Contents

ENVS363/563

Geographic Data Science

Welcome to Geographic Data Science, a course taught by Dr. Dani Arribas-Bel in the Autumn of 2020 at the University of Liverpool.

Contact

Dani Arribas-Bel - D.Arribas-Bel [at] liverpool.ac.uk
Senior Lecturer in Geographic Data Science
Office 508, Roxby Building,
University of Liverpool - 74 Bedford St S,
Liverpool, L69 7ZT,
United Kingdom.

Note

A PDF version of this course is available for download here

Citation

JOSE 10.21105/jose.00042


```
@article{darribas_gds_course,
  author = {Dani Arribas-Bel},
  title = {A course on Geographic Data Science},
  year = 2019,
  journal = {The Journal of Open Source Education},
  volume = 2,
  number = 14,
  doi = {https://doi.org/10.21105/jose.00042}
}
```

Overview

Aims

The module provides students with little or no prior knowledge core competences in Geographic Data Science (GDS). This includes the following:

- Advancing their statistical and numerical literacy.
- Introducing basic principles of programming and state-of-the-art computational tools for GDS.
- Presenting a comprehensive overview of the main methodologies available to the Geographic Data Scientist, as well as their intuition as to how and when they can be applied.
- Focusing on real world applications of these techniques in a geographical and applied context.

Learning outcomes

By the end of the course, students will be able to:

- Demonstrate advanced GIS/GDS concepts and be able to use the tools programmatically to import, manipulate and analyse spatial data in different formats.
- Understand the motivation and inner workings of the main methodological approcahes of GDS, both analytical and visual.
- Critically evaluate the suitability of a specific technique, what it can offer and how it can help answer questions of interest.
- Apply a number of spatial analysis techniques and explain how to interpret the results, in a process of turning data into information.
- When faced with a new data-set, work independently using GIS/GDS tools programmatically to extract valuable insight.

Feedback strategy

The student will receive feedback through the following channels:

- Formal assessment of three summative assignments: two tests and a computational essay. This will be on the form of reasoning of the mark assigned as well as comments specifying how the mark could be improved. This will be provided no later than three working weeks after the deadline of the assignment submission.
- Direct interaction with Module Leader and demonstrators in the computer labs. This will take place in each of the scheduled lab sessions of the course.
- Online forum maintained by the Module Leader where students can contribute by asking and answering questions related to the module.

Key texts and learning resources

Access to materials, including lecture slides and lab notebooks, is centralized through the use of a course website available in the following url:

> https://darribas.org/gds_course

Specific videos, (computational) notebooks, and other resources, as well as academic references are provided for each learning block.

In addition, the currently-in-progress book *"Geographic Data Science with PySAL and the PyData stack"* provides and additional resource for more in-depth coverage of similar content.

Syllabus

Week 1: Introduction

- Lecture: Geographic Data Science.
- Tutorial: Tools + Manipulating data in Python - Tidy Data.

Week 2: Modern Computational Environments

- Lecture: Modern Computational Environments.
- Tutorial: Manipulating data in Python - Advanced Tricks.

Week 3: Spatial Data

- Lecture: Spatial Data.
- Tutorial: Manipulating geospatial data in Python.

Week 4: (Geo)Visualization + Choropleths

- Lecture: (Geo)Visualization + Choropleths.
- Tutorial: Mapping deprivation.

Week 5: Spatial Weights

- Lecture: Spatial Weights.
- Tutorial:
  - **TEST 1** (1h): Thursday Oct. 24th
  - Spatial Weights with PySAL.

Week 6: ESDA

- Lecture: Exploratory Spatial Data Analysis (ESDA).
- Tutorial: ESDA in Python.

Week 7: Clustering

- Lecture: Clustering.
- Tutorial: Geodemographic analysis.

Week 8: Point Data

- Lecture: Point Data.
- Tutorial: Exploring Twitter patterns.

Week 9

- Lecture: Assignment preparation.
- Tutorial:
  - **TEST 2** (1h): Thursday Nov. 21st
  - Assignment Clinic

Week 10: (Spatial) causal inference

- Lecture: Spatial causal inference.

- Tutorial: Assignment Clinic.

Week 11: Geographic Data Science in Action

- Lecture: Geographic Data Science *in the wild*.
- Tutorial: Assignment Clinic.

ASSIGNMENT due on Thursday, December 5th-2019.

Bibliography

[AB14]
Daniel Arribas-Bel. Accidental, open and everywhere: emerging data sources for the understanding of cities. *Applied Geography*, 49():45 – 53, 2014. The New Urban World. URL: http://www.sciencedirect.com/science/article/pii/S0143622813002178, doi:http://dx.doi.org/10.1016/j.apgeog.2013.09.012.

[Don17]
David Donoho. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, 2017.

[LR17]
David Lazer and Jason Radford. Data ex machina: introduction to big data. *Annual Review of Sociology*, 2017.

[McK12]
Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

[RABWng]
Sergio J. Rey, Daniel Arribas-Bel, and Levi J. Wolf. *Geographic Data Science with PySAL and the PyData stack*. CRC press, forthcoming.

[SONeil13]
Rachel Schutt and Cathy O'Neil. *Doing data science: Straight talk from the frontline*. " O'Reilly Media, Inc.", 2013.

[SAB19]
Alex Singleton and Daniel Arribas-Bel. Geographic data science. *Geographical Analysis*, 2019.

[Som18]
James Somers. The scientific paper is obsolete. *The Atlantic*, 2018.

[Wic14]
Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):??–??, 9 2014. URL: http://www.jstatsoft.org/v59/i10.

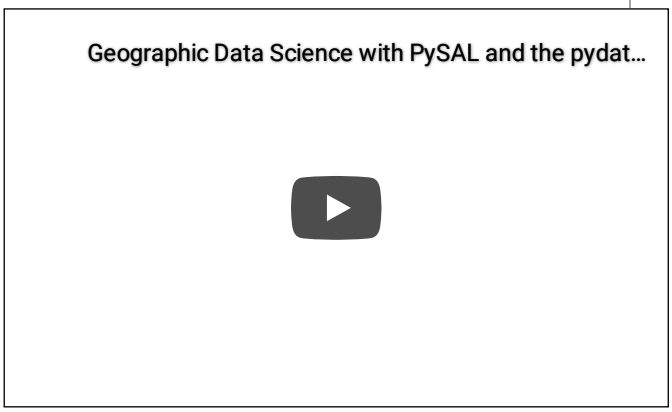
Concepts

The concepts in this block are delivered through:

- Two video clips
- Accompanying slides
- `[Optional]` further readings for the interested and curious mind

This course

Let us start from the beginning, here is a snapshot of what this course is about! In the following clip, you will find out about the philosophy behind the course, how the content is structured, and why this is all designed like this. And, also, a little bit about the assessment…



Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

What is *Geographic Data Science*?

Once it is clearer how this course is going to run, let's dive right into why this course is necessary. The following clip is taken from a keynote response by Dani Arribas-Bel at the first Spatial Data Science Conference, organised by CARTO and held in Brooklyn in 2017. The talk provides a bit of background and context, which will hopefully help you understand a bit better what Geographic Data Science is.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

20:50

Further readings

To get a better picture, the following readings complement the overview provided above very well:

1. The introductory chapter to “Doing Data Science” [SONeil13], by Cathy O’Neil and Rachel Schutt is general overview of why we needed Data Science and where if came from.
2. A slightly more technical historical perspective on where Data Science came from and where it might go can be found in David Donoho’s recent overview [Don17].
3. A geographic take on Data Science, proposing more interaction between Geography and Data Science [SAB19].

Bonus

The chapter is available free online HTML | PDF

Hands-on

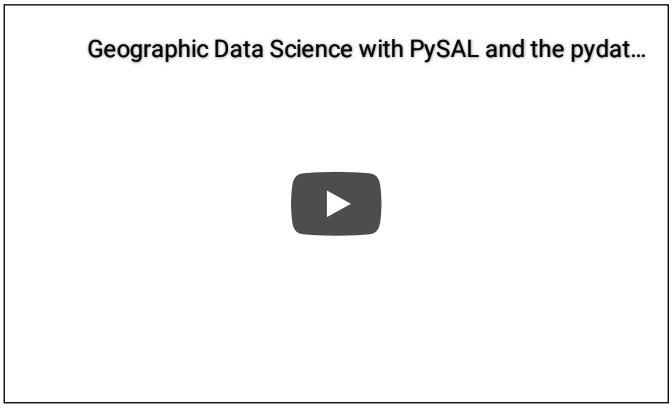
In this first

Following this course interactively

Maybe use this for start up a notebook:

> http://darribas.org/gds19/content/labs/begin.html

Video with walk through Jupyter Lab



Re-write this:

> http://darribas.org/gds19/content/labs/lab_00.html

Software infrastructure for the course

- Point to available guides + install options

Files

Do-It-Yourself

To do:

- Make sure you have the setup installed and/or access to a campus computer to complete the course
- Launch JupyterLab, and explore

Concepts

The ideas behind this block are better communicated through narrative than video or lectures. Hence, the concepts section are delivered through a few references you are expected to read. These will total up about one and a half hours of your focused time.

Open Science

The first part of this block is about setting the philosophical background. Why do we care about the processes and tools we use when we do computational work? Where do the current paradigm come from? Are we on the verge of a new model? For all of this, we we have two reads to set the tone. Make sure to get those in first thing before moving on to the next bits.

- First half of Chapter 1 in "Geographic Data Science with PySAL and the PyData stack" [RABWng]. — Read the chapter here. Estimated time: 15min.
- The 2018 Atlantic piece *"The scientific paper is obsolete"* on computational notebooks, by James Somers [Som18]. — Read the piece here. Estimated time: 35min.

Modern Scientific Tools

Once we know a bit more about why we should care about the tools we use, let's dig into those that will underpin much of this course. This part is interesting in itself, but will also valuable to better understand the practical aspects of the course. Again, we have two reads here to set the tone and complement the practical introduction we saw in the Hands-on and DIY parts of the previous block. We are closing the circle here:

- Second half of Chapter 1 in "Geographic Data Science with PySAL and the PyData stack" [RABWng]. — Read the chapter here. Estimated time: 15min.
- The chapter in the GIS&T Book of Knowledge on computational notebooks, by Geoff Boeing and Dani Arribas-Bel.

Hands-on

Once we know a bit about what computational notebooks are and why we should care about them, let's jump to using them! This section introduces you to using Python for manipulating tabular data. Please read through it carefully and pay attention to how ideas about manipulating data are translated into Python code that "does stuff". For this part, you can read directly from the course website, although it is recommended you follow the section interactively by running code on your own.

Once you have read through and have a bit of a sense of how things work, jump on the Do-It-Yourself section, which will provide you with a challenge to complete it on your own, and will allow you to put what you have already learnt to good use. Happy hacking!

Data munging

Real world datasets are messy. There is no way around it: datasets have "holes" (missing data), the amount of formats in which data can be stored is endless, and the best structure to share data is not always the optimum to analyze them, hence the need to munge them. As has been correctly pointed out in many outlets (e.g.), much of the time spent in what is called (Geo-)Data Science is related not only to sophisticated modeling and insight, but has to do with much more basic and less exotic tasks such as obtaining data, processing, turning them into a shape that makes analysis possible, and exploring it to get to know their basic properties.

For how labor intensive and relevant this aspect is, there is surprisingly very little published on patterns, techniques, and best practices for quick and efficient data cleaning, manipulation, and transformation. In this session, you will use a few real world datasets and learn how to process them into Python so they can be transformed and manipulated, if necessary, and analyzed. For this, we will introduce some of the bread and butter of data analysis and scientific computing in Python. These are fundamental tools that are constantly used in almost any task relating to data analysis.

This notebook covers the basic and the content that is expected to be learnt by every student. We use a prepared dataset that saves us much of the more intricate processing that goes beyond the introductory level the session is aimed at. As a companion to this introduction, there is an additional notebook (see link on the website page for Lab 01) that covers how the dataset used here was prepared from raw data downloaded from the internet, and includes some additional exercises you can do if you want dig deeper into the content of this lab.

In this notebook, we discuss several patterns to clean and structure data properly, including tidying, subsetting, and aggregating; and we finish with some basic visualization. An additional extension presents more advanced tricks to manipulate tabular data.

Before we get our hands data-dirty, let us import all the additional libraries we will need, so we can get that out of the way and focus on the task at hand:

```
# This ensures visualizations are plotted inside the notebook
%matplotlib inline

import os              # This provides several system utilities
import pandas as pd    # This is the workhorse of data munging in Python
import seaborn as sns  # This allows us to efficiently and beautifully plot
```

Dataset

We will be exploring some demographic characteristics in Liverpool. To do that, we will use a dataset that contains population counts, split by ethnic origin. These counts are aggregated at the Lower Layer Super Output Area (LSOA from now on). LSOAs are an official Census geography defined by the Office of National Statistics. You can think of them, more or less, as neighbourhoods. Many data products (Census, deprivation indices, etc.) use LSOAs as one of their main geographies.

To make things easier, we will read data from a file posted online so, for now, you do not need to download any dataset:

```
# Read table
db = pd.read_csv("https://darribas.org/gds_course/content/data/liv_pop.csv",
                 index_col='GeographyCode')
```

Important

Make sure you are connected to the internet when you run this cell

Let us stop for a minute to learn how we have read the file. Here are the main aspects to keep in mind:

- We are using the method `read_csv` from the `pandas` library, which we have imported with the alias `pd`.
- In this form, all that is required is to pass the path to the file we want to read, which in this case is a web address.
- The argument `index_col` is not strictly necessary but allows us to choose one of the columns as the index of the table. More on indices below.
- We are using `read_csv` because the file we want to read is in the `csv` format. However, `pandas` allows for many more formats to be read and write. A full list of formats supported may be found here.
- To ensure we can access the data we have read, we store it in an *object* that we call `db`. We will see more on what we can do with it below but, for now, just keep in mind that allows us to save the result of `read_csv`.

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on this link and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
db = pd.read_csv("liv_pop.csv", index_col="GeographyCode")
```

Data, sliced and diced

Now we are ready to start playing and interrogating the dataset! What we have at our fingertips is a table that summarizes, for each of the LSOAs in Liverpool, how many people live in each, by the region of the world where they were born. Now, let us learn a few cool tricks built into `pandas` that work out-of-the box with a table like ours.

- Inspecting what it looks like. We can check the top (bottom) X lines of the table by passing X to the method `head` (`tail`). For example, for the top/bottom five lines:

```
db.head()
```

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

```
db.tail()
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
E01033764	2106	32	49	15	0
E01033765	1277	21	33	17	3
E01033766	1028	12	20	8	7
E01033767	1003	29	29	5	1
E01033768	1016	69	111	21	6

- Getting an overview of the table:

```
db.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 5 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   Europe                         298 non-null    int64
 1   Africa                         298 non-null    int64
 2   Middle East and Asia           298 non-null    int64
 3   The Americas and the Caribbean  298 non-null    int64
 4   Antarctica and Oceania         298 non-null    int64
dtypes: int64(5)
memory usage: 14.0+ KB
```

- Getting an overview of the *values* of the table:

```
db.describe()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
count	298.00000	298.000000	298.000000	298.000000	298.000000
mean	1462.38255	29.818792	62.909396	8.087248	1.949664
std	248.67329	51.606065	102.519614	9.397638	2.168216
min	731.00000	0.000000	1.000000	0.000000	0.000000
25%	1331.25000	7.000000	16.000000	2.000000	0.000000
50%	1446.00000	14.000000	33.500000	5.000000	1.000000
75%	1579.75000	30.000000	62.750000	10.000000	3.000000
max	2551.00000	484.000000	840.000000	61.000000	11.000000

Note how the output is also a `DataFrame` object, so you can do with it the same things you would with the original table (e.g. writing it to a file).

In this case, the summary might be better presented if the table is "transposed":

```
db.describe().T
```

	count	mean	std	min	25%	50%	7
Europe	298.0	1462.382550	248.673290	731.0	1331.25	1446.0	157
Africa	298.0	29.818792	51.606065	0.0	7.00	14.0	3
Middle East and Asia	298.0	62.909396	102.519614	1.0	16.00	33.5	6
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0	2.00	5.0	1
Antarctica and Oceania	298.0	1.949664	2.168216	0.0	0.00	1.0	

- Equally, common descriptive statistics are also available:

```
# Obtain minimum values for each table
db.min()
```

```
Europe                          731
Africa                            0
Middle East and Asia              1
The Americas and the Caribbean    0
Antarctica and Oceania            0
dtype: int64
```

```
# Obtain minimum value for the column `Europe`
db['Europe'].min()
```

```
731
```

Note here how we have restricted the calculation of the maximum value to one column only.

Similarly, we can restrict the calculations to a single row:

```
# Obtain standard deviation for the row `E01006512`,
# which represents a particular LSOA
db.loc['E01006512', :].std()
```

```
457.8842648530303
```

- Creation of new variables: we can generate new variables by applying operations on existing ones. For example, we can calculate the total population by area. Here is a couple of ways to do it:

```
# Longer, hardcoded
total = db['Europe'] + db['Africa'] + db['Middle East and Asia'] + \
        db['The Americas and the Caribbean'] + db['Antarctica and Oceania']
# Print the top of the variable
total.head()
```

```
GeographyCode
E01006512    1880
E01006513    2941
E01006514    2108
E01006515    1208
E01006518    1696
dtype: int64
```

```
# One shot
total = db.sum(axis=1)
# Print the top of the variable
total.head()
```

```
GeographyCode
E01006512    1880
E01006513    2941
E01006514    2108
E01006515    1208
E01006518    1696
dtype: int64
```

Note how we are using the command `sum`, just like we did with `max` or `min` before but, in this case, we are not applying it over columns (e.g. the max of each column), but over rows, so we get the total sum of populations by areas.

Once we have created the variable, we can make it part of the table:

```
db['Total'] = total
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
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

- Assigning new values: we can easily generate new variables with scalars, and modify those.

```
# New variable with all ones
db['ones'] = 1
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
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

And we can modify specific values too:

```
db.loc['E01006512', 'ones'] = 3
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
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

- Permanently deleting variables is also within reach of one command:

```
del db['ones']
db.head()
```

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

- Index-based querying.

We have already seen how to subset parts of a `DataFrame` if we know exactly which bits we want. For example, if we want to extract the total and European population of the first four areas in the table, we use `loc` with lists:

```
eu_tot_first4 = db.loc[['E01006512', 'E01006513', 'E01006514', 'E01006515'], \
                       ['Total', 'Europe']]
eu_tot_first4
```

GeographyCode	Total	Europe
E01006512	1880	910
E01006513	2941	2225
E01006514	2108	1786
E01006515	1208	974

- Querying based on conditions.

However, sometimes, we do not know exactly which observations we want, but we do know what conditions they need to satisfy (e.g. areas with more than 2,000 inhabitants). For these cases, `DataFrames` support selection based on conditions. Let us see a few examples. Suppose we want to select…

… areas with more than 2,500 people in Total:

```
m5k = db.loc[db['Total'] > 2500, :]
m5k
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
E01006513	2225	61	595	53	7	2941
E01006747	2551	163	812	24	2	3552
E01006751	1843	139	568	21	1	2572

… areas where there are no more than 750 Europeans:

```
nm5ke = db.loc[db['Europe'] < 750, :]
nm5ke
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
E01033757	731	39	223	29	3	1025

… areas with exactly ten person from Antarctica and Oceania:

```
oneOA = db.loc[db['Antarctica and Oceania'] == 10, :]
oneOA
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
E01006679	1353	484	354	31	10	2232

Pro-tip: these queries can grow in sophistication with almost no limits. For example, here is a case where we want to find out the areas where European population is less than half the population:

```
eu_lth = db.loc[(db['Europe'] * 100. / db['Total']) < 50, :]
eu_lth
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
E01006512	910	106	840	24	0	1880

- Combining queries.

Now all of these queries can be combined with each other, for further flexibility. For example, imagine we want areas with more than 25 people from the Americas and Caribbean, but less than 1,500 in total:

```
ac25_l500 = db.loc[(db['The Americas and the Caribbean'] > 25) & \
                   (db['Total'] < 1500), :]
ac25_l500
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
E01033750	1235	53	129	26	5	1448
E01033752	1024	19	114	33	6	1196
E01033754	1262	37	112	32	9	1452
E01033756	886	31	221	42	5	1185
E01033757	731	39	223	29	3	1025
E01033761	1138	52	138	33	11	1372

- Sorting.

Among the many operations `DataFrame` objects support, one of the most useful ones is to sort a table based on a given column. For example, imagine we want to sort the table by total population:

```
db_pop_sorted = db.sort_values('Total', ascending=False)
db_pop_sorted.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
E01006747	2551	163	812	24	2	3552
E01006513	2225	61	595	53	7	2941
E01006751	1843	139	568	21	1	2572
E01006524	2235	36	125	24	11	2431
E01006787	2187	53	75	13	2	2330

If you inspect the help of `db.sort_values`, you will find that you can pass more than one column to sort the table by. This allows you to do so-called hiearchical sorting: sort first based on one column, if equal then based on another column, etc.

Visual exploration

The next step to continue exploring a dataset is to get a feel for what it looks like, visually. We have already learnt how to unconver and inspect specific parts of the data, to check for particular cases we might be intersted in. Now we will see how to plot the data to get a sense of the overall distribution of values. For that, we will be using the Python library `seaborn`.

- Histograms.

One of the most common graphical devices to display the distribution of values in a variable is a histogram. Values are assigned into groups of equal intervals, and the groups are plotted as bars rising as high as the number of values into the group.

A histogram is easily created with the following command. In this case, let us have a look at the shape of the overall population:

```
_ = sns.distplot(db['Total'], kde=False)
```

Note we are using `sns` instead of `pd`, as the function belongs to `seaborn` instead of `pandas`.

We can quickly see most of the areas contain somewhere between 1,200 and 1,700 people, approx. However, there are a few areas that have many more, even up to 3,500 people.

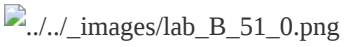
An additional feature to visualize the density of values is called `rug`, and adds a little tick for each value on the horizontal axis:

```
_ = sns.distplot(db['Total'], kde=False, rug=True)
```

- Kernel Density Plots

Histograms are useful, but they are artificial in the sense that a continuous variable is made discrete by turning the values into discrete groups. An alternative is kernel density estimation (KDE), which produces an empirical density function:

```
_ = sns.kdeplot(db['Total'], shade=True)
```



- Line and bar plots

Another very common way of visually displaying a variable is with a line or a bar chart. For example, if we want to generate a line plot of the (sorted) total population by area:

```
_ = db['Total'].sort_values(ascending=False).plot()
```

```
/opt/conda/lib/python3.7/site-packages/pandas/plotting/_matplotlib/core.py:1235:
UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(xticklabels)
```

For a bar plot all we need to do is to change from `plot` to `plot.bar`. Since there are many neighbourhoods, let us plot only the ten largest ones (which we can retrieve with `head`):

```
_ = db['Total'].sort_values(ascending=False)\
                .head(10)\
                .plot.bar()
```

We can turn the plot around by displaying the bars horizontally (see how it's just changing `bar` for `barh`). Let's display now the top 50 areas and, to make it more readable, let us expand the plot's height:

```
_ = db['Total'].sort_values()\
                .head(50)\
                .plot.barh(figsize=(6, 20))
```

Un/tidy data

> **Warning**
>
> This section is a bit more advanced and hence considered optional. Fell free to skip it, move to the next, and return later when you feel more confident.

> *Happy families are all alike; every unhappy family is unhappy in its own way.*
> Leo Tolstoy.

Once you can read your data in, explore specific cases, and have a first visual approach to the entire set, the next step can be preparing it for more sophisticated analysis. Maybe you are thinking of modeling it through regression, or on creating subgroups in the dataset with particular characteristics, or maybe you simply need to present summary measures that relate to a slightly different arrangement of the data than you have been presented with.

For all these cases, you first need what statistician, and general R wizard, Hadley Wickham calls *"tidy data"*. The general idea to "tidy" your data is to convert them from whatever structure they were handed in to you into one that allows convenient and standardized manipulation, and that supports directly inputting the data into what he calls "*tidy*" analysis tools. But, at a more practical level, what is exactly *"tidy data"*? In Wickham's own words:

> *Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.*

He then goes on to list the three fundamental characteristics of *"tidy data"*:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

If you are further interested in the concept of *"tidy data"*, I recommend you check out the original paper (open access) and the public repository associated with it.

Let us bring in the concept of "*tidy data*" to our own Liverpool dataset. First, remember its structure:

```
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
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

Thinking through *tidy* lenses, this is not a tidy dataset. It is not so for each of the three conditions:

- Starting by the last one (*each type of observational unit forms a table*), this dataset actually contains not one but two observational units: the different areas of Liverpool, captured by `GeographyCode`; *and* subgroups of an area. To *tidy* up this aspect, we can create two different tables:

```
# Assign column `Total` into its own as a single-column table
db_totals = db[['Total']]
db_totals.head()
```

	Total
GeographyCode	
E01006512	1880
E01006513	2941
E01006514	2108
E01006515	1208
E01006518	1696

```
# Create a table `db_subgroups` that contains every column in `db` without `Total`
db_subgroups = db.drop('Total', axis=1)
db_subgroups.head()
```

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

Note we use `drop` to exclude "Total", but we could also use a list with the names of all the columns to keep. Additionally, notice how, in this case, the use of `drop` (which leaves `db` untouched) is preferred to that of `del` (which permanently removes the column from `db`).

At this point, the table `db_totals` is tidy: every row is an observation, every table is a variable, and there is only one observational unit in the table.

The other table (`db_subgroups`), however, is not entirely tidied up yet: there is only one observational unit in the table, true; but every row is not an observation, and there are variable values as the names of columns (in other words, every column is not a variable). To obtain a fully tidy version of the table, we need to re-arrange it in a way that every row is a population subgroup in an area, and there are three variables: `GeographyCode`, population subgroup, and population count (or frequency).

Because this is actually a fairly common pattern, there is a direct way to solve it in `pandas`:

```
tidy_subgroups = db_subgroups.stack()
tidy_subgroups.head()
```

```
GeographyCode
E01006512      Europe                          910
               Africa                          106
               Middle East and Asia            840
               The Americas and the Caribbean   24
               Antarctica and Oceania            0
dtype: int64
```

The method `stack`, well, "stacks" the different columns into rows. This fixes our "tidiness" problems but the type of object that is returning is not a `DataFrame`:

```
type(tidy_subgroups)
```

```
pandas.core.series.Series
```

It is a `Series`, which really is like a `DataFrame`, but with only one column. The additional information (`GeographyCode` and population group) are stored in what is called an multi-index. We will skip these for now, so we would really just want to get a `DataFrame` as we know it out of the `Series`. This is also one line of code away:

```
# Unfold the multi-index into different, new columns
tidy_subgroupsDF = tidy_subgroups.reset_index()
tidy_subgroupsDF.head()
```

	GeographyCode	level_1	0
0	E01006512	Europe	910
1	E01006512	Africa	106
2	E01006512	Middle East and Asia	840
3	E01006512	The Americas and the Caribbean	24
4	E01006512	Antarctica and Oceania	0

To which we can apply to renaming to make it look better:

```
tidy_subgroupsDF = tidy_subgroupsDF.rename(columns={'level_1': 'Subgroup', 0: 'Freq'})
tidy_subgroupsDF.head()
```

	GeographyCode	Subgroup	Freq
0	E01006512	Europe	910
1	E01006512	Africa	106
2	E01006512	Middle East and Asia	840
3	E01006512	The Americas and the Caribbean	24
4	E01006512	Antarctica and Oceania	0

Now our table is fully tidied up!

Grouping, transforming, aggregating

One of the advantage of tidy datasets is they allow to perform advanced transformations in a more direct way. One of the most common ones is what is called “group-by” operations. Originated in the world of databases, these operations allow you to group observations in a table by one of its labels, index, or category, and apply operations on the data group by group.

For example, given our tidy table with population subgroups, we might want to compute the total sum of population by each group. This task can be split into two different ones:

- Group the table in each of the different subgroups.
- Compute the sum of `Freq` for each of them.

To do this in `pandas`, meet one of its workhorses, and also one of the reasons why the library has become so popular: the `groupby` operator.

```
pop_grouped = tidy_subgroupsDF.groupby('Subgroup')
pop_grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f682872db10>
```

The object `pop_grouped` still hasn’t computed anything, it is only a convenient way of specifying the grouping. But this allows us then to perform a multitude of operations on it. For our example, the sum is calculated as follows:

```
pop_grouped.sum()
```

	Freq
Subgroup	
Africa	8886
Antarctica and Oceania	581
Europe	435790
Middle East and Asia	18747
The Americas and the Caribbean	2410

Similarly, you can also obtain a summary of each group:

```
pop_grouped.describe()
```

	Freq						
	count	mean	std	min	25%	50%	7
Subgroup							
Africa	298.0	29.818792	51.606065	0.0	7.00	14.0	
Antarctica and Oceania	298.0	1.949664	2.168216	0.0	0.00	1.0	
Europe	298.0	1462.382550	248.673290	731.0	1331.25	1446.0	1
Middle East and Asia	298.0	62.909396	102.519614	1.0	16.00	33.5	
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0	2.00	5.0	

We will not get into it today as it goes beyond the basics we want to conver, but keep in mind that `groupby` allows you to not only call generic functions (like `sum` or `describe`), but also your own functions. This opens the door for virtually any kind of transformation and aggregation possible.

Additional lab materials

The following provide a good "next step" from some of the concepts and tools covered in the lab and DIY sections of this block:

- This NY Times article does a good job at conveying the relevance of data "cleaning" and munging.
- A good introduction to data manipulation in Python is Wes McKinney's "Python for Data Analysis" [McK12].
- To explore further some of the visualization capabilities in at your fingertips, the Python library `seaborn` is an excellent choice. Its online tutorial is a fantastic place to start.
- A good extension is Hadley Wickham' "Tidy data" paper [Wic14], which presents a very popular way of organising tabular data for efficient manipulation.

Do-It-Yourself

```
import pandas
```

This section is all about you taking charge of the steering wheel and choosing your own adventure. For this block, we are going to use what we've learnt before to take a look at a dataset of casualties in the war in Afghanistan. The data was originally released by Wikileaks, and the version we will use is published by The Guardian.

You can read a bit more about the data at The Guardian's data blog

Data preparation

Before you can set off on your data journey, the dataset needs to be read, and there's a couple of details we will get out of the way so it is then easier for you to start working.

The data are published on a Google Sheet you can check out at:

> https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit?hl=en#gid=1

As you will see, each row includes casualties recorded month by month, split by Taliban, Civilians, Afghan forces, and NATO.

To read it into a Python session, we need to slightly modify the URL to access it into:

```
url = ("https://docs.google.com/spreadsheets/d/"\
       "1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/"\
       "export?format=csv&gid=1")
url
```

```
'https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc
/export?format=csv&gid=1'
```

Note how we split the url into three lines so it is more readable in narrow screens. The result however, stored in `url`, is the same as one long string.

This allows us to read the data straight into a DataFrame, as we have done in the previous session:

```
db = pandas.read_csv(url, skiprows=[0, -1])
```

Note also we use the `skiprows=[0, -1]` to avoid reading the top (`0`) and bottom (`-1`) rows which, if you check on the Google Sheet, involves the title of the table.

Now we are good to go!

```
db.head()
```

	Year	Month	Taliban	Civilians	Afghan forces	Nato (detailed in spreadsheet)	Nato - official figures
0	2004.0	January	15	51	23	NaN	11.0
1	2004.0	February	NaN	7	4	5	2.0
2	2004.0	March	19	2	NaN	2	3.0
3	2004.0	April	5	3	19	NaN	3.0
4	2004.0	May	18	29	56	6	9.0

Tasks

Now, the challenge is to put to work what we have learnt in this block. For that, the suggestion is that you carry out an analysis of the Afghan Logs in a similar way as how we looked at population composition in Liverpool. These are of course very different

datasets reflecting immensely different realities. Their structure, however, is relatively parallel: both capture counts aggregated by a spatial (neighbourhood) or temporal unit (month), and each count is split by a few categories.

Try to answer the following questions:

- Obtain the minimum number of civilian casualties (in what month was that?)
- How many NATO casualties were registered in August 2008?
- What is the month with the most total number of casualties?
- Can you make a plot of the distribution of casualties over time?

> **Tip**
>
> You will need to first create a column with total counts

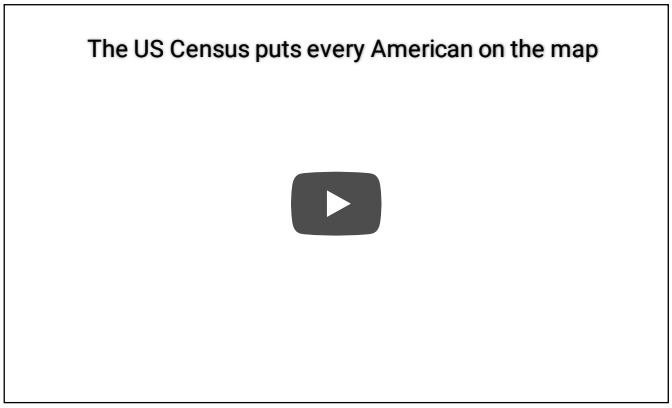
Concepts

This blocks explore spatial data, old and new. We start with an overview of traditional datasets, discussing their benefits and challenges for social scientists; then we move on to new forms of data, and how they pose different challenges, but also exciting opportunities. These two areas are covered with clips and slides that can be complemented with readings. Once conceptual areas are covered, we jump into working with spatial data in Python, which will prepare you for your own adventure in exploring spatial data.

“Good old” (geo) data

To understand what is new in new forms of data, it is useful to begin by considering traditional data. In this section we look at the main characteristics of traditional data available to Social Scientists. Warm up before the main part coming up next!

Before you jump on the clip, please watch the following video by the US Census Burearu, which will be discussed:

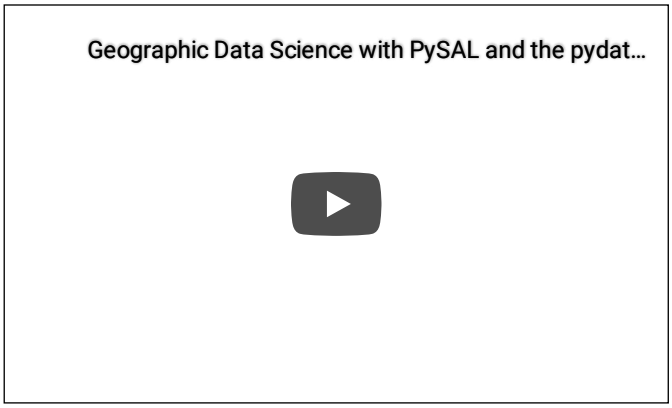


Then go on to the following clip, which will help you put the Census Bureau’s view in perspective:

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

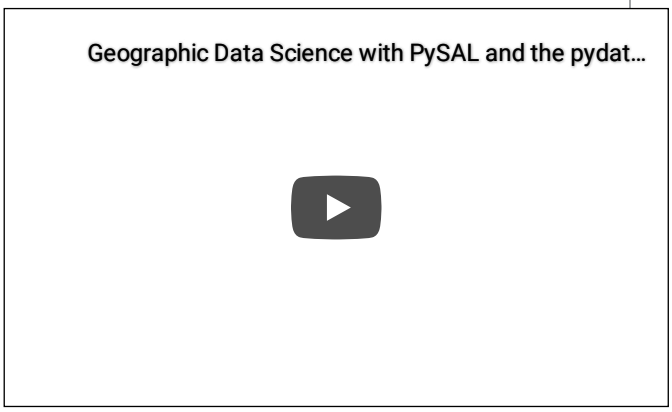


New forms of (geo) data

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



This section discusses two references in particular:

- “Data Ex-Machina”, by Lazer & Radford [LR17]
- And the accidental data paper by Dani Arribas-Bel [AB14]

Although both papers are discussed in the clip, if you are interested in the ideas mentioned, do go to the original sources as they provide much more detail and nuance.

Hands-on

Mapping in Python with `geopandas`

```
%matplotlib inline

import matplotlib.pyplot as plt
import geopandas as gpd
import palettable as pltt
from seaborn import palplot
```

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* sometimes 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:

To learn these concepts, we will be playing 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`:

```
# This might have to look different in your computer
lcp_dir = '../../../gds18_data/Liverpool/'
los_dir = '../../../gds18_data/E08000012_OS/'
```

IMPORTANT: the paths above might have look different in your computer. See this introductory notebook for more details about how to set your paths.

Loading up spatial data

The most direct 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 very similar to 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.

- 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:

```
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`:

```
lsoas.head()
```

	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:

```
lsoas.plot()
```

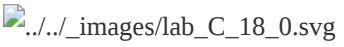
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe040d82940>
```


../../_images/lab_C_15_1.png

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:

```
lsoas.loc[0, 'geometry']
```


../../_images/lab_C_18_0.svg

- Lines

Displaying lines is as straight-forward as polygons. To load railway tunnels in Liverpool and name the rows after the `id` column (or to "index" them):

```
# Read file with tunnel
rwy_tun = gpd.read_file(los_dir + 'RailwayTunnel.shp')
# Index it on column `id`
rwy_tun = rwy_tun.set_index('id')
# Print summary info
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, similarly to the polygon case, if we pick the `"geometry"` column of a table with lines, a single row will display the geometry as well:

```
rwy_tun.loc['0ACD196C313D4F8DE050A00A568A6F6F', 'geometry']
```

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):

```
rwy_tun.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe040ca7b38>
```

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 a quick check of what lines look like.

[Optional 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:

```
namp = gpd.read_file(los_dir + 'NamedPlace.shp')
namp.head()
```

	id	distname	htmlname	clas
0	0EE7A103C03A8FBFE050A00A568A2502	Sugar Brook	Sugar Brook	Hydrog
1	0EE7A104A4B68FBFE050A00A568A2502	Sandfield Park	Sandfield Park	Land
2	0EE7A1041DB18FBFE050A00A568A2502	Sandfield Park	Sandfield Park	Popu
3	0EE7A1041DE48FBFE050A00A568A2502	Gillmoss	Gillmoss	Popu
4	0EE7A1041DE58FBFE050A00A568A2502	Croxteth	Croxteth	Popu

And the plot is produced by running:

```
namp.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe03ebeda90>
```

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):

```
lsoas.plot(alpha=0.1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe03eb68240>
```

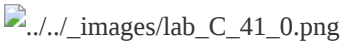
```
lsoas.plot(alpha=1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe03eb39588>
```

- 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:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Plot layer of polygons on the axis
lsoas.plot(ax=ax)
# Remove axis frames
ax.set_axis_off()
# Display
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 an extra line, if we are creating the plot within a figure, as we just did. To include text on top of the figure:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add layer of polygons on the axis
lsoas.plot(ax=ax)
# Add figure title
f.suptitle('Liverpool LSOAs')
# Display
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.

```
# Setup figure and axis with different size
f, ax = plt.subplots(1, figsize=(12, 12))
# Add layer of polygons on the axis
lsoas.plot(ax=ax)
# Display
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:

```
# Setup figure and axis with different size
f, ax = plt.subplots(1, figsize=(12, 4))
# Add layer of polygons on the axis
ax = lsoas.plot(ax=ax)
# Display
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")`.

```
# Setup figure and axis with different size
f, ax = plt.subplots(1, figsize=(12, 4))
# Add layer of polygons on the axis
lsoas.plot(ax=ax)
# Force axis to be on the same unit
plt.axis('equal')
# Display
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:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add layer of polygons on the axis, set fill color (`facecolor`) and boundary
# color (`edgecolor`)
lsoas.plot(linewidth=0.1, facecolor='red', edgecolor='grey', ax=ax)
# Display
plt.show()
```

Note how the lines are much thinner and discreet. In addition, 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, we call `plot` as usual, but pass in two new arguments: `facecolor`, to control the color each polygon is filled with, and `edgecolor`, to control the color of the boundary.
- Draw the map using `plt.show()`.

This approach works very similarly with other geometries, such as lines. For example, if we wanted to plot the railway tunnels in red, we would simply:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add layer with lines, set them red and with different line width
# and append it to the axis `ax`
rwy_tun.plot(linewidth=2, color='red', ax=ax)
# Display
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`).

- 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, we can modify this in a `GeoDataFrame`. First let us check if we have the information stored properly:

```
lsoas.crs
```

```
{'init': 'epsg:27700'}
```

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 quickest 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:

```
# Reproject (`to_crs`) and plot (`plot`) polygons
lsoas.to_crs(epsg=4326).plot()
# Set equal axis
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]

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"

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 most direct plot, one with the polygons from the LSOAs and the tunnels on top of them:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygons on to axis `ax`
lsoas.plot(ax=ax)
# Add a layer with lines on top in axis `ax`
rwy_tun.plot(ax=ax)
# Display
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.

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygons on to axis `ax`
lsoas.plot(ax=ax, facecolor='grey', edgecolor='white', linewidth=0.2)
# Add a layer with lines on top in axis `ax`
rwy_tun.plot(ax=ax, color='red')
# Display
plt.show()
```

[Optional exercise]

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`). Do not try to set the color to green or any other particular one, but you can play with the size of the dot.

Using palettes to create aesthetically pleasant maps

The choice of colors can influence the look and, ultimately, the effectiveness of a map. Although in some cases picking colors that simply allow you to distinguish the different elements might suffice, sometimes, you want to convey certain feelings (warmth, safety, etc.). In those cases, using preexisting palettes can be useful.

In this section, we will learn how to use pre-existent palettes to style your maps. We will be using the library `palettable`, which provides many “canned” palettes. We will also use the handy function `palplot` (from the library `seaborn`) to examine the colors

quickly.

For the sake of the example, let us use a palette based on one of Wes Anderson's movies, Darjeeling Limited:

Here is how you can pull out those colors:

```
wes = pltt.wesanderson.Darjeeling2_5.hex_colors
palplot(wes)
```

Now, note how the object `wes` simply contains a list of colors in the hex standard:

```
wes
```

```
['#D5E3D8', '#618A98', '#F9DA95', '#AE4B16', '#787064']
```

We will use these to style our map. For the sake of the example, let us use the following layers, available from the OS pack:

- `TidalWater`
- `Road`
- `TidalBoundary`
- `FunctionalSite`
- And the LSOA polygons used above.

Let us first read those in:

```
# NOTE: this might take a little bit depending on the machine
tidW = gpd.read_file(los_dir+'TidalWater.shp')
tidB = gpd.read_file(los_dir+'TidalBoundary.shp')
funS = gpd.read_file(los_dir+'FunctionalSite.shp')
road = gpd.read_file(los_dir+'Road.shp')
```

Technically speaking, there is nothing new to learn, other than keeping in mind that we need to add the layers in the right order. Let's go for it:

```
# NOTE: this might take a little bit depending on the machine

# Setup figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Add tidal water (remove boundary lines for the polygons)
tidW.plot(ax=ax, facecolor='#618A98', linewidth=0.)
# Add tidal boundaries
tidB.plot(ax=ax, color='#D5E3D8')
# Add LSOAs
lsoas.plot(ax=ax, facecolor='#F9DA95', edgecolor='#F9DA95', linewidth=0.)
# Add roads
road.plot(ax=ax, color='#AE4B16', linewidth=0.2)
# Add functional sites (remove boundary lines for the polygons)
funS.plot(ax=ax, facecolor='#787064', linewidth=0.)
# Remove axes
ax.set_axis_off()
# Impose same size for units across axes
plt.axis('equal')
# Display
plt.show()
```

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 involves 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:

```
# Set up figure and axes
f, ax = plt.subplots(1)
# Plot polygon layer
lsoas.plot(ax=ax, facecolor='grey', alpha=0.25, linewidth=0.1)
# Plot line layer
rwy_tun.plot(ax=ax, color='green', linewidth=3)
# Save figure to a PNG file
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.

```
# Set up figure and axes
f, ax = plt.subplots(1)
# Plot polygon layer
lsoas.plot(ax=ax, facecolor='grey', alpha=0.25, linewidth=0.1)
# Plot line layer
rwy_tun.plot(ax=ax, color='green', linewidth=3)
# Save figure to a PNG file
plt.savefig('liverpool_railway_tunels.png', dpi=1080)
```

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:

```
import pandas as pd

# Set the path to the location of the Liverpool data from the first practical
# just as you did at the beginning of the previous session
tab_path = 'data/liv_pop.csv'
```

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.

```
# Read file in
lsoa_orig_sub = pd.read_csv(tab_path, index_col=0)
# Create column with totals by area
lsoa_orig_sub['Total'] = lsoa_orig_sub.sum(axis=1)
# Display top of table
lsoa_orig_sub.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode						
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 an 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.

```
lsoas.head()
```

	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`:

```
geo_pop = lsoas.join(lsoa_orig_sub, on='LSOA11CD')
geo_pop.head()
```

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

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 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, earlier versions of `geopandas` appears to have a bug in the code that makes the joined table to loose the CRS. If this is the case, reattaching it is straightforward:

```
geo_pop.crs = lsoas.crs
```

Non-spatial manipulations

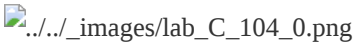
Once we have joined spatial and non-spatial data, we can use the techniques learned in manipulating and slicing non-spatial tables to create much richer maps. In particular, let us recall the example we worked through in the previous session in which we were selecting rows based on their population characteristics. 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):

```
smallest = geo_pop.sort_values('Total').head(10)
```

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

```
f, ax = plt.subplots(1, figsize=(6, 6))
# Base layer with all the areas for the background
geo_pop.plot(facecolor='black', linewidth=0.025, ax=ax)
# Smallest areas
smallest.plot(alpha=1, facecolor='red', linewidth=0, ax=ax)
ax.set_axis_off()
f.suptitle('Areas with smallest population')
plt.axis('equal')
plt.show()
```



[Optional exercise]

Create a map of Liverpool with the two largest areas for each of the different population subgroups in a different color each.

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`:

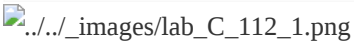
```
cents = geo_pop.centroid
cents.head()
```

```
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
```

Note how `cents` is not an entire table but a single column, or a `GeoSeries` object. This means you can plot it directly, just like a table:

```
cents.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe03e8fac88>
```


../../_images/lab_C_112_1.png

But you don't need to call a `geometry` column to inspect the spatial objects. In fact, if you do it will return an error because there is not any `geometry` column, the object `cents` itself is the geometry.

[Optional exercise]

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 straightforward exercise but computationally a tricky task to perform. The way to perform this operation in `GeoPandas` is through the `contains` method, available for each polygon object.

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

```
poly.contains(pt1)
```

```
True
```

```
poly.contains(pt2)
```

```
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]

This one is fairly advanced, so do not dispair if you cannot solve it. 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:

```
namp.crs
```

```
{'init': 'epsg:27700'}
```

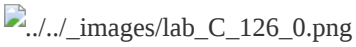
These tells us it uses projection 27700 in the EPSG system. If we look it up, we will find that this corresponds with the Ordnance Survey projection, which is expressed in metres. Hence if we want, for example, a buffer of 500m. around each of these places, we can simply obtain it by:

```
buf = namp.buffer(500)
buf.head()
```

```
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:

```
f, ax = plt.subplots(1)
# Plot buffer
buf.plot(ax=ax, linewidth=0)
# Plot named places on top for reference
# [NOTE how we modify the dot size (`markersize`)
# and the color (`color`)]
namp.plot(ax=ax, markersize=1, color='yellow')
plt.axis('equal')
plt.show()
```



[Optional exercise]

Generate a map of the Liverpool polygons in black and overlay on top of them yellow buffers of 250 metres around each centroid.

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.

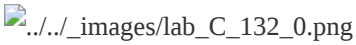
`[Extension I]` Adding base layers from web sources

A popular use of rasters is in the context of web tiles, which are a way of quickly obtaining geographical context to present spatial data. In Python, we can use `contextily` to pull down tiles and display them along with our own geographic data. Let us first import it the package:

```
import contextily as cx
```

We can begin by creating a map in the same way we would do normally, and then use the `add_basemap` command to, er, add a basemap:

```
ax = lsoas.plot(alpha=0.5)
cx.add_basemap(ax, crs=lsoas.crs);
```



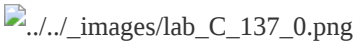
Note that we need to be explicit when adding the basemap to state the coordinate reference system (`crs`) our data is expressed in, `contextily` will not be able to pick it up otherwise. Conversely, we could change our data's CRS into Pseudo-Mercator, the native reference system for most web tiles:

```
lsoas_wm = lsoas.to_crs(epsg=3857)
ax = lsoas_wm.plot(alpha=0.5)
cx.add_basemap(ax);
```

Note how the coordinates are different but, if we set it right, either approach aligns tiles and data nicely.

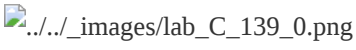
Web tiles can be integrated with other features of maps in a similar way as we have seen above. So, for example, we can change the size of the map, and remove the axis:

```
f, ax = plt.subplots(1, figsize=(9, 9))
lsoas_wm = lsoas.to_crs(epsg=3857)
lsoas_wm.plot(alpha=0.5, ax=ax)
cx.add_basemap(ax)
ax.set_axis_off()
```



Now, `contextily` offers a lot of options in terms of the sources and providers you can use to create your basemaps. For example, we can use satellite imagery instead:

```
f, ax = plt.subplots(1, figsize=(9, 9))
lsoas_wm = lsoas.to_crs(epsg=3857)
lsoas_wm.plot(facecolor='none', edgecolor='red', ax=ax)
cx.add_basemap(ax, url=cx.providers.Esri.WorldImagery)
ax.set_axis_off()
```



Have a look at this Twitter thread to get some further ideas on providers:

```
from IPython.display import HTML
tweet = """
<blockquote class="twitter-tweet" data-lang="en"><p lang="et" dir="ltr">Terrain maps <a
href="https://t.co/VtN9bGG5Mt">pic.twitter.com/VtN9bGG5Mt</a></p>&mdash; Dani Arribas-
Bel (@darribas) <a href="https://twitter.com/darribas/status/1157297596689539072?
ref_src=twsrc%5Etfw">August 2, 2019</a></blockquote>
<script async src="https://platform.twitter.com/widgets.js" charset="utf-8"></script>
"""
HTML(tweet)
```

Dani Arribas-Bel @darribas · Aug 2, 2019

Replying to @darribas

Get world imagery and make a map of a place in two lines:

Dani Arribas-Bel
@darribas

Terrain maps

[Extension II] Advanced GIS operations

- 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

---

This notebook, as well as the entire set of materials, code, and data included in this course are available as an open Github repository available at:
https://github.com/darribas/gds19

Do-It-Yourself

By Dani Arribas-Bel