Metropolis algorythm applied to electoral laws

A good electoral law should be a decent balancement of both representivity and governability. The first principle can lead to proportional laws, the second to majoritary law. So in the last years electoral laws mixing these two features were experimented all over european democracies.
Let's take into account the current italian electoral law and let's focus only on the election of "La Camera dei Deputati". There are 630 deputates in the "small chamber". The 61% is elected in a proportional way(party x gets N% votes, it gets 0.N*0.61*seats). The remaining 37% is elected thorough the collegia mechanism. This means the italian territory is split in 212 electoral collegia. In each collegium different electoral coalitions(or single parties) fight to "win that collegium". In fact, if X coalition/party reaches the highest percentage in the land which represents the collegium it will take that seat in the Parliament.
The question rising up is the following: is it possible to foresee the assignment of the seats in the Parliament, only knowing the national results? The answer, as this program seems to prove, is yes.
The most obvious answer would be to assign the number of collegia according to the national results (X party gets N% votes, it will have N*0.37 seats in the Parliamento). This couldn't be further from the truth because if a party gets a score of 40% it is almost sure that it's going to win more than the 40% of collegia.
Immagine we have three main coalitions (A = 40%,B = 30%,C =20%) and let's consider a single collegium. If we consider the results as probability weight, we can consider to moltiply every score to a random number between 0 and 1. The seat will be obtained by the highest score vehiculated by a random number.
If we repeat this procedure over all the collegia, eventually we will be able to map all the national territory and to reproduce with decent accuracy the national results.
Of course there are many flaws to this algorithm which is totally unable to foresee very localized electoral exploits of some regional parties. So if a party gets the 1% on national scale and in a specific region with 3 collegia it has the 50% it will get 3 seats and no way to foresee that. Actually if we run very long simulations (more than a million times) a

single anomalous result like that can be reproduced.
The algorithm might also be used to highlight some shortcoming of the specific electoral law we're dealing with. It might happen(it sounds absurd) that two parties obtaining almost the same results on national scale will have a completely different assignment of seats. Montecarlo could see that. In fact it is provided an histogram in the graphic part in which the distribution of different electoral simulation for each party are confronted. There is a slightly possibility that a party which got a lesser result could benefit a larger assignment of seats(overlapping between the tails'distributions).
The appropriate number of steps for each simulation can be determined by knowing how many steps need to be performed to compute pi by exploiting Montecarlo methods.

How the repository works

in the file ***Electoral_Montecarlo.py***, a class Montecarlo_electoral is defined. All the methods that will be used in ***Main_electoral.py*** are contained within it.
The program also have a pytest routine **texting_eletcoral.py** where all the methods are tested First of all I have defined the method max_key that returns the key with maximum value within a dictionary. Its purpose will be clear after.

- We start with the definition of the constructor of our class that takes two defaults argument. The first is chooseinput by which we can decide which type of input we can use. It is set on stinput that will allow to write the parameters we need from the standard input of the Console. There are two inputs, "excel" and "txt" that allows to harvest the data from excel or txt files. The second argument of the constructor is hence the filename(including its filepath) of the file we want to pick up data from. If a different word from *stinput,excel,txt* is inserted a ValueError will be raised of course.

- Different variables of the class are defined. In particular we set the number of Deputies and the majoritary and proportional coefficients as the italian ones, but they can change as soon as we import results from different source.
  That is done via the second method, **Import_Results()**. Through this method we acquire the values for the main paremeters we are interested at, i.e. the name of the parties and their Results in the elections(or in polls).
  The filling of the parameters can be performed in three different ways, depending on the input we have decided. For what concerns outer input, i.e. excel or txt files, a FileNotFoundError will be raised if filename doesn't correspond to any existing file on our computers.

- The way data has to be assembled in a txt files is the following:
- The way data has to be assembled in an excel file is the following
- We can highlight that eventually the results we get is homogeneous for every input method we have chosen. Eventually we put together the parties'names and their results in a dictionary called Results and that will play a key role in the algorithm.
- The method **check_input()** is important as well because it verifies that the data have been acquired accordingly. This means that the names of the parties must be alphanumeric, that the results of the parties must be interpreted as a number and that they must be consistent (the sum of the results can't be larger than 1, let alone the value of a single party) and both maj and prop coefficients must be consistent too (their sum can't be larger than 1 of course).
  If everything is correct, strings are converted into float and we can proceed with the algorithm.
- The method **Fill_Seats()** is the core of the program since the algorithm is whithin it.
  Starting from our Results dictionary at first we assign the seats corresponding to the proportional part and then we run a for cycle over all electoral collegia to assign the others, through the Montecarlo method I have explained above. Eventually this method returns a dictionary with the name of the parties as keys and the number of Seats achieved in a simulation.
- Since we are dealing with stocastic processes, we want to eliminate aleatority and we can do that by performing many times (1000) **Fill_Seats()**. That is done thorugh **Complete_Simulation()** which eventually returns a dictionary of the same shape of **Fill_Seats()** but this times the values of the Seats have been mediated and so we expect the result to be closer to the true Result.
  This method also returns a class parameter **allResults** which is a dictionary with keys displayed by the name of each party and values as a list of the different values the party has achieved throughout the whole simulation. It will be very important in the graphic part We can check that in the testing module, where the algorithm was tested with 2018 italian general elections. We asked a 5%(this algorithm cannot foresee the seats assigned abroad) discrepancy between the simulated results and the actual number of seats and the test was passed.
- Let's now take a glance at the graphic part that is useful to graphically visualize our Results. It takes place through the method **Graphics** that has 1 argument set as an empty dictionary named "real" by default. When the function is called and no argument is put within ist brackets that means we just want to display the

simulated data. Conversely if we insert a dictionary as real,that dictionary should represent the real results of the election we were simultaing. That also means that we want to display both the simulated results and the real ones.
Of course we need to pay attention at how the real dictionary is built up. We have to make sure that it has the same number of parties and that the names of the parties are the same, otherwise a ValueError will be raised (that was checked in the pytest routine).

- **Graphics**() starts with the execution of **Complete_Simulation()** and retunrs an histogram with just the simulated data or with both simulated data and real data depending on the value of the argument "real".
The method also provides a second histrogram by exploiting the parameter **allResults** I have described above. So this second histogram will be a collection of the results for the different parties all over the simulations. This graph can be useful to evaluate the oscillations of the parties and to evaluate all possible oscillations during a general election.
This method ends with the histograms being salved in folder where the programs code are currently located.

Main

- The procedure we follow to execute the program is rather simple. We open the file ***Main_electoral.py*** and we start initialating an object of the class Montecarlo_electoral. If we choose an outer input, excel or txt we need to pay attention at the existence of such a file.

- Then we execute the method **Import_Results()** to harvest the data we need from the input file

- We then execute the method **chech_input** to check wether the input is good or not. If it is, it can be used for the algorithm. On the other hand if the input is not so good several ValueErrors may be raised, depending on the input mistake we have done and the method will return False.

- Now we can decide to simply display the results via standard output by simply calling the function **Complete_Simulation** or through graphic display by means of the method **Graphics()**. In the example in ***Main_electoral.py*** I wanted to display the 2018 italian general elections and so I used the 2018 italian general elections result as an argument for **Graphics()**.

- The program will end with the graph saved and the simulation completed.