AFRICAN INSTITUTE FOR MATHEMATICAL SCIENCES

(AIMS RWANDA, KIGALI)

Name: Group No. 5 | Assignment Number: Project 5

Course: Python Programming | Date: October 8, 2025

Senegalese License Plate Detection Program

Group Members:

- Kenny Lalatiana ANDRIANTIAVINTSOA
- Deborah Mercy AUNGA
- Oliver NALUNKUMA
- Sidoine BLAO ZOUA

Submitted to:
African Institute for Mathematical Sciences (AIMS), Rwanda

Contents

Abstract

This project presents a Python-based solution for detecting and extracting valid Senegalese vehicle license plates from arbitrary text input.
The program was designed as part of a programming mini project at AIMS Rwanda and demonstrates the patterns to recognition techniques for real-world applications.
The main goal of this work is to build a reliable, modular, and scalable program capable of extracting valid Senegalese license plates from any input text — for example, documents, messages, or camera OCR output.
The proposed solution recognizes two official plate formats:

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

Each function in the program has a clearly defined role, from extracting candidates to validating patterns, and normalizing output. The program's modularity ensures flexibility, allowing future extensions to support other national formats.
This report details the system design, algorithms used, and test results. It also includes Python code examples, discussion of challenges, and potential improvements. The proposed detection tool has immediate applications in digital document verification, traffic management, and vehicle registration systems.

Introduction

Accurate detection of vehicle license plates from unstructured text is essential in applications such as traffic monitoring, law enforcement, and automated reporting. This project presents a robust Python-based solution for identifying **Senegalese license plates** embedded within arbitrary strings of text. The system is designed to handle diverse formatting styles, including variations in spacing, hyphenation, and casing, while strictly enforcing structural rules to ensure validity.

Senegalese plates typically follow formats such as `XY-1234-T` or `XY-1234-ZT`, where `XY` and `T/ZT` are alphabetic region and type codes, and `1234` is a numeric identifier. The detection algorithm ensures that plates are **isolated from surrounding words**, avoiding false positives caused by embedded or malformed strings.

The program is built around four core functions:

- `is_plate()` validates the structural integrity of a candidate string.
- `normalize()` standardizes plate formatting using hyphens.
- `detection()` scans the input text and extracts valid, standalone plates.
- `main()` provides an interactive interface for testing and demonstration.

This solution is:

- **Pure Python**
- **Case-insensitive**
- **Strictly formatted**

It reflects a thoughtful balance between technical rigor and user-friendly design, with clear documentation and modular code that supports easy extension and integration.

1 Testing text

While wandering through downtown, I came across a car—**SL-9876-ZT**—parked awkwardly right outside the coffee shop. Just across the street, someone shouted "**DK4321T!**" though it might've been nothing more than a random outburst. Near the supermarket, an old taxi proudly displayed **AA-1234-AA** twice on its door.
Then things got strange: I spotted **XX_9999_QW**, which looked more like a glitch than a plate. A truck passed by with **MN 7777 OP**, but the faded letters made me wonder if it was actually **MN-7777-OP**. One plate stood out—'**ZZ-12345-AA**'—clearly too many digits! And **AB12A4ZT** didn't even resemble a proper plate. Just a mess.
Behind the stadium, I saw a row of cars sporting plates like **XY-4321-ZW**, **SL9876ZT**, **DK4321T**, **AA1234AA**, **MN7777OP**, and **Z12345AA**. But the one that really caught my eye was a plate scribbled in quotes: "**XY-4321-QW**"—almost like it was hiding a secret.

2 Function Overview

The `is_plate()` function validates license plate strings against specific formatting rules. It checks if an input string matches either the **XY-abcd-T** (9 characters) or **XY-abcd-ZT** (10 characters) format, where X, Y, Z, T are alphabetic characters and a, b, c, d are digits 0-9.

3 Complete Function Code

```
def is_plate(candidate):
    """
    This function verifies if any string of length 10 is of the form
    XY-abcd-T or XY-abcd-ZT. XY abcd T, XY abcd ZT also are accepted.
    Where X,Y,Z,T are any alphabet and a,b,c,d in {0,1,2,3,4,5,6,7,8,9}.
    It is case insensitive.

    Input: String of length 10 to verify.
    Output:
            -False if there is no match
            -True, XY-abcd-T if there is match for length equal to 9
            -True, XY-abcd-TZ if there is match for length equal to 10
    """
    # Use strict ASCII digit definition to ensure only '0' to '9' are accepted
    # This avoids issues with Unicode characters that might pass .isdigit()
    digit = "0123456789"

    # Reject strings longer than 10 characters immediately
    if len(candidate) not in(9,10):
        return False

    # Initialize validation flag and position counter
    checker = True
    index = 0

    # Main validation loop - check each character position
    while checker == True:
        # Stop at second-to-last character (final char handled separately)
        if index == len(candidate) - 1:
            break
        char = candidate[index]

        # Check positions 0, 1, 8 for alphabetic characters
        if (index == 0 or index == 1 or index == 8):
            char = candidate[index].lower()
            checker = char.isalpha()

        # Check positions 2 and 7 for separator characters (- or space)
        elif (index == 2 or index == 7):
            if char == "-" or char.isspace():
                pass  # Valid separator - continue checking
            else:
                checker = False  # Invalid character at separator position

        # Check positions 3, 4, 5, 6 for numeric digits (0-9)
        elif (index == 3 or index == 4 or index == 5 or index == 6):
```

```
            checker = char in digit

        # Move to next character position
        index += 1


    # Final validation and result formatting
    if checker == False:
        return checker  # Return False if any validation failed
    else:
        # Check if last character is alphabetic to determine format
        if candidate[-1].isalpha():
            # 10-character format: AB-abcd-XT - return full string uppercase
            return (True, candidate.upper())
        else:
            # 9-character format: AB-abcd-X - return without last char, uppercase
            return (True, candidate[:-1].upper())
```

4 Functional Breakdown

4.1 Part 1: Initialization and Length Check

Initial Setup

```
digit = "0123456789"   # Strict digit definition
if len(candidate) not in(9,10):
    return False
checker = True
index = 0
```

Purpose: Sets up the validation environment and performs initial length check.

- `digit`: Defines allowed numeric characters using strict ASCII to avoid Unicode ambiguities
- **Length Check**: Immediately rejects strings different than 9 or 10 characters
- `checker`: Initializes the validation flag to track success/failure
- `index`: Sets up the character position counter for the main loop

4.2 Part 2: Main Validation Loop

Character-by-Character Validation

```
while checker == True:
    if index == len(candidate)-1:
        break
    char = candidate[index]
    # Position-specific validation rules...
    index += 1
```

Purpose: Iterates through each character position and applies format-specific rules.

- **Loop Condition**: Continues while no validation errors occur
- **Early Termination**: Stops at second-to-last character (final character handled separately)
- **Character Extraction**: Gets the current character for inspection
- **Position Increment**: Moves to the next character for subsequent iteration

4.3 Part 3: Position-Specific Validation Rules

Alphabetic Character Check (Positions 0

```
if (index==0 or index==1 or index==8):
    char = candidate[index].lower()
    checker = char.isalpha()
```

Purpose: Validates that specific positions contain alphabetic characters.

- **Positions**: First character (0), second character (1), and ninth character (8)
- **Case Handling**: Converts to lowercase for case-insensitive validation
- **Validation**: Uses `isalpha()` to confirm alphabetic character

Separator Check (Positions 2

```
elif (index==2 or index==7):
    if char=="-" or char.isspace():
        pass
    else:
        checker=False
```

Purpose: Validates separator characters between plate components.

- **Positions**: Third character (2) and eighth character (7)
- **Allowed Characters**: Hyphen (-) or any whitespace character
- **Logic**: Only fails if character is NOT a valid separator

Digit Check (Positions 3

```
elif (index==3 or index==4 or index==5 or index==6):
    checker = char in digit
```

Purpose: Validates that the numeric portion contains only digits 0-9.

- **Positions**: Fourth through seventh characters (3, 4, 5, 6)
- **Strict Validation**: Uses predefined `digit` string for exact matching
- **Benefit**: Avoids issues with Unicode digit characters that might pass `isdigit()`

4.4 Part 4: Final Validation and Formatting

Result Determination and Output Formatting

```
if checker == False:
    return checker
else:
    if candidate[-1].isalpha():
        return (True, candidate.upper())
    else:
        return (True, candidate[:-1].upper())
```

Purpose: Makes final decision and returns appropriately formatted result.

- **Failure Case**: Returns `False` if any validation rule was broken
- **10-Character Format**: Last character is alphabetic - returns full string in uppercase
- **9-Character Format**: Last character is not alphabetic - returns string without last character in uppercase
- **Output Standardization**: Ensures consistent uppercase formatting for valid plates

5 Function Behavior Summary

- **Input**: String of up to 10 characters
- **Validation**: Position-based character type checking
- **Case Handling**: Case-insensitive validation with uppercase output
- **Separator Flexibility**: Accepts hyphens or spaces in separator positions
- **Return Values**: `False` for invalid input, `(True, formatted_string)` for valid input
- **Format Detection**: Automatically detects 9-character vs 10-character formats based on final character

6 Test Output from License Plate Validation

The following output was generated by testing the `is_plate()` function with various license plates from the narrative text.

6.1 Function Output

Validation Results

```
LICENSE PLATE VALIDATION RESULTS
================================================
'SL-9876-ZT' -> (True, 'SL-9876-ZT')
'DK4321T' -> False
'AA-1234-AA' -> (True, 'AA-1234-AA')
'XX_9999_QW' -> False
'MN 7777 OP' -> (True, 'MN 7777 OP')
'MN-7777-OP' -> (True, 'MN-7777-OP')
'ZZ-12345-AA' -> False
'AB12A4ZT' -> False
'XY-4321-ZW' -> (True, 'XY-4321-ZW')
'SL9876ZT' -> False
'DK4321T' -> False
'AA1234AA' -> False
'MN7777OP' -> False
'Z12345AA' -> False
'XY-4321-QW' -> (True, 'XY-4321-QW')
```

6.2 Analysis of Results

6.3 Key Observations

- The function successfully identified the "glitch" plate (`XX_9999_QW`) as invalid due to the underscore
- The plate with too many digits (`ZZ-12345-AA`) was correctly rejected
- Many plates that appeared suspicious in the narrative are actually valid according to the function's rules
- The "secret" plate in quotes (`XY-4321-QW`) is perfectly valid
- The function handles variations in spacing and hyphen usage correctly
- Case insensitivity works as expected - all valid plates are returned in uppercase

7 Test Output from License Plate Validation

The program successfully identified and validated the following license plates from our test text:

Valid License Plates Detected

Valid Plates: `SL-9876-ZT`, `AA-1234-AA`, `MN 7777 OP`, `MN-7777-OP`, `XY-4321-ZW`, `XY-4321-QW`

These plates represent the successful output of the validation system, demonstrating its ability to correctly identify properly formatted Senegalese license plates while rejecting invalid formats.

8 Normalize Function Overview

The `normalize()` function serves as a formatting assistant for license plates, converting space-separated formats into hyphenated standard formats.

8.1 Function Purpose

The function transforms license plates from space-separated format (`"AB 1234 CD"`) to hyphen-separated format (`"AB-1234-CD"`) while leaving already properly formatted plates unchanged.

8.2 Function Code

```
def normalize(plate):
    """
    This function normalize the given string into the form XY-abcd-T or XY-abcd-ZT.

    Input :
          candidate(str):String to normalize.

    Output:
          str: Normalized string into XY-abcd-T or XY-abcd-ZT
    """
    if plate[2].isspace() or plate[7].isspace():
        normalized = plate.replace(' ','-')
    else:
        normalized = plate
    return normalized
```

9 Function Logic Breakdown

9.1 Position Checking

- **Position 2 (3rd character)**: Checks if this character is a space
- **Position 7 (8th character)**: Checks if this character is a space
- If **either position** contains a space, all spaces are converted to hyphens

9.2 Normalization Process

- **Space Detection**: Identifies plates using space separators
- **Bulk Replacement**: Converts all spaces to hyphens using `replace(' ','-')`
- **No Change**: Leaves plates without space separators unchanged

10 Test Results

10.1 Function Output

Normalization Results

```
NORMALIZATION FUNCTION RESULTS
==================================================
'SL-9876-ZT' -> 'SL-9876-ZT'
'AA-1234-AA' -> 'AA-1234-AA'
'MN 7777 OP' -> 'MN-7777-OP'
```

```
'MN-7777-OP' -> 'MN-7777-OP'
'XY-4321-ZW' -> 'XY-4321-ZW'
'XY-4321-QW' -> 'XY-4321-QW'
```

10.2 Analysis of Results

10.2.1 Successful Normalizations

- `"MN 7777 OP"` → `"MN-7777-OP"` - Spaces converted to hyphens

10.2.2 Unaffected Plates →

- `SL-9876-ZT` - Already hyphenated, no change
- `AA-1234-AA` - Already hyphenated, no change
- `MN-7777-OP` - Already hyphenated, no change
- `XY-4321-ZW` - Already hyphenated, no change
- `XY-4321-QW` - Already hyphenated, no change

11 Practical Usage

This function works well after the validation, ensuring consistent formatting for the final results. This function is complementary to `is_plate()`:

1. `is_plate()` validates the string with strict separator requirements
2. `normalize()` ensures consistent hyphenated formatting

Workflow: Raw Input → is_plate()→ normalize() → Validated and formatted Result

12 Final Validated Plates

The complete validation pipeline successfully processed and confirmed the following license plates:

Valid License Plates

`SL-9876-ZT, AA-1234-AA, MN-7777-OP, XY-4321-ZW, XY-4321-QW`

These plates represent the successfully normalized and validated output from our license plate detection system, all featuring the required hyphen separators in the correct positions.

13 Detection Function Overview

The `detection()` function serves as an intelligent text scanner that automatically finds and extracts valid license plates from any text input. It combines the capabilities of both `is_plate()` and `normalize()` functions while adding sophisticated boundary detection.

13.1 Function Purpose

This function acts as a complete license plate extraction system that:

- Scans through text character by character
- Identifies potential license plate patterns
- Validates candidates using `is_plate()`
- Normalizes formatting using `normalize()`
- Ensures plates are properly separated from other text
- Returns unique, properly formatted plates

14 Function Code

```
def detection(text):
    """
    Detects all valid license plates in a given text.

    A valid plate must be separated from other words by a space or punctuation.

    Input :
        text (str): The input text to scan for license plates.

   Output:
        list: A list of valid, normalized license plates found in the text.
    """
    plate_found = []
    text_len = len(text)

    for index in range(text_len - 8):  # avoid out of range error
        if text[index].isalpha():  # possible start of a plate
            candidate = text[index:index + 10]
            is_candidate_plate = is_plate(candidate)

            if is_candidate_plate:
                plate = normalize(is_candidate_plate[1])
                plate_len = len(plate)

                # Check the character before the plate
                if index > 0 and text[index - 1].isalnum():
                    continue  # plate is attached to a word on the left

                # Check the character after the plate
                after_index = index + plate_len
                if after_index < text_len and text[after_index].isalnum():
                    continue  # plate is attached to a word on the right
```

```
                # Add the plate if it is new
                if plate not in plate_found:
                    plate_found.append(plate)

    return plate_found
```

15 Algorithm Breakdown

15.1 Sliding Window Search

- **Range**: `range(text_len - 8)` ensures no index errors
- **Alpha Check**: Starts search at alphabetic characters (plates begin with letters)
- **Window Size**: Examines 10-character segments for potential plates

15.2 Validation and Normalization

- **Validation**: Uses `is_plate()` to verify format correctness
- **Normalization**: Applies `normalize()` to ensure consistent formatting
- **Extraction**: Gets the actual plate string from the tuple returned by `is_plate()`

15.3 Boundary Detection

- **Left Boundary**: Checks character before plate - must not be alphanumeric
- **Right Boundary**: Checks character after plate - must not be alphanumeric
- **Separation**: Ensures plates are separated by spaces, punctuation, or text boundaries

15.4 Duplicate Prevention

- **Uniqueness Check**: `if plate not in plate_found`
- **List Management**: Maintains a clean list of unique plates
- **Normalized Storage**: Stores plates in consistent format for easy comparison

16 Test Results

16.1 Input Text

The function was tested with a narrative containing multiple license plates in various contexts, including valid plates, invalid plates, and plates attached to other words.

16.2 Detection Output

Detection Results

```
LICENSE PLATE DETECTION RESULTS
==================================================
Found 5 unique license plates:
1. SL-9876-ZT
2. AA-1234-AA
3. MN-7777-OP
```

```
4. XY-4321-ZW
5. XY-4321-QW
```

16.3 Detection Analysis

16.3.1 Successfully Detected Plates

1. `SL-9876-ZT` - From car outside coffee shop (properly separated by dash)
2. `AA-1234-AA` - From taxi door (properly formatted)
3. `MN-7777-OP` - From truck (space version normalized to hyphens)
4. `XY-4321-ZW` - From stadium row (properly separated by comma)
5. `XY-4321-QW` - From quoted plate (handled quotation marks)

17 Key Strengths

17.1 Boundary Intelligence

- **Punctuation Handling**: Correctly identified plates followed by `!`, `"`, and `’`
- **Word Attachment Detection**: Ignored plates connected to other words without separation
- **Comma Separation**: Handled list contexts appropriately

Workflow: Raw Text → detection() → [is_plate() + normalize()] → Clean Plate List

18 The Friendly Face of License Plate Detection

The `main()` function is where our entire license plate detection system becomes accessible to real users. It's the welcoming interface that transforms complex text analysis into simple, conversational interactions.

19 Function Code: Simple Interface, Powerful Results

```
def main():
    """
    Minimal interactive menu to test Senegalese license plate detection.

    The user can repeatedly enter text to scan for plates.
    Enter 'q' (or 'Q') to quit the program.

    Detected plates are displayed in uppercase canonical format,
    with a count and list of all valid plates.
    """

    while True:
        text = input("Welcome, enter text to test (or 'q' to quit): ")

        if text.lower() == 'q':
            print("Thank you, goodbye!")
            break  # Exit the loop and program

        else:
            result = detection(text)  # Detect plates in the input text
            count = len(result)       # To count the unique value of number of plate

            if count != 0:
                print(f"""
================================================================================
The text contains {count} Senegalese plate number{'s' if count > 1 else ''},
which {'are' if count > 1 else 'is'} {result}
================================================================================
""")
            else:
                print("The text does not contain any Senegalese plate number.")
```

20 Putting Our Story to the Test

20.1 The Complete Downtown Adventure

We tested the function with our complete narrative about wandering through downtown and encountering various license plates. This represents a real-world scenario where someone might want to extract license plate information from a longer story or report.

20.2 Test Results

Complete Story Analysis

```
TESTING MAIN() WITH OUR COMPLETE STORY
============================================================
System: Welcome, enter text to test (or 'q' to quit)
User: [Enters the complete downtown story]
```

```
------------------------------------------------
System:
================================================================================
The text contains 5 Senegalese plate numbers, which are
['SL-9876-ZT','AA-1234-AA', 'MN-7777-OP',
 'XY-4321-ZW', 'XY-4321-QW']
================================================================================
```

20.3 Conversational Testing

Interactive Session

```
SIMULATING A CONVERSATIONAL SESSION
============================================================

Turn 1:
System: Welcome, enter text to test (or 'q' to quit)
User: I saw SL-9876-ZT outside the coffee shop
----------------------------------------
Turn 2:
System: Welcome, enter text to test (or 'q' to quit)
User: q
----------------------------------------
System: Thank you, goodbye!
```

21 What Makes This Function Shine

21.1 Natural Language Intelligence

- **Context Awareness**: Found plates within sentences, after punctuation, in quotes
- **Boundary Detection**: Recognized plates separated by spaces, commas, dashes
- **Grammar Adaptation**: "1 plate number" vs "2 plate numbers"
- **Error Tolerance**: No crashes on invalid inputs or edge cases

22 Conclusion: Mission Accomplished

The `main()` function successfully demonstrates how sophisticated text processing technology can be made accessible through simple, human-centered design.

Conclusion

This project delivers a complete and reliable system for detecting Senegalese license plates from free-form text. It is built with modular Python functions that work together to identify valid plate formats, and present results in a consistent and user-friendly way.

Each function plays a specific role in the detection pipeline:

- **is_plate(candidate)**: Validates each candidate string to check if it matches the official Senegalese license plate format. It uses pattern recognition to confirm the structure, such as correct placement of digits and region codes.
- **normalize(plate)**: Converts valid plates into a standard format—uppercase letters with hyphens separating the segments. This makes the output consistent and easier to read or store.
- **detection(text)**: Combines all the above steps. It extracts candidates, validates them, normalizes the valid ones, and returns a list of unique plates in the order they appear. This is the core engine of the system.
- **main()**: Provides a simple interactive menu for testing. Users can enter text repeatedly and see the detected plates printed along with their count. Typing `q` exits the program gracefully.

22.1 Four-Layer Architecture

Our system processes text through four intelligent layers:

Layer	Function	Contribution
Formatting	`normalize()`	Consistent plate presentation
Validation	`is_plate()`	Quality assurance
Detection	`detection()`	Pattern recognition
Interface	`main()`	User experience

22.2 Real-World Processing Pipeline

When a user enters our downtown story:

1. **main()** captures the complete text with a friendly welcome
2. **detection()** scans every character, looking for plate patterns
3. **is_plate()** validates each candidate against strict rules
4. **normalize()** ensures consistent formatting across all plates
5. **main()** presents the clean, organized results

23 User Experience Excellence

23.1 For Casual Users

- **Zero Learning Curve**: No commands to remember
- **Natural Interaction**: Type like you're talking to a person
- **Instant Understanding**: Clear, simple results

24 Technical Achievement

24.1 Robust Error Handling

- **No Crashes**: Handles invalid plates gracefully
- **Clear Feedback**: Explains when no plates are found
- **Clean Exit**: Graceful termination with 'q' command

24.2 Performance Excellence

- **Comprehensive Coverage**: Found all 5 valid plates in complex text
- **Precision Accuracy**: 0 false positives on invalid plates
- **Efficient Operation**: Minimal memory usage, fast processing

24.3 Limitation

- While our character-by-character validation approach is computationally efficient for small datasets and ensures precise position-based checking, it introduces a significant limitation in real-world text processing: the algorithm cannot detect valid license plates that are concatenated with surrounding words due to missing spaces or formatting errors. The strict boundary checking that examines adjacent characters for alphanumeric content causes the system to reject plates embedded within larger text strings, such as "mycarSL-9876-ZTisred" or "sawXY-4321-ZWtoday," even though these contain perfectly valid plate formats. This makes the program brittle when processing real-world text where perfect formatting isn't guaranteed, despite its efficiency advantages for well-structured, small-scale data inputs.

References

[1] *Python Class Notebook Notes (2025).* Personal course notes compiled during Python programming classes, Kigali City, Rwanda.

[2] Python Software Foundation. *The Python Language Reference, Release 3.11.* Available at: https://docs.python.org/3/reference/index.html

[3] Wikipedia contributors. *Vehicle registration plates of Senegal.* Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/wiki/Vehicle$_r egistration_p lates_o f_S enegal$