Contents

ENVS363/563

Geographic Data Science

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

Contact

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

Note

A PDF version of this course is available for download here

Citation

JOSE 10.21105/jose.00042


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

Overview

Aims

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

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

Learning outcomes

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

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

Feedback strategy

The student will receive feedback through the following channels:

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

Key texts and learning resources

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

> https://darribas.org/gds_course

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

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

Syllabus

Week 1: Introduction

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

Week 2: Modern Computational Environments

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

Week 3: Spatial Data

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

Week 4: (Geo)Visualization + Choropleths

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

Week 5: Spatial Weights

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

Week 6: ESDA

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

Week 7: Clustering

- Lecture: Clustering.
- Tutorial: Geodemographic analysis.

Week 8: Point Data

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

Week 9

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

Week 10: (Spatial) causal inference

- Lecture: Spatial causal inference.

- Tutorial: Assignment Clinic.

Week 11: Geographic Data Science in Action

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

ASSIGNMENT due on Thursday, December 5th-2019.

Bibliography

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

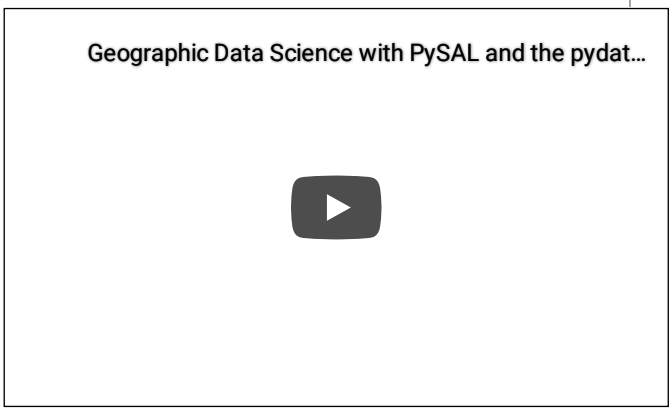
This course

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]



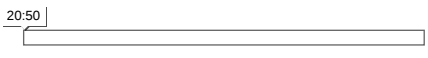
What is *Geographic Data Science*?

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

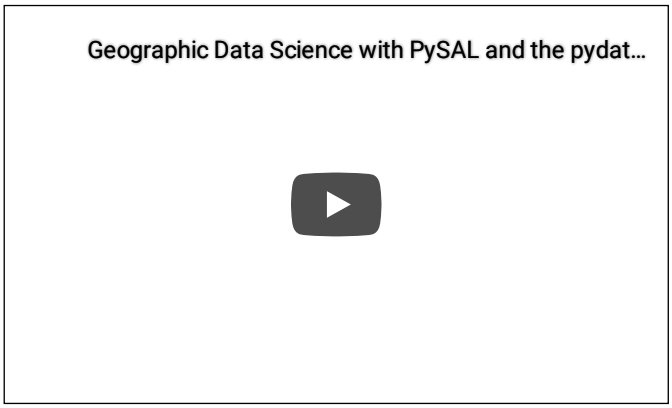
In this first

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

Files

Do-It-Yourself

To do:

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

Concepts

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

Open Science

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

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

Modern Scientific Tools

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

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

Hands-on

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

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

Data munging

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

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

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

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

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

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

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

Dataset

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

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

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

Important

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

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

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

Alternative

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

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

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

Data, sliced and diced

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

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

```
db.head()
```

	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

- Getting an overview of the table:

```
db.info()
```

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

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

```
db.describe()
```

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

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

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

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

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

- Equally, common descriptive statistics are also available:

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

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

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

```
731
```

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

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

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

```
457.8842648530303
```

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

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

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

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

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

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

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

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

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

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

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

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

And we can modify specific values too:

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

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

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

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

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

- Index-based querying.

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

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

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

- Querying based on conditions.

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

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

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

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

… areas where there are no more than 750 Europeans:

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

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

… areas with exactly ten person from Antarctica and Oceania:

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

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

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

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

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

- Combining queries.

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

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

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

- Sorting.

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

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

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

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

Visual exploration

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

- Histograms.

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

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

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

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

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

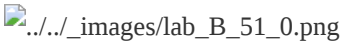
An additional feature to visualize the density of values is called `rug`, and adds a little tick for each value on the horizontal axis:

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

- Kernel Density Plots

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

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



- Line and bar plots

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

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

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

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

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

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

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

Un/tidy data

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

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

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

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

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

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

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

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

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

```
db.head()
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
type(tidy_subgroups)
```

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

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

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

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

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

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

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

Now our table is fully tidied up!

Grouping, transforming, aggregating

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

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

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

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

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

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

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

```
pop_grouped.sum()
```

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

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

```
pop_grouped.describe()
```

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

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

Additional lab materials

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

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

Do-It-Yourself

```
import pandas
```

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

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

Data preparation

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

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

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

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

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

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

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

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

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

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

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

Now we are good to go!

```
db.head()
```

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

Tasks

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

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

Try to answer the following questions:

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

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

By Dani Arribas-Bel