1. **Card separation**

This part will be dedicated to setting up our dataset, which we will start by separating each of the ‘*train-XXX*’ images into several images representing a card. And the result obtained will be the basis on which the following parts will be built.

To be able to automate the image separation process and the assignment of identifiers to the results, we have used the Pillow library (*Pillow is a fork of the Python Imaging Library (PIL). PIL is a library that offers several standard procedures for manipulating images, it performs various operations on images such as cropping, resizing, adding text to images, rotating, greyscaling, etc*) to be able to implement the function illustrated in figure 1.

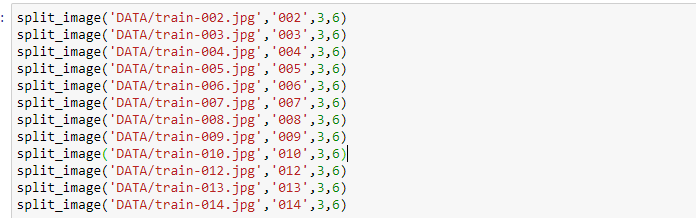


**Figure 1 :** Card separation function.

Our function takes as parameters the source image that we want to fragment, the first three characters of the identifier (the rest will be calculated), and two numbers designating the number of fragments that we want to obtain.

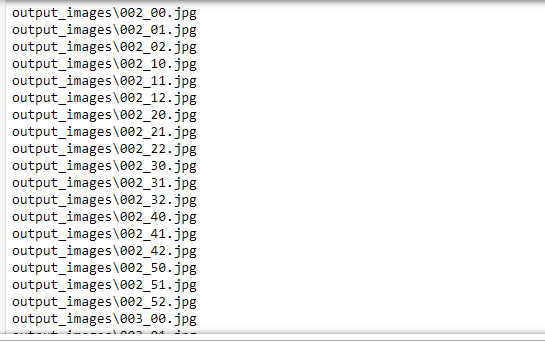
The function begins by retrieving the length and width of the source image, and calculating the length and width of each fragment according to the dimensions of the main image and the number of rows/columns passed as parameters. Then using loops it separates the image into ‘*w\_num’* lines, each of these lines will be fragmented into ‘*h\_num’* columns (in order to obtain elementary results). Finally it associates an identifier to each element, it displays the path where it will store it, and it stores it.

The figure below illustrates the call to the splitting function on each of the ‘*train-XXX*’ images.



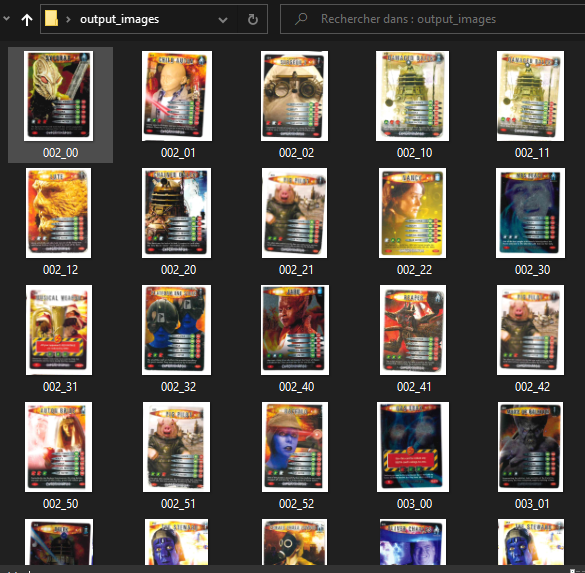
**Figure 2 :** Use of the fragmentation function.

The figure 3 illustrates the result of executing the function at the jupyter notebook editor



**Figure 3 :** The result of the fragmentation function on jupyter notebook.

As mentioned before, the function stores the results in file. For this storage, Figure 4 illustrates some of the results in the output file.



**Figure 4 :** The result of the fragmentation function.

1. **Image matching**

This part will be dedicated to the implementation of a function that will allow us to compare the images based on a set of parameters, in addition to the implementation of the matching feature this part will allow us to test based on the dataset that we have prepared in the previous part, and visualize the different results obtained.

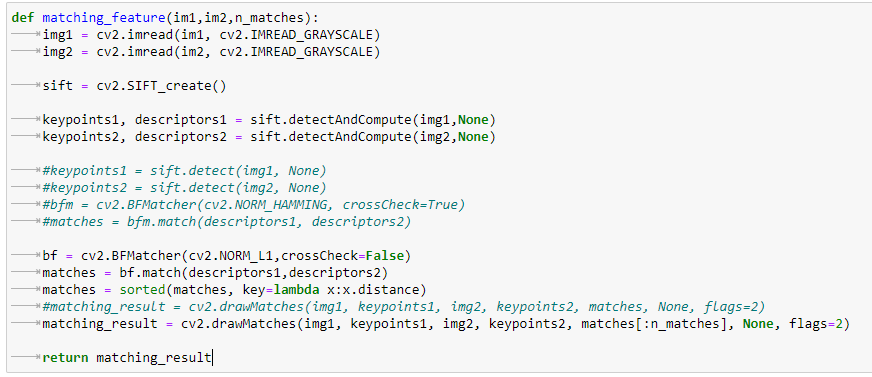
For the implementation part, and as illustrated in the figure below, we must of course start by importing the prerequisites necessary for the operation of our code, these prerequisites are opencv (*Open Computer Vision is a graphics library. it specializes in image processing, whether for photography or video*) and numpy (*a numeric library providing efficient support for large multidimensional arrays, and high-level mathematical routines such as special functions, linear algebra, statistics*).



**Figure 5** : Import of prerequisites.

The function starts by reading the images given in parameters, then it creates an instance of the 'SIFT' object, to be able to apply the detectAndCompute() function to the images, the detect() part will allow us to find the keypoint of the image (the keypoint is a special structure which has many attributes like its (x,y) coordinates, size of the meaningful neighbourhood, angle which specifies its orientation). And the Compute() part computes the descriptors from the keypoints.

Then the function applies a Brute-Force Matcher to the descriptors obtained previously (it takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned), after getting all the matches, the function sorts them based on the distance and get only the first ‘n\_maches’ ones and it draws them using ‘drawMatches’ that draws the small circles on the locations of keypoints.



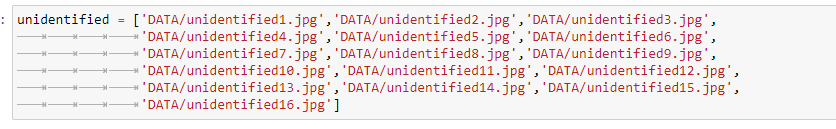
**Figure 6 :** Matching feature code.

To be able to automatically compare all the cards with all the fragments of the train-001 image, we started by gathering them in two different lists, in order to be able to loop them.

Figure 6 and 7 illustrate this process.



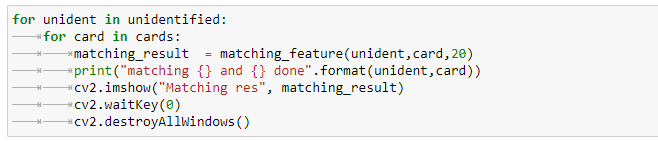
**Figure 7 :** preparing input data (cards).



**Figure 8 :** Preparing input data (train-001).

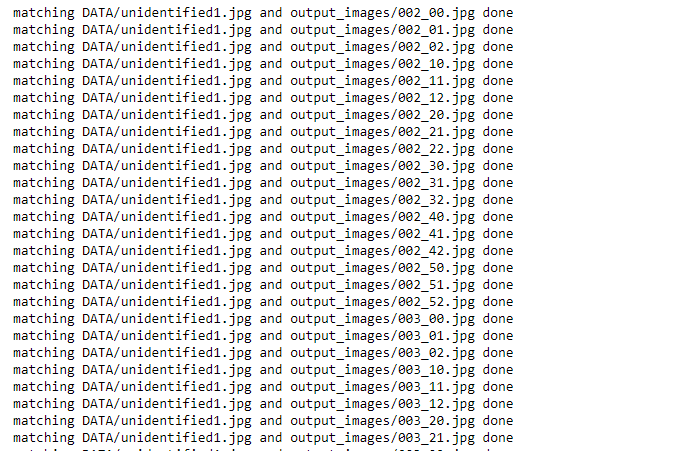
Finally, we are going to test our matching feature, and to do this we perform loops on the two lists built previously, in order to be able to compare each fragment of ‘*train-001*’ with all the cards. Inside the loop we call our matching feature, we display a message to visualize the progress of the execution, and finally we display the matching result using the *imshow()* function of *opencv*.

Figure 9 shows the process followed in detail



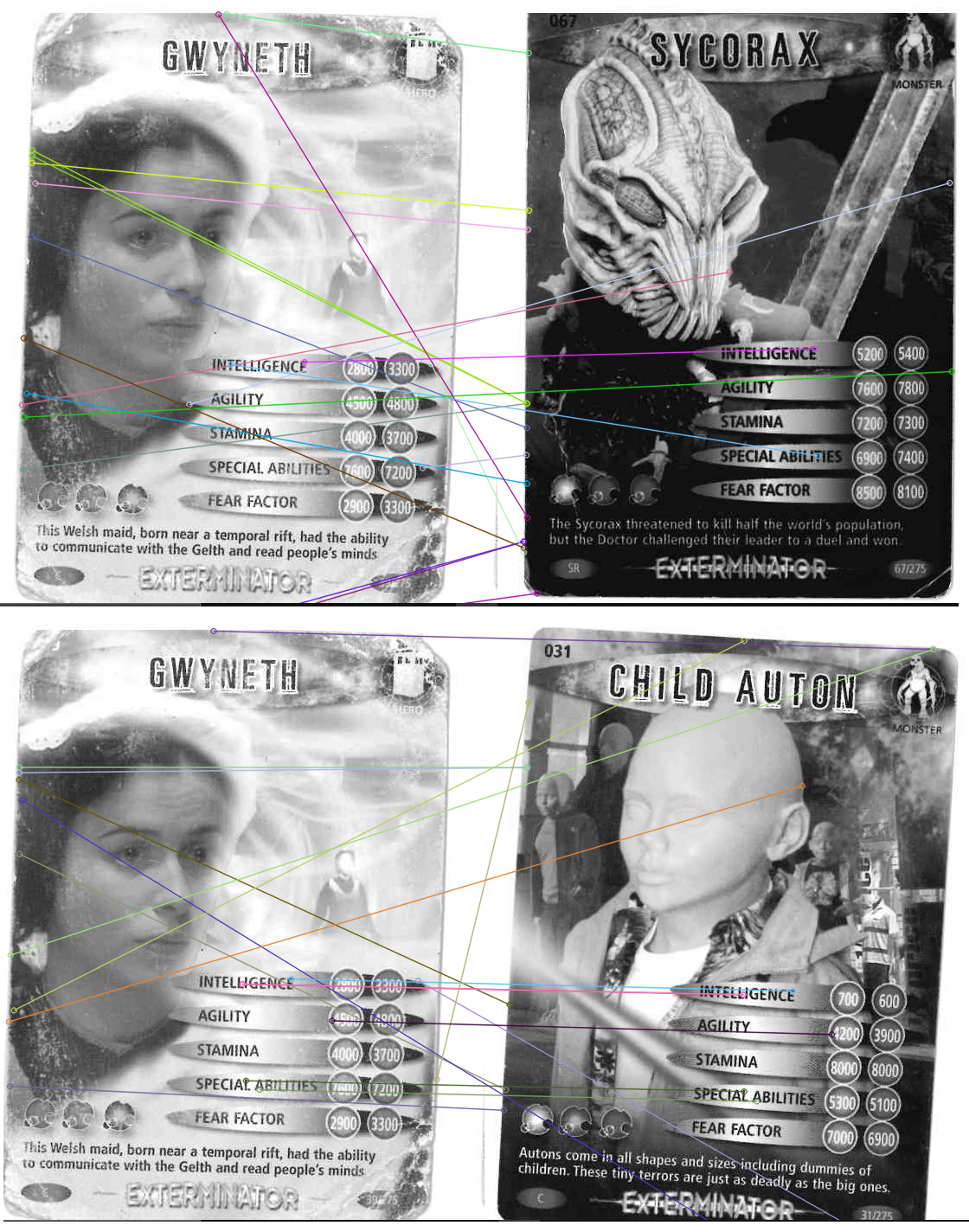
**Figure 9 :** The execution of the matching feature.

The figure below illustrates the result of the execution of our code on jupyter notebook, and as mentioned previously, a message which shows the progress of the process is displayed (the matching of which image with which image).

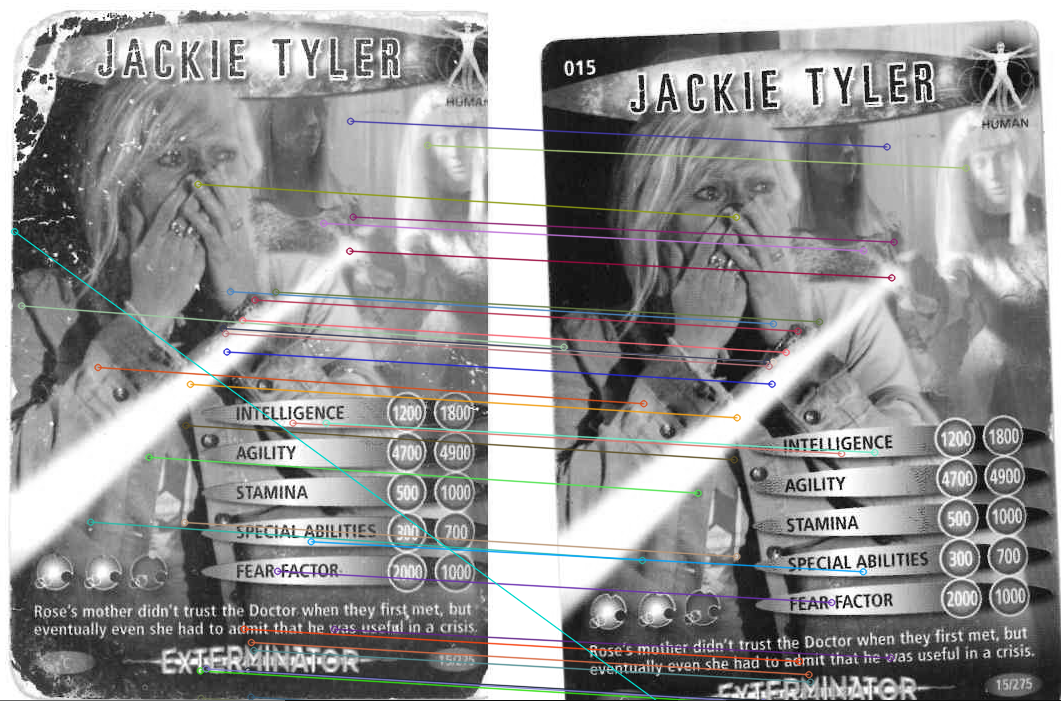


**Figure 10 :** The output of matching feature on jupyter notebook.

Finally for the concrete results of our matching feature, figure 11 shows the result on images that do not have many properties in common, and figure 12 shows the result on two images that are identical.



**Figure 11** : The matching results part 1.



**Figure 12 :** The matching results part 2.