

lab_01.i

September 13, 2017

1 Geographic Data Science - Lab 01, Part I

Dani Arribas-Bel

2 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:

```
In [1]: # 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 easily and beautifully plot
```

2.1 Dataset

We will be exploring some of the characteristics of the population 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 that is small enough to create variation within cities, but large enough also to

preserve privacy. For that reason, many data products (Census, deprivation indices, etc.) use LSOAs as one of their main geographies.

To read a “comma separated values” (.csv) file, we can run:

```
In [2]: # Important! You need to specify the path to the data in *your* machine
        # If you have placed the data folder in the same directory as this notebook,
        # you would do:
        # f = 'liv_pop.csv'
        f = 'data/liv_pop.csv'                                # Path to file containing the table
        db = pd.read_csv(f, index_col='GeographyCode')         # Read the table in
```

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 simple form, all that is required is to pass the path to the file we want to read, which in this case we have created by concatenating two strings. We can see the full path we have used:

```
In [3]: f
```

```
Out[3]: 'data/liv_pop.csv'
```

- 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 written, just replace `read` by `to`! For example, `read_csv` reads in, `to_csv` writes out). A full list of formats supported may be found [here](#).

2.2 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:

```
In [4]: db.head()
```

```
Out[4]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106		840
E01006513	2225	61		595
E01006514	1786	63		193
E01006515	974	29		185
E01006518	1531	69		73

	The Americas and the Caribbean	Antarctica and Oceania
GeographyCode		
E01006512		24
E01006513		53
E01006514		61
E01006515		18
E01006518		19

```
In [5]: db.tail()
```

```

Out[5]:
      Europe  Africa  Middle East and Asia \
GeographyCode
E01033764    2106    32                    49
E01033765    1277    21                    33
E01033766    1028    12                    20
E01033767    1003    29                    29
E01033768    1016    69                   111

      The Americas and the Caribbean  Antarctica and Oceania
GeographyCode
E01033764                        15                        0
E01033765                        17                        3
E01033766                         8                        7
E01033767                         5                        1
E01033768                        21                        6

```

- Getting an overview of the table:

```

In [6]: db.info()

<class 'pandas.core.frame.DataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 5 columns):
Europe                298 non-null int64
Africa                298 non-null int64
Middle East and Asia  298 non-null int64
The Americas and the Caribbean  298 non-null int64
Antarctica and Oceania  298 non-null int64
dtypes: int64(5)
memory usage: 14.0+ KB

```

- Getting an overview of the values of the table:

```

In [7]: db.describe()

Out[7]:
      Europe  Africa  Middle East and Asia \
count    298.000000  298.000000          298.000000
mean    1462.38255   29.818792           62.909396
std     248.67329   51.606065          102.519614
min      731.00000    0.000000           1.000000
25%     1331.25000    7.000000           16.000000
50%     1446.00000   14.000000           33.500000
75%     1579.75000   30.000000           62.750000
max     2551.00000  484.000000          840.000000

      The Americas and the Caribbean  Antarctica and Oceania
count                298.000000          298.000000
mean                 8.087248           1.949664
std                 9.397638           2.168216
min                 0.000000           0.000000
25%                 2.000000           0.000000
50%                 5.000000           1.000000
75%                10.000000           3.000000
max                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”:

```
In [8]: db.describe().T
```

```
Out[8]:
```

	count	mean	std	min	\
Europe	298.0	1462.382550	248.673290	731.0	
Africa	298.0	29.818792	51.606065	0.0	
Middle East and Asia	298.0	62.909396	102.519614	1.0	
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0	
Antarctica and Oceania	298.0	1.949664	2.168216	0.0	

	25%	50%	75%	max
Europe	1331.25	1446.0	1579.75	2551.0
Africa	7.00	14.0	30.00	484.0
Middle East and Asia	16.00	33.5	62.75	840.0
The Americas and the Caribbean	2.00	5.0	10.00	61.0
Antarctica and Oceania	0.00	1.0	3.00	11.0

- Equally, common descriptive statistics are also available:

```
In [9]: # Obtain minimum values for each table
db.min()
```

```
Out[9]: Europe          731
Africa                  0
Middle East and Asia    1
The Americas and the Caribbean  0
Antarctica and Oceania  0
dtype: int64
```

```
In [10]: # Obtain minimum value for the column 'Europe'
db['Europe'].min()
```

```
Out[10]: 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:

```
In [11]: # Obtain standard deviation for the row 'E01006512',
# which represents a particular LSOA
db.loc['E01006512', :].std()
```

```
Out[11]: 457.88426485303029
```

- Simple 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:

```
In [12]: # 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()
```

```
Out[12]: GeographyCode
E01006512    1880
E01006513    2941
E01006514    2108
E01006515    1208
E01006518    1696
dtype: int64
```

```
In [13]: # One shot
total = db.sum(axis=1)
# Print the top of the variable
total.head()
```

```
Out[13]: 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:

```
In [14]: db['Total'] = total
db.head()
```

```
Out[14]:
```

	Europe	Africa	Middle East and Asia \
GeographyCode			
E01006512	910	106	840
E01006513	2225	61	595
E01006514	1786	63	193
E01006515	974	29	185
E01006518	1531	69	73

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01006512	24	0	1880
E01006513	53	7	2941
E01006514	61	5	2108
E01006515	18	2	1208
E01006518	19	4	1696

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

```
In [15]: # New variable with all ones
db['ones'] = 1
db.head()
```

```
Out[15]:
```

	Europe	Africa	Middle East and Asia \
GeographyCode			
E01006512	910	106	840
E01006513	2225	61	595
E01006514	1786	63	193
E01006515	974	29	185
E01006518	1531	69	73

	The Americas and the Caribbean	Antarctica and Oceania	Total \
GeographyCode			
E01006512	24	0	1880
E01006513	53	7	2941
E01006514	61	5	2108

E01006515	18	2	1208
E01006518	19	4	1696

ones	
GeographyCode	
E01006512	1
E01006513	1
E01006514	1
E01006515	1
E01006518	1

And we can modify specific values too:

```
In [16]: db.loc['E01006512', 'ones'] = 3
db.head()
```

```
Out[16]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106	840	
E01006513	2225	61	595	
E01006514	1786	63	193	
E01006515	974	29	185	
E01006518	1531	69	73	

	The Americas and the Caribbean	Antarctica and Oceania	Total	\
GeographyCode				
E01006512		24	0	1880
E01006513		53	7	2941
E01006514		61	5	2108
E01006515		18	2	1208
E01006518		19	4	1696

ones	
GeographyCode	
E01006512	3
E01006513	1
E01006514	1
E01006515	1
E01006518	1

- Deleting variables is also trivial:

```
In [17]: del db['ones']
db.head()
```

```
Out[17]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106	840	
E01006513	2225	61	595	
E01006514	1786	63	193	
E01006515	974	29	185	
E01006518	1531	69	73	

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			

E01006512	24	0	1880
E01006513	53	7	2941
E01006514	61	5	2108
E01006515	18	2	1208
E01006518	19	4	1696

- Simple 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:

```
In [18]: eu_tot_first4 = db.loc[['E01006512', 'E01006513', 'E01006514', 'E01006515'], \
                                ['Total', 'Europe']]
```

```
eu_tot_first4
```

```
Out[18]:
```

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:

```
In [19]: m5k = db.loc[db['Total'] > 2500, :]
m5k
```

```
Out[19]:
```

GeographyCode	Europe	Africa	Middle East and Asia	\
E01006513	2225	61	595	
E01006747	2551	163	812	
E01006751	1843	139	568	

GeographyCode	The Americas and the Caribbean	Antarctica and Oceania	Total
E01006513	53	7	2941
E01006747	24	2	3552
E01006751	21	1	2572

... areas where there are no more than 750 Europeans:

```
In [20]: nm5ke = db.loc[db['Europe'] < 750, :]
nm5ke
```

```
Out[20]:
```

GeographyCode	Europe	Africa	Middle East and Asia	\
E01033757	731	39	223	

GeographyCode	The Americas and the Caribbean	Antarctica and Oceania	Total
E01033757	29	3	1025

... areas with exactly ten person from Antarctica and Oceania:

```
In [21]: oneOA = db.loc[db['Antarctica and Oceania'] == 10, :]  
oneOA
```

```
Out[21]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006679	1353	484		354

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01006679	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:

```
In [22]: eu_lth = db.loc[(db['Europe'] * 100. / db['Total']) < 50, :]  
eu_lth
```

```
Out[22]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106		840

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01006512	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:

```
In [23]: ac25_1500 = db.loc[(db['The Americas and the Caribbean'] > 25) & \  
                             (db['Total'] < 1500), :]  
ac25_1500
```

```
Out[23]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01033750	1235	53		129
E01033752	1024	19		114
E01033754	1262	37		112
E01033756	886	31		221
E01033757	731	39		223
E01033761	1138	52		138

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01033750	26	5	1448
E01033752	33	6	1196
E01033754	32	9	1452
E01033756	42	5	1185
E01033757	29	3	1025
E01033761	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:

```
In [24]: db_pop_sorted = db.sort_values('Total', ascending=False)
         db_pop_sorted.head()
```

```
Out[24]:
```

	Europe	Africa	Middle East and Asia \
GeographyCode			
E01006747	2551	163	812
E01006513	2225	61	595
E01006751	1843	139	568
E01006524	2235	36	125
E01006787	2187	53	75

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01006747	24	2	3552
E01006513	53	7	2941
E01006751	21	1	2572
E01006524	24	11	2431
E01006787	13	2	2330

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

2.3 Visual exploration

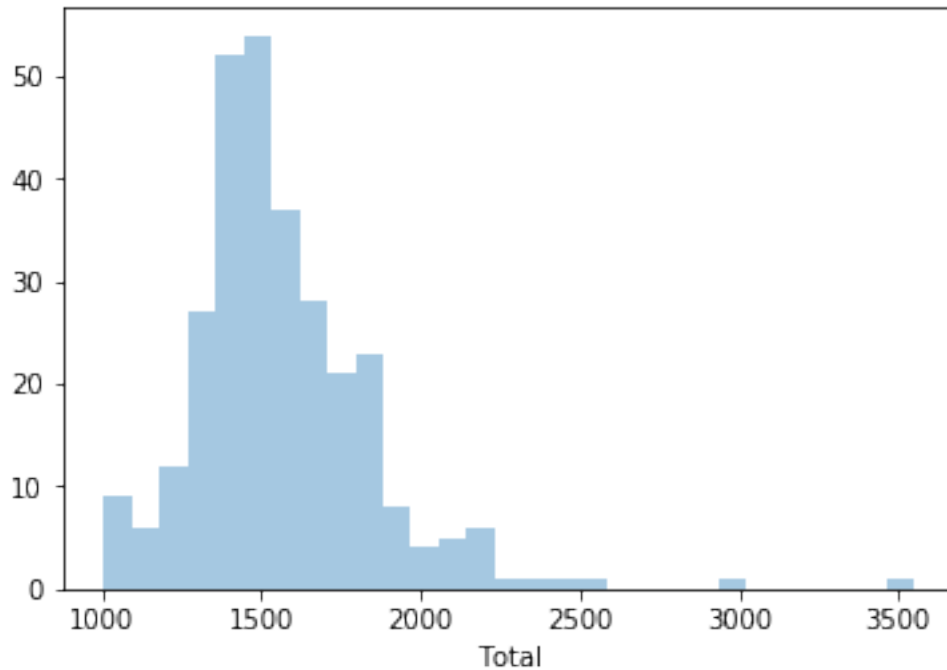
The next step to continue exploring a dataset is to get a feel for what it looks like, visually. We have already learnt how to uncover and inspect specific parts of the data, to check for particular cases we might be interested in. Now we will see how to plot the data to get a sense of the overall distribution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

- Histograms.

One of the simplest 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:

```
In [25]: _ = sns.distplot(db['Total'], kde=False)
```

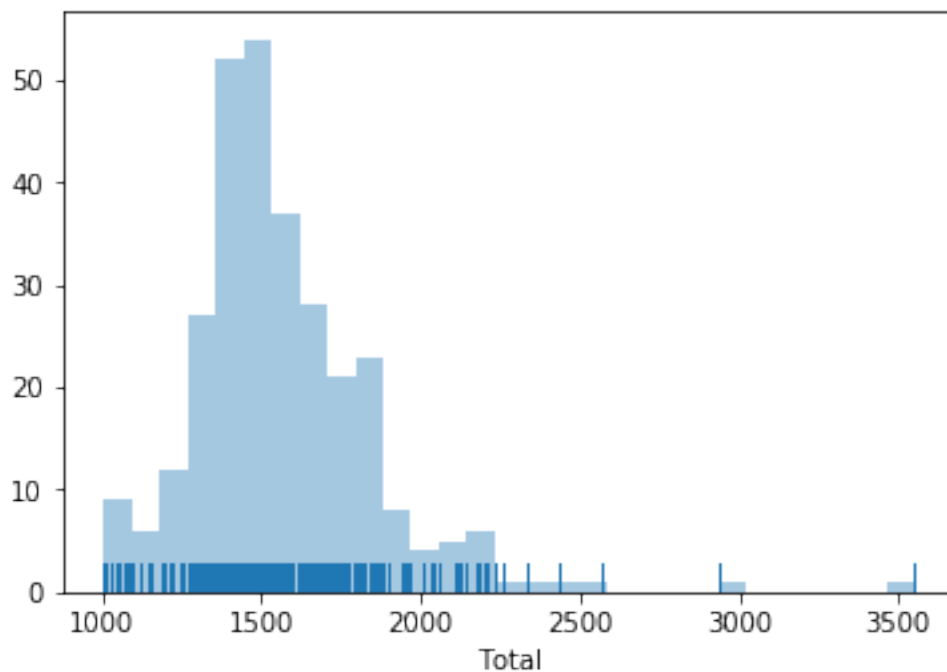


Note we are using `sns` instead of `pd`, as the function belongs to `seaborn` instead of `pandas`.

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:

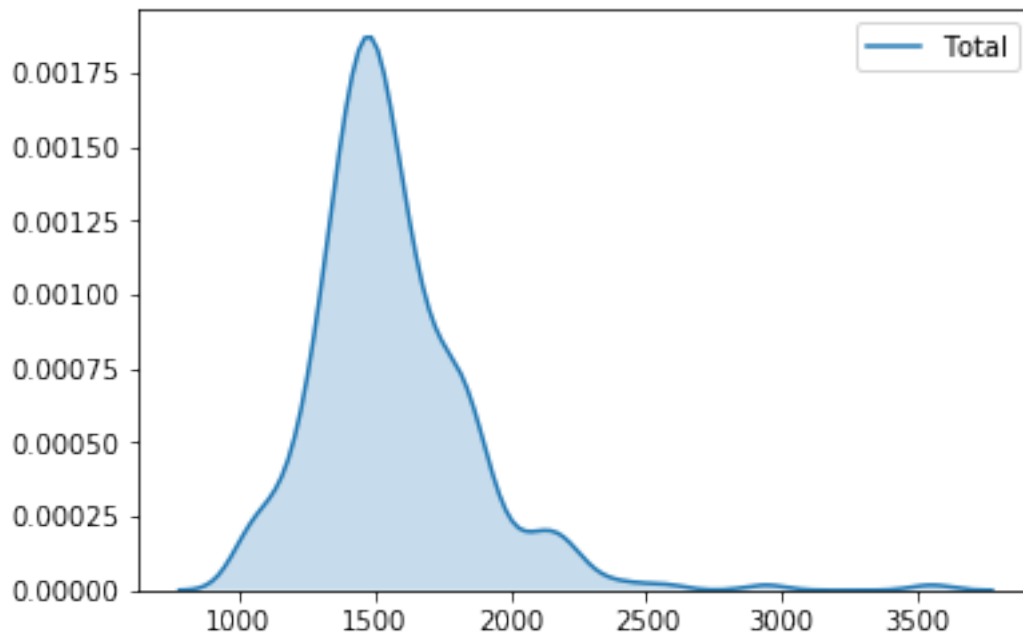
```
In [26]: _ = sns.distplot(db['Total'], kde=False, rug=True)
```



- 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:

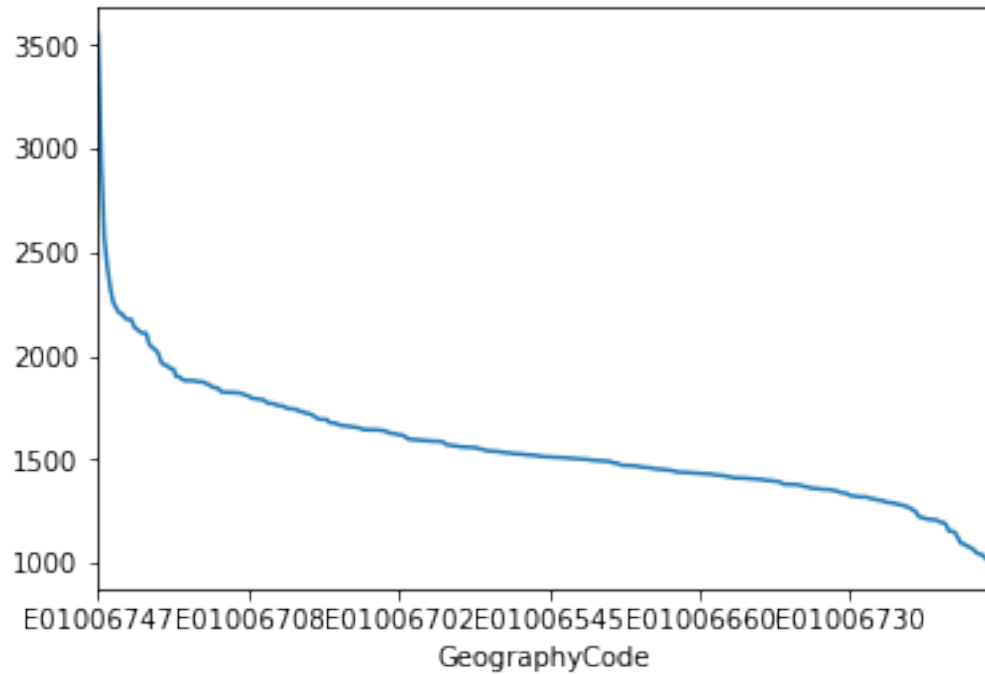
```
In [27]: _ = sns.kdeplot(db['Total'], shade=True)
```



- Line and bar plots

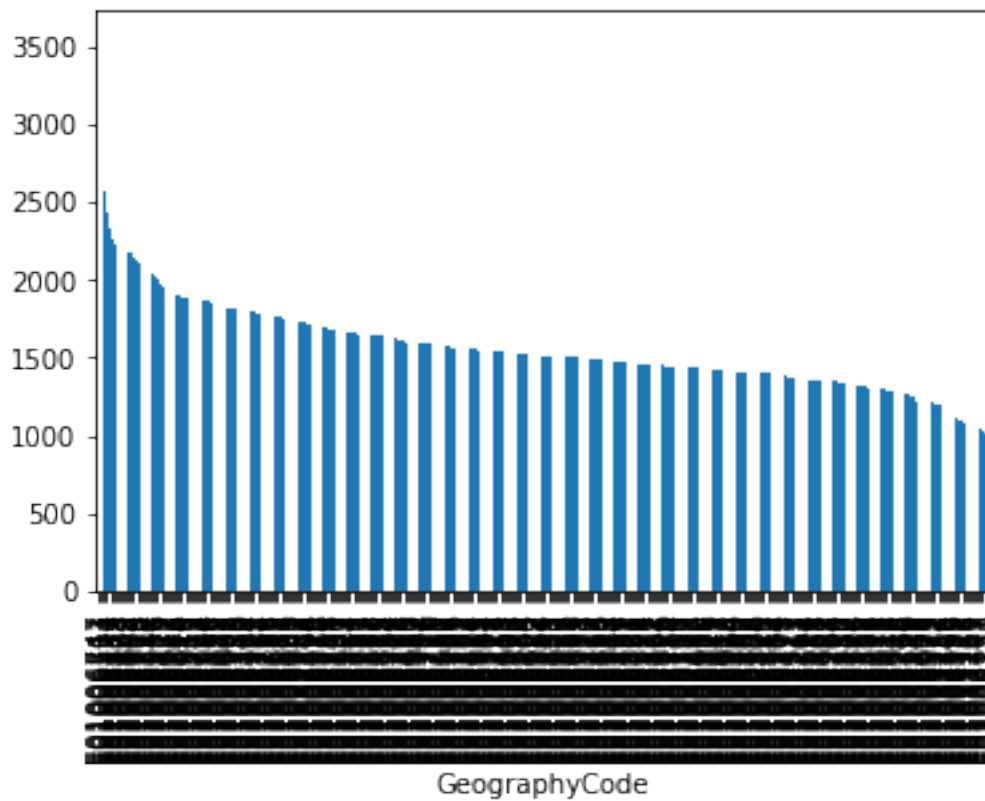
Another very common way of visually displaying a variable is with a line or a bar chart. For example, if we want to generate a line plot of the (sorted) total population by area:

```
In [28]: _ = db['Total'].sort_values(ascending=False).plot()
```



For a bar plot all we need to do is to change an argument of the call:

```
In [29]: _ = db['Total'].sort_values(ascending=False).plot(kind='bar')
```



Note that the large number of areas makes the horizontal axis unreadable. We can try to turn the plot around by displaying the bars horizontally (see how it's just changing `bar` for `barh`). To make it readable, let us expand the plot's height:

```
In [30]: _ = db['Total'].sort_values().plot(kind='barh', figsize=(6, 20))
```



2.3.1 Un/tidy data

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 easy 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:

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

```
Out[31]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106		840
E01006513	2225	61		595
E01006514	1786	63		193
E01006515	974	29		185
E01006518	1531	69		73

	The Americas and the Caribbean	Antarctica and Oceania	Total
GeographyCode			
E01006512		24	0 1880
E01006513		53	7 2941
E01006514		61	5 2108
E01006515		18	2 1208
E01006518		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:

```
In [32]: # Assign column 'Total' into its own as a single-column table
db_totals = db[['Total']]
db_totals.head()
```

```
Out[32]:
```

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

```
In [33]: # Create a table 'db_subgroups' that contains every column in 'db' without 'Total'
db_subgroups = db.drop('Total', axis=1)
db_subgroups.head()
```

```
Out[33]:
```

	Europe	Africa	Middle East and Asia	\
GeographyCode				
E01006512	910	106		840
E01006513	2225	61		595
E01006514	1786	63		193
E01006515	974	29		185
E01006518	1531	69		73

	The Americas and the Caribbean	Antarctica and Oceania
GeographyCode		
E01006512	24	0
E01006513	53	7
E01006514	61	5
E01006515	18	2
E01006518	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.

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`:

```
In [34]: tidy_subgroups = db_subgroups.stack()
tidy_subgroups.head()
```

```
Out[34]:
```

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`:


```
In [35]: type(tidy_subgroups)
```

```
Out[35]: 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:

```
In [36]: # Unfold the multi-index into different, new columns
tidy_subgroupsDF = tidy_subgroups.reset_index()
tidy_subgroupsDF.head()
```

```
Out[36]:
```

	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:

```
In [37]: tidy_subgroupsDF = tidy_subgroupsDF.rename(columns={'level_1': 'Subgroup', 0: 'Freq'})
tidy_subgroupsDF.head()
```

```
Out[37]:
```

	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!

2.3.2 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.

```
In [38]: pop_grouped = tidy_subgroupsDF.groupby('Subgroup')
pop_grouped
```

```
Out[38]: <pandas.core.groupby.DataFrameGroupBy object at 0x110a01780>
```

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:

```
In [39]: pop_grouped.sum()
```

```
Out[39]:
```

	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:

```
In [40]: pop_grouped.describe()
```

```
Out[40]:
```

	Freq count	mean	std	min
Subgroup				
Africa	298.0	29.818792	51.606065	0.0
Antarctica and Oceania	298.0	1.949664	2.168216	0.0
Europe	298.0	1462.382550	248.673290	731.0
Middle East and Asia	298.0	62.909396	102.519614	1.0
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0

	25%	50%	75%	max
Subgroup				
Africa	7.00	14.0	30.00	484.0
Antarctica and Oceania	0.00	1.0	3.00	11.0
Europe	1331.25	1446.0	1579.75	2551.0
Middle East and Asia	16.00	33.5	62.75	840.0
The Americas and the Caribbean	2.00	5.0	10.00	61.0

We will not get into it today as it goes beyond the basics we want to cover, 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.

2.4 If you finish early...

Practice your data tidying skills with a different dataset. For example, you can have a look at the Guardian's version of [Wikileaks' Afghanistan war logs](https://www.theguardian.com/world/2013/jul/27/wikileaks-afghanistan-war-logs). The table is stored on a GoogleDoc on the following address:

```
https://docs.google.com/spreadsheets/d/1EAX8\_ksSCmoWW\_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit?hl=en#gid=1
```

And its structure is as follows:

```
In [41]: from IPython.display import IFrame
url = 'https://docs.google.com/spreadsheets/d/1EAX8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit?hl=en#gid=1'
IFrame(url, 700, 400)
```

```
Out[41]: <IPython.lib.display.IFrame at 0x110a7ecc0>
```

Follow these steps:

- Download the table as a `csv` file (File -> Download as -> `.csv`, current sheet).

- Read it into Python.
- Explore it by creating a few plots.
- Examine its level of tidiness and turn it into a fully tidy dataset.
- Obtain a monthly total count of casualties and create a line or a bar plot of them.

Geographic Data Science'17 - Lab 1, part I by Dani Arribas-Bel is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.