Exercise 2 – Part 1

General instructions:

- Fill your code in the skeleton_part1.py file provided with the assignment, as instructed in this document (replace pass statements in functions with your implementations)
- Ensure that your file 'complies' before submitting (does not include any syntax errors)
- To validate your solution thoroughly read the requirements and ensure you cover all
- In questions, in which you are requested to print or raise an error, ensure that the message exactly matches the expected one (don't forget whitespaces, newlines, etc.); if you raise an exception, use the type that is specified (e.g., ValueError)
- In this exercise, you are **only** expected to handle erroneous that are specified in this document
- For this part, submit a single .py file. Rename skeleton_part1.py file to include 'part_1_' followed by the submitted ids (including 'sifrat bikoret'). E.g., part_1_012345678_112345678_212345678.py
- The due date is published in Moodle
- The submission is in triples, unless approved
- To make your code more modular and avoid code duplication, you are **encouraged** to add auxiliary functions in the skeleton file
- Try to write efficient solution
- **Do not change the functions declaration** defined in the skeleton
- Only place code inside functions (do not add any global variables unless instructed to)
- Do not use reserved words as variable/function names (e.g., don't use len, list, dict)
- If more imported are required, place them at the top of the functions that require them (this is not the best practice, but required for the ease of testing)
- At the end of each question you will find clarifying examples

Notes for this part of exercise 2:

- The data you will require can be found in the data.pickle file in this zip; extract it to the script folder (e.g., where skeleton_part1.py is located)
- The skeleton file includes a main function, use it and fill the code in the missing parts
- You may change the main function, but it will not be checked, any changes to __main__ should be made for the purpose of developing and testing your code
- Only the functions that are described below will be checked
- Ensure that your solution **does not relay** on variables defined in the global scope or in the __main__ function
- `pd`: refers to the pandas module
- `np:` refers to the numpy module
- `df:` refers to an instance of pandas.core.frame.DataFrame
- `sr:` refers to an instance of pandas.core.series.Series
- You are only required to handle errors that are specified:
  - Ensure that you raise the specified Error Type and the it includes the exact error message as described in this document
- In all functions, return the correct types (e.g., do not confuse a list and pd.Series)
- In some questions, you will be required to return the n smallest/biggest elements. If there are more than n elements that have the smallest/biggest values (e.g., a list where all elements have the same value), you may arbitrarily return n of the smallest/biggest.
  For example, assume you have a dictionary that maps students to exam score, and holds the scores of a hundred students. You are requested to return five students with the highest scores. Now let's assume that ten students got a 100, then you may choose any five of these students

Part A: Basic Operations

- In the following questions, you may assume that dataframe in the 'data.pickle' file has the following structure:
  - Area: object
  - Item: object
  - Element: object
  - Year: int64
  - Unit: object
  - Value: float64
- The column Element only contains the values: ['Export Quantity', 'Export Value', 'Import Quantity', 'Import Value']
- We refer to this format at format_1

Part A.1

```
def get_total_rows(df):
    '''
    :param df: pd.Dataframe in format_1
    :return: int
    '''
    pass
```

Complete the body of the function `get_total_rows(df)` that gets a `df` and returns the number of rows it contains

An example execution and its output:

```
>>> total_rows
10418605
```

No Error handling is required

Part A.2

```
def get_sorted_columns(df):
    '''
    :param df: pd.Dataframe in format_1
    :return: list
    '''
    pass
```

Complete the body of the function `get_sorted_columns(df)` that gets a `df` and returns a sorted list of its columns (sorted lexicographically in ascending order)

An example execution and its output:

```
>>> columns_sorted
['Area', 'Element', 'Item', 'Unit', 'Value', 'Year']
```

No Error handling is required

Part A.3

```
def count_unique_values(df):
    '''
    :param df: pd.Dataframe in format_1
    :return: pd.Series
    '''
    pass
```

Complete the body of the function `count_unique_values(df)` that receives a `df` in format_1 and returns a pd.Series with the number of unique values in each columns ; the series index should be `df.columns`.

An example execution and its output:

```
>>> sr = count_unique_values(df)
>>> type(sr)
      pandas.core.series.Series
>>> print(sr)
      Area          243
      Item          469
      Element         4
      Year            1
      Unit            2
      Value       56859
      dtype: int64
```

No Error handling is required

-- Part A.4 --

```
def get_index_as_list(df, column_index):
    '''
    :param df: pd.Dataframe in format_1
    :param column_index: boolean, if True returns the column index, otherwise, the
row index
    :return: list
    '''
    pass
```

Complete the body of `get_index_as_list(df, column_index):`
The function obtains a `df`, a *Boolean* variable `column_index` and returns:

a. its columns as a list if the `column_index=True`

b. its rows index as a list if the `column_index=False`

An example execution and its output:

```
>>> res1 = get_index_as_list(df, True)
>>> res1
['Area', 'Item', 'Element', 'Year', 'Unit', 'Value']

>>> res2 = get_index_as_list(df, False)
>>> res2[:4]
[0, 1, 2, 3, 4]
```

No Error handling is required

-- Part A.5 --

```
def find_min_year(df):
    '''
    find the smallest year value in df
    :param df: pd.Dataframe in format_1
    :return: np.int64
    '''
    pass
```

Complete the body of the function `find_min_year(df)`. The function should return the minimal year (smallest np.int64 in the dataframe in the "Year" column)

```
>>> year = find_min_year(df)
>>> year
1961
```

No Error handling is required

--------------------------------------- Part A.6 ---------------------------------------

```
def apply_fun_over_numric_columns(df, columns, fun):

    '''
    applies function fun over the columns of df
    :param df:  pd.Dataframe in format_1
    :param columns: list of columns that fun should be applied over
    :param fun: a numpy function from the following list:
        np.prod, np.std ,np.var , np.sum , np.min, np.max, np.mean , np.median
    :return: a pd.Series: its index is columns; its values is the result of
applying fun over the selected columns;
    :raise: ValueError
    '''
    pass
```

Complete the body of the function, `apply_fun_over_numric_columns(df, columns, fun)`, that obtains list of column names, a function of the following list:

- np.prod
- np.std
- np.var
- np.sum
- np.min
- np.max
- np.mean
- np.median

The function applies `fun` over the selected columns of the `df`; The function should return a pd.Series; its index is the selected columns (i.e., `columns`), its values are the result of applying `fun` over the selected columns.

An example execution and its output:

```
>>> year_value_means = apply_fun_over_numric_columns(df, columns, np.mean)
>>> year_value_means

Value    30855.756797

Year      1987.901231

dtype: float64
```

Raising exceptions (instruction and checking order):

1. If one of the values in `columns` is not a numeric column or does not appear in the `df` (namely, "Year" or "Value") raise a `ValueError` exception with one of the following messages:
   a. Case 1: if the column does not appear in the dataframe columns,
      Error message: "The specified column is missing from the dataframe"

   b. Case 2: if the column appears as a dataframe column, but is not numeric, Error message: "The specified column must be numeric"

2. If the provided function is not one of the specified ones, raise a `ValueError` exception with the following message
   a. "Invalid function! function must be one of np.prod, np.std, np.var, np.sum, np.min, np.max, np.mean, np.median"
3. First check the values in `columns` according their appearance in `columns` (not in the `df`), then check the `fun` parameter

Guidance: use pd.Dataframe.apply

First check requirement 1 programmatically (see **Raising exceptions**). Instead of checking if the column name is "Value" or "Year", check if the column name is in df.columns and check that it is numeric (use np.issubdtype(pd.Series, np.number))

Note, you have just wrote a very generic and useful function that process a dataframe and applies a function over a subset of its numeric columns

Part B: Reshaping

As we discussed in class, in the provided dataframe the description of imports and exports is split across two types of rows, ones describing the import/export quantities (in tonnes) and the others the import/export values (in 1000$).

For example the following two rows describe that the country 'Gabon' imported '20' 'tonnes' of 'Almont Shelled' at a value of '27' '1000 us$'

Area	Item	Element	Year	Unit	Value
Gabon	Almonds shelled	Import Value	2000	1000 US$	27
Gabon	Almonds shelled	Import Quantity	2000	tonnes	20

To make the analysis of the Part B easier (to be published), we would like to reshape the dataframe, so that each combination of (Area, Item, Year) has a single row of Import information and a single row of Export Information. In other words, we would like the quantity and the price to be in the same row. Furthermore, we would like to drop the unit column as it is takes the same value across the dataframe (either tonnes for quantity or 1000$ for value).

Finally, we would like rows that correspond to import information ("Import Value", "Import Quantity") to have an "Import" Element value and rows with export information to have "Export" Element value. Therefore, the above rows should be merged to a single row:

Area	Item	Element	Year	Quantity(tons)	Price(k,usd)
Gabon	Almonds shelled	Import	2000	27	20

```
def reshape(df):
    '''
    The function joins rows that share ('area', 'item', 'year')
    if they are of export type or of import type
    Rows that only have a single export / import values are removed;
    :param df: pd.Dataframe of format_1
    :return: a pd.Dataframe of format_2
    '''
    pass
```

Complete the code of the function reshape.

The function obtains a dataframe in format_1 and returns a dataframe in format_2 (see below).

Rows that share **("Area", "Item", "Year")** values should be **merged** if:

1. One row contains 'Import Quantity' and the other 'Import Value' in the 'Element' column
2. One row contains 'Export Quantity' and the other 'Export Value' in the 'Element' column

Notes:

1. Combinations of ("Area", "Item", "Year") that only includes import quantity information or import value information should be discarded
2. Combinations of ("Area", "Item", "Year") that only include export quantity information or export value information should be discarded

General guidance:

- Remove the unnecessary column ("Unit")
- You may use `pd.groupby` to split `df` according to the Element field
- Use `pd.merge` to merge the "Import Value" and "Import Quantity" groups to a single dataframe (make sure that you use the right merge policy: inner, outer, left, right)
- Follow similar steps for the export rows
- Add the new "Element" column to each of the dataframes
- Concatenate the two dataframes
- Use `pd.rename` to rename the 'Value' columns to Quantity(tons), Price(k,usd)

Note: Since handling the import rows and the export rows is very similar, consider adding an auxiliary function. This would allow you to avoid code duplication.

Useful functions: `pd.Datagrame.drop`, `pd.Datagrame.rename`, `pd.Groupby.get_group`, `pd.Groupby.merge`

Validation: The main function includes validating function; ensure that it passes before you continue to the other Parts B

- The output of the reshape method is a dataframe of the following format:
    - Area: object
    - Item: object
    - Year: int64
    - Quantity(tons): float64
    - Price(k,usd): float64
    - Element: object
- Element only contains the values: ['Import', 'Export']
- We refer to this format at format_2