lab_03

October 11, 2015

1 Mapping in Python with geopandas

```
In [1]: %matplotlib inline

        import matplotlib.pyplot as plt
        import geopandas as gpd
        import pysal as ps
        from pysal.contrib.viz import mapping as maps
```

In this lab, we will learn how to load, manipulate and visualize spatial data. In some senses, spatial data have become so pervasive that nowadays, they are usually included simply as "one more column" in a table. However, *spatial is special* and there are few aspects in which geographic data differ from standard numerical tables. In this session, we will extend the skills developed in the previous one about non-spatial data, and combine them. In the process, we will discover that, although with some particularities, dealing with spatial data in Python largely resembles dealing with non-spatial data. For example, in this lab you will learn to make slick maps like this one with just a few commands:

Or interactive ones like this one where the smallest areas of Liverpool are highlighted in red:

```
In [2]: from IPython.display import IFrame
        IFrame("figs/lab03_liverpool_smallest.html", width=560, height=315)
```

```
Out[2]: <IPython.lib.display.IFrame at 0x7fbd68e45d90>
```

To learn these concepts, we will be playing again with the geography of Liverpool. In particular we will use Census geographies (Available as part of the Census Data pack used before, see link) and Ordnance Survey geospatial data, available to download also from the CDRC data store (link). To make the rest of the notebook easier to follow, let us set the paths to the main two folders here. We will call the path to the Liverpool Census pack `lcp_dir`, and that to the OS geodata `los_dir`:

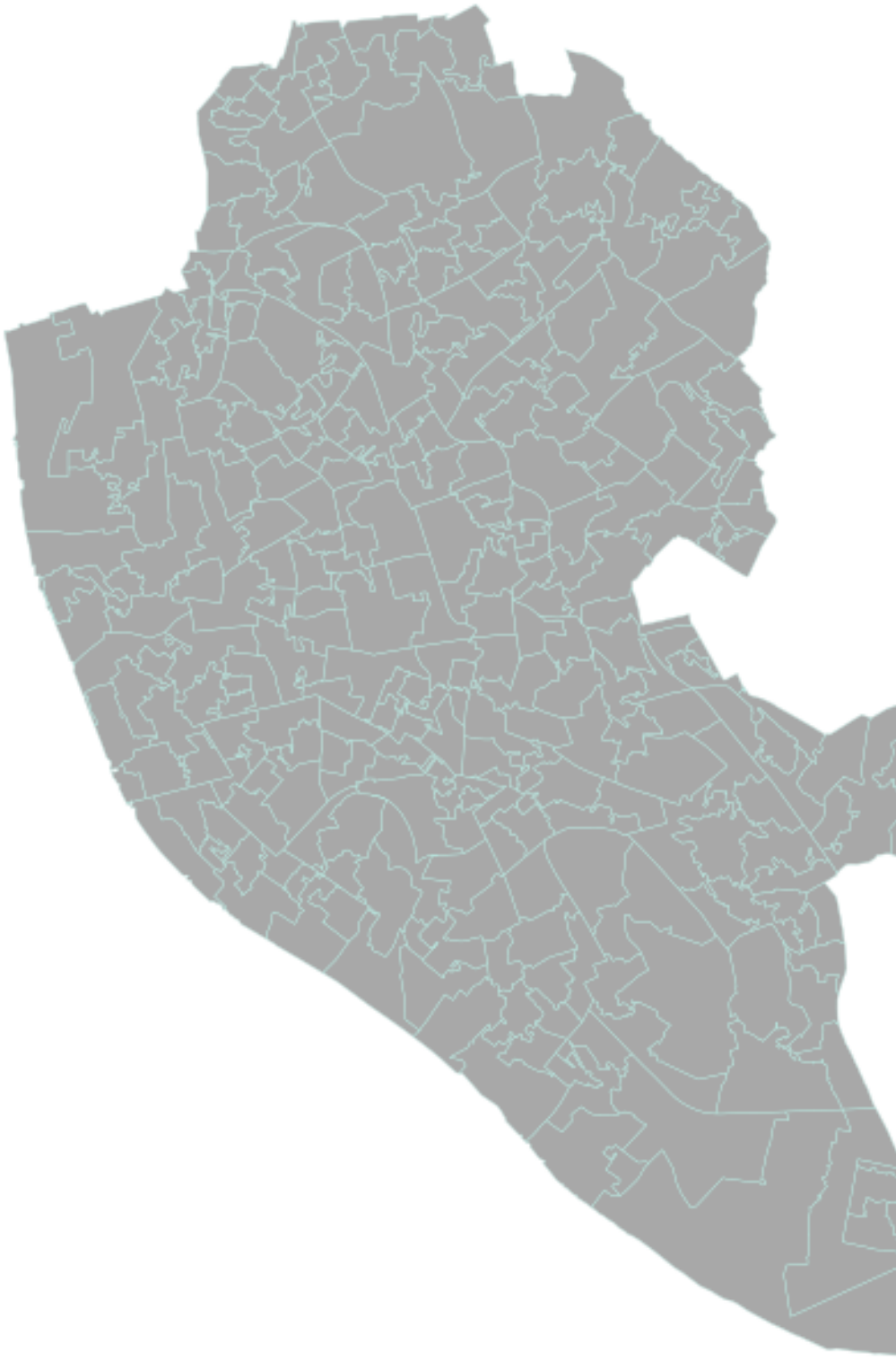
```
In [3]: lcp_dir = '../../../../data/Liverpool/'
        los_dir = '../../../../data/E08000012/'
```

1.1 Loading up spatial data

The easiest way to get from a file to a quick visualization of the data is by loading it as a `GeoDataFrame` and calling the `plot` command. The main library employed for all of this is `geopandas` which is a geospatial extension of the `pandas` library, already introduced before. `geopandas` supports exactly the same functionality that `pandas` does (in fact since it is built on top of it, so most of the underlying machinery is pure `pandas`), plus a wide range of spatial counterparts that make manipulation and general "munging" of spatial data as easy as non-spatial tables.

In two lines of code, we will obtain a graphical representation of the spatial data contained in a file that can be in many formats; actually, since it uses the same drivers under the hood, you can load pretty much the same kind of vector files that QGIS permits. Let us start by plotting single layers in a crude but quick form, and we will build style and sophistication into our plots later on.

LSOAs in Liverpool



- Polygons

Let us begin with the most common type of spatial data in the social science: polygons. For example, we can load the geography of LSOAs in Liverpool with the following lines of code:

```
In [4]: lsoas_link = lcp_dir + 'shapefiles/Liverpool_lsoa11.shp'
        lsoas = gpd.read_file(lsoas_link)
```

Now `lsoas` is a `GeoDataFrame`. Very similar to a traditional, non-spatial `DataFrame`, but with an additional column called `geometry`:

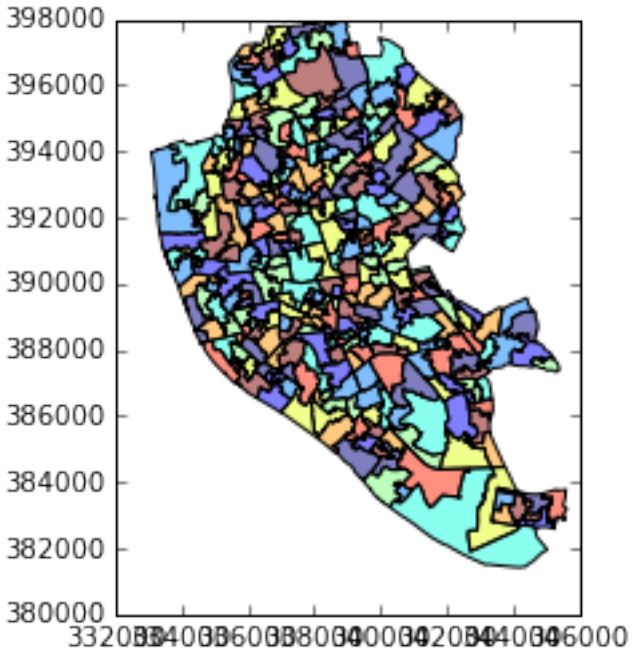
```
In [5]: lsoas.head()
```

```
Out[5]:     LSOA11CD                                           geometry
        0  E01006512  POLYGON ((336103.358 389628.58, 336103.416 389...
        1  E01006513  POLYGON ((335173.781 389691.538, 335169.798 38...
        2  E01006514  POLYGON ((335495.676 389697.267, 335495.444 38...
        3  E01006515  POLYGON ((334953.001 389029, 334951 389035, 33...
        4  E01006518  POLYGON ((335354.015 388601.947, 335354 388602...
```

This allows us to quickly produce a plot by executing the following line:

```
In [6]: lsoas.plot()
```

```
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd68dd13d0>
```

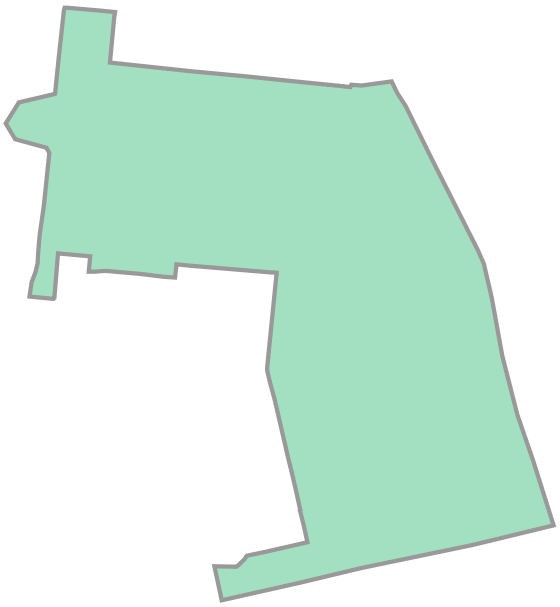


This might not be the most aesthetically pleasant visual representation of the LSOAs geography, but it is hard to argue it is not quick to produce. We will work on styling and customizing spatial plots later on.

Pro-tip: if you call a single row of the `geometry` column, it'll return a small plot ith the shape:

```
In [7]: lsoas.loc[0, 'geometry']
```

Out[7]:



- Lines

Displaying lines is as straight-forward as polygons. To load railway tunnels in Liverpool:

```
In [8]: rwy_tun = gpd.read_file(los_dir + 'RailwayTunnel.shp')\
                    .set_index('id')
        rwy_tun.info()
```

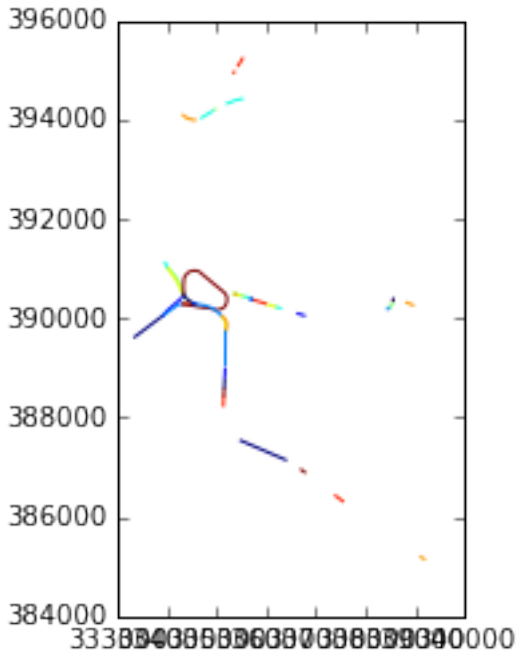
```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 44 entries, 0ACD196C321E4F8DE050A00A568A6F6F to 0ACD196C313D4F8DE050A00A568A6F6F
Data columns (total 2 columns):
featcode    44 non-null float64
geometry    44 non-null object
dtypes: float64(1), object(1)
memory usage: 1.0+ KB
```

Note how we have also indexed the table on the `id` column.

A quick plot is similarly generated by (mind that because there are over 18,000 segments, this may take a little bit):

```
In [9]: rwy_tun.plot()
```

```
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd649e3950>
```



Again, this is not the prettiest way to display the roads maybe, and you might want to change a few parameters such as colors, etc. All of this is possible, as we will see below, but this gives us an easy check of what lines look like.

[In-class exercise]
Obtain the graphical representation of the line with `id` = `0ACD196C32214F8DE050A00A568A6F6F`.

- Points

Finally, points follow a similar structure. If we want to represent named places in Liverpool:

```
In [10]: namp = gpd.read_file(los_dir + 'NamedPlace.shp')
         namp.head()
```

Out[10]:

	classifica	distname	featcode	fontheight	\
0	Hydrography	Sugar Brook	15804	Small	
1	Landcover	Sandfield Park	15805	Small	
2	Populated Place	Sandfield Park	15801	Medium	
3	Populated Place	Gillmoss	15801	Medium	
4	Populated Place	Croxteth	15801	Medium	

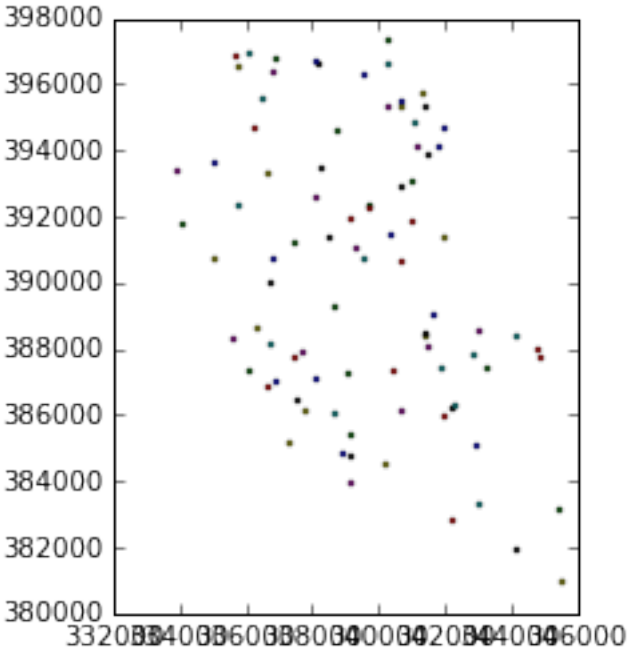
	geometry	htmlname	id	\
0	POINT (339605 396261)	Sugar Brook	0EE7A103C03A8FBFE050A00A568A2502	
1	POINT (339758 392357)	Sandfield Park	0EE7A104A4B68FBFE050A00A568A2502	
2	POINT (339768 392217)	Sandfield Park	0EE7A1041DB18FBFE050A00A568A2502	
3	POINT (340269 396567)	Gillmoss	0EE7A1041DE48FBFE050A00A568A2502	
4	POINT (340296 395304)	Croxteth	0EE7A1041DE58FBFE050A00A568A2502	

```
   orientatio
0          25
1           0
2           0
3           0
4           0
```

And the plot is produced by running:

```
In [11]: namp.plot()
```

```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd5efefb90>
```



1.2 Styling plots

It is possible to tweak several aspects of a plot to customize if to particular needs. In this section, we will explore some of the basic elements that will allow us to obtain more compelling maps.

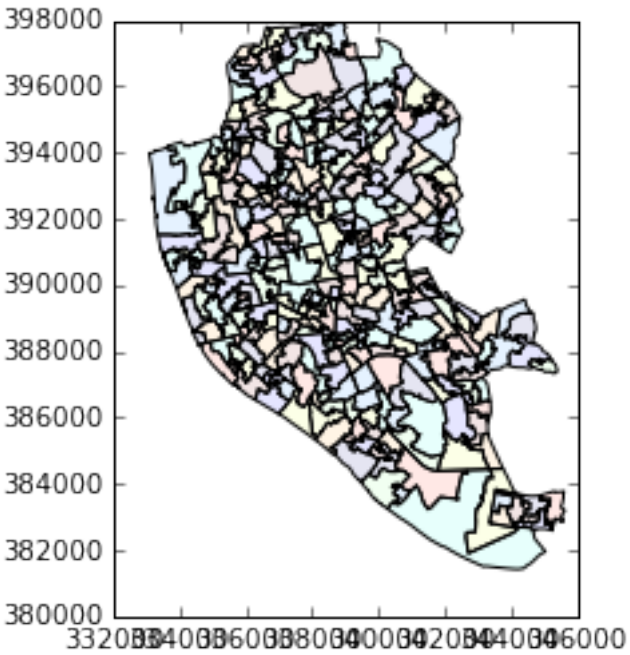
NOTE: some of these variations are very straightforward while others are more intricate and require tinkering with the internal parts of a plot. They are not necessarily organized by increasing level of complexity.

- Changing transparency

The intensity of color of a polygon can be easily changed through the `alpha` attribute in plot. This is specified as a value betwee zero and one, where the former is entirely transparent while the latter is the fully opaque (maximum intensity):

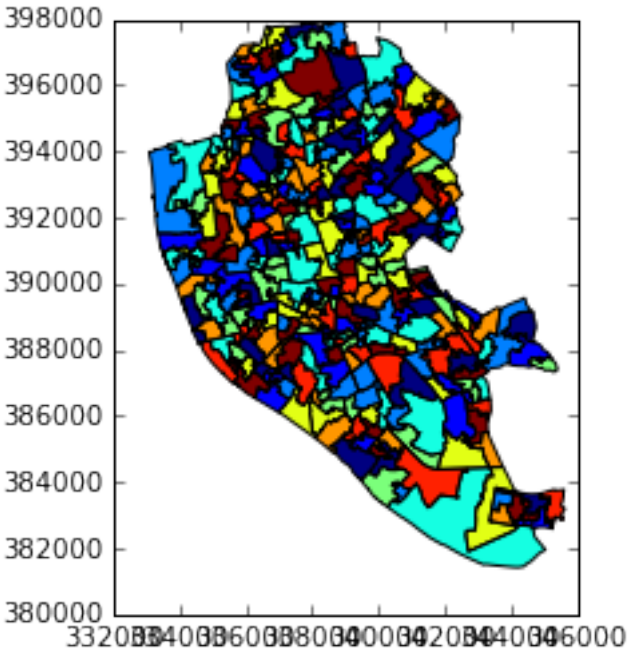
```
In [12]: lsoas.plot(alpha=0.1)
```

```
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd5eda5610>
```



In [13]: lsoas.plot(alpha=1)

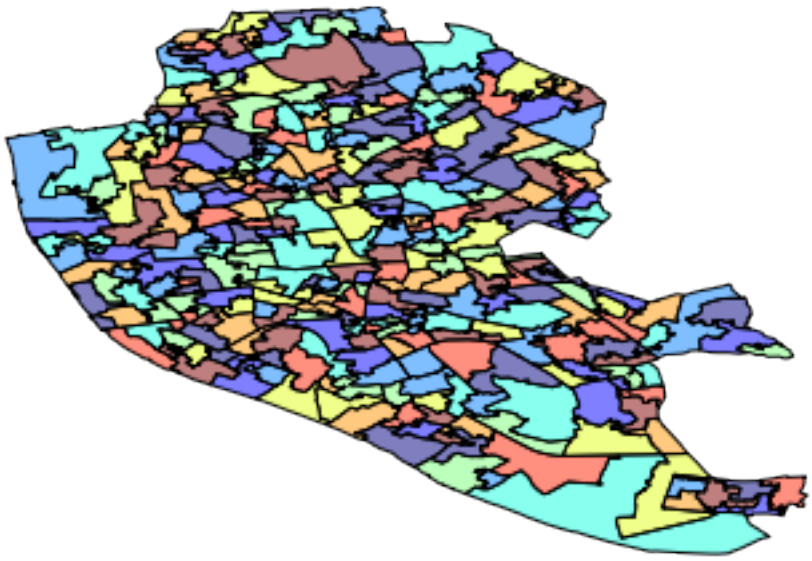
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd5e223a90>



- Removing axes

Although in some cases, the axes can be useful to obtain context, most of the times maps look and feel better without them. Removing the axes involves wrapping the plot into a figure, which takes a few more lines of aparently useless code but that, in time, it will allow you to tweak the map further and to create much more flexible designs:

```
In [14]: f, ax = plt.subplots(1)
         ax = lsoas.plot(axes=ax)
         ax.set_axis_off()
         plt.show()
```



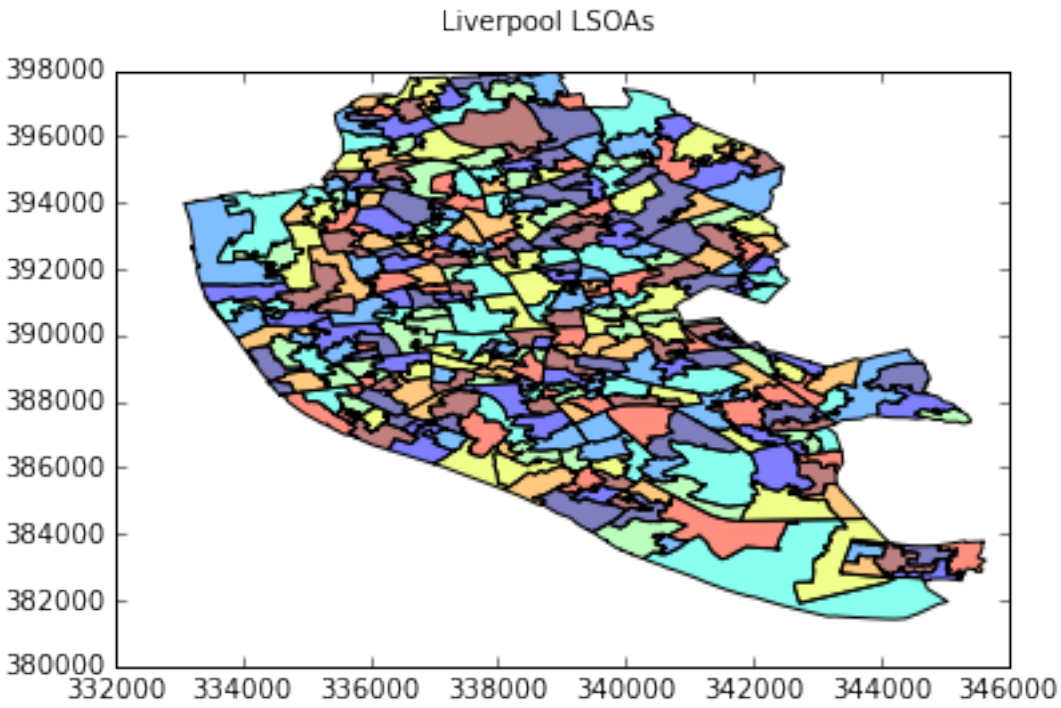
Let us stop for a second a study each of the previous lines:

1. We have first created a figure named `f` with one axis named `ax` by using the command `plt.subplots` (part of the library `matplotlib`, which we have imported at the top of the notebook). Note how the method is returning two elements and we can assign each of them to objects with different name (`f` and `ax`) by simply listing them at the front of the line, separated by commas.
2. Second, we plot the geographies as before, but this time we tell the function that we want it to draw the polygons on the axis we are passing, `ax`. This method returns the axis with the geographies in them, so we make sure to store it on an object with the same name, `ax`.
3. On the third line, we effectively remove the box with coordinates.
4. Finally, we draw the entire plot by calling `plt.show()`.

- Adding a title

Adding a title is a simple extra line, if we are creating the plot within a figure, as we just did. To include text on top of the figure:

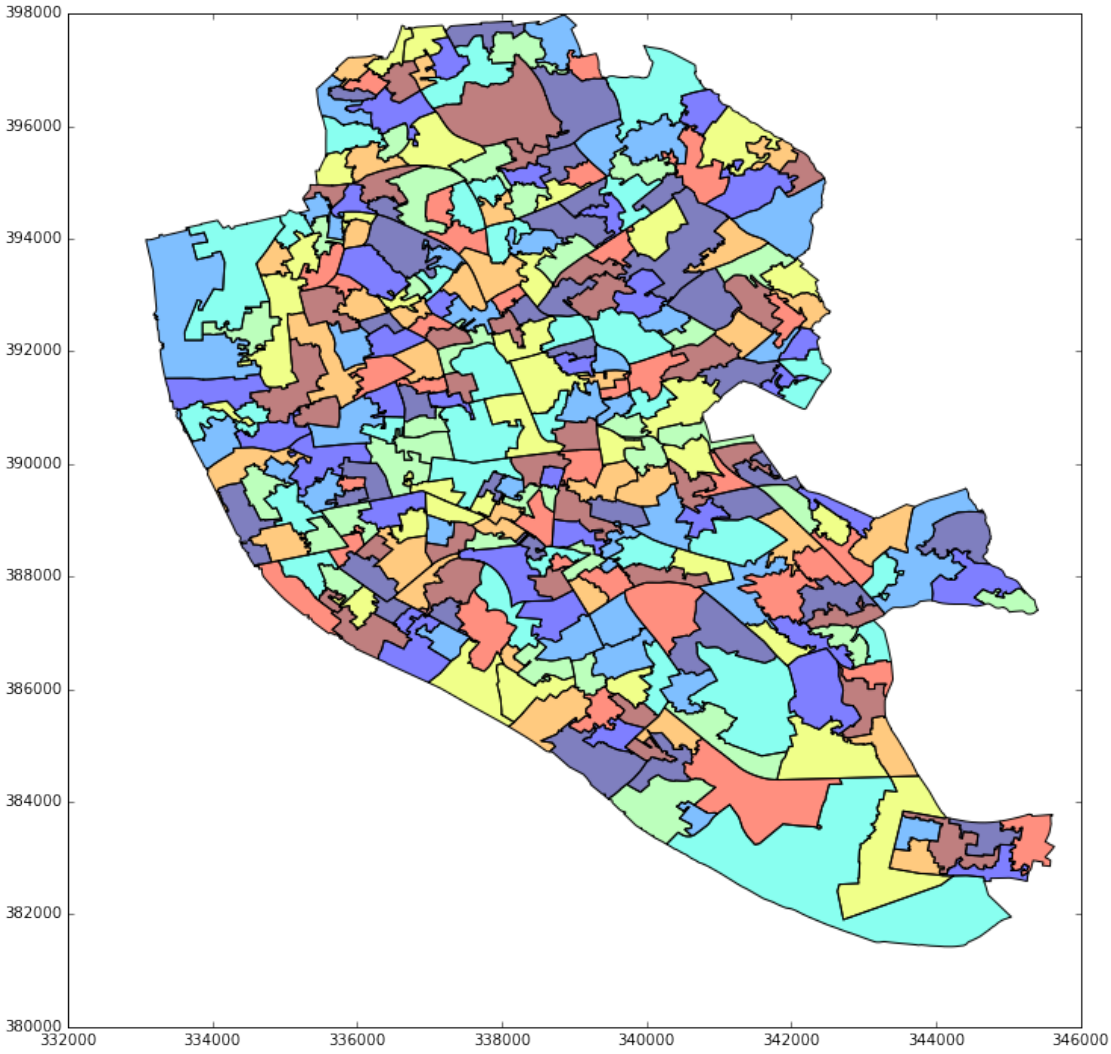
```
In [15]: f, ax = plt.subplots(1)
         ax = lsoas.plot(axes=ax)
         f.suptitle('Liverpool LSOAs')
         plt.show()
```



- Changing the size of the map

The size of the plot is changed equally easily in this context. The only difference is that it is specified when we create the figure with the argument `figsize`. The first number represents the width, the X axis, and the second corresponds with the height, the Y axis.

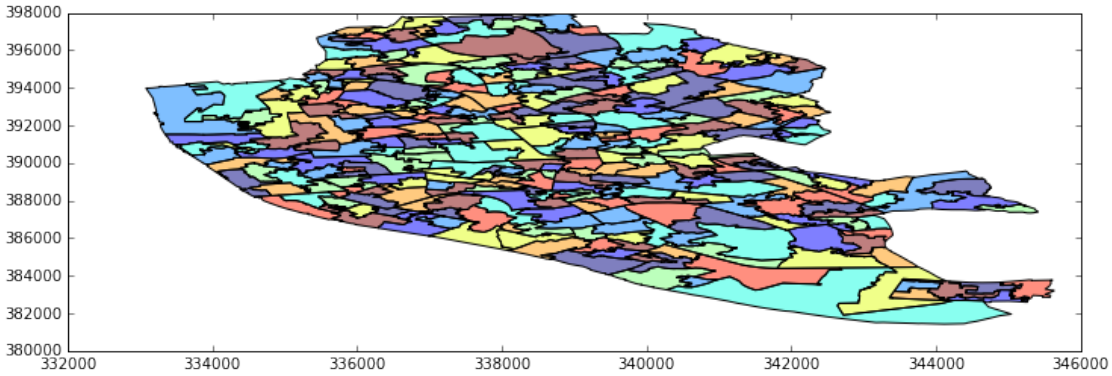
```
In [16]: f, ax = plt.subplots(1, figsize=(12, 12))
         ax = lsoas.plot(axes=ax)
         plt.show()
```



- Scaling plots

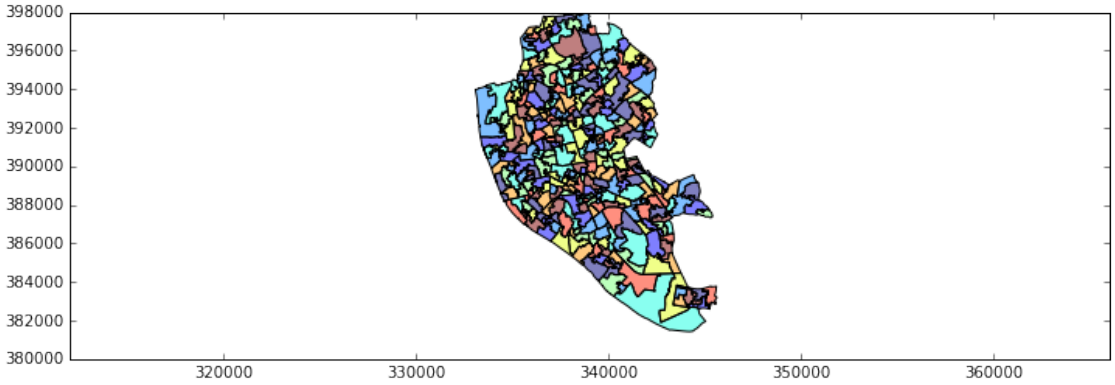
You will notice that the ability to change the size of the figure is very powerful as it makes possible to obtain many different sizes and shapes for plots. However, this also may introduce some distortions in the way the shapes are represented. For example, a very wide figure can make the viewer think that polygons are in reality more "stretched out" than they are in reality:

```
In [17]: f, ax = plt.subplots(1, figsize=(12, 4))
         ax = lsoas.plot(axes=ax)
         plt.show()
```



Although in some contexts this may be desirable (or at least, accepted), in many it will not. From a cartographic point of view, maps need to be as good representatios of reality as they can. We can ensure the scaling ratio between both axes remains fixed, whichever the shape of the figure. To do this, we only need to add a single extra line of code: `plt.axis('equal')`.

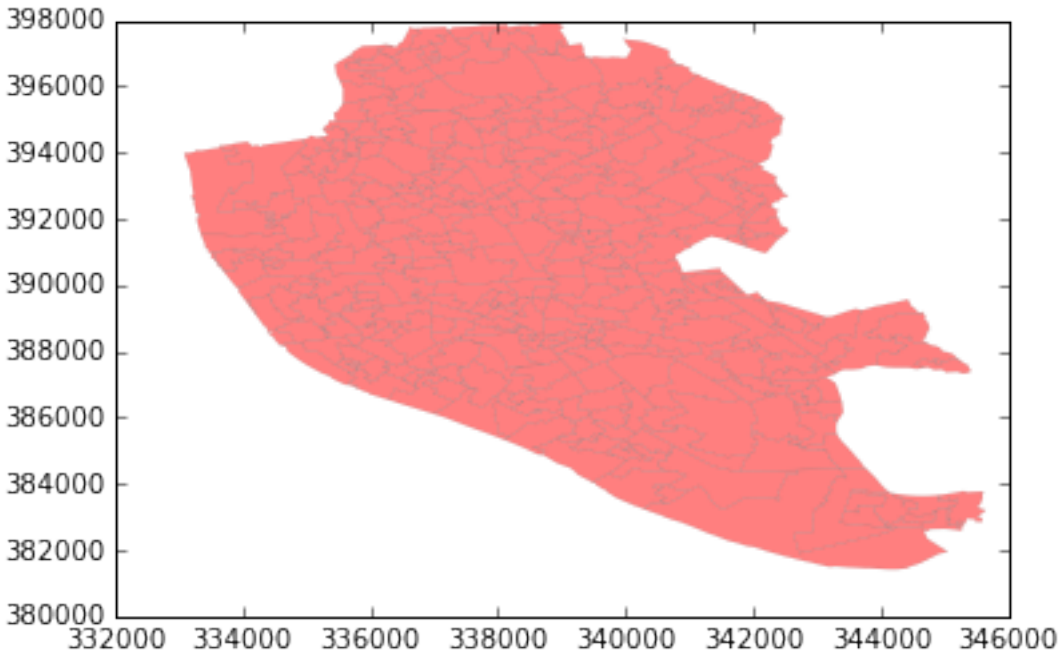
```
In [18]: f, ax = plt.subplots(1, figsize=(12, 4))
         ax = lsoas.plot(axes=ax)
         lims = plt.axis('equal')
         plt.show()
```



- Modifying borders

Border lines sometimes can distort or impede proper interpretation of a map. In those cases, it is useful to know how they can be modified. Although not too complicated, the way to access borders in `geopandas` is not as straightforward as it is the case for other aspects of the map, such as size or frame. Let us first see the code to make the *lines thinner* and *grey*, and then we will work our way through the different steps:

```
In [19]: f, ax = plt.subplots(1)
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, linewidth=0.1, edgecolor='grey')
         plt.show()
```



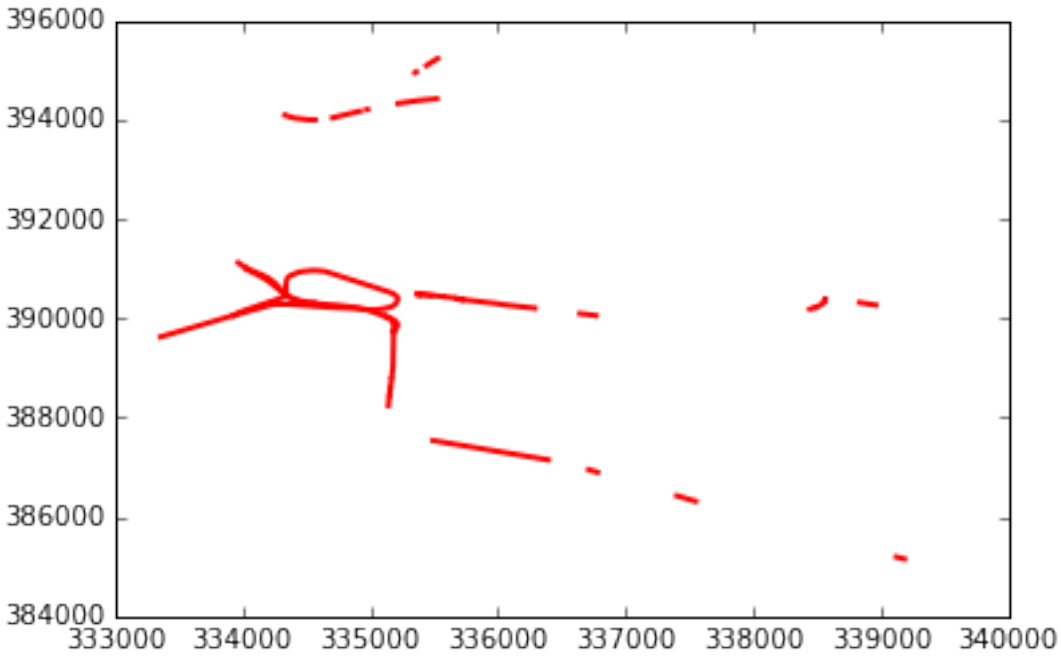
Note how the lines are much thinner and discreet. In addition, because of the way we plot the data, all the polygons are colored in the same (default) color, light red.

Let us examine line by line what we are doing in the code snippet:

- We begin by creating the figure (`f`) object and one axis inside it (`ax`) where we will plot the map.
- Then, instead of calling the utility function `plot` on the `DataFrame`, we loop over every element of the column `geometry`. As we know from above, this dataset contains polygons representing areas of Liverpool, represented geometrically in polygons. For convenience, we call them in the loop `poly`.
- Line 3 is the crucial part in the cell. In there, we plot *each* of the polygons individually, using the function `plot_multipolygon`, which in part of the `plotting` submodule in `geopandas` (which, recall, we have renamed `gpd` for convenience). The way this function works is you need to always pass the axis (`ax` in this case) where the geometry will be drawn, then the actual geometry (`poly` here), and then you can optionally pass some arguments that tweak the default appearance. Accessing each polygon at once affords us further flexibility and allows us to set the width of the border lines (`linewidth`) as well as their colors (`edgecolor`). Note that, because we are plotting at the polygon level, instead of at the `DataFrame` level, each polygon is colored in the same way.
- Draw the map using `plt.show()`.

Note that we are calling the fuction `plot_multipolygon`. This is so because we know we want to plot polygons. If instead we wanted to display lines, as those we read into `rwy_tun`, we would take a similar approach, but using `plot_multilinestring`:

```
In [20]: f, ax = plt.subplots(1)
         for line in rwy_tun['geometry']:
             gpd.plotting.plot_multilinestring(ax, line, linewidth=2, color='red')
         plt.show()
```

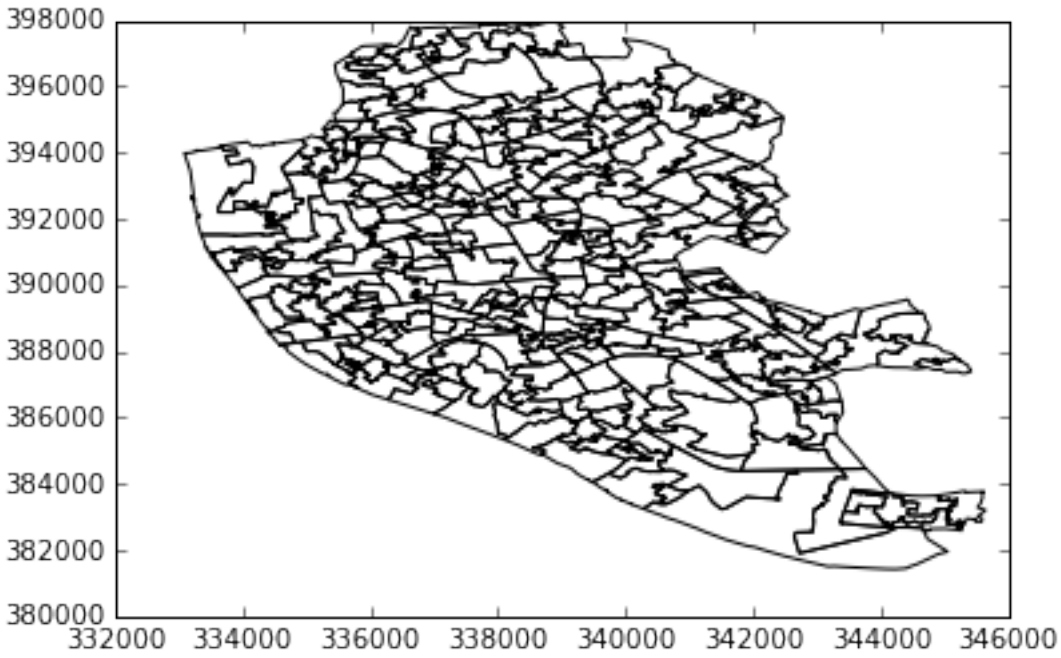


Important, note that in the case of lines the parameter to control the color is simply `color`. This is because lines do not have an area, so there is no need to distinguish between the main area (`facecolor`) and the border lines (`edgecolor`).

- Choosing a single color for every polygon

If you understand the logic behind plotting elements individually, as shown above, modifying the color of the polygon itself is very straightforward, you only need to tweak the parameter `facecolor` to your preferred color. For instance, let us draw the borders without coloring the polygons, leaving them white:

```
In [21]: f, ax = plt.subplots(1)
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, facecolor='white')
         plt.show()
```



- Transforming CRS

The coordindate reference system (CRS) is the way geographers and cartographers have to represent a three-dimentional object, such as the round earth, on a two-dimensional plane, such as a piece of paper or a computer screen. If the source data contain information on the CRS of the data, it is very easy to modify this in a `GeoDataFrame`. First let us check if we have the information stored properly:

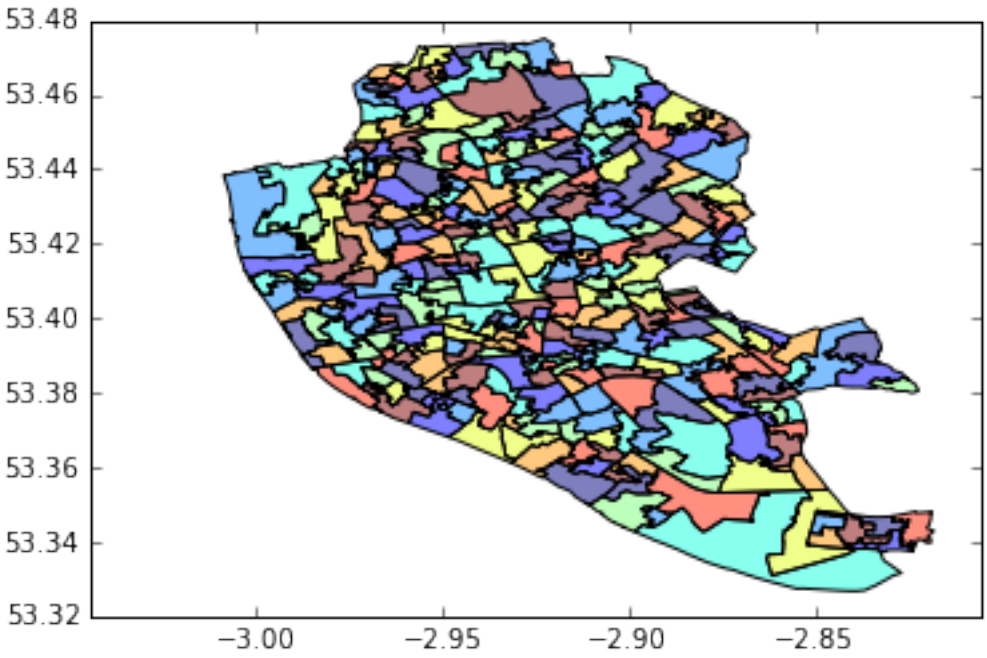
```
In [22]: lsoas.crs
```

```
Out[22]: {u'datum': u'OSGB36',
          u'k': 0.9996012717,
          u'lat_0': 49,
          u'lon_0': -2,
          u'no_defs': True,
          u'proj': u'tmerc',
          u'units': u'm',
          u'x_0': 400000,
          u'y_0': -100000}
```

As we can see, there is information stored about the reference system: it is using the datum "OSGB36", which is a projection in meters (`m` in units). There are also other less decipherable parameters but we do not need to worry about them right now.

If we want to modify this and "reproject" the polygons into a different CRS, the easiest way is to find the EPSG code online (epsg.io is a good one, although there are others too). For example, if we wanted to transform the dataset into lat/lon coordinates, we would use its EPSG code, 4326:

```
In [23]: lsoas.to_crs(epsg=4326).plot()
         lims = plt.axis('equal')
```

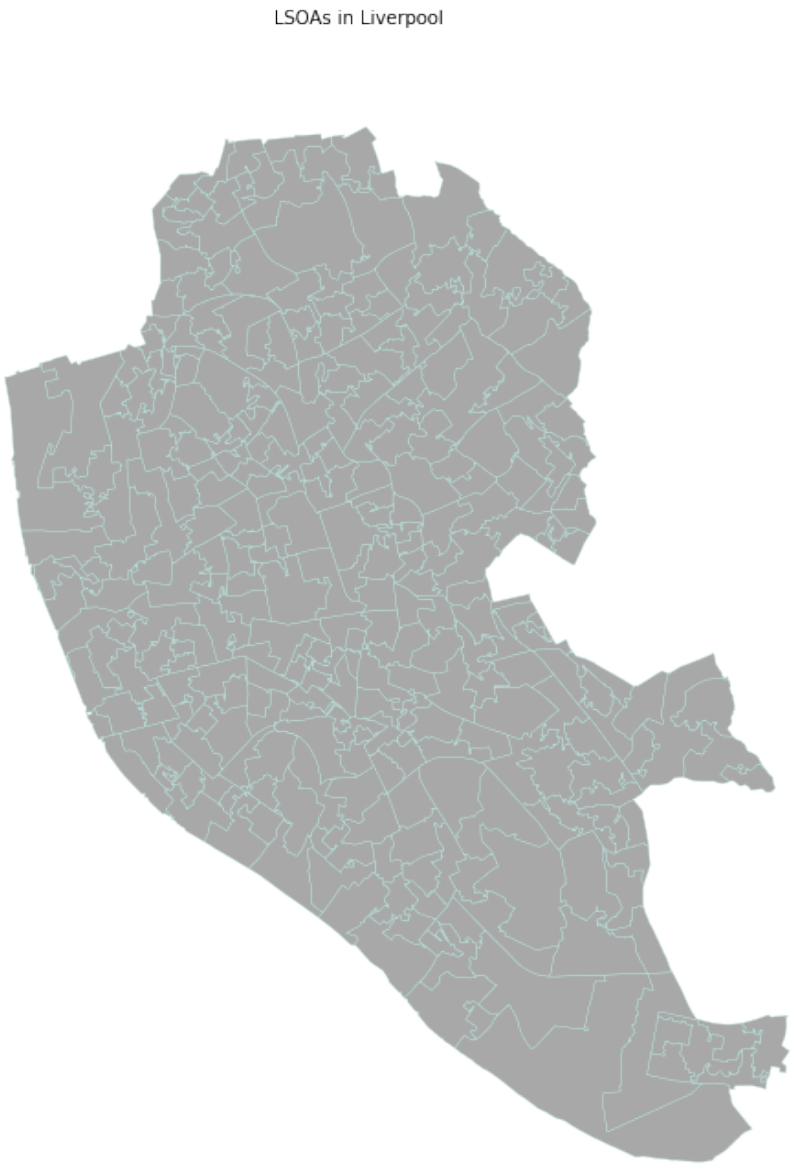


Because the area we are visualizing is not very large, the shape of the polygons is roughly the same. However, note how the *scale* in which they are plotted differs: while before we had coordinate points ranging 332,000 to 398,000, now these are expressed in degrees, and range from -3.05 to -2,80 on the longitude, and between 53.32 and 53.48 on the latitude.

[Optional exercise. Difficulty: 4/5]
Make a map of the LSOAs that features the following characteristics:

- Includes a title
- Does not include axes frame
- It is proportioned and has a figure size of 10 by 11.
- Polygons are all in the color "#525252" and fully opaque.
- Lines have a width of 0.3 and are of color "#B9EBE3"

```
In [24]: f, ax = plt.subplots(1, figsize=(10, 11))
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, facecolor='#525252', linewidth=0.3, edgecolor='#B
         ax.set_axis_off()
         f.suptitle("LSOAs in Liverpool")
         plt.axis('equal')
         plt.savefig('figs/lab03_liverpool_lsoas.png', dpi=75)
         plt.show()
```

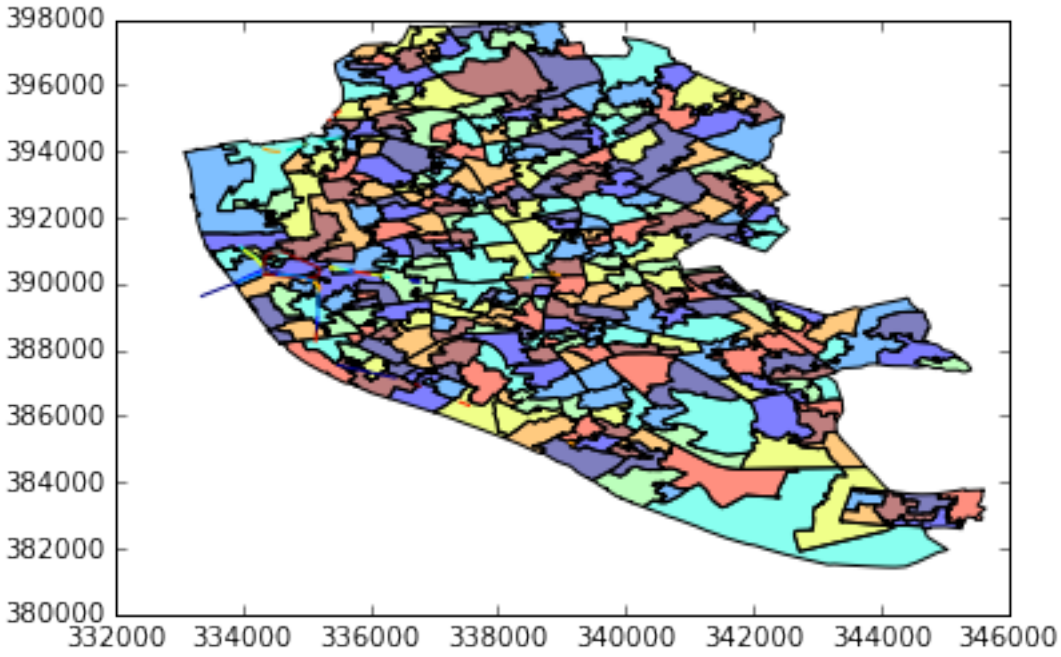


1.3 Composing multi-layer maps

So far we have considered many aspects of plotting *a single* layer of data. However, in many cases, an effective map will require more than one: for example we might want to display streets on top of the polygons of neighborhoods, and add a few points for specific locations we want to highlight. At the very heart of GIS is the possibility to combine spatial information from different sources by overlaying it on top of each other, and this is fully supported in Python.

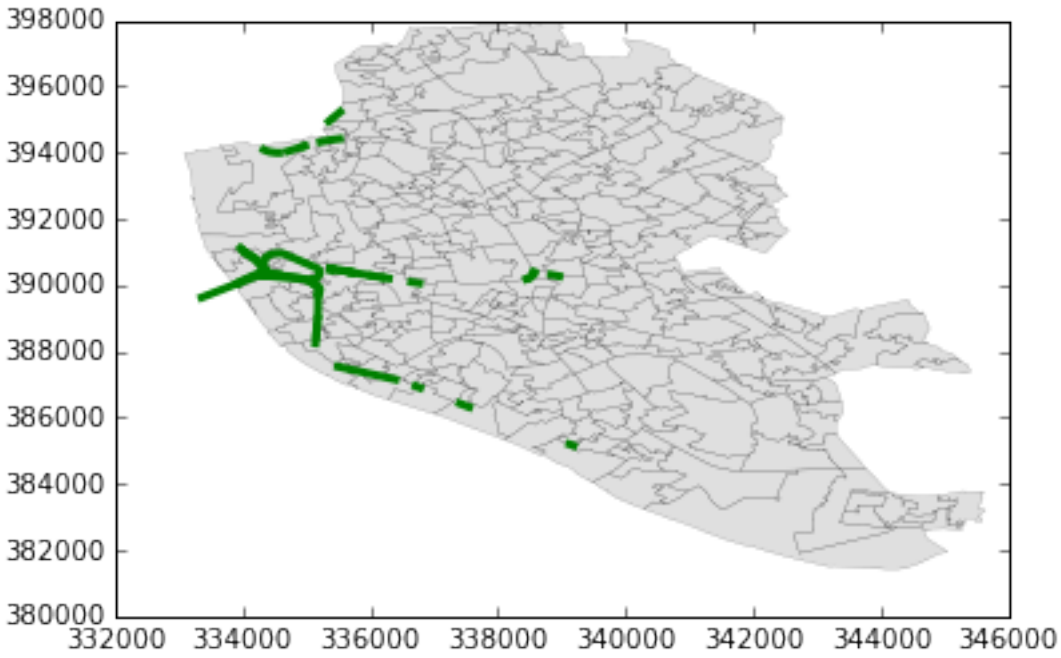
Essentially, combining different layers on a single map boils down to adding each of them to the same axis in a sequential way, as if we were literally overlaying one on top of the previous one. For example, let us get the simplest possible plot, one with the polygons from the LSOAs and the tunnels on top of them:

```
In [25]: f, ax = plt.subplots(1)
         lsoas.plot(axes=ax)
         rwy_tun.plot(axes=ax)
         plt.show()
```



Because the default colors are not really designed to mix and match several layers, it is hard to tell them apart. However, we can use all the skills and tricks learned on styling a single layer, to make a multi-layer more sophisticated and, ultimately, useful.

```
In [26]: f, ax = plt.subplots(1)
         # Plot polygons in light grey
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, facecolor='grey', alpha=0.25, linewidth=0.1)
         # Overlay railway tunnels on top in strong green
         for line in rwy_tun['geometry']:
             gpd.plotting.plot_multilinestring(ax, line, color='green', linewidth=3)
         plt.show()
```



[Optional exercise. Difficulty: 3/5]

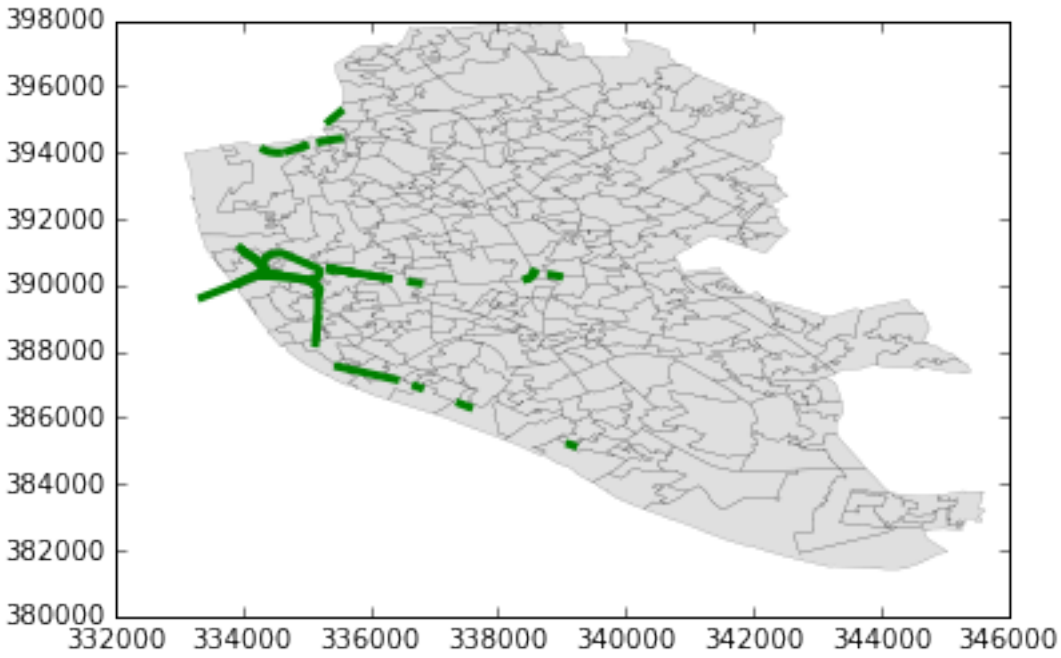
Create a similar map to the one above, but replace the railway tunnels by the named places points used at the beginning (and saved into `namp`).

Pro-tip: keep in mind you are plotting points no lines, so be sure to check the help of the function `gpd.plotting.plot_point`.

1.4 Saving maps to figures

Once we have produced a map we are content with, we might want to save it to a file so we can include it into a report, article, website, etc. Exporting maps in Python is as simple as replacing `plt.show` by `plt.savefig` at the end of the code block to specify where and how to save it. For example to save the previous map into a `png` file in the same folder where the notebook is hosted:

```
In [27]: f, ax = plt.subplots(1)
         # Plot polygons in light grey
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, facecolor='grey', alpha=0.25, linewidth=0.1)
         # Overlay railway tunnels on top in strong green
         for line in rwy_tun['geometry']:
             gpd.plotting.plot_multilinestring(ax, line, color='green', linewidth=3)
         plt.savefig('liverpool_railway_tunels.png')
```

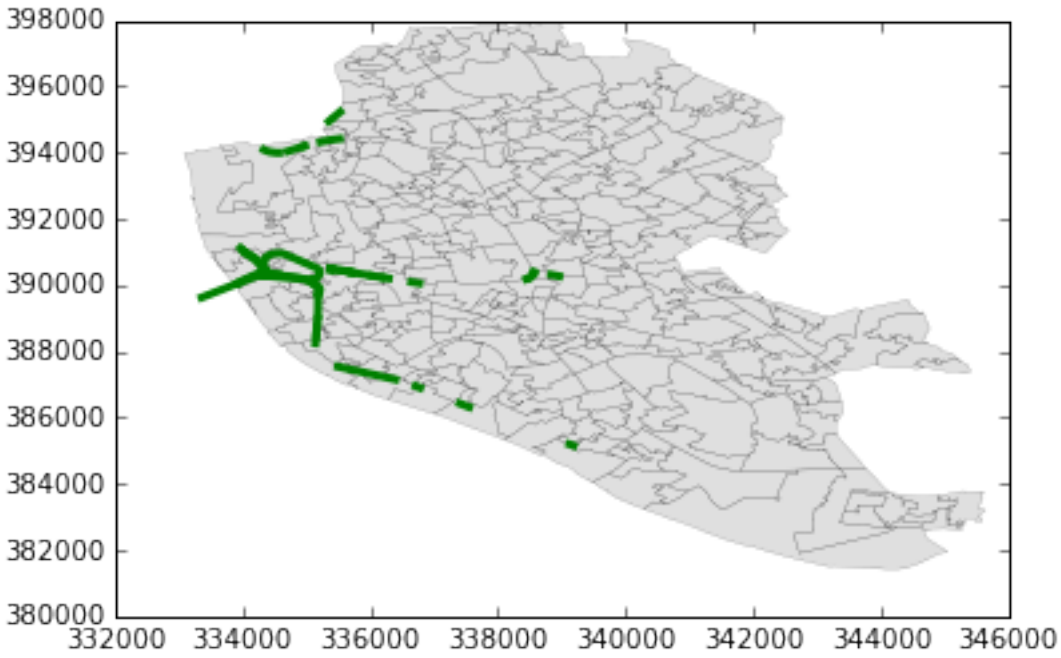


If you now check on the folder, you'll find a `png` (image) file with the map.

The command `plt.savefig` contains a large number of options and additional parameters to tweak. Given the size of the figure created is not very large, we can increase this with the argument `dpi`, which stands for "dots per inch" and it's a standard measure of resolution in images. For example, for a high definition (HD) quality image, we can use 1080:

[Note]: if this takes too long, try with 500 instead, which will still give you a good quality image that renders more easily.

```
In [28]: f, ax = plt.subplots(1)
         # Plot polygons in light grey
         for poly in lsoas['geometry']:
             gpd.plotting.plot_multipolygon(ax, poly, facecolor='grey', alpha=0.25, linewidth=0.1)
         # Overlay railway tunnels on top in strong green
         for line in rwy_tun['geometry']:
             gpd.plotting.plot_multilinestring(ax, line, color='green', linewidth=3)
         plt.savefig('liverpool_railway_tunels.png', dpi=1080)
```



1.5 Manipulating spatial tables (`GeoDataFrames`)

Once we have an understanding of how to visually display spatial information contained, let us see how it can be combined with the operations learnt in the previous session about manipulating non-spatial tabular data. Essentially, the key is to realize that a `GeoDataFrame` contains most of its spatial information in a single column named `geometry`, but the rest of it looks and behaves exactly like a non-spatial `DataFrame` (in fact, it is). This concedes them all the flexibility and convenience that we saw in manipulating, slicing, and transforming tabular data, with the bonus that spatial data is carried away in all those steps. In addition, `GeoDataFrames` also incorporate a set of explicitly spatial operations to combine and transform data. In this section, we will consider both.

Let us refresh some of the techniques we learned in the previous session about non-spatial tabular data and see how those can be combined with the mapping of their spatial counter-parts. To do this, we will revisit the population data we explored in the previous section:

```
In [29]: import pandas as pd

         # Set the path to the location of the Liverpool Census Data-Pack,
         # just as you did at the beginning of the previous session
         path = '../../../../data/Liverpool/'
```

Remember the data we want need to be extracted and renamed for the variables to have human readable names. Here we will do it all in one shot, but you can go back to the notebook of the previous session to follow the steps in more detail.

```
In [30]: # Read original table
         lsoa_orig = pd.read_csv(path+'tables/QS203EW_lsoa11.csv', index_col='GeographyCode')
         # Select necessary variables
         region_codes = ['QS203EW0002', 'QS203EW0032', 'QS203EW0045', \
                         'QS203EW0063', 'QS203EW0072']
         lsoa_orig_sub = lsoa_orig.loc[:, region_codes]
```

```
# Rename variables from codes into names
variables = pd.read_csv(path+'variables_description.csv', index_col=0)
code2name = {}
lookup_table = variables.set_index('ColumnVariableCode') # Reindex to be able to query
for code in region_codes:
    code2name[code] = lookup_table.loc[code, 'ColumnVariableDescription']
for code in code2name:
    code2name[code] = code2name[code].replace(': Total', '')
lsoa_orig_sub = lsoa_orig_sub.rename(columns=code2name)
# Calculate total populations by area
lsoa_orig_sub['Total'] = lsoa_orig_sub.sum(axis=1)
lsoa_orig_sub.head()
```

Out[30]:

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
E01006512	910	106	840	24	0	1880
E01006513	2225	61	595	53	7	2941
E01006514	1786	63	193	61	5	2108
E01006515	974	29	185	18	2	1208
E01006518	1531	69	73	19	4	1696

- Join tabular data

Now we have both tables loaded into the session: on the one hand, the spatial data are contained in `lsoas`, while all the tabular data are in `lsoa_orig_sub`. To be able to work with the two together, we need to *connect* them. In `pandas` language, this is called "join" and the key element in joins are the *indices*, the names assigned to each row of the tables. This is what we determine, for example, when we indicate `index_col` when reading a `csv`. In the case above, the index is set on `GeographyCode`. In the case of the `GeoDataFrame`, there is not any specific index, but a simple unamed sequence. The spatial table does have however a column called `LSOA11CD` which represents the code for each polygon, and this one actually matches those in `GeographyCode` in the population table.

```
In [31]: lsoas.head()
```

Out[31]:

	LSOA11CD	geometry
0	E01006512	POLYGON ((336103.358 389628.58, 336103.416 389...
1	E01006513	POLYGON ((335173.781 389691.538, 335169.798 38...
2	E01006514	POLYGON ((335495.676 389697.267, 335495.444 38...
3	E01006515	POLYGON ((334953.001 389029, 334951 389035, 33...
4	E01006518	POLYGON ((335354.015 388601.947, 335354 388602...

Having the same column, albeit named differently, in both tables thus allows us to combine, *"join"*, the two into a single one where rows are matched that we will call `geo_pop`:

```
In [32]: geo_pop = lsoas.join(lsoa_orig_sub, on='LSOA11CD')
         geo_pop.head()
```

Out[32]:

	LSOA11CD	geometry	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
0	E01006512	POLYGON ((336103.358 389628.58, 336103.416 389...	910	106	840	24	0	1880
1	E01006513	POLYGON ((335173.781 389691.538, 335169.798 38...	2225	61	595	53	7	2941
2	E01006514	POLYGON ((335495.676 389697.267, 335495.444 38...	1786	63	193	61	5	2108
3	E01006515	POLYGON ((334953.001 389029, 334951 389035, 33...	974	29	185	18	2	1208
4	E01006518	POLYGON ((335354.015 388601.947, 335354 388602...	1531	69	73	19	4	1696

Let us quickly run through the logic of joins:

- First, it is an operation in which you are "attaching" some data to a previously existing one. This does not always need to be like this but, for now, we will only consider this case. In particular, in the operation above, we are attaching the population data in `lsoa_orig_sub` to the spatial table `lsoas`.
- Second, note how the main table does not need to be indexed in the shared column for the join to be possible, it only needs to contain it. In this case, the index of `lsoas` is a simple sequence, but the relevant codes are stored in the column `LSOA11CD`.
- Third, the table that is being attached *does* need to be indexed on the relevant column. This is fine with us because `lsoa_orig_sub` *is* already indexed on the relevant ID codes.
- Finally, note how the join operation contains two arguments: one is obviously the table we want to attach; the second one, preceded by "`on`" relates to the column in the main table that is required to be matched with the index of the table being attached. In this case, the relevant ID codes are in the column `LSOA11CD`, so we specify that.

One final note, there appears to be a bug in the code that makes the joined table to loose the CRS. Reattaching it is straightforward:

```
In [33]: geo_pop.crs = lsoas.crs
```

1.5.1 Non-spatial manipulations

Once we have joined spatial and non-spatial data, we can use the techniques learned in manipulating and slicing non-spatial tables to create much richer maps. In particular, let us recall the example we worked through in the previous session in which we were selecting rows based on their population characteristics. In addition to being able to to select them, now we will also be able to visualize them in maps.

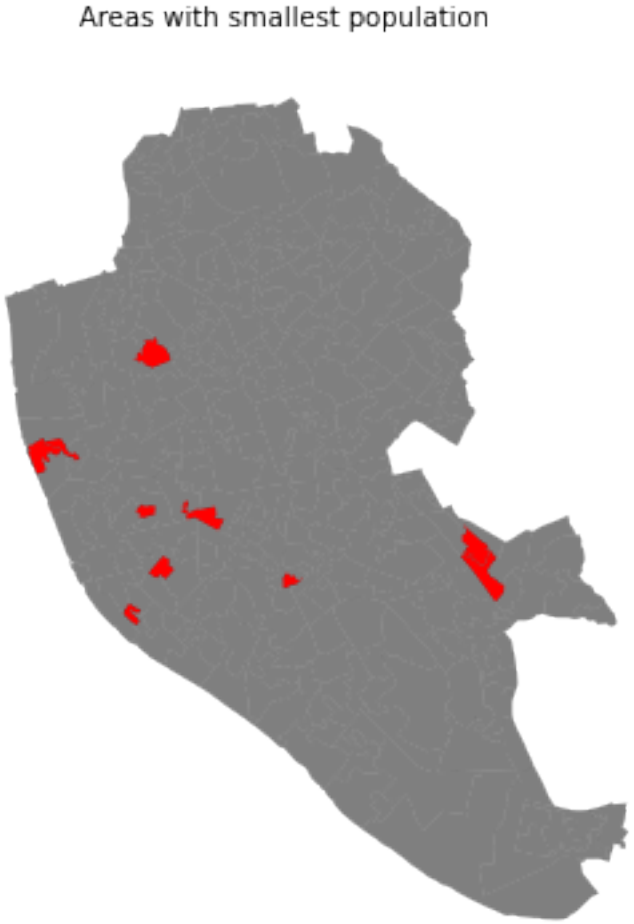
For example, let us select again the ten smallest areas of Liverpool (note how we pass a number to `head` to keep that amount of rows):

```
In [34]: smallest = geo_pop.sort_values('Total').head(10)
```

Now we can make a map of Liverpool and overlay on top of them these areas:

```
In [35]: f, ax = plt.subplots(1, figsize=(6, 6))
         # Base layer with all the areas for the background
```

```
for poly in geo_pop['geometry']:
    gpd.plotting.plot_multipolygon(ax, poly, facecolor='black', linewidth=0.025)
# Smallest areas
for poly in smallest['geometry']:
    gpd.plotting.plot_multipolygon(ax, poly, alpha=1, facecolor='red', linewidth=0)
ax.set_axis_off()
f.suptitle('Areas with smallest population')
plt.axis('equal')
plt.show()
```



[Optional exercise. Difficulty: 4/5]
Create a map of Liverpool with the two largest areas for each of the different population subgroups in a different color each.

1.5.2 Spatial manipulations

In addition to operations purely based on values of the table, as above, `GeoDataFrame`s come built-in with a whole range of traditional GIS operations. Here we will run through a small subset of them that contains some of the most commonly used ones.

- Centroid calculation

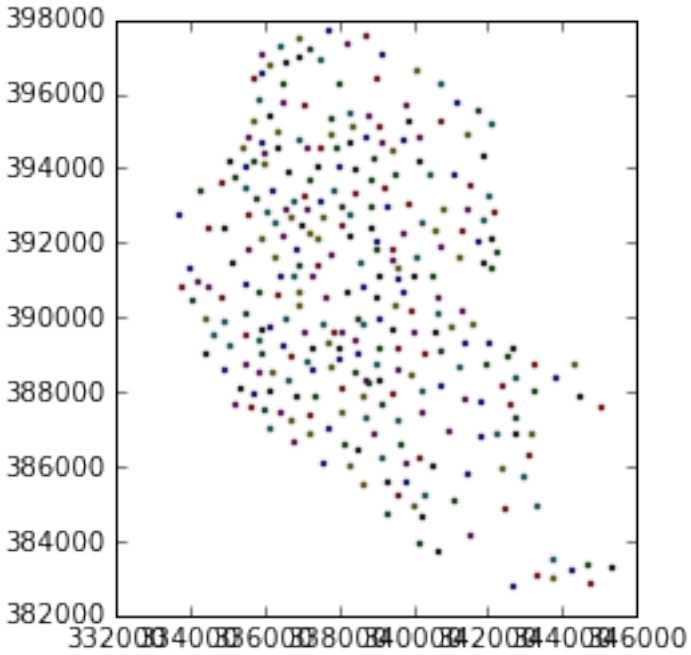
Sometimes it is useful to summarize a polygon into a single point and, for that, a good candidate is its centroid (almost like a spatial analogue of the average). The following command will return a `GeoSeries` (a single column with spatial data) with the centroids of a polygon `GeoDataFrame`:

```
In [36]: cents = geo_pop.centroid
         cents.head()

Out[36]: 0     POINT (336154.2863649924 389733.635773753)
         1    POINT (335535.9278617767 390060.8596847057)
         2    POINT (335525.0607160075 389484.4020273419)
         3    POINT (335117.4359614222 389195.6749620197)
         4     POINT (335532.691301766 388692.8415110898)
         dtype: object

In [37]: cents.plot()

Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd5cbb24d0>
```



[Optional exercise. Difficulty: 1/5]
Create a map with the polygons of Liverpool in the background and overlay on top of them their centroids.

- Point in polygon (PiP)

Knowing whether a point is inside a polygon is conceptually a simple exercise but computationally a tricky task to perform. The simplest way to perform this operation in `GeoPandas` is through the `contains` method, available for each polygon object.

```
In [38]: poly = geo_pop['geometry'][0]
         pt1 = cents[0]
         pt2 = cents[1]

In [39]: poly.contains(pt1)

Out[39]: True

In [40]: poly.contains(pt2)

Out[40]: False
```

Performing point-in-polygon in this way is instructive and useful for pedagogical reasons, but for cases with many points and polygons, it is not particularly efficient. In these situations, it is much more advisable to perform then as a "spatial join". If you are interested in these, see the link provided below to learn more about them.

[Optional exercise. Difficulty: 5/5]
Find in which polygons the named places in `namp` fall into. Return, for each named place, the LSOA code in which it is located.

- Buffers

Buffers are one of the classical GIS operations in which an area is drawn around a particular geometry, given a specific radious. These are very useful, for instance, in combination with point-in-polygon operations to calculate accessibility, catchment areas, etc.

To create a buffer using `geopandas`, simply call the `buffer` method, passing in the radious. Mind that the radious needs to be specified in the same units as the CRS of the geography you are working with. For example, for the named places, we can consider their CRS:

```
In [41]: namp.crs

Out[41]: {'init': u'epsg:27700'}
```

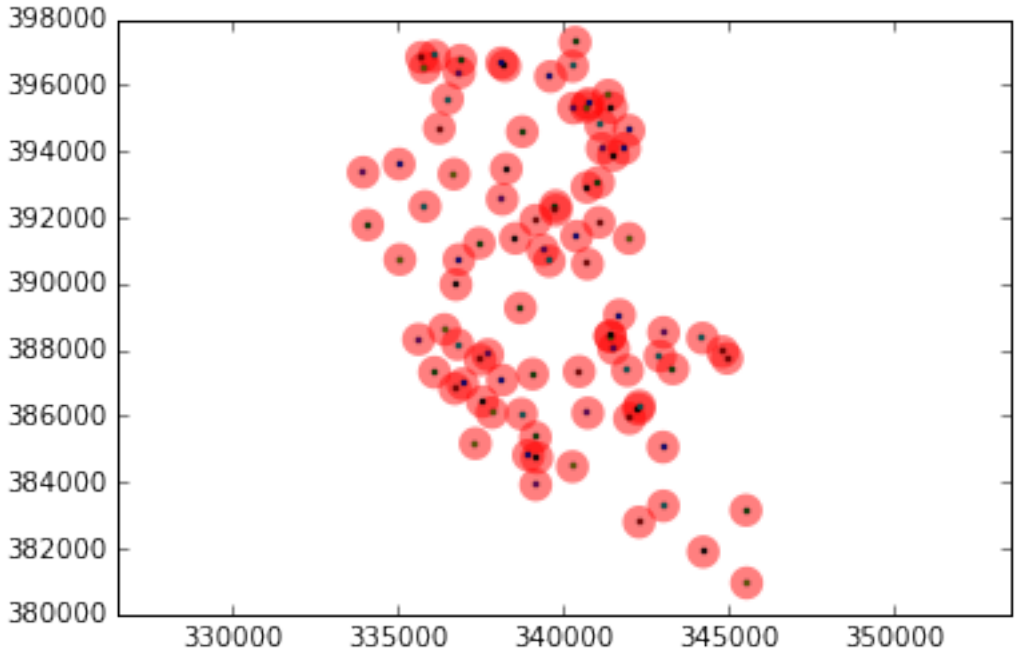
These tells us it uses projection 27700 in the EPSG system. If we look it up, we will find that this corresponds with the Ordnance Survey projection, which is expressed in metres. Hence if we want, for example, a buffer of 500m. around each of these places, we can simply obtain it by:

```
In [42]: buf = namp.buffer(500)
         buf.head()

Out[42]: 0    POLYGON ((340105 396261, 340102.5923633361 396...
         1    POLYGON ((340258 392357, 340255.5923633361 392...
         2    POLYGON ((340268 392217, 340265.5923633361 392...
         3    POLYGON ((340769 396567, 340766.5923633361 396...
         4    POLYGON ((340796 395304, 340793.5923633361 395...
         dtype: object
```

And plotting it is equally straighforward:

```
In [43]: f, ax = plt.subplots(1)
         # Plot buffer
         for poly in buf:
             gpd.plotting.plot_multipolygon(ax, poly, linewidth=0)
         # Plot named places on top for reference
         namp.plot(axes=ax)
         plt.axis('equal')
         plt.show()
```



NOTE The folowing are extensions and as such are not required to complete this section. They are intended as additional resources to explore further possibilities that Python allows to play with representing spatial data.

1.6 [Extension I] Creating interactive maps

```
In [44]: import mplleaflet
```

Introducing interactivity in cartography is a huge subfield that involves many considerations, most of them going well beyond the scope of this course. However, for very simple cases, it is straightforward to add geometries (points, polygons, lines) from a `GeoDataFrame` to a web map based on OpenStreetMap, for example. This can be done using the additional library `mplleaflet`.

As an illustration, we will display the LSOA polygons:

```
In [45]: f, ax = plt.subplots(1)
         for line in rwy_tun['geometry']:
             gpd.plotting.plot_multilinestring(ax, line, color='yellow', linewidth=3)
         mplleaflet.display(fig=f, crs=rwy_tun.crs)
```

```
Out[45]: <IPython.core.display.HTML object>
```

Note that essentially, all it takes is to produce the map in the standard way and, at the end of the code block, add a line that passes the figure object (`f`) and the CRS into the converter that will return the interactive map.

[Optional exercise. Difficulty: 2/5]
Create an interactive map of the smallest population areas in Liverpool

---

1.7 [Extension II] Adding base layers from raster imagery

Sometimes, in addition to vector data, we need to rely on raster sources and combine both in the same map. For all things raster, Python has the library `rasterio`, which provides simplified access to raster sources and many tools for manipulation, transformation, and integration.

In this example (based on this other one by Carson Farmer), we read in open data from the Ordnance Survey for the whole UK. The data corresponds with the GB Overview Map, and is obtained through an OS Open Data license. The original source file can be downloaded from the following link:

https://www.ordnancesurvey.co.uk/opendatadownload/products.html

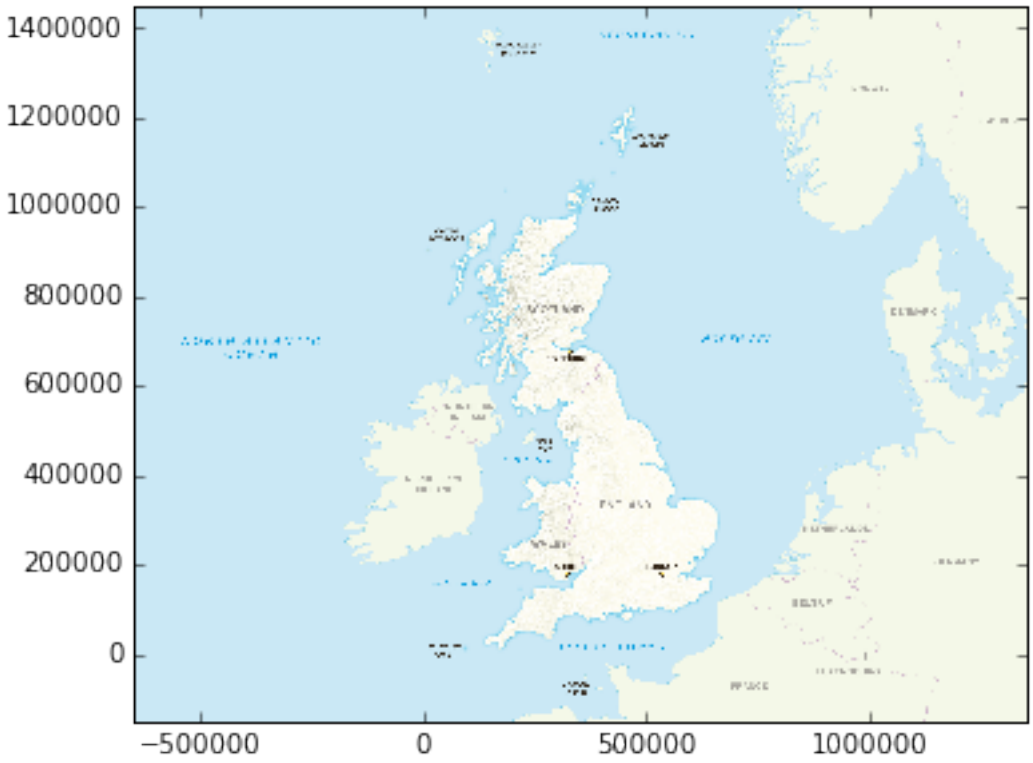
To import the data using `rasterio`, we need to run the following code:

```
In [46]: import rasterio
         import numpy as np

         # Reading in data
         source = rasterio.open('figs/lab03_GBOverview.tif', 'r')
         red = source.read(1)
         green = source.read(2)
         blue = source.read(3)
         pix = np.dstack((red, green, blue))
         bounds = (source.bounds.left, source.bounds.right, \
                   source.bounds.bottom, source.bounds.top)
```

Once we have the data read, plotting it follows a similar pattern as we have seen before, except now we are using the `plt.imshow` command, which displays an image:

```
In [47]: # Plotting
         f = plt.figure(figsize=(6, 6))
         ax = plt.imshow(pix, extent=bounds)
```



[Optional exercise. Difficulty: 3/5]
Reproduce the raster map above in larger size (e.g. 14 by 14) and add the polygons of Liverpool.

1.8 [Extension III] Advanced GIS operations

[NOTE] If you are going to try some of these operations, make sure to have the library `rtree` correctly installed.

- Spatial joins

  https://github.com/geopandas/geopandas/blob/master/examples/spatial_joins.ipynb

- Spatial overlays

  https://github.com/geopandas/geopandas/blob/master/examples/overlays.ipynb

---