

Creating Maps in Python using GeoPandas

A Short Introduction

Andrzej A. Romaniuk

Academics often create maps to illustrate their research, visualize areas of interest, or present data in a way that is easier to understand. To do this effectively without spending too much time, they need simple and accessible tools and well-explained mapmaking workflows. This guide introduces example workflows using GeoPandas and related libraries, offering a practical way to create maps entirely in Python. It is designed for researchers lacking any experience in cartography, helping them avoid the complexity of specialized mapping software while still producing complex and useful visualizations.

Table of contents

1 A Short Introduction	2
2 What exactly is (geo)spatial data?	3
3 Where to Download Such Data?	5
4 GeoPandas, a go-to Python Library for Map Creation	7
5 Setup, Importing Data, Basic Plotting and Exporting Results	10
6 Creating Multi-Layered Maps	16
7 Creating Choropleth Maps	27
8 Interactive Maps With Folium	38
9 Where To Look for Further Resources	44
10 A few Words at the End	45



1 A Short Introduction

Regardless of whether we discuss botanical studies, archaeological investigations, or linguistic research, maps are a common feature in analytical reports, research papers, and conference presentations. Visualising data in this form can greatly help anchor often abstract rows of numbers into a context that is more understandable to both the researcher and their audience. When investigating specific regional conditions (e.g., forest coverage, average income per region, vaccination levels or period-specific archaeological sites) academics frequently work with spatial data, such as geographic locations or regional boundaries.

But how can one easily recreate, for example, a choropleth map where each county's colour represents, on a continuous scale, the percentage of people who voted in the latest elections? Or pinpoint the rough geographic locations of several sampling sites on a map? Or perhaps simply display the outlines of selected countries relevant to the discussion at hand? And beyond that, where can one find publicly available data to plot maps?

Thankfully, we have simple and intuitive solutions for creating and customising complex maps in Python, with appropriate geospatial data freely available online. For non-specialist audiences interested in mapmaking the most useful is *GeoPandas*, a package that helps both with import-ing geospatial data to Python environment and working with such data, including visualisation. *GeoPandas* also work well with other libraries to expand what is available for their users, including e.g. *Folium* for interactive html maps.

What this guide is about?

This guide will help you understand the standard workflow when creating maps for research work, presentations and publications. It will introduce you to the spatial data required for map generation (formats, standards, online resources), the libraries that simplify working with these files in Python, and showcase simple map generation as well as examples of complex, multilayered maps - including ways of dealing with common problems when generating those maps. However, it does not aim to be an exhaustive source of options and solutions, as there are plenty of resources already available online to help you with understanding the range of possibilities in Python-based mapmaking - the last chapter will point you to additional online resources for further learning.

Requirements

For the sake of this guide, I assume you have a **basic understanding of Python** as a programming tool, and a **Python environment installed** in which to work (e.g., Jupyter Notebook, PyCharm, IDLE, VS Code). If not, please first check, for example, one of the Python lessons provided by The Carpentries initiative, for example [this one](#). If you work from an university account you will likely have to ask your system administrator to install necessary software for you.

2 What exactly is (geo)spatial data?

To put it briefly, geospatial data refers to any information tied to a specific location on Earth, such as **geographic coordinates** (latitude and longitude) and **related spatial references** (e.g., addresses, postal codes, or administrative boundaries). If we want to create any type of map, we need such data.

Geospatial data can be stored in simple CSV or Excel files, as the absolute minimum requirement for Python to generate geometry is knowing the X and Y coordinates of the points representing objects, and coding/name for these objects. However, there are also many vector (which use points, lines, and polygons) and raster (which use a cell grid, like an image) data formats specifically designed for storing such information. Among them there are three vector standards worth knowing, with first two used through this guide:

1) **Shapefile**

Shapefile is a common geospatial vector data format used in geographic information system (GIS) software. It stores the geometry (vector data) of geographical features and associated attributes in multiple files, often zipped together for convenience: *.shp* (the main file containing geometry), *.shx* (index file for fast access), and *.dbf* (tabular information linked to shapes). Shapefiles are often used to represent boundaries of regions (e.g., countries, states, census tracts) and transportation networks. While not an open standard, as the original developer still holds rights to it and publicly maintains the format till today, the shapefile format is widely used in GIS work due to all dedicated software supporting shapefiles.

2) **GeoJSON**

A good alternative to the aging shapefile is *GeoJSON*, a widely used format for encoding various geographic data structures in *JavaScript Object Notation* (JSON). It is lightweight, possible to read by a human, and well-suited for modern web mapping applications. Additionally, GeoJSON is an open, publicly available standard, defined by a community-driven organization (IETF). As a result, it is not tied to a particular software ecosystem, but compatible with multiple platforms, and remaining a widely supported and robust format for geospatial data.

3) **Keyhole Markup Language (KML)**

Keyhole Markup Language may be known to some people due to its use in online mapping software, such as Google Earth or Google Maps. It is an XML-based file format that can store geographic locations (points, lines, polygons) and attributes (names, descriptions, and style information). KML files can be very detailed but tend to take up more memory space compared to other formats like GeoJSON or Shapefile, restricting their utilization to either cloud services or generating smaller maps.

International Standardisation of Boundary Data

Geographical and administrative boundaries can be drawn in various ways, with the latter especially varying from country to country. This posed challenges for national and international administrative work, statistical analysis, and modern cartography in general. As a result, a number of standardized classification systems were introduced to store appropriate boundary data. The most well-known example is the **Nomenclature of Units for Territorial Statistics** (NUTS) system, developed in the 1970s for EU countries (see [here](#)).

NUTS consists of three levels, each representing an increasing number of entities at finer scales: - **First level (NUTS 1):** Major administrative regions, usually the largest within a country (e.g., Bavaria in Germany). - **Second level (NUTS 2):** Standard level for regional policy applications, reflecting sub-regions within the major regions of NUTS 1 (e.g., Upper Bavaria). - **Third level (NUTS 3):** Districts, counties, or smaller administrative units below NUTS 2 (e.g., Munich).

The UK itself used the NUTS system until quite recently. However, it is now transitioning to its own **International Territorial Level** (ITL) system, see [Wikipedia entry](#). ITL appears to be designed primarily for comparisons at the Organisation for Economic Co-operation and Development (OECD) level, which is essentially international in scope. As a result, ITL level 3 is more similar to NUTS level 2, meaning it may not always be suitable for finer-scale analysis, especially in northern Scotland.

The ITL levels are as follows: - **First level (ITL 1):** Large territorial units, such as countries or large regions. - **Second level (ITL 2):** Intermediate territorial units, such as provinces, states, or large administrative regions. - **Third level (ITL 3):** Small territorial units, including cities, towns, districts, or local administrative areas.

3 Were to Download Such Data?

As geospatial data are widely used across various specializations and for different purposes, resources are commonly available across the internet. Government portals tend to be the most reliable sources, though there are also initiatives providing resources for the public benefit.

1) **Natural Earth** www.naturalearthdata.com/downloads/

Natural Earth is a public domain map dataset offering free geographic data at three scales: 1:10m, 1:50m, and 1:110m (with “m” denoting millions). It is widely used by cartographers, GIS professionals, and researchers to create maps and conduct spatial analysis. The dataset includes both vector and raster data, covering political boundaries, physical features, cultural landmarks, and more. It is completely free to use, including for commercial purposes. However, it has a very specific policy towards [disputed regions](#), favouring ‘de facto’ ownership to ‘de jure’ international ruling may restrict current usefulness of their datasets.

2) **ONS Open Geography Portal** geoportal.statistics.gov.uk/

The Open Geography Portal, provided by the Office for National Statistics (ONS), is the UK’s official platform for distributing free and open access geographic data and resources. It offers a wealth of different data, especially valuable for those working with UK-specific datasets. All content is available under the Open Government Licence v3.0, ensuring free usage and redistribution.

3) **EU GISCO system** ec.europa.eu/eurostat/web/gisco/overview

The Geographic Information System of the Commission (GISCO), delivered by Eurostat, is designed to provide EU members with geographic information at EU, country, and regional levels. It is an invaluable resource for geospatial data and provides compatible information compiled over the years from each member state. Moreover, it provides primarily internationally established and recognised borders, what simplifies regular mapping.

4) **Regional Example: National Register of Boundaries** www.geoportal.gov.pl/en/

Exploring local repositories for appropriate data is always worthwhile. The Polish National Register of Boundaries is a great resource for anyone working with data within Polish borders and looking for boundary information. It’s also an excellent example of a repository designed with no prior experience in downloading such data required.

If needed: Compatible datasets

Additionally, there is a plenty of data online that can be already used with geospatial data, either due to already containing coordinates that can be plotted on the map or through including standards such as ITL or NUTS. One good example is [UK Data Service](#), with their [ReShare](#)

database containing multiple data repositories, freely available for either non-commercial (CC-BY-NC-SA) or commercial (CC-BY-SA) use. The UK Data Service, funded by the Economic and Social Research Council (ESRC), provides public access to a curated collection of economic, social, and population data. This includes surveys, census data, administrative data, qualitative research, and international datasets.

For the sake of this guide, we will stick to Natural Earth and Open Geography Portal resources for geospatial data, and ReShare database for compatible, plottable datasets.

4 GeoPandas, a go-to Python Library for Map Creation

GeoPandas, often abbreviated as *gpd* or *gp*, is an open-source Python library that builds on *Pandas* data structures, functions and methods by integrating geospatial capabilities from *Shapely*. It simplifies importing, analyzing, and visualizing geospatial data, reducing the need for more complex tools like PostGIS or managing multiple separate Python libraries.

Perhaps one of the most important features of *GeoPandas* is its ability to handle a wide range of geospatial file formats. It supports formats like CSV files with coordinate columns, Shapefiles, GeoJSON, KML, GeoPackage, PostGIS, Spatialite, and DXF, able to import data from those files and encode it into a *GeoDataFrame*, a data object format similar to *Pandas DataFrame* but tailored to Geospatial data. Moreover, *GeoDataFrame* can be later exported in multiple different data formats, including geospatial ones. This makes it highly versatile, allowing users to easily work with data from different sources without the need for additional tools.

Preparing for the installation

GeoPandas requires Python 3.6 or higher. You can check your version in the Command Prompt or Terminal by typing:

```
python --version
```

...or within the environment you work with, by using *sys* module to run *.version* :

```
import sys  
print(sys.version)
```

If you need to update Python to the newest version, you can simply download and install the latest **Python release**. It is the simplest way for environments using the system-wide installation, just remember to check *Add Python to PATH* during installation.

IDLE, the official Python development environment, is installed alongside Python as default. For the rest of environments, once new Python version is installed: - In the *Video Studio Code* navigate to *View -> Command Palette* (Ctrl+Shift+P on Windows or Cmd+Shift+P on macOS). In the popup type *Python: Select Interpreter* and then choose the updated Python version to run as default. - In *PyCharm*, first set up the project you will be working on, then navigate to *File -> Settings -> Project: NAME > Python Interpreter*. There you have an option to add a new interpreter, installed from your PC.

Jupyter Notebook also requires installing new kernel to work with current Python version. In your command prompt window or terminal, type:

```
python -m pip install --upgrade ipykernel
```

It is also possible to install a new kernel, to use it for a specific Python version:

```
python -m ipykernel install --user --name pythonX --display-name "Python X"
```

...where X is Python version.

If you use software providing merged ecosystem for multiple environments, such as e.g. *Anaconda*, it is recommended to handle updates within its terminal, called *Anaconda Prompt*. Once open, type:

```
conda update python
```

You can verify installation by typing

```
python --version
```

GeoPandas Installation

The simplest way to install GeoPandas is to open your Command Prompt (Windows) or Terminal (Mac/Linux) and type:

```
pip install geopandas
```

If you use Anaconda, in Anaconda Prompt you can write and execute this code:

```
conda install -c conda-forge geopandas
```

However, there are also built-in terminals or internal tools that can help you with that: - In *PyCharm*, you can write the same commands within its own terminal, accessible through an icon in the bottom left icon row. Alternatively, you can go to *File > Settings > Project > Python Interpreter*, click '+' and type *geopandas*. Once you see it as an option, click on it, and then click *Install Packages*. - *VS Code* also has own terminal, accessible through *View > Terminal* and then pressing *Ctrl + ~*. - For *Jupyter Notebook* (and *Lab*), it is possible to install packages by typing *!pip* and *!conda* commands in a code cell and then executing its contents.

Verifying GeoPandas Installation

To verify whether installation succeeded, open a new file, import geopandas and print its current version:

```
import geopandas as gpd
print(gpd.__version__)
```

If you want to update geopandas, you can use a modified pip or conda command:

```
# for cmd or Mac terminal
pip install --upgrade geopandas
# for Anaconda Prompt
conda update -c conda-forge geopandas
```

Other Key Libraries

We also require a few other libraries for a regular workflow with *GeoPandas*: - *matplotlib*, specifically *pyplot*, *colors* and *patches*, to create good visualisations. - *pandas*, to be sure we can properly format data loaded to our environment. - *adjustText*, useful when plotting labels directly on a map. - *textwrap*, especially useful if we have very long labels and need to wrap them.

Additionally, *GeoPandas* works well with other libraries designed for mapmaking, especially *Folium*: - *Folium*, for interactive maps in html format. - *branca*, to manually set colour steps and scales for our *folium*-based maps.

Matplotlib is always installed along Python installation. Regarding the rest, you can install those the same way you installed *GeoPandas*.

5 Setup, Importing Data, Basic Plotting and Exporting Results

At the start of every project we have to import appropriate libraries. As you likely know by now, we can do it by using *import*, with additional *as* to create a usable alias. At minimum, to make a map and be able to export it as a file we will require GeoPandas and pyplot:

```
import geopandas as gpd
import matplotlib.pyplot as plt
```

Once we have both libraries imported in our script, we will have to import a file containing geospatial data into Python environment in order to proceed. For that end we will use GeoPandas own function, called *.read_file()*. At minimum, it needs a path to the file to work, either the system path or a working url. *.read_file()* returns a *GeoDataFrame* object, containing all the data obtained from the file.

```
# A simple, low-res map of the world can be downloaded from Natural Earth url=(
    "https://naturalearth.s3.amazonaws.com/110m_cultural/"
    "ne_110m_admin_0_countries.zip"
)
# I use () to break the url down to fit the guide boundaries ;)

# We put url into .read_file() and save as world_simple world_simple =
gpd.read_file(url)
```

Now *world_simple* is a *GeoDataFrame* object, readable to many python fuctions and methods. We can explore its contents e.g. by using *.head()* or *.columns*:

```
print(world_simple.head(n=5))
print(world_simple.columns)
```

	featurecla	scalerank	LABELRANK	SOVEREIGNT	SOV_A3	\
0	Admin-0 country	1	6	Fiji	FJI	
1	Admin-0 country	1	3	United Republic of Tanzania	TZA	
2	Admin-0 country	1		Western Sahara	SAH	

	ADM0_DIF	LEVEL	TYPE	TLC	ADMIN	...	\
0	0	2	Sovereign country	1	Fiji	...	
1	0	2	Sovereign country	1	United Republic of Tanzania	...	
2	0	2	Indeterminate	1	Western Sahara	...	

	FCLASS_TR	FCLASS_ID	FCLASS_PL	FCLASS_GR	FCLASS_IT \
0	None	None	None	None	None
1	None	None	None	None	None
2	Unrecognized	Unrecognized	Unrecognized	None	None

	FCLASS_NL	FCLASS_SE	FCLASS_BD	FCLASS_UA \
0	None	None	None	None
1	None	None	None	None
2	Unrecognized	None	None	None

	geometry
0	MULTIPOLYGON(((180 -16.06713, 180 -16.55522, ...
1	POLYGON((33.90371 -0.95, 34.07262 -1.05982, 3...
2	POLYGON((- 8.66559 27.65643, -8.66512 27.58948...

[3 rows x 169 columns]

```
Index(['featurecla', 'scalerank', 'LABELRANK', 'SOVEREIGNT', 'SOV_A3',
      'ADM0_DIF', 'LEVEL', 'TYPE', 'TLC', 'ADMIN',
      ...,
      'FCLASS_TR', 'FCLASS_ID', 'FCLASS_PL', 'FCLASS_GR', 'FCLASS_IT', 'FCLASS_NL',
      'FCLASS_SE', 'FCLASS_BD', 'FCLASS_UA', 'geometry'],
      dtype='object', length=169)
```

As we can see above, the object contains both proper geometry (packed within the *geometry* column) and related data. In short, we can plot it. To do basic plotting, we will use *.plot()*. By default *.plot()* does not need additional arguments to be defined, but it is always useful to tinker with the figure size and fill/line colours to enhance readability.

```
world_simple.plot(
    figsize=(10, 6),          # Dimensions (width, height); default is 6.4 to 4.8 # In-map
    edgecolor='black',        # line colours; default is 'none'
    linewidth=0.6,           # Width of the lines drawn; by default it is 0.5 # Fill
    color='lightblue')       # colour; default is 'steelblue'
```

While not necessary, we can also include a caption in our figures, if we deem it necessary. I will do it later to number figures generated. For that end, we can use *.figtext()*:

```
# We can also add a caption
plt.figtext(
    0.5,      # x range from 0(left) to 1 (right), 0.5 means Located centrally
    0.03,     # y range from 0 (bottom) to 1 (top), here near the bottom
    "Figure 1: A simple map plotting example using GeoPandas.", # Text to plot
    ha="center", # Horizontal alignment to x stated before
    fontsize=12, # Font size to use
    style="normal") # Style to use e.g. "italic" or "oblique"
```

In some environments you will need to write `plt.show()` to see the results of map generation.

```
plt.show()
```

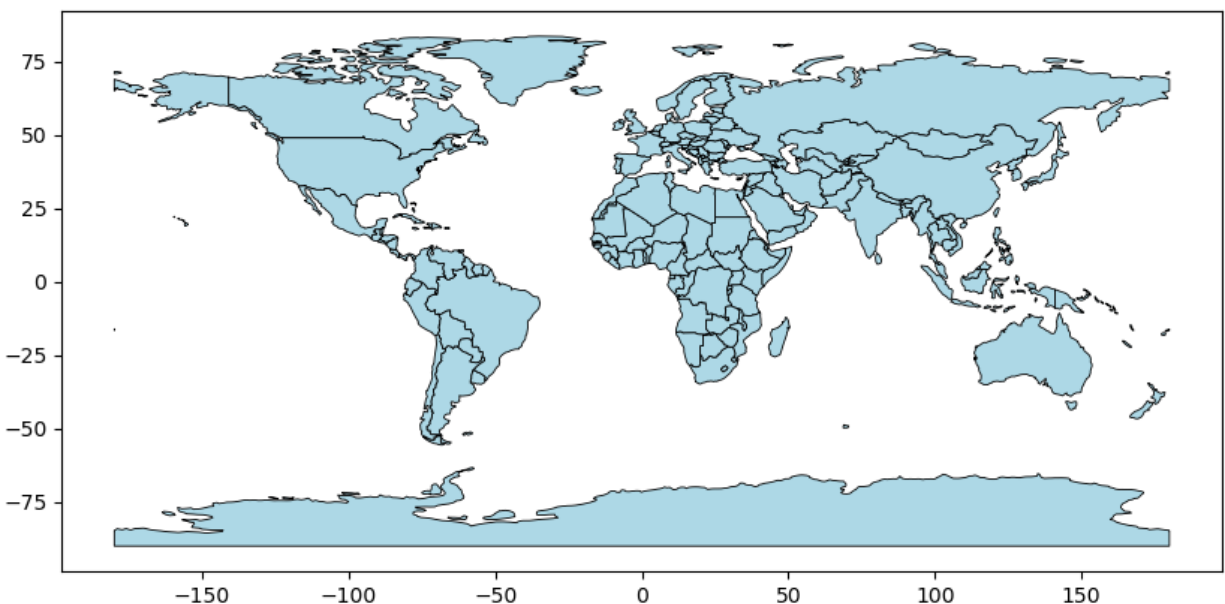


Figure 1: A simple map plotting example using GeoPandas.

Figure 1 looks good, but it would look even better with labeled axes and perhaps a title at the top. Also, if we want to highlight a point on the map; for example, the approximate location of Edinburgh, where SSI, UoE, is located, we can do this by layering elements according to their sequence of generation. The first generated element will be the lowest layer, while the most recent will be on top. To do this, we first need to establish *ax*, which represents the axes (a *canvas* of sorts) where these layers will be drawn. After that, we can plot the main map using `.plot()`, and on top of it, we can add a `.scatter()` function to plot the point, specifying the coordinates.

Both layers will be assigned to *ax*, but we will show two way of doing so, 1) including in function arguments, and 2), attaching function to *ax* via:

```

# Firstly, we define axes area
fig, ax = plt.subplots(figsize=(10, 6))

# We plot the first layer
world_simple.plot(
    ax=ax,                #We define world_simple as a layer within ax
    edgecolor='black',
    linewidth=0.6,
    color='lightblue')

#   Edinburgh   coordinates
edinburgh_lon   =   -3.1883
edinburgh_lat = 55.9533

# Add a dot for Edinburgh
ax.scatter(
    edinburgh_lon, # We state longitude here
    edinburgh_lat, # We state latitude here
    color='red',   # Marker colour
    marker='o',    # Marker shape, with 'o' referring to circles #
    s=50,          # Marker/point size, default being 20
    label="Edinburgh" )

# Add labels for x and y axes
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")

# Add a title (optional)
ax.set_title("Map with Edinburgh Marked")

# Show the plot
plt.show()

```

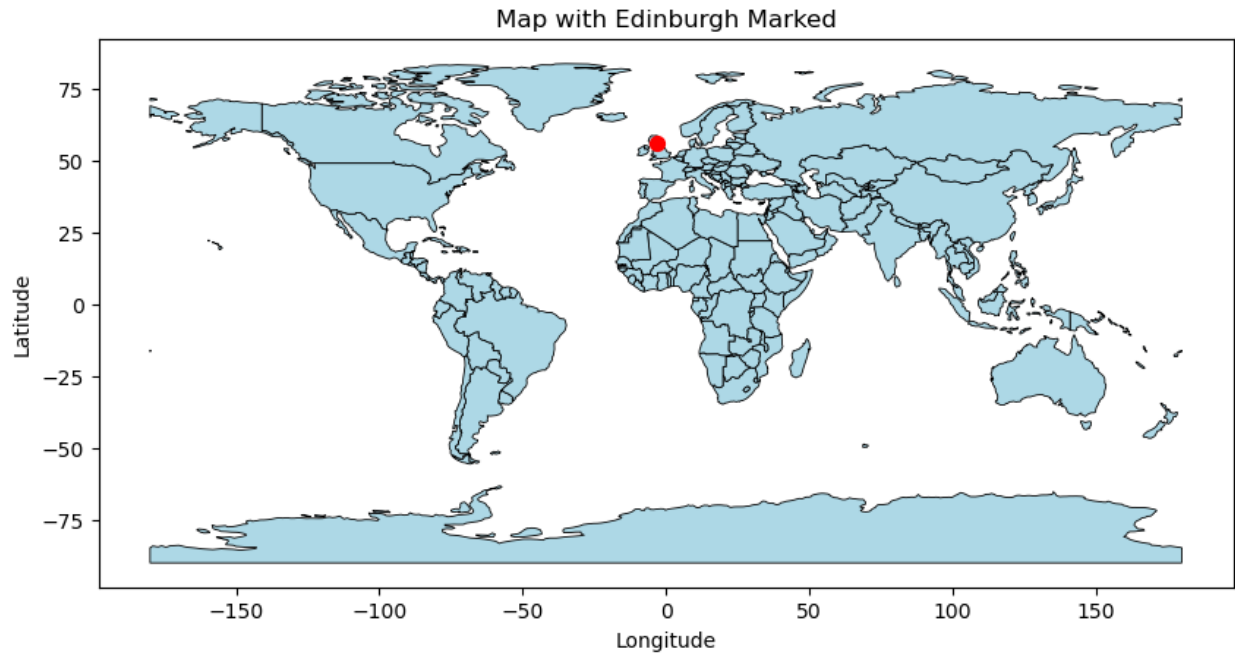


Figure 2: A simple two-layered map.

To save the map, we can use the `pyplot.savefig()` function. This function requires at least a name for the generated figure, including the desired format. The default resolution is `dpi = 100`, but we can define a higher resolution for our work. The current standard for publications is a dpi of at least 300:

```
plt.savefig(
    "world_simple.png",    # Name and end format
    dpi=300)               # resolution we want
```

It is also possible to export the `GeoDataFrame` object itself into multiple different formats. We can do this by using the `.to_file()` function. The easiest formats to handle are newer ones:

```
world_simple.to_file("world_simple.geojson", driver="GeoJSON") # GeoJSON
gdf.to_file("world_simple.kml", driver='KML')                  # KML
world_simple.to_file("world_simple.gpkg", driver="GPKG")       # GPKG
```

We can also export data as *shapefiles*. However, it is an older format with many restrictions, and we may be forced to reformat the data before proceeding. Additionally, since the *shapefile* format is actually based on three different file types, it will generate three files that must be stored together, ideally in a *zip* file.

```
# Reformatted for larger integers (int64), to avoid error
world_simple['POP_EST'] = world_simple['POP_EST'].astype('int64')

# Should save without problems
world_simple.to_file("world_simple.shp")
```

6 Creating Multi-Layered Maps

Let's assume we need a map of Europe, with country borders and all the capitals displayed, plus a division between EU and non-EU countries. How should we approach this task?

The first step, before we plot anything, is to find the data we need. Specifically, we need two geospatial datasets: one containing the geometry defining country borders and one containing the coordinates of major cities.

For **country boundaries** we may be inclined to look at what is available on [Natural Earth](#) website. There, we can find so-called "cultural" maps with "Small scale data" (1:110m), suitable for maps containing multiple countries up to a global scale. In particular, the [Admin 0 – Countries](#) collection seems most useful for our purposes. However, as *Natural Earth* follows own disputed borders policy, favoring the current physical state rather than one based on international law, their data may not be the best solution for our work, particularly when it comes to Europe (especially the Balkan region or the ongoing conflict in Ukraine).

In a result, a better solution may be *Eurostat's* own country geographic data, **available under [this link](#)**. The data reflects *de jure* conditions, making our work much simpler, with standard coding for internationally recognized countries (see [here](#)). From the site, we will download the data for the year 2024 in *SHP* format, and at the biggest scale (1:60 million). For our purposes, the most useful coordinate system is EPSG 4326, also known as the World Geodetic System 1984 (WGS 84). In simple terms, it is a spatial reference system (SRS) adopted in 1984 as a standard for GPS tracking and mapping applications (for more, see [here](#)). It is an ideal standard for regular map plotting, saving us time and hurdles with recalculating coordinates between the files.

Please download the zipped data and, without unpacking it, place it in your working directory. If you have no idea what the working directory is, it is essentially a location on your computer where the environment you work in starts looking for files if not instructed to look under a specific path. You can check your working directory from within the environment by typing this:

```
import os
print(os.getcwd())
```

We also need city data, but here we can make use of the *Natural Earth* collection at the 1:100m scale, as capitals in Europe are not currently under any major disputes. The so-called **"Populated Places"** dataset seems perfect for what we want to plot. Since *Natural Earth* uses the same projection standard (*EPSG:4326*), the coordinate system does not need recalculating. Additionally, it can be imported into our environment directly from the link, as shown in the overview chapter before.

Our setup should look something like this:


```

# All necessary libraries
import geopandas as gpd
import matplotlib.pyplot as plt
# Function adjust_text, useful for working with labels
from adjustText import adjust_text
# Lines and patches libraries, ideal to plot lines, circles and squares # for the map
legend
from matplotlib.lines import Line2D
from matplotlib.patches import Patch

# Path to the zipped file (assuming it is in your working directory) world_loc =
"CNTR_RG_60M_2024_4326.shp.zip"
# url to the data at Natural Earth
cities_url = (
    "https://naturalearth.s3.amazonaws.com/110m_cultural/"
    "ne_110m_populated_places.zip"
)

# Importing datasets
world = gpd.read_file(world_loc) cities =
gpd.read_file(cities_url)

```

What if we plot both data frames right away? We can stack two maps on top of each other and narrow the drawing area to focus roughly on Europe. To do this, we will use the `.xlim()` and `.ylim()` functions to restrict the area shown. For the first map, we will use the already shown `.plot()` function in conjunction with the pre-established axes area:

```

# Establishing the axes area
fig, ax = plt.subplots(figsize=(10, 6))

# Plotting the first layer (countries)
world.plot(ax=ax, edgecolor='black', linewidth=0.6, color='yellow')

# Plotting the second layer (cities)
cities.plot(ax=ax, color='blue', markersize=10)

# Narrowing the plot area to Europe
plt.xlim(-25, 50.5)
plt.ylim(34.5, 71.5)

```

```
# Including labels
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")

# Map title
ax.set_title("Map of Europe")

# Show the output
plt.show()
```

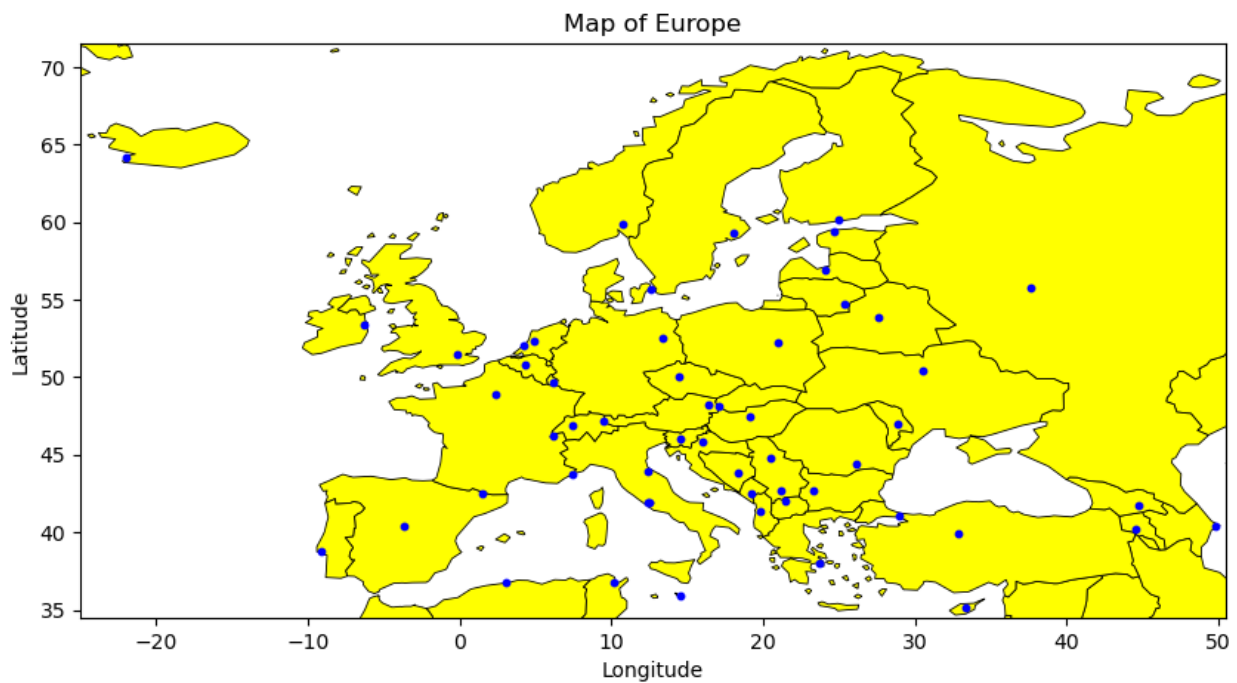


Figure 3: A two-layered map (bottom - world, top - cities), narrowed to Europe.

Figure 3 does look good, but when we look at some countries, we may notice more than one city shown, such as for the Netherlands or Turkey. To make this map more useful, we would need to remove non-capital cities. It may also be best to only plot some countries - such as those that have part of their territory in Europe or are technically considered European due to cultural reasons. This can be further narrowed to EU countries. Finally, we should label the capitals for general clarity.

We can filter both data frames to get only the rows we are interested in, but to do so we have to check what they consist of, and what columns to use to discern between data we want and one to be discarded:

```
# world data
print(world.head(n=4))
# it seems there is a unique country code in the data frame
print(world['ISO3_CODE'].tolist())

# city data
print(cities.head(n=4))
# each row has associated city name, including two versions
print(cities['NAMEA'].tolist())
print(cities['NAMEASCII'].tolist())
```

For the world data, we have an *ISO3_CODE* column, which contains internationally recognized standards for providing 3-letter codes for recognized countries. These codes usually resemble their abbreviations, and it's not too hard to find a list of them online, including on sites like [here](#) and [here](#) for more details.

For the cities, there are two columns: one providing the accurate name and the other a transliteration into a standard set of characters compatible with *ASCII* encoding. As there are no other means, this may be the best solution for filtering. Given that some ASCII names may be quite far from what is usually seen on maps, we will use the *NAMES* column. However, it also means dealing with unique signs specific to various European languages, and we should take this into account if issues arise.

Our code can look like the one below. First, we create lists of what we seek, which we will later apply to each data frame using the *.isin()* function. Thanks to this, we will get only the rows in the filtered column that contain at least one element from the specified list (i.e., return 'TRUE'). We will save the results as three new data frames: border data for Europe overall, EU-specific border data, and European capitals.

```
# Filter for Europe
#( including Azerbaijan, Georgia, Turkey and Kazakhstan,
# as technically partially in Europe,
# and Armenia/Cyprus due to cultural connections)
europe_list = ['ALB', 'AND', 'ARM', 'AZE', 'AUT', 'BEL', 'BIH', 'BLR', 'BGR',
               'CHE', 'CYP', 'CZE', 'DEU', 'DNK', 'ESP', 'EST', 'FIN', 'FRA', 'GEO', 'GBR', 'GRC',
               'HRV', 'HUN', 'ISL', 'IRL', 'ITA', 'KAZ', 'KSV', 'LIE', 'LTU', 'LTU', 'LUX', 'LVA',
               'MKD', 'MLT', 'MDA', 'MNE', 'NLD', 'NOR', 'POL', 'PRT', 'ROU', 'RUS', 'SRB',
               'SVK', 'SVN', 'SWE', 'TUR', 'UKR', 'VAT']

# Filter specifically for the European Union
```

```

eu_list = ['AUT', 'BEL', 'BGR', 'CYP', 'CZE', 'DEU', 'DNK', 'ESP', 'EST', 'FIN',
           'FRA', 'GRC', 'HRV', 'HUN', 'IRL', 'ITA', 'LVA', 'LTU', 'LUX', 'MLT', 'NLD', 'POL', 'PRT',
           'ROU', 'SVK', 'SVN', 'SWE']

# ... and or capitals (without nur-sultan, as it lies outside of xlim set) capitals_list =
['Ankara','Tirana','Andorra','Yerevan','Baku','Vienna','Brussels',
  'Sarajevo','Minsk','Sofia','Bern','Nicosia','Prague','Berlin','Copenhagen', 'Madrid',
  'Tallinn','Helsinki','Paris','Tbilisi',
  'London','Athens','Zagreb','Budapest','Reykjavík','Dublin',
  'Rome','Pristina','Vaduz','Vilnius','Luxembourg City','Riga',
  'Skopje','Valletta','Chişinău','Podgorica','Amsterdam','Oslo',
  'Warsaw','Lisbon','Bucharest','Moscow','Belgrade','Bratislava',
  'Ljubljana','Stockholm','Kyiv','Vatican City']

# Application of filter to appropriate data frames europe =
world[world['ISO3_CODE'].isin(europe_list)] eu =
world[world['ISO3_CODE'].isin(eu_list)] capitals =
cities[cities['NAME'].isin(capitals_list)]

```

Once done, we can start plotting the map. This map will have three layers, starting with European countries (Europe), with EU countries shown in a different color (EU), and capitals shown as large, blue dots. Like the previous one, this map will also be narrowed to European coordinates, but we will also ensure it stays within the appropriate aspect ratio between latitude and longitude by using the `set_aspect()` function and setting it to equal.

```

# Preparing the axes
fig, ax = plt.subplots(figsize=(20, 12))

# First, we plot the countries layer
europe.plot(
    ax=ax,                # figure we are building #
    edgecolor="brown",    # In-map line colours # Fill
    color="beige"         # colour
)

# Then, we plot EU countries on top
eu.plot(
    ax=ax,                # figure we are building #
    edgecolor="brown",    # In-map line colours # Fill
    color="wheat"         # colour
)

```

```
)

# Thirdly, we plot individual capitals
capitals.plot(ax=ax, color='blue', markersize=10)

# Let's narrow coordinates to Europe
plt.xlim(-25, 50.5)
plt.ylim(34.5, 71.5)

# We should also maintain an equal aspect ratio for the map
ax.set_aspect('equal')

# Once done, we can plot the map
plt.show()
```

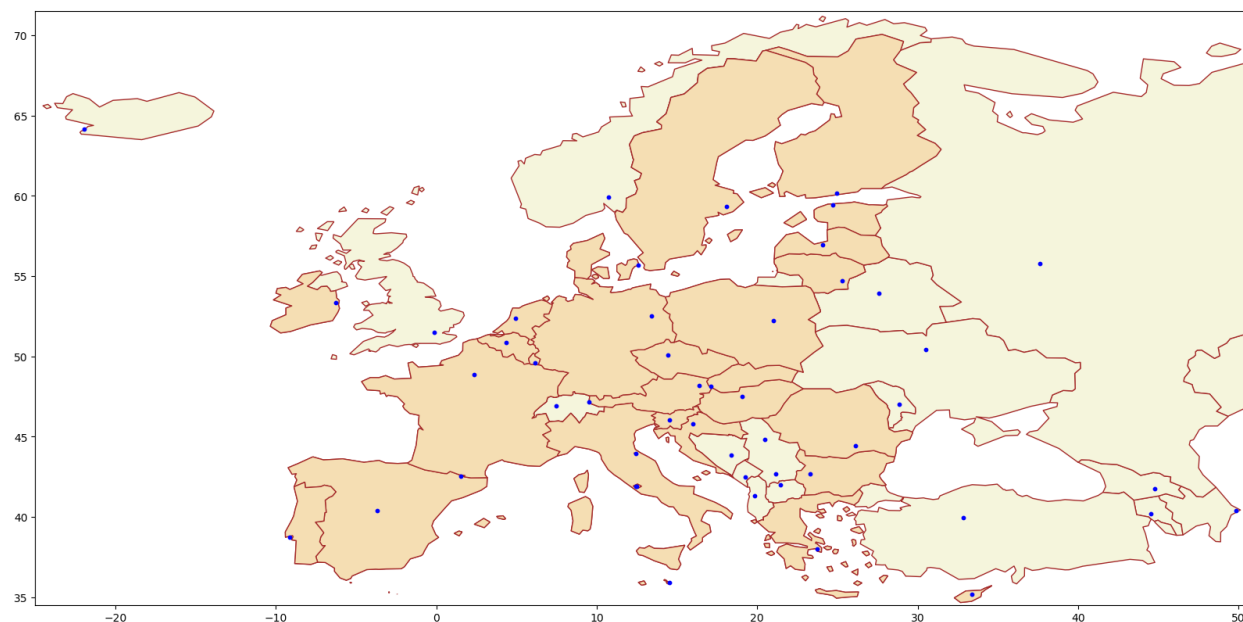


Figure 4: A three-layered map from filtered data frames.

Figure 4 looks promising, but we still need to plot the capital names. We have three pieces of information to do that: the capital names in the *Name* column, and the *x* and *y* coordinates in the *geometry* column, specifically in a sub-element called *centroid* (i.e., the geometric centre of a figure). We can use *.annotate()* from *pyplot* and a for loop to quickly draw all the labels we need. However, we will need a method to effectively work with data within multiple data cells, including nested values within the *geometry* column. We can do this in several ways, for example, by using the built-in *zip()* function. However, *.itertuples()* or *.iterrows()* from *GeoPandas* may be a better choice. Especially, *.iterrows()* lets you iterate over rows in a data frame as pairs of index and

rows.

```
# Approach 1
for name, x, y in zip(
    capitals["NAME"],
    capitals.geometry.centroid.x,
    capitals.geometry.centroid.y):
    ax.annotate(
        text=name,
        xy=(x, y),
        horizontalalignment='left',
        color='blue'
    )
    # We take 3 values from columns in zip() #
    # Column NAME stated
    # centroid x val, within geometry
    # centroid y val, within geometry
    # We create an annotation on axes
    # Text to include
    # Coordinates of the text
    # Alignment of the text (to xy point) # Colour
    # of the text

# Approach 2
for row in capitals.iteruples(index=False):# For each row in the data frame:
    ax.annotate(
        text=row.NAME,
        xy=(row.geometry.centroid.x,
            row.geometry.centroid.y),
        horizontalalignment='left',
        color='blue'
    )
    # We create an annotation on axes
    # Text to include
    # centroid x val, within geometry
    # centroid y val, within geometry
    # Alignment of the text (to xy point) #
    # Colour of the text

# Approach 3
for idx, row in capitals.iterrows():
    ax.annotate(
        text=row['NAME'],
        xy=(
            row['geometry'].centroid.x, # centroid x val, within geometry
            row['geometry'].centroid.y),# centroid y val, within geometry
        horizontalalignment='left',
        color='blue'
    )
    # For each index/row in the data frame: #
    # We create an annotation on axes
    # Text to include
    # Alignment of the text (to xy point) #
    # Colour of the text
```

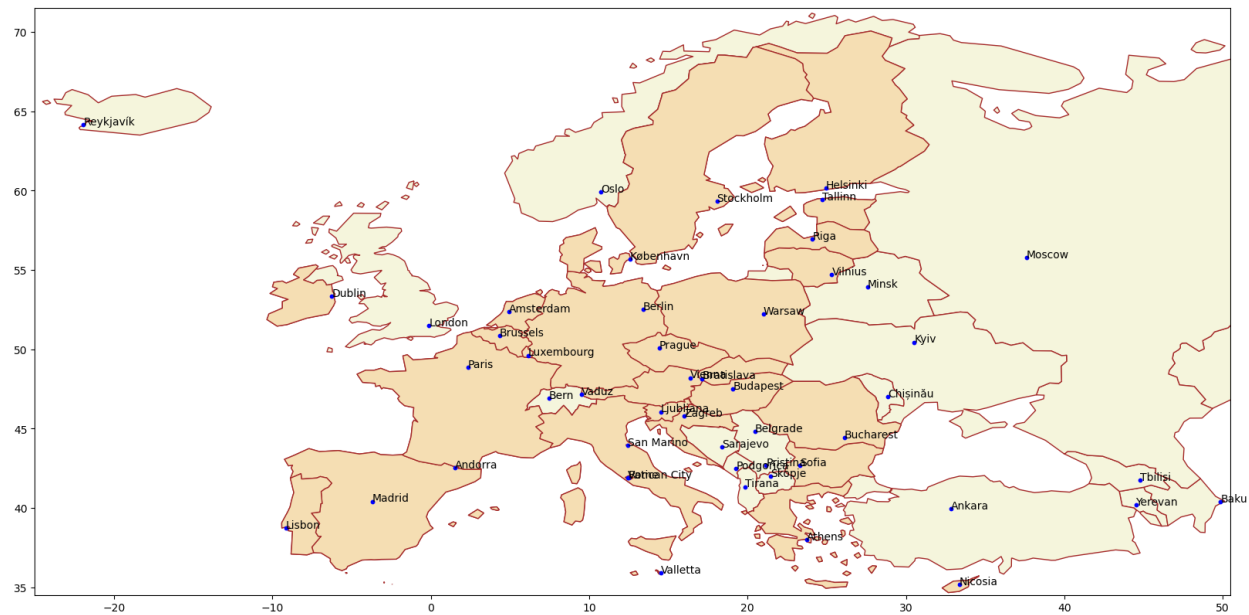


Figure 5: A three-layered map with annotations.

As we can see by looking at **Figure 5**, labeling features on maps can be quite tricky. Names can overlap each other and other features, making the labels and the map unreadable — for example, Rome and Vatican City above. We will approach this by plotting labels using the `.text()` function while recording each label created in the `texts` list. Then, we will use the `.adjust_text()` function to ask Python to enable dodging and arrows when necessary. It may seem a bit complex, but do not worry, it is actually pretty easy. There are just a lot of optional arguments available to make the labels look better, and adjust the strength and scope of possible dodging.

```
# We can plot all the capitals labels this way:
texts = [] # an empty list we will use to store created labels for idx,
row in capitals.iterrows(): # for each index/row in capitals data frame
    text = ax.text( # each label is added to axes
        row.geometry.x, # X-coordinate (longitude)
        row.geometry.y, # Y-coordinate (latitude)
        row['NAME'], # Capital name from the 'NAME' column
        fontsize=10, # Font size for the labels
        ha='center', # Horizontal alignment (centered)
        va='center', # Vertical alignment (centered)
        color='black', # Color of the label text
        fontweight='bold' # Bold font for better visibility
    )
    texts.append(text) # once text created, added to the texts list

# Now we can adjust labels to avoid overlap
```

```

adjust_text(
    texts,                                # the list we populated previously #
    ax=ax,                                # what we plot on
    expand_points=(1.5, 1.5),              # Expand area where text starts
    expand_text=(1.5, 1.5),                # Expand area text may cover
    arrowprops=dict(                       # Dictionary of arrow properties
        arrowstyle="-",                   # Arrow style, e.g., "->", "-|>", etc. # Arrow
        color='black',                    # color
        lw=0.5)                           # Line width for the arrow
)

# Now, let's name x and y labels and the map itself
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
ax.set_title("The Map of Europe, with Capitals Shown")

```

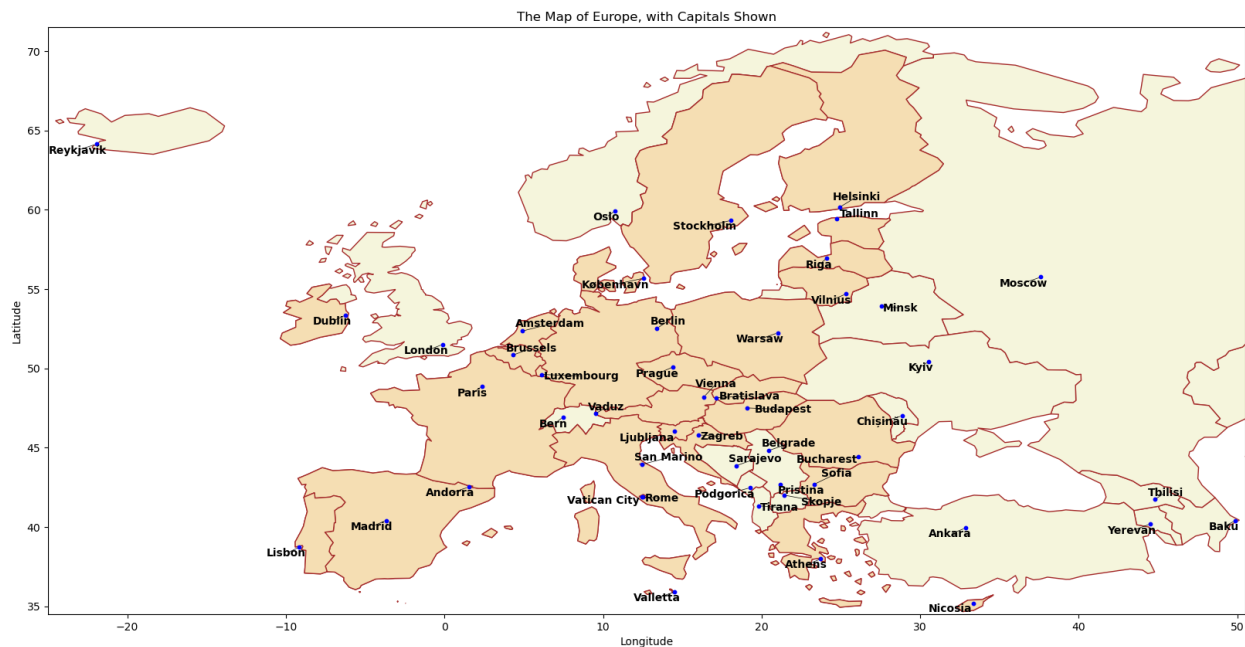


Figure 6: A map with labels autoadjusted to dodge each other and the point they refer to.

Given that **Figure 6** is essentially what we hoped to achieve with the initial idea, the last thing we have to do is explain what is actually seen on the map. For that, we can plot a legend, first by establishing its properties in a custom list of objects, containing all the lines, circles, and squares, as well as the associated text. Then we can use `.legend()` to plot the list contents on the top, final layer:


```

# Creating and custom legend
legend_elements = [
    Patch(color='beige',
          label='European Countries'),
    Patch(color='wheat',
          label='EU Countries'
          ),
    Line2D([0], [0],
           color='blue',
           marker='o',
           markersize=10,
           linestyle=' ',
           label='Capitals'
           )
]

# Object that will store all legend elements # First
# element, as a rectangle

# Second element, also a rectangle

# Third element, a circle

# Plotting the legend on axes
ax.legend(
    handles=legend_elements, # Label elements to be displayed
    loc='upper left',        # Location of the label
    fontsize=12)             # Font size for label text

# And, finally... plot!
plt.show()

```

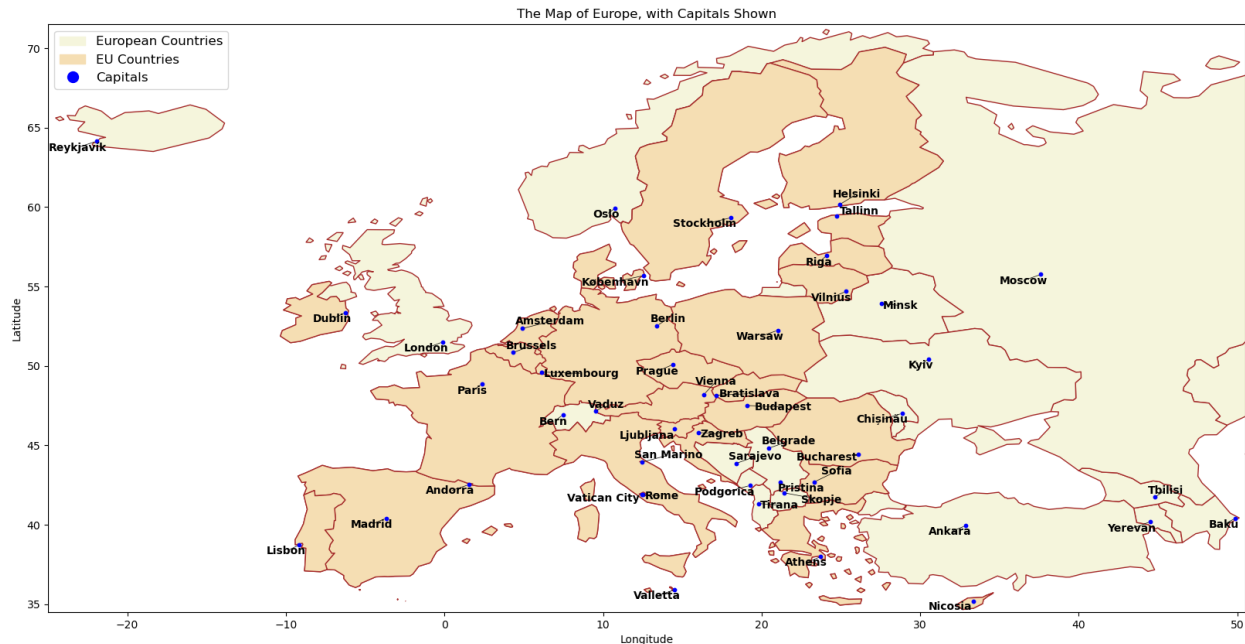


Figure 7: A complete map, with a legend in top-left corner.

Exporting Results

Figure 7 is the end product we wanted to generate. Now, we can save the map through using `.savefig()`, as it was shown a chapter before:

```
plt.savefig("europe_map_with_capitals.png", dpi=300)
```

Given that we have also created three separate datasets by modifying and filtering the original two, we may also be interested in exporting them. We can export them into various file formats, but this time we will use the *GeoJSON* format:

```
world_simple.to_file("europe.geojson", driver="GeoJSON")
world_simple.to_file("eu.geojson", driver="GeoJSON")
world_simple.to_file("capitals.geojson", driver="GeoJSON")
```

7 Creating Choropleth Maps

As we now know how to create multi-layered maps and add various labels to them, it is time to become familiar with one of the most common data visualization methods involving maps: **choropleths**. These are thematic maps designed to visualize specific data in a geographic context through the use of a colour scale or gradient. A good example is population density maps, where more densely populated areas are often shaded in darker tones of a chosen colour, while less densely populated regions are shaded lighter.

To create an example of such a map, we will need two datasets: one for geographic data and one for the data we want to plot as a choropleth. Ideally, both should be compatible to ensure we can join them into a single file before plotting. To this end, we will need datasets that follow the same standard.

As mentioned in one of the early chapters, one of the standards is the **Nomenclature of Units for Territorial Statistics** (NUTS) system. For the sake of this exercise, let's assume we were interested in UK, specifically Scotland. The latest version of the NUTS system for the UK was released in 2018, with the most useful level for regional work being NUTS 3. The Open Geography Portal provides the 2018 data for download [here](#). The data is available in several formats, including *shapefile*, but for this exercise, we will use the *GeoJSON* version. If you want to follow what I do, please download the file and place it in your working directory.

A good example of a recent dataset compatible with *NUTS level 3* system is the [Economic Inactivity, Health Conditions, Work-Limiting Health Conditions and EU and Non-EU Populations in NUTS3 Regions of the UK, 2018-2021](#). Published in 2024 on the previously mentioned *ReShare* platform, it provides a wealth of data that can be plotted directly with NUTS 3-level spatial data, thanks to its adherence to the same regional structures. You can download the *xlsx* file, zipped, from [here](#). Please extract the *xlsx* file and place it in your working directory.

Your initial setup for this map should look something like that:

```
# Importing required libraries import
geopandas as gpd
import matplotlib.pyplot as plt import
pandas as pd
import matplotlib.colors as mcolors #
adjust_text for working with labels from
adjustText import adjust_text

# Path to a GeoJSON file
geojson_path = (
    "NUTS_Level_3_January_2018_FCB_in_the_United_Kingdom_2022"
```

```

"-8077497628547136415.geojson"
)
# Path to the dataset
health_data_path = '857426_Data_Documentation.xlsx'
# Importing the dataset - saved in the working directory
nuts3_uk = gpd.read_file(geojson_path)
health_data = pd.read_excel(health_data_path, sheet_name=2, engine='openpyxl')

# We can plot the geodata to check how NUTS map of UK looks like
nuts3_uk.plot()
plt.show()

```

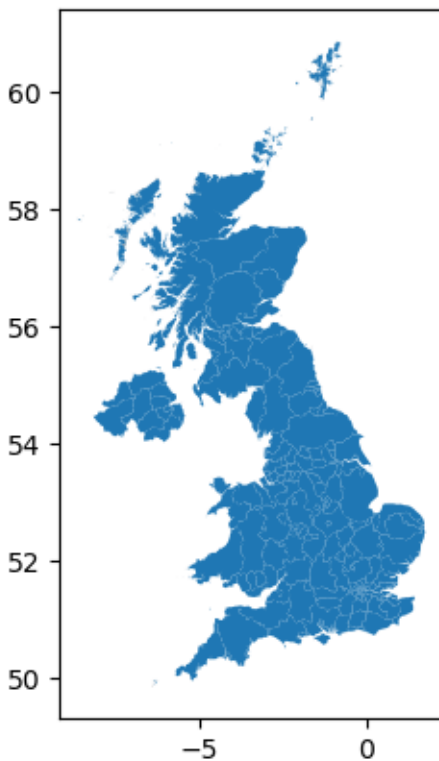


Figure 8: NUTS 3 map of UK.

Figure 8 shows that the data can be plotted without much issue, covering most of the UK except for some autonomous islands. Most importantly, all of Scotland is present. However, we need to filter the data in a similar way to how we did with countries in the previous chapter, leaving only the NUTS regions that represent Scotland. To do so, we first need to check both dataframes for a common column.

Let's first see geospatial data:

```
# See what is inside the NUTS 3 geospatial data for UK
print(nuts3_uk.head(n=4))

# We can see that both regions names and NUTS level 3 codes are available
print(nuts3_uk['nuts318nm'].tolist()) # Names
print(nuts3_uk['nuts318cd'].tolist()) # Codes
```

...and now the dataset:

```
# See what is inside the dataset we want to plot
print(health_data.head(n=4))

# NUTS coding is included
print(nuts3_uk['NUTS163_code'].tolist())
```

Both dataframes include the NUTS 3 coding system, and we will use it to filter only the data relevant to us. Since it is the same system, even though from different years, we can use a single list to filter each dataframe and save the results as separate objects:

```
# We prepare to filter data in both data frames
scotland_codes = ['UKM50','UKM61','UKM62','UKM63','UKM64','UKM65','UKM66',
                  'UKM71','UKM72','UKM73','UKM75','UKM76','UKM77','UKM78',
                  'UKM81','UKM82','UKM83','UKM84','UKM91','UKM92','UKM93',
                  'UKM94','UKM95']

# And now we can apply it to both
nuts3_Scotland = nuts3_uk[nuts3_uk['nuts318cd'].isin(scotland_codes)]
health_Scotland = health_data[health_data['NUTS163_cd'].isin(scotland_codes)]
```

Now we have two dataframes, but why not combine them into a single one? Given GeoDataFrame can contain both geometry and associated data, it is possible to merge these objects using `.merge()`, a function used in both the *Pandas* and *GeoPandas* libraries to join datasets. We perform a so-called left join, meaning we add data to `nuts3_Scotland` (left) from `health_Scotland` (right) in a way that the left side is the primary dataset, and any non-matching data from the right side is discarded.

```
merged_Scotland = nuts3_Scotland.merge( # nuts3 dataset as main (left) one
    health_Scotland,                    # Health dataset as merged (right) one #
    left_on='nuts318cd',                # we state the columns to use as id's
```

```

    right_on='NUTS163_code',          # along which the merge will happen
    how='left'                        # what does not fit from health dataset # is
)                                    discarded

# Just to be sure the outcome is a GeoDataFrame we save it as one
merged_Scotland = gpd.GeoDataFrame(merged_Scotland)

```

How does the new dataframe look? You can always take a peek:

```
print(merged_Scotland.columns)
```

We now have a lot of plottable data aligned with the appropriate regional coordinates. One of the first columns shows the regional population in 2018, before COVID and early into the disruptions brought by Brexit, and in 2021, a year into the COVID pandemic and after the UK officially left the EU. Given this data, we might want to plot the population change per region. We can do this by first creating a new column called 'Pop_change', which results from subtracting the 2018 population data from the 2021 data. We can simply type: `merged_Scotland['Pop_change'] = merged_Scotland['Pop1664_2021'] - merged_Scotland['Pop1664_2018']` and it should work. However, it is preferable to use `.assign()` from the *Pandas* library to avoid potential errors if the numerical formatting differs from what is expected.

```

merged_Scotland = merged_Scotland.assign(
    Pop_change=merged_Scotland['Pop1664_2021'] - merged_Scotland['Pop1664_2018']
)

```

Now we can plot the data. As usual, we will start by establishing the axes and then add a layer to it with the *Pop_change* data plotted in a predefined colour gradient.

```

# First we establish axes
fig, ax = plt.subplots(figsize=(10, 10))

# Now we plot merged_Scotland
merged_Scotland.plot(
    ax=ax,                # We draw on established axes
    column='Pop_change',  # Column we want to plot
    cmap='coolwarm',      # How we map colour, blue-red gradient in this case #
    linewidth=0.5,        # The width of regions borders
    edgecolor='darkgray', # Border lines colour
    legend=True           # Should the legend (colour system) be included?
)

```

```

)

# We add a tittle...
ax.set_title('Scotland - population change 2018-2021', fontsize=12)

# and remove axis:
ax.set_axis_off()

# And, finally, we show the plot:
plt.show()

```

As *Pop_change* has values for all the geometries stated in the file, plotting should complete successfully. But what if we have missing data? The next step would be to plot the missing values (filtered out using the *.isna()* function) as a separate layer, like this:

```

# Overlay missing values in a neutral color
merged_Scotland[merged_Scotland['Pop_change'].isna()].plot(
    ax=ax,                # We draw on established axes
    color="lightgray",    # Colour we use to denote NA's
    edgecolor="darkgreen", # Edge the same as in the original layer
    hatch="//"           # We apply a line pattern in a form of diagonal lines
)

```

Scotland - population change 2018-2021

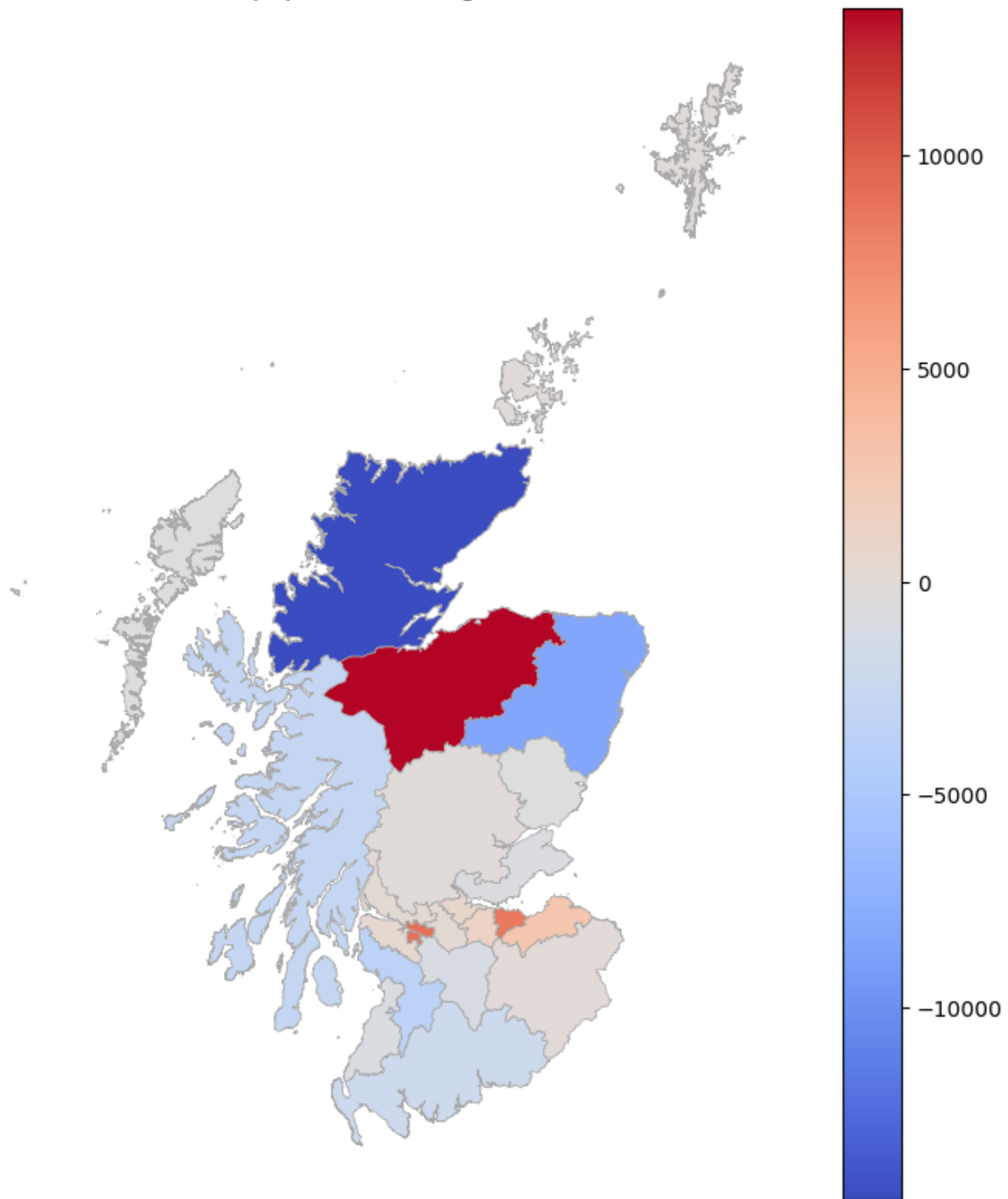


Figure 9: Choropleth map of Scotland NUTS 3 level regions.

Figure 9 looks fine, but it would be easier to read with a better colour scale — ideally, with white denoting no change and green, instead of red, showcasing areas with a population increase. This can be done by manually creating a *cmap* object, which contains the colours we want to

include and the number of steps (shades between colours) to create. We can do this using `.LinearSegmentedColormap.from_list()`. However, if we want to align the colour scale with our data in `Pop_change`, we have to define yet another object, usually called `norm` — for normal distribution. Using `.TwoSlopeNorm()`, we can define the maximum and minimum of the distribution, as well as its central point. Finally, when using the `.plot()` function, we can simply input our results as `cmap=cmap` and `norm=norm`. How it would look like for this example? See the code below:

```
# First, we create a list of colours we want to use colors =
['blue', 'white', 'green']

# Now, we create our own cmap object
cmap = mcolors.LinearSegmentedColormap.from_list(
    "custom_gradient", # It is going to be our own gradient
    colors,             # Here we put the list we made
    N=100)              # Number of individual colours in gradient

# Having colour mapping defined, now we have to arrange it over # a
specific range:
norm = mcolors.TwoSlopeNorm(
    vmin=merged_Scotland['Pop_change'].min(), # minimum as Pop_change minimum
    vcenter=0,                               # centre
    vmax=merged_Scotland['Pop_change'].max()) # maximum as Pop_change maximum

# Then we can start plotting:
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

merged_Scotland.plot(
    column='Pop_change',
    cmap=cmap,           #We put our defined mapping here #...
    norm=norm,           and here our predefined range
    linewidth=0.5,
    edgecolor='darkgray',
    legend=True,
    ax=ax)

# And now we can do the rest:
ax.set_title('Scotland - population change 2018-2021', fontsize=12)
ax.set_axis_off()
plt.show()
```

Scotland - population change 2018-2021

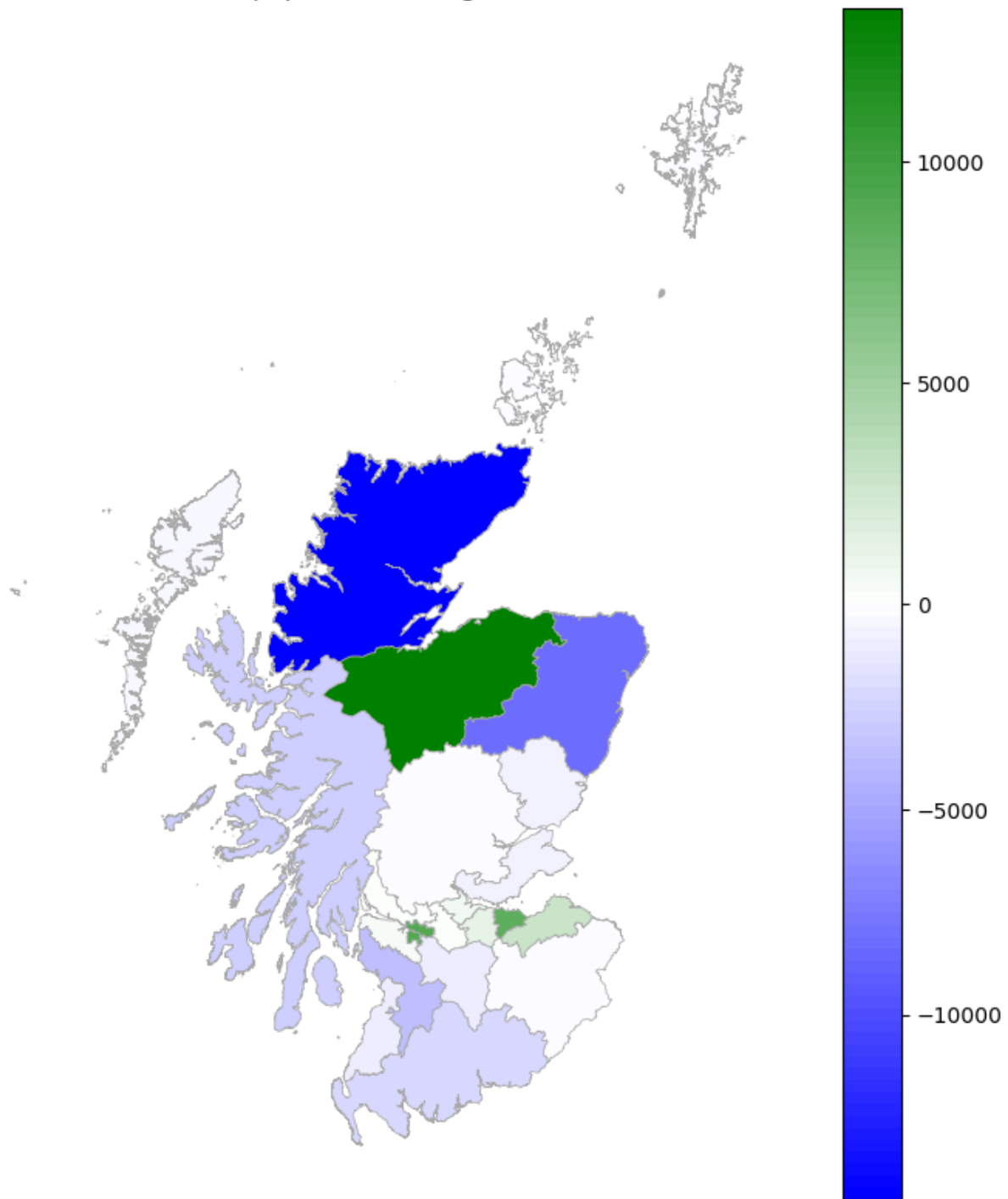


Figure 10: Choropleth map with a custom colour gradient.

Perhaps the last thing we can do to enhance this map is add labels to each region shown. We can reuse a method from the European example, but we need to adjust it a bit to fit our needs.

As you may have discovered on your own, some NUTS 3 region names for Scotland are quite

long, often containing names for several historical and/or administrative regions included as one. However, those are strings, i.e. text data, and as such we can break them and wrap into new lines. We can do so by using `.wrap()` from the `textwrap` library to point towards text to be wrapped and state the maximum width (in signs) after which the text will be broken into a new line. The `wrap()` function would thus return a list of lines. This will be wrapped in `.join()` function, with `\n`, a sign denoting a new line, in front. That way it will be joining each line broken by `.wrap()` to `\n` and thus forcing text to render line by line.

It sounds complicated, but is, in fact, pretty simple. See the code below:

```
# We start the same as in the case of European example texts
= []
for idx, row in merged_Scotland.iterrows():
    wrapped_label = "\n".join(          #we join a new line \n to a text #
        textwrap.wrap(                 and define the text, including:
            row['nuts318nm'],           # the column name from which to take text #
            width=12                    and a specific length after which to wrap
        )
    )
    text = ax.text(
        row.geometry.centroid.x,
        row.geometry.centroid.y,
        wrapped_label, fontsize=6,      # Instead of the row stated, we have wrapped text
        ha='center',
        va='center',
        color='black',
        fontweight='bold')

    texts.append(text)

# ...and then we can continue with adjusting labels, if we wish.
adjust_text(
    texts,
    ax=ax,
    expand_points=(6, 6),
    expand_text=(6, 6),
    force_text=1,
    arrowprops=dict(
        arrowstyle="-",
        color='darkred',
        lw=1)
```

Scotland - population change 2018-2021

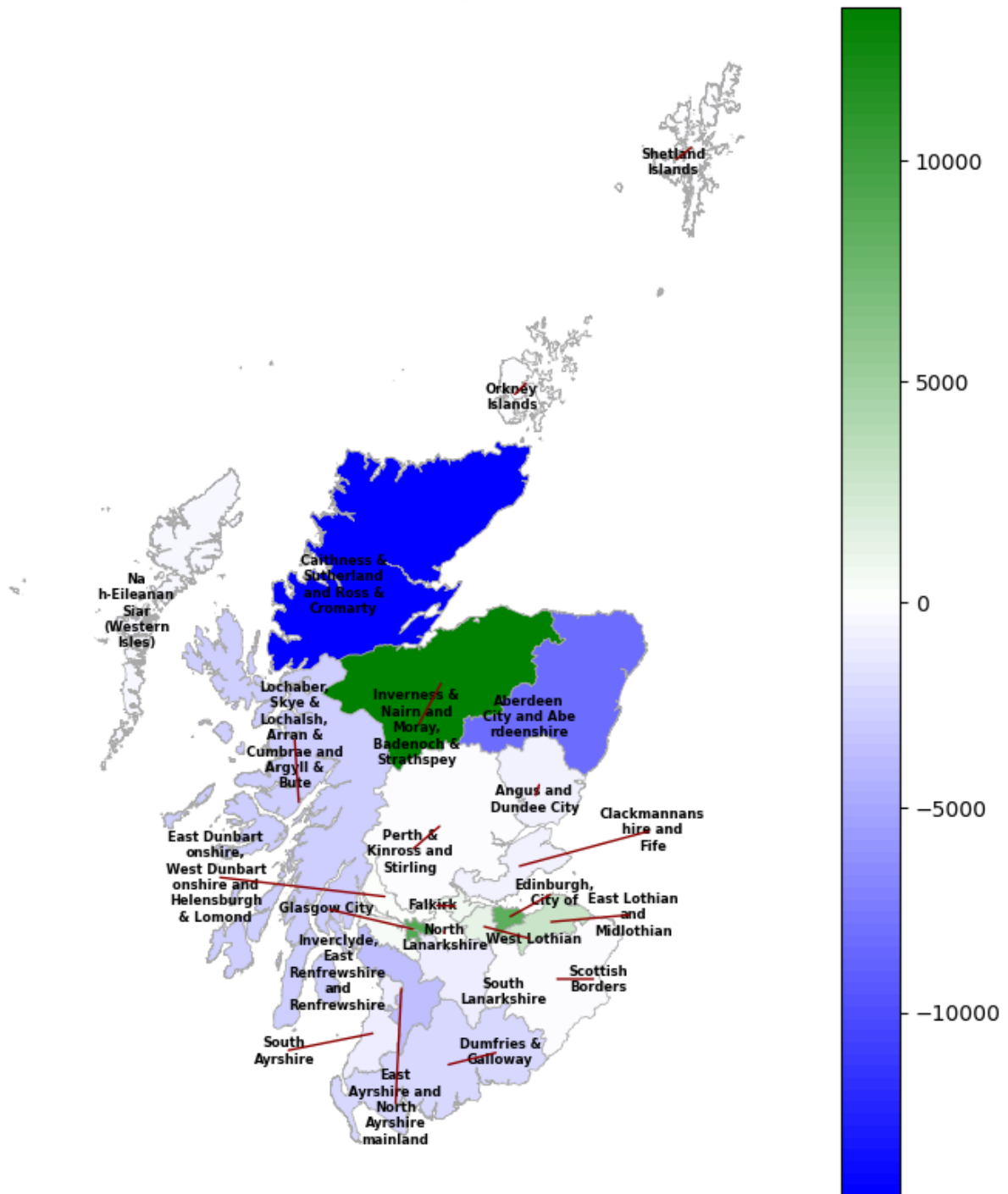


Figure 11: Choropleth map with NUTS 3 level regions labelled.

Exporting Results

Figure 11 can be exported as the rest of maps we did so far. Regarding the GeoDataFrame, which this time contains much more data that is not necessarily geospatial in nature, can be safely exported as a GeoJSON file:

```
plt.savefig("scotland_pop_change.png", dpi=300)  
merged_Scotland.to_file("merged_Scotland.geojson", driver="GeoJSON")
```

8 Interactive Maps With Folium

Once you are familiar with creating maps with *GeoPandas*, another option worth exploring is the use of the *Folium* library to create interactive maps. *Folium* works similarly to *GeoPandas*, but instead of plotting map geometry on abstract axes, it uses *Leaflet.js* to overlay it on pre-rendered images or tilesets displaying geographical features, roads, terrain, satellite imagery, etc. The default option is *OpenStreetMap (OSM)* (see [here](#)), but it can also overlay data on topographic, high-contrast, or low-detail maps. This is especially useful if you wish to showcase your data on a much more detailed map but are not willing to go through the hassle of creating one from scratch. Additionally, maps are exported to an *HTML* file that can be opened and navigated like a digital map. The only drawback is the potentially long generation time for such files.

To save time, we will use the data frame we worked with in the previous example. In the setup below, we will download the file we exported back then:

```
#Necessary libraries
import folium
import pandas as pd
import geopandas as gpd
from branca.colormap import LinearColormap

# Loading data from previous example
geojson_path = 'merged_Scotland.geojson'
merged_Scotland = gpd.read_file(geojson_path)
```

To plot anything with Folium, we first need to define the canvas we will be working with. However, since the final map will be an HTML file with the whole world available, we need to define what the user will see when they open the map, specifically the location (longitude, latitude) and the zoom level the map will start with:

```
map_Scotland = folium.Map(
    location=[55.9533, -3.1883], #Location used to center user view
    zoom_start=7)                #Zoom level the user starts with
```

If you want to choose a specific background, you should define another argument, *tiles*. For example, if you are not interested in the details of the standard OSM map, you might prefer the minimalist CartoDB positron style (see below). Built-in options are available [here](#), and all the custom options shown [here](#).

```
map_Scotland = folium.Map(
    tiles="CartoDB positron",          # Specific map to be drawn as a background
    location=[55.9533, -3.1883],
    zoom_start=7)
```

In contrast to what we have seen before, *Folium* cannot plot within established Python environments, only in the final document. To check how the map we created looks, we should save it and then open the newly created HTML file. Just in case, we will also add a small dot where Edinburgh is located:

```
# Add a point where Edinburgh is
folium.Marker(
    [55.9533, -3.1883], # Coordinates
    popup="Edinburgh"   # What is show to a map user
).add_to(map_Scotland) # Marker is added to our map object

# Save to a file
map_Scotland.save("map_Scotland_interactive1.html")
```

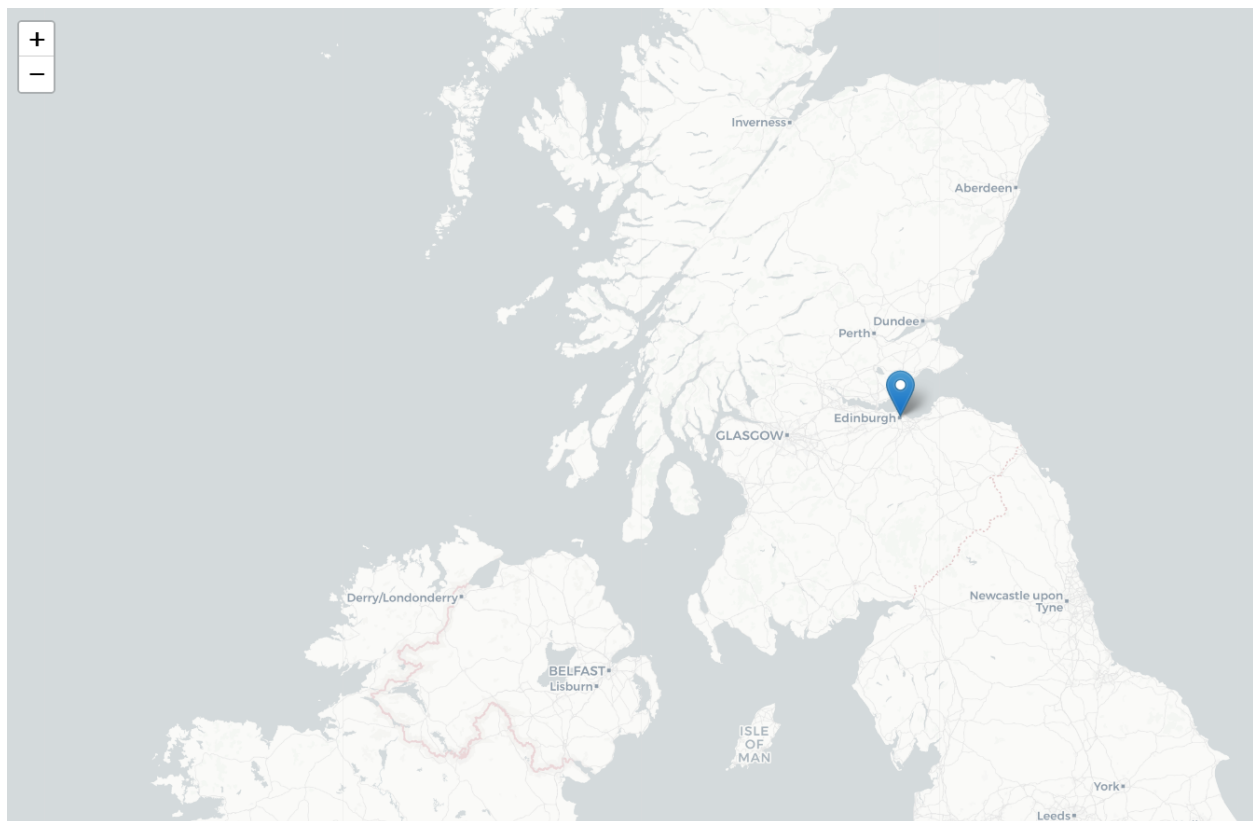


Figure 12: HTML map centered on Edinburgh.

Figure 12 shows a promising starting point for our map. Folium enables plotting a choropleth map directly as a layer through the **.Choropleth()** function. It requires a relatively large amount of input data, including which dataset contains the geospatial data for plotting objects and which data we want to visualize through these objects. This includes specifying the exact column names to consider, with the first always being the one used to identify the objects to be drawn. See the code below:

```
# We create a starting map again
map_Scotland = folium.Map(
    tiles="CartoDB positron",
    location=[55.9533, -3.1883],
    zoom_start=7
)

# ... and we use .Choropleth() to add a layer
folium.Choropleth(          # We plot a Choropleth map
    geo_data=merged_Scotland, # Source of geospatial data
    name='choropleth',       # How we want to name this layer #
    data=merged_Scotland,    # Source of plottable data
    columns=['nuts318cd', 'Pop_change'], # Columns we are interested in
    key_on='feature.properties.nuts318cd', # What columns used for identification
    fill_color="RdBu_r",      # A colour scale to use (_r reverses scale)
    fill_opacity=0.5,         # Infill opacity (0 - transparent, 1 - solid colour) # Line
    line_opacity=0.3,         # opacity
    legend_name='Population Change', # Label for the automatically created legend
    highlight=True            # Highlight regions when hovered
).add_to(map_Scotland)       # We add the layer to our map

# ...and save it html file
map_Scotland.save("map_Scotland_interactive2.html")
```

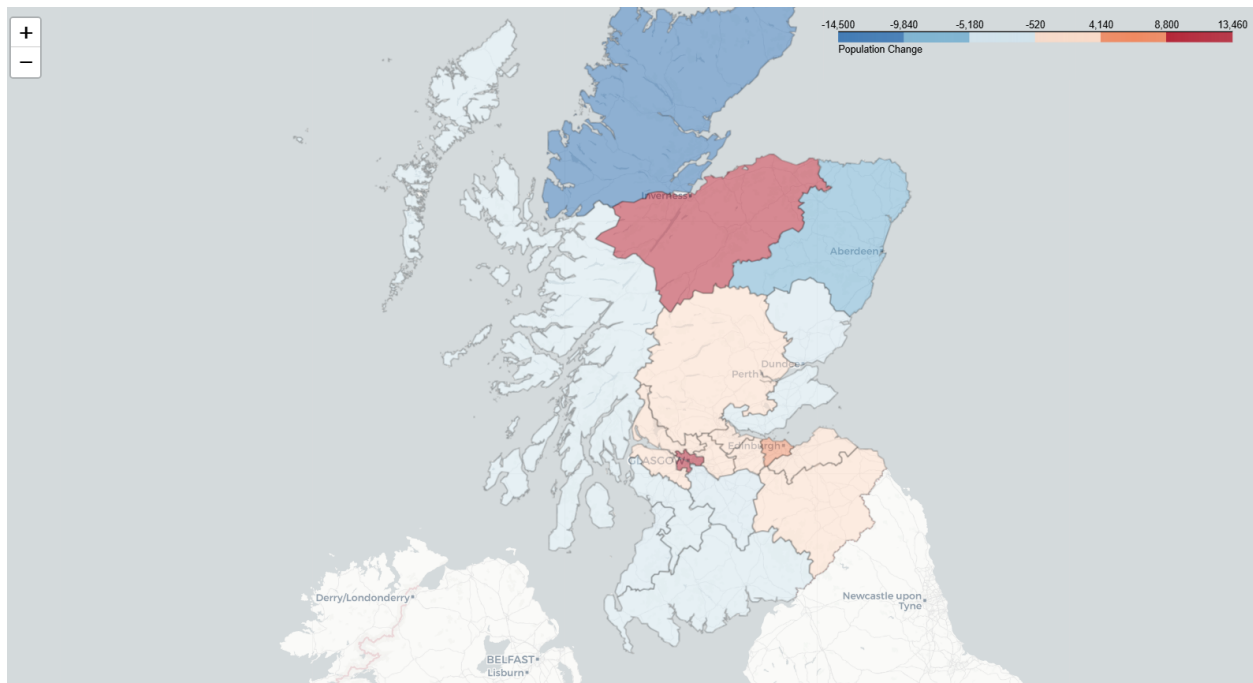



Figure 13: HTML map, with choropleth layer on top.

We can already see a few things that may need adjustment, including the color scale being off (with a mean of -520 used as the scale's central point) and the lack of labels to further explain the map. However, this is also where issues with *Folium* unfortunately arise, as it is still a developing library. Fortunately, we can achieve a lot with other libraries built to support *Folium*, such as *Branca*. In particular, the *branca.colormap* sublibrary allows us to use *LinearColormap()* to encode our own color scale in a gradient format.

However, in the current version of *Folium*, I have noticed that it does not work with *.Choropleth()*, despite widespread information suggesting that it should. That said, we can always plot regions manually using the *.GeoJson()* function from *Folium*. While it is a more basic function, it is also more versatile for experienced users. Additionally, we can take this opportunity to use the *.Popup()* function to create labels that appear when a user hovers over them. The final code may be somewhat complex, but it remains easily adjustable and reusable.

The code containing all these solutions can be seen below:

```
# Again, create the map centered around Scotland
map_Scotland = folium.Map(
    tiles="CartoDB positron",
    location=[55.9533, -3.1883],
    zoom_start=7
)
```

```

# We define our own colour scale
linear_color = LinearColormap(
    ["blue", "white", "red"], index=[          # Colours we include in mapping #
                                              # Index of values, 1 per colour

    merged_Scotland["Pop_change"].min(), 0,
    merged_Scotland["Pop_change"].max()],

    vmin=merged_Scotland["Pop_change"].min(), # Scale end - minimum value
    vmax=merged_Scotland["Pop_change"].max()  # Scale end - maximum value
)

# ...and manually apply it through GeoJson styling
folium.GeoJson(
    merged_Scotland,          # Source of data
    style_function=lambda feature: { # We define how we create features (regions)
        'fillColor': linear_color(    # Fill set to our custom scale
            feature['properties']['Pop_change'] # ..and use data from 'Pop_change') #
        # Just in case, we also define colour when no data is available if feature['properties']
        # ['Pop_change'] is not None else 'gray',
        'color': 'black', # The color of the border around each feature/region 'weight': 0.3,
        # The weight (thickness) of the border line
        'fillOpacity': 0.6 # Transparently of the infill colour
    },
    popup=folium.Popup( # Create a popup for each feature (region) on click
        feature['properties'].get('nuts318nm', 'No Info'), # Data to show
        max_width=300 # Maximum width of the popup
    ),
    name="Styled Choropleth" # Name of the GeoJson
).add_to(map_Scotland)

# Add only the custom LinearColormap legend
linear_color.add_to(map_Scotland)

# Save the map
map_Scotland.save("map_Scotland_interactive3.html")

```

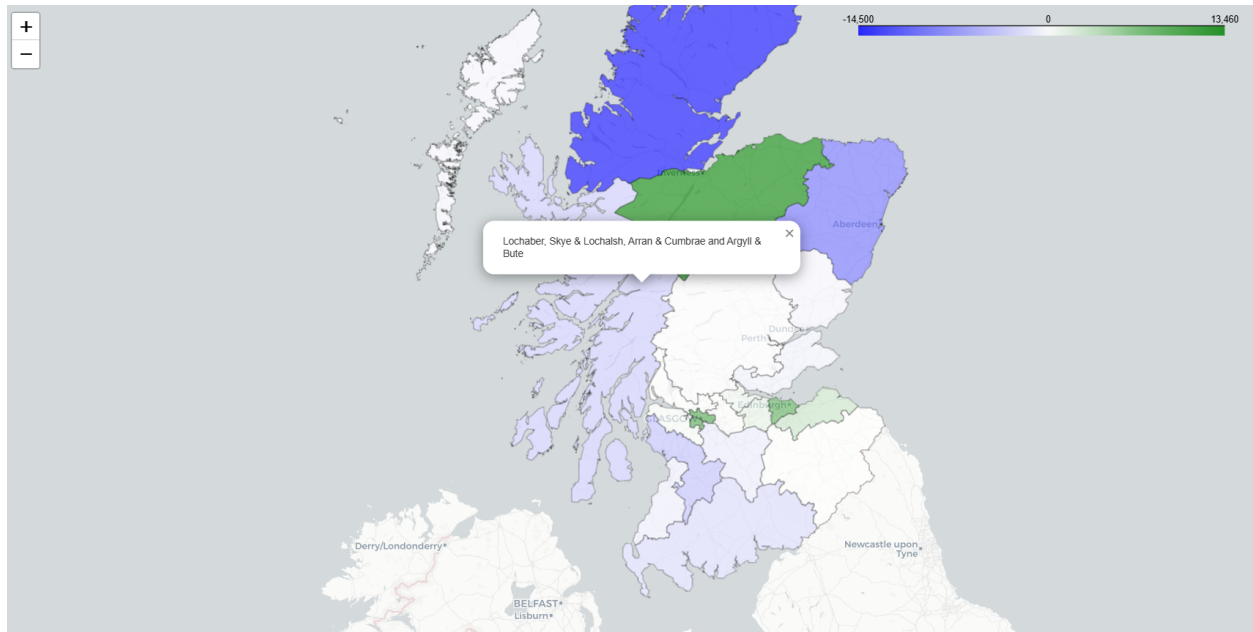


Figure 14: HTML map plotting choropleth manually to fit a custom colour gradient.

Exporting Results

In this case, we do not need to do much, as we already have shareable files, each containing all the data for the map in a single file. Despite the challenges with *Folium*, it is well worth learning and keeping an eye on its frequent updates.

9 Where To Look for Further Resources

As you have likely noticed, there are multiple ways to utilize the functions and methods already shown, not to mention possibilities not covered in this guide. While the release of *GeoPandas* 1.0 is a relatively recent development (2024), there are already plenty of official resources to learn from, along with a slowly growing wealth of user-created content. In contrast, *Folium* is still a developing library with fewer available resources, but it has a vibrant community that contributes to existing official resources and shares knowledge on platforms like YouTube.

Official Online Resources

- 1) **Official Introduction to GeoPandas** geopandas.org
- 2) **GeoPandas User Guide** geopandas.org
- 3) **GeoPandas: Map Examples** geopandas.org
- 4) **Folium User Guide** python-visualization.github.io/folium

Community-Made Online Resources

- 1) **GeoPandas Introduction by Yan Holtz** python-graph-gallery.com
- 2) **GeoPandas Cheat Sheet by Drew Seewald** github.com/atseewal
- 3) **Real Python on YouTube** www.youtube.com/@realpython, especially see this [video](#)
- 4) **Stack Overflow (GeoPandas)** stackoverflow.com
- 5) **Stack Overflow (Folium)** stackoverflow.com

Complex Subject Overviews

- 1) **Introduction to GIS Programming** geog-312.gishub.org
- 2) **LinkedIn Learning: Geospatial Raster Data Analytics** www.linkedin.com/learning

10 A few Words at the End

That is it. I hope this guide has demonstrated the utility of GeoPandas, Folium, and other libraries in everyday mapmaking for academia and data analysis. GeoPandas has matured into a fully developed library with a stable update pathway, ensuring its usefulness for years, if not decades, to come. While Folium has a more sporadic update schedule and is less commonly used, it remains one of the best tools of its kind. However, if you decide to explore other Python-based mapping libraries, that is completely fine. After all, the key is not just sticking to specific tools but understanding the underlying concepts and frameworks that drive spatial analysis, and I am happy I could help you with it.

Given the opportunity, I would like to thank everyone who has worked with me over the past year and a half, including colleagues from the Software Sustainability Institute, EPCC (especially Giacomo Peru and Selina Aragon) and the University of Edinburgh at large. Through coordinating the **DUSC upskilling workshops** and later managing the **Research Software Practices in the Social Sciences** dissemination project, of which this guide is a part, I have learned a great deal about the current academic softwarescape and had the opportunity to meet many fascinating people from vastly diverse, academic as well as government and corporate, backgrounds.

I would also like to thank Mike Spencer from Smart Data Foundry for introducing me to Quarto, through which this guide pdf version was rendered.

Where to Follow - Author

- **LinkedIn** www.linkedin.com/in/andrzej-aleksander-romaniuk-74145292/
- **ORCID** orcid.org/0000-0002-4977-9241
- **ResearchGate** www.researchgate.net/profile/Andrzej-Romaniuk
- **GitHub** github.com/AndrzejRomaniuk
- **BlueSky** [@andrzejromaniuk.bsky.social](https://bsky.social/andrzejromaniuk.bsky.social)

Where to Follow - Software Sustainability Institute

- **Main page** www.software.ac.uk/
- **RSPiSS project page** [.../programmes/research-software-practices-social-sciences](https://www.software.ac.uk/programmes/research-software-practices-social-sciences)
- **YouTube** www.youtube.com/user/software.saved
- **LinkedIn** www.linkedin.com/company/software-sustainability-institute/
- **X** x.com/SoftwareSaved
- **Mastodon** mastodon.social/@SoftwareSaved
- **BlueSky** [softwaresaved.bsky.social](https://software.saved.bsky.social)

A Note About Funding

This work was supported by **UKRI-ESRC** through additional funding The **UK Software Sustainability Institute: Phase 4**, supported through the **UKRI Digital Research Infrastructure Programme** through grant number **AH/Z000114/1**.

