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.

Note

A PDF version of this course is available for download here

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.

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.

[ABGLopezVM19]
Daniel Arribas-Bel, M-À Garcia-López, and Elisabet Viladecans-Marsal. Building (s and) cities: delineating urban areas with a machine learning algorithm. *Journal of Urban Economics*, pages 103217, 2019.

[Bre15]
Cynthia Brewer. *Designing better Maps: A Guide for GIS users*. ESRI press, 2015.

[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

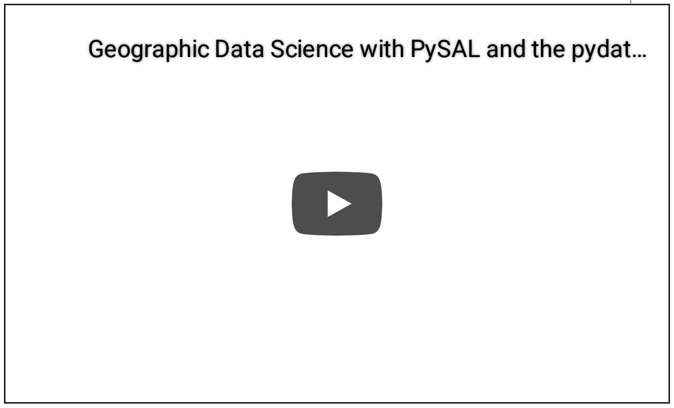
This course

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]

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

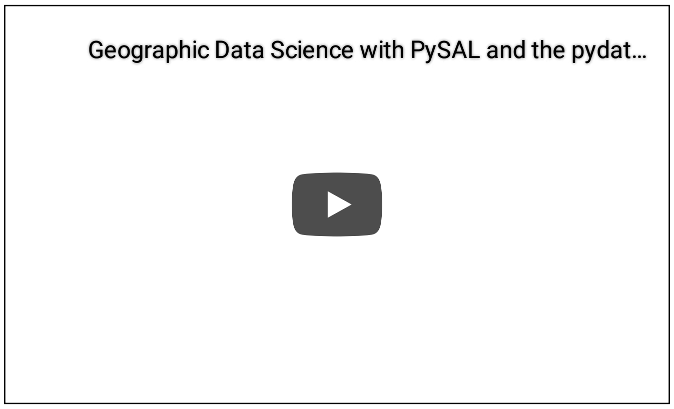
In this first

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

Downloading data for this course

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.

Inspect

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()
```

	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

```
db.tail()
```

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

Or 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
```

Summarise

Or 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%	
Europe	298.0	1462.382550	248.673290	731.0	1331.25	1446.0	1
Africa	298.0	29.818792	51.606065	0.0	7.00	14.0	
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	
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
```

Create new columns

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	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

A different spin on this is assigning new values: we can 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	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

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	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

Delete columns

Permanently deleting variables is also within reach of one command:

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

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

Index-based queries

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
```

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

Condition-based queries

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
```

	Europe	**Africa**	**Middle East and Asia**	**The Americas and the Caribbean**	**Antarctica and Oceania**	**Tota**
GeographyCode						
E01006513	2225	61	595	53	7	294
E01006747	2551	163	812	24	2	355
E01006751	1843	139	568	21	1	257

… *areas where there are no more than 750 Europeans*:

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

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01033757	731	39	223	29	3	102

… *areas with exactly ten person from Antarctica and Oceania*:

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

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006679	1353	484	354	31	10	223

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
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188

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	Tota
GeographyCode						
E01033750	1235	53	129	26	5	144
E01033752	1024	19	114	33	6	119
E01033754	1262	37	112	32	9	145
E01033756	886	31	221	42	5	118
E01033757	731	39	223	29	3	102
E01033761	1138	52	138	33	11	137

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	Tota
GeographyCode						
E01006747	2551	163	812	24	2	355
E01006513	2225	61	595	53	7	294
E01006751	1843	139	568	21	1	257
E01006524	2235	36	125	24	11	243
E01006787	2187	53	75	13	2	233

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)
```

../../_images/lab_B_47_0.png

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)
```

../../_images/lab_B_49_0.png

- 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)
```

../../_images/lab_B_51_0.png

- 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)
```

../../_images/lab_B_53_1.png

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()
```

../../_images/lab_B_55_0.png

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))
```

../../_images/lab_B_57_0.png

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:

```python
# 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:

```python
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.

```python
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

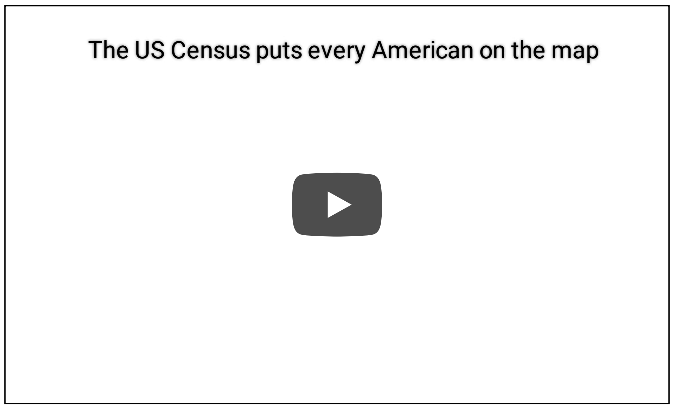
Concepts

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:

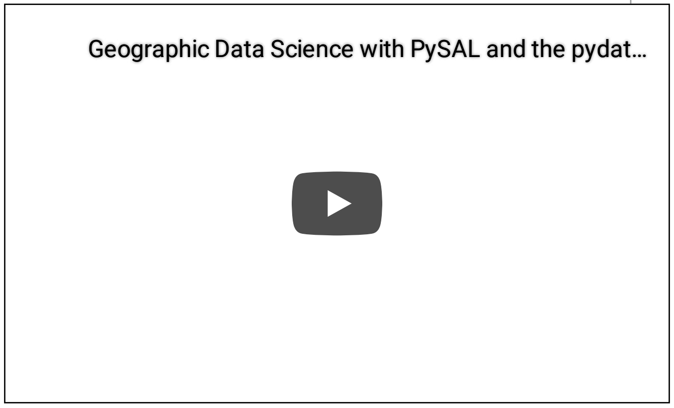


Then go on to the following clip, which will help you put the Census Bureau’s view in perspective:

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

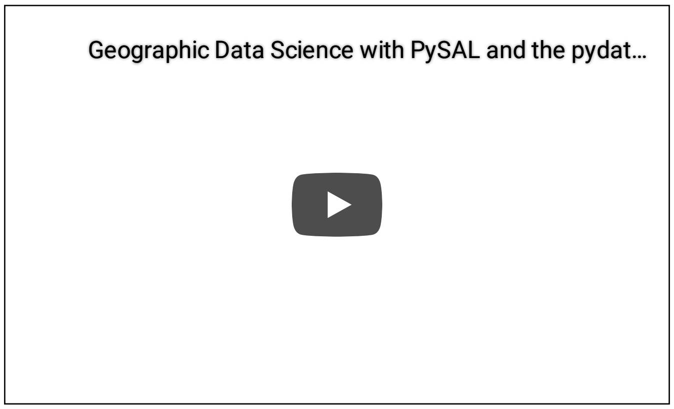


New forms of (geo) data

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

```
%matplotlib inline

import geopandas
import osmnx
import contextily as cx
from keplergl import KeplerGl
import matplotlib.pyplot as plt
```

In this lab, we will learn how to load, manipulate and visualize spatial data. In some senses, spatial data 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.

Datasets

To learn these concepts, we will be playing with three main datasets. Same as in the previous block, these datasets can be loaded dynamically from the web, or you can download them manually, keep a copy on your computer, and load them from there.

> **Important**
>
> Make sure you are connected to the internet when you run these cells as they need to access data hosted online

Cities

First we will use a polygon geography. We will use an open dataset that contains the boundaries of Spanish cities. We can read it into an object named `cities` by:

```
cities = geopandas.read_file("https://ndownloader.figshare.com/files/20232174")
```

Note

This dataset is derived from [ABGLopezVM19], which proposes a machine learning algorithm to delineate city boundaries from building footprints.

Streets

In addition to polygons, we will play with a line layer. For that, we are going to use a subset of street network from the Spanish city of Zaragoza.

Note

This dataset froms of a project called "Las calles de las mujeres", a community-driven initiative exploring the extent to which streets are named after women.

Check out more about the project, including an interactive map at:

https://geochicasosm.github.

The data is available on the following web address:

```
url = ("https://github.com/geochicasosm/lascallesdelasmujeres"\
       "/raw/master/data/zaragoza/final_tile.geojson")
url
```

```
'https://github.com/geochicasosm/lascallesdelasmujeres/raw/master/data/zaragoza/fi
nal_tile.geojson'
```

And you can read it into an object called `streets` with:

```
streets = geopandas.read_file(url)
```

Bars

The final dataset we will rely on is a set of points demarcating the location of bars in Zaragoza. To obtain it, we will use `osmnx`, a Python library that allows us to query OpenStreetMap. Note that we use the method `pois_from_place`, which queries for points of interest (POIs, or `pois`) in a particular place (Zaragoza in this case). In addition, we can specify a set of tags to delimit the query. We use this to ask *only* for amenities of the type "bar":

```
pois = osmnx.pois.pois_from_place("Zaragoza, Spain",
                                  tags={"amenity": "bar"}
                                 )
```

You do not need to know at this point what happens behind the scenes when we run `pois_from_place` but, if you are curious, we are making a query to OpenStreetMap (almost as if you typed "bars in Zaragoza, Spain" within Google Maps) and getting the response as a table of data, instead of as a website with an interactive map. Pretty cool, huh?

Inspecting 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 the same functionality that `pandas` does, plus a wide range of spatial extensions that make manipulation and general "munging" of spatial data 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 Desktop GIS packages like QGIS permit. 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

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

```
cities.head()
```

	city_id	n_buildings	geometry
0	ci000	2348	POLYGON ((385390.071 4202949.446, 384488.697 4...
1	ci001	2741	POLYGON ((214893.033 4579137.558, 215258.185 4...
2	ci002	5472	POLYGON ((690674.281 4182188.538, 691047.526 4...
3	ci003	14608	POLYGON ((513378.282 4072327.639, 513408.853 4...
4	ci004	2324	POLYGON ((206989.081 4129478.031, 207275.702 4...

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

```
cities.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_21_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:

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

../../_images/lab_C_24_0.svg

Lines

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:

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

../../_images/lab_C_27_0.svg

A quick plot is similarly generated by:

```
streets.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_29_1.png

Again, this is not the prettiest way to display the streets 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.

Points

Points take a similar approach for quick plotting:

```
pois.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_33_1.png

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):

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

```
<AxesSubplot:>
```

../../_images/lab_C_38_1.png

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
cities.plot(ax=ax)
# Remove axis frames
ax.set_axis_off()
# Display
plt.show()
```

../../_images/lab_C_41_0.png

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
streets.plot(ax=ax)
# Add figure title
f.suptitle("Streets in Zaragoza")
# Display
plt.show()
```

../../_images/lab_C_45_0.png

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
cities.plot(ax=ax)
# Display
plt.show()
```

../../_images/lab_C_48_0.png

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 thicker* and *black*, and then we will work our way through the different steps:

```
# Setup figure and axis
f, ax = plt.subplots(1, figsize=(12, 12))
# Add layer of polygons on the axis, set fill color (`facecolor`) and boundary
# color (`edgecolor`)
cities.plot(linewidth=1,
            facecolor='red',
            edgecolor='black',
            ax=ax
           )
```

```
<AxesSubplot:>
```

../../_images/lab_C_51_1.png

Note how the lines are thicker. In addition, all the polygons are colored in the same (default) color, light red. However, because the lines are thicker, we can only see the polygon filling for those cities with an area large enough.

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: `linewidth` for the width of the line; `facecolor`, to control the color each polygon is filled with; and `edgecolor`, to control the color of the boundary.

This approach works very similarly with other geometries, such as lines. For example, if we wanted to plot the streets 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`
streets.plot(linewidth=2, color='red', ax=ax)
```

```
<AxesSubplot:>
```

../../_images/lab_C_53_1.png

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:

```
cities.crs
```

```
<Projected CRS: EPSG:25830>
Name: ETRS89 / UTM zone 30N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe - 6°W to 0°W and ETRS89 by country
- bounds: (-6.0, 35.26, 0.0, 80.53)
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: European Terrestrial Reference System 1989
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

As we can see, there is information stored about the reference system: it is using the standard Spanish projection, which is expressed in meters. 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
cities.to_crs(epsg=4326).plot()
# Set equal axis
lims = plt.axis('equal')
```

../../_images/lab_C_59_0.png

The shape of the polygons is slightly different. Furthermore, note how the *scale* in which they are plotted differs.

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.

For this section, let’s select only Zaragoza from the `streets` table and convert it to lat/lon so it’s aligned with the streets and POIs layers:

```
zgz = cities.loc[[112], :].to_crs(epsg=4326)
zgz
```

	city_id	n_buildings	geometry
112	ci122	23604	POLYGON ((-0.93057 41.60615, -0.93092 41.60622...

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's plot the boundary of Zaragoza and its bars:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
```

```
<AxesSubplot:>
```

../../_images/lab_C_65_1.png

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:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
# Save figure to a PNG file
plt.savefig('zaragoza_bars.png')
```

../../_images/lab_C_68_0.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 quality image, we could use 500:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
# Save figure to a PNG file
plt.savefig('zaragoza_bars.png', dpi=500)
```

../../_images/lab_C_70_0.png

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.

GeoDataFrames come with a whole range of traditional GIS operations built-in. Here we will run through a small subset of them that contains some of the most commonly used ones.

Area calculation

One of the spatial aspects we often need from polygons is their area. “How big is it?” is a question that always haunts us when we think of countries, regions, or cities. To obtain area measurements, first make sure you GeoDataFrame is projected. If that is the case, you can calculate areas as follows:

```
city_areas = cities.area
city_areas.head()
```

```
0    8.449666e+06
1    9.121270e+06
2    1.322653e+07
3    6.808121e+07
4    1.072284e+07
dtype: float64
```

This indicates that the area of the first city in our table takes up 8,450,000 squared metres. If we wanted to convert into squared kilometres, we can divide by 1,000,000:

```
areas_in_sqkm = city_areas / 1000000
areas_in_sqkm.head()
```

```
0     8.449666
1     9.121270
2    13.226528
3    68.081212
4    10.722843
dtype: float64
```

Length

Similarly, an equally common question with lines is their length. Also similarly, their computation is relatively straightforward in Python, provided that our data are projected. Here we will perform the projection (to_crs) and the calculation of the length at the same time:

```
street_length = streets.to_crs(epsg=25830).length
street_length.head()
```

```
0     37.338828
1    104.510732
2    365.969719
3     97.101436
4     94.002218
dtype: float64
```

Since the CRS we use (`EPSG:25830`) is expressed in metres, we can tell the first street segment is about 37m.

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

```
0    POINT (386147.759 4204605.994)
1    POINT (216296.159 4579397.331)
2    POINT (688901.588 4180201.774)
3    POINT (518262.028 4069898.674)
4    POINT (206940.936 4127361.966)
dtype: geometry
```

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()
```

```
<AxesSubplot:>
```

../../_images/lab_C_84_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.

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 = cities.loc[112, "geometry"]
pt1 = cents[0]
pt2 = cents[112]
```

And we can perform the checks as follows:

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

```
False
```

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

```
True
```

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.

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.

For this example, we will use the bars table, but will project it to the same CRS as `cities`, so it is expressed in metres:

```
pois_projected = pois.to_crs(cities.crs)
pois_projected.crs
```

```
<Projected CRS: EPSG:25830>
Name: ETRS89 / UTM zone 30N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe - 6°W to 0°W and ETRS89 by country
- bounds: (-6.0, 35.26, 0.0, 80.53)
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: European Terrestrial Reference System 1989
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

To create a buffer using `geopandas`, simply call the `buffer` method, passing in the radious. For example, to draw a 500m. buffer around every bar in Zaragoza:

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

```
603129991    POLYGON ((676071.164 4612191.116, 676068.756 4...
673807647    POLYGON ((675740.197 4612333.309, 675737.789 4...
759470843    POLYGON ((676083.551 4614558.818, 676081.143 4...
765953540    POLYGON ((675653.012 4611997.279, 675650.604 4...
772973291    POLYGON ((675616.277 4614955.710, 675613.870 4...
dtype: geometry
```

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`)]
pois_projected.plot(ax=ax, markersize=1, color='yellow')
```

```
<AxesSubplot:>
```

../../_images/lab_C_97_1.png

Adding base layers from web sources

Many single datasets lack context when displayed on their own. A common approach to alleviate this is to use 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.

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 = cities.plot(color="black")
cx.add_basemap(ax, crs=cities.crs);
```

../../_images/lab_C_99_0.png

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:

```
cities_wm = cities.to_crs(epsg=3857)
ax = cities_wm.plot(color="black")
cx.add_basemap(ax);
```

../../_images/lab_C_101_0.png

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. Let's use Zaragoza for this example:

```
f, ax = plt.subplots(1, figsize=(9, 9))
zgz.plot(alpha=0.25, ax=ax)
cx.add_basemap(ax, crs=zgz.crs)
ax.set_axis_off()
```

../../_images/lab_C_104_0.png

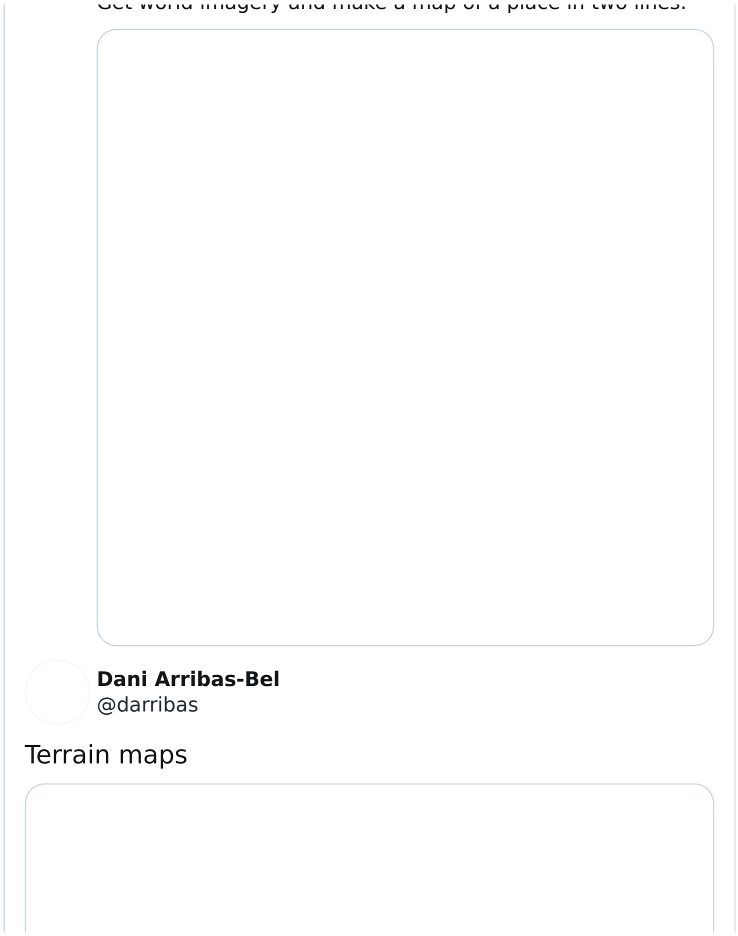
Now, `contextily` offers a lot of options in terms of the sources and providers you can use to create your basemaps. For example, we can use satellite imagery instead:

```
f, ax = plt.subplots(1, figsize=(9, 9))
zgz.plot(alpha=0.25, ax=ax)
cx.add_basemap(ax,
               crs=zgz.crs,
               source=cx.providers.Esri.WorldImagery
              )
ax.set_axis_off()
```

../../_images/lab_C_106_0.png

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

Dani Arribas-Bel @darribas · Aug 2, 2019
Replying to @darribas
Get world imagery and make a map of a place in two lines:


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

Dani Arribas-Bel
@darribas

Terrain maps

And consider checking out the documentation website for the package:

> https://contextily.readthedocs.io/en/latest/

Interactive maps

Everything we have seen so far relates to static maps. These are useful for publication, to include in reports or to print. However, modern web technologies afford much more flexibility to explore spatial data interactively.

One of the most prominents examples of interactive maps integrated with notebooks is [Kepler.GL](). Using its Jupyter integration, we can easily convert data (expressed in lon/lat) into interactive maps, and then play further to tweak its appearance. In this context, we will only show how you can take a `GeoDataFrame` into an interactive map in two lines of code:

```python
# NOTE: this may not render on the website

# Create a new map
m = KeplerGl()
# Append your data in lon/lat
m.add_data(streets, name="Streets")
# Display
m
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/geodataframe.py:852: UserWarning:
Geometry column does not contain geometry.
  warnings.warn("Geometry column does not contain geometry.")
```

Further resources

More advanced GIS operations are possible in `geopandas` and, in most cases, they are extensions of the same logic we have used in this document. If you are thinking about taking the next step from here, the following two operations (and the documentation provided) will give you the biggest "bang for the buck":

- 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

Do-It-Yourself

In this session, we will practice your skills in mapping with Python. Fire up a notebook you can edit interactively, and let's do this!

Data preparation

Polygons

For this section, you will have to push yourself out of the comfort zone when it comes to sourcing the data. As nice as it is to be able to pull a dataset directly from the web at the stroke of a url address, most real-world cases are not that straight forward. Instead, you usually have to download a dataset manually and store it locally on your computer before you can get to work.

We are going to use data from the Consumer Data Research Centre (CDRC) about Liverpool, in particular an extract from the Census. You can download a copy of the data at:

> **Important**
>
> You will need a username and password to download the data. Create it for free at:
>
> > https://data.cdrc.ac.uk/user/register

> Liverpool Census'11 Residential data pack download

Once you have the `.zip` file on your computer, right-click and “Extract all”. The resulting folder will contain all you need. For the sake of the example, let’s assume you place the resulting folder in the same location as the notebook you are using. If that is the case, you can load up a `GeoDataFrame` of Liverpool neighborhoods with:

```
import geopandas
liv =
geopandas.read_file("Census_Residential_Data_Pack_2011_E08000012/data/Census_Residenti
al_Data_Pack_2011/Local_Authority_Districts/E08000012/shapefiles/E08000012.shp")
```

Lines

For a line layer, we are going to use a different bit of `osmnx` functionality that will allow us to extract all the highways

```
bikepaths = osmnx.geocode_to_gdf("Liverpool, UK", )
```

```
bikepaths = osmnx.graph_from_place("Liverpool, UK", network_type="bike")
```

```
len(bikepaths)
```

```
23460
```

Points

For points, we will use an analogue of the POI layer we have used in the Lab: pubs in Liverpool, as recorded by OpenStreetMap.

Let’s import `osmnx`:

```
import osmnx
```

And make a similar query to retrieve the table:

```
pubs = osmnx.pois.pois_from_place("Liverpool, UK",
                                  tags={"amenity": "pub"}
                                 )
```

Tasks

Task I: *Tweak your map*

With those three layers, try to complete the following tasks:

- Make a map of the Liverpool neighborhoods that includes the following characteristics:
  - Features a title
  - Does not include axes frame
  - It has a figure size of 10 by 11
  - Polygons are all in color `"#525252"` and 50% transparent
  - Boundary lines (“edges”) have a width of 0.3 and are of color `"#B9EBE3"`
  - Includes a basemap with the Stamen watercolor theme

Note

Not all of the requirements above are not equally hard to achieve. If you can get some but not all of them, that's also great! The point is you learn something every time you try.

Task II: *Non-spatial manipulations*

For this one we will combine some of the ideas we learnt in the previous block with this one.

Focus on the LSOA `liv` layer and use it to do the following:

1. Calculate the area of each neighbourhood
2. Find the five smallest areas in the table. Create a new object (e.g. `smallest` with them only)
3. Create a multi-layer map of Liverpool where the five smallest areas are coloured in red, and the rest appear in black.

Task III: *The gender gap on the streets*

This one is a bit more advanced, so don't despair if you can't get it on your first try. It also relies on the streets dataset from the "Hands-on" section, so you will need to load it up on your own. Here're the questions for you to answer:

> *Which group accounts for longer total street length in Zaragoza: men or women? By how much*?

The suggestion is that you get to work right away. However, if this task seems too daunting, you can expand the tip below for a bit of help.

Tip

Concepts

This block is all about Geovisualisation and displaying statistical information on maps. We start with an introduction on *what* geovisualisation is; then we follow with the modifiable areal unit problem, a key concept to keep in mind when displaying statistical information spatially; and we wrap up with tips to make awesome choropleths, thematic maps. Each section contains a short clip and a set of slides, plus a few (optional) readings.

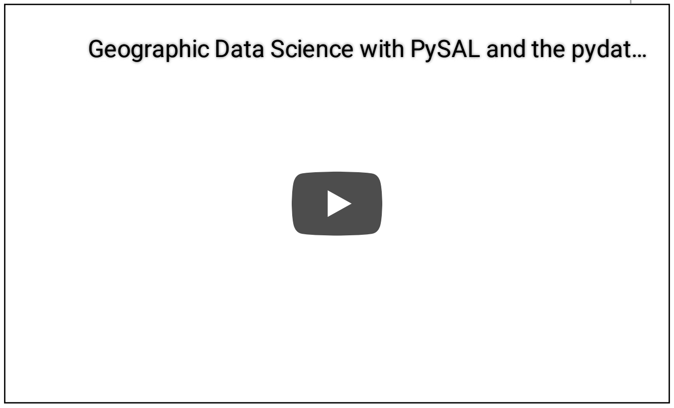
Geovisualisation

Geovisualisation is an area that underpins much what we will discuss in this course. Often, we will be presenting the results of more sophisticated analyses as maps. So getting the principles behind mapping right is critical. In this clip, we cover *what* is (geo)visualisation and why it is important.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



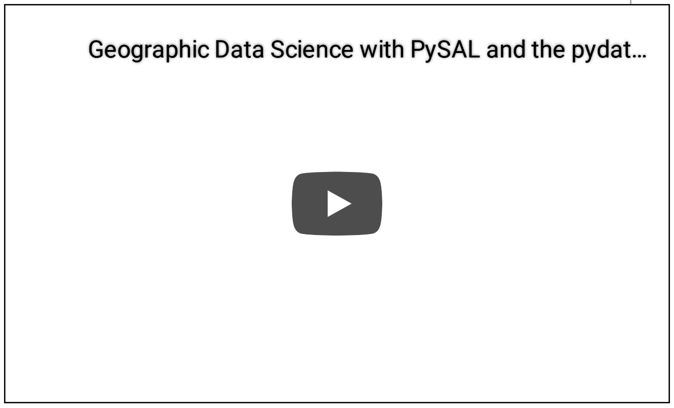
Geographical containers for data

This section tries to get you to think about the geographical containers we use to represent data in maps. By that, we mean the areas, delineations and aggregations we, implicitly or explicitly, incur in when mapping data. This is an important aspect, but Geographers have been aware of them for a long time, so we are standing on the shoulders of giants.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



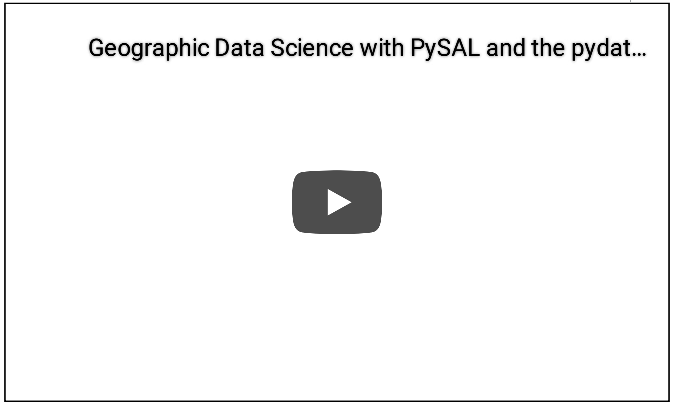
Choropleths

Choropleths are thematic maps and, these days, are everywhere. From elections, to economic inequality, to the distribution of population density, there's a choropleth for everyone. Although technically, it is easy to create choropleths, it is even easier to make *bad* choropleths. Fortunately, there are a few principles that we can follow to create effective choropleths. Get them all delivered right to the conform of your happy place in the following clip and slides!

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



Further readings

The clip above contains a compressed version of the key principles behind successful choropleths. For a more comprehensive coverage, please refer to:

- Choropleths chapter on the GDS book (in progress) [RABWng].
- Cynthia Brewer's "Designing Better Maps" [Bre15] covers several core aspects of building effective geovisualisations.

The chapter is available for free here

Hands-on

Choropleths in Python

Important

This is an adapted version, with a bit less content and detail, of the chapter on choropleth mapping by Rey, Arribas-Bel and Wolf (*in progress*) [RABWng]. Check out the full chapter, available for free at:

https://geographicdata.science/book/notebooks/05_choropleth.html

In this session, we will build on all we have learnt so far about loading and manipulating (spatial) data and apply it to one of the most commonly used forms of spatial analysis: choropleths. Remember these are maps that display the spatial distribution of a variable encoded in a color scheme, also called *palette*. Although there are many ways in which you can convert the values of a variable into a specific color, we will focus in this context only on a handful of them, in particular:

- Unique values
- Equal interval
- Quantiles

- Fisher-Jenks

Before all this mapping fun, let us get the importing of libraries and data loading out of the way:

```
%matplotlib inline

import geopandas
from pysal.lib import examples
import seaborn as sns
import pandas as pd
from pysal.viz import mapclassify
import numpy as np
import matplotlib.pyplot as plt
```

Data

To mirror the original chapter this section is based on, we will use the same dataset: the Mexico GDP per capita dataset, which we can access as a PySAL example dataset.

> **Note**
>
> You can read more about PySAL example datasets here

We can get a short explanation of the dataset through the `explain` method:

```
examples.explain("mexico")
```

```
mexico
======

Decennial per capita incomes of Mexican states 1940-2000
--------------------------------------------------------

* mexico.csv: attribute data. (n=32, k=13)
* mexico.gal: spatial weights in GAL format.
* mexicojoin.shp: Polygon shapefile. (n=32)

Data used in Rey, S.J. and M.L. Sastre Gutierrez. (2010) "Interregional inequality
dynamics in Mexico." Spatial Economic Analysis, 5: 277-298.
```

Now, to download it from its remote location, we can use `load_example`:

```
mx = examples.load_example("mexico")
```

This will download the data and place it on your home directory. We can inspect the directory where it is stored:

```
mx.get_file_list()
```

```
['/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.shp',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexico.csv',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/README.md',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.dbf',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.prj',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexico.gal',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.shx']
```

For this section, we will read the ESRI shapefile, which we can do by pointing `geopandas.read_file` to the `.shp` file. The utility function `get_path` makes it a bit easier for us:

```
db = geopandas.read_file(examples.get_path("mexicojoin.shp"))
```

And, from now on, `db` is a table as we are used to so far in this course:

```
db.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 35 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   POLY_ID     32 non-null     int64
 1   AREA        32 non-null     float64
 2   CODE        32 non-null     object
 3   NAME        32 non-null     object
 4   PERIMETER   32 non-null     float64
 5   ACRES       32 non-null     float64
 6   HECTARES    32 non-null     float64
 7   PCGDP1940   32 non-null     float64
 8   PCGDP1950   32 non-null     float64
 9   PCGDP1960   32 non-null     float64
 10  PCGDP1970   32 non-null     float64
 11  PCGDP1980   32 non-null     float64
 12  PCGDP1990   32 non-null     float64
 13  PCGDP2000   32 non-null     float64
 14  HANSON03    32 non-null     float64
 15  HANSON98    32 non-null     float64
 16  ESQUIVEL99  32 non-null     float64
 17  INEGI       32 non-null     float64
 18  INEGI2      32 non-null     float64
 19  MAXP        32 non-null     float64
 20  GR4000      32 non-null     float64
 21  GR5000      32 non-null     float64
 22  GR6000      32 non-null     float64
 23  GR7000      32 non-null     float64
 24  GR8000      32 non-null     float64
 25  GR9000      32 non-null     float64
 26  LPCGDP40    32 non-null     float64
 27  LPCGDP50    32 non-null     float64
 28  LPCGDP60    32 non-null     float64
 29  LPCGDP70    32 non-null     float64
 30  LPCGDP80    32 non-null     float64
 31  LPCGDP90    32 non-null     float64
 32  LPCGDP00    32 non-null     float64
 33  TEST        32 non-null     float64
 34  geometry    32 non-null     geometry
dtypes: float64(31), geometry(1), int64(1), object(2)
memory usage: 8.9+ KB
```

The data however does not include a CRS:

```
db.crs
```

To be able to add baselayers, we need to specify one. Looking at the details and the original reference, we find the data are expressed in longitude and latitude, so the CRS we can use is `EPSG:4326`. Let’s add it to `db`:

```
db.crs = "EPSG:4326"
db.crs
```

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

Now we are fully ready to map!

Choropleths

Unique values

A choropleth for categorical variables simply assigns a different color to every potential value in the series. The main requirement in this case is then for the color scheme to reflect the fact that different values are not ordered or follow a particular scale.

In Python, creating categorical choropleths is possible with one line of code. To demonstrate this, we can plot the Mexican states and the region each belongs to based on the Mexican Statistics Institute (coded in our table as the `INEGI` variable):

```
db.plot(column="INEGI",
        categorical=True,
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_21_1.png

Let us stop for a second in a few crucial aspects:

- Note how we are using the same approach as for basic maps, the command `plot`, but we now need to add the argument `column` to specify which column in particular is to be represented.
- Since the variable is categorical we need to make that explicit by setting the argument `categorical` to `True`.
- As an optional argument, we can set `legend` to `True` and the resulting figure will include a legend with the names of all the values in the map.
- Unless we specify a different colormap, the selected one respects the categorical nature of the data by not implying a gradient or scale but a qualitative structure.

Equal interval

If, instead of categorical variables, we want to display the geographical distribution of a continuous phenomenon, we need to select a way to encode each value into a color. One potential solution is applying what is usually called "equal intervals". The intuition of this method is to split the *range* of the distribution, the difference between the minimum and maximum value, into equally large segments and to assign a different color to each of them according to a palette that reflects the fact that values are ordered.

Creating the choropleth is relatively straightforward in Python. For example, to create an equal interval on the GDP per capita in 2000 (`PCGDP2000`), we can run a similar command as above:

```
db.plot(column="PCGDP2000",
        scheme="equal_interval",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_25_1.png

Pay attention to the key differences:

- Instead of specifyig `categorical` as `True`, we replace it by the argument `scheme`, which we will use for all choropleths that require a continuous classification scheme. In this case, we set it to `equal_interval`.
- As above, we set the number of colors to 7. Note that we need not pass the bins we calculated above, the plotting method does it itself under the hood for us.
- As optional arguments, we can change the colormap to a yellow to green gradient, which is one of the recommended ones by ColorBrewer for a sequential palette.

Now, let's dig a bit deeper into the classification, and how exactly we are encoding values into colors. Each segment, also called bins or buckets, can also be calculated using the library `mapclassify` from the `PySAL` family:

```
classi = mapclassify.EqualInterval(db["PCGDP2000"], k=7)
classi
```

```
EqualInterval

      Interval         Count
----------------------------
[ 8684.00, 15207.57] |    10
(15207.57, 21731.14] |    10
(21731.14, 28254.71] |     5
(28254.71, 34778.29] |     4
(34778.29, 41301.86] |     2
(41301.86, 47825.43] |     0
(47825.43, 54349.00] |     1
```

The only additional argument to pass to `Equal_Interval`, other than the actual variable we would like to classify is the number of segments we want to create, `k`, which we are arbitrarily setting to seven in this case. This will be the number of colors that will be plotted on the map so, although having several can give more detail, at some point the marginal value of an additional one is fairly limited, given the ability of the brain to tell any differences.

Once we have classified the variable, we can check the actual break points where values stop being in one class and become part of the next one:

```
classi.bins
```

```
array([15207.57142857, 21731.14285714, 28254.71428571, 34778.28571429,
       41301.85714286, 47825.42857143, 54349.        ])
```

The array of breaking points above implies that any value in the variable below 15,207.57 will get the first color in the gradient when mapped, values between 15,207.57 and 21,731.14 the next one, and so on.

The key characteristic in equal interval maps is that the bins are allocated based on the magnitude on the values, irrespective of how many obervations fall into each bin as a result of it. In highly skewed distributions, this can result in bins with a large number of observations, while others only have a handful of outliers. This can be seen in the summary table printed out above, where ten states are in the first group, but only one of them belong to the one with highest values. This can also be represented visually with a kernel density plot where the break points are included as well:

../../_images/lab_D_31_0.png

Technically speaking, the figure is created by overlaying a KDE plot with vertical bars for each of the break points. This makes much more explicit the issue highlighed by which the first bin contains a large amount of observations while the one with top values only encompasses a handful of them.

Quantiles

One solution to obtain a more balanced classification scheme is using quantiles. This, by definition, assigns the same amount of values to each bin: the entire series is laid out in order and break points are assigned in a way that leaves exactly the same amount of observations between each of them. This "observation-based" approach contrasts with the "value-based" method of equal intervals and, although it can obscure the magnitude of extreme values, it can be more informative in cases with skewed distributions.

The code required to create the choropleth mirrors that needed above for equal intervals:

```
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_35_1.png

Note how, in this case, the amount of polygons in each color is by definition much more balanced (almost equal in fact, except for rounding differences). This obscures outlier values, which get blurred by significantly smaller values in the same group, but allows to get more detail in the "most populated" part of the distribution, where instead of only white polygons, we can now discern more variability.

To get further insight into the quantile classification, let's calculate it with `mapclassify`:

```
classi = mapclassify.Quantiles(db["PCGDP2000"], k=7)
classi
```

```
Quantiles

      Interval         Count
----------------------------
[ 8684.00, 11752.00] |     5
(11752.00, 13215.43] |     4
(13215.43, 15996.29] |     5
(15996.29, 20447.14] |     4
(20447.14, 26109.57] |     5
(26109.57, 30357.86] |     4
(30357.86, 54349.00] |     5
```

And, similarly, the bins can also be inspected:

```
classi.bins
```

```
array([11752.        , 13215.42857143, 15996.28571429, 20447.14285714,
       26109.57142857, 30357.85714286, 54349.        ])
```

The visualization of the distribution can be generated in a similar way as well:

../../_images/lab_D_42_0.png

Fisher-Jenks

Equal interval and quantiles are only two examples of very many classification schemes to encode values into colors. Although not all of them are integrated into `geopandas`, `PySAL` includes several other classification schemes (for a detailed list, have a look at this link). As an example of a more sophisticated one, let us create a Fisher-Jenks choropleth:

```
db.plot(column="PCGDP2000",
        scheme="fisher_jenks",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_45_1.png

The same classification can be obtained with a similar approach as before:

```
classi = mapclassify.FisherJenks(db["PCGDP2000"], k=7)
classi
```

```
FisherJenks

      Interval         Count
----------------------------
[ 8684.00, 13360.00] |    10
(13360.00, 18170.00] |     8
(18170.00, 24068.00] |     4
(24068.00, 28460.00] |     4
(28460.00, 33442.00] |     3
(33442.00, 38672.00] |     2
(38672.00, 54349.00] |     1
```

This methodology aims at minimizing the variance *within* each bin while maximizing that *between* different classes.

```
classi.bins
```

```
array([13360., 18170., 24068., 28460., 33442., 38672., 54349.])
```

Graphically, we can see how the break points are not equally spaced but are adapting to obtain an optimal grouping of observations:

../../_images/lab_D_51_0.png

For example, the bin with highest values covers a much wider span that the one with lowest, because there are fewer states in that value range.

Zooming into the map

Zoom into full map

A general map of an entire region, or urban area, can sometimes obscure local patterns because they happen at a much smaller scale that cannot be perceived in the global view. One way to solve this is by providing a focus of a smaller part of the map in a separate figure. Although there are many ways to do this in Python, the most straightforward one is to reset the limits of the axes to center them in the area of interest.

As an example, let us consider the quantile map produced above:

```
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=False
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_56_1.png

If we want to focus around the capital, Mexico DF, the first step involves realising that such area of the map (the little dark green polygon in the SE centre of the map), falls within the coordinates of -102W/-97W, and 18N/21N, roughly speaking. To display a zoom map into that area, we can do as follows:

```
# Setup the figure
f, ax = plt.subplots(1)
# Draw the choropleth
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=False,
        ax=ax
       )
# Redimensionate X and Y axes to desired bounds
ax.set_ylim(18, 21)
ax.set_xlim(-102, -97)
```

```
(-102.0, -97.0)
```

../../_images/lab_D_58_1.png

Partial map

The approach above is straightforward, but not necessarily the most efficient one: not that, to generate a map of a potentially very small area, we effectively draw the entire (potentially very large) map, and discard everything except the section we want. This is not straightforward to notice at first sight, but what Python is doing in the code cell above is plottin the entire `db` object, and only then focusing the figure on the X and Y ranges specified in `set_xlim`/`set_ylim`.

Sometimes, this is required. For example, if we want to retain the same coloring used for the national map, but focus on the region around Mexico DF, this approach is the easiest one.

However, sometimes, we only need to plot the *geographical features* within a given range, and we either don't need to keep the national coloring (e.g. we are using a single color), or we want a classification performed *only* with the features in the region.

For these cases, it is computationally more efficient to select the data we want to plot first, and then display them through `plot`. For this, we can rely on the `cx` operator:

```
subset = db.cx[-102:-97, 18:21]
subset.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_D_60_1.png

We query the range of spatial coordinates similarly to how we query indices with `loc`. Note however the result includes full geographic features, and hence the polygons with at least some area within the range are included fully. This results in a larger range than originally specified.

This approach is a "spatial slice". If you remember when we saw non-spatial slices (enabled by the `loc` operator), this is a similar approach but our selection criteria, instead of subsetting by indices of the table, are based on the spatial coordinates of the data represented in the table.

Since the result is a `GeoDataFrame` itself, we can create a choropleth that is based only on the data in the subset:

```
subset.plot(column="PCGDP2000",
            scheme="quantiles",
            k=7,
            cmap="YlGn",
            legend=False
           )
```

```
<AxesSubplot:>
```

../../_images/lab_D_62_1.png

Do-It-Yourself

Let’s make a bunch of choropleths! In this section, you will practice the concepts and code we have learnt in this block. Happy hacking!

Data preparation

> **Note**
>
> The AHAH dataset was invented by a University of Liverpool team. If you want to find out more about the background and details of the project, have a look at the information page at the CDRC website.

We are going to use the Access to Healthy Assets and Hazards (AHAH) index. This is a score that ranks LSOAs (the same polygons we used in block C) by the proximity to features of the environment that are considered positive for health (assets) and negative (hazards). The resulting number gives us a sense of how “unhealthy” the environment of the LSOA is. The higher the score, the less healthy the area is assessed to be.

To download the Liverpool AHAH pack, please go over to:

> **Important**
>
> You will need a username and password to download the data. Create it for free at:
>
> > https://data.cdrc.ac.uk/user/register

> Liverpool AHAH GeoData pack

Once you have the `.zip` file on your computer, right-click and “Extract all”. The resulting folder will contain all you need. For the sake of the example, let’s assume you place the resulting folder in the same location as the notebook you are using. If that is the case, you can load up a `GeoDataFrame` of Liverpool neighborhoods with:

```
import geopandas
lsoas =
geopandas.read_file("Access_to_Healthy_Assets_and_Hazards_AHAH_E08000012/data/Access_to_H
ealthy_Assets_and_Hazards_AHAH/Local_Authority_Districts/E08000012/shapefiles/E08000012.s
hp")
```

Now, this gets us the geometries of the LSOAs, but not the AHAH data. For that, we need to read in the data and join it to `ahah`. Assuming the same location of the data as above, we can do as follows:

```
import pandas
ahah_data =
pandas.read_csv("Access_to_Healthy_Assets_and_Hazards_AHAH_E08000012/data/Access_to_Healt
hy_Assets_and_Hazards_AHAH/Local_Authority_Districts/E0800001/tables/E08000012.csv")
```

To read the data, and as follows to join it:

```
ahah = lsoas.join(ahah_data.set_index("lsoa11cd"), on="lsoa11cd")
```

Now we’re ready to map using the `ahah` object.

Tasks

Task I: AHAH choropleths

Create the following choropleths and, where possible, complement them with a figure that displays the distribution of values using a KDE:

- Equal Interval with five classes
- Quantiles with five classes
- Fisher-Jenks with five classes
- Unique Values with the following setup:
  - Split the LSOAs in two classes: above and below the average AHAH score
  - Assign a qualitative label (`above` or `below`) to each LSOA
  - Create a unique value map for the labels you have just created

Task II: Zoom maps

Generate the following maps:

- Zoom of the city centre of Liverpool with he same color for every LSO
- Quantile map of AHAH for all of Liverpool, zoomed into north of the city centre
- Zoom to north of the city centre with a quantile map of AHAH for the section only

---