Project Report: PyPassword Generator

1. Cover Page

Field	Detail
Project Title	PyPassword Generator
Submitted by	KARTIK YADAV
Registration Number	25BAS10071
Date of Submission	November 2025

2. Introduction

This document details the development of the PyPassword Generator, a simple command-line application designed to create strong, randomized passwords based on user-defined length requirements for letters, numbers, and symbols. In an age where digital security is paramount, tools that automate the creation of complex and unique passwords are essential. This project utilizes Python's built-in `random` module to ensure high entropy and unpredictability in the generated output, promoting better user security.

3. Problem Statement

The lack of strong, unique passwords across various online services is a major security vulnerability. Users often rely on weak, predictable passwords or reuse the same password, making them susceptible to dictionary attacks and credential stuffing. The problem addressed by this project is the need for a simple, accessible tool that generates cryptographically strong, randomized passwords that meet specific complexity requirements (length of letters, symbols, and numbers).

4. Functional Requirements

1. **User Input:** The system must prompt the user for the desired number of letters, symbols, and numbers.
2. **Character Set Definition:** The system must define distinct lists for all possible lowercase/uppercase letters, numbers (0-9), and common symbols.
3. **Character Selection:** The system must randomly select the specified count of characters from each respective list.
4. **Password Combination:** The system must combine all selected characters into a single list or string.
5. **Randomization/Shuffling:** The system must fully randomize the order of the combined characters to maximize complexity and prevent predictable patterns.
6. **Output Display:** The system must display the final, generated password to the user.

5. Non-functional Requirements

1. **Usability:** The application must be easy to use via the command-line interface with clear prompts.
2. **Performance:** The password generation process must be instantaneous (near-zero latency).
3. **Security (Password Strength):** The generated password must be highly randomized and meet the complexity requirements provided by the user.
4. **Maintainability:** The code must be well-structured and easy to understand and modify.

6. System Architecture

Given the project's simplicity, the architecture is a **Single-Tier, Standalone Command-Line Application**.

- **Presentation Layer:** Handled by the console's `input()` and `print()` functions.
- **Application Logic Layer:** Handled by the Python script, which manages user input, character selection, list manipulation, and randomization using the `random` module.
- **Data Layer:** Minimal; character sets are hardcoded as Python lists.

The architecture is highly coupled but perfectly suited for a non-persistent, single-session utility tool.

7. Design Diagrams

Use Case Diagram

This diagram shows the user's primary interaction points with the system.

Workflow Diagram

This illustrates the step-by-step process of the application logic.

Sequence Diagram

Class/Component Diagram

8. Design Decisions & Rationale

The final implementation (labeled 'hard version' in the source code) uses the following key design decisions:

1. **List for Composition:** Characters are first collected into a list (`password_list`) rather than concatenated directly into a string.
   - **Rationale:** Lists are mutable, which is essential for the second key decision: shuffling.
2. **Full Shuffling:** The `random.shuffle()` function is applied to the entire `password_list`.
   - **Rationale:** This is the *most critical* step for creating a secure password. If the characters were simply concatenated (e.g., all letters first, then all symbols, then all numbers), the password would be easy to predict. Shuffling the entire list ensures true randomness and high entropy.

9. Implementation Details

The core implementation involves four stages:

1. **Initialization:** Defining the character source lists (`letters`, `numbers`, `symbols`).
2. **Input:** Collecting the three integer inputs from the user.

Population: Using `for` loops and `random.choice()` to append the required number of random characters from the source lists into the `password_list`.

```
for char in range(0 , nr_letters):
   password_list.append(random.choice(letters))
# ... repeated for numbers and symbols
```

3.
4. **Finalization:**
   - Shuffling the list: `random.shuffle(password_list)`

Joining the list into the final string:

```
password = ""
for char in password_list:
   password += char
```

   -

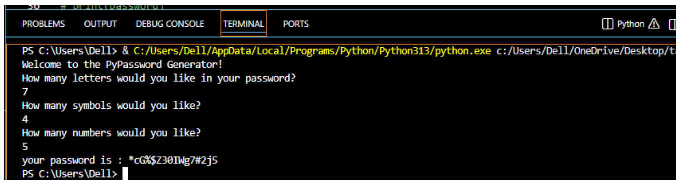
10.Source code

```
import random

letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B',
'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

numbers = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

symbols = ['!', '#', '$', '%', '&', '(', ')', '*', '+']



print("Welcome to the PyPassword Generator!")

nr_letters = int(input("How many letters would you like in your password?\n"))

nr_symbols = int(input(f"How many symbols would you like?\n"))

nr_numbers = int(input(f"How many numbers would you like?\n"))




#My version

# random_letters = random.choices(letters,k=nr_letters)

# random_numbers = random.choices(numbers,k=nr_numbers)

# random_symbols = random.choices(symbols,k=nr_symbols)

#

# password = random_letters + random_numbers + random_symbols

# print(password)

# random.shuffle(password)
# print(password)

# for i in password :

#     print(i, end='')




#Easy version

# password = ""

# for char in range(0 , nr_letters):

#     password += random.choice(letters)

#

# for char in range(0 , nr_numbers):

#     password += random.choice(numbers)

#

# for char in range(0, nr_symbols):

#     password += random.choice(symbols)

#

# print(password)




#My version for hard code

# all_range = letters+symbols+numbers

# range_ = nr_symbols + nr_letters + nr_numbers
# password = ""

# for char in range(0,range_):

#     password += random.choice(all_range)

# print(password)



#hard version

password_list = []

for char in range(0 , nr_letters):

    password_list.append(random.choice(letters))



for char in range(0 , nr_numbers):

    password_list.append(random.choice(numbers))



for char in range(0, nr_symbols):

    password_list.append(random.choice(symbols))



random.shuffle(password_list)



password = ""

for char in password_list:

    password += char
```

11. Screenshots / Results

(Since this is a report template, provide an example console session here.)

Console Input	Console Output
Welcome to the PyPassword Generator!	**your password is : 8#vY2k(E+7p**
How many letters would you like in your password?	
8	
How many symbols would you like?	
2	
How many numbers would you like?	
3	



```
PS C:\Users\Dell> & C:/Users/Dell/AppData/Local/Programs/Python/Python313/python.exe c:/Users/Dell/OneDrive/Desktop/t
Welcome to the PyPassword Generator!
How many letters would you like in your password?
7
How many symbols would you like?
4
How many numbers would you like?
5
your password is : *cG%$Z30IWg7#2j5
PS C:\Users\Dell>
```

12. Testing Approach

The testing was conducted manually on the command line using the following approaches:

1. **Length Validation:**
   - Test Case: Request 4 letters, 2 symbols, 2 numbers.
   - Expected Result: Password must be exactly 8 characters long.
2. **Content Validation:**
   - Test Case: Request 1 letter, 0 symbols, 0 numbers.
   - Expected Result: Password must be a single letter.

3. **Randomization Test (Key Test):**
   - Test Case: Request a high number of characters (e.g., 5 letters, 5 symbols, 5 numbers) and run the program multiple times.
   - Expected Result: The character types must appear in random order (e.g., Symbol $\rightarrow$ Letter $\rightarrow$ Number $\rightarrow$ Symbol, etc.) in every run, confirming the efficacy of `random.shuffle()`.

13. Challenges Faced

The primary challenge was ensuring the final password was truly *randomly* ordered, not just randomly *selected*.

Challenge	Solution Implemented in Code
Ordered output: Initial versions (like the commented 'Easy version') would group characters by type (e.g., `letterssymbolsnumbers`).	Using a list (`password_list`) to hold all characters.
Shuffling: Ensuring the entire sequence is randomized.	Applying `random.shuffle()` to the `password_list` before iterating over it to construct the final string.

14. Learnings & Key Takeaways

- **Mutability for Transformation:** The necessity of using a mutable data structure (a Python list) to perform transformations like shuffling before converting to the final immutable output (a string).
- **Importance of Entropy:** Understanding that security is not just about the *inclusion* of different character types, but the *unpredictability* of their order.
- **The Python `random` Module:** Gaining proficiency with powerful functions like `random.choice()` (for selecting one item) and `random.shuffle()` (for randomizing a sequence).

15. Future Enhancements

1. **Password Strength Meter:** Integrate logic to visually assess and rate the generated password's strength (e.g., "Good," "Excellent") based on length and variety.
2. **Input Validation:** Add error handling to ensure users only enter positive integers for the required counts, preventing crashes from non-numeric input.

3. **Exclude Ambiguous Characters:** Add an option to exclude characters that can be visually confused (e.g., 'l', '1', 'I', 'o', '0', 'O').
4. **GUI Interface:** Convert the command-line interface (CLI) to a simple Graphical User Interface (GUI) using Tkinter or PyQt for broader usability.

16. References

1. Python 3 Documentation (for `random` module).
2. Open-Source Cryptography Principles (concept of password entropy).