Poland            312696
dtype: int64
```

np.log10(populations2):

```
Slovakia          6.736430
Czech Republic    7.027341
Hungary           6.990017
Poland            7.584173
dtype: float64
```

Beware: when we combine two Series, e.g. by +, Pandas will use index, not position, to pair up elements.

[102]:
```
a = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
b = pd.Series([10, 20, 30], index=['c', 'a', 'e'])
c = pd.Series([100, 200])
display(Markdown("**Series a:**"), a)
display(Markdown("**Series b:**"), b)
display(Markdown("**Series c:**"), c)
display(Markdown("**Series a+b:**"), a + b)
display(Markdown("**Series a+c:**"), a + c)
```

Series a:

```
a    1
b    2
c    3
d    4
dtype: int64
```

Series b:

```
c    10
a    20
```

```
e    30
dtype: int64
```

Series c:

```
0    100
1    200
dtype: int64
```

Series a+b:

```
a    21.0
b     NaN
c    13.0
d     NaN
e     NaN
dtype: float64
```

Series a+c:

```
a   NaN
b   NaN
c   NaN
d   NaN
0   NaN
1   NaN
dtype: float64
```

1.2.8 Working with DataFrame columns

- `DataFrame` is similar to a dictionary of `Series` objects (columns).
- For example, `table['area']` or `table.area` is the **column** of country areas.
- New columns can be added to a DataFrame: `table['density'] = table['population'] / table['area']`
- But `table[0:2]` are the first 2 **rows** of the table.
  - To be save, use `loc[]` / `iloc[]` rather than just `[]`.
- By `table[table['population'] > 1e7]` we get countries with more that 10 million people (CZ, PL).

[103]:
```
display(Markdown("**`table['area']`:**"), table['area'])
display(Markdown("**`table.area`:**"), table.area)
display(Markdown("**Adding density:**"))
display(Markdown("`table['density'] = table['population'] / table['area']`"))
table['density'] = table['population'] / table['area']
display(Markdown("**`table[0:2]`:**"), table[0:2])
display(Markdown("**`table[table['population'] > 1e7]`:**"),
        table[table['population'] > 1e7])
```

`table['area']`:

```
0     49035
1     78866
```

```
2     93030
3    312696
Name: area, dtype: int64
```

table.area:

```
0     49035
1     78866
2     93030
3    312696
Name: area, dtype: int64
```

Adding density:

`table['density'] = table['population'] / table['area']`

table[0:2]:

	country	region	population	area	landlocked	density
0	Slovakia	Europe	5450421	49035	True	111.153686
1	Czech Republic	Europe	10649800	78866	True	135.036644

table[table['population'] > 1e7]:

	country	region	population	area	landlocked	density
1	Czech Republic	Europe	10649800	78866	True	135.036644
3	Poland	Europe	38386000	312696	False	122.758206

1.2.9 Selecting table rows with query

- Method query is very useful for selecting DataFrame rows satisfying some properties.
- In examples below, @ substitutes variable value.
- While loc[] and iloc[] raise an exception if the requested value is not found, query can return an empty table.

[104]:
```
display(Markdown("**`table.query(\"country=='Slovakia'\")`:**"),
        table.query("country=='Slovakia'"))

def get_country(table, country):
  """Get a given country from the table"""
  return table.query("country == @country")
display(Markdown("**The same but for Hungary and using a function:**"),
        get_country(table, 'Hungary'))

display(Markdown("**Query with an empty result:**"))
display(Markdown("`table.query(\"population < 10e6 and not landlocked\")`:"))
display(table.query("population < 10e6 and not landlocked"))
```

table.query("country=='Slovakia'"):

	country	region	population	area	landlocked	density
0	Slovakia	Europe	5450421	49035	True	111.153686

The same but for Hungary and using a function:

```
   country  region  population   area  landlocked     density
2  Hungary  Europe     9772756  93030        True  105.049511
```

Query with an empty result:

`table.query("population < 10e6 and not landlocked")`:

```
Empty DataFrame
Columns: [country, region, population, area, landlocked, density]
Index: []
```

1.2.10 Importing and exporting data

- Import and export is possible using many file formats (text-based CSV, JSON, HTML; binary Excel, HDF5 etc.).
- We will mostly use CSV (=comma separated values) format.
  - Each table row is one line of the file.
  - Columns are separated by commas.
  - Columns containing commas or end-of-line characters may be enclosed in quotation marks.
  - Sometimes a different column separator is used, e.g. tab `"\t"`.
- Writing our table to a csv file: `table.to_csv("countries.csv")`.
  - If run in Colab, this will create a temporary file, which you can save on your computer (see the right panel, tab Files).
- Conversely, `table2 = pd.read_csv("countries.csv", index_col=0)` will read data from the file to a new `DataFrame` called `table2`.
- Input and output functions allow you to set many optional arguments to tweak the format.

1.3 Example: a table of country populations from the United Nations

- Obtained from the UN webpage https://data.un.org/
- We will read the table in CSV format directly from a URL.
- We need to play a bit with settings:
  - We skip the top two lines.
  - We supply our own (simpler) column names.
  - We specify character encoding (default is UTF8) and that thousands are separted by a comma in numerical values, such as 1,000,000.
  - Note that empty fields (missing values) are imported as `np.NaN`.

[105]:
```
# original source:
# url = 'https://data.un.org/_Docs/SYB/CSV/
↪SYB65_1_202209_Population,%20Surface%20Area%20and%20Density.csv'
# our local copy:
url = 'https://bbrejova.github.io/viz/data/Un_population.csv'
column_names = ['Region ID', 'Region', 'Year',
                'Series', 'Value', 'Footnotes', 'Source']
un_table = pd.read_csv(url, encoding='latin-1', names=column_names,
                       skiprows=2, thousands=",")
```

```
# print the first 5 rows to check the result
un_table.head()
```

[105]:
```
   Region ID                         Region  Year  \
0          1  Total, all countries or areas  2010
1          1  Total, all countries or areas  2010
2          1  Total, all countries or areas  2010
3          1  Total, all countries or areas  2010
4          1  Total, all countries or areas  2010

                                              Series    Value Footnotes  \
0            Population mid-year estimates (millions)  6985.60       NaN
1  Population mid-year estimates for males (milli…  3514.41       NaN
2  Population mid-year estimates for females (mil…  3471.20       NaN
3                   Sex ratio (males per 100 females)   101.20       NaN
4     Population aged 0 to 14 years old (percentage)    27.10       NaN

                                              Source
0  United Nations Population Division, New York, …
1  United Nations Population Division, New York, …
2  United Nations Population Division, New York, …
3  United Nations Population Division, New York, …
4  United Nations Population Division, New York, …
```

[106]:
```
# print the last 5 rows, to see if the bottom looks ok
un_table.tail()
```

[106]:
```
      Region ID    Region  Year  \
7868        716  Zimbabwe  2022
7869        716  Zimbabwe  2022
7870        716  Zimbabwe  2022
7871        716  Zimbabwe  2022
7872        716  Zimbabwe  2022

                                                Series  Value  \
7868  Population mid-year estimates for females (mil…   8.61
7869                  Sex ratio (males per 100 females)  89.40
7870     Population aged 0 to 14 years old (percentage)  40.60
7871         Population aged 60+ years old (percentage)   4.80
7872                                 Population density  42.20

                                        Footnotes  \
7868  Projected estimate (medium fertility variant).
7869  Projected estimate (medium fertility variant).
7870  Projected estimate (medium fertility variant).
7871  Projected estimate (medium fertility variant).
7872  Projected estimate (medium fertility variant).
```

```
                                                   Source
7868  United Nations Population Division, New York, …
7869  United Nations Population Division, New York, …
7870  United Nations Population Division, New York, …
7871  United Nations Population Division, New York, …
7872  United Nations Population Division, New York, …
```

[107]:
```
# check types of columns; strings are imported as object, which is expected
un_table.dtypes
```

[107]:
```
Region ID      int64
Region        object
Year           int64
Series        object
Value        float64
Footnotes     object
Source        object
dtype: object
```

- Each country has data for several years.
- There are several values per country and year, e.g. total population, the number of men and women, sizes of three age groups.
- The first part of the table contains various continents and regions, later individual countries arranged alphabetically from `'Afghanistan'` to `'Zimbabwe'`.

1.3.1 A simple table with total population across years

We will create a simpler table `country_pop`.

- It will contain only countries, not regions.
- It will contain only rows with total population, all available years.
- It will contain columns `Country` (originally `Region`) ,`Year`, and `Population` (originally `Value`).

[108]:
```
# get all rows for Afghanistan, choose the label for the first of them
first_country = un_table.query('Region == "Afghanistan"').index[0]
# get all rows from the first Afghanistan onwards and all columns
un_countries = un_table.iloc[first_country:, :]
# get only rows with total population and select only some columns using loc
country_pop = (un_countries
               .query('Series=="Population mid-year estimates (millions)"')
               .loc[:, ['Region', 'Year', 'Value']]
               .rename(columns={'Value':'Population', 'Region':'Country'}))
# print the start of the result
country_pop.head()
```

[108]:

	Country	Year	Population
930	Afghanistan	2010	28.19
937	Afghanistan	2015	33.75
945	Afghanistan	2020	38.97
953	Afghanistan	2022	41.13
960	Albania	2010	2.91

1.4 Tidy data, wide and long tables

- The original UN table has in column `Value` various values, including population size, sex ratio, population density, etc.
- In general, one column of a table should contain values of the same type.
- This is true in our `country_pop` table with columns `Country`, `Year`, and `Population`.
- This type of table is called **long** and is usually preferable.
- For some analysis, we may want to have countries as rows and years as columns; this is called a **wide table**.
- Pandas has methods to convert between the two formats, e.g. `wide_to_long`, `melt`, `pivot`, `unstack` etc.
- See the article Tidy data by Hadley Wickham for a longer discussion.

1.5 Back to example: comparing populations in 2010 and 2022

- We select only two years from `country_pop`.
- Function pivot will use the column `Country` as the row index, values from column `Year` as new column names and values from column `Population` as values to populate the table itself.
- Finally we rename the columns so that they are strings starting with a letter; otherwise they are harder to be used in `query`.

[109]:
```
display(Markdown("**Original `country_pop` table:**"), country_pop.head())


pop = (country_pop.query("Year==2010 or Year==2022")
       .pivot(index='Country', columns=['Year'], values='Population')
       .rename(columns={2010:'pop2010', 2022:'pop2022'}))

display(Markdown("**New `pop` table:**"), pop.head())
```

Original `country_pop` table:

	Country	Year	Population
930	Afghanistan	2010	28.19
937	Afghanistan	2015	33.75
945	Afghanistan	2020	38.97
953	Afghanistan	2022	41.13
960	Albania	2010	2.91

New pop table:

```
Year            pop2010  pop2022
Country
Afghanistan       28.19    41.13
Albania            2.91     2.84
Algeria           35.86    44.90
American Samoa     0.05     0.04
Andorra            0.07     0.08
```

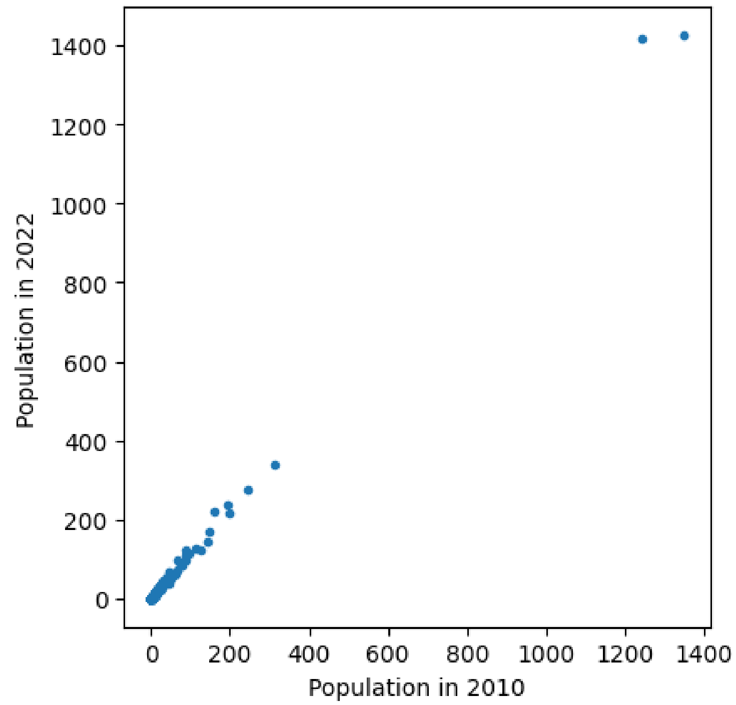
[110]:
```
# compute the difference between years for each country (positive = increase)
pop['difference'] = pop['pop2022'] - pop['pop2010']
# relative difference, as a fraction of population in 2010
#  (value 1 means 100% increase)
pop['relDifference'] = pop['difference'] / pop['pop2010']
pop.head()
```

[110]:
```
Year            pop2010  pop2022  difference  relDifference
Country
Afghanistan       28.19    41.13       12.94       0.459028
Albania            2.91     2.84       -0.07      -0.024055
Algeria           35.86    44.90        9.04       0.252091
American Samoa     0.05     0.04       -0.01      -0.200000
Andorra            0.07     0.08        0.01       0.142857
```

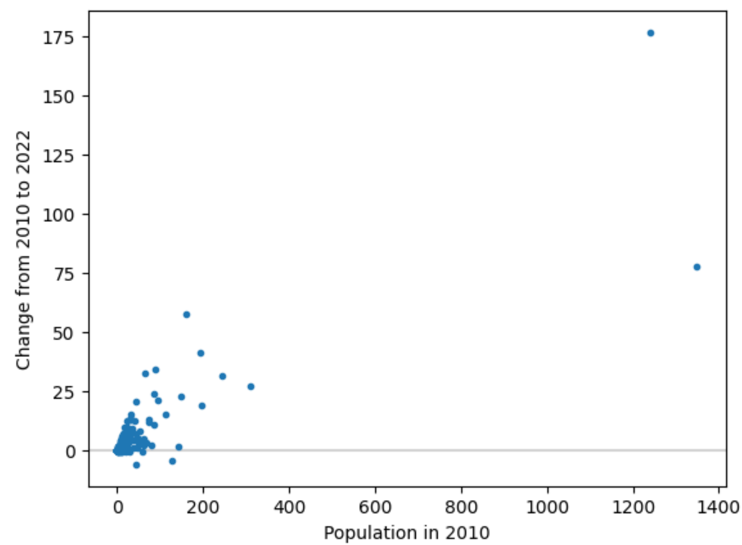
Now we will use this table to create some plots and tables.

- What can you observe from these data displays?
- Are some of these visualizations more useful than others or are they complementary? How could we improve them?
- What other questions you could ask about this table and how would you answer them using plots or tables?

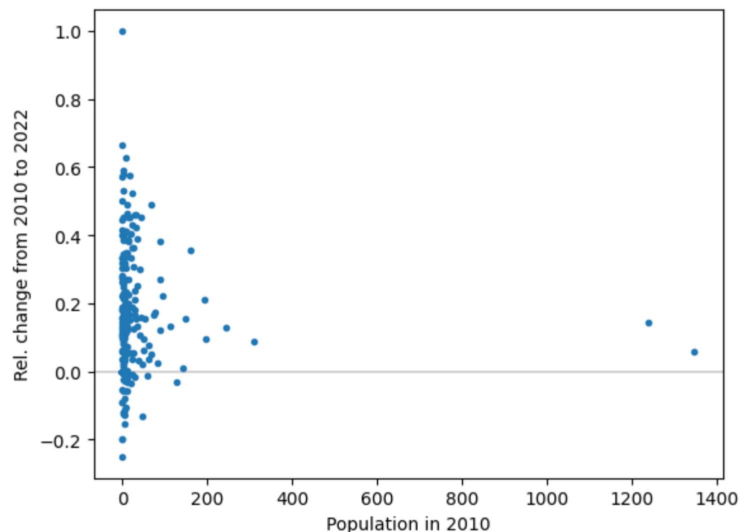
[111]:
```
figure, axes = plt.subplots()
axes.plot(pop.pop2010, pop.pop2022, '.')
axes.set_aspect('equal')
axes.set_xlabel('Population in 2010')
axes.set_ylabel('Population in 2022')
pass
```



[112]:
```
figure, axes = plt.subplots()
axes.axhline(0, color="lightgrey")
axes.plot(pop.pop2010, pop.difference, '.')
axes.set_xlabel('Population in 2010')
axes.set_ylabel('Change from 2010 to 2022')
pass
```



[113]:
```
figure, axes = plt.subplots()
axes.axhline(0, color="lightgrey")
axes.plot(pop.pop2010, pop.relDifference, '.')
axes.set_xlabel('Population in 2010')
axes.set_ylabel('Rel. change from 2010 to 2022')
pass
```



[114]:
```
pop.sort_values('relDifference').head(10)
```

[114]:
```
Year                        pop2010  pop2022  difference  relDifference
Country
Saint Martin (French part)     0.04     0.03       -0.01      -0.250000
American Samoa                 0.05     0.04       -0.01      -0.200000
Marshall Islands               0.05     0.04       -0.01      -0.200000
Bosnia and Herzegovina         3.81     3.23       -0.58      -0.152231
Ukraine                       45.68    39.70       -5.98      -0.130911
Puerto Rico                    3.72     3.25       -0.47      -0.126344
Lithuania                      3.14     2.75       -0.39      -0.124204
Latvia                         2.10     1.85       -0.25      -0.119048
Republic of Moldova            3.68     3.27       -0.41      -0.111413
Bulgaria                       7.59     6.78       -0.81      -0.106719
```

[115]:
```
pop.sort_values('relDifference', ascending=False).head(10)
```

[115]:
```
Year                           pop2010  pop2022  difference  relDifference
Country
Anguilla                          0.01     0.02        0.01       1.000000
```

Turks and Caicos Islands	0.03	0.05	0.02	0.666667
Jordan	6.93	11.29	4.36	0.629149
Oman	2.88	4.58	1.70	0.590278
Qatar	1.71	2.70	0.99	0.578947
Niger	16.65	26.21	9.56	0.574174
Mayotte	0.21	0.33	0.12	0.571429
Equatorial Guinea	1.09	1.67	0.58	0.532110
Angola	23.36	35.59	12.23	0.523545
Bonaire, St. Eustatius & Saba	0.02	0.03	0.01	0.500000

[116]:

```
neighbor_countries =␣
  ↪['Slovakia','Czechia','Hungary','Poland','Austria','Ukraine']
neighbors = pop.loc[neighbor_countries, : ]
display(neighbors)
```

Year	pop2010	pop2022	difference	relDifference
Country				
Slovakia	5.40	5.64	0.24	0.044444
Czechia	10.46	10.49	0.03	0.002868
Hungary	9.99	9.97	-0.02	-0.002002
Poland	38.60	39.86	1.26	0.032642
Austria	8.36	8.94	0.58	0.069378
Ukraine	45.68	39.70	-5.98	-0.130911

1.6 Summary and outlook

- We will work mostly with tabular data.
- We will store them in `DataFrame` from Pandas library.
- This is more convenient and more efficient than regular Python lists.
- We have seen several functions for basic manipulation:
  - `iloc[]`, `loc[]`, `query`, `head`, `set_index`, `reset_index`, `rename`, `pivot`, `copy`, `sort_values`, operations and functions on `Series`.
- Next lecture will be focused on examples of different chart types.
- More Pandas later.

Lecture 3a
Overview of Plot Types

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

Plan for today

- Types of variables (columns)
- Gallery of different plot types, some discussion of their properties
- Some notes on how to draw them in Python (more in a notebook)

Types of variables (columns)

Categorical / qualitative

- **Nominal:** values have no fixed ordering (for example, gender, country, color)
- **Ordinal:** values are ordered (for example education level primary / secondary / university; star ranking 0-5)

Numerical / quantitative

- **Discrete:** typically counts
- **Continuous:** typically measurements

Types of variables (columns)

Numerical / quantitative

- **Discrete:** typically counts
- **Continuous:** typically measurements

Numerical variables also categorized as follows:

- **Ratio (pomerová):** if zero means "none", and it is meaningful to compute ratios / percentages (mass, length, duration, cost, ...)
- **Interval:** does not have "true zero", we can subtract but not make ratios (temperature in degrees C, date)

Data for today

- Various country indicators downloaded from the World Bank for years 2000, 2010, 2020
- Population, area, GDP per capita, life expectancy, fertility (number of children per woman)
- Also classification into regions and income groups
- Which are categorical / numerical?

We will also use Gapminder life expectancy 1990-2021 from I01

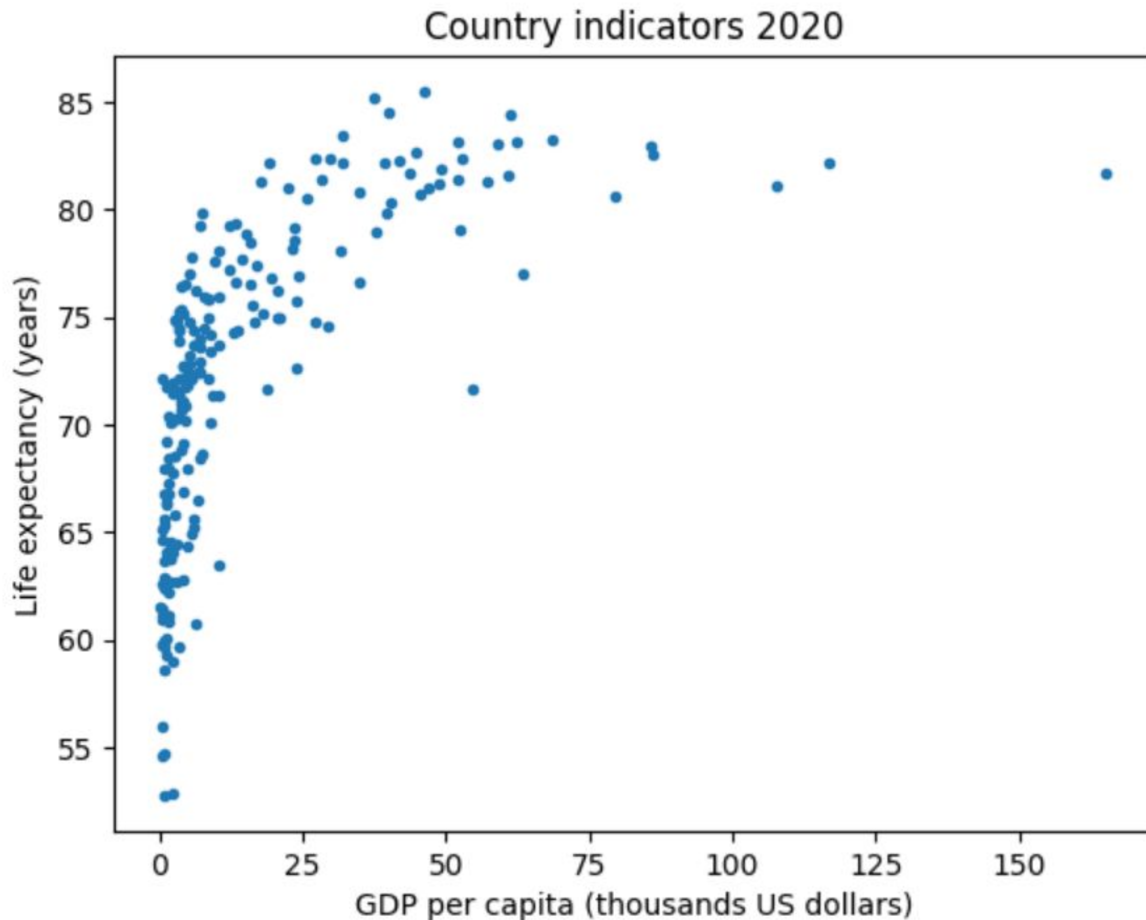
Scatter plot (bodový graf)

Good for two numerical variables (x and y).

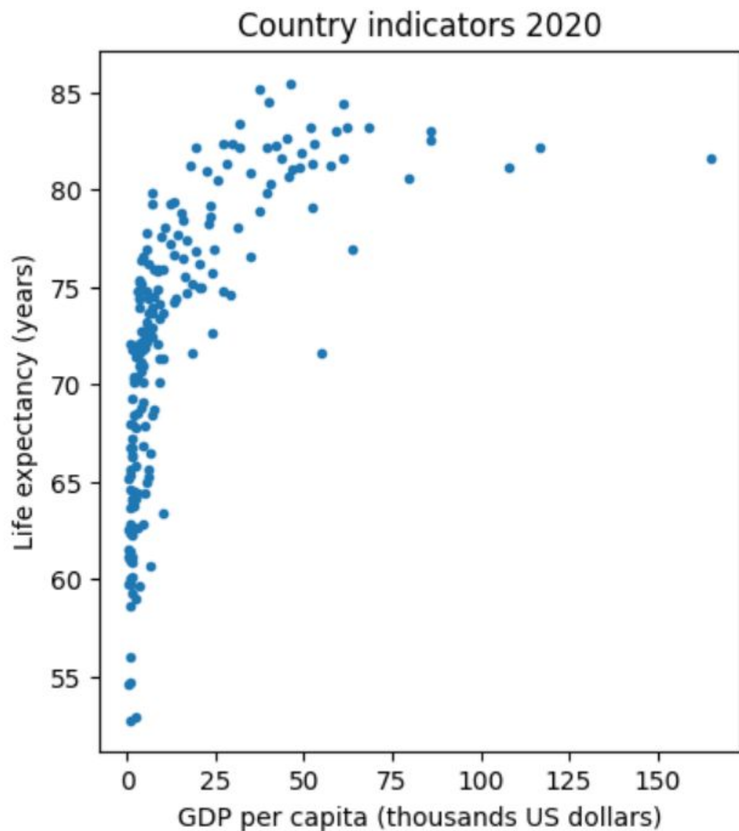
In this plot, many points near left boundary, most space empty.

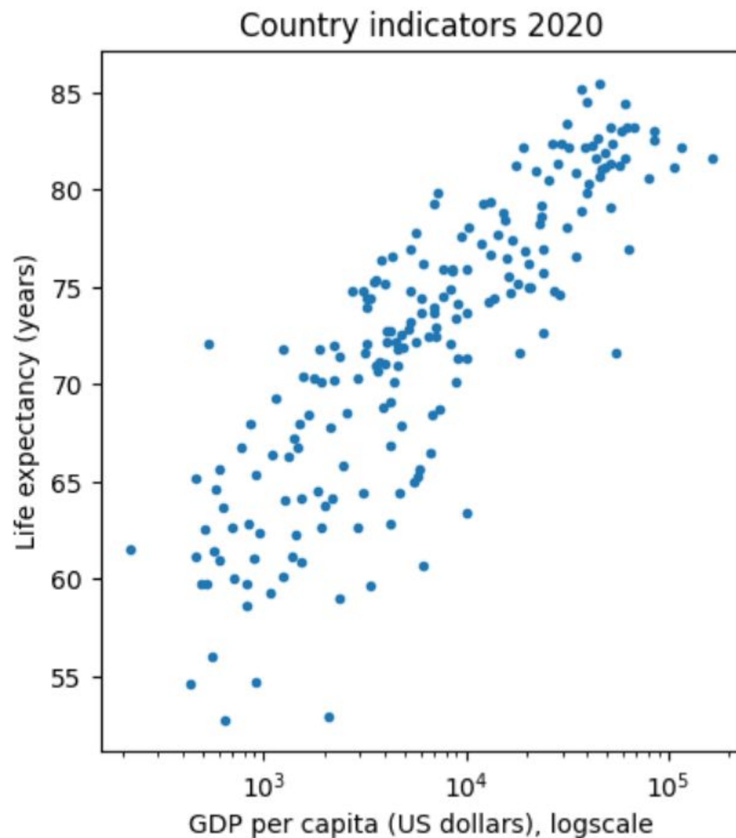
Solution 1: combine overall view and detail

Solution 2: log scale



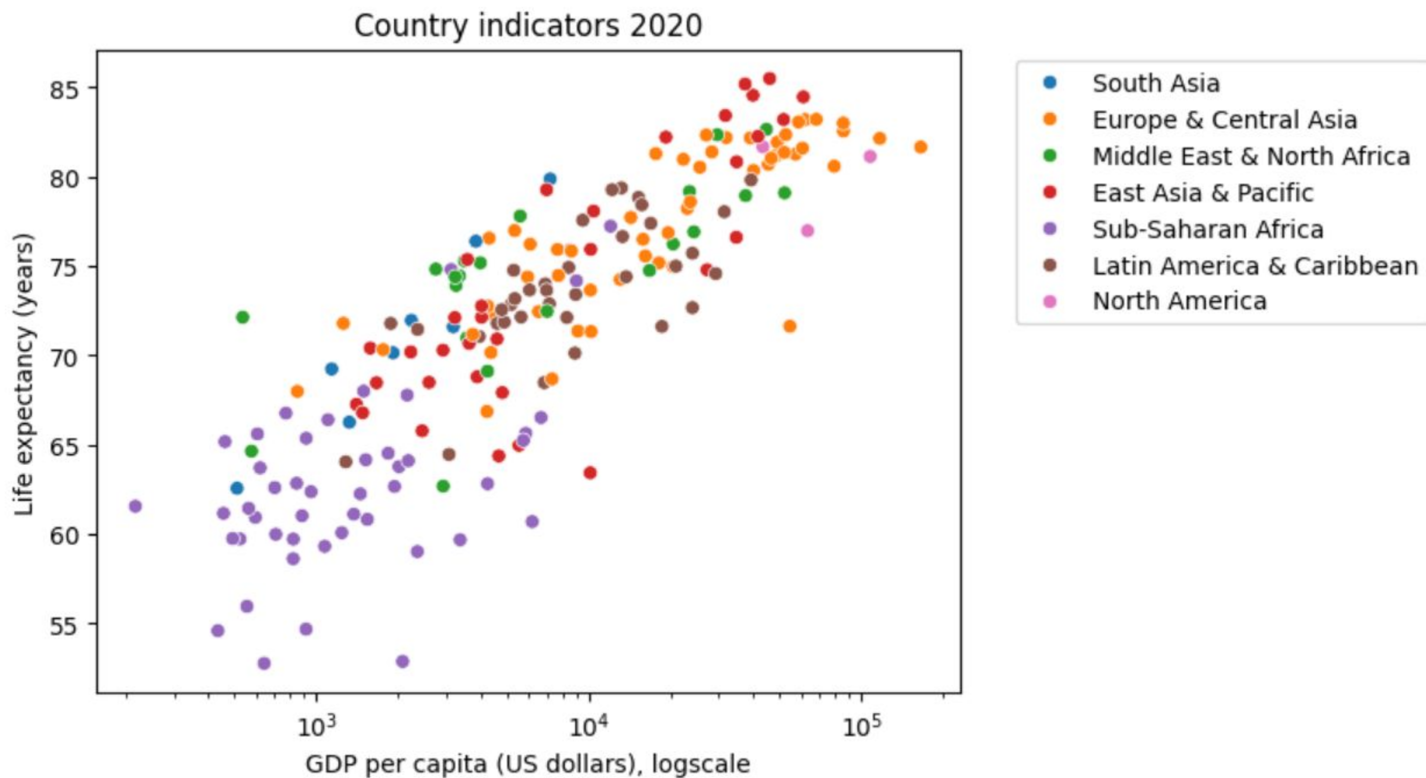
Country indicators 2020
Life expectancy (years)
85
80
75
70
65
60
55
0
25
50
75
100
125
150
GDP per capita (thousands US dollars)
Only GDP per capita < 25000 USD
Life expectancy (years)
0
5
10
15
20
25
GDP per capita (thousands US dollars)



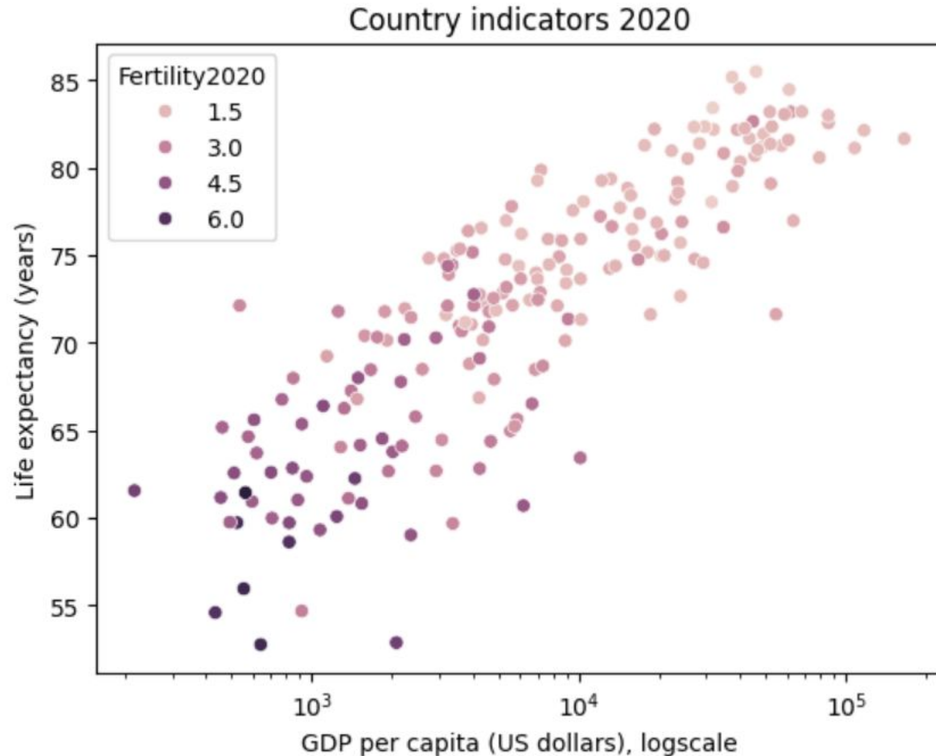


Log-scale x-axis: draw at log(x) instead of x, but axis ticks show values of x

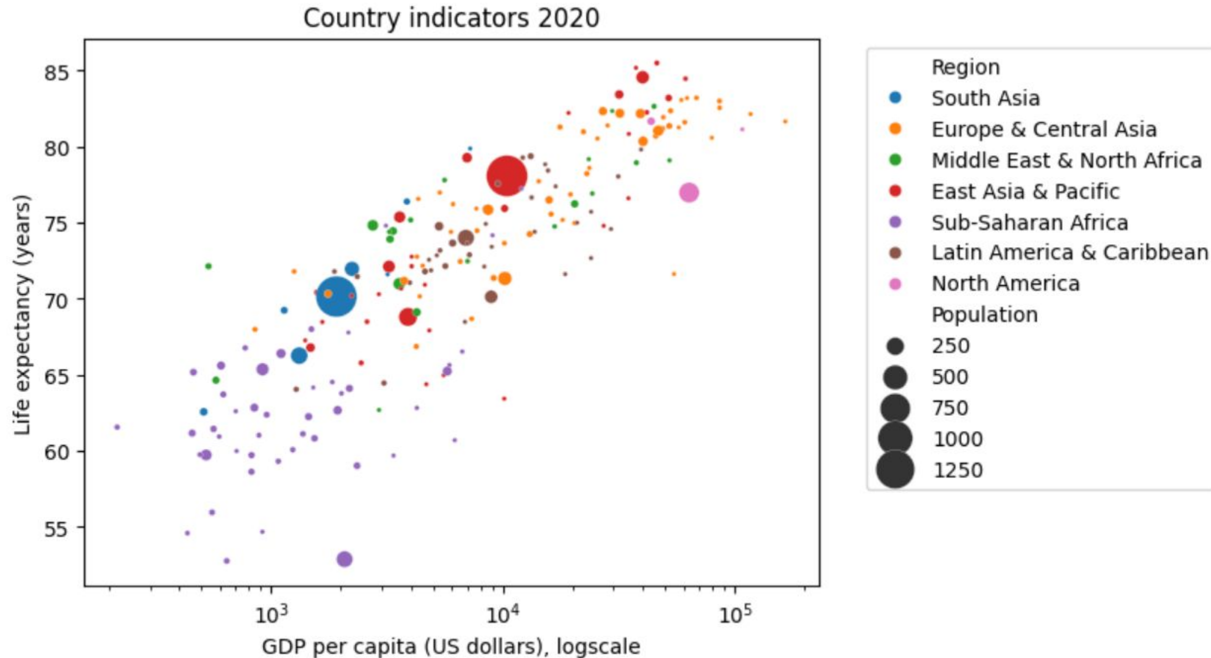
Adding a categorical variable with color



Adding a numerical variable with color scale

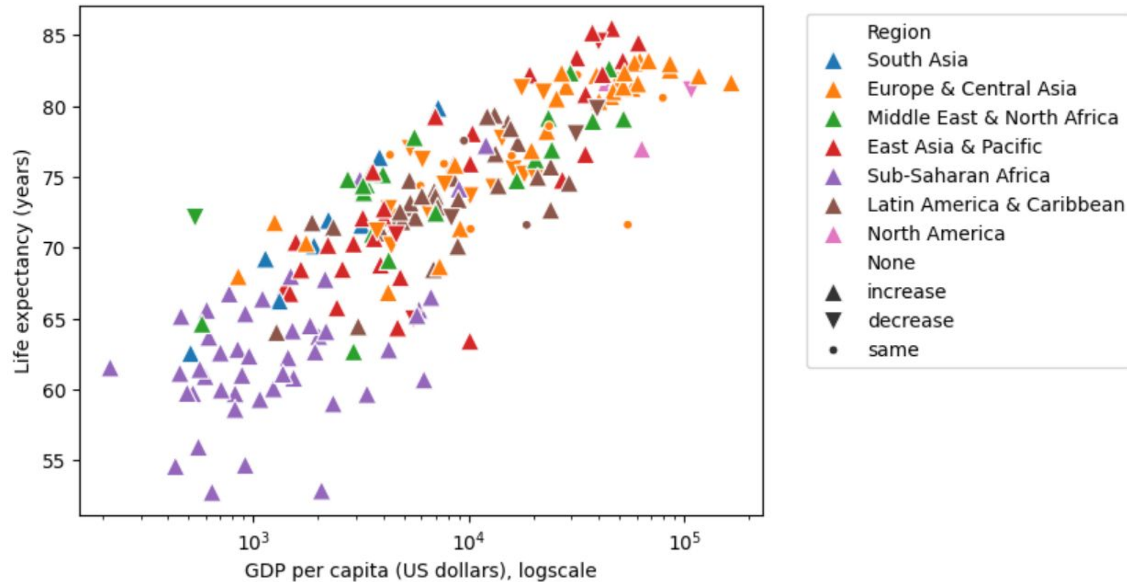


Adding numerical variable with marker size



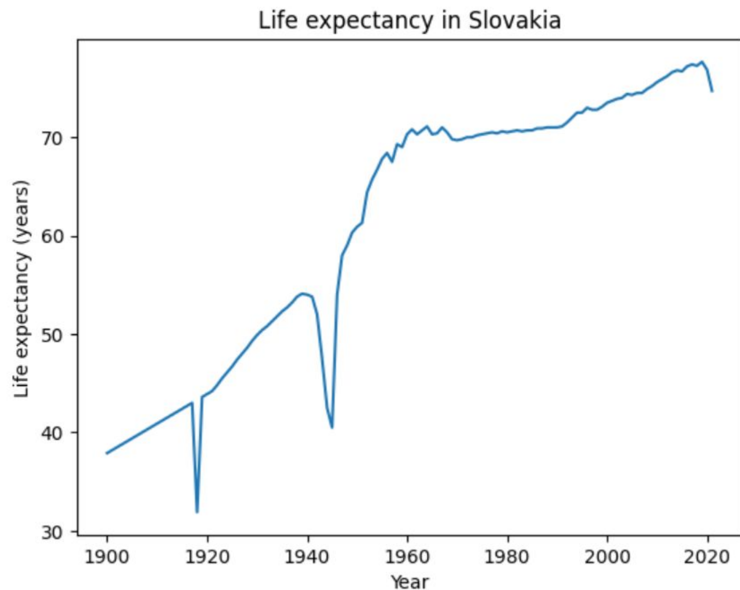
Variable value should be proportional to circle area, not diameter!

Adding categorical variable with marker shape



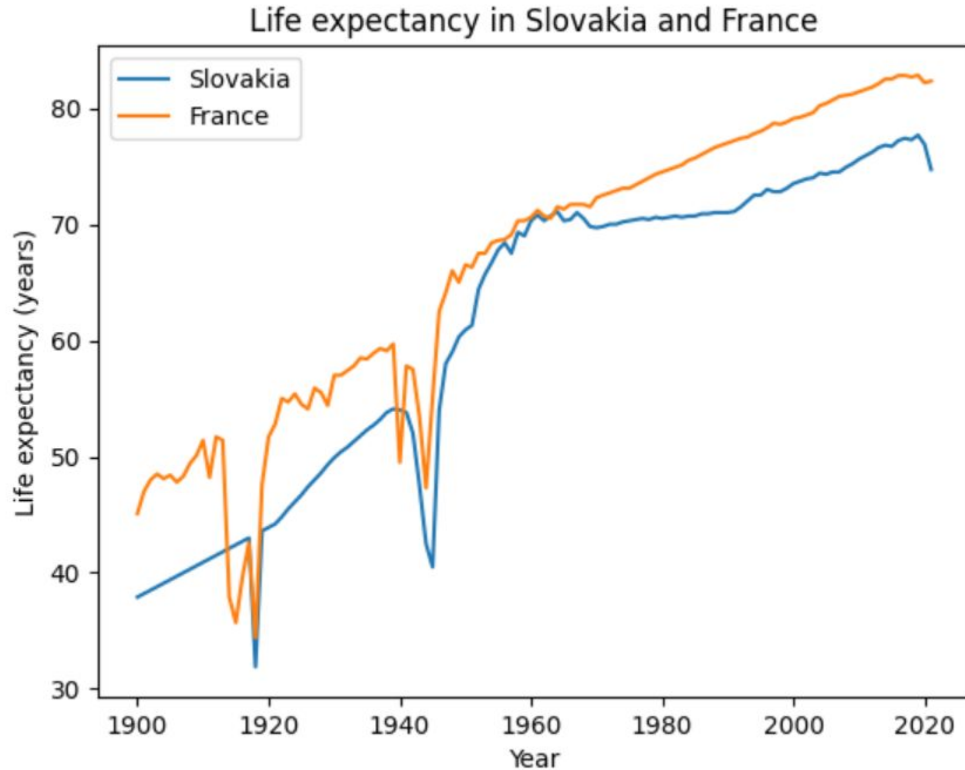
Hard to read, particularly for many data points
Showing population change between 2000 and 2020
If less than 1% change, marked as equal

Line graph (čiarový graf)

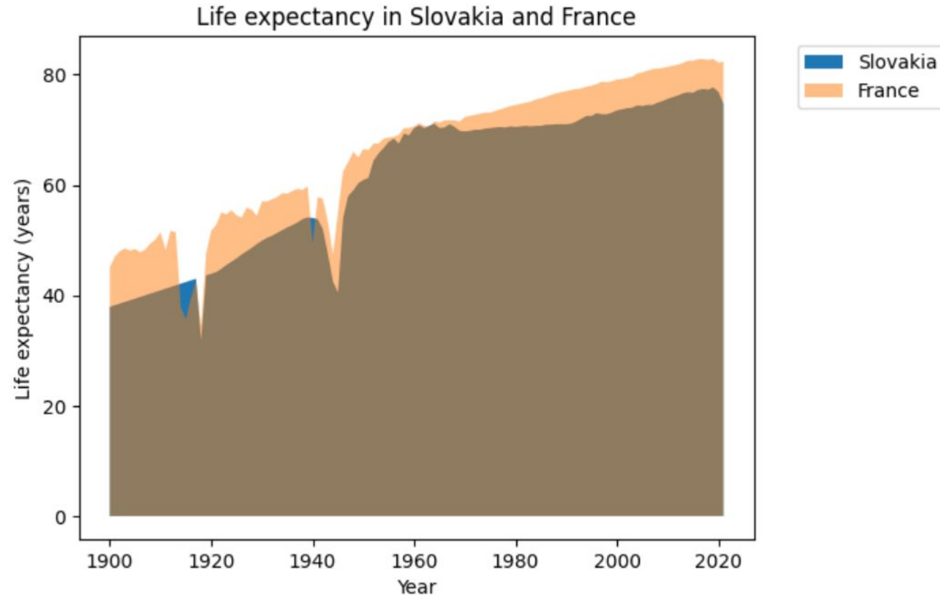


Emphasizing continuity between data points
Data points can be also shown as markers

Adding categorical variable with color



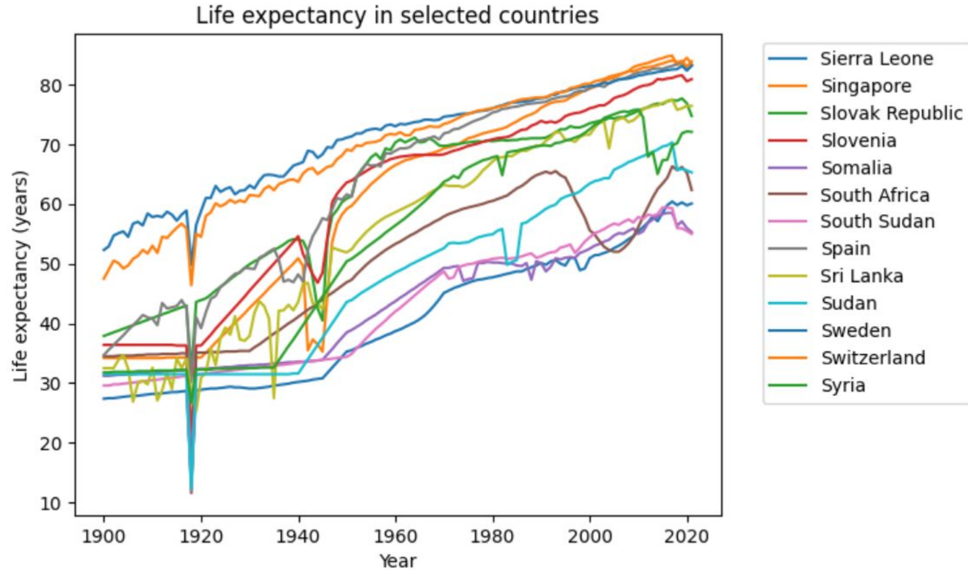
Area graph (plošný graf)



Y-axis must start at 0

Emphasizes differences more than line graph, but also more cluttered

Line graph with many lines



Hard to follow individual lines, but shows general trends and comparisons.
Countries with names Si..Sw, and having population at least 1 million.
Note that colors start to repeat.

Small multiples

A small plot for each value of a categorical variable

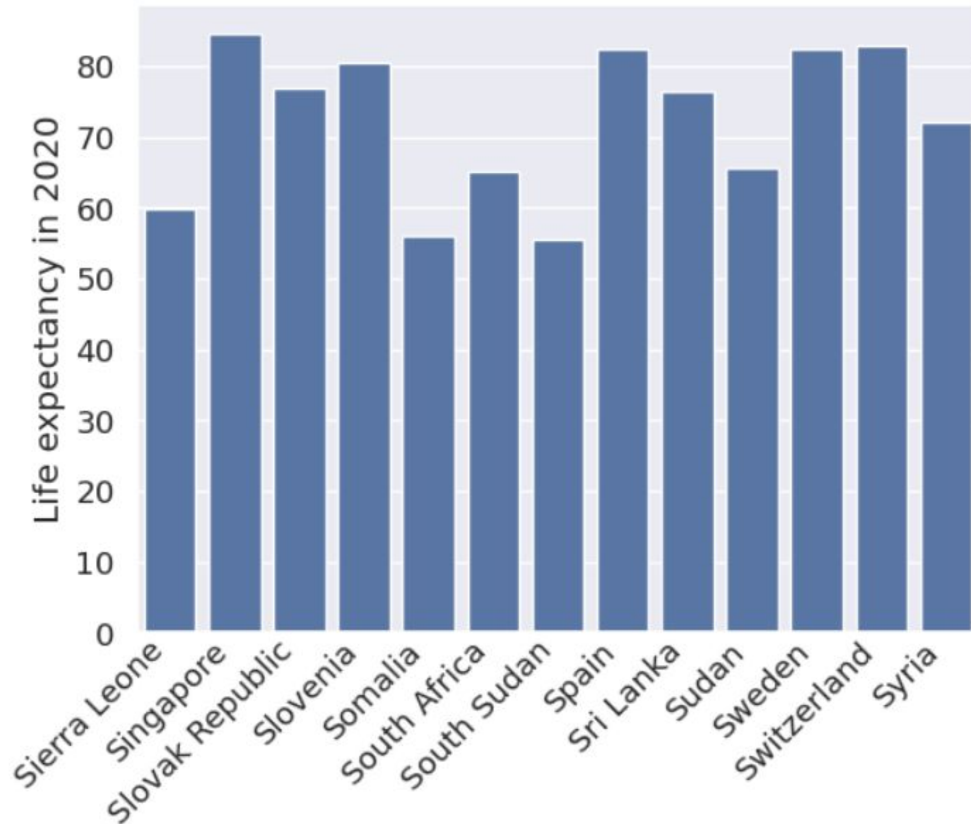
Must have the same axes!

Exact comparison difficult, but it is possible to notice different trends

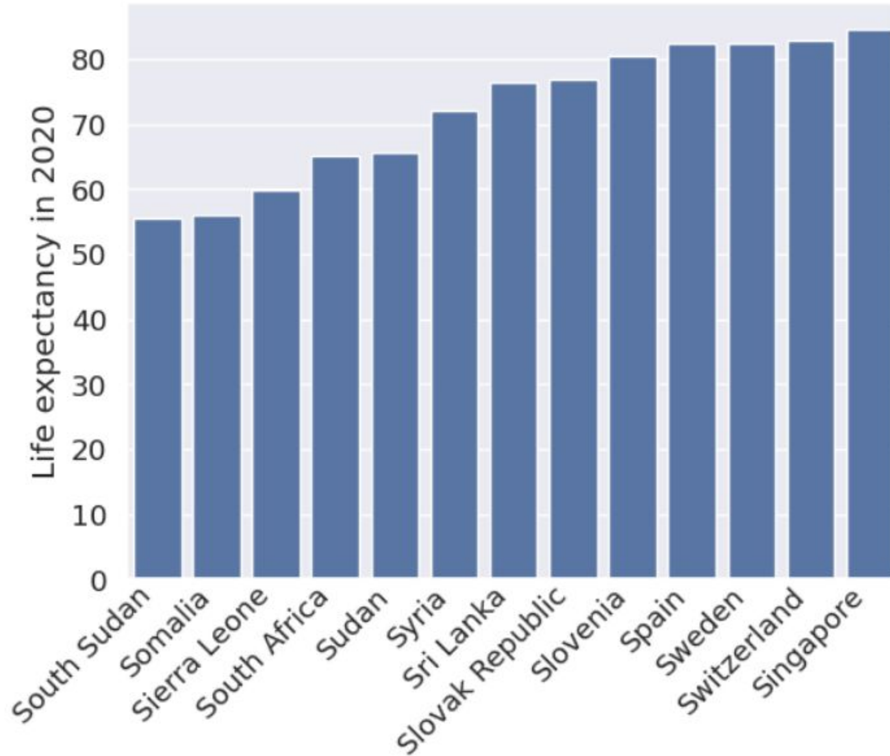
Bar graph (stĺpcový/pruhový graf)

X-axis is categorical

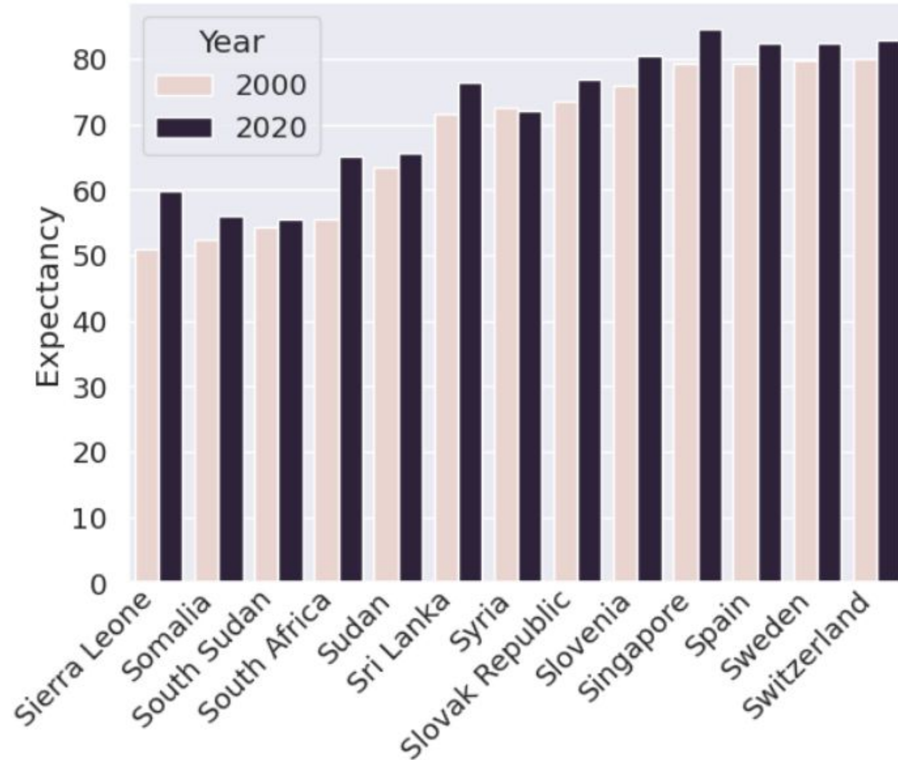
Y-axis must start at 0



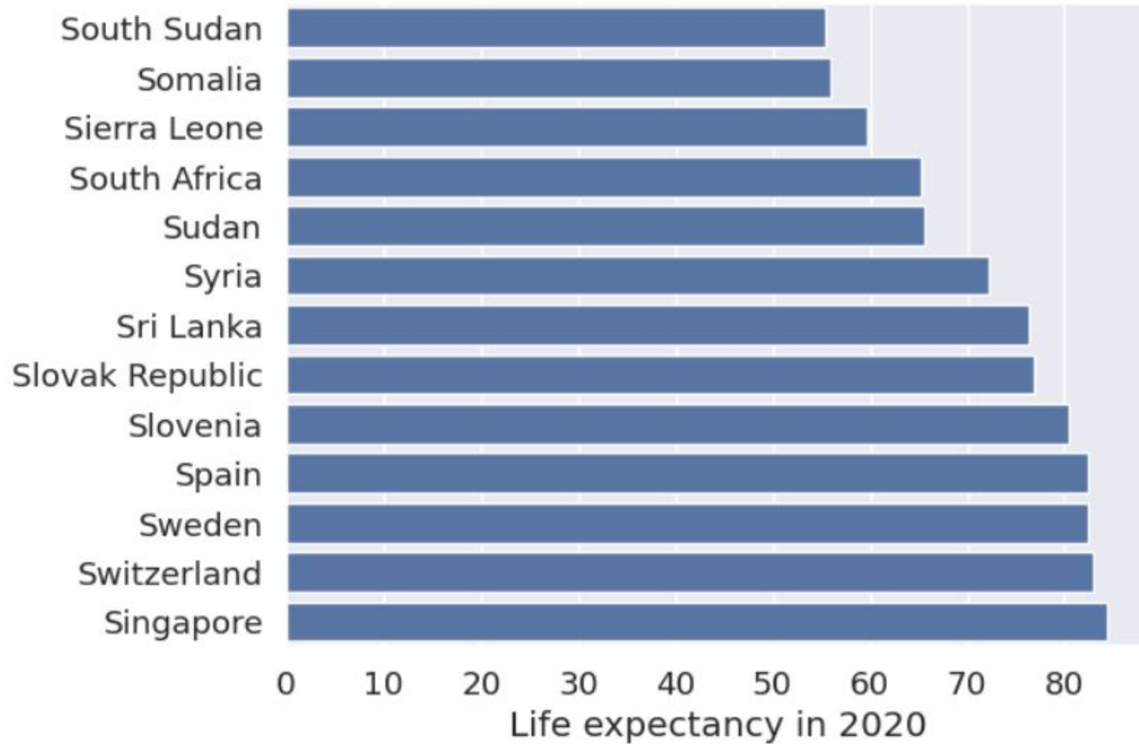
Bar graph with sorted columns



Bar graph with colored columns



Bar graphs can be horizontal



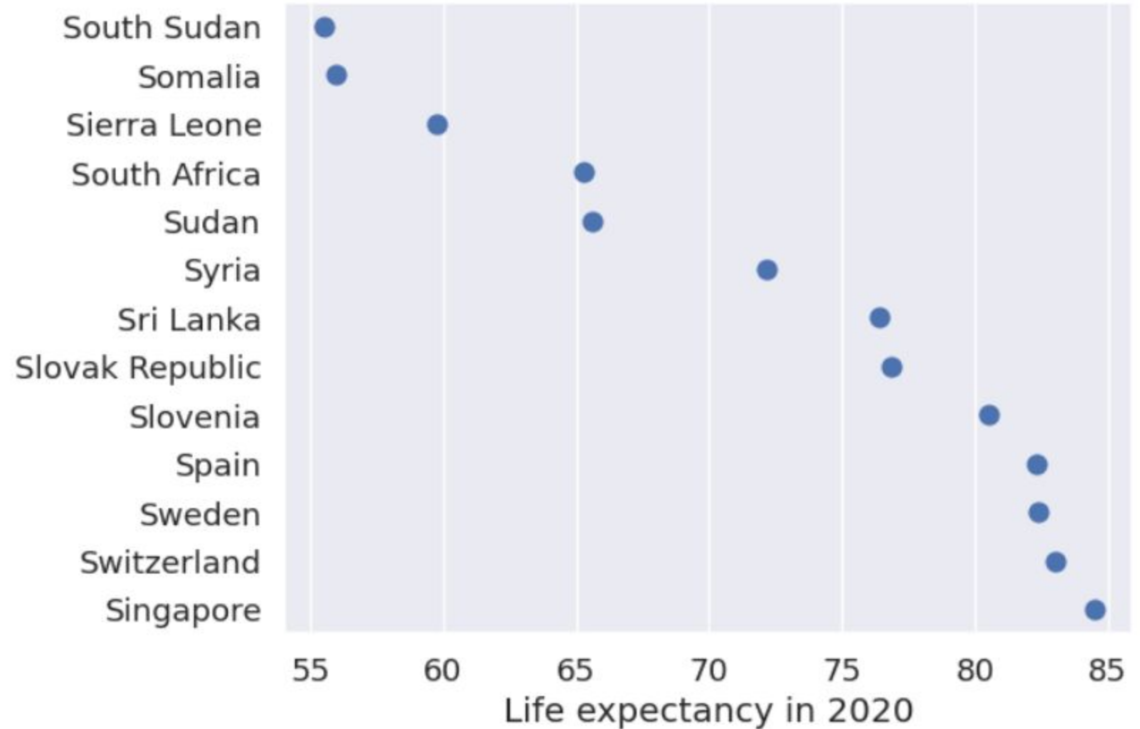
Dot plot

As bar graph but only dots shown at the top of the bar

Less clutter

X-axis does not need to start at 0 - better use of space

Can use multiple colors

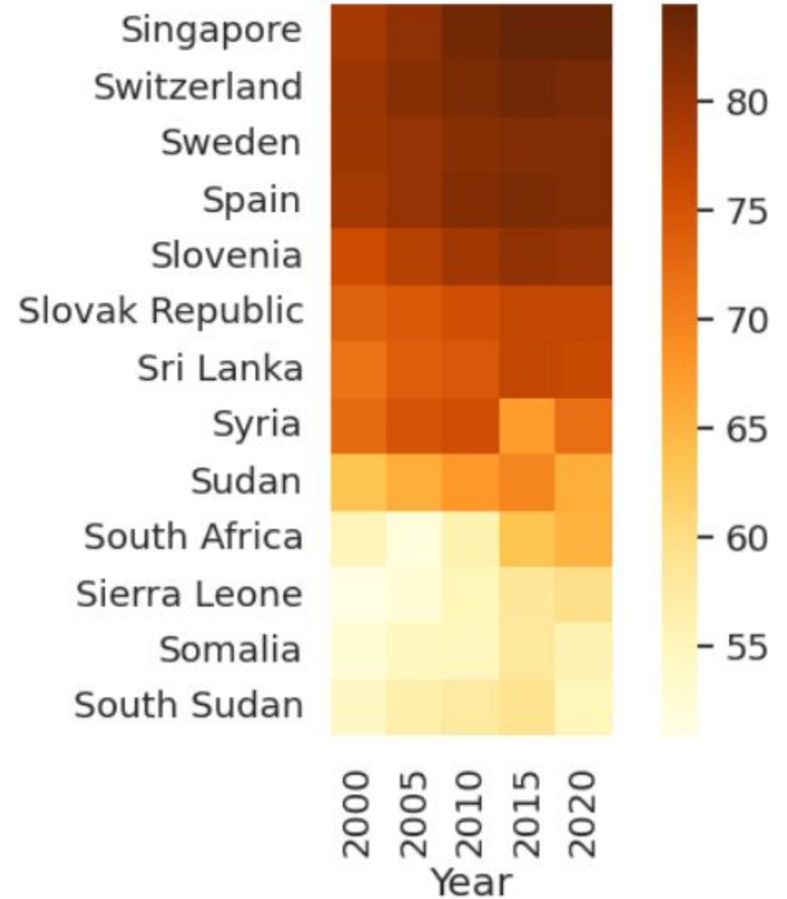


Heatmap

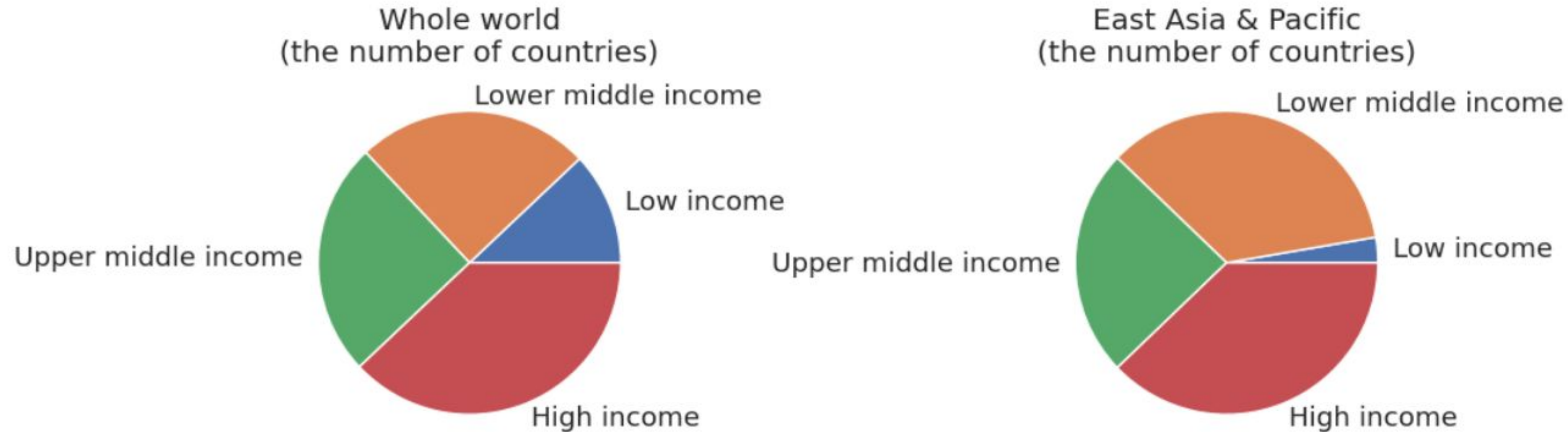
Both axes categorical

Numerical value shown in a color scale

Compact display, but color scales harder to read

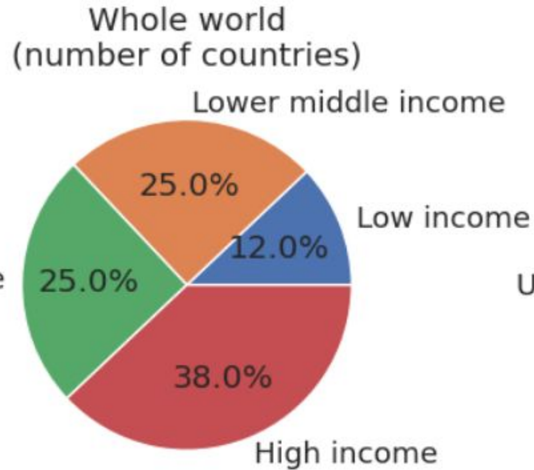


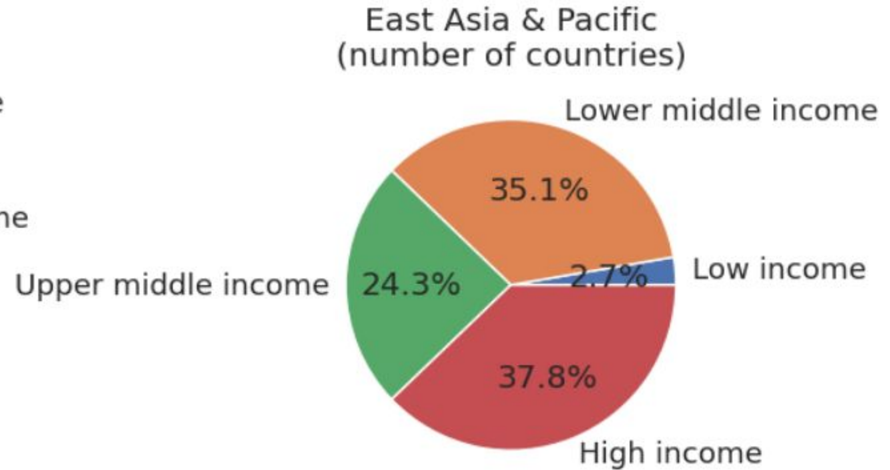
Pie chart (koláčový graf)



Obvious that percentages displayed
Very large values are easy to see (here high income)
Hard to compare similar values to each other
Space use not good

Pie chart with values labeled



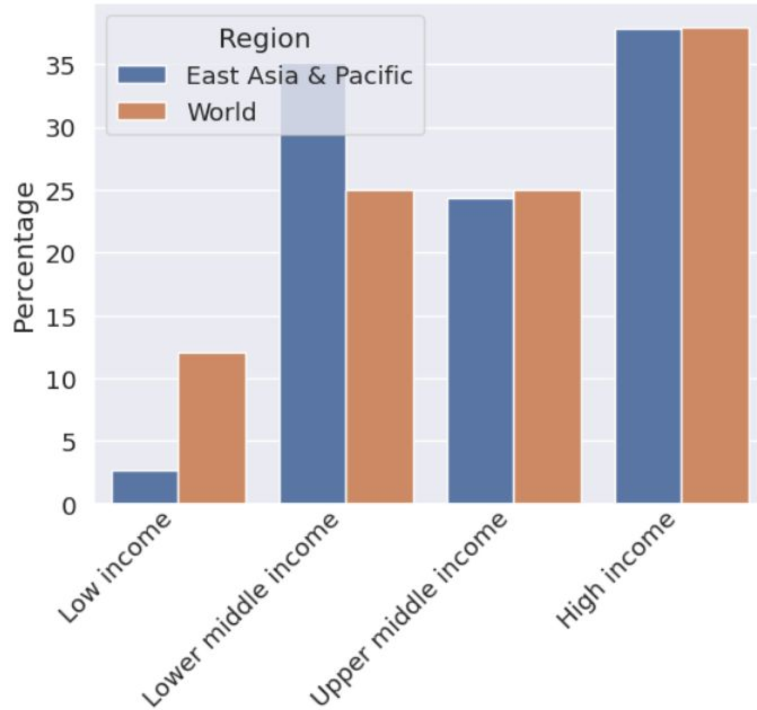


Easier to compare but still not ideal
Labeling values also useful in other types of graphs

Stacked (skladaný) bar graph instead of pie chart

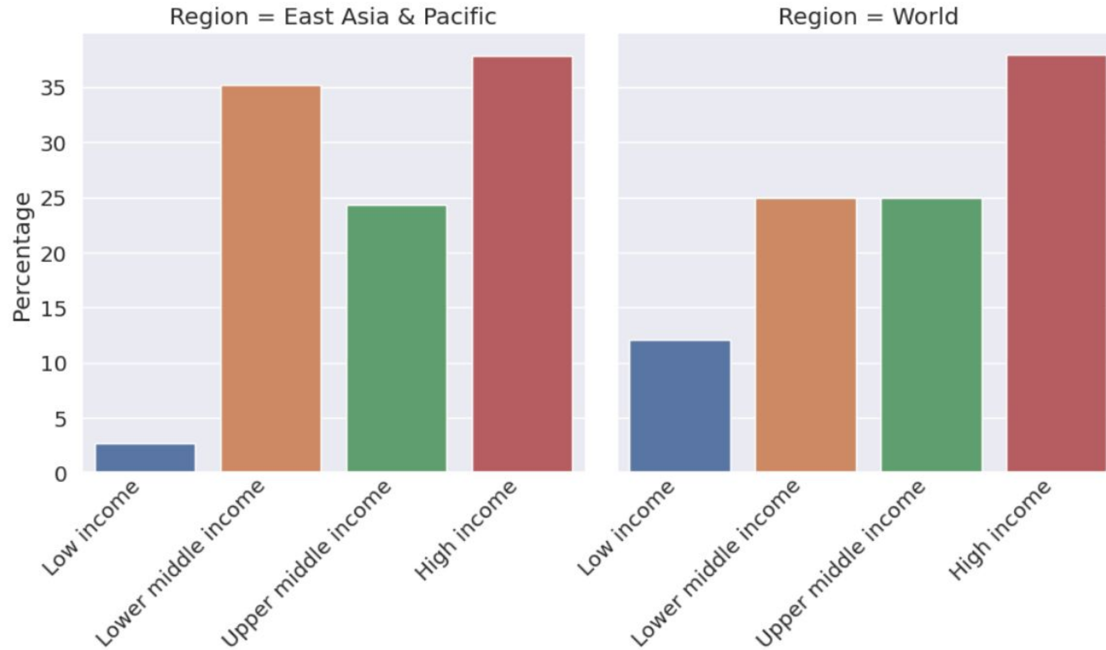
Rectangles easier to compare than wedges
Benefits from labeled values
Middle colors hard to compare across bars
Similar idea: stacked area plot (change in percentages over time)

Colored bar graph instead of pie chart



Easy to compare East Asia vs whole world.
Not obvious that we show parts of a whole.

Colored bar graph instead of pie chart



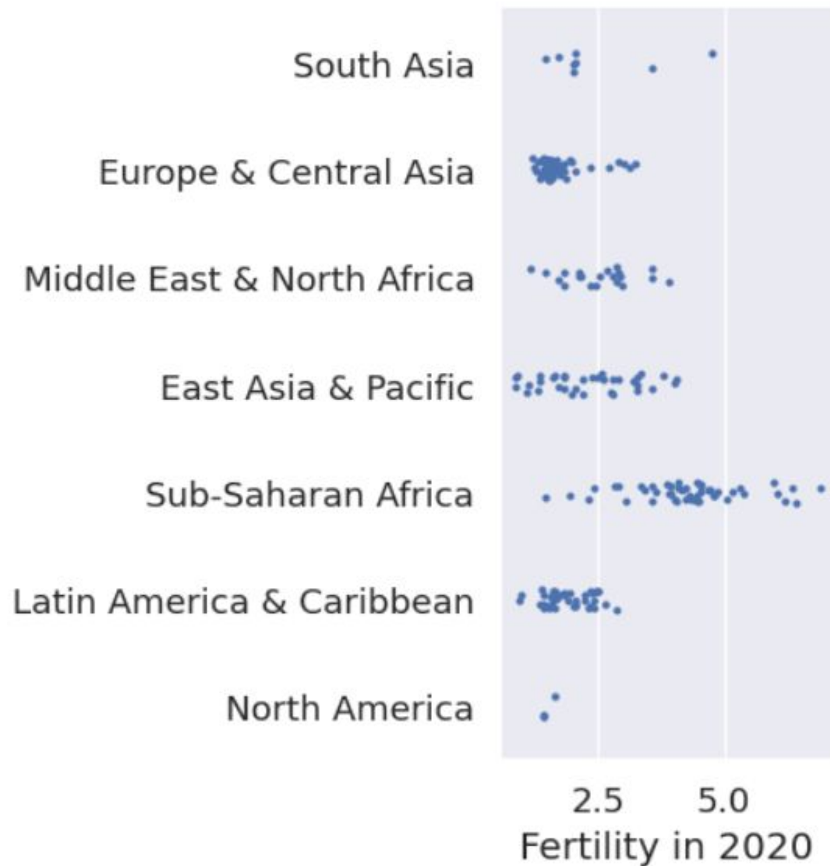
Easy to compare income groups within region

Strip plot

One axis categorical

Other axis shows individual data points

Jitter added in categorical axis to avoid point overlap

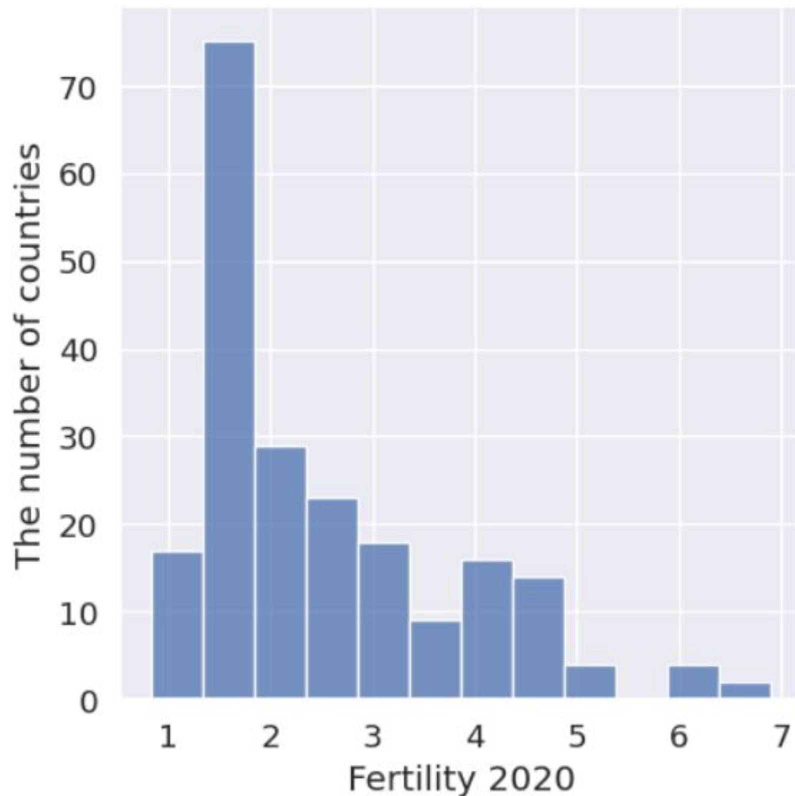


Histogram

For 1D numerical data

Split values into bins, show bin sizes as bar graph

We could use colors to display 2 or more histograms



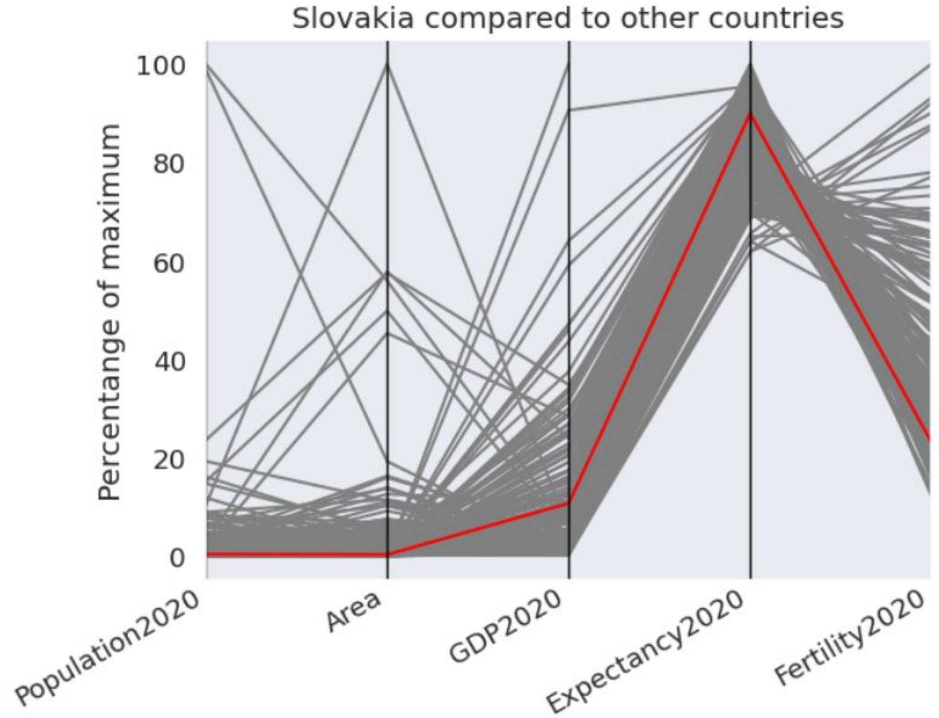
Parallel coordinates

Good for multidimensional numerical data

Each column one dimension

Here scaled as % of maximum value

Hard to see individual lines, but can show trends, compare groups shown in color or selected data point vs others



Parallel categories

Good for multidimensional categorical data

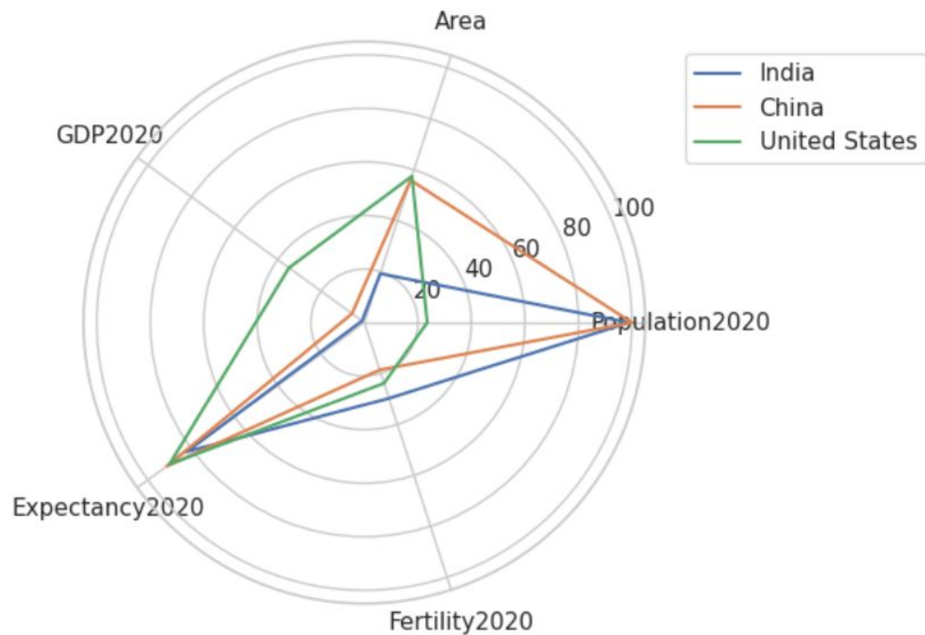
Each column one dimension

The widths of ribbons correspond to the number of countries

Radar chart (radarový graf)

Hard-to-read version of parallel coordinates

Perhaps some justification in cyclical domains, such as average temperature in months of a year



Now some Python

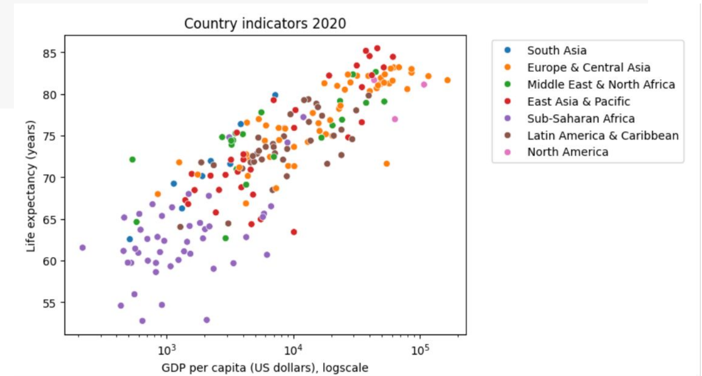
Overview of libraries

- Matplotlib
- Seaborn: an extension of Matplotlib, convenient for many types of plots
- Plotly: basic usage similar to Seaborn, plots interactive by default

Part of the main table `countries`

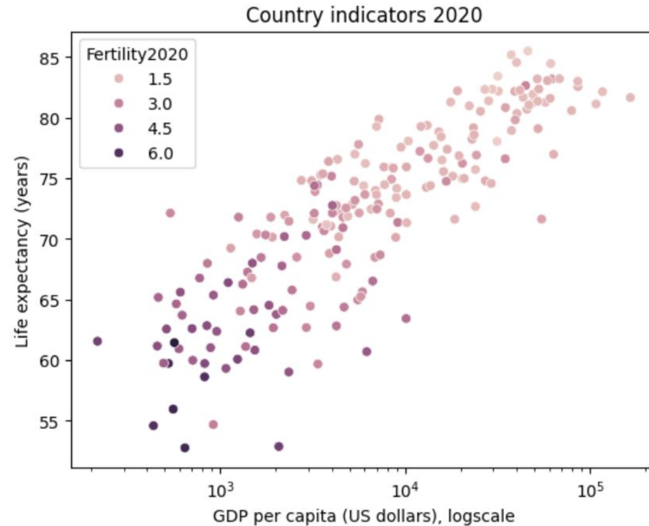
Country	ISO3	Region	Income Group	Population2000	Population2010	Population2020	Area	GDP2000	GDP2010
Afghanistan	AFG	South Asia	Low income	19542983.0	28189672.0	38972231.0	652860.0	NaN	562.499219
Albania	ALB	Europe & Central Asia	Upper middle income	3089026.0	2913021.0	2837849.0	28750.0	1126.683340	4094.349686
Algeria	DZA	Middle East & North Africa	Lower middle income	30774621.0	35856344.0	43451666.0	2381741.0	1780.376063	4495.921476
American Samoa	ASM	East Asia & Pacific	High income	58229.0	54849.0	46189.0	200.0	NaN	10446.863206
Andorra	AND	Europe & Central Asia	High income	66097.0	71519.0	77699.0	470.0	21620.465102	48237.890541

```python
# create plot using Seaborn
axes = sns.scatterplot(data=countries, x='GDP2020', y='Expectancy2020',
                       hue='Region')
# set plot properties using methods from Matplotlib
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
axes.semilogx()
# place legend outside the plot:
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```

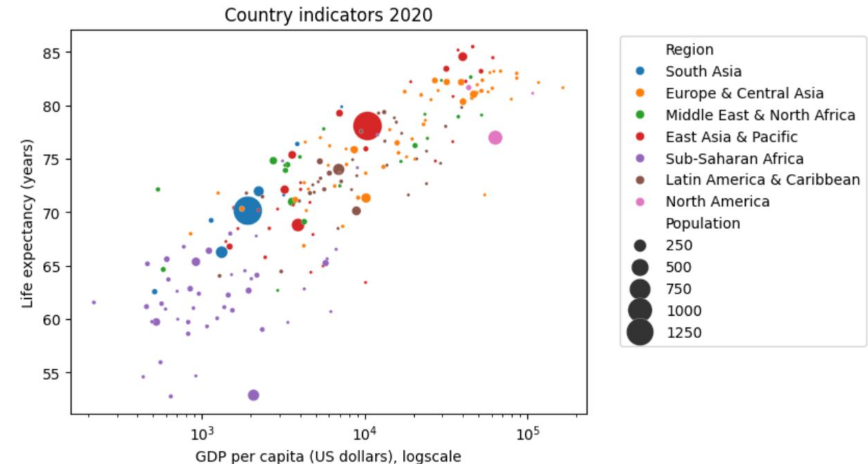


```python
# The same plot in Plotly
# We want to use index (country name) in the figure for tooltip info
# therefore we create a temporary table with column Country instead of index
temp_table = countries.reset_index()
# how to rename automated axis labels
fig_labels = {'GDP2020':'GDP per capita (US dollars), logscale',
              'Expectancy2020':'Life expectancy (years)'}
# create Plotly plot, add countyr name to tooltip data
fig = px.scatter(data_frame=temp_table,
                 x="GDP2020", y="Expectancy2020", color="Region",
                 hover_data=['Country'], log_x=True,
                 labels = fig_labels)
fig.show()
```

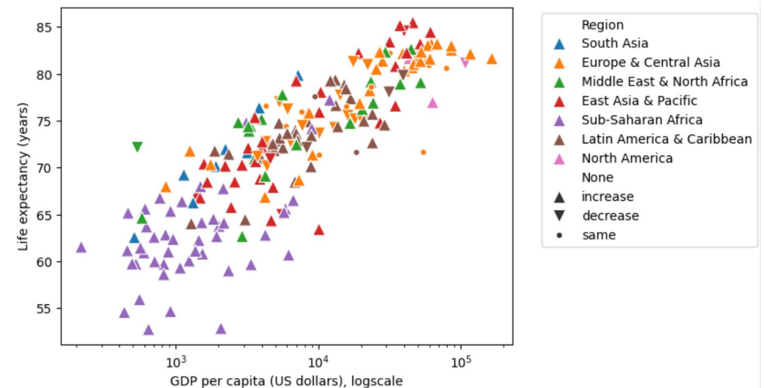
```python
# Seaborn automatically detects if the column used as hue is categorical or numerical
axes = sns.scatterplot(data=countries, x='GDP2020', y='Expectancy2020',
                       hue='Fertility2020')
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
axes.semilogx()
pass
```



```python
# add a column representing population in millions to table countries
countries['Population'] = countries['Population2020'] / 1e6
# create the plot
# parameter sizes sets the minimum and maximum point size to be used
axes = sns.scatterplot(data=countries,
                       x='GDP2020', y='Expectancy2020', hue='Region',
                       size='Population', sizes=(5, 400))
# set titles, log axes and legend location as before
```



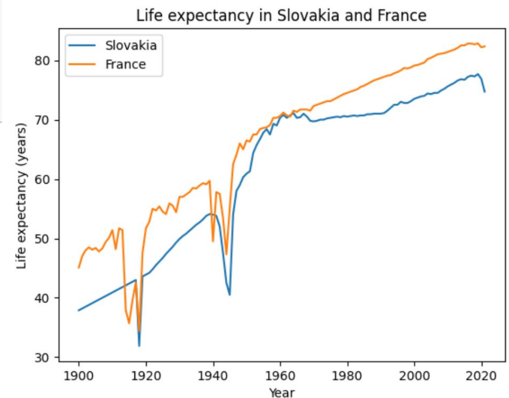
```python
# compute relative differences in population between years 2010 and 2020
diff = (countries.Population2020 - countries.Population2010) / countries.Population2010
# new series with values 'increase', 'decrease' and 'same'
diff_class = diff.apply(lambda x : 'decrease' if x < -0.01
                        else 'increase' if x > 0.01 else 'same')

# create plot
# parameter s sets marker size
axes = sns.scatterplot(data=countries,
                       x='GDP2020', y='Expectancy2020', hue='Region',
                       style=diff_class, s=100,
                       markers={'increase':'^', 'decrease':'v', 'same':'.'})
```



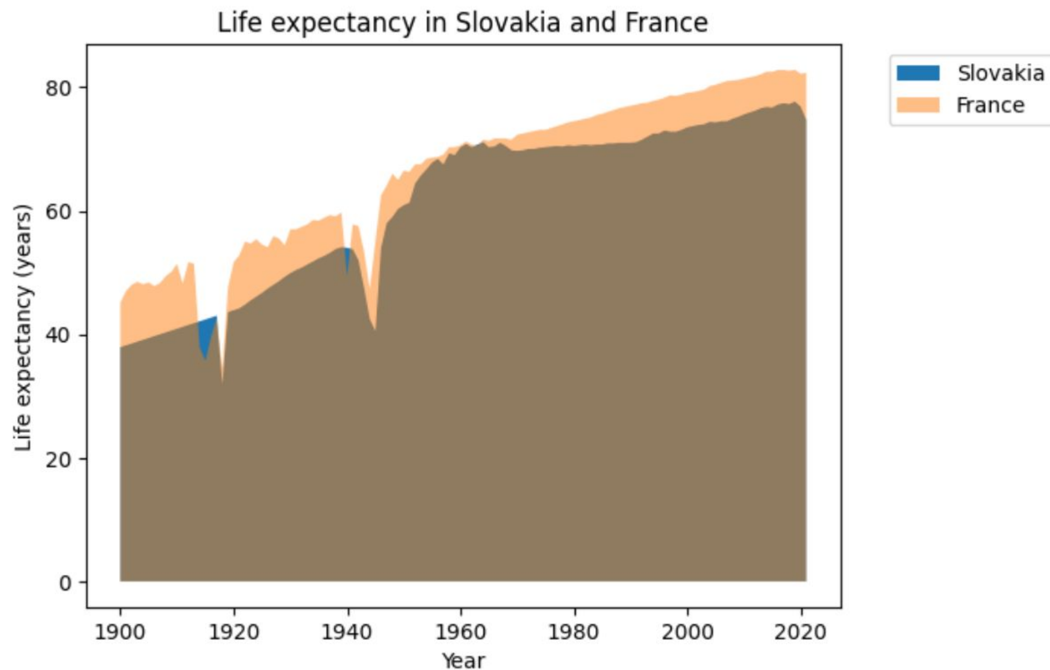
```
display(life_exp_years)
```

	1900	1901	1902	1903	1904	1905	1906	1907	1908	1909	...	2
Country												
Afghanistan	29.4	29.5	29.5	29.6	29.7	29.7	29.8	29.9	29.9	30.0	...	
Albania	35.4	35.4	35.4	35.4	35.4	35.4	35.4	35.4	35.4	35.4	...	
Algeria	30.2	30.3	30.4	31.4	25.4	28.1	29.6	29.5	29.5	31.0	...	
Angola	29.0	29.1	29.2	29.3	29.3	29.4	29.4	29.5	29.6	29.7	...	
Antigua and Barbuda	33.8	33.8	33.8	33.8	33.8	33.8	33.8	33.8	33.8	33.8	...	
...	...	...	...	...	...	...	...	...	...	...	...	

```python
# list of numerical years from column names
years = [int(x) for x in life_exp_years.columns]

figure, axes = plt.subplots()
# plot two lines
axes.plot(years, life_exp_years.loc['Slovak Republic'], label='Slovakia')
axes.plot(years, life_exp_years.loc['France'], label='France')
# plot settings
axes.set_xlabel('Year')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Life expectancy in Slovakia and France')
axes.legend()
pass
```



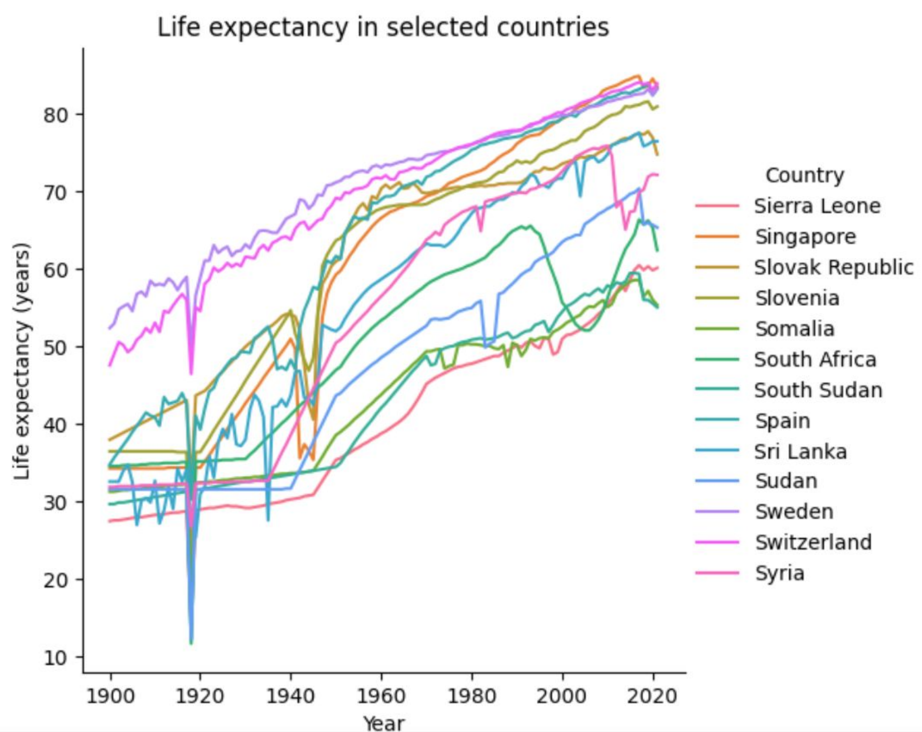
```python
figure, axes = plt.subplots()
# two filled areas, the second is semi-transparent
axes.fill_between(years, 0, life_exp_years.loc['Slovak Republic'], label='Slovakia')
axes.fill_between(years, 0, life_exp_years.loc['France'], label='France', alpha=0.5)
# plot settings as before...
```



```python
# Lines for many countries easy in Seaborn
# ... works better with long table
life_exp_sel_long = (
    life_exp_sel.reset_index()
    .melt(id_vars=['Country'])
    .rename(columns={'variable':'Year', 'value':'Expectancy'})
    .astype({'Year':'int32'})
  )
display(life_exp_sel_long)
```

	Country	Year	Expectancy
0	Sierra Leone	1900	27.400000
1	Singapore	1900	34.200000
2	Slovak Republic	1900	37.900000
3	Slovenia	1900	36.400000
4	Somalia	1900	31.200000
...	...	...	...
1581	Sri Lanka	2021	76.399000
1582	Sudan	2021	65.267000
1583	Sweden	2021	83.156098
1584	Switzerland	2021	83.851220
1585	Syria	2021	72.063000

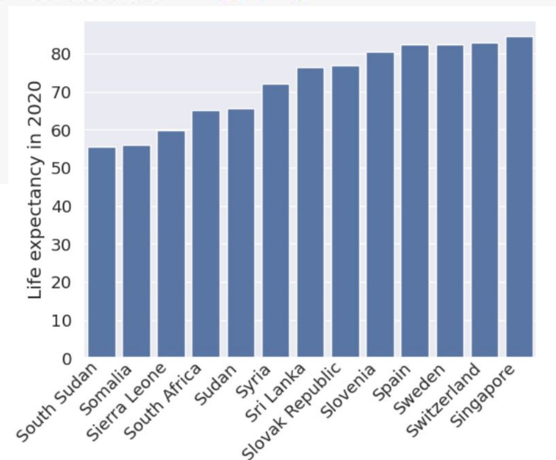
```
grid = sns.relplot(data=life_exp_sel_long, x='Year', y='Expectancy',
                   hue='Country', kind="line")
grid.set_axis_labels('Year', 'Life expectancy (years)')
grid.set(title='Life expectancy in selected countries')
```



```python
sns.set_theme(font_scale=1.2)
# create a grid of small multiple plots
# use col_wrap=5 columns
grid = sns.relplot(data=life_exp_sel_long,
                   x='Year', y='Expectancy', col='Country',
                   col_wrap=5, kind="line", height=3, aspect=1)

grid.set_axis_labels('Year', 'Life expectancy (years)')
grid.set_titles("{col_name}")  # title of each plot will be country name
```

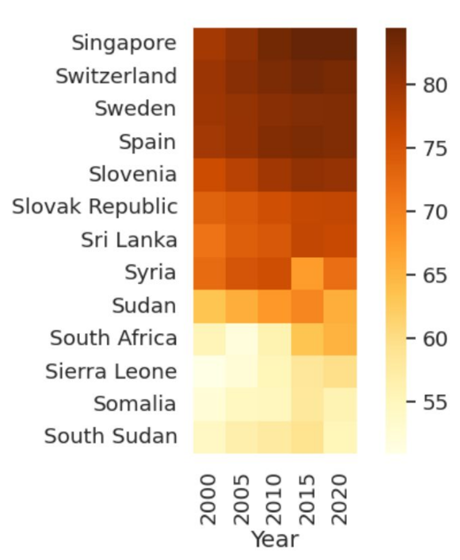
```
def rotate_bar_labels(axes, angle=45):
    """Auxiliary function for rotating bar plot labels by 45 degrees"""
    axes.tick_params(axis='x', labelrotation=angle, pad=-5)
    plt.setp(axes.get_xticklabels(), ha='right')
```

```
# sorting
life_exp_sel_2020_sorted = life_exp_sel_2020.sort_values('Expectancy')
# plotting
axes = sns.barplot(data=life_exp_sel_2020_sorted,
                   x='Country', y='Expectancy', color="C0")
axes.set_ylabel("Life expectancy in 2020")
axes.set_xlabel(None)
rotate_bar_labels(axes)
```



```
# set of years to be used
sel_years={2000, 2005, 2010, 2015, 2020}
# create desired wide table
life_exp_sel_wide = (life_exp_sel_long.query('Year in @sel_years')
                     .pivot(index='Country', columns='Year', values='Expectancy')
                     .sort_values(2020, ascending=False))
# show the table
display(life_exp_sel_wide)
```

```
axes = sns.heatmap(data=life_exp_sel_wide, square=True, cmap="YlOrBr")
```



Year	2000	2005	2010	2015	2020
Country					
Singapore	79.3	81.1	83.2	84.4	84.465854
Switzerland	80.1	81.5	82.5	83.5	83.000000
Sweden	79.8	80.6	81.5	82.2	82.356098
Spain	79.4	80.5	82.0	82.6	82.331707
Slovenia	76.0	77.7	79.5	80.8	80.531707
Slovak Republic	73.5	74.3	75.6	76.7	76.865854

1 Lecture 3b: Source code for plots from Lecture 3a + introduction to Seaborn and Plotly libraries

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

This notebook contains the source code for all the plots shown in the first part of the lecture. In also introduces two new plotting libraries: Seaborn and Plotly.

1.1 Seaborn library

- Seaborn library is an extension of Matplotlib.
- Seaborn is more convenient for many types of plots; we will use it for more complex scatter plots and line plots, for bar plots, strip plots, histograms and heatmaps.
- In Seaborn functions, a whole DataFrame can be added using option `data=`. DataFrame column names are then used as `x`, `y`, `hue` (color), `col` (one of subfigures).
- Seaborn creates Matplotlib objects (e.g. figure, axes) which can be then modified using Matplotlib methods.
- The first example of this library is in section Categorical variable via color

1.2 Plotly library for interactive plots

- Another popular library is Plotly.
- It provides some additional plot types and all plots are interactive.
- For example, in the scatter plot, we can find information about each dot by hovering a mouse over it.
- We can also zoom into parts of the plot by selecting a rectangle.
- A menu with additional options appears in the top right corner of the plot.
- Plotly is also used the first time in section Categorical variable via color.

1.3 Used libraries

[1]:
```
import numpy as np
import pandas as pd
from IPython.display import Markdown
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
```

1.4 Importing World Bank data

Country indicators from World Bank, https://databank.worldbank.org/home under CC BY 4.0 license.

Country population, surface area in km squared, GDP per capita (current US$), life expectancy at birth (years), fertility rate (births per woman); in years 2000, 2010, 2020.

[2]:
```python
url = 'https://bbrejova.github.io/viz/data/World_bank.csv'
countries = pd.read_csv(url).set_index('Country')
display(countries)
```

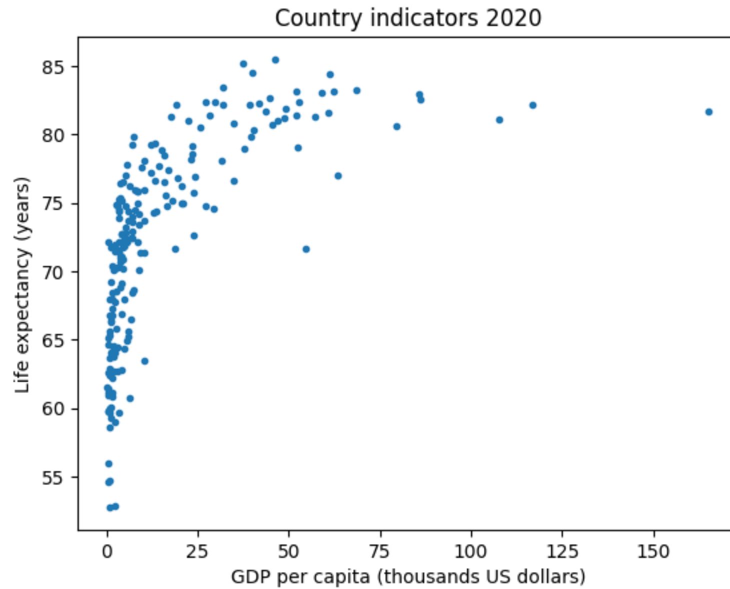
```
                   ISO3                      Region         Income Group  \
Country
Afghanistan         AFG                  South Asia           Low income
Albania             ALB       Europe & Central Asia  Upper middle income
Algeria             DZA  Middle East & North Africa  Lower middle income
American Samoa      ASM         East Asia & Pacific          High income
Andorra             AND       Europe & Central Asia          High income
…                  …                         …                   …
Virgin Islands      VIR   Latin America & Caribbean          High income
West Bank and Gaza  PSE  Middle East & North Africa  Upper middle income
Yemen               YEM  Middle East & North Africa           Low income
Zambia              ZMB          Sub-Saharan Africa  Lower middle income
Zimbabwe            ZWE          Sub-Saharan Africa  Lower middle income

                    Population2000  Population2010  Population2020       Area  \
Country
Afghanistan             19542983.0      28189672.0      38972231.0   652860.0
Albania                  3089026.0       2913021.0       2837849.0    28750.0
Algeria                 30774621.0      35856344.0      43451666.0  2381741.0
American Samoa             58229.0         54849.0         46189.0      200.0
Andorra                    66097.0         71519.0         77699.0      470.0
…                              …             …               …        …
Virgin Islands            108642.0        108356.0        106291.0      350.0
West Bank and Gaza       2922153.0       3786161.0       4803269.0     6020.0
Yemen                   18628701.0      24743945.0      32284046.0   527970.0
Zambia                   9891135.0      13792087.0      18927715.0   752610.0
Zimbabwe                11834676.0      12839770.0      15669667.0   390760.0

                         GDP2000       GDP2010       GDP2020  Expectancy2000  \
Country
Afghanistan                  NaN    562.499219    512.055098       55.298000
Albania              1126.683340   4094.349686   5343.037704       75.404000
Algeria              1780.376063   4495.921476   3354.153164       70.478000
American Samoa               NaN  10446.863206  15609.777220             NaN
Andorra             21620.465102  48237.890541  37207.238871             NaN
…                          …            …            …              …
Virgin Islands               NaN  39905.128418  39411.045254       76.619512
West Bank and Gaza   1476.171850   2557.075624   3233.568638       70.388000
Yemen                 519.591639   1249.063085    578.512010       62.588000
Zambia                364.026145   1469.361450    956.831729       45.231000
Zimbabwe              565.284390    937.840340   1372.696674       44.686000

                    Expectancy2010  Expectancy2020  Fertility2000  \
```

```
Country
Afghanistan            60.851000       62.575000          7.534
Albania                77.936000       76.989000          2.231
Algeria                73.808000       74.453000          2.566
American Samoa               NaN             NaN            NaN
Andorra                      NaN             NaN            NaN
…                            …             …            …
Virgin Islands         77.865854       79.819512          2.060
West Bank and Gaza     73.004000       74.403000          5.443
Yemen                  67.280000       64.650000          6.318
Zambia                 56.799000       62.380000          5.926
Zimbabwe               50.652000       61.124000          3.974

                    Fertility2010  Fertility2020
Country
Afghanistan                 6.099          4.750
Albania                     1.656          1.400
Algeria                     2.843          2.942
American Samoa                NaN            NaN
Andorra                     1.270            NaN
…                           …            …
Virgin Islands              2.300          2.030
West Bank and Gaza          4.383          3.570
Yemen                       4.855          3.886
Zambia                      5.363          4.379
Zimbabwe                    4.025          3.545

[217 rows x 16 columns]
```

1.5 A simple scatterplot

To create a simple scatterplot, commands from the previous lectures suffice. Note that we divide GDP by 1000 and add this information to the axis title. This makes the axis easier to read.

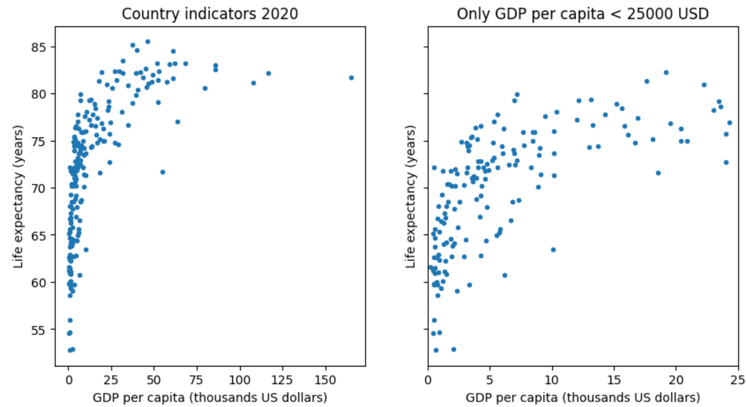
[3]:
```
figure, axes = plt.subplots()
axes.plot(countries.GDP2020 / 1000, countries.Expectancy2020, '.')
axes.set_xlabel('GDP per capita (thousands US dollars)')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
pass
```



1.6 Zooming in

Limits on x axis are set using `set_xlim` method in order to zoom in on countries with lower GDP.

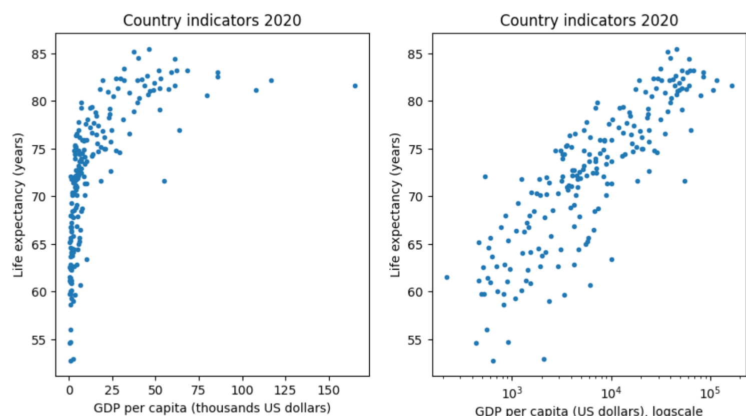
[4]:
```
# create two subplots
figure, axes = plt.subplots(1, 2, figsize=(10, 5), sharey=True)

# the left subplot - full range of data
axes[0].plot(countries.GDP2020 / 1000, countries.Expectancy2020, '.')
axes[0].set_xlabel('GDP per capita (thousands US dollars)')
axes[0].set_ylabel('Life expectancy (years)')
axes[0].set_title('Country indicators 2020')

# the right subplot - smaller values of GDP
axes[1].plot(countries.GDP2020 / 1000, countries.Expectancy2020, '.')
axes[1].set_xlabel('GDP per capita (thousands US dollars)')
axes[1].set_ylabel('Life expectancy (years)')
axes[1].set_title('Only GDP per capita < 25000 USD')
axes[1].set_xlim(0, 25)
pass
```



1.7 Log-scale plot

In this plot, the log-scale on the x-axis is switched on by `semilogx` method; similarly there is `semilogy` for the y-axis and `loglog` for both axes.

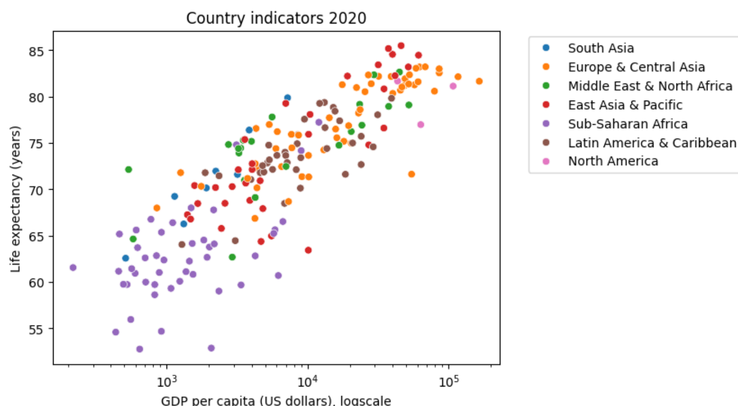
[5]:
```
figure, axes = plt.subplots(1, 2, figsize=(10, 5))

# linear scale plot
axes[0].plot(countries.GDP2020 / 1000, countries.Expectancy2020, '.')
axes[0].set_xlabel('GDP per capita (thousands US dollars)')
axes[0].set_ylabel('Life expectancy (years)')
axes[0].set_title('Country indicators 2020')

# log scale plot
axes[1].plot(countries.GDP2020, countries.Expectancy2020, '.')
axes[1].set_xlabel('GDP per capita (US dollars), logscale')
axes[1].set_ylabel('Life expectancy (years)')
axes[1].set_title('Country indicators 2020')
axes[1].semilogx()
pass
```



1.8 Categorical variable via color

Here we color countries by their region. Seaborn function `scatterplot` can do this easily via `hue` parameter. This function returns Matplotlib axes which can be then modified by familiar methods such as `set_xlabel`.

[6]:
```
# create plot using Seaborn
axes = sns.scatterplot(data=countries, x='GDP2020', y='Expectancy2020',
                       hue='Region')
# set plot properties using methods from Matplotlib
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
axes.semilogx()
# place legend outside the plot:
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```



- The same plot in Plotly is even easier and interactive.
- Both Plotly and Seaborn automatically label axes with column names, such as `GDP2020`.
- Here we override such automated labels with longer ones using a dictionary `fig_labels`.

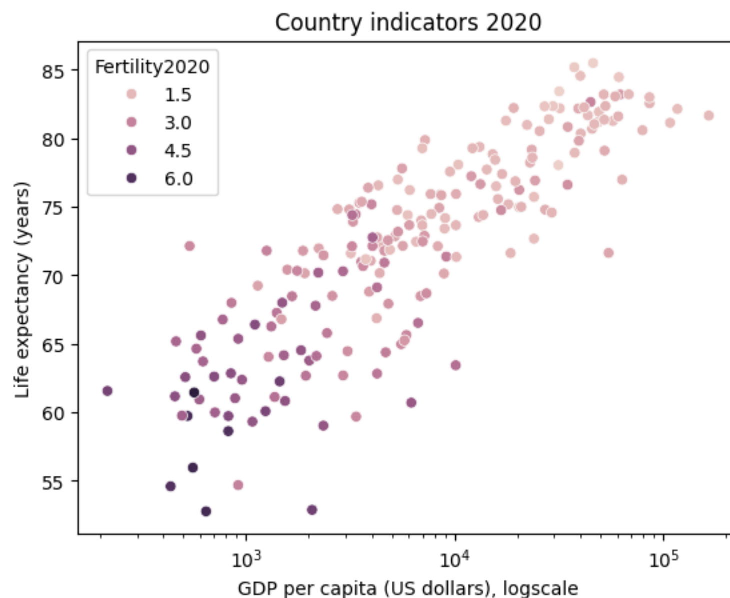
[7]:
```
# we want to use index (country name) in the figure for tooltip info
# therefore we create a temporary table with column Country instead of index
temp_table = countries.reset_index()
# how to rename automated axis labels
fig_labels = {'GDP2020':'GDP per capita (US dollars), logscale',
              'Expectancy2020':'Life expectancy (years)'}
# create Plotly plot, add countyr name to tooltip data
fig = px.scatter(data_frame=temp_table,
                 x="GDP2020", y="Expectancy2020", color="Region",
```

```
                  hover_data=['Country'], log_x=True,
                  labels = fig_labels)
fig.show()
```

1.9 Numerical variable via color

Seaborn automatically detects if the column used as `hue` is a categorical or numerical variable. In the previous graph, regions were used as `hue` and Seaborn chose a color palette with a different color for each category. Here we have a numerical variable so a continuous palette with different shades of pink and purple is used by default. We will discuss color palettes later in the course.

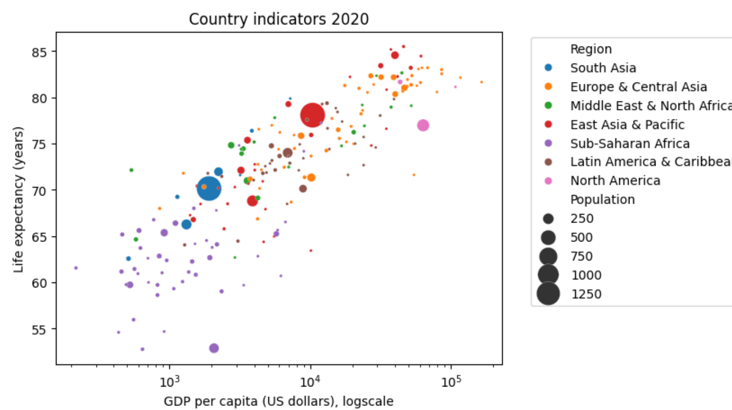
[8]:
```
axes = sns.scatterplot(data=countries, x='GDP2020', y='Expectancy2020',
                        hue='Fertility2020')
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
axes.semilogx()
pass
```



1.10 Numerical variable as point size

We will now use the population of each country as the size of each point (also called bubble), and we will color countries by regions. Sizing points according to the values in a specified table column is again simple to do in Seaborn using paremeter `size` in `sns.scatterplot`. Parameter `sizes` sets the minimum and maximum point size to be used. For simplicity, population in millions is added as a new column to `countries`.

```
[9]: # add a column representing population in millions to table countries
countries['Population'] = countries['Population2020'] / 1e6
# create the plot
axes = sns.scatterplot(data=countries,
                       x='GDP2020', y='Expectancy2020', hue='Region',␣
 ↪size='Population',
                       sizes=(5, 400))
# set various plot properties
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Country indicators 2020')
axes.semilogx()
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```

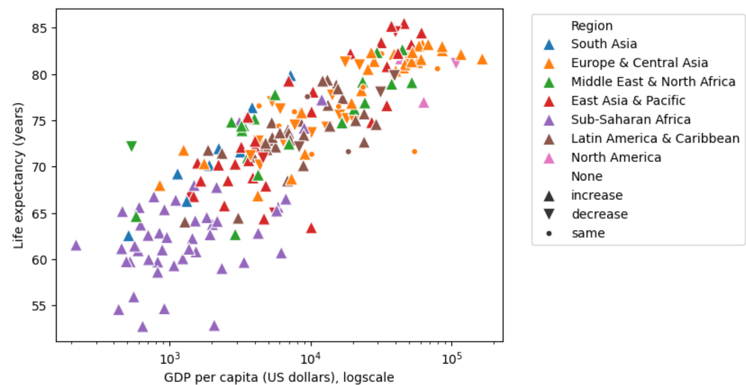


1.11 Categorical variable as marker type

- We add a new column named `Population change` with categories `increase`, `decrease` and `same` depedning on how the population of a country changed between 2010 and 2020. Category `same` is applied to countries with population change less then 1% in either direction.
- This column is created using `apply` command, which applies a function (here a lambda expression) to `diff` Series containing relative change in population.
- This column is then used as argument `style` in `sns.scatterplot`. Size of markers is set to 100 (more than default) by argument `s`. Particular markers are selected by `markers` argument.
- Note that in the `scatterplot` we use both columns of `countries` table and separate Series.

```
[10]: # compute relative differences in population between years 2010 and 2020
diff = (countries.Population2020 - countries.Population2010) / countries.
 ↪Population2010
# new series with values 'increase', 'decrease' and 'same'
diff_class = diff.apply(lambda x : 'decrease' if x < -0.01
                        else 'increase' if x > 0.01 else 'same')
```

```
# create plot
axes = sns.scatterplot(data=countries,
                       x='GDP2020', y='Expectancy2020', hue='Region',
                       style=diff_class, s=100,
                       markers={'increase':'^', 'decrease':'v', 'same':'.'})
# plot settings
axes.semilogx()
axes.set_xlabel('GDP per capita (US dollars), logscale')
axes.set_ylabel('Life expectancy (years)')
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```



1.12 Importing Gapminder life expectancy

We import life expectancy data provided free by the Gapminder foundation under the CC-BY license. The data set gives for each year and each country an estimate of how may years would newborn babies live on average if the trends in mortality of different age groups that were prevailing in the year of their birth would prevail through their entire life.

```
[11]: url="https://bbrejova.github.io/viz/data/life_expectancy_years.csv"
life_exp = pd.read_csv(url, index_col=0)
life_exp_years = life_exp.iloc[:, 1:]
display(life_exp)
```

```
                    ISO3  1900  1901  1902  1903  1904  1905  1906  1907  \
Country
Afghanistan          AFG  29.4  29.5  29.5  29.6  29.7  29.7  29.8  29.9
Albania              ALB  35.4  35.4  35.4  35.4  35.4  35.4  35.4  35.4
Algeria              DZA  30.2  30.3  30.4  31.4  25.4  28.1  29.6  29.5
Angola               AGO  29.0  29.1  29.2  29.3  29.3  29.4  29.4  29.5
Antigua and Barbuda  ATG  33.8  33.8  33.8  33.8  33.8  33.8  33.8  33.8
…                      …     …     …     …     …     …     …     …     …
Venezuela            VEN  32.4  32.4  32.4  32.4  32.4  32.4  32.5  32.5
Vietnam              VNM  31.2  31.1  31.1  31.1  31.1  31.0  31.0  31.0
```

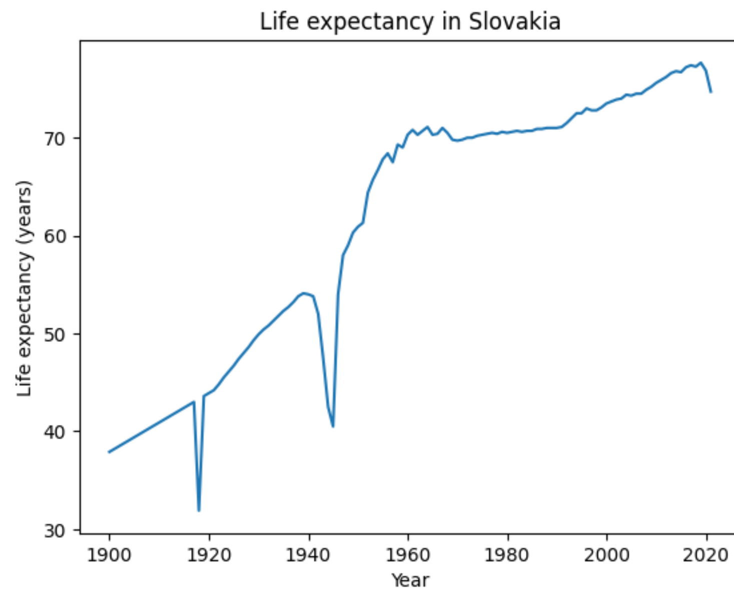
```
Yemen                 YEM  23.5  23.5  23.5  23.5  23.5  23.6  23.6  23.6
Zambia                ZMB  33.6  33.6  33.6  33.7  33.7  33.8  33.8  33.8
Zimbabwe              ZWE  34.1  34.1  34.1  34.1  34.1  34.1  34.1  34.1

                      1908  …  2012  2013  2014  2015  2016  2017    2018  \
Country                     …
Afghanistan           29.9  …  60.8  61.3  61.2  61.2  61.2  63.4  63.081
Albania               35.4  …  77.8  77.9  77.9  78.0  78.1  78.2  79.184
Algeria               29.5  …  76.8  76.9  77.0  77.1  77.4  77.7  76.066
Angola                29.6  …  61.3  61.9  62.8  63.3  63.8  64.2  62.144
Antigua and Barbuda   33.8  …  76.7  76.8  76.8  76.9  77.0  77.0  78.511
…                      …  …    …    …    …    …    …    …      …
Venezuela             32.5  …  75.2  75.2  75.0  75.0  75.3  75.3  71.979
Vietnam               30.9  …  73.8  74.0  74.1  74.3  74.4  74.5  73.976
Yemen                 23.6  …  68.3  68.9  69.0  68.6  68.1  68.1  64.575
Zambia                33.9  …  58.8  60.0  61.1  62.0  62.8  63.2  62.342
Zimbabwe              34.2  …  54.9  56.8  58.5  59.6  60.5  61.4  61.414

                        2019    2020    2021
Country
Afghanistan           63.565  62.575  61.982
Albania               79.282  76.989  76.463
Algeria               76.474  74.453  76.377
Angola                62.448  62.261  61.643
Antigua and Barbuda   78.691  78.841  78.497
…                        …       …       …
Venezuela             72.161  71.095  70.554
Vietnam               74.093  75.378  73.618
Yemen                 65.092  64.650  63.753
Zambia                62.793  62.380  61.223
Zimbabwe              61.292  61.124  59.253

[184 rows x 123 columns]
```

1.13 A simple line graph

Here we use `plot` from matplotlib to plot life expectancy over the years for Slovakia. Years are column names which need to be converted from string to integer using Python list comprehension.

[12]:
```python
# list of numerical years from column names
years = [int(x) for x in life_exp_years.columns]
# simple plot for one row of the table
figure, axes = plt.subplots()
axes.plot(years, life_exp_years.loc['Slovak Republic'])
# plot settings
axes.set_xlabel('Year')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Life expectancy in Slovakia')
```

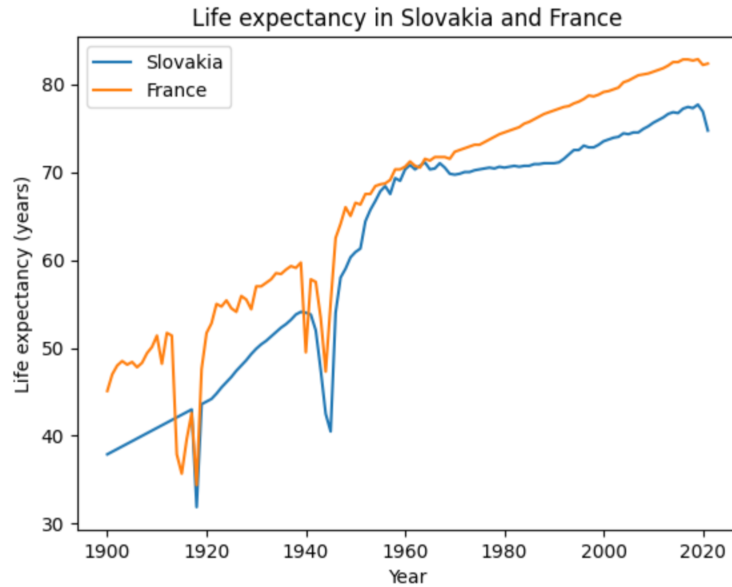
```
pass
```



1.14 A line graph with multiple lines

Here we plot two lines, each by a separate call to `plot`. Each line has a label to show in the legend.

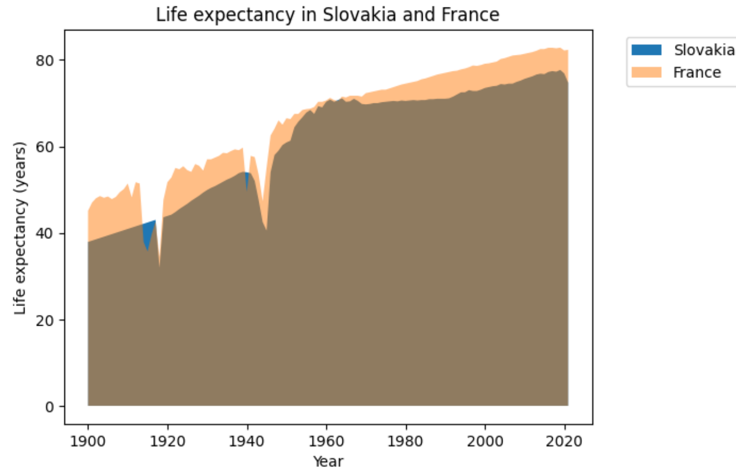
[13]:
```
figure, axes = plt.subplots()
# plot two lines
axes.plot(years, life_exp_years.loc['Slovak Republic'], label='Slovakia')
axes.plot(years, life_exp_years.loc['France'], label='France')
# plot settings
axes.set_xlabel('Year')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Life expectancy in Slovakia and France')
axes.legend()
pass
```



1.15 Area graph

Here we fill in the area between x-axis (value 0) and a table row using `fill_between` method. France is plotted on top and is set to be semi-transparent using `alpha=0.5`.

[14]:
```
figure, axes = plt.subplots()
# two filled areas, the second is semi-transparent
axes.fill_between(years, 0, life_exp_years.loc['Slovak Republic'],␣
 ↪label='Slovakia')
axes.fill_between(years, 0, life_exp_years.loc['France'], label='France',␣
 ↪alpha=0.5)
# plot settings
axes.set_xlabel('Year')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Life expectancy in Slovakia and France')
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```



1.16 Line graph with many lines

- Here we want to plot lines for all countries alphabetically between Si and Sz and having at least million inhabitants.
- First we select such countries from `countries` to table `selection`.
- Using `intersection`, we get only countries from our selection that are also in Gapminder table (`life_exp`).
- Part of the Gapminder table for these coutnries is then stored as `life_exp_sel`.

[15]:
```
selection = countries.query('Population2020 > 1e6 and Country >= "Si" and␣
↪Country <= "Sz"')
life_exp_iso3 = life_exp.reset_index().set_index('ISO3')
life_exp_sel = life_exp_iso3.loc[life_exp_iso3.index.intersection(selection.
↪ISO3), :].set_index('Country')
display(life_exp_sel)
```

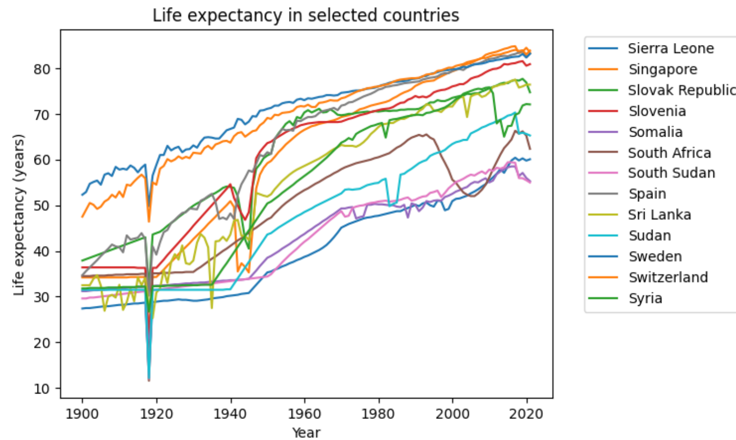
```
                1900  1901  1902  1903  1904  1905  1906  1907  1908  1909  \
Country
Sierra Leone    27.4  27.5  27.5  27.6  27.7  27.8  27.9  27.9  28.0  28.1
Singapore       34.2  34.2  34.2  34.2  34.2  34.2  34.2  34.2  34.2  34.2
Slovak Republic 37.9  38.2  38.5  38.8  39.1  39.4  39.7  40.0  40.3  40.6
Slovenia        36.4  36.4  36.4  36.4  36.4  36.4  36.4  36.4  36.4  36.4
Somalia         31.2  31.2  31.3  31.4  31.4  31.5  31.5  31.6  31.7  31.7
South Africa    34.5  34.5  34.5  34.6  34.6  34.6  34.6  34.7  34.7  34.8
South Sudan     29.6  29.6  29.8  29.8  29.9  30.0  30.1  30.2  30.3  30.4
Spain           34.7  35.6  36.4  37.2  38.0  38.9  39.7  40.5  41.4  41.0
Sri Lanka       32.5  32.5  32.5  33.9  34.7  32.4  26.9  30.0  30.4  29.8
Sudan           31.5  31.5  31.5  31.5  31.5  31.5  31.5  31.5  31.5  31.5
Sweden          52.3  52.9  54.7  55.1  55.4  54.5  56.7  57.0  56.4  58.4
Switzerland     47.5  49.0  50.5  50.1  49.2  49.7  50.8  51.3  52.3  51.7
Syria           31.8  31.8  31.9  31.9  31.9  31.9  31.9  32.0  32.0  32.0
```

```
                  …  2012  2013  2014  2015  2016  2017       2018  \
Country           …
Sierra Leone      …  56.9  57.9  57.1  58.5  59.8  60.4  59.796000
Singapore         …  83.6  83.9  84.2  84.4  84.7  84.8  83.297561
Slovak Republic   …  76.2  76.6  76.8  76.7  77.2  77.4  77.265854
Slovenia          …  79.9  80.2  80.9  80.8  81.0  81.1  81.378049
Somalia           …  56.8  57.4  57.9  58.3  58.5  58.5  56.375000
South Africa      …  59.5  61.1  62.5  63.4  64.4  66.3  65.674000
South Sudan       …  58.2  58.0  58.3  59.4  59.4  59.3  55.950000
Spain             …  82.3  82.6  82.7  82.6  82.9  83.1  83.431707
Sri Lanka         …  76.1  76.4  76.5  76.9  77.2  77.5  75.748000
Sudan             …  68.4  68.7  69.1  69.6  69.8  70.3  65.681000
Sweden            …  81.8  81.9  82.1  82.2  82.4  82.5  82.558537
Switzerland       …  82.9  83.0  83.3  83.5  83.8  84.0  83.753659
Syria             …  67.9  68.7  65.0  67.3  67.4  69.8  70.145000

                      2019       2020       2021
Country
Sierra Leone     60.255000  59.763000  60.062000
Singapore        83.595122  84.465854  83.441463
Slovak Republic  77.665854  76.865854  74.714634
Slovenia         81.529268  80.531707  80.875610
Somalia          57.078000  55.967000  55.280000
South Africa     66.175000  65.252000  62.341000
South Sudan      55.912000  55.480000  54.975000
Spain            83.831707  82.331707  83.178049
Sri Lanka        76.008000  76.393000  76.399000
Sudan            65.876000  65.614000  65.267000
Sweden           83.109756  82.356098  83.156098
Switzerland      83.904878  83.000000  83.851220
Syria            71.822000  72.140000  72.063000

[13 rows x 122 columns]
```

- In Matplotlib, each country from `life_exp_sel` is plotted separately in a for-loop, similarly as for two countries above.
- Note that colors repeat because the default palette is not large enough.

[16]:
```python
figure, axes = plt.subplots()
# loop over countries
for country in life_exp_sel.index:
    axes.plot(years, life_exp_sel.loc[country], label=country)

# plot settings
axes.set_xlabel('Year')
axes.set_ylabel('Life expectancy (years)')
axes.set_title('Life expectancy in selected countries')
```

```
axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```



- To use Seaborn for the same plot, it is better to change `life_exp_sel` table from wide to long format using `melt` method. Year is converted from strings to integers.
- This creates a table with columns Country, Year, Expectancy.

[17]:
```
life_exp_sel_long = (
    life_exp_sel.reset_index()
    .melt(id_vars=['Country'])
    .rename(columns={'variable':'Year', 'value':'Expectancy'})
    .astype({'Year':'int32'})
  )
display(life_exp_sel_long)
```

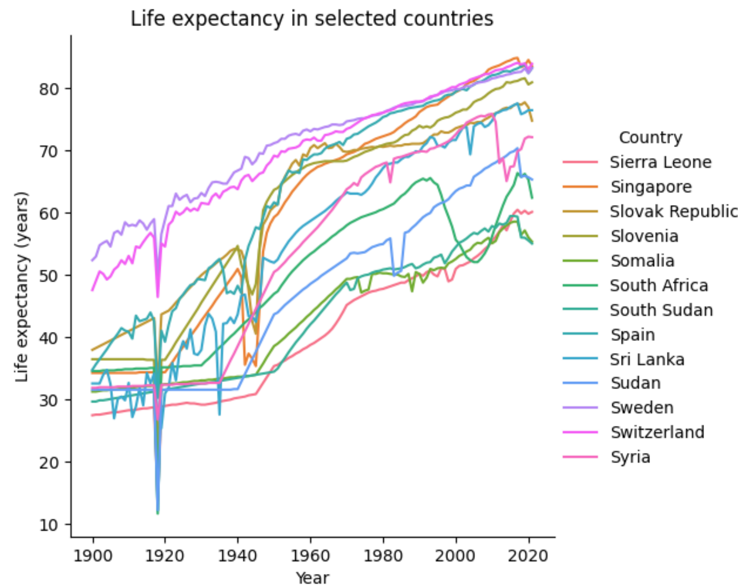
	Country	Year	Expectancy
0	Sierra Leone	1900	27.400000
1	Singapore	1900	34.200000
2	Slovak Republic	1900	37.900000
3	Slovenia	1900	36.400000
4	Somalia	1900	31.200000
…	…	…	…
1581	Sri Lanka	2021	76.399000
1582	Sudan	2021	65.267000
1583	Sweden	2021	83.156098
1584	Switzerland	2021	83.851220
1585	Syria	2021	72.063000

[1586 rows x 3 columns]

- Now we use Seaborn function `relplot`, setting parameters `x`, `y` and `hue` to column names in our long table and specifying that we want lineplit using `kind="line"`.

- The function returns `FacetGrid`, which potentially contains multiple axes, so we ned to use slightly different methods to set labels.
- Seaborn created a sufficiently large color palette but some colors are then hard to distinguish.

[18]:
```
grid = sns.relplot(data=life_exp_sel_long, x='Year', y='Expectancy',
                   hue='Country', kind="line")
grid.set_axis_labels('Year', 'Life expectancy (years)')
grid.set(title='Life expectancy in selected countries')
pass
```



1.17 Small multiples

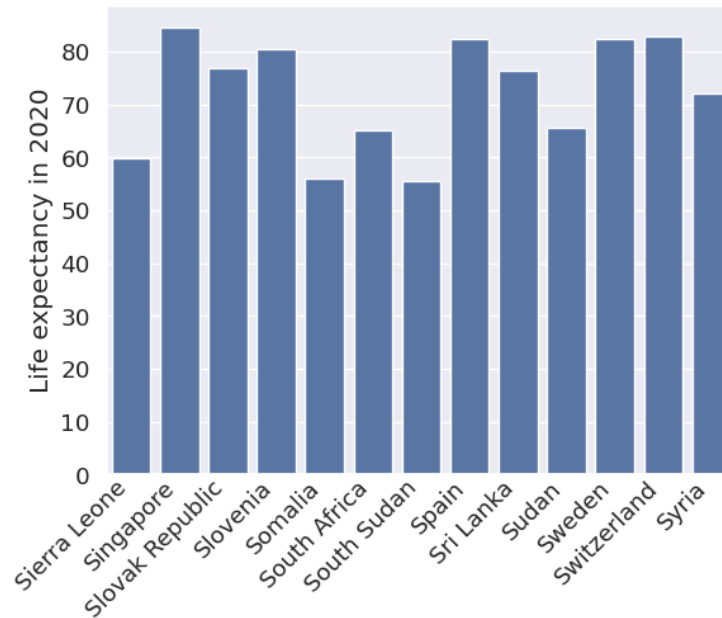
- Small multiples, with each country in our selection as a separate plot, is very easy to do in Seaborn from a long-format table using `relplot`, using column `Country` in option `col` which selects one of subplots for each data point.
- Option `col_wrap` selects how many subplots will be placed ion one row of the overall figure.

[19]:
```
# create grid of small multiple plots
sns.set_theme(font_scale=1.2)
grid = sns.relplot(data=life_exp_sel_long,
                   x='Year', y='Expectancy', col='Country',
                   col_wrap=5, kind="line", height=3, aspect=1)

grid.set_axis_labels('Year', 'Life expectancy (years)')
grid.set_titles("{col_name}")  # title of each plot will be country name
pass
```

1.18 Bar graph

- We plot a bargraph of life expectancy in our selected countries by Seaborn function `barplot`.
- All bars are plotted by the same color using setting `color="C0"`.
- We rotate tick labels on the x axes to fit them in the given space.

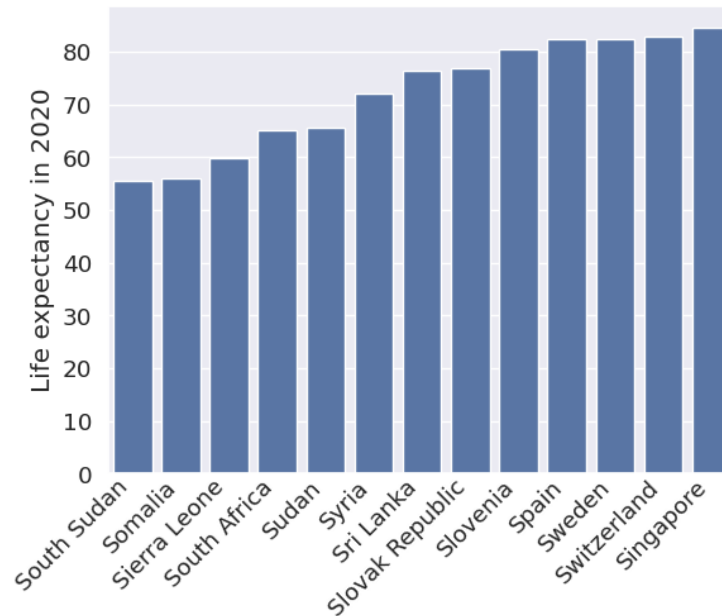
[20]:
```
def rotate_bar_labels(axes, angle=45):
    """Auxiliary function for rotating bar plot labels by 45 degrees"""
    axes.tick_params(axis='x', labelrotation=angle, pad=-5)
    plt.setp(axes.get_xticklabels(), ha='right')


# select one year from the long table
life_exp_sel_2020 = life_exp_sel_long.query('Year==2020')
# create barplot
axes = sns.barplot(data=life_exp_sel_2020,
                   x='Country', y='Expectancy', color="C0")
axes.set_ylabel("Life expectancy in 2020")
axes.set_xlabel(None)
rotate_bar_labels(axes)
pass
```



1.19 Bar graph with sorted columns

Countries are sorted by value in preprocessing, then plotted as before.

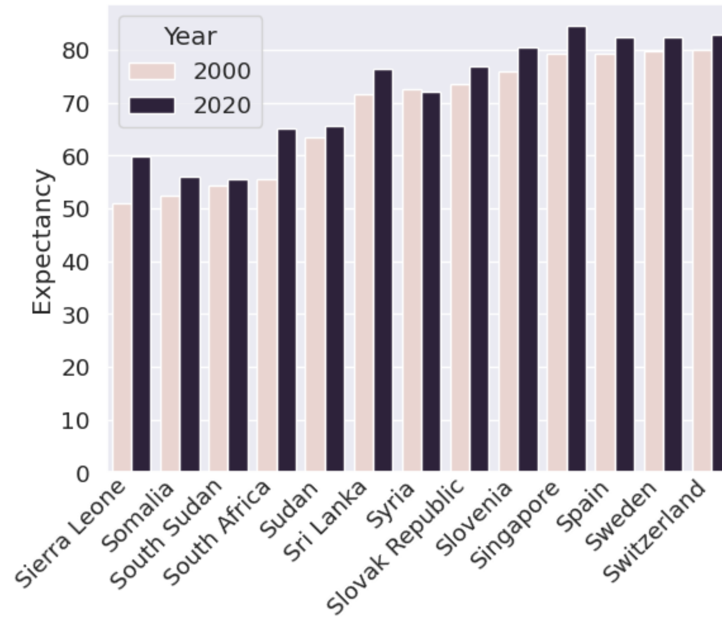
[21]:
```
# sorting
life_exp_sel_2020_sorted = life_exp_sel_2020.sort_values('Expectancy')
# plotting
axes = sns.barplot(data=life_exp_sel_2020_sorted,
                   x='Country', y='Expectancy', color="C0")
axes.set_ylabel("Life expectancy in 2020")
axes.set_xlabel(None)
rotate_bar_labels(axes)
pass
```



1.20 Bar graph with multiple colors

- Now we compare life expectancy in two years in a bargraph with tho colors of columns.
- After selecting appropriate rows of the long table, we use column `Year` in the `hue` parameter of `barplot`.

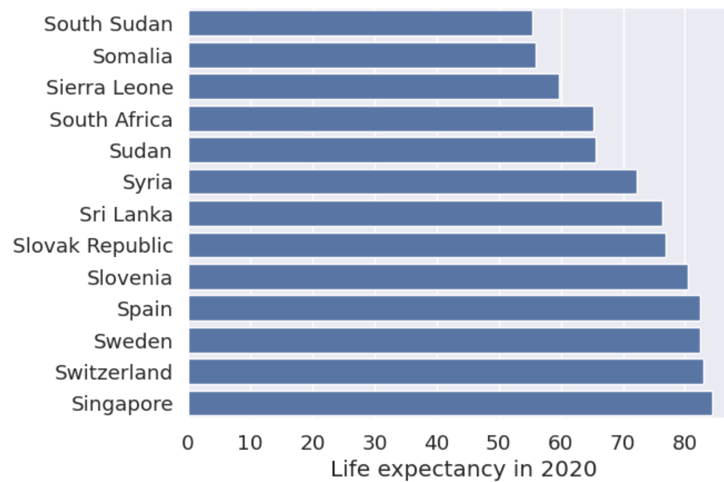
[22]:
```
# select only years 2000 and 2020 from the table, sort
life_exp_sel_comp = life_exp_sel_long.query('Year==2020 or Year==2000').
↪sort_values('Expectancy')
# plotting
axes = sns.barplot(data=life_exp_sel_comp, x='Country', y='Expectancy',␣
↪hue='Year')
axes.set_xlabel(None)
rotate_bar_labels(axes)
pass
```



1.21 Horizontal bar graph

- Longer bar labels are easier to read in a horizontal barplot.
- In Seaborn, it is sufficent to switch `x` and `y` arguments.

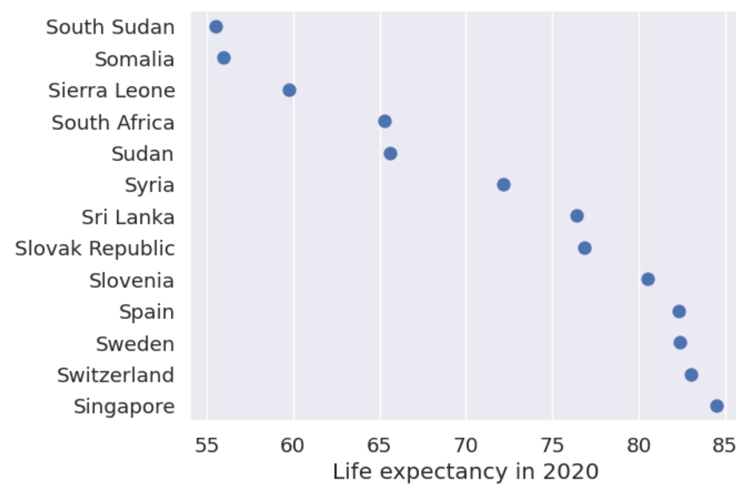
[23]:
```
axes = sns.barplot(data=life_exp_sel_2020_sorted,
                   y='Country', x='Expectancy', color="C0")
axes.set_xlabel("Life expectancy in 2020")
axes.set_ylabel(None)
pass
```



1.22 Dot plot

- Dot plot shows only the end of each bar as a dot.
- Seaborn's pointplot joins these dots by lines by default, `linestyle='none'` prevents this.
- Note that in contrast to barplots, the x axis does not start at 0 (we could make it so by `set_xlim`).

[24]:
```
axes = sns.pointplot(data=life_exp_sel_2020_sorted,
                     y='Country', x='Expectancy',
                     color="C0", linestyle='none')
axes.set_xlabel("Life expectancy in 2020")
axes.set_ylabel(None)
pass
```



1.23 Heatmap

- The goal is to create heatmap with countries as rows, several years as columns and life expectancy values as colors.
- We first need to create a DataFrame with these values in such an arrangment by selecting rows with appropriate years from our long table and pivoting the table by year to make it wide.
- Finally we sort the table by the expectancy in the last year.

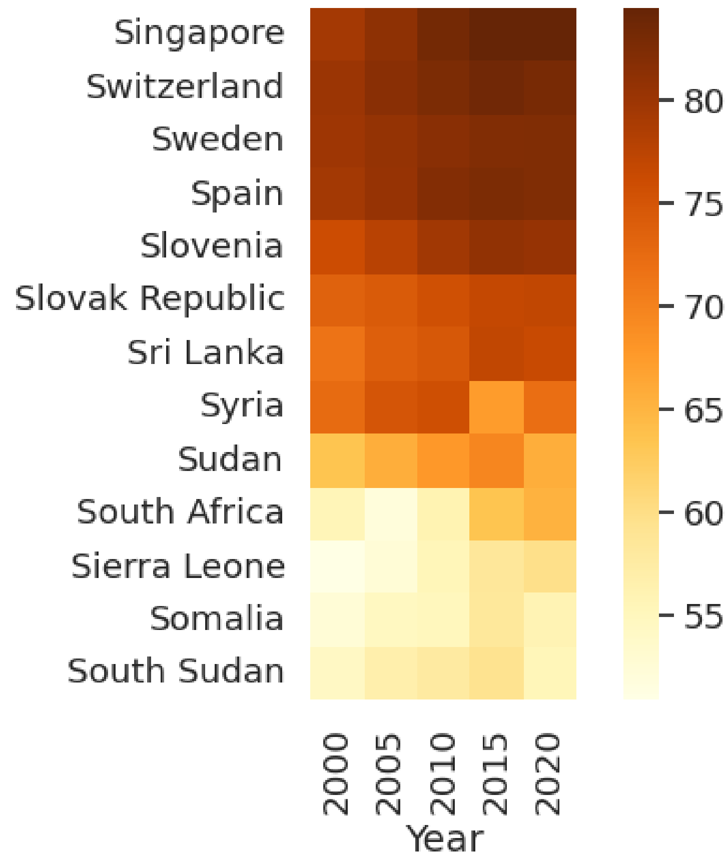
[25]:
```
# set of years to be used
sel_years={2000, 2005, 2010, 2015, 2020}
# create desired wide table
life_exp_sel_wide = (life_exp_sel_long.query('Year in @sel_years')
                     .pivot(index='Country', columns='Year',␣
 ↪values='Expectancy')
                     .sort_values(2020, ascending=False))
# show the table
```

```
display(life_exp_sel_wide)
```

```
Year             2000  2005  2010  2015       2020
Country
Singapore        79.3  81.1  83.2  84.4  84.465854
Switzerland      80.1  81.5  82.5  83.5  83.000000
Sweden           79.8  80.6  81.5  82.2  82.356098
Spain            79.4  80.5  82.0  82.6  82.331707
Slovenia         76.0  77.7  79.5  80.8  80.531707
Slovak Republic  73.5  74.3  75.6  76.7  76.865854
Sri Lanka        71.6  73.8  74.7  76.9  76.393000
Syria            72.5  75.0  75.8  67.3  72.140000
Sudan            63.4  65.7  67.7  69.6  65.614000
South Africa     55.6  52.0  56.1  63.4  65.252000
Sierra Leone     50.9  52.6  55.4  58.5  59.763000
Somalia          52.5  54.7  55.0  58.3  55.967000
South Sudan      54.4  56.7  57.8  59.4  55.480000
```

- Heatmap is plotted by `sns.heatmap` function.
- We have used options to set the shape of individual cells to square and change the color palette ('cmap').

[26]:
```
axes = sns.heatmap(data=life_exp_sel_wide, square=True, cmap="YlOrBr")
axes.set_ylabel(None)
pass
```



1.24 Pie chart

- To prepare data for pie chart, we use two features of Pandas which we will cover in a later lecture: converting the Income Group column to a categorical type and computing the number of countries in various income groups using `groupby`.
- In this way we create two Series: `groups` with counts for the whole world and `groups_asia` for just East Asian countries.

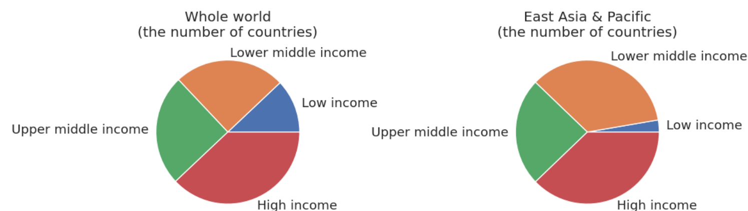
[27]:
```
# creating a categorical type
income_categories = ["Low income", "Lower middle income",
                     "Upper middle income", "High income"]
cat_type = pd.api.types.CategoricalDtype(categories=income_categories,
                                         ordered=True)
# converting Income Group column to cat_type
countries_cat = countries.astype({'Income Group': cat_type})
# aggregation using groupby
groups = countries_cat.groupby('Income Group').size().rename('Count')
# the same but only on countries selected by query
groups_asia = (countries_cat.query('Region=="East Asia & Pacific"')
               .groupby('Income Group').size().rename('Count'))
```

```
display(groups)
```

```
Income Group
Low income             26
Lower middle income    54
Upper middle income    54
High income            82
Name: Count, dtype: int64
```

- The plotting is done by the `pie` function from Matplotlib.
- It gets the series with counts as parameter `x` and country names as `labels`.

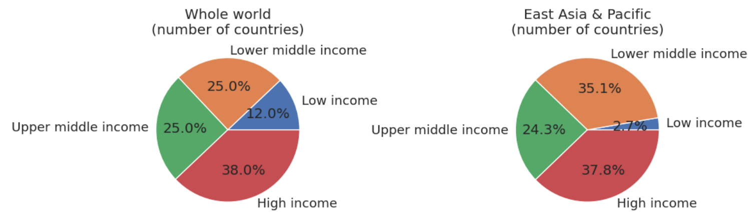
[28]:
```
figure, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].pie(x=groups, labels=groups.index)
axes[0].set_title('Whole world\n(the number of countries)')
axes[1].pie(x=groups_asia, labels=groups_asia.index)
axes[1].set_title('East Asia & Pacific\n(the number of countries)')
figure.subplots_adjust(wspace=1)
pass
```



1.25 Pie chart with labels

- Labels are added by `autopct` setting in `pie`. This setting provides a formatting string for the values, here we print one decimal place.

[29]:
```
figure, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].pie(x=groups, labels=groups.index, autopct="%.1f%%")
axes[0].set_title('Whole world\n(number of countries)')
axes[1].pie(x=groups_asia, labels=groups_asia.index, autopct="%.1f%%")
axes[1].set_title('East Asia & Pacific\n(number of countries)')
figure.subplots_adjust(wspace=1)
pass
```



1.26 Stacked bar graph instead of pie chart

- To prepare data for stacked bar graph, we need to combine our two count Series (`groups` and `groups_asia`) to one long table `groups_concat`.
- This is a DataFrame, because `Income Group` was moved from index to a column.
- We also add percentage column, which will be used in the plot. Percentage is computed by divided counts with the sum of all counts.
- We also add a column with region name, because we will consider two regions (East Asia and the whole world).

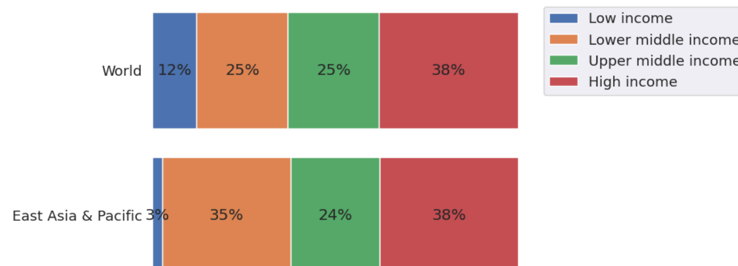
[30]:
```
# first create DataFrame for East Asia
# add Income Group index as a column
temp_asia = groups_asia.reset_index()
# compute percentages and add as a new column
temp_asia['Percentage'] = temp_asia['Count'] * 100 / temp_asia['Count'].sum()
# add Region as a new column, filled with copies of the same string
temp_asia['Region'] = ["East Asia & Pacific"] * len(groups_asia)

# the same three steps for World
temp_world = groups.reset_index()
temp_world['Percentage'] = temp_world['Count'] * 100 / temp_world['Count'].sum()
temp_world['Region'] = ["World"] * len(groups)

# concatenate two DataFrames and display
groups_concat = pd.concat([temp_asia, temp_world], axis=0)
display(groups_concat)
```

	Income Group	Count	Percentage	Region
0	Low income	1	2.702703	East Asia & Pacific
1	Lower middle income	13	35.135135	East Asia & Pacific
2	Upper middle income	9	24.324324	East Asia & Pacific
3	High income	14	37.837838	East Asia & Pacific
0	Low income	26	12.037037	World
1	Lower middle income	54	25.000000	World
2	Upper middle income	54	25.000000	World
3	High income	82	37.962963	World

- Stacked bar graph is not very automated in Matplotib.

- Left coordinate for each rectangle needs to be computed manually, then function `barh` is used (see also tutorial).
- Each bar is labeled with the percentage using `bar_label` function.

[31]:
```
# list of regions and income groups
tmp_regions = groups_concat['Region'].unique()
tmp_groups = groups_concat['Income Group'].unique()
# the first rectangles start at 0
starts = pd.Series([0] * tmp_regions.shape[0])
# create plot
figure, axes = plt.subplots()

# iterate through income groups
for group in tmp_groups:
    # select data for this income group from both regions
    tmp_data = groups_concat.query("`Income Group` == @group")
    # plot
    rectangles = axes.barh(y=tmp_data['Region'], width=tmp_data['Percentage'],␣
↪left=starts, label=group)
    # add labels
    axes.bar_label(rectangles, label_type='center', fmt="%.0f%%")
    # move starts by the size of each rectangle
    starts += tmp_data['Percentage'].reset_index(drop=True)

axes.legend(bbox_to_anchor=(1, 1), loc=2)
# hide plot frame and x-axis ticks
axes.xaxis.set_visible(False)
axes.set_frame_on(False)
pass
```



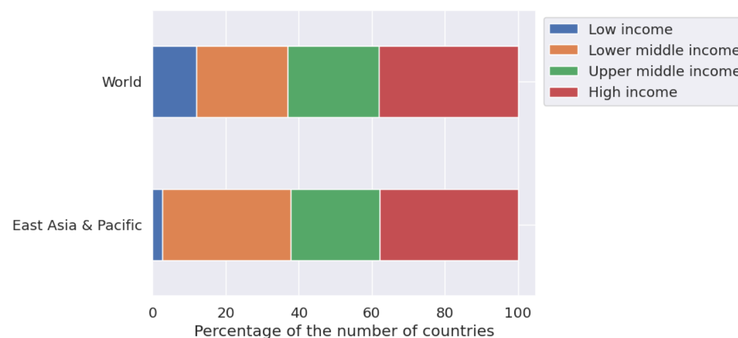
- Stacked bar charts are much easier in Plotly using `px.bar` function.

[32]:
```
fig = px.bar(groups_concat, x="Region", y="Percentage", color="Income Group",
             text="Percentage", text_auto=".0f")
fig.update_layout(font=dict(size=20), xaxis_title=None,)
fig.show()
```

It can also be drawn easily directly by Pandas, but values (sizes of rectangles) are not shown. The table is first converted toa wide form with different income groups as columns.

```
[33]: groups_concat_wide = groups_concat.pivot(columns='Income Group',␣
      ↪index='Region', values='Percentage')
      display(groups_concat_wide)
      axes = groups_concat_wide.plot(kind='barh', stacked=True)
      axes.legend(bbox_to_anchor=(1, 1), loc=2)
      axes.set_ylabel(None)
      axes.set_xlabel('Percentage of the number of countries')
      pass
```

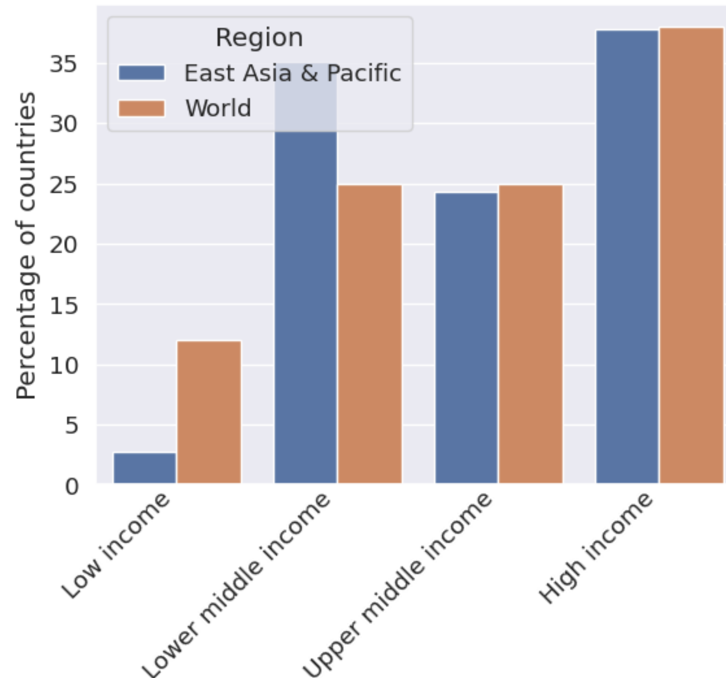
```
Income Group          Low income  Lower middle income  Upper middle income  \
Region
East Asia & Pacific     2.702703            35.135135            24.324324
World                  12.037037            25.000000            25.000000

Income Group          High income
Region
East Asia & Pacific     37.837838
World                   37.962963
```



1.27 Colored bar graphs insteads of pie chart

- As we have seen before, colored bar graphs are easy in Seaborn from a long table.
- Therefore we use `groups_concat` DataFrame.

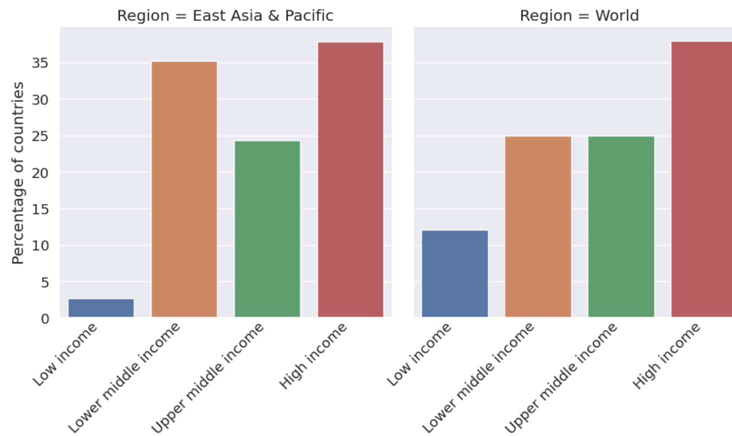
```
[34]: axes = sns.barplot(data=groups_concat,
                         x='Income Group', y='Percentage', hue='Region')
      rotate_bar_labels(axes)
      axes.set_xlabel(None)
      axes.set_ylabel("Percentage of countries")
      pass
```



1.28 Multiple bar graphs instead of pie chart

- In the next plot a separte bar graph for each region.
- This is also very simple in Seaborn using `catplot` with setting `col='Region'` and `kind='bar'`.
- Labels are rotated in each subplot using a for-loop.

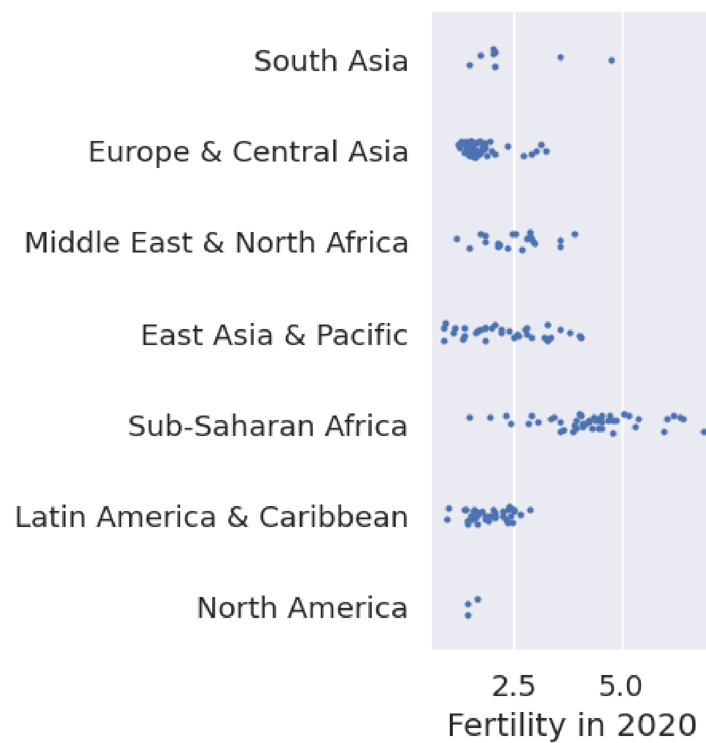
[35]:
```
grid = sns.catplot(kind='bar', data=groups_concat,
                   x='Income Group', y='Percentage',
                   col='Region', hue='Income Group')

# label rotation
for which in [0,1]:
  rotate_bar_labels(grid.axes[0,which])
  grid.axes[0,which].set_xlabel(None)
  grid.axes[0,which].set_ylabel("Percentage of countries")
pass
```



1.29 Strip plot

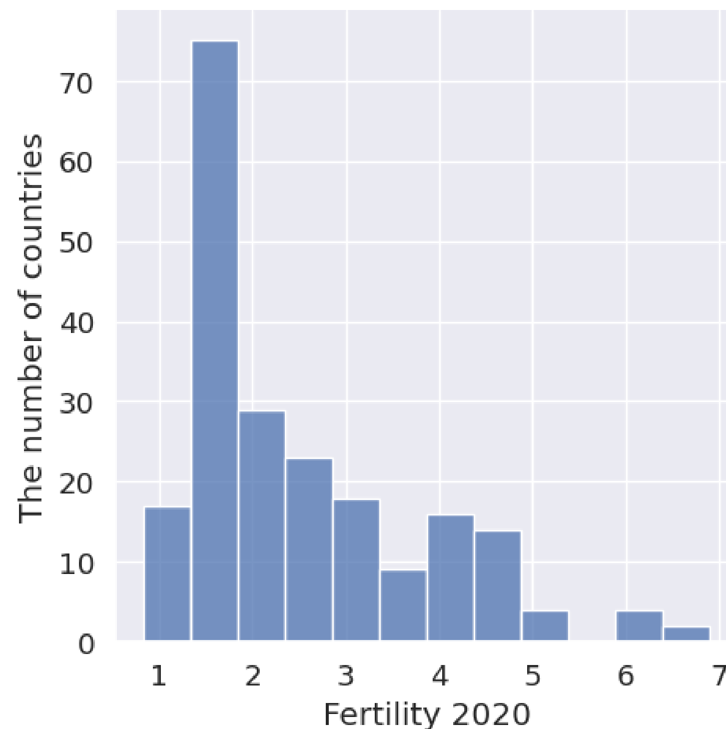
- Strip plot of fertility per region is also very simple in `sns.catplot`.
- Setting `kind='strip'` is default for `catplot`, so it is omitted here.
- Size of dots is reduced to limit overlapping markers.

[36]:
```
grid = sns.catplot(x="Fertility2020", y="Region", data=countries, size=3)
grid.set_axis_labels("Fertility in 2020", "")
pass
```



1.30 Histogram

[37]:
```
grid = sns.displot(countries, x="Fertility2020", binwidth=0.5)
grid.set_axis_labels("Fertility 2020", "The number of countries")
pass
```



1.31 Parallel coordinates

- We want to display varipus properties of individual countries as parallel coordinate plot.
- We first create table `for_parallel` with selected columns and express all numbers as percentage of the maximum value.
- We add `selected` column which has True in row for Slovakia and False elsewhere. This is used to highlight Slovakia in the plot.
- Also ordering is changed to draw Slovakia the last.

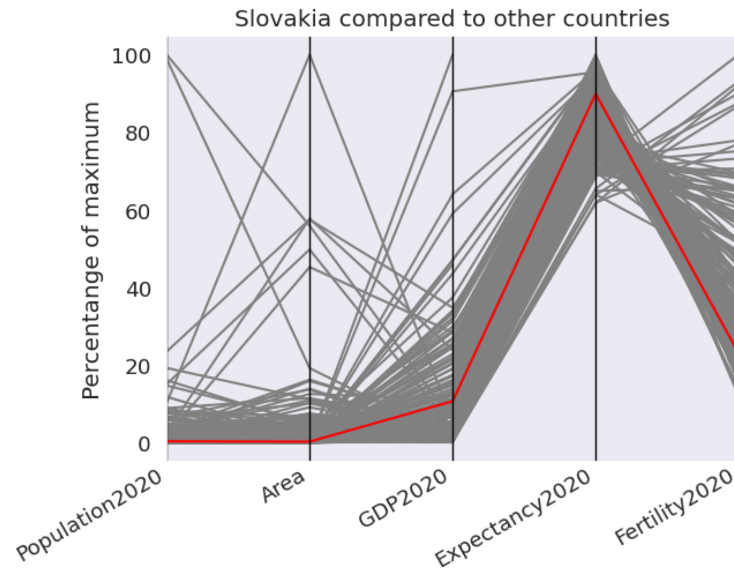
[38]:
```
# selecting columns
for_parallel_sel = countries.loc[:, ['Population2020', 'Area', 'GDP2020',
                                     'Expectancy2020', 'Fertility2020']]
# computing maximum in each column
for_parallel_max = for_parallel_sel.max(axis=0)
# dividing values by the maximum and multiplying by 100 to get percentage
for_parallel = for_parallel_sel.div(for_parallel_max, axis=1) * 100
```

```
# creating column of booleans called 'selected' which highlights Slovakia with␣
↪True
for_parallel['selected'] = countries.index=="Slovak Republic"
# sort by 'selected' to put Slovakia last
for_parallel.sort_values('selected', inplace=True)
# show end of the table
display(for_parallel.tail())
```

```
                 Population2020      Area    GDP2020  Expectancy2020  \
Country
Greece                 0.758174  0.771775   9.651333       95.076168
Greenland              0.003995  2.400538  29.962671       83.788155
Fiji                   0.065227  0.106853   2.638193       79.445541
Zimbabwe               1.110458  2.285380   0.752009       71.492098
Slovak Republic        0.386849  0.286754  10.711280       89.904148

                 Fertility2020  selected
Country
Greece               20.168311     False
Greenland            29.294835     False
Fiji                 36.201393     False
Zimbabwe             51.436448     False
Slovak Republic      23.070226      True
```

- Parallel coordinates are drawn using Pandas `parallel_coordinates` function, which internally calls Matplotlib and returns Axes object.

[39]:
```
axes = pd.plotting.parallel_coordinates(for_parallel, class_column='selected',
                                        color=['gray', 'red'])
axes.get_legend().remove()
axes.set_ylabel("Percentange of maximum")
axes.set_title("Slovakia compared to other countries")
rotate_bar_labels(axes, angle=30)
pass
```



1.32 Parallel categories

- We will use two categorical columns from the countries table, but more categorical columns could be easily added.
- We use the version of the table with a categorical income groups and sort countries by income.
- Now we use `parallel-categories` function from Plotly.
- This function orders each column of the figure by size. By calling `update_traces`, we reorder the first column by the same order as they first occur in our table.

[40]:
```
for_parallel_cat = (countries_cat.loc[:,['Income Group','Region']]
                    .sort_values('Income Group', ascending=False))
fig = px.parallel_categories(for_parallel_cat, width=800)
fig.update_traces(dimensions=[{"categoryorder": "array"}, {}])
fig.update_layout(font_size=20)
fig.update_layout(margin={'l':200,'r':200})
fig.show()
```

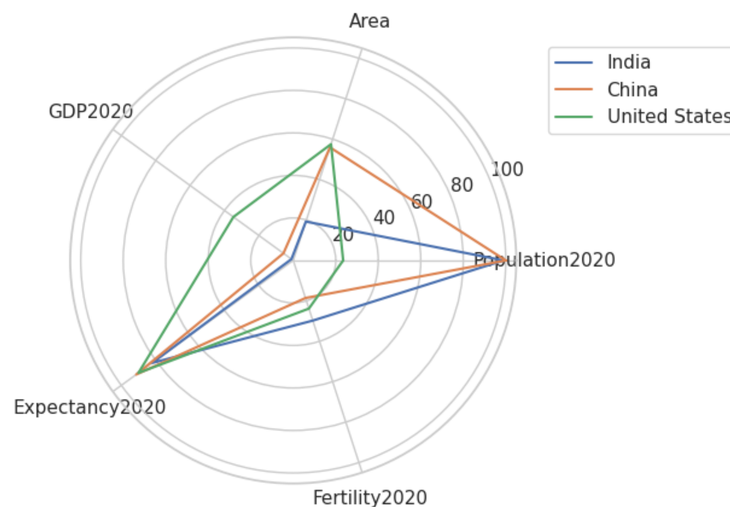
1.33 Radar chart

- Radar charts are not well supported in any of the used libraries.
- Below we compute angles of each axis manually, then use `plot` from Matplotlib.
- When creating axes, we specify polar coordinates `subplot_kw={'projection': 'polar'}`.
- We also use `set_thetagrids`

[41]:
```
# skip 'selected' column, use only 3 countries
for_radar = for_parallel.loc[['India','China','United States'], :].iloc[:, 0:-1]
display(for_radar.head())
```

```
# setup plot with polar coordinates
sns.set_theme(style="whitegrid")
figure, axes = plt.subplots(subplot_kw={'projection': 'polar'})
categories = list(for_radar.columns)
import math
angles = [ i * 2 * math.pi / len(categories) for i in range(len(categories))]
angles_deg = [x / math.pi * 180 for x in angles]
axes.set_thetagrids(angles_deg, labels=categories)

# for plotting, we will need to return to starting point in each line
angles.append(angles[0])

# for each country create list of values, add the starting point, plot
for country in for_radar.index:
    values = list(for_radar.loc[country, :])
    values.append(values[0])
    axes.plot(angles, values, label=country)

axes.legend(bbox_to_anchor=(1.05, 1), loc=2)
pass
```

```
               Population2020       Area     GDP2020  Expectancy2020  \
Country
India               98.957347  19.225710   1.048125       82.049124
China              100.000000  55.929174   5.702240       91.320734
United States       23.493127  57.500095  34.803081       90.038227

               Fertility2020
Country
India              29.759141
China              18.586767
United States      23.817470
```



Lecture 4
Summary statistics

[Data visualization · 1-DAV-105]

Lecture by Broňa Brejová

More details in the [notebook version]

Introduction

Summary statistics (popisné charakteristiky / štatistiky):
quantities that summarize basic properties of a single variable (a table column), such as the mean.

We can also characterize dependencies between pairs of variables.

Together with simple plots, they give us the first glimpse at the data when working with a new data set.

Data set for today

- The same data set as in group tasks 04.
- The data set describes 2049 movies.
- Originally downloaded from https://www.kaggle.com/rounakbanik/the-movies-dataset and preprocessed, keeping only movies with at least 500 viewer votes.

```
url = 'https://bbrejova.github.io/viz/data/Movies_small.csv'
movies = pd.read_csv(url)
display(movies.head())
```

title	year	budget	revenue	original_language	runtime	release_date	vote_average	vote_count	overview
Toy Story	1995	30000000.0	373554033.0	en	81.0	1995-10-30	7.7	5415.0	Led by Woody, Andy's toys live happily in his ...
Jumanji	1995	65000000.0	262797249.0	en	104.0	1995-12-15	6.9	2413.0	When siblings Judy and Peter discover an encha...
Heat	1995	60000000.0	187436818.0	en	170.0	1995-12-15	7.7	1886.0	Obsessive master thief, Neil McCauley leads a ...
GoldenEye	1995	58000000.0	352194034.0	en	130.0	1995-11-16	6.6	1194.0	James Bond must unmask the mysterious head of ...
Casino	1995	52000000.0	116112375.0	en	178.0	1995-11-22	7.8	1343.0	The life of the gambling paradise – Las Vegas ...

Measures of central tendency (miery stredu / polohy)

These represent a **typical value** in a sample $x = x_1,\ldots,x_n$ (one numerical column).

Mean (priemer)

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

This is the arithmetic mean, there are also geometric and harmonic means.

Median (medián) is the middle value when the values ordered by size.

For even n usually defined as the average of the two middle values.

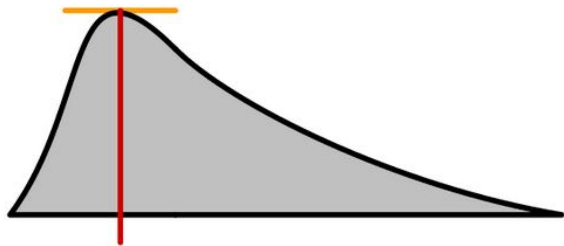
Example:

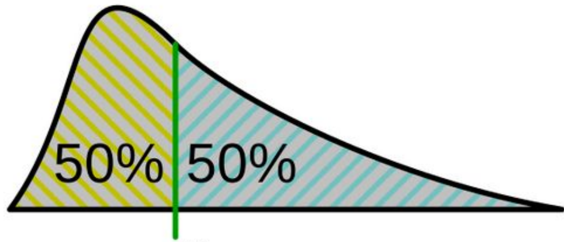
Median of 10, 12, 15, 16, 16 is 15.

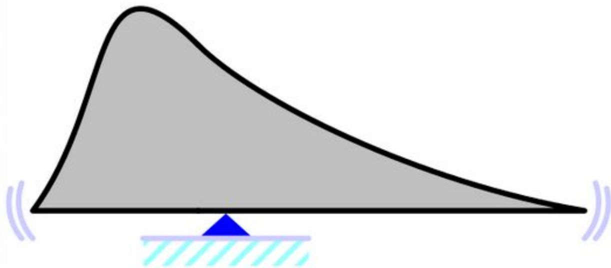
Median of 10, 12, 15, 16, 16, 20 is 15.5.

Measures of central tendency (cont.)

- **Mean (priemer)**
- **Median (medián)**
- **Mode (modus)** is the most frequent value (for a discrete variable).
  Mode of 10,12,15,16,16 is 16.
  For continuous variables, we may look for a mode in a histogram.
  This is sensitive to bin size.







https://commons.wikimedia.org/wiki/File:Visualisation_mode_median_mean.svg

Properties of the measures

If we apply **linear transformation** $a \cdot x_i + b$ with the same a and b to all x_i, mean, median and mode will be transformed **in the same way**.

This corresponds e.g. to the change in the units of measurement (grams vs kilograms, degrees C vs degrees F)

Mean can be heavily influenced by outliers
800, 1000, 1100, 1200, 1800, 2000, 30000: mean 5414.3, median 1200
800, 1000, 1100, 1200, 1800, 2000, 10000: mean 2557.1, median 1200

Therefore we often **prefer median** (e.g. median salary).

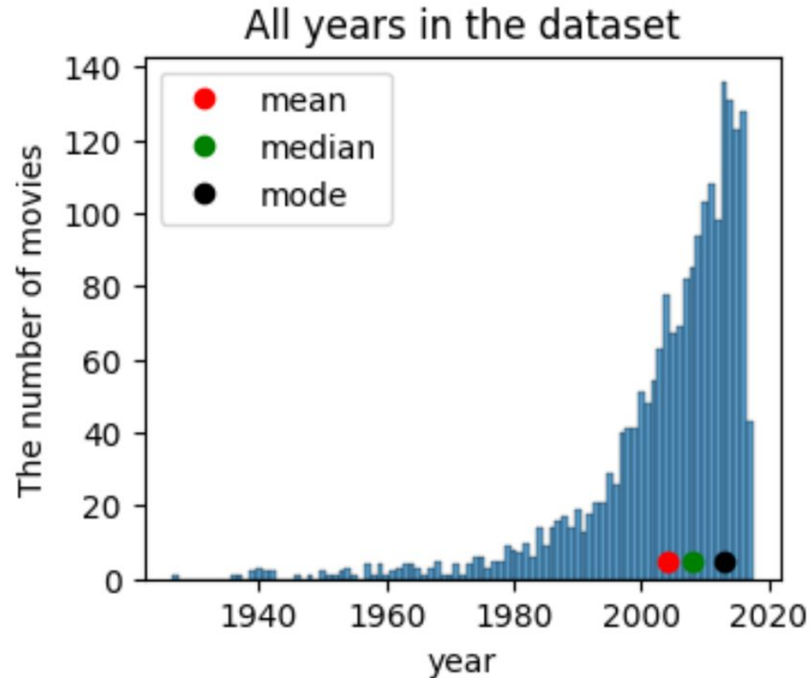
Computation in Pandas

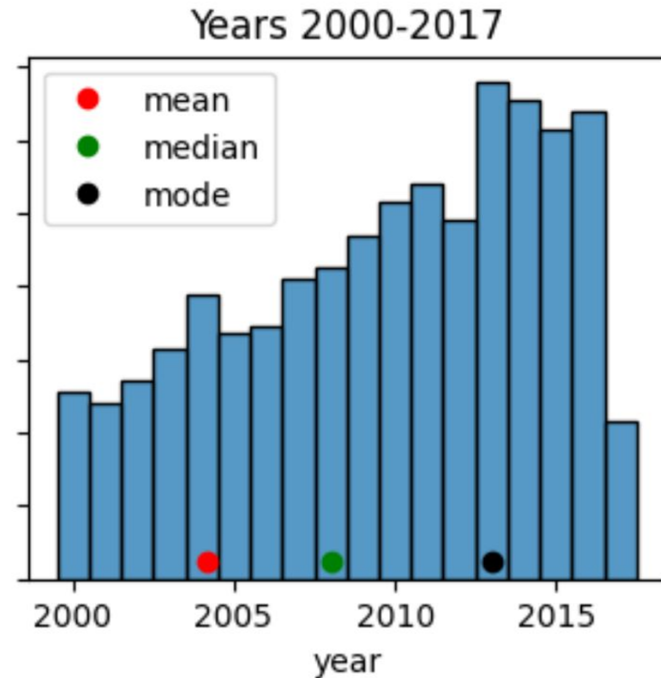
```
display(Markdown("**Properties of the column `year` in our table:**"))
print(f"Mean: {movies['year'].mean():.2f}")
print(f"Median: {movies['year'].median()}")
print(f"Mode:\n{movies['year'].mode()}")
```

Properties of the column year in our table:

```
Mean: 2004.14
Median: 2008.0
Mode:
0    2013
Name: year, dtype: int64
```

Shown in a histogram (whole and detail)





What causes the difference between the mean and the median?

Summarizing many columns or rows

- Functions `mean` and `median` can be applied to all numerical columns
- With `axis=1` we get means or medians in rows

```
display(movies.mean(numeric_only=True))
```

```
year                 2,004.1
budget          55,108,939.7
revenue        198,565,134.3
runtime                112.7
vote_average             6.6
vote_count           1,704.6
dtype: float64
```

Quantiles, percentiles and quartiles (kvantily, percentily, kvartily)

Median is the middle value in a sorted order;
about 50% of values are smaller and 50% larger.

For a percentage p, the $\boldsymbol{p}$**-th percentile** is at position roughly $(p/100) \cdot n$ in the sorted order of values.

Similarly **quantile** (in Pandas), but we give fraction between 0 and 1 rather than percentage.

Quartiles are three values Q_1, Q_2 and Q_3 that split input data into quarters.
Therefore, Q_2 is the median.

Quantiles in Pandas

```
display(Markdown("**Median:**"),
        movies['year'].median())
display(Markdown("**Quantile for 0.5:**"),
        movies['year'].quantile(0.5))
display(Markdown("**All quartiles:**"),
        movies['year'].quantile([0.25, 0.5, 0.75]))
display(Markdown("**With step 0.1:**"),
        movies['year'].quantile(np.arange(0.1, 1, 0.1)))
```

Median:

```
2008.0
```

Quantile for 0.5:

```
2008.0
```

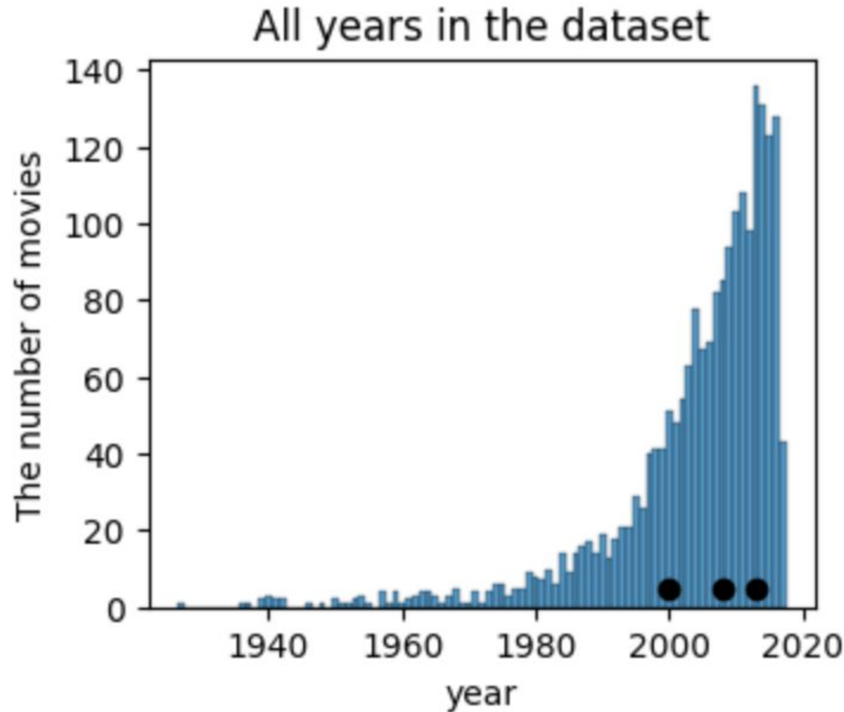
All quartiles:

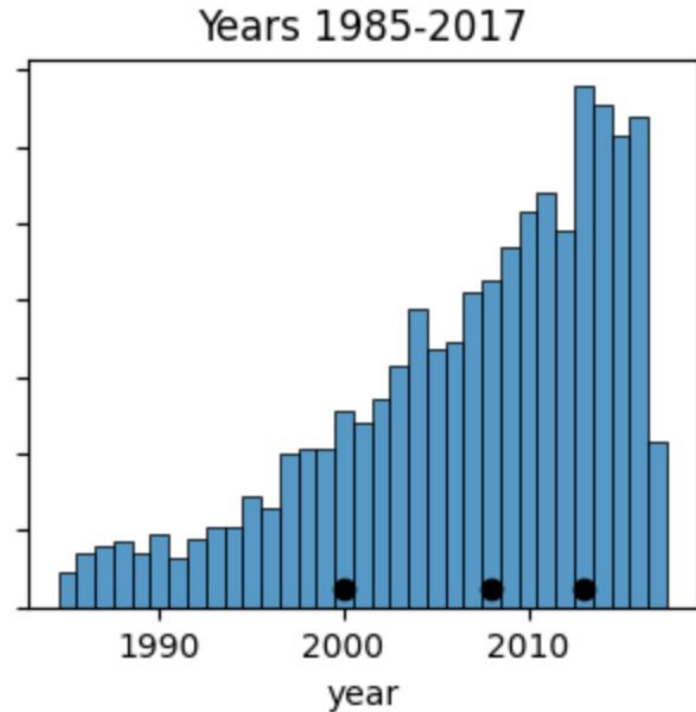
```
0.25    2,000.00
0.50    2,008.00
0.75    2,013.00
Name: year, dtype: float64
```

With step 0.1:

```
0.10    1,988.80
0.20    1,998.00
0.30    2,002.00
0.40    2,005.00
0.50    2,008.00
0.60    2,010.00
0.70    2,012.00
0.80    2,014.00
0.90    2,015.00
Name: year, dtype: float64
```

Quartiles in a histogram (whole and detail)





Quantile interpolation

Optional parameter `interpolation` accepts values `'linear'` (default), `'lower'`, `'higher'`, `'midpoint'`, `'nearest'`.

Minimum is at quantile 0, maximum at quantile 1, the rest evenly spaced between.

The quantile between two elements is influenced only by its two neighbors.

Example: list [0,10,20,100]
p=0: 0, p=1/3: 10, p=2/3: 20, p=1: 100
p=1/4: by default linear interpolation at 3/4 between 0 and 10, i.e. 7.5.

Linear interpolation is continuous as p changes from 0 to 1.

Quantile interpolation

Quantiles for [0, 10, 20, 100]

```
0.01     0.30
0.25     7.50
0.50    15.00
0.75    40.00
dtype: float64
```

Quantiles for [0, 10, 20, 100] with `interpolation='lower'`

```
0.01      0
0.25      0
0.50     10
0.75     20
dtype: int64
```

Measures of variability (miery variability)

Values in the sample may be close to central values or spread widely.

Examples of measures:

Range of values from **minimum** to **maximum** (sensitive to outliers).

Interquartile range IQR (kvartilové rozpätie): range between Q_1 and Q_3 (contains the middle half of the data).

Variance and standard deviation (next)

Variance (rozptyl)

For each x_i square its difference from the mean

Squaring gives non-negative values (and squares are easier to work with mathematically than absolute values).

Variance is the mean of these squares, but we divide by $n-1$ rather than n:

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}$$

Division by n would underestimate the true variance of the underlying population (more in the statistics course).

Standard deviation (smerodajná odchýlka)

Square root of the variance

$$s = \sqrt{s^2}$$

It is in the same units as the original values
(variance is in units squared).

Recall:

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n - 1}$$

Properties

Larger variance and standard deviation mean that data are spread farther from the mean.

If we apply **linear transformation** $a \cdot x_i + b$ with the a and b to all x_i:
Neither variance nor standard deviation change with b.
Variance is multiplied by a^2, standard deviation by $|a|$.

These measures are strongly **influenced by outliers**:
800, 1000, 1100, 1200, 1800, 2000, 30000: st. dev. 10850.0, IQR 850
800, 1000, 1100, 1200, 1800, 2000, 10000: st. dev. 3310.5, IQR 850.

```python
display(Markdown("**Minimum**"),
        movies['year'].min())
display(Markdown("**Maximum**"),
        movies['year'].max())
display(Markdown("**Mean**"),
        movies['year'].mean())
display(Markdown("**Variance**"),
        movies['year'].var())
display(Markdown("**Standard deviation**"),
        movies['year'].std())
q1 = movies['year'].quantile(0.25)
q3 = movies['year'].quantile(0.75)
display(Markdown("**Q1, Q3 and IQR:**"),
        q1, q3, q3-q1)
```

Minimum

```
1927
```

Maximum

```
2017
```

Mean

```
2004.1449487554905
```

Variance

```
161.2714600681735
```

Standard deviation

```
12.699270060447313
```

Q1, Q3 and IQR:

```
2000.0
2013.0
13.0
```

Outliers (odľahlé hodnoty)

Outliers are the values which are far from the typical range of values.

In data analysis, it is important to **check outliers.**

If they represent **errors**, we may try to correct or remove them.

They can also represent **interesting anomalies.**

Different definitions of outliers may be appropriate in different situations.

One possible definition of outliers

The criterion by statistician John Tukey:

Outliers are the values outside of the range $[Q_1 - k \cdot IQR, Q_3 + k \cdot IQR]$, e.g. for $k = 1.5$.

In our example 800, 1000, 1100, 1200, 1800, 2000, 30000:
$Q_1 = 1050$, $Q_3 = 1900$, $IQR = 850$.
$Q_1 - 1.5 \cdot IQR = -225$, $Q_3 + 1.5 \cdot IQR = 3175$.

Outliers are values smaller than −225 or larger than 3175; here only 30000.

The range of outliers is not influenced if we change the outliers.

Computation in Pandas

```
# get quartiles and iqr
q1 = movies['year'].quantile(0.25)
q3 = movies['year'].quantile(0.75)
iqr = q3 - q1
# compute thresholds for outliers
lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr
# count outliers
count = movies.query('year < @lower or year > @upper')['year'].count()
```

Outliers outside of range: [1980.5, 2032.5]

Outlier count: 112

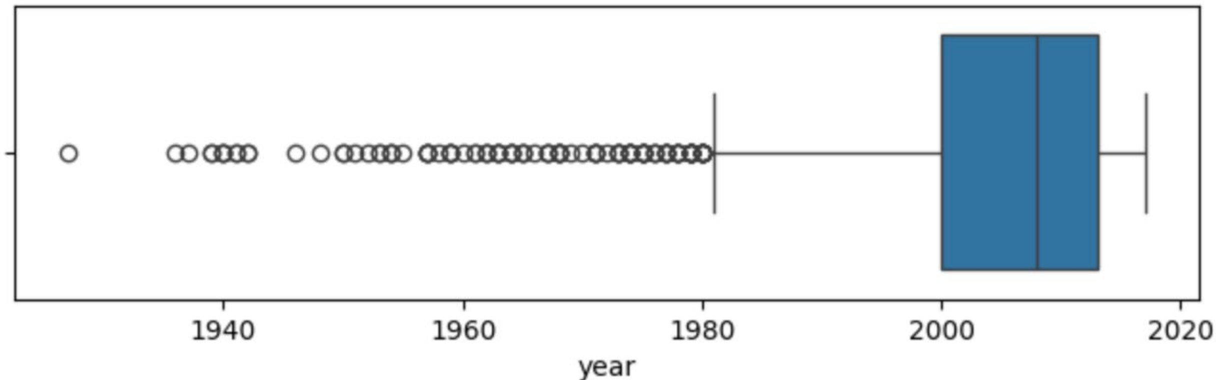
Total count: 2049

Boxplot (krabicový graf)

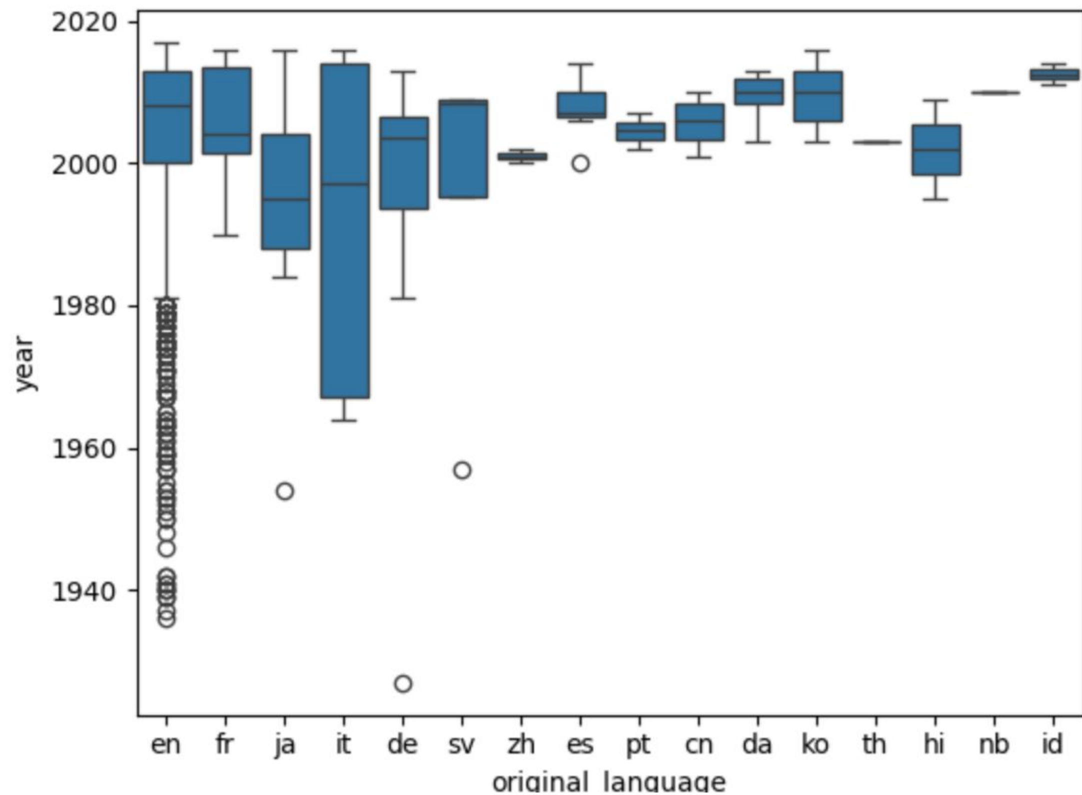
Developed by Mary Eleanor Hunt Spear and John Tukey.

It shows the **five-number summary:** min, $Q1$, median ($Q2$), $Q3$, max

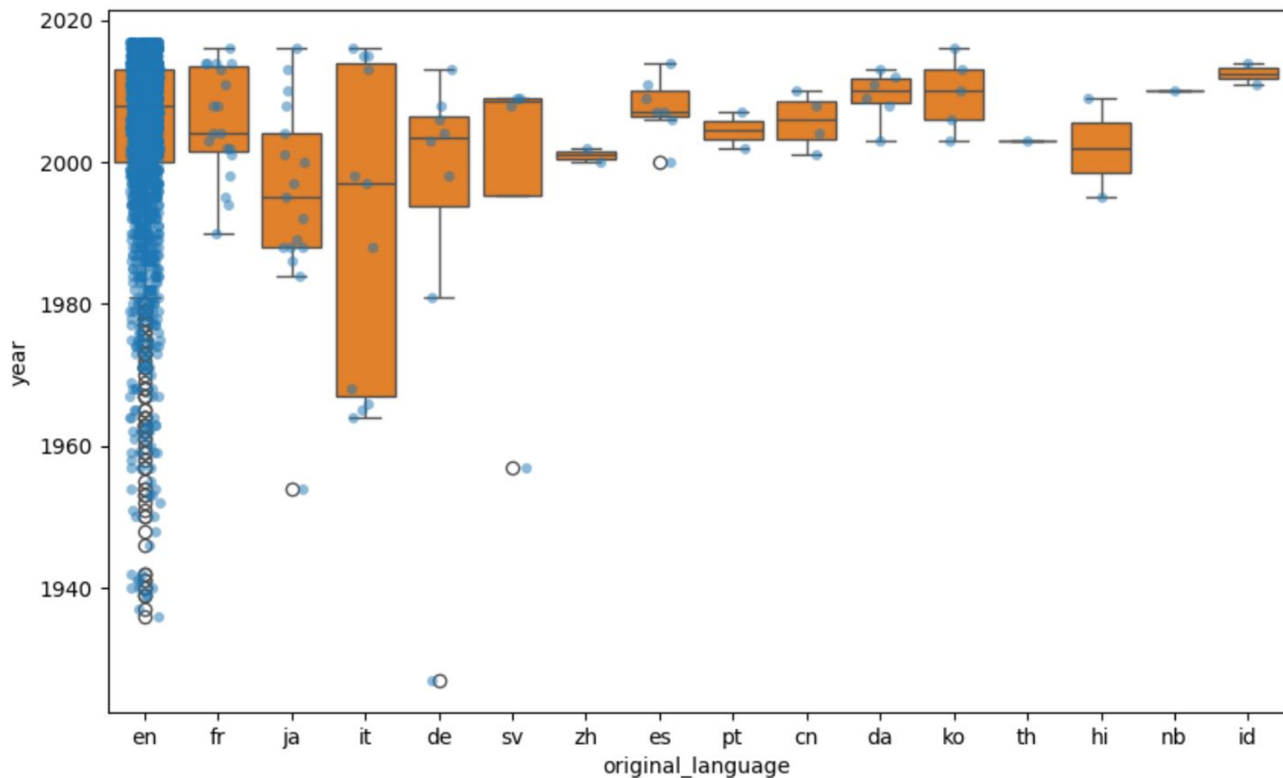
$Q1$ and $Q3$: a box, median: line through box, min and max: whiskers
Outliers often excluded from the whiskers and shown as points.



Boxplots are used for quick comparison



For small datasets we may add strip plot

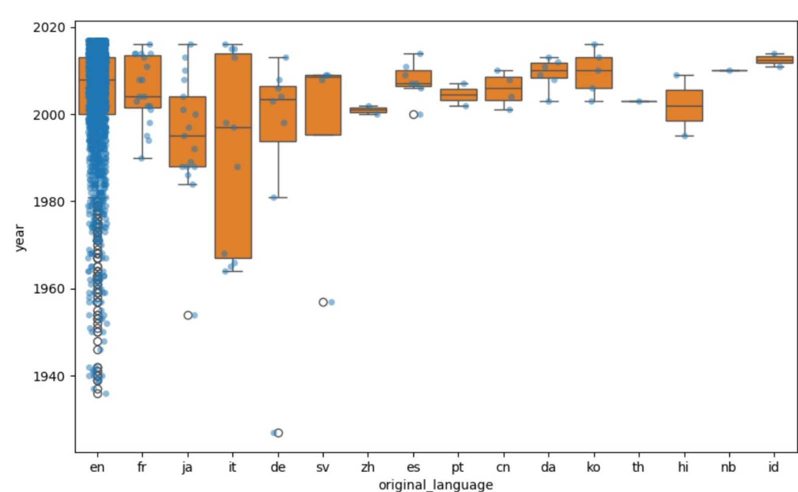


Works well for smaller languages, mess for en.

What do we see for sv, pt, th, hi, nb, id?

Code for the plot

```
axes = sns.boxplot(data=movies, x='original_language',
                   y='year', color='C1')
sns.stripplot(data=movies, x='original_language',
              y='year', color='C0',
              alpha=0.5, size=5, jitter=0.2)
```



Quick overview of a data set: `describe` in Pandas

```
movies.describe()
```

	year	budget	revenue	runtime	vote_average	vote_count
count	2,049.00	1,959.00	1,965.00	2,049.00	2,049.00	2,049.00
mean	2,004.14	55,108,939.70	198,565,134.28	112.66	6.63	1,704.64
std	12.70	53,139,663.86	233,028,732.94	24.76	0.77	1,607.89
min	1,927.00	1.00	15.00	7.00	4.00	501.00
25%	2,000.00	16,000,000.00	52,882,018.00	97.00	6.10	709.00
50%	2,008.00	38,000,000.00	122,200,000.00	109.00	6.60	1,092.00
75%	2,013.00	75,000,000.00	250,200,000.00	124.00	7.20	2,000.00
max	2,017.00	380,000,000.00	2,787,965,087.00	705.00	9.10	14,075.00

```
movies.describe(include='all').transpose()
```

	count	unique	top	freq	mean	std	min	25%	50%
title	2049	2018	Beauty and the Beast	3	NaN	NaN	NaN	NaN	NaN
year	2,049.00	NaN	NaN	NaN	2,004.14	12.70	1,927.00	2,000.00	2,008.00
budget	1,959.00	NaN	NaN	NaN	55,108,939.70	53,139,663.86	1.00	16,000,000.00	38,000,000.00
revenue	1,965.00	NaN	NaN	NaN	198,565,134.28	233,028,732.94	15.00	52,882,018.00	122,200,000.00
original_language	2049	16	en	1958	NaN	NaN	NaN	NaN	NaN
runtime	2,049.00	NaN	NaN	NaN	112.66	24.76	7.00	97.00	109.00
release_date	2049	1740	2014-12-25	6	NaN	NaN	NaN	NaN	NaN
vote_average	2,049.00	NaN	NaN	NaN	6.63	0.77	4.00	6.10	6.60
vote_count	2,049.00	NaN	NaN	NaN	1,704.64	1,607.89	501.00	709.00	1,092.00
overview	2049	2049	Led by Woody, Andy's toys live happily in his ...	1	NaN	NaN	NaN	NaN	NaN

Correlation (korelácia)

We are often interested in **relationships** among variables (data columns).

Next: two correlation coefficients that measure strength of such relationships.

Beware: **correlation does not imply causation**

Correlation does not imply causation

If electricity consumption grows in a very cold weather, there might be **cause-and-effect** relationship: the cold weather is causing people to use more electricity for heating.

If healthier people tend to be happier, which is the cause and which is effect?

Both studied variables can be also influenced by some third, **unknown factor.** For example, within a year, deaths by drowning increase with increased ice cream consumption. Both increases are spurred by warm weather.

The observed correlation can be just a **coincidence**,
see the Redskins rule and a specialized webpage Spurious Correlations.
You can easily find such "coincidences" by comparing many pairs of variables.

Pearson correlation coefficient

It measures linear relationship between two variables.

Consider pairs of values $(x_1,y_1),\ldots,(x_n,y_n)$, where (x_i,y_i) are two different features of the same object.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right).$$

where s_x is the standard deviation of variable x.

Pearson correlation coefficient

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s_x} \right) \left(\frac{y_i - \bar{y}}{s_y} \right).$$

Expression $(x_i - \bar{x})/s_x$ is called the **standard score** or **z-score:** how many standard deviations above or below the mean is x_i?

The product of z-scores for x_i and y_i is positive iff they lie on the same side of the respective means of x and y.

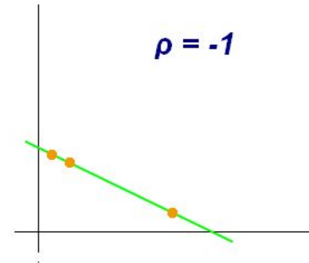
Properties of Pearson correlation coefficient

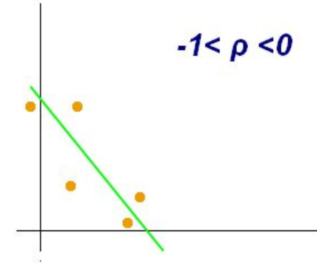
The value of r is always from interval $[-1,1]$.

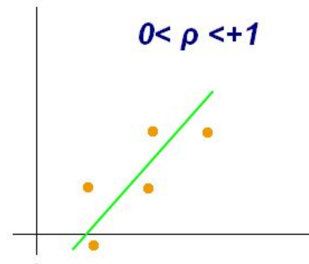
1 if y grows linearly with x,

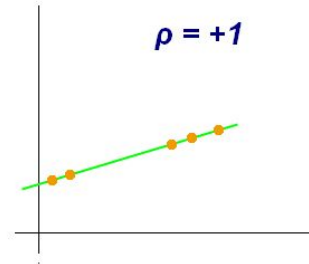
-1 if y decreases linearly

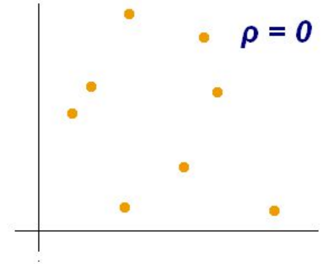
0 means no correlation.







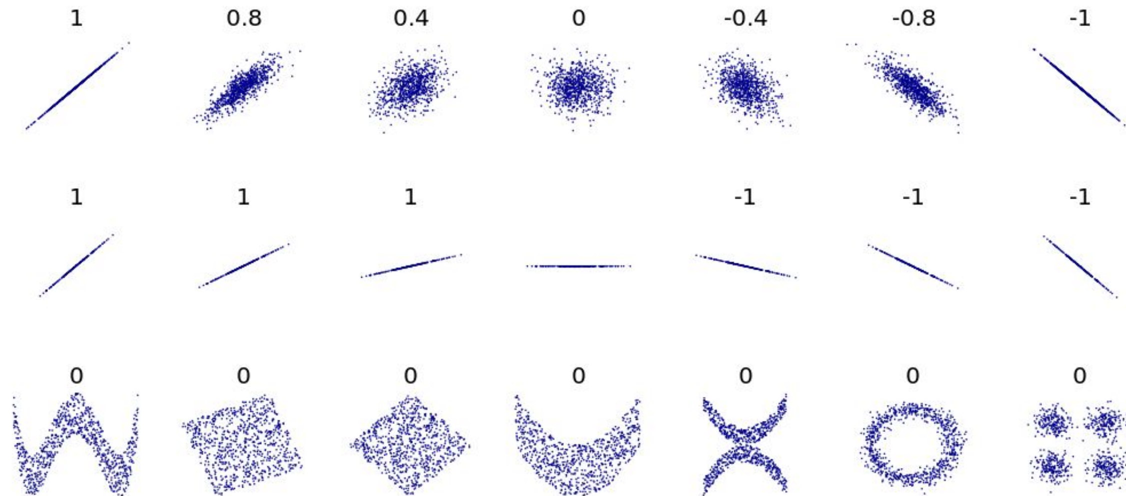




https://commons.wikimedia.org/wiki/File:Correlation_coefficient.png

Some cautions

Pearson correlation measures only linear relationships (bottom row)
Pearson correlation does not depend on the slope of the best-fit line (middle row)



https://commons.wikimedia.org/wiki/File:Correlation_examples2.svg

Properties of Pearson correlation

What if we linearly scale each variable, i.e. ax_i+b, cy_i+d?
What if a>0, a=0, a<0?

What if we switch x and y?

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i-\bar{x}}{s_x}\right)\left(\frac{y_i-\bar{y}}{s_y}\right).$$

Properties of Pearson correlation

Pearson correlation does not change if we linearly scale each variable, i.e. ax_i+b, cy_i+d (for $a,c>0$).

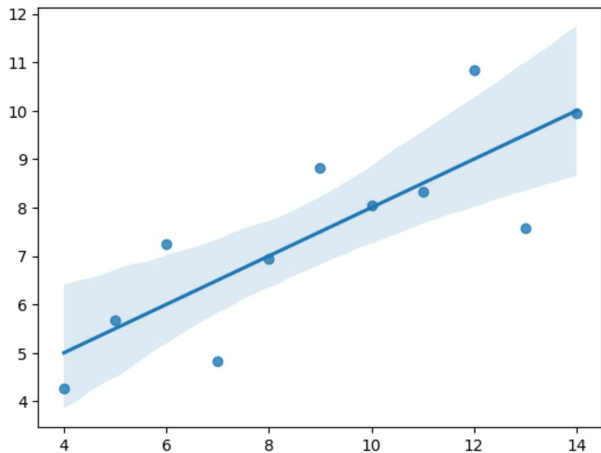
Pearson correlation is symmetric wrt. x and y.

Due to reliance on mean and std.dev. it is sensitive to outliers.

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s_x} \right) \left(\frac{y_i - \bar{y}}{s_y} \right).$$

Linear regression

The process of finding the line best representing the relationship of x and y.
In higher dimensions we can predict one variable as a linear combination of many.
You will study linear regression in later courses.
We may draw regression lines in some plots.




```
x = [10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5]
y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96,
      7.24, 4.26, 10.84, 4.82, 5.68]
sns.regplot(x=x, y=y1)
```

Spearman's rank correlation coefficient

It can detect **non-linear relationships**.

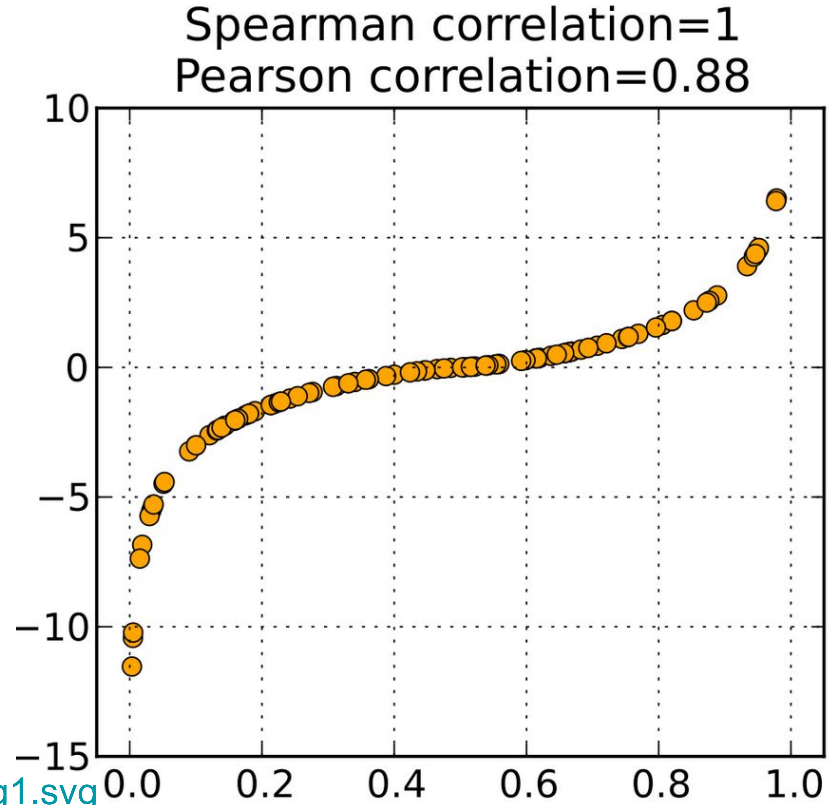
We first convert each variable into **ranks**:
Rank of x_i is its index in the sorted order of $x_1, \ldots, x_n$.
Equal values get the same (average) rank.
For example, the ranks of 10, 0, 10, 20, 10, 20 are 3, 1, 3, 5.5, 3, 5.5.

Then we compute **Pearson correlation coefficient of the two rank sequences**.

Properties of Spearman's rank correlation coeficient

Values of 1, -1 if y monotonically increases or decreases with x.

It is less sensitive to outliers (actual values of x and y are not important).



https://commons.wikimedia.org/wiki/File:Spearman_fig1.svg

Computation in Pandas

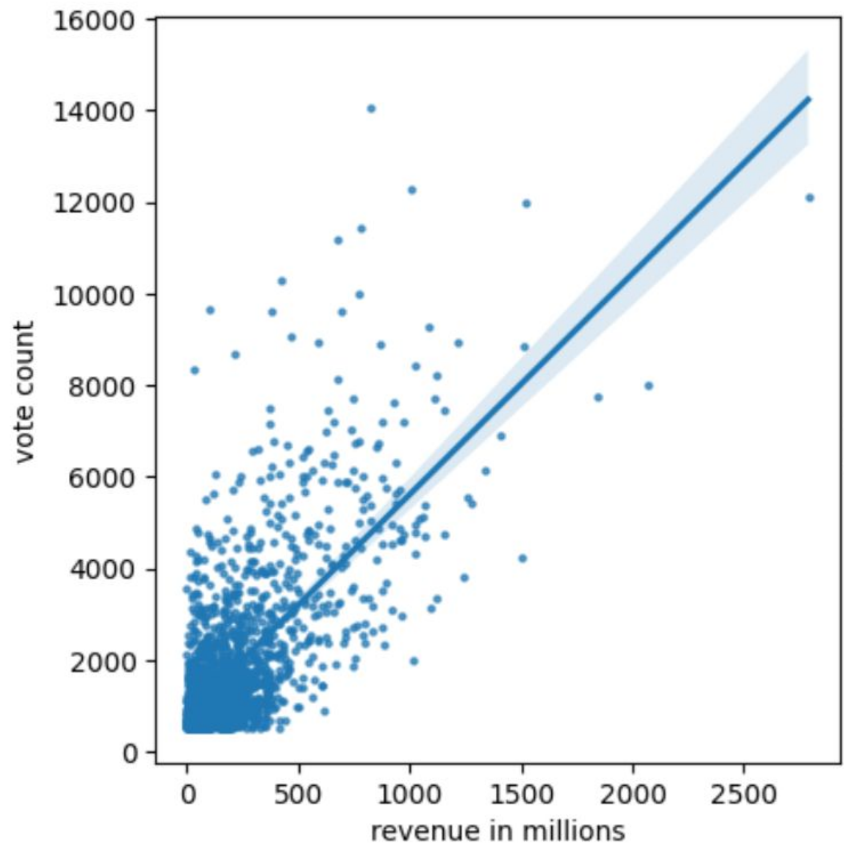
```
movies.corr(numeric_only=True)
```

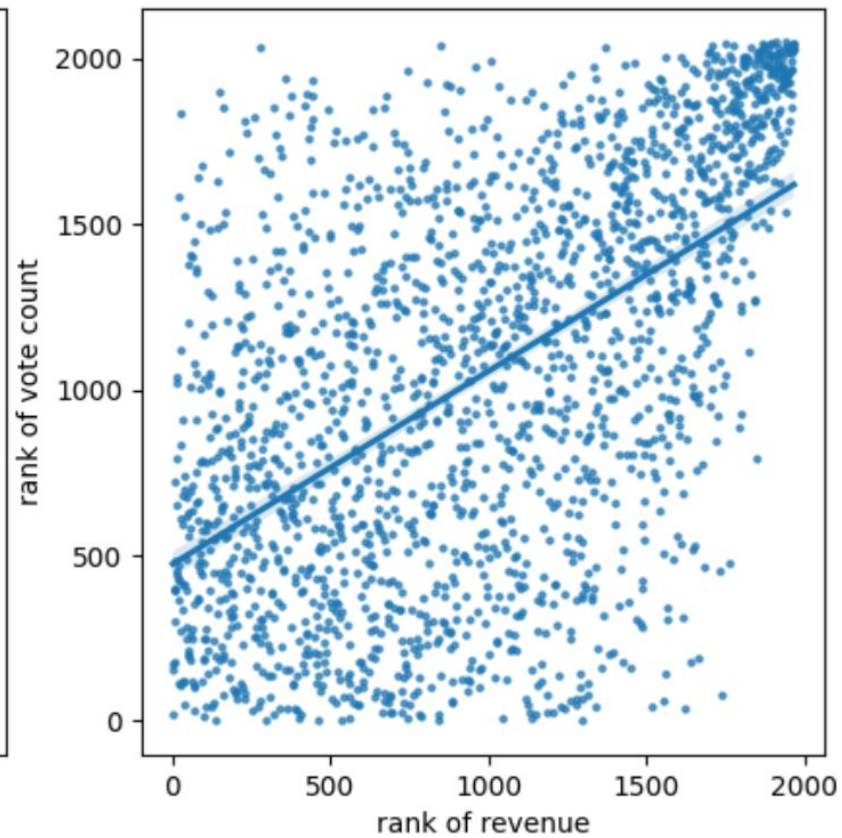
	year	budget	revenue	runtime	vote_average	vote_count
year	1.00	0.28	0.12	-0.07	-0.34	0.12
budget	0.28	1.00	0.69	0.22	-0.18	0.47
revenue	0.12	0.69	1.00	0.25	0.06	0.69
runtime	-0.07	0.22	0.25	1.00	0.31	0.25
vote_average	-0.34	-0.18	0.06	0.31	1.00	0.33
vote_count	0.12	0.47	0.69	0.25	0.33	1.00

Computation in Pandas (cont.)

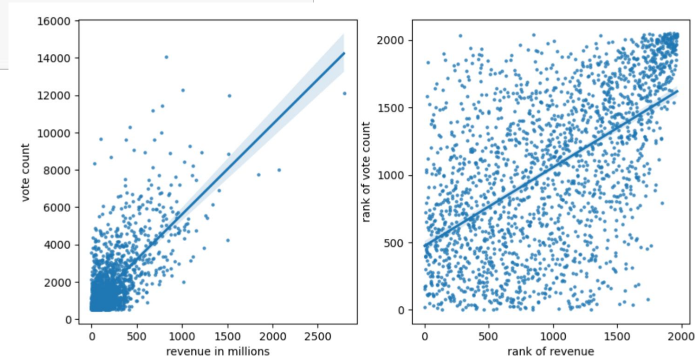
```
movies.corr(method='spearman', numeric_only=True)
```

	year	budget	revenue	runtime	vote_average	vote_count
year	1.00	0.21	0.02	-0.03	-0.27	0.14
budget	0.21	1.00	0.68	0.24	-0.28	0.37
revenue	0.02	0.68	1.00	0.21	-0.08	0.56
runtime	-0.03	0.24	0.21	1.00	0.32	0.27
vote_average	-0.27	-0.28	-0.08	0.32	1.00	0.29
vote_count	0.14	0.37	0.56	0.27	0.29	1.00


vote count
16000
14000
12000
10000
8000
6000
4000
2000
0
0
500
1000
1500
2000
2500
revenue in millions


rank of vote count
2000
1500
1000
500
0
0
500
1000
1500
2000
rank of revenue

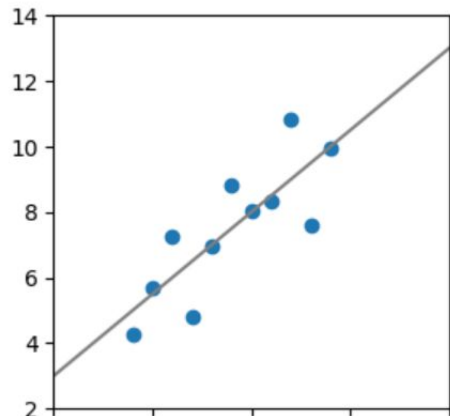
```python
figure, axes = plt.subplots(1, 2, figsize=(10,5))
# plot of values
sns.regplot(x=movies['revenue'] / 1e6, y=movies['vote_count'],
            ax=axes[0], scatter_kws={'alpha':0.7, 's':5})
axes[0].set_xlabel('revenue in millions')
axes[0].set_ylabel('vote count')
# compute ranks
revenue_rank = movies['revenue'].rank()
vote_count_rank = movies['vote_count'].rank()
# plot of ranks
sns.regplot(x=revenue_rank, y=vote_count_rank,
            ax=axes[1], scatter_kws={'alpha':0.7, 's':5})
axes[1].set_xlabel('rank of revenue')
axes[1].set_ylabel('rank of vote count')
```

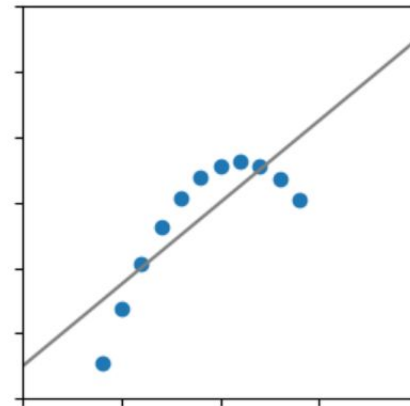


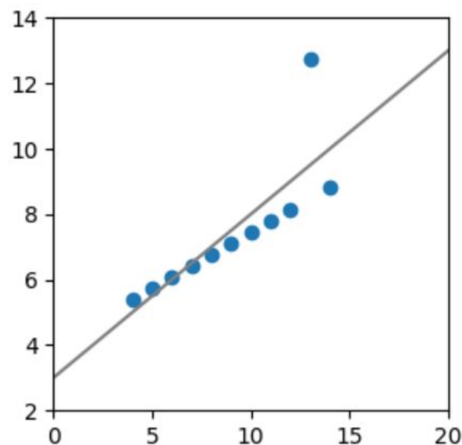
Anscombe's quartet

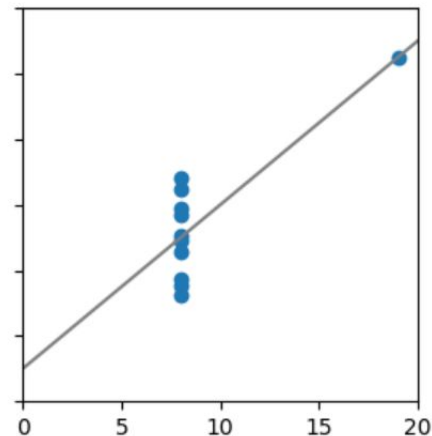
Four **artificial data sets** designed by Francis Anscombe.

The same or very similar values of: means and variances of x and y, Pearson correlation coefficient (0.816) and linear regression line.







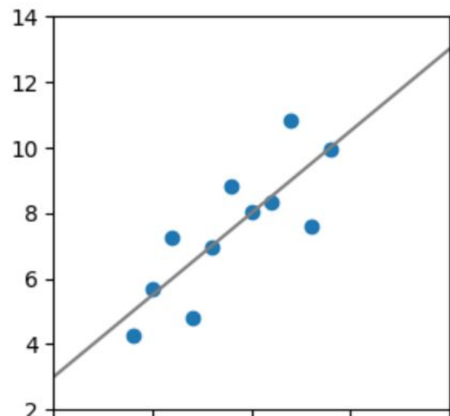


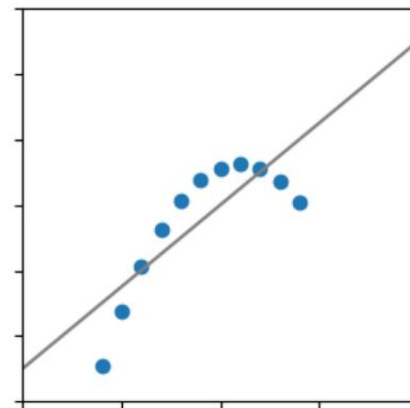
Anscombe's quartet

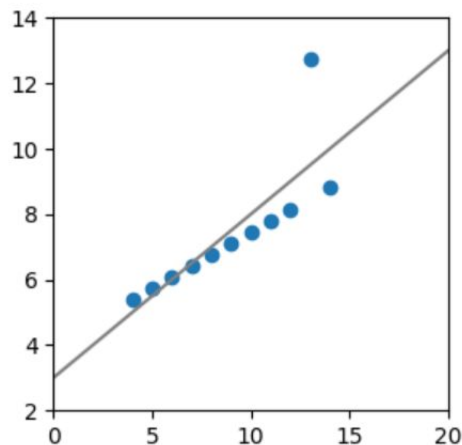
The same summary statistics,
but visually very different

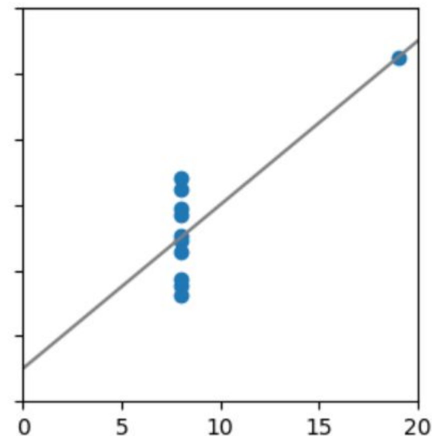
Importance of visualization:
Plots often give us a much better idea of the properties of a data set than simple numerical summaries.

The bottom row illustrates the influence of outliers on correlation and regression.





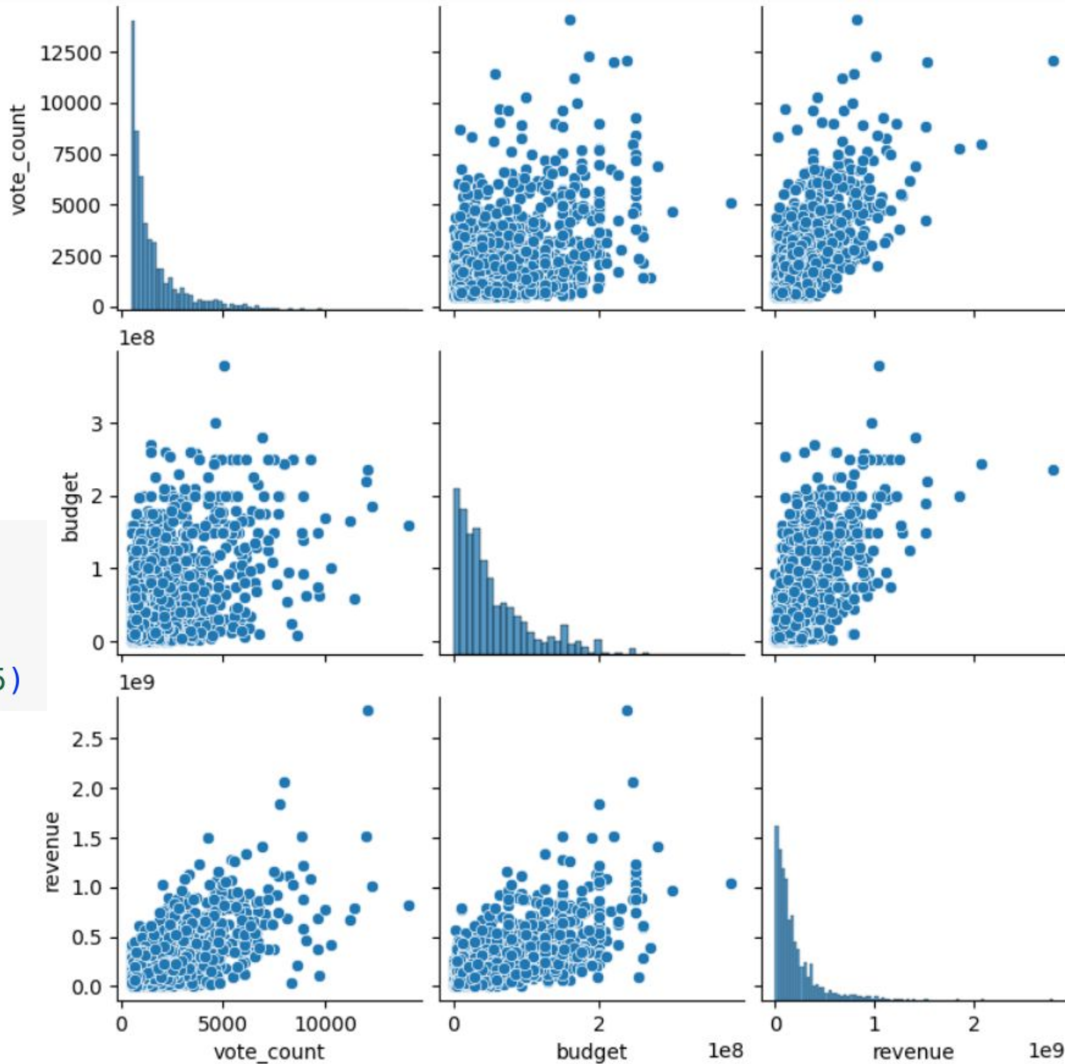




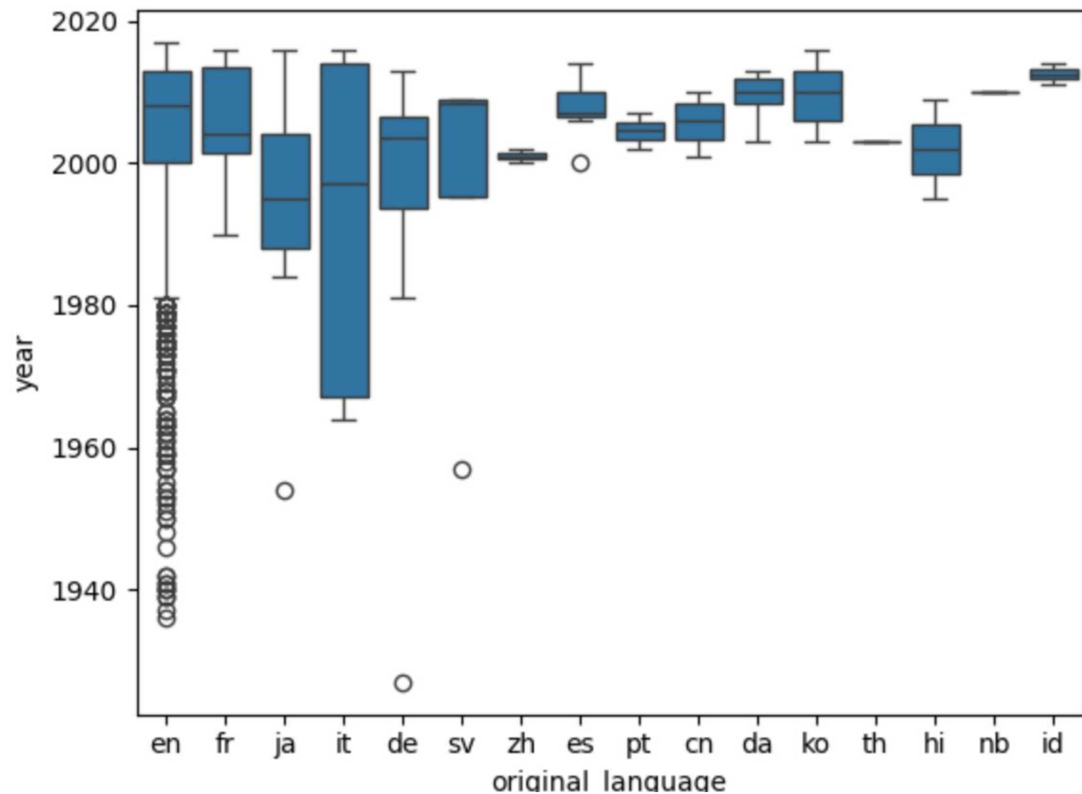
Visual overview of a data set: `pairplot` in Seaborn

```
subset = movies.loc[:, ['vote_count',
                        'budget',
                        'revenue']]
grid = sns.pairplot(subset, height=2.5)
```

All histograms + scatterplots



How to compute summaries for groups?



How to compute summaries for groups?

Method `groupby` splits the table into groups based on values of some column.

We can apply a summary statistics function or `describe` on each group.

```python
movies.groupby('original_language').median(numeric_only=True).head()
```

original_language	year	budget	revenue	runtime	vote_average	vote_count
cn	2,006.00	12,902,809.00	39,388,380.00	108.50	7.20	762.50
da	2,010.00	10,000,000.00	16,740,418.00	119.00	6.80	867.50
de	2,003.50	6,250,000.00	70,000,000.00	129.00	7.60	669.00
en	2,008.00	40,000,000.00	126,397,819.00	109.00	6.60	1,126.00
es	2,007.00	2,000,000.00	30,448,000.00	118.00	7.60	797.00

```
subset = movies.loc[:, ['original_language', 'year', 'budget']]
subset.groupby('original_language').describe().head(3).transpose()
```

	original_language	cn	da	de
year	count	4.00	6.00	8.00
	mean	2,005.75	2,009.33	1,992.50
	std	4.03	3.61	28.13
	min	2,001.00	2,003.00	1,927.00
	25%	2,003.25	2,008.25	1,993.75
	50%	2,006.00	2,010.00	2,003.50
	75%	2,008.50	2,011.75	2,006.50
	max	2,010.00	2,013.00	2,013.00
budget	count	3.00	5.00	8.00
	mean	14,872,795.67	13,440,000.00	18,223,718.75

Summary

Summary statistics:

- mean, median, mode
- percentiles, quantiles, quartiles
- min, max, interquartile range, variance, standard deviation
- Pearson and Spearman correlation

More details in a statistics course

Visualization:

- boxplot
- scatter plots with regression lines
- pairplot

Pandas:

- functions for computing statistics, `describe`
- `groupby`

Next week: more Pandas

1 Lecture 4: Summary statistics

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Introduction

- Summary statistics (popisné charakteristiky / štatistiky) are quantities that summarize basic properties of a single variable (a table column), such as the mean.
- We can also characterize dependencies between pairs of variables.
- Together with simple plots, such as histograms, they give us the first glimpse at the data when working with a new data set.
- We start by loading the movie data set, which we use to illustrate these terms.

1.2 Importing the movie data set

- The same data set as in group tasks 04.
- The data set describes 2049 movies.
- The data set was downloaded from https://www.kaggle.com/rounakbanik/the-movies-dataset and preprocessed, keeping only movies with at least 500 viewer votes.

[1]:
```
import numpy as np
import pandas as pd
from IPython.display import Markdown
import matplotlib.pyplot as plt
import seaborn as sns
pd.options.display.float_format = '{:,.2f}'.format
```

[2]:
```
url = 'https://bbrejova.github.io/viz/data/Movies_small.csv'
movies = pd.read_csv(url)
display(movies.head())
```

```
       title  year         budget        revenue original_language  runtime  \
0  Toy Story  1995 30,000,000.00 373,554,033.00                en    81.00
1    Jumanji  1995 65,000,000.00 262,797,249.00                en   104.00
2       Heat  1995 60,000,000.00 187,436,818.00                en   170.00
3  GoldenEye  1995 58,000,000.00 352,194,034.00                en   130.00
4     Casino  1995 52,000,000.00 116,112,375.00                en   178.00

  release_date  vote_average  vote_count  \
0   1995-10-30          7.70    5,415.00
1   1995-12-15          6.90    2,413.00
2   1995-12-15          7.70    1,886.00
3   1995-11-16          6.60    1,194.00
4   1995-11-22          7.80    1,343.00

                                            overview
0  Led by Woody, Andy's toys live happily in his …
```

```
1    When siblings Judy and Peter discover an encha…
2    Obsessive master thief, Neil McCauley leads a …
3    James Bond must unmask the mysterious head of …
4    The life of the gambling paradise - Las Vegas …
```

1.3 Measures of central tendency (miery stredu / polohy)

These represent a typical value in a sample x with values $x_1, \dots, x_n$ (one numerical column of a table).

- **Mean (priemer)** $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$
  - This is the arithmetic mean, there are also geometric and harmonic means.
- **Median (medián)** is the middle value when the values ordered from smallest to largest.
  - For even n usually defined as the average of the two middle values.
  - Median of 10, 12, 15, 16, 16 is 15.
  - Median of 10, 12, 15, 16, 16, 20 is 15.5.
- **Mode (modus)** is the most frequent value (for a discrete variable).
  - Mode of 10,12,15,16,16 is 16.
  - For continuous variables, we may look for a mode in a histogram (this is sensitive to bin size).

https://commons.wikimedia.org/wiki/File:Visualisation_mode_median_mean.svg Cmglee, CC BY-SA 3.0

1.3.1 Properties of the measures

- If we apply linear transformation $a \cdot x_i + b$ with the same constants a and b to all values x_i, mean, median and mode will be also transformed in the same way.
  - This corresponds e.g. to the change in the units of measurement (grams vs kilograms, degrees C vs degrees F)
- Mean can be heavily influenced by outliers.
  - Mean of 800, 1000, 1100, 1200, 1800, 2000 and 30000 is 5414.3, median 1200.
  - Mean of 800, 1000, 1100, 1200, 1800, 2000 and 10000 is 2557.1, median 1200.
- Therefore we often prefer median (e.g. median salary).

1.3.2 Computation in Pandas

Below we apply functions `mean`, `median`, `mode` to a Series (column `year` of our table).

Note that `mode` returns a Series of results (for case of ties). Here just a single value 2013.

Note the use of Python f-strings to print the results.

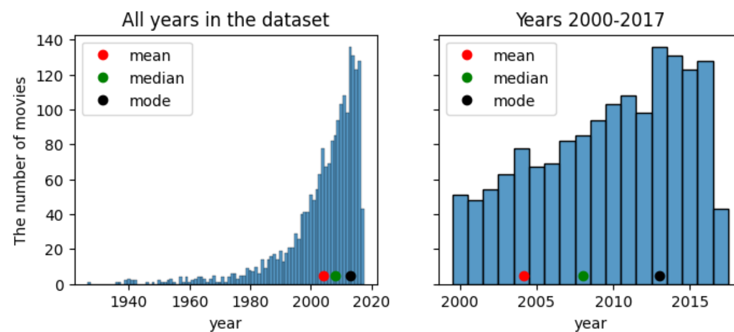
```
[3]: display(Markdown("**Properties of the column `year` in our table:**"))
print(f"Mean: {movies['year'].mean():.2f}")
print(f"Median: {movies['year'].median()}")
print(f"Mode:\n{movies['year'].mode()}")
```

Properties of the column year in our table:

```
Mean: 2004.14
Median: 2008.0
Mode:
0    2013
Name: year, dtype: int64
```

Let us see these values in a histogram of the column values (overall view and detail).

[4]:
```python
# set up figure with two plots
figure, axes = plt.subplots(1, 2, figsize=(8,3), sharey=True)

# plot histograms, use discrete=True to have each year in one bin
sns.histplot(data=movies, x='year', discrete=True, ax=axes[0])
sns.histplot(x=movies.query('year>=2000')['year'], discrete=True, ax=axes[1])

# titles and axis labels
axes[0].set_ylabel("The number of movies")
axes[0].set_title('All years in the dataset')
axes[1].set_title('Years 2000-2017')

# compute three summary statics, set up their color and label
stats = [{'label':'mean', 'value':movies['year'].mean(), 'color':'red'},
         {'label':'median', 'value':movies['year'].median(), 'color':'green'},
         {'label':'mode', 'value':movies['year'].mode(), 'color':'black'}]
# add dots for all statistics to both plots (at y=5)
for a in axes:
  for s in stats:
    a.plot(s['value'], 5, 'o', color=s['color'], label=s['label'])
  a.legend()

pass
```



- Functions `mean` and `median` can be applied to all numerical columns in a table.
- With `axis=1` we can compute means or medians in rows.

```
[5]: display(Markdown("**`movies.mean(numeric_only=True)`:**"), movies.
     ↪mean(numeric_only=True))
     display(Markdown("**`movies.median(numeric_only=True)`:**"), movies.
     ↪median(numeric_only=True))
```

`movies.mean(numeric_only=True)`:

```
year                    2,004.14
budget             55,108,939.70
revenue           198,565,134.28
runtime                   112.66
vote_average                6.63
vote_count              1,704.64
dtype: float64
```

`movies.median(numeric_only=True)`:

```
year                    2,008.00
budget             38,000,000.00
revenue           122,200,000.00
runtime                   109.00
vote_average                6.60
vote_count              1,092.00
dtype: float64
```

1.4 Quantiles, percentiles and quartiles (kvantily, percentily, kvartily)

- Median is the middle value in a sorted order.
- Therefore about 50% of values are smaller and 50% larger.
- For a different percentage p, the p-th **percentile** is at position roughly $(p/100) \cdot n$ in the sorted order of values.
- Similarly **quantile** (in Pandas), but we give fraction between 0 and 1 rather than percentage.
- Specifically **quartiles** are three values Q_1, Q_2 and Q_3 that split input data into quarters.
  - Therefore, Q_2 is the median.
- Many definitions exist regarding situations when the desired fraction falls between two values (we can take lower, higher, mean, weighted mean etc).

1.4.1 Computation in Pandas

- Function `quantile` gets a single value between 0 and 1 or a list of values and returns corresponding quantiles.
- To get quantiles for 0.1, 0.2, ..., 0.9, we generate a regular sequence of values using `np.arange`.

```
[6]: display(Markdown("**Median:**"), movies['year'].median())
     display(Markdown("**Quantile for 0.5:**"), movies['year'].quantile(0.5))
     display(Markdown("**All quartiles:**"), movies['year'].quantile([0.25, 0.5, 0.
     ↪75]))
     display(Markdown("**With step 0.1:**"), movies['year'].quantile(np.arange(0.1,␣
     ↪1, 0.1)))
```

Median:

```
2008.0
```

Quantile for 0.5:

```
2008.0
```

All quartiles:

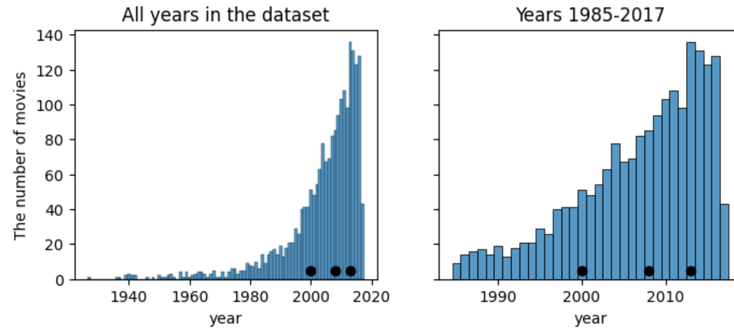
```
0.25   2,000.00
0.50   2,008.00
0.75   2,013.00
Name: year, dtype: float64
```

With step 0.1:

```
0.10   1,988.80
0.20   1,998.00
0.30   2,002.00
0.40   2,005.00
0.50   2,008.00
0.60   2,010.00
0.70   2,012.00
0.80   2,014.00
0.90   2,015.00
Name: year, dtype: float64
```

The code below plots the quartiles highlighted in a histogram.

[7]:
```
# setup histograms
figure, axes = plt.subplots(1, 2, figsize=(8,3), sharey=True)
sns.histplot(data=movies, x='year', discrete=True, ax=axes[0])
sns.histplot(x=movies.query('year>=1985')['year'], discrete=True, ax=axes[1])
axes[0].set_ylabel("The number of movies")
axes[0].set_title('All years in the dataset')
axes[1].set_title('Years 1985-2017')

# compute and display quartiles
quartiles = movies['year'].quantile([0.25, 0.5, 0.75])
for a in axes:
    a.plot(quartiles, [5] * len(quartiles), 'o', color='black')
pass
```



- The code below illustrates how the `quantile` function works when returning quantiles which do not correspond to a single input value.
- Optional parameter `interpolation` accepts values `'linear'` `(default)`, `'lower'`, `'higher'`, `'midpoint'`, `'nearest'`.
- Imagine the lowest element at quantile 0, the highest element at quantile 1 and the rest evenly spaced between. The quantile at position between two elements is influenced only by its two neighbors.

For example, consider list of values [0,10,20,100]. * Values taken from the list: $p = 0$: 0, $p = 1/3$: 10, $p = 2/3$: 20, $p = 1$: 100 * Example of a different value: $p = 1/4$: by default linear interpolation at 3/4 between 0 and 10, i.e. 7.5.

Note that linear interpolation is continuous as p changes from 0 to 1.

[8]:
```
a = pd.Series([0, 100])
b = pd.Series([0, 10, 20, 30, 100])
c = pd.Series([0, 10, 20, 100])
quantiles = [0.01, 0.25, 0.5, 0.75]
display(Markdown(f"**Quantiles for {list(a)}**"), a.quantile(quantiles))
display(Markdown(f"**Quantiles for {list(b)}**"), b.quantile(quantiles))
display(Markdown(f"**Quantiles for {list(c)}**"), c.quantile(quantiles))
display(Markdown(f"**Quantiles for {list(c)} with `interpolation='lower'`**"),
        c.quantile(quantiles, interpolation='lower'))
```

Quantiles for [0, 100]

```
0.01     1.00
0.25    25.00
0.50    50.00
0.75    75.00
dtype: float64
```

Quantiles for [0, 10, 20, 30, 100]

```
0.01     0.40
0.25    10.00
0.50    20.00
```

```
0.75   30.00
dtype: float64
```

Quantiles for [0, 10, 20, 100]

```
0.01    0.30
0.25    7.50
0.50   15.00
0.75   40.00
dtype: float64
```

Quantiles for [0, 10, 20, 100] with `interpolation='lower'`

```
0.01     0
0.25     0
0.50    10
0.75    20
dtype: int64
```

1.5 Measures of variability (miery variability)

- Values in the sample may be close to their mean or median, or they can spread widely.
- It is important to consider how representative is the mean or median of the whole set.

Examples of measures:

- Range of values from **minimum** to **maximum** (sensitive to outliers).
- **Interquartile range IQR (kvartilové rozpätie)**: range between Q_1 and Q_3 (contains the middle half of the data).
- Variance and standard deviation (described next).

1.5.1 Variance and standard deviation (rozptyl a smerodajná odchýlka)

Variance

- For each value in the sample compute its difference from the mean and square it: $(x_i - \bar{x})^2$.
- After squaring, we get non-negative values (and squares are easier to work with mathematically than absolute values).
- Variance is the mean of these squares, but we divide by $n-1$ rather than n:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$

- We divide by $n-1$ rather than n, because we would otherwise underestimate the true variance of the underlying population (more in the statistics course).
- For large n, the difference between dividing by n and $n-1$ is negligible.

Standard deviation

- Square root of the variance

$$s = \sqrt{s^2}$$

- It is expressed in the same units as the original values (variance is in units squared).

Properties

- Larger variance and standard deviation mean that data are spread farther from the mean
- If we apply linear transformation $a \cdot x_i + b$ with the same constants a and b to all values x_i:
  - Neither variance nor standard deviation change with b.
  - Variance is multiplied by a^2, standard deviation by $|a|$.
- These measures are strongly influenced by outliers:
  - For 800, 1000, 1100, 1200, 1800, 2000, 30000 st. dev. is 10850.0, IQR 850.
  - For 800, 1000, 1100, 1200, 1800, 2000, 10000 st. dev. is 3310.5, IQR 850.

1.5.2 Computation in Pandas

We can use functions `min`, `max`, `std`, `var`, which work similarly to `mean`.

[9]:
```
display(Markdown("**Minimum**"), movies['year'].min())
display(Markdown("**Maximum**"), movies['year'].max())
display(Markdown("**Mean**"), movies['year'].mean())
display(Markdown("**Variance**"), movies['year'].var())
display(Markdown("**Standard deviation**"), movies['year'].std())
q1 = movies['year'].quantile(0.25)
q3 = movies['year'].quantile(0.75)
display(Markdown("**Q1, Q3 and interquartile range:**"), q1, q3, q3-q1)
```

Minimum

```
1927
```

Maximum

```
2017
```

Mean

```
2004.1449487554905
```

Variance

```
161.2714600681735
```

Standard deviation

```
12.699270060447313
```

Q1, Q3 and interquartile range:

```
2000.0
```

```
2013.0
```

```
13.0
```

1.6 Outliers (odľahlé hodnoty)

- Outliers are the values which are far from the typical range of values.
- In data analysis, it is important to **check the outliers**.
- If they represent errors, we may try to correct or remove them.

- They can also represent interesting anomalies.
- Different definitions of outliers may be appropriate in different situations.
- The criterion by statistician John Tukey is often used:
  - Outliers are the values outside of the range $Q_1 - k \cdot IQR, Q_3 + k \cdot IQR$, e.g. for $k = 1.5$.
- In our example 800, 1000, 1100, 1200, 1800, 2000, 30000:
  - $Q_1 = 1050$, $Q_3 = 1900$, $IQR = 850$.
  - $Q_1 - 1.5 \cdot IQR = -225$, $Q_3 + 1.5 \cdot IQR = 3175$.
  - Outliers are values smaller than -225 or larger than 3175; here only 30000.
  - The range of outliers is not influenced if we change outliers values (as long as they stay outside of range Q1-Q3).

1.6.1 Computation in Pandas

- The code below finds outliers in the `year` column.
- We compute the lower and upper thresholds manually from quartiles.
- Then we use `query` to select rows and count how many there are.
- Function `count` counts the values in a Series or columns of a DataFrame, ignoring missing values.

[10]:
```
# get quartiles and iqr
q1 = movies['year'].quantile(0.25)
q3 = movies['year'].quantile(0.75)
iqr = q3 - q1
# compute thresholds for outliers
lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr
# count outliers
count = movies.query('year < @lower or year > @upper')['year'].count()
# print results
display(Markdown(f"**Outliers outside of range:** [{lower}, {upper}]"))
display(Markdown(f"**Outlier count:** {count}"))
display(Markdown(f"**Total count:** {movies['year'].count()}"))
```

Outliers outside of range: [1980.5, 2032.5]

Outlier count: 112

Total count: 2049

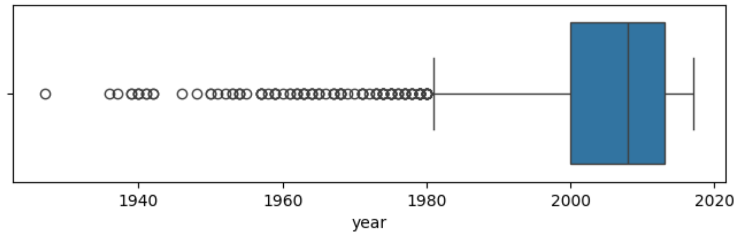
1.7 Boxplot (krabicový graf)

- Boxplots were developed by Mary Eleanor Hunt Spear and John Tukey.
- For a single numerical variable it shows the **five-number summary** consisting of the minimum, Q_1, median (Q_2), Q_3 and the maximum.
- Median is shown as a thick line, Q_1 and Q_3 as a box and minimum and maximum as "whiskers".
- Outliers are often excluded from the whiskers and shown as individual points.
- Summaries of different samples are often compared in a single boxplot.
- Boxplots allow clear comparison of basic characteristics.

1.7.1 Boxplots in Seaborn

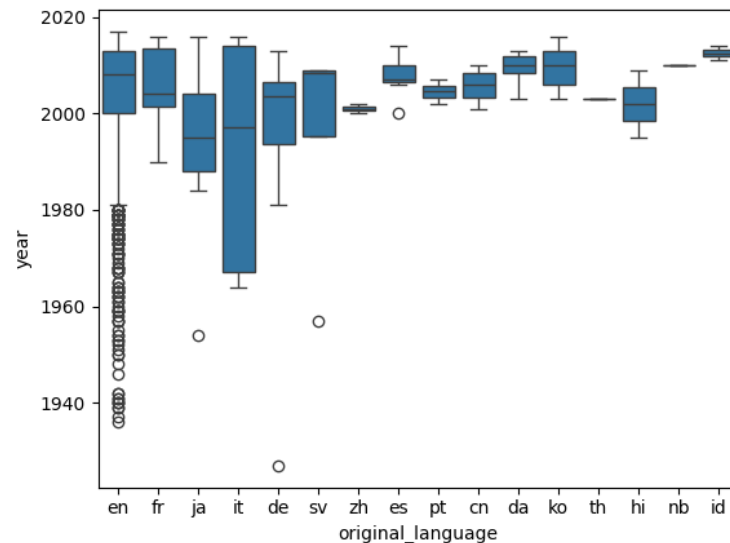
- We use `boxplot` finction from Seaborn.
- Below is a simple horizontal boxplot of the `year` column.
- Recall that quartiles are 2000, 2008 and 2013, minimum 1927, maximum 2017, outliers outside of [1980.5, 2032.5].

[11]:
```
axes = sns.boxplot(data=movies, x='year')
axes.figure.set_size_inches(8,2)
```



- Below is a vertical boxplot of the `year` column split into groups according to language.
- This is achieved by specifying both `x` and `y` options.

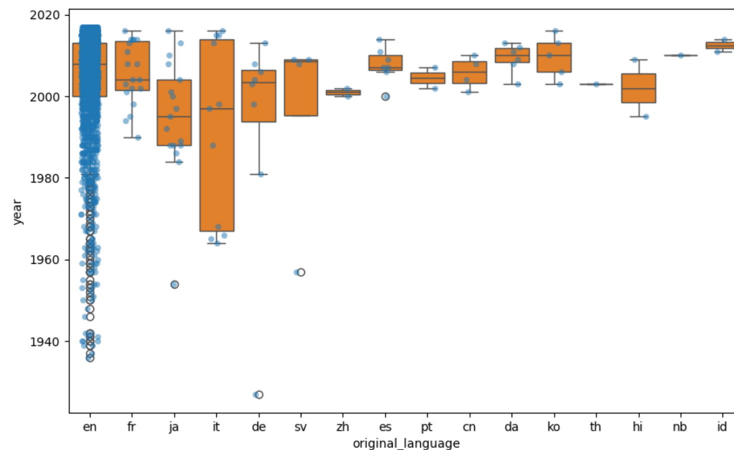
[12]:
```
sns.boxplot(data=movies, x='original_language', y='year')
pass
```



- Below we draw a strip plot on top of the boxplot.
- This allows us to see both individual data points and the summary.
- Here it does not work very well for `en`, better suited for smaller datasets.

- We see that some languages have extremely low number of points, boxplots not ideal in that case.

[13]:
```
axes = sns.boxplot(data=movies, x='original_language', y='year', color='C1')
sns.stripplot(data=movies, x='original_language', y='year', color='C0',
              alpha=0.5, size=5, jitter=0.2)
axes.figure.set_size_inches(10,6)
pass
```



1.8 Quick overview of a data set: `describe` in Pandas

Function `describe` gives a quick overview of a data set with many statistics described today.

[14]:
```
movies.describe()
```

[14]:
```
              year         budget          revenue  runtime  vote_average  \
count 2,049.00       1,959.00         1,965.00 2,049.00      2,049.00
mean  2,004.14  55,108,939.70   198,565,134.28   112.66          6.63
std      12.70  53,139,663.86   233,028,732.94    24.76          0.77
min   1,927.00           1.00            15.00     7.00          4.00
25%   2,000.00  16,000,000.00    52,882,018.00    97.00          6.10
50%   2,008.00  38,000,000.00   122,200,000.00   109.00          6.60
75%   2,013.00  75,000,000.00   250,200,000.00   124.00          7.20
max   2,017.00 380,000,000.00 2,787,965,087.00   705.00          9.10

       vote_count
count    2,049.00
mean     1,704.64
std      1,607.89
min        501.00
25%        709.00
50%      1,092.00
```

```
75%       2,000.00
max      14,075.00
```

- By default `describe` only considers numerical columns.
- Other columns can be included by `include='all'`.
- Different statistics reported for categorical columns (`unique`, `top`, `freq`).

[15]:
```
movies.describe(include='all').transpose()
```

[15]:
```
                     count unique  \
title                 2049   2018
year              2,049.00    NaN
budget            1,959.00    NaN
revenue           1,965.00    NaN
original_language     2049     16
runtime           2,049.00    NaN
release_date          2049   1740
vote_average      2,049.00    NaN
vote_count        2,049.00    NaN
overview              2049   2049

                                                             top  freq  \
title                                         Beauty and the Beast     3
year                                                         NaN   NaN
budget                                                       NaN   NaN
revenue                                                      NaN   NaN
original_language                                             en  1958
runtime                                                      NaN   NaN
release_date                                          2014-12-25     6
vote_average                                                 NaN   NaN
vote_count                                                   NaN   NaN
overview           Led by Woody, Andy's toys live happily in his …     1

                            mean            std      min            25%  \
title                        NaN            NaN      NaN            NaN
year                    2,004.14          12.70 1,927.00       2,000.00
budget             55,108,939.70  53,139,663.86     1.00 16,000,000.00
revenue           198,565,134.28 233,028,732.94    15.00 52,882,018.00
original_language            NaN            NaN      NaN            NaN
runtime                   112.66          24.76     7.00          97.00
release_date                 NaN            NaN      NaN            NaN
vote_average                6.63           0.77     4.00           6.10
vote_count              1,704.64       1,607.89   501.00         709.00
overview                     NaN            NaN      NaN            NaN

                             50%            75%            max
title                        NaN            NaN            NaN
```

```
year                          2,008.00       2,013.00         2,017.00
budget                   38,000,000.00  75,000,000.00   380,000,000.00
revenue                 122,200,000.00 250,200,000.00 2,787,965,087.00
original_language                  NaN            NaN              NaN
runtime                         109.00         124.00           705.00
release_date                       NaN            NaN              NaN
vote_average                      6.60           7.20             9.10
vote_count                    1,092.00       2,000.00        14,075.00
overview                           NaN            NaN              NaN
```

1.9 Correlation (korelácia)

- We are often interested in relationships among different variables (data columns).
- We will see two correlation coefficients that measure the strength of such relationships.
- Beware: **correlation does not imply causation**.
  - If electricity consumption grows in a very cold weather, there might be **cause-and-effect** relationship: the cold weather is causing people to use more electricity for heating.
  - If healthier people tend to be happier, which is the cause and which is effect?
  - Both studied variables can be also influenced by some third, unknown factor. For example, within a year, deaths by drowning increase with increased ice cream consumption. Both increases are spurred by warm weather.
  - The observed correlation can be just a coincidence, see the Redskins rule and a specialized webpage Spurious Correlations.
  - You can easily find such "coincidences" by comparing many pairs of variables (a practice called data dredging).

1.9.1 Pearson correlation coefficient

- It measures linear relationship between two variables.
- Consider pairs of values $(x_1, y_1), \ldots, (x_n, y_n)$, where (x_i, y_i) are two different features of the same object.

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

- Or equivalently:

$$r = \frac{1}{n-1}\sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right).$$

- where s_x is the standard deviation of variable x.
- Expression $(x_i - \bar{x})/s_x$ is called the **standard score** or **z-score**, and it tells us how many standard deviations above or below the mean value x_i is.
- The product of $(x_i - \bar{x})/s_x$ and $(y_i - \bar{y})/s_y$ is positive if x_i and y_i lie on the same side of the respective means of x and y and negative if they lie on the opposite sides.

1.9.2 Properties of Pearson correlation coefficient

Values of Pearson correlation coefficient

- The value of r is always from interval $[-1, 1]$.

- It is 1 if y grows linearly with x, -1 if y decreases linearly with increasing x.
- Zero means no correlation.
- Values between 0 and 1 mean intermediate value of positive correlation, values between -1 and 0 negative correlation.

https://commons.wikimedia.org/wiki/File:Correlation_coefficient.png Kiatdd, CC BY-SA 3.0

Some cautions

- Pearson correlation measures only linear relationships (x and y in the bottom row have non-linear relationships but their correlation is 0).
- Pearson correlation does not depend on the slope of the best-fit line (see the middle row below).

https://commons.wikimedia.org/wiki/File:Correlation_examples2.svg public domain

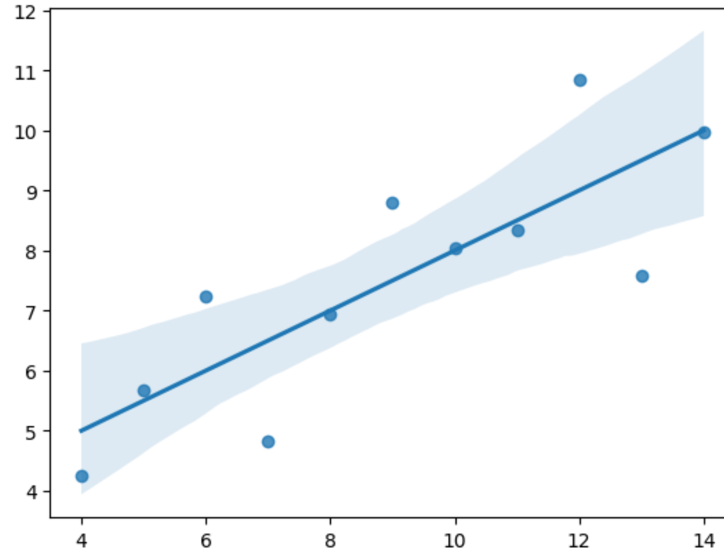
Other properties

- Pearson correlation does not change if we linearly scale each variable, i.e. $ax_i + b$, $cy_i + d$ (for $a, c > 0$).
- Pearson correlation is symmetric $r_{x,y} = r_{y,x}$.

1.9.3 Linear regression

- The process of finding the line best representing the relationship of x and y is called linear regression.
- It can be used in higher dimensions to predict one variable as a linear combination of many others.
- You will study linear regression in later courses, but we may draw regression lines in some plots.

[16]:
```
x = [10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5]
y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]
sns.regplot(x=x, y=y1)
pass
```



1.9.4 Spearman's rank correlation coefficient

- It can detect non-linear relationships.
- We first convert each variable into ranks:
  - Rank of x_i is its index in the sorted order of $x_1, \dots, x_n$.
  - Equal values get the same (average) rank.
  - For example, the ranks of 10, 0, 10, 20, 10, 20 are 3, 1, 3, 5.5, 3, 5.5.
- Then we compute Pearson correlation coefficient of the two rank sequences.
- Values of 1, -1 if y monotonically increases or decreases with x.
- It is less sensitive to distant outliers (actual values of x and y are not important).

https://commons.wikimedia.org/wiki/File:Spearman_fig1.svg Skbkekas, CC BY-SA 3.0

1.9.5 Computation in Pandas

Function `corr` computes correlation between all pairs of numerical columns. There is also a version to compare two Series.

In our table, the highest Pearson correlation is 0.69 for pairs (`budget`, `revenue`), (`vote_count`, `revenue`)

[17]:
```
movies.corr(numeric_only=True)
```

[17]:

	year	budget	revenue	runtime	vote_average	vote_count
year	1.00	0.28	0.12	-0.07	-0.34	0.12
budget	0.28	1.00	0.69	0.22	-0.18	0.47
revenue	0.12	0.69	1.00	0.25	0.06	0.69
runtime	-0.07	0.22	0.25	1.00	0.31	0.25
vote_average	-0.34	-0.18	0.06	0.31	1.00	0.33
vote_count	0.12	0.47	0.69	0.25	0.33	1.00

With Spearman rank correlation, the correlation between `revenue` and `budget` remains similar, but correlation between `vote_count` and `budget` decreases from 0.69 to 0.56.

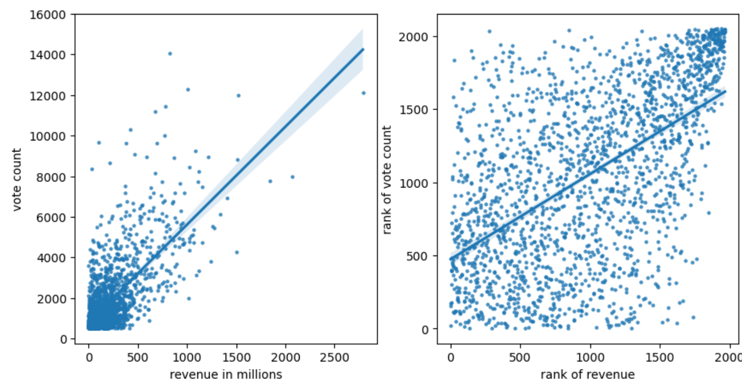
[18]:
```
movies.corr(method='spearman', numeric_only=True)
```

[18]:

	year	budget	revenue	runtime	vote_average	vote_count
year	1.00	0.21	0.02	-0.03	-0.27	0.14
budget	0.21	1.00	0.68	0.24	-0.28	0.37
revenue	0.02	0.68	1.00	0.21	-0.08	0.56
runtime	-0.03	0.24	0.21	1.00	0.32	0.27
vote_average	-0.27	-0.28	-0.08	0.32	1.00	0.29
vote_count	0.14	0.37	0.56	0.27	0.29	1.00

- Here we illustrate the regression line for `revenue` versus `vote_count`.
- We use Seaborn `regplot` to draw scatterplot together with the regression line.
- Points are made smaller and transparent by `scatter_kws={'alpha':0.7, 's':5}`.
- The plot on the right shows ranks instead of actual values.
- Ranks are computed using rank function for Series.
- Pearson correlation coefficient is probably driven by outliers.

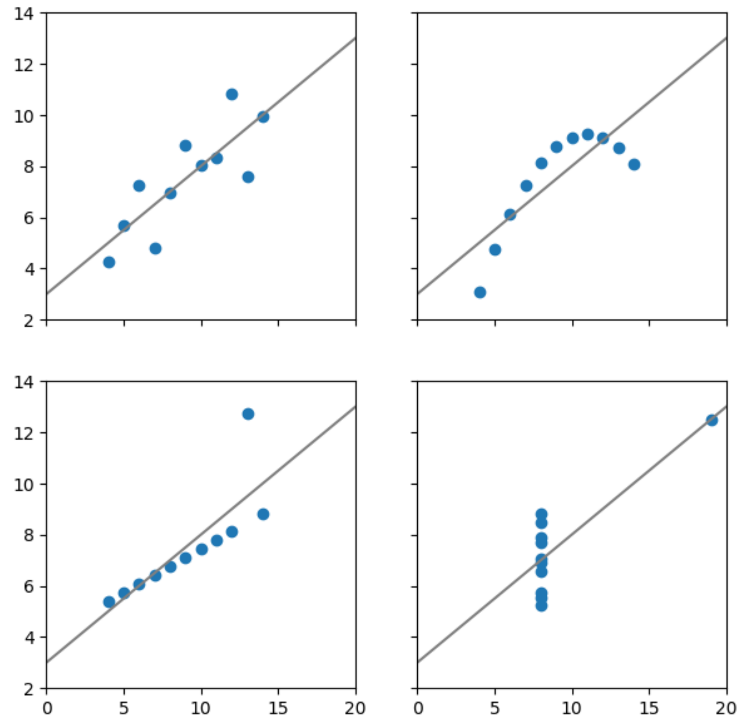
[19]:
```
# figure with two plots
figure, axes = plt.subplots(1, 2, figsize=(10,5))
# plot of values
sns.regplot(x=movies['revenue'] / 1e6, y=movies['vote_count'],
            ax=axes[0], scatter_kws={'alpha':0.7, 's':5})
axes[0].set_xlabel('revenue in millions')
axes[0].set_ylabel('vote count')
# compute ranks
revenue_rank = movies['revenue'].rank()
vote_count_rank = movies['vote_count'].rank()
# plot of ranks
sns.regplot(x=revenue_rank, y=vote_count_rank,
            ax=axes[1], scatter_kws={'alpha':0.7, 's':5})
axes[1].set_xlabel('rank of revenue')
axes[1].set_ylabel('rank of vote count')
pass
```



1.10 Anscombe's quartet and importance of visualization

- Anscombe's quartet are four artificial data sets designed by Francis Anscombe.
- All have the same or very similar values of means and variances of both x and y, Pearson correlation coefficient (0.816) and linear regression line.
- But visually we see each has a very different character.
- The bottom row illustrates the influence of outliers on correlation and regression.
- Overall this shows that plots give us a much better idea of the properties of a data set than simple numerical summaries.

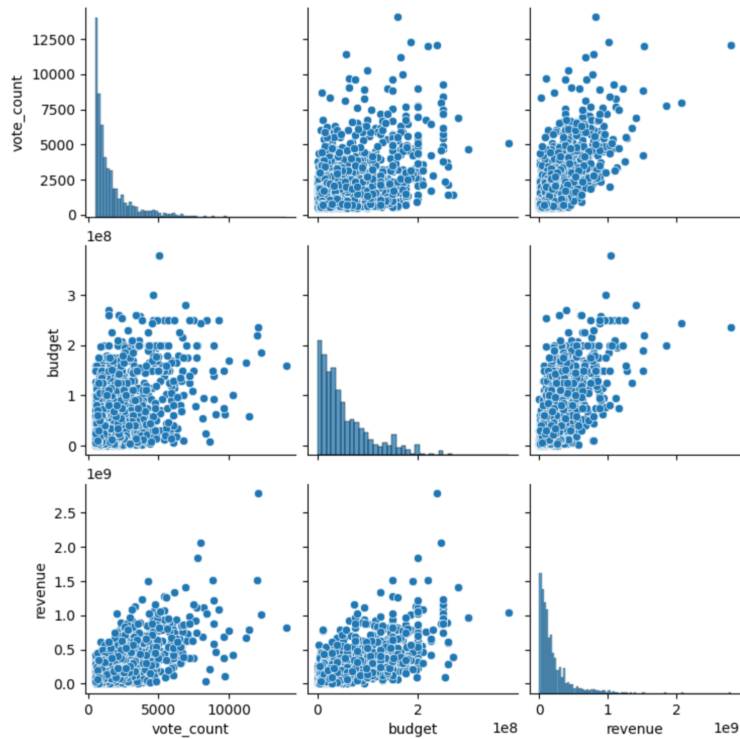
[20]:
```
# adapted from https://matplotlib.org/stable/gallery/specialty_plots/anscombe.
↪html
x = [10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5]
y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]
y2 = [9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74]
y3 = [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73]
x4 = [8, 8, 8, 8, 8, 8, 8, 19, 8, 8, 8]
y4 = [6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89]
datasets = [(x, y1), (x, y2), (x, y3), (x4, y4)]

figure, axes = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(7, 7))
axes[0, 0].set(xlim=(0, 20), ylim=(2, 14))

for ax, (x, y) in zip(axes.flat, datasets):
    ax.plot(x, y, 'o')
    # linear regression
    slope, intercept = np.polyfit(x, y, deg=1)
    ax.axline(xy1=(0, intercept), slope=slope, color='gray')
```



1.10.1 Visual overview of a data set: `pairplot` in Seaborn

- Seaborn `pairplot` generates a matrix of plots for all numerical columns.
- The diagonal contains histograms of individual columns.
- Off-diagonal entries are scatterplots of two columns.
- Here only 3 columns shown for simpler examination.

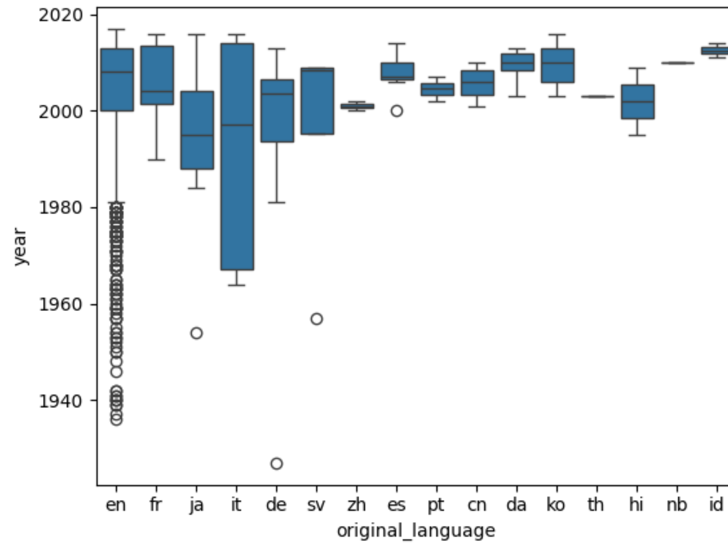
[21]:
```
subset = movies.loc[:, ['vote_count', 'budget', 'revenue']]
grid = sns.pairplot(subset, height=2.5)
pass
```



1.11 Computing summaries of subsets of data: groupby from Pandas

- We have seen that Seaborn can create plots where data are split into groups according to a categorical variable.
- One example are boxplots, which we have seen today.
- How can we compute summary statistics for each such group in Pandas?

[22]:
```
sns.boxplot(data=movies, x='original_language', y='year')
pass
```



- Pandas DataFrame supports function `groupby` which splits the table into groups based on values of some column.
- We can apply a summary statistics function on each group.
- Below we compute medians of all numerical columns for each language and show the first 5 languages.

[23]:
```
movies.groupby('original_language').median(numeric_only=True).head()
```

[23]:
```
                       year        budget         revenue  runtime  \
original_language
cn                 2,006.00 12,902,809.00   39,388,380.00   108.50
da                 2,010.00 10,000,000.00   16,740,418.00   119.00
de                 2,003.50  6,250,000.00   70,000,000.00   129.00
en                 2,008.00 40,000,000.00  126,397,819.00   109.00
es                 2,007.00  2,000,000.00   30,448,000.00   118.00

                   vote_average  vote_count
original_language
cn                         7.20      762.50
da                         6.80      867.50
de                         7.60      669.00
en                         6.60    1,126.00
es                         7.60      797.00
```

- We can also apply `describe` on the `groupby` groups.
- Here only two numerical columns of the original table are shown.

[24]:
```
subset = movies.loc[:, ['original_language', 'year', 'budget']]
subset.groupby('original_language').describe().head()
```

[24]:

	year						
	count	mean	std	min	25%	50%	75%
original_language							
cn	4.00	2,005.75	4.03	2,001.00	2,003.25	2,006.00	2,008.50
da	6.00	2,009.33	3.61	2,003.00	2,008.25	2,010.00	2,011.75
de	8.00	1,992.50	28.13	1,927.00	1,993.75	2,003.50	2,006.50
en	1,958.00	2,004.30	12.54	1,936.00	2,000.00	2,008.00	2,013.00
es	7.00	2,007.71	4.39	2,000.00	2,006.50	2,007.00	2,010.00

		budget			
	max	count	mean	std	min
original_language					
cn	2,010.00	3.00	14,872,795.67	4,479,793.25	11,715,578.00
da	2,013.00	5.00	13,440,000.00	12,369,640.25	3,800,000.00
de	2,013.00	8.00	18,223,718.75	30,623,544.47	1,530,000.00
en	2,017.00	1,891.00	56,637,200.97	53,394,829.52	1.00
es	2,014.00	5.00	7,500,000.00	8,046,738.47	1,500,000.00

	25%	50%	75%	max
original_language				
cn	12,309,193.50	12,902,809.00	16,451,404.50	20,000,000.00
da	7,400,000.00	10,000,000.00	11,000,000.00	35,000,000.00
de	4,100,000.00	6,250,000.00	15,084,937.50	92,620,000.00
en	18,000,000.00	40,000,000.00	80,000,000.00	380,000,000.00
es	2,000,000.00	2,000,000.00	13,000,000.00	19,000,000.00

1.12 Summary

We have seen several summary statistics:

- mean, median, mode
- percentiles, quantiles, quartiles
- min, max, interquartile range, variance, standard deviation
- Pearson and Spearman correlation

Visualization:

- boxplot
- scatter plots with regression lines
- pairplot

Pandas:

- functions for computing statistics, `describe`
- `groupby`
- next week: more Pandas

More details in a statistics course.

1 Lecture 5: Advanced Pandas

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

As usual, we start by importing libraries. We also import the country data set from World Bank https://databank.worldbank.org/home under CC BY 4.0 license (see Lecture 03b).

[1]:
```
import numpy as np
import pandas as pd
from IPython.display import Markdown
import matplotlib.pyplot as plt
import seaborn as sns
pd.options.display.float_format = '{:,.2f}'.format
```

[2]:
```
url = 'https://bbrejova.github.io/viz/data/World_bank.csv'
countries = pd.read_csv(url).set_index('Country')
```

1.1 Hierarchical index (MultiIndex)

1.1.1 A small example table

To illustrate a hierarchical index, we first create a very small table consisting of two countries and their population in two years, and convert this table from wide to long format.

[3]:
```
example_countries = countries.loc[["Slovak Republic", "Austria"],
                                  ["Population2010", "Population2020"]]
display(Markdown("**A small subset of countries table:**"), example_countries)
# change to long format
example_long = (example_countries.reset_index()
                                 .melt(id_vars=['Country'],
                                       var_name='Year',
                                       value_name='Population'))
# change year from string such as "Population2010" to int 2010
example_long.Year = example_long.Year.apply(lambda x : int(x[-4:]))
display(Markdown("**Changed to long format:**"), example_long)
```

A small subset of countries table:

	Population2010	Population2020
Country		
Slovak Republic	5,391,428.00	5,458,827.00
Austria	8,363,404.00	8,916,864.00

Changed to long format:

	Country	Year	Population
0	Slovak Republic	2010	5,391,428.00
1	Austria	2010	8,363,404.00

```
2  Slovak Republic  2020 5,458,827.00
3          Austria  2020 8,916,864.00
```

1.1.2 An index with duplicate labels

The original wide table had country as index, but in the long table, each country can have multiple rows. Pandas still allows us to use country as index with duplicate values. Selecting the name of the country then gives us multiple rows.

[4]:
```
# set country name as index in a copy of the table
example_long_indexed = example_long.set_index('Country')
# display table with index
display(Markdown("**Table with country as index:**"), example_long_indexed)
# select Slovakia from this table
display(Markdown("**Selecting multiple rows using `example_long_indexed.
↪loc['Slovak Republic']`:**"))
display(example_long_indexed.loc['Slovak Republic'])
```

Table with country as index:

```
                 Year   Population
Country
Slovak Republic  2010 5,391,428.00
Austria          2010 8,363,404.00
Slovak Republic  2020 5,458,827.00
Austria          2020 8,916,864.00
```

Selecting multiple rows using `example_long_indexed.loc['Slovak Republic']`:

```
                 Year   Population
Country
Slovak Republic  2010 5,391,428.00
Slovak Republic  2020 5,458,827.00
```

1.1.3 Finally the hierarchical index

Our table can be more naturally indexed by a pair (country, year), which uniquely specifies a row. An index consisting of two or more levels is called hierarchical or multi-level.

- MultiIndex can be created by `set_index` with a list of columns to use as index.
- For faster operations, it is a good idea to sort the table by the index using `sort_index`.
- In `loc` use a tuple with one value per level, or only several initial levels.
- To specify other levels, use `xs`.

[5]:
```
# create MultiIndex by choosing a list of columns
example_multiindexed = example_long.set_index(['Country', 'Year']).sort_index()
display(Markdown("**Table with a multiindex:**"), example_multiindexed)
```

Table with a multiindex:

```
                    Population
Country        Year
Austria        2010 8,363,404.00
               2020 8,916,864.00
Slovak Republic 2010 5,391,428.00
               2020 5,458,827.00
```

[6]:
```
display(Markdown("**Selecting a row by using a tuple in `loc`:**"))
display(example_multiindexed.loc[('Slovak Republic', 2010)])
```

Selecting a row by using a tuple in `loc`:

```
Population   5,391,428.00
Name: (Slovak Republic, 2010), dtype: float64
```

[7]:
```
display(Markdown("**Selecting all rows for a country using a shorter tuple in␣
 ↪`loc`:**"))
display(example_multiindexed.loc[('Slovak Republic',)])
```

Selecting all rows for a country using a shorter tuple in `loc`:

```
       Population
Year
2010 5,391,428.00
2020 5,458,827.00
```

[8]:
```
display(Markdown("**Selecting all rows for a year using `xs`:**"))
display(example_multiindexed.xs(2010, level='Year'))
```

Selecting all rows for a year using `xs`:

```
                  Population
Country
Austria         8,363,404.00
Slovak Republic 5,391,428.00
```

[9]:
```
display(Markdown("**Names of index levels can be used in `query`:**"))
display(example_multiindexed.query('Year > 2015'))
```

Names of index levels can be used in `query`:

```
                       Population
Country         Year
Austria         2020 8,916,864.00
Slovak Republic 2020 5,458,827.00
```

1.2 Combining tables

1.2.1 Concatenating tables using `concat`

- Function `concat` can be used to concatenate several tables.

- At the default settings, it combines along axis 0, meaning that the rows of second table are added after the rows of the first table.
- We will also use it for `axis=1`, in which case it finds rows with the same index in both tables and combines their columns.
- By default, the result has union of rows of the two tables, but intersection can be obtained by `join='inner'`.

Example Create a second small table of countries and display both tables. Then illustrate various concatenation modes using these tables.

[10]:
```
example_countries2 = countries.loc[["Slovak Republic", "Austria", "Hungary"],
                                   ["Area", "Region"]]
display(Markdown("**The first small table:**"), example_countries)
display(Markdown("**The second small table:**"), example_countries2)
```

The first small table:

```
                 Population2010  Population2020
Country
Slovak Republic    5,391,428.00    5,458,827.00
Austria            8,363,404.00    8,916,864.00
```

The second small table:

```
                     Area                 Region
Country
Slovak Republic 49,030.00  Europe & Central Asia
Austria         83,879.00  Europe & Central Asia
Hungary         93,030.00  Europe & Central Asia
```

[11]:
```
display(Markdown("**Tables concatenated along axis 0:**"))
display(pd.concat([example_countries, example_countries2]))
```

Tables concatenated along axis 0:

```
                 Population2010  Population2020      Area  \
Country
Slovak Republic    5,391,428.00    5,458,827.00       NaN
Austria            8,363,404.00    8,916,864.00       NaN
Slovak Republic             NaN             NaN 49,030.00
Austria                     NaN             NaN 83,879.00
Hungary                     NaN             NaN 93,030.00

                                Region
Country
Slovak Republic                    NaN
Austria                            NaN
Slovak Republic  Europe & Central Asia
Austria          Europe & Central Asia
Hungary          Europe & Central Asia
```

[12]:
```
display(Markdown("**Tables concatenated along axis 1:**"))
display(pd.concat([example_countries, example_countries2], axis=1))
```

Tables concatenated along axis 1:

```
                 Population2010  Population2020      Area  \
Country
Slovak Republic    5,391,428.00    5,458,827.00 49,030.00
Austria            8,363,404.00    8,916,864.00 83,879.00
Hungary                     NaN             NaN 93,030.00

                               Region
Country
Slovak Republic  Europe & Central Asia
Austria          Europe & Central Asia
Hungary          Europe & Central Asia
```

[13]:
```
display(Markdown("**Tables concatenated along axis 1 with inner join:**"))
display(pd.concat([example_countries, example_countries2], axis=1,␣
  ↪join='inner'))
```

Tables concatenated along axis 1 with inner join:

```
                 Population2010  Population2020      Area  \
Country
Slovak Republic    5,391,428.00    5,458,827.00 49,030.00
Austria            8,363,404.00    8,916,864.00 83,879.00

                               Region
Country
Slovak Republic  Europe & Central Asia
Austria          Europe & Central Asia
```

1.2.2 Merging tables with `merge`

- Function `merge` works similarly as `concat` with `axis=1`, but it will match lines of two tables using any specified columns, not necessarily index.
- If values in these columns repeat, it combines all matching pairs of rows.
- Setting `how` in `merge` allows us to include rows that do not have a matching row in the other table.

[14]:
```
# a small example of how all combinations of matching rows are returned:
tab1 = pd.DataFrame({'name': ['a','a','a','b'], 'value': [1,2,3,4]})
tab2 = pd.DataFrame({'name': ['a','a','b'], 'value': [10,20,30]})
display(Markdown("**DataFrame `tab1`:**"))
display(tab1)
display(Markdown("**DataFrame `tab2`:**"))
display(tab2)
display(Markdown("**Result of `pd.merge(tab1, tab2, on='name')`:**"))
```

```
display(pd.merge(tab1, tab2, on='name'))
```

DataFrame tab1:

	name	value
0	a	1
1	a	2
2	a	3
3	b	4

DataFrame tab2:

	name	value
0	a	10
1	a	20
2	b	30

Result of pd.merge(tab1, tab2, on='name'):

	name	value_x	value_y
0	a	1	10
1	a	1	20
2	a	2	10
3	a	2	20
4	a	3	10
5	a	3	20
6	b	4	30

Example of using merge on countries

- Countries belong to various international organizations and a single country can belong to many. We will represent this as a table having one row for each pair of country and an organization it belongs to.
- To combine this with other country data, we apply merge to get a table in which each country is copied for each organization it is in.
- Then we can for example compute the total number of people living in countries covered by individual organizations.

[15]:
```
# we create a small membership table by parsing a CSV-format string
import io
membership_str = io.StringIO("""Country,Member
Slovak Republic,NATO
Slovak Republic,EU
Slovak Republic,UN
Austria,UN
Austria,EU
""")
membership = pd.read_csv(membership_str)
display(Markdown("**A small country membership table:**"), membership)
```

A small country membership table:

```
            Country Member
0  Slovak Republic   NATO
1  Slovak Republic     EU
2  Slovak Republic     UN
3          Austria     UN
4          Austria     EU
```

[16]:
```
# merging tables using column Country in both
example_membership = pd.merge(example_countries, membership, on='Country')
display(Markdown("**Merged table:**"), example_membership)
```

Merged table:

```
            Country  Population2010  Population2020 Member
0  Slovak Republic    5,391,428.00    5,458,827.00   NATO
1  Slovak Republic    5,391,428.00    5,458,827.00     EU
2  Slovak Republic    5,391,428.00    5,458,827.00     UN
3          Austria    8,363,404.00    8,916,864.00     UN
4          Austria    8,363,404.00    8,916,864.00     EU
```

[17]:
```
# compute the total number of people in EU (here only for our two countries)
display(example_membership.query('Member == "EU"')['Population2020'].sum())
```

```
14375691.0
```

As we will see in the next section, we can also use `groupby` to compute sums for all organizations.

[18]:
```
display(Markdown("**The sum of country populations for each organization**␣
  ↪(only for our two countries)"))
display(example_membership.groupby('Member')['Population2020'].sum())
```

The sum of country populations for each organization (only for our two countries)

```
Member
EU      14,375,691.00
NATO     5,458,827.00
UN      14,375,691.00
Name: Population2020, dtype: float64
```

Similar operations are often done in relational databases, where `merge` is called join. Aggregation as in `groupby` is also frequently used. More in a specialized database course in the third year.

1.3 Aggregation, split-apply-combine (groupby)

We have already seen simple examples of aggregation by `groupby` in Lecture 04. Here we discuss it in more detail.

Pandas follow the split-apply-combine strategy introduced in R by Hadley Wickham.

Split: split data into groups, often by values in some column, such as `Region` in the `countries` table.

Apply: apply some computation on each group, obtaining some result (single value, Series, DataFrame).

Combine: concatenate results for all groups together to a new table.

Typical operations in the apply step:

- **aggregation**: e.g. compute group size, mean, median etc.
- **transformation**: e.g. compute percentage or rank of each item within a group
- **filtering**: e.g. include only groups that are large enough

In Pandas, this is done by combination of groupby for the split step and additional functions for the apply step. The combine step is done implicitly. Pandas library provides many options, we will cover only basics.

1.3.1 Simple aggregation in the apply step

Apply functions such as `sum`, `mean`, `median`, `min`, `max`, `size`, `count`, `describe` after `groupby`.

- `size` gives the number of rows in the group.
- `count` gives the number of non-missing values in each column.

[19]:
```
display(Markdown("**The number of countries in each region:**"))
display(countries.groupby('Region').size())
```

The number of countries in each region:

```
Region
East Asia & Pacific           37
Europe & Central Asia         58
Latin America & Caribbean     42
Middle East & North Africa    21
North America                  3
South Asia                     8
Sub-Saharan Africa            48
dtype: int64
```

[20]:
```
display(Markdown("**Sums of country indicators in each region**"))
display(Markdown(" (including nonsense sums such as life expectation or GDP per␣
  ↪capita)"))
display(countries.groupby('Region').sum(numeric_only=True))
```

Sums of country indicators in each region

(including nonsense sums such as life expectation or GDP per capita)

```
                               Population2000   Population2010   Population2020  \
Region
East Asia & Pacific          2,025,976,167.00 2,187,065,378.00 2,340,350,517.00
Europe & Central Asia          862,786,208.00   889,169,626.00   922,353,365.00
Latin America & Caribbean      521,281,151.00   588,873,865.00   650,534,988.00
Middle East & North Africa     321,037,455.00   397,997,552.00   479,966,650.00
```

```
North America                312,909,973.00   343,397,156.00   369,582,572.00
South Asia                 1,406,945,496.00 1,660,546,144.00 1,882,531,621.00
Sub-Saharan Africa           671,212,484.00   879,797,424.00 1,151,302,077.00

                                    Area    GDP2000      GDP2010      GDP2020  \
Region
East Asia & Pacific        24,794,669.42 233,980.83   506,478.43   569,610.58
Europe & Central Asia      28,813,751.77 883,386.74 1,752,994.15 1,966,242.67
Latin America & Caribbean  20,523,017.36 194,902.34   462,544.16   522,816.20
Middle East & North Africa 11,385,553.90 172,013.59   327,153.10   293,809.50
North America              19,715,550.00 116,885.13   198,088.01   214,882.96
South Asia                  5,135,270.00   5,525.87    16,479.92    21,370.63
Sub-Saharan Africa         24,328,265.87  43,582.79   108,587.66    96,165.83

                            Expectancy2000  Expectancy2010  Expectancy2020  \
Region
East Asia & Pacific               2,436.61        2,454.65        2,520.88
Europe & Central Asia             4,053.36        4,204.81        4,255.06
Latin America & Caribbean         2,931.71        3,013.73        2,957.72
Middle East & North Africa        1,496.97        1,557.33        1,568.09
North America                       234.66          240.36          239.79
South Asia                          511.70          546.95          568.09
Sub-Saharan Africa                2,547.64        2,803.97        3,003.10

                            Fertility2000  Fertility2010  Fertility2020
Region
East Asia & Pacific                103.92          91.58          80.30
Europe & Central Asia               94.74         100.09          93.30
Latin America & Caribbean          108.07          90.74          76.61
Middle East & North Africa          71.54          59.86          52.64
North America                        5.31           5.28           4.45
South Asia                          31.62          24.66          19.56
Sub-Saharan Africa                 262.34         237.78         205.94
```

[21]:
```
display(Markdown("**Specifically sum only population in 2020 per region:**"))
display(countries.groupby('Region')['Population2020'].sum())
```

Specifically sum only population in 2020 per region:

```
Region
East Asia & Pacific          2,340,350,517.00
Europe & Central Asia          922,353,365.00
Latin America & Caribbean      650,534,988.00
Middle East & North Africa     479,966,650.00
North America                  369,582,572.00
South Asia                   1,882,531,621.00
Sub-Saharan Africa           1,151,302,077.00
Name: Population2020, dtype: float64
```

1.3.2 Transformation in the apply step

Here we use `transform` method which get a function which is used on every group and should produce a group with the same index. We could write our own function (e.g. a lambda expression) or we can use one the built-in functions specified by a string.

Here we compute for each country what percentage is its population from the population of the region.

[22]:
```
# group countries by region, compute the sum of each region
# and copy the regional sum for each country
region_sums = countries.groupby('Region')['Population2020'].transform('sum')
display(Markdown("**For each country, what is the total population of its␣
↪region:**"))
display(region_sums)
# now divide the population of the country by the regional total
pop_within_group = countries['Population2020'] / region_sums
display(Markdown("**For each country, what fraction is its population within␣
↪region:**"))
display(pop_within_group.head())
```

For each country, what is the total population of its region:

```
Country
Afghanistan          1,882,531,621.00
Albania                922,353,365.00
Algeria                479,966,650.00
American Samoa       2,340,350,517.00
Andorra                922,353,365.00
                             …
Virgin Islands         650,534,988.00
West Bank and Gaza     479,966,650.00
Yemen                  479,966,650.00
Zambia               1,151,302,077.00
Zimbabwe             1,151,302,077.00
Name: Population2020, Length: 217, dtype: float64
```

For each country, what fraction is its population within region:

```
Country
Afghanistan      0.02
Albania          0.00
Algeria          0.09
American Samoa   0.00
Andorra          0.00
Name: Population2020, dtype: float64
```

Bellow we see an alternative form of the same computation when transformation is done via a lambda function that takes a list `x` of country sizes within a region and divides them by the sum of `x`.

The use of lambda functions applied on each element is often convenient but might be slow on large data.

[23]:
```
pop_within_group = (countries.groupby('Region')['Population2020']
                    .transform(lambda x : x / x.sum()))
display(Markdown("**For each country, what fraction is its population within␣
↪region:**"))
display(pop_within_group.head())
```

For each country, what fraction is its population within region:

```
Country
Afghanistan       0.02
Albania           0.00
Algeria           0.09
American Samoa    0.00
Andorra           0.00
Name: Population2020, dtype: float64
```

Lambda expression `lambda x : x / x.sum()` above is a shorthand for defining a function which gets `x` and returns `x / x.sum()`. Below we show a version with function explictly defined.

[24]:
```
def group_fraction(x):
    return x / x.sum()
pop_within_group = (countries.groupby('Region')['Population2020']
                    .transform(group_fraction))
display(Markdown("**For each country, what fraction is its population within␣
↪region:**"))
display(pop_within_group.head())
```

For each country, what fraction is its population within region:

```
Country
Afghanistan       0.02
Albania           0.00
Algeria           0.09
American Samoa    0.00
Andorra           0.00
Name: Population2020, dtype: float64
```

[25]:
```
display(Markdown("**Add back region name using concat:**"))
pop_within_group2 = pd.concat([pop_within_group, countries['Region']], axis=1)
display(pop_within_group2.head())

display(Markdown("**Look up value for Slovakia:**"))
display(pop_within_group2.loc["Slovak Republic"])
```

Add back region name using concat:

```
                Population2020                    Region
```

```
Country
Afghanistan                 0.02                South Asia
Albania                     0.00       Europe & Central Asia
Algeria                     0.09  Middle East & North Africa
American Samoa              0.00         East Asia & Pacific
Andorra                     0.00       Europe & Central Asia
```

Look up value for Slovakia:

```
Population2020                       0.01
Region              Europe & Central Asia
Name: Slovak Republic, dtype: object
```

[26]:
```
display(Markdown("**Check that the sum of each region is 1:**"))
display(pop_within_group2.groupby('Region').sum())
```

Check that the sum of each region is 1:

```
                            Population2020
Region
East Asia & Pacific                   1.00
Europe & Central Asia                 1.00
Latin America & Caribbean             1.00
Middle East & North Africa            1.00
North America                         1.00
South Asia                            1.00
Sub-Saharan Africa                    1.00
```

1.3.3 Filtering in the apply step

Finally, groupby can be followed by filter to use only some of the groups in the result.

Here we report all countries in regions that have at least one billion inhabitants.

[27]:
```
# filter gets a function returning a boolean value for each group
filtered = (countries.groupby("Region")
            .filter(lambda x : x['Population2020'].sum() > 1e9))
display(Markdown("**Filtered data:**"))
display(filtered.head())
display(Markdown("**Check sums in regions for selected countries:**"))
display(filtered.groupby('Region')['Population2020'].sum())
```

Filtered data:

```
               ISO3               Region         Income Group  Population2000  \
Country
Afghanistan     AFG           South Asia           Low income   19,542,983.00
American Samoa  ASM  East Asia & Pacific          High income       58,229.00
Angola          AGO   Sub-Saharan Africa  Lower middle income   16,394,062.00
Australia       AUS  East Asia & Pacific          High income   19,028,802.00
Bangladesh      BGD           South Asia  Lower middle income  129,193,327.00
```

```
                 Population2010  Population2020          Area   GDP2000  \
Country
Afghanistan       28,189,672.00   38,972,231.00    652,860.00       NaN
American Samoa        54,849.00       46,189.00        200.00       NaN
Angola            23,364,186.00   33,428,486.00  1,246,700.00    556.88
Australia         22,031,750.00   25,649,247.00  7,741,220.00 21,870.42
Bangladesh       148,391,139.00  167,420,950.00    147,570.00    413.10

                  GDP2010   GDP2020  Expectancy2000  Expectancy2010  \
Country
Afghanistan        562.50    512.06           55.30           60.85
American Samoa  10,446.86 15,609.78             NaN             NaN
Angola           3,586.66  1,450.91           46.02           56.73
Australia       52,147.02 51,868.25           79.23           81.70
Bangladesh         776.86  2,233.31           65.78           68.64

                 Expectancy2020  Fertility2000  Fertility2010  Fertility2020
Country
Afghanistan               62.58           7.53           6.10           4.75
American Samoa              NaN            NaN            NaN            NaN
Angola                    62.26           6.64           6.19           5.37
Australia                 83.20           1.76           1.93           1.58
Bangladesh                71.97           3.22           2.34           2.00
```

Check sums in regions for selected countries:

```
Region
East Asia & Pacific    2,340,350,517.00
South Asia             1,882,531,621.00
Sub-Saharan Africa     1,151,302,077.00
Name: Population2020, dtype: float64
```

1.3.4 Grouping by multiple values

Function `groupby` can get a single column, but also a list of columns or a Series which will be used as if it was a column of the table.

[28]:
```
display(Markdown("**Populations split by both region and income group**"))
display(countries.groupby(['Region', "Income Group"])['Population2020'].sum())
```

Populations split by both region and income group

```
Region                      Income Group
East Asia & Pacific         High income              223,971,823.00
                            Low income                25,867,467.00
                            Lower middle income      301,779,468.00
                            Upper middle income    1,788,731,759.00
Europe & Central Asia       High income              522,292,344.00
                            Lower middle income       94,487,207.00
```

```
                              Upper middle income     305,573,814.00
Latin America & Caribbean     High income              34,033,357.00
                              Lower middle income      40,120,621.00
                              Upper middle income     547,890,556.00
Middle East & North Africa    High income              68,156,525.00
                              Low income               53,056,642.00
                              Lower middle income     304,739,289.00
                              Upper middle income      54,014,194.00
North America                 High income             369,582,572.00
South Asia                    Low income               38,972,231.00
                              Lower middle income   1,843,044,952.00
                              Upper middle income         514,438.00
Sub-Saharan Africa            High income                  98,462.00
                              Low income              549,157,331.00
                              Lower middle income     533,054,222.00
                              Upper middle income      68,992,062.00
Name: Population2020, dtype: float64
```

- Now we create a Series classifying each country as small, medium and large using cutoff 1 million for small and 100 million for medium.
- We then use this series in `groupby`.
- The classification is created by `pd.cut` function.

[29]:
```
bin_ends = [0, 1e6, 1e8, 1e10]
bin_labels = ["small", "medium", "large"]
size_groups = pd.cut(countries['Population2020'],
                     bins=bin_ends, labels=bin_labels).rename("SizeCategory")
display(Markdown("**Country size classification:**"))
display(size_groups.head())
```

Country size classification:

```
Country
Afghanistan       medium
Albania           medium
Algeria           medium
American Samoa     small
Andorra            small
Name: SizeCategory, dtype: category
Categories (3, object): ['small' < 'medium' < 'large']
```

Now we can use `size_groups` Series in `groupby`.

Parameter `observed=True` is related to the fact that `size_groups` is has a categorial variable type to be explained next.

[30]:
```
# now use size_groups in groupby
display(Markdown("**The number of countries in each size group:**"))
display(countries.groupby(size_groups, observed=True).size())
display(Markdown("**The number of countries in each size group and region:**"))
```

```
display(countries.groupby(['Region', size_groups], observed=True).size())
```

The number of countries in each size group:

```
SizeCategory
small      57
medium    146
large      14
dtype: int64
```

The number of countries in each size group and region:

```
Region                      SizeCategory
East Asia & Pacific         small           18
                            medium          15
                            large            4
Europe & Central Asia       small           12
                            medium          45
                            large            1
Latin America & Caribbean   small           19
                            medium          21
                            large            2
Middle East & North Africa  small            1
                            medium          19
                            large            1
North America               small            1
                            medium           1
                            large            1
South Asia                  small            2
                            medium           3
                            large            3
Sub-Saharan Africa          small            4
                            medium          42
                            large            2
dtype: int64
```

1.4 Categorical variables

Categorical variables have values from a small set, such as region and income group in the table of countries. So far we have represented them only as strings, but we can explicitly convert them to a categorical data type in Pandas.

This has several advantages: * Strings are internally replaced by numerical IDs within the table, potentially saving memory. * Categories can be ordered and then sorting, minimum, maximum etc works as desired, not alphabetically. * Pandas is aware of the full set of possible values. For example categories without members can appear in the `groupby` results.

Example Income groups in our table are strings, we will convert them to an ordered categorical variable.

[31]:
```
# creating a categorical type
cat_type = pd.api.types.CategoricalDtype(categories=["Low income",
                                                     "Lower middle income",
                                                     "Upper middle income",
                                                     "High income"],
                                         ordered=True)
# converting Income Group column to cat_type in a new DataFrame
countries_cat = countries.astype({'Income Group': cat_type})

display(Markdown("**Income Group column in the old table:**"),
        countries['Income Group'].head(3))
display(Markdown("**Income Group column in the new table:**"),
        countries_cat['Income Group'].head(3))
```

Income Group column in the old table:

```
Country
Afghanistan             Low income
Albania        Upper middle income
Algeria        Lower middle income
Name: Income Group, dtype: object
```

Income Group column in the new table:

```
Country
Afghanistan             Low income
Albania        Upper middle income
Algeria        Lower middle income
Name: Income Group, dtype: category
Categories (4, object): ['Low income' < 'Lower middle income' < 'Upper middle␣
↪income' < 'High income']
```

[32]:
```
display(Markdown("**Minimum and maximum income group in the table with␣
↪categorical values:**"
                 "  (manually fixed order):"))
display(countries_cat['Income Group'].min())
display(countries_cat['Income Group'].max())

display(Markdown("**Minimum and maximum income group in the table with␣
↪strings**"
                 "  (alphabetical order):"))
display(countries['Income Group'].dropna().min())
display(countries['Income Group'].dropna().max())
```

Minimum and maximum income group in the table with categorical values: (manually fixed order):

```
'Low income'
```

```
'High income'
```

Minimum and maximum income group in the table with strings (alphabetical order):

'High income'

'Upper middle income'

- Note that if categories do not need a fixed order, they can be created automatically by the astype function as in the code below.
- Notice that groupby creates even empty groups which would not happen with strings. This is caused by observed=False setting.

[33]:
```
# convert region to an unordered category
countries_cat2 = countries_cat.astype({'Region': 'category'})
# count the number of countries for each combination of income group and region
countries_cat2.groupby(['Income Group', 'Region'], observed=False).size()
```

[33]:
```
Income Group         Region
Low income           East Asia & Pacific            1
                     Europe & Central Asia          0
                     Latin America & Caribbean      0
                     Middle East & North Africa     2
                     North America                  0
                     South Asia                     1
                     Sub-Saharan Africa            22
Lower middle income  East Asia & Pacific           13
                     Europe & Central Asia          4
                     Latin America & Caribbean      4
                     Middle East & North Africa     8
                     North America                  0
                     South Asia                     6
                     Sub-Saharan Africa            19
Upper middle income  East Asia & Pacific            9
                     Europe & Central Asia         16
                     Latin America & Caribbean     19
                     Middle East & North Africa     3
                     North America                  0
                     South Asia                     1
                     Sub-Saharan Africa             6
High income          East Asia & Pacific           14
                     Europe & Central Asia         38
                     Latin America & Caribbean     18
                     Middle East & North Africa     8
                     North America                  3
                     South Asia                     0
                     Sub-Saharan Africa             1
dtype: int64
```

1.5 Dates and times

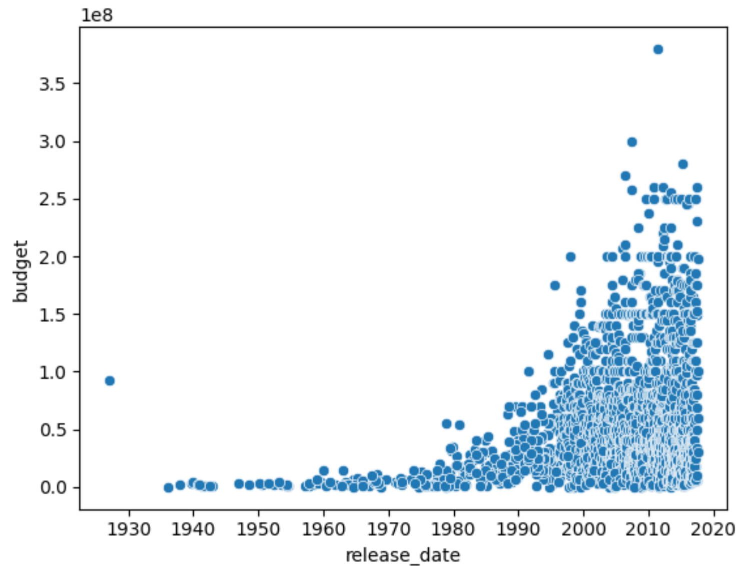
An important type of data sets are time series, where some variables are measured repeatedly over time. Pandas has an extensive support for work with times and dates. Here we show only a small example.

- We illustrate this on the movie dataset from Kaggle (see lecture 04).
- The column labeled `release_date` is recognized as date by passing `parse_dates` parameter to `read_csv`.
- Then we call function `day_name()` to get the day of week for each release day and use `value_counts` to see which days are most frequent as movie release dates.
- We also use the release date as the x-coordinate in a scatterplot.

[34]:
```
# import data, including parsing of dates
url = 'https://bbrejova.github.io/viz/data/Movies_small.csv'
movies = pd.read_csv(url, parse_dates=['release_date'])
# get days of week for realse dates
days = movies['release_date'].apply(lambda x : x.day_name())
days.value_counts()
```

[34]:
```
release_date
Friday       639
Thursday     515
Wednesday    474
Tuesday      175
Saturday      94
Monday        87
Sunday        65
Name: count, dtype: int64
```

[35]:
```
# use release date is x-coordinate
sns.scatterplot(data=movies, x='release_date', y='budget')
pass
```



1.6 Missing values

Data sets are often incomplete, and Pandas provides techniques for working with missing data.

- Missing data are typically imported as `np.nan` (not-a-number).
- These cannot occur in int-type columns, so ints are converted to floats, but can be handled in a special way.

Bellow we show a small example what happens when working with missing data, including functions `isna`, `dropna`, `fillna`.

[36]:
```
# create a small series with one missing value
a = pd.Series([1, 2, np.nan, 3])
display(Markdown("**`a.sum()` skips missing values:**"),
        a.sum())
display(Markdown("**`a.count()` counts non-missing values:**"),
        a.count())
display(Markdown("**`a.mean()` also considers only non-missing:**"),
        a.mean())
display(Markdown("**`a > 2` evaluates missing values as `False`, similarly `<`,␣
 ↪`==`:**"),
        a > 2)
display(Markdown("**`a == np.nan` also evaluates as `False`:**"),
        a == np.nan)
display(Markdown("**`a.isna()` can be used to detect missing values:**"),
        a.isna())
display(Markdown("**`a.dropna()` omits missing values:**"),
        a.dropna())
display(Markdown("**`a.fillna(-1)` replaces them with a specified value:**"),
```

```
        a.fillna(-1))
```

`a.sum()` skips missing values:

```
6.0
```

`a.count()` counts non-missing values:

```
3
```

`a.mean()` also considers only non-missing:

```
2.0
```

`a > 2` evaluates missing values as `False`, similarly `<`, `==`:

```
0    False
1    False
2    False
3     True
dtype: bool
```

`a == np.nan` also evaluates as `False`:

```
0    False
1    False
2    False
3    False
dtype: bool
```

`a.isna()` can be used to detect missing values:

```
0    False
1    False
2     True
3    False
dtype: bool
```

`a.dropna()` omits missing values:

```
0   1.00
1   2.00
3   3.00
dtype: float64
```

`a.fillna(-1)` replaces them with a specified value:

```
0    1.00
1    2.00
2   -1.00
3    3.00
dtype: float64
```

1.7 Pandas efficiency

Below we show several examples how different ways of implementing the same operation can have very different running time on large data. Pandas functions are usually much faster than manual iteration. However, if you do not work on huge data sets, the difference is not so important.

To measure time, we use a special Jupyter command `%timeit`. * It runs the code several times to estimate the time per one repeat.

[37]:
```
# generate a Series of million random numbers and also convert it to Python list
length = int(1e6)
xs = pd.Series(np.random.uniform(0,100, length))
xl = list(xs)
```

Below we see that method `sum()` on Series is faster than Python built-in `sum` on a Python list, but Python built-in `sum` on Series is much slower, because it iterates over elements of Series.

[38]:
```
display(Markdown("**Method `sum` on `Series` `xs.sum()`:**"))
%timeit result = xs.sum()
display(Markdown("**Python `sum` on Python list `sum(xl)`:**"))
%timeit result = sum(xl)
display(Markdown("**Python `sum` on Series `sum(xs)`:**"))
%timeit result = sum(xs)
```

Method `sum` on `Series` `xs.sum()`:

```
1.12 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Python `sum` on Python list `sum(xl)`:

```
6.77 ms ± 401 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Python `sum` on Series `sum(xs)`:

```
58.3 ms ± 351 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Below we compare three ways of generating a sequence of squared values. Multiplying Series with `*` is the fastest, Python list comprehension is much slower and `apply` function from Pandas is even slower.

[39]:
```
display(Markdown("**Pandas `Series` multiplication `x2s = xs * xs`**:"))
%timeit x2s = xs * xs
display(Markdown("**Python list comprehension on a list `x2l = [x * x for x in␣
 ↪xl]`:**"))
%timeit x2l = [x * x for x in xl]
display(Markdown("**Pandas `apply` function `x2s = xs.apply(lambda x : x *␣
 ↪x)`**"))
%timeit x2s = xs.apply(lambda x : x * x)
```

Pandas `Series` multiplication `x2s = xs * xs`:

```
1.67 ms ± 50.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Python list comprehension on a list `x2l = [x * x for x in xl]`:

```
53.6 ms ± 6.34 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Pandas apply function `x2s = xs.apply(lambda x : x * x)`

```
234 ms ± 12.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The code below creates the Series of squares by creating a Series filled with zeroes and then assigning individual values using for-loop. This is again much slower than all methods above, so to make the code reasonably fast, we run it on data which is 100 times smaller than above.

[40]:
```
length2 = 10000
xs_small = xs.iloc[0:length2]
def assignments(len, x):
  x2 = pd.Series([0.0] * len)
  for i in range(len):
    x2[i] = x[i] * x[i]
  return x2
%timeit x2s_small = assignments(length2, xs_small)
```

```
173 ms ± 1.02 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Finally the code below is even worse. It appends individual squares to a Series which starts with size 1. We run it on even smaller list of size 1000.

[41]:
```
length3 = 1000
xs_tiny = xs.iloc[0:length3]
def assignments(len, x):
  x2 = pd.Series([0.0])
  for i in range(len):
    x2[i] = x[i] * x[i]
  return x2
%timeit x2s_tiny = assignments(length3, xs_tiny)
```

```
286 ms ± 16 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Lecture 6
Maps, graphs, time series

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

More details in the notebook version

Part I: Maps

Maps

Each map is a visualization of data about location of objects.

Conventions about colors and symbols, orientation etc. allow us to quickly understand a map.

Example:
A topographic map from US

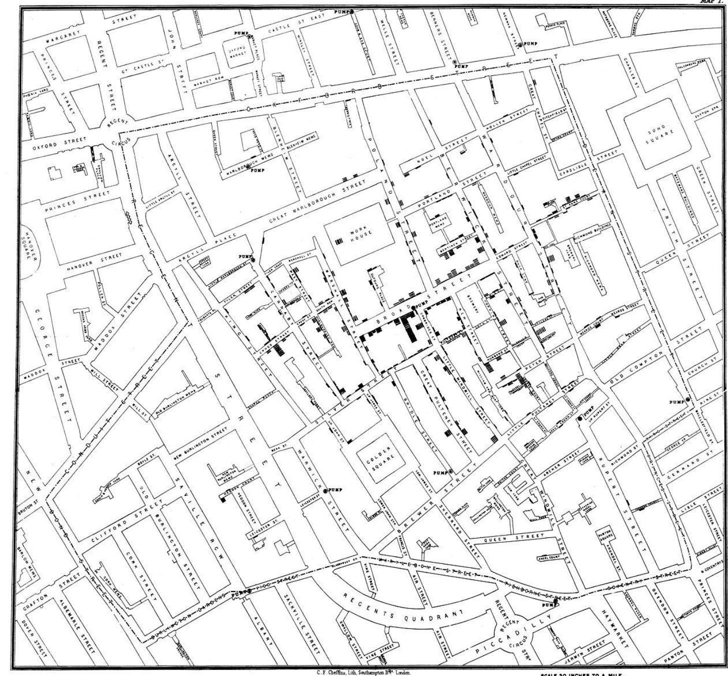
https://en.wikipedia.org/wiki/Map#/media/File:Topographic_map_example.png

Data visualization in maps

Thematic maps (tematické mapy) visualize data other than typical geographical features

Recall Snow's map of cholera cases (1854)

Additional examples: [Wikipedia](), [GeoPlot library gallery]().



https://commons.wikimedia.org/wiki/File:Snow-cholera-map-1.jpg

Map projection (kartografické zobrazenie)

A transformation to project the surface of a globe onto a plane

Each projection introduces some distortion

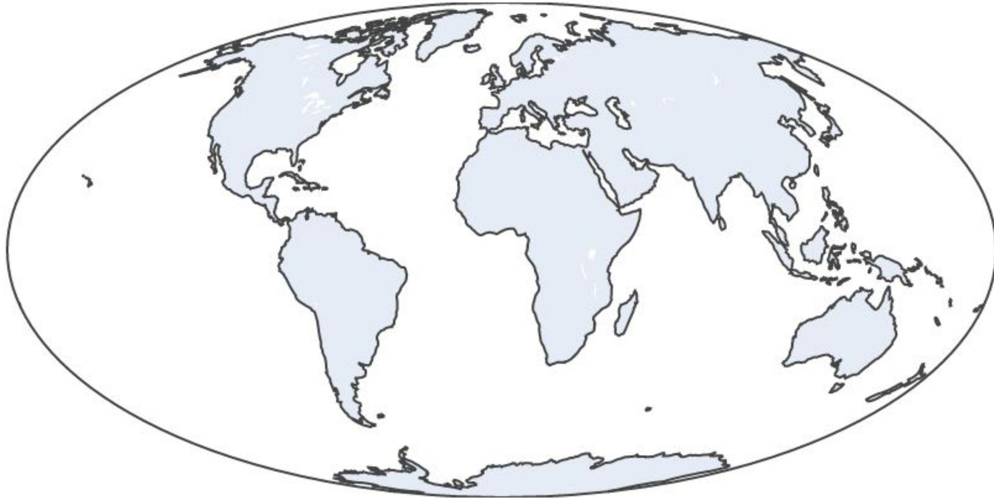
Conformal projections preserve local angles, but distort other aspects, such as lengths, areas etc.

For example, **Mercator projection** (1569) developed for navigation, but shows Greenland bigger than Africa, while in fact it is 14x smaller.

Map projection (kartografické zobrazenie)

Equal-area projections preserve areas (cannot be conformal at the same time).

These are typically good for data visualization, as they make areas comparable.



Example: **Mollweide equal-area projection** (1805)

Map projection (kartografické zobrazenie)

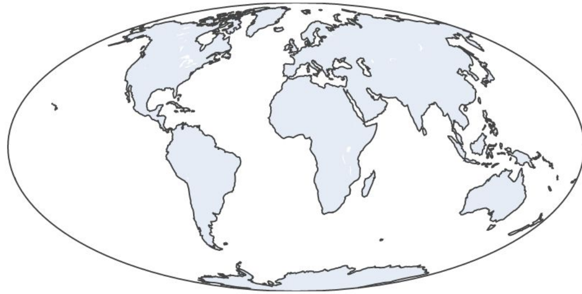
Orthographic projection is similar to a photograph of the Earth from a very distant point.

It is not an equal-area projection, but our sense of perspective may compensate.

It displays one hemisphere.

Recommended projections (Cairo, The Truthful Art)

Whole world: e.g. Mollweide equal-area projection (1805)
Continents / large countries: e.g. Lambert azimuthal equal-area projection (1772)
Countries in mid-latitudes: e.g. Albers equal-area conic projection (1805)
Polar regions: e.g. Lambert azimuthal equal-area projection (1772)



Mollweide

Lambert

Projection examples in Plotly

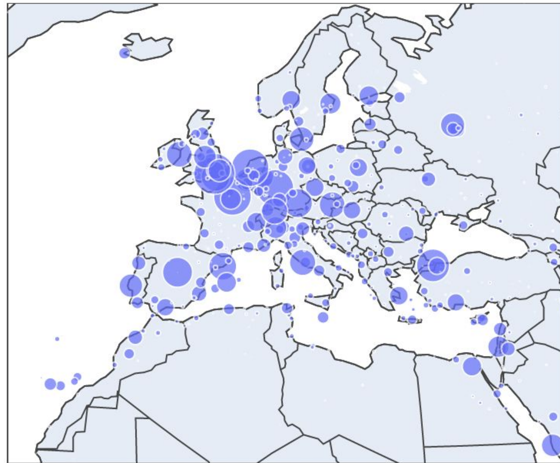
```
def show_world(projection, scope=None):
  # create a map figure with an empty scatterplot
  fig = go.Figure(go.Scattergeo())
  # set the desired projection
  fig.update_geos(projection_type=projection)
  # we can also limit the scope of the map
  if scope is not None:
    fig.update_geos(scope=scope)
  # finally, make the image smaller and with 0 margins
  fig.update_layout(height=200,
                    margin={"r":0,"t":0,"l":0,"b":0})
  # show the figure
  fig.show()
```

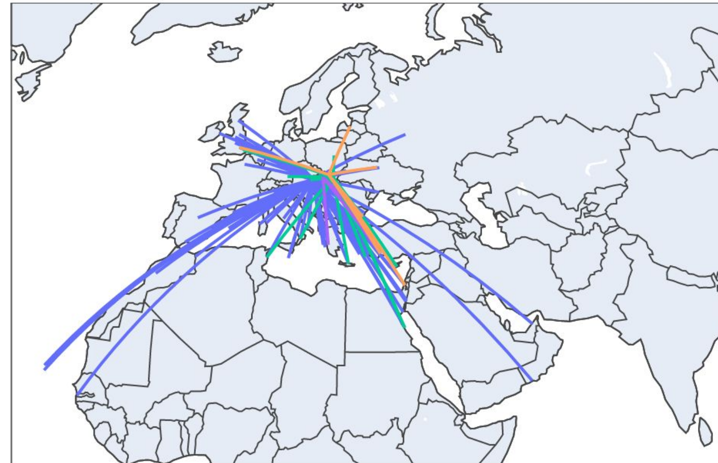
```
show_world("azimuthal equal area", "europe")
```

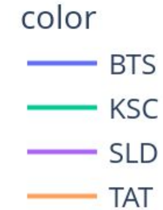
Adding data as points and lines to a map

Geographic coordinates of places can be projected as x and y.
Additional values can be shown using marker color / size or line color / width.

Example: airport locations in Europe and airline connections from Slovakia.







Our airport dataset

- The dataset of 2173 international airports of the world from the World Bank under the CC-BY 4.0 license, and preprocessed.
- For each airport its 3-letter code, name, country, 3-letter code of the country, the number of airplane seats per year (from unknown years) and the location.
- Stored in **GeoJSON** format.
- We parse the file using **GeoPandas** library for working with geographical data.
- It is an extension of Pandas DataFrame, with location information.

Importing airports in GeoPandas

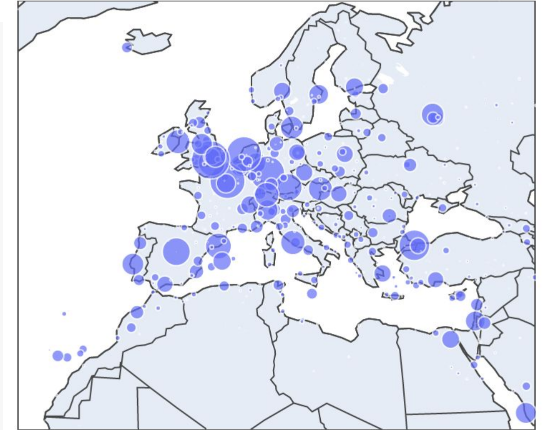
```
import geopandas as gpd
```

```
airports = gpd.read_file("https://bbrejova.github.io/viz/data/airports.geojson")
display(airports.query('Country == "Slovakia"'))
```

	Orig	Name	TotalSeats	Country	ISO3	geometry
1489	BTS	M.R. Stefanik	1211732.116	Slovakia	SVK	POINT (17.21670 48.16670)
1490	ILZ	Zilina	3986.360	Slovakia	SVK	POINT (18.76670 49.23330)
1491	KSC	Barca	323259.132	Slovakia	SVK	POINT (21.25000 48.66670)
1492	PZY	Piestany Airport	1403.892	Slovakia	SVK	POINT (17.83330 48.63330)
1493	SLD	Sliac	11876.753	Slovakia	SVK	POINT (19.13330 48.63330)
1494	TAT	Tatry/Poprad	39612.286	Slovakia	SVK	POINT (20.24030 49.07190)

Drawing airport bubble graph in Plotly

```
fig = px.scatter_geo(
    airports,
    lat=airports.geometry.y,
    lon=airports.geometry.x,
    size="TotalSeats",
    hover_name="Name"
    )
fig.update_geos(
    projection_type="azimuthal equal area",
    lonaxis_range= [-20, 40],
    lataxis_range= [20, 70],
    showcountries = True
)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```



Interactive plot: zooming, panning, tooltips

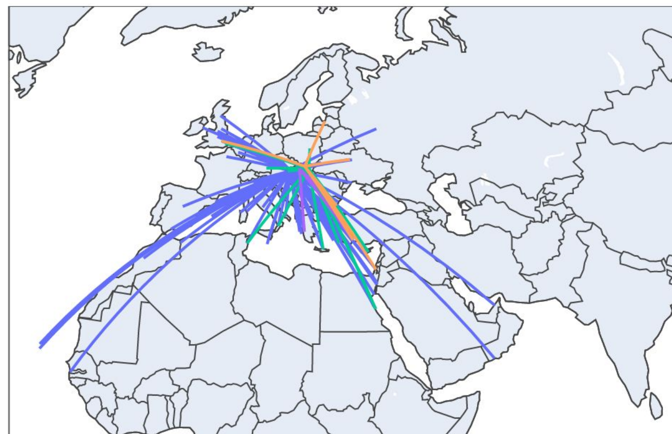
Importing airport connections in GeoPandas

```
connections = gpd.read_file("https://bbrejova.github.io/viz/data/airport_pairs_svk.geojson")
display(connections.head())
```

	OrigCode	DestCode	TotalSeats	geometry
0	BTS	ADB	7370.433	LINESTRING (17.21670 48.16670, 27.15620 38.29430)
1	BTS	AGP	15152.501	LINESTRING (17.21670 48.16670, -4.49810 36.67170)
2	BTS	AHO	14740.866	LINESTRING (17.21670 48.16670, 8.28890 40.63060)
3	BTS	AQJ	3275.748	LINESTRING (17.21670 48.16670, 35.01940 29.61250)
4	BTS	ATH	19654.488	LINESTRING (17.21670 48.16670, 23.94440 37.93640)

Drawing airline connections in Plotly

```
lats = []
lons = []
origCodes = []
destCodes = []

for index, row in connections.iterrows():
  x, y = row['geometry'].xy
  lats.extend(list(y) + [None])
  lons.extend(list(x) + [None])
  origCodes.extend([row['OrigCode']] * len(x) + [None])
  destCodes.extend([row['DestCode']] * len(x) + [None])

fig = px.line_geo(lat=lats, lon=lons, hover_name=destCodes, color=origCodes)
fig.update_geos(
  projection_type="azimuthal equal area",
  lonaxis_range= [-25, 55],
  lataxis_range= [10, 60],
  showcountries = True
)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```



color

BTS

KSC

SLD

TAT

Displaying variables that vary over space

Elevation can be measured at any point on land.

How is it visualized on geographic maps?

Displaying variables that vary over space

Elevation can be measured at any point on land.

How is it visualized on geographic maps?

https://upload.wikimedia.org/wikipedia/commons/d/d9/Slovakia_general_relief_map.svg

https://en.wikipedia.org/wiki/Map#/media/File:Topographic_map_example.png

Isarithmic maps / isoline maps / heatmaps

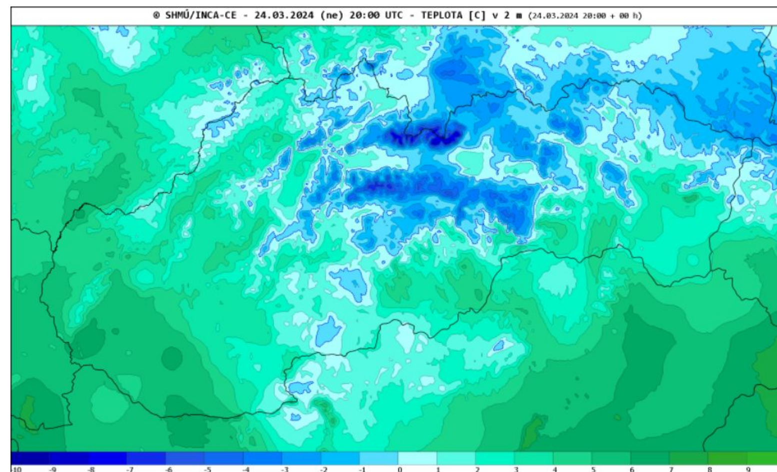
Display a continuous variable over the map area (elevation, temperature etc.)

Value in each point can be shown by a color scale.
Some contour lines can be displayed as well.

A **contour line** (isoline, isopleth, isarithm, izočiara)
connects points of the same value.

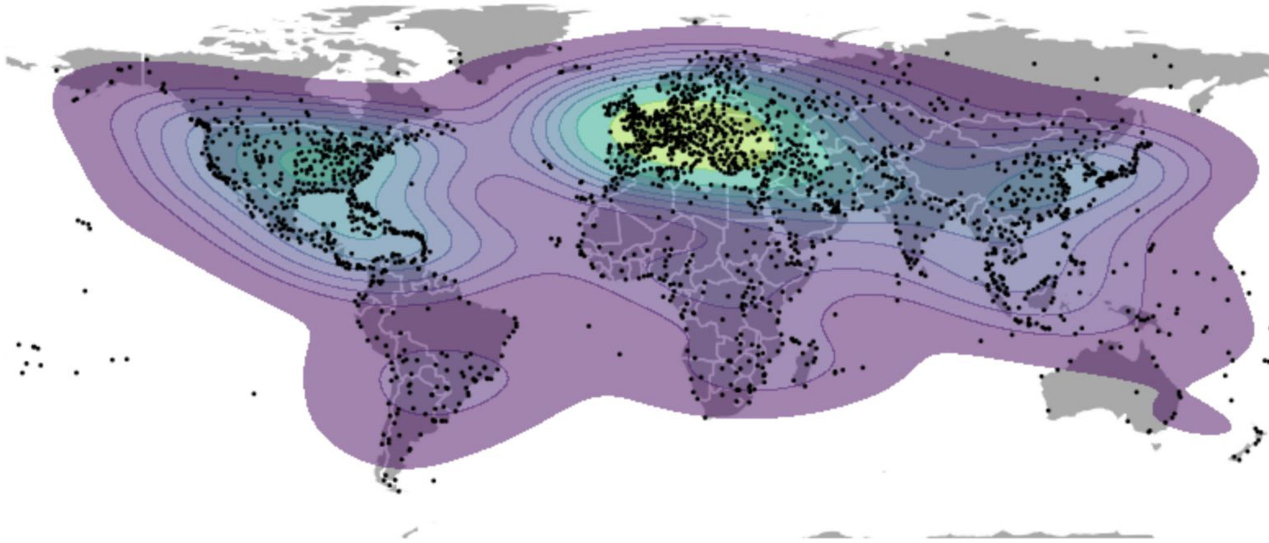
Example: short-term forecasts from
the Slovak Hydrometeorological Institute.

https://www.shmu.sk/sk/?page=1&id=meteo_inca_base



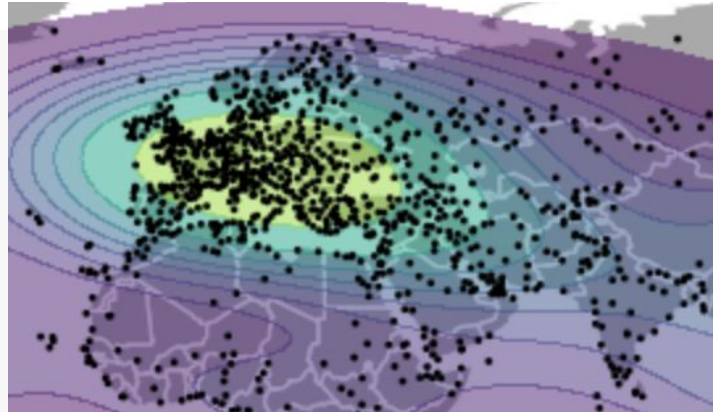
Density of airports as a isarithmic map

We use kdeplot function from the Geoplot library.
KDE stands for kernel density estimation (next lecture).



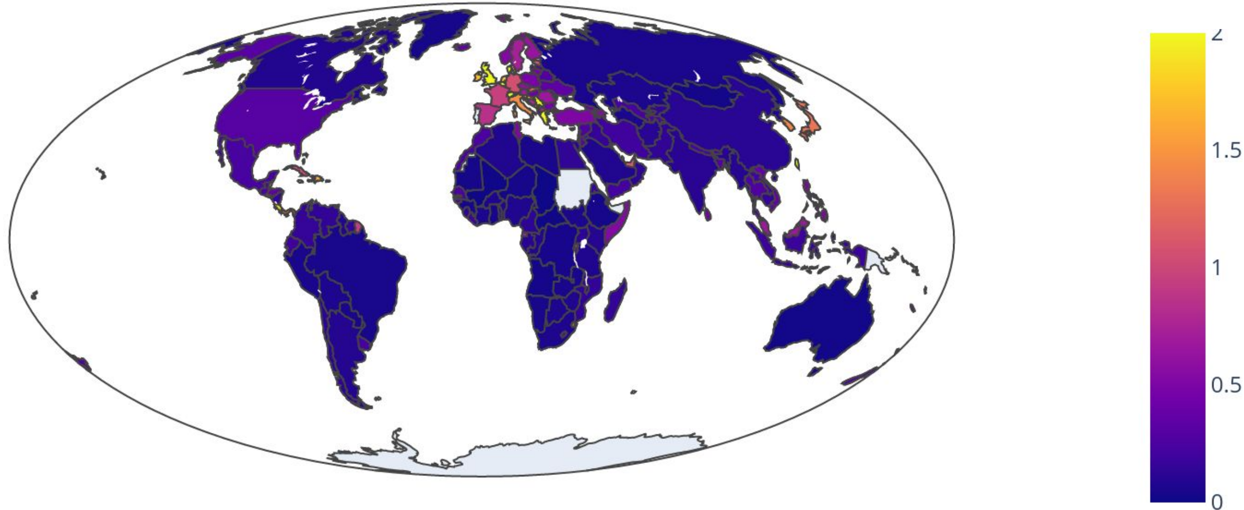
Density of airports in Geoplot library

```
import geoplot as gplt
# plot countries as a background
ax = gplt.polyplot(
    countries.explode(index_parts=True),
    edgecolor='white',
    facecolor='darkgray',
    figsize=(10, 5),
)
# plot semi-transparent isarithmic map
gplt.kdeplot(
    airports, cmap='viridis',
    fill=True, alpha=0.5, ax=ax
)
# plot points on top
gplt.pointplot(airports, s=1, color='black', ax=ax
pass
```



Choropleth maps (kartogramy)

These show numerical / categorical values for administrative regions (countries, districts, etc.) via colors applied to the whole region.
Example: The number of airports in a country per 10000 km^2



Types of variables over regions

Spatially extensive: apply to the unit as a whole. If we subdivide the region, spatially extensive variable will be often the sum of its parts.
Examples: total population, area, the number of airports in the country

Spatially intensive: may stay the same if you divide the unit, provided the unit is homogeneous without regional differences.
Examples: population density, life expectancy, GDP per person.

Spatially extensive variables are not appropriate for choropleths
(large value for a large country is visually attributed to each small subregion)

Computing airport stats per country in GeoPandas

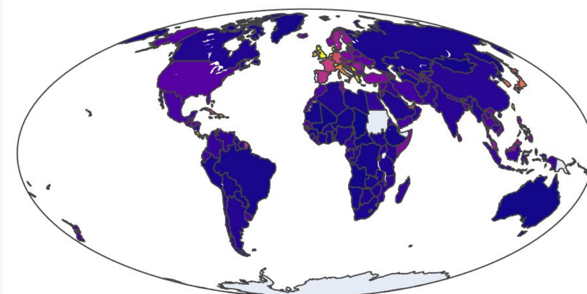
```
# compute the number of airports per country by groupby
airports_per_country = airports.groupby('ISO3').size()
# add the new column to a copy of the old table
countries2 = countries.copy(deep=True)
countries2['Airports'] = airports_per_country
# remove countries where airports or location are missing
countries2.dropna(subset=['geometry', 'Airports'], inplace=True)
# add columns with airport density and airports per million people
countries2['Airport_density'] = (countries2['Airports']
                                 / countries2['Area'] * 10000)
countries2['Airports_per_mil'] = (countries2['Airports']
                                  / countries2['Population'] * 1e6)
display(countries2.loc['SVK'])
```

Computing airport stats per country in GeoPandas

```
Type                                           Sovereign country
Name                                                    Slovakia
Population                                             5454073.0
Region                                     Europe & Central Asia
geometry            POLYGON ((22.558137648211755 49.0857380234671...
Area                                                47069.779734
Airports                                                     6.0
Airport_density                                         1.274703
Airports_per_mil                                        1.100095
Name: SVK, dtype: object
```

Drawing choropleth map in Plotly

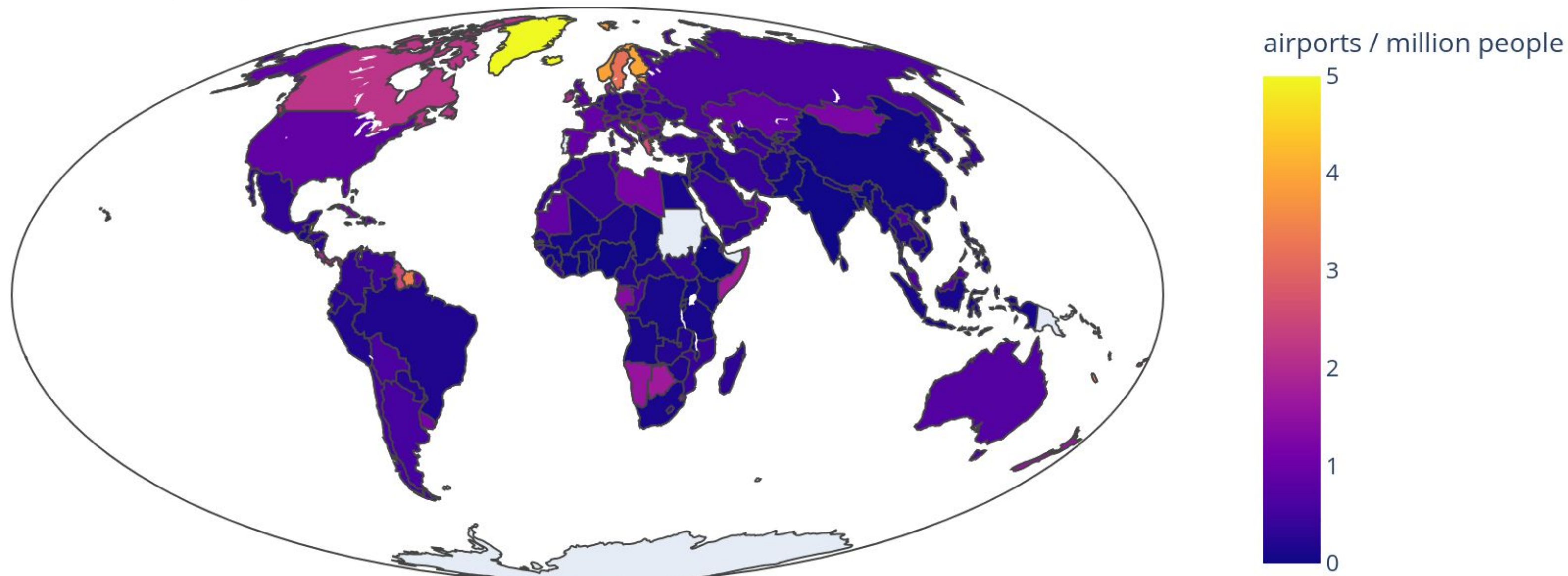
```
def draw_choropleth(data, column, range_color=None, label=None):
  fig = px.choropleth(
    data, locations=data.index, color=column,
    range_color=range_color,
    labels={column:label},
    hover_name="Name",
    projection = "mollweide"
  )
  fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
  fig.show()

display(Markdown("**The number of airports per 10000 squared km**"))
draw_choropleth(countries2, 'Airport_density', (0, 2), 'airports / 10000 km2')
```



Another intensive variable

```
display(Markdown("**The number of airports per million inhabitans**"))
draw_choropleth(countries2, 'Airports_per_mil', (0, 5), 'airports / million people')
```

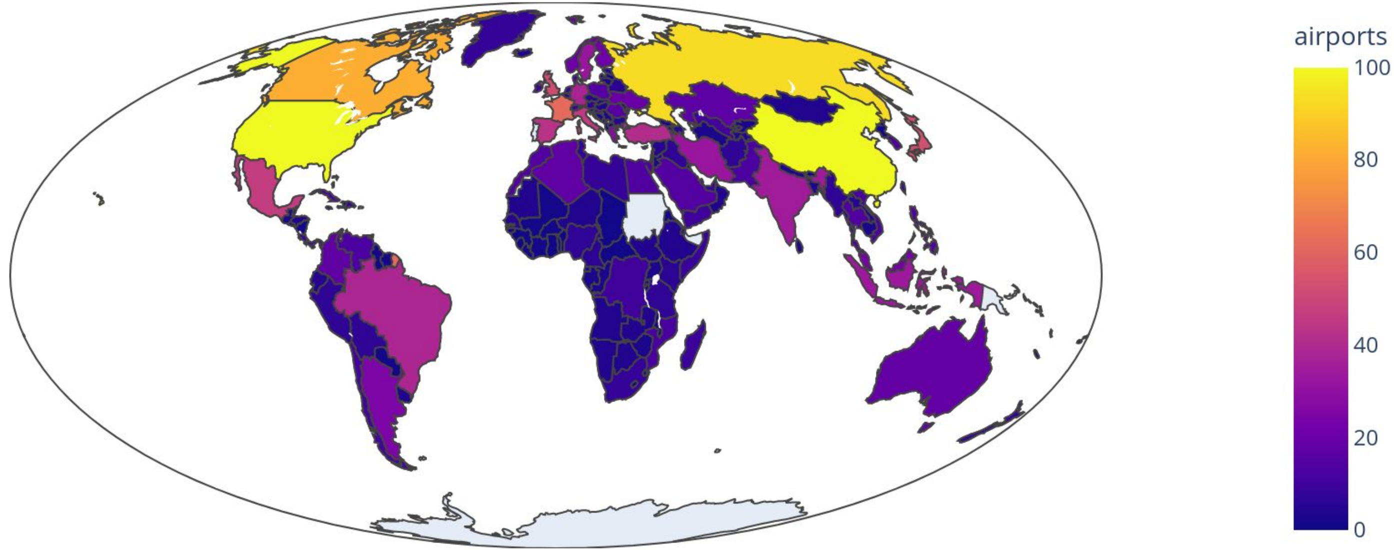
The number of airports per million inhabitans



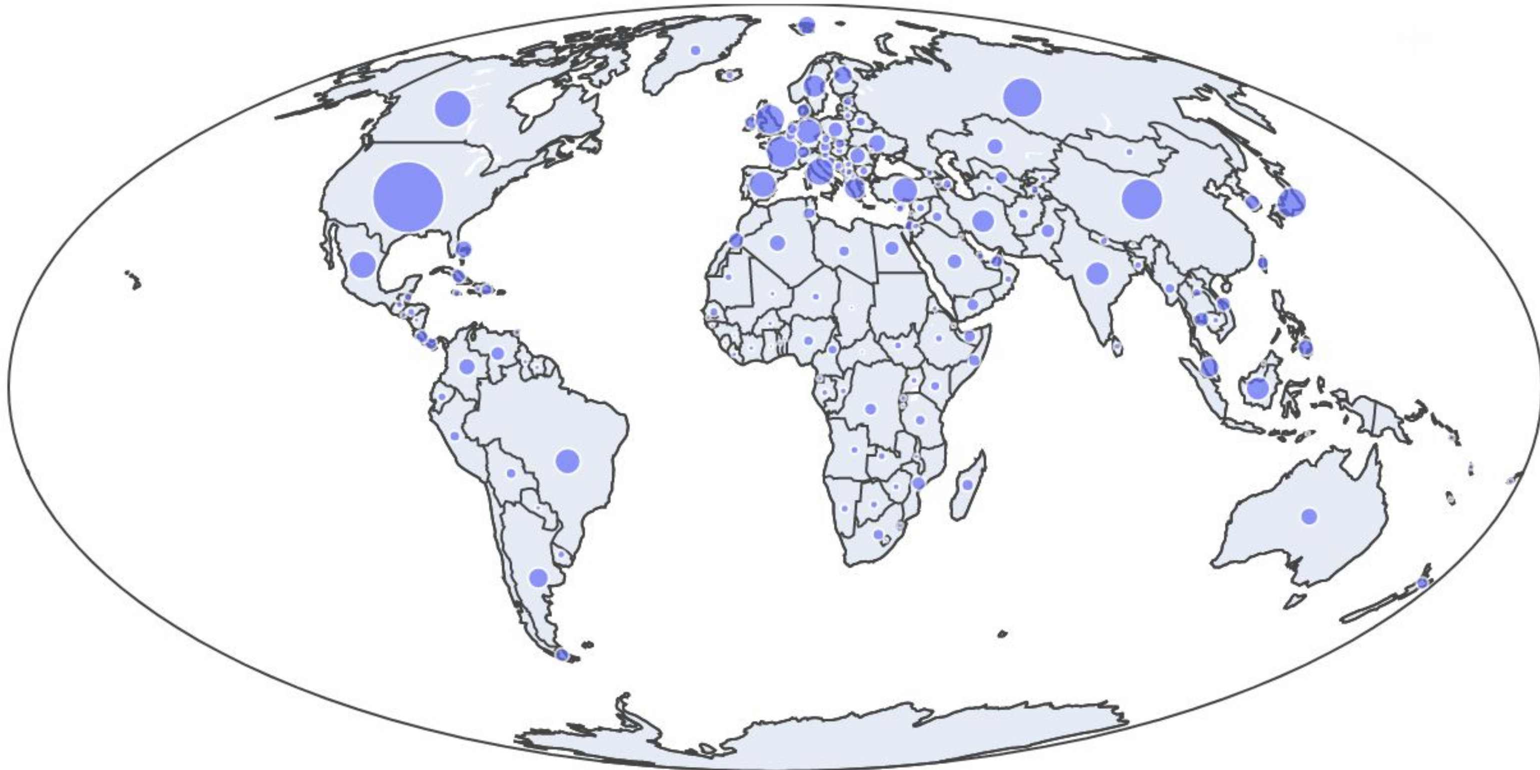
The number of airports is extensive (not good)

```
display(Markdown("**The number of airports in a country**"))
draw_choropleth(countries2, 'Airports', (0, 100), 'airports')
```

The number of airports in a country



The number of airports as a bubble plot (better)



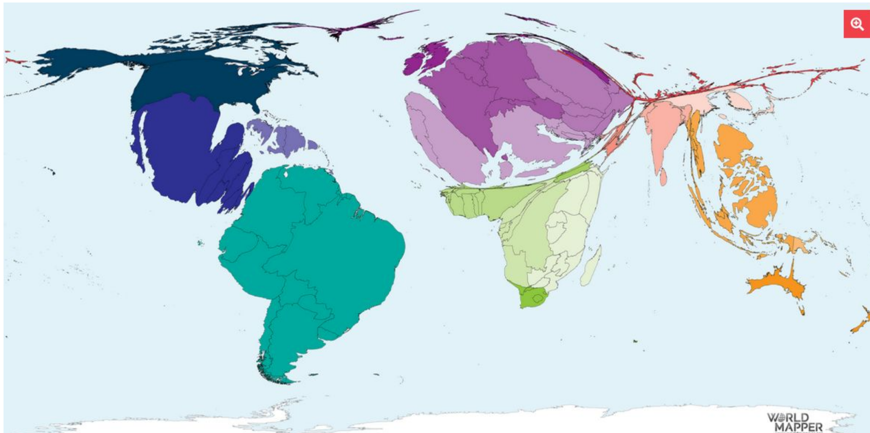
Cartogram vs kartogram

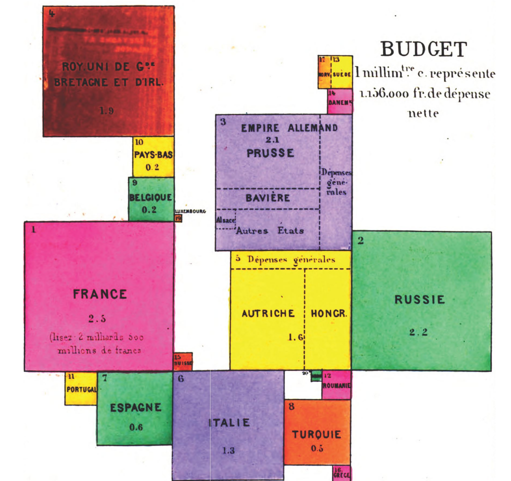
A choropleth map is called kartogram in Slovak.

In English, cartogram is a map with regions rescaled according to some variable

Examples: World's Catholic Population by World Mapper

Levasseur's cartogram of country budgets





https://worldmapper.org/maps/catholic-population-2005/

https://commons.wikimedia.org/wiki/File:Levasseur_cartogram.png

Summary of maps

- Many data sets contain geographic entities (countries, cities, coordinates)
- Displaying such data on maps shows spatial relationships
- Use appropriate equal-area projections
- Bubble plots: add points of various sizes
- Isarithmic maps / isoline maps / heatmaps: continuously varying variables
- Choropleth map: variables characterizing whole region
    - The variable should be intensive, e.g. normalized by area or population

Several useful libraries

- Geopandas for working with geographical data, extension of DataFrame
- Geoplot and Plotly for visualization

Part II: Graphs / networks

Graph / network

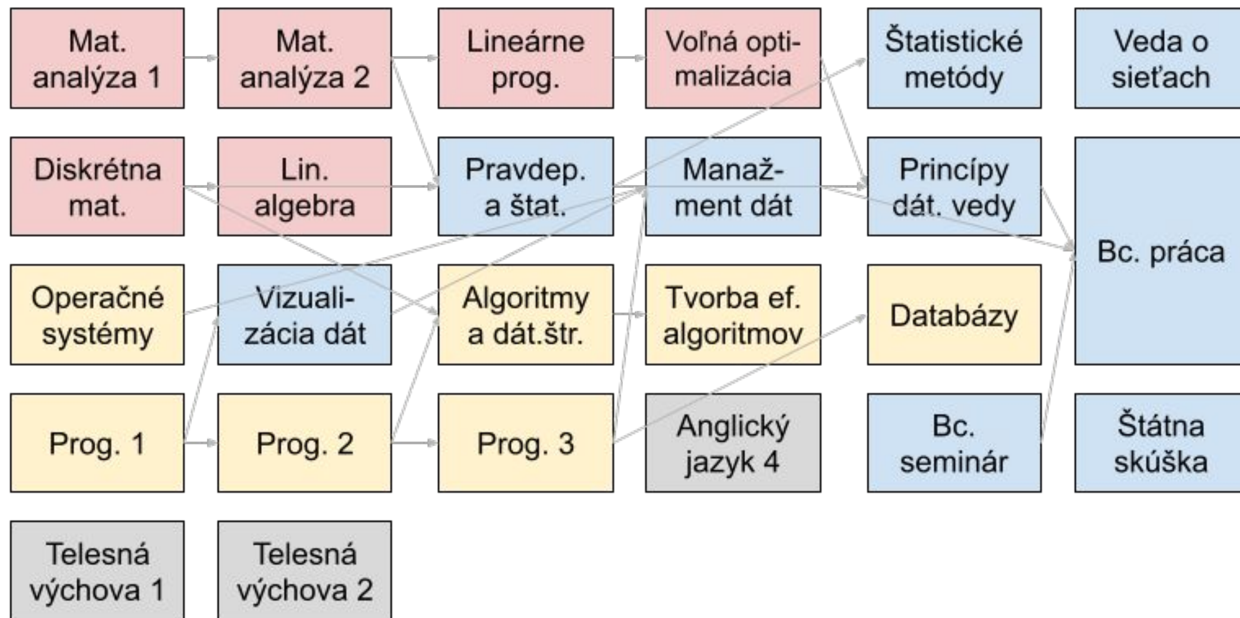
Vertices (vrcholy; also nodes, uzly) often real-world entities

Edges (hrany; also links, arcs) often relationships / connections between pairs of vertices

Many real-world examples:
places connected by roads, computers connected by network cables,
people connected by family or work relationships,
companies connected by financial transactions, texts connected by references,
tasks or courses connected by dependencies,
any objects connected based on similarity / shared features, ...

Graphs are very important in both computer science and data science.

Example: dependencies between DAV courses



Graph / network (cont.)

Edges: Directed (orientované) or undirected (for symmetric relationships)
Recall: how did you define directed / undirected edges in discrete mathematics?

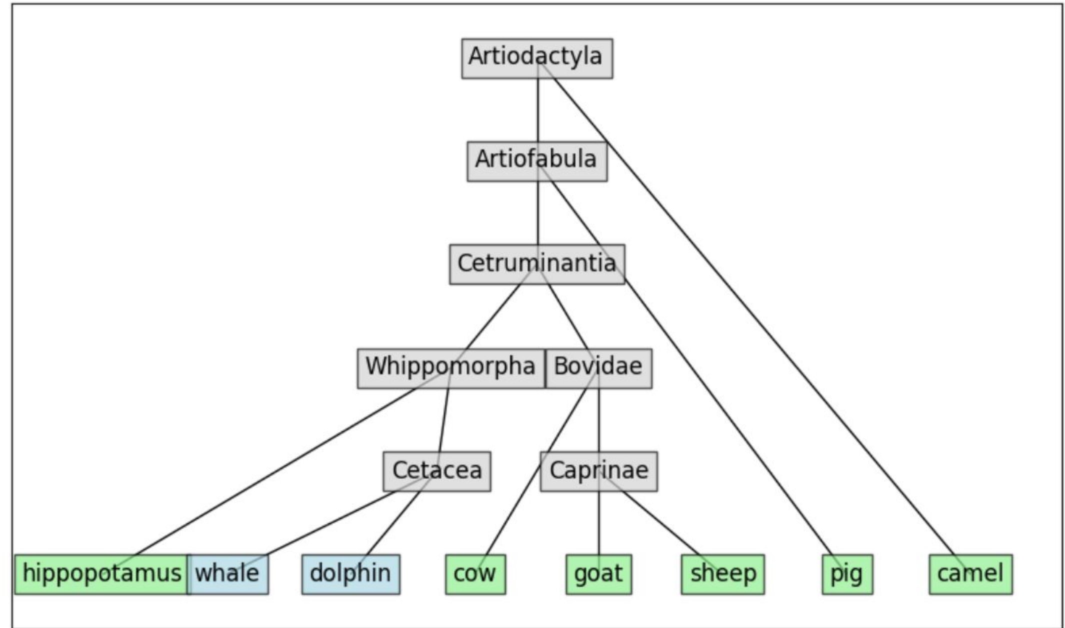
Graphs are covered in several courses: discrete mathematics, programming, design of efficient algorithms, network science.

Trees

An undirected graph is called a tree if it is connected and without cycles.

In practice we usually encounter rooted (directed) trees,
which have a single root,
all other vertices
can be reached from the root
via a unique path.

Creates parent / child
relationships between nodes



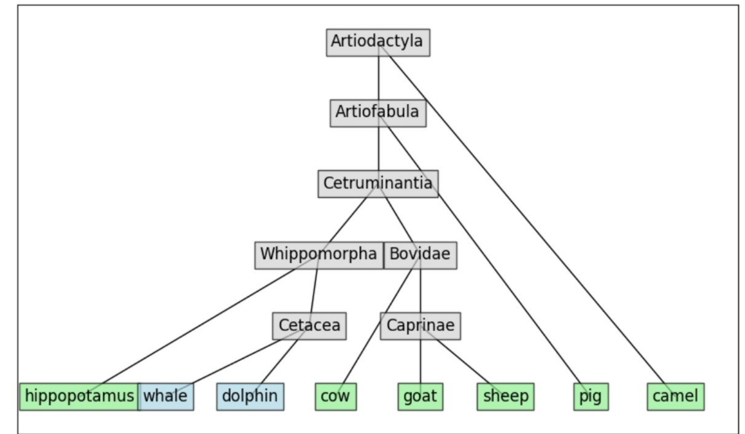
Trees and hierarchies

Trees can express hierarchies in which each entity has a single direct superior
Examples:

Company structure in which each employee (except for the head of the company) has a single supervisor (similarly army command)

Administrative divisions (country, region, district)

Species taxonomy
(animals, mammals, primates, ...)



More general hierarchies

Some hierarchies allow multiple direct superiors, for example:

- family tree where each person has two parents (and they may be distantly related),
- geometrical shapes, where a square is both a special case of a regular polygon and a special case of a rectangle and both of these are a special case of a polygon.

These hierarchies can be represented as directed acyclic graphs

- Acyclic means that by following edges we never get back to the starting node (nobody is their own ancestor).

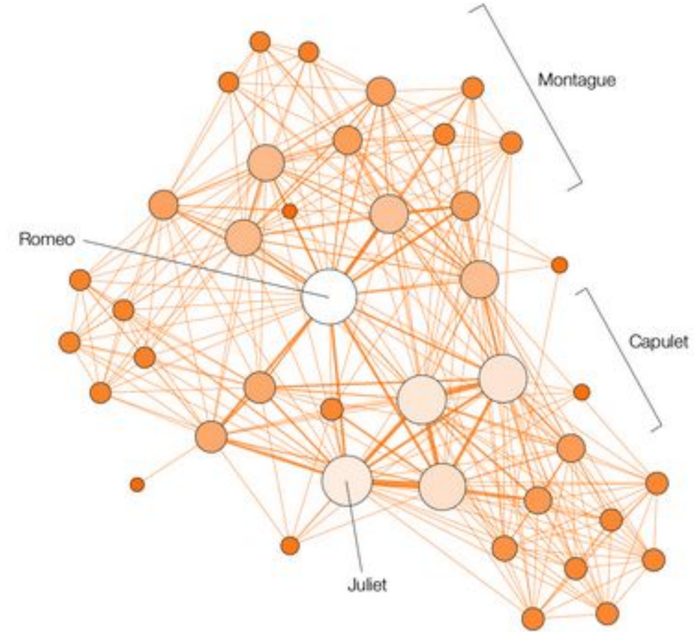
What to study / visualize in real-life graphs?

Details of connections for a particular node (requires zooming in large networks).

Overall structure of the graph: connected components, density of edges, presence of cycles, weak places (bridges and articulations), densely connected clusters

Do nodes with some property cluster together? (Are they connected by many edges?)

Example: character co-occurrence in Shakespeare



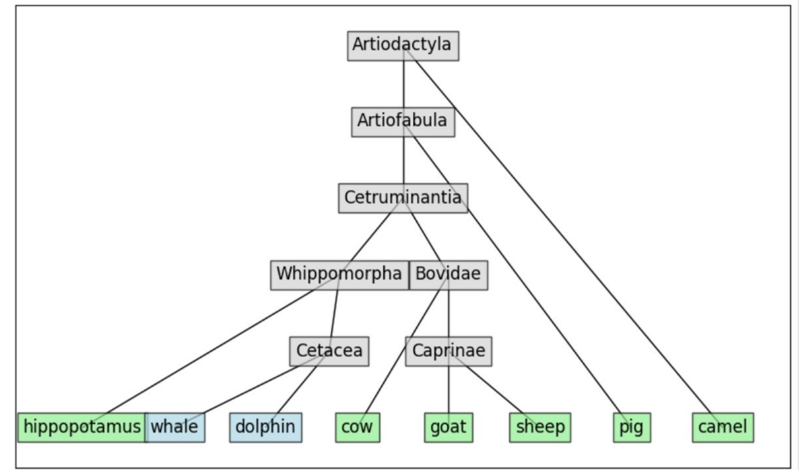
http://www.martingrandjean.ch/network-visualization-shakespeare/

Basics of graph drawing

Vertices typically shown as markers (circles, rectangles etc.), possibly with labels, size, color, ...

Edges shown as lines connecting them, possibly of different color or width. They can be straight lines, arcs, polygonal lines or arbitrary curves.

Edge direction displayed as arrows
or all edges drawn to point in one direction,
e.g. downwards.



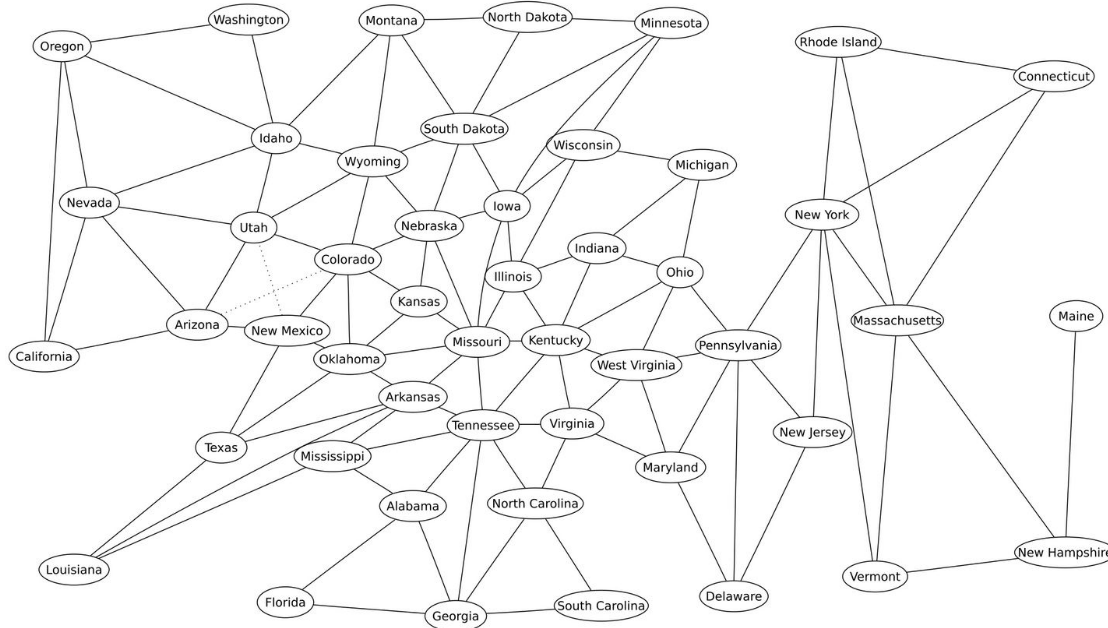
Desirable properties of a graph drawing

Nodes do not overlap.

Edges are not too long and have a simple shape without many bends.

The number of edge crossings is small.

The graph uses the space of the figure well without large empty regions.

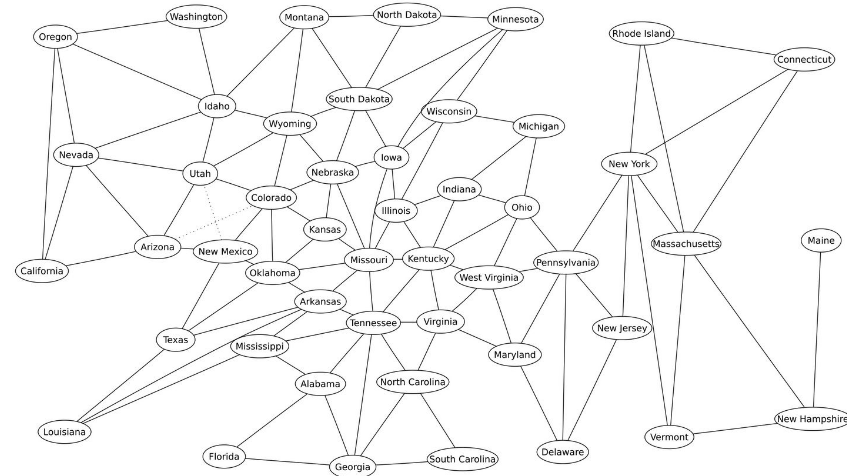


https://commons.wikimedia.org/wiki/File:UnitedStatesGraphViz.svg

Node positioning in graph drawing

Sometimes the position of nodes is given by their properties, e.g. on a map (airline connections), level of a hierarchy, timeline.

Otherwise we place nodes to optimize desirable properties, e.g. using **force-directed layout**, which assigns attractive forces (springs) between nodes connected by edges and repulsive forces between other pairs of nodes.



https://commons.wikimedia.org/wiki/File:UnitedStatesGraphViz.svg

Our hierarchy: manually created DataFrame of animals

Taxonomy of even-toed ungulates (párnokopytníky)

Level along tree (1=leaves, 6=root)

Category: land / sea / group

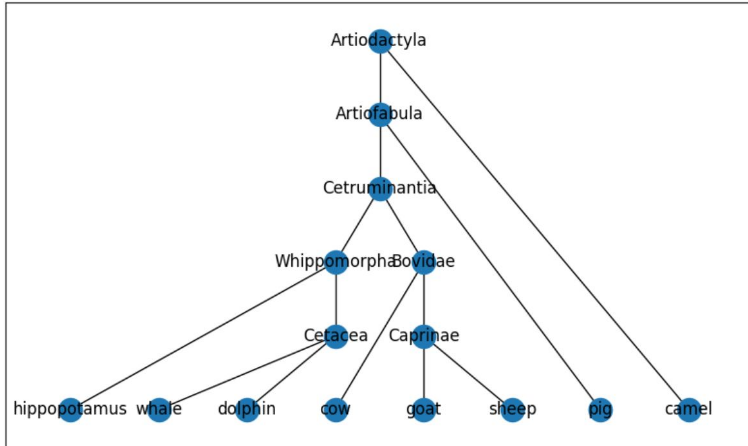
	name	parent	level	category
0	camel	Artiodactyla	1	land
1	pig	Artiofabula	1	land
2	sheep	Caprinae	1	land
3	goat	Caprinae	1	land
4	cow	Bovidae	1	land
5	dolphin	Cetacea	1	sea
6	whale	Cetacea	1	sea
7	hippopotamus	Whippomorpha	1	land
8	Caprinae	Bovidae	2	group
9	Cetacea	Whippomorpha	2	group
10	Bovidae	Cetruminantia	3	group
11	Whippomorpha	Cetruminantia	3	group
12	Cetruminantia	Artiofabula	4	group
13	Artiofabula	Artiodactyla	5	group
14	Artiodactyla	NaN	6	group

NetworkX: library for working with graphs

```
# create empty graph in NetworkX
G = nx.Graph()

# adding each table row as a node
for index, row in animals.iterrows():
  G.add_node(row['name'], level=row['level'],
             category=row['category'])

# adding an edge to each node from its parent
for index, row in animals.iterrows():
  if row['parent'] is not np.nan:
    G.add_edge(row['parent'], row['name'])
```

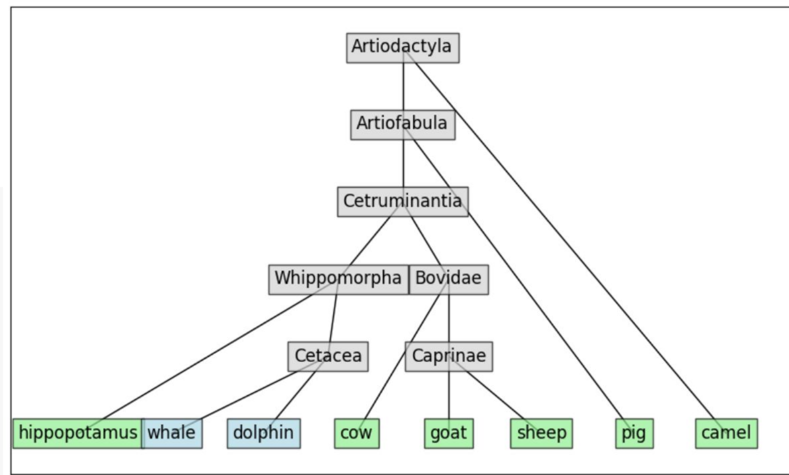
Basic graph drawing in NetworkX

```
# computing coordinates of nodes
coordinates = nx.multipartite_layout(G, subset_key="level",
                                     align='horizontal')
# drawing the graph
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx(G, coordinates, ax=axes)
```



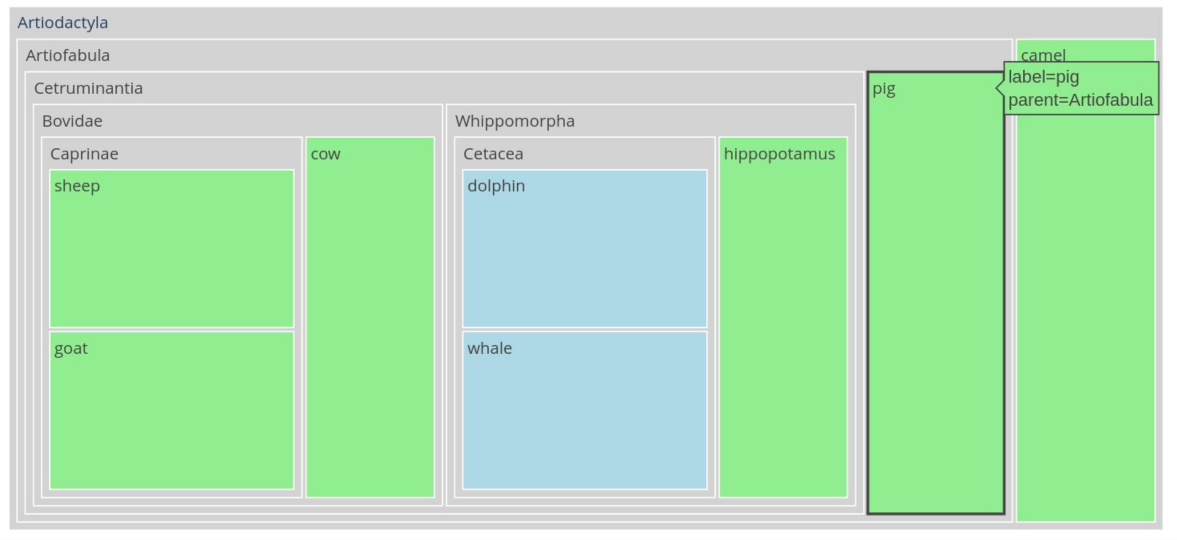
Improving the plot

```
# again compute coordinates and move some of them manually
coordinates2 = nx.multipartite_layout(G, subset_key="level",
                                      align='horizontal')
coordinates2["Whippomorpha"] += (-0.05, 0)
coordinates2["Cetacea"] += (-0.08, 0)

# plot edges only, omit nodes for now
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx_edges(G, coordinates2, ax=axes)

# plot each category of nodes by a different color
color_dict = {'group':'lightgray', 'land':'lightgreen', 'sea':'lightblue'}
for category in color_dict:
  # create a list of nodes on the category
  category_nodes = [v for v in G.nodes if G.nodes[v]['category']==category]
  # select subgraph H of G
  H = G.subgraph(category_nodes)
  # create a dictionary of node label attributes
  label_options = {"ec": "black", "fc": color_dict[category], "alpha": 0.7}
  # draw the node labels as boxes
  nx.draw_networkx_labels(H, coordinates2, font_size=12, b
                          box=label_options, ax=axes)
```



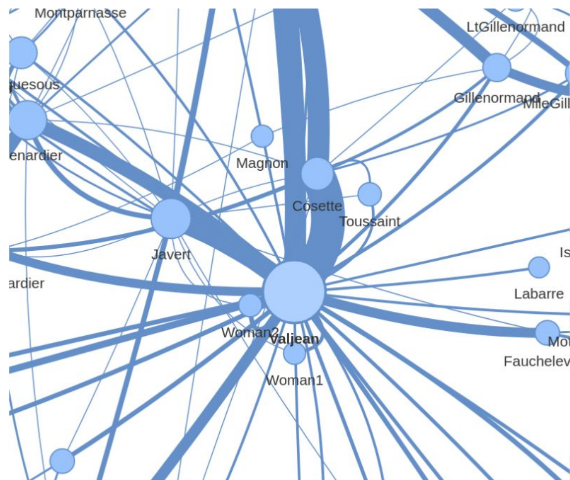
Hierarchy as a treemap in Plotly Express

```
import plotly.express as px
fig = px.treemap(
    names=animals['name'],
    parents=animals['parent'],
    color=animals['category'],
    color_discrete_map=color_dict
)
fig.show()
```



Interactive graphs in Pyvis

- Example from NetworkX: character co-occurrence in Les Misérables by Victor Hugo.
- Edges weighted by frequency



```
# initializing an empty network, setup plot properties
pyvis_net = Network("500px", "500px", notebook=True,
                    cdn_resources='in_line')
# loading network from NetworkX
pyvis_net.from_nx(nx.les_miserables_graph())

# get a dictionary of neighbors for each node
neighbors = pyvis_net.get_adj_list()
# add additional node properties
# used as tooltip and size
for node in pyvis_net.nodes:
    node["title"] = node["id"]
    node["value"] = len(neighbors[node["id"]])

# saving the visualization in an html file
pyvis_net.show("net.html")
# displaying the html file in the notebook
from IPython.display import display, HTML
display(HTML('net.html'))
```

Summary of graphs

- Graphs important in **many applications**
- **NetworkX** library has many functions for working with graphs, several layout algorithms for visualization
- **Pyvis** allows interactive visualization
- **Plotly** can visualize trees as **treemaps**

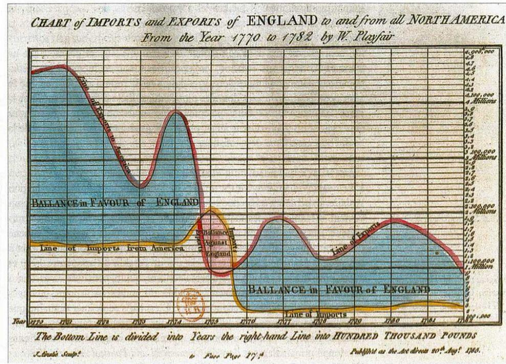
Part III: Time series

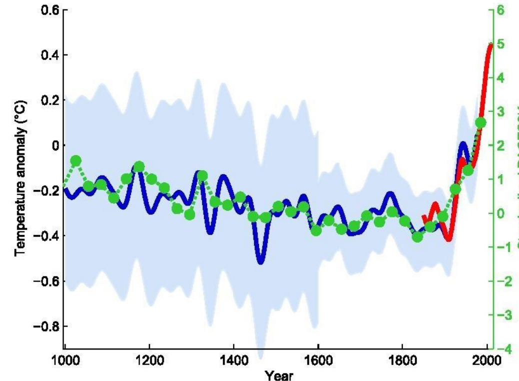
Time series (časové rady)

Sequences of measurements over time (regular or irregular time intervals).

Typically displayed as a **line graph**, with time as x-axis, time flowing from left to right (a cultural convention in western countries).

Other options: bar graphs, heat maps, box plots, ...





Recall:
Playfair 1786,
Mann, Bradley & Hughes 1999

https://commons.wikimedia.org/wiki/File:1786_Playfair_-_Chart_of_import_and_exports_of_England_to_and_from_all_North_America_from_the_year_1770_to_1782.jpg https://en.wikipedia.org/wiki/File:T_comp_61-90.pdf

Typical features of a time series

Overall trend (increasing / decreasing / flat; rate of change),

Seasonality (daily / weekly / yearly cycles),

Noise (general variability / outliers)

Two Google trend time series

Google trends compare frequency of search terms over time and to each other.

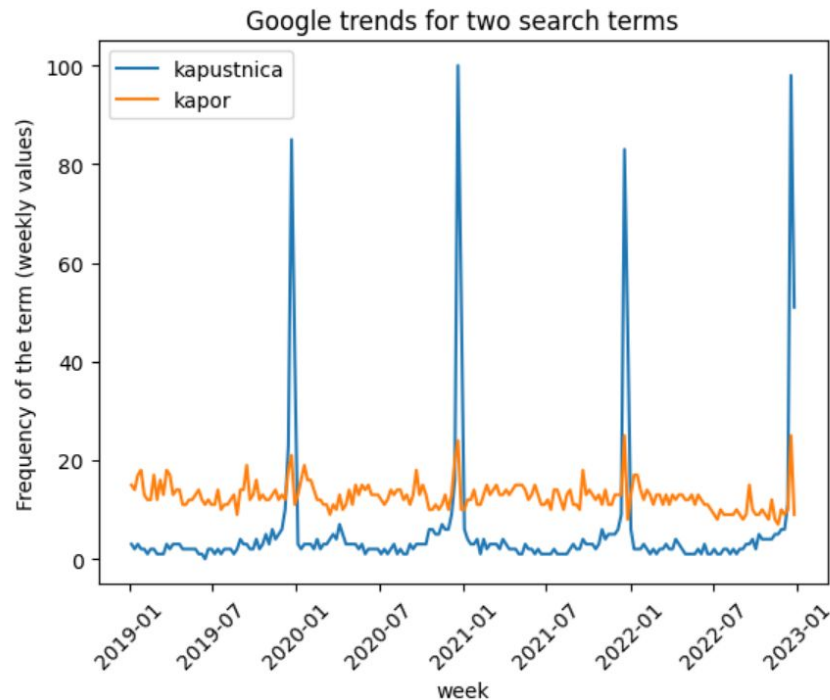
Here we use Christmas-related terms kapustnica and kapor.

```
display(trends1.head())
```

week	kapustnica	kapor
2019-01-06	3	15
2019-01-13	2	14
2019-01-20	3	17
2019-01-27	2	18
2019-02-03	2	13

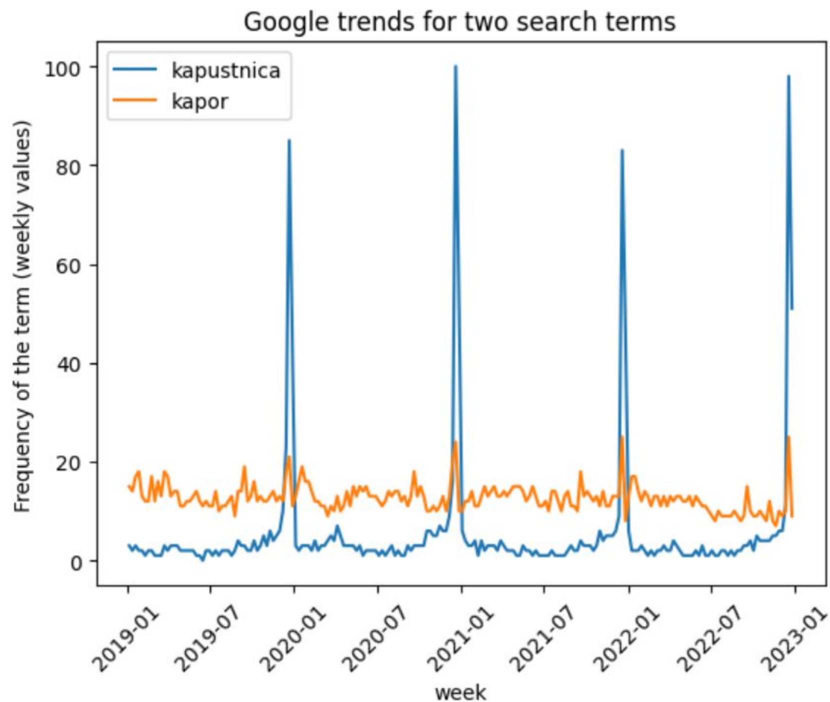
What can we see on the plot?
(trend, seasonality, noise)

Comparison of the two terms?



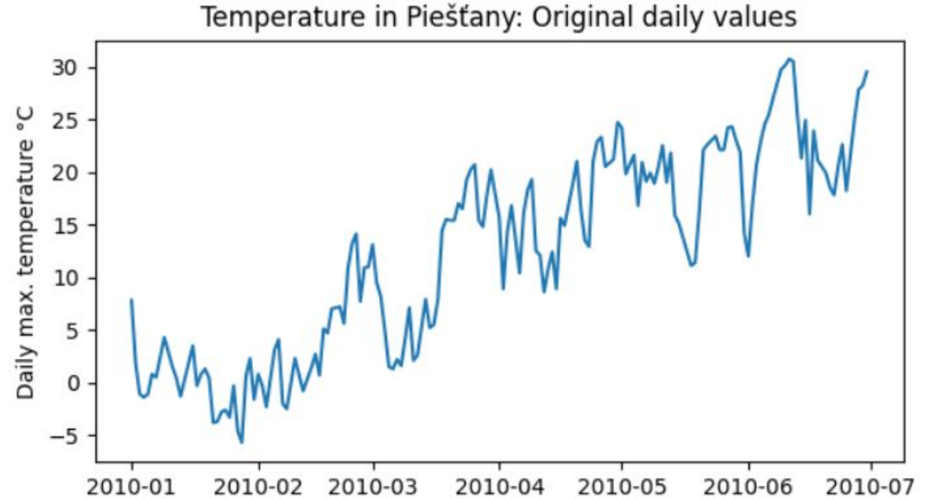

```
axes = sns.lineplot(trends1, dashes=False)
axes.set_ylabel("Frequency of the term (weekly values)")
axes.set_title("Google trends for two search terms")
# rotate tick labels
axes.tick_params(axis='x', labelrotation = 45)
```

Search terms related to kapor: zbgis kataster, zbgis mapa, zgbis, katasterportal list vlastnictva, dážďovka.



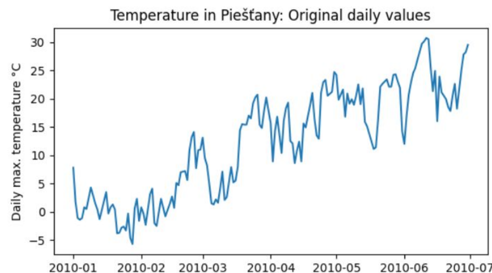
Trend: temperatures are growing in spring

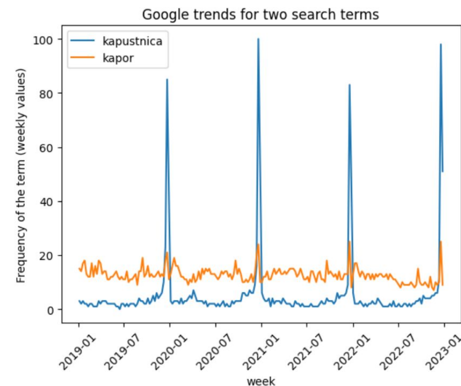
Second dataset:
Maximum daily temperature values from Piešťany January-June 2010, from US National Oceanic and Atmospheric Administration



Smoothing data (vyrovnanie, vyhladenie)

Our time series are quite noisy.





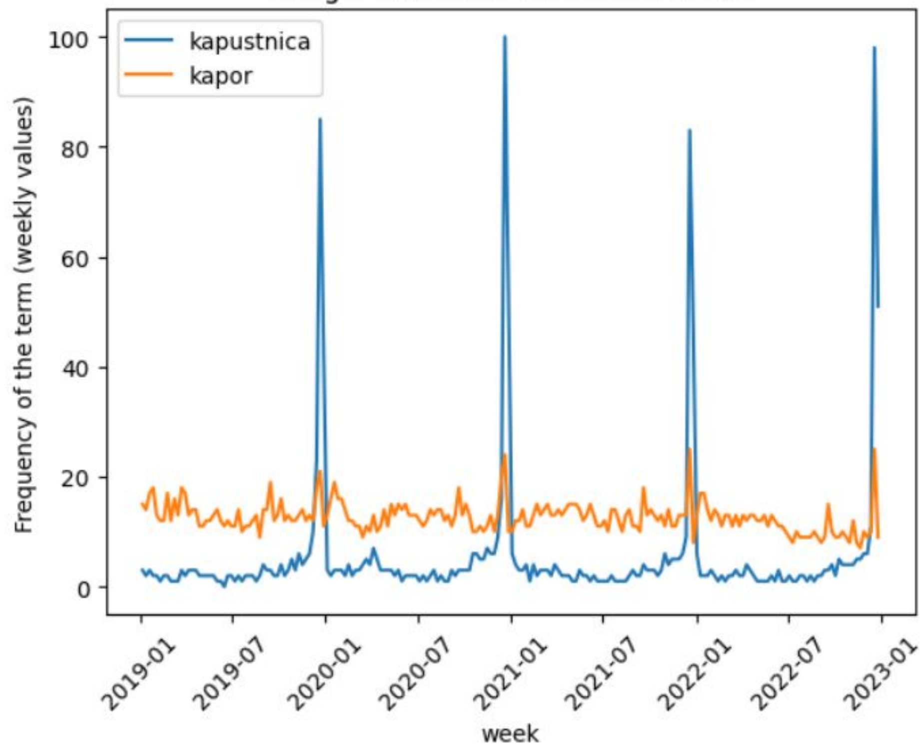
Two options for **smoothing data:**

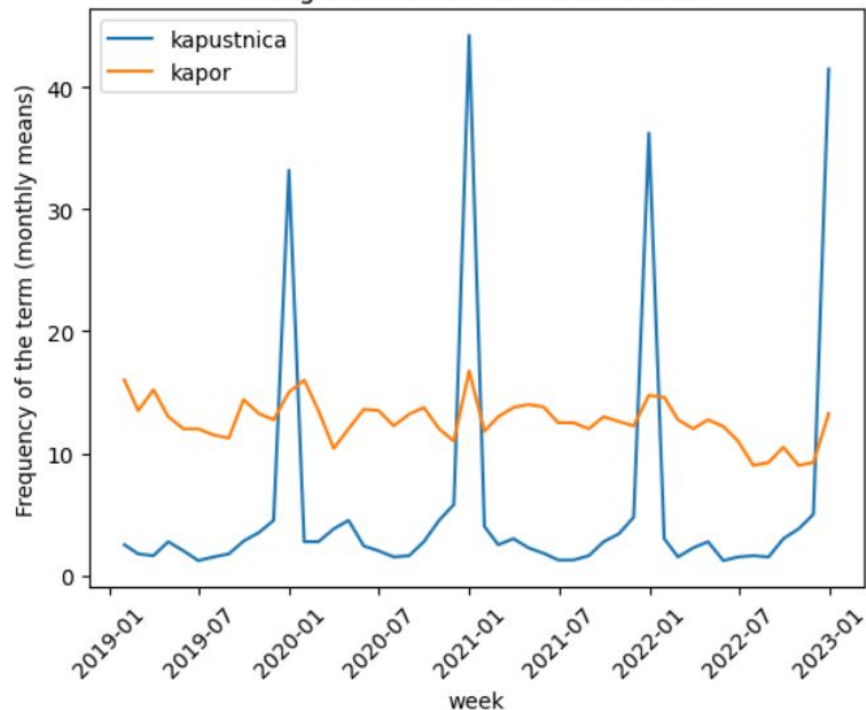
Aggregating them in longer time intervals (e.g. months).

Sliding window (kĺzavé okno): we choose a window size *w* and compute a new series, each value being mean or other summary of *w* consecutive windows in the input.

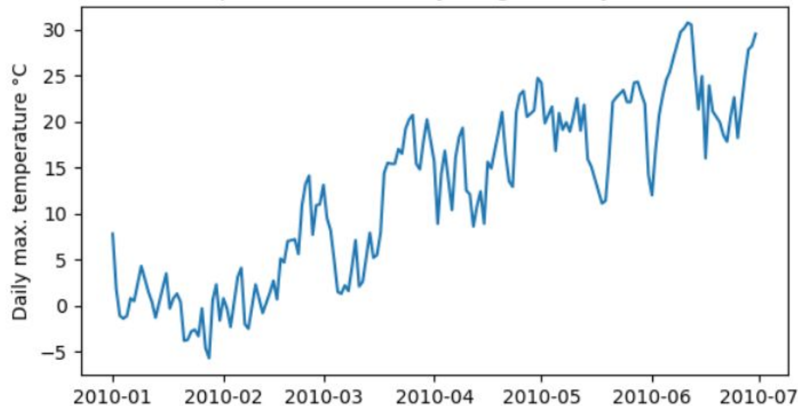
For example with values 2,6,4,2,8,2 and window size 4, we get window means 3.5, 5, 4.

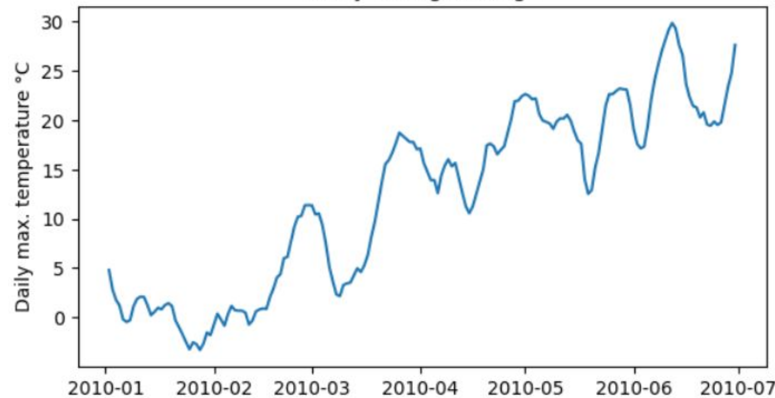
Smoothing Google trends by monthly aggregation

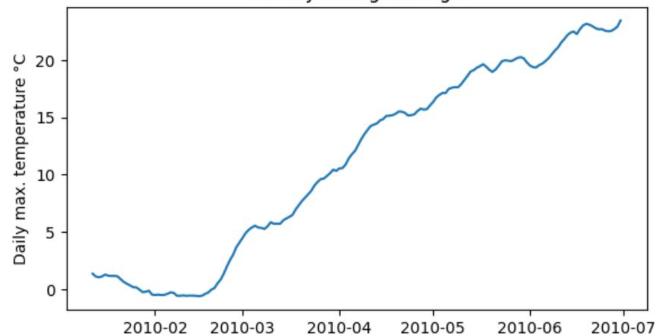




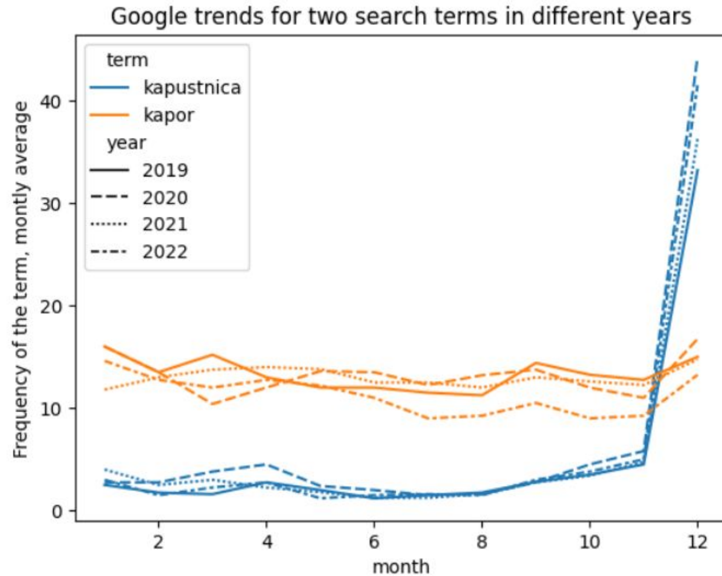
Smoothing temperatures by sliding window

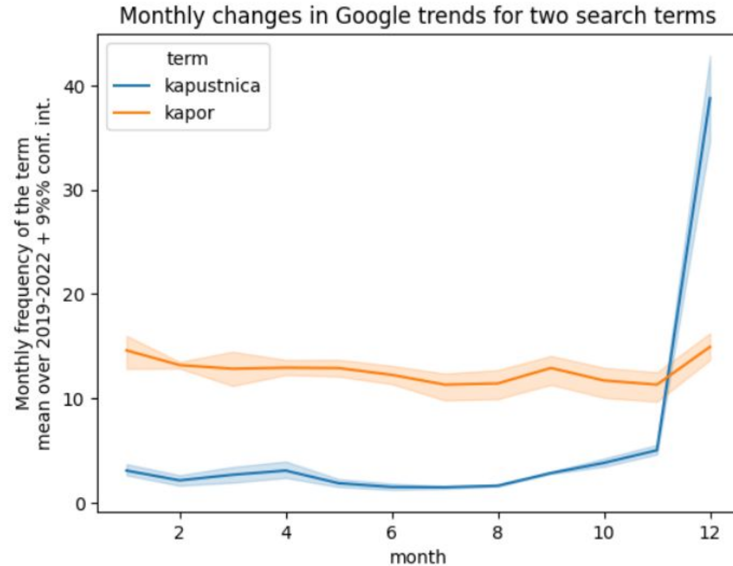






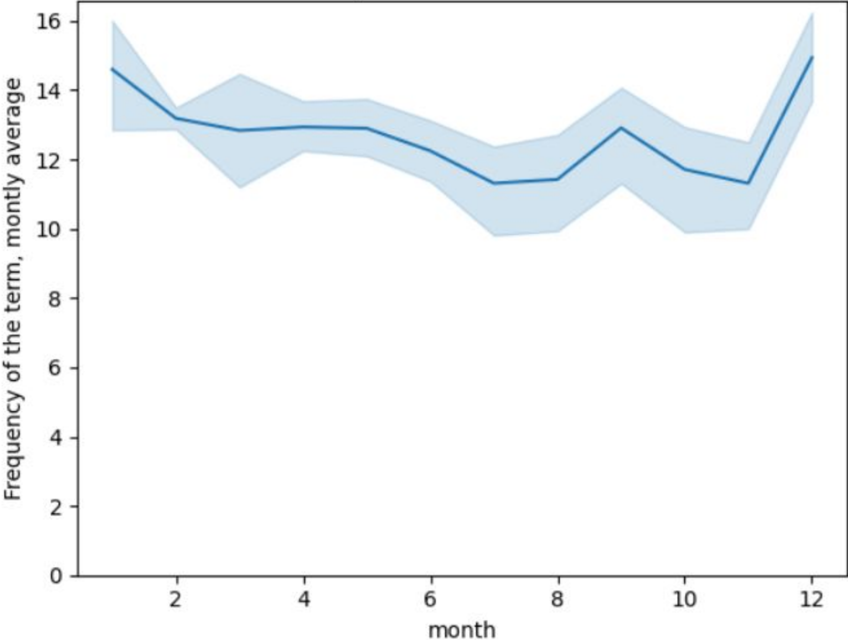
Overlapping timescales to display seasonality / showing uncertainty

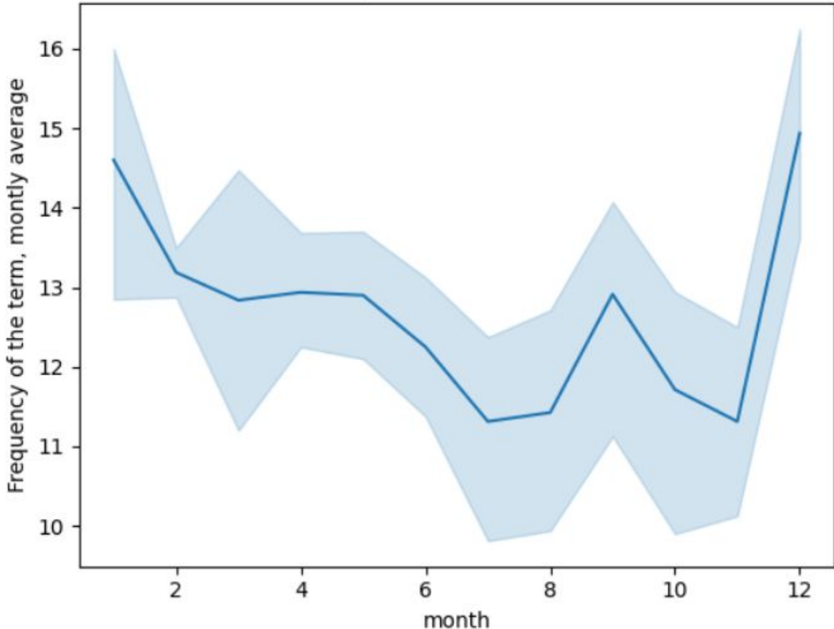




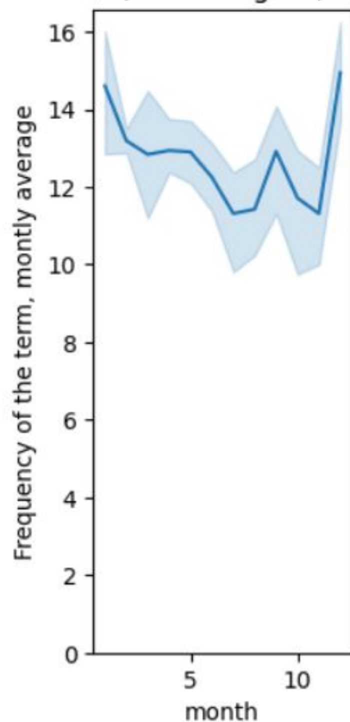
Right: multiple years summarized as the mean and its 95% confidence interval expressing uncertainty in the true value of the mean due to noise in data.

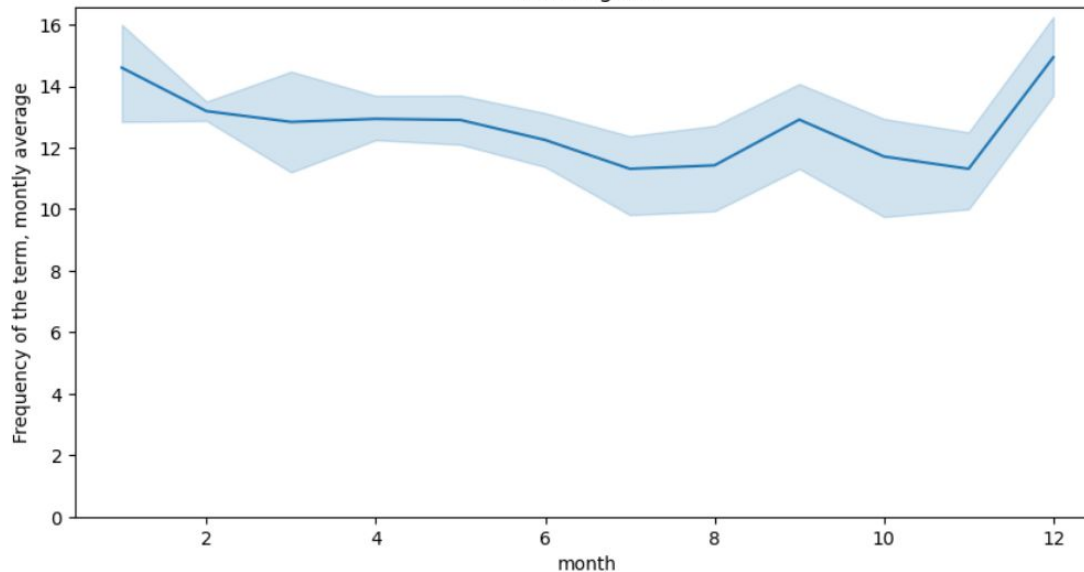
Importance of scales



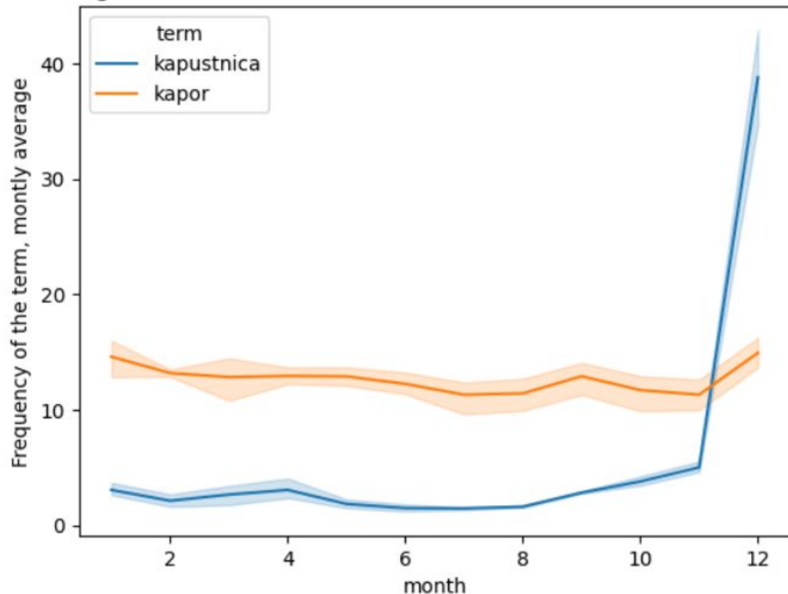


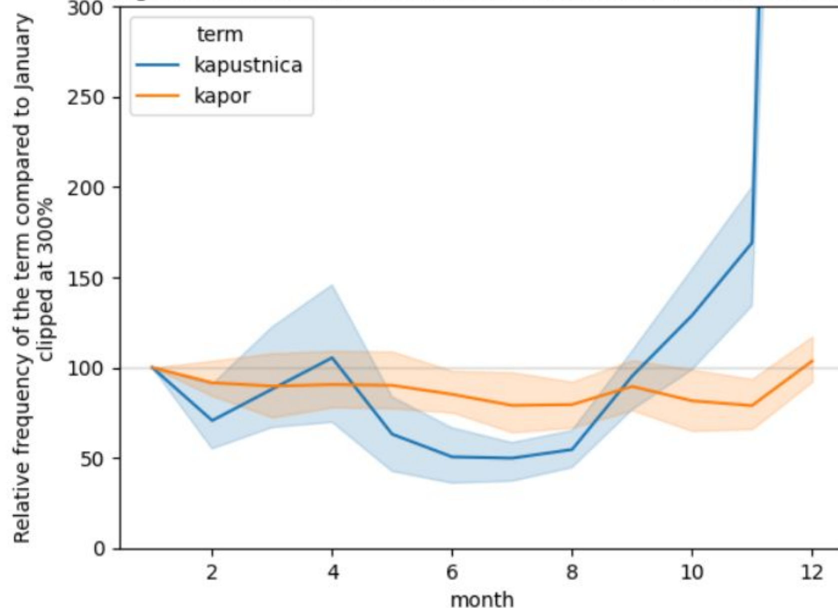
Importance of scales





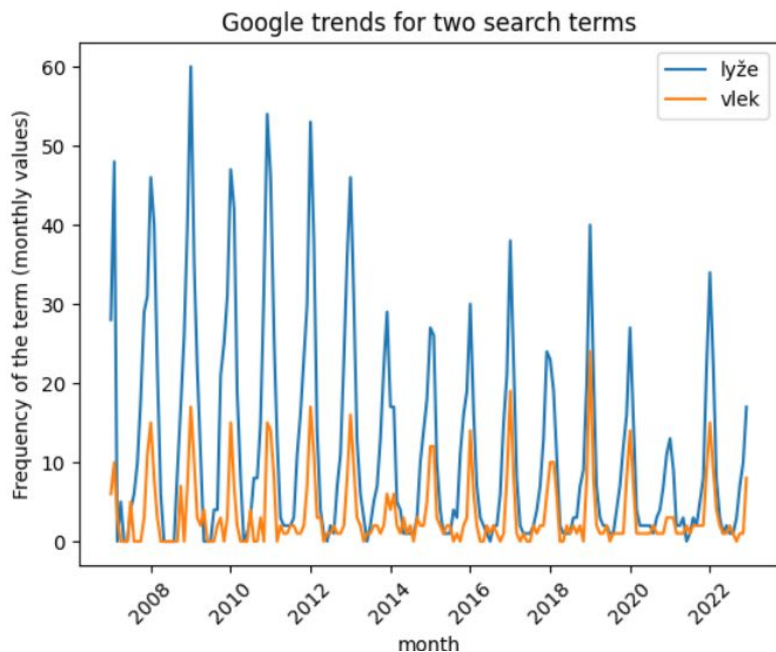
Relative scales

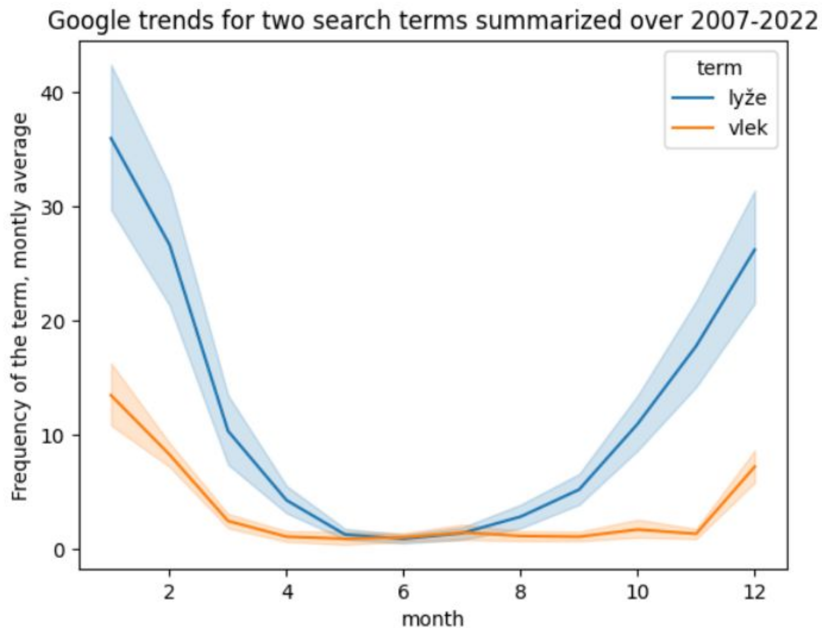




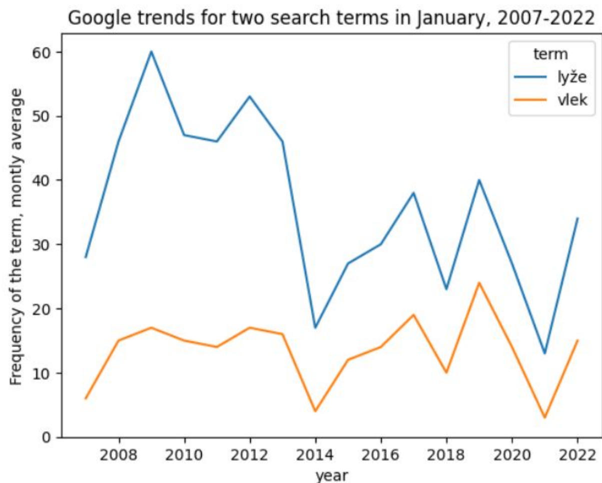
Right: monthly values relative to January
(can be used to compare trends even if overall values vary different)

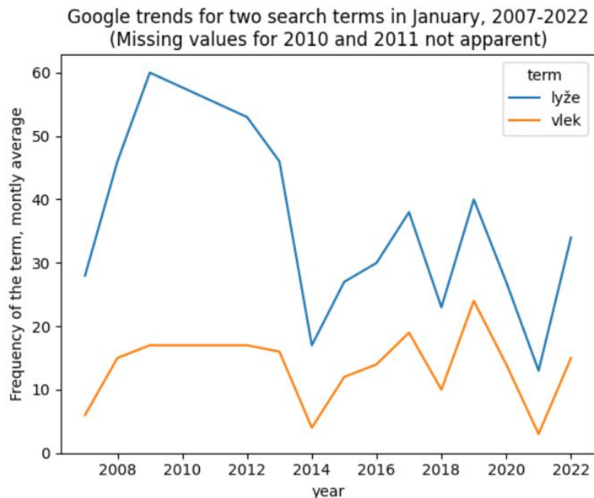
One more pair of Google trend lines

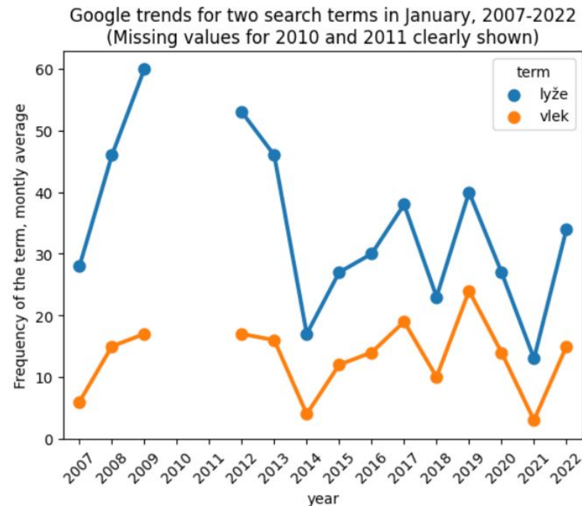




Acknowledging missing values







Middle and right: values for 2010 and 2011 missing
Middle: missing values are not visible, misleading plot
Right: missing values are easy to spot

Summary of time series

Typical goals are to observe and study:

- overall trend (increasing / decreasing / flat; rate of change),
- seasonality (daily / weekly / yearly cycles),
- noise (general variability / outliers)

Useful techniques:

- smoothing by aggregation and sliding window
- overlapping timescales
- relative scales
- showing uncertainty and missing values

1 Lecture 6: Maps, graphs, time series

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Package installation

We need to install several packages which are not pre-installed in Colab.

[1]:
```
! pip install geoplot pyvis
```

[2]:
```
# importing our usual libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from IPython.display import Markdown


# new libraries for today:
#  geopandas is a library for working with geographical data
import geopandas as gpd
#  geoplot is a library for visualizing geographical data
import geoplot as gplt
#  we also will use a submodule of plotly
import plotly.graph_objects as go
#  networkx is a library for working with graphs and networks
import networkx as nx
# Pyvis is a library for drawing networks
from pyvis.network import Network
```

1.2 Maps

1.2.1 Introduction

- Each map is a visualization of data about location of objects.
- Maps have a rich set of conventions about colors and symbols, orientation etc. This allows us to quickly understand a map.

Examples:

- A topographic map from US looks similar to maps used in Slovakia, but striped lines are typically used for railroads in Slovakia (example).
- A map of European countries from 1721 can still be easily read by today's audience.

Data visualization in maps

- Maps visualizing data other than typical geographical features are usually called thematic maps (tematické mapy).

- We will see several examples, for others see e.g. Wikipedia, GeoPlot library gallery.
- Also recall Snow's map of cholera cases from the first lecture.

1.2.2 Map projection (kartografické zobrazenie)

- A map projection is a transformation to project the surface of a globe onto a plane.
- Each projection introduces **some distortion**.

Conformal projections preserve local angles, but distort other aspects, such as lengths, areas etc.

- For example, Mercator projection (1569) was developed for navigation, but shows Greenland bigger than Africa, while in fact it is 14x smaller.

Equal-area projections preserve areas (cannot be conformal at the same time).

- Equal-area projections are typically good for data visualization, as they make areas comparable.

Orthographic projection is similar to a photograph of the Earth from a very distant point.

- It is not an equal-area projection, but our sense of perspective may compensate.
- It displays one hemisphere.

Recommended projections (Cairo, The Truthful Art):

- Whole world: e.g. Mollweide equal-area projection (1805)
- Continents / large countries: e.g. Lambert azimuthal equal-area projection (1772)
- Countries in mid-latitudes: e.g. Albers equal-area conic projection (1805)
- Polar regions: e.g. Lambert azimuthal equal-area projection (1772)

Examples of projections in Plotly

- Plotly allows us to set projections for our plot. Here we use it to illustrate the projections on the map of continent outlines.
- The maps are interactive.

[3]:
```
def show_world(projection, scope=None):
  """A function to display the whole Earth or a desired
  area (scope) using a selected projection. Both arguments
  are strings that name projections or scopes supported by Plotly."""
  # create a map figure with an empty scatterplot
  fig = go.Figure(go.Scattergeo())
  # set the desired projection
  fig.update_geos(projection_type=projection)
  # we can also limit the scope of the map
  if scope is not None:
    fig.update_geos(scope=scope)
  # finally, make the image smaller and with 0 margins
  fig.update_layout(height=200, margin={"r":0,"t":0,"l":0,"b":0})
  # show the figure
  fig.show()
```

```
[4]: display(Markdown("**Orthographic projection**"))
show_world("orthographic")
```

Orthographic projection

```
[5]: display(Markdown("**Mollweide equal-area projection**"))
show_world("mollweide")
```

Mollweide equal-area projection

```
[6]: display(Markdown("**Mercator conformal projection** (not recommended for data␣
     ↪visualization)"))
show_world("mercator")
```

Mercator conformal projection (not recommended for data visualization)

```
[7]: display(Markdown("**Lambert azimuthal equal-area projection**"))
show_world("azimuthal equal area", "europe")
```

Lambert azimuthal equal-area projection

1.2.3 Adding data as points and lines to a map

- Geographic coordinates of places can be projected as x and y. Additional values can be shown using marker color and size or line color and width.
- We illustrate this using datasets of airport locations and airline connections.

Importing datasets

- The dataset of international airports of the world was downloaded from the World Bank under the CC-BY 4.0 license, and preprocessed. It includes the number of seats within a year, which is from unknown years, possibly not comparable between countries.
- Our preprocessed file is in GeoJSON format used for describing simple geographical features. It contains both location data and other attributes.
- We parse the file using GeoPandas, which is a library for working with geographical data.
- It is an extension of Pandas DataFrame, with location information.
- Each row of the table contains one airport, with its 3-letter code, name, country, 3-letter code of the country, the number of airplane seats per year and the location.

```
[8]: display(Markdown("**Importing the list of airports**"))
# parse the file
airports = gpd.read_file("https://bbrejova.github.io/viz/data/airports.geojson")
# show the first 5 rows
display(Markdown("**The first five rows:**"), airports.head())
# show the total number of rows
display(Markdown(f"**The number of rows:** {airports.shape[0]}"))
display(Markdown("**International airports in Slovakia**"))
display(airports.query('Country == "Slovakia"'))
```

Importing the list of airports

The first five rows:

```
  Orig                      Name   TotalSeats      Country ISO3  \
0  HEA                     Herat    22041.971  Afghanistan  AFG
1  JAA                  Jalalabad     6343.512  Afghanistan  AFG
2  KBL       Kabul International  1016196.825  Afghanistan  AFG
3  KDH  Kandahar International    39924.262  Afghanistan  AFG
4  MZR             Mazar-e-Sharif    58326.513  Afghanistan  AFG

                    geometry
0  POINT (62.22670 34.20690)
1  POINT (70.50000 34.40000)
2  POINT (69.21390 34.56390)
3  POINT (65.84750 31.50690)
4  POINT (67.20830 36.70420)
```

The number of rows: 2173

International airports in Slovakia

```
     Orig              Name   TotalSeats   Country ISO3  \
1489  BTS      M.R. Stefanik  1211732.116  Slovakia  SVK
1490  ILZ             Zilina     3986.360  Slovakia  SVK
1491  KSC              Barca   323259.132  Slovakia  SVK
1492  PZY   Piestany Airport     1403.892  Slovakia  SVK
1493  SLD              Sliac    11876.753  Slovakia  SVK
1494  TAT       Tatry/Poprad    39612.286  Slovakia  SVK

                       geometry
1489  POINT (17.21670 48.16670)
1490  POINT (18.76670 49.23330)
1491  POINT (21.25000 48.66670)
1492  POINT (17.83330 48.63330)
1493  POINT (19.13330 48.63330)
1494  POINT (20.24030 49.07190)
```

- We will later need a datasets of coutry boundaries proved by Natural Earth.
- Below we estimate country area from its low resolution borders. For example, area of Slovakia estimated as 47070km2, while World Bank lists as 49030km2.

[9]:
```
# read world countries as a dataset provided by Natural Earth
countries = gpd.read_file("https://bbrejova.github.io/viz/data/
↪country_boundaries.geojson")
# set 3-letter code as the index
countries = countries.set_index('ISO3')
# estimate country area in square km from geometry
# first the geometry is projected by equal-area projection
# the result is in square meters, converted to squared km (divide by 1e6)
```

```
# beware that areas are approximate due to low resolution borders
country_areas = countries['geometry'].to_crs({'proj':'cea'}).area  / 1e6
# add areas to countries
countries['Area'] = country_areas

display(Markdown("**Table of countries of the world**"))
display(countries.head())
display(Markdown("**Data for Slovakia**"))
display(countries.loc['SVK', :])
```

Table of countries of the world

```
                   Type                      Name   Population  \
ISO3
FJI   Sovereign country                      Fiji     889953.0
TZA   Sovereign country                  Tanzania   58005463.0
B28       Indeterminate                 W. Sahara     603253.0
CAN   Sovereign country                    Canada   37589262.0
USA             Country  United States of America  328239523.0

                          Region  \
ISO3
FJI          East Asia & Pacific
TZA           Sub-Saharan Africa
B28   Middle East & North Africa
CAN                North America
USA                North America

                                               geometry          Area
ISO3
FJI   MULTIPOLYGON (((180.00000 -16.06713, 180.00000…  1.928760e+04
TZA   POLYGON ((33.90371 -0.95000, 34.07262 -1.05982…  9.327793e+05
B28   POLYGON ((-8.66559 27.65643, -8.66512 27.58948…  9.666925e+04
CAN   MULTIPOLYGON (((-122.84000 49.00000, -122.9742…  1.003773e+07
USA   MULTIPOLYGON (((-122.84000 49.00000, -120.0000…  9.509851e+06
```

Data for Slovakia

```
Type                                          Sovereign country
Name                                                   Slovakia
Population                                            5454073.0
Region                                    Europe & Central Asia
geometry      POLYGON ((22.558137648211755 49.08573802346714…
Area                                               47069.779734
Name: SVK, dtype: object
```

All airports as points using Plotly

- We use `scatter_geo` function from Plotly Express.

- We set parts of `geometry` column as latitude and longitude. Column `Name` is used as a tooltip.

[10]:
```
fig = px.scatter_geo(
    airports,
    lat=airports.geometry.y,
    lon=airports.geometry.x,
    hover_name="Name",
    projection="mollweide"
    )
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

Adding the size of the airport

- We use size of the circle to represent the number of seats.
- Scatterplots with point sizes are often called bubble graphs.
- We focus on Europe, add country borders and change the projection.

[11]:
```
fig = px.scatter_geo(
    airports,
    lat=airports.geometry.y,
    lon=airports.geometry.x,
    size="TotalSeats",
    hover_name="Name"
    )
fig.update_geos(
    projection_type="azimuthal equal area",
    lonaxis_range= [-20, 40],
    lataxis_range= [20, 70],
    showcountries = True
)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

Airline connections from Slovakia as lines

- We import another table (originating from World bank as above), which shows international airline connections from Slovak airports (in an unknown year).
- Each connection is given by two airport codes, the number of airplane seats within a year, and geometry with a segment connecting the two airport locations.
- Line color will correspond to the airport of origin in Slovakia.
- The code is adapted from examples in the Plotly documentation.

[12]:
```
display(Markdown("**Importing airline connections**"))
connections = gpd.read_file("https://bbrejova.github.io/viz/data/
 ↪airport_pairs_svk.geojson")
```

```
display(connections.head())
```

Importing airline connections

```
  OrigCode DestCode  TotalSeats  \
0      BTS      ADB    7370.433
1      BTS      AGP   15152.501
2      BTS      AHO   14740.866
3      BTS      AQJ    3275.748
4      BTS      ATH   19654.488

                                            geometry
0  LINESTRING (17.21670 48.16670, 27.15620 38.29430)
1  LINESTRING (17.21670 48.16670, -4.49810 36.67170)
2   LINESTRING (17.21670 48.16670, 8.28890 40.63060)
3  LINESTRING (17.21670 48.16670, 35.01940 29.61250)
4  LINESTRING (17.21670 48.16670, 23.94440 37.93640)
```

[13]:
```
def draw_lines(connections):
  # create two lists with x and y coordinates of polylines
  # separated by None
  lats = []
  lons = []
  # also create lists of origin and destination codes parallel to lists above
  origCodes = []
  destCodes = []

  # iterate through table rows
  for index, row in connections.iterrows():
    # get lists of x and y coordinates (of length 2 in this case)
    x, y = row['geometry'].xy
    # add coordinates and None separator to lists
    lats.extend(list(y) + [None])
    lons.extend(list(x) + [None])
    # add airport codes for each coordinate and None separator
    origCodes.extend([row['OrigCode']] * len(x) + [None])
    destCodes.extend([row['DestCode']] * len(x) + [None])

  # create figure with these lists
  fig = px.line_geo(lat=lats, lon=lons, hover_name=destCodes, color=origCodes)
  # setup projection
  fig.update_geos(
    projection_type="azimuthal equal area",
    lonaxis_range= [-25, 55],
    lataxis_range= [10, 60],
    showcountries = True
  )
  fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
```

```
  fig.show()

# call the function to draw the map
display(Markdown("**Airline connections from Slovak airports**"))
draw_lines(connections)
```

Airline connections from Slovak airports

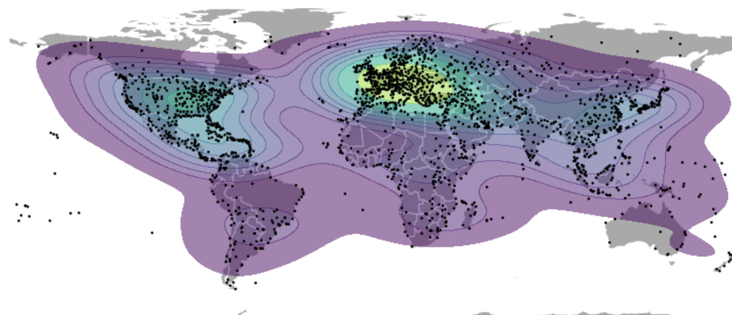
1.2.4 Isarithmic maps / isoline maps / heatmaps

- These maps display a continuous variable over the map area (elevation, temperature and other weather phenomena etc.).
- Value in each point can be shown by a color scale.
- Some contour lines can be displayed as well.
- A contour line (isoline, isopleth, isarithm, izočiara) connects points of the same value.
- Example: short-term forecasts from the Slovak Hydrometeorological Institute.

Density of airports

- Here we show world airports as both points and their local density as a isarithmic map.
- This is achieved using `kdeplot` function from the Geoplot library.
- KDE stands for kernel density estimation, and we will explain it in the next lecture.

[14]:
```
# plot countries as a background
ax = gplt.polyplot(
    countries.explode(index_parts=True),
    edgecolor='white',
    facecolor='darkgray',
    figsize=(10, 5),
)
# plot semi-transparent isarithmic map
gplt.kdeplot(
    airports, cmap='viridis',
    fill=True, alpha=0.5, ax=ax
)
# plot points on top
gplt.pointplot(airports, s=1, color='black', ax=ax)
pass
```



1.2.5 Choropleth maps (kartogramy)

- Often we have numerical / categorical values for administrative regions (countries, districts, etc.).
- Choropleth maps show such variables via colors applied to the whole region.

Variables over regions:

- **Spatially extensive** variables apply to the unit as a whole (e.g. total population, area, the number of airports in the country). If we subdivide the region, spatially extensive variable will be often the sum of its parts (but not always, e.g. perimeter)
- **Spatially intensive** variables may stay the same if you divide the unit, provided the unit is homogeneous without regional differences. Examples include population density, life expectancy, GDP per person.
- Spatially extensive variables are not appropriate for choropleths, because large value for a large country is visually attributed to each small subregion of the country. If counts are of interest, better use a bubble graph with marker of appropriate size in the region center.

Beware:

- A choropleth map is called kartogram in Slovak.
- English word **cartogram** means a map with regions rescaled according to some variable (such as the Levasseur's cartogram of country budgets and a modern example).

Choropleth maps of airports per country We will show three choropleth maps:

- the number of airports per 10000 km2 (spatially intensive variable),
- the number of airports per million inhabitants (also spatially intensive),
- the number of airports (spatially extensive, not recommended for choropleth), We will also show the number of airports as a bubble graph (more appropriate than extensive variable).

All choropleth maps are created by Plotly. The bubble graph is also created by plotly, and the bubble is placed to the representative point of each country.

[15]:
```
# compute the number of airports per country by groupby
airports_per_country = airports.groupby('ISO3').size()
# add the new column to a copy of the old table
countries2 = countries.copy(deep=True)
# add the number of airports as a new column
countries2['Airports'] = airports_per_country
# remove countries where airports or location are missing
countries2.dropna(subset=['geometry', 'Airports'], inplace=True)
# add columns with airport density and airports per million people
countries2['Airport_density'] = (countries2['Airports']
                                 / countries2['Area'] * 10000)
countries2['Airports_per_mil'] = (countries2['Airports']
                                  / countries2['Population'] * 1e6)
# show the new table
```

```
display(Markdown("**The first five rows of `countries2` table:**"))
display(countries2.head())
display(Markdown("**The values for Slovakia:**"))
display(countries2.loc['SVK'])
```

The first five rows of `countries2` table:

```
                   Type                      Name   Population  \
ISO3
FJI   Sovereign country                      Fiji     889953.0
TZA   Sovereign country                  Tanzania   58005463.0
CAN   Sovereign country                    Canada   37589262.0
USA             Country  United States of America  328239523.0
KAZ         Sovereignty                Kazakhstan   18513930.0

                     Region  \
ISO3
FJI     East Asia & Pacific
TZA      Sub-Saharan Africa
CAN           North America
USA           North America
KAZ   Europe & Central Asia

                                               geometry          Area  \
ISO3
FJI   MULTIPOLYGON (((180.00000 -16.06713, 180.00000…  1.928760e+04
TZA   POLYGON ((33.90371 -0.95000, 34.07262 -1.05982…  9.327793e+05
CAN   MULTIPOLYGON (((-122.84000 49.00000, -122.9742…  1.003773e+07
USA   MULTIPOLYGON (((-122.84000 49.00000, -120.0000…  9.509851e+06
KAZ   POLYGON ((87.35997 49.21498, 86.59878 48.54918…  2.728701e+06

      Airports  Airport_density  Airports_per_mil
ISO3
FJI        2.0         1.036935          2.247310
TZA        7.0         0.075045          0.120678
CAN       82.0         0.081692          2.181474
USA      291.0         0.305998          0.886548
KAZ       17.0         0.062301          0.918228
```

The values for Slovakia:

```
Type                                                 Sovereign country
Name                                                          Slovakia
Population                                                   5454073.0
Region                                           Europe & Central Asia
geometry            POLYGON ((22.558137648211755 49.08573802346714…
Area                                                      47069.779734
Airports                                                           6.0
Airport_density                                               1.274703
```

```
Airports_per_mil                                          1.100095
Name: SVK, dtype: object
```

[16]:
```
def draw_choropleth(data, column, range_color=None, label=None):
  fig = px.choropleth(
    data, locations=data.index, color=column,
    range_color=range_color,
    labels={column:label},
    hover_name="Name",
    projection = "mollweide"
  )
  fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
  fig.show()

display(Markdown("**The number of airports per 10000 squared km**"))
draw_choropleth(countries2, 'Airport_density', (0, 2), 'airports / 10000 km2')
```

The number of airports per 10000 squared km

[17]:
```
display(Markdown("**The number of airports per million inhabitans**"))
draw_choropleth(countries2, 'Airports_per_mil', (0, 5), 'airports / million␣
 ↪people')
```

The number of airports per million inhabitans

[18]:
```
display(Markdown("**The number of airports in a country**"))
draw_choropleth(countries2, 'Airports', (0, 100), 'airports')
```

The number of airports in a country

[19]:
```
# make a new table of countries in which geometry is replaced
# with a single representative point
countries3 = countries2.copy(deep=True)
countries3['geometry'] = countries2['geometry'].representative_point()

# plot as a bubble plot
display(Markdown("**The number of airports in a country**"))
fig = px.scatter_geo(
    countries3,
    lat=countries3.geometry.y,
    lon=countries3.geometry.x,
    size="Airports",
    hover_name="Name",
    projection = "mollweide"
    )
fig.update_geos(showcountries = True)
fig.update_layout(height=300, margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

The number of airports in a country

1.2.6 Summary of maps

- Many data sets contain geographic entities (countries, cities, coordinates).
- Displaying such data on maps highlights spatial relationships.
- In your maps, use appropriate equal-area projections.

Types of thematic maps:

- Bubble plots contain points of various sizes and colors.
- Isarithmic maps / isoline maps / heatmaps display continuously varying variables, such as elevation or temperature using color scales or isolines.
- Choropleth map display variables characterizing whole region using colors. The variable used in choropleth should be intensive, e.g. normalized by area or population.

Several useful libraries:

- Geopandas for working with geographical data, extension of DataFrame
- Geoplot and Plotly for visualization

1.3 Graphs and hierarchies

1.3.1 Graphs

- A **graph / network** consists of **vertices** (vrcholy; also nodes, uzly) and **edges** (hrany; also links, arcs).
- Vertices often represent real-world **entities**.
- Edges often represent **relationships** and connections between pairs of vertices.

Many real-world examples of graphs: places connected by roads, computers connected by network cables, people connected by family or work relationships, companies connected by financial transactions, texts connected by references, tasks or courses connected by dependencies, any objects connected based on similarity / shared features.

- Edges can be directed (orientované) or undirected depending on whether the relationship is symmetrical.
- Recall: how did you define directed / undirected edges in discrete mathematics?

Graphs are very important in both computer science and data science. They are covered in several courses: discrete mathematics, programming, design of efficient algorithms, network science.

1.3.2 Trees and hierarchies

- An undirected graph is called a **tree** if it is connected and without cycles.
- In practice we usually encounter rooted (directed) trees, which have a single **root**, all other vertices can be reached from the root via a unique path.
- This gives rise to parent / child relationships between nodes (parent is the node closer to the root).
- Trees can express hierarchies in which each entity has a single direct superior, for example:
  - company structure in which each employee (except for the head of the company) has a single supervisor (similarly army command),
  - administrative divisions (country, region, district),

    - species taxonomy (animals, mammals, primates, …).
- However some hierarchies allow multiple direct superiors, for example:
    - family tree where each person has two parents (and they may be distantly related),
    - geometrical shapes, where a square is both a special case of a regular polygon and a special case of a rectangle and both of these are a special case of a polygon.
- These hierarchies can be represented as directed acyclic graphs.
    - Acyclic means that by following edges we never get back to the starting node (nobody is their own ancestor).

1.3.3 What do we study / visualize in real-life graphs?

- Details of connections for a particular node (requires zooming in large networks).
- Overall structure of the graph: connected components, density of edges, presence of cycles, weak places (bridges and articulations), clusters of densely connected nodes.
- Do nodes with some property cluster together? (Are they connected by many edges?)

See for example character co-occurrence in Shakespeare's tragedies.

1.3.4 Basics of graph drawing

- Vertices are typically displayed as markers (circles, rectangles etc.), possibly with labels, size, color, …
- Edges are displayed as lines connecting them, possibly of different color or width. They can be straight lines, arcs, polygonal lines or arbitrary curves.
- Edge direction displayed as arrows or in a hierarchy edges may be drawn to point in one direction, e.g. downwards.

Desirable properties:

- Nodes do not overlap.
- Edges are not too long and have a simple shape without many bends.
- The number of edge crossings is small.
- The graph uses the space of the figure well without large empty regions.

Node positioning:

- Sometimes the position of nodes is given by their properties, e.g. on a map (see airline connections), level of a hierarchy, timeline.
- Otherwise we try to place nodes to optimize desirable properties, e.g. using force-directed layout, which assigns attractive forces (springs) between nodes connected by edges and repulsive forces between other pairs of nodes.

Examples:

- https://en.wikipedia.org/wiki/Graphviz#/media/File:UnitedStatesGraphViz.svg
- https://upload.wikimedia.org/wikipedia/commons/9/90/Visualization_of_wiki_structure_using_prefuse_

1.3.5 Displaying a simple hierarchy in NetworkX

- We start by creating a simple tree representing taxonomy of selected even-toed ungulates (párnokopytníky) as a Pandas `DataFrame`.

- Each row of the data frame describes each node, giving its name, parent, level along the tree (leaves are 1, root is 5) and category, which is `land` for land animals, `sea` for sea animals and `group` for taxonomy groups.
- Group `Artiodactyla` is the root without a parent.

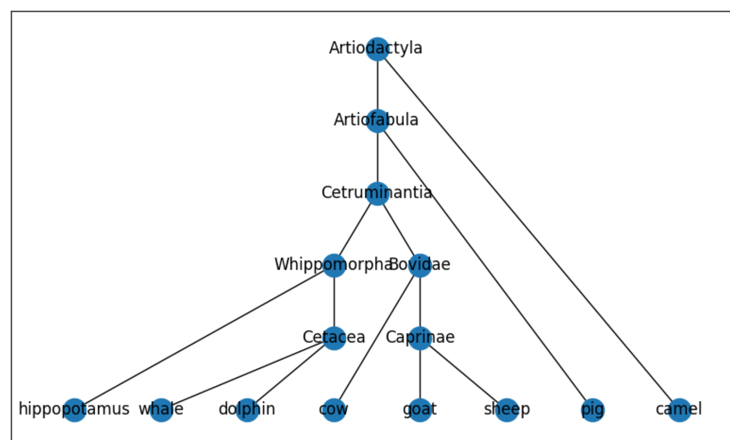
[20]:
```
from io import StringIO

animal_csv = StringIO("""name,parent,level,category
camel,Artiodactyla,1,land
pig,Artiofabula,1,land
sheep,Caprinae,1,land
goat,Caprinae,1,land
cow,Bovidae,1,land
dolphin,Cetacea,1,sea
whale,Cetacea,1,sea
hippopotamus,Whippomorpha,1,land
Caprinae,Bovidae,2,group
Cetacea,Whippomorpha,2,group
Bovidae,Cetruminantia,3,group
Whippomorpha,Cetruminantia,3,group
Cetruminantia,Artiofabula,4,group
Artiofabula,Artiodactyla,5,group
Artiodactyla,,6,group""")

animals = pd.read_csv(animal_csv)
animals['category'] = animals['category'].astype('category')
display(animals)
```

	name	parent	level	category
0	camel	Artiodactyla	1	land
1	pig	Artiofabula	1	land
2	sheep	Caprinae	1	land
3	goat	Caprinae	1	land
4	cow	Bovidae	1	land
5	dolphin	Cetacea	1	sea
6	whale	Cetacea	1	sea
7	hippopotamus	Whippomorpha	1	land
8	Caprinae	Bovidae	2	group
9	Cetacea	Whippomorpha	2	group
10	Bovidae	Cetruminantia	3	group
11	Whippomorpha	Cetruminantia	3	group
12	Cetruminantia	Artiofabula	4	group
13	Artiofabula	Artiodactyla	5	group
14	Artiodactyla	NaN	6	group

- NetworkX is a large library for working with graphs, it implements many graph algorithms.
- Below we convert our DataFrame to `Graph` class from NetworkX by adding nodes and edges.
- Nodes and edges can have arbitrary attributes attached, here `level` and `category`.

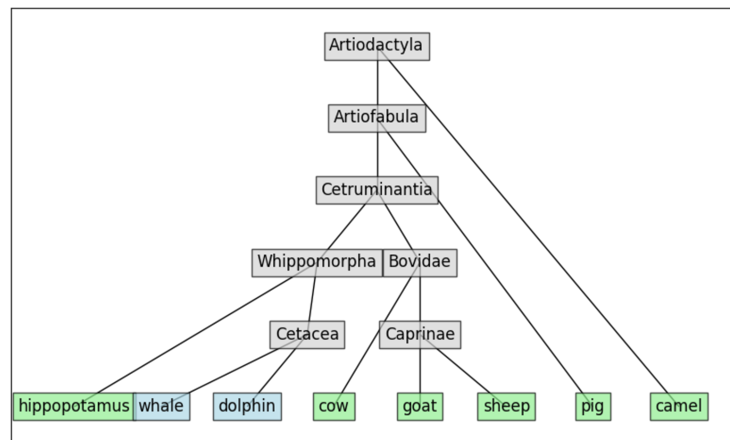
- Then we plot basic representation of the graph.
- The plotting works in two steps: first we compute coordinates of all nodes using `multipartite_layout`.
- Then we plot the network using `draw_networkx` into Matplotlib axes.
- The plot is not very nice, we will improve it below.

[21]:
```
# create empty graph in NetworkX
G = nx.Graph()

# adding each table row as a node
for index, row in animals.iterrows():
  G.add_node(row['name'], level=row['level'], category=row['category'])

# adding an edge to each node from its parent
for index, row in animals.iterrows():
  if row['parent'] is not np.nan:
    G.add_edge(row['parent'], row['name'])

# computing coordinates of nodes
coordinates = nx.multipartite_layout(G, subset_key="level", align='horizontal')
# drawing the graph
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx(G, coordinates, ax=axes)

pass
```



- Below we show an improved version of the plot.
- Two nodes are moved manually to reduce overlaps.
- Edges are plotted first.
- Then we draw node lables as boxes with text.
- Each category of nodes is draw separately with a different background color.

[22]:
```
# again compute coordinates and move some of them manually
coordinates2 = nx.multipartite_layout(G, subset_key="level", align='horizontal')
coordinates2["Whippomorpha"] += (-0.05, 0)
coordinates2["Cetacea"] += (-0.08, 0)

# plot edges only, omit nodes for now
(figure, axes) = plt.subplots(figsize=(10, 6))
nx.draw_networkx_edges(G, coordinates2, ax=axes)

# plot each category of nodes by a different color
color_dict = {'group':'lightgray', 'land':'lightgreen', 'sea':'lightblue'}
for category in color_dict:
  # create a list of nodes on the category
  category_nodes = [v for v in G.nodes if G.nodes[v]['category']==category]
  # select subgraph H of G
  H = G.subgraph(category_nodes)
  # create a dictionary of node label attributes
  label_options = {"ec": "black", "fc": color_dict[category], "alpha": 0.7}
  # draw the node labels as boxes
  nx.draw_networkx_labels(H, coordinates2, font_size=12, bbox=label_options,␣
 ↪ax=axes)

pass
```



1.3.6 Hierarchy as a treemap in Plotly Express

- PlotlyExpress can be used to easily create treemaps.
- Leaves of the tree are empty labeled rectangles, upper categories are enclosing boxes.
- To select box colors, we use `color_dict` created above.

[23]:
```
import plotly.express as px
fig = px.treemap(
    names=animals['name'],
    parents=animals['parent'],
    color=animals['category'],
    color_discrete_map=color_dict
)
fig.show()
```

1.3.7 Book character connections in Pyvis

- Here we use an example network from the NetworkX library.
- It represents character co-occurence in the novel Les Misérables by Victor Hugo.
- Edges are weighted by how often characters co-occur.
- To make the plot interactive, we use Pyvis library.
- We convert NetworkX graph to `Network` class from Pyvis.
- We add two new features for each node: `title` (used as a tooltip, name of the character) and `value` (used as a size of the node, representing its number of neighbors, i.e. degree).
- Visualization is saved as a HTML file which is then displayed using HTML class from IPython library.

[24]:
```
# initializing an empty network, setup plot properties
pyvis_net = Network("500px", "500px", notebook=True, cdn_resources='in_line')
# loading network from NetworkX
pyvis_net.from_nx(nx.les_miserables_graph())

# get a dictionary of neighbors for each node
neighbors = pyvis_net.get_adj_list()
# add additional node properties
# used as tooltip and size
for node in pyvis_net.nodes:
    node["title"] = node["id"]
    node["value"] = len(neighbors[node["id"]])

# saving the visualization in an html file
pyvis_net.show("net.html")
# displaying the html file in the notebook
from IPython.display import display, HTML
display(HTML('net.html'))
pass
```

```
net.html
```

1.3.8 Summary of graphs

- Graphs are important in many applications.
- NetworkX library has many functions for working with graphs, including several layout algorithms for visualization.

- Pyvis allows interactive visualization of graphs.
- Plotly can visualize trees as treemaps.

1.4 Time series (časové rady)

- Time series are sequences of measurements or values over time (in regular or irregular time intervals).
- Typically displayed as a line graph, with time as x-axis, time flowing from left to right (a cultural convention in western countries).
- Other options for drawing time series exist (bar graphs, heat maps, box plots, …).

Typical features of a time series:

- overall trend (increasing / decreasing / flat; rate of change),
- seasonality (daily / weekly / yearly cycles),
- noise (general variability / outliers)

We have seen some examples in the first lecture:

- Playfair's atlas, foreign trade
- Hockey stick graph of global temperature (Fig.3a)

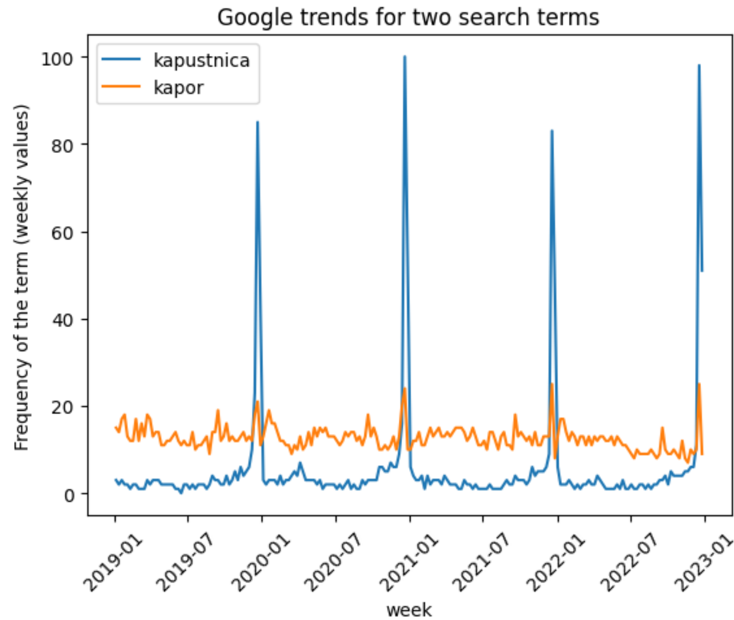
1.4.1 Two Google trend time series

- Google trends allow users to compare frequency of search terms over time and to each other.
- Here we use Christmas-related terms *kapustnica* and *kapor*.

[25]:
```
url = "https://bbrejova.github.io/viz/data/kapustnica-kapor.csv"
trends1 = pd.read_csv(url, parse_dates=['week']).set_index('week')
display(trends1.head())
```

week	kapustnica	kapor
2019-01-06	3	15
2019-01-13	2	14
2019-01-20	3	17
2019-01-27	2	18
2019-02-03	2	13

[26]:
```
axes = sns.lineplot(trends1, dashes=False)
axes.set_ylabel("Frequency of the term (weekly values)")
axes.set_title("Google trends for two search terms")
# rotate tick labels
axes.tick_params(axis='x', labelrotation = 45)
pass
```



- With kapustnica we see a clear seasonal trend.
- But kapor behaves differently. As related search terms Google reports zbgis kataster, zbgis mapa, zgbis, katasterportal list vlastnictva, dážďovka. Can you explain this?

1.4.2 Smoothing data (vyrovnanie, vyhladenie)

- Time series above is measured weekly and is quite noisy.
- We can smooth the data e.g. by **aggregating** them in longer time intervals. Here we compute mean value in each month (4 or 5 weeks).
- This is done using `resample` method from Pandas.
- An alternative is to use a **sliding window** (kĺzavé okno), where we choose a window size. e.g. 4 weeks and compute a new series, each value being mean or other summary of 4 consecutive windows in the input.
- For example with values 2,6,4,2,8,2 and window size 4, we get window means 3.5, 5, 4.

[27]:
```
# aggregate google trends from weekly to monthly mean (ME means month-end, in␣
↪older versions use M)
trends1monthly = trends1.resample('ME').mean()
display(Markdown("**New table of monthly means** (the first 5 rows)"))
display(trends1monthly.head())
display(Markdown("**The number of values aggregated in each month** (the first␣
↪5 values)"))
display(trends1.resample('ME').size().head())
```

New table of monthly means (the first 5 rows)

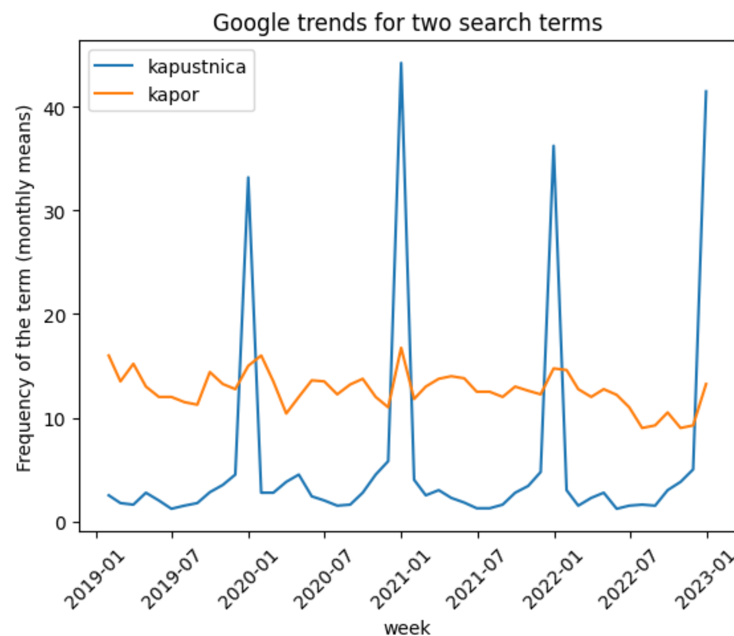
```
            kapustnica  kapor
week
```

```
2019-01-31         2.50   16.0
2019-02-28         1.75   13.5
2019-03-31         1.60   15.2
2019-04-30         2.75   13.0
2019-05-31         2.00   12.0
```

The number of values aggregated in each month (the first 5 values)

```
week
2019-01-31    4
2019-02-28    4
2019-03-31    5
2019-04-30    4
2019-05-31    4
Freq: ME, dtype: int64
```

[28]:
```
axes = sns.lineplot(trends1monthly, dashes=False)
axes.set_ylabel("Frequency of the term (monthly means)")
axes.set_title("Google trends for two search terms")
axes.tick_params(axis='x', labelrotation = 45)
pass
```



1.4.3 Trend: temperatures are growing in spring

- We will also look at a dataset displaying a trend: series of temperature values from Piešťany from January to June 2010, downloaded from US National Oceanic and Atmospheric Administration.
- We show both the original data and values smoothed with rolling average.

[29]:
```python
# read a dataset of temperatures in Piestany
url="https://bbrejova.github.io/viz/data/piestany-weather.csv"
weather = pd.read_csv(url, parse_dates=['DATE']).set_index('DATE')
# select only columns with daily maximum temperatures
temperature = weather["TMAX"]
# select only period from January to June 2010
spring2010 = temperature[pd.Timestamp('2010-01-01'):pd.Timestamp('2010-06-30')]
display(spring2010)
```

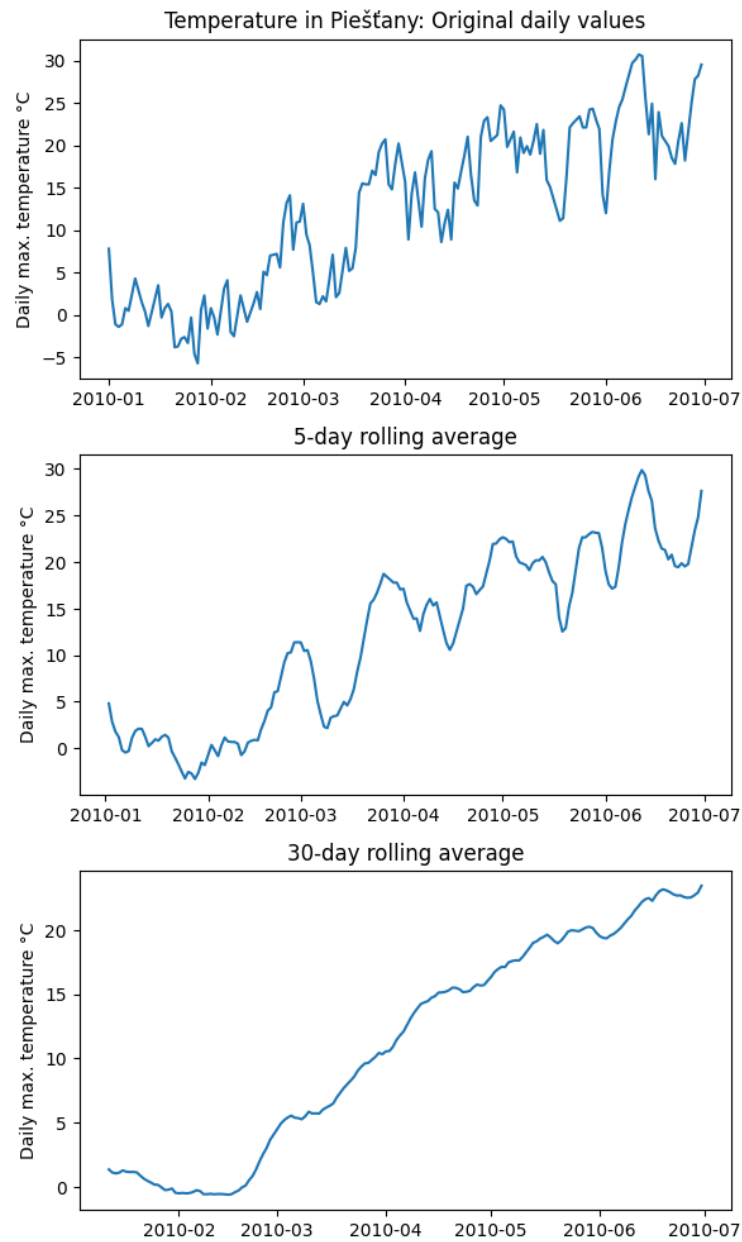
```
DATE
2010-01-01     7.8
2010-01-02     1.8
2010-01-03    -1.1
2010-01-04    -1.4
2010-01-05    -1.1
              …
2010-06-26     NaN
2010-06-27    25.0
2010-06-28    27.8
2010-06-29    28.2
2010-06-30    29.5
Name: TMAX, Length: 181, dtype: float64
```

[30]:
```python
# compute rolling averages in a window of 5 and 30 days
spring2010rolling5 = spring2010.rolling(5, min_periods=2).mean()
spring2010rolling30 = spring2010.rolling(30, min_periods=10).mean()
spring2010rolling5.head(10)
```

[30]:
```
DATE
2010-01-01         NaN
2010-01-02    4.800000
2010-01-03    2.833333
2010-01-04    1.775000
2010-01-05    1.200000
2010-01-06   -0.200000
2010-01-07   -0.460000
2010-01-08   -0.300000
2010-01-09    1.125000
2010-01-10    1.866667
Name: TMAX, dtype: float64
```

[31]:
```python
(figure, axes) = plt.subplots(3, 1, figsize=(6, 10))
sns.lineplot(spring2010, ax=axes[0])
sns.lineplot(spring2010rolling5, ax=axes[1])
sns.lineplot(spring2010rolling30, ax=axes[2])
axes[0].set_title("Temperature in Piešťany: Original daily values")
axes[1].set_title("5-day rolling average")
```

```
axes[2].set_title("30-day rolling average")
for i in range(3):
  axes[i].set_ylabel("Daily max. temperature °C")
  axes[i].set_xlabel(None)
figure.tight_layout(pad=1.0)
pass
```



1.4.4 Overlapping timescales to display seasonality

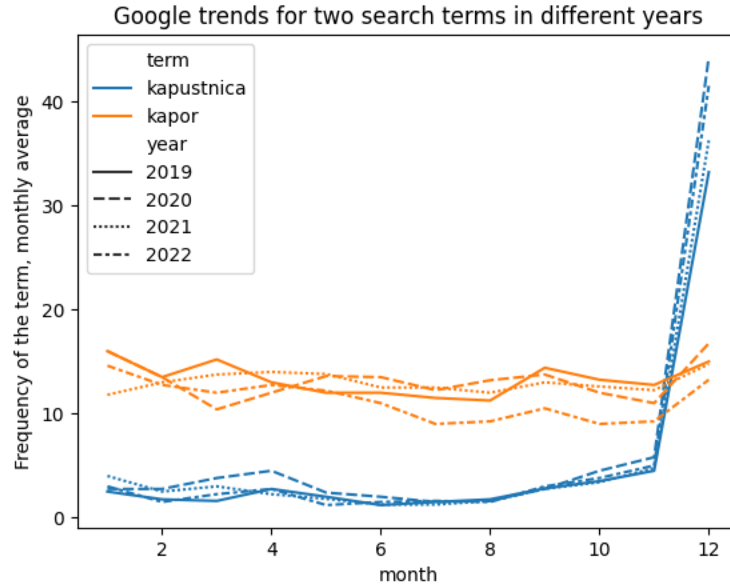
- We can better see cyclical trends if we plot each cycle on the same x-axis scale.
- In our Google example, we will use the month as the x axis and plot individual years as lines.

[32]:
```
# convert monthly table to long format with separate rows for kapustnica and␣
↪kapor
trends1monthlyLong = trends1monthly.reset_index().melt(id_vars=['week'])
trends1monthlyLong.rename(columns={'variable':'term', 'value':'frequency'},␣
↪inplace=True)
# create separate columns with year and month
trends1monthlyLong['month'] = trends1monthlyLong['week'].dt.month
trends1monthlyLong['year'] = trends1monthlyLong['week'].dt.year
display(Markdown("**Monthly table in the long format**"))
display(trends1monthlyLong)
```

Monthly table in the long format

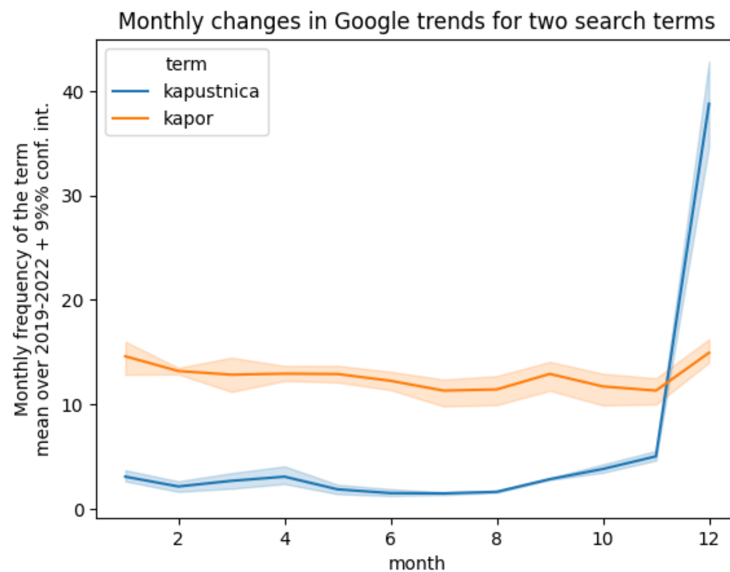
```
          week        term  frequency  month  year
0   2019-01-31  kapustnica       2.50      1  2019
1   2019-02-28  kapustnica       1.75      2  2019
2   2019-03-31  kapustnica       1.60      3  2019
3   2019-04-30  kapustnica       2.75      4  2019
4   2019-05-31  kapustnica       2.00      5  2019
..         …           …         …      …     …
91  2022-08-31       kapor       9.25      8  2022
92  2022-09-30       kapor      10.50      9  2022
93  2022-10-31       kapor       9.00     10  2022
94  2022-11-30       kapor       9.25     11  2022
95  2022-12-31       kapor      13.25     12  2022

[96 rows x 5 columns]
```

[33]:
```
# use month as x, separate years by line style and search terms by color
axes = sns.lineplot(trends1monthlyLong, x='month', y='frequency', hue='term',␣
↪style='year')
axes.set_title("Google trends for two search terms in different years")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



- We can see that across years the trend is quite stable.
- Below we see another version of the figure where multiple lines for years are replaced with mean and its 95% confidence interval expressing our uncertainty in the true value of the mean due to noise in data.

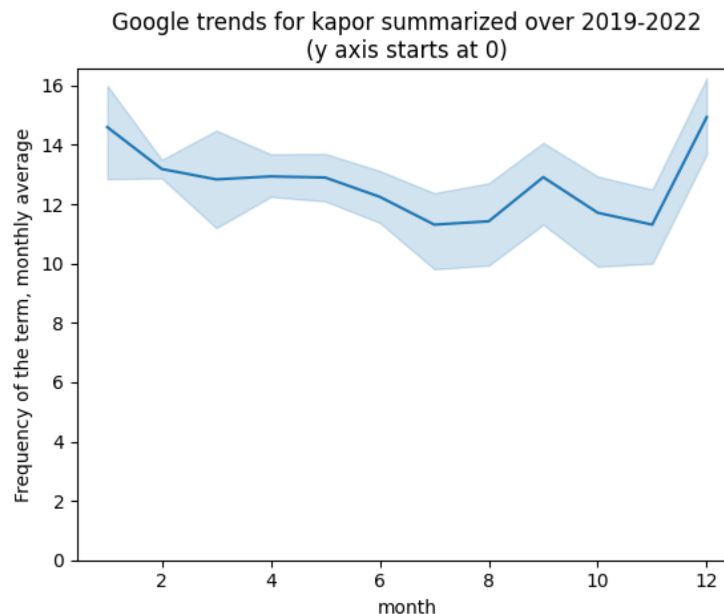
[34]:
```
axes = sns.lineplot(trends1monthlyLong, x='month', y='frequency', hue='term')
axes.set_title("Monthly changes in Google trends for two search terms")
axes.set_ylabel("Monthly frequency of the term\nmean over 2019-2022 + 9%% conf.␣
↪int.")
pass
```



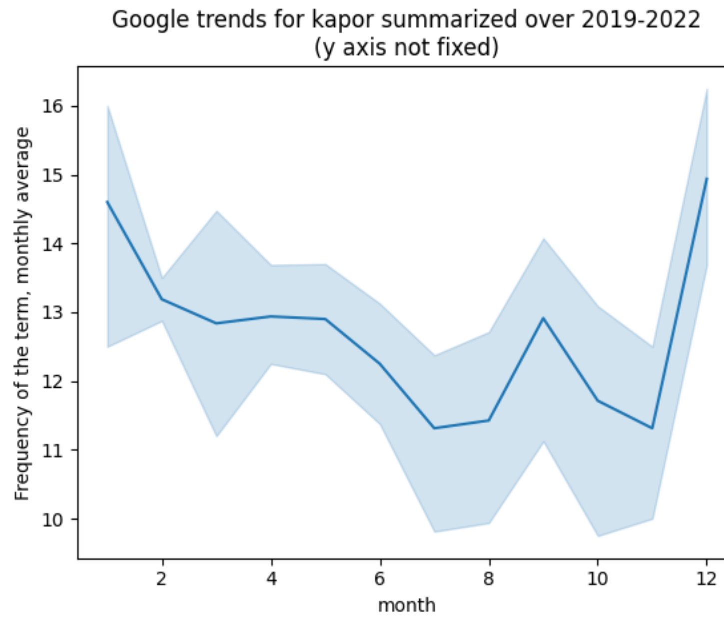
1.4.5 Importance of scales

- The plot below shows that if we do not start y axis at 0, differences in kapor searches may appear exaggerated.
- The next two plots show that even with y axis starting at 0, the time series may appear more variable with narrower aspect ratio of the figure.

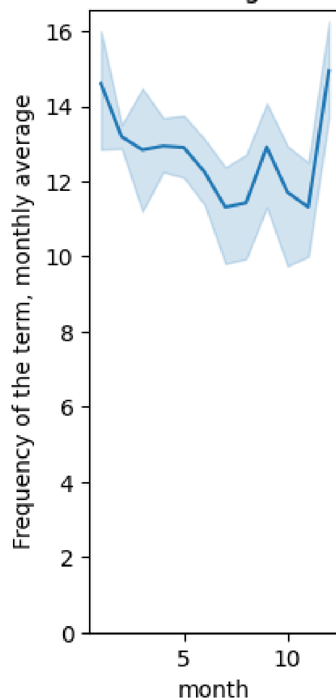
[35]:
```
kapor = trends1monthlyLong.query("term=='kapor'")
axes = sns.lineplot(kapor, x='month', y='frequency')
axes.set_title("Google trends for kapor summarized over 2019-2022\n(y axis␣
↪starts at 0)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



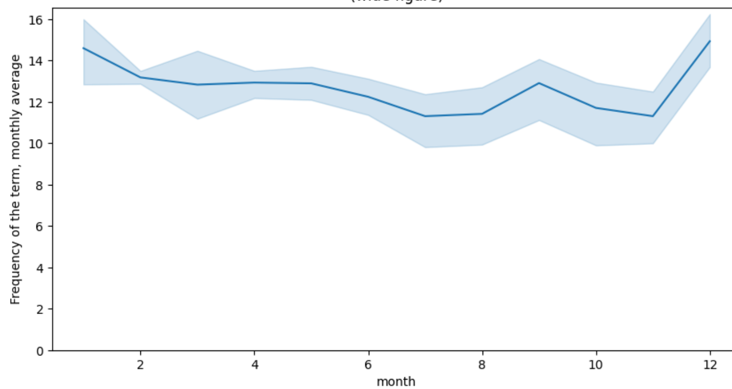
[36]:
```
axes = sns.lineplot(kapor, x='month', y='frequency')
axes.set_title("Google trends for kapor summarized over 2019-2022\n(y axis not␣
↪fixed)")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



[37]:
```
(figure, axes) = plt.subplots(figsize=(2,5))
sns.lineplot(kapor, x='month', y='frequency', ax=axes)
axes.set_title("Google trends for kapor summarized over 2019-2022\n(narrow␣
↪figure)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



[38]:
```
(figure, axes) = plt.subplots(figsize=(10,5))
sns.lineplot(kapor, x='month', y='frequency', ax=axes)
axes.set_title("Google trends for kapor summarized over 2019-2022\n(wide␣
↪figure)")
axes.set_ylabel("Frequency of the term, monthly average")
axes.set_ylim(ymin=0)
pass
```



1.4.6 Relative scales

- When we care about rate of increase or decrease, it might be better to express values as a percentage compared to initial value.
- Here we compare values in each month with values in January of the same year.
- In this way even two time series with quite different values can be plotted in the same plot (e.g. revenue of a small and a large company and their relative changes within a year).

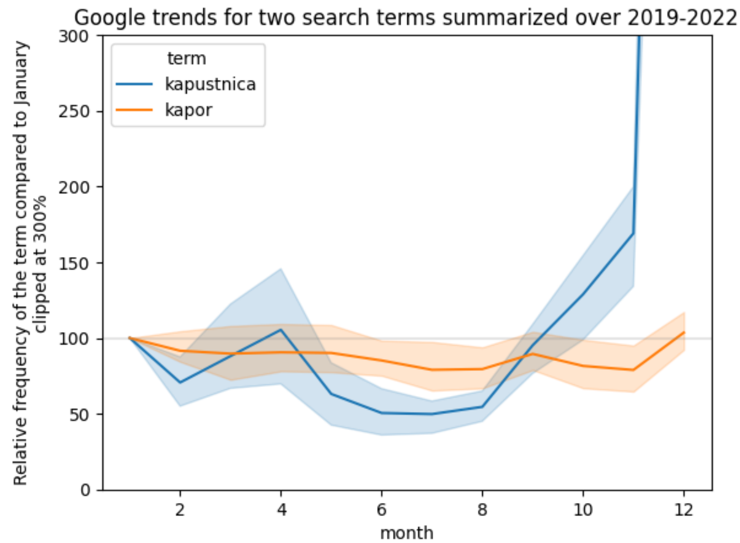
[39]:
```
# compute relative values by transforming each group of monthly values
# by dividing them by the first value (January)
relValue = (trends1monthlyLong.groupby(['year','term'])['frequency']
            .transform(lambda x : x * 100 / x.iloc[0]))
# add relative values as a column to the long table
relTable = trends1monthlyLong.assign(relValue=relValue)
display(Markdown("**Relative values added**"))
relTable.head()
```

Relative values added

[39]:

	week	term	frequency	month	year	relValue
0	2019-01-31	kapustnica	2.50	1	2019	100.0
1	2019-02-28	kapustnica	1.75	2	2019	70.0
2	2019-03-31	kapustnica	1.60	3	2019	64.0
3	2019-04-30	kapustnica	2.75	4	2019	110.0
4	2019-05-31	kapustnica	2.00	5	2019	80.0

[40]:
```
axes = sns.lineplot(relTable, x='month', y='relValue', hue='term')
axes.set_ylim(ymin=0, ymax=300)
axes.axhline(100, color="gray", alpha=0.2)
axes.set_title("Google trends for two search terms summarized over 2019-2022")
axes.set_ylabel("Relative frequency of the term compared to January\nclipped at␣
↪300%")

pass
```



1.4.7 One more pair of Google trend lines

- Again very seasonal: lyže and vlek.
- This time we have monthly data over a longer period of time.
- We display the original data as well as yearly seasonal trend.
- The peak month is January for both queries, so we also display January values changing over the years.

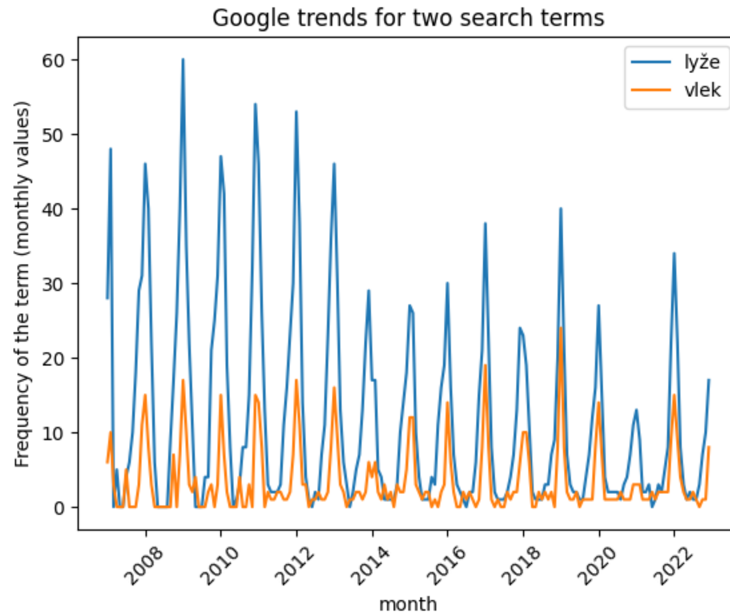
[41]:

```
url = "https://bbrejova.github.io/viz/data/lyze-vlek.csv"
trends2 = pd.read_csv(url, parse_dates=['month']).set_index('month')
display(trends2.head())
```

	lyže	vlek
month		
2007-01-01	28	6
2007-02-01	48	10
2007-03-01	0	3
2007-04-01	5	0
2007-05-01	0	0

[42]:

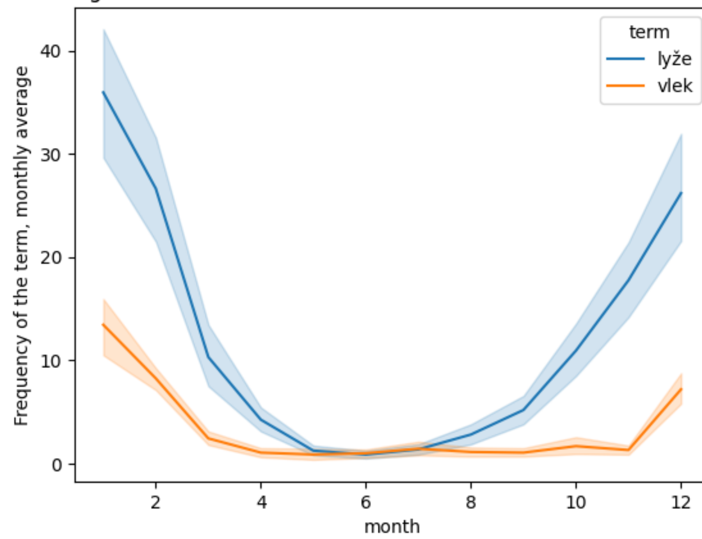
```
axes = sns.lineplot(trends2, dashes=False)
axes.set_title("Google trends for two search terms")
axes.set_ylabel("Frequency of the term (monthly values)")
# rotate tick labels
axes.tick_params(axis='x', labelrotation = 45)
pass
```



[43]:
```
trends2long = trends2.reset_index().melt(id_vars=['month'])
trends2long.rename(columns={'month':'date','variable':'term', 'value':
↪'frequency'}, inplace=True)
trends2long['month'] = trends2long['date'].dt.month
trends2long['year'] = trends2long['date'].dt.year
display(trends2long)
```

```
           date  term  frequency  month  year
0    2007-01-01  lyže         28      1  2007
1    2007-02-01  lyže         48      2  2007
2    2007-03-01  lyže          0      3  2007
3    2007-04-01  lyže          5      4  2007
4    2007-05-01  lyže          0      5  2007
..          …     …          …      …     …
379  2022-08-01  vlek          1      8  2022
380  2022-09-01  vlek          0      9  2022
381  2022-10-01  vlek          1     10  2022
382  2022-11-01  vlek          1     11  2022
383  2022-12-01  vlek          8     12  2022

[384 rows x 5 columns]
```

[44]:
```
axes = sns.lineplot(trends2long, x='month', y='frequency', hue='term')
axes.set_title("Google trends for two search terms summarized over 2007-2022")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



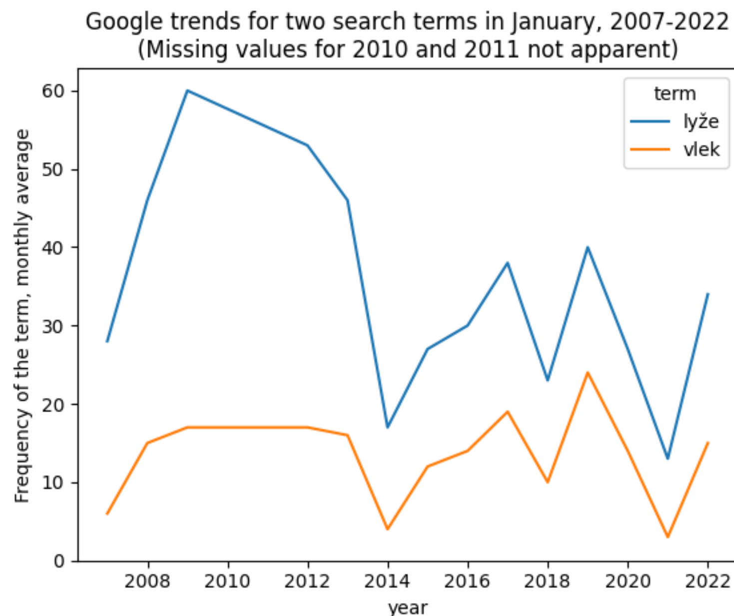
[45]:
```
lyzeJan = trends2long.query("month==1")
axes = sns.lineplot(lyzeJan, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January, 2007-2022")
axes.set_ylabel("Frequency of the term, monthly average")

pass
```

The drop in 2021 was due to pandemics, but what about 2014 and 2018?

1.4.8 Acknowledging missing values

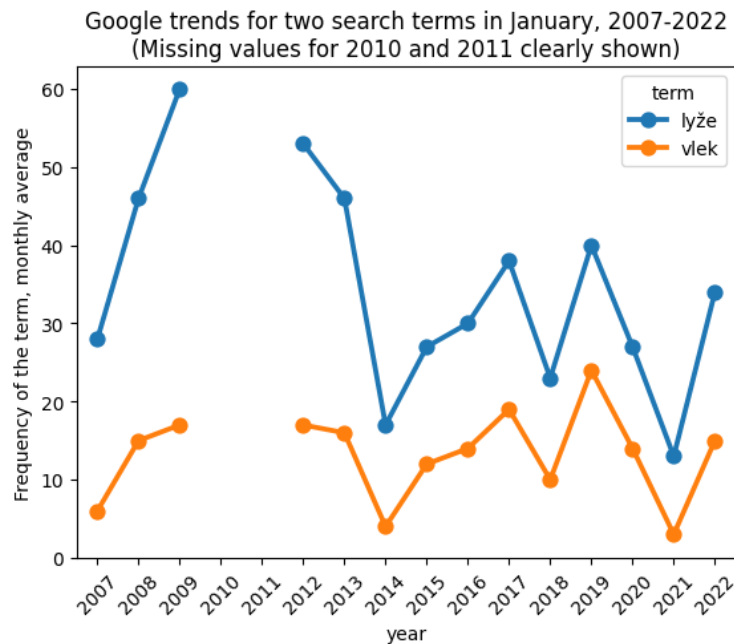
- Let us imagine that January values for 2010 and 2011 are missing.
- If we draw a lineplot in Seaborn, years 2009 and 2012 are connected by a straight line and viewer does not know that something is missing.
- This is not a good idea.
- Below we use pointplot which nicely shows the missing data and also locations of measured values.

[46]:
```
# remove values for two years
to_remove = lyzeJan["year"].isin([2010,2011])
lyzeJanMissing = lyzeJan.copy(deep=True)
lyzeJanMissing.loc[to_remove, 'frequency'] = np.nan
axes = sns.lineplot(lyzeJanMissing, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January,␣
↪2007-2022\n(Missing values for 2010 and 2011 not apparent)")
axes.set_ylabel("Frequency of the term, monthly average")
pass
```



[47]:
```
axes = sns.pointplot(lyzeJanMissing, x='year', y='frequency', hue='term')
axes.set_ylim(ymin=0)
axes.set_title("Google trends for two search terms in January,␣
↪2007-2022\n(Missing values for 2010 and 2011 clearly shown)")
axes.set_ylabel("Frequency of the term, monthly average")
```

```
axes.tick_params(axis='x', labelrotation = 45)
pass
```



1.4.9 Summary of time series

Typical goals are to observe and study:

- overall trend (increasing / decreasing / flat; rate of change),
- seasonality (daily / weekly / yearly cycles),
- noise (general variability / outliers)

Useful techniques:

- smoothing by aggregation and sliding window
- overlapping timescales
- relative scales
- showing uncertainty and missing values

1 Lecture 7: More statistics

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

1.1 Importing libraries and data

As usual, we start by importing libraries. We add `scipy.stats` library for working with probability distributions. One more library will be added at the end of the lecture.

[29]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from IPython.display import Markdown

import scipy.stats
```

In this lecture, we will use the World bank dataset from Lecture 03b (table `countries`) downloaded from WorldBank https://databank.worldbank.org/home under CC BY 4.0 license.

We will also work with an informal survey of preferences and opinions of young people done in 2013 among students of FSEV UK and their friends. Datatset was downloaded from https://www.kaggle.com/miroslavsabo/young-people-survey . Columns correspond to survey questions, rows to respondents. The list of questions and meaning of responses is given in this document.

[30]:
```
url = 'https://bbrejova.github.io/viz/data/fsev-responses.csv'
fsev = pd.read_csv(url)
display(Markdown("**Size of fsev table:**"), fsev.shape)
```

Size of fsev table:

```
(1010, 150)
```

[31]:
```
url = 'https://bbrejova.github.io/viz/data/World_bank.csv'
countries = pd.read_csv(url).set_index('Country')
display(countries.describe())
```

```
       Population2000  Population2010  Population2020          Area  \
count    2.170000e+02    2.170000e+02    2.170000e+02  2.160000e+02
mean     2.821267e+07    3.201312e+07    3.592913e+07  6.235930e+05
std      1.157060e+08    1.282292e+08    1.398323e+08  1.828856e+06
min      9.638000e+03    1.024100e+04    1.106900e+04  1.000000e+01
25%      6.049510e+05    7.055170e+05    7.972020e+05  1.122250e+04
50%      5.056174e+06    5.768613e+06    6.579900e+06  1.017050e+05
75%      1.639406e+07    2.112004e+07    2.564925e+07  4.786050e+05
max      1.262645e+09    1.337705e+09    1.411100e+09  1.709825e+07
```

```
            GDP2000        GDP2010        GDP2020  Expectancy2000  \
count    199.000000     209.000000     210.000000      211.000000
mean    8292.850729   16135.528397   17547.135072       67.358588
std    13036.320032   24118.156125   26171.548454        9.795515
min      122.961660     222.660583     216.827417       44.518000
25%      654.638840    1706.414917    2262.246896       61.362000
50%     1996.515578    5735.422857    6370.903532       70.417073
75%    10721.262761   21447.858255   22805.261142       74.505000
max    81763.827669  161780.745361  182537.387370       81.370000

       Expectancy2010  Expectancy2020  Fertility2000  Fertility2010  \
count      210.000000      209.000000     211.000000     211.000000
mean        70.580052       72.309699       3.211021       2.890908
std          8.831970        7.482700       1.716951       1.490287
min         45.596000       52.777000       0.912000       1.042000
25%         64.480750       66.797000       1.840000       1.765000
50%         72.810500       72.889000       2.660000       2.340000
75%         77.578598       78.041000       4.395000       3.827000
max         83.109000       85.497561       7.732000       7.485000

       Fertility2020
count     211.000000
mean        2.525145
std         1.273400
min         0.837000
25%         1.557500
50%         2.040000
75%         3.257000
max         6.892000
```

[32]:
```
display(Markdown("**Values of life expectancy in 2020 in individual countries:
↪**"))
display(countries['Expectancy2020'].dropna())
```

Values of life expectancy in 2020 in individual countries:

```
Country
Afghanistan            62.575000
Albania                76.989000
Algeria                74.453000
Angola                 62.261000
Antigua and Barbuda    78.841000
                         …
Virgin Islands         79.819512
West Bank and Gaza     74.403000
Yemen                  64.650000
Zambia                 62.380000
Zimbabwe               61.124000
```

```
Name: Expectancy2020, Length: 209, dtype: float64
```

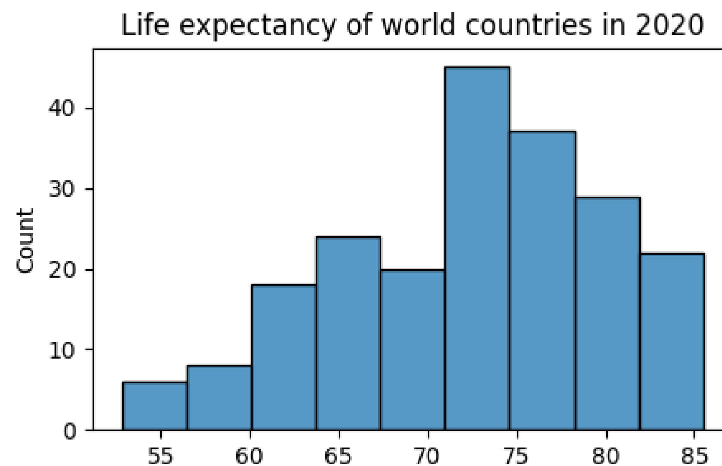
1.2 Histograms

Histograms are well known, and we have seen them in Lecture 03. We split the range of a variable into equally sized bins, count the number of data points in each bin and plot the counts as a bar graph.

Histograms allow us to observe many aspects of distribution of values of a variable:

- range of values, outliers
- central tendency
- unimodality / multimodality
- variance
- symmetry / skewness (šikmosť)

[33]:
```
axes = sns.histplot(data=countries, x='Expectancy2020')
axes.set_title('Life expectancy of world countries in 2020')
axes.set_xlabel(None)
axes.figure.set_size_inches(5, 3)
pass
```

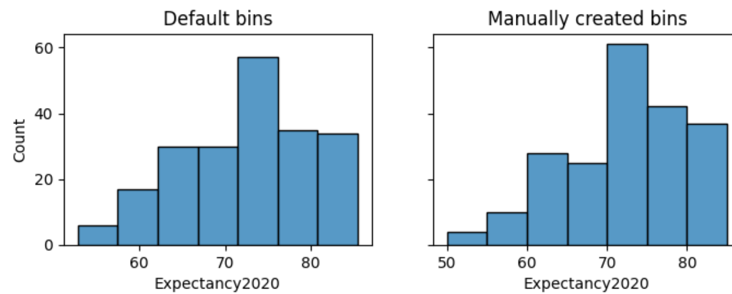


1.2.1 Custom bins

- Seaborn library makes bins by splitting the range into equally sized intervals, but perhaps a more meaningfull plot uses round values at bin boundaries, e.g. intervals of 5 years 50-55, 55-60, 60-65,…
- We can use manually created bin boundaries in Seaborn.
- Plotly library tries to create more meaningful bins by default.

[34]:
```
# create a figure with two plots
figure, axes = plt.subplots(1, 2, figsize=(8,2.5), sharey=True)
```

```
# the first plot has histogram with default bins of width 5
sns.histplot(data=countries, x='Expectancy2020', binwidth=5, ax=axes[0])
axes[0].set_title('Default bins')

# the second plot has manually set bin boundaries 50,55,60,...,85
sns.histplot(data=countries, x='Expectancy2020',
             bins=range(50, 90, 5), ax=axes[1])
axes[1].set_title('Manually created bins')
pass
```

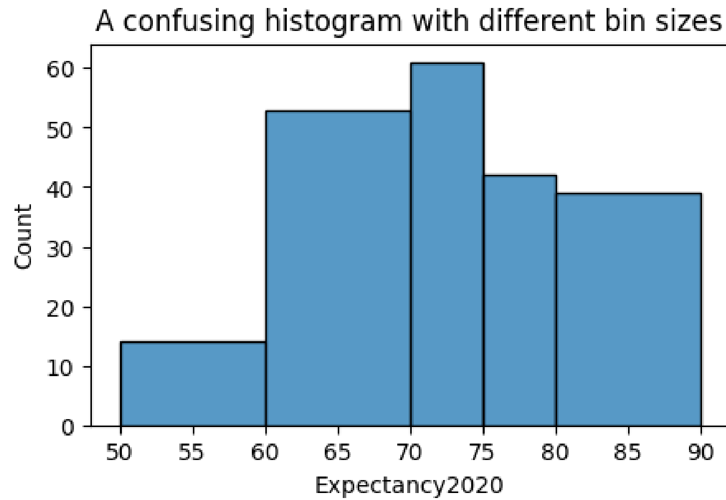


[35]:
```
# in Plotly, we specify the maximum number of bins,
# library may choose a lower number to get "nice" bin boundaries
fig = px.histogram(countries, x="Expectancy2020",
                   nbins=8, width=500, height=350)
fig.show()
```

1.2.2 Use equally-sized bins

- Manually created bin boundaries can be arbitrary, but if bin width is unequal, the resulting plot is confusing.
- If you really need special bins (e.g. age <18 years, 18-65 years, >65 years), make a categorical variable, then plot it as a bar graph (typically displayed as bars with equal width, spaces between bars), clearly mark the meaning of each bar.

[36]:
```
axes = sns.histplot(data=countries, x='Expectancy2020',
                    bins=[50, 60, 70, 75, 80, 90])
axes.set_title('A confusing histogram with different bin sizes')
axes.figure.set_size_inches(5, 3)
pass
```



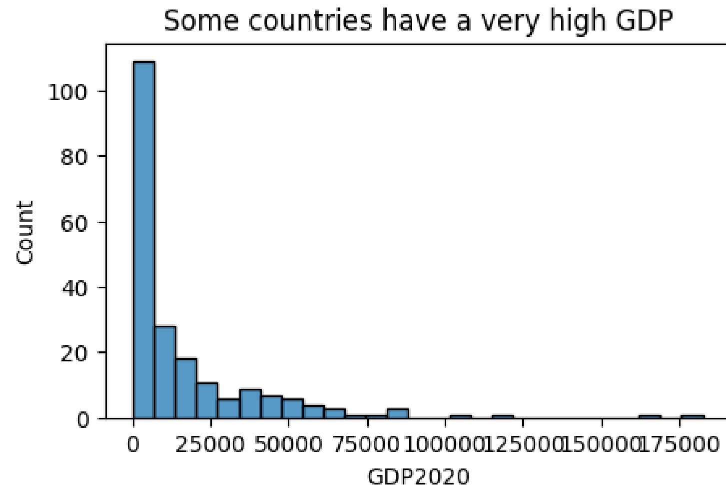
1.2.3 Removing outliers

- Histograms are great for spotting outliers.
- But extreme values reduce the space given to more regular values, so perhaps we want to remove them in subsequent analysis.

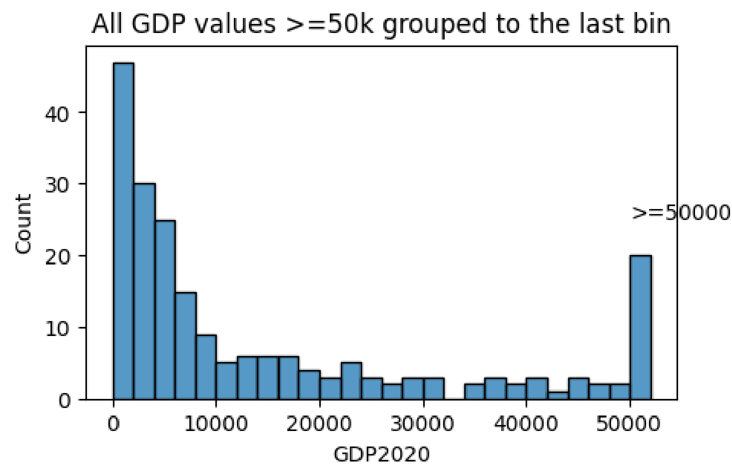
We have several options:

- Remove them from the dataset if we believe them to be errors.
- Or remove them from the plot only, e.g. by `set_xlim` or by using custom bins with a smaller range.
- Or clip values: replace values above some threshold with the threshold value (function `clip` in Pandas). Thus they will be present in the last bin. This bin should be then clearly marked.

[37]:
```
axes = sns.histplot(data=countries, x='GDP2020')
axes.set_title('Some countries have a very high GDP')
axes.figure.set_size_inches(5, 3)
pass
```



[38]:
```
# replace values larger than 51k with 51k
gdp_clipped = countries['GDP2020'].clip(0, 51000)
# make histogram with manual bins, with last bin 50k-52k
axes = sns.histplot(x=gdp_clipped, bins=np.arange(0, 53000, 2000))
axes.figure.set_size_inches(5, 3)
# mention clipping in plot title
axes.set_title('All GDP values >=50k grouped to the last bin')
# also add a text label to the bin with clipped values
axes.text(x=50000, y=25, s='>=50000')
pass
```



1.2.4 Problems with precision

When working with integers or even real numbers given with a small number of decimal points, we can get artifacts related to different number of possible values falling to different bins.

To illustrate this, we uniformly sample million points from the set $\{0, 0.01, 0.02, \ldots, 0.99\}$.

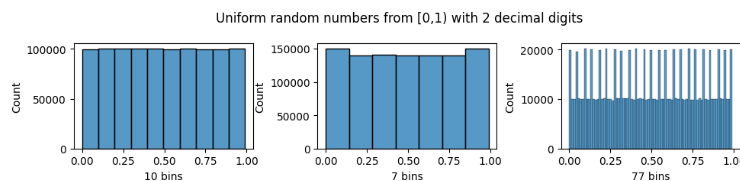
- There will be a similar number of samples for each possible value from this set.
- We show histograms with 10, 7 and 77 equally sized bins.
- For 10 bins, each bin summarizes 10 of the possible values and the sizes are approximately the same.
- For 7 bins, the first and the last bin summarize 15 possible values each and remaining bins summarize 14 possible values each. The first and last bin are thus slightly higher.
- For 77 bins, some bins summarize 2 different values, others only 1. We see clear differences in bar height.

If we are unaware of this, we may draw incorrect conclusions from the second and third plot.

[39]:
```
sample_uniform = np.random.randint(0, 100, 1000000) / 100
display(Markdown('**Example of data (first values):**'), sample_uniform[0:5])
figure, axes = plt.subplots(1, 3, figsize=(10,2.5))
figure.tight_layout(pad=3)
for (i, bin) in enumerate([10, 7, 77]):
  sns.histplot(x=sample_uniform, bins=bin, ax=axes[i])
  axes[i].set_xlabel(f"{bin} bins")
figure.suptitle("Uniform random numbers from [0,1) with 2 decimal digits")
pass
```

Example of data (first values):

array([0.44, 0.65, 0.72, 0.14, 0.46])



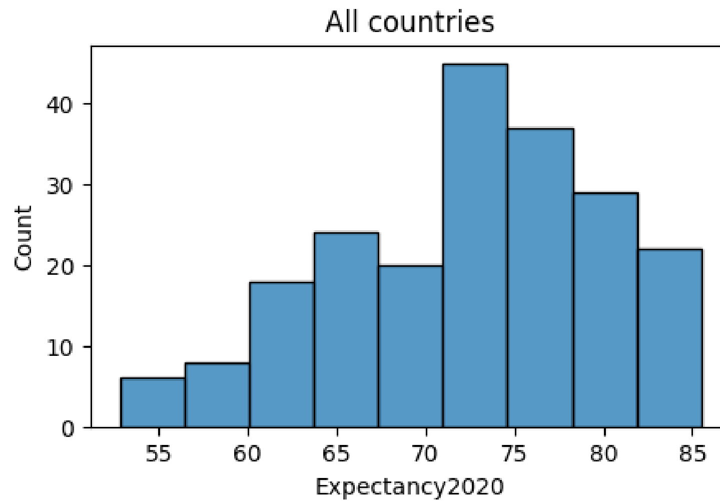
1.2.5 Small samples

Beware drawing strong conclusions from small samples.

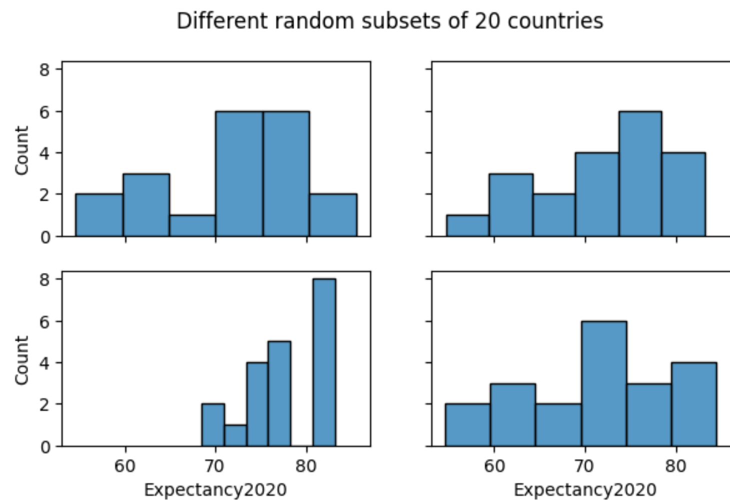
- Below we again first show the histogram of life expectancy over all countries.
- Then we show histograms of four random subsets of 20 countries each.
- Any estimates (including histograms) from small samples are subject to random noise.

[40]:
```
axes = sns.histplot(data=countries, x='Expectancy2020')
axes.set_title('All countries')
axes.figure.set_size_inches(5, 3)
```

```
pass
```



[41]:
```
figure, axes = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(7,4))
for row in axes:
  for subplot in row:
    expectancy_sample = countries['Expectancy2020'].dropna().sample(20)
    sns.histplot(x=expectancy_sample, ax=subplot)
figure.suptitle("Different random subsets of 20 countries")
pass
```



1.2.6 Summary: Histogram bin size

To summarize, the tricky part of using histograms is to choose the bin size or the number of bins.

Smaller bins mean more details are visible, but some of those details may be artefacts:

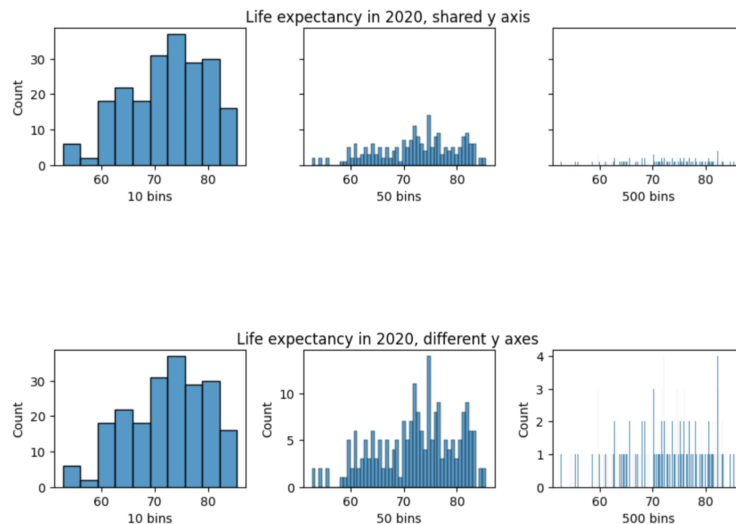
- random fluctuations due to small number of points in the bin, or
- effects related to insufficient resolution of the data.

Thus choose bin size based on:

- the amount of data,
- the precision of input values,
- the meaningful resulution of the results.

In the example below we show the life expectancy data with different number of bins. Do 50 or 500 bins show more meaningful information than 10 bins?

[42]:
```
# one set of plots has share y axis, one has not
figure1, axes1 = plt.subplots(1, 3, figsize=(10,2), sharey=True)
figure2, axes2 = plt.subplots(1, 3, figsize=(10,2))
# plot the same plots for each set
for (figure, axes, title) in [(figure1, axes1, 'shared y axis'),
                              (figure2, axes2, 'different y axes')]:
  # iterate over different numbers of bins
  for (i, bin) in enumerate([10, 50, 500]):
    sns.histplot(data=countries, x='Expectancy2020', bins=bin, ax=axes[i])
    axes[i].set_xlabel(f"{bin} bins")
  # title of the whole figure
  figure.suptitle(f"Life expectancy in 2020, {title}")
  figure.subplots_adjust(wspace=0.3)
pass
```

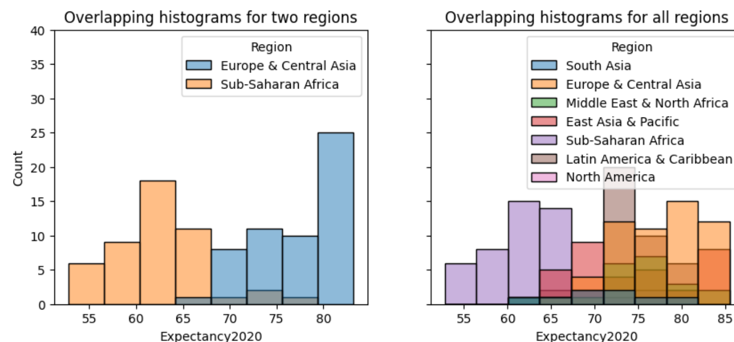


1.2.7 Comparing distributions with histograms

- We can compare distributions of a numerical variable split into groups by a categorical variable.
- For example in our `countries` table, we can compare life expectancy in different regions of the world.
- Seaborn provides several options for doing so.

The first two plots use semi-transparent overlapping histograms, which work well for two regions (left), but are a mess for many regions (right).

[43]:
```
# select two regions
countries_subset = countries.query('Region == "Europe & Central Asia" '
                                   + 'or Region == "Sub-Saharan Africa"')
# create figure with 2 plots
figures, axes = plt.subplots(1, 2, figsize=(10,4), sharey=True)
# plot both histograms
sns.histplot(data=countries_subset, x='Expectancy2020', hue='Region', ax =␣
↪axes[0])
sns.histplot(data=countries, x='Expectancy2020', hue='Region', ax=axes[1])
# make bigger y-axis to accomodate legend
axes[0].set_ylim(0,40)
# titles
axes[0].set_title('Overlapping histograms for two regions')
axes[1].set_title('Overlapping histograms for all regions')
pass
```

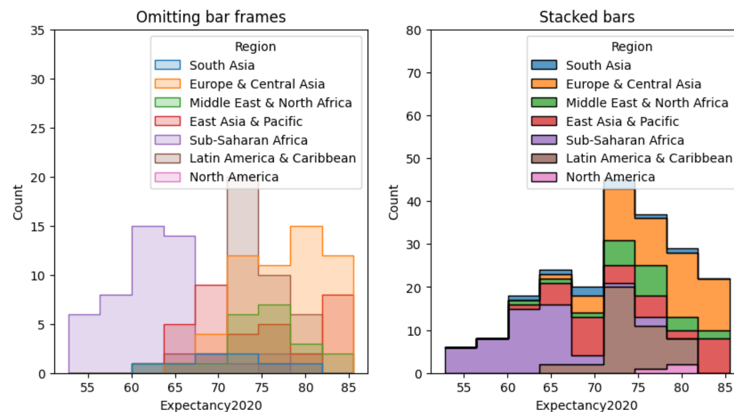


The next two plots attempt to improve the situation.

- On the left we omit bar outlines to simplify the plot.
- On the right we use stacked bars, showing contribution of each region to the whole.

[44]:
```
# create figure with 2 plots
figures, axes = plt.subplots(1, 2, figsize=(10, 5))
# plot both histograms
sns.histplot(data=countries, x='Expectancy2020', hue='Region',
```

```
             element='step', ax = axes[0])
sns.histplot(data=countries, x='Expectancy2020', hue='Region', element='step',
             multiple="stack", ax=axes[1])
# make bigger y-axis to accomodate legend
axes[0].set_ylim(0,35)
axes[1].set_ylim(0,80)
# titles
axes[0].set_title('Omitting bar frames')
axes[1].set_title('Stacked bars')
pass
```



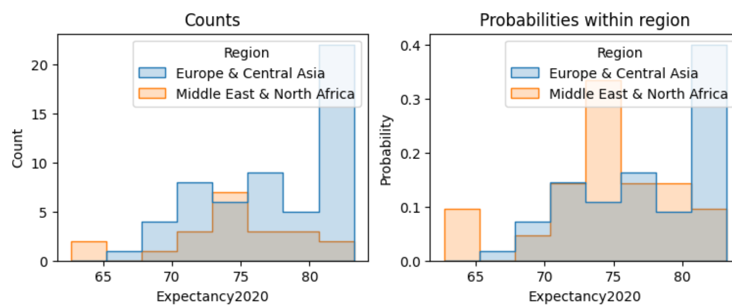
- Regions contain different number of countries.
- To better compare distribution of the expectancy within region, we should normalize the count to probabilities.
- Use `common_norm=False` to normalize each region separately.

[45]:
```
display(Markdown("**Countries in regions:**"))
display(countries.groupby('Region').size().sort_values())

# select two regions of very different sizes
countries_subset2 = countries.query('Region == "Europe & Central Asia" '
                                     + 'or Region == "Middle East & North␣
↪Africa"')
# plot counts and probabilities
figure, axes = plt.subplots(1, 2, figsize=(9,3))
sns.histplot(data=countries_subset2, x='Expectancy2020', hue='Region',␣
↪element='step', ax=axes[0])
sns.histplot(data=countries_subset2, x='Expectancy2020', hue='Region',␣
↪element='step',
             stat="probability", common_norm=False, ax=axes[1])
axes[0].set_title('Counts')
axes[1].set_title('Probabilities within region')
```

```
pass
```

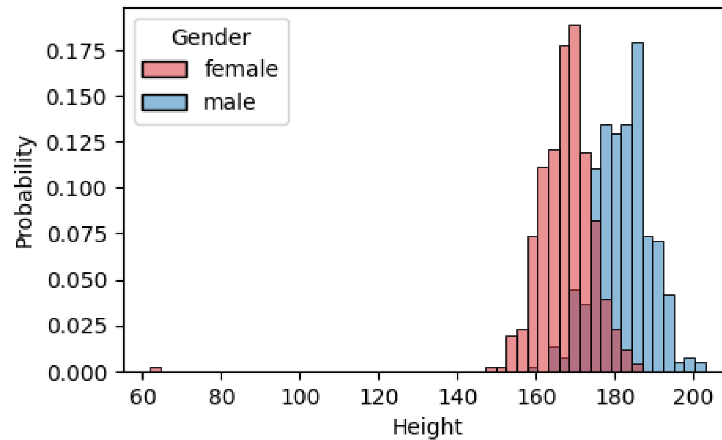
Countries in regions:

```
Region
North America                 3
South Asia                    8
Middle East & North Africa   21
East Asia & Pacific          37
Latin America & Caribbean    42
Sub-Saharan Africa           48
Europe & Central Asia        58
dtype: int64
```



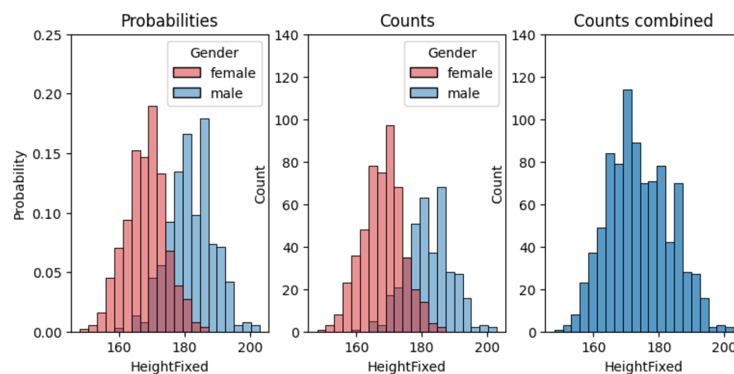
- Using FSEV survey, we can compare self-reported heights of women and men.
- We will use only values of adults above 18 years of age.
- Besides the expected trend, we also see a clear outlier, perhaps an error (although people of such heights exist, the cases are extremely rare).
- We will replace it with NaN.

[46]:
```
fsev.iloc[:,144].value_counts()
adults = fsev.query("Age >= 18").copy(deep=True)
axes = sns.histplot(data=adults, x='Height', hue='Gender',
                    hue_order=['female', 'male'], palette=['C3', 'C0'],
                    stat="probability", common_norm=False)
axes.figure.set_size_inches(5, 3)
pass
```



Below we see histograms after removal of the extreme value.

```
[47]: # replace all values lower than 100cm by NaN, in a new column HeighFixed
adults['HeightFixed']=adults['Height'].mask(adults['Height']<100, np.nan)
figure, axes = plt.subplots(1, 3, figsize=(9,4))
sns.histplot(data = adults, x='HeightFixed', hue='Gender',
             hue_order=['female', 'male'], palette=['C3', 'C0'],
             stat="probability", common_norm=False, ax=axes[0])
sns.histplot(data = adults, x='HeightFixed', hue='Gender',
             hue_order=['female', 'male'], palette=['C3', 'C0'],
             ax=axes[1])
sns.histplot(data = adults, x='HeightFixed', ax=axes[2])
axes[0].set_ylim(0,0.25)
axes[1].set_ylim(0,140)
axes[2].set_ylim(0,140)
axes[0].set_title('Probabilities')
axes[1].set_title('Counts')
axes[2].set_title('Counts combined')
pass
```



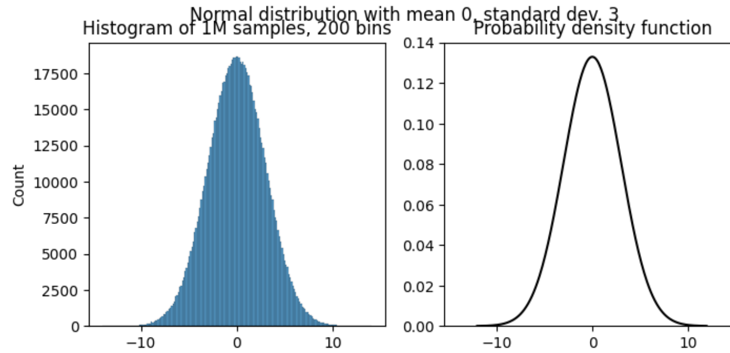
1.3 Probability distributions

- For a continuous variable, we can imagine having infinitely many data points and making histogram with infinitely small bins, keeping the area under the histogram equal to one.
- Thus we obtain **probability density function** (PDF) (hustota rozdelenia pravdepodobnosti).
- We often assume that our data are a small sample from one of the well-characterized probability distributions (rozdelenie pravdepodobnosti).

1.3.1 Normal (Gaussian) distribution

- The normal (or Gaussian) distribution has two parameters: mean μ and standard deviation σ.
- Its density is $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$.
- This is the well-known bell shape.
- Below we plot both histogram of a million samples from this distribution and the density given by the function above.
- These two plots are very similar.

[48]:
```
figure, axes = plt.subplots(1, 2, sharex=True, figsize=(8, 3.5))

# sample million points from the normal distrib. with mean 0 and std. dev. 3
sample_normal = np.random.normal(0, 3, 1000000)
# create histogram of the sampled points
sns.histplot(x=sample_normal, bins=200, ax=axes[0])
axes[0].set_title('Histogram of 1M samples, 200 bins')

# create an object representing normal distrib. with mean 0 and std. dev. 3
normal = scipy.stats.norm(0, 3)
# create equally-spaced points
x = np.arange(-12, 12, 0.1)
# compute values of pdf in these points
y = normal.pdf(x)
# plot the function
axes[1].plot(x, y, 'k-')
axes[1].set_title('Probability density function')
axes[1].set_ylim(0, 0.14)

figure.suptitle("Normal distribution with mean 0, standard dev. 3")
pass
```



- Normal distribution often arises in situations where a variable is a result of many small influences.
- One example is the height of a person within one gender and population.
- Below we fit the normal distribution to the histogram of the adult male heights from the FSEV survey.

[49]:
```
# select male height, drop missing values
male_heights = adults.query("Gender=='male'")['Height'].dropna()
# compute the characteristics (means, stdev)
display(Markdown("**Mean male height:**"),
        male_heights.mean(),
        Markdown("**Std. dev. male height:**"),
        male_heights.std())

# compute the best fit
parameters = scipy.stats.norm.fit(male_heights)
display(Markdown("**Best fit:**"), parameters)

# get function values for regularly distributed x values
x = np.arange(150, 200, 1)
pdf_fitted = scipy.stats.norm.pdf(x, loc=parameters[0], scale=parameters[1])

# plot histogram, normalized as density (area=1)
figure, axes = plt.subplots(figsize=(5,3))
sns.histplot(x=male_heights, stat='density', ax=axes)
# add a line for fitted density
axes.plot(x, pdf_fitted, 'k-')
axes.set_title('Male heights with normal distribution fit')
pass
```

Mean male height:

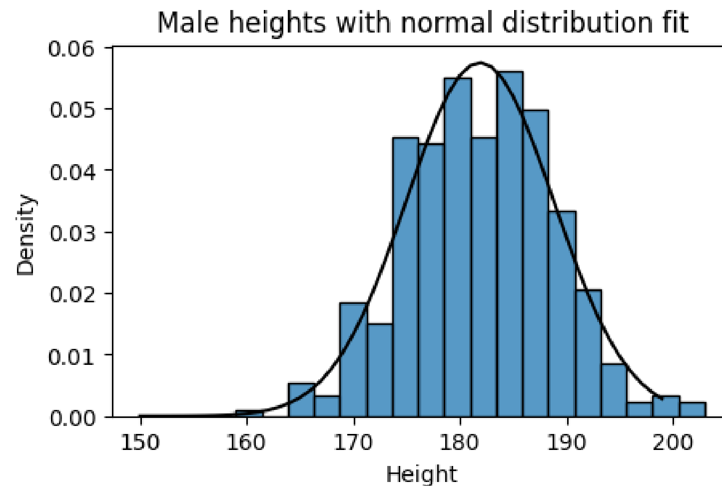
181.91820580474933

Std. dev. male height:

```
6.957251247475206
```

Best fit:

```
(181.91820580474933, 6.948066753375318)
```



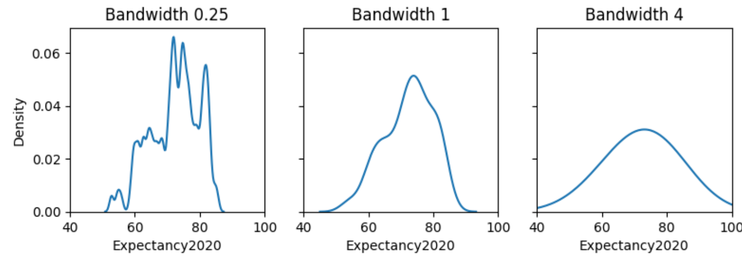
1.4 Kernel Density Estimation (KDE)

- KDE creates a smoothed version of a histogram.
- We choose a kernel function. e.g. the normal distribution.
- For each point in the dataset, the method creates a "kernel" centered at that point.
- It then adds up the heights of all kernel copies.
- The amount of smoothing is controled by the bandwidth parameter (standard deviation for the normal distribution).
- More information is in the scikit-learn documentation.

https://commons.wikimedia.org/wiki/File:Comparison_of_1D_histogram_and_KDE.png Drleft at English Wikipedia, CC BY-SA 3.0

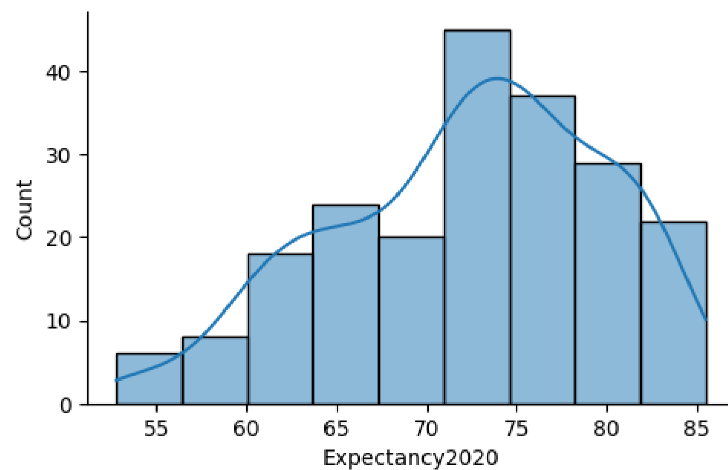
- KDE can be conveniently computed directly in Seaborn's displot/kdeplot function.
- The bandwidth is adjusted by `bw_adjust`, with default 1.
- A small bandwidth leads to a bumpy plot not representing real trends.
- A large badwidth can obscure real trends.

[50]:
```
figure, axes = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(9,2.5))
for axes, bandwidth in [(axes[0], 0.25), (axes[1], 1), (axes[2], 4)]:
  sns.kdeplot(x=countries["Expectancy2020"], ax=axes, bw_adjust=bandwidth)
  axes.set_title(f'Bandwidth {bandwidth}')
  axes.set_xlim(40,100)
pass
```



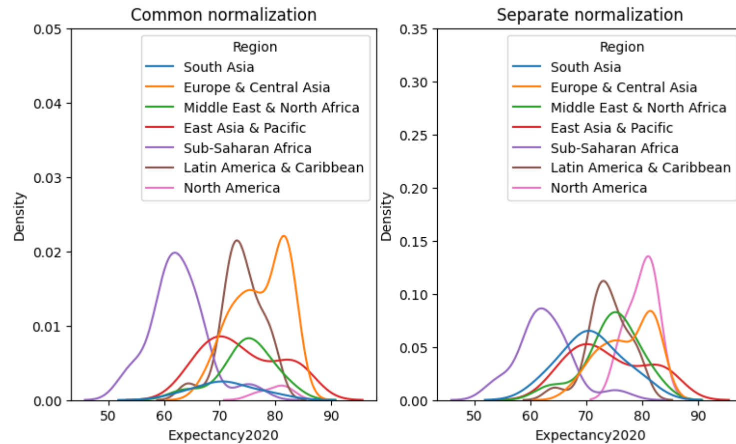
We can display combined histogram and KDE.

[51]:
```
axes = sns.displot(countries, x="Expectancy2020", kde=True)
axes.figure.set_size_inches(5, 3)
pass
```



KDE plots can be also better for comparing multiple distributions, as their smooth curves are easier to follow than histograms.

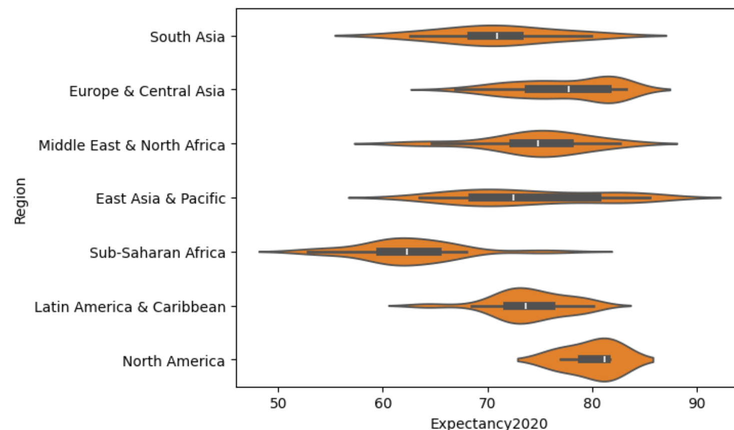
[52]:
```
figure, axes = plt.subplots(1, 2, sharex=True, figsize=(9,5))
sns.kdeplot(x=countries["Expectancy2020"], hue=countries["Region"], ax=axes[0])
axes[0].set_ylim(0, 0.05)
axes[0].set_title('Common normalization')
sns.kdeplot(x=countries["Expectancy2020"], hue=countries["Region"],
            common_norm=False, ax=axes[1])
axes[1].set_title('Separate normalization')
axes[1].set_ylim(0, 0.35)
pass
```



1.5 Violin plots

- Violin plots are often used instead of boxplots to compare distributions for different values of a categorical variable.
- Each violin consist of two symmetric KDE plots.
- They can be accompanied by a boxplot or strip plot.
- More variants can be found in the Seaborn tutorial.

[53]:
```
sns.violinplot(data=countries, y="Region", x="Expectancy2020", color="C1")
pass
```



1.6 Cumulative distribution function

For a probability density function $f(x)$:

- Its cumulative distribution function (CDF) (distribučná funkcia) is the area under the curve from left up to point x.

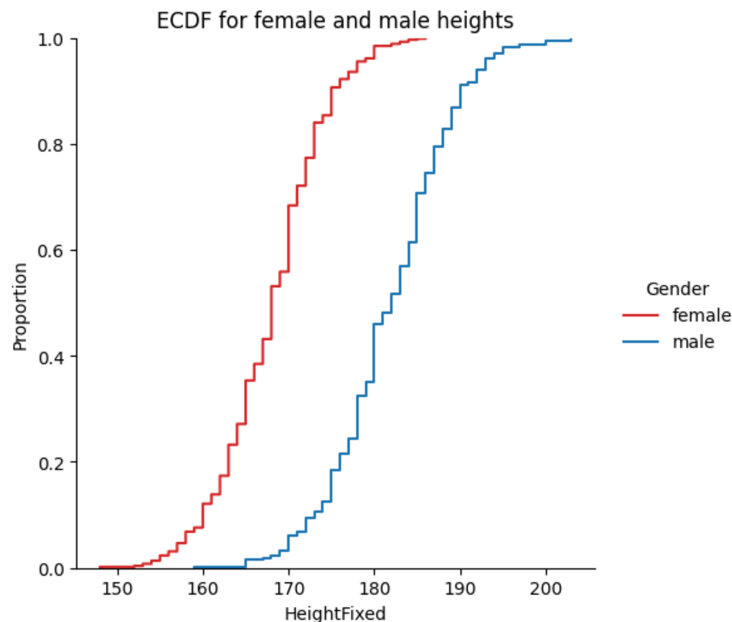
- $F(x) = \int_{-\infty}^{x} f(t)\,dt$.
- CDF is non-decreasing.
- $\lim_{x\to-\infty} F(x) = 0$ and $\lim_{x\to\infty} F(x) = 1$.
- $F(x)$ is the probability that the random point from the distribution is $\leq x$.

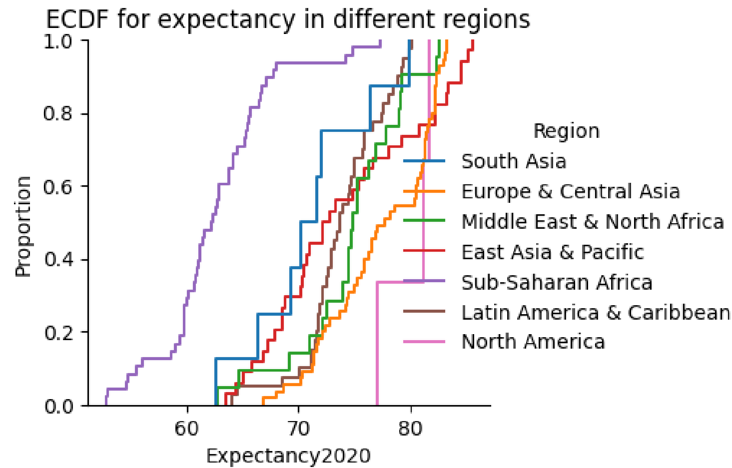
https://commons.wikimedia.org/wiki/File:Normal_Distribution_CDF.svg Inductiveload, Public domain

1.6.1 Empirical cumulative distribution function (ECDF)

- This is a similar concept for a finite sample.
- For each x, $F(x)$ is the fraction of the sample which is $\leq x$.
- This gives us a step-wise function which can be visualized.
- Unlike histograms and KDE, no parameters need to be set (bins, bandwidth).
- Allows comparison of multiple distributions and their quantiles (how?).
- But harder to interpret than histogram in terms of shape.

[54]:
```
grid = sns.displot(adults, x="HeightFixed", hue="Gender", kind="ecdf",
                   hue_order=['female', 'male'], palette=['C3', 'C0'])
grid.axes[0,0].set_title('ECDF for female and male heights')
grid = sns.displot(countries, x="Expectancy2020", hue="Region", kind="ecdf")
grid.axes[0,0].set_title('ECDF for expectancy in different regions')
grid.figure.set_size_inches(5, 3)
pass
```

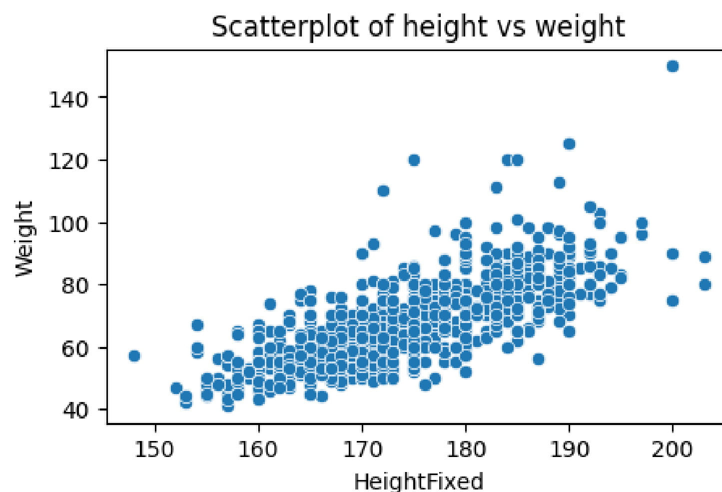




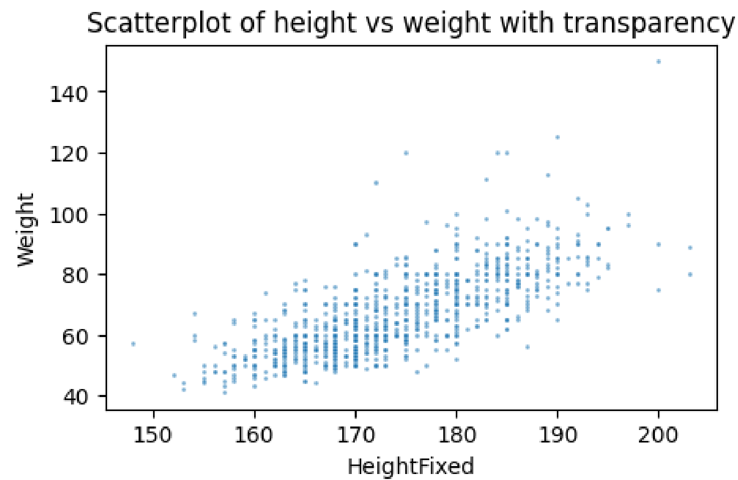
1.7 Two-dimensional histograms / KDE

- Instead of scatterplots, we can compute 2D histograms or smooth them by KDE.
- This works well if we have many overlapping points.
- Below we show several variants for the plot height vs. weight of adults in the FSEV survey (with the height outlier removed).
- In scatterplot there is a cloud of overlapping points and it is not clear which parts of it are denser. This can be somewhat improved with smaller points and transparency.

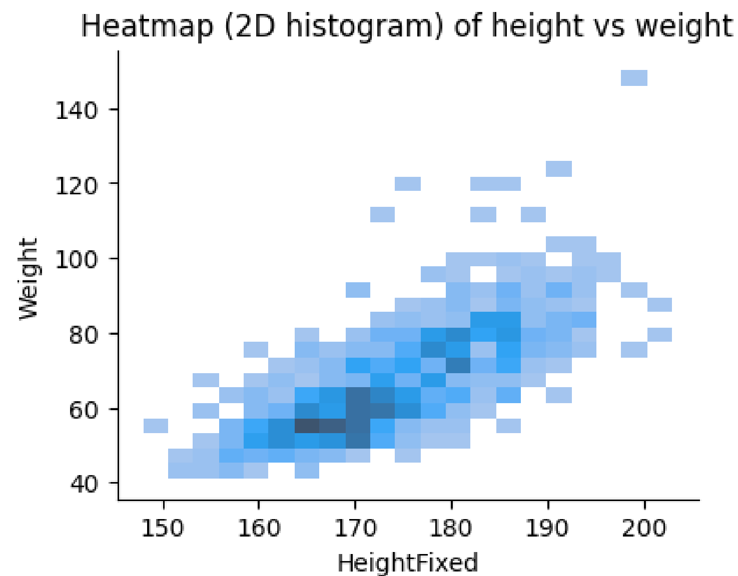
[55]:
```
axes = sns.scatterplot(data=adults, x='HeightFixed', y='Weight')
axes.figure.set_size_inches(5, 3)
axes.set_title('Scatterplot of height vs weight')
pass
```



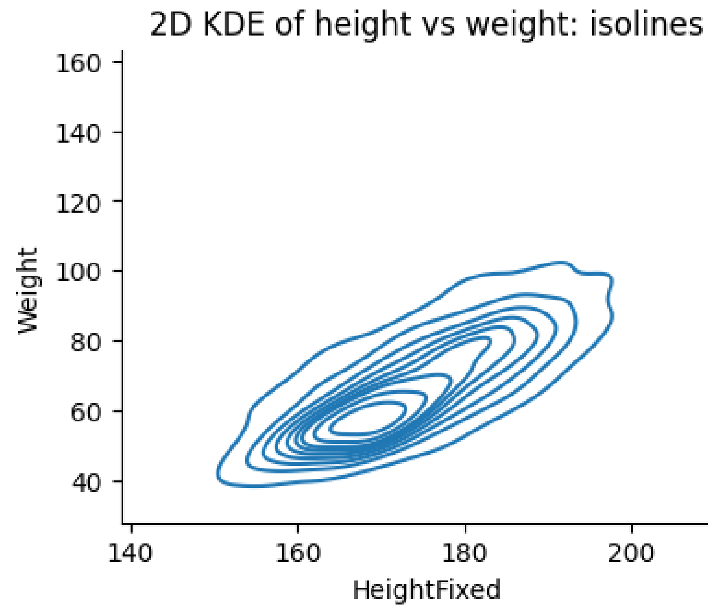
[56]:
```python
# plot with smaller points (s=4) and transparency (alpha=0.6)
axes = sns.scatterplot(data=adults, x='HeightFixed', y='Weight', s=4, alpha=0.6)
axes.figure.set_size_inches(5, 3)
axes.set_title('Scatterplot of height vs weight with transparency')
pass
```



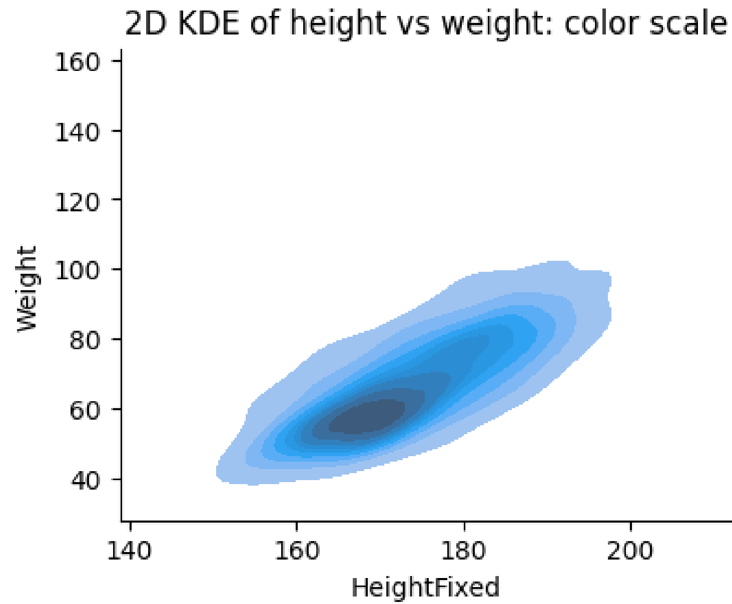
[57]:
```python
grid = sns.displot(data=adults, x='HeightFixed', y='Weight')
grid.figure.set_size_inches(4, 3)
grid.ax.set_title('Heatmap (2D histogram) of height vs weight')
pass
```



[58]:
```
grid = sns.displot(data=adults, x='HeightFixed', y='Weight', kind="kde")
grid.figure.set_size_inches(4, 3)
grid.ax.set_title('2D KDE of height vs weight: isolines')
pass
```



[59]:
```
grid = sns.displot(data=adults, x='HeightFixed', y='Weight', kind="kde",␣
  ↪fill=True)
grid.figure.set_size_inches(4, 3)
grid.ax.set_title('2D KDE of height vs weight: color scale')
pass
```



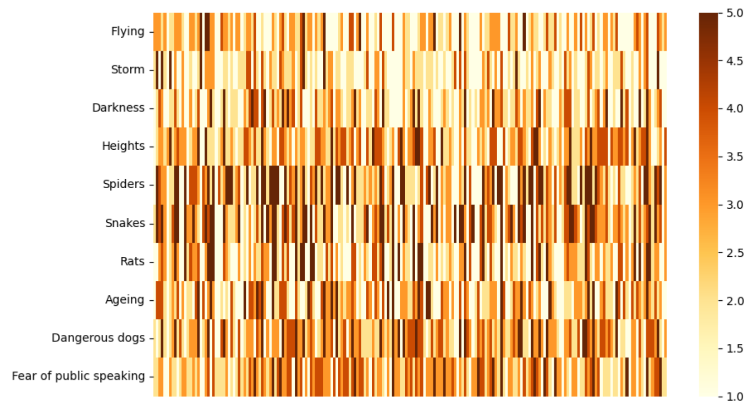
1.8 Clustering multi-dimensional data

- The FSEV survey contains questions about phobias and fears, each with answers 1-5 (5 means highest fear).
- We will first randomly select 200 participants without missing values.
- We will display them as a heatmap.

[60]:
```
# columns 63-72 are fears, drop rows with missing values, sample 200 people
fsev_sample = fsev.iloc[:, 63:73].dropna().sample(200)
# show sample of the data
display(fsev_sample.head())
```

```
     Flying  Storm  Darkness  Heights  Spiders  Snakes  Rats  Ageing  \
995     3.0    2.0       1.0      1.0      1.0       2   1.0     1.0
27      3.0    5.0       2.0      2.0      5.0       5   1.0     4.0
978     3.0    1.0       4.0      3.0      4.0       4   3.0     4.0
568     2.0    5.0       3.0      3.0      3.0       5   4.0     4.0
206     3.0    2.0       1.0      2.0      3.0       3   3.0     2.0

     Dangerous dogs  Fear of public speaking
995             2.0                      2.0
27              1.0                      2.0
978             2.0                      3.0
568             5.0                      3.0
206             2.0                      1.0
```

[61]:
```
figure, axes = plt.subplots(figsize=(10,6))
sns.heatmap(fsev_sample.transpose(), xticklabels=False, ax=axes, cmap="YlOrBr")
pass
```



- Heatmap does not show clear trends, but we see that some phobias have higher values than others.
- We display this more explicitly by sorted means.
- Then we "standardize" values for individual phobias by subtracting the mean and dividing by the standard deviation.
- After this change, each phobia has mean 0 and standard deviation 1.
- People with positive scores fear that subject more than average, with negative scores less than average.

[62]:
```
display(Markdown("**Phobias sorted by mean score in the survey:**"))
display(fsev_sample.mean().sort_values())
```

Phobias sorted by mean score in the survey:

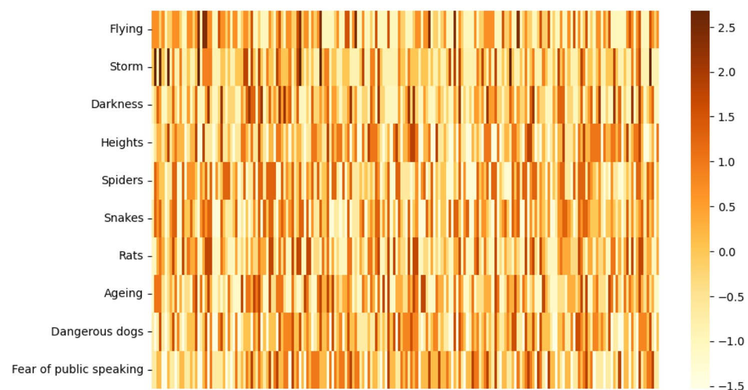
```
Storm                      1.970
Flying                     2.180
Darkness                   2.310
Rats                       2.475
Ageing                     2.575
Heights                    2.740
Fear of public speaking    2.815
Dangerous dogs             2.920
Spiders                    2.975
Snakes                     3.015
dtype: float64
```

[63]:
```
fsev_sample_standardized = (fsev_sample - fsev_sample.mean()) / fsev_sample.
 ↪std()
display(fsev_sample_standardized.head())
```

```
        Flying     Storm  Darkness   Heights   Spiders    Snakes      Rats  \
995   0.694374  0.026563 -1.019166 -1.411326 -1.281210 -0.675758 -1.022784
27    0.694374  2.682911 -0.241177 -0.600219  1.313646  1.321556 -1.022784
978   0.694374 -0.858886  1.314803  0.210888  0.664932  0.655785  0.364042
568  -0.152423  2.682911  0.536813  0.210888  0.016218  1.321556  1.057455
206   0.694374  0.026563 -1.019166 -0.600219  0.016218 -0.009987  0.364042

        Ageing  Dangerous dogs  Fear of public speaking
995  -1.147529       -0.709371                -0.686886
27    1.038241       -1.480427                -0.686886
978   1.038241       -0.709371                 0.155919
568   1.038241        1.603796                 0.155919
206  -0.418939       -0.709371                -1.529692
```

[64]:

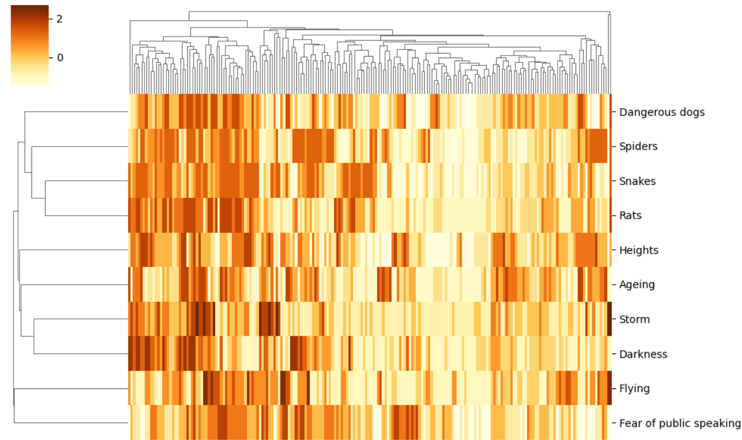
```
figure,axes = plt.subplots(figsize=(10,6))
sns.heatmap(fsev_sample_standardized.transpose(), xticklabels=False,␣
↪cmap="YlOrBr")
pass
```



- Heatmap now does not show much.
- Bellow we reorder its rows and columns by clustering (zhlukovanie), which puts similar rows and similar columns together.
- The degree of similarity is reflected also in the trees (recall last lecture about hierarchies).
- Some areas of dark and light colors now appear.

[65]:

```
sns.clustermap(fsev_sample_standardized.transpose(),
               xticklabels=False, figsize=(10,6), cmap="YlOrBr")
pass
```



1.9 Dimensionality reduction

Methods for dimensionality reduction project high-dimensional data into lower-dimensional (usually 2D) space, while trying to preserve some structure in the original data.

- **Principal component analysis** (PCA) uses a linear projection: each new dimension is a linear combination (weighted sum) of the original dimensions. Weights are chosen to maximize variance.

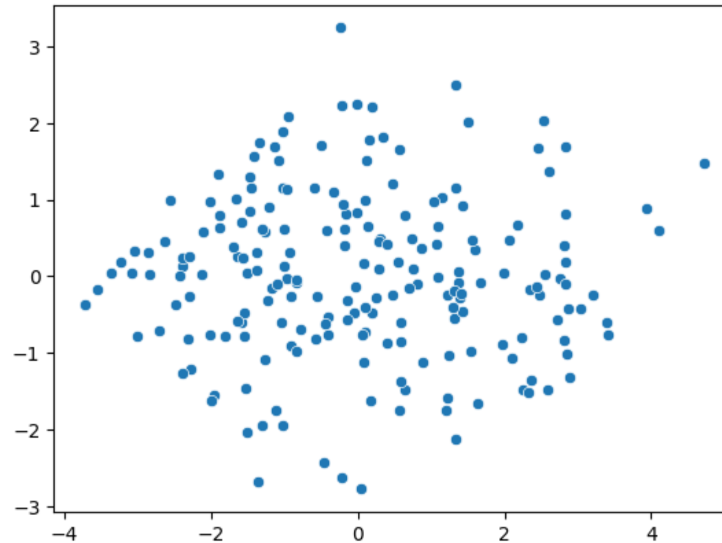
Some methods do not use linear projections, but try to preserve distances between points, for example: * Multidimensional scaling (MDS), * T-distributed Stochastic Neighbor Embedding (t-SNE).

We will use PCA from the scikit-learn library for machine learning in Python.

[66]:
```
from sklearn.decomposition import PCA
# compute PCA of our standardized data with 2 dimensions
fsev_pca = PCA(n_components=2).fit_transform(fsev_sample_standardized)
display(Markdown("**PCA transformed values** (first five lines):"))
display(fsev_pca[0:5, :])
axes = sns.scatterplot(x=fsev_pca[:, 0], y=fsev_pca[:, 1])
```

PCA transformed values (first five lines):

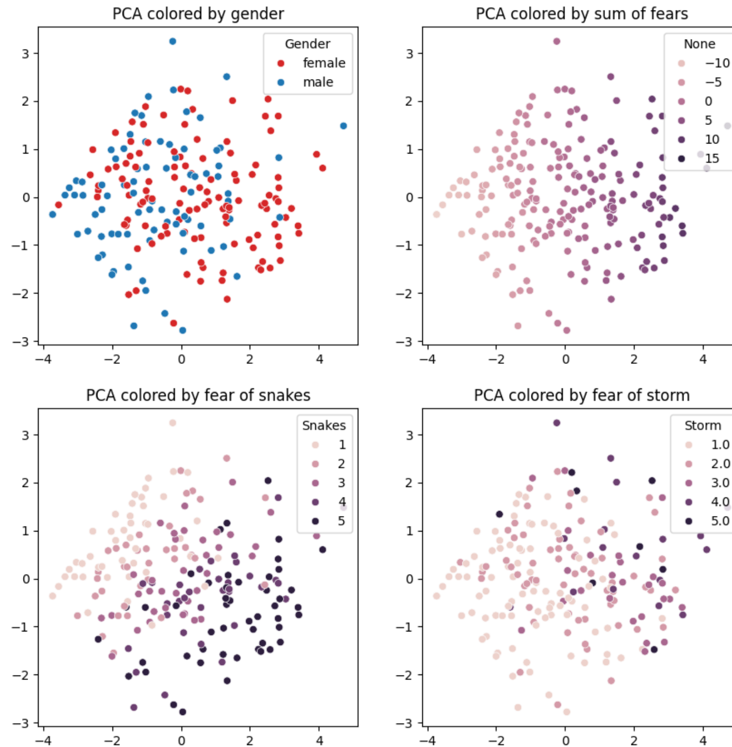
```
array([[-2.40005395,  0.13763973],
       [ 0.79985413, -0.09193566],
       [ 1.09446725, -0.01171503],
       [ 2.83751212,  0.19095389],
       [-0.90658354, -0.90915457]])
```



The scatterplot starts to be interesting if we can color points by some known factors.

- Below we see that men and women are quite mixed but women are shifted to the left.
- It seems that the first dimension strongly correlates with the overall fearfuless of a person.
- Fears of snakes and storms are strongly related to the overall fearfulness, but they also have trends along the y-axis.

[67]:
```
# all columns for our sample to select gender and all data
fsev_sample_all = fsev.loc[fsev_sample.index,]
figure, axes = plt.subplots(2, 2, figsize=(10,10))
sns.scatterplot(x=fsev_pca[:, 0], y=fsev_pca[:, 1],
                hue=fsev_sample_all['Gender'],
                hue_order=['female', 'male'], palette=['C3', 'C0'],
                ax=axes[0,0])
axes[0,0].set_title('PCA colored by gender')
sns.scatterplot(x=fsev_pca[:, 0], y=fsev_pca[:, 1],
                hue=fsev_sample_standardized.sum(axis=1), ax=axes[0,1])
axes[0,1].set_title('PCA colored by sum of fears')
sns.scatterplot(x=fsev_pca[:, 0], y=fsev_pca[:, 1],
                hue=fsev_sample_all['Snakes'], ax=axes[1,0])
axes[1,0].set_title('PCA colored by fear of snakes')
sns.scatterplot(x=fsev_pca[:, 0], y=fsev_pca[:, 1],
                hue=fsev_sample_all['Storm'], ax=axes[1,1])
axes[1,1].set_title('PCA colored by fear of storm')
pass
```



1.10 Conclusion and other courses

We briefly covered several statistical concepts often used in visualization:

- histogram,
- kernel density estimation,
- empirical cumulative distribution function,
- clustering,
- dimensionality reduction.

You will learn more in the next years of your study:

- Fundamentals of Probability and Statistics, 2W (DAV) or 3W (BIN)
- Principles of Data Science 3W (DAV)
- Linear Algebra this semester

Lecture 8
Visual perception and colors

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

Acknowledgement: materials inspired by lectures from Martina Bátorová in 2021

Aside: data analysis / visualization project phases

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis (many tables and graphs for your use)
- Formation of **hypotheses**
- Testing hypotheses (careful reanalysis, new data, other sources)
- **Explanatory** visualizations for the final report / presentation (best views selected for the audience)

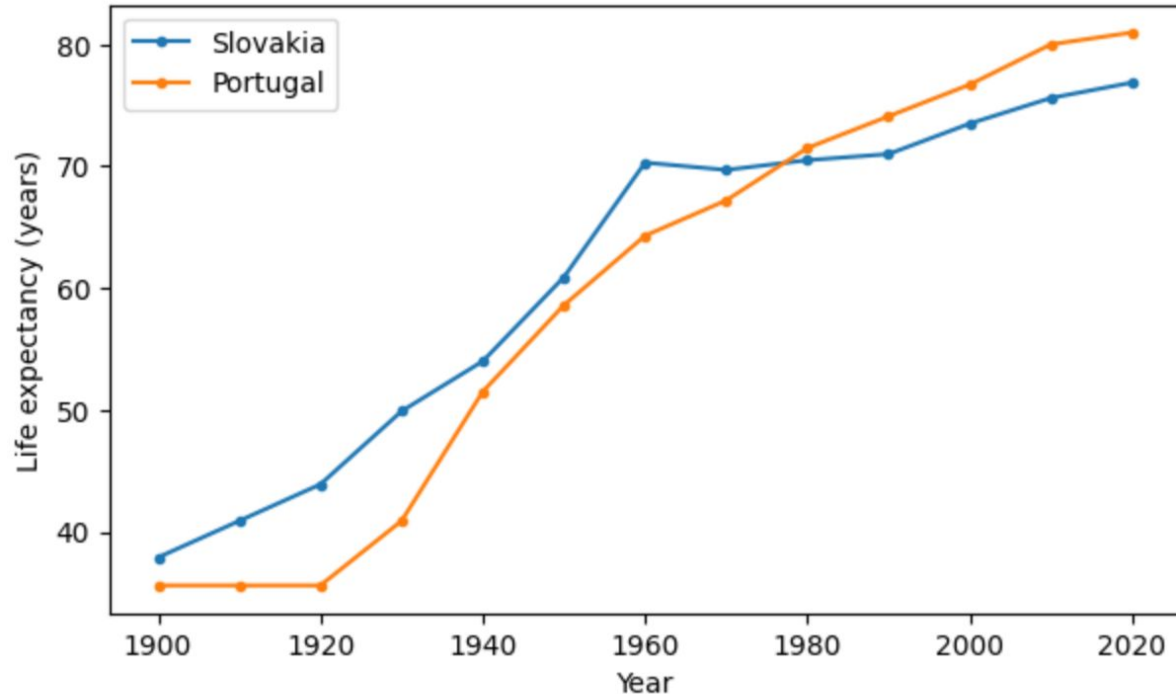
More in two weeks

Why talk about visual perception in visualization?

In which period of time was life expectancy higher in Slovakia than in Portugal?

Country	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010	2020
Slovak Republic	37.9	40.9	43.9	49.9	54.0	60.9	70.3	69.7	70.5	71.0	73.5	75.6	76.9
Portugal	35.6	35.6	35.6	40.9	51.5	58.6	64.3	67.2	71.5	74.1	76.7	80.0	81.0

In which period of time was life expectancy higher in Slovakia than in Portugal?



Visual brain, table vs. graph

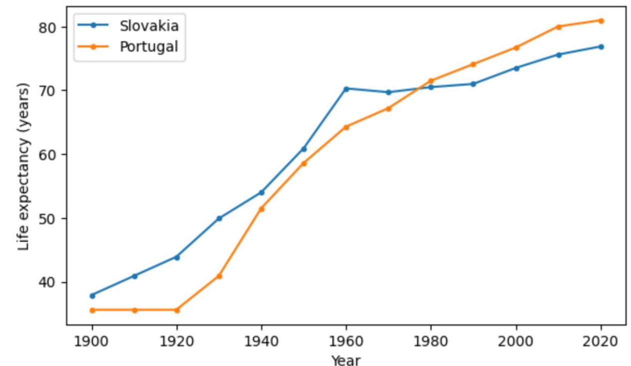
We "read" tables, verbal processing

We "see" plots, visual processing

Visual processing is very parallel and fast, evolved to spot predators

In which situations is a table preferable to a plot?

Country	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010	2020
Slovak Republic	37.9	40.9	43.9	49.9	54.0	60.9	70.3	69.7	70.5	71.0	73.5	75.6	76.9
Portugal	35.6	35.6	35.6	40.9	51.5	58.6	64.3	67.2	71.5	74.1	76.7	80.0	81.0



Human visual perception

What happens when we look at the figure below?

Human visual perception

Human visual perception

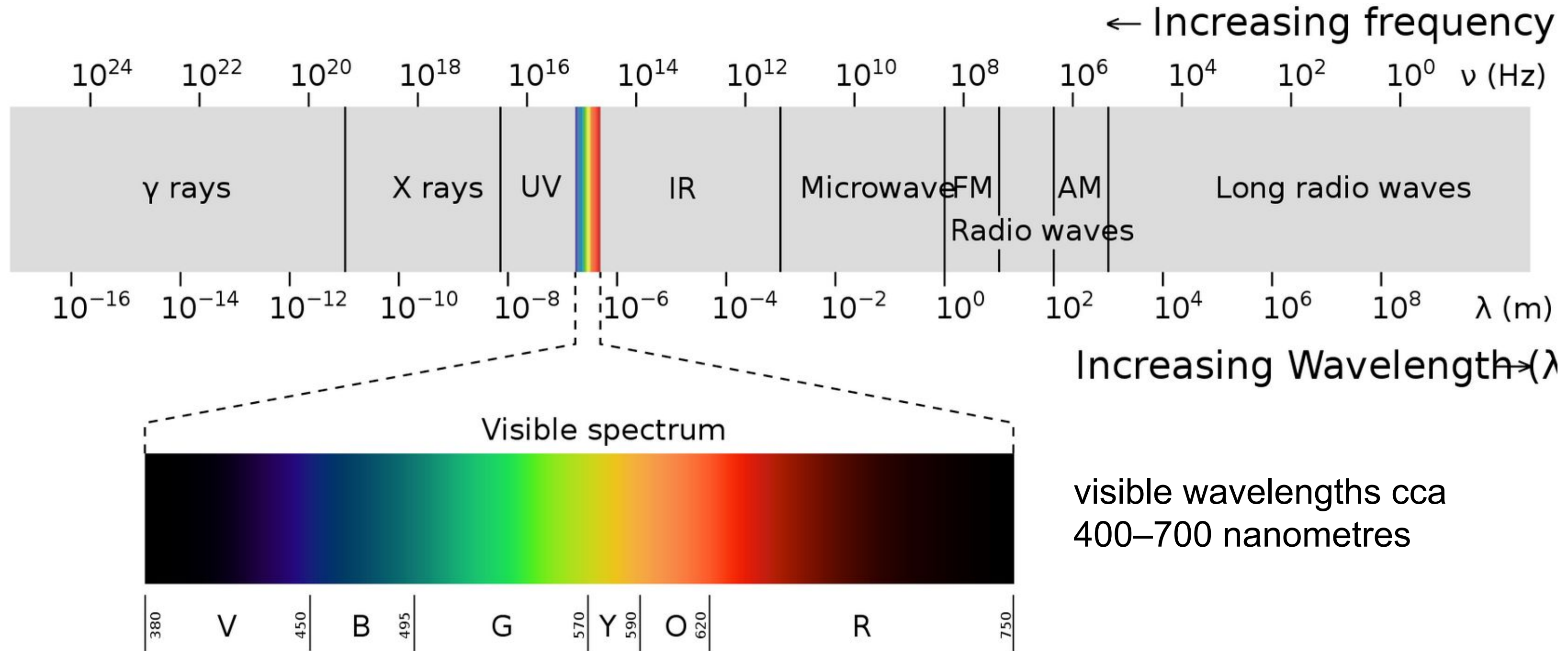
What happens when we look at the figure?

- The **light** from the screen / projector hits the retinas of our **eyes**
- Photoreceptor cells **transduce** (convert) this signal into nerve impulses
- In the brain:
  - detection of **basic features**
  - recognition of **patterns**
  - interpretation, assignment of **meaning**

Today: Focus on the light, eyes and colors, later stages next week

Light

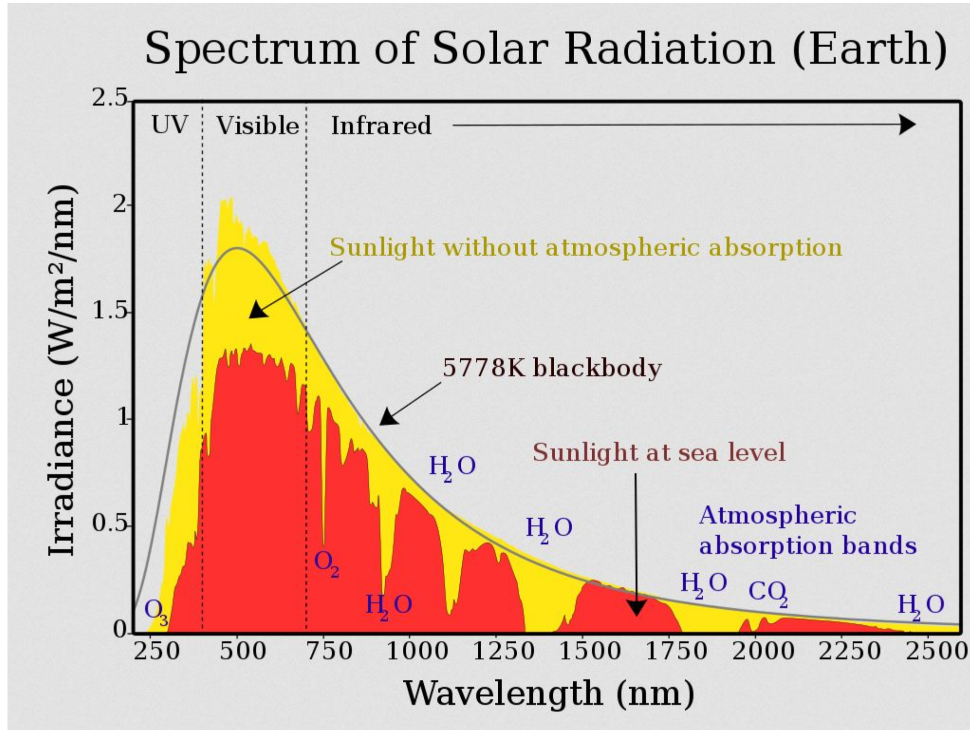
Visible light as a part of electromagnetic spectrum



visible wavelengths cca
400–700 nanometres

https://commons.wikimedia.org/wiki/File:EM_spectrum.svg

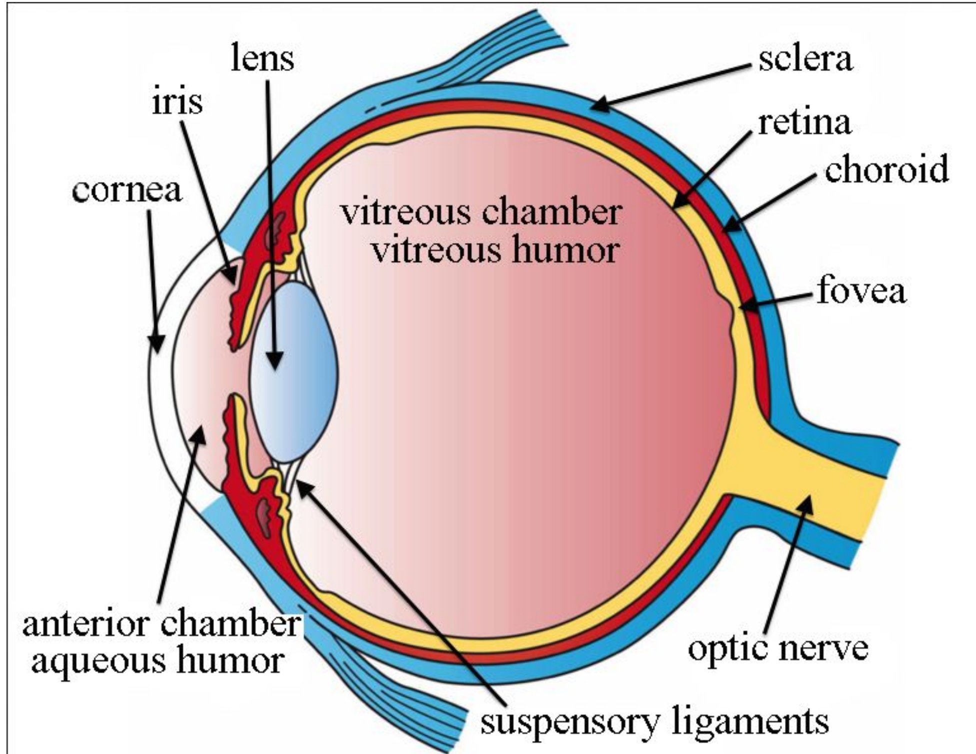
Sunlight is a mixture of different wavelengths



https://commons.wikimedia.org/wiki/File:Solar_spectrum_en.svg
https://commons.wikimedia.org/wiki/File:WhereRainbowRises.jpg

Human eye

Human eye



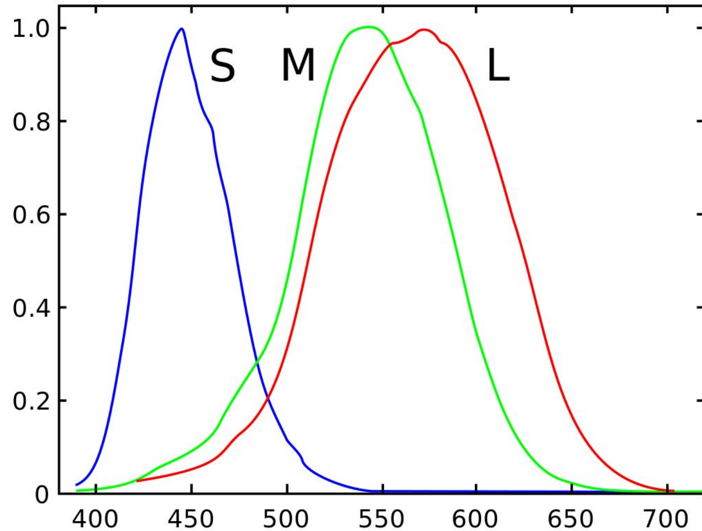
https://commons.wikimedia.org/wiki/File:Three_Main_Layers_of_the_Eye.png

Retina (sietnica): light-sensitive layer

Lens (šošovka): focus light to retina

Pupil (zrenica): hole in iris (dúhovka) where light enters the eye, its size regulated by the amount of light

Photoreceptor cells in the retina

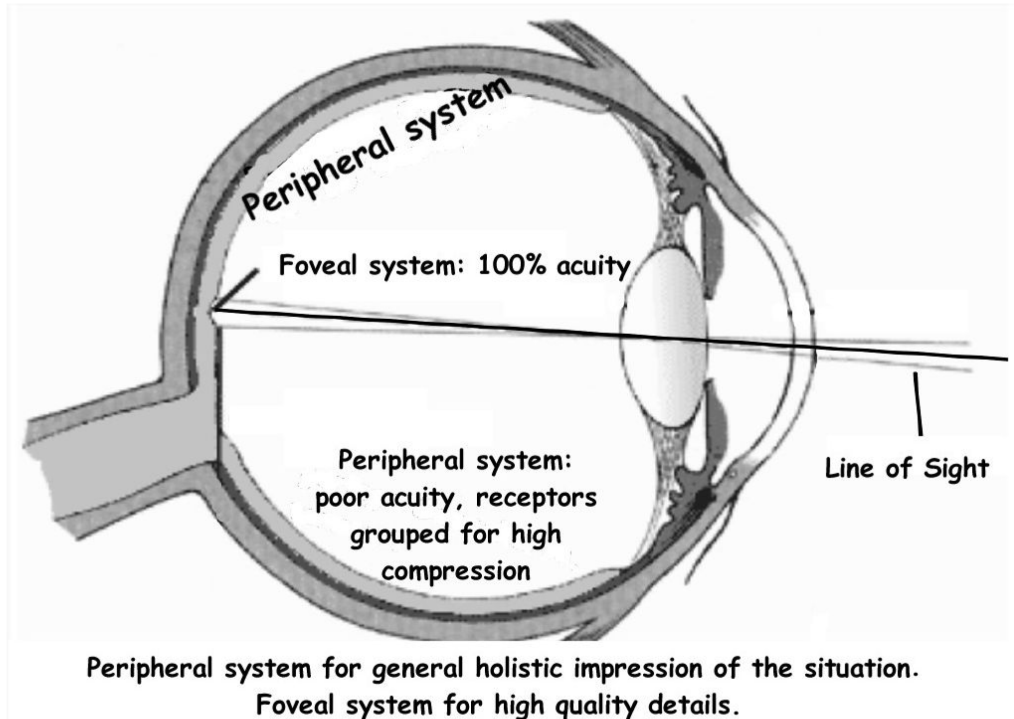


Rods (tyčinky): more sensitive to low light, not used for color vision

Cones (čapíky): color vision, three different types sensitive to different wavelengths (blue, green, red)

https://commons.wikimedia.org/wiki/File:Cones_SMJ2_E.svg

Foveal vs peripheral vision



https://commons.wikimedia.org/wiki/File:Double_system_e.jpg

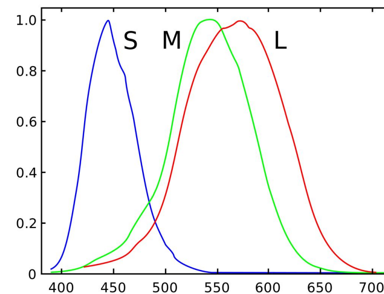
Fovea: central zone with many cones, sharp color vision, only about 1-2°

Peripheral vision: mostly rods, fast monochrome vision

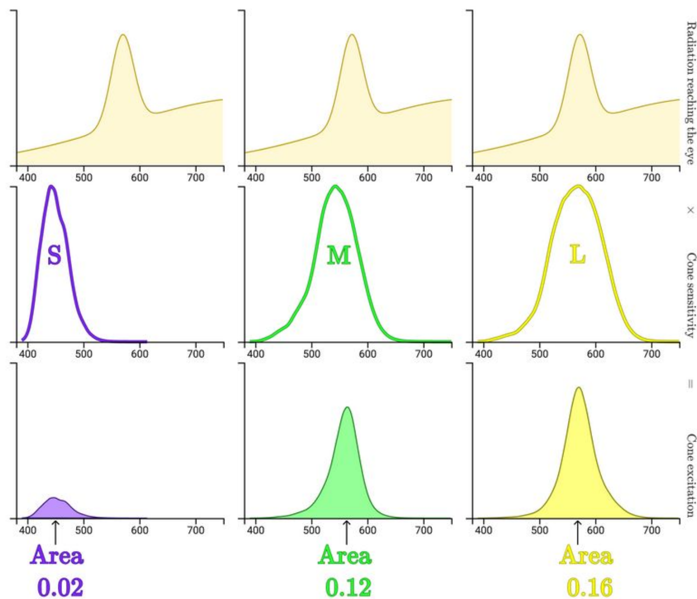
The eyes make fast **movements** (saccades) between fixations on different points of interest to create a composite image

Colors and color spaces

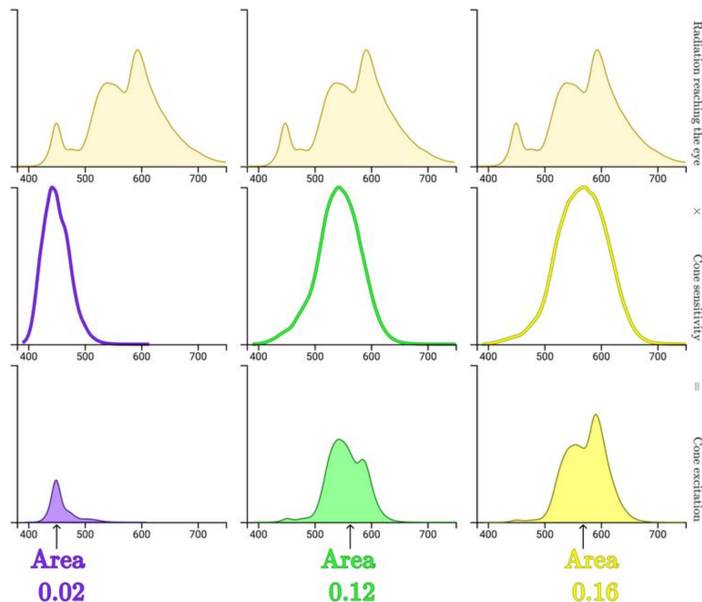
Metamers: light with different spectra that appear the same (to typical humans)



Cone excitation by a point on a lemon

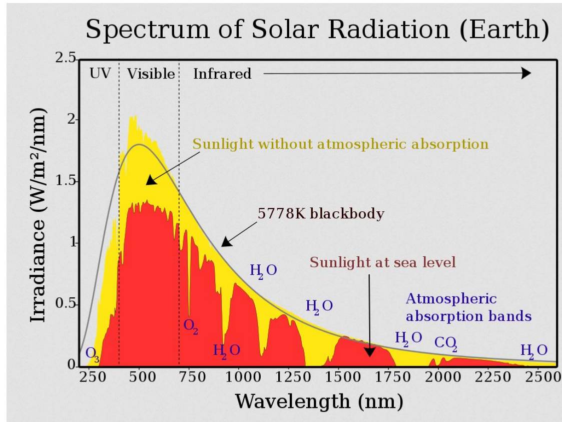


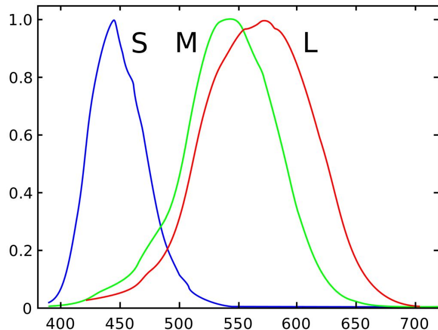
Cone excitation by a pixel of a lemon on a screen



https://jamie-wong.com/post/color/

Color spaces, LMS





A **color space** is an organization of colors.

Our eye projects a full light spectrum into three values: response of the three types of cones.

S (short), M (medium), L (large) wavelength

LMS color space uses these three values to represent a color.

Metameric colors have the same values.

Derived models, e.g. CIE with better properties.

https://commons.wikimedia.org/wiki/File:Solar_spectrum_en.svg
https://commons.wikimedia.org/wiki/File:Cones_SMJ2_E.svg

Do you know some other color spaces?

Additive color models, RGB

Monitors, projectors etc.

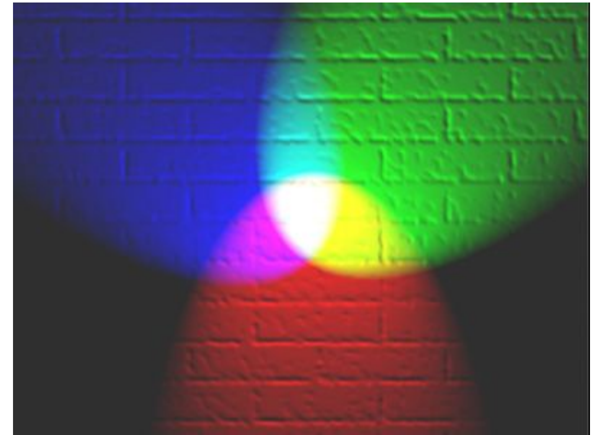
Component lights in **primary colors**,
other colors mixtures of these (adding up light).

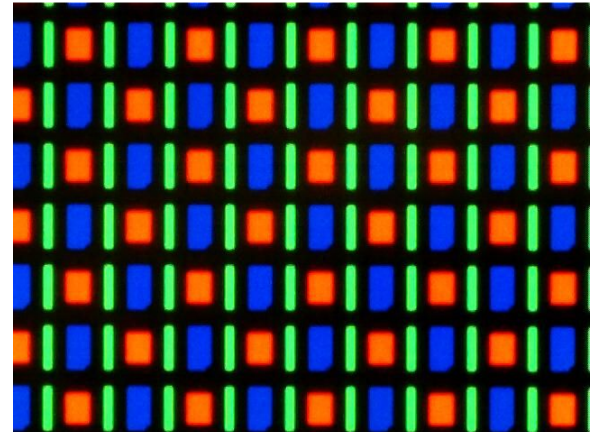
White can be achieved by combining colors.

RGB uses **red, green, blue** as primary
(corresponds to LMS peaks).

The **gamut** is the set of colors representable by a device, usually a subset of the visible spectrum.

https://commons.wikimedia.org/wiki/File:RGB_illumination.jpg
https://commons.wikimedia.org/wiki/File:Nexus_one_screen_microscope.jpg

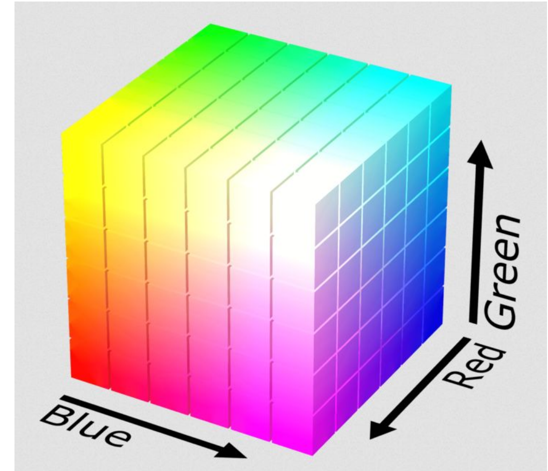


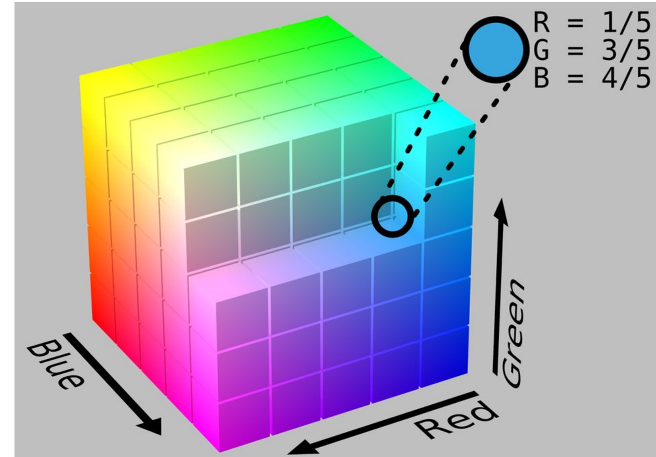


Colors in RGB space (RGB cube)

RGB is often used to specify colors.
Each coordinate e.g. a real number between 0 and 1 or integer between 0 and 255.
Also hexadecimal notation, e.g. #ff0000 is pure red.

Greytones on the main diagonal (x,x,x).





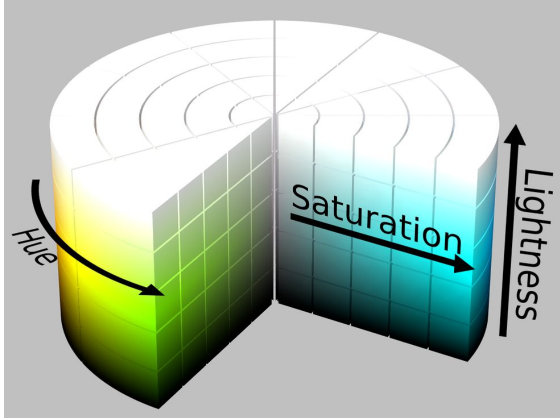
https://commons.wikimedia.org/wiki/File:RGB_color_solid_cube.png
https://commons.wikimedia.org/wiki/File:RGB_Cube_Show_lowgamma_cutout_b.png

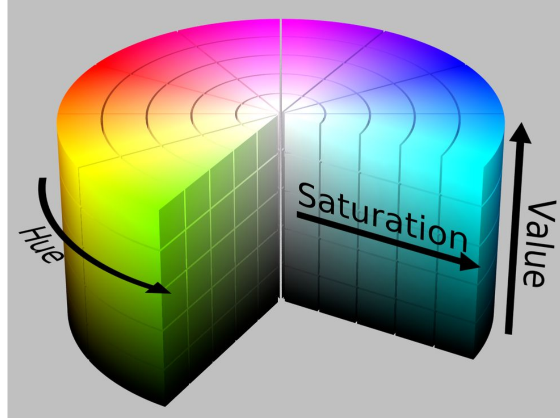
HSL and HSV color models

Transformations of RGB model with more intuitive coordinates.

Useful for color pickers, color palettes, image transformations etc.

Hue, saturation, lightness / hue, saturation, value.





https://commons.wikimedia.org/wiki/File:HSL_color_solid_cylinder_saturation_gray.png
https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder_saturation_gray.png

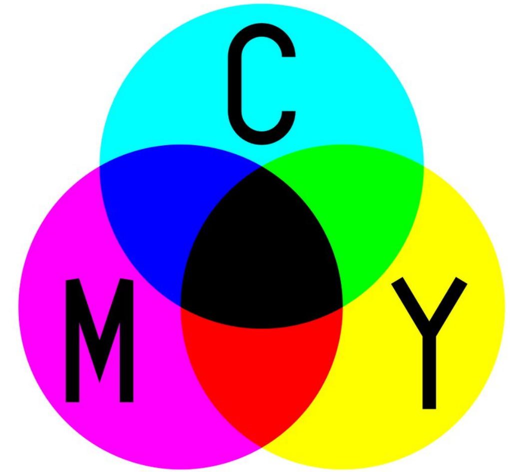
Subtractive models, pigments

Pigments block part of the light spectrum.

Adding more pigments blocks (subtracts) more light.

Black can be achieved by combining colors.

Example: CMY(K) color model used in **printing**.



https://commons.wikimedia.org/wiki/File:SubtractiveColor.svg

CMY(K) color model

Primary colors cyan, magenta and yellow
- **cyan** absorbs red
- **magenta** absorbs green
- **yellow** absorbs blue

Black (K) added because
- it is cheaper
- it hides artifacts in dark colors

Conversion from RGB to CMYK is difficult, device-dependent.

https://commons.wikimedia.org/wiki/File:Barns_grand_tetons.jpg
https://commons.wikimedia.org/wiki/File:CMY_separation_%E2%80%93_no_black.jpg
https://commons.wikimedia.org/wiki/File:CMYK_separation_%E2%80%93_maximum_black.jpg

Color wheel, RYB model

Subtractive model developed in art, for mixing pigments

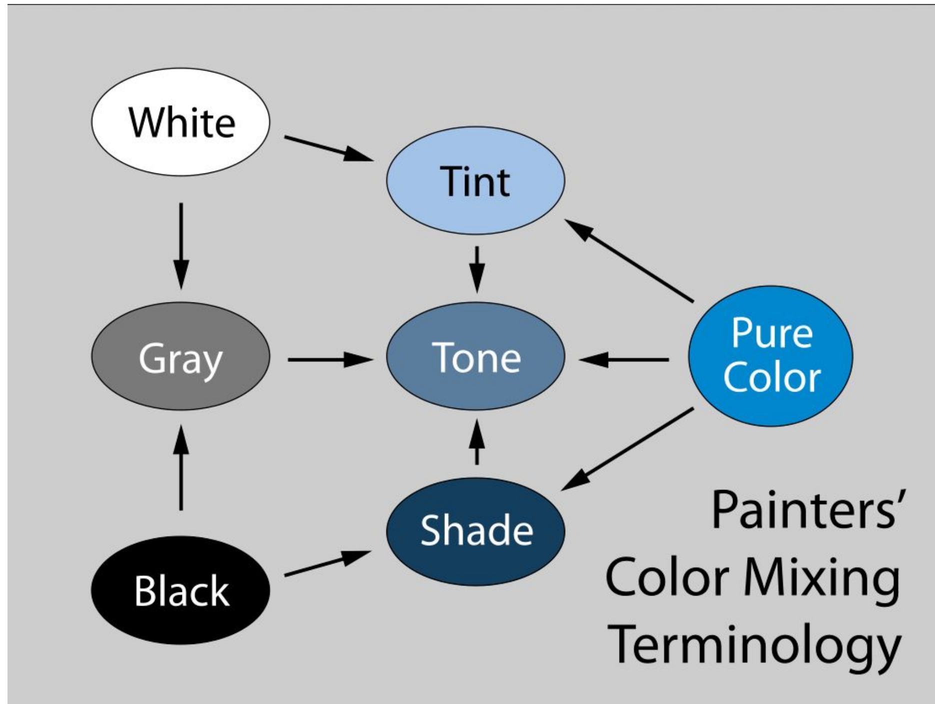
Dates back to 17th century

Primary colors red, yellow, blue

Secondary orange, green, purple (each a mix of two primaries)

https://www.w3schools.com/colors/colors_wheels.asp

Tint, tone, shade - more painter terminology



https://commons.wikimedia.org/wiki/File:Tint-tone-shade.svg

Examples of color schemes

Monochromatic: tints / tones / shades of the same hue

Complementary: 2 colors opposite on the color wheel
(e.g. orange and blue)

Split complementary: color and 2 neighbors
of its complement (e.g. orange and blue-green, blue-purple)

Analogous: 3-5 adjacent colors on the wheel

Each of these can be desaturated (tints / tones / shades)

See also https://color.adobe.com/create/color-wheel

https://www.w3schools.com/colors/colors_wheels.asp

Color and meaning, cultural differences

Colors often symbolize different things both within and between cultures:

- for example red: blood, love, passion, life, anger, violence, danger, emergency, speed, heat ...
  - China: good luck, prosperity vs. Europe: warning
  - "in the red" mean losses in English, what about China?
- mourning color is black in Europe, white in the East

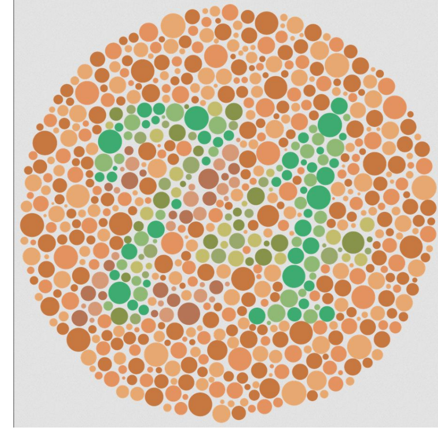
Colors in data visualization

Not everybody will be able to see your colors

Color blindness or color vision deficiency

Various forms and causes

Most often genetic red–green color blindness, where L or M opsin gene is mutated (8% of males)



https://commons.wikimedia.org/wiki/File:Ishihara_9.svg

Technical problems
- Projectors often distort colors
- Black-and-white printing

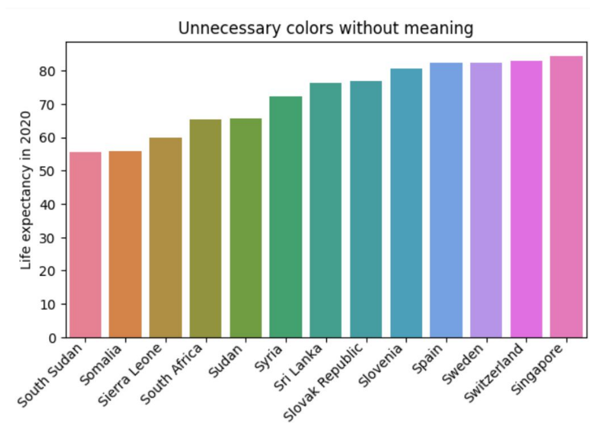
Choose your colors wisely to avoid these problems as much as possible

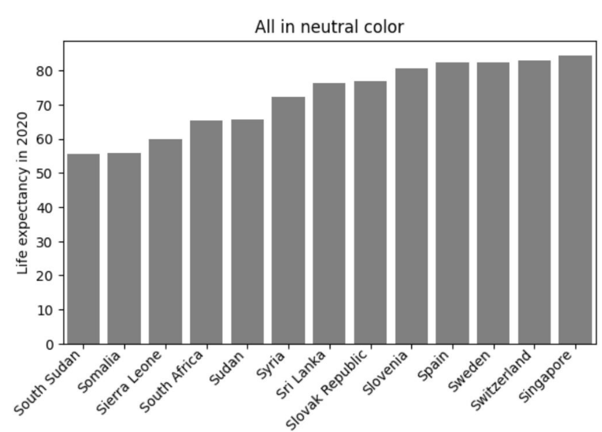
Color draws attention, use it sparingly

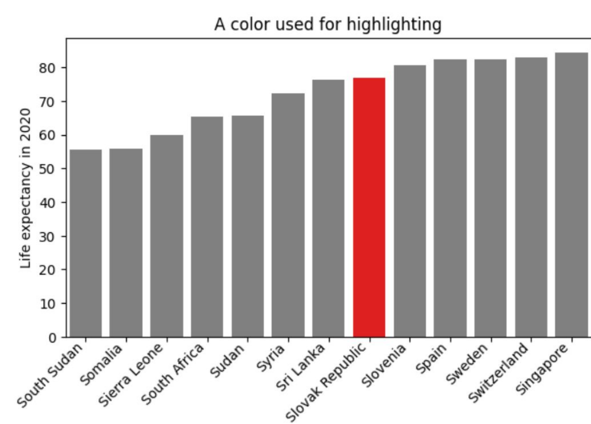
Rely on neutral tones

Use color sparingly, avoid unnecessary colors

Colors are great for highlighting points of interest



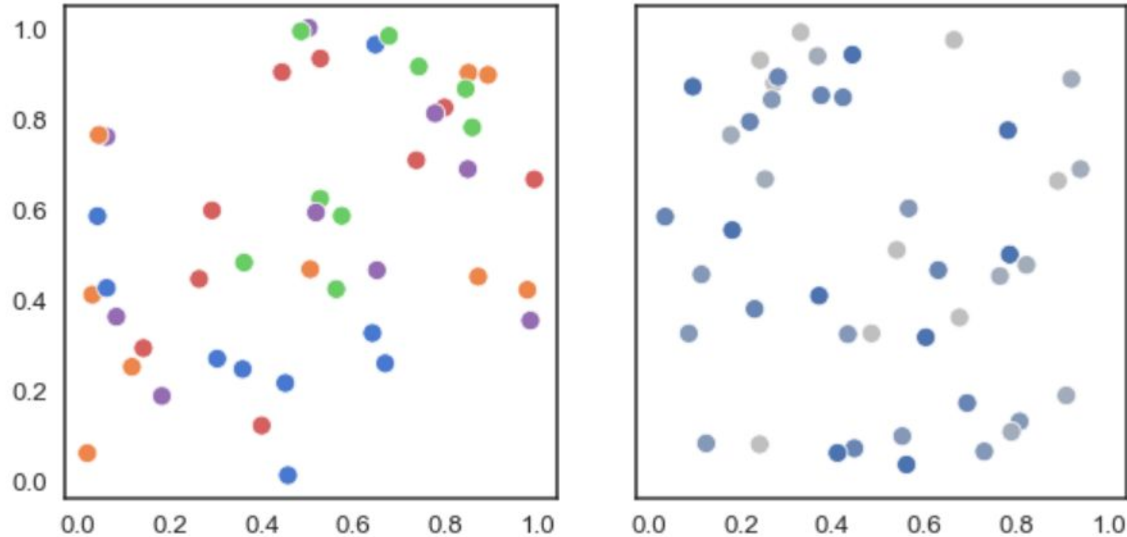




Qualitative palettes (for categorical variables)

Typically vary hues, easier to distinguish than lightness of the same hue

Try to keep the number of colors low



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

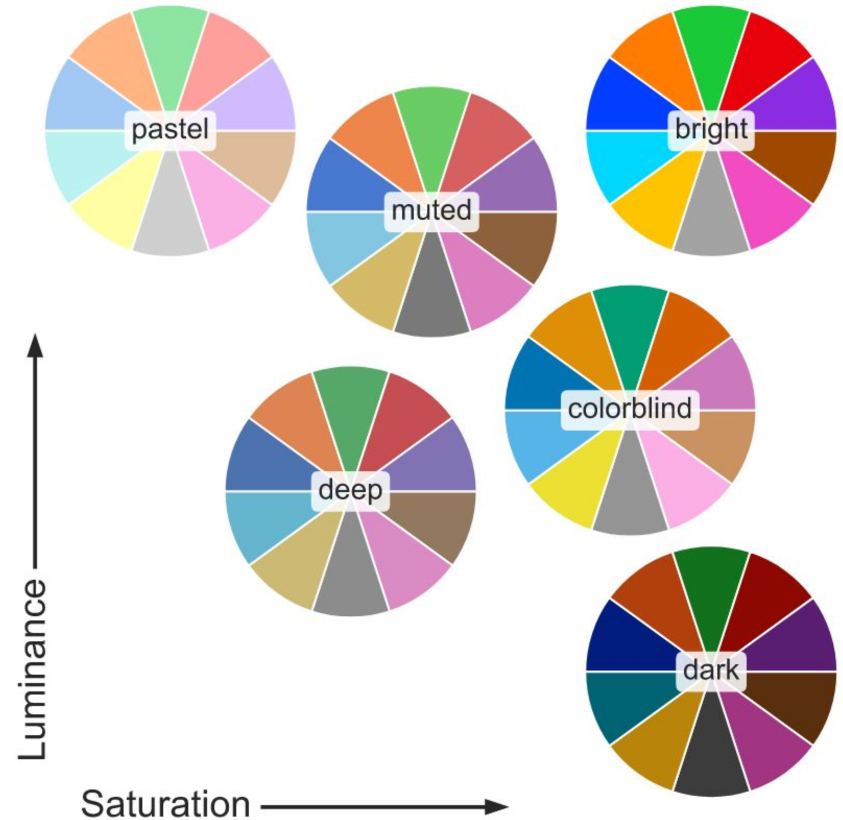
Qualitative palettes (for categorical variables)

Examples from Seaborn library

on the right:

Color Brewer tool:

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



Quantitative sequential palettes (for numerical variables)

Vary lightness rather than (just) hue (hue is circular, hard to interpret)

Discrete vs continuous:

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

Quantitative diverging palettes

For numerical variables with a special midpoint, often 0 (such as increase / decrease)

Midpoint displayed as a special neutral color, e.g. grey

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

Summary

Visual perception:

- light on the retina, transduction on photoreceptor cells
- feature detection, pattern formation, interpretation

Light and color:

- Visible light as a part of electromagnetic spectrum
- Three types of cones sensitive to different wavelengths of light

Color spaces (LMS, RGB, HSL, HSV, CMYK, RYB)

Colors in visualization (qualitative and quantitative palettes, color blindness, sparing use of colors)

Graphics file formats (for exporting graphs)

Raster formats: store your plot as pixels at some resolution

- resolution leads to tradeoff between size and quality
- prefer lossless compression (PNG rather than JPEG)
- transparent background may be also a good idea

Vector formats: store your plot as geometric objects (SVG, PDF, EPS)

- can be arbitrarily enlarged, editable
- can be large if many points / lines (subsample data?)
- vector formats usually preferable unless not supported by software
- it is a good idea to include fonts in the file
- beware that these file formats may include bitmaps

Lecture 9
Pre-attentive attributes, gestalt, illusions

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

Acknowledgement: materials inspired by lectures from Martina Bátorová in 2021

Human visual perception

What happens when we look at the figure?

- The **light** from the screen / projector hits the retinas of our **eyes**
- Photoreceptor cells **transduce** (convert) this signal into nerve impulses
- In the brain:
  - detection of **basic features**
  - recognition of **patterns**
  - interpretation, assignment of **meaning**

Human visual perception

What happens when we look at the figure?

- The **light** from the screen / projector hits the retinas of our **eyes**
- Photoreceptor cells **transduce** (convert) this signal into nerve impulses
- In the brain:
  - detection of **basic features**
  - recognition of **patterns**
  - interpretation, assignment of **meaning**

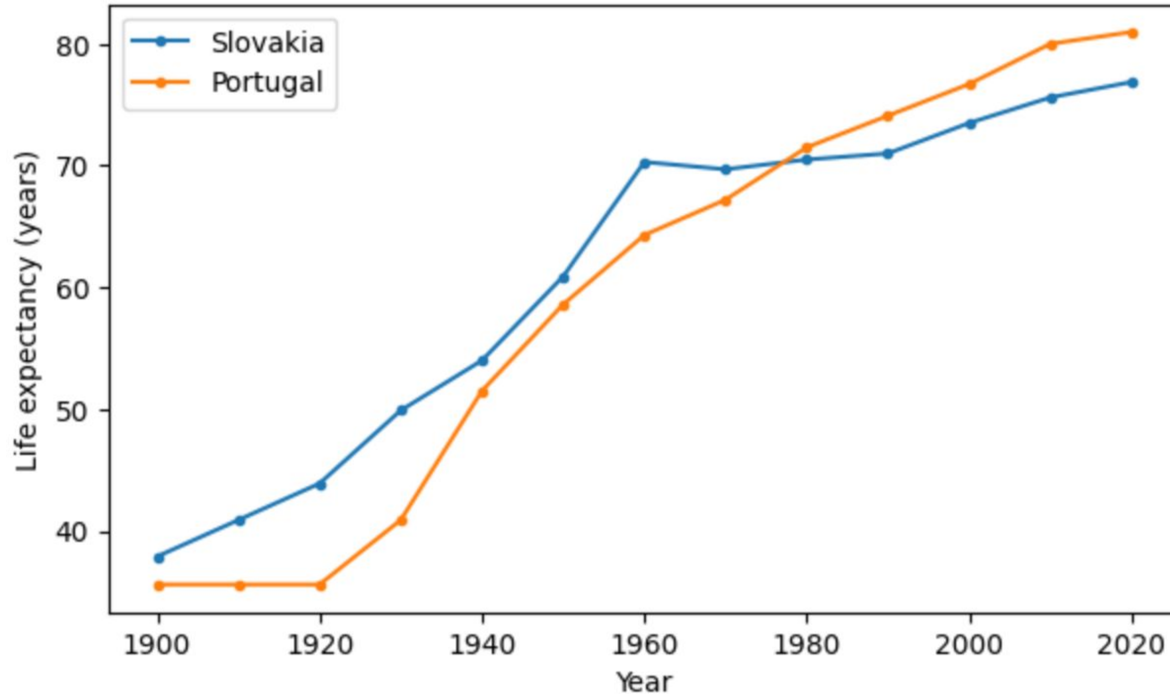
Today: Detection of features and patterns, use for visualization

Note: Human visual perception is very good for detecting **motion** (danger/prey). This is relevant for animated visualization, but not covered today.

In which period of time was life expectancy higher in Slovakia than in Portugal?

Country	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010	2020
Slovak Republic	37.9	40.9	43.9	49.9	54.0	60.9	70.3	69.7	70.5	71.0	73.5	75.6	76.9
Portugal	35.6	35.6	35.6	40.9	51.5	58.6	64.3	67.2	71.5	74.1	76.7	80.0	81.0

In which period of time was life expectancy higher in Slovakia than in Portugal?



How many copies of digit six do you see?

1014508

2530653

6821550

3702967

8622988

What about now?

1014508

2530**6**53

6821550

37029**6**7

8**6**22988

What about Slovakia vs Portugal in this table?

	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010	2020
Slovakia	**37.9**	**40.9**	**43.9**	**49.9**	**54.0**	**60.9**	**70.3**	**69.7**	70.5	71.0	73.5	75.6	76.9
Portugal	35.6	35.6	35.6	40.9	51.5	58.6	64.3	67.2	**71.5**	**74.1**	**76.7**	**80.0**	**81.0**

Pre-attentive attributes

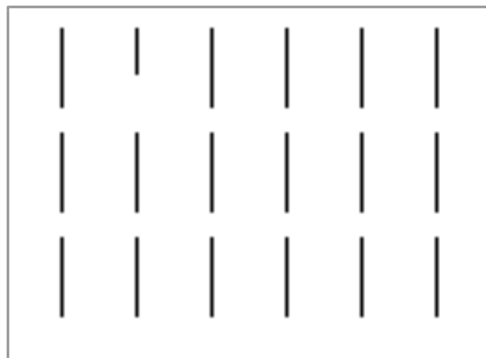
- Features of the seen objects detected by our brain very **fast**
- Prior to and **without** the need of conscious **awareness**
- Brain uses them to **guide attention** / gaze to interesting parts of the scene
- Their correct use allows fast and effortless understanding of our visualizations

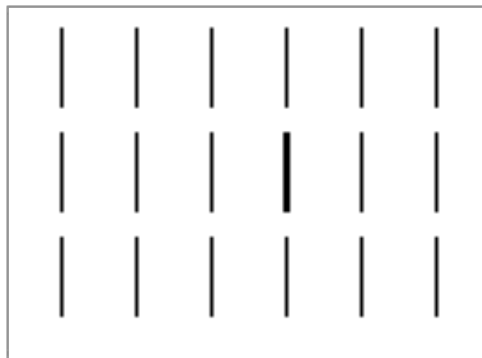
Next:
Examples of important pre-attentive attributes (form, color, position)
following Few 2009

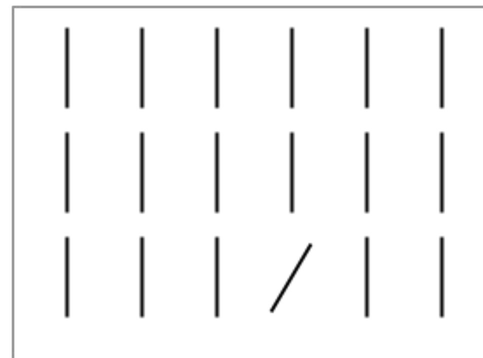
See also https://www.csc2.ncsu.edu/faculty/healey/PP/

Pre-attentive attributes: form

We can quickly distinguish one object that differs from the rest

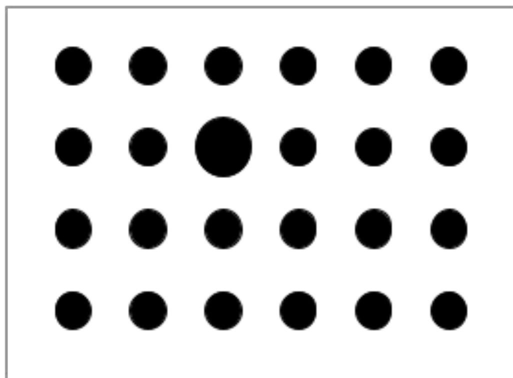




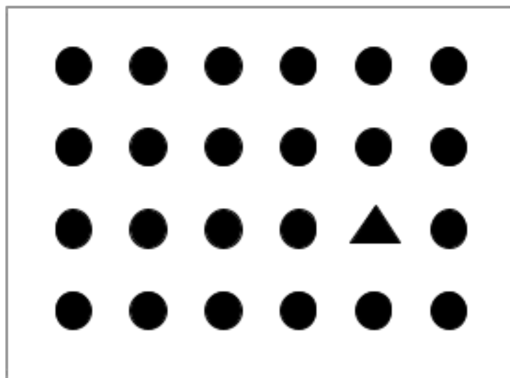


Pre-attentive attributes: form

We can quickly distinguish one object that differs from the rest



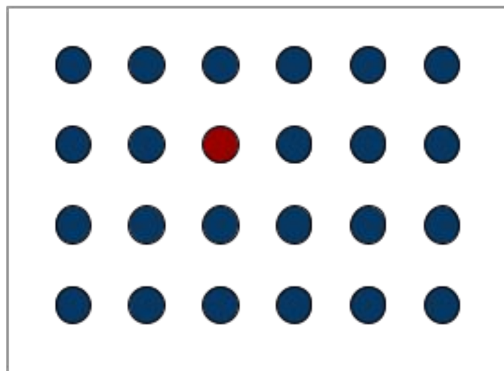
Size



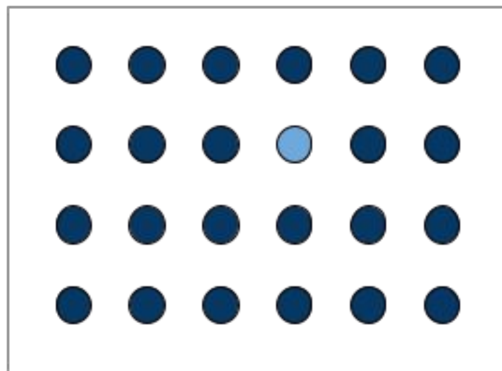
Shape

Pre-attentive attributes: color

We can quickly distinguish one object that differs from the rest

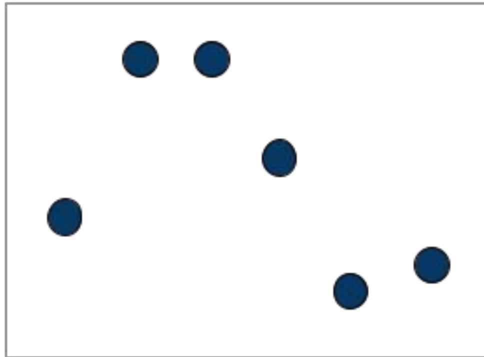


Hue

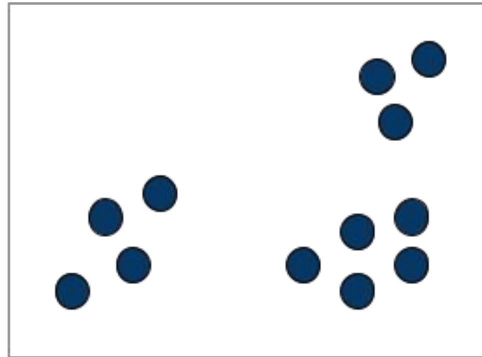


Lightness

Pre-attentive attributes: position



2D



Groups

Hierarchy of graph elements

Cleveland, McGill 1985

Experiments with volunteers of how well they **judge ratios** between values graphically encoded in different ways.

Not all pre-attentive attributes are equally good for **quantitative reasoning**.

Prefer elements on the left side for accuracy

Length (aligned)

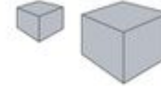
Length

Slope

Angle

Area

Volume



Color intensity

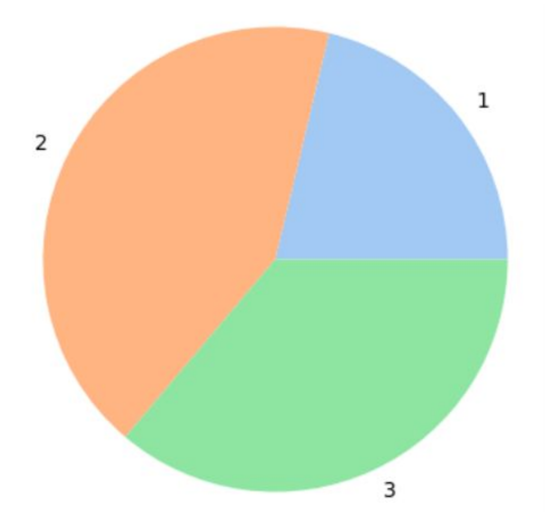
Color hue

Accurate

Generic

Based on https://paldhous.github.io/ucb/2015/dataviz/week2.htm

The same data with length / area / color / angle



10,20,17

Chart selection tools

In lecture 3 and later, we have seen many types of graphs

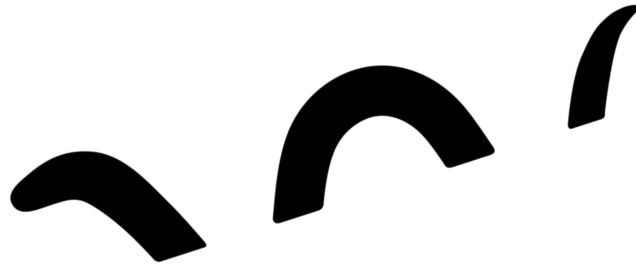
Some websites list them based on variable type and purpose for easier selection:

- https://www.data-to-viz.com/
- https://extremepresentation.typepad.com/blog/2006/09/choosing_a_good.html

Let us look at some the suggestions from the first website in terms of the hierarchy of graph elements

From parts to the whole: gestalt

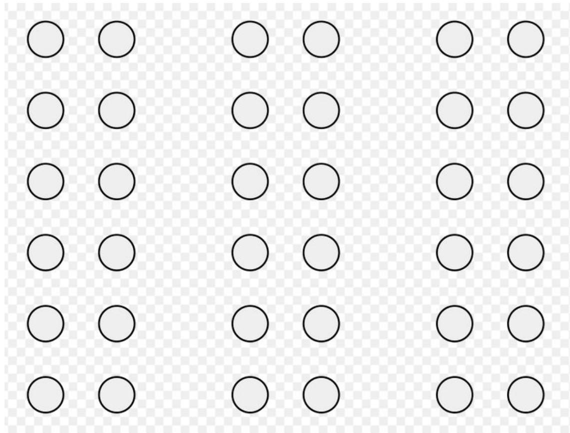
- Gestalt psychology (early 20th century, Austria and Germany)
- **Gestalt** means **pattern**
- Our brains group individual shapes into larger patterns
- The brain favors speed to precision (illusions, errors)
- It also favors symmetry and simplicity
- Several gestalt principles are relevant in data visualization

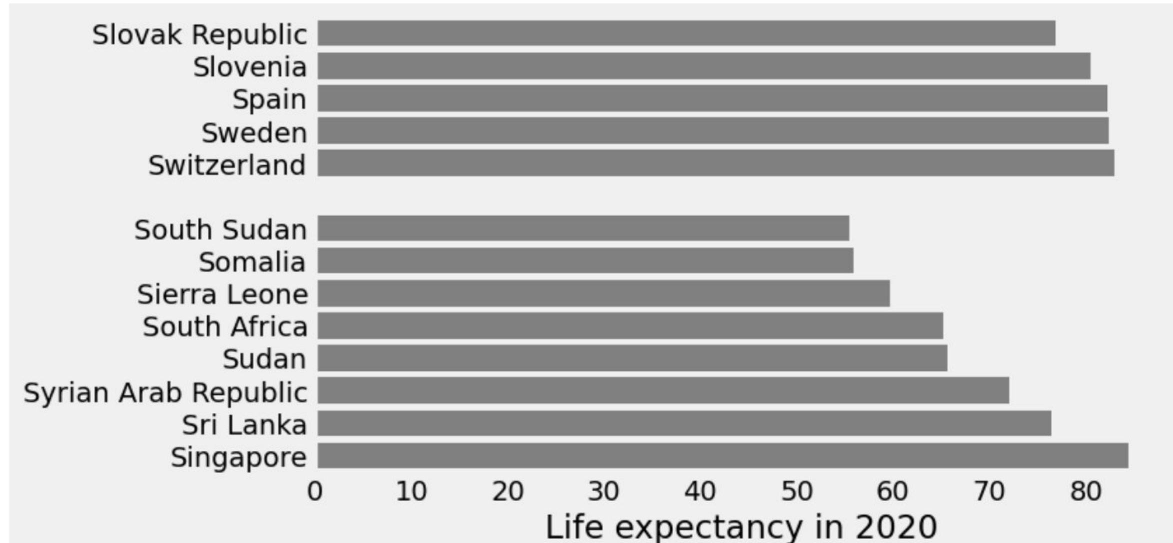


https://commons.wikimedia.org/wiki/File:Reification.svg

Principle of proximity

- Objects located close to each other are perceived as a group
- Good use of space in plots / tables / presentations can improve readability

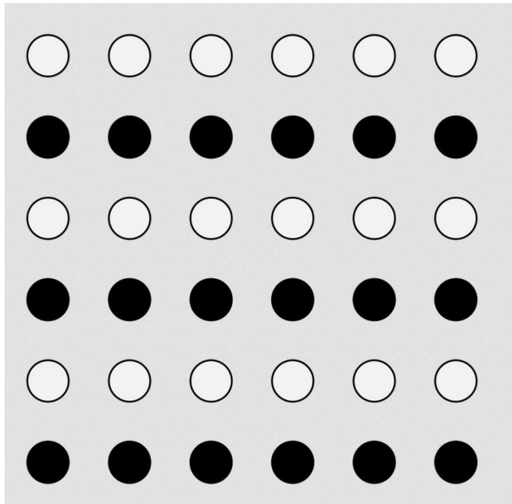




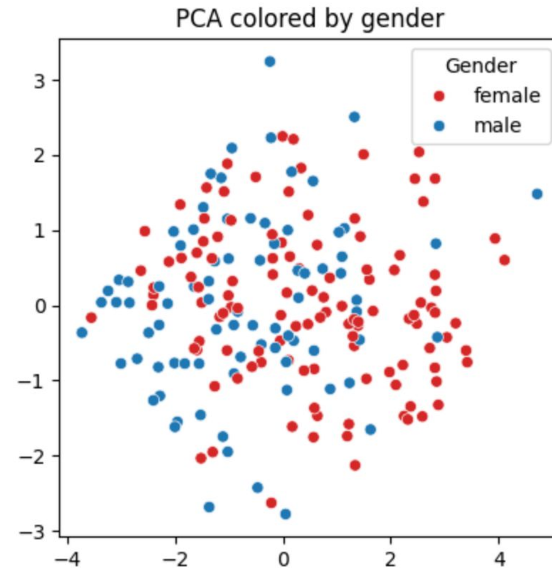
https://commons.wikimedia.org/wiki/File:Gestalt_proximity.svg

Principle of similarity

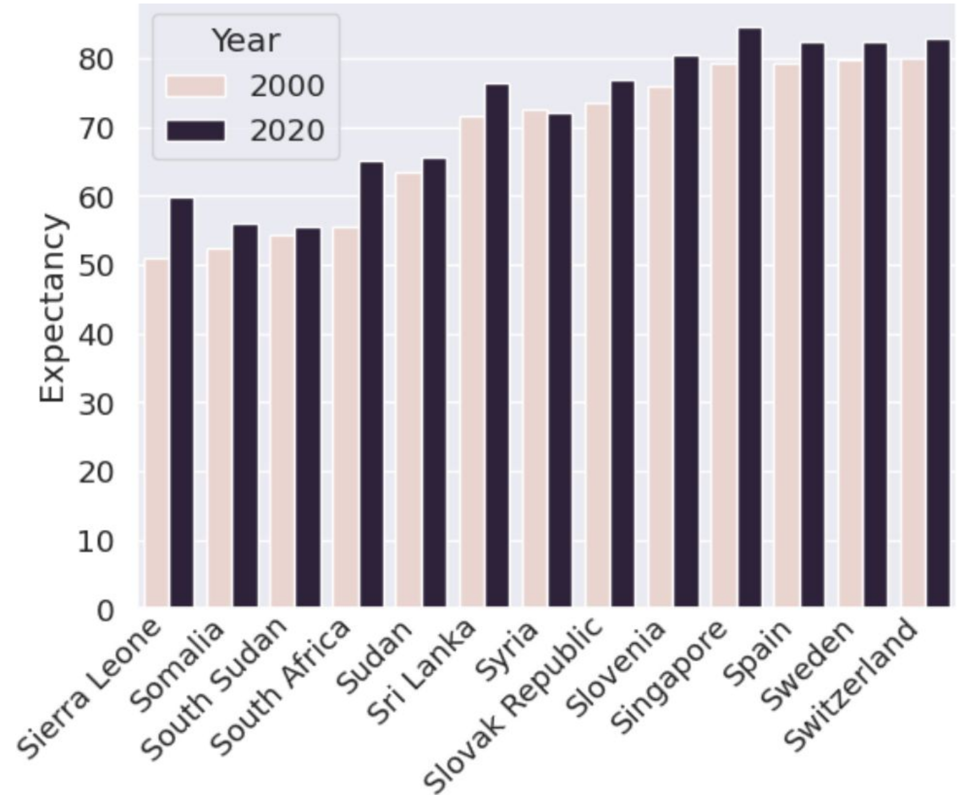
- Similar objects are perceived as a group even of not close by
- Various plots use color / shape to distinguish groups



https://commons.wikimedia.org/wiki/File:Gestalt_similarity.svg



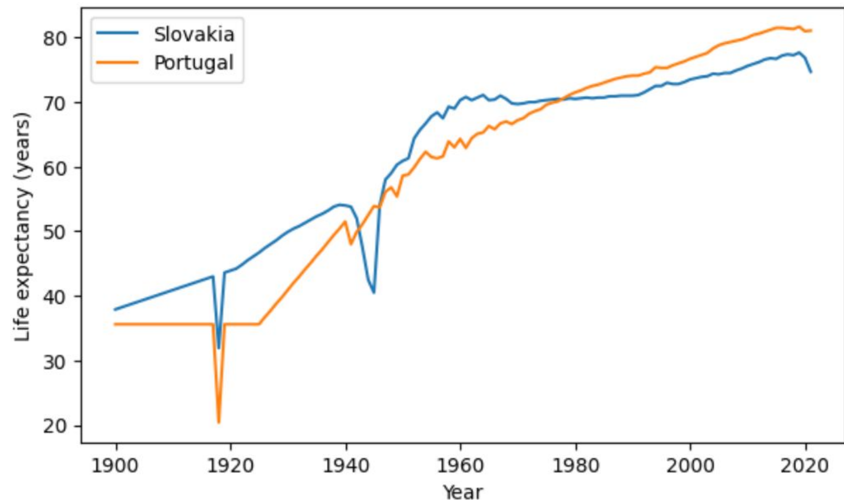
How are both principles used here?

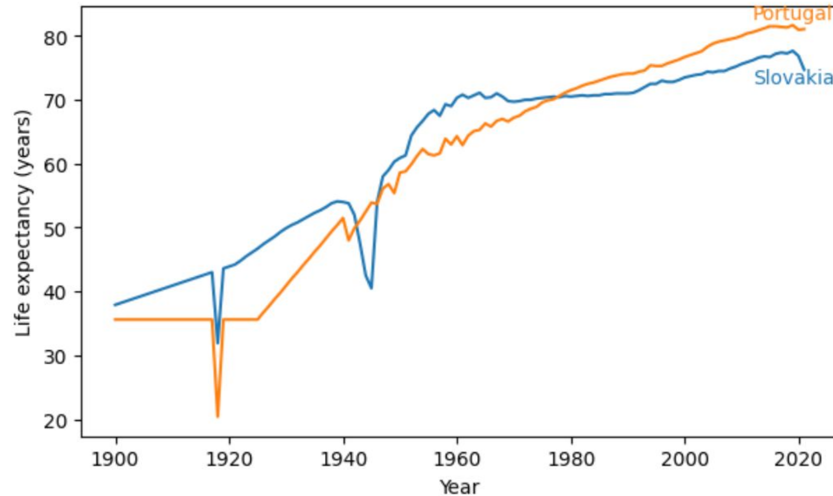


Example

separate legend vs marking lines with text in the same color

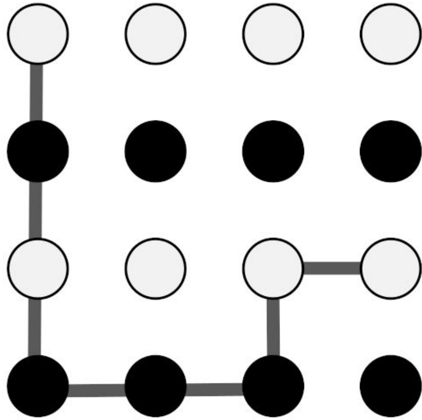
- using principles of proximity and similarity

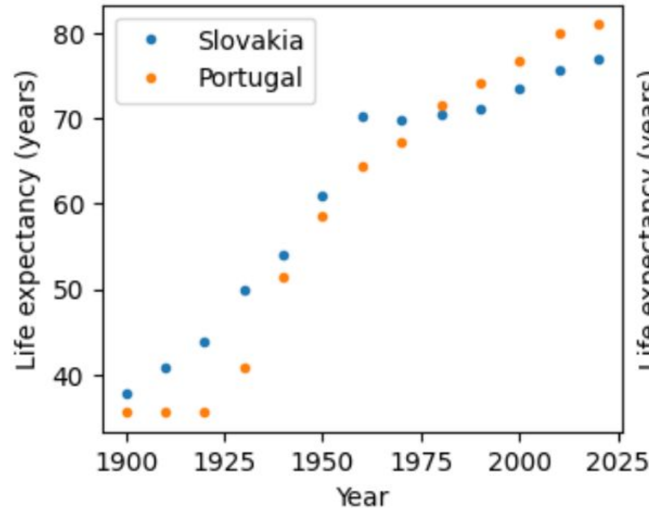


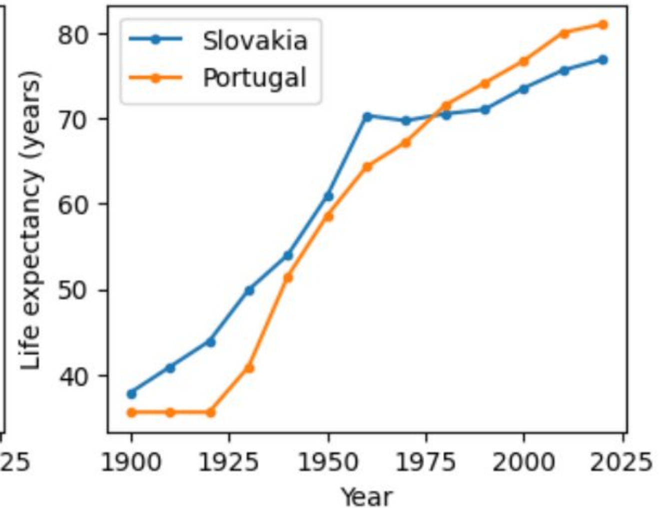


Principle of connection

- Connected objects are perceived to form a group
- Stronger than proximity and similarity
- Consider carefully when to use line graph vs. scatter plot

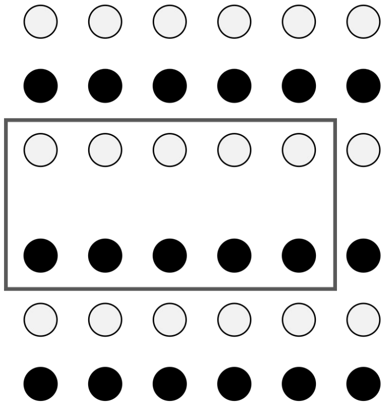


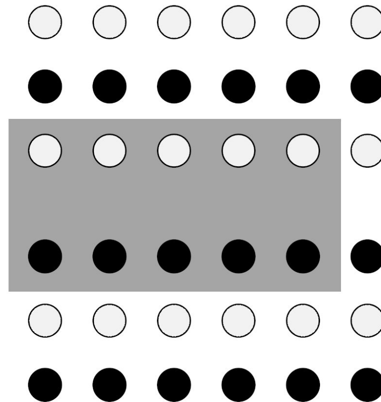


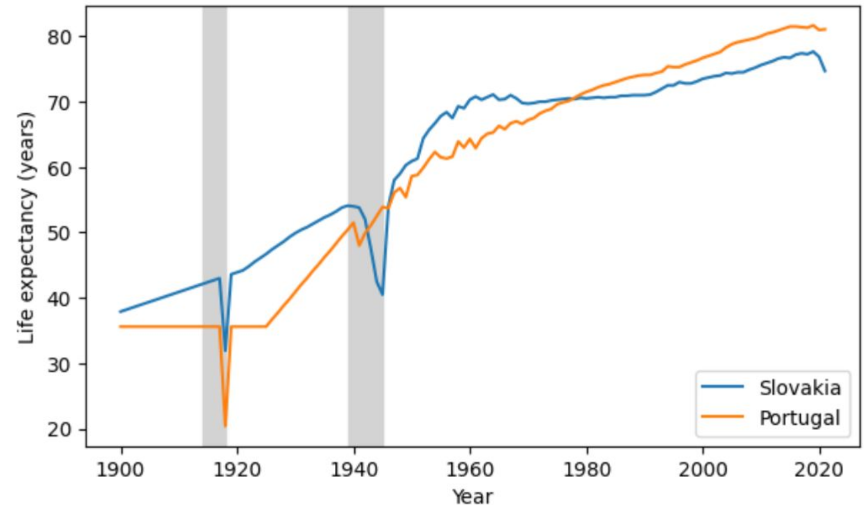


Principle of enclosure

- Enclosed objects are perceived as a member of the group
- Stronger than proximity and similarity
- Useful for highlighting in plots; little is enough (e.g. light background)

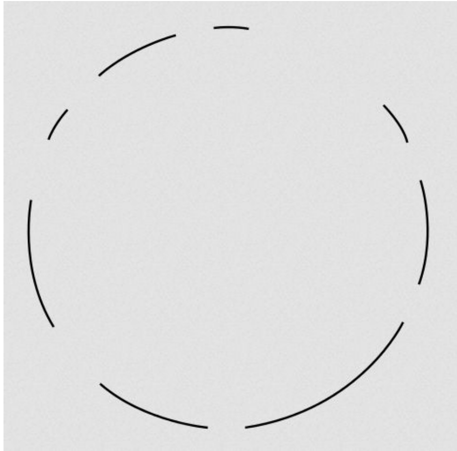


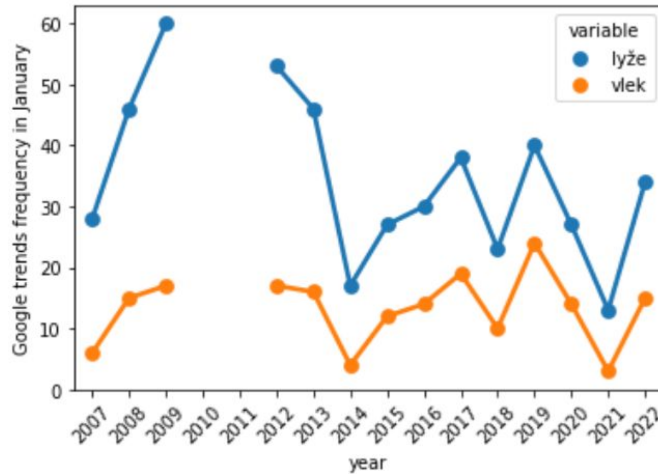




Principle of closure

- Our brain fills gaps in figures, connects interrupted lines
- Useful / dangerous when interruptions by design

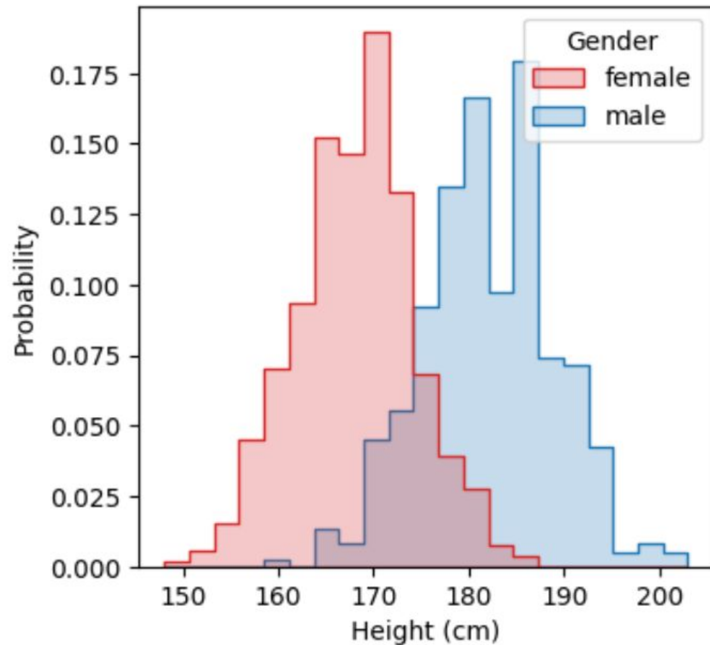


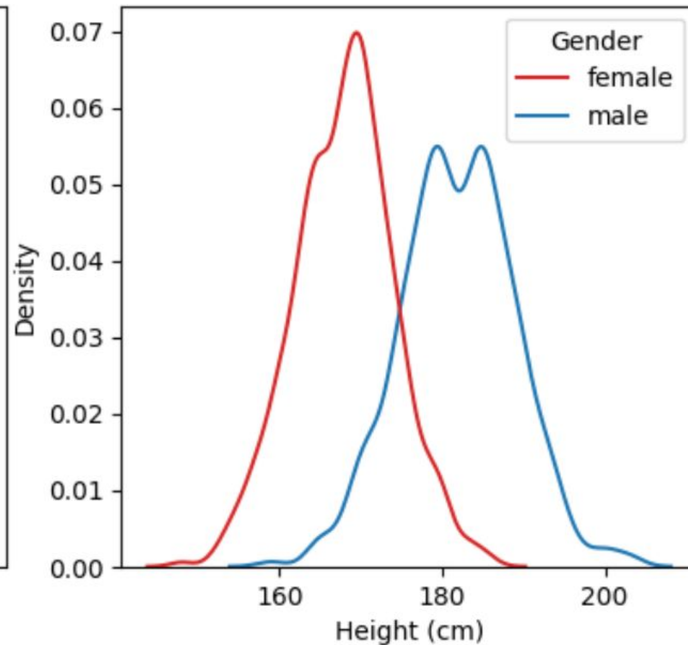


https://commons.wikimedia.org/wiki/File:Gestalt_closure.svg

Principle of continuity

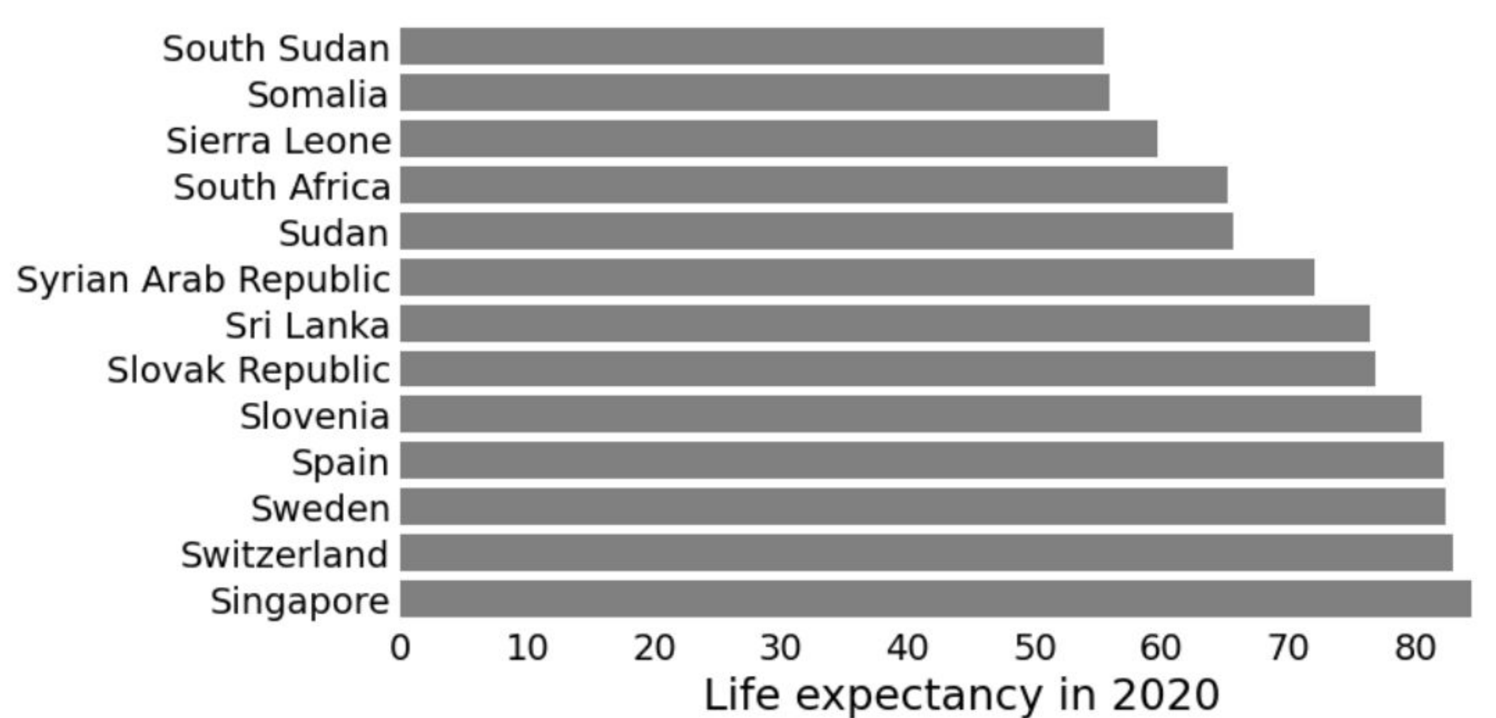
Smooth lines are easier to follow than angular and sharp





Frames not necessary, gestalt principles fills them in

(principles of closure and continuity)

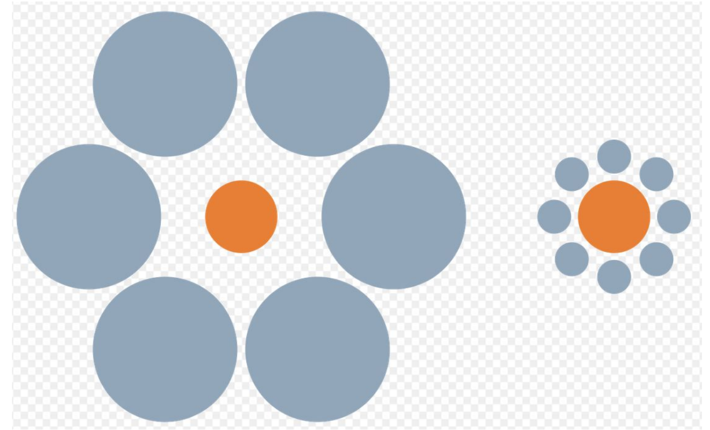


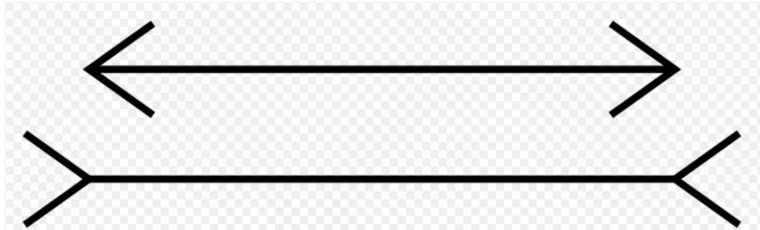
Illusions

- Fast visual processing leads to errors
- These are demonstrated by many optical illusions
- Beware not to create illusions in your plots

Illusions: length and size

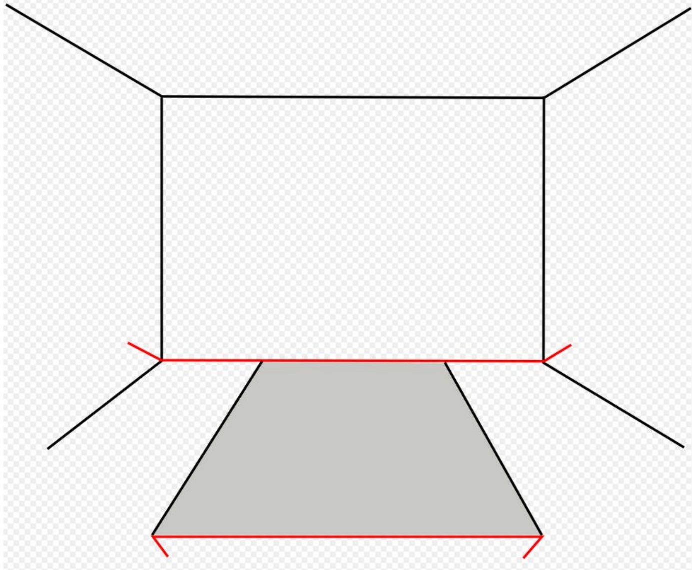
Müller-Lyer and Ebbinghaus illusions

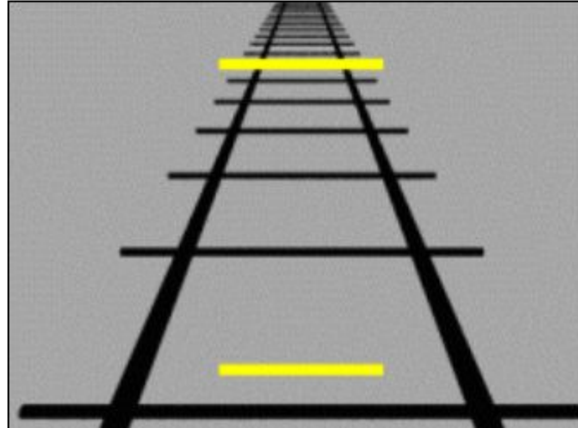




https://commons.wikimedia.org/wiki/File:M%C3%BCller-Lyer_illusion.svg
https://commons.wikimedia.org/wiki/File:Mond-vergleich.svg

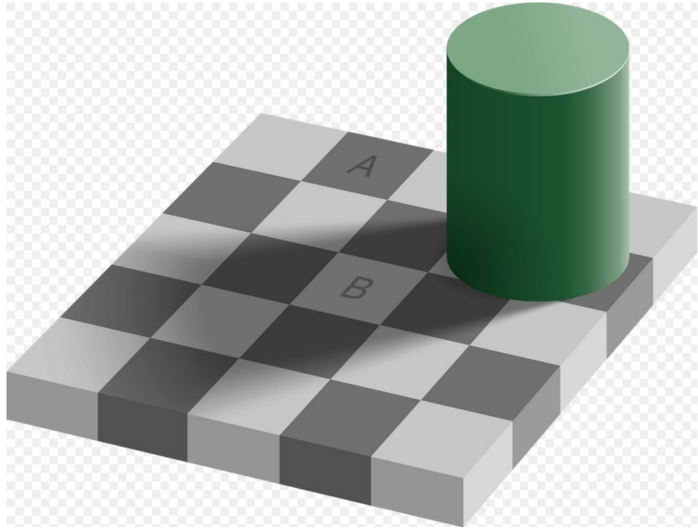
Illusions: length, perspective, spatial compensation

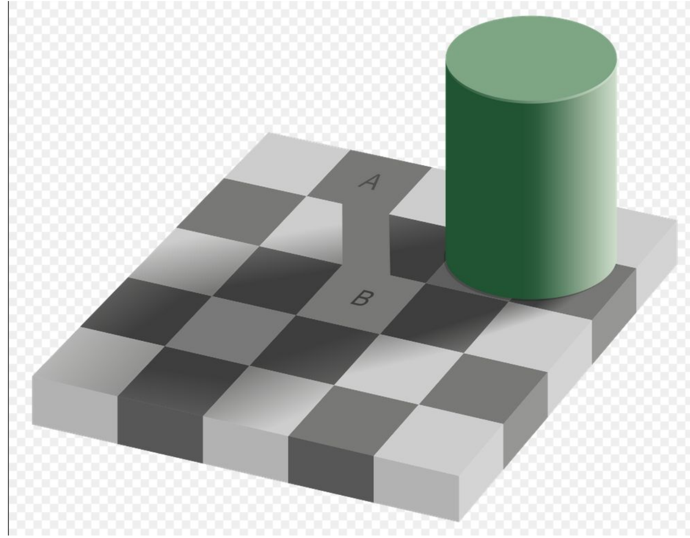




https://commons.wikimedia.org/wiki/File:Mueller_lyer.svg
https://commons.wikimedia.org/wiki/File:Ponzo_illusion.gif

Illusions: color





https://en.wikipedia.org/wiki/File:Checker_shadow_illusion.svg
https://commons.wikimedia.org/wiki/File:Grey_square_optical_illusion_proof2.svg

Illusions: color

Mach bands: when bands touch, the edge effect exaggerates their difference

Illusions: color

In heatmap the perception of colors influenced by their surroundings

```
a = np.array([
    [10, 10, 10,  0,  0,  0,  0],
    [10, 37, 10,  0,  0,  0,  0],
    [10, 10, 10,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0, 40, 40, 40],
    [ 0,  0,  0,  0, 40, 37, 40],
    [ 0,  0,  0,  0, 40, 40, 40],
    ])
pal = sns.color_palette("dark:white_r", as_cmap=True)
axes = sns.heatmap(data=a, square=True,
                   cmap=pal, cbar=False)
axes.axis('off')
```

Working memory

- **Iconic memory:** extremely short-term (<1s), simple pre-attentive processing
- **Visual short-term memory:** many seconds, but very small capacity (only 3-5 items)
- **Long-term memory:** store and recall selected information

Since working memory is small, looking at a plot with many colors requires back-and-forth between legend and plot

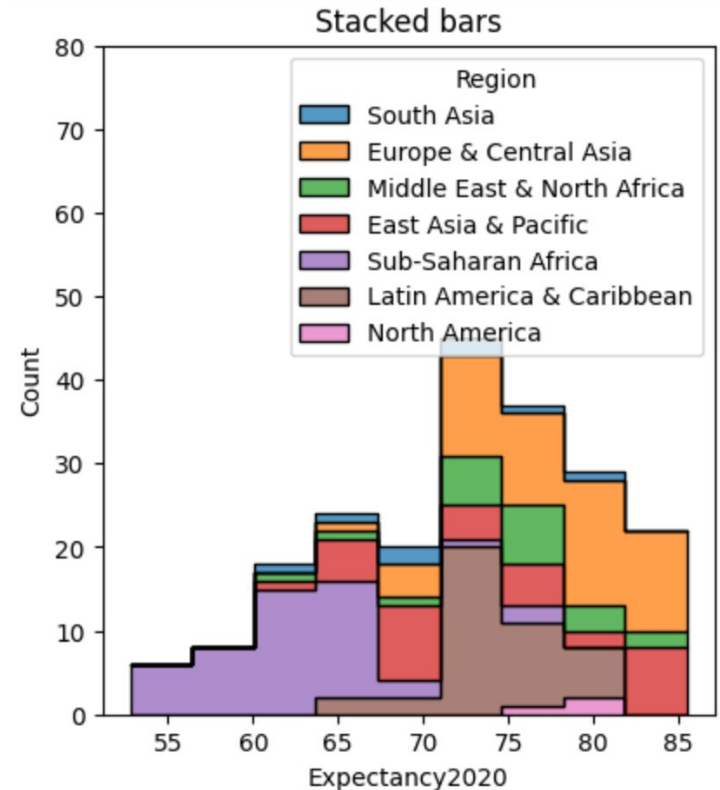


Chart and table junk

- **Chart junk**: elements of plots not necessary to convey information
- They unhelpfully catch our attention through pre-attentive attributes
- Most visualization can be improved by simplification
- Some redundancy can be useful

Nice visualizations of the simplification process:

- https://www.darkhorseanalytics.com/blog/data-looks-better-naked
- Also tables, maps and the unpopular pie charts

Summary

- **Pre-attentive attributes** are processed by our brains very fast
- Choosing the right attributes from the **hierarchy** allows accurate quantification
- Principles of **gestalt** describe how the brain connects part to the whole
- The brain can also make errors in visual processing as seen in **illusions**
- Removing **chart junk** concentrates our attention to the important elements

Additional sources

- Utilizing Gestalt Principles to Improve Your Data Visualization Design
- http://daydreamingnumbers.com/blog/gestalt-laws-data-visualization/
- Albert Cairo: The Functional Art
- C.N. Knaflic: Storytelling with Data
- Stephen Few: Now You See it

Visualizing text data

Visualizing text data

Working with natural text is difficult

- Complex grammar, ambiguous meaning, synonyms, etc.
- Lot of machine learning research
- Nonetheless sometimes simple statistics on frequencies of words or groups of words can be useful
- Usually we remove *stop words* (frequent words such as "and", "is"...) and apply *lemmatization* (convert inflected words to canonical form, such as "seen" -> "see")

Word clouds

State of the Union Address, 2002 vs. 2011

act afghanistan allies american attack best budget camps children citizens coalition congress continue corps country create danger depend destruction develop economy encourage enemies evil extend fight free freedom government health help history home homeland hope increase islamic jobs join lives mass military moment months nation opportunity peace people police power protect rebuild regimes resolve retirement security spending states tax terror terrorists thank thousands together tonight training true united war ways weapons women work workers world

President Bush, January 29, 2002

afghan ago already american behind believe best better building business care century challenge chance change child children clean college company compete congress country create cuts deficit democrats different don done dream economy education energy family future generation give goal government health help home idea innovation internet invest jobs laughter law life live money nation passed people percent possible projects race reform republicans research responsibility schools spending states step students success support sure tax teachers technology things together tonight troops willing win work workers world years

President Obama, January 25, 2011

https://commons.wikimedia.org/wiki/File:State_of_the_union_word_clouds.png

Word clouds

- Display the most common words from a text
- Size of words grows with frequency
- Arranged to be visually pleasing
- Not the best option for understanding/comparing word frequencies
- You can also display word frequencies using **bar graphs** and other plot types

Tag cloud

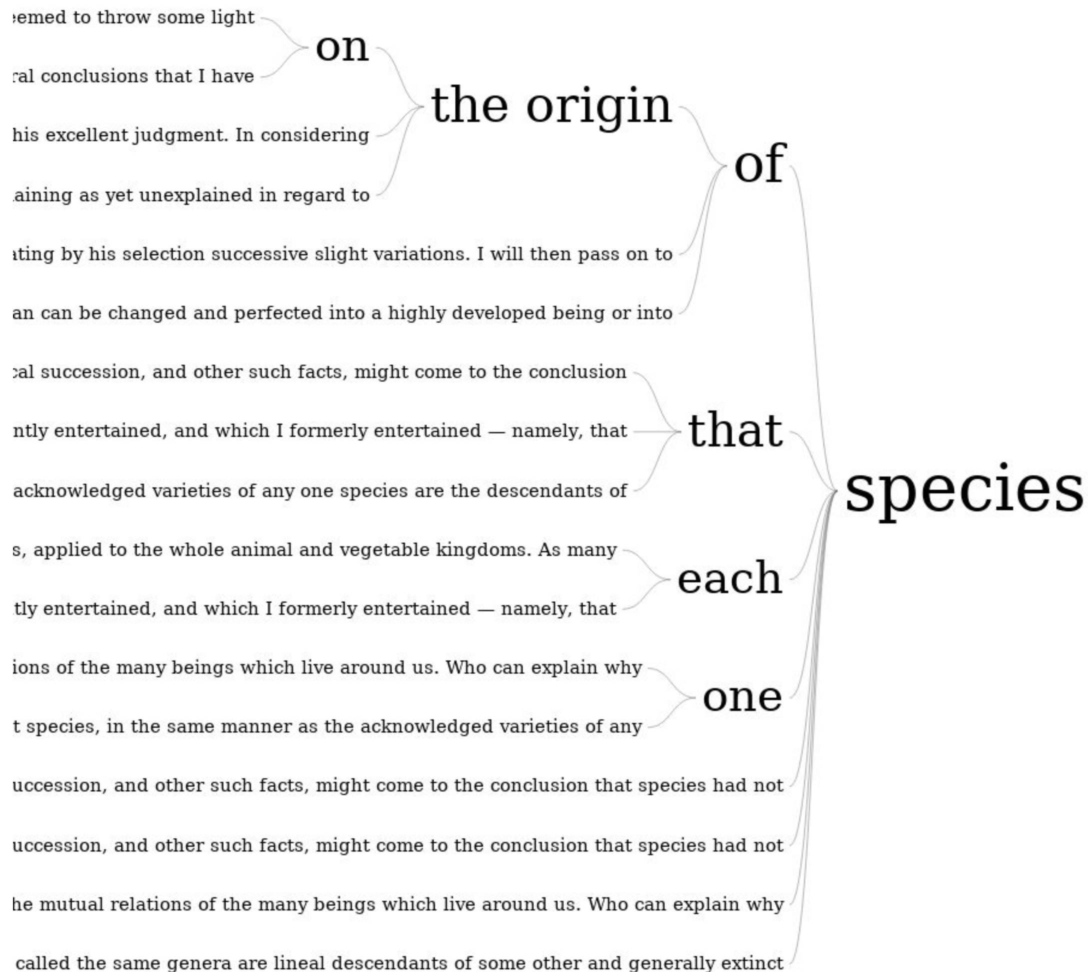
- Endings of German city names typical for individual regions
- Combination of a word cloud and map
- Figure from Reckziegel et al 2018

Word tree

Shows with words most often follow or precede a given word using a hierarchy

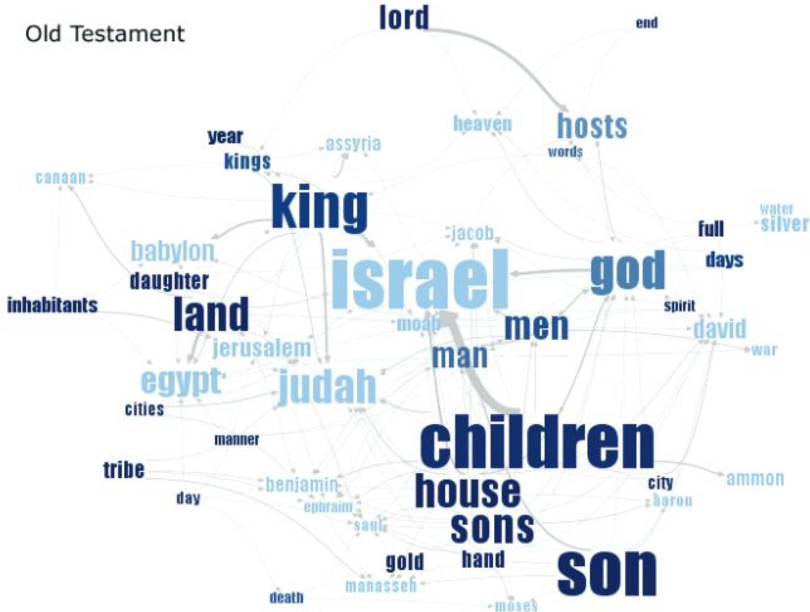
Text: Introduction to The Origin of Species by Charles Darwin, 1859, 1872

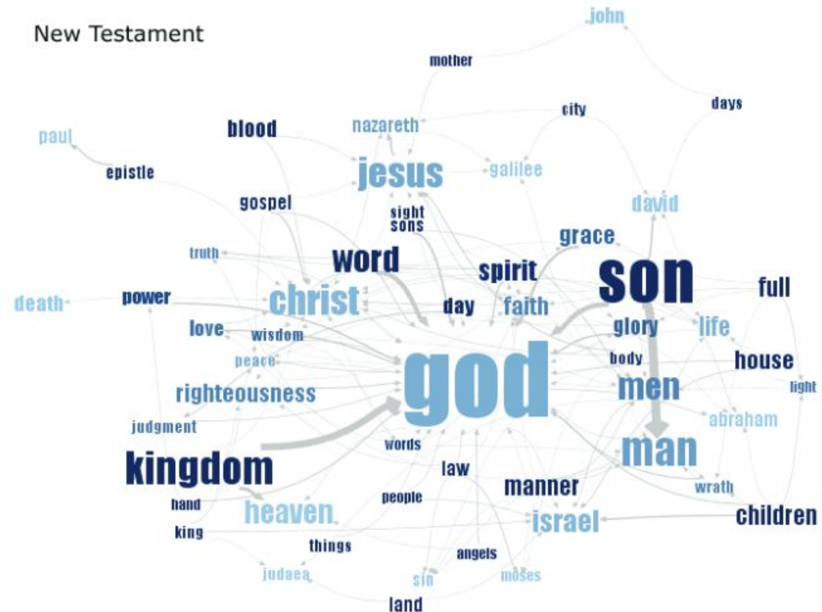
Figure source



Phrase nets

Phrases of type "X of Y", X connected to Y in a graph; source van Ham et al 2009





Text visualization: additional sources

- Courses Data management (2L), Principles of Data Science (3Z)
- Text visualization browser https://textvis.lnu.se/
- Lecture from Univ. of Washington
- Drawing Elena Ferrante's Profile: Finding out who is Elena Ferrante, bestselling Italian author (My Brilliant Friend) by comparing word frequencies etc. (see e.g. page 100)

Lecture 10
Presentation of results

Data visualization · 1-DAV-105

Lecture by Broňa Brejová

Acknowledgement: materials inspired by lectures from Martina Bátorová in 2021

Data analysis project phases

Recall from L08: Data analysis project phases

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
- Formation of **hypotheses**
- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation

Details: obtaining data

- Obtaining data
    - This course: we download whole datasets in a tabular form
    - But often also web scraping, manual collection of data, measurements, surveys,...
    - Requires careful planning

- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
- Formation of **hypotheses**
- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation

Details: preprocessing data

- Obtaining data
- Data preprocessing, **checking**, cleaning
  - Try to understand how (and why) the data was obtained and processed
  - Convert them to a convenient format
  - Check for missing values and suspicious outliers
  - Very important phrase: "Garbage in, garbage out"

- **Exploratory** analysis
- Formation of **hypotheses**
- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation

Details: exploratory analysis

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
  - Try many analyses
  - This course: visualizations and simple statistics
  - Later you learn advanced statistical and machine learning models
  - Less successful attempts may suggest new directions

- Formation of **hypotheses**
- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation

Details: Formation of hypotheses

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
- Formation of **hypotheses**
    - Select visualizations showing interesting trends / exceptions in the data
    - Formulate possible relationships
      (but remember, correlation does not imply causation)

- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation

Details: Testing hypotheses

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
- Formation of **hypotheses**
- Testing hypotheses
    - Recheck your code and data, try other related analyses
    - Try to find other relevant data or existing analyses by other people
    - If important decisions will be based on your result,
      test it particularly thoroughly
      (what would happen if our plot was all wrong?)

- **Explanatory** visualizations for the final report / presentation

Details: Explanatory visualizations

- Obtaining data
- Data preprocessing, **checking**, cleaning
- **Exploratory** analysis
- Formation of **hypotheses**
- Testing hypotheses
- **Explanatory** visualizations for the final report / presentation
  - Formulate your conclusions
  - Support them with your analysis and visualizations
  - Do not include all exploratory analyses
    (but do not hide data contradicting your conclusion)
  - Polish visualizations that you selected

Presentation of results

Presentation of results

- Understand **context**, audience, goals (more later)
- Tell a **story** (more later)
- Choose appropriate **visuals** (text / table / chart, appropriate type of graph, pre-attentive attributes, hierarchy of graph elements)
- Eliminate clutter, **focus attention** on the main points (pre-attentive attributes, such as color, size, spacing)
- Pay attention to **design** (accessibility due to font size and colors, aesthetics...)
- Get **feedback** and a lot of **practice**

(see Cole Nussbaumer Knaflic: Storytelling with data)

Understand the context of your presentation

Before writing any text or preparing any presentation try to find out:

- Who is your expected audience?
- What do they know and what do you need to explain?
- What might be interesting / new for them?
- What is the medium (live presentation, video, printed text, website)?
- What is the appropriate length (time, number of pages)?
- What do you want to achieve by the presentation?
  (inform / entertain / inspire to take a specific action)

Examples

Try to list some examples of situations where data visualization might be presented: who are speakers and audiences, what are goals

Situations where data visualizations are presented

- A **company** presents to potential **consumers**, persuades them to **buy** their products
- A **charity** presents to general **public**, persuades them to **donate**
- A **nonprofit / government** present to general **public**, persuades them to take **action** (live healthily, protect environment, ...)
- An **employee** presents to **colleagues**, persuades them to **change** processes
- A **politician** presents to general **public**, persuades them to **vote** for something
- A **journalist** writes for general **public**, informs them about important **issues**
- A **teacher** presents to **students**, teaches them a given **topic**
- A **student** presents to a **teacher**, demonstrates his / her **achievements** and skills
- A speaker talks to general **public**, **entertains / informs** about interesting topics
- A speaker talks to **experts**, informs about **new discoveries**, technologies etc.

Presentation of results

- Understand context
- **Tell a story**
- Choose appropriate visuals
- Eliminate clutter, focus attention on the main points
- Get feedback and a lot of practice

Storytelling

- We are easily captivated by a good story (book, movie, play)
  - We do not want to interrupt reading / watching
  - We can recall the plot afterwards
  - We want to achieve similar effects by your presentation
- Traditional stories structured as basic plot - twists - ending
- This roughly corresponds to introduction, actual content, conclusion
- Repetition useful in stories as well as in presentation
- Suspense and surprise

(see Cole Nussbaumer Knaflic: Storytelling with data)

Storytelling: structuring presentation

- One option is to describe your process of discovery roughly **chronologically** (omitting some dead ends): identifying question, getting data, analyzing data, coming to conclusion, recommending action
- Another option is to **lead with the ending**: starting with a call to action, backing it up with data

(see Cole Nussbaumer Knaflic: Storytelling with data)

Cognitive biases

Cognitive bias (kognitívne skreslenie)

- Cognitive bias is a systematic deviation from rational judgement
- A brain mechanism to create shortcuts, allow fast reasoning
- Term introduced by Amos Tversky and Daniel Kahneman in 1972

Very long list of biases discovered by researchers:

https://commons.wikimedia.org/wiki/File:The_Cognitive_Bias_Codex_-_180%2B_biases,_designed_by_John_Manoogian_III_(jm3).png

Three cognitive biases

- **Patternicity bias:** See non-existent patterns in data, even in random noise (related to seeing faces in the clouds)
- **Storytelling bias:** Invent "stories", explanations, cause-effect relationships for these patterns
- **Confirmation bias:** It is hard to discard our beliefs. We search for evidence that back our theories and interpret contradicting evidence the opposite way.

See Alberto Cairo: The Truthful Art

Cognitive biases in analysis and presentation

- Beware of biases in yourselves during analysis and in your audience during presentation
- *"The first principle is that you must not fool yourself---and you are the easiest person to fool"* Richard Feynman

Do not oversimplify

Story from Alberto Cairo: The Truthful Art

"*Study finds that more than a quarter journalism grads wish they'd chosen a different career*" Poynter Institute, 2013

Storytelling bias suggests:

- A change from printed to online media leads to worse job market for journalists
- Cairo as a journalism professor starts to worry about his future

Journalism grads (cont.)

"*Study finds that more than a quarter journalism grads wish they'd chosen a different career*" Poynter Institute, 2013

Actual value is 28%, as found by a survey

- This value by itself is presumably correct
- However it is not put into perspective, compared with other values

Results of Cairo's investigation

- The dissatisfaction among journalism students did not change much over the years
- Decreases in the number of news reporters and their low salaries
- Survey results imply sampling error which should be considered
- (Ideally compare to grads from other fields)

He suggests reformulating the message of the story:

"*Even if jobs prospects for journalists have worsened substantially and they may worsen even further in the future, the percentage of grads who wish they'd chosen a different career hasn't changed at all in more than a decade.*"

Properties of good visualization

- **Truthful** (based on thorough and honest research, high quality data, appropriate analysis, correct math, no bugs in code)
- **Functional** (constitutes an accurate depiction of the data, allows meaningful comparisons)
- **Beautiful** (attractive, intriguing, aesthetically pleasing for target audience)
- **Insightful** (reveals evidence hard to see otherwise)
- **Enlightening** (changes our minds for the better)

Alberto Cairo: The Truthful Art (journalist's perspective)

Back to thoughts on good visualization

Last lecture

Pre-attentive attributes are quickly recognized by our brain (size, color, position,...)

Hierarchy of graph elements: not all attributes are good for accurate quantitative reasoning

Gestalt principles: how brain connects elements into larger patterns (proximity, similarity, connection, enclosure, closure, continuity,)

Errors in visual processing lead to **illusions**

This informs our chart type choice (bars vs pies) and elimination of chart junk

Additional aspects of good plot choice

Basic setup: Selecting variables, choosing type of plot, assigning variables to x, y, color...

Data transformations: filtering (e.g. select data from one region), aggregating (e.g. summary per region) to avoid overplotting

Additional settings: sorting (e.g. bar graph columns), rescaling (log axis), re-expressing (e.g. absolute value vs relative change), zooming

Focus and explanation: highlighting, annotating (adding notes to plot)

Inspired by Stephen Few: Now you see it

Speed is not always everything

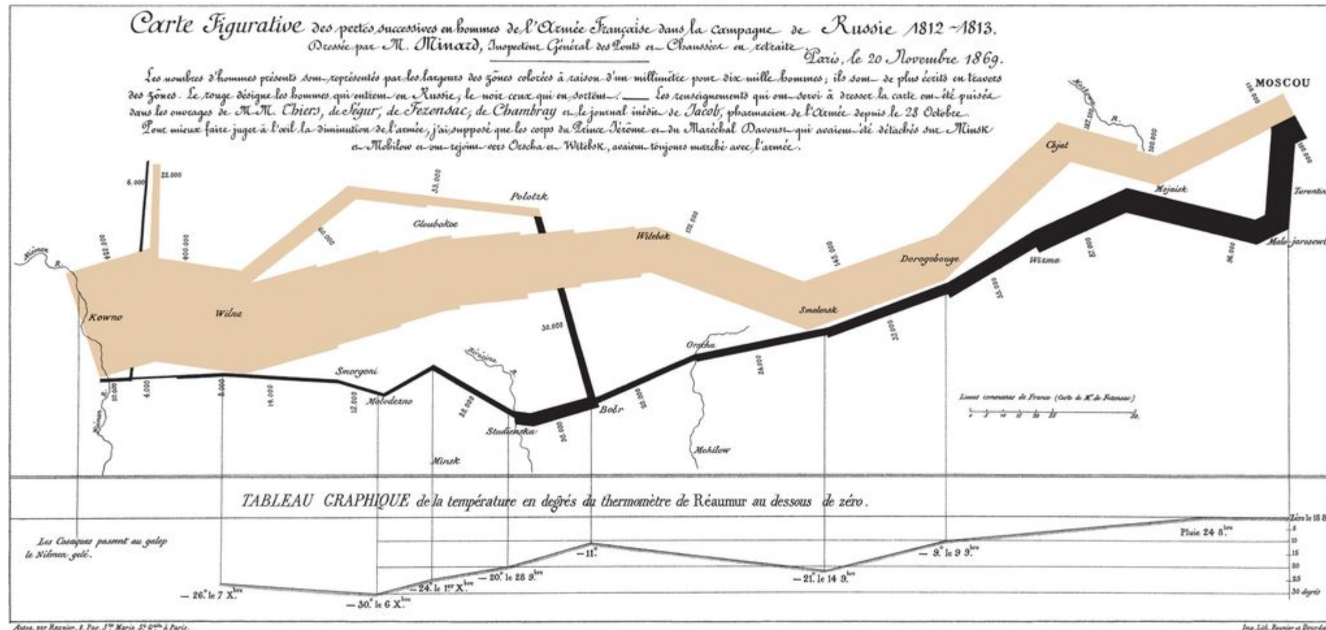
While there is a place for rapidly-understood graphs, it is too limiting to make speed a requirement in science and technology, where the use of graphs ranges from detailed in-depth data analysis to quick presentation. […]
The important criterion for a graph is not simply how fast we can see a result; rather it is whether through the use of the graph we can see something that would have been harder to see otherwise or that could not have been seen at all.

William Cleveland, The Elements of Graphing Data, Chapter 2

Recall: exploratory vs. explanatory analysis, sometimes audience can explore too

Recall Minard's plot of French army losses

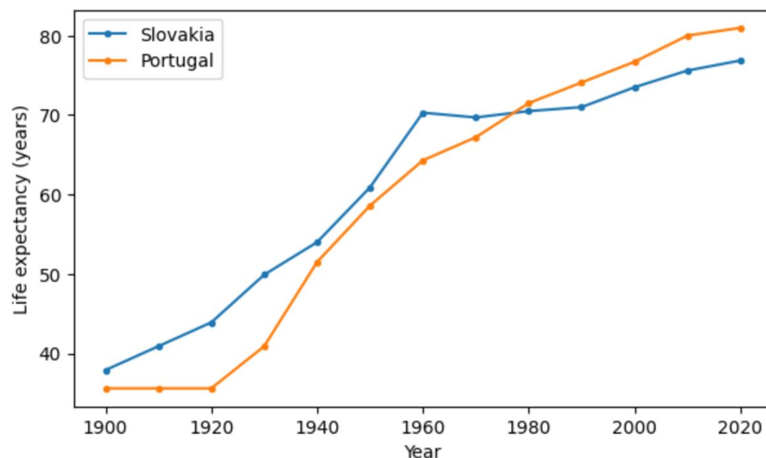
Easy to see big picture but also many minute details



https://commons.wikimedia.org/wiki/File:Minard.png

Tables vs. graphs

When is it good to include a table instead of / in addition to a graph?



	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010	2020
Slovakia	**37.9**	**40.9**	**43.9**	**49.9**	**54.0**	**60.9**	**70.3**	**69.7**	70.5	71.0	73.5	75.6	76.9
Portugal	35.6	35.6	35.6	40.9	51.5	58.6	64.3	67.2	**71.5**	**74.1**	**76.7**	**80.0**	**81.0**

Tables vs. graphs

Advantages of tables:

- **Very few numbers** typically better given directly than in a graph
- In a long table, each reader can **find items** of personal interest (e.g. results of a sport competition, statistics for all countries)
- A table gives **exact values**
- Readers can **re-analyze** the same data (table preferably machine-readable)
- Numbers at very **different scales** are sometimes difficult to display even with log axes

See also
https://www.storytellingwithdata.com/blog/2011/11/visual-battle-table-vs-graph

Examples of bad graphs and their improvements

- http://www.perceptualedge.com/examples.php
- https://eagereyes.org/pie-charts
- https://viz.wtf/

Infographics

Some examples of infographics

Several examples that are close to data visualization:

- Income by religious group in US (image, website)
- Deadliest pandemics (website)
- War casualties (website)
- Game of Thrones relationships (website)
- Emergency medical services in Slovakia 2019 (website)

Some explain other types of information:

- Sitting and standing is bad (website)

Data visualization (DV) vs infographics (IG)

- **Target audience:** IG general public, DV often experts
- **Storytelling:** often in IG, can be created from multiple DV
- **Design and aesthetics:** more elaborate in IG, includes graphics elements and clipart (considered chart junk in DV)
- **Process of creation:** many simple tools for DV, IG time consuming, often created by collaboration of data analysis, domain experts and graphic designers

https://www.statsilk.com/blog/real-difference-between-infographics-and-data-visualizations

1 Lecture 11: Infographics, interactivity, other tools, specialized plots

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

Acknowledgement: some materials inspired by lectures from Martina Bátorová in 2021

1.0.1 Several examples of infographics

Several examples that are close to data visualization:

- Income by religious group in US (image, website)
- Deadliest pandemics (website)
- War casualties (website)
- Game of Thrones relationships (website)
- Emergency medical services in Slovakia 2019 (website)

Some explain other types of information:

- Sitting and standing is bad (website)

1.1 Data visualization (DV) vs infographics (IG)

- **Target audience:** IG general public, DV often experts
- **Storytelling:** often in IG, can be created from multiple DV
- **Design and aesthetics:** more elaborate in IG, includes graphics elements and clipart (considered chart junk in DV)
- **Process of creation:** many simple tools for DV, IG time consuming, often created by collaboration of data analysis, domain experts and graphic designers

See also https://www.statsilk.com/blog/real-difference-between-infographics-and-data-visualizations

1.2 Interactivity

Interactive visualization engages audience, allows them to explore data in depth and according to their interest.

1.2.1 Examples

- PhD gender gap (website)
- Making it big (website)
- US cities with the same name (website)

1.2.2 Techniques in interactivity visualization

Similar to decisions made in designing a static plot:

- Selecting variables (x, y, color, …)
- Filtering data (selecting table rows)
- Highlighting points or groups

- Aggregating (display countries or region summaries)
- Zooming / panning
- Rescaling (log-scale) / reexpressing (e.g. % instead of counts)
- Sorting (e.g. bars in bargraphs)
- Displaying details (tooltips)
- Annotating
- Bookmarking

(Stephen Few)

1.2.3 Dashboard

- A display consisting of mutiple plots, summarizing current state of important indicators (e.g. of a business, pandemics, …)
- Inspired by dashboards in cars and planes
- Often interactive, but main features in default view

Two SARS-CoV-2 examples:

- WHO
- Nextstrain
  - many options: selecting color, filtering, highlighting, aggregating, zooming and panning (maps and tree), rescaling (time vs divergence), tooltips, bookmarking

1.2.4 Interactivity in Plotly Express

All Plotly plots by default have some interactivity:

- Filtering groups
- Zooming / panning
- Details
- Spike lines

Example 1: Country indicators from World Bank, https://databank.worldbank.org/home under CC BY 4.0 license.

Regions can be switched on and off.

[1]:
```
import plotly.express as px
import pandas as pd
url = 'http://compbio.fmph.uniba.sk/vyuka/viz/images/9/9d/World_bank.csv'
countries = pd.read_csv(url)

px.scatter(
        countries, x="GDP2018", y="Expectancy2018", color="Region",
        hover_data=['Country'],
        title="Country indicators 2018", log_x=True,
        width=800, height=500
    )
```

Example 2: Life expectancy data provided free by the Gapminder foundation under the CC-BY license.

Compare data along the x coordinate.

```
[2]: url="http://compbio.fmph.uniba.sk/vyuka/viz/images/3/33/
     ↪Gapminder_life_expectancy_years.csv"
     orig_expectancy = pd.read_csv(url)
     expectancy = pd.melt(orig_expectancy, id_vars=["country"], var_name="year")
     expectancy['year'] = expectancy['year'].astype(int)
```

```
[3]: selected = expectancy.query("country=='Slovak Republic' or country=='France'")
     fig=px.line(
             selected, x="year", y="value", color="country",
             title="Life expectancy", width=800, height=500
         )
     fig.update_layout(hovermode="x unified")
```

1.2.5 More interaction with Dash by Plotly

- Dash library by Plotly allows adding control elements (selectors, sliders, buttons, …)
- We have seen an example in L01

1.3 Other visualization tools

Non-programmers typically create plots in spreadsheets:

- Excel (examples)
- Google sheets (examples)

System R: programming language for statistical computing

- Together with Python, very popular in data science
- Built-in plots
- Also other libraries, notably ggplot2 based on system called Grammar of Graphics (cheat-sheet)

Javascript

- Programming language popular in web programming
- Google charts for Javascript (examples)
- D3.js library (Data-Driven Documents)

Tableau

- Advanced visualization tools, commercial
- Gallery

Microsoft Power BI

- Interactive data visualization software with a focus on business intelligence
- An example

1.4 Several specialized visualization types

1.4.1 UML diagrams in computer science

- Display relationships between different classes or other components and aspects of software

https://commons.wikimedia.org/wiki/File:UML_diagrams_overview.svg Derfel73; Pmerson

1.4.2 Waterfall chart

- Used in bussiness analysis: financial, inventory, human resources etc.
- Displays effects decreasing or increasing a given value
- The first and last columns are bars displaying starting and final value
- Intermediate columns float, displaying changes from previous total
- Description

https://commons.wikimedia.org/wiki/File:Waterfallchart_ex2.jpg FusionCharts Blog, CC BY-SA 4.0

1.4.3 Funnel charts

- Display losses within a business process, e.g from website visit to actual purchase
- Horizontal bar chart with centered bars
- Beware: different from funnel plot in medical meta-analyses of multiple publications

[4]:
```
# example from https://plotly.com/python/funnel-charts/
data = dict(
    number=[39, 27.4, 20.6, 11, 2],
    stage=["Website visit", "Downloads", "Potential customers", "Requested␣
 ↪price", "invoice sent"])
fig = px.funnel(data, x='number', y='stage')
fig.show()
```

1.4.4 Gantt chart

- Used in management to display project schedule with different tasks and their planned duration
- Can also display current status of tasks and their dependencies

https://commons.wikimedia.org/wiki/File:GanttChartAnatomy.svg

1.4.5 Candlestick chart

- Similar to boxplot, used in financial data, e.g. stocks, currency exchange rates
- Line: minimum and maximum, box: opening and close, color: increase or decrease

https://commons.wikimedia.org/wiki/File:Candlestick_Chart_in_MetaTrader_5.png