Guide for calculation of BoPs consumption intensities

Jorge Ortigosa – 27 June 2020 draft version.

NOTE : The final version of this guide belongs to the EU-JRC and therefore I am not allowed to disclose it here. What follows is just one of the many previous drafts.

1.INTRODUCTION

The work done so far (AA LCIND2 and previous projects) focuses on the calculation of the Consumer Footprint (CF) at the EU level[1], meaning that only the average consumption of an average EU citizen is analysed. The consumption pattern of an average EU citizen considers the apparent consumption at the EU level, i.e. the consumption intensity of each representative product is calculated as:

$$Apparent\ consumption(AC) = Domestic\ production + Imports - Exports$$

However, due to different cultural and geographical backgrounds the consumption patterns of EU citizens can vary between Member States. Cultural differences might affect food consumption due to culinary practices, historically available products, and country-specific trends. Geographical characteristics can affect consumption intensity patterns, such as the energy demand for heating households. Within the AA Consumption Footprint, Member State-specific Consumer Footprint have been assessed by calculating the specific consumption intensities for each country.

In order to do this, since repeating the procedure followed so far would imply manually building all five BoPs for each country considered, i.e. retrieving and organizing the data, which would require a huge amount of time, it was decided to automatize this process using Python programming language to create a number of scripts and functions which would for each group of representative products or basket of products (BoP) and country retrieve the required data and calculate the apparent consumption. This endeavor has not been exempt of obstacles, such as getting automatically access to the multiple data sources used in the BoPs, harmonizing the different formats and naming systems these sources provided the information in, dealing with the data gaps most of these sources show or implement the numerous exceptions and particular cases that render any automatization effort more complicated.

[1] EU level means that it considers the Member States of the European Union at the time of assessment, e.g. 2010 assessment considers EU-27, while 2015 considers already EU-28 (with the addition of Croatia).

2.BOPs

Ideally, it should be possible to apply the same set of functions to build up all the BoPs and calculate their AC, rather than writing specific functions for each of them. Unfortunately, the differences among them rendered this task impossible. However, some of the BoPs have some elements in common, such as the data sources (see table) used or the calculations performed. This made it possible to write a unique set of functions that could be used to build them all and calculate their respective ACs. As for the remaining two BoPs, Mobility and Housing, specific sets of functions were developed to build each of them.

Altogether, a total of 7 scripts were written, which were applied to build the different BoPs and calculate their ACs as described in the table below:

Table 1. Data sources, developed scripts and goal of each script, by BoP.

BoP	Data Sources	Scripts	Goal
HHGoods, Food, Appliances	EUROSTAT(PRODCOM/COMEXT), FAO	*data_inputs*	Read and filter excel and csv files; extract yearly population figures by country
		predownload	Support the extraction of data from the different data sources.
		postdownload	Perform operations on the raw data to prepare and harmonize it with the chosen naming system before further operations
		calculations	Perform the calculations required to obtain the apparent consumption.
		pcomext	Retrieve the required data from PRODCOM and COMEXT.
		fao	Retrieve the required data from FAO.
Mobility	EUROSTAT (transport data), EU Pocketbook	*bop_mobility*	Retrieve the required data, filters it and performs both, intermediate and final calculations
Housing	EU building database, EUROSTAT		

3.The Scripts

3.1 BOP HH, FOOD & APP

3.1.1 Inputs

In order to build these BoPs and calculate the respective ACs through the above mentioned scripts two main inputs are required: data from the selected data sources and information on how to treat that data for each particular case.

3.1.2 Data Sources:

The raw material these three BoPs are built on is data from two different data sources: EUROSTAT[2] and FAO[3]. Within EUROSTAT, the required data comes from PRODCOM (DS-066341) and COMEXT (DS-016890) datasets which contain data on production and trade. FAO data, which is only necessary for BoP Food comes from: *Food Balance: Food Supply - Crops Primary Equivalent* (4), *Food Balance: New Food Balances* (19), *Food Balance: Food Balances (old methodology and population)* (20), *Production: Crops* (54), *Production: Livestock Primary* (57) and *Trade: Crops and livestock products* (71).

EUROSTAT data is retrieved, for each country and period of interest, via EUROSTAT SDMX service by means of python *eurostat* package[4]. As for FAO, the data of the mentioned datasets will be downloaded at once for all countries and years, via FAO Fenixservice bulkdownload service[5] by means of python *urllib.request* package within the *urllib* library[6].

It may be worth it to mention that, despite both belong to EUROSTAT, PRODCOM and COMEXT have different products codes and country names and codes systems. For instance, PRODCOM contains data for “Luxembourg”, which country code is “018”, whilst COMEXT contains data for “Luxemburg”, which country code is “LU”.

In order to unify the different formats, column headers, products codes and country codes the different datasets show and make it possible to bring them together, it was decided to use PRODCOM ones, with the exception of FAO products and groups codes, which will remain unchanged.

[2] https://ec.europa.eu/eurostat
[3] http://www.fao.org/home/en/
[4] https://pypi.org/project/eurostat/
[5] http://fenixservices.fao.org/
[6] https://pypi.org/project/urllib3/

3.1.3 Inventory file

In addition to the data sources, the other key input to run the functions is an inventory of all the products, representative products and product groups in each of the BoPs, along with information on how to perform the double counting and the upscaling for each of them for instance.

This inventory is saved in an excel file that is read in the first steps of the building process. Being an external document to the scripts in a very common format, makes it possible for any user to modify it at will, adding products or changing coefficients for instance.

An inventory example can be found in the Annex. In the table below an explanation of the information contained in each of its columns may be found.

Table 2. Fields and descriptors of the inventory file.

Column	Information
BoP	The BoP the product belongs to. Use the following abbreviations: *bop_hg* (BoP Household Goods), *bop_app* (BoP Appliances) and *bop_food* (BoP Food)
prod_groups	The product group the product belongs to. Use the abbreviations, as indicated in the Annex.
source	The origin of the data of the product. It can take two values, either EUROSTAT or FAO.
FAO_group	For the BoP Food products, this is the code of the group they belong to in FAO database. For the other BoPs products it will remain empty.
FAO_datasets	For the BoP Food products, this is the number of the FAO dataset within the FAO dataset dictionary, the data of the product can be found in. For the other BoPs products it will remain empty.
prod_codes	The product code in the data source indicated in column *source*.
prod_names	The product name.
rep_prod_names	The names of the representative products. There is no relation between a representative product and the product in the *prod_names* columns on its left.
components	The products (PRODCOM CODES) a representative product comprehends referred to by its product code.
split	For some product groups, different PRODCOM codes represent a number of representative products and the sum of PRODCOM values should be divided into RPS, e.g. leisure footwear and fashion footwear. The number the representative products quantities must be divided by the given split value.
EXP_COEF	In specific cases, the representative products cannot be related to PRODCOM codes and the sum of the product group must be divided by coefficients to divide the values among the different representative products, e.g. detergents. Coefficient the representative product exports values have to be multiplied by.
IMP_COEF	In specific cases, the representative products cannot be related to PRODCOM codes and the sum of the product group must be divided by coefficients to divide the values among the different representative products, e.g. detergents. Coefficient the representative product import values have to be multiplied by.
PROD_COEF	In specific cases, the representative products cannot be related to PRODCOM codes and the sum of the product group must be divided by coefficients to divide the

	values among the different representative products, e.g. detergents. Coefficient the representative product production values have to be multiplied by.
Lifespan	The lifespan of the representative product.
DC_PROD	The representative products which are used to calculate the quantity to be subtracted due to double counting. Empty if no subtraction due to double counting is to be applied.
DC_COEF	The percentages of the representative products in *DC_PROD* that will be subtracted due to double counting. It is 1 if no subtraction due to double counting is to be applied.
DC_REP_PROD	It is 1 if the product has to undergo the double counting and 0 when it does not.
FBS_UPS	This column indicates when the upscaling has to be done using a different data source than PRODCOM. It is 1 for the representative products for which the upscaling must be done using the FAO Food Balance Sheets (FBS) product group quantity. It is 0 for the representative products for which the upscaling must be done using the product group quantity calculated through the sum of the single products quantities.

3.1.4 PRODCOM-COMEXT codes table

As it was already mentioned, although PRODCOM and COMEXT data refers, with minor differences, to the same products, they do not share the same products codes system. This means that a single product would have at least two different codes one for PRODCOM and one for COMEXT, although, in some cases the correspondence is not one-to-one i.e. the product represented by one PRODCOM code may correspond to more than one product in COMEXT and thus to more than one COMEXT code, or the other way around. In addition to this, these correspondences are not static but vary over the years. For the sake of simplicity it was decided to use PRODCOM naming system in every case, i.e. the products codes, the countries, the indicators, etc.

This means, in the case of the product codes, that, in several occasions along the process of retrieving and processing data COMEXT codes will have to be converted to PRODCOM codes and the other way around. In order to do so we need a table with the correspondences between both products' codes systems for each year.

Table 3. Example of PCOMEXT file.

PCOM	CN	Year
36637790	96180000	2003
399900Z0	94060090	2003
15111140	2011000	2004
15111140	2012020	2004
15111140	2012030	2004
.....	.....	.....

3.1.5 Scripts

Information about each function workings, inputs and outputs can be found in the functions themselves. Here a brief description of each of them will be provided along with some comments.

data_inputs.py

This script contains four functions: *read_file*, *filter_file*, *inventory_dtype* and *population*. In general, they serve the purpose of loading additional information from sources and formats different from the main sources. The first three are focused on excel and csv files whilst the fourth one extracts information on population per country and year from EUROSTAT.

`read_file`

Arg.:(file_name, file_folder, skiprows, sheetname, dtype_dict)

Provided the file name (including the extension), folder path, first row with data, sheet name (or position beginning from 0), and the data type each column of the file contains it loads all the information contained in a .xlsx or .csv in the shape of a data frame.

In [11]:
```
dtype_dict = {"prod_codes" : str, "EXP_COEF" : float, "IMP_COEF" : float, "PROD_COEF" : float, "components" : str,
              "split" : float, "DC_COEF" : str, "DC_PRODS" : str, "FBS_UPS" : float}

inventory = read_file("inventory.xlsx", INPUTS_FOLDER, 4, 0, dtype_dict)

inventory.head()
```

Out[11]:

	BoP	prod_groups	Source	FAO_group	FAO_datasets	prod_codes	prod_names	rep_prod_names	components	split	EXP_COEF	IMP_COEF	PROD_
0	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421150	Perfumes	NaN	NaN	NaN	1.0	1.0	
1	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421170	Toilet waters	NaN	NaN	NaN	1.0	1.0	
2	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421250	Lip make-up preparations	NaN	NaN	NaN	1.0	1.0	
3	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421270	Eye make-up preparations	NaN	NaN	NaN	1.0	1.0	
4	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421300	Manicure or pedicure preparations	NaN	NaN	NaN	1.0	1.0	

Figure 1: `read_file(inventory)`

The *dtype_dict* variable tells the function what kind of data will be found in each column. If no information is provided the function tries to recognize the types of data itself, so only in the cases a type of data different from the obvious one is to be assigned to one column, should it be explicitly included in the dictionary. In the examples above, for reasons that will be explained later on, *prod_codes*, *Years*, *components*, *DC_COEF* and *DC_PRODS* were all assigned the type `str` despite of being integer numbers.

In [25]:
```
dtype_dict = {"PCOM" : str, "CN" : str, "Year" : str}

pcomext_table = read_file("pcomext_table.xlsx", INPUTS_FOLDER, 0, 0, dtype_dict)

pcomext_table.head()
```

Out[25]:

	PCOM	CN	Year	Unnamed: 5
0	15111140	2011000	1995	NaN
1	15111140	2012020	1995	NaN
2	15111140	2012030	1995	NaN
3	15111140	2012050	1995	NaN
4	15111190	2012090	1995	NaN

Figure 2 : `read_file(pcomext_table)`

`filter_file`
Arg.:(items,input_df)

Given a data frame and some items of interest, the function will look for the item of interest in all the columns from left to right until it finds the first column containing the given item. Then it will produce a data frame which only contains the rows of the provided data frame which in that given column have the input item.

In [19]:
```
filtered_inventory = filter_file(["bop_app"], inventory)

filtered_inventory.head()
```

```
The provided items are in columns: ['BoP']
```

Out[19]:

	BoP	prod_groups	Source	FAO_group	FAO_datasets	prod_codes	prod_names	rep_prod_names	compone
445	bop_app	PG_Dishwasher	EUROSTAT	NaN	NaN	27511200	Household dishwashing machines	RP_Dishwasher1	27511
446	bop_app	PG_Dishwasher	EUROSTAT	NaN	NaN	27511200	Household dishwashing machines	RP_Dishwasher2	27511
447	bop_app	PG_WashingDryer	EUROSTAT	NaN	NaN	27511300	Cloth washing and drying machines, of the hous...	RP_WashingDryer1	27511
448	bop_app	PG_WashingDryer	EUROSTAT	NaN	NaN	27511300	Cloth washing and drying machines, of the hous...	RP_WashingDryer2	27511
449	bop_app	PG_Fridge	EUROSTAT	NaN	NaN	27511110	Combined refrigerators-freezers, with separate...	RP_Fridge	27511110;27511133;27511135;27511150;27511

Figure 3 : `filter_file`

As it can be seen the given item was found in column *BoP* and the resulting data frame only contains the rows for which the value in the mentioned column is *bop_app*.

`inventory_dtype`
Arg.: None

It simply produces a dictionary with the inventory file columns data types. This way it is not necessary to write the whole data type dictionary every time the inventory file is loaded. If the column data types are to be changed this will have to be done within the function itself.

```
In [21]:
1 dtype_dict = inventory_dtype()
2
3 inventory = read_file("inventory.xlsx", INPUTS_FOLDER, 4, 0, dtype_dict)
4
5 inventory.head()
```

Out[21]:

	BoP	prod_groups	Source	FAO_group	FAO_datasets	prod_codes	prod_names	rep_prod_names	components	split	EXP_COEF	IMP_COEF	PROD_
0	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421150	Perfumes	NaN	NaN	NaN	1.0	1.0	
1	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421170	Toilet waters	NaN	NaN	NaN	1.0	1.0	
2	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421250	Lip make-up preparations	NaN	NaN	NaN	1.0	1.0	
3	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421270	Eye make-up preparations	NaN	NaN	NaN	1.0	1.0	
4	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421300	Manicure or pedicure preparations	NaN	NaN	NaN	1.0	1.0	

Figure 4 : `inventory_dtype`

p`opulation`
Arg.:(start_year, end_year)

Given a start year and an end year it retrieves data from EUROSTAT *demo_r_d2jan* dataset and produces a data frame containing the population of all European countries in the years within the given period. The list of countries it retrieves the data for is within the function and can only be modified there.

```
In [23]:
1 population("2010", "2012")
```

Out[23]:

	DECL	Population	Year
0	Austria	8351643.0	2010
1	Austria	8375164.0	2011
2	Austria	8408121.0	2012
3	Belgium	10839905.0	2010
4	Belgium	11000638.0	2011
...	...	...	...

Figure 5 : `population`

predownload.py

This script contains five functions which perform a number of operations that take place before downloading the data from EUROSTAT (for FAO data a different set of functions is used). These functions are therefore not to be run alone but within other functions.

ds_conversion
Arg.:(DS, col)

As it was indicated previously, PRODCOM data set naming systems and format was adopted as a reference, so that the other data sources will have to adapt to it. This function provides the information required to do so, including the correspondence between PRODCOM, COMEXT and FAO countries naming systems, units' conversion factors, columns names correspondences etc.

Having all this information in the same place, from which other functions could retrieve whatever they need, would allow anybody to make changes without having to dig into many different functions.

```
In [5]: 1 ds_conversion("DS-016890", "COUNTRY_CHANGE")

Out[5]: {'Belgium': 'Belgium (And Luxbg -> 1998)',
         'Germany': 'Germany (Incl Dd From 1991)',
         'Czechia': 'Czechia (Cs->1992)',
         'Luxemburg': 'Luxembourg',
         'EUROPEAN UNION (28)': 'Eu28_Intra'}
```

Figure 6 : ds_conversion(COUNTRY_CHANGE)

```
In [6]: 1 ds_conversion("FAO", "UNIT")

Out[6]: {'CONSQNT': 1000,
         'PRODQNT': 1000,
         'IMPQNT': 1000,
         'EXPQNT': 1000,
         'EXPVAL': 1,
         'IMPVAL': 1}
```

Figure 7 : ds_conversion(UNITS)

In Fig.6 the function produces a dictionary with the correspondence between PRODCOM and COMEXT countries naming systems. In Fig.7, the result is a dictionary with the units' conversion factors to be applied to each indicator in the data extracted from FAO datasets.

check_repeated
Arg.:(elements_list)

Return the repeated elements in a list.

check_belong
Arg.:(elements_list, database)

Return the elements in a list that do not belong to a given database.

check_dataset
Arg.:(elements_list, database)

Applies **check_repeated** and **check_belong.**

ccode
Arg.:(countries_list, DS)

In both, PRODCOM and COMEXT have a coding system for the countries, for instance, in PRODCOM "France" is referred with the code 001 and in COMEXT with the code FR. This function translates country codes to country names and the other way around.

```
In [8]:  1 ccode(["Italy", "France", "Spain"], "DS-066341")
Out[8]: ['005', '001', '011']

In [9]:  1 ccode(["005", "001", "011"], "DS-066341")
Out[9]: ['Italy', 'France', 'Spain']
```

Figure 8 : ccode

pcomext.py

The goal of the functions contained in this script is locating and downloading EUROSTAT's datasets data.

search_one_word
Arg.:(word, products_dict)

If a dictionary composed by products codes and descriptions is provided as argument, it extracts those product codes which corresponding description includes a given word.

```
In [10]:  1 products_dict = {"10111230" : "Fresh or chilled carcases and half-carcases, of pig meat",
          2                  "10113230" : "Frozen carcases and half-carcases, of pig meat",
          3                 "10115060" : "Lard and other pig fat",
          4                 "10202450" : "Smoked herrings (including fillets)",
          5                 "10201600" : "Frozen fish livers and roes",
          6                 "10513070" : "Dairy spreads of a fat content by weight < 80%"}
          7
          8 search_one_word("meat", products_dict)
Out[10]: {'10111230': 'fresh or chilled carcases and half-carcases, of pig meat',
          '10113230': 'frozen carcases and half-carcases, of pig meat'}
```

Figure 9 : search_one_word

search_many_words
Arg.:(words_list, DS)

From a given EUROSTAT dataset, it extracts all those products codes which corresponding description includes number of words.

```
In [13]:  1 search_many_words(["pig", "meat"], "DS-066341")
          2

{'10111230': 'fresh or chilled carcases and half-carcases, of pig meat (including fresh meat packed with salt as a temporary pr
eservative)', '10111250': 'fresh or chilled hams, shoulders and cuts thereof with bone in, of pig meat (including fresh meat pa
cked with salt as a temporary preservative)', '10111290': 'fresh or chilled pig meat (including fresh meat packed with salt as
a temporary preservative; excluding carcases and half-carcases, hams, shoulders and cuts thereof with bone in)', '10113230': 'f
rozen carcases and half-carcases, of pig meat', '10113250': 'frozen hams, shoulders and cuts with bone in, of pig meat', '10113
290': 'frozen pig meat (excluding carcases and half-carcases, hams, shoulders and cuts thereof with bone in)', '10131300': 'mea
t salted, in brine, dried or smoked; edible flours and meals of meat or meat offal (excluding pig meat, beef and veal salted, i
n brine, dried or smoked)'}

Out[13]: ['10111230',
          '10111250',
          '10111290',
          '10113230',
          '10113250',
          '10113290',
          '10131300']
```

Figure 10 : search_many_words

check_pcomext
Arg.:(products, countries, pcomext_table_year, DS)

It checks whether a list of products and a list of countries have elements repeated or belong to a given EUROSTAT dataset. Both the products and the countries have to be provided in PRODCOM nomenclature. If this dataset is not PRODCOM, the function also converts both the products and the countries to the corresponding dataset nomenclature and, in the case of the countries, to the corresponding country codes system.

```
In [35]:  1 check_pcomext(["10111140", "10111190"], ["Spain", "Italy"], pcomext_table_2010, "DS-016890")

Out[35]: (['2011000', '2012020', '2012030', '2012050', '2012090', '2013000'],
          ['ES', 'IT'])
```

Figure 11 : check_pcomext

In the example, notice that for 2010, the two PRODCOM products codes were assigned a total of 6 COMEXT product codes, i.e. to some PRODCOM products correspond more than one COMEXT product.

get_pcomext
Arg.:(start_year, end_year, product_codes, countries, pcomext_table, DS)

From the given EUROSTAT dataset (either PRODCOM or COMEXT), it extracts the information of the products provided for the years and countries specified. The results are provided following the particular dataset nomenclature.

```
In [45]: 1 get_pcomext("2010","2010", ["10111140", "10111190"], ["Spain", "Italy"], pcomext_table, "DS-066341")
```

```
Progress: 100.0%
```

Out[45]:

	INDICATORS	DECL	PRCCODE	FREQ	2010
0	EXPQNT	011	10111140	A	7.800590e+07
1	EXPVAL	011	10111140	A	2.424099e+08
2	IMPQNT	011	10111140	A	3.362440e+07
3	IMPVAL	011	10111140	A	1.158349e+08
4	PRODQNT	011	10111140	A	4.606080e+08
5	PRODVAL	011	10111140	A	6.677179e+08
6	EXPQNT	005	10111140	A	4.650670e+07
7	EXPVAL	005	10111140	A	1.143386e+08

Figure 12 : `get_pcomext`

postdownload.py

The functions in this script are to be applied to the raw data retrieved from the different datasets and their goal is to transform it into a common format (see table below) so that the data from the different datasets can be used together.

DECL	PRCCODE	Year	EXPVAL	EXPQNT	IMPVAL	IMPQNT	PRODVAL	PRODQNT
Belgium	14111000	2010	NaN	52003540	NaN	79676300	NaN	80234670
...	...	...	...	...	...	...	...	...

Table 4 : harmonized common format

As above mentioned, PRODCOM features were adopted to as the common ones, thus “DECL” and “PRCCODE” are PRODCOM data headers for countries and product codes, whilst the rest, except for “Year”, which is new, are PRODCOM indicators. The country names in “DECL” and the product codes in “PRCCODE” also follow PRODCOM naming system.

The specific information to do these changes for the data coming from each of the datasets considered will be accessed by the functions through the **predownload.py** function **`ds_conversion`.**

`rename`

`Arg.:(input_df, DS)`

Turns the original headers of the input data frame coming from the given dataset into the corresponding ones in PRODCOM headers system.

```
In [11]:  1 raw_trade_df.head()
```

Out[11]:

	PARTNER	FLOW	INDICATORS	PRODUCT	REPORTER	FREQ	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	BE	1	QUANTITY_IN_100KG	33030010	AT	A	400.0	350.0	362.0	291.0	370.0	407.0	561.0	437.0	171.0
1	BE	1	QUANTITY_IN_100KG	33030010	BE	A	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	BE	1	QUANTITY_IN_100KG	33030010	BG	A	4.0	10.0	15.0	21.0	36.0	46.0	36.0	29.0	4.0
3	BE	1	QUANTITY_IN_100KG	33030010	CY	A	12.0	28.0	5.0	7.0	20.0	48.0	41.0	42.0	10.0
4	BE	1	QUANTITY_IN_100KG	33030010	CZ	A	106.0	106.0	29.0	35.0	158.0	535.0	853.0	182.0	1320.0

```
In [10]:  1 postdw.rename(raw_trade_df, "DS-016890")
          2 trade1_df.head()
```

Out[10]:

	DECL	FLOW	INDICATORS	PRCCODE	REPORTER	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	BE	1	QUANTITY_IN_100KG	33030010	AT	400.0	350.0	362.0	291.0	370.0	407.0	561.0	437.0	171.0
1	BE	1	QUANTITY_IN_100KG	33030010	BE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	BE	1	QUANTITY_IN_100KG	33030010	BG	4.0	10.0	15.0	21.0	36.0	46.0	36.0	29.0	4.0
3	BE	1	QUANTITY_IN_100KG	33030010	CY	12.0	28.0	5.0	7.0	20.0	48.0	41.0	42.0	10.0

Figure 13 : `rename`

`aggregate`

Arg.:(input_df, DS)

Aggregates the information in the input data frame according to a set of columns which depends on the dataset the data comes from. For instance, if, for the data coming from a given dataset, the aggregation is to be made for columns DECL, PRCCODE and Year, that means that the values displayed in rows sharing the same DECL, PRCCODE and Year will be added up.

```
In [10]:  1 trade1_df.head()
```

Out[10]:

	DECL	FLOW	INDICATORS	PRCCODE	REPORTER	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	BE	1	QUANTITY_IN_100KG	33030010	AT	400.0	350.0	362.0	291.0	370.0	407.0	561.0	437.0	171.0
1	BE	1	QUANTITY_IN_100KG	33030010	BE	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	BE	1	QUANTITY_IN_100KG	33030010	BG	4.0	10.0	15.0	21.0	36.0	46.0	36.0	29.0	4.0
3	BE	1	QUANTITY_IN_100KG	33030010	CY	12.0	28.0	5.0	7.0	20.0	48.0	41.0	42.0	10.0
4	BE	1	QUANTITY_IN_100KG	33030010	CZ	106.0	106.0	29.0	35.0	158.0	535.0	853.0	182.0	1320.0

```
In [12]:  1 trade2_df = postdw.aggregate(trade1_df, "DS-016890")
          2 trade2_df.head()
```

Out[12]:

	DECL	FLOW	INDICATORS	PRCCODE	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	BE	1	QUANTITY_IN_100KG	33030010	10663.0	15099.0	25666.0	45444.0	29257.0	21133.0	22526.0	25515.0	23163.0
1	BE	1	QUANTITY_IN_100KG	33030090	48551.0	40142.0	23824.0	28018.0	31000.0	30431.0	32367.0	30358.0	32851.0
2	BE	1	QUANTITY_IN_100KG	33041000	5401.0	6106.0	7306.0	11117.0	12943.0	13687.0	15923.0	30749.0	27849.0
3	BE	1	QUANTITY_IN_100KG	33042000	5362.0	6142.0	12395.0	11985.0	13644.0	16716.0	21120.0	32679.0	31471.0
4	BE	1	QUANTITY_IN_100KG	33043000	4495.0	7884.0	7529.0	7542.0	9757.0	14676.0	15747.0	14824.0	16593.0

Figure 14 : `aggregate`

combine_cols
Arg.:(input_df, DS)

In some cases, there is no direct correspondence between a dataset's columns and those of PRODCOM. Sometimes the same information that in PRODCOM data is displayed in just one column may be divided into two or three columns in other datasets data frames.

This function combines those columns to create a new one that corresponds to the column in PRODCOM data frames.

```
In [12]:
1 trade2_df.head()
```

Out[12]:

	DECL	FLOW	INDICATORS	PRCCODE	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	BE	1	QUANTITY_IN_100KG	33030010	10663.0	15099.0	25666.0	45444.0	29257.0	21133.0	22526.0	25515.0	23163.0
1	BE	1	QUANTITY_IN_100KG	33030090	48551.0	40142.0	23824.0	28018.0	31000.0	30431.0	32367.0	30358.0	32851.0
2	BE	1	QUANTITY_IN_100KG	33041000	5401.0	6106.0	7306.0	11117.0	12943.0	13687.0	15923.0	30749.0	27849.0
3	BE	1	QUANTITY_IN_100KG	33042000	5362.0	6142.0	12395.0	11985.0	13644.0	16716.0	21120.0	32679.0	31471.0
4	BE	1	QUANTITY_IN_100KG	33043000	4495.0	7884.0	7529.0	7542.0	9757.0	14676.0	15747.0	14824.0	16593.0

```
In [13]:
1 trade3_df = postdw.combine_cols(trade2_df, "DS-016890")
2 trade3_df.head()
```

Out[13]:

	DECL	PRCCODE	2010	2011	2012	2013	2014	2015	2016	2017	2018	INDICATORS
0	BE	33030010	10663.0	15099.0	25666.0	45444.0	29257.0	21133.0	22526.0	25515.0	23163.0	IMPQNT
1	BE	33030090	48551.0	40142.0	23824.0	28018.0	31000.0	30431.0	32367.0	30358.0	32851.0	IMPQNT
2	BE	33041000	5401.0	6106.0	7306.0	11117.0	12943.0	13687.0	15923.0	30749.0	27849.0	IMPQNT
3	BE	33042000	5362.0	6142.0	12395.0	11985.0	13644.0	16716.0	21120.0	32679.0	31471.0	IMPQNT
4	BE	33043000	4495.0	7884.0	7529.0	7542.0	9757.0	14676.0	15747.0	14824.0	16593.0	IMPQNT

Figure 15 : combine_cols

In the example above, a combination of the information in the first data frame columns "FLOW" and "INDICATORS" was used to fill out the column "INDICATORS" in the resulting data frame. For instance, the rows in the first data frame with "FLOW" equal to 1 and "INDICATORS" equal to QUANTITY_IN_100KG were assigned "IMPQNT" in the resulting data frame.

reorganize
Arg.:(input_df, exclude, restore, drop_col)

Turn columns values to headers and the other way around. In the example below the elements in "INDICATORS" become headers and the different year headers become values within a new column "Year".

```
In [13]:
1 trade3_df.head()
```

Out[13]:

	DECL	PRCCODE	2010	2011	2012	2013	2014	2015	2016	2017	2018	INDICATORS
0	BE	33030010	10663.0	15099.0	25666.0	45444.0	29257.0	21133.0	22526.0	25515.0	23163.0	IMPQNT
1	BE	33030090	48551.0	40142.0	23824.0	28018.0	31000.0	30431.0	32367.0	30358.0	32851.0	IMPQNT
2	BE	33041000	5401.0	6106.0	7306.0	11117.0	12943.0	13687.0	15923.0	30749.0	27849.0	IMPQNT
3	BE	33042000	5362.0	6142.0	12395.0	11985.0	13644.0	16716.0	21120.0	32679.0	31471.0	IMPQNT
4	BE	33043000	4495.0	7884.0	7529.0	7542.0	9757.0	14676.0	15747.0	14824.0	16593.0	IMPQNT

```
In [16]:
1 years = [str(y) for y in list(range(2010, 2019))]
2 trade4_df = postdw.reorganize(trade3_df, years, ["INDICATORS"], [])
3 trade4_df.head()
```

Out[16]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	BE	33030010	2010	26825.0	57051953.0	10663.0	33358949.0
1	BE	33030010	2011	19144.0	46379642.0	15099.0	37640265.0
2	BE	33030010	2012	24193.0	51070030.0	25666.0	53738179.0
3	BE	33030010	2013	28177.0	55840493.0	45444.0	85104060.0

Figure 16 : `reorganize`

`convert_units`

Arg.:(input_df, DS)

Convert the products' quantity indicators units to kg and the value indicators to euros if they were not originally provided in these units. The information necessary to perform the conversion is provided by function **`ds_conversion`**.

```
In [16]:
1 trade4_df.head()
```

Out[16]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	BE	33030010	2010	26825.0	57051953.0	10663.0	33358949.0
1	BE	33030010	2011	19144.0	46379642.0	15099.0	37640265.0
2	BE	33030010	2012	24193.0	51070030.0	25666.0	53738179.0
3	BE	33030010	2013	28177.0	55840493.0	45444.0	85104060.0
4	BE	33030010	2014	26810.0	61093304.0	29257.0	91483884.0

```
In [17]:
1 trade5_df = postdw.convert_units(trade4_df, "DS-016890")
2 trade5_df.head()
```

Out[17]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	BE	33030010	2010	2682500.0	57051953.0	1066300.0	33358949.0
1	BE	33030010	2011	1914400.0	46379642.0	1509900.0	37640265.0
2	BE	33030010	2012	2419300.0	51070030.0	2566600.0	53738179.0
3	BE	33030010	2013	2817700.0	55840493.0	4544400.0	85104060.0
4	BE	33030010	2014	2681000.0	61093304.0	2925700.0	91483884.0

Figure 17 : `convert_units`

`pcode_conversion`

Arg.:(input_df, pcomext_table, DS)

If the input data frame comes from a different dataset than PRODCOM, this function converts the products codes into the PRODCOM equivalent ones, by means of the *pcomext_table*.

```
In [17]:
1 trade5_df.head()
```

Out[17]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	BE	33030010	2010	2682500.0	57051953.0	1066300.0	33358949.0
1	BE	33030010	2011	1914400.0	46379642.0	1509900.0	37640265.0
2	BE	33030010	2012	2419300.0	51070030.0	2566600.0	53738179.0
3	BE	33030010	2013	2817700.0	55840493.0	4544400.0	85104060.0
4	BE	33030010	2014	2681000.0	61093304.0	2925700.0	91483884.0

```
In [18]:
1 trade6_df = postdw.pcode_conversion(trade5_df, pcomext_table, "DS-016890")
2 trade6_df.head()
```

Out[18]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
2	BE	20421150	2012	2419300.0	51070030.0	2566600.0	53738179.0
11	BE	20421170	2012	5483300.0	118352330.0	2382400.0	57277089.0
20	BE	20421250	2012	411900.0	19033979.0	730600.0	26196290.0
29	BE	20421270	2012	887100.0	42834137.0	1239500.0	32323817.0

Figure 18 : `pcode_conversion`

`prices`

`Arg.:(input_df)`

This function has been developed for data gap filling purposes. If the input data frame contains value and quantity indicators, the function calculates the products prices by dividing the mentioned indicators. The function has been thought to be applied on the data from COMEXT.

```
In [25]:
1 trade8_df.head()
```

Out[25]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	Belgium	14111000	2010	747700.0	49580680.0	829400.0	36779619.0
1	Belgium	14111000	2011	933000.0	52026865.0	838800.0	36665747.0
2	Belgium	14111000	2012	642100.0	43417240.0	1052800.0	34111340.0
3	Belgium	14111000	2013	608800.0	35455199.0	602300.0	37292808.0
4	Belgium	14111000	2014	356100.0	33473809.0	640500.0	41478819.0

```
In [24]:
1 prices_df = postdw.prices(trade8_df)
2 prices_df.head()
```

Out[24]:

	PRCCODE	DECL	Year	EXPPRICE	IMPPRICE	AVGPRICE
0	14111000	Belgium	2010	66.310927	44.344850	55.327889
1	14111000	Belgium	2011	55.762985	43.712145	49.737565
2	14111000	Belgium	2012	67.617567	32.400589	50.009078
3	14111000	Belgium	2013	58.237843	61.917330	60.077587

Figure 19 : `prices`

Since COMEXT does not provide production figures, only prices for exports and imports can be calculated directly. The average prices are calculated for each product and year as an average of the other two.

apply_prices
Arg.:(input_df, prices_df)

This function has been developed for data gap filling purposes. The products' prices calculated with the function *prices* are used here to calculate indicators quantities and values for the same products in a different data frame. This function has been thought to apply the prices calculated out of COMEXT data onto the PRODCOM data.

```
In [27]:
1 pcom9
```

Out[27]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN
4	Belgium	14111000	2014	NaN	49328650.0	NaN	73703310.0	NaN	NaN

```
In [28]:
1 filling_df = postdw.apply_prices(pcom9_df, prices_df)
2 filling_df.head()
```

Out[28]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	784237.869630	NaN	1.796743e+06	NaN	NaN	NaN
1	Belgium	14111000	2011	861913.149101	NaN	1.764831e+06	NaN	NaN	NaN
2	Belgium	14111000	2012	660026.702296	NaN	2.052531e+06	NaN	NaN	NaN
3	Belgium	14111000	2013	748465.388334	NaN	1.033950e+06	NaN	NaN	NaN

Figure 20 : apply_prices

The average prices will be used to calculate production quantities and values.

fill_gaps
Arg.:(input_df, filling_df)

This function has been developed for data gap filling purposes. This function fills out the gaps in the input data frame with the corresponding values (i.e. same country, product, year and indicator) from the extra data frame provided which is in turn a product of **apply_prices** (for example, *filling_df* coming from the calculation of quantities based on economic value data (from the previous function)).

```
In [27]:
1 pcom9
```

Out[27]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN
4	Belgium	14111000	2014	NaN	49328650.0	NaN	73703310.0	NaN	NaN

```
In [29]:
1 filled_pcom_df = postdw.fill_gaps(pcom9_df, filling_df)
2 filled_pcom_df.head()
```

Out[29]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	784237.869630	52003540.0	1.796743e+06	79676300.0	NaN	NaN
1	Belgium	14111000	2011	861913.149101	48062850.0	1.764831e+06	77144540.0	NaN	NaN
2	Belgium	14111000	2012	660026.702296	44629400.0	2.052531e+06	66503210.0	NaN	NaN
3	Belgium	14111000	2013	748465.388334	43589010.0	1.033950e+06	64019430.0	NaN	NaN

Figure 21 : `fill_gaps`

`interpol`

Arg.:(input_df)

This function has been developed for data gap filling purposes. Another strategy to fill possible gaps in the data is calculating them through the interpolation of the existing values. This function does this for each indicator column, and only when there are one or two missing values between two existing ones belonging to the same product. This limitation can be of course be modified in the function.

```
In [45]:
1 pcom9_df.head()
```

Out[45]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
2812	Belgium	20421990	2014	NaN	49567780.0	NaN	59203360.0	NaN	1666719.0
2813	Belgium	20421990	2015	NaN	70734740.0	NaN	76097930.0	NaN	NaN
2814	Belgium	20421990	2016	NaN	76298760.0	NaN	90121170.0	NaN	2335741.0
2815	Belgium	20421990	2017	NaN	83152590.0	NaN	86199900.0	NaN	4572968.0

```
In [50]:
1 inter_pcom_df = postdw.interpol(pcom9_df)
2 inter_pcom_df.head()
```

Out[50]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
2812	Belgium	20421990	2014	NaN	49567780.0	NaN	59203360.0	NaN	1666719.0
2813	Belgium	20421990	2015	NaN	70734740.0	NaN	76097930.0	NaN	2001230.0
2814	Belgium	20421990	2016	NaN	76298760.0	NaN	90121170.0	NaN	2335741.0
2815	Belgium	20421990	2017	NaN	83152590.0	NaN	86199900.0	NaN	4572968.0

Figure 22 : `Interpol`

fao.py

The goal of the functions contained in this script is locating and downloading FAO's data. The use of proxies to access the bulkdownload service is not mandatory but depends on the dispositive and internet connection.

fao_datasets
Arg.:(proxies)

Extracts a list with the FAO datasets available through the bulkdownload service.

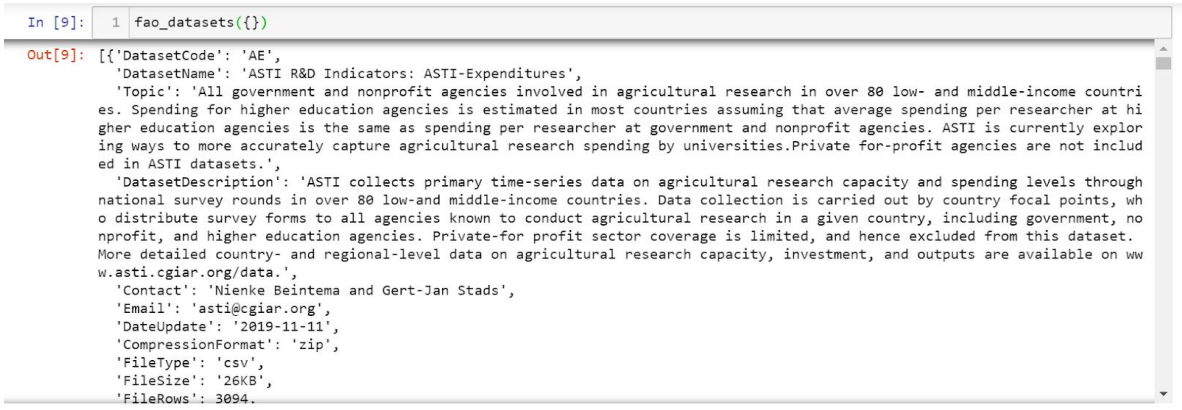

Figure 23 : fao_datasets

search_one_word
Arg.:(datasets, proxies, key_word)

Provides the names of all the available datasets containing a given word in their name.

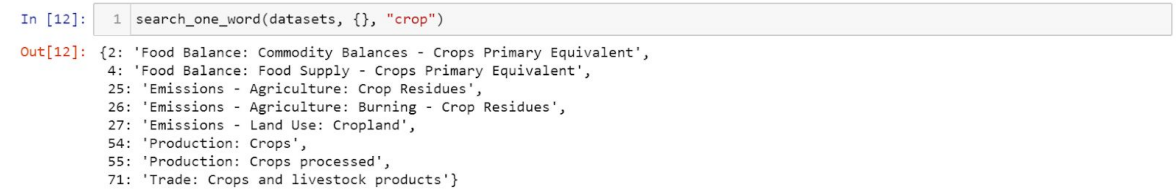

Figure 24 : search_one_word (FAO)

The numbers you may see in the image above are the dataset number and the same ones column "FAO_datasets" in the inventory file refers to.

download_faozip
Arg.:(proxies, datasets, datasets_numbers, end_folder, storage folder)

Downloads the datasets requested using their numeric identifier. The datasets are downloaded as .zip files which are saved in a folder made for that purpose. The previous content of that folder is moved, before the downloading begins to a back-up or storage folder, so that the previously downloaded datasets remain available.

extract_zip
Arg:(zip_folder, zip_file_name)

Extracts the files in the given .zip files present in a folder.

fao_extr
Arg.:(proxies, datasets, faodata_folder, backup_folder, datasets_numbers)

Downloads, unzip and reads the datasets corresponding to the datasets numeric identifiers provided.

```
In [17]:
1 food_balance = fao_extr({}, datasets, FAODATA_FOLDER, STORAGE_FOLDER, [2])
2 food_balance.head()
```

Out[17]:

	Area Code	Area	Item Code	Item	Element Code	Element	Year Code	Year	Unit	Value	Flag
0	2	Afghanistan	2617	Apples and products	5510	Production	1961	1961	tonnes	15100.0	S
1	2	Afghanistan	2617	Apples and products	5510	Production	1962	1962	tonnes	15100.0	S
2	2	Afghanistan	2617	Apples and products	5510	Production	1963	1963	tonnes	15100.0	S
3	2	Afghanistan	2617	Apples and products	5510	Production	1964	1964	tonnes	18400.0	S
4	2	Afghanistan	2617	Apples and products	5510	Production	1965	1965	tonnes	20400.0	S

Figure 25 : fao_extr

In the example above the .zip files corresponding to dataset 2 (*"Food Balance : Commodity Balances – Crops Primary Equivalent"*) were downloaded, unzip and their content read.

check_fao
Arg.:(input_df, fao_codes, countries, start_year, end_year)

First, it checks whether there are elements repeated in the provided products' codes and countries and then, in the given input data frame, it looks for the non-repeated ones and for the years comprehended between the start and end year provided. It also converts the provided country names (if present in the input data frame) to PRODCOM countries naming system.

```
In [31]:
1 check_fao(food_balance, ["2615", "2616", "2618", "2619", "45345346"], ["Italy", "Italy", "Spainx"], "2010", "2012")

These elements are repeated:  ['Italy']
These elements do not belong to the DS:  ['Spainx']
These elements do not belong to the DS:  ['45345346']

Out[31]: (['2615', '2616', '2618', '2619'],
          ['Italy', 'Italy'],
          ['2010', '2011', '2012'])
```

Figure 26 : check_fao

As it can be appreciated in the figure above, the function let the user know in case it cannot find any of the elements introduced and returns only those present in the input data frame.

`filter_faodata`
Arg.:(df, fao_codes, countries, start_year, end_year)

Performs *check_fao* on the variables given and generates a subset of input data frame provided according to the indicated filters (i.e. fao codes, countries and timeframe).

```
In [32]:
1 food_balance_filtered = filter_faodata(food_balance, ["2615", "2616", "2618", "2619", "45345346"], ["Italy", "Italy", "Spain
2 food_balance_filtered.head()

These elements are repeated:  ['Italy']
These elements do not belong to the DS:  ['Spainx']
These elements do not belong to the DS:  ['45345346']
```

Out[32]:

	Area Code	Area	Item Code	Item	Element Code	Element	Year Code	Year	Unit	Value	Flag
2358572	106	Italy	2615	Bananas	5510	Production	2010	2010	tonnes	342.0	S
2358573	106	Italy	2615	Bananas	5510	Production	2011	2011	tonnes	351.0	S
2358574	106	Italy	2615	Bananas	5510	Production	2012	2012	tonnes	340.0	S
2358625	106	Italy	2615	Bananas	5610	Import Quantity	2010	2010	tonnes	658391.0	S

Figure 27 : `filter faodata`

`fao_extra`
Arg.:(fao_df, inventory_df, col)

Some of the BoP products and products groups (the latter only for FAO products, see "FAO_groups" in the inventory file) do not match a single FAO product or group but the sum of several. This function creates the necessary new FAO products and groups by adding up the values of the ones composing them.

```
In [36]:
1 extra_fao_groups = fao_extra(food_balance_filtered, ups_df, "FAO_group")
2 extra_fao_groups.head()
```

Out[36]:

	Area	Area Code	Element	Element Code	Flag	Item	Item Code	Unit	Value	Year	Year Code
0	Italy	NaN	Export Quantity	NaN	NaN	NaN	2615;2616;2618;2619	NaN	222747.0	2010	NaN
0	Italy	NaN	Export Quantity	NaN	NaN	NaN	2615;2616;2618;2619	NaN	209072.0	2011	NaN
0	Italy	NaN	Export Quantity	NaN	NaN	NaN	2615;2616;2618;2619	NaN	162207.0	2012	NaN
0	Italy	NaN	Food supply quantity (tonnes)	NaN	NaN	NaN	2615;2616;2618;2619	NaN	778373.0	2010	NaN

Figure 28 : `fao_extra`

In the figure above, "2615;2616;2618;2619", a FAO group which previously did not exist, has been created by adding up the values of the single groups that compose it.

calculation.py

The goal of the functions in this script is performing the upscaling of the BoPs products, make the necessary modification in the presence of double counting and calculating the representative products apparent consumption.

`consumption`

Arg.:(input_df)

If the input data frame does not contain consumption figures (“CONSQNT” and “CONSVAL”), this function calculates them out of the production, exports and imports value. If it has consumption values, the function will only calculate them for those products for which the original one is missing.

```
In [17]:
consumption_df = consumption_df[consumption_df["Year"] == "2010"]
inventory_df = ups_df[ups_df["BoP"] == "bop_hg"]
```

```
In [18]:
consint_one = calc.consint_one(inventory_df, consumption_df, "QNT")
consint_one.head()
```

```
The provided items are in columns: ['prod_codes']
```

```
In [15]:
pcom9_df.head()
```

Out[15]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN
4	Belgium	14111000	2014	NaN	49328650.0	NaN	73703310.0	NaN	NaN

```
In [16]:
consumption_df = calc.consumption(pcom9_df)
consumption_df.head()
```

Out[16]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL	CONSQNT	CONSVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN	0.0	27672760.0
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN	0.0	29081690.0
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN	0.0	21873810.0
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN	0.0	20430420.0

Figure 29 : `consumption`

It should be taken into account that, as long as the three indicators required for calculating the consumption (i.e. production, exports and imports) exist in the input data frame, the function will calculate the consumption, even if for some products some of the indicators does not provide a value. That means for instance that, if for a product the exports are missing, the function will consider 0 as export value and provide a consumption value that would be the sum of production and imports. **This may lead to negatives values for instance.** Therefore, any data gap filling steps should be performed prior to calculating the consumption with this function.

The option exists to only calculate consumption in the cases where there are values of the three indicators and leave consumption empty if not.

consint_one
Arg.:(inventory_df, input_df, fao_groups, what)

For one year and one country data, it calculates the BoP representative products consumption as well as that of the product groups these representative products belong to. The "what" argument indicates the function whether it is the consumption quantity ("QNT") or the consumption value ("VAL") it has to calculate.

```
In [33]:
1 ci = calc.consint_one(ups_df, cons_df, fbs_df, "QNT")
2 ci.head()
```

Out[33]:

rep_prod_names	components	split	...	FBS_UPS	IMPQNT	PG_IMPQNT	EXPQNT	PG_EXPQNT	PRODQN
RP_PigMeat	10111230;10111250;10111290;10113230;10113250;1...	1.0	...	1.0	122596400.0	341367500.0	-792790900.0	1.327911e+09	465271501.
RP_BeefMeat	10111140;10111190;10113100	1.0	...	1.0	56118500.0	341367500.0	-122351000.0	1.327911e+09	96445980.
RP_Meat_based_dishes	10851100	1.0	...	0.0	0.0	0.0	-0.0	0.000000e+00	40788188.
RP_Cod	10201100;10201200;10201330;10201360;10201400;1...	1.0	...	1.0	72842850.0	149771900.0	-38604900.0	9.295740e+07	25593460.

Figure 30 : consint_one

For some FAO products in the Food BoP it was decided to use the FAO Food Balance Sheets (FBS) products group consumption data instead of the one calculated out of the addition of the different products consumption. These values are provided as a data frame ("fao_groups") in the function arguments, and, in the representative products for which the inventory column "FBS_UPS" is 1, they will substitute the product group consumption values.

Notice that the resulting data frame contains all the information in the inventory file plus the information retrieved from the data sources.

double_counting
Arg.:(input_df, what)

Applies the double counting subtractions, both to the representative products consumption and to the products group consumptions (columns "DC_TOTAL", "PG_CONSQNT_DC" and "CONSQNT_DC").

```
In [34]:
1 ci_dc = calc.double_counting(ci, "QNT")
2 ci_dc.head()
```

Out[34]:

s	split	...	EXPQNT	PG_EXPQNT	PRODQNT	PG_PRODQNT	CONSQNT	PG_CONSQNT	PG_FBS	DC_TOTAL	PG_CONSQNT_DC	CONSQNT_DC
.	1.0	...	-792790900.0	1.327911e+09	465271501.0	1.140275e+09	-204922999.0	759000000.0	759000000.0	-13460102.04	7.455399e+08	-204922999.0
)	1.0	...	-122351000.0	1.327911e+09	96445980.0	1.140275e+09	30213480.0	759000000.0	759000000.0	-13460102.04	7.455399e+08	30213480.0
)	1.0	...	-0.0	0.000000e+00	40788188.0	4.078819e+07	40788188.0	0.0	NaN	0.00	0.000000e+00	40788188.0
.	1.0	...	-38604900.0	9.295740e+07	25593460.5	4.620387e+07	59831410.5	300000000.0	300000000.0	0.00	3.000000e+08	59831410.5

Figure 31 : `double_counting`

`upscale_one`

Arg.:(input_df, what)

Applies the upscaling to the representative products (column “CONSQNT_UPS”) and also multiplies the result by the corresponding lifespan (provided for each product in the inventory file).

Out[36]:

ODQNT	CONSQNT	PG_CONSQNT	PG_FBS	DC_TOTAL	PG_CONSQNT_DC	CONSQNT_DC	CONSQNT_SHARE	CONSQNT_SHARE_SUM	CONSQNT_UPS
75e+09	-204922999.0	759000000.0	759000000.0	-13460102.04	7.455399e+08	-204922999.0	-0.274865	0.188725	-1.085828e+09
75e+09	30213480.0	759000000.0	759000000.0	-13460102.04	7.455399e+08	30213480.0	0.040526	0.188725	1.600925e+08
19e+07	40788188.0	NaN	NaN	NaN	NaN	40788188.0	NaN	0.000000	inf
87e+07	59831410.5	300000000.0	300000000.0	NaN	3.000000e+08	59831410.5	0.199438	0.343333	1.742664e+08

Figure 32 : `upscale_one`

`consumption_intensity`

Arg.:(inventory, input_df, fbs_df, what)

The three previous functions, **`consint_one`**, **`double_counting`** and **`upscale`** were made to be only applied on one-year one-country data. This function does the same operations the mentioned functions performed on several countries and several years data. In addition to this, it divides the resulting representative products upscaled consumption by the corresponding population number in order to get the AC per person (column “CONSQNT_UPS” and “CONSQNT_UPS_P”).

```
In [43]:
1 ci_all = consumption_intensity(ups_df, all_df, faog9_df, "QNT")
2 ci_allo_concat = pd.concat(ci_all)
3 ci_allo_concat.head()
```

Out[43]:

PG_CONSQNT_DC	CONSQNT_DC	CONSQNT_SHARE	CONSQNT_SHARE_SUM	CONSQNT_UPS	DECL	Year	Population	CONSQNT_DC_P	CONSQNT_UPS_P
788028710.5	-186389534.0	-0.236526	0.224125	-8.316313e+08	Belgium	2012	11075889.0	-16.828404	-75.084834
788028710.5	36987384.0	0.046937	0.224125	1.650300e+08	Belgium	2012	11075889.0	3.339451	14.899933
51428150.0	51428150.0	1.000000	1.000000	5.142815e+07	Belgium	2012	11075889.0	4.643253	4.643253
293000000.0	63646203.0	0.217223	0.364787	1.744751e+08	Belgium	2012	11075889.0	5.746374	15.752692

Figure 33 : `consumption_intensity`

`apply_lifespan`

Arg.: (input_df)

This function is to be applied after **`consumption_intensity`** and it could be considered as a different methodology for applying the lifespan. It divides the consumption intensity values by the lifespan (note than they had already been multiplied by the lifespan in **`upscale_one`**) and, for each representative product and year it add ups all the consumption intensity values for all the years covered by the product lifespan counting from that year. I.e a Fridge lifespan is 15 years, that means that, after applying this function, the fridge new consumption intensity for 2000 will be the sum of all the yearly consumption intensity from 2000 until 2015.

3.2 BOP MOBILITY

3.2.1 Inputs

To build and calculate BoP Mobility ACs using the script mentioned in Table 1 two different main inputs are required: data from three different data sources and a frame were all the information can be orderly merged.

3.2.2 Data Sources

The main data source is, as in the other BoPs, EUROSTAT, with the difference that, in this case a greater number of datasets is required and there is no need to access them through the SDMX service. What follows is a list with the EUROSTAT datasets used and their content.

EUROSTAT dataset	Content
road_eqs_mopeds	Two wheels and buses number of vehicles
road_eqs_motorc	
road_eqs_busmot	
rail_tf_traveh	Millions of trains km
avia_panc	National, intra-eu, extra-eu flights number of passengers
avia_paincc	
avia_paexcc	
road_pa_mov	Two wheels million passenger km
road_tf_road	Buses million vehicle km
road_eqs_carpda	Passenger cars, by type of motor energy
road_eqs_carmot	Passenger cars, by type of motor energy and size of engine
road_eqs_carage	Passenger cars in active by period of production

Table 5 : BoP Mobility EUROSTAT datasets

The information retrieved from the EUROSTAT dataset will have to be complemented with data from the Statistical pocketbook of Mobility and Transport[7], which would have to be downloaded and stored in advance, and reference values from the BoP Mobility report, which had already been extracted from the report and copied inside a function. The table below shows the information provided by each of these two sources.

Source	Content	Short name
PB(pb2019-section23, sheet = cars)	Passenger cars million passenger km	*cars*
PB(pb2019-section23, sheet = rail_pkm)	Trains million passenger km	*rail_pkm*
BoP Mobility Report Table. 60 (pag.121)	National, intra-eu, extra-eu flights average distance	*bop121*
BoP Mobility Report (pag.142)	Reference two wheels vehicles contribution to total passengers km in the EU.	*bop142*
BoP Mobility Report pag.143	Reference passenger cars occupancy factor	*bop143_148*
BoP Mobility Report pag. 148	Reference two wheels vehicles occupancy factor	*bop143_148*

Table 6 : BoP Mobility other data sources

[7] https://ec.europa.eu/transport/facts-fundings/statistics/pocketbook-2019_en

3.2.3 Mobility frame file

To order the different data inputs coming from the different data sources an excel file containing a list with all the means of transport types considered and their main features is used. The information on this list is disposed so that the script function can easily add the different data sources data as columns matching it with the right vehicle type on the list.

```
In [2]:
# Loading the mobility frame file

mob_frame = di.read_file("mobility_frame.xlsx", INPUTS_FOLDER, 0, 0, {})
mob_frame.head()
```

Out[2]:

	transport	vehicle_type	code	type	age	id_1	id_2	id_3
0	Road transport	Passenger_Cars	SP 1	Gasoline <1.4 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC_LT1400	PET
1	Road transport	Passenger_Cars	SP 2	Gasoline <1.4 L	Euro_4	EU4	CC_LT1400	PET
2	Road transport	Passenger_Cars	SP 3	Gasoline <1.4 L	Euro_5	EU5	CC_LT1400	PET
3	Road transport	Passenger_Cars	SP 4	Gasoline 1.4 - 2.0 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC1400-1999	PET

Figure 34 : `mobility_frame`

In the table below you may find the columns to be added to the mobility frame file and filled out with the data from the different data sources.

Column name	Short name
Coefficient technology	*tech_coeff*
Coeffcient engine displacement	*eng_coeff*
Coefficient type of fuel	*fuel_coeff*
Million passenger km	*mill_pkm*
Occupancy factor	*ocp_factor*
Contribution to passenger km	*contr_pkm*
Km per flight	*flight_km*
Number of passenger	*num_p*
Milions of trains km	*trkm*
Numbers of vehicles	*num_vehi*
Milion vehicle km	*mill_vehikm*

Table 7 : Mobility column names

3.2.4 Scripts

bop_mobility.py

This script contains the functions necessary to extract the right data from the data sources in Table 4 and Table 5, filter it, calculate different magnitudes out of it and join it with the mobility frame file.

ds_transport
Arg.:(DS, col_name)

This function provides information, for each of the datasets from each data source, on how to filter the data, which units conversion factors to apply, how to merge the data with the mobility frame file, which new categories are to be added, etc.

```
In [3]:
# Units conversion factors for rail_tf_traveh data.

mob.ds_transport("rail_tf_traveh", "UNITS")

Out[3]: 0.001

In [4]:
# Correspondancy between rail_tf_traveh data frame and mobility frame data frame columns.
# These are used to bring together both data frames.

mob.ds_transport("rail_tf_traveh", "SORT")

Out[4]: {'id_1': 'vehicle', 'year': 'year', 'geo\\time': 'geo\\time'}
```

Figure 35 : ds_transport

Whilst EUROSTAT dataset keep their names, pocketbooks and mobility report pages are referred to with the shorthand names displayed in Table 5.

check_countries
Arg.:(countries_names)

Checks whether the country names provided are repeated and whether they are in a dictionary within the function. Then it turns them into country codes.

```
In [5]:
mob.check_countries(["Italy", "France", "France", "Spainx"])

These elements are repeated:  ['France']
These elements do not belong to the DS:  ['Spainx']

Out[5]: ['IT', 'FR', 'FR']
```

Figure 36 : check_countries

bop_report
Arg.:(bop_sheet, col_name)

This function contains all the information extracted from the mobility report and provides it according to the sheet and element (“col_name” argument) requested.

```
In [6]:
mob.bop_report("bop143_148", "BOP_TABLE")

Out[6]: {'vehicle': ['Passenger_Cars', '2W'], 'value': [1.62, 1.1]}
```

Figure 37 : bop_report

Notice that the mobility report sheets received slightly different names here.

get_mobility
Arg.:(start_year, end_year, countries, EUROSTAT_dataset)

From the provided EUROSTAT dataset, this function, unlike **get_pcomext**, downloads all the information available and then filters it according to the years and countries arguments provided. It also filters it following information provided by dictionary "filters_dict" in **ds_transport**.

```
In [8]:
1 czechia_rail = mob.get_mobility("2010", "2010", ["Czechia"], "rail_tf_traveh")
2 czechia_rail.head()
```

Out[8]:

	year	geo\time	unit	vehicle	trkm
0	2010	CZ	THS_TRKM	LOC_ELC	68.666
1	2010	CZ	THS_TRKM	RCA_ELC	15.114
2	2010	CZ	THS_TRKM	TOTAL	153.792
3	2010	CZ	THS_TRKM	LOC_DIE	14.585

Figure 38 : get_mobility

get_bop
Arg.:(start_year, end_year, countries, bop)

Provides the information from the mobility report. Since the information from the mobility report is used as reference regardless the year and, except for the national flights, also the country, this function adds the year and country column to the retrieved information so that it can be correctly merged with the information from other sources.

```
In [9]:
1 germany_ocp_factors = mob.get_bop("2010", "2012", ["Germany"], "bop143_148")
2 germany_ocp_factors.head()
```

Out[9]:

	vehicle	ocp_factor	geo\time	year
0	Passenger_Cars	1.62	DE	2010
1	2W	1.10	DE	2010
0	Passenger_Cars	1.62	DE	2011
1	2W	1.10	DE	2011

Figure 39 : get_bop

get_pocketbook
Arg.:(start_year, end_year, countries, file, file_folder, sheetname)

Extracts the information from the provided pocketbook file sheet and filters it according to the years and countries given as arguments.

```
In [10]:
1 belgium_cars = mob.get_pocketbook("2010", "2010", ["Belgium"], "pb2019-section23.xls", INPUTS_FOLDER, "cars")
2 belgium_cars.head()
```

Out[10]:

	geo\time	year	mill_pkm	vehicle
0	BE	2010	109387.760219	Passenger_Cars

Figure 40 : get_pocketbook

add_col
Arg.:(input_df, DS)

Adds a new column to the provided data frame containing data from "DS". "DS" being either a EUROSTAT dataset, a pocketbook sheet or a mobility report page. In the two last cases the short names in Table 5 shall be used. The name and content of the additional column is given by the "add_col_dict" in **ds_transport**.

add_extra_type
Arg.:(input_df, DS)

Some vehicle types features in the mobility frame file do not exist as such in the information retrieved from the different data sources. In these cases, new categories that match what is in the mobility frame file have to be created out of the data available in the data sources.

For each EUROSTAT dataset, pocketbook or report page, the new types to be added are given by the "extra_types" dict in **ds_transport**.

calc_fuel_coeff
Arg.:(start_year, end_year, countries_names)

For the countries and period provided, it extracts the corresponding data from EUROSTAT *road_eqs_carpda* dataset and calculates the fuel type coefficient.

```
In [5]:  1 fuel_coeff_belgium = mob.calc_fuel_coeff("2017", "2017", ["Belgium"])
         2 fuel_coeff_belgium.head()
```

Out[5]:

	unit	mot_nrg	geo\time	year	fuel_coeff
0	NR	ALT	BE	2017	0.008180
1	NR	BIOETH	BE	2017	0.000020
2	NR	DIE	BE	2017	0.578073
3	NR	DIE_X_HYB	BE	2017	0.577144

Figure 41 : calc_fuel_coeff

calc_eng_coeff
Arg.:(start_year, end_year, countries_names)

For the countries and period provided, it extracts the corresponding data from EUROSTAT *road_eqs_carmot* dataset and calculates the engine displacement coefficient.

```
In [13]:
1 belgium_eng_coeff = mob.calc_eng_coeff("2010", "2010", ["Belgium"])
2 belgium_eng_coeff.head()
```

Out[13]:

	unit	mot_nrg	engine	geo\time	year	eng_coeff
0	NR	DIE	CC1400-1999	BE	2010	0.740648
1	NR	DIE	CC_GE2000	BE	2010	0.154983
2	NR	DIE	CC_LT1400	BE	2010	0.104370
3	NR	DIE	TOTAL	BE	2010	1.000000

Figure 42 : `calc_eng_coeff`

`calc_tech_coeff`

Arg.:(start_year, end_year, countries_names)

For the countries and period provided, it extracts the corresponding data from EUROSTAT *road_eqs_carage* dataset and calculates the technology coefficient.

```
In [15]:
1 belgium_tech_coeff = mob.calc_tech_coeff("2010", "2010", ["Belgium"])
2 belgium_tech_coeff.head()
```

Out[15]:

	geo\time	year	age_type	tech_coeff	value
0	BE	2010	CONV	0.024469	5276000.0
1	BE	2010	EU1	0.097877	5276000.0
2	BE	2010	EU2	0.097877	5276000.0
3	BE	2010	EU3	0.246001	5276000.0

Figure 43 : `calc_tech_coeff`

mob_merge

Arg.:(mob_frame, col_name, start_year, end_year, countries_names, file_name, file_folder)

For the countries and period provided, it extracts all the data referred to the column name given(“col_name”, see Table 6), may it come from EUROSTAT datasets, pocketbook or bop report pages, does the calculation required in case it is required (to calculate any of the coefficients for instance) and merge it with the mobility frame.

```
In [16]:
1 belgium_mill_vehikm = mob.mob_merge(mob_frame, "mill_vehikm" , "2013", "2014", ["Belgium"], "", "")
2 belgium_mill_vehikm.head()
```

Out[16]:

	transport	vehicle_type	code	type	age	id_1	id_2	id_3	geo\time	year	mill_vehikm
0	Road transport	Passenger_Cars	SP 1	Gasoline <1.4 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC_LT1400	PET	BE	2013	NaN
1	Road transport	Passenger_Cars	SP 2	Gasoline <1.4 L	Euro_4	EU4	CC_LT1400	PET	BE	2013	NaN
2	Road transport	Passenger_Cars	SP 3	Gasoline <1.4 L	Euro_5	EU5	CC_LT1400	PET	BE	2013	NaN
3	Road transport	Passenger_Cars	SP 4	Gasoline 1.4 - 2.0 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC1400-1999	PET	BE	2013	NaN

Figure 44 : `mob_merge`

share_columns
Arg.:(input_df)

Provided as an input a data frame containing the mobility frame plus all the columns already added, it calculates the number of vehicles share per vehicle type (new column *num_vehi_share*) and the share of train km per train type (new column *trkm_share*).

consumption
Arg.:(input_df)

Provided as an input a data frame containing the mobility frame plus all the columns already added, including the share columns calculated with **share_columns**, this function calculates the apparent consumption for each vehicle type in the mobility frame.

mobility
Arg.:(mob_frame, start_year, end_year, countries, pocketbook, file_folder)

For the period and countries provided it repeats **mob_merge** for all the columns in Table 6, then applies **share_columns** and finally calculates the apparent consumption through **consumption**. The result is a data frame with all the columns in Table 6 plus the share columns and the apparent consumption columns for all the countries and years requested.

```
In [17]:
1 czechia_mobility = mob.mobility(mob_frame, "2010", "2010", ["Czechia"], "pb2019-section23.xls", INPUTS_FOLDER)
2 czechia_mobility.head()
```

Out[17]:

	transport	vehicle_type	code	type	age	id_1	id_2	id_3	geo\time	year	...	flight_km	contr_pkm
0	Road transport	Passenger_Cars	SP 1	Gasoline <1.4 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC_LT1400	PET	CZ	2010	...	NaN	NaN
1	Road transport	Passenger_Cars	SP 2	Gasoline <1.4 L	Euro_4	EU4	CC_LT1400	PET	CZ	2010	...	NaN	NaN
2	Road transport	Passenger_Cars	SP 3	Gasoline <1.4 L	Euro_5	EU5	CC_LT1400	PET	CZ	2010	...	NaN	NaN
3	Road transport	Passenger_Cars	SP 4	Gasoline 1.4 - 2.0 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC1400-1999	PET	CZ	2010	...	NaN	NaN

Out[17]:

id_2	id_3	geo\time	year	...	flight_km	contr_pkm	mill_pkm	mill_vehikm	eng_coeff	tech_coeff	ocp_factor	num_vehi_share	trkm_share	Result
C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.777714	1.62	NaN	NaN	20075.956248
C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.186254	1.62	NaN	NaN	4807.986548
C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.036032	1.62	NaN	NaN	930.133533
CC1400-1999	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.305327	0.777714	1.62	NaN	NaN	9317.993576

Figure 45 : mobility

4.Running the functions

Here it will be explained how to run the functions presented in the previous version in order to build up the BoP and calculate the apparent conusmption. The examples provided have all been run on a jupyter notebook.

4.1 Required packages

A number of python packages are required for running the functions. Therefore, these will have to be loaded every time the functions are run at the very beginning.

```
In [1]:
1  import os
2  import sys
3  import pandas as pd
4  from collections import Counter
5  import itertools
6  from itertools import compress
7  from itertools import product
8  import numpy as np
9  import eurostat as estat
10 import math
11
```

Figure 46 : packages loading

In addition to these, depending on the BoP considered, also the corresponding scripts will have to be loaded. In the example below the scripts for BoPs Food, Appliances and Household goods are loaded. Notice that, for the sake of simplicity all the scripts are assigned a short name.

```
In [2]:
1  import data_inputs as di
2  import predownload as predw
3  import pcomext as pcx
4  import postdownload as postdw
5  import fao
6  import calculation as calc
7
```

Figure 47 : scripts loading

4.2 Folders structure

In order to render easier the running of the scripts and the retrieval of the results it is recommended to create a folder for the purpose and five subfolders within it named as follows: *inputs, outputs, scripts, faodata* and *faodata_storage*. The table below displays what is to be copied in each of the folders.

Main folder	Subfolders	Files
BoPs	scripts	*data_inputs.py, predownload.py, postdownload.py, pcomext.py, calculations.py, fao.py, bop_mobility.py*

	inputs	*pcomext_table.xlsx, mobility_frame.py, pb2019-section23.xls, inventory.xlsx*
	outputs	…
	faodata	…
	faodata_storage	…

Table 8 : Recommended folders structure

Outputs, faodata and *faodata_storage* are of course empty before any function has been run.

If the folders are going to be used often it might be convenient to defined a variable for each of them. This will make it easier to refer to them when necessary.

In [3]:
```
OUTPUTS_FOLDER = "../outputs"
INPUTS_FOLDER = "../inputs"
FAODATA_FOLDER = "../faodata"
STORAGE_FOLDER = "../faodata_storage"
```

Figure 48 : folders definition

4.3 Running BOP HH, FOOD & APP

4.3.1 Loading data

For these BoPs the first thing to do is loading the *inventory* and *pcomext_table* files, which should have been previously saved in the *inputs* folder.

In [21]:
```
dtype_dict = inventory_dtype()

inventory = read_file("inventory.xlsx", INPUTS_FOLDER, 4, 0, dtype_dict)

inventory.head()
```

Out[21]:

	BoP	prod_groups	Source	FAO_group	FAO_datasets	prod_codes	prod_names	rep_prod_names	components	split	EXP_COEF	IMP_COEF	PROD_
0	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421150	Perfumes	NaN	NaN	NaN	1.0	1.0	
1	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421170	Toilet waters	NaN	NaN	NaN	1.0	1.0	
2	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421250	Lip make-up preparations	NaN	NaN	NaN	1.0	1.0	
3	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421270	Eye make-up preparations	NaN	NaN	NaN	1.0	1.0	
4	bop_hg	PG_PCP	EUROSTAT	NaN	NaN	20421300	Manicure or pedicure preparations	NaN	NaN	NaN	1.0	1.0	

Figure 49 : reading inventory file

```
In [7]:
pcomext_dtype = {"Year" : str, "PCOM" : str, "CN" : str}
pcomext_table = di.read_file("pcomext_table.xlsx", INPUTS_FOLDER, 0, 0, pcomext_dtype)
pcomext_table.head()
```

Out[7]:

	PCOM	CN	Year
0	15111140	2011000	1995
1	15111140	2012020	1995
2	15111140	2012030	1995
3	15111140	2012050	1995

Figure 50 : reading pcomext_table file

If the PRODCOM and COMEXT data of interest has previously been downloaded and saved onto an excel file in the *outputs* folder, it will have to be loaded using **`read_file`** too.

```
In [8]:
dtype_dict = {"DECL" : str, "PRCCODE" : str}
pcom_raw_df = di.read_file("Pcom_Food_Belgium_2010-2018.xlsx", OUTPUTS_FOLDER, 0, 0, dtype_dict)
pcom_raw_df.head()
```

Out[8]:

	Unnamed: 0	INDICATORS	DECL	PRCCODE	FREQ	2010	2011	2012	2013	2014	2015	2016	201
0	0	EXPQNT	017	10111230	A	300403800.0	311794600.0	361986600.0	411305400.0	436111500.0	461596300.0	432069900.0	434924700.
1	1	EXPQNT	017	10111250	A	116714600.0	107140600.0	97361300.0	87159900.0	76005500.0	70139700.0	66662400.0	59835100.
2	2	EXPQNT	017	10111290	A	146113200.0	137906600.0	128789200.0	121821400.0	109465800.0	106519400.0	99457400.0	97467400.
3	3	EXPQNT	017	10113230	A	315300.0	226800.0	120200.0	428100.0	222800.0	472000.0	1410100.0	380000.

Figure 51 : reading PRODCOM data file

```
In [9]:
dtype_dict = {"REPORTER" : str, "PARTNER" : str, "PRODUCT" : str, "FLOW" : str}
trade_raw_df = di.read_file("Trade_Food_Belgium_2010-2018.xlsx", OUTPUTS_FOLDER, 0, 0, dtype_dict)
trade_raw_df.head()
```

Out[9]:

	Unnamed: 0	Unnamed: 0.1	PARTNER	FLOW	INDICATORS	PRODUCT	REPORTER	FREQ	2010	2011	2012	2013	2014	2015	2016	2017	2018
0	0	0	BE	1	QUANTITY_IN_100KG	10062011	AT	A	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	1	1	BE	1	QUANTITY_IN_100KG	10062011	BE	A	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	2	2	BE	1	QUANTITY_IN_100KG	10062011	BG	A	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	3	3	BE	1	QUANTITY_IN_100KG	10062011	CY	A	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 52 : reding trade(COMEXT) data file

If not the data will have to be downloaded using **`get_pcomext`** as follows:

```
In [10]:
products_codes = inv_df[inv_df["BoP"] == "bop_food"]

pcom_raw_df = get_pcomext("2010", "2018", products_codes, ["Belgium"], pcomext_table, "DS-066341")

trade_raw_df = get_pcomext("2010", "2018", products_codes, ["Belgium"], pcomext_table, "DS-016890")
```

Figure 53 : PRODCOM/COMEXT data download

Notice that the codes of the products of interest are extracted from the inventory data frame is by filtering it for the particular BoP. BoP Food in the figure above.

The Appliances and Household Goods BoPs only require PRODCOM and COMEXT data, however, for the BoP Food, also FAO data will have to be retrieved. In order to do so the following steps have to be followed:

1. Get FAO products codes, FAO groups and FAO datasets from the inventory file.

```
In [17]:
1 fao_codes = inv_df[inv_df["Source"] == "FAO"]["prod_codes"].dropna().tolist()
2 fao_codes

Out[17]: ['2531', '1062', '388', '176', '490', '515', '486', '221', '231']

In [18]:
1 fao_groups = list(set(inv_df["FAO_group"].dropna().tolist()))
2 fao_groups

Out[18]: [2531, 2918, 2919, '2615;2616;2618;2619', 2960, 2551, 2943]

In [20]:
1 fao_datasets = list(set(inv_df["FAO_datasets"].dropna().tolist()))
2 fao_datasets

Out[20]: ['57;71', '19;20', 4, '54;71']
```

Figure 54 : FAO products, groups and datasets example.

2. Get a list of the available FAO datasets in the FAO bulkdownload facility.

```
In [29]:
1 fao_ds = fao.fao_datasets({})
2 fao_ds

Out[29]: [{'DatasetCode': 'AE',
  'DatasetName': 'ASTI R&D Indicators: ASTI-Expenditures',
  'Topic': 'All government and nonprofit agencies involved in agricultural research in over 80 low- and middle-income countri
es. Spending for higher education agencies is estimated in most countries assuming that average spending per researcher at hi
gher education agencies is the same as spending per researcher at government and nonprofit agencies. ASTI is currently explor
ing ways to more accurately capture agricultural research spending by universities.Private for-profit agencies are not includ
ed in ASTI datasets.',
  'DatasetDescription': 'ASTI collects primary time-series data on agricultural research capacity and spending levels through
national survey rounds in over 80 low-and middle-income countries. Data collection is carried out by country focal points, wh
o distribute survey forms to all agencies known to conduct agricultural research in a given country, including government, no
nprofit, and higher education agencies. Private-for profit sector coverage is limited, and hence excluded from this dataset.
More detailed country- and regional-level data on agricultural research capacity, investment, and outputs are available on ww
w.asti.cgiar.org/data.',
  'Contact': 'Nienke Beintema and Gert-Jan Stads',
  'Email': 'asti@cgiar.org',
  'DateUpdate': '2019-11-11',
  'CompressionFormat': 'zip',
  'FileType': 'csv',
  'FileSize': '26KB',
  'FileRows': 3094,
```

Figure 55 : FAO available datasets example

3. Download, unzip and read the data from the previously defined datasets of interest.

```
In [30]:
fao_df = fao.fao_extr({}, fao_ds, FAODATA_FOLDER, STORAGE_FOLDER, fao_datasets)
fao_df.head()
```

Out[30]:

	Area Code	Area	Item Code	Item	Element Code	Element	Year Code	Year	Unit	Value	Flag	Note
0	2	Afghanistan	2501	Population	511	Total Population - Both sexes	1961	1961	1000 persons	8954.0	NaN	NaN
1	2	Afghanistan	2501	Population	511	Total Population - Both sexes	1962	1962	1000 persons	9142.0	NaN	NaN
2	2	Afghanistan	2501	Population	511	Total Population - Both sexes	1963	1963	1000 persons	9340.0	NaN	NaN
3	2	Afghanistan	2501	Population	511	Total Population - Both sexes	1964	1964	1000 persons	9547.0	NaN	NaN

Figure 56 : FAO_extr example

4. Filter the data according to the FAO groups and products of interest.

```
In [22]:
fao_products_df = fao.filter_faodata(fao_df, fao_codes, ["Belgium"], "2010", "2018")
fao_products_df.head()
```

Out[22]:

	Area Code	Area	Item Code	Item	Element Code	Element	Year Code	Year	Unit	Value	Flag	Note
670713	255	Belgium	2531	Potatoes and products	5511	Production	2010	2010	1000 tonnes	3456.0	S	NaN
670714	255	Belgium	2531	Potatoes and products	5511	Production	2011	2011	1000 tonnes	4129.0	S	NaN
670715	255	Belgium	2531	Potatoes and products	5511	Production	2012	2012	1000 tonnes	2930.0	S	NaN
670716	255	Belgium	2531	Potatoes and products	5511	Production	2013	2013	1000 tonnes	3428.0	S	NaN
670727	255	Belgium	2531	Potatoes and products	5611	Import Quantity	2010	2010	1000 tonnes	1841.0	S	NaN

```
In [23]:
fao_groups_df = fao.filter_faodata(fao_df, fao_groups, ["Belgium"], "2010", "2018")
fao_groups_df.head()
```

Out[23]:

	Area Code	Area	Item Code	Item	Element Code	Element	Year Code	Year	Unit	Value	Flag	Note
670713	255	Belgium	2531	Potatoes and products	5511	Production	2010	2010	1000 tonnes	3456.0	S	NaN
670714	255	Belgium	2531	Potatoes and products	5511	Production	2011	2011	1000 tonnes	4129.0	S	NaN
670715	255	Belgium	2531	Potatoes and products	5511	Production	2012	2012	1000 tonnes	2930.0	S	NaN
670716	255	Belgium	2531	Potatoes and products	5511	Production	2013	2013	1000 tonnes	3428.0	S	NaN

Figure 57 : Filtered FAO data examples.

4.3.2 Processing raw data

Before calculating the AC, the raw data from the different sources has to be processed in order to have it all in the same format. This will be done by applying successive functions. These will all be the same regardless the data origin, however, the arguments provided may vary.

```
In [31]:
pcom1_df = postdw.rename(pcom_raw_df, "DS-066341")
pcom2_df = pcom1_df.drop(pcom1_df.columns[0], axis = 1)
pcom3_df = postdw.pcode_conversion(pcom2_df, pcomext_table, "DS-066341")
pcom4_df = postdw.ccode_conversion(pcom3_df, "DS-066341")
pcom5_df = postdw.aggregate(pcom4_df, "DS-066341")
pcom6_df = postdw.combine_cols(pcom5_df, "DS-066341")
years = [str(y) for y in list(range(2010, 2019))]
pcom7_df = postdw.reorganize(pcom6_df, years, ["INDICATORS"], [])
pcom8_df = postdw.convert_units(pcom7_df, "DS-066341")
pcom9_df = pcom8_df.replace(0, np.nan)
pcom9_df.head()
```

Out[31]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	10111140	2010	61352000.0	258267700.0	17476700.0	37154730.0	24672872.0	126649996.0
1	Belgium	10111140	2011	68016800.0	293188290.0	18001100.0	47057150.0	24607432.0	128170126.0
2	Belgium	10111140	2012	68032600.0	303171790.0	16727900.0	47724010.0	27489992.0	149660717.0
3	Belgium	10111140	2013	64044400.0	295682780.0	12713800.0	38190850.0	35500969.0	187955307.0

Figure 58 : PRODCOM data processing example

In [32]:

```
trade1_df = postdw.rename(trade_raw_df, "DS-016890")
trade2_df = postdw.aggregate(trade1_df, "DS-016890")
trade3_df = postdw.combine_cols(trade2_df, "DS-016890")
years = [str(y) for y in list(range(2010, 2019))]
trade4_df = postdw.reorganize(trade3_df, years, ["INDICATORS"], [])
trade5_df = postdw.convert_units(trade4_df, "DS-016890")
trade6_df = postdw.pcode_conversion(trade5_df, pcomext_table, "DS-016890")
trade7_df = postdw.ccode_conversion(trade6_df, "DS-016890")
trade8_df = postdw.aggregate(trade7_df, "DS-016890")
trade8_df.head()
```

Out[32]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	Belgium	100000Z4	2017	2.376333e+05	3.969094e+06	1.004067e+06	1.518781e+07
1	Belgium	100000Z4	2018	1.949333e+05	3.772581e+06	7.531333e+05	1.290455e+07
2	Belgium	10111140	2010	1.996380e+07	4.557625e+07	5.454710e+07	2.324149e+08
3	Belgium	10111140	2011	2.000880e+07	5.086414e+07	6.329210e+07	2.659198e+08

Figure 59 : COMEXT data processin example

In [33]:

```
faop1_df = postdw.rename(fao_products_df, "FAO")
faop2_df = postdw.pcode_conversion(faop1_df, pcomext_table, "FAO")
faop3_df = postdw.ccode_conversion(faop2_df, "FAO")
faop4_df= postdw.aggregate(faop3_df, "FAO")
faop5_df = postdw.combine_cols(faop4_df, "FAO")
faop6_df = postdw.reorganize(faop5_df, ["Value"], ["Year"], ["Year"])
years = [str(y) for y in list(range(2010, 2019))]
faop6_df.columns = [str(c) for c in faop6_df.columns]
faop7_df= postdw.reorganize(faop6_df, years, ["INDICATORS"], [])
faop8_df = postdw.rename(faop7_df, "FAO")
faop9_df = postdw.convert_units(faop8_df, "FAO")
faop9_df.head()
```

Out[33]:

	DECL	PRCCODE	Note	Year	EXPQNT	EXPVAL	CONSQNT	IMPQNT	IMPVAL	PRODQNT
0	Belgium	1062	NaN	2010	78354000.0	130240.0	NaN	60931000.0	94204.0	168909000.0
1	Belgium	1062	NaN	2011	46382000.0	117246.0	NaN	56693000.0	97608.0	169727000.0
2	Belgium	1062	NaN	2012	53862000.0	118964.0	NaN	47030000.0	95346.0	153515000.0
3	Belgium	1062	NaN	2013	67729000.0	134891.0	NaN	52743000.0	103597.0	173530000.0

Figure 60 : FAO products data processing example

In [34]:

```
faog0_df = fao.fao_extra(fao_groups_df,inv_df, "FAO_group")
faog1_df = postdw.rename(faog0_df, "FAO")
faog2_df = postdw.pcode_conversion(faog1_df, pcomext_table, "FAO")
faog3_df = postdw.ccode_conversion(faog2_df, "FAO")
faog4_df= postdw.aggregate(faog3_df, "FAO")
faog5_df = postdw.combine_cols(faog4_df, "FAO")
faog6_df = postdw.reorganize(faog5_df, ["Value"], ["Year"], ["Year"])
years = [str(y) for y in list(range(2010, 2019))]
faog6_df.columns = [str(c) for c in faog6_df.columns]
faog7_df= postdw.reorganize(faog6_df, years, ["INDICATORS"], [])
faog8_df = postdw.rename(faog7_df, "FAO")
faog9_df = postdw.convert_units(faog8_df, "FAO_BS")
faog9_df.head()
```

Out[34]:

	DECL	PRCCODE	Note	Year	EXPQNT	CONSQNT	IMPQNT	PRODQNT
0	Belgium	2531	NaN	2010	3.372000e+09	9.241992e+11	1.841000e+09	3.456000e+09
1	Belgium	2531	NaN	2011	3.511000e+09	1.000043e+12	1.930000e+09	4.129000e+09
2	Belgium	2531	NaN	2012	3.857000e+09	9.318065e+11	2.252000e+09	2.930000e+09
3	Belgium	2531	NaN	2013	3.929000e+09	1.037175e+12	2.109000e+09	3.428000e+09

Figure 61 : FAO groups data processing example

Notice that for COMEXT data aggregate is applied twice, the first time to aggregate the partners countries and the second to aggregate the possible repeated products codes after performing the products codes conversion, for it may happen that several COMEXT products codes are converted into the same PRODCOM product code.

4.3.3 Data gap filling

Unfortunately, for many different reasons, there are always missing values in the data retrieved from the different data sources. These missing values will lead, in the best case to slightly different results in the AC, and in the worst case to completely nonsense AC figures. This depends of its amount and distribution. If for instances there is only one missing value for each product and country, even if this happens for all products, it probably will not have a huge effect in the end results. However, if the same number of missing values is localized in a single indicator within a product or group of products, their effect might be much worse. In order to avoid, at least partially, the problems the missing values may lead to two different strategies were followed.

Interpolation

This approach, implemented through function **`interpol`**, is focused on isolated missing values, which will be filled out using the existing surrounding values from other years, as long as these correspond to the same product, country and indicator, i.e. missing values cannot be interpolated with values from other countries, other products or other indicators. In addition to this, a limit of two consecutive missing values, which should be surrounded by existing values belonging to the same country, product and indicators, has been set. This means that, if there are three consecutive missing values, or the missing values belong to the last years of the period considered no modification will be introduced. E.g. if in the period 2010-2018, the values for 2017 and 2018 are missing, they will not be filled out because they are not surrounded by existing values from the same product, country and indicator. This limitation can of course been change within **`interpol`**.

```
In [45]:
1 pcom9_df.head()
```

Out[45]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
2812	Belgium	20421990	2014	NaN	49567780.0	NaN	59203360.0	NaN	1666719.0
2813	Belgium	20421990	2015	NaN	70734740.0	NaN	76097930.0	NaN	NaN
2814	Belgium	20421990	2016	NaN	76298760.0	NaN	90121170.0	NaN	2335741.0
2815	Belgium	20421990	2017	NaN	83152590.0	NaN	86199900.0	NaN	4572968.0

```
In [50]:
1 inter_pcom_df = postdw.interpol(pcom9_df)
2 inter_pcom_df.head()
```

Out[50]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
2812	Belgium	20421990	2014	NaN	49567780.0	NaN	59203360.0	NaN	1666719.0
2813	Belgium	20421990	2015	NaN	70734740.0	NaN	76097930.0	NaN	2001230.0
2814	Belgium	20421990	2016	NaN	76298760.0	NaN	90121170.0	NaN	2335741.0
2815	Belgium	20421990	2017	NaN	83152590.0	NaN	86199900.0	NaN	4572968.0

Figure 62 : interpolation example

In the figure above, in the original data frame *pcom9_df* the "PRODVAL" value for product 20421990 for Belgium in 2015 was missing. Since the missing value was surrounded with existing values from the same country, product and indicator, **`interpol`** filled out the missing value with a figure calculated through the interpolation of the closets values, as it can be in the resulting data frame *inter_pcom_df*.

Trade filling

This second approach is more focused on large numbers of consecutive missing values, although it can of course tackle single isolated values. The key idea behind this approach is using COMEXT data to fill out the missing values in the PRODCOM data and is implemented through three different functions**, `prices`**, **`apply_prices`** and **`fill_gaps`**.

The first step is applying the **`prices`** function on the already processed COMEXT data. This function calculates the products imports and exports prices by dividing the indicators quantities columns by their corresponding indicator values columns.

```
In [25]:
1 trade8_df.head()
```

Out[25]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL
0	Belgium	14111000	2010	747700.0	49580680.0	829400.0	36779619.0
1	Belgium	14111000	2011	933000.0	52026865.0	838800.0	36665747.0
2	Belgium	14111000	2012	642100.0	43417240.0	1052800.0	34111340.0
3	Belgium	14111000	2013	608800.0	35455199.0	602300.0	37292808.0
4	Belgium	14111000	2014	356100.0	33473809.0	640500.0	41478819.0

```
In [24]:
1 prices_df = postdw.prices(trade8_df)
2 prices_df.head()
```

Out[24]:

	PRCCODE	DECL	Year	EXPPRICE	IMPPRICE	AVGPRICE
0	14111000	Belgium	2010	66.310927	44.344850	55.327889
1	14111000	Belgium	2011	55.762985	43.712145	49.737565
2	14111000	Belgium	2012	67.617567	32.400589	50.009078
3	14111000	Belgium	2013	58.237843	61.917330	60.077587

Figure 55 : `prices` example

"AVGPRICE" column values are calculated through an average of "EXPPRICE" and "IMPPRICE" values.

The second step is using the previously calculated products prices to calculate missing values in the PRODCOM processed data through **`apply_prices`**.

```
In [27]:
1 pcom9
```

Out[27]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN
4	Belgium	14111000	2014	NaN	49328650.0	NaN	73703310.0	NaN	NaN

```
In [28]:
1 filling_df = postdw.apply_prices(pcom9_df, prices_df)
2 filling_df.head()
```

Out[28]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	784237.869630	NaN	1.796743e+06	NaN	NaN	NaN
1	Belgium	14111000	2011	861913.149101	NaN	1.764831e+06	NaN	NaN	NaN
2	Belgium	14111000	2012	660026.702296	NaN	2.052531e+06	NaN	NaN	NaN
3	Belgium	14111000	2013	748465.388334	NaN	1.033950e+06	NaN	NaN	NaN

Figure 63 : `apply_prices` example

In the figure above, in the processed PRODCOM data frame pcom9_df, all product 14111000 “EXPQNT”, “IMPQNT”, “PRODQNT” and “PRODVAL “ values for Belgium and years 2010 to 2014 are missing. Function **`apply_prices`**, divides “EXPVAL” by “EXPPRICE”, “IMPVAL” by “IMPPRICE” and “PRODVAL” by “AVGPRICE”. Then it also multiplies “EXPQNT” by “EXPPRICE”, “IMPQNT” by “IMPPRICE” and “PRODQNT” by “AVGPRICE”. As it can be notices, this leads to nothing for “PRODQNT” and “PRODVAL”, since there were no values for either.

The last step is filling the gaps in the PRODCOM data frame using the values calculated through **`apply_prices`**. This is done with **`fill_gaps`**.

```
In [27]:
1 pcom9
```

Out[27]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	NaN	52003540.0	NaN	79676300.0	NaN	NaN
1	Belgium	14111000	2011	NaN	48062850.0	NaN	77144540.0	NaN	NaN
2	Belgium	14111000	2012	NaN	44629400.0	NaN	66503210.0	NaN	NaN
3	Belgium	14111000	2013	NaN	43589010.0	NaN	64019430.0	NaN	NaN
4	Belgium	14111000	2014	NaN	49328650.0	NaN	73703310.0	NaN	NaN

```
In [29]:
1 filled_pcom_df = postdw.fill_gaps(pcom9_df, filling_df)
2 filled_pcom_df.head()
```

Out[29]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	14111000	2010	784237.869630	52003540.0	1.796743e+06	79676300.0	NaN	NaN
1	Belgium	14111000	2011	861913.149101	48062850.0	1.764831e+06	77144540.0	NaN	NaN
2	Belgium	14111000	2012	660026.702296	44629400.0	2.052531e+06	66503210.0	NaN	NaN
3	Belgium	14111000	2013	748465.388334	43589010.0	1.033950e+06	64019430.0	NaN	NaN

Figure 64 : `fill_gaps` example

The two strategies are meant to be complementary and to be used together as in . It may be worth it to mention that, the order in which the two of them are applied does not change the results in terms of filled out missing values.

```
In [24]:
1  prices_df = postdw.prices(trade8_df)
2
3  filling_df = postdw.apply_prices(pcom9_df, prices_df)
4
5  filled_pcom_df = postdw.fill_gaps(pcom9_df, filling_df)
6
7  inter_pcom_df = postdw.interpol(filled_pcom_df)
8
9  inter_pcom_df.head()
10
```

Out[24]:

	DECL	PRCCODE	Year	EXPQNT	EXPVAL	IMPQNT	IMPVAL	PRODQNT	PRODVAL
0	Belgium	10111140	2010	61352000.0	258267700.0	17476700.0	37154730.0	24672872.0	126649996.0
1	Belgium	10111140	2011	68016800.0	293188290.0	18001100.0	47057150.0	24607432.0	128170126.0
2	Belgium	10111140	2012	68032600.0	303171790.0	16727900.0	47724010.0	27489992.0	149660717.0
3	Belgium	10111140	2013	64044400.0	295682780.0	12713800.0	38190850.0	35500969.0	187955307.0

Figure 65 : Data gap filling example

4.3.4 Calculation

Once all the data from all the sources has been processed, the formats harmonized and the missing values filled out, to the possible extent, it has to be put all together in a single data frame,

and then the representative products AC per capita can be calculated by applying **consumption_intensity**.

```
In [61]:
1 all_df = pd.concat([inter_pcom_df, faop9_df], sort = True)
2 ci_all = consumption_intensity(inv_df, all_df, faog9_df, "QNT")
3 ci_all["Belgium"]
```

Out[61]:

	BoP	prod_groups	Source	FAO_group	FAO_datasets	prod_codes	prod_names	rep_prod_names	
0	bop_food	PG_Meat	EUROSTAT	2943	19;20	10113230	Frozen carcases and half-carcases, of pig meat	RP_PigMeat	10111230;10111250;10111290;10113
1	bop_food	PG_Meat	EUROSTAT	2943	19;20	10111190	Fresh or chilled cuts, of beef and veal	RP_BeefMeat	10111140;10
2	bop_food	PG_Fish&SeaFood	EUROSTAT	2960	19;20	10202425	Smoked Pacific, Atlantic and Danube salmon (i...	RP_Salmon	10201100;10201200;10201330;10201

Figure 66 : `consumption_intensity` example

Notice that the result of this function is a dictionary of data frames, one for each country considered in the data inputs. It is possible to extract each of them by calling it with the country name.

4.4 Running BOP Mobility

4.4.1 Loading data

Here the only data that has to be loaded is the mobility frame file (see 3.2.3), which contains the information of the different means of transport types considered. This is done like in 4.2.1 with **`data_inputs`**.

```
In [2]:
1 # Loading the mobility frame file
2
3 mob_frame = di.read_file("mobility_frame.xlsx", INPUTS_FOLDER, 0, 0, {})
4 mob_frame.head()
```

Out[2]:

	transport	vehicle_type	code	type	age	id_1	id_2	id_3
0	Road transport	Passenger_Cars	SP 1	Gasoline <1.4 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC_LT1400	PET
1	Road transport	Passenger_Cars	SP 2	Gasoline <1.4 L	Euro_4	EU4	CC_LT1400	PET
2	Road transport	Passenger_Cars	SP 3	Gasoline <1.4 L	Euro_5	EU5	CC_LT1400	PET
3	Road transport	Passenger_Cars	SP 4	Gasoline 1.4 - 2.0 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC1400-1999	PET

Figure 67 : mobility frame example

Calculation

For BoP Mobility, the only other step is running **`mobility`** with the parameters of interest.

In [17]:
```
czechia_mobility = mob.mobility(mob_frame, "2010", "2010", ["Czechia"], "pb2019-section23.xls", INPUTS_FOLDER)
czechia_mobility.head()
```

Out[17]:

	transport	vehicle_type	code	type	age	id_1	id_2	id_3	geo\time	year	...	flight_km	contr_pkm
0	Road transport	Passenger_Cars	SP 1	Gasoline <1.4 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC_LT1400	PET	CZ	2010	...	NaN	NaN
1	Road transport	Passenger_Cars	SP 2	Gasoline <1.4 L	Euro_4	EU4	CC_LT1400	PET	CZ	2010	...	NaN	NaN
2	Road transport	Passenger_Cars	SP 3	Gasoline <1.4 L	Euro_5	EU5	CC_LT1400	PET	CZ	2010	...	NaN	NaN
3	Road transport	Passenger_Cars	SP 4	Gasoline 1.4 - 2.0 L	Conventional;Euro_1;Euro_2;Euro_3	CONV;EU1;EU2;EU3	CC1400-1999	PET	CZ	2010	...	NaN	NaN

Out[17]:

	id_2	id_3	geo\time	year	...	flight_km	contr_pkm	mill_pkm	mill_vehikm	eng_coeff	tech_coeff	ocp_factor	num_vehi_share	trkm_share	Result
	C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.777714	1.62	NaN	NaN	20075.956248
	C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.186254	1.62	NaN	NaN	4807.986548
	C_LT1400	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.657839	0.036032	1.62	NaN	NaN	930.133533
	CC1400-1999	PET	CZ	2010	...	NaN	NaN	63570.0	NaN	0.305327	0.777714	1.62	NaN	NaN	9317.993576

Figure 68 : `mobility` example