Introduction to spatial data with Geopandas

Lino Galiana
Version of September 29, 2026

Geocoded data have been more and more used these recent years in research, public policies or business decisions. Data scientists use them a lot, whether they come from open data or geocoded digital traces. For spatial data, the `GeoPandas` package extends the functionalities of the `Pandas` ecosystem to enable handling complex geographical data in a simple manner. This chapter presents the challenge of handling spatial data with `Python`.

⚠️ **Automatically generated PDF.** This document is produced from the course website and may contain formatting issues. For an up-to-date version adapted to your reading, visit the course page: pythonds.linogaliana.fr.

Table of contents

1 Introduction **2**
- 1.1 What's so special with spatial data ? 2
- 1.2 Structure of spatial data 2
- 1.3 Spatial data analysis is not new 2
- 1.4 Where can we find French spatial data? 3
- 1.5 Objectives 4
- 1.6 Data 4
- 1.7 Preliminary installations 4

2 From `Pandas` to `Geopandas` **5**
- 2.1 Anatomy of a `GeoPandas` object 5

3 Reading and enriching spatial data **6**
- 3.1 Shapefile (`.shp`) and geopackage (`.gpkg`) formats 6
- 3.2 GeoJSON and TopoJSON 6
- 3.3 Other data formats 7
- 3.4 An exercise to discover spatial data 7

4 The coordinate reference system (CRS) **9**
- 4.1 Principle 9
- 4.2 The Mercator System 10
- 4.3 Handling with `GeoPandas` 11
- 4.4 Exercise to understand how important a projection system can be 12

5 Import and explore spatial datasets **15**
- 5.1 Locate data on a map 15
- 5.2 Application 16
- 5.3 Geometric operations 19

6 Enrichments through the spatial dimension: spatial joins **21**
- 6.1 Principle 21
- 6.2 Example: locating stations in their neighborhood 21
- 6.3 Additional exercise 24

7 References **26**
- Informations additionnelles 26

Bibliography **28**

Skills you will acquire in this chapter

- Get familiar with the core data structures in **GeoPandas**: `GeoSeries` and `GeoDataFrame`, which extend `pandas.Series` and `pandas.DataFrame` for working with spatial geometries
- Read geospatial data from various formats (GeoPackage, Shapefile, GeoJSON…) using the `.read_file()` method
- Use standard `pandas` methods on a `GeoDataFrame`, such as `head()`, while keeping spatial data intact
- Quickly visualize geospatial data using the built-in `.plot()` method to generate static maps
- Understand and work with coordinate reference systems (CRS): set a CRS using `set_crs()` and reproject data with `to_crs()`

1 Introduction

1.1 What's so special with spatial data ?

Previous chapters have introduced how structured data can be leveraged using the `Pandas` library. We will now explore the analysis of more complex data, namely spatial data. These datasets are more sophisticated than tabular data since they not only share the properties of tabular data (flattened data in a structure of columns and rows) but also include an additional geographical dimension. This dimension can be more or less complex depending on the nature of the data: it can be points (2D location coordinates), lines (a series of points), directional lines (the same structure as before but with a direction), polygons (a set of points), etc. This diversity of geographic objects aims to allow information systems and representations of numerous geographic objects.

From now on, we will refer to *"spatial data"* as all data that pertain to the geographical characteristics of objects (location, contours, links). The geographical characteristics of objects are described using a **coordinate system**. These allow the representation of the geographic object in a two-dimensional Euclidean space (x, y). The transition from the real space (the Earth, which is a three-dimensional sphere) to the planar space is done through a **projection system**.

1.2 Structure of spatial data

Spatial data typically gather two types of data:

1. **Geographical data** (or geometries): geometric objects such as points, vectors, polygons, or meshes (*raster*). Example: the shape of each municipality, the coordinates of a building, etc. ;
2. **Attributive data** (or attributes): measurements and characteristics associated with geometric objects. Example: the population of each municipality, the number of windows and the number of floors of a building, etc.

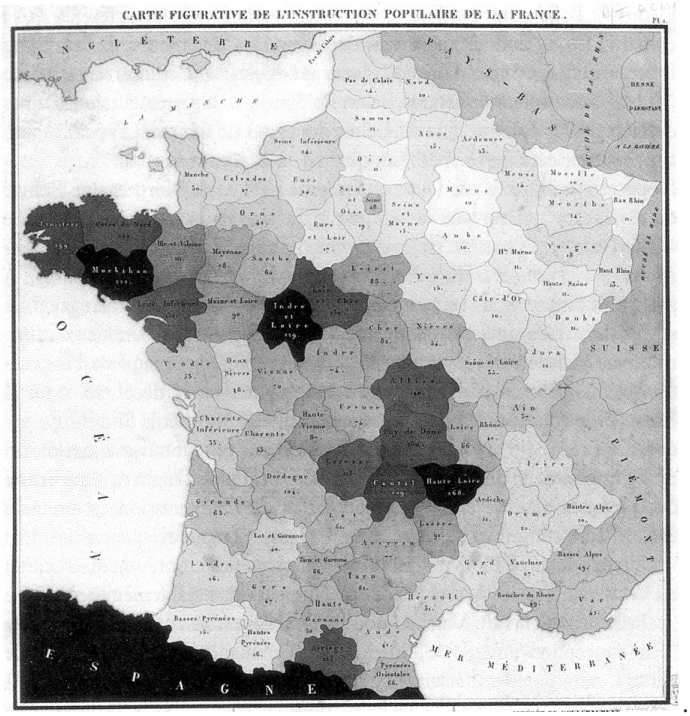
Spatial data are frequently processed using a geographic information system (GIS), which is an information system capable of storing, organizing, and presenting alphanumeric data spatially referenced by coordinates in a reference system (CRS). `Python` has functionalities allowing it to perform the same tasks as a GIS (processing spatial data, creating cartographic representations).

1.3 Spatial data analysis is not new

Initially used mainly for military or administrative purposes, cartographic production has been very common to represent socioeconomic information since at least the 19th century. The most well-

known representation in this field is the choropleth map[1].

According to C.-houh Chen, W. Härdle, A. Unwin, and M. Friendly [1], the first representation of this type was proposed by Charles Dupin in 1926 to represent education levels in France. The emergence of choropleth maps is indeed inseparable from the organization of power into entities considered unitary: world maps often represent color gradients based on nations, national maps based on administrative levels (regions, departments, municipalities, but also states or *landers*).



While the production of geographical information may have been closely tied to military use and administrative management of a territory, the digitalization of the economy has multiplied the actors concerned with collecting and making geographical data available. Thus, handling and representing spatial data is no longer the exclusive domain of geographers and geomatics engineers. Data scientists must be able to quickly explore the structure of a geographic dataset just as they would with traditional tabular datasets.

1.4 Where can we find French spatial data?

During our journey of discovering `Pandas`, we have already encountered some geolocated sources, notably produced by INSEE. This institution publishes numerous local statistics, such as the Filosofi data we encountered in the previous chapter. Beyond INSEE, all institutions of the public statistical system (INSEE and ministerial statistical services) publish many aggregated data sources at different geographical scales: sub-municipal level (e.g., 200m tiles), municipal level, or supra-municipal levels (administrative or study area zonings).

More generally, many French administrations outside the public statistical system disseminate geographic data on data.gouv. For example, we previously exploited a dataset from ADEME whose geographical dimension was the municipality.

The central player in the French public geographic data ecosystem is IGN. Well known to hiking enthusiasts for its *"Top 25"* maps, which can be found on the geoportail, IGN is also responsible for mapping the legal boundaries of French administrative entities (AdminExpress base), forests (BDForêt), roads (BDRoute), buildings (BDTopo), etc. We briefly mentioned the `cartiflette` library

[1]Despite all its limitations, which we will revisit, the choropleth map is nonetheless informative. Knowing how to produce one quickly to grasp the main structuring facts of a dataset is particularly useful.

in the previous chapter, which allows flexible retrieval of administrative map bases (AdminExpress base) with `Python`; we will go further in this chapter.

The public authorities are no longer the only ones producing and disseminating spatial data. Collecting GPS coordinates has become almost automatic, and many actors collect, exploit, and even sell geospatial data on their users. These data can be very precise and rich on certain issues, such as mobility. However, it is necessary to keep in mind when wanting to extrapolate statistics built on these data that they concern users of the specific service, who may not be representative of the behaviors of the population as a whole.

1.5 Objectives

This chapter illustrates some central principles of data analysis through practical examples:

- Manipulations on the attributes of datasets;
- Geometric manipulations;
- Management of cartographic projections;
- Quick creation of maps (to be explored further in a later chapter).

1.6 Data

In this chapter, we will use the following data:

- Free floating bike rents (Vélib) data;
- French administrative design through a `Python` package named `cartiflette` which facilitates the retrieval of this source.

1.7 Preliminary installations

This tutorial requires some package installations to be reproducible

```
!pip install pandas fiona shapely pyproj rtree
!pip install contextily
!pip install geopandas
!pip install topojson
```

To be able to run this tutorial, the following imports will be useful.

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

To install `cartiflette`, you need to use the following commands from a `Jupyter Notebook` (if you use the command line directly, you can remove the `!` at the beginning of the line):

```
!pip install py7zr geopandas openpyxl tqdm s3fs --quiet
!pip install PyYAML --quiet
!pip install cartiflette --quiet
```

2 From `Pandas` to `Geopandas`

The `Geopandas` package is a toolkit designed to facilitate the manipulation of spatial data. **The great strength of `Geopandas` is that it allows for the manipulation of spatial data as if it were traditional data**, because it is based on the ISO 19125 *simple feature access* standard defined jointly by the *Open Geospatial Consortium (OGC)* and the *International Organization for Standardization (ISO)*. `Geopandas` relies on `Pandas` for handling the tabular aspect of spatial data and on `Shapely` and `GDAL` for geometric manipulations. However, just as `Pandas` allows you to use `Numpy` without knowing it, working with `Geopandas` lets you leverage `Shapely` and `GDAL` without having to deal with them directly.

Compared to a standard *DataFrame*, a `Geopandas` object includes an additional column: `geometry`. This column stores the coordinates of geographic objects (or sets of coordinates in the case of boundaries). A `Geopandas` object inherits the properties of a `Pandas` *DataFrame* but offers methods tailored to the handling of spatial data. Therefore, with `GeoPandas`, you have attributes based on the principle of *tidy* data discussed in the previous chapter while the associated geometry is managed consistently alongside the attributes.

Thus, with `Geopandas`, you can perform attribute manipulations as with `pandas`, but you can also manipulate the spatial dimension of the data. Specifically, you can:

- Calculate distances and areas;
- Aggregate zonings quickly (e.g., grouping municipalities into departments);
- Find out which municipality a building is located in based on its geographic coordinates;
- Recalculate coordinates in another projection system;
- Create a map quickly and easily.

> Tip
>
> Manipulating data in a `Geopandas` object is significantly slower than in a traditional `DataFrame` (since `Python` has to handle geographic information during data manipulation). When working with large datasets, it may be preferable to perform operations on the data before attaching geometry to it.

Compared to specialized software like `QGIS`, `Python` allows for the automation of data processing and visualization. In fact, `QGIS` itself uses `Python`...

2.1 Anatomy of a `GeoPandas` object

In summary, a `GeoPandas` object consists of the following elements:

```
Simple feature collection with 6 features and 2 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: 2.169349 ymin: 48.80894 xmax: 2.320866 ymax: 48.92685
Geodetic CRS:  WGS 84
```

	NOM	POPULATION	geometry
0	Paris	2165423	POLYGON ((2.3642 48.8164, 2.38077 48.8217, 2.3...
1	Levallois-Perret	66082	POLYGON ((2.28739 48.90364, 2.28427 48.90229, ...
2	Bois-Colombes	28841	POLYGON ((2.26639 48.90629, 2.26494 48.91027, ...
3	Malakoff	30950	POLYGON ((2.27818 48.81425, 2.27449 48.81343, ...
4	Clichy	63089	POLYGON ((2.30377 48.89415, 2.30769 48.89601, ...
5	Nanterre	96277	POLYGON ((2.2291 48.90603, 2.22795 48.90792, 2...

1. **Attributes**: These are the values associated with each geographic level. This is the usual tabular dimension, treated similarly to a standard `Pandas` object.
2. **Geometries**: These are numerical values interpreted to represent the geographic dimension. They allow representation in a specific reference frame (the reference system).
3. **Reference System**: This system transforms positions on the globe (3D with an asymmetrical sphere) into a 2D plane. There are numerous systems, identifiable by an EPSG code (4326, 2154, etc.). Their manipulation is facilitated by `Geopandas`, which relies on `Shapely`, just as `Pandas` relies on `Numpy` or `Arrow`.

3 Reading and enriching spatial data

In the previous chapter, we discussed flat data formats such as `CSV` and newer formats like `Parquet`. These are suitable for tabular data. For geographic data, which store information across multiple dimensions (attributes and geometry), there are specialized formats.

3.1 Shapefile (`.shp`) and geopackage (`.gpkg`) formats

The historical format for storing spatial data is the shapefile. It is a proprietary format developed by ESRI that has become a de facto standard. In this format, data

3.2 GeoJSON and TopoJSON

The development of a format to rival the hegemony of the shapefile is intrinsically linked to the emergence of web technologies in the cartography sector. These web technologies rely on Javascript and adhere to the JSON format standards.

The GeoJSON format stores both attributes and geometries in a single file. This makes it quite convenient to use and it has become the preferred format among web developers. However, storing all information in one file can make it quite large if the geometries are very detailed, though it is still less bulky than the shapefile. `GeoPandas` is well-equipped to read files in the GeoJSON format, and data-sharing platforms like `data.gouv` favor this format over the shapefile.

To reduce file size, the TopoJSON format has recently emerged. It is built on the same principles as GeoJSON but reduces the volume of stored geometric data by simplifying it, retaining only arcs and directions between points. As this format is new, it is not yet well-integrated into the `Python` ecosystem.

The newest addition to the gallery of geospatial formats is the `Parquet` format. We devote an entire chapter to this format, presenting its many practical features for *data scientists* (Parquet and Data in the cloud).

Originally, this format was designed for tabular data, that is, data without a geographic dimension. Nevertheless, it can also efficiently store complex information, such as multidimensional vectors representing geographic coordinates. Since early 2026, data structures specifically designed for geospatial information have been added to the `Parquet` standard. As a result, a standard `Parquet` file can now represent geographic information (more details here).

If it gains the same popularity within the geospatial community as it has in other domains where it has already become established, this format is likely to become essential for geographic analysis in the coming years.

3.3 Other data formats

The ecosystem of geographic data formats is much more fragmented than that of structured data. Each format offers advantages that make it suitable for certain types of data but has limitations that prevent it from becoming a standard for other types of data.

For example, GPS data extracted from various applications (e.g., `Strava`) are stored in the GPX format. This format is particularly well-suited for geolocated traces with altitude. However, it is not the most appropriate format for storing directional lines, which are essential for routing applications.

The shapefile and geojson formats are sufficiently versatile to adapt to various types of geographic data, even if they are not the optimal format for every type of data. Within this generalist class of formats, the Geoparquet might be the next trending format. As its name suggests, it is an extension of the Parquet format to geographic data. This format is not yet mature but is worth keeping an eye on, as the large user base of the Parquet ecosystem could lead to a rapid shift if a stable implementation of Geoparquet emerges.

This page provides a more detailed comparison of the main geographic data formats. The help section of `Geopandas` offers code snippets for various situations.

3.4 An exercise to discover spatial data

The goal of this exercise is to illustrate the similarity between `GeoPandas` objects and the `Pandas` objects we previously explored.

We will directly import the `AdminExpress` data (official boundaries of municipalities produced by IGN) using `cartiflette`:

> Exercise 1: Discovering spatial objects
>
> First, we will retrieve geographic data using the `cartiflette` package and its `carti_download` function.
>
> 1. Use the code below to download administrative borders for the departments of the inner suburbs (75, 92, 93, and 94) in a simplified manner using the `cartiflette` package
> 2. Look at the first few rows of the data. Identify the difference from a standard dataframe.
> 3. Display the projection system (attribute `crs`) of `communes_borders`. This controls the transformation of the three-dimensional terrestrial space into a flat surface. Use `to_crs` to transform the data into Lambert 93, the official system (EPSG code 2154).
> 4. Display the municipalities of Hauts de Seine (department 92) and use the `plot` method.
> 5. Keep only Paris and plot the borders on a map: what is the problem for an analysis of intramural Paris?
>
> You will quickly notice the problem. We do not have the boundaries of Parisian districts, which greatly impoverishes the map of Paris.
>
> 6. This time, use the argument `borders="COMMUNE_ARRONDISSEMENT"` to obtain a consolidated map with the municipalities and districts in large cities. Convert to Lambert 93.

```
# 1. Chargement des données de Cartiflette
from cartiflette import carti_download
communes_borders = carti_download(
    crs = 4326,
    values = ["75", "92", "93", "94"],
    borders="COMMUNE",
    vectorfile_format="geojson",
    filter_by="DEPARTEMENT",
```

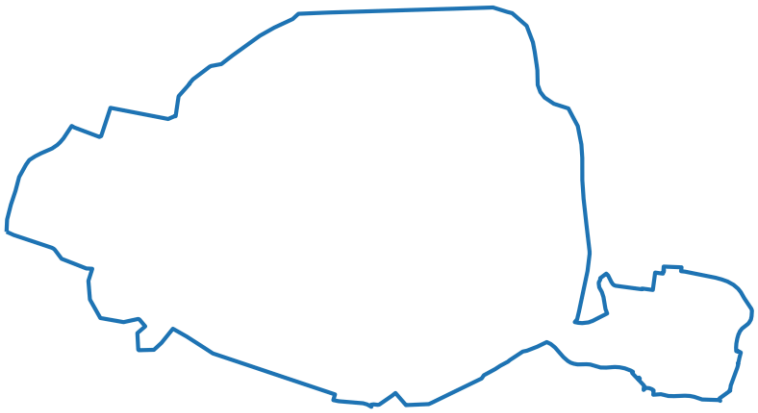
```
    source="EXPRESS-COG-CARTO-TERRITOIRE",
    year=2022)
```

The visualization provided in the question shows that our `GeoDataFrame` includes a `geometry` column containing the necessary information to determine the municipal boundaries.

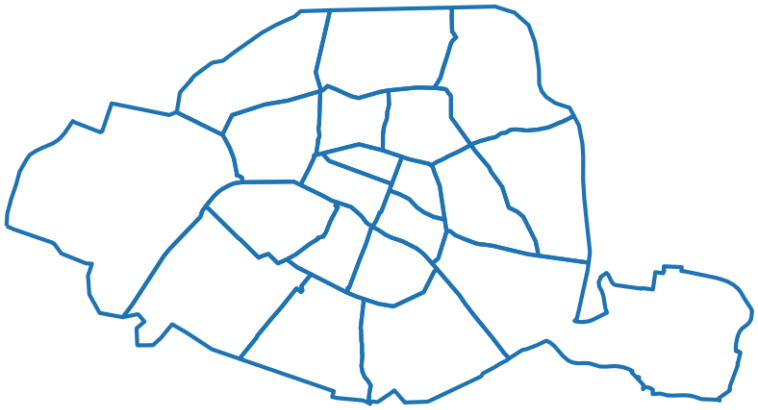
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

The data is in WGS84, we reproject it to Lambert 93

In question 5, the issue for Paris is easily noticeable: the boundaries of the arrondissements are missing. This significantly reduces the detail of the map of Paris.



At the end of question 6, we obtain the expected map for inner Paris:



4 The coordinate reference system (CRS)

4.1 Principle

Spatial data is richer than traditional data because it usually includes additional elements to place objects in a Cartesian space. This additional dimension can be simple (a point has two additional pieces of information: x and y) or quite complex (polygons, lines with direction, etc.).

Cartographic analysis, therefore, borrows concepts from geometry to represent objects in space. **Projections** are at the heart of managing spatial data. These projections consist of transforming a position in terrestrial space into a position on a plane. This is a projection operation from a three-dimensional space into a two-dimensional space. This post offers rich elements on the subject, including the following image that clearly shows the principle of a projection:

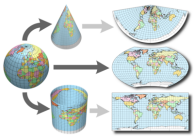


Figure 1: Different types of projection

This operation is not neutral. One of the consequences of the remarkable theorem of Gauss is that the surface of the Earth cannot be mapped without distortion. A projection cannot simultaneously keep distances and angles intact (i.e., positions). Thus, there is no universally better projection, which opens the door to the coexistence of many different projections, designed for different tasks. A poor projection system can distort visual appreciation and also cause errors in calculations on the spatial dimension.

Projection systems are subject to international standards and are often designated by so-called EPSG codes. This site is a good reference. The most common for French users are as follows (more info here):

- `2154`: Lambert 93 projection system. This is the official projection system. Most data published by the administration for the metropolis is available in this projection system.
- `27572`: Lambert II extended. This is the former official projection system. Older spatial data may be in this format.
- `4326`: WGS 84 or pseudo-Mercator system or *Web Mercator*. This is not actually a projection system but a coordinate system (longitude/latitude) that simply allows angular positioning on

the ellipsoid. It is used for GPS data. This is the most common system, especially when working with web map backgrounds.

4.2 The Mercator System

As mentioned above, one of the best-known projections is the Web Mercator projection (EPSG code 3857), which projects spherical longitude and latitude data from the WGS84 system (EPSG 4326) onto flat maps. This is the geodetic system used by the American GNSS **GPS**, which is used by the vast majority of global navigation systems, let alone Google Maps. This projection preserves angles, which means it distorts distances. It was originally designed to represent the Northern Hemisphere. The farther you move away from it, the more distances are distorted. This leads to well-known distortions (Greenland being oversized, Africa reduced in size, Antarctica being immense, etc.). However, the Mercator projection keeps positions intact. This property explains its use in GPS systems and navigation map backgrounds like *Google Maps*. It is a projection primarily designed for navigation, not for representing socio-economic information on Earth. This projection is inseparable from the great explorations of the Renaissance, as highlighted by this Twitter thread by Jules Grandin.

Figure 2: *Example of country re-projection from the site thetruesize.com*

Figure 3: “Don’t trust the Mercator projection” on `Reddit`

To go further, the following interactive map, created by Nicolas Lambert and derived from this *Observable notebook*, illustrates the distorting effect of the Mercator projection, among others, on our perception of country sizes.

There are actually many possible representations of the world, some more intricate than others. Projections are numerous, and some can have surprising shapes. For example, the Spilhaus projection proposes centering the view on the oceans rather than land. That’s why it is sometimes referred to as the world as seen by fish.

Tip for France

For France, in the WGS84 system (4326):

- Longitude (x) revolves around 0° (from −5.2 to +9.6 to be precise)
- Latitude (y) around 45 (between +41.3 and +51.1)

In the Lambert 93 system (2154):

- Coordinates x: between 100,000 and 1,300,000
- Latitude (y): between 6,000,000 and 7,200,000

More details

4.3 Handling with `GeoPandas`

Regarding the management of projections with `GeoPandas`, the official documentation is very well done. It provides the following warning that is good to keep in mind:

> Be aware that most of the time you don’t have to set a projection. Data loaded from a reputable source (using the geopandas.read_file() command) should always include projection information. You can see an object’s current CRS through the GeoSeries.crs attribute.
>
> From time to time, however, you may get data that does not include a projection. In this situation, you have to set the CRS so geopandas knows how to interpret the coordinates.

Figure 4: *Image borrowed from XKCD https://xkcd.com/2256/ which can also be found on https://blog.chrislansdown.com/2020/01/17/a-great-map-projection-joke/*

The two main methods for defining the projection system used are:

- `df.set_crs`: This command is used to specify what projection system is used, i.e., how the coordinates *(x,y)* are related to the Earth's surface. **This command should not be used to transform the coordinate system, only to define it**.
- `df.to_crs`: **This command is used to project the points of a geometry into another, i.e., to recalculate the coordinates according to another projection system**.

In the specific case of producing a map with an OpenStreetMaps background or a dynamic leaflet map, it is necessary to de-project the data (for example from Lambert-93) to land in the unprojected WGS 84 system (code EPSG 4326). This site dedicated to geographic projections can be useful for finding the projection system of a file where it is not indicated.

The next exercise will help convince you of the importance of delegating as much of the projection system management as possible to `GeoPandas`. The issue is not only about the relevance of the representation of geographic objects on the map. Indeed, all geometric operations (area calculations, distance, etc.) can be affected by the choices made regarding the projection system.

4.4 Exercise to understand how important a projection system can be

Here is some code using `cartiflette` again to retrieve French borders (divided by region):

```
from cartiflette import carti_download
france = carti_download(
      values = ["France"],
      crs = 4326,
      borders = "REGION",
      vectorfile_format="geojson",
      simplification=50,
```

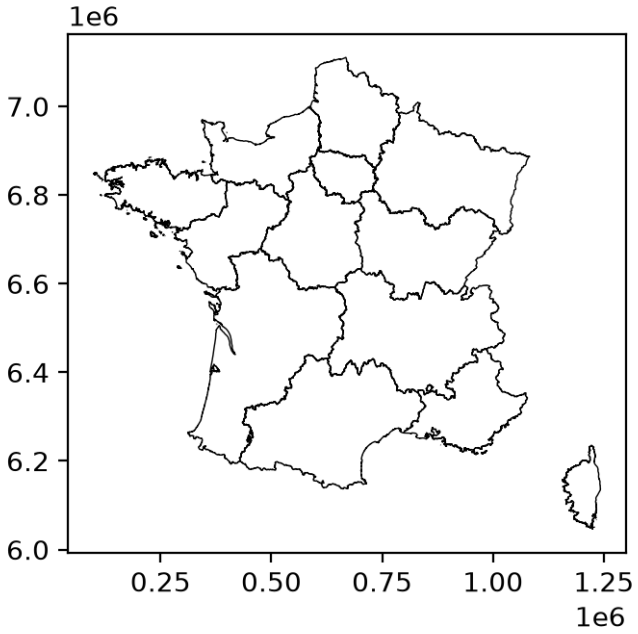
```
        filter_by="FRANCE_ENTIERE",
        source="EXPRESS-COG-CARTO-TERRITOIRE",
        year=2022)
france = france.loc[france['INSEE_REG']>10]
```

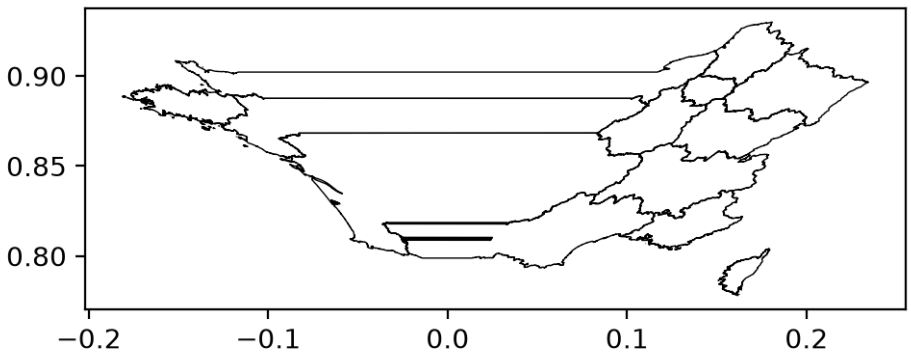
Exercise 2: Projections, representations, and approximations

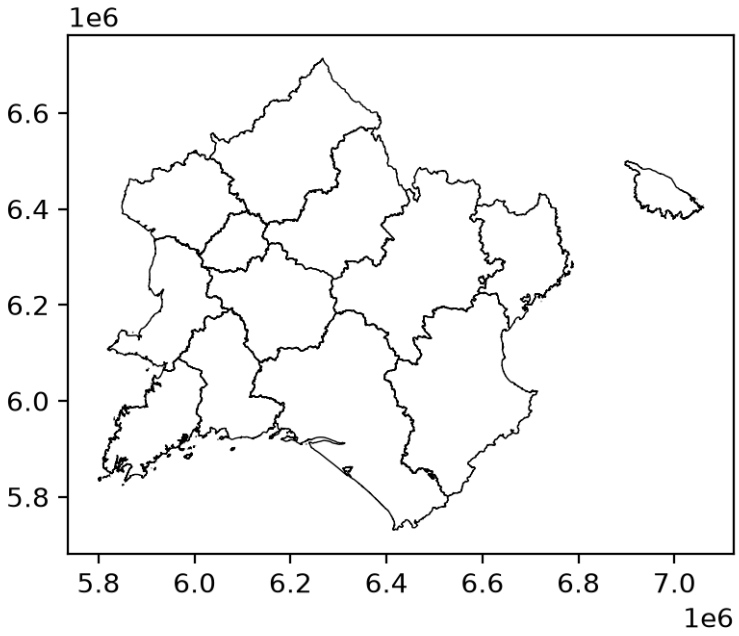
1. Experiment with representing the borders of France using several projections:

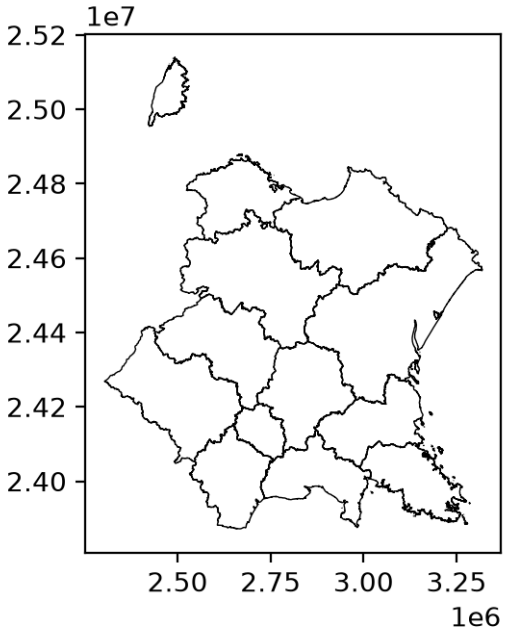
- Mercator WGS84 (EPSG: 4326)
- Healpix projection (`+proj=healpix +lon_0=0 +a=1`)
- Projection for Tahiti (EPSG: 3304)
- Albers projection for the United States (EPSG: 5070)

2. Calculate the area in km^2 of the French regions in the following two projection systems: World Mercator WGS84 (EPSG: 3395) and Lambert 93 (EPSG: 2154). Calculate the difference in km^2 for each region.







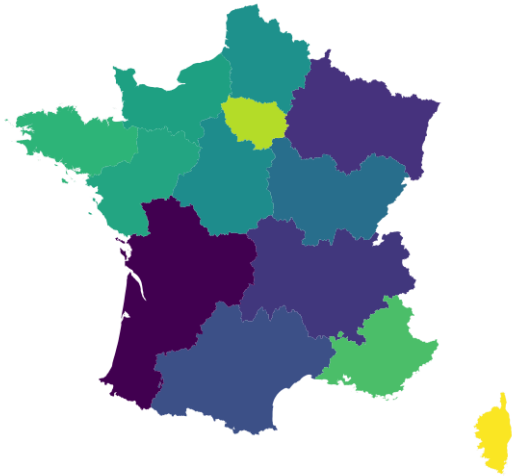


With question 1 illustrating some pathological cases, we understand that projections have a distorting effect which is clearly visible when we represent them side by side in the form of maps:

However, the problem is not just visual; it is also numerical. Geometric calculations lead to quite notable differences depending on the reference system used.

We can represent these approximations on a map[2] to get an idea of the regions where the measurement error is the greatest (subject of question 2).

[2]This map is not too neat; it's normal. We will see how to make beautiful maps later.



This type of measurement error is normal at the scale of the French territory. Projections inherited from Mercator distort distances, especially when approaching the equator or the poles.

Therefore, it is always necessary to reproject the data into the Lambert 93 system (the official system for mainland France) before performing geometric calculations.

5 Import and explore spatial datasets

Often, the communal division serves only as a background for maps, providing context and reference points. Generally, it is used to contextualize another dataset.

To illustrate this approach, we will use the data on the locations and capacities of Vélib stations, available on the open data site of the city of Paris, and directly queryable via the URL https://opendata.paris.fr/explore/dataset/velib-emplacement-des-stations/download/?format=geojson&timezone=Europe/Berlin&lang=fr. This dataset will help us illustrate some classic issues in spatial data analysis.

```
velib_data = 'https://opendata.paris.fr/explore/dataset/velib-emplacement-des-stations/
download/?format=geojson&timezone=Europe/Berlin&lang=fr'
stations = gpd.read_file(velib_data)
stations.head(2)
```

	capacity	stationcode	coordonnees_geo	name		geometry
0	23	9021	[48.87778639599673, 2.337429821491241]	Saint Georges - d'Aumale	POINT	(2.33743 48.87779)
1	50	23009	[48.88873214926474, 2.288[illegible]	Antoine Louise Michel - Tocqué	POINT	(2.28816 48.88873)

5.1 Locate data on a map

The first step, before thoroughly exploring the data, is to display it on a contextual map to ensure the geographic coverage of the data. In our case, this will give us an intuition about the locations of the stations, particularly their uneven density in the Parisian urban space.

5.2 Application

In the next exercise, we propose to quickly create a map with three layers:

- The locations of stations as points;
- The borders of communes and arrondissements for context;
- The borders of departments with wider lines for additional context.

We will delve deeper into cartographic work in the next chapter. However, being able to quickly position your data on a map is always useful in exploratory work.

Before the exercise, use the following function from the `cartiflette` package to retrieve the base map of the departments of the inner suburbs:

```
idf = carti_download(
      values = ["11"],
      crs = 4326,
      borders = "DEPARTEMENT",
      vectorfile_format="geojson",
      filter_by="REGION",
      source="EXPRESS-COG-CARTO-TERRITOIRE",
      year=2022)

petite_couronne_departements = (
  idf
  .loc[idf['INSEE_DEP'].isin(["75","92","93","94"])]
  .to_crs(2154)
)
```

Exercise 3: Import and explore the Velib data

Let's start by retrieving the data needed to produce this map.

1. Check the geographic projection of `station` (attribute `crs`). If it is different from the commune data, reproject the latter to the same projection system as the Velib stations.
2. Keep only the top 50 stations (variable `capacity`).

We can now build the map sequentially using the `plot` method, with the help of this documentation.

3. First, use `boundary.plot` to represent the base layer of commune and arrondissement boundaries:
   - Use the options `edgecolor = "black"` and `linewidth = 0.5`
   - Name this object `base`
4. Add the layer of departments with the options `edgecolor = "blue"` and `linewidth = 0.7`.
5. Add the positions of the stations and adjust the size according to the `capacity` variable. The aesthetics of the obtained points can be controlled with the options `color = "red"` and `alpha = 0.4`.
6. Remove the axes and add a title with the options below:

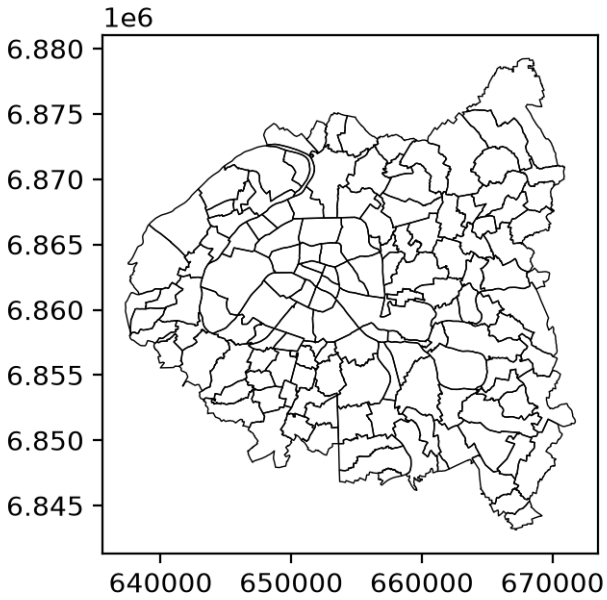
```
base.set_axis_off()
base.set_title("The 50 main Velib stations")
```

7. Following the model below, use the `contextily` package to add a contextual OpenStreetMap base map:

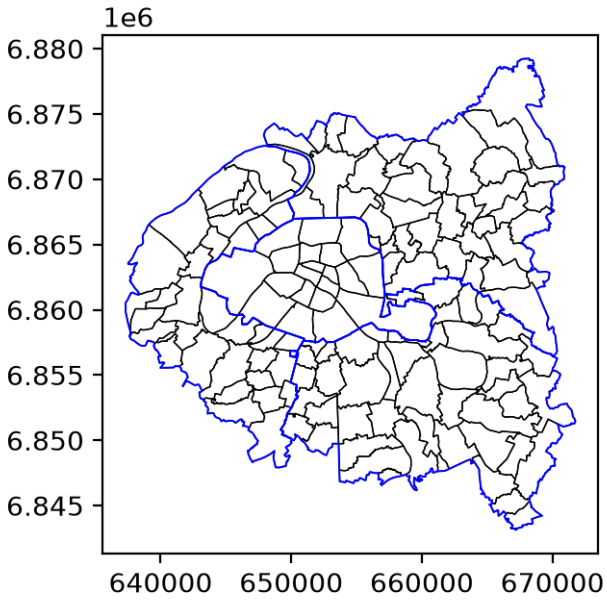
```
import contextily as ctx
ax = ...
ctx.add_basemap(ax, source = ctx.providers.OpenStreetMap.Mapnik)
```

⚠️ `contextily` expects data in the Pseudo Mercator representation system (EPSG: 3857), so it will be necessary to reproject your data before creating the map.

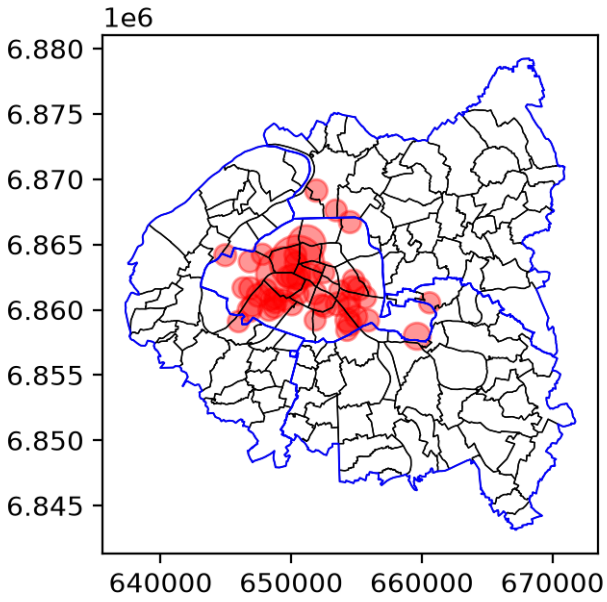
The base layer obtained after question 3



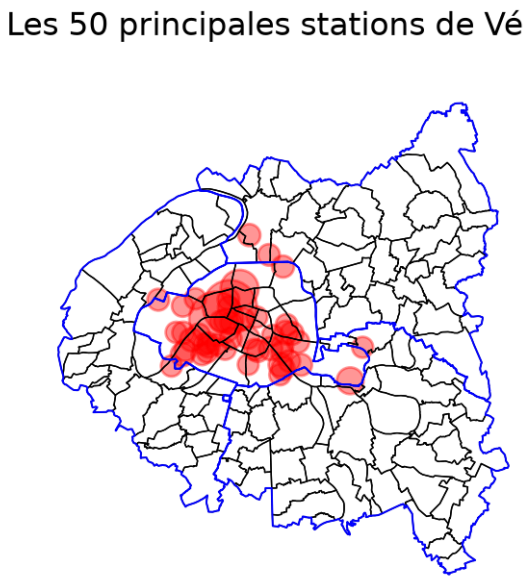
Then by adding the departmental boundaries (question 4).

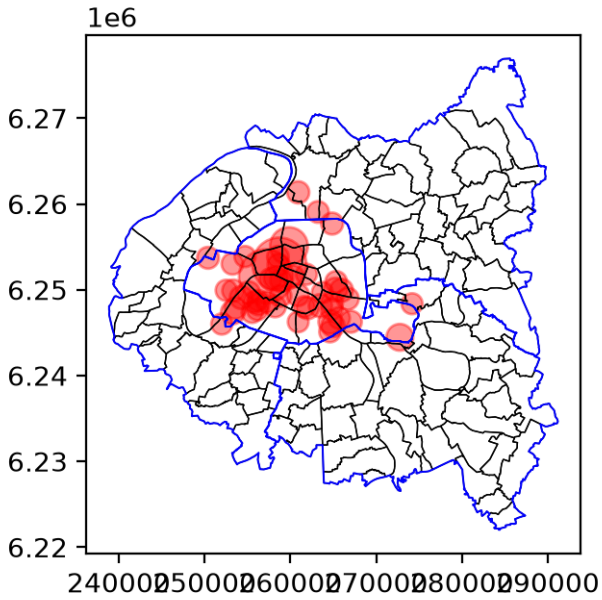


Then the stations (question 5).



Next, if we remove the axes (question 6), we get:





The map speaks for itself. However, for those less familiar with Parisian geography, it could be made even clearer with the addition of a contextual background map *openstreetmap*. In fine, this results in the following map:

```
<Axes: title={'center': 'Les 50 principales stations de Vélib'}>
```

```
<Figure size 1100x700 with 0 Axes>
```

5.3 Geometric operations

In addition to simplified graphical representation, the main advantage of using `GeoPandas` is the availability of efficient methods for manipulating spatial dimensions. Many of these methods come from the package `Shapely`.

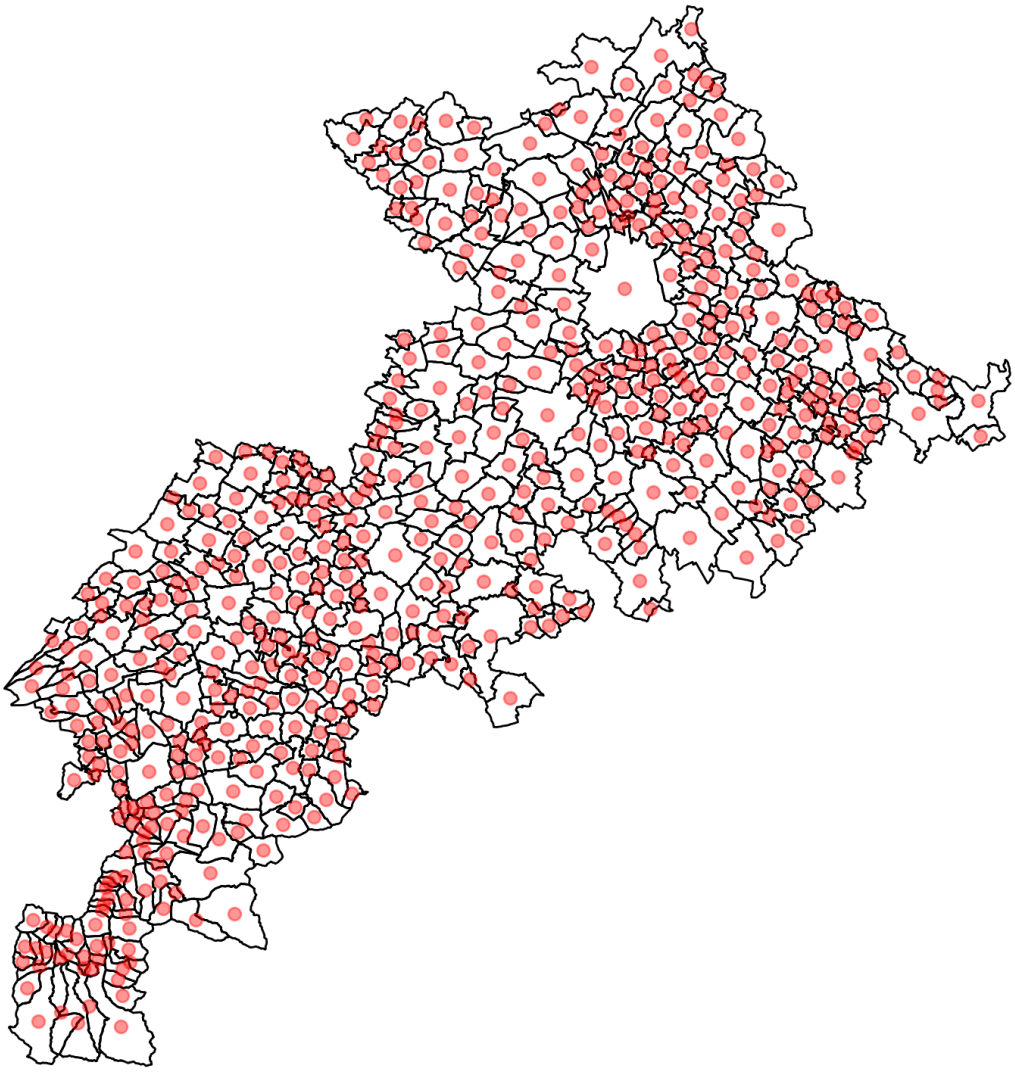
We have already seen the `to_crs` method for reprojecting data vectorized without worrying. We have also mentioned the `area` method to calculate areas. There are many such methods, and the goal of this chapter is not to be exhaustive but to serve as a general introduction to delve deeper later. Among the most useful methods, we can mention `centroid`, which, as its name suggests, finds the centroid of each polygon, thus transforming surface data into point data. For example, to roughly represent the centers of villages in Haute-Garonne (31), after downloading the appropriate base map, we will do the following:

```
from cartiflette import carti_download
communes_31 = carti_download(
      values = ["31"],
      crs = 4326,
      borders="COMMUNE",
      vectorfile_format="geojson",
      filter_by="DEPARTEMENT",
      source="EXPRESS-COG-CARTO-TERRITOIRE",
      year=2022)

# on reprojete en 3857 pour le fond de carte
communes_31 = communes_31.to_crs(3857)
```

```
# on calcule le centroide
dep_31 = communes_31.copy()
communes_31['geometry'] = communes_31['geometry'].centroid

ax = communes_31.plot(figsize = (10,10), color = 'red', alpha = 0.4, zorder=2)
dep_31.to_crs(3857).plot(
  ax = ax, zorder=1,
  edgecolor = "black",
  facecolor="none", color = None
)
#ctx.add_basemap(ax, source = ctx.providers.Stamen.Toner)
ax.set_axis_off()
ax
```



Therefore, with `Geopandas`, the entire `Pandas` grammar can be used to process the attribute dimension of the data while the geographic dimension can be manipulated with appropriate methods.

6 Enrichments through the spatial dimension: spatial joins

6.1 Principle

The previous map already illustrates the power of cartographic representation. With just a few lines of code and minimal operations on our data, we already gain a better understanding of the phenomenon we wish to observe. Specifically, we clearly detect a center-periphery structure in our data, which is not surprising but reassuring to find at first glance.

We also notice that the most used stations, outside of the hypercenter of Paris, are generally located on major roads or near parks. Again, nothing surprising, but it is reassuring to see this in our data.

Now, we can explore the structure of our dataset more thoroughly. However, if we look at it, we notice that we have limited information in the raw dataset.

```
stations.head(2)
```

	capacity	stationcode	coordonnees_geo	name	geometry
0	23	9021	[48.87778639599673, 2.337429821491241]	Saint Georges - d'Aumale	POINT (651405.084 6864400.013)
1	50	23009	[48.88873214926474, 2.288157641887665]	Anatole France - Louise Michel	POINT (647802.379 6865648.556)

In the previous chapter, we presented how associating datasets through a common dimension can enhance their value. In that case, it involved matching data based on common information in both datasets.

Now we have an additional implicit information in our two datasets: the geographical dimension. Spatial join refers to the association of datasets based on the geographical dimension. There are many different types of spatial joins: finding points within a polygon, finding intersections between multiple areas, linking a point to its nearest neighbor in another source, etc.

6.2 Example: locating stations in their neighborhood

In this exercise, we will assume that:

- The locations of the `velib` stations are stored in a dataframe named `stations`.
- The administrative data is in a dataframe named `petite_couronne`.

Exercise 4: Associate Stations with Their Corresponding Communes and Arrondissements

1. Perform a spatial join to enrich the station data by adding information from `petite_couronne`. Call this object `stations_info`.
2. Count the number of stations and the median size of the stations by arrondissement.
3. Create the objects `stations_19e` and `arrondissement_19e` to store, respectively, the stations belonging to the 19th arrondissement and the boundaries of the arrondissement.
4. Count the number of velib stations and the number of velib spots by arrondissement or commune. Represent each of these pieces of information on a map.
5. Represent the map of stations in the 19th arrondissement with the following code:

```
base = petite_couronne.loc[petite_couronne['INSEE_DEP']=="75"].boundary.plot(edgecolor =
"k", linewidth=0.5)
arrondissement_19e.boundary.plot(ax = base, edgecolor = "red", linewidth=0.9)
stations_19.plot(ax = base, color = "red", alpha = 0.4)
base.set_axis_off()
base.set_title("Les stations Vélib du 19e arrondissement")
base
```

Following the previous examples, only represent the 19th arrondissement and add an open-streetmap background to better locate the stations.

6. Represent the same information but in density (divide by the surface area of the arrondissement or commune in km²).

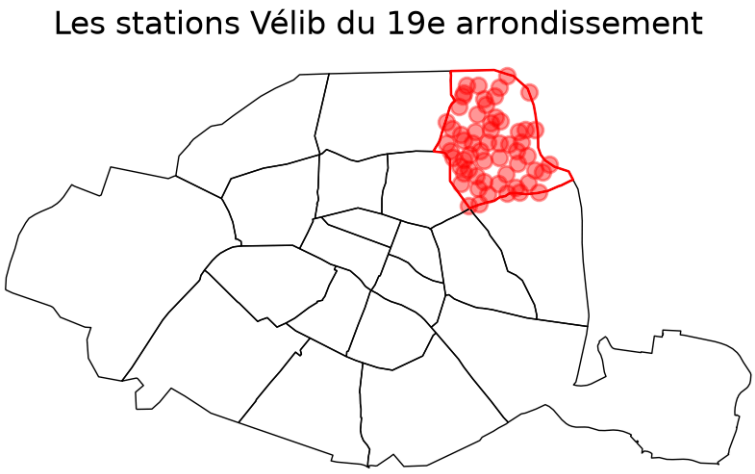
After the spatial join, the dataset presents the following structure:

	capacity	stationcode	coordonnees_geo	name	geometry	index_right	INSEE_DEP	INSEE_REG	ID	NOM	...	AAV2020	TAAV2017	TDAAV2017	CATEAAV2020	BV2012	LIBELLE_DEPARTEMENT	LIBELLE_REGION	PAYS	SOURCE	AREA
0	23	9021	[48.877786395979673, 2.3374298249149241]	Saint Georges - d'Aumale	POINT (651405.084 6864400.013)	5	75	11	ARR_MUNI0000000097360043	Paris 9e Arrondissement	...	001	5	50	11	75056	Paris	Île-de-France	France	IGN:EXPRESS-COG-CARTO-TERRITOIRE	NaN
1	50	25009	[48.88873214926474, 2.2881576418876652]	Amatole France - Louise Michel	POINT (647892.379 6865648.556)	20	92	11	COMMUNE_0000000097356057	Levallois-Perret	...	001	5	50	12	75056	Hauts-de-Seine	Île-de-France	France	IGN:EXPRESS-COG-CARTO-TERRITOIRE	metropole

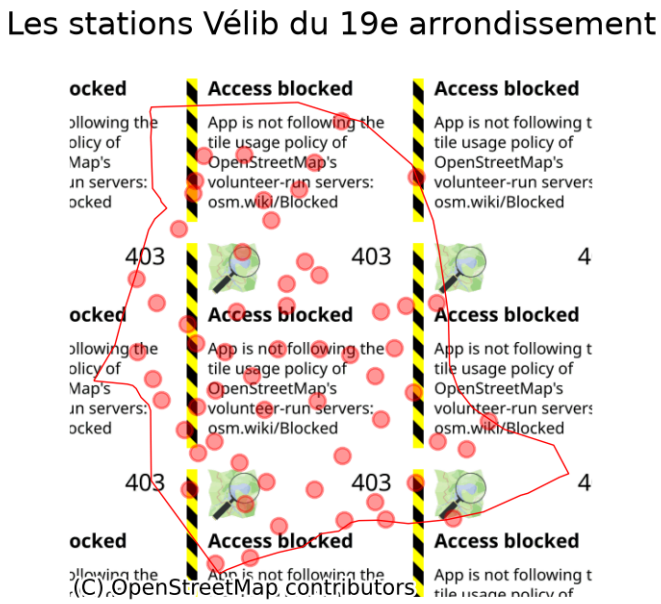
We can therefore calculate statistics by district, just as we would with a `Pandas DataFrame` (question 2):

	NOM	capacity	
		count	median
30	L'Île-Saint-Denis	1	25.0
7	Bois-Colombes	2	30.0
29	Joinville-le-Pont	2	40.0
34	Le Pré-Saint-Gervais	2	21.0
45	Noisy-le-Sec	2	29.0
...	...	...	...
55	Paris 17e Arrondissement	63	34.0
50	Paris 12e Arrondissement	68	42.0
51	Paris 13e Arrondissement	70	34.5
59	Paris 20e Arrondissement	70	26.0
53	Paris 15e Arrondissement	89	36.0

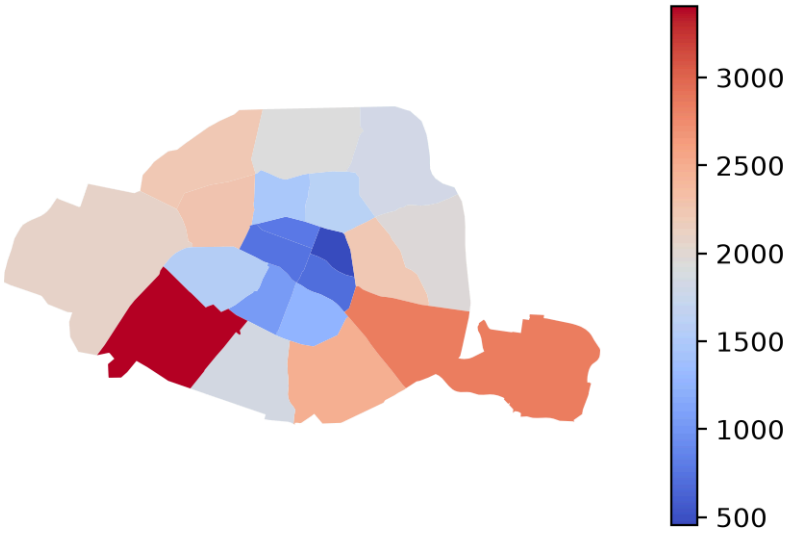
However, maps will probably be more illustrative. To begin with question 3, we can represent the stations of the 19th arrondissement, first within the whole of Paris.

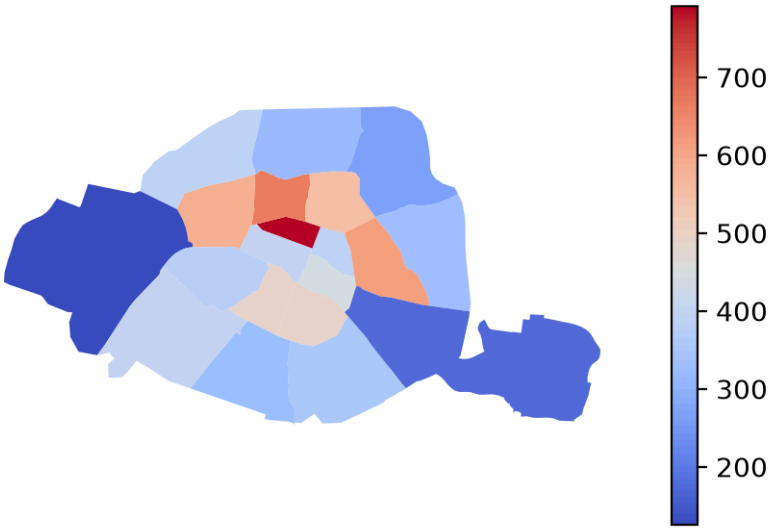


We can then zoom in on this arrondissement and create a map with a more detailed background:



Our map after question 5 looks like the following one





With this map, based on color gradients (choropleth map), the reader falls victim to a classic illusion. The most visible arrondissements on the map are the largest ones. It is logical that these areas are also better equipped with Vélib stations. Although the availability of Vélib stations is likely more related to population density and infrastructure, the size effect is likely the most visible phenomenon on our map, even though it might not be the primary factor in reality.

If we instead represent the capacity in terms of density, to account for the varying sizes of the arrondissements, the conclusions are reversed and better align with the expectations of a center-periphery model. The central arrondissements are better equipped. If we made a map with proportional circles rather than a choropleth map, this would be even more visible; however, cartography is not the focus of this chapter.

```
<Axes: >
```

6.3 Additional exercise

The previous exercises allowed for familiarization with the processing of spatial data. However, it is often necessary to juggle more complex geometric dimensions. This can include, for example, changing territorial scales in the data or introducing merges/dissolutions of geometries.

We will illustrate this with an additional exercise that demonstrates how to work with data in urban economic models where we assume movement to the nearest point (Hotelling's model).

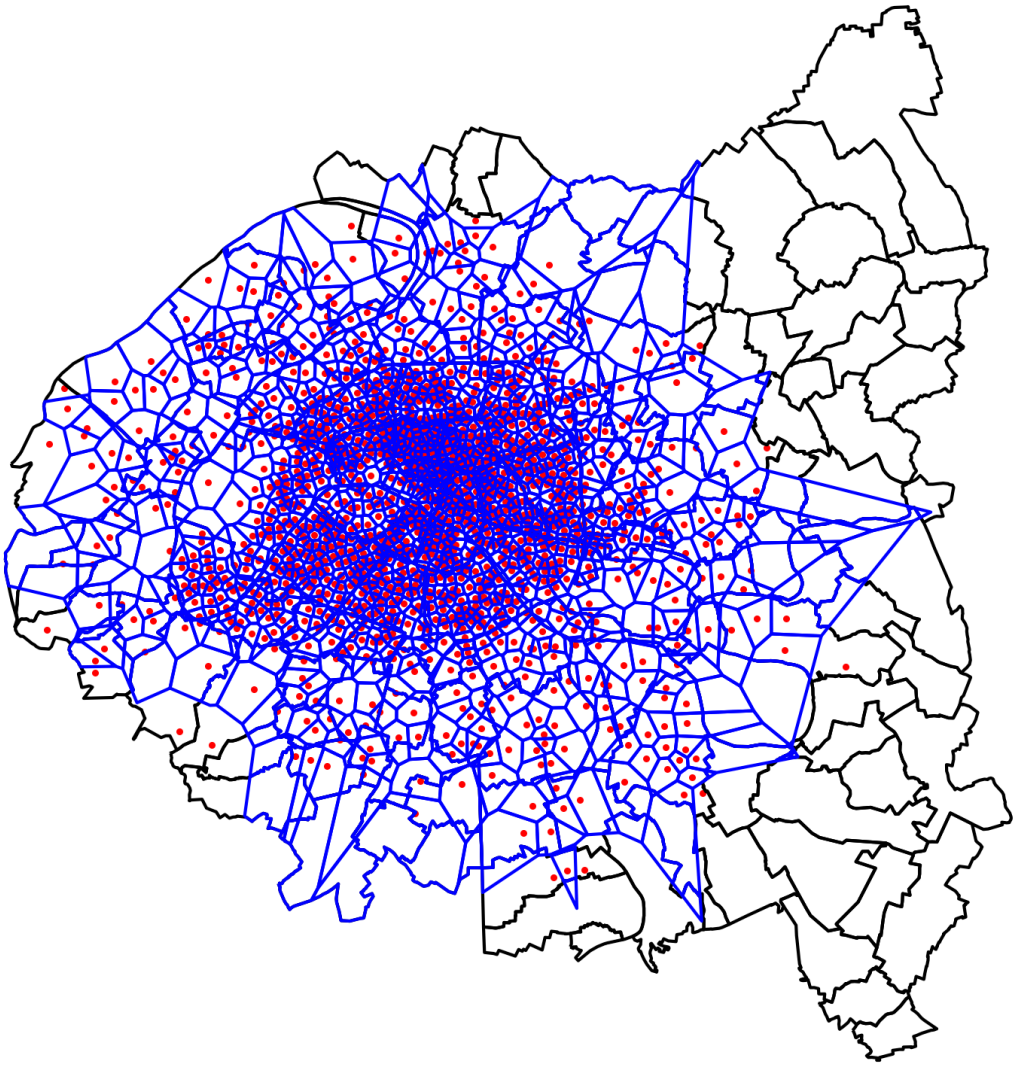
Let's imagine that each Vélib user moves exclusively to the nearest station (assuming there is never a shortage or overcapacity). What is the map of Vélib coverage? To answer this type of question, we frequently use the Voronoi tessellation, a classic operation for transforming points into polygons.

The following exercise allows for familiarization with the construction of Voronoi[3].

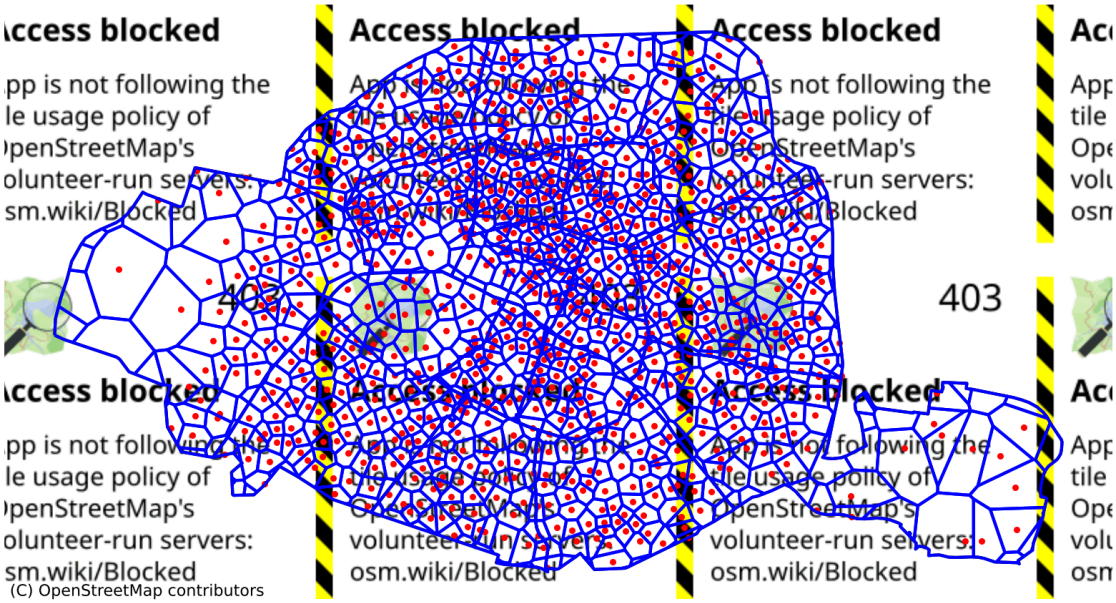
> Exercise 5 (optional): Coverage map of the stations
>
> This exercise is more complex because it involves going back to `Shapely`, a lower-level library than `GeoPandas`.
>
> This exercise is left open-ended. A possible source of inspiration is this discussion on StackExchange.
>
> The objective is to make two coverage maps: one at the level of the petite couronne and the other only within Paris intramuros.

The first coverage map, at the agglomeration level, shows the higher density of Velib stations in central Paris:

[3] In this working paper on mobile phone data, it is shown that this approach is not without bias in phenomena where the spatial proximity hypothesis is overly simplistic.



If we zoom in on Paris intramuros, we also see heterogeneity in coverage. There is less heterogeneity in coverage areas since the density is high, but we still notice divergences between certain areas.



7 References

Informations additionnelles

i Python environment

This site was built automatically through a `Github` action using the `Quarto` reproducible publishing software (version 1.10.18).

The environment used to obtain the results is reproducible via uv. The `pyproject.toml` file used to build this environment is available on the `linogaliana/python-datascientist` repository

```
[project]
name = "python-datascientist"
version = "0.1.0"
description = "Source code for Lino Galiana's Python for data science course"
readme = "README.md"
requires-python = "≥3.13,<3.14"
dependencies = [
    "altair≥6.0.0",
    "cartiflette",
    "contextily==1.6.2",
    "duckdb≥0.10.1",
    "folium≥0.19.6",
    "gdal==3.11.4",
    "graphviz==0.20.3",
    "great-tables≥0.12.0",
    "gt-extras≥0.0.8",
    "ipykernel≥6.29.5",
    "jupyter≥1.1.1",
    "jupyter-cache≥1.0.0",
    "kaleido≥0.2.1",
```

```
    "langchain-community>=0.3.27",
    "loguru==0.7.3",
    "markdown>=3.8",
    "nbclient>=0.10.0",
    "nbformat>=5.10.4",
    "nltk>=3.9.1",
    "pandas>=3.0",
    "pip>=25.1.1",
    "plotly>=6.1.2",
    "plotnine>=0.15",
    "polars>=1.8.2",
    "pyarrow>=17.0.0",
    "pynsee>=0.1.8",
    "python-dotenv>=1.0.1",
    "python-frontmatter>=1.1.0",
    "pywaffle>=1.1.1",
    "requests>=2.32.3",
    "scikit-image>=0.24.0",
    "scikit-learn>=1.8.0",
    "scipy>=1.13.0",
    "seaborn>=0.13.2",
    "selenium<4.39.0",
    "spacy>=3.8.4",
    "webdriver-manager>=4.0.2",
    "wordcloud==1.9.3",
]

[tool.uv.sources]
cartiflette = { git = "https://github.com/inseefrlab/cartiflette" }
gdal = [
  { index = "gdal-wheels", marker = "sys_platform == 'linux'" },
  { index = "geospatial_wheels", marker = "sys_platform == 'win32'" },
]

[[tool.uv.index]]
name = "geospatial_wheels"
url = "https://nathanjmcdougall.github.io/geospatial-wheels-index/"
explicit = true

[[tool.uv.index]]
name = "gdal-wheels"
url = "https://gitlab.com/api/v4/projects/61637378/packages/pypi/simple"
explicit = true

[dependency-groups]
dev = [
    "nb-clean>=4.0.1",
]
```

To use exactly the same environment (version of `Python` and *packages*), please refer to the documentation for `uv`.

Bibliography

[1] C.-houh Chen, W. Härdle, A. Unwin, and M. Friendly, “A brief history of data visualization,” *Handbook of data visualization*, pp. 15–56, 2008.