Web Mapping and Geovisualisation

Gabriele Filomena, Elisabetta Pietrostefani, and Matt Howard

2025-02-18

Table of contents

Welcome

This is the website for "Web Mapping and Geovisualisation" (module **ENVS456**) at the University of Liverpool. This course is designed 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 2024-25

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.

Matt Howard - 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.

Asking for Help

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.

Assessment:

Two pieces of coursework (Assignment 1 50% and Assignment 2 50%). Equivalent to 2,500 words each. See guidilines on the assignment pages, submission and marking criteria here.

Syllabus

Week 1 MH

- Lecture: Introduction to the module
- Lab: Powerful examples and Python Refresher

Week 2 GF

- Lecture: Geovisualisation: Design Principles and Statistical Visualisation
- Lab: Static Maps

Week 3 MH

- Lecture: The Web's Architecture
- Lab: Web Architectures and APIs

Week 4 MH

- Lecture: Data architechtures & formats. Room: Chemistry Building, Gossage LT
- Lab: Data architechtures & Tiles. Room: Central Teaching Lab - PC Teaching Centre

Week 5 GF

- Lecture: Interactive Maps. Room: Chemistry Building, Gossage LT
- Lab: Interactive Maps. Room: Rendall Teaching Centre - PC Teaching Centre

Week 6 MH

- Lab (h 11.00 am): Q&A & Assignment I Clinic. Room 502 Building, PC Teaching Centre A + B (if necessary)
- **Assignment I Due**

Week 7 GF

- Lecture: OpenStreetMap Data & Spatial Network. Room: Chemistry Building, Gossage LT

- Lab: Retrieving Data From OpenStreetMap. Room: Rendall Teaching Centre - PC Teaching Centre

Week 8 GF

- Lecture: Dashboards. Room: Chemistry Building, Gossage LT
- Lab: Designing Dashboard for Geovisualisation. Room: Central Teaching Lab - PC Teaching Centre

Week 9 GF

- Lab (h 11.00 am): Advanced Tools. Room: Central Teaching Lab - PC Teaching Centre

Week 11 GF

- Lab (h 11.00 am): Review & Assignment II clinic. Room: Central Teaching Lab - PC Teaching Centre
- **Assignment II Due (Week 13)**

Assessments: General Remarks

Submission

Please follow these templates consistently:

- Assignment I
- Assignment II

Download the template as follows: and edit them accordingly.

You will submit through Canvas a `.html` file obtained from a Python `.ipynb` Jupyter Notebook file. To do so, in your `.ipynb` file, follow these steps: `File` –> `Save and Export as..` –> `HTML`. Prior to this step, the notebook needs to be rendered (i.e. all the cells should be executed).

Other file formats will not be accepted.

Important for Assignment II, before exporting your `.ipynb` to a `.html` file: follow the steps described in the template to include an interactive dashboard in the static `html` that will be sumbitted as your assignment. This is to guarantee that your dashboard works in the submission file. You are responsible of your dashboard working as expected.

Marking Criteria

This course follows the standard marking criteria (the general ones and those relating to GIS assignments in particular) set by the School of Environmental Sciences. Please make sure to check the student handbook and familiarise with them. In addition to these generic criteria, the following specific criteria will be used in cases where computer code is part of the work being assessed:

Mark Range	Description	Narrative	Map(s) and/or Dashboard Design	Technical Skills	Interactive Map/Dashboard Fully Working (required)	API Call (required) (A1)
0-15	Minimal or no effort, incomplete, or no functionality.	Limited or unclear problem and justification.	Minimal effort, lacking justification for design choices.	Code does not run, no documentation provided.	No	No
16-39	Basic functionality with significant issues or missing components.	Problem and justification are somewhat unclear or incomplete.	Design choices lack depth or clear connection to the dashboard's aim.	Code does not run or produces incorrect output. Some documentation is provided, but it lacks clarity or detail.	Partially	No
40-49	Meets minimum requirements with some functionality and basic clarity.	Problem and justification are clear but not well-integrated into the overall framework.	Basic design choices are made, and some connection to the interactive map's (A1) or dashboard's aim (A2) is present, but the design lacks refinement.	Code runs and produces the expected output. Documentation is present but lacks depth or proper formatting.	Partially	No

Mark Range	Description	Narrative	Map(s) and/or Dashboard Design	Technical Skills	Interactive Map/Dashboard Fully Working (required)	API Call (required) (A1)
50-59	Meets expectations with functional output and clear structure.	Clear problem identification and justification, but limited integration with other components.	Reasonable design choices are made with some level of thought towards usability and presentation.	Code runs and produces the expected output. Extensive documentation explaining the logic is provided.	Partially	Yes
60-69	Good overall quality with functional and thoughtful design.	Problem and justification are clear, and some integration with the overall framework is evident.	Thoughtful design choices are made and connected to the interactive map's (A1) or dashboard's aim (A2), with some justification for interactivity and widgets (A2).	Code runs and produces the expected output. Extensive, properly formatted documentation is provided, showing a good understanding of concepts.	Yes	Yes

Mark Range	Description	Narrative	Map(s) and/or Dashboard Design	Technical Skills	Interactive Map/Dashboard Fully Working (required)	API Call (required) (A1)
70-79	High-quality work with clear evidence of advanced skills and integration of components.	Problem and justification are clear and well-integrated with all components of the assignment, creating a cohesive narrative.	Excellent design choices are made, with strong connections to the interactive map's (A1) or dashboard's aim (A2) and well-justified interactivity (A1 and A2) and widgets (A2).	Code runs and produces the expected output. Evidence of advanced skills is demonstrated in the code design.	Yes	Yes

Mark Range	Description	Narrative	Map(s) and/or Dashboard Design	Technical Skills	Interactive Map/Dashboard Fully Working (required)	API Call (required) (A1)
80-100	Exceptional work with innovative contributions and flawless execution.	Problem and justification are excellently presented and fully integrated into a cohesive, professional-quality narrative.	Outstanding design choices with creative elements, advanced interactivity, and exceptional attention to detail in connecting the components to interactive map (A1) or the dashboard (A2).	Code runs and produces the expected output. Extensive, properly formatted documentation is provided, including novel contributions (e.g., algorithm optimizations, novel methods to improve functionality).	Yes	Yes

DOs and DONTs

- Do not include "temporary" maps unless you really need to specifically show something to your reader.

- Do not include maps that have no actual differences, apart from few things (e.g. you changed the zoom level).
- Mix the accompanying text, in markdown cells, with the code.
- Do not include all the text at the beginning.
- Provide some theoretical context and motivation to your topic.
- Present 2 or 3 NICE maps and the final interactive map (Assignment I) and the one included in your dashboard (Assignment II)

Assignment I

- **Title:** Exploring APIs and Interactive Maps in Python.
- **Type:** Coursework.
- **Due: Thursday March 6th (2.00 pm) - Week 6**.
- 50% of the final mark.
- Submission on Canvas, `.html` files only.

Context, Design, Data, and Assemblage

In this assessment, you will have the opportunity to explore different sources and combine them in a map that can be explored interactively through a web browser. This assignment requires you to identify a research problem from literature, source relevant data from the web in different formats, assemble them, and document the process. To be successful, you will need to demonstrate your understanding not only of the technical aspects involved in the process but also of the conceptual notions underpinning them. Below are the required components for your submission:

1. **Context and Problem**: Identify a research problem with a geographical connotation. Discuss concisely recent research around it in physical or human geography (around 7–8 references). Introduce how you will explore and visualise dimensions of the problem (e.g., gentrification, access to healthy food in cities, urban heat islands, etc.).
2. **Data and Backend**: Draft a list of spatial datasets relevant to your research problem and demonstrate your ability to develop your own API request function in Python. Include datasets containing spatial information or linkable to other spatial sources. Highlight the data/variables worth considering and their role in representing the problem. Demonstrate your understanding of core "backend" web mapping concepts. Include an explanation of how tilesets, client-server architecture, and APIs are implemented and contribute to your map's functionality.
3. **Design**: Create Good Looking static maps to represent your datasets, focusing on spatial units (e.g., buildings, cities). Move onto interactive visualisation with folium, incorporating interactivity for categorical and numerical variables. Seek feedback to refine your ideas. Use inspiration from web map examples discussed in the course to ensure effective representation of data.
4. **Assemblage**: Enhance your map by incorporating widgets for dataset exploration and experimenting with tilesets, such as creating your own in Mapbox. Address design considerations, including the map's extent, zoom levels, and variable visibility at different zoom levels. Ensure consistency and aesthetic appeal to complete this stage successfully.

Expected Content

Code

- Introductory Static Maps (2 to 3), presenting the topic and the geographic context.
- An API request written by your own.
- All the necessary steps for making your API work and for data cleaning/exploration.
- An interactive final map. This should be fed with data obtained through the API request.

You CANNOT employ for your main maps the following libraries: Holoviews, Geoviews, and Plotly

Text in Markdown Cells, 1,000 words, distributed across the notebook:

- About 250 words introducing the research problem, the context, and existing recent research on the topic.
- About 200 words presenting and motivating the chosen data sources, in relation to your research problem. Here you should engage not only with what data you are using but why and what they bring to the map. Everything should be in the map for a reason.
- About 200 words with your description of what your API is, how it works and how it will made your map possible.
- About 200 words with a description of how your interactive map works, its components and your design ideas.
- About 150 words to summarise your research problem and how you tackled it by means of geovisualisation tools (Conclusion).

Evaulation

The assignment will be evaluated based on 3 main pillars, on which you will have to be successful to achieve a good mark:

1. **Narrative**. The ability to identify and present a research problem, motivate and justify one's map, as well as the ability to bring each component of the assignment into a coherent whole that "fits together".
2. **Map design abilities.** The ability to demonstrate the understanding of geovisualisation and interactivity design principles.
3. **Technical skills**. The ability to master Python scripting and technologies that allow one to create a compelling map, but also to access interesting and sophisticated data sources.

How is this assignment useful?

This assessment includes several elements that will help you improve critical aspects of your web mapping skills:

- **Design**: this is not about making maps, this is about making good maps. And behind every good map there is a set of conscious choices that you will have to think throug to be successful (what map? what data? how to present the data? etc.).
- **Technology**: at the end of the day, building good web maps requires solid understanding of current technology that goes beyond what the average person can be expected to know. In this assignment, you will need to demonstrate you are proficient in a series of tasks manipulating geospatial data in a web environment.
- **Presentation**: in many real-world contexts, your work is as good as it can come across to the audience it is intended to. This means that it is vital to be able to communicate not only what you are doing but why and on what building blocks it is based on.

Assignment II

- **Title:** *A dashboard that explores a Spatial Dataset.*
- **Type:** Coursework.
- **Due: Monday May 12th (2.00 pm) - Week 13**.
- 50% of the final mark.
- Submission on Canvas, `.html` files only.

This assignment requires you to build a dashboard for a **spatial data set of your choice**. To be successful, you will need to demonstrate your understanding not only of technical elements, but of the design process required to create a product that can communicate complex ideas effectively. There are three core building blocks you will have to assemble to build your dashboard: the main maps(s), base map, and widgets.

1. **Context and Problem**: Identify a research problem with a geographical connotation. Discuss concisely recent research around it in physical or human geography (around 7–8 references). Introduce how you will explore and visualise dimensions of the problem (e.g., gentrification, access to healthy food in cities, urban heat islands, etc.).
2. **The Dashboard**. Import your data and start building a dashboard with `panel`. Think about what you want to show, which interactive elements you will allow the user to access and how they will let them modify the experience of your dashboard. The dashboard must incorporate interactive map(s), besides allowing the user to play with the dataset. Interactive maps should be built with `folium` or (optionally) with `pydeck`.
3. **The basemap**. Design your own basemap through scripting (e.g. assembling a basemap with OpenStreetMap features in a unique layer) or use available TileSets. Think about the data in the background, which colors, the zoom levels that will be allowed, and how it all comes together to create a backdrop for your main message that is conducent to the experience you want to create. Use the basemap to enhance the visualisation experience of the user.
4. **Additional widgets**. One of the advantages of dashboards in comparison to standard web maps is that they allow to bring elements of analysis to a more finished product. Think about what you want your users to be able to analyse, why, and how that will modify the main map.

Expected Content

Code

- Introductory Static Maps (2 to 3), presenting the topic and the geographic context.
- An API request (optional) data calls, and necessary data cleaning operations.
- All the necessary steps for building and refining the funcitoning of your Dashboard with `panel`.
- An interactive final dashboard (one) that also incoporates an interactive map.

You CANNOT employ for your main maps the following libraries: Holoviews, Geoviews, and Plotly

Text in Markdown Cells, 1,000 words, distributed across the notebook

- About 250 words introducing the research problem, the context, and existing recent research on the topic.
- About 200 words presenting and motivating the chosen data sources, in relation to your research problem. Here you should engage not only with what data you are using but why and what they bring to the dashboard.
- About 200 words for the overall idea of the dashboard. What do you want to communicate? What is the story you want to tell?
- About 200 words where you describe your design choices around interactivity, including both cartographic elements (e.g. zooming, panning) as well as additional interactivity built around components such as widgets.
- About 150 words to summarise your research problem and how you tackled it by means of geovisualisation tools (Conclusion).

Evaluation

The assignment will be evaluated based on 3 main pillars, on which you will have to be successful to achieve a good mark:

1. **Narrative**. The ability to identify and present a research problem, motivate and justify one's map, as well as the ability to bring each component of the assignment into a coherent whole that "fits together".
2. **Dashboard and Map(s) design**. It is very important to think through every step of preparing this assignment as if it was part of something bigger towards which it contributes. Critically introduce every aspect considered when designing the map(s), by explicitly connecting it to the overall aim of the dashboard. One should clearly and

critically describe how they engaged with every design choice (e.g. adding certain widgets or interactivity functions in the dashboard).

3. **Technical skills**. The ability to master Python scripting and technologies that allow one to create an interactive, informative and compelling (geographic) dashboard, as well as to access interesting and sophisticated data sources.

How is this assignment useful?

This assignment combines several elements that will help you improve critical aspects of web mapping:

- **Design**: this is not about making maps, this is about making good maps. And behind every good map there is a set of conscious choices that you will have to think through to be successful (what map? what data? how to present the data? etc.).
- **Technology**: at the end of the day, building good web maps requires familiarity with the state-of-the-art in terms of web mapping tools. In this assignment, you will need to demonstrate your mastery of some of the key tools that are leading both industry and academia.
- **Presentation**: in many real-world contexts, your work is as good as it can come across to the audience it is intended to. This means that it is vital to be able to communicate not only what you are doing but why and on what building blocks it is based on.

Setting up the Working Environment

Follow the instructions for yuour Operating System and test your installation. If you experience any issues, write a message on the Ms Teams channel of the module. Setting up the Python environment is necessary for:

- Executing the Jupyter Notebooks of the Lab sessions of the course.
- Preparing your own Jupyter Notebooks for the assignments (one each).

We will use `Minconda` to handle our working environment. *Miniconda is a free minimal installer for conda. It is a small bootstrap version of Anaconda that includes only conda, Python, the packages they both depend on, and a small number of other useful packages (like pip, zlib, and a few others)*

Set up Miniconda (and Python) on Ms Windows

Installation

1. Install Miniconda:
    - *Option 1*: On a UoL Machine: Download and install Miniconda from here. This will install Miniconda and Python in `C:\`. If this process is aborted because it requires administrator rights, press `Start`, select `Install University Applications`, type and choose `Miniconda`.
    - *Option 2*, **Recommended**: Install Miniconda on your personal Laptop: Follow the instructions here.
2. During the installation, leave the default settings. In particular, when asked whom to "Install Miniconda for", choose "Just for me".

Important: If you do choose to work on University Machines you will have to reinstall `Miniconda` every lab session unless you use a PC where Miniconda has been installed already.

Alternatively, you can work on the lab notebooks directly on the web. This does not require to install `Miniconda`. However, it represents a much slower option, especially when setting up the environment. To do so, you can access the data and the lab notebooks in the `\labs` directory from a virtual copy of the course repository here. If you opt for this option, you do not need to follow the rest of the instructions below.

Set up the Directories

1. Create a folder where you want to keep your work conducted throughout this course. For example, call it `envs456`. You can save it wherever you want. If you are working on a university machine, it could be worth creating it in `M:/`, which should your "virtual" hard-disk.
2. Download the data and the images for running and rendering the jupyter notebooks.
3. Unzip the folders and move the nested folders into the folder `envs456`.
4. Create another folder called `labs`

The folder structure should look like:

```
envs456/
  data/
  labs_img/
  labs/
```

Set up the Python Environment

1. Download the `envs456.yml` from GitHub by cliciking `Download raw file`, top right at this page
2. Save it in the folder `envs456` created before.
3. Type in the search bar and find the `Anaconda Prompt (miniconda 3)`. Launch it. The terminal should appear.

3. In the **Anaconda Terminal** write: `conda env create -n envs456 --file C:\envs456\envs456.yml` and press `Enter`; if the file is located elsewhere you'll need to use the corresponding file path.
4. If you are prompted any questions, press `y`. This process will install all the packages necessary to carry out the lab sessions.
5. In the **Anaconda Terminal** write `conda activate envs456` and press `Enter`. This activates your working environment.
6. *Necessary* on University machines, otherwise *Optional*: Configuration of Jupyter Notebooks

    - In the **Anaconda Terminal**, write `jupyter server --generate-config` and press enter. This, at least in Windows, should create a file to: `C:\Users\username\.jupyter\jupyte`
    - Open the file with a text editor (e.g. Notepad++), do a `ctrl-f` search for: `c.ServerApp.root_dir`, uncomment it by removing the and change it to `c.ServerApp.notebook_dir = 'C:\\your\\new\\path`, for example the directory where you created the `envs456` folder. In the University Machines, it is advised to work on the directory `M:\`.
    - Save the file and close it.

Start a Lab Session

1. Download the Jupyter Notebook of the session in your folder. Choose one jupyter notebook and click `Dowload raw file` as shown below

2. Save the file in the `labs` folder within your `envs456` folder on your machine.
3. Type in the search bar, find and open the `Anaconda Prompt (miniconda 3)`.
4. In the **Anaconda Terminal** write and run `conda activate envs456`.
5. In the **Anaconda Terminal** write and run `jupyter notebook`. This should open Jupyter Notebook in your default browser.

6. Navigate to your course folder in and double click on the notebook downloaded in step 1.
7. You can now work on your copy of the notebook.

Follow these instructions and test your installation **prior to the first Lab Session** (Wed, 31st of January). If you experience any issues, write a message on the Ms Teams channel of the module. Setting up the Python environment is necessary for:

- Executing the Jupyter Notebooks of the Lab sessions of the course.
- Preparing your own Jupyter Notebooks for the assignments (one each).

Set up Miniconda (and Python) on MAC

Installation

To install Miniconda on your personal laptop, Follow the instructions here. During the installation, leave the default settings. In particular, when asked whom to "Install Miniconda for", choose "Just for me".

Set up the Directories

1. Create a folder where you want to keep your work conducted throughout this course. For example, call it `envs456`. You can save it wherever you want. For example, Elisabetta has named her folder `envs456` and it's in her Dropbox in `Users/PIETROST/Library/CloudStorage/Dropbox/envs456`
2. Download the data and the images for running and rendering the jupyter notebooks.
3. Unzip the folders and move the nested folders into the folder `envs456`.
4. Create another folder called `labs`

The folder structure should look like:

```
envs456/
   data/
   labs_img/
   labs/
```

Set up the Python Environment

1. Download the `envs456.yml` from GitHub by cliciking `Download raw file`, top right, at this page
2. Save it in the folder `envs456` created before.
3. Type in the search bar and open the **Terminal**.
4. In the **Terminal** write `conda env create -n envs456 --file envs456.yml` and press `Enter`. This will need to be modified according to where you placed the `envs456` folder. For example, Elisabetta has named her folder `envs456` and it's in her Dropbox in `Users/PIETROST/Library/CloudStorage/Dropbox/envs456/envs456.yml`. If you created the `envs456` folder on your desktop, the path would be `Desktop/envs456`.

4. If you are prompted any questions, press `y`. This process will install all the packages necessary to carry out the lab sessions.

Start a Lab Session

1. Download the Jupyter Notebook of the session in your folder. Choose one jupyter notebook and click `Dowload raw file` as shown below

2. Save the file in the `labs` folder within your `envs456` folder on your machine.
3. Type in the search bar, find and open the **Terminal**.
4. In the **Terminal** write and run `conda activate envs456`.
5. In the **Terminal** write and run `jupyter notebook`.

6. This should open Jupyter Notebook in your default browser. You should see something like this:

7. Navigate to your folder. You can now work on your copy of the notebook.

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. Copyright (c) 2018, Joris Van den Bossche.

```
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)
```

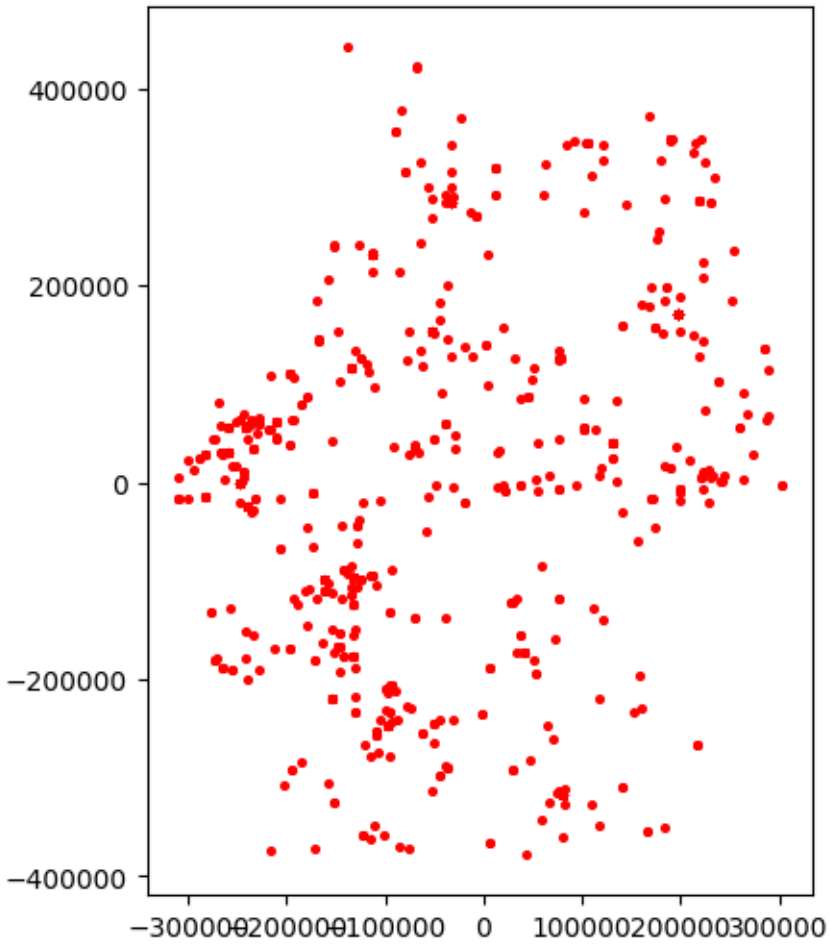
Creating the `GeoDataFrame` from the `DataFrame`

```
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")
```

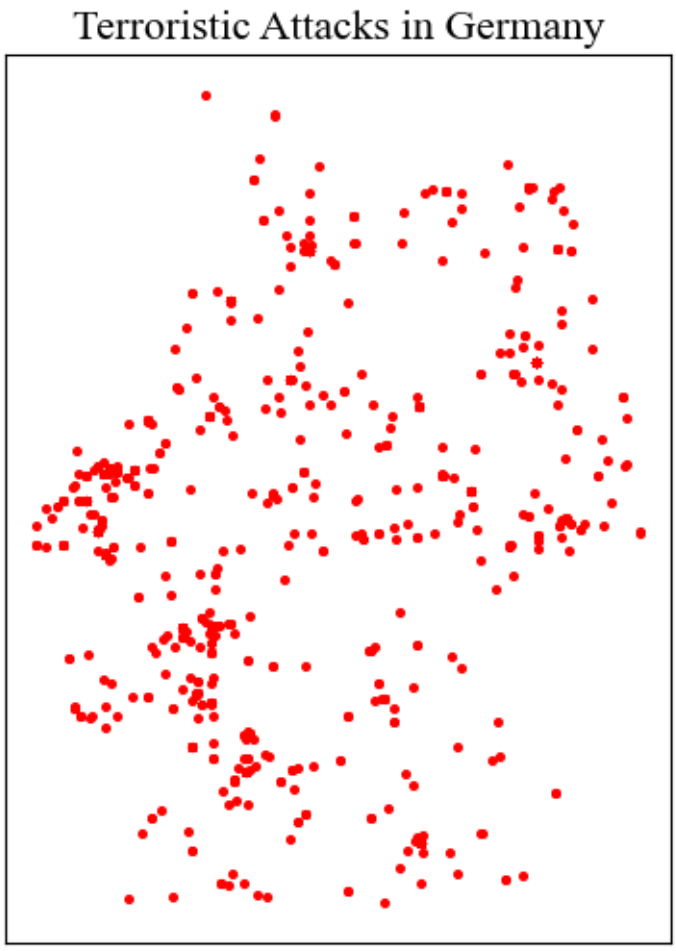
Basic plotting

```
# 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:

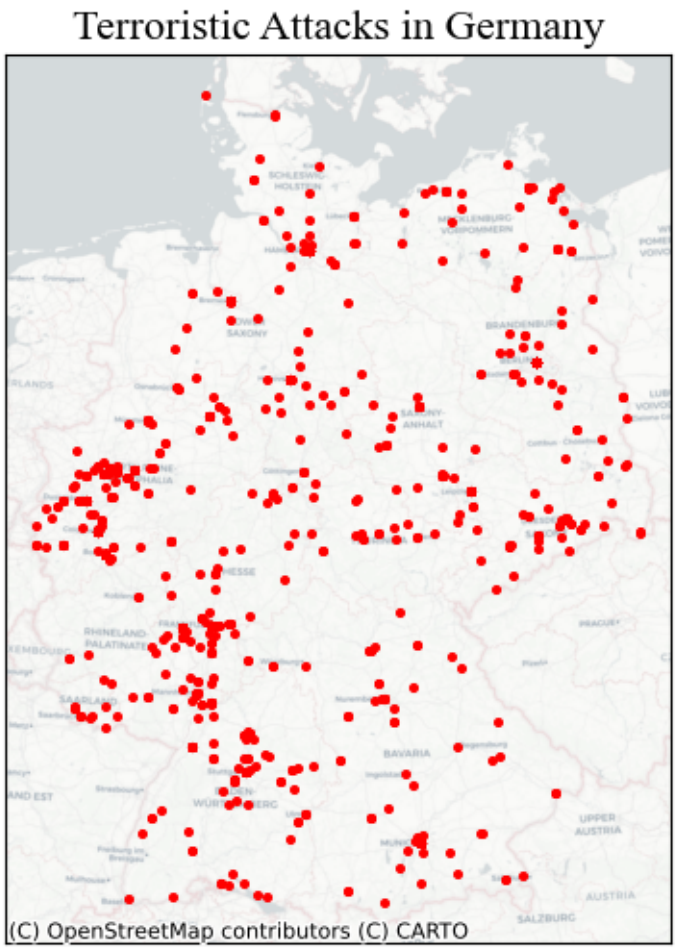
```
# removing ticks
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.tick_params(axis= 'both', which= 'both', length=0)
title_parameters = {'fontsize':'16', 'fontname':'Times New Roman'}
ax.set_title("Terroristic Attacks in Germany", **title_parameters)
fig
```



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

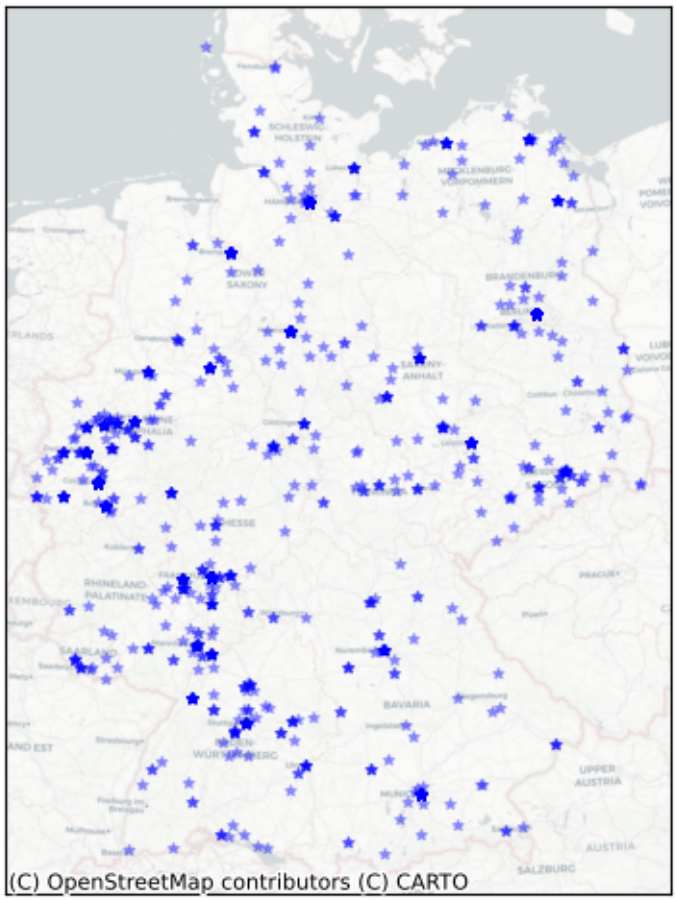
- `markersize`: numerical value (for now)
- `marker`: see https://matplotlib.org/stable/api/markers_api.html

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)
```

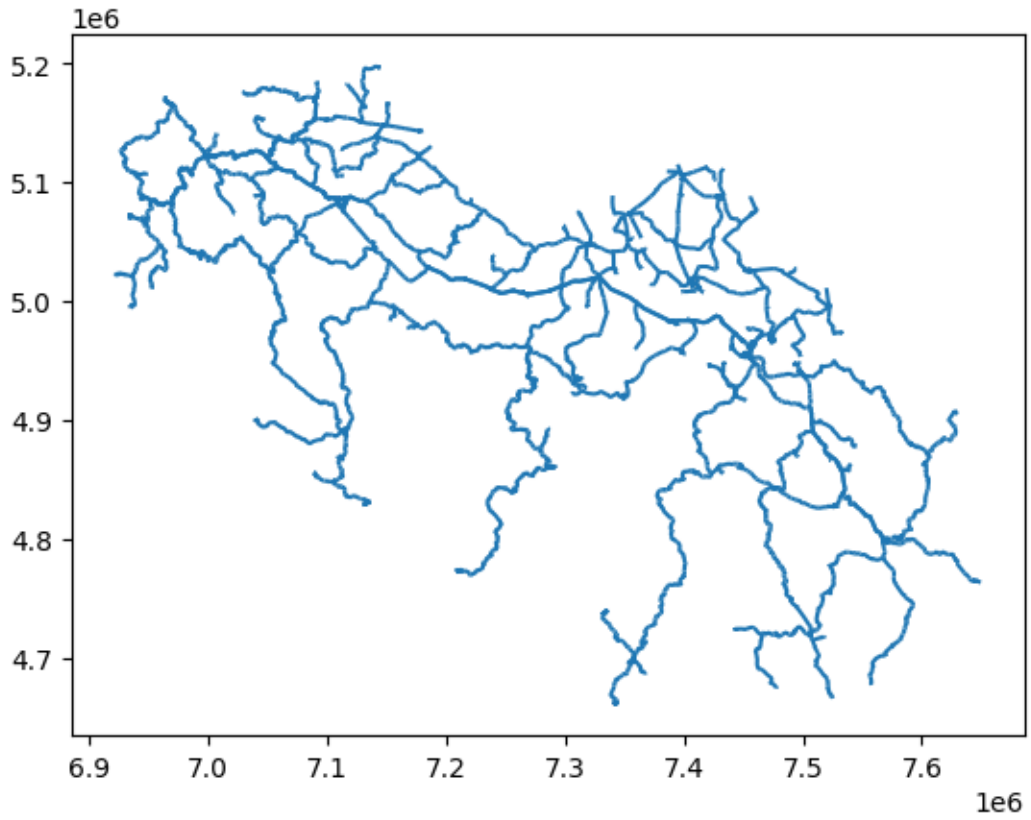
```
# prepare the axis and coordinate
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf.plot(ax=ax, markersize = 15, color = 'blue', marker = '*', alpha = 0.3)
ctx.add_basemap(ax, crs= gdf.crs.to_string(), source= source)
ax_ticks_off(ax)
```



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
```

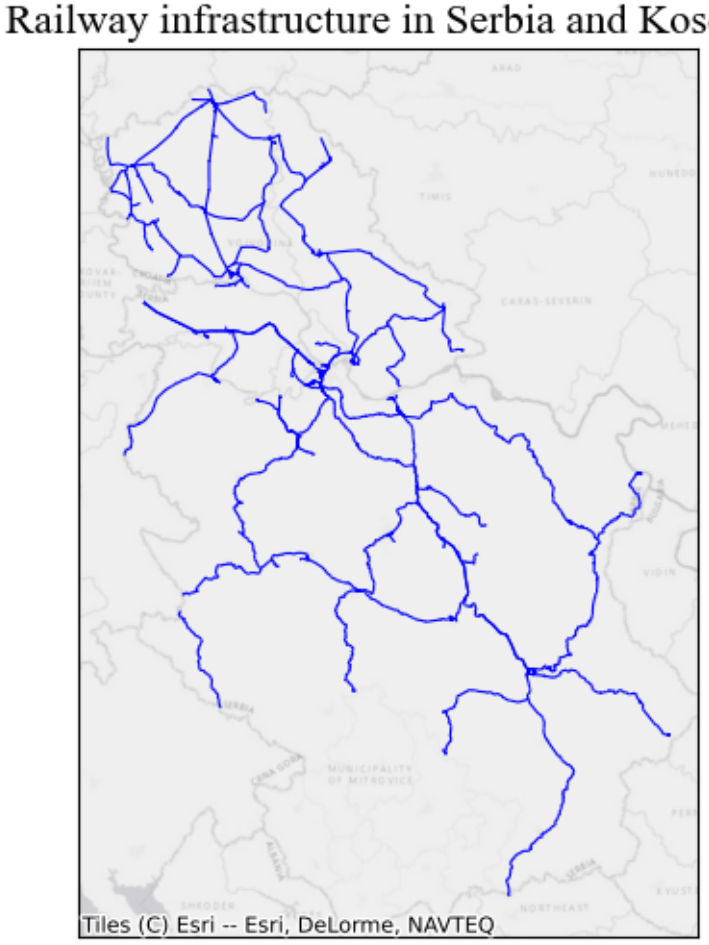
```
Text(0.5, 1.0, 'Railway infrastructure in the West Balkans')
```

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

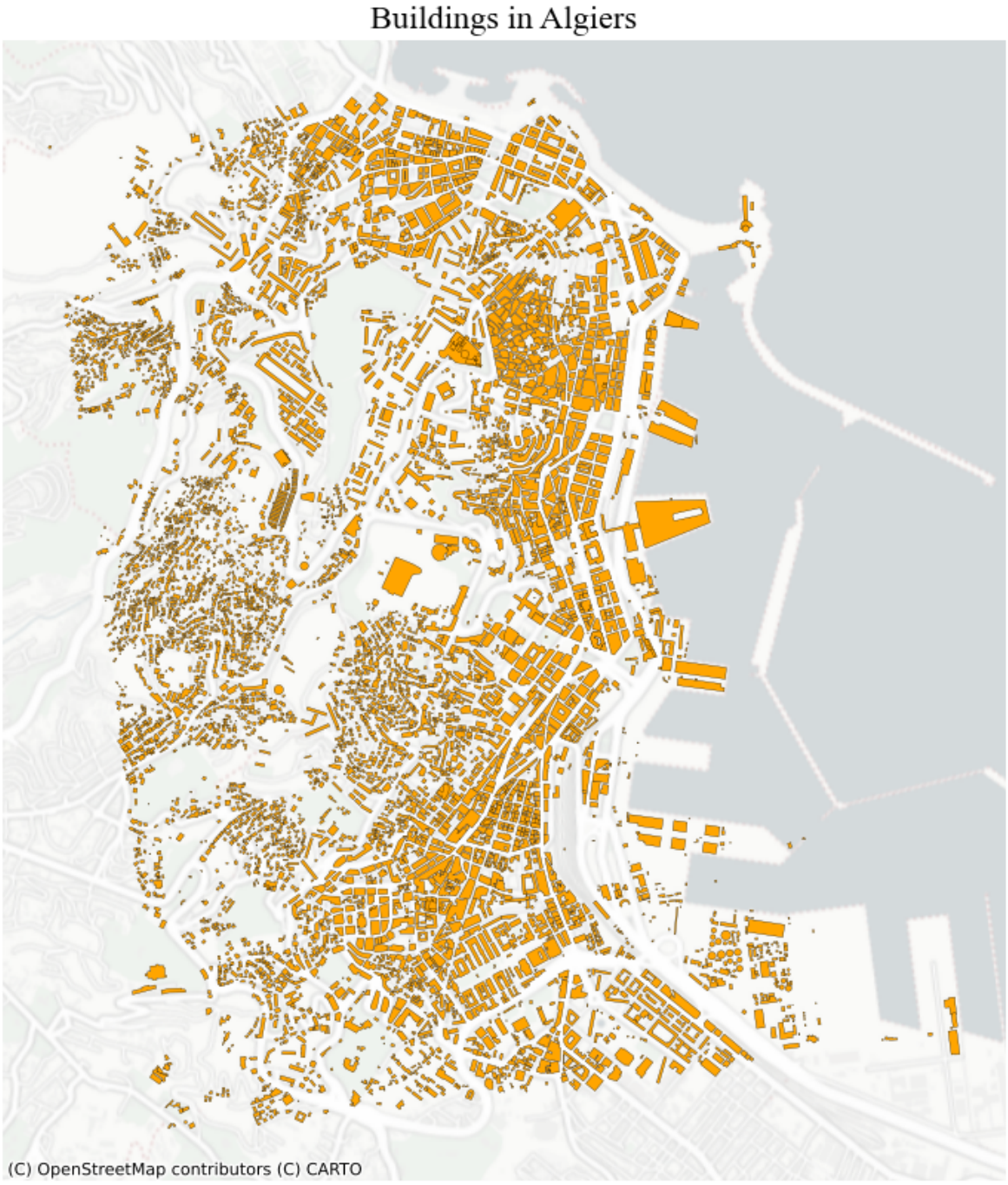
We are again using `OSMNX` to download data from `OpenStreetMap` automatically. In this case, we will get building footprints from the city of Algiers in Alageria.

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.

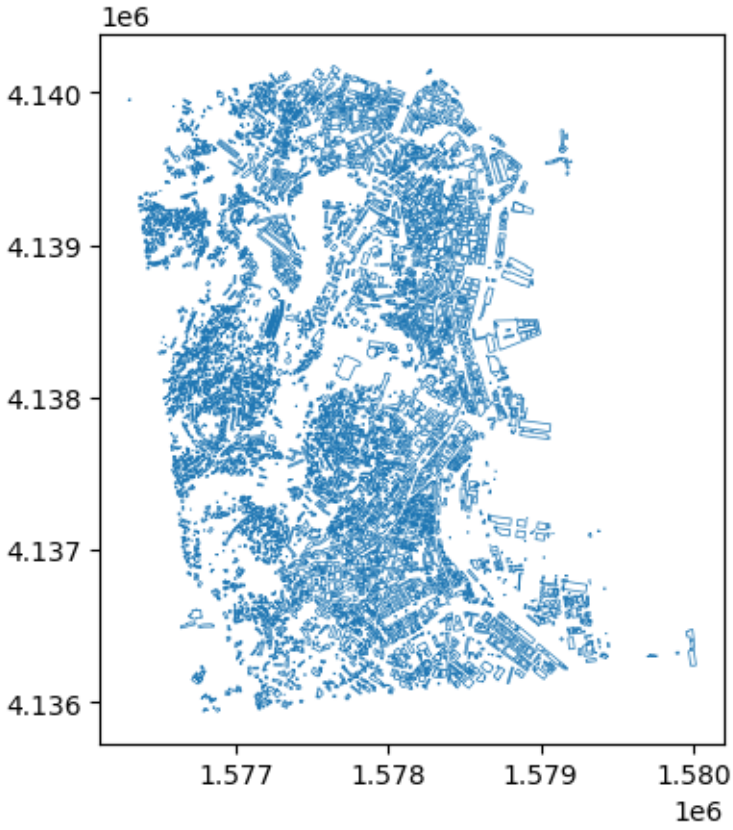
```
algeria_crs = 'EPSG:30729'
tags = {"building": True} #OSM tags
buildings = ox.features_from_address("Algiers, Algeria", tags = tags, dist = 2000)
buildings = buildings.reset_index()
 # sometimes building footprints are represented by Points, let's disregard them
buildings = buildings[(buildings.geometry.geom_type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

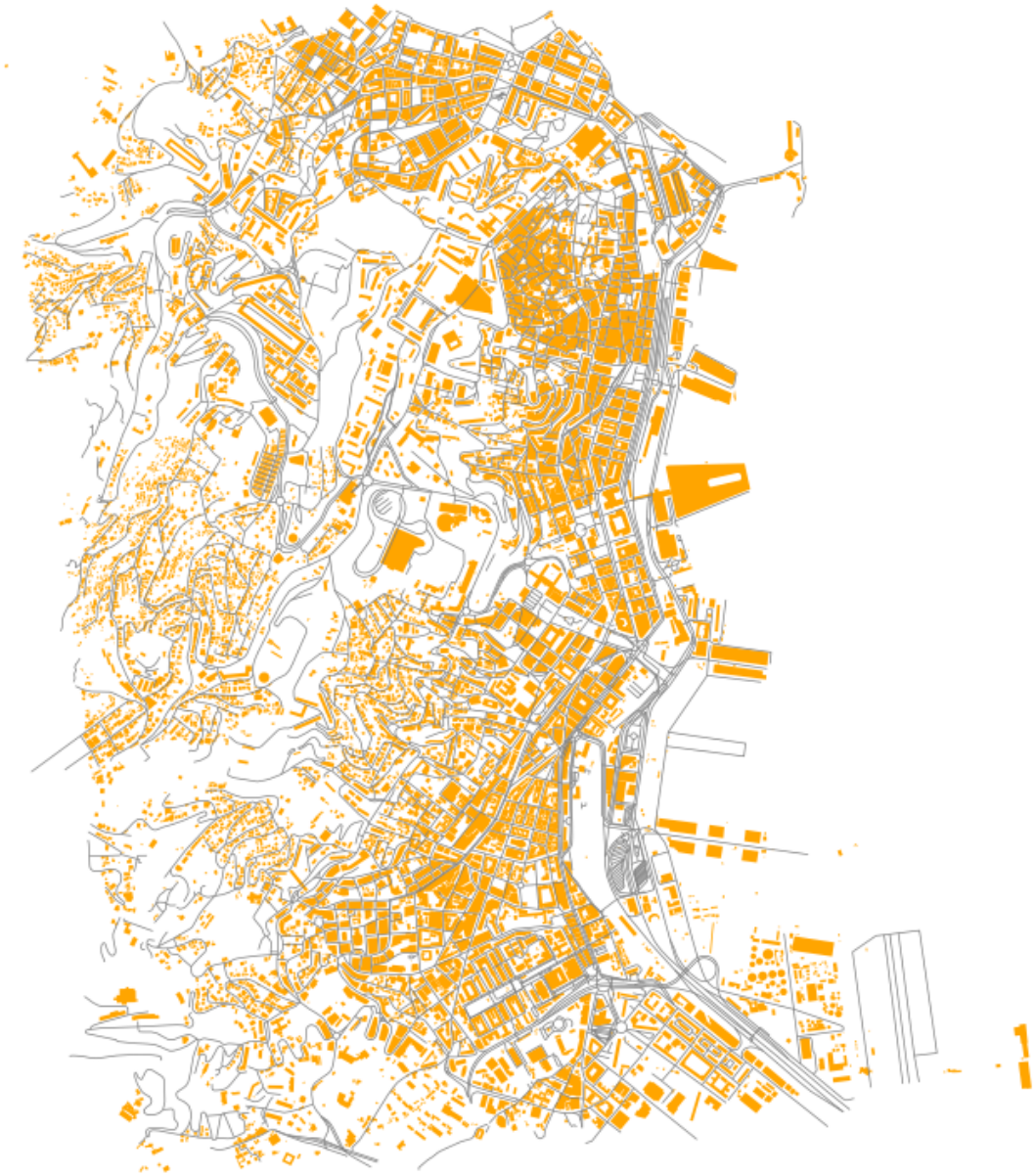
```
tags = {"highway": True} #OSM tags
roads = ox.features_from_address("Algiers, Algeria", tags = tags, dist = 2000)
roads = roads.reset_index()
roads = roads.to_crs(algeria_crs)
 # sometimes building footprints are represented by Points, let's disregard them
roads = roads[roads.geometry.geom_type ==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

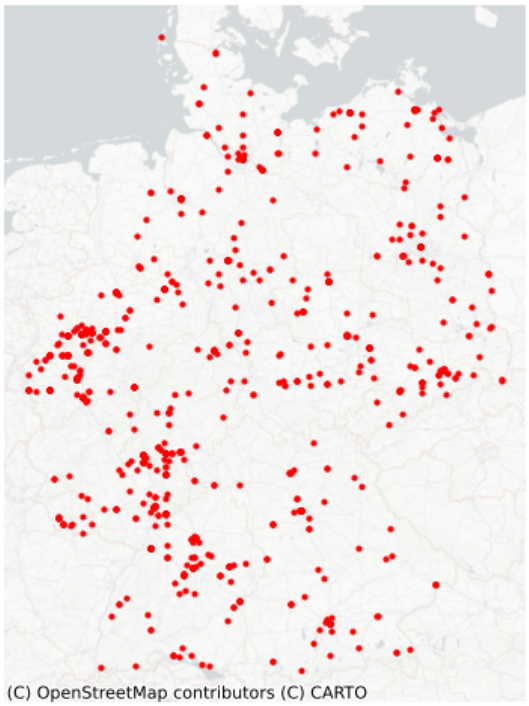
Buildings and Roads in Algiers

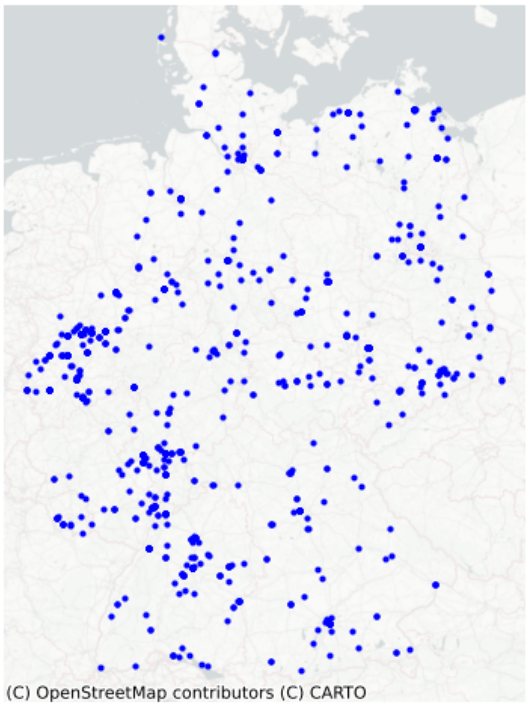


3.1.5 Sub-plots

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)
```

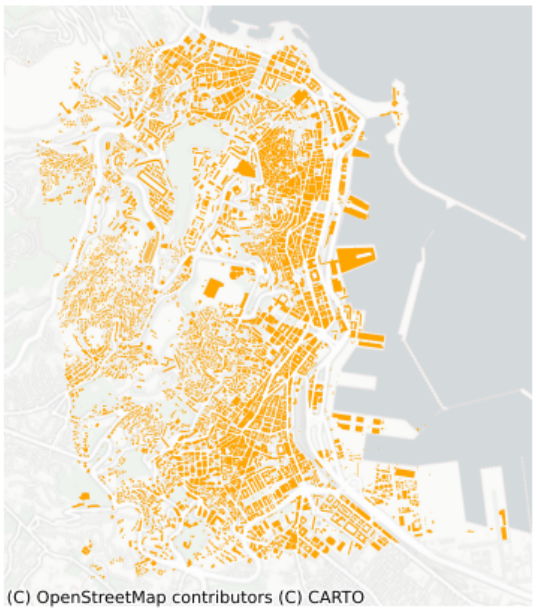




- Plot different layers.

```
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)
```

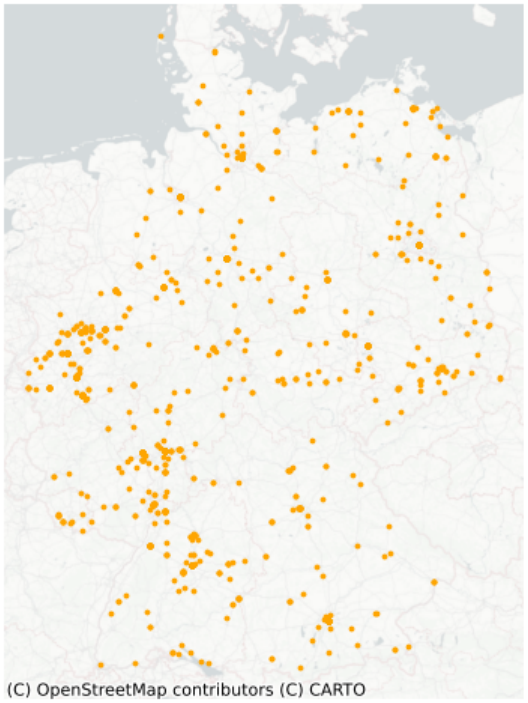
```
for ax in axes:
    ax.set_axis_off()
    ctx.add_basemap(ax, crs= buildings.crs.to_string(), source = source)
```

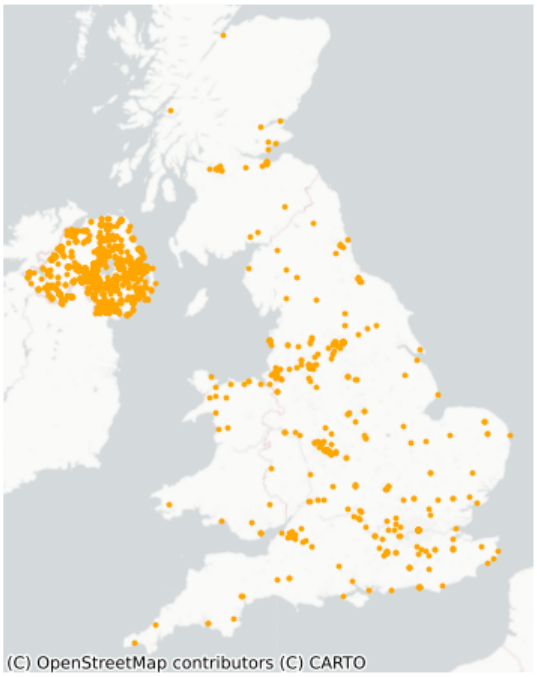


- Analyse phenomena across different geographical areas. For example, terrorism in Germany and in the UK.

```
# 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 `GeoDataFrames` 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).

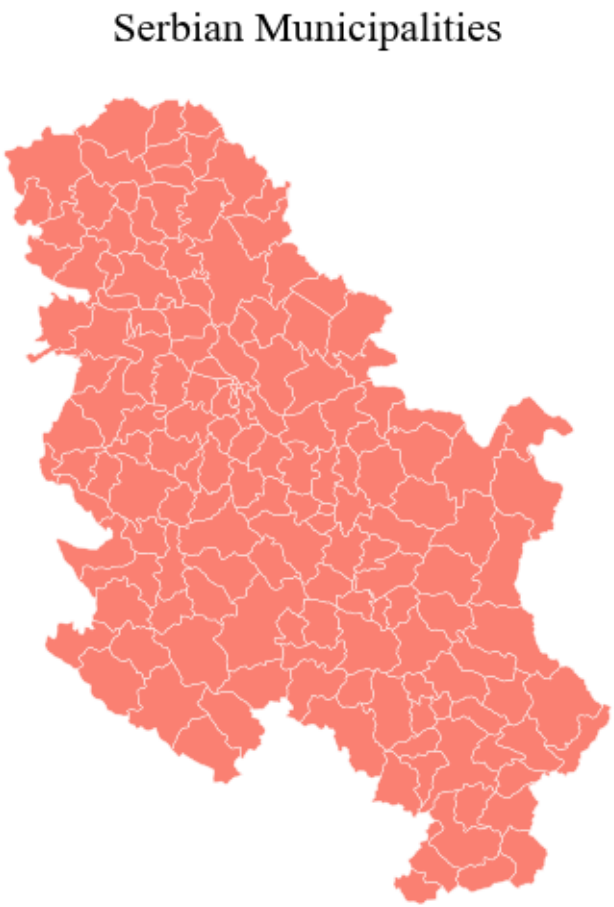
Later on, we will go back to the terrorism dataset.

```
# 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).

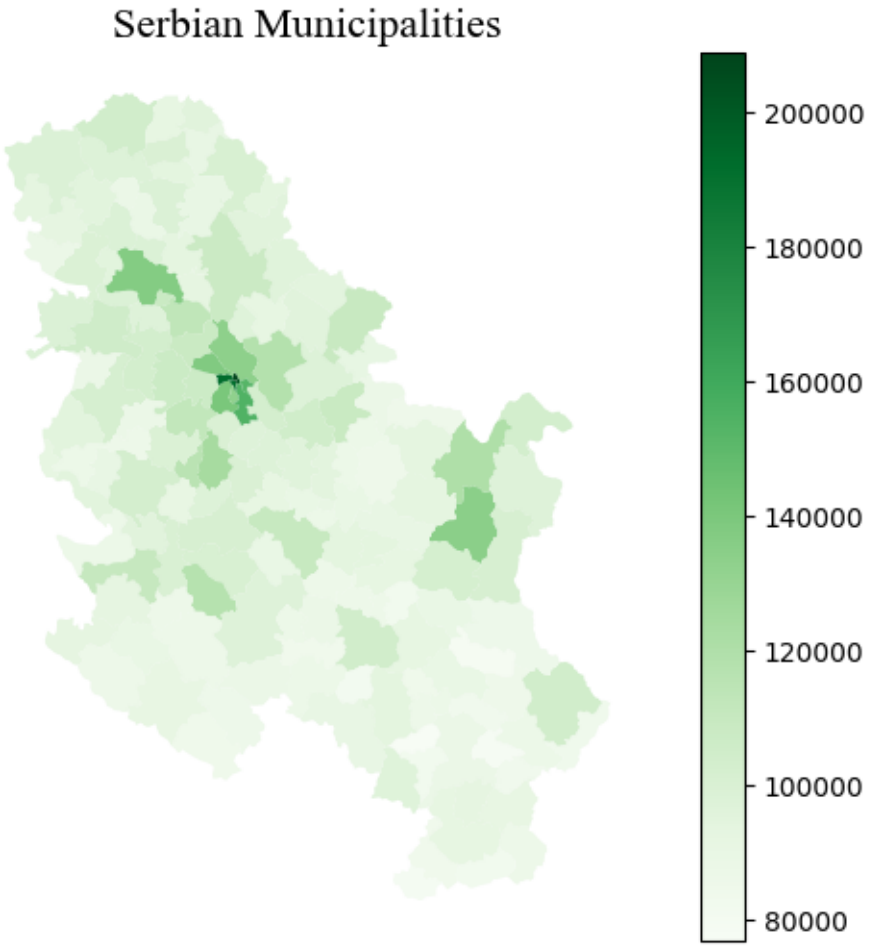
```
data = pd.read_csv("../data/serbia_data.csv") #some slavic characters
data.drop('name_en', axis = 1, inplace = True)
serbia_admin = pd.merge(serbia_admin, data, left_on = "townID", right_on = "id")
serbia_admin = serbia_admin[serbia_admin.id.notna()]
serbia_admin['id'] = serbia_admin['id'].astype('int64')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

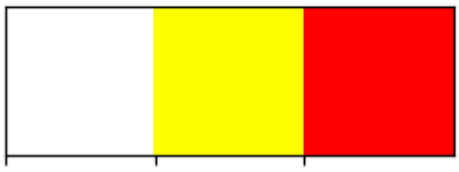
Built-in colour maps can be found here https://matplotlib.org/stable/gallery/color/colormap_reference.html. However one can create new ones as follows from a list of colours:

```
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)
```

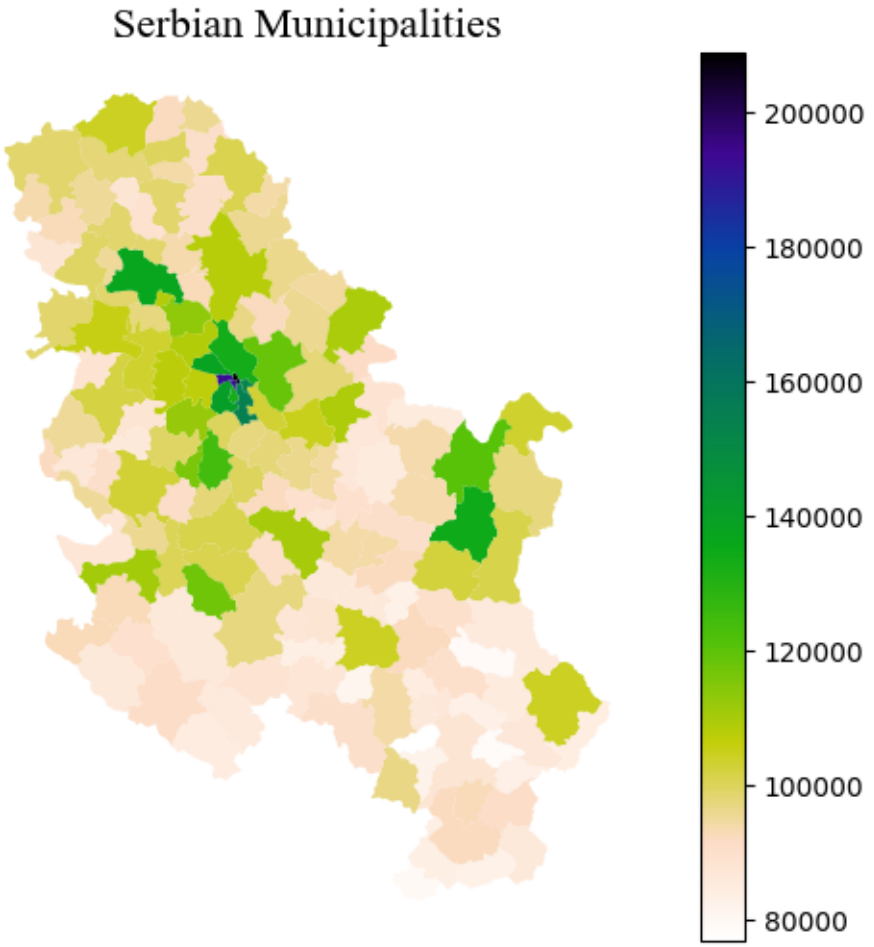


Let's try a new colormap and let's also set a number of classes to divide the data in, through the parameter `k`.

```
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

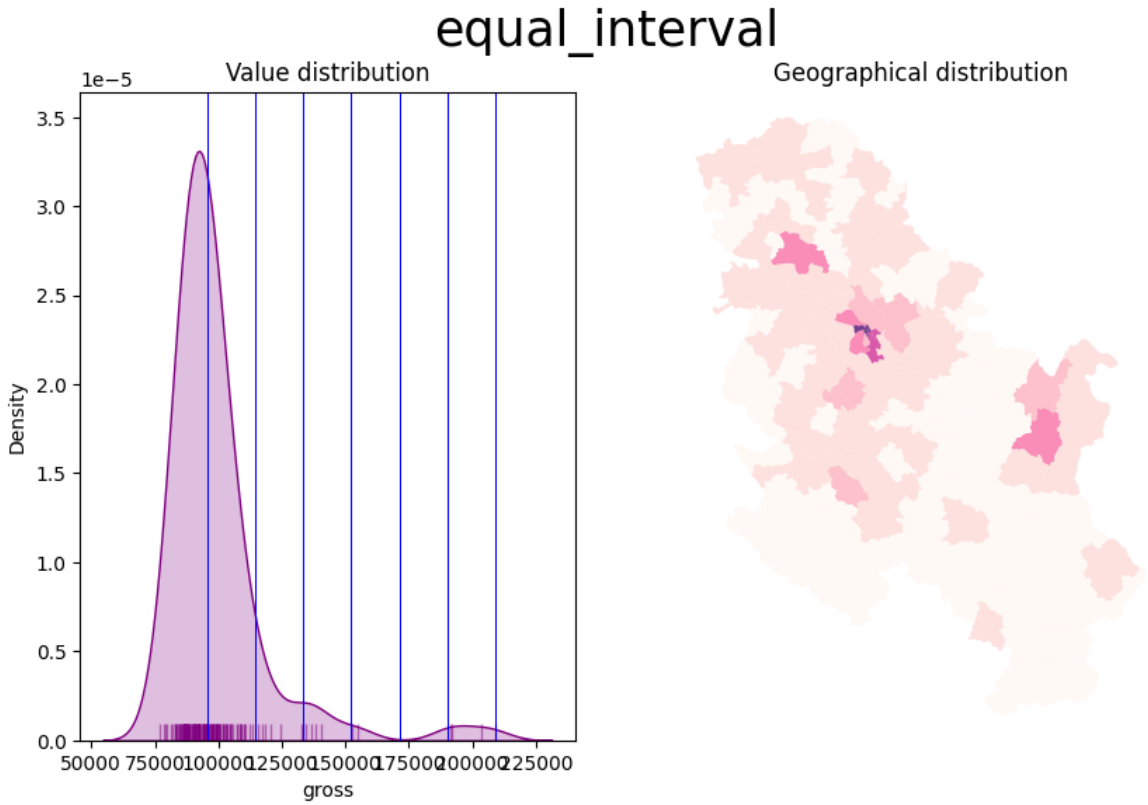
```
# Function for plotting the map and the distribution of the value in bins

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
```

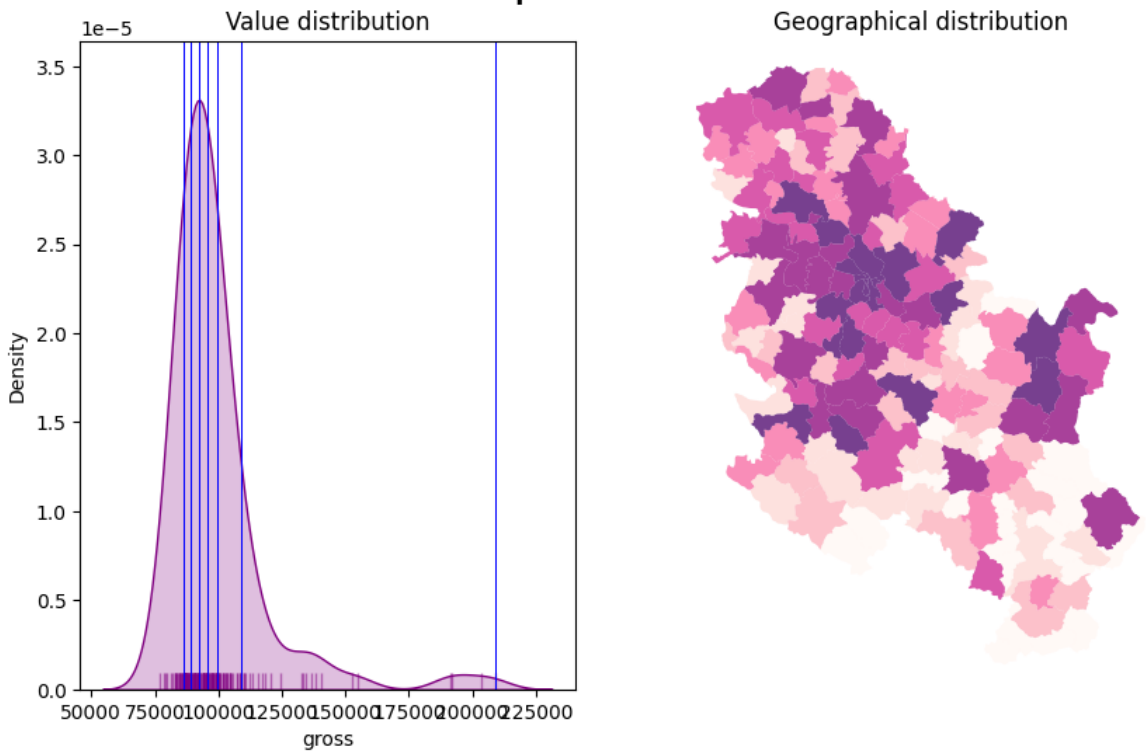
```
    sns.kdeplot(gdf[column], fill=True, color='purple', ax=ax1)
    sns.rugplot(gdf[column], alpha=0.5, color='purple', ax=ax1)
    for cut in classification.bins:
        ax1.axvline(cut, color='blue', linewidth=0.75)
    ax1.set_title('Value distribution')
    # Map
    p = gdf.plot(column=column, 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()
```

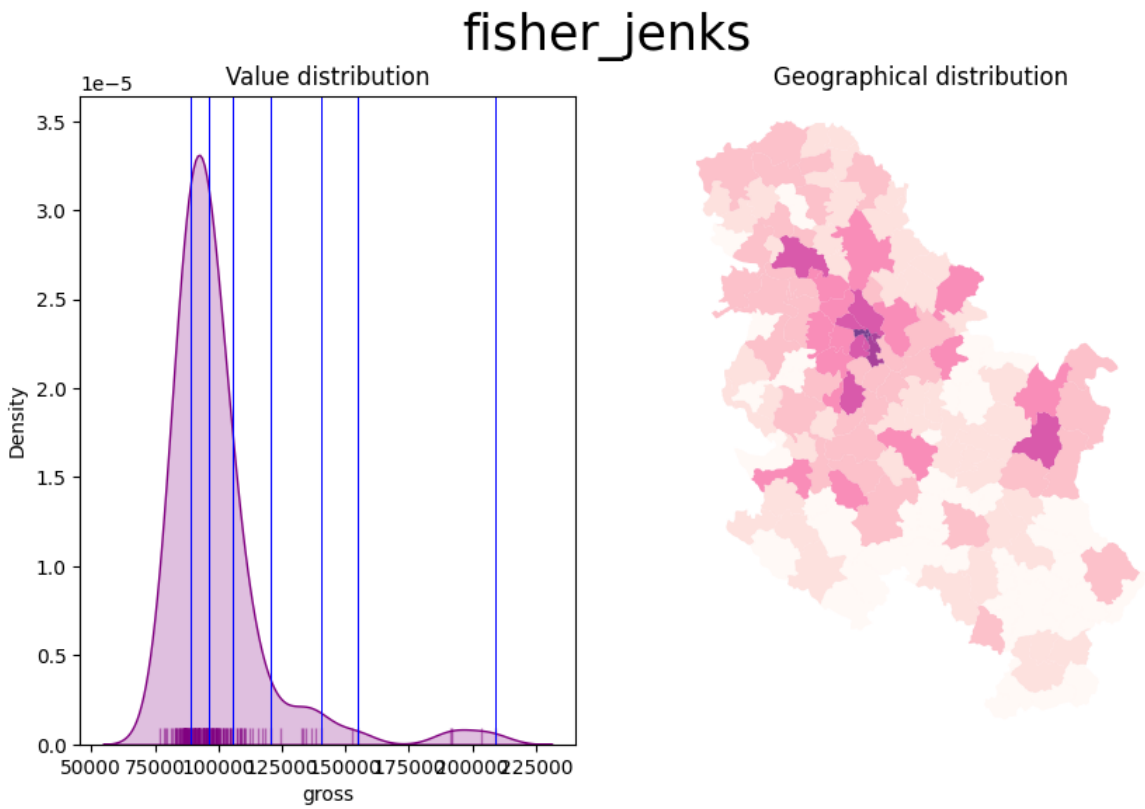
- The *Equal intervals* method splits the range of the distribution, the difference between the minimum and maximum value, into equally large segments and to assign a different colour to each of them according to a palette that reflects the fact that values are ordered.
- To obtain a more balanced classification, 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
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



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.

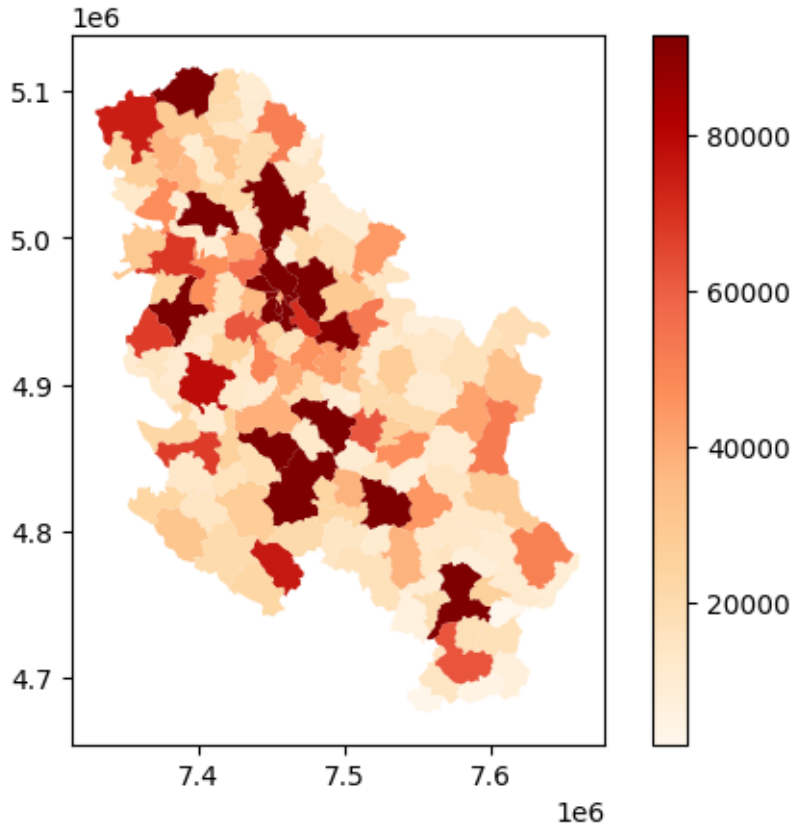
When employed in a plotting function, this normalization object ensures that the data values are scaled to fit a pre-defined range (for instance, `norm = mpl.colors.Normalize(vmin=0, vmax=40)`). Any values falling below 0 are mapped to the lowest colour on the colormap scale, while values exceeding 40 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)
```

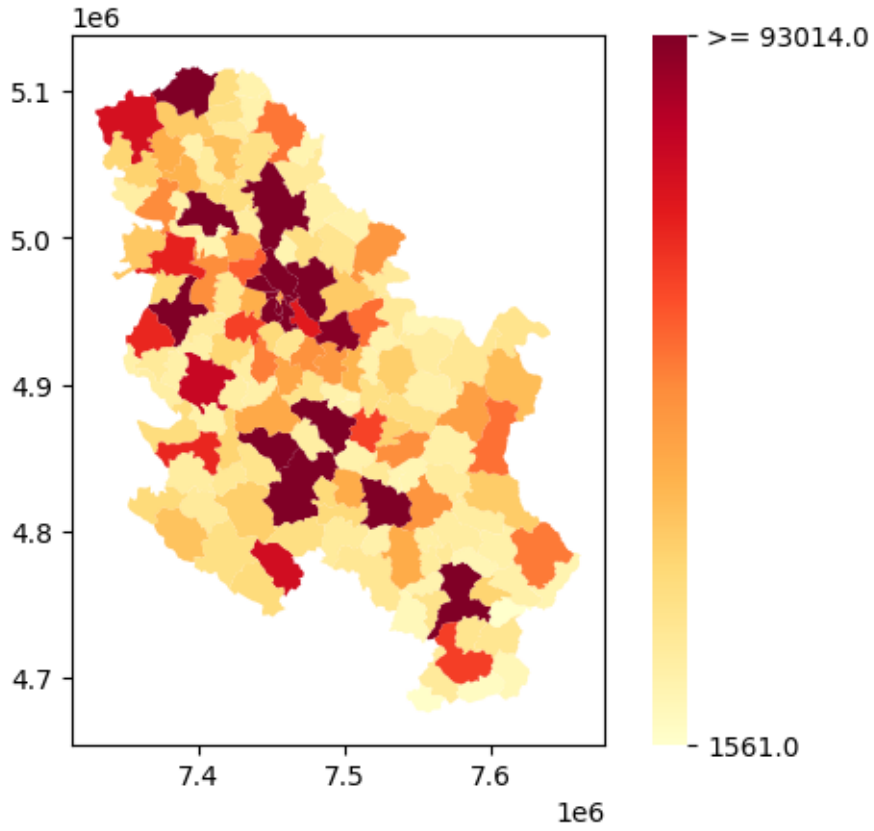


Important:

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.

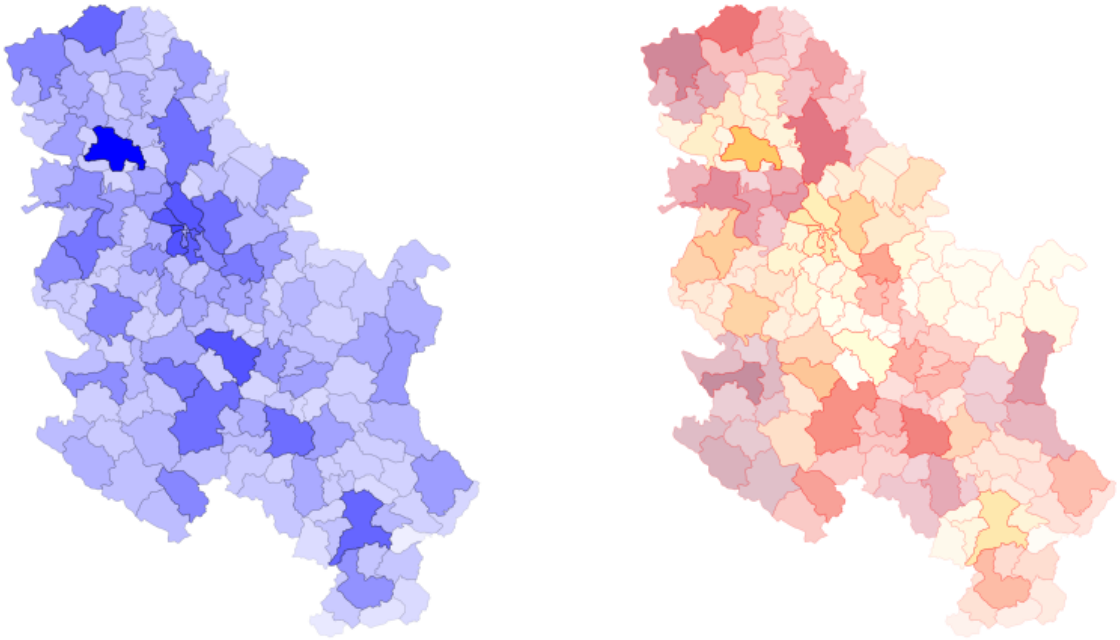
3.2.1.5 Varying alpha transparency based on an array

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:

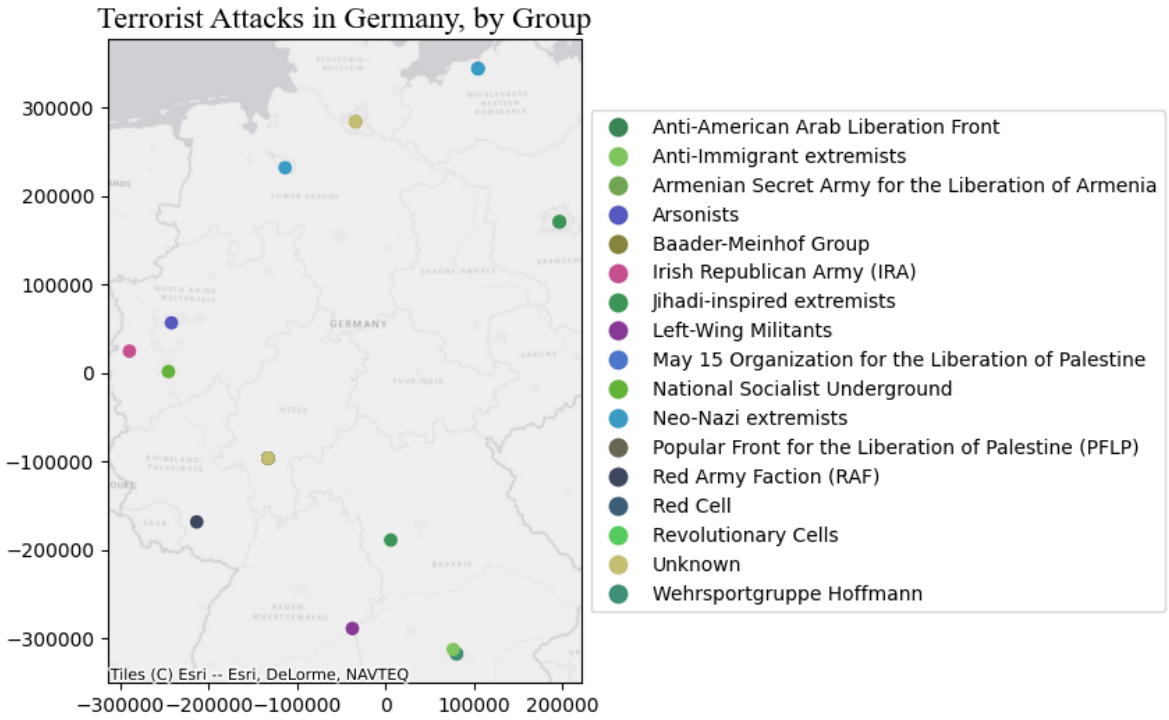
```
# Generate random colormap
def rand_cmap(nlabels):
    """
    It generates a categorical random color map, given the number of classes

    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.

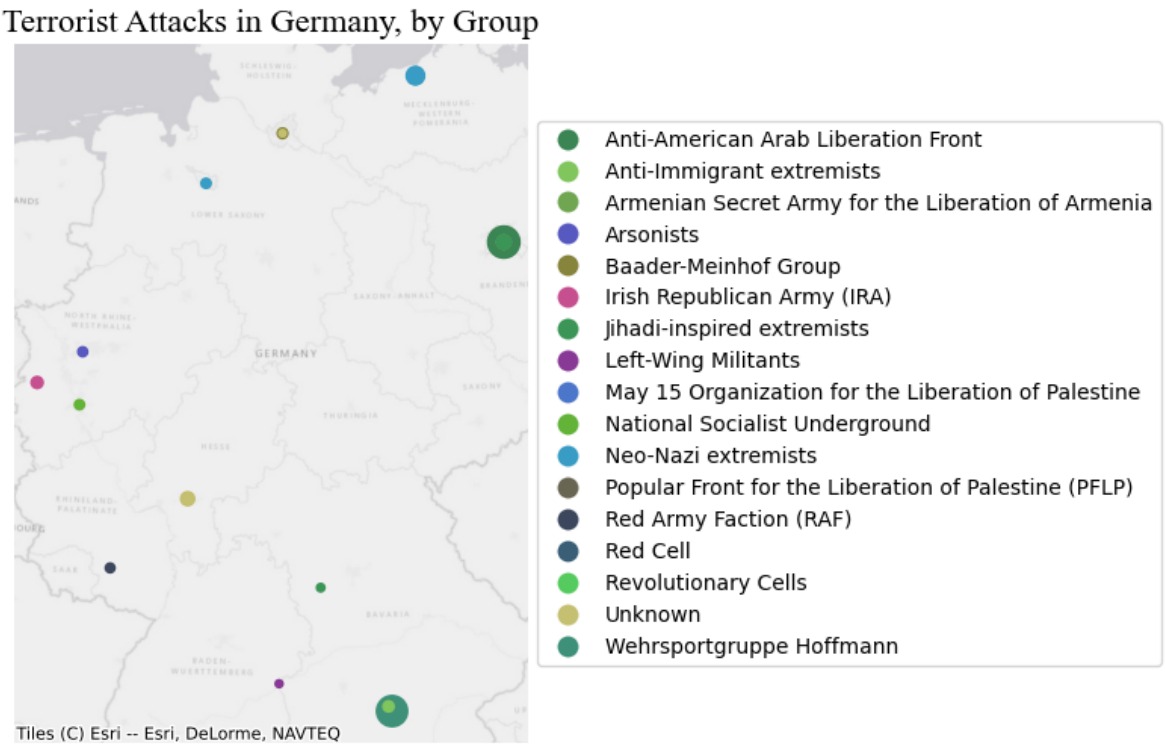
```
legend_kwds={"loc": "center left", "bbox_to_anchor": (1, 0.5)}
```

```
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)
```

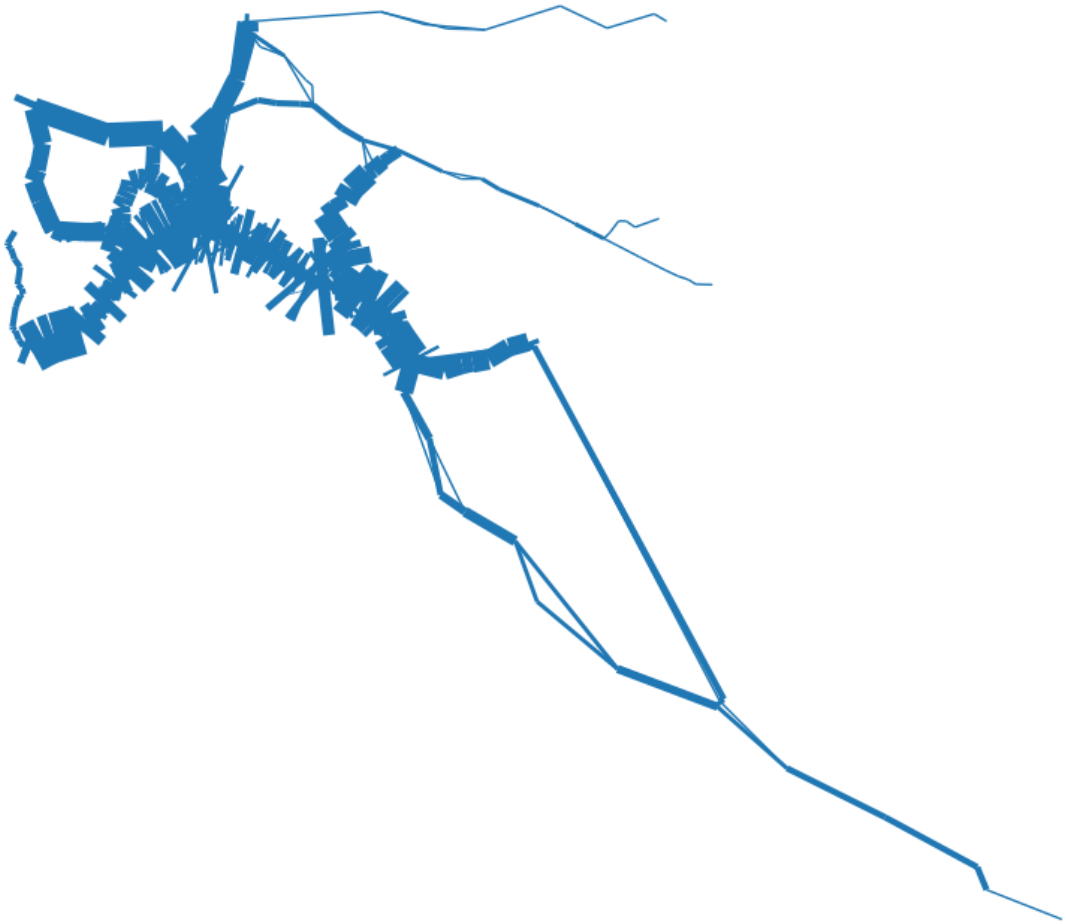
We have: - 'REG': regional trains. - 'IC': intercity trains. - 'FB': similar to IC, but slightly faster. - 'ICN': sleeper trains. - 'U': urban trains (Genoa). - 'EC/EN': international trains. - 'AV': High-speed trains.

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')
```

4 Web Architectures and APIs

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

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

4.1 What do APIs actually do?

In this lab, we will unpack how Application Programming Interfaces ("APIs") work and we cover the basics of accessing an API using Python. Instead of downloading a data set, APIs allow programmers, statisticians (or students) to request data directly from a server to a local machine. When you work with web APIs, two different computers — a client and server — will interact with each other to request and provide data, respectively.

4.1.1 RESTful Web APIs are all around you.

Web APIs

- Allow you query a remote database over the internet
- Take on a variety of formats
- Adhere to a particular style known as Representation State Transfer or REST (in most cases)
- RESTful APIs are convenient because we use them to query database using URLs

Consider a simple Google search:

Ever wonder what all that extra stuff in the address bar was all about? In this case, the full address is Google's way of sending a query to its databases requesting information related to the search term *liverpool top attractions*.

In fact, it looks like Google makes its query by taking the search terms, separating each of them with a +, and appending them to the link https://www.google.com/#q=. Therefore, we should be able to actually change our Google search by adding some terms to the URL:

Learning how to use RESTful APIs is all about learning how to format these URLs so that you can get the response you want.

4.1.2 Group activity

Get into groups of 5 or 6 students. Using your friend the internet, look up answers to the following questions. Each group will be assigned one question and asked to present their findings in 5 min to discuss with the entire class.

1. What is a `URL` and how can it help us query data? What is a response status and what are the possible categories?
2. What is a `GET` request? How does a `GET` request work?
3. What are API keys and how do you obtain them? What kinds of restrictions to they impose on users? Find an example of an API key, what does it look like?
4. (For 2 groups) More and more APIs pop up every day. Do a bit of quick research and find 2 different examples of APIs that you would be interested in using. 2 groups, 2 or 3 APIs each.

There are two ways to collect data through APIs in Python:

- **Plug-n-play packages.** Many common APIs are available through user-written Python (or R) Packages. These packages offer functions that conveniently "wrap" API queries and format the response. These packages are usually much more convenient than writing our own query, so it is worth searching for a package that works with the API we need.
- **Writing our own API request.** If no wrapper function is available, web have to write our own API request and format the response ourselves using Python. This is tricky, but definitely doable.

4.2 API Python libraries

Exercise: `census` pair activity:

Some Python packages "wrap" API queries and format the response. Lucky us! In pairs, let's have a look at `census`, a wrapper or the United States Census Bureau's API You can also have a look at the different APIs available from the United States Census Bureau.

```
import pandas as pd
from census import Census
```

To get started working, set you Census API key. A key can be obtained from http://api.census.gov/data/key_signup.html.

```
census_api_key = "" # Replace 'YOUR_CENSUS_API_KEY_HERE' with your actual Census API key.
```

```
# Set API key
c = Census(census_api_key)
```

- Variables in tidycensus are identified by their Census ID, e.g. B19013_001.
- Entire tables of variables can be requested with the table argument, e.g. table = 'B19001'.
- Users can request multiple variables at a time, and set custom names with a named vector.

In Python we can use the library `census` to access the American Community Survey (ACS) 5-Year Data (2016-2020)

Exercise:

In pairs explore some of the different variables available in the 5-Year ACS (2016-2020). Make a note of 3 variables you would be interested in exploring. The ACS2 variablespage might help.

Let's explore income data for example.

```
# Retrieve income data by state using the ACS table B19001
# Note: The variable 'B19001_001E' represents "Estimate!!Total" in table B19001
income_data = c.acs5.state(('NAME', 'B19001_001E'), Census.ALL, year=2020)
income_data[:10] # just first ten "rows"
```

```
[{'NAME': 'Pennsylvania', 'B19001_001E': 5106601.0, 'state': '42'},
 {'NAME': 'California', 'B19001_001E': 13103114.0, 'state': '06'},
 {'NAME': 'West Virginia', 'B19001_001E': 734235.0, 'state': '54'},
 {'NAME': 'Utah', 'B19001_001E': 1003345.0, 'state': '49'},
 {'NAME': 'New York', 'B19001_001E': 7417224.0, 'state': '36'},
 {'NAME': 'District of Columbia', 'B19001_001E': 288307.0, 'state': '11'},
 {'NAME': 'Alaska', 'B19001_001E': 255173.0, 'state': '02'},
 {'NAME': 'Florida', 'B19001_001E': 7931313.0, 'state': '12'},
 {'NAME': 'South Carolina', 'B19001_001E': 1961481.0, 'state': '45'},
 {'NAME': 'North Dakota', 'B19001_001E': 320873.0, 'state': '38'}]
```

```
income_df = pd.DataFrame(income_data)
income_df.head()
```

	NAME	B19001_001E	state
0	Pennsylvania	5106601.0	42
1	California	13103114.0	06
2	West Virginia	734235.0	54

	NAME	B19001_001E	state
3	Utah	1003345.0	49
4	New York	7417224.0	36

Exercise:

Discuss the format of the data obtained with your partner and then use the function `acs5.state` to explore the 3 variables you discussed in the previous exercise.

You can also get more variables by passing the names in a `tuple`. The code below, for example, fetches all the columns from the B19001 table, which include various income brackets. The tuple in the request generates a list of column names (like 'B19001_001E', 'B19001_002E', ..., 'B19001_017E').

```
variables = tuple(f'B19001_{str(i).zfill(3)}E' for i in range(1, 18))
```

4.2.0.1 This is what the line above does:

- `range(1, 18)`: This part generates a sequence of numbers from 1 to 17. In Python, range(start, stop) generates numbers from start up to but not including stop.
- `for i in range(1, 18)`: This is a loop within a comprehension that iterates over each number in the range from 1 to 17.
- `str(i).zfill(3)`: For each number `i`, this converts `i` to a string. Then, `.zfill(3)` pads the string with zeros to make it 3 characters long. For example, if `i` is 2, `str(i).zfill(3)` becomes '002'.
- `f'B19001_{str(i).zfill(3)}E'`: This is an `f-string`, a way to format strings in Python. It inserts the zero-padded string into a larger string. So, for i = 2, you would get 'B19001_002E'.
- Finally, the comprehension is wrapped in `tuple(...)`, which converts the entire series of strings into a tuple.

```
wide_data = c.acs5.state(('NAME',) + variables, Census.ALL, year=2020)
# Convert to a Pandas DataFrame
wide_df = pd.DataFrame(wide_data)
wide_df.head()
```

	NAME	B19001_001E	B19001_002E	B19001_003E	B19001_004E	B19001_005E	B19001_006E
0	Pennsylvania	5106601.0	296733.0	206216.0	223380.0	227639.0	226678.0
1	California	13103114.0	614887.0	507398.0	435382.0	474093.0	454373.0
2	West Virginia	734235.0	62341.0	43003.0	45613.0	44635.0	40805.0

	NAME	B19001_001E	B19001_002E	B19001_003E	B19001_004E	B19001_005E	B19001_006E
3	Utah	1003345.0	36211.0	27395.0	28460.0	32497.0	36116.0
4	New York	7417224.0	471680.0	340614.0	303901.0	298025.0	276764.0

Let's make our query a bit more precise. We are going to query data on median household income and median age by county in the state of Louisiana from the 2016-2020 ACS. We can use the library `us` to get the code of each of the state by passing their name.

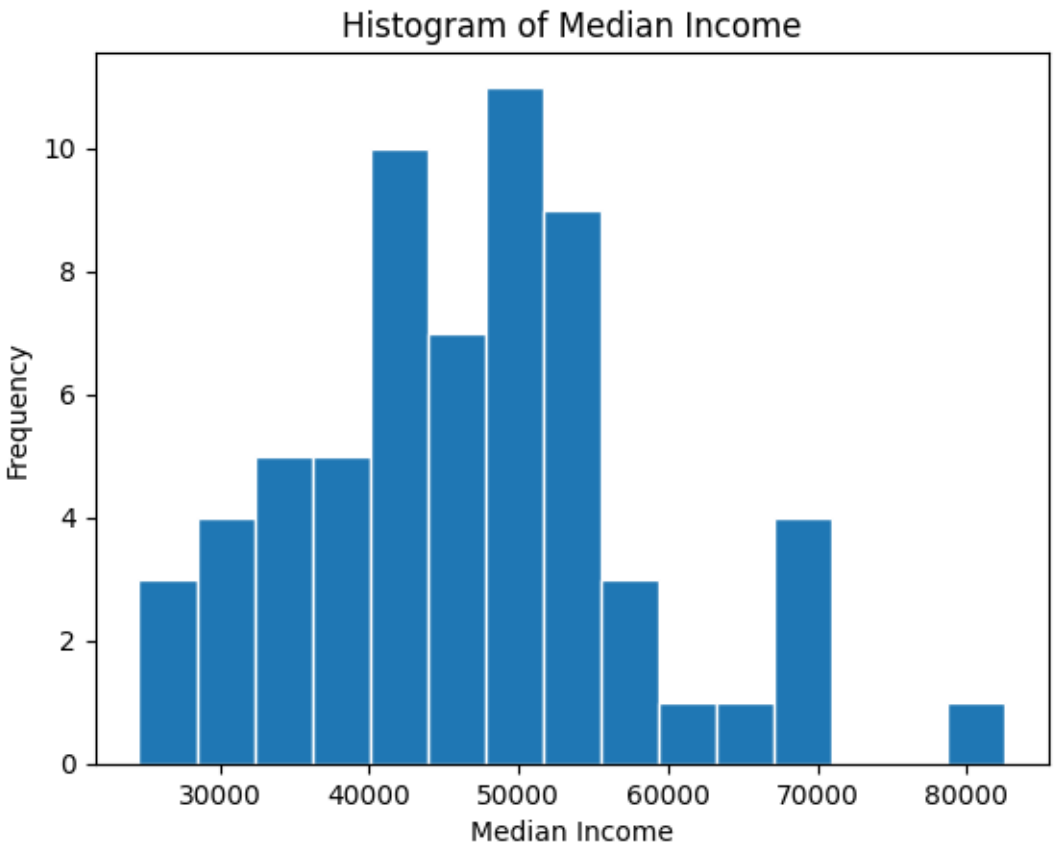
```
import us
# Get the state object for Louisiana
state_obj = us.states.lookup('Louisiana')
# Get the FIPS code (state code)
code = state_obj.fips
```

```
# Retrieve wide format data for median income and median age by county in Louisiana for the y
louisiana_data = c.acs5.state_county(('B19013_001E', 'B01002_001E'), code, Census.ALL, year=2
# Convert to a Pandas DataFrame
louisiana_df = pd.DataFrame(louisiana_data)
# Renaming the columns for clarity
louisiana_df.rename(columns={'B19013_001E': 'median_income', 'B01002_001E': 'median_age'}, in
louisiana_df.head()  # Display the first few rows
```

	median_income	median_age	state	county
0	82594.0	36.0	22	005
1	49256.0	37.5	22	011
2	42003.0	37.8	22	017
3	56902.0	44.9	22	023
4	36294.0	37.4	22	029

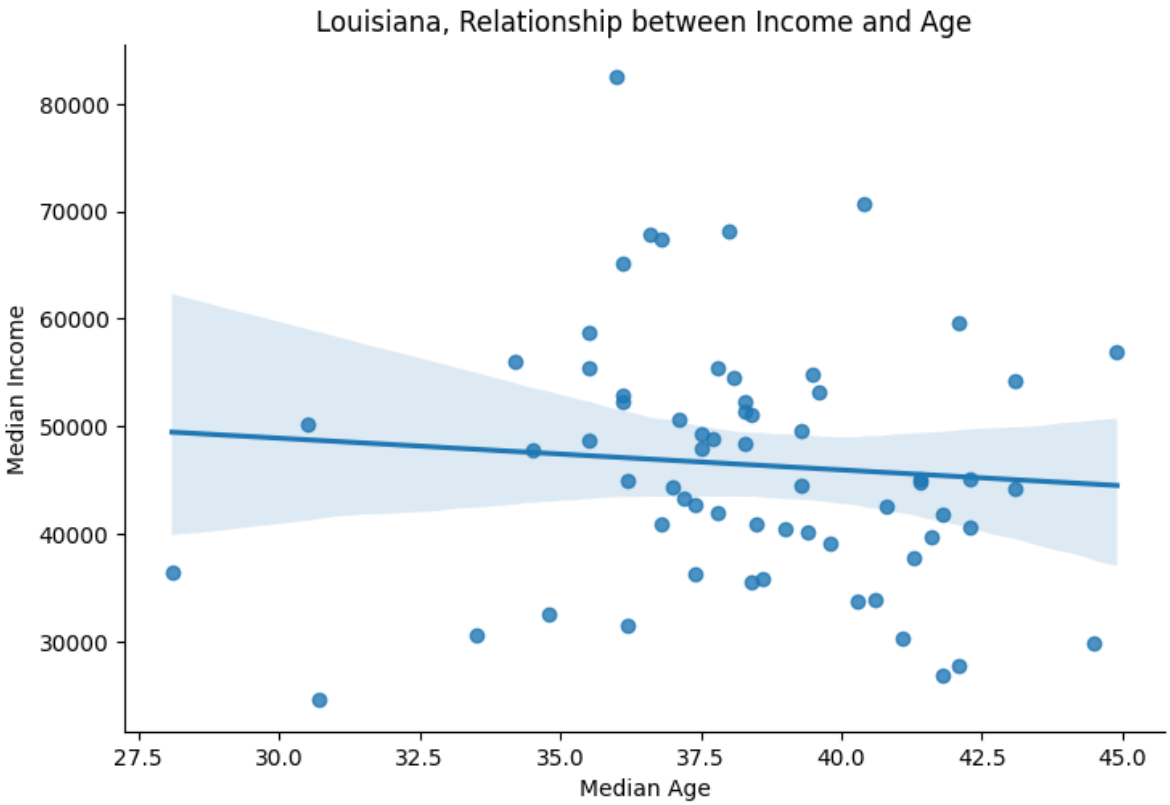
Let's plot one of our variables.

```
import matplotlib.pyplot as plt

# Plotting histogram for median income with 15 bins
plt.hist(louisiana_df['median_income'], bins=15, edgecolor='white')
plt.xlabel('Median Income')
plt.ylabel('Frequency')
plt.title('Histogram of Median Income')
plt.show()
```



We can also explore correlations between variables. Let's use `seaborn`.

```
import seaborn as sns

# Creating a scatter plot with a linear regression line
sns.lmplot(x='median_age', y='median_income', data=louisiana_df, aspect=1.5)
plt.xlabel('Median Age')
plt.ylabel('Median Income')
plt.title('Louisiana, Relationship between Income and Age')
plt.show()
```



Exercise:

In pairs, modify the state, variables and year parameters following the approach adopted above and produce some other simple scatter plots (cloud of points) that suggest correlations between your variables of interest.

4.3 Your own API request demo

The Python libraries commonly used for API requests are `requests` and `json`.

JSON stands for JavaScript Object Notation. Despite its association with JavaScript, JSON is widely used due to its readability and ease of use by computers, making it the primary format for data transmission through APIs. Most APIs return their responses in JSON format. The `json` library in Python allows you to parse and convert these JSON responses into Python data structures. JSON structures are composed of key-value pairs, similar to Python dictionaries.

In Python, to make an API request and handle the response, you would typically use the `requests` library. A standard API request involves sending a `GET` request to the server's URL, which specifies the data you wish to retrieve. For instance, to request the locations of

all the hire bike stations in London from the **Transport for London** API, you would use the `requests.get()` method. This method requires a URL that directs the request to the appropriate server.

```
import requests
import json

response = requests.get("https://api.tfl.gov.uk/BikePoint/")
response
```

```
<Response [200]>
```

Good! The response code 200 indicates a successful request

Most GET request URLs for API querying have three or four components:

1. Authentication Key/Token: A user-specific character string appended to a base URL telling the server who is making the query; allows servers to efficiently manage database access.
2. Base URL: A link stub that will be at the beginning of all calls to a given API; points the server to the location of an entire database.
3. Search Parameters: A character string appended to a base URL that tells the server what to extract from the database; basically, a series of filters used to point to specific parts of a database.
4. Response Format: A character string indicating how the response should be formatted; usually one of .csv, .json, or .xml.

```
# Making the API request
response = requests.get("https://api.tfl.gov.uk/BikePoint/")

# Parsing the response content as JSON and converting it to a DataFrame
bike_stations = pd.DataFrame(response.json())

# Printing the column names
print(bike_stations.columns)
```

```
Index(['$type', 'id', 'url', 'commonName', 'placeType', 'additionalProperties',
       'children', 'childrenUrls', 'lat', 'lon'],
      dtype='object')
```

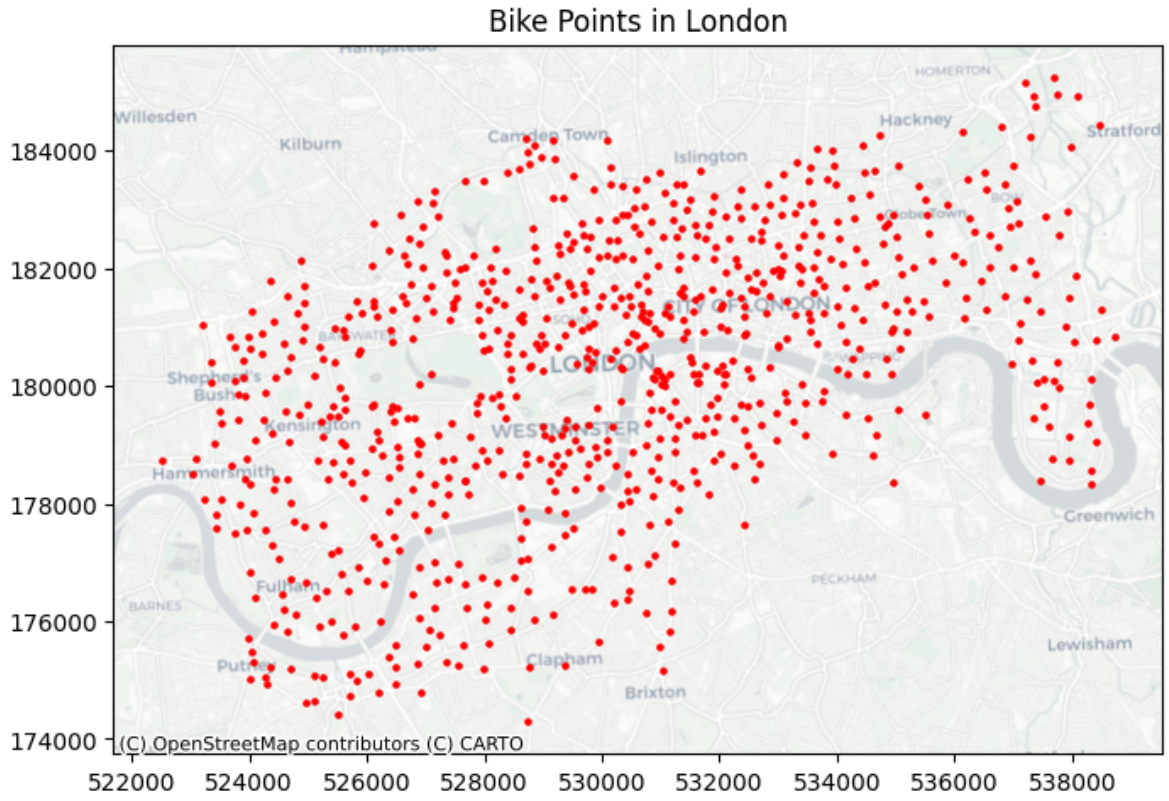
```
# Creating a new column 'Station ID' by extracting the numeric part from the 'id' column
bike_stations['Station ID'] = bike_stations['id'].str.extract(r'BikePoints_(\d+)', expand=Fal
bike_stations.head()
```

	$type	id	url	commonName	placeType	additionalProperties	children	childrenUrls	lat	lon	Sta
0	Tfl.Api.Presentation.Entities.Place, Tfl.Api.P...	BikePoints_1	/Place/BikePoints_1	River Street , Cle							
1	Tfl.Api.Presentation.Entities.Place, Tfl.Api.P...	BikePoints_2	/Place/BikePoints_2	Phillimore Garden							
2	Tfl.Api.Presentation.Entities.Place, Tfl.Api.P...	BikePoints_3	/Place/BikePoints_3	Christopher Stree							
3	Tfl.Api.Presentation.Entities.Place, Tfl.Api.P...	BikePoints_4	/Place/BikePoints_4	St. Chad's Street							
4	Tfl.Api.Presentation.Entities.Place, Tfl.Api.P...	BikePoints_5	/Place/BikePoints_5	Sedding Street, Sl							

By now you should be able to visualise the data that you have obtained through this API.

```
%matplotlib inline
import geopandas as gpd
import contextily as ctx

# Convert DataFrame to GeoDataFrame
gdf = gpd.GeoDataFrame(bike_stations, geometry=gpd.points_from_xy(bike_stations.lon, bike_sta
# Setting the Coordinate Reference System (CRS) to WGS84 (EPSG:4326)
gdf.set_crs(epsg=4326, inplace=True)
# project to British Grid
gdf.to_crs(epsg= 27700, inplace = True)

# Plotting bikepoints on a map
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
gdf.plot(ax = ax, color='red', markersize=5)
source = ctx.providers.CartoDB.Positron
ctx.add_basemap(ax, crs= gdf.crs.to_string(), source= source)
ax.set_title('Bike Points in London')
```

```
Text(0.5, 1.0, 'Bike Points in London')
```



4.4 Geocoding API

Below is a short exploration of a the `geocoder` provided by the library `geopy`. The `geocoder` relies on various external geocoding services. When using `Nominatim`, it accesses data from OpenStreetMap. OpenStreetMap's data includes a vast array of geographical information sourced from contributors globally. This includes street addresses, points of interest, and other location-based data. When you provide an address to the Nominatim geocoder, it queries OpenStreetMap's databases to find the corresponding geographical coordinates.

In your own time, try to use it to automatically embed coordinates between addresses.

```
from geopy.geocoders import Nominatim

# Create a DataFrame with addresses
some_addresses = pd.DataFrame({
    'name': ["South Campus Teaching Hub", "Sefton Park", "Stanley Street"],
    'addr': ["140 Chatham St, Liverpool L7 7BA", "Sefton Park, Liverpool L17 1AP", "4 Stanley
})
```

```
# Initialize geocoder
geolocator = Nominatim(user_agent="geoapiExercises")

# Function for geocoding
def geocode(address):
    location = geolocator.geocode(address)
    return pd.Series([location.latitude, location.longitude], index=['latitude', 'longitude']

# Geocode the addresses
lat_longs = some_addresses['addr'].apply(geocode)

# Adding the results back to the DataFrame
some_addresses = some_addresses.join(lat_longs)
some_addresses
```

	name	addr	latitude	longitude
0	South Campus Teaching Hub	140 Chatham St, Liverpool L7 7BA	53.400188	-2.964437
1	Sefton Park	Sefton Park, Liverpool L17 1AP	53.389846	-2.923999
2	Stanley Street	4 Stanley St, Liverpool L1 6AA	53.407363	-2.987240

4.4.0.1 Reverse Geocoding

You can also reverse geo-code the data, open the output and see what the result is.

```
# Reverse geocoding (optional)
def reverse_geocode(lat, lon):
    location = geolocator.reverse((lat, lon))
    return location.address

# Apply reverse geocoding
some_addresses[['latitude', 'longitude']].apply(lambda x: reverse_geocode(x['latitude'], x['
```

```
0    148, Chatham Street, Canning / Georgian Quarte...
1    Sefton Park, Smithdown Road, Wavertree, Liverp...
2    Davies Street, Cavern Quarter, Liverpool, Live...
dtype: object
```

4.4.0.2 Geographic Data through APIs and the web

APIs can help us generate and create spatial data. In this lab, we have both:

- Written **our own API request.**
- Used **Plug-n-play packages.**

There are many APIs where we can get data these days. A few other examples include

- The London DataStore API
- Thames Water API
- London Air API
- Crime data

4.5 *Group activity answers*

1. Uniform Resource Location (`URL`) is a string of characters that, when interpreted via the Hypertext Transfer Protocol (`HTTP`). URLs point to a data resource, notably files written in Hypertext Markup Language (`HTML`) or a subset of a database.

   - 1xx informational response - the request was received, continuing process
   - 2xx successful - the request was successfully received, understood, and accepted
   - 3xx redirection - further action needs to be taken in order to complete the request
   - 4xx client error - the request contains bad syntax or cannot be fulfilled
   - 5xx server error - the server failed to fulfil an apparently valid request

2. `GET` requests a representation of a data resource corresponding to a particular `URL`. The process of executing the `GET` method is often referred to as a `GET request` and is the main method used for querying RESTful databases. `HEAD`, `POST`, `PUT`, `DELETE`: other common methods, though mostly never used for database querying.

   Surfing the web is basically equivalent to sending a bunch of `GET` requests to different servers and asking for different files written in `HTML`. Suppose, for instance, I wanted to look something up on Wikipedia. Your first step would be to open your web browser and type in http://www.wikipedia.org. Once you hit return, you would see the page below. Several different processes occurred, however, between you hitting "return" and the page finally being rendered:

   1. The web browser took the entered character string, used the command-line tool "Curl" to write a properly formatted `HTTP GET` request, and submitted it to the server that hosts the Wikipedia homepage.
   2. After receiving this request, the server sent an `HTTP` response, from which `Curl` extracted the HTML code for the page (partially shown below).

3. The raw `HTML` code was parsed and then executed by the web browser, rendering the page as seen in the window.
4. Most APIs requires a key or other user credentials before you can query their database. Getting credentials for an API requires that you register with the organization. Once you have successfully registered, you will be assigned one or more keys, tokens, or other credentials that must be supplied to the server as part of any API call you make. To make sure users are not abusing their data access privileges (e.g., by making many rapid queries), each set of keys will be given rate limits governing the total number of calls that can be made over certain intervals of time.

3. Most APIs requires a key before you can query their database. This usually requires you to register with the organization. Most APIs are set up for developers, so you will likely be asked to register an "application." All this really entails is coming up with a name for your app/bot/project and providing your real name, organization, and email. Note that some more popular APIs (e.g., Twitter, Facebook) will require additional information, such as a web address or mobile number. Once you have registered, you will be assigned one or more keys, tokens, or other credentials that must be supplied to the server as part of any API call you make. Most API keys limits the total number of calls that can be made over certain intervals of time. This is so users do not busing their data access privileges.

4.6 References

- Brief History of the Internet, by the Internet Society, is a handy (and free!) introduction to how it all came to be.
- Haklay, M., Singleton, A., Parker, C. 2008. "Web Mapping 2.0: The Neogeography of the GeoWeb". Geography Compass, 2(6):2011–2039
- A blog post from JoeMorrison commenting on the recent change of licensing for some of the core software from Mapbox
- Terman, R., 2020. Computational Tools for Social Science

5 Data Architectures and Tiles

5.1 Intro

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

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

In this lab, we will explore and familiarise with some of the most common data formats for web mapping: GeoJSON and Mbtiles.

5.2 GeoJSONs

To get familiar with the format, we will start by creating a GeoJSON file from scratch. Head over to the following website:

https://geojson.io/

In there, we will create together a small example to better understand the building blocks of this file format.

We will pay special attention to the following aspects:

- Readability.
- Coordinate system.
- Ability to add non-spatial information attached to each record.
- How to save it as a file.

Excercise:

Create a GeoJSON file for the following data and save them to separate files:

1. Your five favourite spots in Liverpool
2. A polygon of what you consider to be the boundary of the neighbourhood where you live and the city centre of Liverpool. Name each.
3. A route that captures one of your favourite walks around the Liverpool region

If you are comfortable, upload the files to Microsoft Teams to share them with peers.

5.2.1 GeoJSON in Python

With the files from the exercise at hand, we will learn how to open them in a Python environment. Then, let's begin by importing the necessary libraries; `geojson` is used for handling GeoJSON files.

```
import geopandas as gpd
```

Now, place the geojson files you have created in the data folder used in these sessions. As always, the data folder should be stored in the directory where the notebook is running from. For this example, we will assume that the file is called `map.geojson`. We can read the file as:

```
liverpool = gpd.read_file("../data/map.geojson")
liverpool.head()
```

	geometry
0	POINT (-2.96320 53.40471)
1	POINT (-2.96206 53.40471)
2	POINT (-2.96513 53.40163)
3	LINESTRING (-2.96510 53.40163, -2.96505 53.402...
4	POLYGON ((-2.97048 53.40888, -2.96968 53.40573...

We can also plot and explore the content of the GeoDataFrame with `Folium`. Folium, which we will see more in detail later on, helps create interactive maps from data stored in `geopandas.GeoDataFrame`.

```
import folium

liverpool_centroid = (53.41058, -2.97794)
# Create a Folium map centered around this point
map = folium.Map(location=liverpool_centroid, zoom_start=13, tiles="CartoDB.DarkMatterNoLabel

# Add the liverpool data to the map, this will plot each geometry in the GeoDataFrame
folium.GeoJson(liverpool).add_to(map)
map
```

```
<folium.folium.Map at 0x2684e976cd0>
```

Once read, the geojson behaves exactly like any GeoDataFrame we have seen so far. We can therefore operate on it and tap into the functionality from `pandas` and `geopandas`. For example, we can and reproject the layer to the to British National Grid.

```
liverpool_bng = liverpool.to_crs(crs = "EPSG: 27700")
```

When we inspected our geojson, we noted that the spatial data is stored in the following format `POINT (-2.977367 53.40753)`. This is called "well known text" (`wkt`) and is a representation that spatial databases like PostGIS use as well. Another way to store spatial data as text for storage or transfer, less (human) readable but more efficient is the "well known blurb" (`wkb`). We can use the `shapely` library to handle the WKT representation of the geometry and then convert it to WKB format.

```
from shapely import wkt
from shapely.geometry import Point
import shapely.wkb

# Load the WKT representation of the point
wkt_string = "POINT (-2.977367 53.40753)"

# Convert the WKT representation into a Shapely Point object
point = wkt.loads(wkt_string)

# Convert the Point object into WKB format
wkb_data = shapely.wkb.dumps(point)
wkb_data.hex()
```

```
'01010000005e8429caa5d107c0c7116bf129b44a40'
```

Excercise: - Read the `GeoJSON` created for your favorite walks in Liverpool and calculate their length

Once you are happy with the data as we will hypothetically need it, you can write it out to any other file format supported in `geopandas`. For example, we can create a Geopackge file with the same information. For this, we can use the function `to_file`. See an example below:

```
# Write 'liverpool_bng' to a GeoPackage file
liverpool_bng.to_file("../data/liverpool_bng.gpkg", layer="liverpool_bng", driver="GPKG")
```

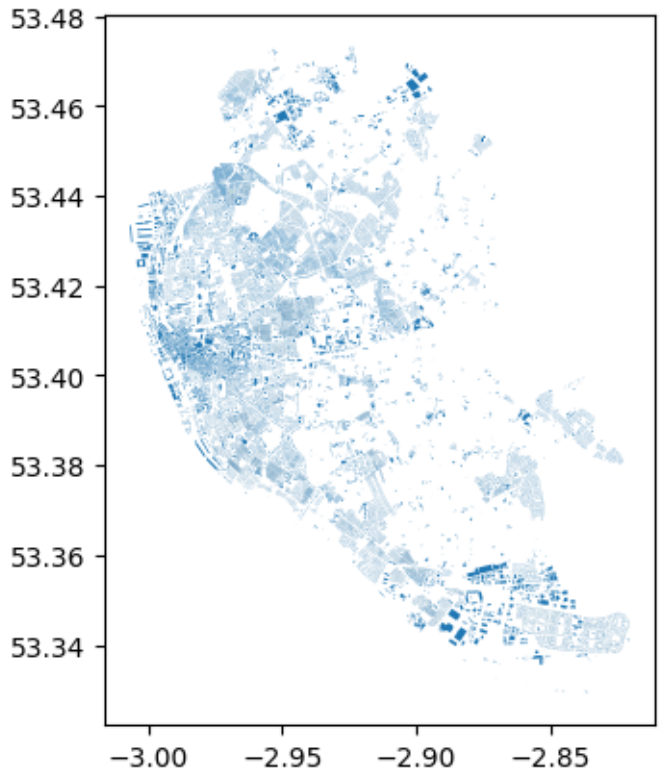
5.3 Tilesets

In this section we will dive into the concept of tiles to understand why they have been so transformative in the world of web mapping. We have already seen the usage of tilesets above with `folium` and with `contextily` (although within a static context). We will see that `folium`, integrates different tileset options already.

For this section, let's start by getting the building footprints from OpenStreetMap with `osmnx`

```
import osmnx as ox
tags = {"building": True} #OSM tags
buildings = ox.features_from_place("Liverpool, UK", tags = tags)
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.plot()
```



Let's save the GeoDataFrame a geojson and call it `buildings_liverpool.geojson`.

```
buildings[['osmid','geometry']].to_file('../data/buildings_liverpool.geojson',  driver='GeoJ
```

5.4 Mapbox

Register for a MapBox account here.

Once your account is created and verified, Save both your username and public token as variables below:

```
mapbox_user = '' #Add your username here
mapbox_token = '' #Add your public token here (find this on the right hand side of the Mapbo
```

5.5 Optional: Generating .mbtiles

Note that this section is optional for those who are interested in trying out more advanced data formats. It is not required for this lab or the assignment

IMPORTANT: This step cannot be ran on University Machines as the installation requires admin rights. Please skip to the next section - 'Mapbox Studio'

.mbtiles is a tile based data format, designed to assist the user in "making a scale-independent view of your data, so that at any level from the entire world to a single building, you can see the density and texture of the data rather than a simplification from dropping supposedly unimportant features or clustering or aggregating them. [Source]"

The following steps require the installation of `tippecanoe` on your machine. This cannot be installed directly onto Windows devices, and will require the installation of a local linux environment onto your device if you intend to use it.

5.5.1 MAC or Linux Devices (Python)

In Python, the `togeojsontiles` package allows for us to convert our geojsons into .mbtiles files within our python code. Ensure you have installed **all** the requirements (ensuring that `togeojsontiles` is installed within the envs456 environment, not *base* and that 'tippecanoe' is properly set up from here), before trying the code below: