lab_01_ii

September 13, 2017

1 Geographic Data Science - Lab 01, Part II

Dani Arribas-Bel

This notebook elaborates on the previous session and shows some more advanced tricks that will allow you to perform data cleaning and processing in cases where the original source data used are not made available ready for analysis (as we did in the previous session). In particular, we will show how you can transform data downloaded from the internet into the table you used to explore population patterns in Liverpool.

Before anything, let us import the libraries we will need:

```
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
```

1.0.1 The Liverpool Census Data Pack

Throughout this notebook (and later on as well), we will use the CDRC's Census Data Pack for the city of Liverpool (link) and explore some of the city's socio-demogaphic characteristics. This is a large package crafted by the CDRC that brings together several Census tables in a consistent way for the city of Liverpool. We will only be able to use just a few of them but, since they are consistently organized, the procedure used should teach you how to explore other variables on your own. In particular, in this session, we will be using a table that lists **population by country of birth**.

The pack is composed of two types of data: tabular and spatial. Tabular data are numerical tables that contain information relating to many socio-economic variables for different units (areas); spatial data contains the geometries of the areas in which Liverpool is divided into. Since there are many variables contained in several tables, that can be linked to more than one geography, the pack also includes two "compass files" that help you find what you are looking for: one table that lists and describes the different datasets available; and a much more detailed table that lists and describes each and every single variable available in the pack.

The remainder assumes you have downloaded and unpacked the data.

IMPORTANT: if you are working on a university lab computer, make sure to store the downloaded files (as well as the notebook) in the `M:` drive. This will ensure it is safe and does not get erased.

Specify the path to the folder in the following cell, so you can correctly run the code without errors:

```
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:
        # path = 'Liverpool/'
        path = '../../../gds17_data/Liverpool/'

        # Check to see if the path is correct and works. If you have set it
```

```
# correctly, you should obtain the following list
os.listdir(path)
```

```
Out[2]: ['datasets_description.csv',
 'metadata.xml',
 'readme.txt',
 'shapefiles',
 'tables',
 'variables_description.csv']
```

1.0.2 Creating the table from the previous notebook

It is not only that data are not ready to analyze when you get a hold on them. Sometimes, there is not such thing as the dataset to analyze. Instead, what you have is a collection of separated files, sometimes with different structures, that you need to bring together to begin with. This is one of the areas where a bit of scripting skills can help you a long way. While in a traditional point-and-click program like Microsoft Excel or SPSS, you would have to repeat the steps every time you wanted to incorporate a new dataset, with a bit of Python ninja tricks, you can write code that will do it for you as many times as you need.

We will begin jumping straight into the analysis of population in Liverpool, organized by country of birth, at the Local Super Output Area (LSOA) level. Because the Census Data Pack contains a lot of data and very many different tables, you will have to bear with us and trust that what we are extracting is exactly the data of interest. This will speed up the process to walk through the reading, processing and manipulating of a dataset. Once you are familiar with these skills, the final section goes into how to explore the entire pack with more detail.

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

```
In [3]: lsoa_orig = pd.read_csv(path+'tables/QS203EW_lsoa11.csv', index_col='GeographyCode')
        lsoa_orig.head()
```

```
Out[3]:                QS203EW0001  QS203EW0002  QS203EW0003  QS203EW0004  \
        GeographyCode
        E01006512             1880          910          766          699
        E01006513             2941         2225         2033         1806
        E01006514             2108         1786         1632         1503
        E01006515             1208          974          910          877
        E01006518             1696         1531         1468         1446

                       QS203EW0005  QS203EW0006  QS203EW0007  QS203EW0008  \
        GeographyCode
        E01006512               26           21           20            0
        E01006513               98           28          101            0
        E01006514               44           18           67            0
        E01006515               16            5           12            0
        E01006518                7            6            9            0

                       QS203EW0009  QS203EW0010     ...       QS203EW0069  \
        GeographyCode                               ...
        E01006512                0            0     ...                 5
        E01006513                0            0     ...                 5
        E01006514                0            0     ...                19
        E01006515                0            0     ...                 4
        E01006518                0            0     ...                 3

                       QS203EW0070  QS203EW0071  QS203EW0072  QS203EW0073  \
        GeographyCode
```

```
E01006512            0            5            0            0
E01006513            1            4            7            0
E01006514            2           17            5            0
E01006515            2            2            2            0
E01006518            0            3            4            0

               QS203EW0074  QS203EW0075  QS203EW0076  QS203EW0077  QS203EW0078
GeographyCode
E01006512                0            0            0            0            0
E01006513                7            6            1            0            0
E01006514                4            2            2            1            0
E01006515                2            2            0            0            0
E01006518                4            4            0            0            0

[5 rows x 78 columns]
```

Before we continue with the data, let us have a look at the object `lsoa_orig`. It is a different "animal" than we have seen so far:

```
In [4]: type(lsoa_orig)
```

```
Out[4]: pandas.core.frame.DataFrame
```

It is a "pandas data frame". Similar to R's "data.frame" class, it is one of the most essential data structures in Python for data analysis, and we will use it intensively. Data frames are sophisticated costructs that can perform several advanced tasks and have many properties. We will be discovering them as we progress on the course but, for now, let us keep in mind they are tables, indexed on rows and columns that can support mixed data types and can be flexibly manipulated.

Now we have read the file, we can inspect it. For example, to show the first lines of the table:

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

```
Out[5]:        QS203EW0001  QS203EW0002  QS203EW0003  QS203EW0004  \
GeographyCode
E01006512             1880          910          766          699
E01006513             2941         2225         2033         1806
E01006514             2108         1786         1632         1503
E01006515             1208          974          910          877
E01006518             1696         1531         1468         1446

               QS203EW0005  QS203EW0006  QS203EW0007  QS203EW0008  \
GeographyCode
E01006512               26           21           20            0
E01006513               98           28          101            0
E01006514               44           18           67            0
E01006515               16            5           12            0
E01006518                7            6            9            0

               QS203EW0009  QS203EW0010      ...       QS203EW0069  \
GeographyCode                                ...
E01006512                0            0      ...                 5
E01006513                0            0      ...                 5
E01006514                0            0      ...                19
E01006515                0            0      ...                 4
E01006518                0            0      ...                 3
```

```
                 QS203EW0070  QS203EW0071  QS203EW0072  QS203EW0073  \
GeographyCode
E01006512                  0            5            0            0
E01006513                  1            4            7            0
E01006514                  2           17            5            0
E01006515                  2            2            2            0
E01006518                  0            3            4            0

                 QS203EW0074  QS203EW0075  QS203EW0076  QS203EW0077  QS203EW0078
GeographyCode
E01006512                  0            0            0            0            0
E01006513                  7            6            1            0            0
E01006514                  4            2            2            1            0
E01006515                  2            2            0            0            0
E01006518                  4            4            0            0            0

[5 rows x 78 columns]
```

Let us also quickly check the dimensions of the table:

```
In [6]: lsoa_orig.shape
```

```
Out[6]: (298, 78)
```

This implies 298 rows by 78 columns. That is a lot of columns, all named under obscure codes. For now, just trust that the columns we want are:

```
In [7]: region_codes = ['QS203EW0002', 'QS203EW0032', 'QS203EW0045', \
                        'QS203EW0063', 'QS203EW0072']
```

To keep only those with us, we can slice the table using the `loc` operator:

```
In [8]: # Select only the columns with names in the list `region_codes`
        lsoa_orig_sub = lsoa_orig.loc[:, region_codes]

        lsoa_orig_sub.head()
```

```
Out[8]:          QS203EW0002  QS203EW0032  QS203EW0045  QS203EW0063  QS203EW0072
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 how we use the operator `loc` (for locator) on the dataframe, followed by squared brackets and, inside it, two alternatives:

- We can use `:` to keep all the elements (rows in this case).
- And we can use a list of strings (or simply one would work too) with the names what we want to select.

We can further inspect the dataset with an additional command called `info`, that lists the names of the columns and how many non-null elements each contains:

```
In [9]: lsoa_orig_sub.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 5 columns):
QS203EW0002    298 non-null int64
QS203EW0032    298 non-null int64
QS203EW0045    298 non-null int64
QS203EW0063    298 non-null int64
QS203EW0072    298 non-null int64
dtypes: int64(5)
memory usage: 14.0+ KB
```

[Renaming columns]

IMPORTANT: some of the elemnts in this part are more advanced hence optional. If you want to move quickly through the lab, simply run the code cells without paying much attention to what it does. Once you have become more familiar with the rest of the tutorial, return here and work through the logic.

The table we have compiled contains exactly what we wanted. However, the names of the columns are a bit unintuitive, to say the least. It would be much handier if we could rename the columns into something more human readable. The easiest way to do that in `pandas` is by creating a dictionary that maps the original name into the desired one, and then applying it to the `DataFrame` with the command `rename`. Let us walk through the steps necessary, one by one:

- Create a dictionary that maps the codes to the names. For this, we can use the list we have created before (`region_variables`), and what we have learnt about querying tables, combined with a small `for` loop.

First we need to bring up the variable names into a separate table (see the final section for more detail):

```
In [10]: # Read file with variable descriptions
         variables = pd.read_csv(path+'variables_description.csv', index_col=0)
```

```
In [11]: # Create a "dictionary" to store names of the variables
         # and their description
         code2name = {}
         # Set the index to be the code of each variable
         lookup_table = variables.set_index('ColumnVariableCode') # Reindex to be able to query
         # Run over every region code, select its description/name and store it
         # in the `code2name` dictionary
         for code in region_codes:
             code2name[code] = lookup_table.loc[code, 'ColumnVariableDescription']
         code2name
```

```
Out[11]: {'QS203EW0002': 'Europe: Total',
          'QS203EW0032': 'Africa: Total',
          'QS203EW0045': 'Middle East and Asia: Total',
          'QS203EW0063': 'The Americas and the Caribbean: Total',
          'QS203EW0072': 'Antarctica and Oceania: Total'}
```

- Because we know that each of these variables are totals for each group, we can further declutter the names by removing the piece of the string `": Total"`. A simple loop can help us:

```
In [12]: # Loop over every code in the `code2name` dictionary and chop off
         # ": Total" so the name is shorter and neater
         for code in code2name:
             code2name[code] = code2name[code].replace(': Total', '')
         code2name
```

```
Out[12]: {'QS203EW0002': 'Europe',
          'QS203EW0032': 'Africa',
          'QS203EW0045': 'Middle East and Asia',
          'QS203EW0063': 'The Americas and the Caribbean',
          'QS203EW0072': 'Antarctica and Oceania'}
```

- With the dictionary in hand, renaming the columns is as easy as:

```
In [13]: # Rename each column in `lsoa_orig_sub` from its code to its name
         lsoa_orig_sub = lsoa_orig_sub.rename(columns=code2name)

         lsoa_orig_sub.head()
```

```
Out[13]:                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
```

And this is it! The table stored in `lsoa_orig_sub` is essentially the same as we played with in the previous session.

1.1 Delving deeper into the Census Data Pack

We started this notebook assuming we already knew what variables in particular we wanted, out of the hundreds available on the Liverpool Census Data Pack. Unfortunately, that is not always the case, and sometimes you have to explore an entire dataset by yourself to find what you are looking for. To dip your toes into the sea of the Census Data Pack, in this section we will walk through how to systematically identify a variable and extract it.

The folder contains data at different scales. We will be using the Local Super Output Area (LSOAs). The folder is structured in the following way:

```
In [14]: # This command lists the files in the folder passed (`path` in this case)
         os.listdir(path)
```

```
Out[14]: ['datasets_description.csv',
          'metadata.xml',
          'readme.txt',
          'shapefiles',
          'tables',
          'variables_description.csv']
```

For now, we will ignore the spatial information contained in the folder `shapefiles` and focus on the `tables` one. If you have a peek at the folder, it contains many files. You can get their names into a Python list with the following command:

```
In [15]: # Create a list with the names of all the tables available
         csvs = os.listdir(path + 'tables')
```

And count them using the core fuction `len`, which returns the length of a list:

```
In [16]: # Obtain the length of the list `csvs`
         len(csvs)

Out[16]: 303
```

That is right, 303 files! Luckily, to navigate that sea of seemingly non-sensical letters, there is a codebook that explains things a bit. You can open it with a text editor or a spreadsheet program but, since it is a `csv` file, we can also ingest it with Python:

```
In [17]: # Load up the file and set the first column as index
         codebook = pd.read_csv(path + 'datasets_description.csv', index_col=0)
```

Now we have read the file, we can inspect it. For example, to show the first lines of the table:

```
In [18]: codebook.head()

Out[18]:                                        DatasetTitle
         DatasetId
         CT0010                    Ethnic group write-ins
         KS101EW                Usual resident population
         KS102EW                            Age structure
         KS103EW    Marital and civil partnership status
         KS104EW                      Living arrangements
```

You can use the index chosen to query rows. For example, if we want to see what dataset code `QS203EW` corresponds to:

```
In [19]: # Extract the value for the column `DatasetTitle` and the row `QS203EW`
         # This is effectively the name of the dataset with that code
         codebook.loc['QS203EW', 'DatasetTitle']

Out[19]: 'Country of birth (detailed)'
```

If we want to see what that dataset contains, there is another file in the folder called `variables_description.csv` that has further information. We can bring it in the same way we did before and, again, we will index it using the first column of the table, the ID of the dataset where the variable belongs to:

```
In [20]: variables = pd.read_csv(path+'variables_description.csv', index_col=0)
```

To have a sense of how large it is, we can call its `shape` property, which returns the number of rows and columns, respectively:

```
In [21]: # Get the dimensions of the table `variables`
         variables.shape

Out[21]: (2563, 3)
```

2,563 different variables!!! Let us see what the structure of the table is:

```
In [22]: variables.head()
```

```
Out[22]:           ColumnVariableCode ColumnVariableMeasurementUnit  \
DatasetId
CT0010            CT00100001                         Count
CT0010            CT00100002                         Count
CT0010            CT00100003                         Count
CT0010            CT00100004                         Count
CT0010            CT00100005                         Count

                                     ColumnVariableDescription
DatasetId
CT0010                           All categories: Ethnic group
CT0010     English/Welsh/Scottish/Northern Irish/British
CT0010                                                  Irish
CT0010                             Gypsy or Irish Traveller
CT0010                                            Other White
```

If we are interested in exploring the country of birth (code `QS203EW`), we can subset the table using `loc` in a similar way as before. The only difference is that now we do not want to restrict the column to only one, so we use the colon `:` instead of a particular name, including thus all the columns. Let us also save the subset by assigning it to a new object, `birth_orig`:

```
In [23]: # Select all the column values for the row 'QS203EW'
         birth_orig = variables.loc['QS203EW', :]

         birth_orig.shape
```

```
Out[23]: (78, 3)
```

To be clear, the table above contains all the variables that the dataset `QS203EW` is comprised of. This means that, for every row in this table, there is a column in the actual dataset which, for the LSOAs, is on the file `QS203EW_lsoa11.csv`, in the `tables` folder.

This is still a lot. Arguably, to get a first sense of the data and start exploring it, we do not need every single disaggregation available. Let us look at the names and codes of the first 25 variables to see if we can spot any pattern that helps us simplify (note how we now use `:` first to indicate we want all the rows):

```
In [24]: # Select all the rows for the two columns 'ColumnVariableCode' and
         # 'ColumnVariableDescription', and show the top 25
         birth_orig.loc[:, ['ColumnVariableCode', 'ColumnVariableDescription']].head(25)
```

```
Out[24]:           ColumnVariableCode  \
DatasetId
QS203EW          QS203EW0001
QS203EW          QS203EW0002
QS203EW          QS203EW0003
QS203EW          QS203EW0004
QS203EW          QS203EW0005
QS203EW          QS203EW0006
QS203EW          QS203EW0007
QS203EW          QS203EW0008
QS203EW          QS203EW0009
QS203EW          QS203EW0010
QS203EW          QS203EW0011
QS203EW          QS203EW0012
QS203EW          QS203EW0013
QS203EW          QS203EW0014
```

```
QS203EW          QS203EW0015
QS203EW          QS203EW0016
QS203EW          QS203EW0017
QS203EW          QS203EW0018
QS203EW          QS203EW0019
QS203EW          QS203EW0020
QS203EW          QS203EW0021
QS203EW          QS203EW0022
QS203EW          QS203EW0023
QS203EW          QS203EW0024
QS203EW          QS203EW0025

                                    ColumnVariableDescription
DatasetId
QS203EW                     All categories: Country of birth
QS203EW                                        Europe: Total
QS203EW                         Europe: United Kingdom: Total
QS203EW                       Europe: United Kingdom: England
QS203EW              Europe: United Kingdom: Northern Ireland
QS203EW                      Europe: United Kingdom: Scotland
QS203EW                         Europe: United Kingdom: Wales
QS203EW         Europe: Great Britain not otherwise specified
QS203EW        Europe: United Kingdom not otherwise specified
QS203EW                                     Europe: Guernsey
QS203EW                                       Europe: Jersey
QS203EW       Europe: Channel Islands not otherwise specified
QS203EW                                  Europe: Isle of Man
QS203EW                                      Europe: Ireland
QS203EW                          Europe: Other Europe: Total
QS203EW             Europe: Other Europe: EU Countries: Total
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Member cou...
QS203EW     Europe: Other Europe: EU countries: Accession ...
QS203EW     Europe: Other Europe: EU countries: Accession ...
```

Note how we have been able to pass a list of variables we wanted to select as columns, and `pandas` has returned the dataframe "sliced" with only those, cutting off the rest.

It looks like the variable name follows a hierarchical pattern that dissaggregates by regions of the world. A sensible first approach might be to start considering only the largest regions. To do that, we need a list of the variable name for those aggregates since, once we have it, subsetting the dataframe will be straightforward. There are several ways we can go about it:

- Since there are not that many regions, we can hardcode them into a list, the same we have used above:

```
In [25]: region_codes = ['QS203EW0002', 'QS203EW0032', 'QS203EW0045', \
                         'QS203EW0063', 'QS203EW0072']
```

[Advanced extension. Optional]

- However, this approach would not get us very far if the list was longer. For that, a much more useful way is to write a loop that builds the list for us. To do this, we can remember some of the tricks learnt in the previous session about writing `for` loops and `if` statements and combine them with new ones about working with strings.

```
In [26]: regions = []
         for var in birth_orig['ColumnVariableDescription']:
             # Split the name of the variable in pieces by ': '
             pieces = var.split(': ')
             # Keep the first one (top hierarchy) and append ': Total'
             name = pieces[0] + ': Total'
             # If the name create matches the variable (exists in the original list),
             # add the name to the list
             if name == var:
                 regions.append(name)
         regions
```

```
Out[26]: ['Europe: Total',
          'Africa: Total',
          'Middle East and Asia: Total',
          'The Americas and the Caribbean: Total',
          'Antarctica and Oceania: Total']
```

Let us work slowly by each step of this loop:

- We first create an empty list where we will store the names of the regions.
- We begin a loop over every single row the column containing the names (`ColumnVariableDescription`).
- For each name, which is a string, we split it in pieces using `": "` as the points in the string where we want to break it, obtaining a list with the resulting pieces. For instance if we have `Europe: Total`, we essentially do:

```
In [27]: 'Europe: Total'.split(': ')
```

```
Out[27]: ['Europe', 'Total']
```

- We keep the first element, as it contains the name we want to maintain.
- In order to build the actual name of the variable, we join it to `": Total"`, obtaining the string we want to keep:

```
In [28]: 'Europe' + ': Total'
```

```
Out[28]: 'Europe: Total'
```

- We then check that the string we have built is the same as the variable we began with. If so, we save it on the list we created in the beginning. This step is a bit counter-intuitive, but is done to ensure a) that the name of the variable exists, and b) that it is saved only once.

Now we have the names, we need to convert them into the codes. There are several ways to go about it, but here we will show one that relies on the indexing capabilities of `pandas`. Essentially we take `birth_orig` and index it on the names of the variables, to then subset it, keeping only those in our list (the variables we want to retain).

```
In [29]: # Set the column `ColumnVariableDescription` as the index and keep only those
         # in the list `regions`
         subset = birth_orig.set_index('ColumnVariableDescription').reindex(regions)

         subset
```

```
Out[29]:                                     ColumnVariableCode  \
         ColumnVariableDescription
         Europe: Total                              QS203EW0002
         Africa: Total                              QS203EW0032
         Middle East and Asia: Total                QS203EW0045
         The Americas and the Caribbean: Total      QS203EW0063
         Antarctica and Oceania: Total              QS203EW0072

                                              ColumnVariableMeasurementUnit
         ColumnVariableDescription
         Europe: Total                                                Count
         Africa: Total                                                Count
         Middle East and Asia: Total                                  Count
         The Americas and the Caribbean: Total                        Count
         Antarctica and Oceania: Total                                Count
```

Once this is done, all left to do is to retrieve the codes:

```
In [30]: # Convert the column `ColumnVariableCode` in the table `subset`
         # into a list
         region_codes = list(subset['ColumnVariableCode'])

         region_codes
```

```
Out[30]: ['QS203EW0002', 'QS203EW0032', 'QS203EW0045', 'QS203EW0063', 'QS203EW0072']
```

Which is the same that we hardcoded originally, only it has been entirely picked up by our python code, not by a human.
