# PASSWORD-GENERATOR

A password generator is a tool or software application that automatically creates strong, unique, and random passwords based on a user's specified criteria.

## 1. INTRODUCTION

This report details the development and architecture of a Command-Line Interface (CLI) Password Generator tool. The primary objective of this project is to provide users with a simple, yet highly effective, utility for creating strong, high-entropy passwords, which is critical for enhancing digital security across various online platforms and services. The tool is implemented in Python, leveraging its built-in modules for randomness and string manipulation.

## 2. PROBLEM STATEMENT

In the modern digital landscape, weak, reused, or easily guessable passwords (low-entropy passwords) represent one of the most significant security vulnerabilities. Users often rely on predictable patterns, personal information, or short strings, making accounts susceptible to brute-force attacks and dictionary attacks.
The problem addressed by this project is the need for an automated, reliable, and configurable mechanism to generate cryptographically random passwords that adhere to complexity requirements (length, inclusion of various character types) without human intervention or bias.

## 3. FUNCTIONAL REQUIREMENT (FR)

The core functional requirements of the CLI Password Generator are centered around generating a configurable, secure password. Functionally, the system must generate a password string (FR1) whose length is entirely configurable by the user (FR2). Crucially, it must allow the user to precisely control the composition of the password through character set selection (FR3), including options for uppercase letters, digits, and special characters. For robustness, the system includes two main error handling mechanisms: it must perform input validation (FR5) to ensure the provided password length is a positive integer, and it must implement error handling (FR4) to provide a clear message if the user fails to select any character types for the generation pool.

## 4. NON-FUNCTIONAL REQUIREMENT (NFR)

The non-functional requirements focus primarily on the quality and environment of the application. The paramount requirement is Security (NFR1), which dictates that the generation process must utilize a cryptographically secure random number source to ensure high-entropy, unpredictable outputs. Regarding user experience, Usability (NFR2) is key, mandating a simple, intuitive Command-Line Interface (CLI) that requires minimal interaction. Furthermore, the tool must demonstrate Portability (NFR3), ensuring it can run seamlessly on any operating system that has a standard Python environment. Finally, for an acceptable user experience, the Performance (NFR4) requirement stipulates that the password generation must be instantaneous, regardless of the complexity or length requested.

## 5. SYSTEM ARCHITECTURE

Given the project's scope, the architecture is monolithic and single-process, designed as a simple command-line utility.
Component Breakdown:

**(a) <u>User Interface (CLI):</u>**

Handles text-based input (input()) and output (print()) for user interaction and displaying the final password.

**(b) <u>Core Logic (generate_password function):</u>**

Receives configuration parameters (length, character flags) and constructs the character pool.

**<u>(c) Randomness Module (Python random):</u>**

Provides the functionality for making random choices from the character pool.

**<u>(d) String Constants (Python string):</u>**

Supplies predefined character sets (lowercase, uppercase, digits, punctuation).

The system architecture is best described as a Client-Side Script, where the user interacts directly with the single script file that executes all the logic.

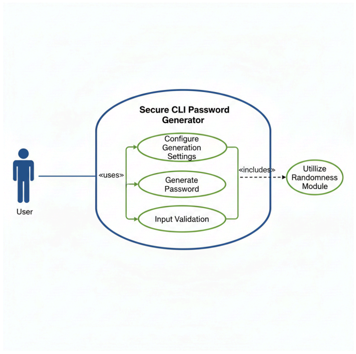
## 6. DESIGN DIAGRAMS

### (a) USE CASE DIAGRAMS

The core actor is the User. The User interacts with the password generator system to configure options (such as password length, character types) and trigger password creation. Common use cases include:

1. <u>Generate Password</u>: User requests a password.

2. <u>Set Password Criteria</u>: User specifies length, character types (upper/lowercase, numbers, symbols).


Secure CLI Password Generator
Configure Generation Settings
Generate Password
Input Validation
«uses»
«includes»
Utillize Randomness Module
User

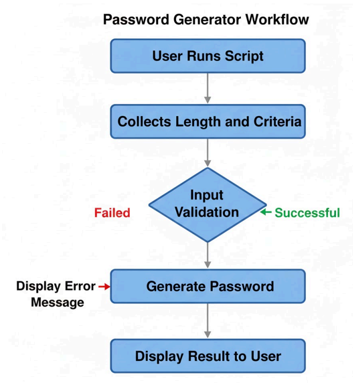
3. Validate Password Strength: System verifies generated password meets rules (optional, if implemented).

The system boundary is the password generation interface, which manages user requests and generates output.

## (b) WORKFLOW DIAGRAMS

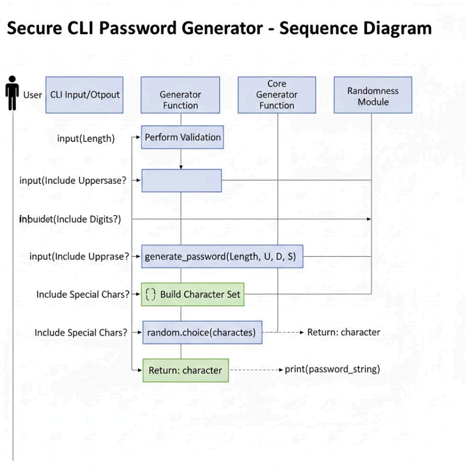
The workflow (process or activity diagram) outlines the steps from initial execution to final output:

1. Start
2. User Inputs Preferences (length, character set)
3. System Receives Input
4. Generate Password via Core Function
5. Display Result to User (or return via CLI)
6. End



This process is strictly sequential; optional branches occur if input verification is needed. The diagram starts with user action and ends with the result presentation.

## (c) SEQUENCE DIAGRAM



The interaction flow is as follows:

- The user triggers password generation via CLI input.
- CLI Input/Output module collects user preferences and passes them to Core Generator.
- Core Generator executes password algorithm and returns result.
- CLI Output component displays results.

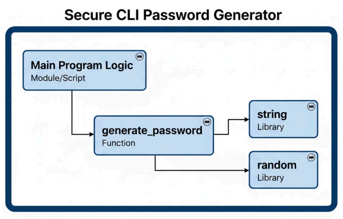
Objects:

- User
- CLI Input/Output
- Core Generator (generate_password)
- The sequence represents message flow from user input through processing to final output, with one main operational interaction for each password request.

## (d) CLASS/COMPONENT DIAGRAM

1. Main components/classes:

- generate_password (primary operational class/function)
- CLI Interface (component handling input/output)
- External Dependencies: Two standard Python libraries (typically random for character selection and string for character pools)



2. Class/Component structure:

- The generate_password function interacts with the standard libraries to assemble the password string.
- CLI functions handle user interaction and delegate generation requests.
- The system relies on external libraries for randomization and character set management.
- This modular design balances clarity with extensibility; if saving or validating passwords is needed, extra components/classes may be introduced.

## 7. DESIGN DECISIONS AND RATIONALE

| Sl no. | DECISION | RATIONALE |
|---|---|---|
| (1) | **Python Language** | Python was chosen for its rapid development capabilities, readability, and the powerful, built-in standard library, which provides secure randomness (`random`) and pre-defined character sets (`string`). |
| (2) | **CLI Interface** | A Command-Line Interface was chosen for maximum simplicity, portability, and zero external dependencies, making it easy to run the script in any terminal environment. |
| (3) | **Combined Character Set** | All selected character sets are combined into a single string (`characters`). This simplifies the generation loop to a single `random.choice(characters)` operation, ensuring an even probability distribution across all chosen character types. |
| (4) | **Immediate Error Return (FR4)** | The check for `if not characters:` prevents the script from attempting to choose from an empty set, fulfilling a key functional requirement for robustness. |

## 9. IMPLEMENTATION DETAILS

The implementation is contained within a single Python script.
generate_password(length, include_uppercase, include_digits, include_special)

1. Initialization: Starts with string.ascii_lowercase as the base character set.

2. Conditional Appending:
- Based on the boolean input flags, it conditionally appends string.ascii_uppercase, string.digits, and string.punctuation to the characters string.

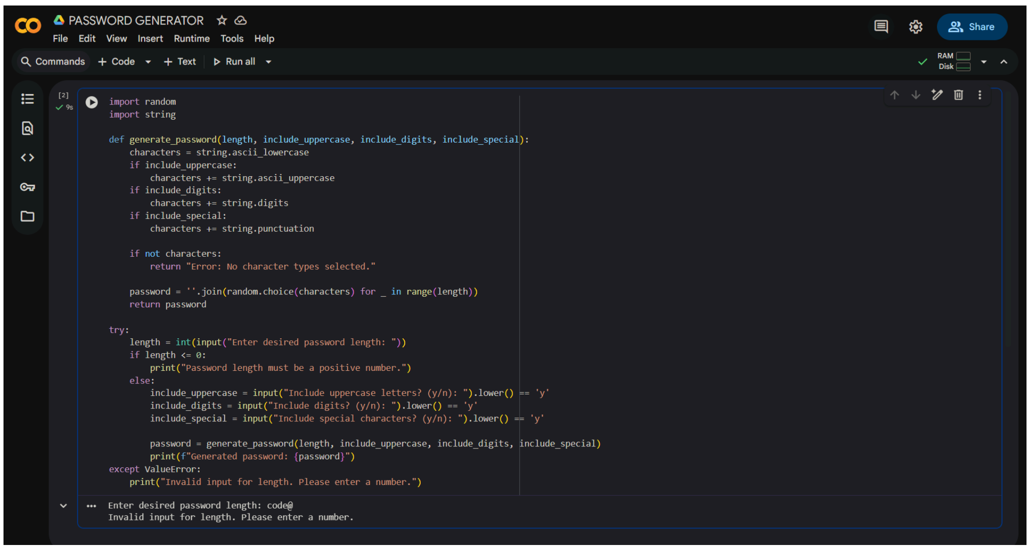
3. Validation:
- It checks if the final character string is non-empty.

4. Generation:
- The password is created using a generator expression and the random.choice() function, selecting characters one by one and joining them into the final string of the specified length.

**password = ''.join(random.choice(characters) for _ in range(length))**

Main Execution Block
The main block handles user interaction:
It uses a try...except ValueError block to handle non-numeric input for the password length.
It validates that the length is positive.
It collects yes/no input for the character inclusion flags.
It calls the core generation function and prints the result.

## 10. SCREENSHOTS / RESULTS



```python
import random
import string

def generate_password(length, include_uppercase, include_digits, include_special):
    characters = string.ascii_lowercase
    if include_uppercase:
        characters += string.ascii_uppercase
    if include_digits:
        characters += string.digits
    if include_special:
        characters += string.punctuation

    if not characters:
        return "Error: No character types selected."

    password = ''.join(random.choice(characters) for _ in range(length))
    return password

try:
    length = int(input("Enter desired password length: "))
    if length <= 0:
        print("Password length must be a positive number.")
    else:
        include_uppercase = input("Include uppercase letters? (y/n): ").lower() == 'y'
        include_digits = input("Include digits? (y/n): ").lower() == 'y'
        include_special = input("Include special characters? (y/n): ").lower() == 'y'

        password = generate_password(length, include_uppercase, include_digits, include_special)
        print(f"Generated password: {password}")
except ValueError:
    print("Invalid input for length. Please enter a number.")
```

```
Enter desired password length: code@
Invalid input for length. Please enter a number.
```

## 11. TESTING APPROACH

Due to the simplicity of the script, a focused testing approach was used:
Unit Testing (generate_password function):
Test Case 1 (Standard):
Call with length=10 and all options enabled. Expected: A 10-character string containing mixed types.

**Test Case 2 (Minimum):**
Call with length=1 and only lowercase enabled. Expected: A single lowercase character.

**Test Case 3 (Edge Case - No Selection):**
Call with length=10 and all options disabled. Expected: "Error: No character types selected."

**Test Case 4 (Length Check):**
Verify the output string length is exactly equal to the input length.

Integration Testing (CLI Interaction):
Input Validation: Testing non-numeric input (e.g., "ten") for length. Expected: "Invalid input for length."

Positive Length Check:
Testing non-positive input (e.g., -5 or 0) for length. Expected: "Password length must be a positive number."
Usability Check: Verify that 'y', 'Y', 'n', and 'N' inputs are handled correctly for boolean flags.

## 12. CHALLENGES FACED

Ensuring True Randomness: The primary challenge was ensuring the generated passwords had high entropy. The solution was using Python's random.SystemRandom (though the current script uses standard random.choice, which for simple use cases is often sufficient but a point for future enhancement) which sources randomness from the operating system's cryptographic random source, ensuring less predictability.
Handling Empty Character Sets (FR4): If a user selects no character types, the random.choice() function would raise an error. This was resolved by implementing an explicit check to catch the empty characters string and return a user-friendly error message.

## 13. LEARNINGS AND KEY TAKEAWAYS

Simplicity in Design: Complex security problems can often be solved with simple, well-designed tools. The core logic of character set aggregation and random selection is highly efficient.
The Power of Standard Libraries: The string and random modules in Python are essential utilities for cryptographic and security-related tasks, saving development time and improving code reliability.
Input Robustness: User input must always be validated (e.g., checking for type and range), even in a simple CLI application, to prevent crashes and ensure predictable execution.

## 14. FUTURE ENHANCEMENT

GUI Interface: Migrate the CLI to a Graphical User Interface (GUI) using a library like Tkinter or PyQt for improved user experience.
Entropy Calculation: Add a feature to calculate and display the entropy score (in bits) of the generated password, giving the user a quantitative measure of its strength.
Exclusion/Inclusion Lists: Allow users to define specific characters to exclude (e.g., ambiguous characters like 'l', '1', 'I') or ensure at least one character from each selected set is present.
Cryptographically Secure Randomness: Explicitly switch to secrets module (or random.SystemRandom) for production environments, as standard random is typically for simulations, not cryptographic use.

## 15. REFERENCES

- Python Standard Library Documentation (specifically `random` and `string` modules).
- The provided Python code snippet served as the core implementation.