Web Mapping and Geovisualisation

Gabriele Filomena and Elisabetta Pietrostefani

2024-07-24

Table of contents

Welcome

This is the website for "Web Mapping and Geovisualisation" (module **ENVS456**) at the University of Liverpool. This course is designed and delivered by Dr. Gabriele Filomena and Dr. Elisabetta Pietrostefani from the Geographic Data Science Lab at the University of Liverpool, United Kingdom. The module has two main aims. It seeks to provide hands-on experience and training in:

- The design and generation of web-based mapping and geographical information tools.
- The use of software to access, analyse and visualize web-based geographical information.

 A compilation of this web course is hosted as a GitHub repository that you can access:

- As an html website.
- As a GitHub repository.

Contact

Gabriele Filomena - gfilo [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 1xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Elisabetta Pietrostefani - e.pietrostefani [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 6xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Overview

Aims

This module aims to provide hands-on experience and training in: - The design and generation of (good looking) web-based mapping and geographical information tools. - The use of software to access, analyse and visualize web-based geographical information.

Learning Outcomes

By the end of the module, students should be able to:

(2) Visualise and represent geo-data through static and dynamic maps.
(3) Recognise and describe the component of web based mapping infrastructure.
(4) Collect Web-based data.
(5) Generate interactive maps and dashboards.
(6) Understand basic concepts of spatial network analysis.
(7) Manipulate geo-data through scripting in Python.

Feedback

Formal assessment. Two pieces of coursework (50%/50%). Equivalent to 2,500 words each

Verbal face-to-face feedback. Immediate face-to-face feedback will be provided during computer, discussion and clinic sessions in interaction with staff. This will take place in all live sessions during the semester. *Teams Forum*. Asynchronous written feedback will be provided via Teams. Students are encouraged to contribute by asking and answering questions relating to the module content. Staff will monitor the forum Monday to Friday 9am-5pm, but it will be open to students to make contributions at all times. Response time will vary depending on the complexity of the question and staff availability.

1 Introduction & Python Refresher

The **Lecture slides** can be found here.

This **lab**'s notebook can be downloaded from here.

1.1 Part I: Powerful Web Mapping Examples

This part of the lab has two main components: 1. The first one will require you to find a partner and work together with her/him 2. And the second one will involve group discussion.

1.1.1 Paired Activity

In pairs, find **three** examples where web maps are used to communicate an idea. Complete the following sheet for each example:

- **Substantive**
  - `Title`: Title of the map/project
  - `Author`: Who is behind the project?
  - `Big idea`: a "one-liner" on what the project tries to accomplish –
  - `Message`: what does the map try to get accross
- **Technical**
  - `URL`:
  - `Interactivity`: does the map let you interact with it in any way? Yes/No
  - `Zoomable`: can you explore the map at different scales? Yes/No
  - `Tooltips`:
  - `Basemap`: Is there an underlying map providing geographical context? Yes/No. If so, who is it provided by?
  - `Technology`: can you guess what technology does this map rely on?

Post each sheet as a separate item on the Teams channel for Lab No.1

1.1.1.1 Example

The project "WHO Coronavirus (COVID-19) Dashboard"

- **Substantive**
  - `Title`: WHO Coronavirus (COVID-19) Dashboard
  - `Author`: World Health Organization
  - `Big idea`: Shows confirmed COVID-19 cases and deaths by country to date
  - `Message`: The project displays a map of the world where COVID-19 cases are shown by country. This element is used to show which countries have had more cases (large trends). A drop down button allows us to visualise the map by a) Total per 100,000 population b) % change in the last 7 days c) newly reported in the last 7 days d) newly reported in the last 24 hours.
- **Technical**
  - `URL`: https://covid19.who.int/
  - `Interactivity`: Yes
  - `Zoomable`: Yes
  - `Tooltips`: Yes
  - `Basemap`: No
  - `Technology`: Unknown

Here are a couple of other COVID-19 examples of web-maps that where basemaps and technology is easier to spot.

- "London School of Hygiene & Tropical Medicine - COVID-19 tracker"
- "Tracking Coronavirus in the United Kingdom: Latest Map and Case Count"

1.1.2 Class discussion

We will select a few examples posted and collectively discuss (some of) the following questions:

1. What makes them powerful, what "speaks" to us?
2. What could be improved, what is counter-intuitive?
3. What design elements do they rely on?
4. What technology do they use?

1.1.3 References

- For an excellent coverage of "visualisation literacy", Chapter 11 of Andy Kirk's "Data Visualisation" is a great start. Lab: Getting up to speed for web mapping
- A comprehensive overview of computational notebooks and how they relate to modern scientific work is available on Ch.1 of the GDS book.
- A recent overview of notebooks in Geography is available in Boeing & Arribas-Bel (2021)

1.2 Part II: Python/Pandas (Refresher)

Gabriele Filomena has prepared this notebook by readapting material shared on this repository.

1.2.1 Python

A quick overview of ubiquitous programming concepts including data types, for loops, if-then-else conditionals, and functions.

```
import numpy as np
import pandas as pd
```

```
# integers (int)
x = 100
type(x)
```

```
# floating-point numbers (float)
x = 100.5
type(x)
```

```
# sequence of characters (str)
x = 'Los Angeles, CA 90089'
len(x)
```

```
# list of items
x = [1, 2, 3, 'USC']
len(x)
```

```
# sets are unique
x = {2, 2, 3, 3, 1}
x
```

```
# tuples are immutable sequences
latlng = (34.019425, -118.283413)
type(latlng)
```

```
# you can unpack a tuple
lat, lng = latlng
type(lat)
```

```
# dictionary of key:value pairs
iceland = {'Country': 'Iceland', 'Population': 372520, 'Capital': 'Reykjavík', '% Foreign Po
type(iceland)
```

```
# you can convert types
x = '100'
print(type(x))
y = int(x)
print(type(y))
```

```
# you can loop through an iterable, such as a list or tuple
for coord in latlng:
    print('Current coordinate is:', coord)
```

```
# loop through a dictionary keys and values as tuples
for key, value in iceland.items():
    print(key, value)
```

```
# booleans are trues/falses
x = 101
x > 100
```

```
# use two == for equality and one = for assignment
x == 100
```

```
# if, elif, else for conditional branching execution
x = 101
if x > 100:
    print('Value is greater than 100.')
elif x < 100:
    print('Value is less than 100.')
else:
    print('Value is 100.')
```

```
# use functions to encapsulate and reuse bits of code
def convert_items(my_list, new_type=str):
    # convert each item in a list to a new type
    new_list = [new_type(item) for item in my_list]
    return new_list

l = [1, 2, 3, 4]
convert_items(l)
```

1.2.2 `pandas` Series and DataFrames

pandas has two primary data structures we will work with: `Series` and `DataFrame`.

1.2.2.1 Pandas Series

```
# a pandas series is based on a numpy array: it's fast, compact, and has more functionality
# it has an index which allows you to work naturally with tabular data
my_list = [8, 5, 77, 2]
my_series = pd.Series(my_list)
my_series
```

```
# look at a list-representation of the index
my_series.index.tolist()
```

```
# look at the series' values themselves
my_series.values
```

```
# what's the data type of the series' values?
type(my_series.values)
```

```
# what's the data type of the individual values themselves?
my_series.dtype
```

1.2.2.2 Pandas DataFrames

```
# a dict can contain multiple lists and label them
my_dict = {'hh_income'  : [75125, 22075, 31950, 115400],
           'home_value' : [525000, 275000, 395000, 985000]}
my_dict
```

```
# a pandas dataframe can contain one or more columns
# each column is a pandas series
# each row is a pandas series
# you can create a dataframe by passing in a list, array, series, or dict
df = pd.DataFrame(my_dict)
df
```

```
# the row labels in the index are accessed by the .index attribute of the DataFrame object
df.index.tolist()
```

```
# the column labels are accessed by the .columns attribute of the DataFrame object
df.columns
```

```
# the data values are accessed by the .values attribute of the DataFrame object
# this is a numpy (two-dimensional) array
df.values
```

1.2.3 Loading data in Pandas

Usually, you'll work with data by loading a dataset file into pandas. CSV is the most common format. But pandas can also ingest tab-separated data, JSON, and proprietary file formats like Excel .xlsx files, Stata, SAS, and SPSS.

Below, notice what pandas's `read_csv` function does:

1. Recognize the header row and get its variable names.
2. Read all the rows and construct a pandas DataFrame (an assembly of pandas Series rows and columns).
3. Construct a unique index, beginning with zero.
4. Infer the data type of each variable (i.e., column).

```
# load a data file
# note the relative filepath! where is this file located?
# use dtype argument if you don't want pandas to guess your data types
df = pd.read_csv('../data/GTD_2022.csv', low_memory = False)
```

```
to_replace = [-9, -99, "-9", "-99"]
for value in to_replace:
    df = df.replace(value, np.NaN)

df['eventid'] = df['eventid'].astype("Int64")
```

```
# dataframe shape as rows, columns
df.shape
```

```
# or use len to just see the number of rows
len(df)
```

```
# view the dataframe's "head"
df.head()
```

```
# view the dataframe's "tail"
df.tail()
```

```
# column data types
df.dtypes
```

```
# or
for dt in df.columns[:10]:
    print(dt, type(dt))
```

1.2.4 Selecting and slicing data from a DataFrame

```
# CHEAT SHEET OF COMMON TASKS
# Operation                       Syntax           Result
#------------------------------------------------------------
# Select column by name           df[col]          Series
# Select columns by name          df[col_list]     DataFrame
# Select row by label             df.loc[label]    Series
# Select row by integer location  df.iloc[loc]     Series
# Slice rows by label             df.loc[a:c]      DataFrame
# Select rows by boolean vector   df[mask]         DataFrame
```

1.2.4.1 Select DataFrame's column(s) by name

```
# select a single column by column name
# this is a pandas series
df['country']
```

```
# select multiple columns by a list of column names
# this is a pandas dataframe that is a subset of the original
df[['country_txt', 'year']]
```

```
# create a new column by assigning df['new_col'] to some values
# people killed every perpetrator
df['killed_per_attacker'] = df['nkill'] / df['nperps']

# inspect the results
df[['country', 'year', 'nkill', 'nperps', 'killed_per_attacker']].head(15)
```

1.2.4.2 Select row(s) by label

```
# use .loc to select by row label
# returns the row as a series whose index is the dataframe column names
df.loc[0]
```

```
# use .loc to select single value by row label, column name
df.loc[15, 'gname'] #group name
```

```
# slice of rows from label 5 to label 7, inclusive
# this returns a pandas dataframe
df.loc[5:7]
```

```
# slice of rows from label 17 to label 27, inclusive
# slice of columns from country_txt to city, inclusive
df.loc[17:27, 'country_txt':'city']
```

```
# subset of rows from with labels in list
# subset of columns with names in list
df.loc[[1, 350], ['country', 'gname']]
```

```
# you can use a column of identifiers as the index (indices do not *need* to be unique)
df_gname = df.set_index('gname')
df_gname.index.is_unique
```

```
df_gname.head(3)
```

```
# .loc works by label, not by position in the dataframe
try:
    df_gname.loc[0]
except KeyError as e:
    print('label not found')
```

```
# the index now contains gname values, so you have to use .loc accordingly to select by row l
df_gname.loc['Taliban'].head()
```

1.2.4.3 Select by (integer) position - Independent from actual Index

```
# get the row in the zero-th position in the dataframe
df.iloc[0]
```

```
# you can slice as well
# note, while .loc is inclusive, .iloc is not
# get the rows from position 0 up to but not including position 3 (ie, rows 0, 1, and 2)
df.iloc[0:3]
```

```
# get the value from the row in position 3 and the column in position 2 (zero-indexed)
df.iloc[3, 6] #country_txt
```

1.2.4.4 Select/filter by value

You can subset or filter a dataframe for based on the values in its rows/columns.

```
# filter the dataframe by urban areas with more than 25 million residents
df[df['nkill'] > 30].head()
```

```
# you can chain multiple conditions together
# pandas logical operators are: | for or, & for and, ~ for not
# these must be grouped by using parentheses due to order of operations
df[['country','nkill', 'nwound']][(df['nkill'] > 200) & (df['nwound'] > 10)].head()
# columns on the left-hand side are here used to slice the resulting output
```

```
# ~ means not... it essentially flips trues to falses and vice-versa
df[['country','nkill', 'nwound']][~(df['nkill'] > 200) & (df['nwound'] > 10)]
```

1.2.5 Grouping and summarizing

```
# group by terroristic group name
groups = df.groupby('gname')
```

```
# what is the median number of people killed per event across the different groups?
groups['nkill'].median().sort_values(ascending=False)
```

```
# look at several columns' medians by group
groups[['nkill', 'nwound', 'nperps']].median()
```

```
# you can create a new dataFrame by directly passing columns between "[[ ]]", after the grou
# to do so, you also need to pass a function that can deal with the values (e.g. sum..etc)
western_europe = df[df.region_txt == 'Western Europe']
western_europe.groupby('country_txt')[['nkill', 'nwound']].sum().sort_values('nkill', ascend
```

1.2.6 Indexes

Each `DataFrame` has an index. Indexes do not have to be unique (but that would be for the best)

```
# resetting index (when loading a .csv file pandas creates an index automatically, from 0 to
df.reset_index(drop = True).sort_index().head() # this does not assign the new index though,
```

```
#this does assign the new index to your df
df = df.reset_index(drop = True).sort_index()
df.head()
```

```
# index isn't unique
df.index.is_unique
```

```
# you can set a new index
# drop -> Delete columns to be used as the new index.
# append ->  whether to append columns to existing index.
df = df.set_index('eventid', drop=True, append=False)
df.index.name = None # remove the index "name"
df.head()

# this index is not ideal, but it's the original source's id
```

1.3 Part III: Geospatial Vector data in Python

Gabriele Filomena has prepared this notebook by readapting material shared on this repository. Copyright (c) 2018, Joris Van den Bossche.

```
%matplotlib inline

import geopandas as gpd
```

1.3.1 Importing geospatial data

GeoPandas builds on Pandas types `Series` and `Dataframe`, by incorporating information about geographical space.

- `GeoSeries`: a Series object designed to store shapely geometry object
- `GeoDataFrame`: object is a pandas DataFrame that has a column with geometry (that contains a *Geoseries*)

We can use the GeoPandas library to read many of GIS file formats (relying on the `fiona` library under the hood, which is an interface to GDAL/OGR), using the `gpd.read_file` function. For example, let's start by reading a shapefile with all the countries of the world (adapted from http://www.naturalearthdata.com/downloads/110m-cultural-vectors/110m-admin-0-countries/, zip file is available in the `/data` directory), and inspect the data:

```
countries = gpd.read_file("../data/ne_countries.zip")
# or if the archive is unpacked:
# countries = gpd.read_file("../data/ne_countries.shp")
```

```
countries.head()
```

```
countries.plot()
```

We observe that:

- Using `.head()` we can see the first rows of the dataset, just like we can do with Pandas.
- There is a `geometry` column and the different countries are represented as polygons
- We can use the `.plot()` (matplotlib) method to quickly get a *basic* visualization of the data

1.3.2 What's a GeoDataFrame?

We used the GeoPandas library to read in the geospatial data, and this returned us a `GeoDataFrame`:

```
type(countries)
```

A GeoDataFrame contains a tabular, geospatial dataset:

- It has a 'geometry' column that holds the geometry information (or features in GeoJSON).
- The other columns are the **attributes** (or properties in GeoJSON) that describe each of the geometries.

Such a `GeoDataFrame` is just like a pandas `DataFrame`, but with some additional functionality for working with geospatial data: * A `geometry` attribute that always returns the column with the geometry information (returning a `GeoSeries`). The column name itself does not necessarily need to be 'geometry', but it will always be accessible as the `geometry` attribute. * It has some extra methods for working with spatial data (area, distance, buffer, intersection, …) see here, for example.

```
countries.geometry.head()
```

```
type(countries.geometry)
```

```
countries.geometry.area
```

It's still a `DataFrame`, so we have all the `pandas` functionality available to use on the geospatial dataset, and to do data manipulations with the attributes and geometry information together. For example, we can calculate the average population over all countries (by accessing the 'pop_est' column, and calling the `mean` method on it):

```
countries['pop_est'].mean()
```

```
africa = countries[countries['continent'] == 'Africa']
```

```
africa.plot();
```

The rest of the tutorial is going to assume you already know some pandas basics, but we will try to give hints for that part for those that are not familiar.

Important:

- A `GeoDataFrame` allows to perform typical tabular data analysis together with spatial operations
- A `GeoDataFrame` (or *Feature Collection*) consists of:
  - **Geometries** or **features**: the spatial objects
  - **Attributes** or **properties**: columns with information about each spatial object

1.3.3 Geometries: Points, Linestrings and Polygons

Spatial **vector** data can consist of different types, and the 3 fundamental types are:

- **Point** data: represents a single point in space.
- **Line** data ("LineString"): represented as a sequence of points that form a line.
- **Polygon** data: represents a filled area.

And each of them can also be combined in multi-part geometries (See https://shapely.readthedocs.io/en/stable/m objects for extensive overview).

For the example we have seen up to now, the individual geometry objects are Polygons:

```
print(countries.geometry[2])
```

Let's import some other datasets with different types of geometry objects.

A dateset about cities in the world (adapted from http://www.naturalearthdata.com/downloads/110m-cultural-vectors/110m-populated-places/, zip file is available in the `/data` directory), consisting of `Point` data:

```
cities = gpd.read_file("../data/ne_cities.zip")
```

```
print(cities.geometry[0])
```

And a dataset of rivers in the world (from http://www.naturalearthdata.com/downloads/50m-physical-vectors/50m-rivers-lake-centerlines/, zip file is available in the `/data` directory) where each river is a `(Multi-)LineString`:

```
rivers = gpd.read_file("../data/ne_rivers.zip")
```

```
print(rivers.geometry[0])
```

1.3.4 The `shapely` library

The individual geometry objects are provided by the `shapely` library

```
from shapely.geometry import Point, Polygon, LineString
```

```
type(countries.geometry[0])
```

To construct one ourselves:

```
p = Point(0, 0)
```

```
print(p)
```

```
polygon = Polygon([(1, 1), (2,2), (2, 1)])
```

```
polygon.area
```

```
polygon.distance(p)
```

Important:

Single geometries are represented by `shapely` objects:

- If you access a single geometry of a GeoDataFrame, you get a shapely geometry object

- Those objects have similar functionality as geopandas objects (GeoDataFrame/GeoSeries). For example:
  - `single_shapely_object.distance(other_point)` -> distance between two points
  - `geodataframe.distance(other_point)` -> distance for each point in the geodataframe to the other point

1.3.5 Plotting

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize=(15, 10))
countries.plot(ax = ax, edgecolor='k', facecolor='none')
rivers.plot(ax=ax)
cities.plot(ax=ax, color='red')
ax.set(xlim=(-20, 60), ylim=(-40, 40))
```

1.3.6 Creating GeoDataFrames (withouth specifying the CRS)

```
gpd.GeoDataFrame({
    'geometry': [Point(1, 1), Point(2, 2)],
    'attribute1': [1, 2],
    'attribute2': [0.1, 0.2]})
```

```
# Creating a GeoDataFrame from an existing dataframe
# For example, if you have lat/lon coordinates in two columns:
df = pd.DataFrame(
    {'City': ['Buenos Aires', 'Brasilia', 'Santiago', 'Bogota', 'Caracas'],
     'Country': ['Argentina', 'Brazil', 'Chile', 'Colombia', 'Venezuela'],
     'Latitude': [-34.58, -15.78, -33.45, 4.60, 10.48],
     'Longitude': [-58.66, -47.91, -70.66, -74.08, -66.86]})
```

```
gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.Longitude, df.Latitude))
gdf
```

2 Practice

Throughout the exercises in this course, we will work with several datasets about the city of Paris.

Here, we start with the following datasets:

- The administrative districts of Paris (https://opendata.paris.fr/explore/dataset/quartier_paris/): `paris_districts_utm.geojson`
- Real-time (at the moment I downloaded them ..) information about the public bicycle sharing system in Paris (vélib, https://opendata.paris.fr/explore/dataset/stations-velib-disponibilites-en-temps-reel/information/): `data/paris_bike_stations_mercator.gpkg`

Both datasets are provided as spatial datasets using a GIS file format.

Excercise 1:

We will start by exploring the bicycle station dataset (available as a GeoPackage file: `data/paris_bike_stations_mercator.gpkg`)

- Read the stations datasets into a GeoDataFrame called `stations`.
- Check the type of the returned object
- Check the first rows of the dataframes. What kind of geometries does this datasets contain?
- How many features are there in the dataset?

Hints

- Use `type(..)` to check any Python object type
- The `gpd.read_file()` function can read different geospatial file formats. You pass the file name as first argument.
- Use the `.shape` attribute to get the number of features

Exercise 2:

- Make a quick plot of the `stations` dataset.
- Make the plot a bit larger by setting the figure size to (12, 6) (hint: the `plot` method accepts a `figsize` keyword).

Exercise 3:

Next, we will explore the dataset on the administrative districts of Paris (available as a GeoJSON file: `../data/paris_districts_utm.geojson`)

- Read the dataset into a GeoDataFrame called `districts`.
- Check the first rows of the dataframe. What kind of geometries does this dataset contain?
- How many features are there in the dataset? (hint: use the `.shape` attribute)
- Make a quick plot of the `districts` dataset (set the figure size to (12, 6)).

Exercise 4:

What are the largest districts (biggest area)?

- Calculate the area of each district.
- Add this area as a new column to the `districts` dataframe.
- Sort the dataframe by the area column from largest to smallest values (descending).

Hints

- Adding a column can be done by assigning values to a column using the same square brackets syntax: `df['new_col'] = values`
- To sort the rows of a DataFrame, use the `sort_values()` method, specifying the colum to sort on with the `by='col_name'` keyword. Check the help of this method to see how to sort ascending or descending.

2.1 Part IV: Coordinate reference systems & Projections

Gabriele Filomena has prepared this notebook by readapting material shared on this repository.


```
countries = gpd.read_file("../data/ne_countries.zip")
cities = gpd.read_file("../data/ne_cities.zip")
rivers = gpd.read_file("../data/ne_rivers.zip")
```

2.1.1 Coordinate reference systems

Up to now, we have used the geometry data with certain coordinates without further wondering what those coordinates mean or how they are expressed.

> The **Coordinate Reference System (CRS)** relates the coordinates to a specific location on earth.

For an in-depth explanation, see https://docs.qgis.org/2.8/en/docs/gentle_gis_introduction/coordinate_referer

2.1.1.1 Geographic coordinates

Degrees of latitude and longitude.

E.g. 48°51 N, 2°17 E

The most known type of coordinates are geographic coordinates: we define a position on the globe in degrees of latitude and longitude, relative to the equator and the prime meridian. With this system, we can easily specify any location on earth. It is used widely, for example in GPS. If you inspect the coordinates of a location in Google Maps, you will also see latitude and longitude.

Attention!

in Python we use (lon, lat) and not (lat, lon)

- Longitude: [-180, 180]{{1}}
- Latitude: [-90, 90]{{1}}

2.1.2 Projected coordinates

`(x, y)` coordinates are usually in meters or feet

Although the earth is a globe, in practice we usually represent it on a flat surface: think about a physical map, or the figures we have made with Python on our computer screen. Going from the globe to a flat map is what we call a *projection*.

We project the surface of the earth onto a 2D plane so we can express locations in cartesian x and y coordinates, on a flat surface. In this plane, we then typically work with a length unit such as meters instead of degrees, which makes the analysis more convenient and effective.

However, there is an important remark: the 3 dimensional earth can never be represented perfectly on a 2 dimensional map, so projections inevitably introduce distortions. To minimize such errors, there are different approaches to project, each with specific advantages and disadvantages.

Some projection systems will try to preserve the area size of geometries, such as the Albers Equal Area projection. Other projection systems try to preserve angles, such as the Mercator projection, but will see big distortions in the area. Every projection system will always have some distortion of area, angle or distance.

Projected size vs actual size (Mercator projection):

2.1.3 Coordinate Reference Systems in Python / GeoPandas

A GeoDataFrame or GeoSeries has a `.crs` attribute which holds (optionally) a description of the coordinate reference system of the geometries:

```
countries.crs
```

For the `countries` dataframe, it indicates that it uses the EPSG 4326 / WGS84 lon/lat reference system, which is one of the most used for geographic coordinates.

It uses coordinates as latitude and longitude in degrees, as can you be seen from the x/y labels on the plot:

```
countries.plot()
```

The `.crs` attribute returns a `pyproj.CRS` object. To specify a CRS, we typically use some string representation:

- **EPSG code** Example: `EPSG:4326` = WGS84 geographic CRS (longitude, latitude)

For more information, see also http://geopandas.readthedocs.io/en/latest/projections.html.

2.1.3.1 Transforming to another CRS

We can convert a GeoDataFrame to another reference system using the `to_crs` function.

For example, let's convert the countries to the World Mercator projection (http://epsg.io/3395):

```
# remove Antartica, as the Mercator projection cannot deal with the poles
countries = countries[(countries['name'] != "Antarctica")]
countries_mercator = countries.to_crs(epsg=3395)  # or .to_crs("EPSG:3395")
countries_mercator.plot()
```

Note the different scale of x and y.

2.1.3.2 Why using a different CRS?

There are sometimes good reasons you want to change the coordinate references system of your dataset, for example:

- Different sources with different CRS -> need to convert to the same crs.

- Different countries/geographical areas with different CRS.
- Mapping (distortion of shape and distances).
- Distance / area based calculations -> ensure you use an appropriate projected coordinate system expressed in a meaningful unit such as meters or feet (**not degrees!**).

Important:

All the calculations (e.g. distance, spatial operations, etc.) that take place in `GeoPandas` and `Shapely` assume that your data is represented in a 2D cartesian plane, and thus the result of those calculations will only be correct if your data is properly projected.

2.2 Practice

Again, we will go back to the Paris datasets. Up to now, we provided the datasets in an appropriate projected CRS for the exercises. But the original data were actually using geographic coordinates. In the following exercises, we will start from there.

Going back to the Paris districts dataset, this is now provided as a GeoJSON file (`"../data/paris_districts.geojson"`) in geographic coordinates.

For converting the layer to projected coordinates, we will use the standard projected CRS for France is the RGF93 / Lambert-93 reference system, referenced by the `EPSG:2154` number.

Exercise: Projecting a GeoDataFrame

- Read the districts datasets (`../data/paris_districts.geojson"`) into a GeoDataFrame called `districts`.
- Look at the CRS attribute of the GeoDataFrame. Do you recognize the EPSG number?
- Make a plot of the `districts` dataset.
- Calculate the area of all districts.
- Convert the `districts` to a projected CRS (using the `EPSG:2154` for France). Call the new dataset `districts_RGF93`.
- Make a similar plot of `districts_RGF93`.
- Calculate the area of all districts again with `districts_RGF93` (the result will now be expressed in m^2).

Hints

- The CRS information is stored in the `.crs` attribute of a GeoDataFrame.
- Making a simple plot of a GeoDataFrame can be done with the `.plot()` method.
- Converting to a different CRS can be done with the `.to_crs()` method, and the CRS can be specified as an EPSG number using the `epsg` keyword.

3 Static Maps in Python

The **Lecture slides** can be found here.

This **lab**'s notebook can be downloaded from here.

3.1 Part I: Basic Maps

In this session, we will use the libraries `matplotlib` and `contextily` to plot the information represented into different `GeoDataFrames`. We will look into plotting `Point`, `LineString` and `Polygon GeoDataFrames`. Most of the plots here are rather ugly but, at this point, the goal is to get familiar with the parameters of the plot function and what can be done with them.

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import geopandas as gpd
import pandas as pd
import osmnx as ox
import contextily as ctx
import seaborn as sns
```

3.1.1 Plotting Points

Load the data of terrorist attacks 1970-2020 and choose a country. Germany is used as a case study here but feel free to change the country. If you do so, also change the `crs` (see https://epsg.io).

```
attacks = pd.read_csv("../data/GTD_2022.csv", low_memory = False)
```

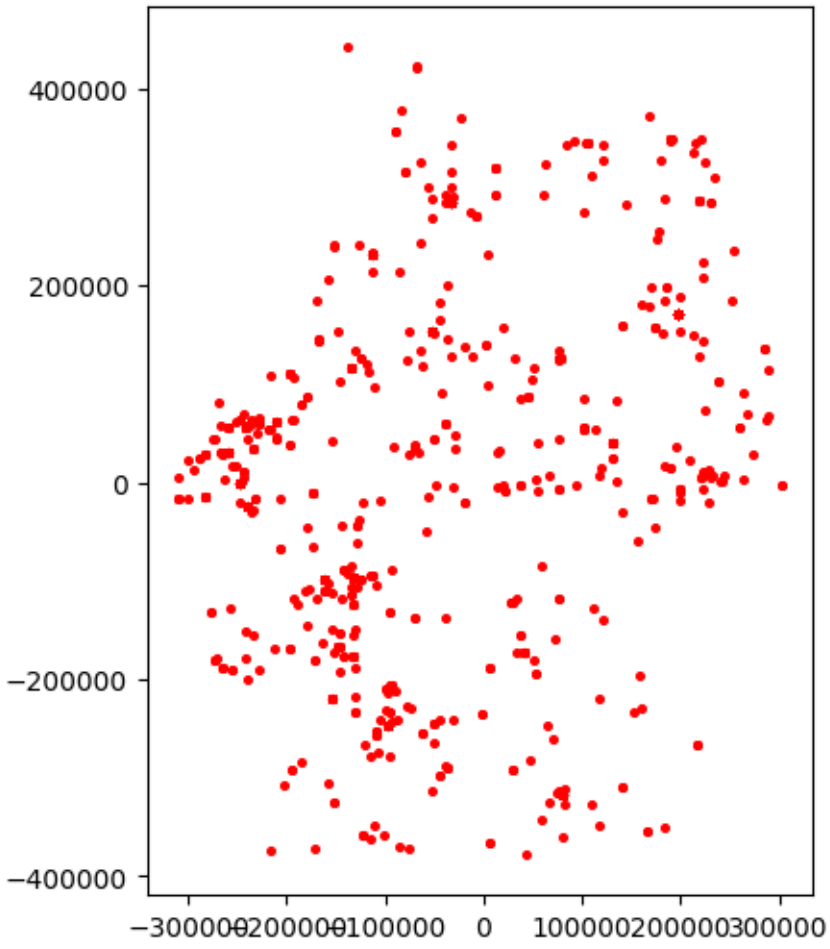
Creating the `GeoDataFrame` from the `DataFrame`

```
germany = ['West Germany (FRG)', 'Germany', 'East Germany (GDR)'] # Germany was split till 19
df = attacks[attacks.country_txt.isin(germany)].copy()

# Uncomment the lines below for other countries that haven't changed their denominations/boun
# country = 'France'
# df = attacks[attacks.country_txt == country].copy()#
wgs = 'EPSG:4326'
germany_crs = 'EPSG:4839'
gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.longitude, df.latitude), crs = wgs)
gdf = gdf[~gdf.geometry.is_empty] # remove empty geometries
gdf.to_file("../data/germany.shp")
gdf = gdf.to_crs(germany_crs)
```

```
C:\Users\gfilo\AppData\Local\Temp\ipykernel_700\3815068564.py:11: UserWarning: Column names l
  gdf.to_file("data/germany.shp")
```

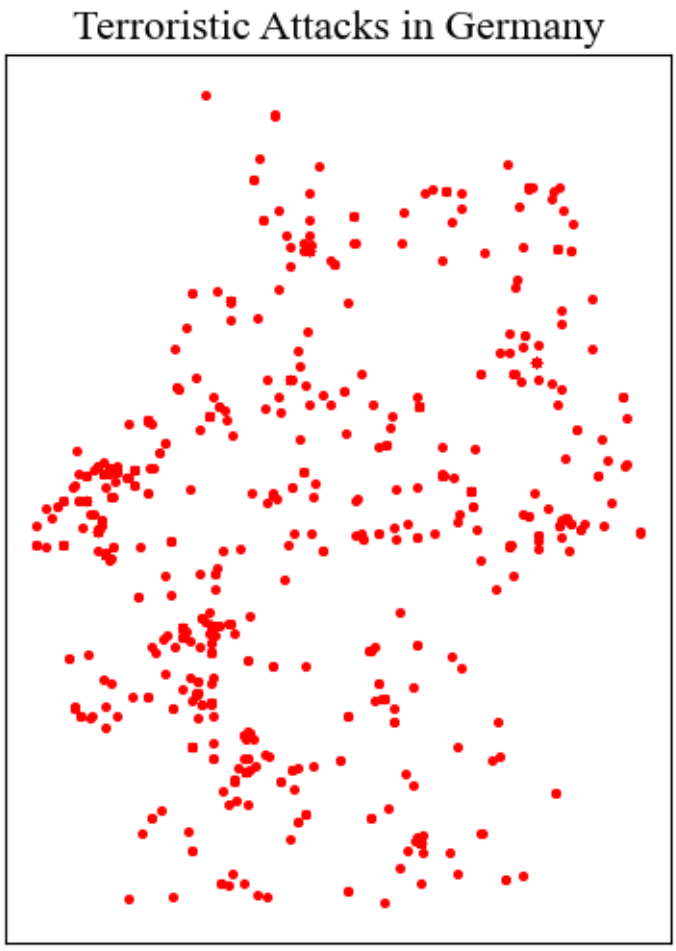
Basic plotting

```
# prepare the axis and coordinate
nr_rows = 1
nr_cols = 1
fig, ax = plt.subplots(nr_cols, nr_rows, figsize=(8, 6))
gdf.plot(ax=ax, color='red', markersize=7)
```



Slightly improving the plot:

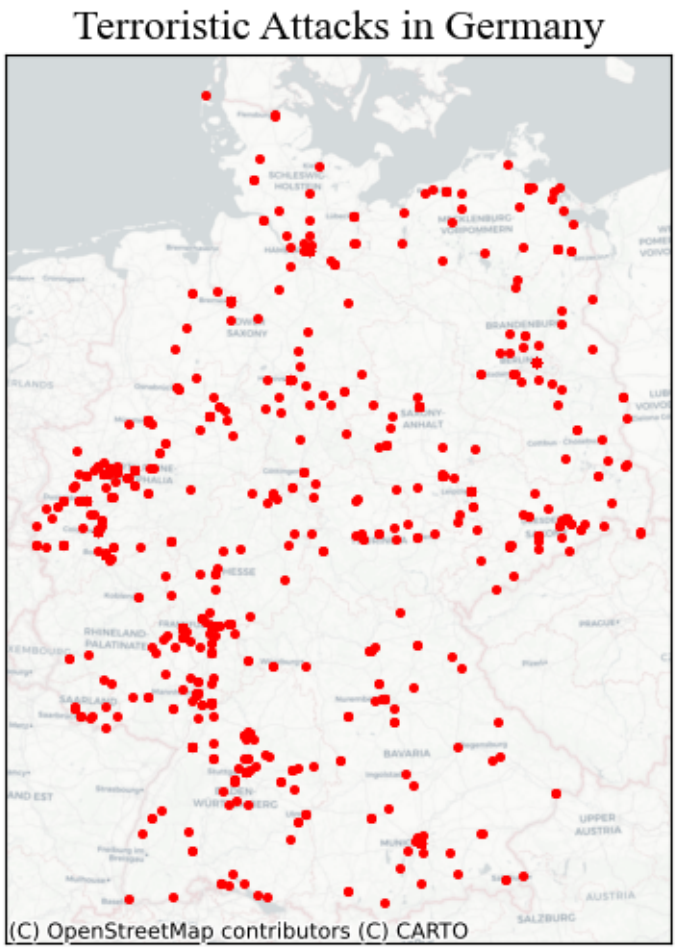
```
# removing ticks
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.tick_params(axis= 'both', which= 'both', length=0)
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Terroristic Attacks in Germany", **title_parameters)
fig
```



3.1.1.1 Adding some context: Base Maps with `Contextily`

see providers and options here https://xyzservices.readthedocs.io/en/stable/introduction.html

```
source = ctx.providers.CartoDB.Positron
ctx.add_basemap(ax, crs= gdf.crs.to_string(), source= source)
# replot
fig
```




```
<Figure size 640x480 with 0 Axes>
```

3.1.1.2 Parameters specific to `Point` in the `plot` method

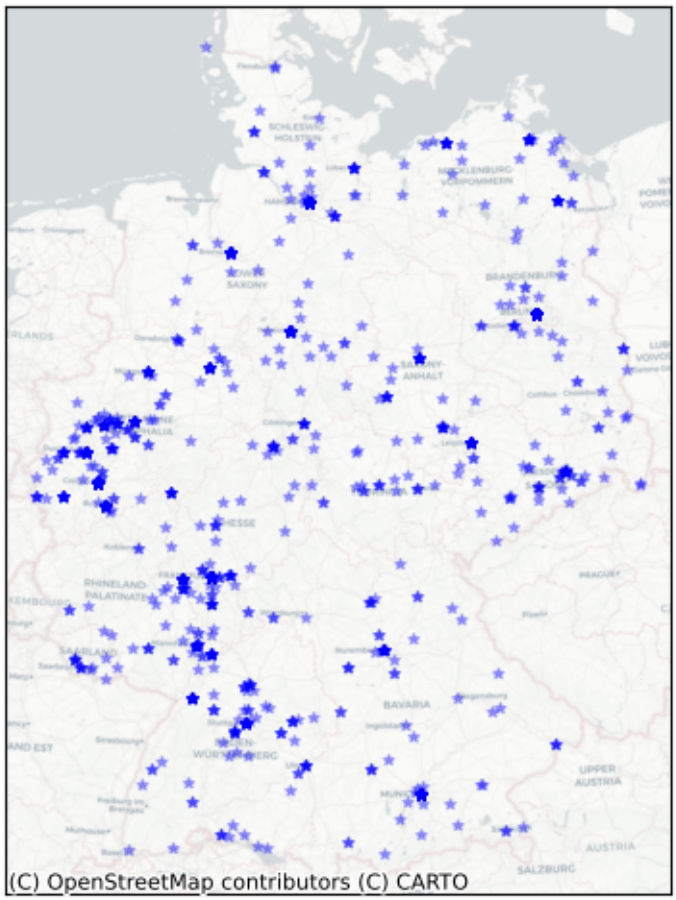
- `markersize`: numerical value (for now)
- `marker`: see https://matplotlib.org/stable/api/markers_api.html

3.1.1.2.1 Other properties, shape independent:

- `color`: https://matplotlib.org/3.1.0/gallery/color/named_colors.html
- `alpha`: regulates transparency of the shape: 0 to 1

```
# first, let's make a function

def ax_ticks_off(ax):
    ax.xaxis.set_ticklabels([])
    ax.yaxis.set_ticklabels([])
    ax.tick_params(axis= 'both', which= 'both', length=0)
```

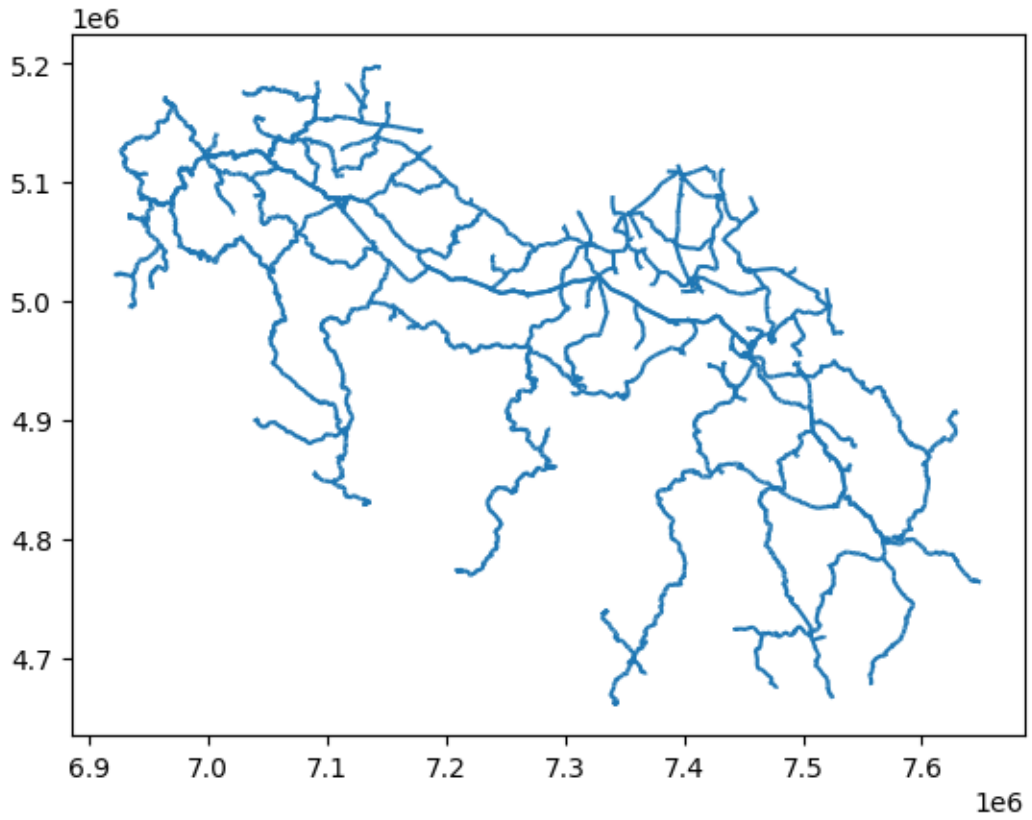
```
# prepare the axis and coordinate
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf.plot(ax=ax, markersize = 15, color = 'blue', marker = '*', alpha = 0.3)
ctx.add_basemap(ax, crs= gdf.crs.to_string(), source= source)
ax_ticks_off(ax)
```



3.1.2 Plotting LineStrings

Let's import railway tracks in the Western Balkans (Slovenia, Croatia, Bosnia & Herzegovina, Montenegro, Serbia, Kosovo)

```
wb_crs = 'EPSG:31277'
lines_gdf = gpd.read_file("..\data\wb_railways.shp")
lines_gdf.plot()
```




```
# prepare the plot
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
lines_gdf.plot(ax=ax, linewidth = 0.8, color = 'blue', alpha = 1)
ctx.add_basemap(ax, crs= lines_gdf.crs.to_string(), source = ctx.providers.Esri.WorldGrayCan
ax_ticks_off(ax)
ax.set_title("Railway infrastructure in the West Balkans", **title_parameters) #parameters a
```

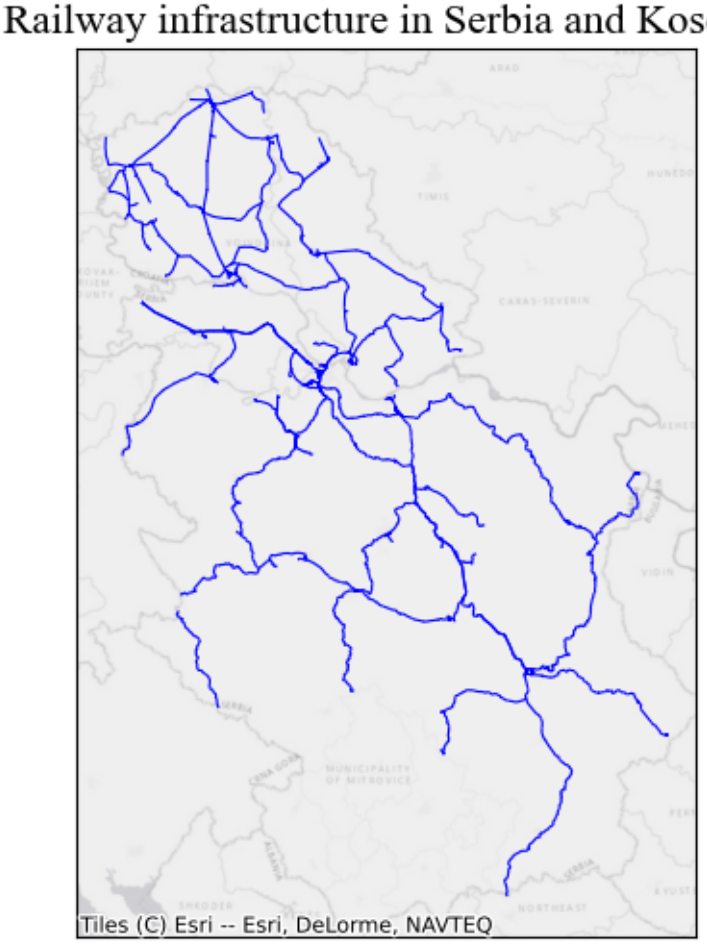
```
Text(0.5, 1.0, 'Railway infrastructure in the West Balkans')
```

One can also filter prior to plotting, based on the columns in the GeoDataFrame. First we download Serbia's Boundary with `OSMNX`, more on that later on. Then we filter `lines_gdf` with a `within` operation.

```
serbia = ox.geocode_to_gdf('Serbia')
serbia = serbia.to_crs(wb_crs)
serbia_lines = lines_gdf[lines_gdf.geometry.within(serbia.iloc[0].geometry)].copy() #there's
```

```
# prepare the plot
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
serbia_lines.plot(ax=ax, linewidth = 0.8, color = 'blue', alpha = 1)
ctx.add_basemap(ax, crs= lines_gdf.crs.to_string(), source = ctx.providers.Esri.WorldGrayCan
ax_ticks_off(ax)
ax.set_title("Railway infrastructure in Serbia and Kosovo", **title_parameters) #parameters
```

```
Text(0.5, 1.0, 'Railway infrastructure in Serbia and Kosovo')
```



3.1.2.1 Parameters specific to `LineString`:

- `linewidth`: numerical value (for now).
- `capstyle`: controls how Matplotlib draws the corners where two different line segments meet. See https://matplotlib.org/stable/gallery/lines_bars_and_markers/capstyle.html
- `joinstyle`': controls how Matplotlib draws the corners where two different line segments meet. https://matplotlib.org/stable/gallery/lines_bars_and_markers/joinstyle.html

```
# prepare the plot
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
serbia_lines.plot(ax=ax, linewidth = 0.9, color = 'black', alpha = 1, capstyle = 'round', jo
ax.set_axis_off() # we don't need the ticks function
ax.set_title("Railway infrastructure in Serbia", **title_parameters) #parameters as above
```

```
Text(0.5, 1.0, 'Railway infrastructure in Serbia')
```

Railway infrastructure in Serbia

3.1.3 Plotting Polygons

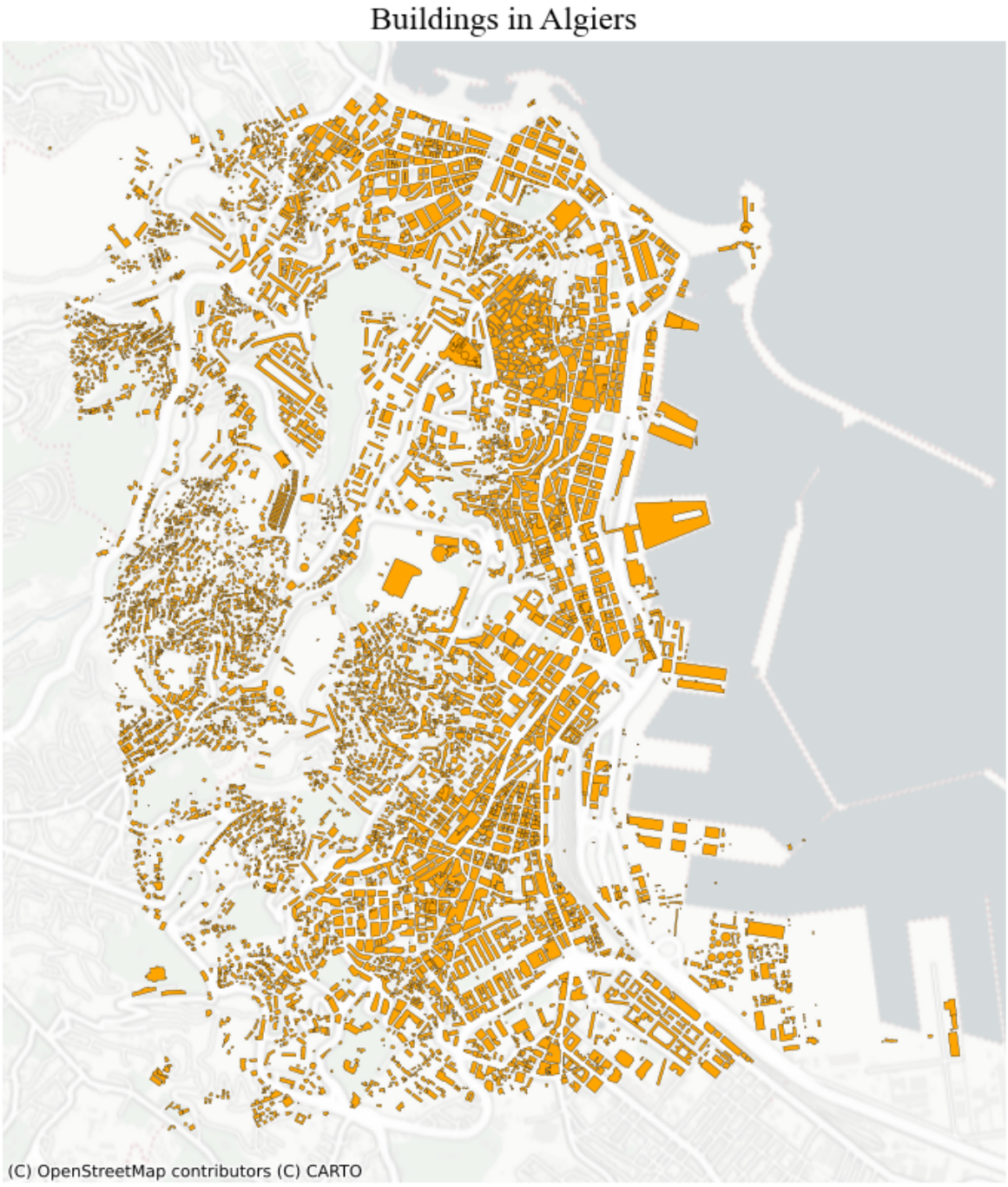
We are again using `OSMNX` to download data from `OpenStreetMap` automatically. In this case, we will get building footprints from the city of Algiers in Alageria.

3.1.3.1 Parameter specific to `Polygon`:

- `edgecolor`: the outline of the polygon, by default = `None` (often better).
- `linewidth`: the width of the outline of the polygon.

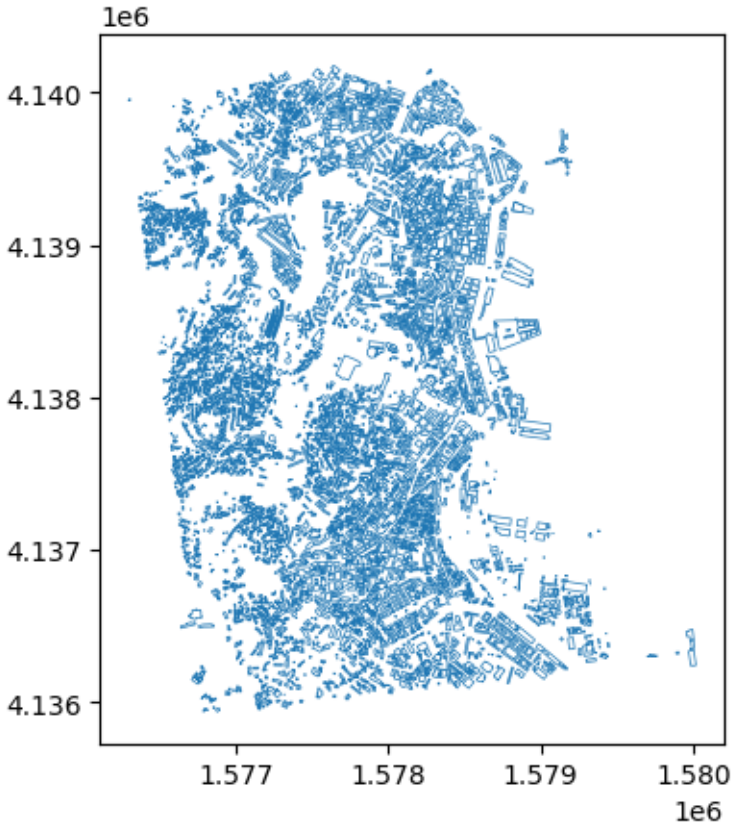
```
algeria_crs = 'EPSG:30729'
tags = {"building": True} #OSM tags
buildings = ox.features_from_address("Algiers, Algeria", tags = tags, dist = 2000)
buildings = buildings.reset_index()
 # sometimes building footprints are represented by Points, let's disregard them
buildings = buildings[(buildings.geometry.geom_type == 'Polygon') | (buildings.geometry.geom
buildings = buildings.to_crs(algeria_crs)
```

```
fig, ax = plt.subplots(1, 1, figsize=(15, 10))
ax.set_title("Buildings in Algiers", **title_parameters)
ax.set_axis_off() # we don't need the ticks function
buildings.plot(ax=ax, color = 'orange', edgecolor = 'black', lw = 0.2)
source = ctx.providers.CartoDB.PositronNoLabels
ctx.add_basemap(ax, crs= buildings.crs.to_string(), source= source)
```



For polygons, you can also plot just the boundaries of the geometries by:

```
buildings.boundary.plot(lw = 0.5)
```



3.1.4 Plotting more than one layer together

Let's also download roads for Algiers

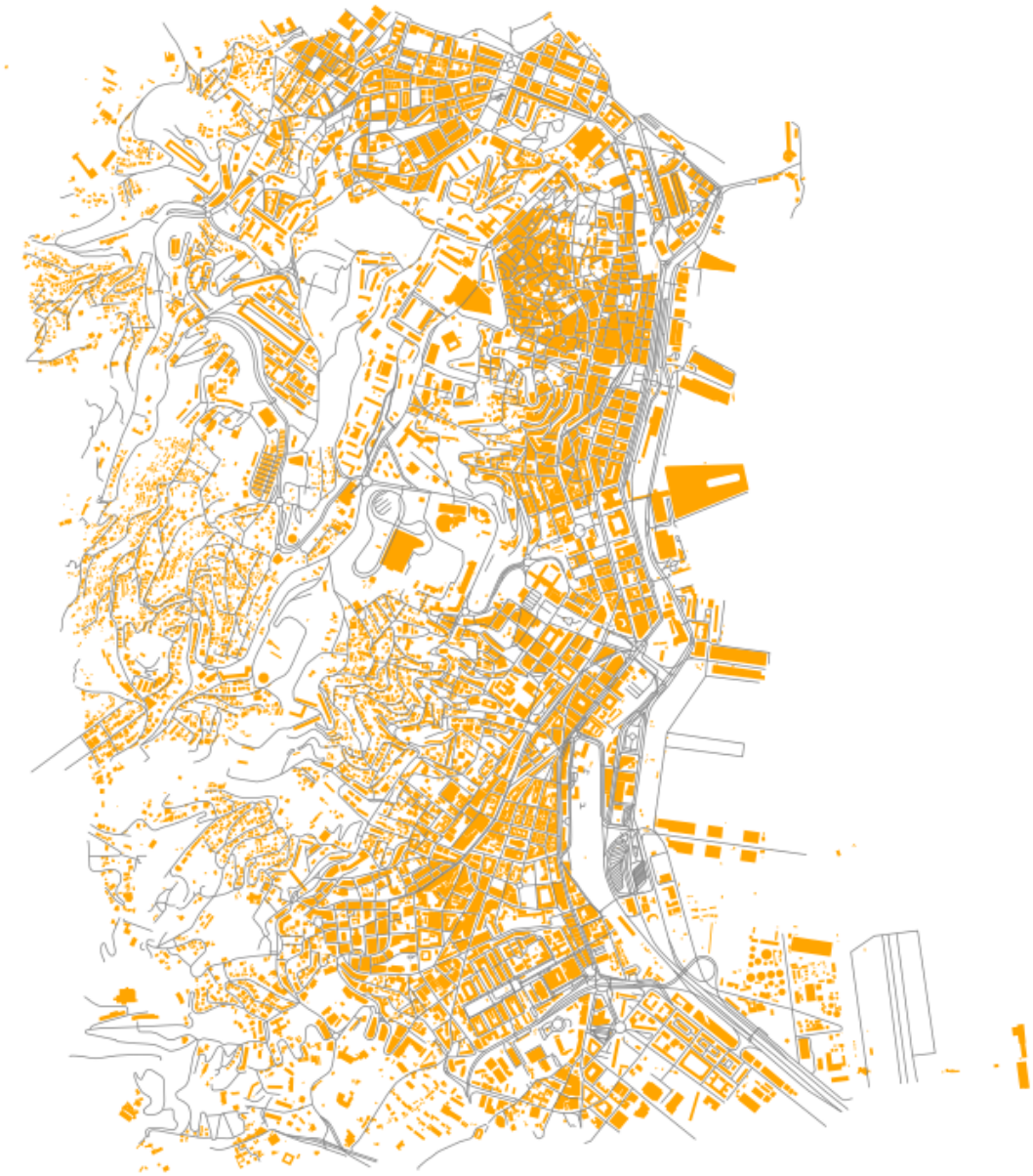
```
tags = {"highway": True} #OSM tags
roads = ox.features_from_address("Algiers, Algeria", tags = tags, dist = 2000)
roads = roads.reset_index()
roads = roads.to_crs(algeria_crs)
 # sometimes building footprints are represented by Points, let's disregard them
roads = roads[roads.geometry.geom_type == 'LineString']
```

And plot everything togehter. It's important to keep in mind that the last layer is always rendered on top of the others. In other words, they may cover the previous ones.

However, you can prevent this by passing arguments to the parameter `zorder` in the `plot` method. The layer with the higher zorder value will be plotted on top.

```
fig, ax = plt.subplots(1, 1, figsize=(15, 10))
ax.set_title("Buildings and Roads in Algiers", **title_parameters)
ax.set_axis_off() # we don't need the ticks function
# only roads within the extent of the buildings layer
roads[roads.geometry.within(buildings.unary_union.envelope)].plot(ax=ax, color = 'grey', lw =
buildings.plot(ax=ax, color = 'orange')
```

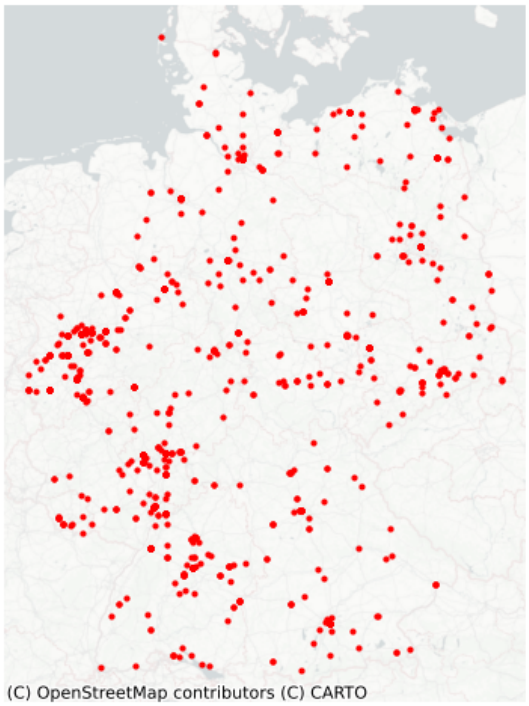
Buildings and Roads in Algiers

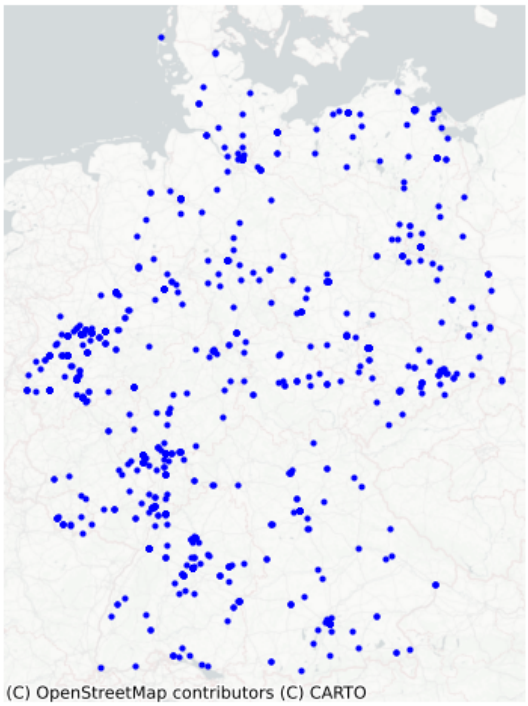


3.1.5 Sub-plots

To obtain multiple sub-plots, we manipulate the `nrows`, `ncols` parameters. We can use this approach to: * Plot the same layer with different properties.

```
fig, axes = plt.subplots(1, 2, figsize=(10, 6))
colors = ['red', 'blue']

for n, ax in enumerate(axes):
    gdf.plot(ax=ax, markersize = 4, color = colors[n])
    ax.set_axis_off()
    ctx.add_basemap(ax, crs= gdf.crs.to_string(), source= source)
```

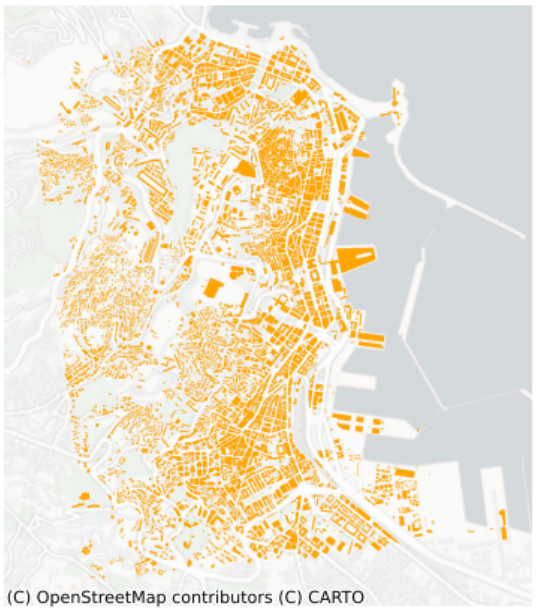




- Plot different layers.

```
fig, axes = plt.subplots(1, 2, figsize=(10, 6))
gdfs = [buildings, roads]
colors = ['orange', 'grey']

buildings.plot(ax=axes[0], color = 'orange', edgecolor = 'none')
roads.plot(ax=axes[1], color = 'gray', lw = 0.5)
```

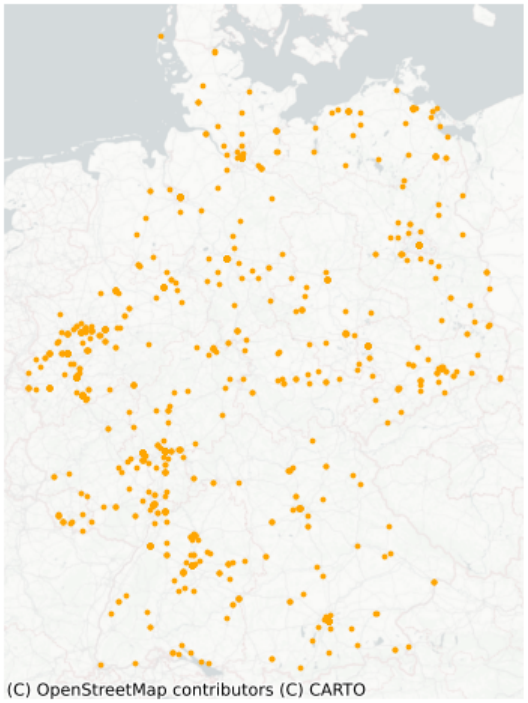
```
for ax in axes:
    ax.set_axis_off()
    ctx.add_basemap(ax, crs= buildings.crs.to_string(), source = source)
```

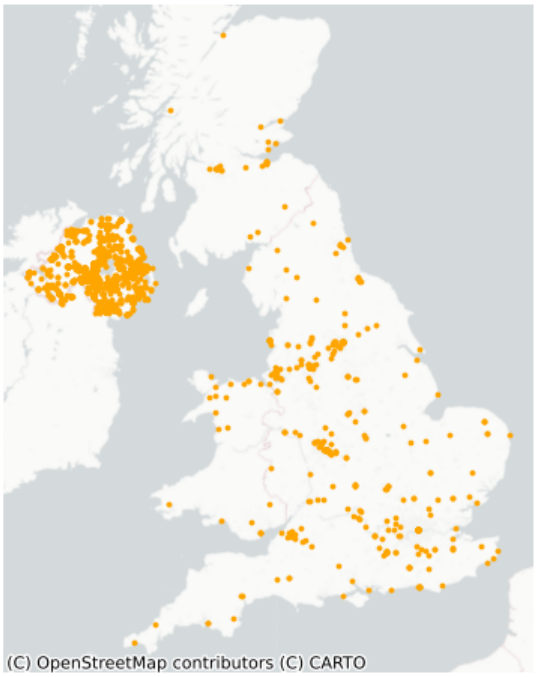


- Analyse phenomena across different geographical areas. For example, terrorism in Germany and in the UK.

```
# let's prepare the gdf for the UK
df_uk = attacks[attacks.country_txt == 'United Kingdom'].copy()
uk_crs = 'EPSG:27700'
gdf_uk = gpd.GeoDataFrame(df_uk, geometry=gpd.points_from_xy(df_uk.longitude, df_uk.latitude
gdf_uk = gdf_uk.to_crs(uk_crs)
```

```
fig, axes = plt.subplots(1, 2, figsize=(10, 6))
gdfs = [gdf, gdf_uk]

for n, ax in enumerate(axes):
    gdf_tmp = gdfs[n]
    gdf_tmp.plot(ax=ax, color = 'orange', markersize = 3)
    ax.set_axis_off()
    ctx.add_basemap(ax, crs= gdf_tmp.crs.to_string(), source= source)
```





Exercise:

- Think about the plots above and how they could be improved.
- Copy and paste the code and execute the functions playing with the different parameters.
- Produce a neat map using the `GeoDataFrame`s available in this notebook or the ones employed in the previous sessions, making use of the elements/parameters discussed here.
- Try out different tiles for the basemap to familiarise yourself with what's available.

3.2 Part II: Choropleth Mapping

```
import geoplot.crs as gcrs
import geoplot as gplt
```

Data

For this second part of the tutorial, we will use some data at the municipality level for Serbia. The data contains information regarding poverty level, average income, population and tourism. The data is taken from https://data.stat.gov.rs/?caller=SDDB&languageCode=en-US and

can be associated to the polygons representing the administrative boundaries of the municipalities. These boundaries can be found here https://data.humdata.org/dataset/geoboundaries-admin-boundaries-for-serbia?force_layout=desktop. While most of the data refers to 2023, the admin boundaries file traces back to 2017. Thus, it may contain obsolete information (few changes may occur).

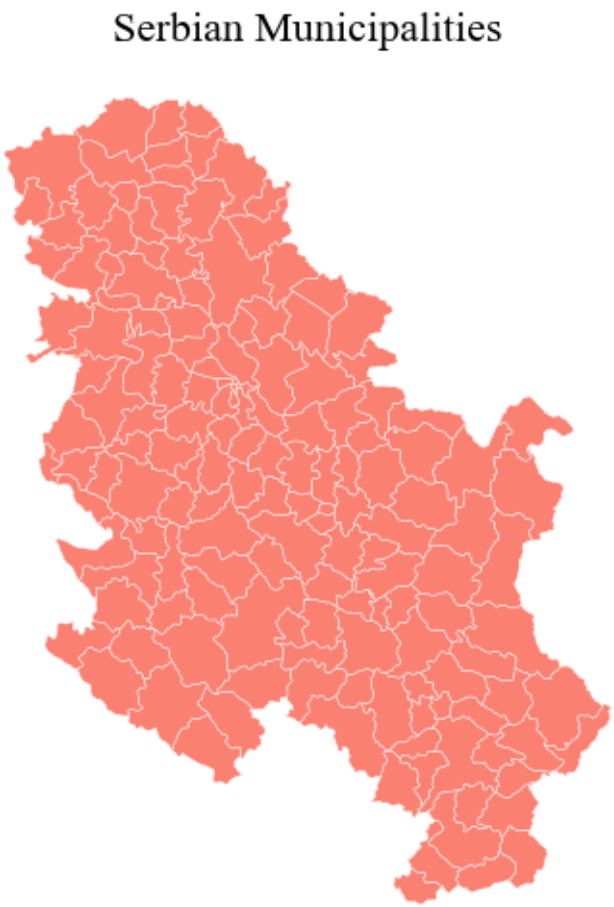
Later on, we will go back to the terrorism dataset.

```
# This will be different on your computer and will depend on where
# you have downloaded the files
serbia_crs = 'EPSG:31277'
wgs = 'EPSG:4326'
serbia_admin = gpd.read_file('../data/serbia_admin.shp')
serbia_admin.set_index('townID', inplace = True, drop = True)
serbia_admin = serbia_admin.to_crs(serbia_crs)
```

Let's plot the `GeoDataFrame` following the last session's steps.

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
serbia_admin.plot(ax = ax, color = 'salmon', linewidth = 0.3, edgecolor = 'white')
ax.set_axis_off()
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Serbian Municipalities", **title_parameters) #parameters as above
```

```
Text(0.5, 1.0, 'Serbian Municipalities')
```



The we load the data and merge it into the `GeoDataFrame`, before getting rid of municipalities that do not have a corresponding shape/record in the GeoDataFrame (probably the result of changes in the national subdivisions).

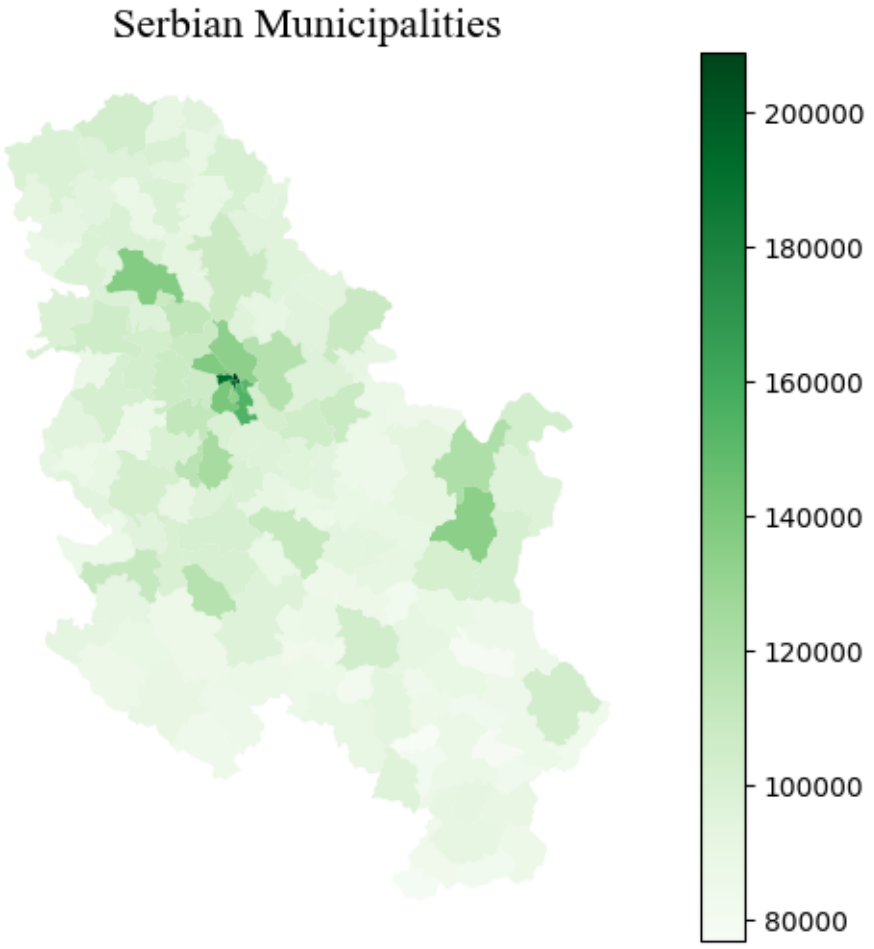
```
data = pd.read_csv("../data/serbia_data.csv") #some slavic characters
data.drop('name_en', axis = 1, inplace = True)
serbia_admin = pd.merge(serbia_admin, data, left_on = "townID", right_on = "id")
serbia_admin = serbia_admin[serbia_admin.id.notna()]
serbia_admin['id'] = serbia_admin['id'].astype('int64')
serbia_admin.head()

#let's save the so-obtained gdf for later (encoding for dealing with slavic characters).
serbia_admin.to_file("../data/serbia_data.shp", encoding='utf-8')
```

Creating a choropleth map is rather straightforward and can ben done by using few other

parameters. Reflect on what you see and whether the map below is informative.

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
serbia_admin.plot(ax = ax, column = 'gross', linewidth = 0.3, cmap = 'Greens', legend = True)
ax.set_axis_off()
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Serbian Municipalities", **title_parameters) #parameters as above
```

```
Text(0.5, 1.0, 'Serbian Municipalities')
```



3.2.1 Choropleth Maps for Numerical Variables

We are essentially using the same approach employed for creating basic maps, the method `plot`, but we now need to pass arguments to some new parameters to specify which column

is to be represented and how. As an optional argument, one can set legend to `True` and the resulting figure will include a colour bar.

- `column`: the name of the column representing the variable that we want to use to colour-code our shapes.
- `scheme`: the scheme used to colour the shapes based on the variable values.
- `cmap`: the colormap used to show variation.

3.2.1.1 Colormaps

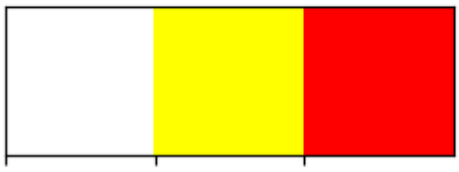
Built-in colour maps can be found here https://matplotlib.org/stable/gallery/color/colormap_reference.html. However one can create new ones as follows from a list of colours:

```
from seaborn import palplot
from matplotlib.colors import LinearSegmentedColormap

colors = [(0.00, 0.00, 0.00,1), (0.248, 0.0271, 0.569, 1), (0.0311, 0.258, 0.646,1),
            (0.019, 0.415, 0.415,1), (0.025, 0.538, 0.269,1), (0.0315, 0.658, 0.103,1),
            (0.331, 0.761, 0.036,1),(0.768, 0.809, 0.039,1), (0.989, 0.862, 0.772,1),
            (1.0, 1.0, 1.0)]
palplot(colors)
```

```
kindlmann = LinearSegmentedColormap.from_list('kindlmann', colors)
```

or from colour names:

```
colors = ["white", "yellow", "red"]
palplot(colors)
```

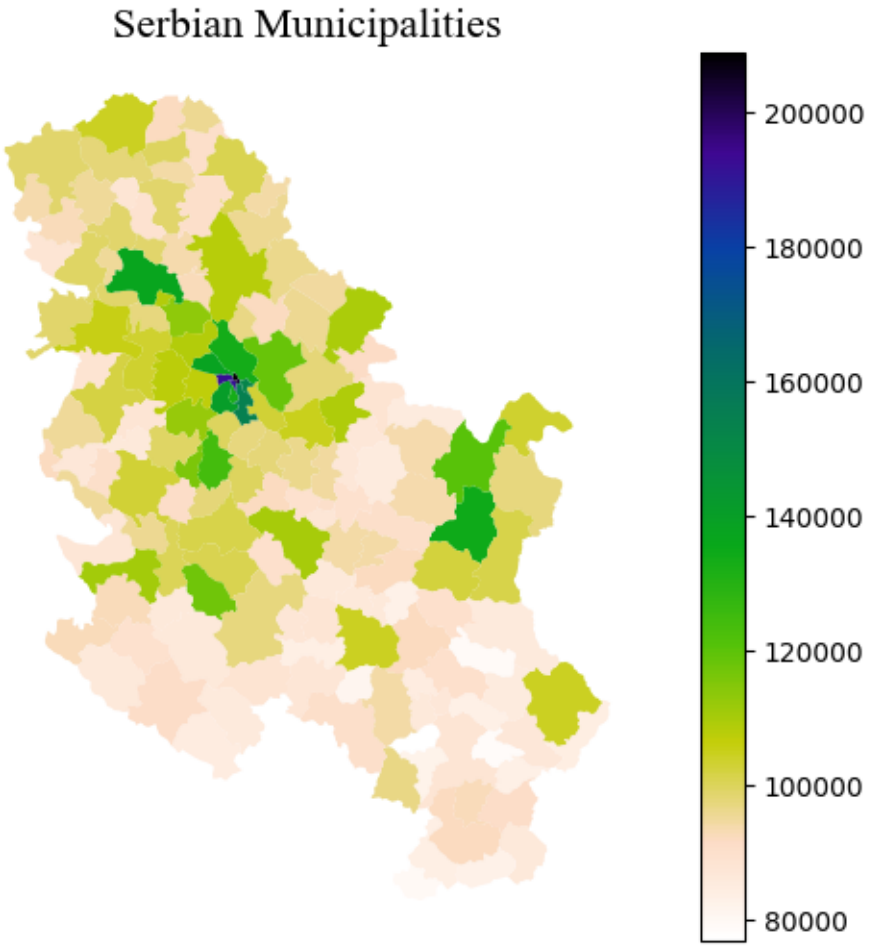


Let's try a new colormap and let's also set a number of classes to divide the data in, through the parameter `k`.

```
white_to_red = LinearSegmentedColormap.from_list("name", ["yellow","red"])
```

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
serbia_admin.plot(ax = ax, column = 'gross', linewidth = 0.3, cmap = kindlmann.reversed(), le
ax.set_axis_off()
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Serbian Municipalities", **title_parameters) #parameters as above
```

```
Text(0.5, 1.0, 'Serbian Municipalities')
```



With `GeoPandas`, when you use the `plot` method with `legend=True` the type of legend that appears depends on the data being visualized:

- Continuous Data: For columns with continuous data (like population estimates, temperatures, etc.), a colour bar is generated as the legend. This color bar represents a range of values with a gradient, indicating how data values correspond to colours on the map.
- Categorical Data: For columns with categorical data (like country names, types of land use, etc.), if you specify `legend=True`, `GeoPandas` will try to create a legend that categorizes these distinct values with different colours. However, creating legends for categorical data is not as straightforward as with continuous data and might require additional handling for a clear and informative legend (see below).

3.2.1.2 Scheme

It is important to keep in mind that choropleth maps strongly depend on the scheme that it is passed (or the default one) to classify the data in groups. The plot above only shows one municipality coloured in dark blue.

Look at the following plots and how three different classifiers produce different results for the same data.

Refer to https://geopandas.org/en/stable/gallery/choropleths.html and https://geographicdata.science/book/no for further details

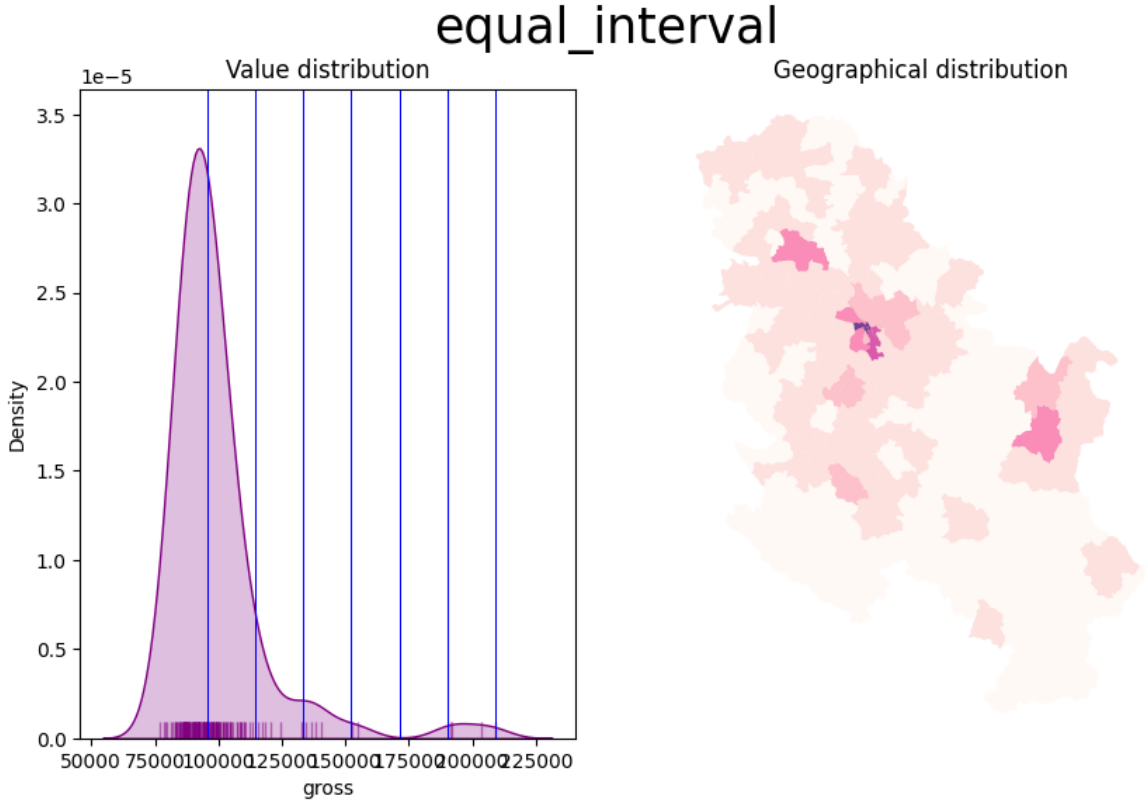
```
# Function for plotting the map and the distribution of the value in bins

from mapclassify import Quantiles, EqualInterval, FisherJenks

def plot_scheme(gdf, column, scheme, figsize=(10, 6)):
    '''
    Arguments
    ---------
    gdf: GeoDataFrame
        The GeoDataFrame to plot
    column: str
        Variable name
    scheme: str
        Name of the classification scheme to use
    figsize: Tuple
        [Optional. Default = (10, 6)] Size of the figure to be created.

    '''
    schemes = {'equal_interval': EqualInterval, 'quantiles': Quantiles, 'fisher_jenks': Fishe
    classification = schemes[scheme](gdf[column], k=7)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=figsize)
    # KDE
```

```
    sns.kdeplot(gdf[column], fill=True, color='purple', ax=ax1)
    sns.rugplot(gdf[column], alpha=0.5, color='purple', ax=ax1)
    for cut in classification.bins:
        ax1.axvline(cut, color='blue', linewidth=0.75)
    ax1.set_title('Value distribution')
    # Map
    p = gdf.plot(column=column, scheme=scheme, alpha=0.75, k=7, cmap='RdPu', ax=ax2, linewid
    ax2.axis('equal')
    ax2.set_axis_off()
    ax2.set_title('Geographical distribution')
    fig.suptitle(scheme, size=25)
    plt.show()
```

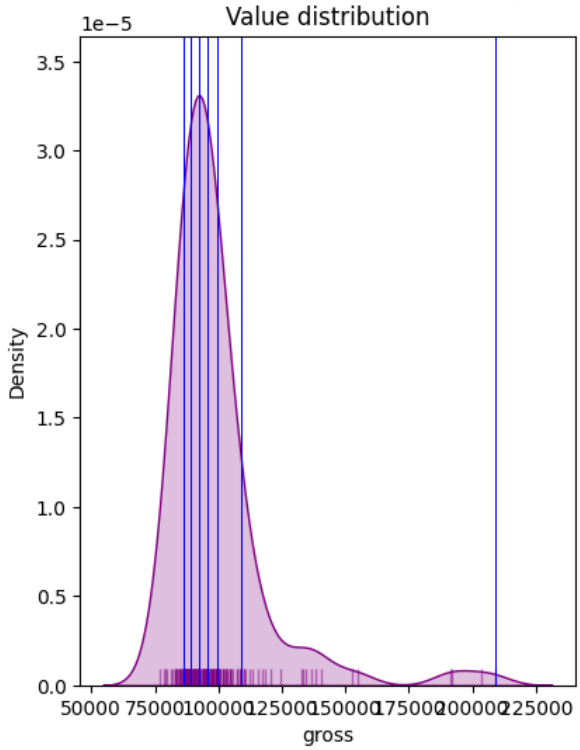
- The *Equal intervals* method splits the range of the distribution, the difference between the minimum and maximum value, into equally large segments and to assign a different colour to each of them according to a palette that reflects the fact that values are ordered.
- To obtain a more balanced classification, one can use the *Quantiles* scheme. This assigns the same amount of values to each bin: the entire series is laid out in order and break points are assigned in a way that leaves exactly the same amount of observations between each of them. This "observation-based" approach contrasts with the "value-based" method of equal intervals and, although it can obscure the magnitude of extreme values, it can be more informative in cases with skewed distributions.
- Amongst many other, the *Fisher Jenks* dynamically minimises the sum of the absolute deviations around class medians. The Fisher-Jenks algorithm is guaranteed to produce an optimal classification for a prespecified number of classes.

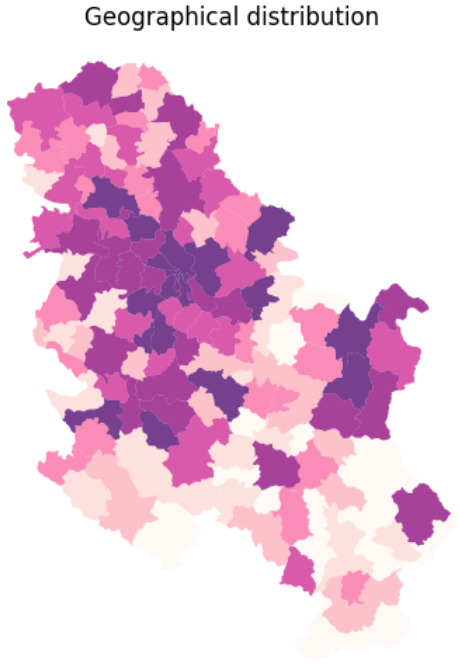
The only additional arguments to pass for producing a choropleth, therefore, are the actual variable we would like to classify and the number of segments we want to create, `k`. This is, in other words, the number of colours that will be plotted on the map so, although having several can give more detail, at some point the marginal value of an additional one is fairly limited, given the ability of the human brain to tell any differences.

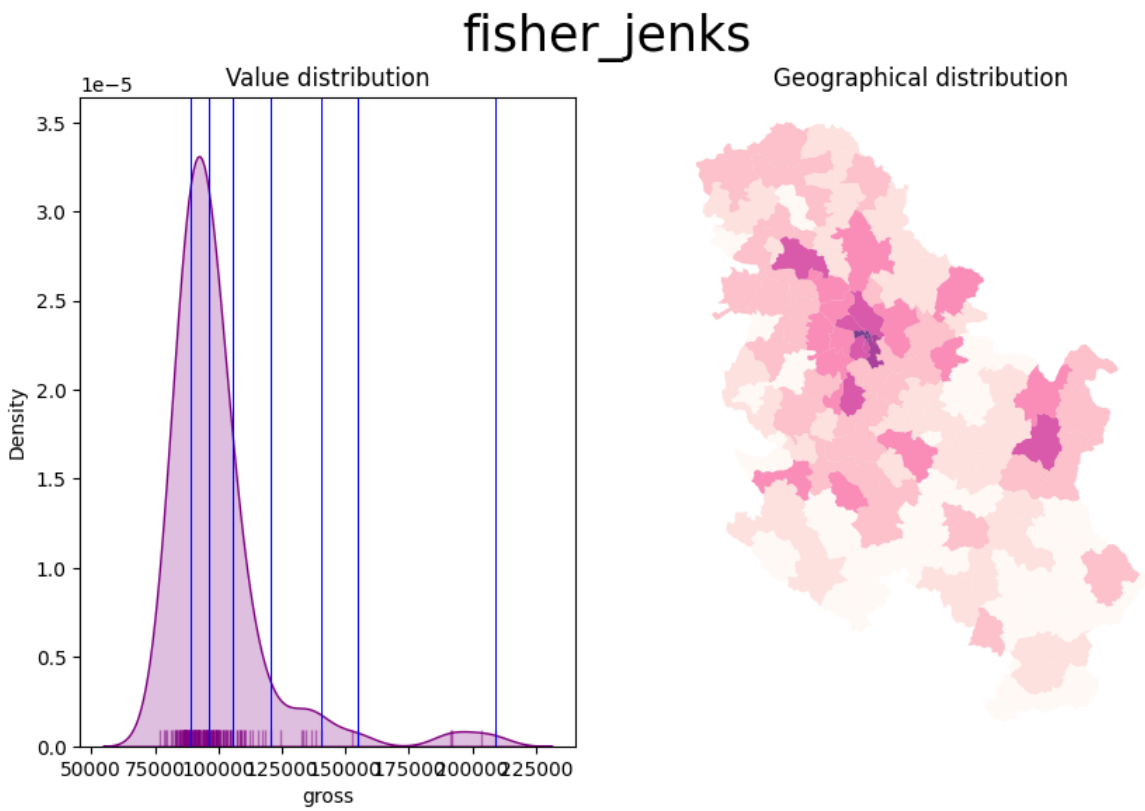
```
schemes = ['equal_interval', 'quantiles', 'fisher_jenks']
for scheme in schemes:
    plot_scheme(serbia_admin, 'gross', scheme)
```


equal_interval
Value distribution
Geographical distribution
1e−5
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Density
50000
75000
100000
125000
150000
175000
200000
225000
gross

quantiles


Value distribution
1e−5
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Density
50000 75000 100000 125000 150000 175000 200000 225000
gross


Geographical distribution



Also consider the Modifiable Areal Unit Problem and how the geographies of the administrative boundaries, in this case, may impact the visualisation.

For example, the most populated area is a municipality in the north that corresponds to the city of Novi Sad. Let's have a look at the data

```
serbia_admin[['name', 'pop', 'Province']].sort_values(by = 'pop', ascending = False).iloc[:1
```

	name	pop	Province
48	Novi Sad	341625.0	Južno-Bački
24	Novi Beograd	186667.0	Grad Beograd
19	Čukarica	154854.0	Grad Beograd
3	Kragujevac	154290.0	Šumadijski
26	Palilula	148292.0	Grad Beograd
34	Zemun	143173.0	Grad Beograd
32	Voždovac	137315.0	Grad Beograd
35	Zvezdara	130225.0	Grad Beograd
39	Leskovac	123201.0	Jablanički

	name	pop	Province
120	Subotica	121250.0	Severno-Bački

In our dataset, the city of Novi Sad is categorised as a municipality by itself, because the administrative boundaries file is not updated. In reality, "since 2002, when the new statute of the city of Novi Sad came into effect, Novi Sad is divided into two city municipalities, Petrovaradin and Novi Sad. From 1989 until 2002, the name Municipality of Novi Sad meant the whole territory of the present-day city of Novi Sad." (see: wikipedia).

On the contrary, Grad Beograd, that is Belgrade, is correctly split into different municipalities and its population, when visualised, is spread out across the different geometries of its municipalities. In other words, our map depends on the geometries of the areas and on how the data was collected. While it could be that these areas were indeed identified by population size in the first place, the point is that the fact that Novi Sad is not split into more areas, as Belgrade is, makes it stound out more clearly from the map (and to some extent a bit unfairly)

This may happen with different types of data, particularly with administrative boundaries and it is crucial to reflect on how Choropleth maps may be impacted. One can look for more granular data or consider to weight the continuous value with the extent of the area (i.e. obtaining density values).

3.2.1.3 An alternative to scheme: ColorMap Normalisation

The `mpl.colors.Normalize` function in `matplotlib` creates a normalization object, which adjusts data values into a range that is ideal for colour mapping in a colormap. This function is particularly beneficial in scenarios where precise control over the mapping of data values to colour representations is needed.

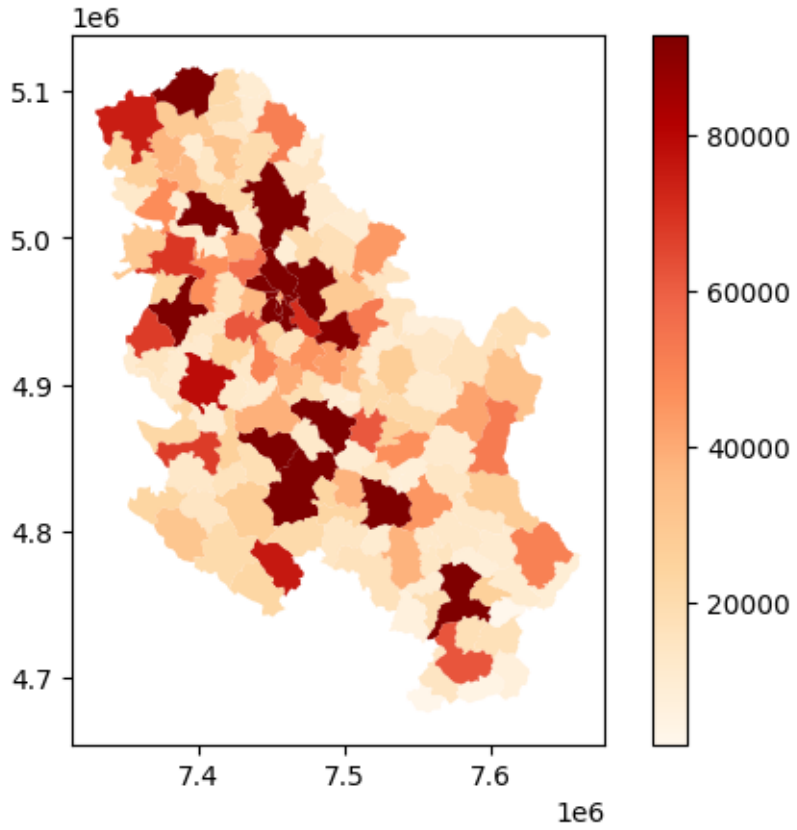
When employed in a plotting function, this normalization object ensures that the data values are scaled to fit a pre-defined range (for instance, `norm = mpl.colors.Normalize(vmin=0, vmax=40)`). Any values falling below 0 are mapped to the lowest colour on the colormap scale, while values exceeding 40 are mapped to the highest colour. This approach is especially useful when aiming to highlight differences within a specific data range; it can significantly enhance the visualization of data, by, for example, emphasizing temperature variations between 0°C and 40°C. This becomes crucial in instances where a few data points with high values (e.g., 50°C) might otherwise lead to a less informative visualization if not 'normalized' and treated as if they corresponded to 40° C values.

For our dataset, we can use as `vmax` the value corresponding to the 90th percentile.

```
serbia_admin['pop'].quantile(0.90)
```

```
93014.0
```

```
import matplotlib as mpl
fig, ax = plt.subplots(1, 1)
vmin = serbia_admin['pop'].min()
vmax = serbia_admin['pop'].quantile(0.90) #
norm = mpl.colors.Normalize(vmin=vmin, vmax=vmax)
serbia_admin.plot(ax = ax, column='pop', cmap='OrRd', legend=True, norm = norm)
```

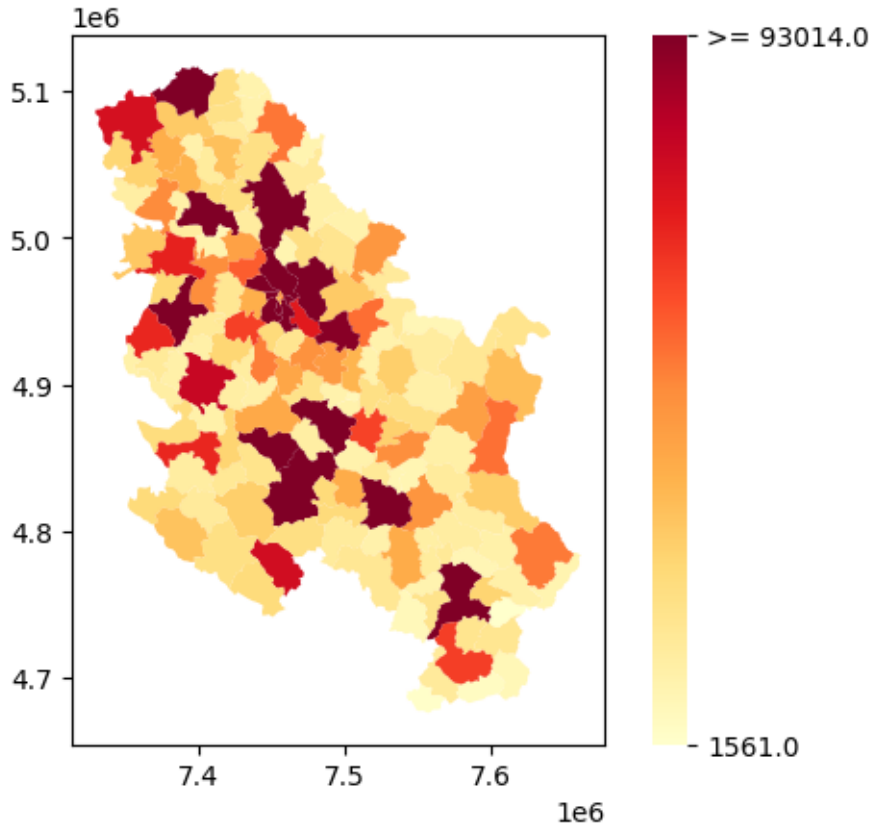


Important:

When passing `norm` in the `plot` method, do not pass the arguments to the `scheme` parameter. For continuous variables, `norm` maps each value directly to a color, making discrete categorization redundant. In other words, it allows for a direct mapping of data values to the color map, eliminating the need for intermediary classification schemes. `norm` ensures a smooth gradient in the color map without artificially segmenting the data.

3.2.1.4 Customising the colorbar

```
import matplotlib.cm as cm

fig, ax = plt.subplots(1, 1)
cmap = 'YlOrRd'
# we leave the legend out
serbia_admin.plot(column='pop', cmap=cmap, norm = norm, ax=ax)

# we add the colorbar separately passing the norm and cmap
cbar = fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax = ax)
cbar.outline.set_visible(False)

# updating ticks VALUES
ticks = [norm.vmin, norm.vmax]
cbar.set_ticks(ticks = ticks)

# updating ticks LABELS
cbar.ax.set_yticklabels([round(t,1) for t in ticks])
cbar.ax.set_yticklabels([round(t,1) if t < norm.vmax else ">= "+str(round(t,1)) for t in cba
```



Above, we removed the outline of the color bar. Then we set the tick values to the min and the max population values, based on our norm object. Then, for the vmax value's label we added a ">=" to remind us that other, higher values are displayed with the darkest color.

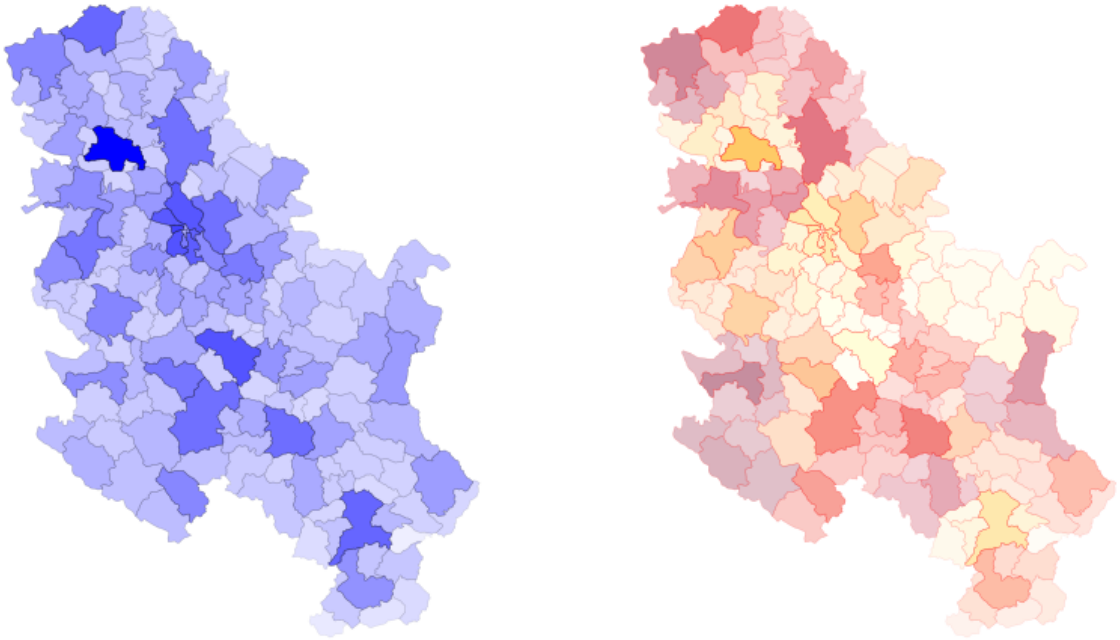
3.2.1.5 Varying alpha transparency based on an array

Finally, we can also convey variation in a continuous scale through transparency. `alpha` doesn't expect column names, so we cannot just pass the name of the column containing the variable. Instead, we have to create an array from 0.0 to 1.0 values. To so we can a) use normalisation methods, or b) rescale the original values within 0 to 1 based on the original min and max values.

For example, with square root normalization:

```
# 1. Create an alpha array based on a normalized value (e.g., population)
import numpy as np
pop_max = serbia_admin['pop'].max()
alpha = np.sqrt(serbia_admin['pop'] / pop_max)
```

```
# Plot with varying alpha values
fig, axes = plt.subplots(1, 2, figsize=(10, 6))
serbia_admin.plot(color = 'blue', ax=axes[0], alpha=alpha, edgecolor='black', linewidth = 0.3
serbia_admin.plot(cmap = 'YlOrRd', ax=axes[1], alpha=alpha, edgecolor='red', linewidth = 0.3)
for ax in axes:
    ax.set_axis_off()
```



Important:

`matplotlib` would not able to plot a color bar from variations in the alpha value since no column is passed directly. We would need, in this case, to build a color bar manually as demonstrated above.

3.2.2 Choropleth Maps for Categorical Variables

A choropleth for categorical variables assigns a different color to every potential value in the series based on certain colormaps (`cmap`). We don't need to specify a scheme in this case, but just to the categorical `column`. Using last's week GeoDataFrame, we can plot terrorist attacks in Germany, for example, by group.

```
gdf = gpd.read_file("../data/germany.shp").to_crs(germany_crs)
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf.plot(ax = ax, column = 'gname', legend = True)
ax.set_axis_off()
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Terrorist Attacks in Germany, by Group", **title_parameters) #parameters as ab
```

```
Text(0.5, 1.0, 'Terrorist Attacks in Germany, by Group')
```

Terrorist Attacks in Germany, by Group

The map above is what you would get from datasets that are not cleaned/manipulated directly or when there are too many categories in the selected column. First, let's get a slimmer slice of the gdf that only contains attacks that cause a number of fatalities and wounded higher than 10.

```
condition = (gdf.nkill + gdf.nwound) > 10
gdf_filtered = gdf[condition].copy()
```

Then, let's build a function that creates a random color map based on the number of categories. This creates random `HUE`-based colors:

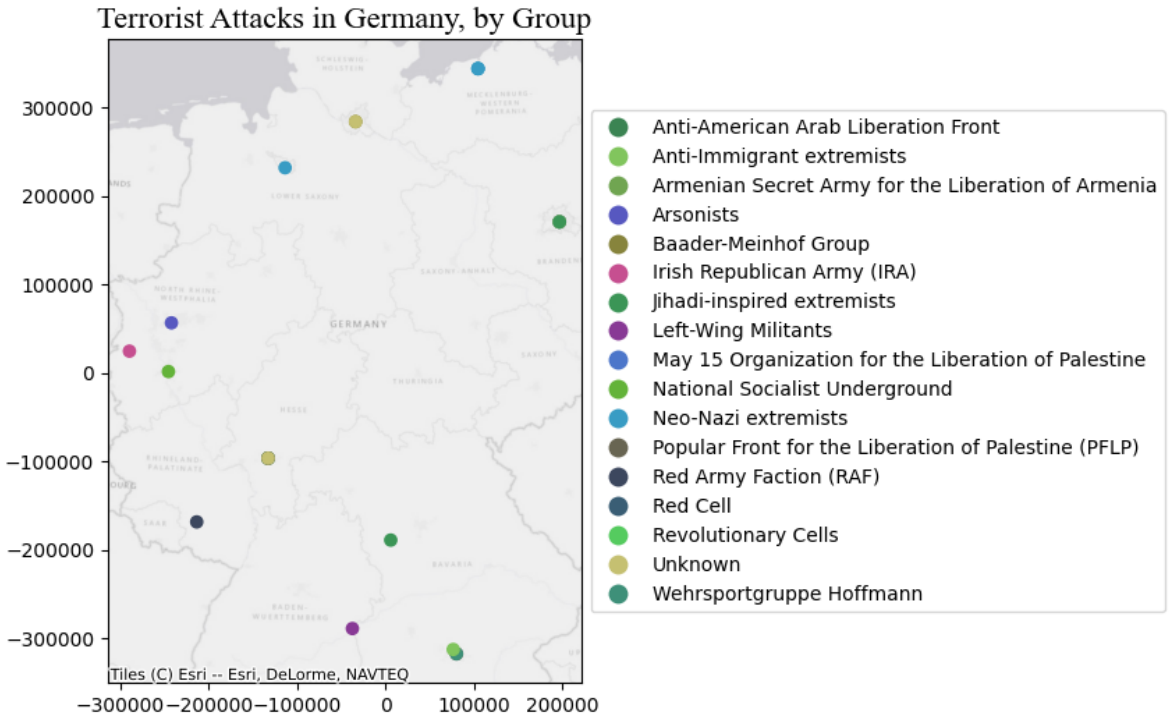
```
# Generate random colormap
def rand_cmap(nlabels):
    """
    It generates a categorical random color map, given the number of classes

    Parameters
    ----------
    nlabels: int
        The number of categories to be coloured.
    type_color: str {"soft", "bright"}
        It defines whether using bright or soft pastel colors, by limiting the RGB spectrum.

    Returns
    -------
    cmap: matplotlib.colors.LinearSegmentedColormap
        The color map.
    """
    # Generate color map for bright colors, based on hsv
    randHSVcolors = [(np.random.uniform(low=0.20, high=0.80),
                          np.random.uniform(low=0.20, high=0.80),
                          np.random.uniform(low=0.20, high= 0.80)) for i in range(nlabels)]

    random_colormap = LinearSegmentedColormap.from_list('new_map', randHSVcolors, N=nlabels)

    return random_colormap
```

```
cmap = rand_cmap(len(gdf_filtered.gname.unique()))
cmap
```

We also place the legend on the centre left. This is done automatically, but the legend and its items can be manipulated directly. Legends in `matplotlib` are extremely complex to personalise. However, do have a look at https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html#ma for both automatic and explicit manipulation.

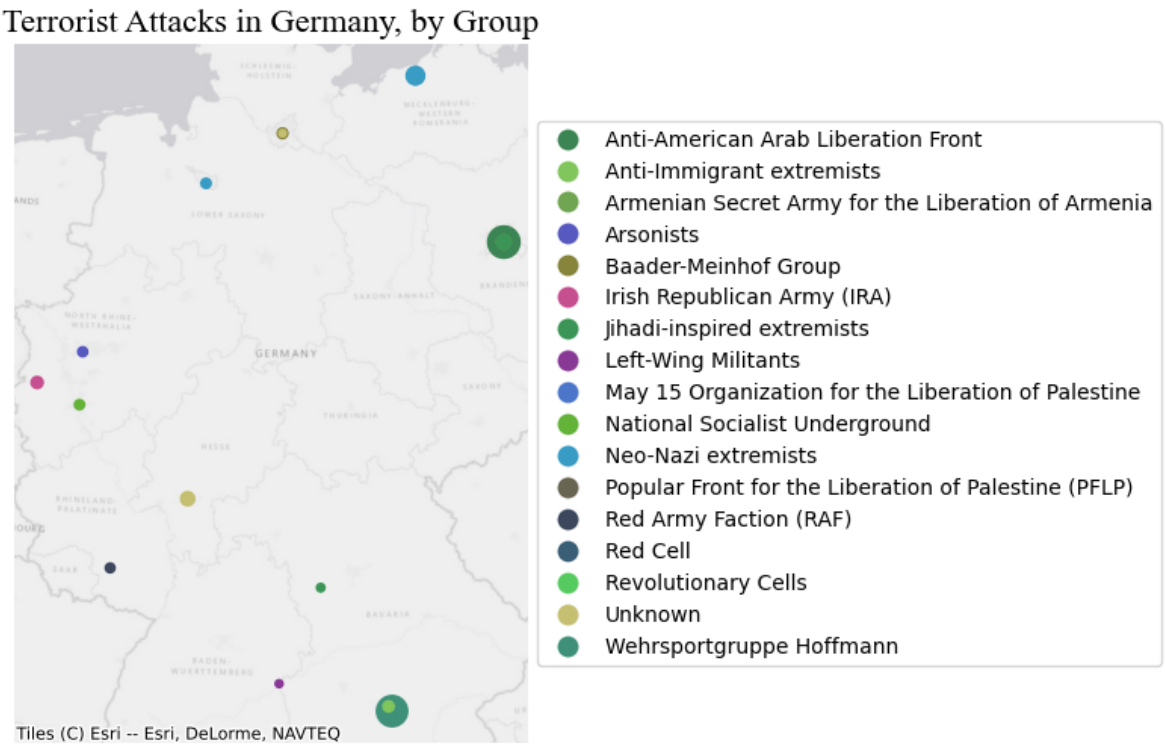
```
legend_kwds={"loc": "center left", "bbox_to_anchor": (1, 0.5)}
```

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf_filtered.plot(ax = ax, column = 'gname', legend = True, cmap = cmap, legend_kwds = legend
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Terrorist Attacks in Germany, by Group", **title_parameters) #parameters as ab
ctx.add_basemap(ax, crs= gdf_filtered.crs.to_string(), source = ctx.providers.Esri.WorldGray
```



We can also convey the impact of the events through the `markersize`. This introduces the concept of *cartogram* (see below).

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf_filtered.plot(ax = ax, column = 'gname', markersize = 'nwound', legend = True, cmap = cma
ax.set_title("Terrorist Attacks in Germany, by Group", **title_parameters) #parameters as abo
ctx.add_basemap(ax, crs= gdf_filtered.crs.to_string(), source = ctx.providers.Esri.WorldGray(
ax.set_axis_off()
```



3.3 Part III: Cartograms - Manipulating the Geometry size for showing the magnitude of a value

Cartograms are maps that represent the spatial distribution of a variable not by encoding it in a color palette but rather by modifying geographical objects. There are many algorithms to distort the shapes of geographical entities according to values, some of them are rather complex.

3.3.1 Polygons

You can obtain cartograms for `Polygon` with `geoplot`: see https://residentmario.github.io/geoplot/

`geoplot` functions pretty much work as `plot`

```
# this library needs the GeoDataFrame to be reverted to WGS
ax = gplt.cartogram(serbia_admin.to_crs(wgs), scale='pop', projection=gcrs.Mercator(), color
# see for projections that work with gplt https://scitools.org.uk/cartopy/docs/v0.15/crs/proj
gplt.polyplot(serbia_admin.to_crs(wgs), facecolor='lightgray', edgecolor='white', ax=ax, lw =
```

3.3.2 Points

For `Point` GeoDataFrames we can just go back to `plot` and pass a column name to `markersize`.

```
attacks = pd.read_csv("../data/GTD_2022.csv", low_memory = False)
country = 'Germany'
```

```
df = attacks[attacks.country_txt == country].copy()
wgs = 'EPSG:4326'
germany_crs = 'EPSG:4839'
gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.longitude, df.latitude), crs = wgs)
gdf = gdf.to_crs(germany_crs)
```

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf.plot(ax = ax, markersize = 'nwound', color = 'purple', legend = True)
ax.set_axis_off()
```

One can also convert polygons into points by using their centroids, and then define the size of the dot proportionally to the value of the variable we want to display.

3.3.3 LineString

For `LineString` we pass the column name to `linewidth`.

Let's load a shapefile of lines. These lines represent frequency of train connections from/to train stations in the region of Liguria (Italy) to other stations within or outside the region. Each line refers to a connection between two specific stations, through a certain type of service and contains information about the frequency of that type of service. For example, the cities of Savona and Finale Ligure might be connected by 5 InterCity trains and 50 regional services. These services correspond to 2 different records.

```
trains_freq = gpd.read_file("../data/trains_liguria.shp" )
trains_freq.crs
```

```
<Projected CRS: EPSG:3003>
Name: Monte Mario / Italy zone 1
Axis Info [cartesian]:
- X[east]: Easting (metre)
- Y[north]: Northing (metre)
Area of Use:
- name: Italy - onshore and offshore - west of 12°E.
- bounds: (5.93, 36.53, 12.0, 47.04)
Coordinate Operation:
- name: Italy zone 1
- method: Transverse Mercator
Datum: Monte Mario
- Ellipsoid: International 1924
- Prime Meridian: Greenwich
```

Let's check the type of services contained here.

```
trains_freq['train_type'].unique()
```

```
array(['REG', 'IC', 'FB', 'ICN', 'U', 'EC/EN', 'AV'], dtype=object)
```

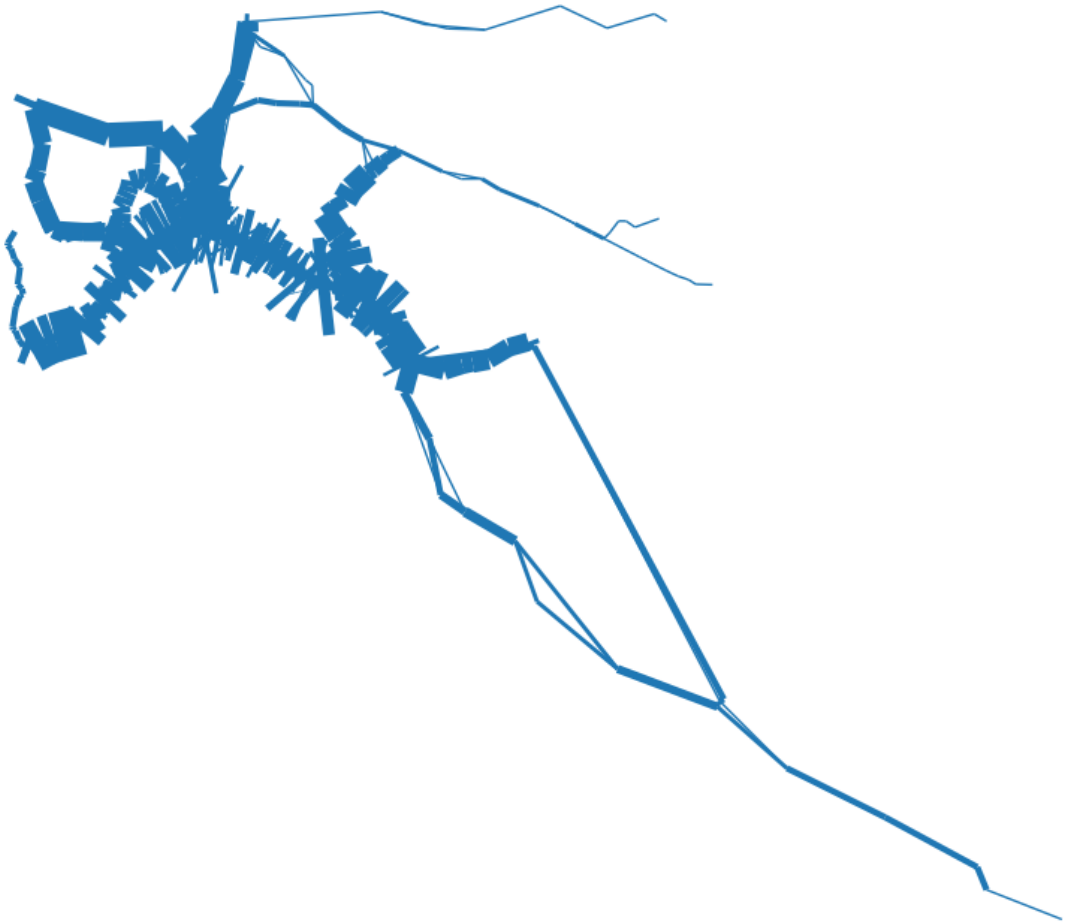
We have: - 'REG': regional trains. - 'IC': intercity trains. - 'FB': similar to IC, but slightly faster. - 'ICN': sleeper trains. - 'U': urban trains (Genoa). - 'EC/EN': international trains. - 'AV': High-speed trains.

Let's keep just regional, intercity, and high-speed trains.

```
to_keep = ['REG', 'IC', 'AV']
trains_freq = trains_freq[trains_freq['train_type'].isin(to_keep)]
```

The usage of `linewdith` is a bit different from `markersize` for some reason. We have to pass an `array` of N values, where N is equal to the GeoDataFrame size. In other words, we have to pass the column we want to use to regulate the line width directly as a list/array. Specifying the column name is not enough.

```
fig, ax = plt.subplots(1, 1, figsize=(10, 15))
trains_freq.plot(ax = ax, linewidth = trains_freq['freq'])
ax.set_axis_off()
```



As you can see, the default arguments and simply passing the column values do not produce pretty results. The first thing to look at is the values that are passed to `linewidth`. In some cases, the min and max values, as well as their distribution, are not ideal for visually conveying the magnitude of the variable attached to the geometry. One option is to use a multiplier factor (see below), or to rescale the values from 0 to 1, for example, and then, again, if necessary use a multiplier.

```
fig, ax = plt.subplots(1, 1, figsize=(15, 20))
lw = trains_freq['freq'] * 0.15
trains_freq.plot(ax = ax, linewidth = lw, capstyle = 'round', joinstyle = 'round', column =
ctx.add_basemap(ax, crs= trains_freq.crs.to_string(), source = ctx.providers.Esri.WorldGrayCa
ax.set_axis_off()
```

While this looks a bit better, this visualisation is not ideal because the frequencies are not snapped to the actual railway network. The lines represent, instead, connection between train stops and therefore their coordinates only include the ones corresponding to the stations where the different services call at. One can devise approaches to:

- Assigning the frequencies, or any other value, to the corresponding infrastructure's section. For example, the railway section between two stations could be associated with a value representing the total number of regional/local services travelling along it.
- Smoothing the lines representing the services by adding further coordinates along the line.

Both these processes go beyond the scopes of this lab and require several considerations depending on the data, the scale, and what information one wants to displays.

Exercise:

Today we've seen how to exploit `matplotlib` to plot `GeoDataFrame` layers. Go through the notebook again if you feel that there's something you need to review. You are not expected to remember each step/method/parameter. Rather, this notebook should be used as a reference for producing maps in Python. Do keep in mind that most of the maps above have been produced with just a bunch of rows, so each of them can be improved and embellished with some more effort.

Now, if you are not overwhelmed, have a look at the very last map and produce some nice visualisation using the same data. You can further improve its clarity, add a legend that refers to the line width, visualise only a certain type of services, or add information/context, for example. In the folder `\data` you can also find a `.shp` file containing all the train stations in Italy, should you need that.

3.3.3.1 Saving figures (check here for details)

```
fig.savefig("fig1.pdf", dpi='figure', format="pdf", bbox_inches = 'tight')
```