Exercise 2 – Part 2

General instructions:

- Fill your code in the skeleton_part2.py file provided with the assignment, as instructed in this document (replace pass statements in functions with your implementations)
- Ensure that your file runs before submitting (does not include any syntax errors)
- To validate your solution make sure you cover all requirements
- In cases where raising exception and error-message printout are required, make sure that the **exception type** you raise and the error-message output match the requirement **precisely.** (e.g., raise 'ValueError' and print the exact message required)
- handle only errors specified in this document
- For this part, submit a single .py file. Rename 'skeleton_part_2.py' file to include 'part_2_' followed by the submitted ids (including 'sifrat bikoret'). E.g., 'part_2_012345678_112345678.py'
- The due date of exercise 2 is valid for both parts 1 and 2 and is published in Moodle.
- The submission is in doubles or singles only.
- You are encouraged to add auxiliary functions to the skeleton functions, in order to make your code modular and reduce code duplication.
- **Do not change the defined functions declaration**
- Write your code only inside function's body (do not add global variables unless specifically instructed to).
- Do not use reserved standard-library words as variable/function names (e.g., len, list, dict, bool)
- If additional libraries are required, place imports at the top of the functions that require them (not the best practice, but used here for the ease of grading)
- At the end of each question you will find clarifying examples

Notes for this part of exercise 2:

- The data for this part is generated by the __main__ function in skeleton_part1.py; The pickle file **fixed_df.pickle**, should be written in the script's folder
- Here, you are requested to complete the code of skeleton_part2.py
- The skeleton file includes a main function, use it to fill the code under the function declarations. DO NOT COMMENT OUT THE MAIN DECLARATION!
- You may change the main function, but it will not be checked. Changes in the __main__ are recommended for the purpose of testing & validation of your code.
- Make sure your solution *does not depend on globals* at the main or out of the main.
- **Only functions' code will be graded**
- `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
- Make sure to **raise the specified Error Types** and **include the exact error messages** required.
- Make sure that the functions' return types are correct (e.g., do not confuse a list and pd.Series) – for this, you can add function-testing line in the main by printing out both the value and type of the returned element.
- 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.

the __main__ function includes validation statements.

If you are unable to complete the code with the data output from part 1 of exercise 2, use `fixed_df.pickle` file provided in this part. (copy-paste it into your part2-script folder).

- The `fixed_df.pickle` holds a dataframe of the following format **(format_2)**:
  - Area: object
  - Item: object
  - Year: int64
  - Quantity(tons): float64
  - Price(k,usd): float64
  - Element: object
- "Element" is a categorical feature, containing the values: ['Import', 'Export']

Part C: Filter & Groupby

Important remark: You may assume in all functions bellow that the `df` structure is of **format_2**

---------- Part C.1 ----------

```
def find_most_frequent_year(df, n):
    '''
    :param df: pd.Dataframe in format_2
    :param n: positive, integer;  n'th most frequent year
    :return: int; the n'th most frequent year
    :raise: ValueError - if n is not a positive integer'''
    pass
```

Complete the body of the function `find_most_frequent_year(df, n)`, which receives the `df` in format_2 and a strictly positive integer `n` (`n` >0), and **returns the `n`'th most frequent year**.
In other words,

- count the number rows each year appears in "Year" column in `df`. Assign it to year_counts as local variable
- Then, sort the year_counts in **descending** order.
- the `n`'th most frequent year would be the `n`'th element in the sorted descending series.
- Note: If several years have identical year_counts, you can ignore repetitiveness, and simply return the `n`'th element in the sorted vector.
- Caution: python indexing starts at 0.

For example, consider the `df` in the example below.
If we count frequency of year appearances and sort by year_counts in descending order we get:
(514322, 2000), (30310, 2008), (4329, 1962)
For `n=1` the year 2000 should be returned
For `n=2` the year 2008 should be returned
For `n=3` the year 1962 should be returned
Notice: if there are repetitive year counts, e.g., [(3528,2008), (3528,1976)] (not shown here) the order of n retains, ignoring identical counts. Therefore, both years will be accepted as a correct answer (i.e., 2008 or 1976).

Example:

Year_counts	Area	Item	Element	Year	Unit	Value
1	Afghanistan	Almonds shelled	Export Quantity	2007	tonnes	0
30310	Albania	Chick peas	Export Quantity	2008	tonnes	NaN
514322	Austria	Cotton linter	Import Value	2000	1000 US$	1
4329	Afghanistan	Fruit, fresh nes	Export Quantity	1962	tonnes	5900

```
>>> df = pd.read_pickle("fixed_df.pickle")
```

```
>>> find_most_frequent_year(df, 4)
2007
>>> find_most_frequent_year(df, 10)
2004
```

Raising exceptions:

Raise `ValueError` if the user provides **n** that is **not** a positive integer, with error message: "n must be a strictly positive integer (>0)"

Don't confuse raising an exception and printing!!!

---------- Part C.2 ----------

```
def filterby_criteria(df, criteria):
    '''
        The function filters all rows that do not have values
        that are included in the dict criteria
    :param df: pd.dataframe of in format_2
    :param criteria: a dictionary mapping columns to values to keep;
    key-- is a column name (string); values- list of values (of any object) to keep
    :return: a df without the rows that do not include the specified values
    :raise: ValueError if key is not in dataframe
    '''
    pass
```

Complete the code of the function `filterby_criteria`. The function obtains a `df` in format_2 and a dictionary that maps columns to values. The function slices rows (indexes) such that it returns only rows whose values match 'criteria' keys values.

Hint: use pd.Dataframe.isin()

Example Run:

```
df_isr13 = filterby_criteria(df, {"Area":["Israel"], "Year":[2013]})
```

Only rows with "Year"=2013 and "Area"="Israel" are included in the output.
See head of an expected result:

	index	Area	Item	Year	Quantity(tons)	Price(k,usd)	Element
934168	934168	Israel	Alfalfa meal and pellets	2013	400.0	100.0	Export
934221	934221	Israel	Almonds shelled	2013	546.0	3665.0	Export
934274	934274	Israel	Anise, badian, fennel, coriander	2013	11.0	71.0	Export
934327	934327	Israel	Apples	2013	15020.0	14570.0	Export
934380	934380	Israel	Apricots	2013	76.0	174.0	Export
934433	934433	Israel	Apricots, dry	2013	0.0	0.0	Export

Raising exceptions:

Raise a `ValueError` if the dictionary includes a key that is not in `df.column`

Error message: "$key is not a column in df", where $key is the missing key.

Check the columns in lexicographical order. You may assume that 'criteria' is a dictionary (if 'criteria' dictionary is empty, return df unchanged).

---------- Part C.3 ----------

```
def find_extremes(df, fun=np.sum, by_item=True, by_value=True, n=5):
    '''
    :param df: pd.DataFrame in format_2; only including importing/exporting rows
    :param by_item: If True group rows by item, otherwise by Area
    :param by_value: If True should find the n most extreme items by value,
otherwise by quantity
    :param fun:  a function to be applied, one of: np.prod(),np.std(),np.var(),
    np.sum(), np.min(), np.max(), np.mean(), np.median()
    :param n: if positive get the least extreme items, otherwise, get the most
extreme items
    :return: list of the n least/most imported/exported items/areas
    raise: ValueError if n is not an int, or if n == 0
    raise: ValueError df contains both import and export values
    raise: ValueError fun is not one of the specified above functions
    '''
    pass
```

Complete the body of the function `find_extremes`.

Function requirements:
The function **returns a list of the** `n` ***least/most imported/exported items/areas*** **by** ***value/quantity*****.**

- for this, **np.Nan's and np.inf's are removed before slicing**. you may use the following statements: `replace([np.inf, -np.inf], np.nan).dropna()`
- The function groups the rows by *items or* by *areas*, then applies `fun` over the *value or over quantity* of each group.
- If `by_value=True`, then the `'Price(k,usd)'` column is used (per item), otherwise the `'Quantity(tons)'` column is used.
- The elements (*items/areas*) are then sorted by their values.
- Finally, if `n` is positive, a `list` of smallest `n` elements is returned, otherwise a list of the largest n elements is returned.

Function arguments and output:

- a `df` in format_2 with rows that include only import/export information
- `by_item`, Boolean: if True group rows by 'Item', otherwise by 'Area'
- `by_value`, Boolean: if True find the most extreme elements (either 'Area' or 'Item') by value, otherwise by quantity
- `fun`: a function to be applied over each group, one of: `np.prod(),np.std(),np.var(), np.sum(), np.min(), np.max(), np.mean(), np.midian()`
- `n`, an integer, **if positive** get the **least extreme elements**, otherwise, get the most extreme elements

You may assume that after applying `fun`, there are at least n elements that are not np.nan, np.inf, -np.inf
Hint: use pd.Groupby; pd.GroupBy.apply()

Example Run:

```
>>> df_exp = filterby_criteria(df, {"Element": ["Export"], "Year":[2013]})
>>> find_extremes(df_exp, by_item=False, by_value=True, n=-5)
['France', 'Netherlands', 'Germany', 'Brazil', 'United States of America']
```

Raising exceptions:

- Raise a `ValueError` if `n` is not an integer or if n == 0;

Error message: 'n must be an integer different than zero'

- Raise a `ValueError` if the 'Element' column contains both 'Import' and 'Export' values

Error message: 'The dataframe must only include Import or Export rows'

(i.e., df["Element"] must be unique)

- Raise a `ValueError` if `fun` does not include one of the specified functions

Error message: 'Invalid function! function must be one of np.prod(),np.std(),np.var(), np.sum(), np.min(), np.max(), np.mean(), np.midian()'

---------- Part C.4 ----------

```
def generate_scatter_import_vs_export(df, countries, year, output):
    '''
    The function produces a scatter plot of the total import/export values of the
specified countries in the specified year
    :param df: a dataframe in format_2
    :param countries: a list of strings specifying countries to include
    :param year: int; only rows of the specified year are used
    :param output: a filename (str) to which the scatter plot is to be saved
    :return: None; saves the plot to output.png
    '''
    pass
```

The function '`generate_scatter_import_vs_export`' generates a scatter plot of the total **export values** as function of the **import values** for a specified list of `countries` in a specific `year`.

The function saves the scatter plot in png format `output` file.

Function arguments:

- `df` is a dataframe in format_2,
- `countries` a list of required countries
- `year` is an integer of required year
- `output` is a string of the scatter output file name

Scatter plot has the following characteristics:

- Y_axis has a label 'Exports'
- X_axis has a label 'Imports'
- Figure title is 'Exports as function of imports'
- Each point should be labeled with its corresponding country
- Function return value is the figure reference object.
  hint: use a variable to store the figure object, then return that variable at the end. e.g., ax = temp_df.plot.scatter(…) …. return ax
- **Bonus (2pt)**: color the points according to the ratio between Export to Import: use green for countries with larger ratios (i.e., countries that export more than they import) and red for countries with small ratio. Use the red-yellow-green color maping: 'RdYlGn':

  ```
  cmap = matplotlib.cm.get_cmap('RdYlGn')
  ```

- **Bonus(2pt)**: set the size of each point to the proportion of its total export+import value

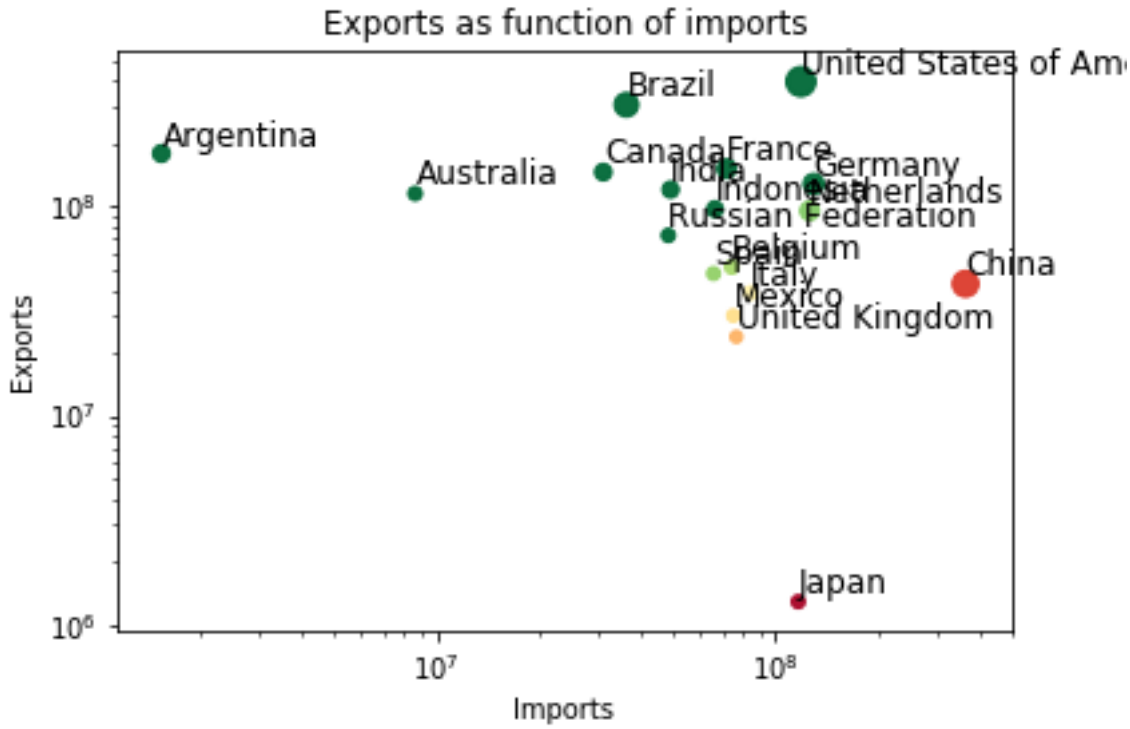
Remark: you are recommended to use `filter_criteria` in your implementation

Hint: use pd.Groupby; pd.Groupby.sum; pd.concat; pd.Dataframe.rename; pd.Dataframe.plt.scatter; matplotlib.axes._subplots.AxesSubplot (to define plot characteristics)

No exception handling is required

Example Run:

```
>>> most_exp_countries = find_extremes(df_exp, by_item=False, by_value=True, n=-12)
>>> most_imp_countries = find_extremes(df_imp, by_item=False, by_value=True, n=-12)
>>> countries = list(set(most_exp_countries + most_imp_countries))
>>> generate_scatter_import_vs_export(df, countries=countries, year=2013,
output='import_vs_export')
```



You are done!!!

Great job!