



# Python project report

---

Text or Graphic editor for finite state automata

A project by

*Lachheb Eya*

*De Moraes Théo*

*Thumereau Bryan*

Supervised by *Zaouche Djaouida*



Github link for the project : <https://github.com/bryanthmr/projet-automatisation>

# Summary

## *I - Project presentation*

## *II - Instructions manual*

- 1) *Program's execution*
- 2) *Text editor operations*

## *III - Code structure*

- 1) *AEF.py*
- 2) *menu.py*
- 3) *main.py*
- 4) *affichage2.py*
- 5) *Directory structures*

## *IV - Tasks distribution and organization*

- 1) *Team members*
- 2) *Tasks distribution*
- 3) *Choice of collaboration tools*
- 4) *Meetings*
- 5) *Difficulties encountered*

# I - Project presentation

The aim of our project is to create a text editor in Python dedicated to Finite State Automata (FSA), providing a user-friendly interface for manipulating, modifying and analyzing automata. The aim is to simplify FSA management while offering advanced features for exploring and optimizing these structures. Here's an overview of the features incremented within our editor :

- 1) Generate an FSA by entering information in the editor
- 2) Integrated import/export functions for loading and saving automata from files
- 3) Structural modification of an FSA by adding or removing states, transitions and alphabet symbols
- 4) Deletion to remove an FSA from the interface
- 5) Word recognition to determine whether a word is recognized by a specific FSA by testing it against the automaton
- 6) The verification :
  - Completeness, which determines whether an automaton is complete
  - Determinism to check whether an automaton is deterministic
  - Equivalence, to check whether two automata are equivalent
- 7) The automaton transformation which allows :
  - Render a complete automaton by adding missing transitions
  - Make an automaton deterministic by giving each state a unique transition for each symbol in its alphabet
- 8) Obtaining a new FSA with its :
  - Complement
  - Mirror
  - Product with another automaton
  - Concatenation with another automaton
- 9) Obtaining :
  - A regular expression from an automaton
  - The language recognized by the automaton

## II – Instructions manual

To help you understand and make the best use of the automaton editing and management system set up as part of this IT project, here are the instructions to follow :

### 1) Program's execution

In order to launch our text editor program correctly, we need at least version 3.10 Python and we have to execute the file named « main.py ». To do this, we must be in the project folder named « projet-automatisation », write in the terminal « python main.py », if you're in the windows subsystem, or write « python3 main.py » for the Linux subsystem. After this, press enter.

Windows :

```
PS H:\Documents\Python\projet-automatisation> python main.py
```

Linux :

```
PS H:\Documents\Python\projet-automatisation> python3 main.py
```

The main menu guides you through the various functions available. To get the most out of your program, be sure to follow the instructions on the screen :

```
=====
                        Select an option :
1. ~Creation
2. ~Modification
3. ~Deletion
4. ~Verification
5. ~Improvement
6. ~Operation
7. ~Display
8. ~Unit tests
8. ~Exit
=====
Select a number >>
```

## 2) Text editor operations

### 2.1 Automata creation and management

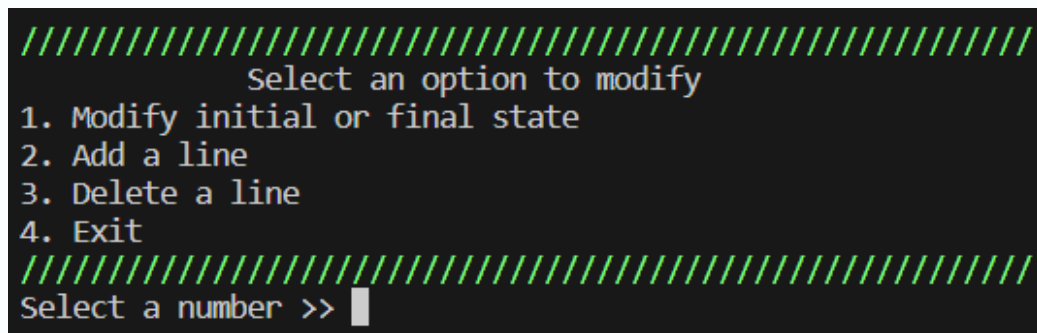
The « AEF.py » module includes functions for creating, modifying, deleting and managing automata. The use of these functions is explained below :

#### 2.1.1 Automaton creation

To create an automaton, run the « main.py » file (see program's execution section) and select the « Create automaton » option from the main by entering the number 1 and pressing enter. Follow the on-screen instructions to specify the CSV file name, initial state, final state and transitions of the automaton. The data will be saved in the « csv » directory.

#### 2.1.2 Automaton modification

To modify an existing automaton, select « Modification » from the main menu by entering the number 2 and pressing enter. You will be prompted to select the CSV file of the automaton to be modified. You can then choose from the following options :



```
////////////////////////////////////  
                          Select an option to modify  
1. Modify initial or final state  
2. Add a line  
3. Delete a line  
4. Exit  
////////////////////////////////////  
Select a number >> |
```

#### 2.1.3 Automaton deletion

The « Deletion » option in the main menu allows you to delete an existing automaton by entering the number 3. You will be prompted to select the CSV file of the automaton to be deleted. Confirmation will be requested before the file is actually deleted.

### 2.2 Automaton checks and improvements

The program also features functions for checking and upgrading existing automata. These options can be accessed from the main menu from “Verification” by entering the number 4.

### 2.2.1 Checks

- Word recognition : You can check whether a word is recognized by the automaton by selecting the corresponding option in the check menu
- Automaton completeness : Check whether the automaton is complete by selecting the appropriate option and specifying the automaton's CSV file
- Automaton determinism : Check whether the automaton is deterministic by selecting the appropriate option and specifying the automaton's CSV file.

```
-----  
                        Select an option :  
1. If a word is recognized  
2. If an automaton is complete  
3. If an automaton is deterministic  
4. If all cycles are unitary  
5. If two automata are equivalent  
6. Return  
-----  
Select a number >> 
```

### 2.2.2 Improvements

The « Improvement » option allows you to :

- Make an automaton complete by specifying the CSV file and following the on-screen instructions
- Make an automaton deterministic by specifying the CSV file and following the on-screen instructions
- Make a pruned automaton by specifying the CSV file and following the on-screen instructions (**not functional**)
- Make a minimal automaton by specifying the CSV file and following the on-screen instructions (**not functional**)

```
*****  
                        Select an option :  
1. Make a complet automaton  
2. Make a deterministic automaton  
3. Make a pruned automaton  
4. Make a minimal automaton  
5. Return  
*****  
Faites votre choix>> 
```

## III – Code structure

The project is developed in Python, using advanced features such as classes, CSV files, conditional structures and loops. In addition, our project's source code is divided into several files, each responsible for a specific part of the system.

### 1) « AEF.py »

The « AEF.py » file contains the definition of the « Automate » class and the functions associated with finite automaton manipulation.

- « Automate » class : This class represents a finite automaton. It has attributes such as « Initial », « Final », et « transition » to store the initial state, final state and transitions respectively. The methods of this class are used to import an automaton from a CSV file, create a new automaton, delete automata, modify automata, and perform various checks.
- Utility functions : the file also contains utility functions, such as « fichierExistence() » which checks the existence of CSV files.

### 2) « menu.py »

The « menu.py » file manages the application's interactive menus.

- « menu.py » function: This function defines the main menu, which guides the user through the program's various functions.
- « verif() » function: A submenu for automaton verifications, such as word recognition, completeness and determinism.
- « operation() » function : A submenu for possible automaton operations, such as product or concatenation between two automata, creation of the automaton's complement or mirror, or regular expression of an automaton.
- « improve() » function: A submenu for possible automaton improvements, such as completion, determinism, pruning and minimization.
- Unit tests : We ensured the reliability of our code by implementing unit tests throughout the development process. This approach enabled any errors or failures to be detected quickly, thereby promoting high code quality. However, these unit tests do not cover all the functionalities, such as the creation/modification of the CSV content of an automaton.

### 3) « main.py »

The « main.py » file is the program's main entry point.

- Automaton initialization : The code creates an instance of the « Automate » class to handle automata.
- Main menu : The program uses the main menu defined in « menu.py » to guide the user through the program's various functions.
- Program execution : Program execution begins by calling the « menu() » function.

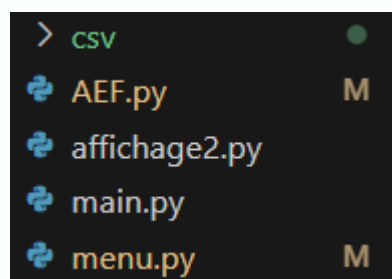
### 4) « affichage2.py »

The « affichage2.py » file allows you to graphically display the automata you have created.

### 5) Directory structures

The project follows a simple directory structure :

- « csv/ » : This directory stores the CSV files which representing the automata. Furthermore, as the « csv/test.csv » file is required for unit testing, ***it should not be deleted.***



Finally, the structure of the code makes it easy to maintain the system. Files are organized in a logical way, allowing you to quickly understand each component of the project. The modularity of the code also makes it easy to add new functionalities.



## IV – Tasks distribution and organization

In this section, we will describe how tasks were divided between team members and how the organization was set up to carry out the project. The aim is to provide an overview of team collaboration.

### Team members

Our team is made up of three members, each bringing key skills and playing specific roles in our project :

#### 1) Eya

*Role :*

Données(), estDéterministe(), deterministe(), \_\_mul\_\_(), modifier(), modifierligne(), modifier\_etat(), estComplet(), affichage(), menu(), mot(), verifier\_mot()

#### 2) Bryan

*Role :*

arden(), estDeterministe(), regex(), init\_equation(), factorisation(), \_\_init\_\_(), importCSV(), miroir(), complement(), \_\_add\_\_(), equivalence()

#### 3) Théo

*Role :*

supprimer(), estComplet(), complet(), numero\_puit(), fichierExistence(), create(), ajout\_ligne(), supprimer\_ligne()

Working together, we combine our skills to create the most efficient and user-friendly Python text editor possible.

## Tasks distribution

The division of tasks within our team was collaborative and adaptive, taking into account our respective skills and availability.

At the outset, we discussed the different parts of the project, such as data management, the user interface, and so on. Taking into account everyone's preferences and knowledge, we identified the areas where each member felt most comfortable. For example, if someone already had experience in managing files to store automata, that person was encouraged to work on that subject. This enabled us to gradually create our functions, correct them and then make them available to the other team members so that we could move forward efficiently.

Secondly, the distribution of tasks was balanced to adapt to everyone's availability, thus avoiding an excessive workload for any one member.

Finally, to ensure that the project progressed smoothly, we set intermediate milestones and realistic deadlines for each part of the work. This enabled us to coordinate our efforts and ensure that the project progressed smoothly.

This approach fostered smooth collaboration within our team, with everyone making a significant contribution while developing their skills within the framework of the project.

## Choice of collaboration tools

To facilitate remote communication and collaboration, the choice of tools was a crucial step. We opted for Discord, an instant messaging application offering text, video and voice capabilities, which was familiar to all team members.

Complementing this, the use of Github, a code-hosting platform, was instrumental in ensuring efficient tracking of code and individual tasks, reinforcing collaboration within the team.

When it came to programming our text editor in Python, we opted to use Visual Studio Code, a powerful code editor mastered by the whole team. This decision has helped to harmonize our development environment and ensure consistency in our collective work.

## Meetings

To maintain constant collaboration within the team, we set up weekly meetings on Discord. These sessions served as synchronization points, enabling us to track the progress of each group member. We paid particular attention to the obstacles encountered and the solutions envisaged. Each meeting was also an opportunity to take stock of our current position. In concrete terms, this meant a weekly review of tasks completed, those still to be done, and any bugs in the code to be resolved. We also regularly encouraged ideas for improving the existing code. Finally, all code integrations were carried out collectively during these meetings.

## Difficulties encountered

As a team of just three members, we had to cope with a heavier workload than other groups, which had a minimum of four people.

Despite our determination and cooperation, this situation sometimes meant that time and resources had to be managed more rigorously. We had to be strategic in allocating tasks and ensure that each member of the team contributed optimally to the project.

This challenge also highlighted our ability to manage priorities effectively and maximise our limited resources.

Ultimately, although the small size of our team presented challenges, it also strengthened our ability to work agilely and make quick decisions to ensure the success of our project.