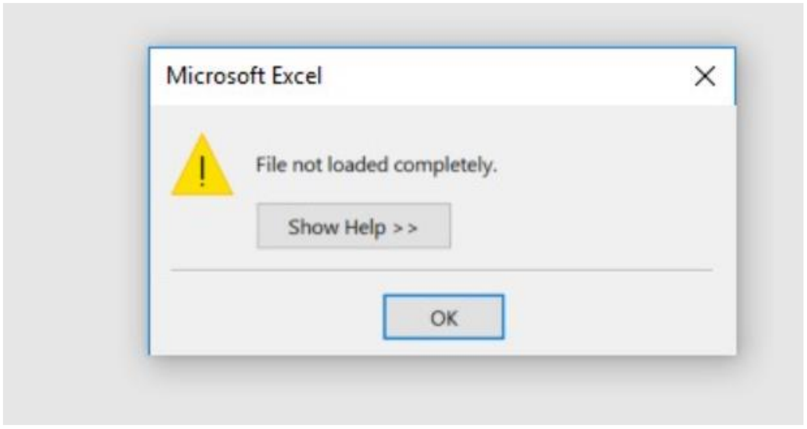
# BDDS Python Tutorial; subsetting large datasets and merging metadata

This is a **Bulk Data Download Services (BDDS)** tutorial providing a walkthrough on how to process large CSV files using the **Python** coding language (with the **Pandas** module). We selected Python for it's popularity, open source access, relevance to data science and accessibility for beginners. The Pandas module was selected because it is fast and has low memory usage. The code only takes seconds to run (depending on the system) and uses around 800 MB of memory including Python's overhead (500MB).

We assume that you have some basic understanding of coding or scripting in order to adapt the code to your needs, but we believe this tutorial is within the reach of novice coders. It is built to process any of the BDDS data packages with as few modifications as possible.

## Context

We wanted to address users' feedback concerning issues when opening the larger datasets in **Excel** e.g. **Other Policy-Relevant Indicators** (**OPRI**) containing more than 2.7 million rows. Figure 1 shows the warning message popping-up when a CSV is partially loaded due to its size exceeding Excel's limit, in this case, exceeding the maximum number of rows.


Microsoft Excel

File not loaded completely.

*Figure 1 excel warning message: file not loaded completely.*

The BDDS is meant to be accessed programmatically either with a statistical package (R, STATA, SAS, SPSS, etc.) or with a coding language. Although it is possible to load it initially in Excel with some manual workaround, we strongly discourage this practice to prevent human errors. Moreover, each BDDS package provides data point level metadata, and labels in separate CSV files. Linking these files with the core datasets requires efficiency and power that goes beyond Excel's standard capabilities.

## Objective

This tutorial fulfils two main objectives. First, to process a large CSV in a programmatic way. Second, to demonstrate how the country/indicator labels and metadata are linked and merged to the data.

Specifically, we will process the **Other Policy-Relevant Indicators** BDDS files using the **Pandas** module in **Python**, providing a walkthrough with the following steps:

1. Read (load) the full CSV file in memory;
2. Create a subset based on lists of countries, indicators and years;
3. Merge indicator and country labels to the subset (using Pandas module);
4. Merge metadata to a data subset;
5. Write the data subset back to CSV.

## Getting started

### Download the Python code

Start by downloading the tutorial Python code package (the link will start the download).

### Download the BDDS data

You will also need the Other Policy-Relevant Indicators (OPRI) data package (available on the UIS BDDS page under ‘Education’).

Unzip the Python code and the data files and take note of the path of the folder where those documents are saved, it will be required as an input to the code during this walkthrough.

### Install the Python software

There are many ways to get Python, but we recommend the convenience of **Anaconda**. This software package includes multiple data science tools, such as Python and dedicated Python Integrated Developer Environments (IDE) like Spyder and Jupyter Notebook. For this demonstration, we will use the **Spyder** IDE (within Anaconda) for running the code, looking at output tables, etc.

Go to the Anaconda download page and choose a download package for your system at the bottom of the page.

## Opening the code

Install **Anaconda** → Open Anaconda  → Click on the **Install** button in the Spyder section (Figure 2, the button will change from "Install" to "Launch") → **Launch** Spyder after it is installed

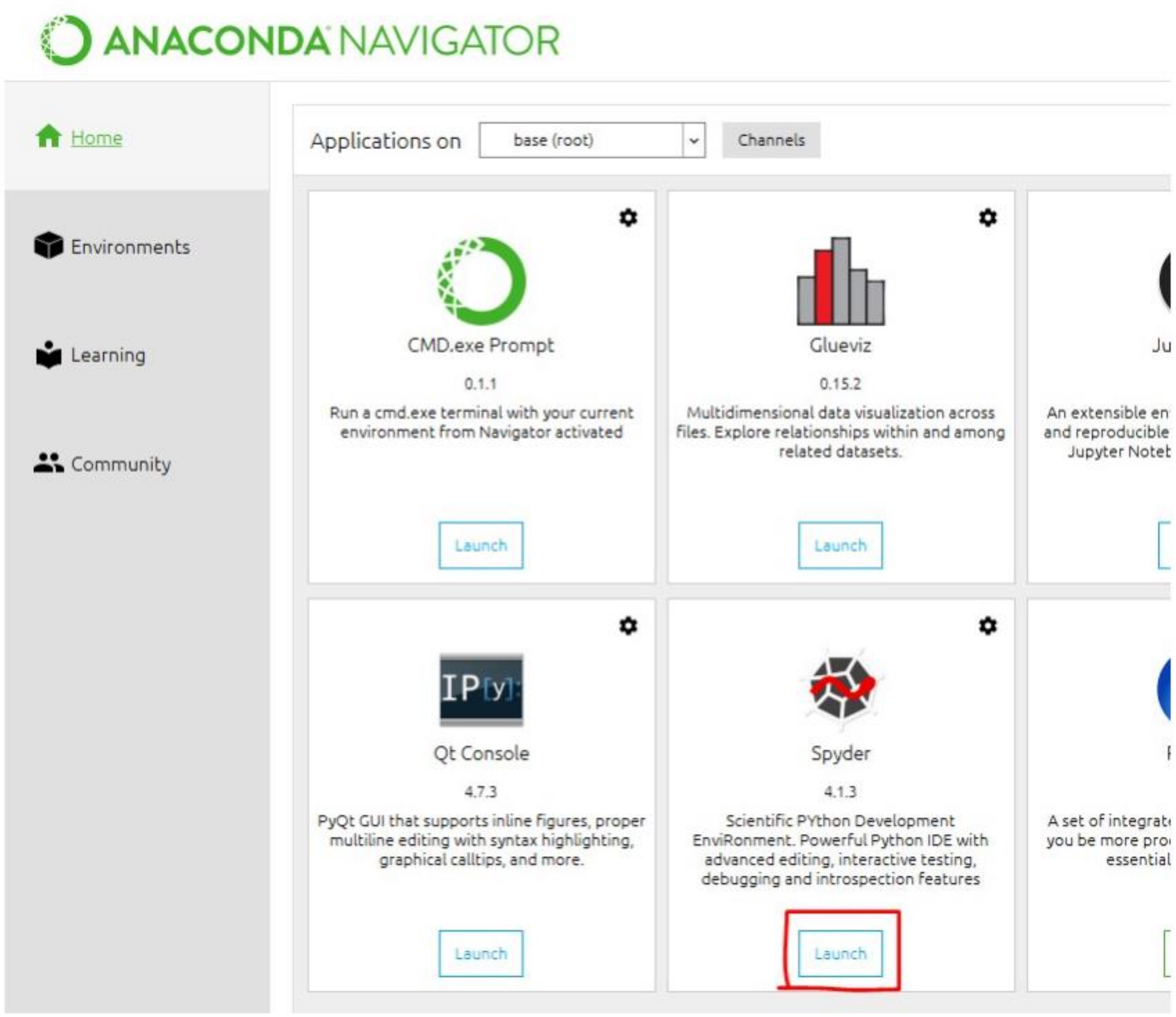

ANACONDA NAVIGATOR
Home
Environments
Learning
Community
Applications on
base (root)
Channels
CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated
Launch
Glueviz
0.15.2
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Launch
Ju
An extensible en
and reproducible
Jupyter Notel
Qt Console
4.7.3
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch
Spyder
4.1.3
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch
A set of integrat
you be more pro
essential

*Figure 2 anaconda interface's Home page*

Within Spyder: Go to **File** menu (Figure 3) → Click **Open...**→ Open the BDDS code from its saved location

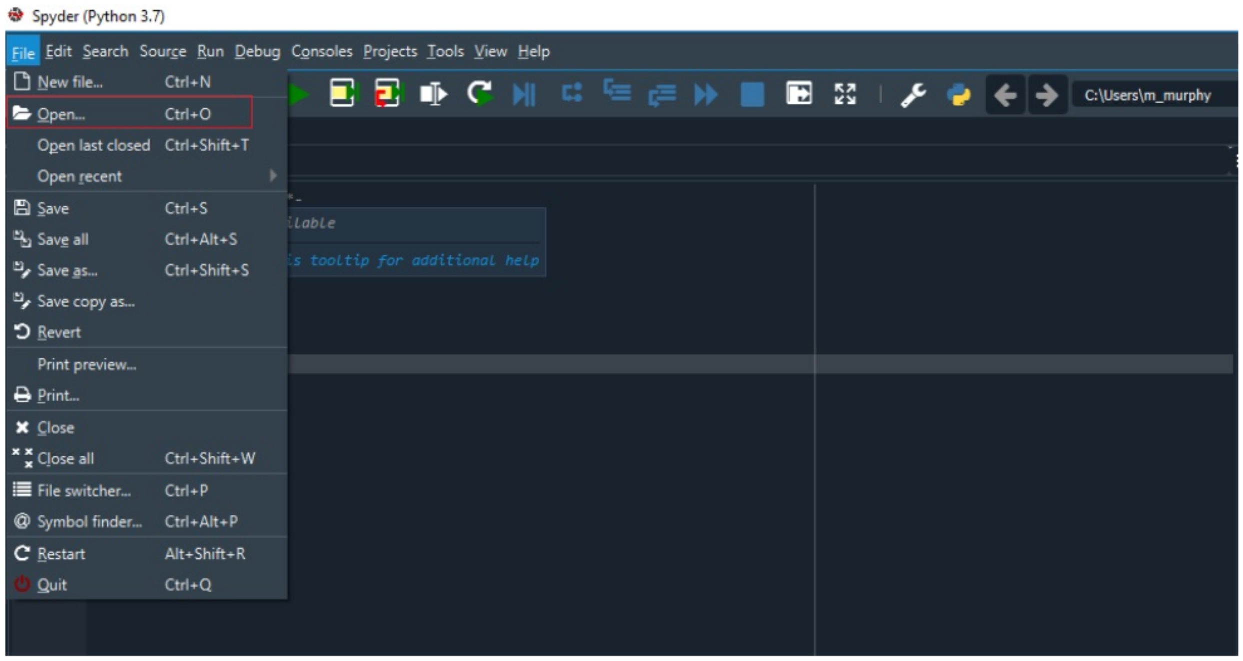


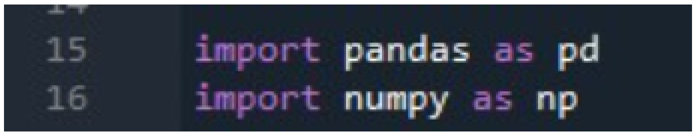
*Figure 3 opening a Python file in Spyder IDE*

This will load the Python code in Spyder. Nothing is executed at this point.

## Executing the code

In the following section, we will look at each block of code in detail. The code contains most of the necessary comments to understand how it works but we will go through each steps adding some colour to those comments.

The code uses two modules (Figure 4) that include built-in functions for data manipulation.

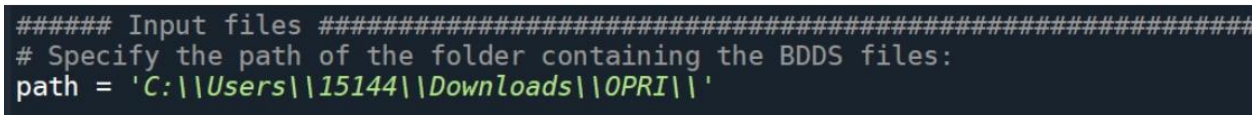


```
import pandas as pd
import numpy as np
```

*Figure 4 modules for data manipulation*

- **Pandas** is a fast, powerful, flexible and easy to use open-source data analysis and manipulation tool with Python
- **Numpy** a fundamental package for scientific computing with Python

## Specifying file path and files names



```
###### Input files ##########################################################
# Specify the path of the folder containing the BDDS files:
path = 'C:\\Users\\15144\\Downloads\\OPRI\\'
```

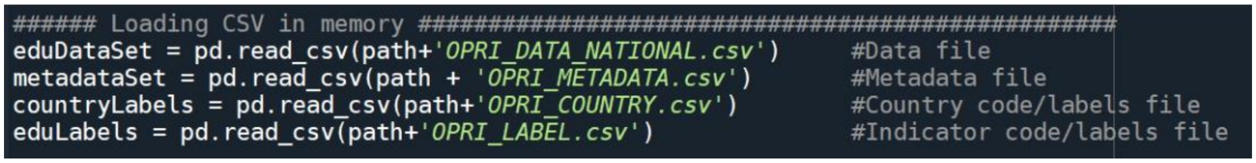
*Figure 5 file path and loading CSV data in memory*

Figure 5 shows the path where the CSV files are saved on the computer.

Change the path to the location on your computer where the BDDS files are saved

This tutorial only uses the following CSV:

- OPRI_DATA_NATIONAL.csv - the core dataset with country data;
- OPRI_METADATA.csv - the metadata table related to the data points in the core dataset;
- OPRI_COUNTRY.csv - the table with country labels and;
- OPRI_LABEL.csv - the table with indicator labels.



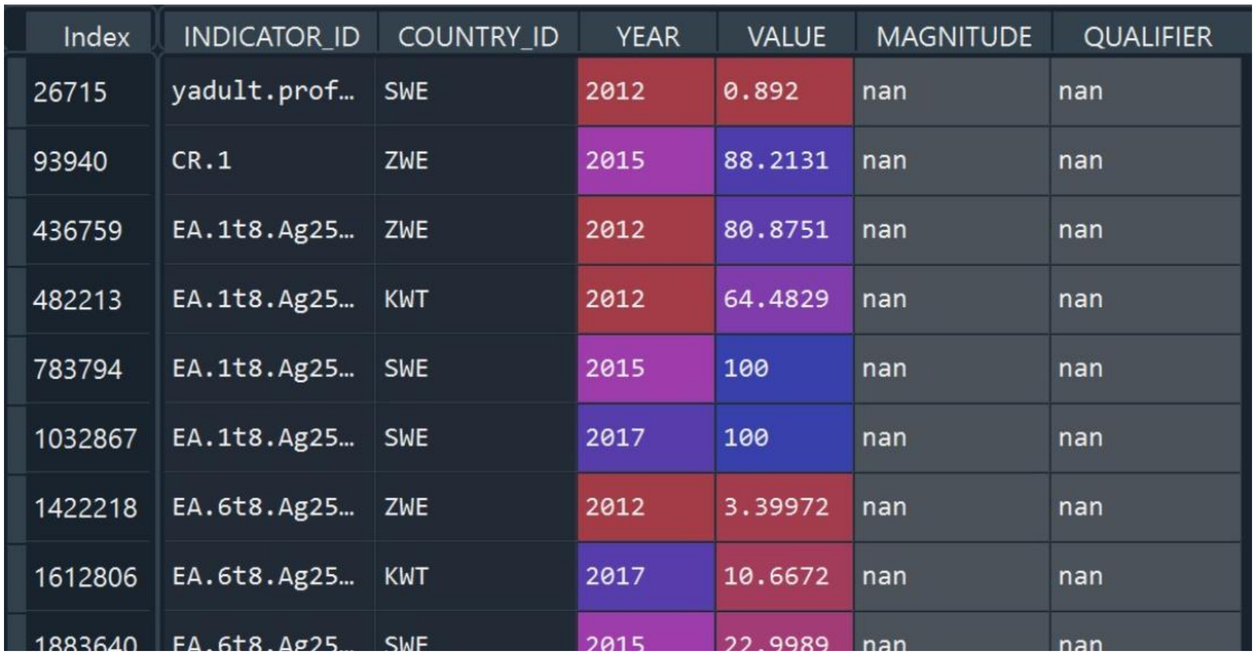
```
###### Loading CSV in memory ################################################
eduDataSet = pd.read_csv(path+'OPRI_DATA_NATIONAL.csv')        #Data file
metadataSet = pd.read_csv(path + 'OPRI_METADATA.csv')          #Metadata file
countryLabels = pd.read_csv(path+'OPRI_COUNTRY.csv')           #Country code/labels file
eduLabels = pd.read_csv(path+'OPRI_LABEL.csv')                 #Indicator code/labels file
```

*Figure 6 loading CSV files to memory*

The code in Figure 6 will load the CSV files to memory in a Pandas DataFrame reproducing the original file's structure.

Modify the file names in the code when you want to load other datasets

The "OPRI_DATA_NATIONAL.csv" saved in the ***eduDataSet*** variable will look like the table in Figure 7.



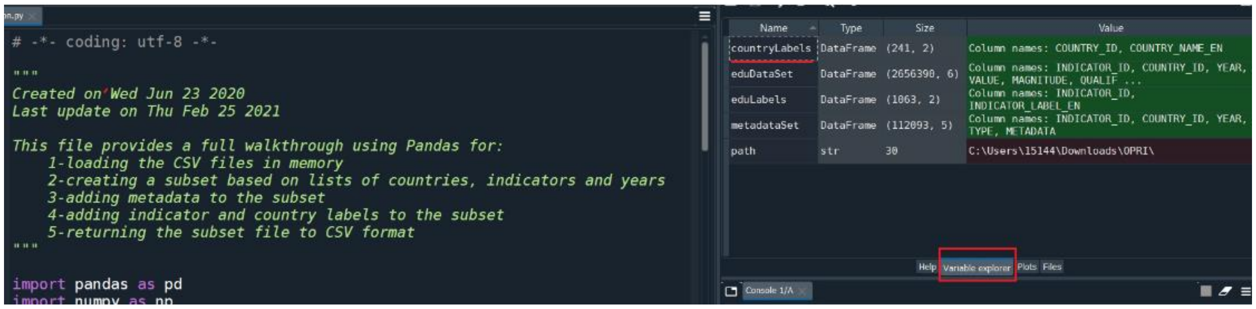
| Index | INDICATOR_ID | COUNTRY_ID | YEAR | VALUE | MAGNITUDE | QUALIFIER |
|---|---|---|---|---|---|---|
| 26715 | yadult.prof… | SWE | 2012 | 0.892 | nan | nan |
| 93940 | CR.1 | ZWE | 2015 | 88.2131 | nan | nan |
| 436759 | EA.1t8.Ag25… | ZWE | 2012 | 80.8751 | nan | nan |
| 482213 | EA.1t8.Ag25… | KWT | 2012 | 64.4829 | nan | nan |
| 783794 | EA.1t8.Ag25… | SWE | 2015 | 100 | nan | nan |
| 1032867 | EA.1t8.Ag25… | SWE | 2017 | 100 | nan | nan |
| 1422218 | EA.6t8.Ag25… | ZWE | 2012 | 3.39972 | nan | nan |
| 1612806 | EA.6t8.Ag25… | KWT | 2017 | 10.6672 | nan | nan |
| 1883640 | EA.6t8.Ag25… | SWE | 2015 | 22.9989 | nan | nan |

*Figure 7 an example of a Pandas' DataFrame*

Run the code from the beginning to the “Load CSV to memory” section (included) by selecting the lines and then pressing F9.

You can now see the CSV loaded in the **Variable explorer** tab (Figure 8) within the Spyder IDE. Double click on the **countryLabels** variable and the DataFrame it contains will pop-up in a new window.

Note that double-clicking on large DataFrame will take a long time to load e.g. the **eduDataSet** containing 2.7 million rows.



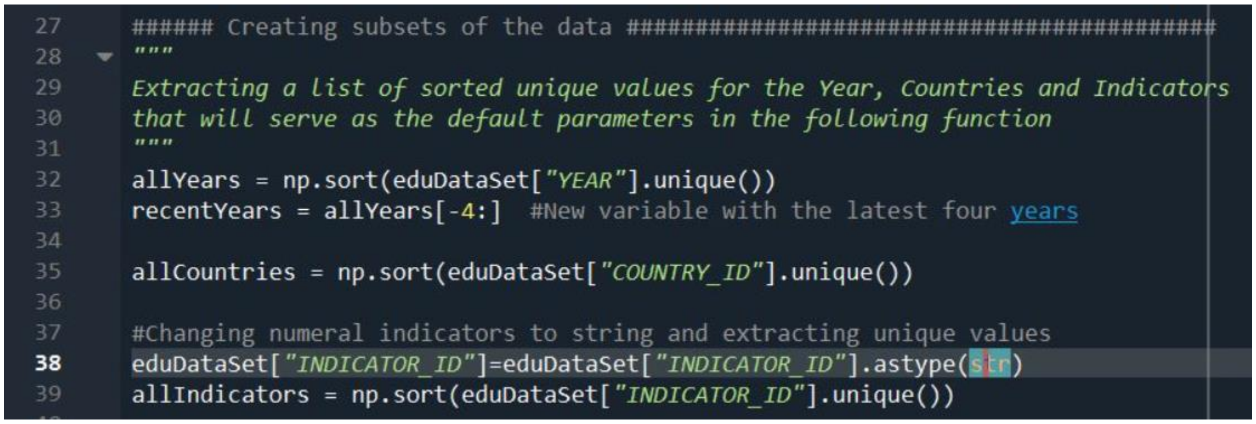
```
# -*- coding: utf-8 -*-

"""
Created on Wed Jun 23 2020
Last update on Thu Feb 25 2021

This file provides a full walkthrough using Pandas for:
    1-loading the CSV files in memory
    2-creating a subset based on lists of countries, indicators and years
    3-adding metadata to the subset
    4-adding indicator and country labels to the subset
    5-returning the subset file to CSV format
"""

import pandas as pd
import numpy as np
```

| Name | Type | Size | Value |
|---|---|---|---|
| countryLabels | DataFrame | (241, 2) | Column names: COUNTRY_ID, COUNTRY_NAME_EN |
| eduDataSet | DataFrame | (2656390, 6) | Column names: INDICATOR_ID, COUNTRY_ID, YEAR, VALUE, MAGNITUDE, QUALIF ... |
| eduLabels | DataFrame | (1063, 2) | Column names: INDICATOR_ID, INDICATOR_LABEL_EN |
| metadataSet | DataFrame | (112093, 5) | Column names: INDICATOR_ID, COUNTRY_ID, YEAR, TYPE, METADATA |
| path | str | 30 | C:\Users\15144\Downloads\OPRI\ |

*Figure 8 opening a variable containing a DataFrame in the Variable Explorer*

## Subsetting the data file

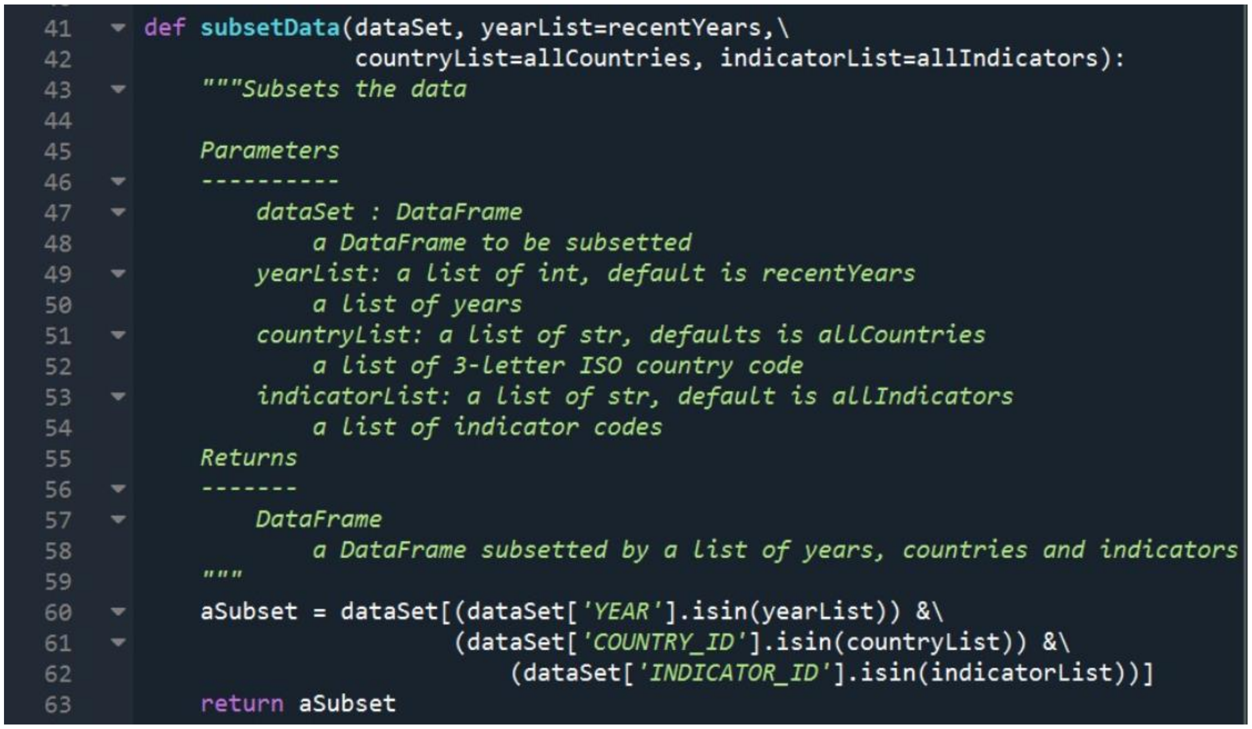
The second step is to extract a subset from a DataFrame based on lists of countries, years and indicators.



```
###### Creating subsets of the data ###########################################
"""
Extracting a list of sorted unique values for the Year, Countries and Indicators
that will serve as the default parameters in the following function
"""
allYears = np.sort(eduDataSet["YEAR"].unique())
recentYears = allYears[-4:]  #New variable with the latest four years

allCountries = np.sort(eduDataSet["COUNTRY_ID"].unique())

#Changing numeral indicators to string and extracting unique values
eduDataSet["INDICATOR_ID"]=eduDataSet["INDICATOR_ID"].astype(str)
allIndicators = np.sort(eduDataSet["INDICATOR_ID"].unique())
```

*Figure 9 extract and sort lists of unique values for years, countries and indicators*

The first lines (Figure 9) sort and save lists of the unique values for countries, years and indicators found in eduDataset. These lists will define the default parameters in the function named subsetData Figure 10.



```python
def subsetData(dataSet, yearList=recentYears,\
                 countryList=allCountries, indicatorList=allIndicators):
    """Subsets the data

    Parameters
    ----------
        dataSet : DataFrame
            a DataFrame to be subsetted
        yearList: a list of int, default is recentYears
            a list of years
        countryList: a list of str, defaults is allCountries
            a list of 3-letter ISO country code
        indicatorList: a list of str, default is allIndicators
            a list of indicator codes
    Returns
    -------
        DataFrame
            a DataFrame subsetted by a list of years, countries and indicators
    """
    aSubset = dataSet[(dataSet['YEAR'].isin(yearList)) &\
                       (dataSet['COUNTRY_ID'].isin(countryList)) &\
                          (dataSet['INDICATOR_ID'].isin(indicatorList))]
    return aSubset
```

*Figure 10 subset function*

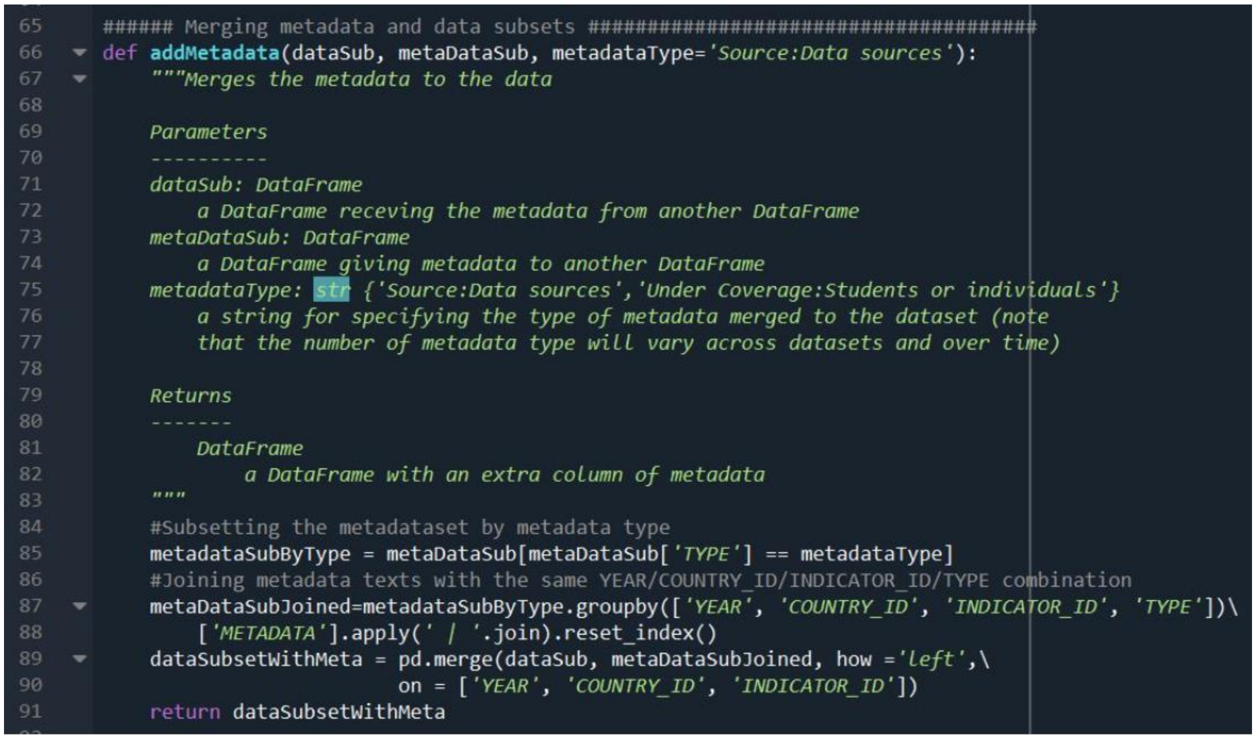
The subsetData function (Figure 10) takes in four arguments: dataSet, yearList, countryList and indicatorList. Left at their default parameters, the subset will contain the last four years of data for all countries and all indicators.

The full dataset OPRI_DATA_NATIONAL.csv starts with more than 2.7 million rows. The default parameters output a DataFrame containing around 250K rows providing a subset that fits within an Excel sheet.

Both the core data and the metadata should be subsetted to speed up the merging process although the speed gain will be marginal on smaller subsets.

An example will be provided showing how to change these parameters to get a smaller (or larger) subset. Before running these examples, we will define another function for merging the metadata to the subset.

## Merging the metadata to the data subset



```python
###### Merging metadata and data subsets ######################################
def addMetadata(dataSub, metaDataSub, metadataType='Source:Data sources'):
    """Merges the metadata to the data

    Parameters
    ----------
    dataSub: DataFrame
        a DataFrame receving the metadata from another DataFrame
    metaDataSub: DataFrame
        a DataFrame giving metadata to another DataFrame
    metadataType: str {'Source:Data sources','Under Coverage:Students or individuals'}
        a string for specifying the type of metadata merged to the dataset (note
        that the number of metadata type will vary across datasets and over time)

    Returns
    -------
        DataFrame
            a DataFrame with an extra column of metadata
    """
    #Subsetting the metadataset by metadata type
    metadataSubByType = metaDataSub[metaDataSub['TYPE'] == metadataType]
    #Joining metadata texts with the same YEAR/COUNTRY_ID/INDICATOR_ID/TYPE combination
    metaDataSubJoined=metadataSubByType.groupby(['YEAR', 'COUNTRY_ID', 'INDICATOR_ID', 'TYPE'])\
        ['METADATA'].apply(' | '.join).reset_index()
    dataSubsetWithMeta = pd.merge(dataSub, metaDataSubJoined, how ='left',\
                         on = ['YEAR', 'COUNTRY_ID', 'INDICATOR_ID'])
    return dataSubsetWithMeta
```

*Figure 11 merging the metadata function*

The **addMetadata** function (Figure 11) will take a dataset and merge it with its metadata.

The function takes in three arguments: dataSub, metadataSub and metadataType. Left at its default 3rd parameter, the function will add a column holding the data source to the subset.

The metadata is data point specific and the function will try matching a data point with the same YEAR/COUNTRY_ID/INDICATOR_ID combination within the metadata file.

Any data point can have zero, one or multiple metadata values associated with it. In the OPRI dataset, there are two types of metadata at the time of writing so a single data point could have a source or an under coverage entry associated with it. As such, it is important to run this function multiple time to get all the metadata types required for your needs. This function could also be modified to merge all metadata in one run.

Furthermore, a single datapoint could have multiple entries for the same type of metadata. This is also taken into account and when such a case occurs, the function will combine the entries within a single cell and each entries will be separated by the "|" symbol.

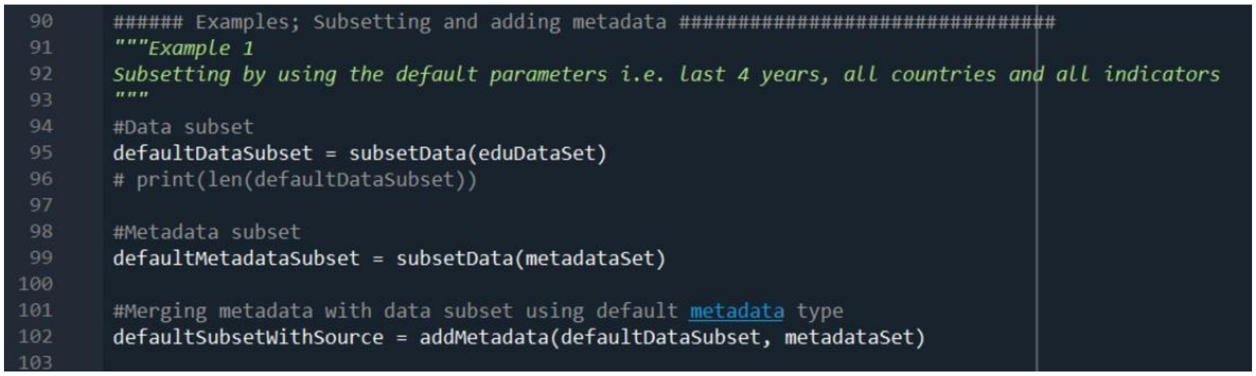
Run the code sections from "Creating subsets of the data" to "Merging metadata and data subsets" (included) by selecting the lines and then pressing F9. This will save the new functions to memory.

## Example subsetting and merging the metadata to the subset

So far, we have constructed two functions that allow to subset the core data set and then match the metadata to each data point of the subset.

The next blocks of code provide examples running these two functions.

### Example 1, running the subsetData function with the default parameters



```
###### Examples; Subsetting and adding metadata ################################
"""Example 1
Subsetting by using the default parameters i.e. last 4 years, all countries and all indicators
"""
#Data subset
defaultDataSubset = subsetData(eduDataSet)
# print(len(defaultDataSubset))

#Metadata subset
defaultMetadataSubset = subsetData(metadataSet)

#Merging metadata with data subset using default metadata type
defaultSubsetWithSource = addMetadata(defaultDataSubset, metadataSet)

```

*Figure 12 executing the subset and metadata functions using the default parameters*

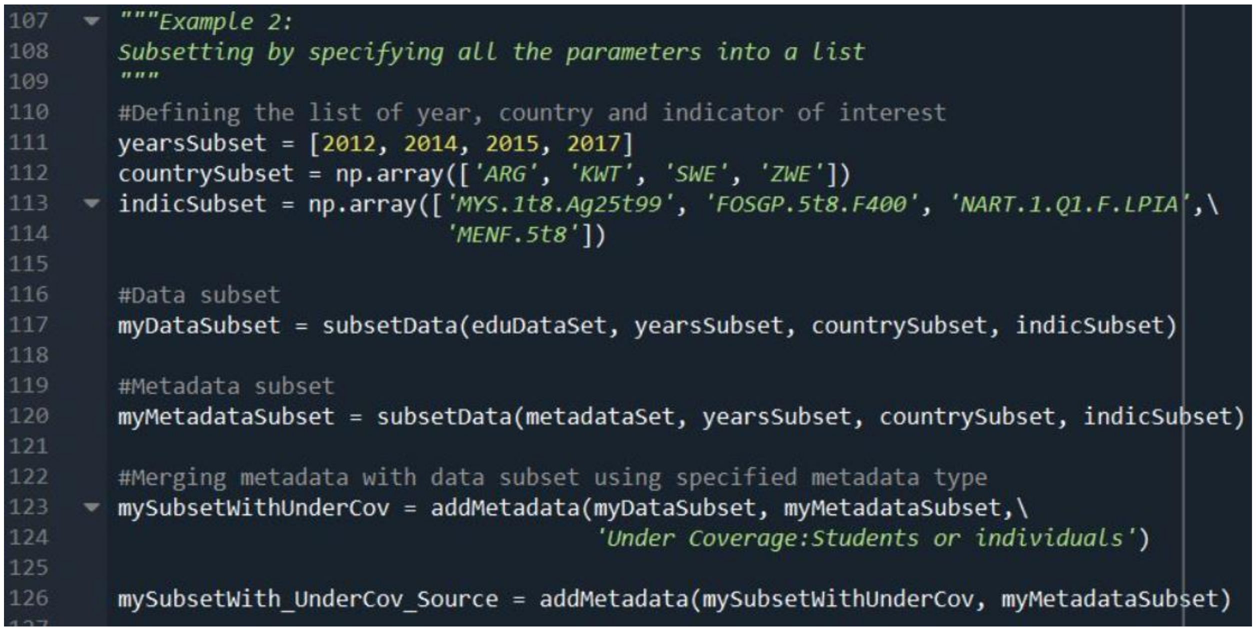
In this example (Figure 12), we only need to specify the dataset to be subsetted since we will otherwise run the **subsetData** function on its default parameters. This means we will extract the last 4 years of data, all indicators and all countries for both the core data set and the metadata set.

Next, we use the **addMetadata** function specifying the data subset and the metadata subset as the first two parameters. For the purpose of this example, we do not specify the third parameter since we will merge the data with the default metadata type, the source of the data.

RUN "Example 1" (Figure 12) by selecting the lines and then pressing F9.

This will save the DataFrames to variables (defaultDataSubset, defaultMetadataSubset, defaultSubsetWithSource) which can be visualized in the variable explorer tab.

### Example 2, running the subsetData function with custom parameters



```python
"""Example 2:
Subsetting by specifying all the parameters into a list
"""
#Defining the list of year, country and indicator of interest
yearsSubset = [2012, 2014, 2015, 2017]
countrySubset = np.array(['ARG', 'KWT', 'SWE', 'ZWE'])
indicSubset = np.array(['MYS.1t8.Ag25t99', 'FOSGP.5t8.F400', 'NART.1.Q1.F.LPIA',\
                        'MENF.5t8'])

#Data subset
myDataSubset = subsetData(eduDataSet, yearsSubset, countrySubset, indicSubset)

#Metadata subset
myMetadataSubset = subsetData(metadataSet, yearsSubset, countrySubset, indicSubset)

#Merging metadata with data subset using specified metadata type
mySubsetWithUnderCov = addMetadata(myDataSubset, myMetadataSubset,\
                                   'Under Coverage:Students or individuals')

mySubsetWith_UnderCov_Source = addMetadata(mySubsetWithUnderCov, myMetadataSubset)
```

*Figure 13 executing the subset and metadata functions using custom parameters*

Figure 13 shows how the function works with all the parameters and the parameters' inputs explicitly stated.

First, custom lists of years, countries and indicators are specified and saved in variables (yearsSubset, countrySubset and indicSubset). These variables are then passed in the **subsetData** function as the second, third and fourth parameters respectively.

Once the dataset and the metadata set have been subsetted, the code merges the metadata and data subsets a first time with the **addMetadata** function. The third parameters is a string specifying 'Under Coverage: Students or individuals' as the type of metadata merged to the subset.

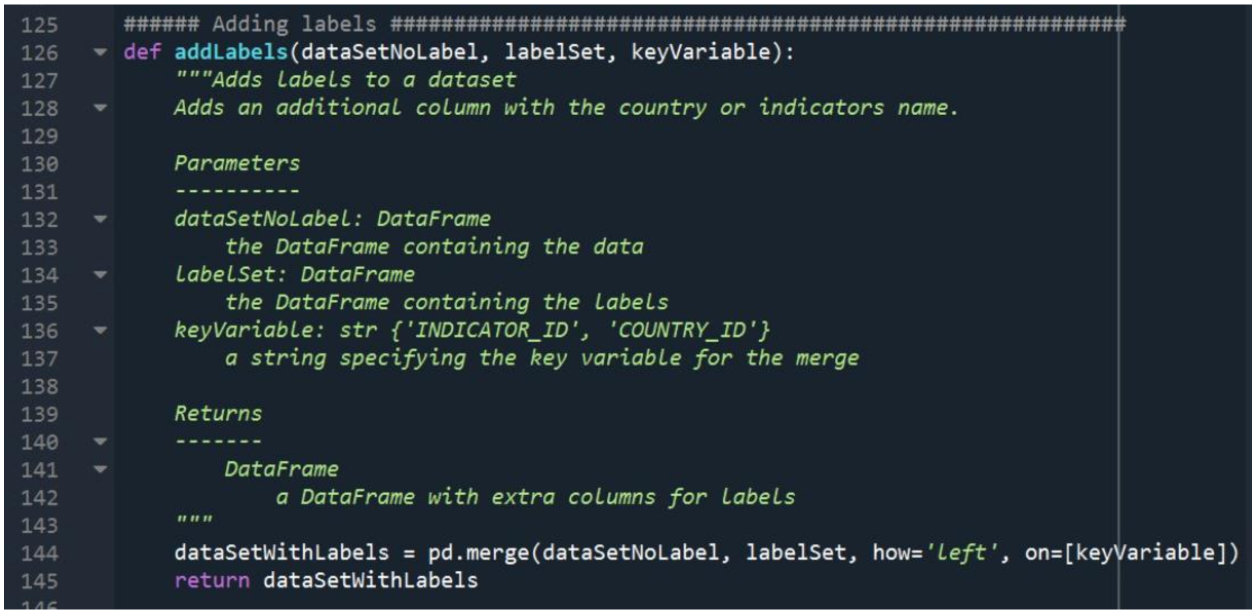
To further the example, the **addMetadata** is then used a second time to add the default metadata type (data source).

The final output is a subset with extra columns with both the source and the under-coverage metadata.

Run "Example 2" (Figure 13) section by selecting the lines and then pressing F9.

This will save the DataFrames to variables.

## Adding labels



```python
###### Adding labels ##############################################################
def addLabels(dataSetNoLabel, labelSet, keyVariable):
    """Adds labels to a dataset
    Adds an additional column with the country or indicators name.

    Parameters
    ----------
    dataSetNoLabel: DataFrame
        the DataFrame containing the data
    labelSet: DataFrame
        the DataFrame containing the labels
    keyVariable: str {'INDICATOR_ID', 'COUNTRY_ID'}
        a string specifying the key variable for the merge

    Returns
    -------
        DataFrame
            a DataFrame with extra columns for labels
    """
    dataSetWithLabels = pd.merge(dataSetNoLabel, labelSet, how='left', on=[keyVariable])
    return dataSetWithLabels
```

*Figure 14 adding labels function with an example*

Before converting back the subset with metadata to CSV, the label for the country and indicator can be attached to the subset. This function will merge the labels from the label dataset to the subset.

The function (Figure 14) takes in three arguments and has no default parameters. The first and second arguments are for specifying the DataFrames for the subset and label tables. The third argument indicates the key on which to merge the DataFrames.

To get both countries and indicators labels you will need to run the function twice.

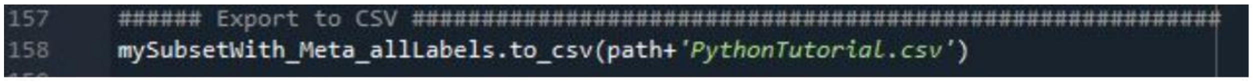
The output is a dataset with extra columns with those labels.

Run the "Adding labels" section (function and example) by selecting the lines and then pressing F9.

This will save the function in memory and run the examples for merging labels.

The function is run twice to include both country and indicator labels. You can now visualize the DataFrame with labels in the Variable Explorer.

## Exporting subset to CSV



```python
###### Export to CSV ##############################################################
mySubsetWith_Meta_allLabels.to_csv(path+'PythonTutorial.csv')
```

*Figure 15 exporting the final subset to CSV*

Lastly, we export the subset back to CSV (Figure 15). There are many options for what we could do with the subset but, for the sake of this tutorial, we assume that the user will want to do some analysis in Excel.

## Modify the file name in the code to a name of your choice and run the last line of code to export the final subset to CSV.

You now have a subset with the metadata and labels included. The subset is saved in the folder specified in the "Input files" section at the beginning of the code.