Project Report: Senegalese License Plate Detection System

Abstract

This project builds a simple Python program that finds Senegalese vehicle license plate numbers that appear inside any piece of text. It works without using external libraries. It supports two valid formats, accepts both hyphens and single spaces as separators, ignores letter case, and prints all plates it finds in a clean, standard form. We also add a second version that uses a regular expression (regex) to do the same job in fewer lines. The goal is clarity, correctness, and ease of understanding.

1. Introduction

We often need to pull structured items (like license plates) out of unstructured text (paragraphs, notes, reports). This project shows how to do that for Senegalese license plates using two approaches:

1. A manual step-by-step parser (no regex)
2. A regex-based version (shorter, but less explicit)

2. What Counts as a Valid Plate

Accepted shapes (letters = A–Z, digits = 0–9):

- XY-1234-T
- XY-1234-ZT

You may also write them with spaces instead of hyphens: `XY 1234 T` or `XY 1234 ZT`.

Rules:

- First two characters: letters
- Separator: hyphen (-) or single space
- Four digits
- Separator: hyphen (-) or single space
- Last part: 1 or 2 letters
- No extra letters/digits glued directly to the left or right

- Case-insensitive input; output always uppercase with hyphens

Examples that SHOULD match:

```
ab-1234-e
ab 1234 e
mn-4321-zt
uv 9876 t
```

Examples that SHOULD NOT match:

```
AAB-1234-T   (three starting letters)
AB-12345-T   (five digits)
AB_1234_T    (underscores are not allowed)
AB-1234-TXQ  (last part too long)
```

3. Main Goals

- Detect any valid plate in a text
- Show each unique plate once, in order of appearance
- Print a helpful message if none are found
- Make the code easy to read and extend
- Provide an alternative regex version for comparison

4. Approach 1: Manual Parsing (No Regex)

4.1. Logical Idea for Manual Parsing

We walk through the text one character at a time. At each position we check if the next characters could form a valid plate. If not, we move one step forward. If yes, we store it (in a standard format) and then jump ahead past it.

4.2. Steps

1. Convert the whole text to uppercase (so we only compare one case).
2. Check two letters.
3. Check a separator (hyphen or space).
4. Check four digits.
5. Check a separator.

6. Collect one or two ending letters.
7. Make sure there is a boundary (start/end of text OR a non-letter/digit) before and after.
8. Normalize to the form: `XY-1234-T` or `XY-1234-ZT`.
9. Avoid duplicates.

4.3. Strengths / Weaknesses

Strengths: Transparent, easy to modify, no hidden behavior.
Weaknesses: More lines of code than a regex.

5. Approach 2: Regex Version

The regex puts the same rules into one pattern. It is shorter and fast. We still normalize output the same way (always with hyphens and uppercase). Lookbehind and lookahead prevent partial matches inside larger words.

5.1. Explanation of the Regex Pattern

- `(?i)` ignore case
- `(?<![A-Z0-9])` left side is not a letter/digit
- `([A-Z]2)` two letters
- `[- ]` separator
- `(\d{4})` four digits
- `[- ]` separator
- `([A-Z]{1,2})` one or two letters
- `(?![A-Z0-9])` right side is not a letter/digit

6. Boundary Condition Management

A critical aspect of the implementation involves robust boundary condition checking to prevent false positives. The algorithm examines the characters immediately preceding and following potential plate matches to ensure they are not alphanumeric. This boundary validation ensures that the system doesn't mistakenly identify substrings within larger words or alphanumeric codes as license plates.

The boundary checking mechanism employs logical conditions that consider both the position within the text string and the alphanumeric properties of adjacent characters. This sophisticated approach handles edge cases such as plates appearing at the beginning or end of text inputs, as well as plates surrounded by punctuation or whitespace.

7. Pattern Normalization and Storage

Upon successful identification and validation of a license plate, the program performs normalization to standardize the output format. Regardless of the original separator type (hyphen or space) found in the input text, all detected plates are converted to the canonical hyphen-separated uppercase format. This normalization ensures consistency in output presentation and facilitates subsequent processing or comparison operations.

The storage system maintains detected plates in an ordered list that preserves the sequence of appearance in the original text. Duplicate elimination is implemented through membership checking before list insertion, ensuring that each unique plate appears only once in the final output while maintaining the first occurrence position in the ordering.

8. Added Comparison Helper

We included a helper function `compare_detectors()` so both methods can be run on the same sample text to show they agree.

9. Testing

We built a test paragraph mixing valid plates, invalid near-misses, different separators, different ending lengths, and case variations. Both methods returned the same valid list.

10. Results (In Simple Terms)

- All expected plates were detected.
- No incorrect matches were printed in testing examples.
- Output is consistent and clean.

11. Limitations

- No fuzzy matching (typos or OCR mistakes not handled).
- Only supports the two Senegal patterns described.
- Does not check whether the plate is officially issued, just the pattern.

12. Conclusion

We now have two clear ways to find Senegalese license plates inside any text: a manual parser and a regex version. Both meet the goals of correctness, clarity, and simple output. The manual version is ideal for learning and future tweaks. The regex version is concise and practical.