African Institute for Mathematical Sciences – AIMS-RW

PyPro-SCiDaS – Mini-Projects.

September 30, 2025

Project 1. `Student Registration Number`

*AIMS-Rwanda would like to automate the registration of its students by assigning each of them a **unique registration number**. The mechanism below will be used to generate this number.*

1. ***Primary Fields:*** *Each student's registration number is composed of **three fields**:*
   - ***Field 1:*** *Extract the first three consonants of the last name (in uppercase).*
   - ***Field 2:*** *Extract the first three letters of the first name (in uppercase).*
   - ***Field 3:*** *A three-digit sequential counter representing the order in which students with the same first two fields register (starting at 001).*

   If the last name does not contain three consonants, pad the missing ones with "X". If the first name has fewer than three letters, also pad with "X". The sequential counter must always be exactly three digits long (e.g., 001, 002, . . . , 999).
2. ***Uniqueness Requirement:*** *Two students with the same first three consonants of last name and first name initials must still get unique numbers. For instance:*
   - *Gnansounou Marcel* ⇒ `GNNMAR001`
   - *Gnonas Marcellin (registered later)* ⇒ `GNNMAR002`
3. ***Additional Requirements (Complexity):***
   - ***Case-insensitivity:*** *The program must work regardless of whether the input is lowercase, uppercase, or mixed case.*
   - ***Accents & special characters:*** *Normalize accented characters (é → E, ç → C, ü → U, etc.).*

- ***Hyphenated and compound names:*** *Ignore spaces and hyphens when extracting consonants or letters (e.g., "Jean-Pierre" or "Van der Merwe").*
- ***Persistent Tracking:*** *Simulate that your program remembers how many students have already been registered with the same prefix so that subsequent calls generate the next available number (using an in-memory dictionary).*
- ***Validation:*** *Ensure that names contain only alphabetic characters, spaces, or hyphens; reject invalid inputs (numbers, symbols) with an error message.*

Your Task: *Write a* ***pure Python program*** *(no external libraries) that:*

1. *Accepts as input: last name and first name of a student.*
2. *Generates the student's registration number according to the rules above.*
3. *Keeps track of already assigned numbers during execution so that new students receive the correct sequence number.*
4. *Prints the final registration number.*

You might want to run a test using the AIMS-RW 2025 class list.

==========

Project 2. `Check a Password`

In this exercise you will write a program that determines whether a password is ***strong enough*** *according to several security rules.*

1. *A good password must:*
   - *Be at least 8 characters long and at most 64 characters.*
   - *Contain at least one uppercase letter, one lowercase letter, one digit, and one special character (special characters are any of:* `!@#$%^&*()-_+=[]{};:'",<.>/?\|‘` .
   - *Not contain any spaces.*
2. *For additional complexity:*
   - *Reject passwords that are entirely alphabetic or entirely numeric.*
   - *Reject passwords that contain 3 or more identical consecutive characters (e.g.,* `aaa`*,* `111`*).*
   - *Reject passwords that are among a small list of commonly used insecure passwords such as:* `"password", "123456", "qwerty", "letmein"`*.*
3. *Your function should return* `True` *if the password is good and* `False` *otherwise.*
4. *Include a main program that:*
   a) *Reads a password from the user.*
   b) *Checks it against all the above rules.*

c) *Prints a clear message indicating whether the password is valid or which rules it failed.*

5. *The entire solution must be written using* ***pure Python*** *(no external libraries such as* `re` *or* `string` *may be used).*

==========

Project 3. `Working Directory Generator`

The goal of this program is to automatically create a ***working directory structure*** *for the students of a given cohort at AIMS-Rwanda. This task will be automated using a program written in* ***pure Python*** *(no external libraries such as* `pandas` *are allowed).*
The program will receive two CSV files as input:

1. *A* ***course list file****: each line contains the name of a course offered to the cohort.*
2. *A* ***student list file****: each line contains the student's last name and first name, separated by a comma.*

The program must then:

1. *Generate a main working directory named* `AIMS-Rwanda-Workspace`*.*
2. *For each student, create a subdirectory inside the main directory using the format:*

   $$\text{“}last\_name, first\_name\text{”}$$

   with all spaces and special characters normalized (e.g., accents removed, etc.).
3. *Inside each student's directory, create one subdirectory per course from the course list.*
4. *Inside each course subdirectory, create an empty text file named* `README.txt`*.*
5. *If any directories or files already exist from a previous run, do not overwrite them but print a message that they were skipped.*
6. *At the end, print a summary of how many student folders and course subfolders were successfully created.*

Additional Complexity:

- *Validate that both CSV files exist and are not empty. Print an error message and stop execution otherwise.*
- *Handle names with hyphens, apostrophes, or accented characters by normalizing them (e.g., “García-López” becomes “Garcia-Lopez”).*
- *Ensure that the program works regardless of letter case (lowercase, uppercase, mixed).*
- *Include a simple main menu that lets the user run the generation process again with a different pair of CSV files, or quit.*

==========

Project 4. `Capitalize It`

Many people do not use capital letters correctly, especially when typing on small devices like smartphones. In this exercise, you will write a program that ***automatically fixes capitalization*** *in a string using* ***pure Python*** *(no external libraries allowed).*
Your program must:

1. *Capitalize the first non-space character of the entire string.*
2. *Capitalize the first non-space character following any period (`.`), exclamation mark (`!`), or question mark (`?`).*
3. *Replace any standalone lowercase "i" (preceded and followed by spaces or punctuation) with uppercase "I".*
4. *Normalize multiple spaces so that there is only a single space between words, but preserve line breaks.*
5. *Preserve all other characters (apostrophes, commas, quotes, etc.) exactly as they are.*
6. *Ignore leading/trailing spaces in the input string.*

Additional Complexity:

- *The program should also handle inputs with inconsistent punctuation spacing, ensuring that there is exactly one space after `.`, `!`, or `?` if another sentence follows.*
- *The program should not falsely capitalize letters inside words (e.g., `apple` should remain lowercase unless it is at the start of a sentence).*
- *Handle text spanning multiple lines correctly, capitalizing sentence starts across line breaks.*

Your program must define a function `capitalize_smartly(text: str) -> str` that performs all the above transformations and returns the corrected string. Include a main program that reads a string from the user, applies the function, and prints the result.

==========

Project 5. `License Plate Detection`

Write a program in ***pure Python*** *(no external libraries) that detects the presence of Senegalese vehicle license plate numbers* inside any text string.
We consider the following formats (letters are A–Z; digits are 0–9):

`XY-abcd-T` *or* `XY-abcd-ZT`

where X, Y, Z, T are letters and a, b, c, d are digits.
Requirements

1. *The input is an arbitrary string; the output is a Boolean. If* `True`*, the program must also print all detected license plates.*
2. *Matching is case-insensitive; detected plates must be* normalized *to uppercase and printed in canonical hyphenated form* `XY-ABCD-T/XY-ABCD-ZT`*.*
3. *Allow either a hyphen or a single space as a separator (*`-` *or* ` `*). For example,* `XY 1234 T` *and* `xy-1234-zt` *must be recognized and normalized.*
4. *Do* not *match substrings embedded within longer alphanumeric tokens (e.g., do not match across letters/digits without a boundary).*
5. *If multiple plates appear, print them in the order they appear, without duplicates.*
6. *If no plate is found, print a clear message stating so.*

Additional Complexity

- *Tolerate trailing punctuation around a plate (e.g., commas or periods).*
- *Count and display the total number of* unique *plates found at the end.*
- *Provide a minimal text menu to test multiple inputs until the user quits.*

==========

Project 6. `Hexadecimal and Decimal Digits`

Write two functions, `hex2int` *and* `int2hex`*, that convert between hexadecimal digits (0, 1, 2, . . . , 9, A, B, C, D, E, F) and base-10 integers.*

1. `hex2int`*:*
   - *Takes a string containing a single hexadecimal digit (uppercase or lowercase).*
   - *Returns its corresponding integer value (0–15).*
   - *If the string is not a valid hexadecimal digit, stop execution with a clear error message.*
2. `int2hex`*:*
   - *Takes an integer between 0 and 15 (inclusive).*
   - *Returns the corresponding hexadecimal digit (uppercase).*
   - *If the integer is outside this range, stop execution with a clear error message.*

Additional Complexity:

- *Extend* `hex2int` *to support input strings with multiple characters (e.g.,* `"1A"`*), returning the correct decimal value for the entire number.*
- *Extend* `int2hex` *to support integers larger than 15, returning their full hexadecimal representation.*
- *Ensure that both functions work without using built-in Python shortcuts like* `int(x, 16)` *or* `hex(x)` *— you must implement the conversion logic manually.*

- *Provide a small interactive main program that:*
  1. *Prompts the user to choose between* `hex→int` *or* `int→hex` *conversion.*
  2. *Reads the input value.*
  3. *Displays the result or a meaningful error message.*