Python File Integrity Monitoring System

Cyber security

Python File Integrity Monitoring System

Student name: Mohammed Ali Mohammed Al-khawlani **Student ID**:202010875

Department: Cybersecurity **Submission** Date:2/2/2026

Abstract

This project presents a tool designed to monitor and track file and directory operations. Its main objective is to detect any changes in the file system, such as creation, deletion, modification, or renaming, and log these events for analysis. The tool uses SHA hashing to verify file integrity and detect unauthorized changes. It also records the original and updated names of renamed files. The monitoring logic is implemented using Object-Oriented Programming (OOP) principles, which allow for modular and scalable design. The results are displayed in a simplified and user-friendly format, making it easier to understand and analyze file system activity.

introduction

In today's rapidly evolving digital world, monitoring file system activity has become a critical task in cybersecurity. The ability to detect unauthorized changes, access attempts, and suspicious behavior within directories and files is essential for protecting sensitive data and maintaining system integrity. This project aims to address this challenge by developing a tool that monitors file and directory operations—such as creation, deletion, modification, and renaming—and logs these events for further analysis. The tool also tracks changes in file names and content, helping identify potential threats or tampering. By implementing this solution using Object-Oriented Programming (OOP), the tool

ensures modularity, scalability, and ease of maintenance. It provides a fast and

efficient way for system administrators and security analysts to observe file system

behavior, detect anomalies, and respond to incidents in real time.

Methodolog

This tool was developed using Python and focuses on monitoring file and directory operations with high accuracy and responsiveness. The system is modular and scalable, built around the following core components:

1. Directory Monitor Class

This is the heart of the system. It handles: * Scanning the target directory. * Comparing current and previous states. * Detecting changes such as creation, deletion, modification, and renaming. * Logging events to a text file.

2. Hashing for Integrity

To detect file modifications, the tool uses SHA hashing. Each file's hash is calculated and stored, allowing the system to detect content changes even if the f ilename remains the same.

3. State Management

The previous state of the directory is stored in a JSON file (monitor_states.json). This includes: * File names. * Directory structure. * Hash values. The tool compares this saved state with the current scan to detect: * New files or folders. * Deleted items. * Modified content. * Renamed files or directories.

4. Logging

All detected operations are recorded in log.txt , with timestamps and operation types. Each entry is formatted for easy parsing and later visualization.

5. Menu-Driven Interface

The user interacts with the tool through a terminal-based menu. Options include: * Start monitoring. * View log summary. * Visualize results. * Reset state. * Exit. This interface ensures ease of use and guides the user through all available features.

Threading Implementation

To improve performance and responsiveness, the tool uses Python's threading module. Monitoring operations run in a background thread, allowing the interface to remain active and responsive. This ensures: * Real-time scanning without freezing the UI. * Efficient handling of large directories. * Smooth user experience.

Visualization Module

The tool includes a visualization module using matplotlib , offering: * Horizontal bar charts. * Combined charts (both views in one figure). Each chart displays the frequency of operations (create, delete, modify, rename) and is saved automatically with a timestamped filename. Colors are mapped using for a clean, professional look.

Libraries Used

os : File system access.

json : State storage.

hashlib : SHA hashing.

re : Regular expressions.

time : Timestamps and delays.

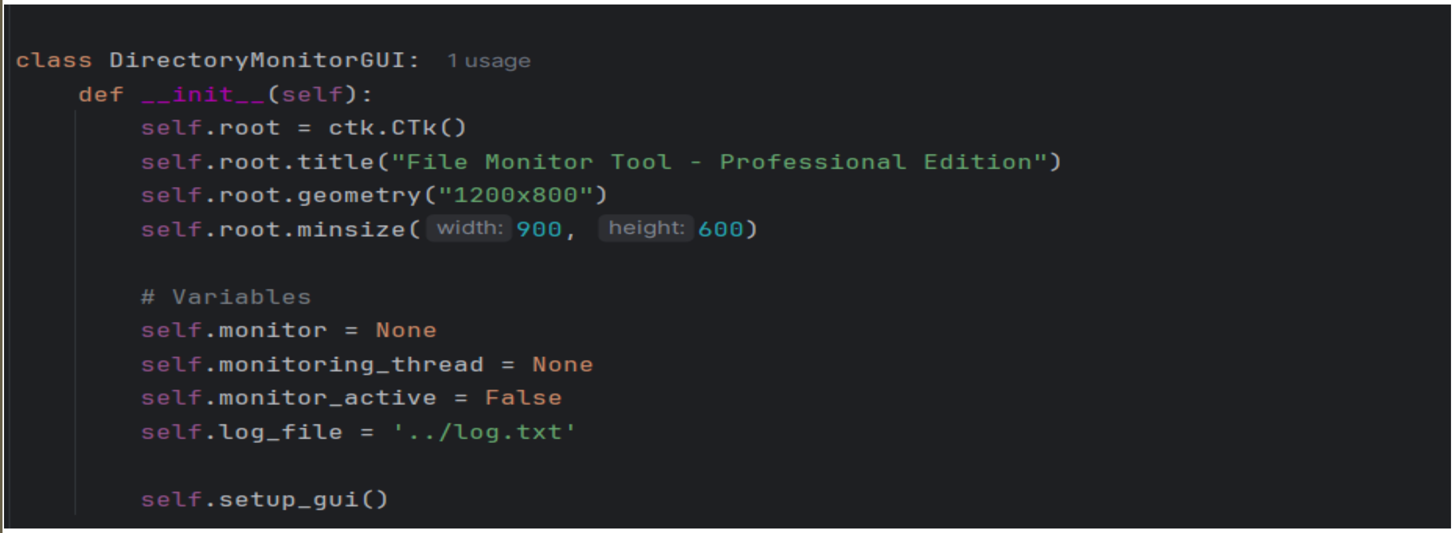
threading : Background execution.

matplotlib : Data visualization.

The tool's design is based on the DirectoryMonitor class, which encapsulates all core monitoring functionalities. This class includes several key methods that work together to detect and log changes.

DirectoryMonitor Class

This class is responsible for scanning the target directory, comparing current and previous states, detecting changes (creation, deletion, modification, renaming), logging events, and saving and loading state data. __init__(self) : This function initializes the monitoring object. It validates the directory path, sets up log and snapshot files, and loads the previous state.



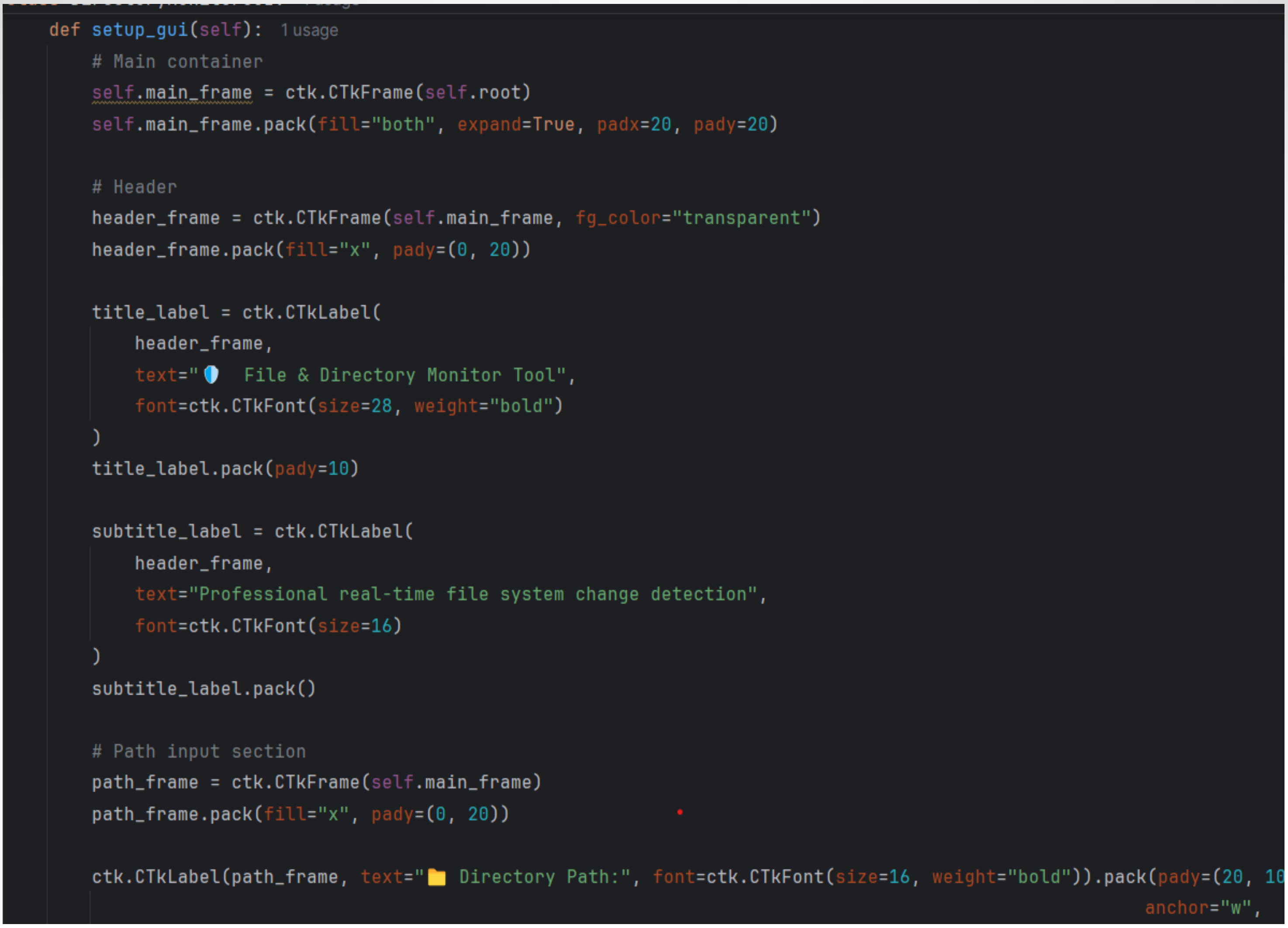
```
class DirectoryMonitorGUI:  1 usage
    def __init__(self):
        self.root = ctk.CTk()
        self.root.title("File Monitor Tool - Professional Edition")
        self.root.geometry("1200x800")
        self.root.minsize( width: 900,  height: 600)

        # Variables
        self.monitor = None
        self.monitoring_thread = None
        self.monitor_active = False
        self.log_file = '../log.txt'

        self.setup_gui()
```

Creates the main frame

- A container that holds all other elements in the window.
- It expands to fill the entire application window.

Adds a header section

- Transparent frame at the top.
- Contains:
    - A **title label**: “ File & Directory Monitor Tool” (large, bold font).
    - A **subtitle label**: “Professional real-time file system change detection” (smaller font).



```
def setup_gui(self):  1 usage
    # Main container
    self.main_frame = ctk.CTkFrame(self.root)
    self.main_frame.pack(fill="both", expand=True, padx=20, pady=20)

    # Header
    header_frame = ctk.CTkFrame(self.main_frame, fg_color="transparent")
    header_frame.pack(fill="x", pady=(0, 20))

    title_label = ctk.CTkLabel(
        header_frame,
        text="🛡  File & Directory Monitor Tool",
        font=ctk.CTkFont(size=28, weight="bold")
    )
    title_label.pack(pady=10)

    subtitle_label = ctk.CTkLabel(
        header_frame,
        text="Professional real-time file system change detection",
        font=ctk.CTkFont(size=16)
    )
    subtitle_label.pack()

    # Path input section
    path_frame = ctk.CTkFrame(self.main_frame)
    path_frame.pack(fill="x", pady=(0, 20))

    ctk.CTkLabel(path_frame, text="📁 Directory Path:", font=ctk.CTkFont(size=16, weight="bold")).pack(pady=(20, 10
                                                                                                         anchor="w",
```

Func 2

Places the text entry box (where the user types the directory path) on the **left side** of the frame.

fill="x" and expand=True make it stretch horizontally.

padx and pady add spacing around it.

Func 3

Creates a button labeled **“ Browse”**.

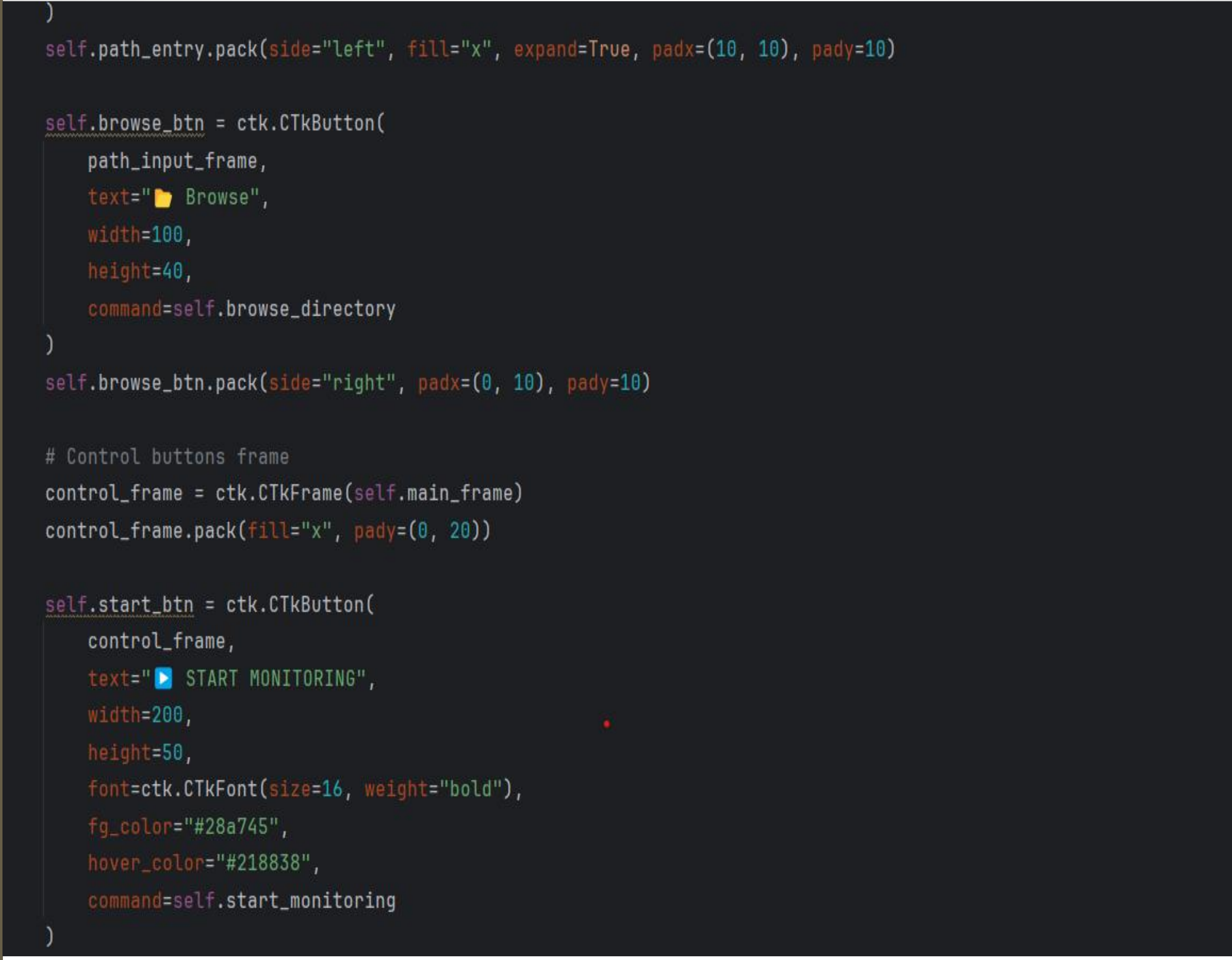
When clicked, it calls the function self.browse_directory (which should open a file dialog to select a folder).

Positioned on the **right side** of the path input frame.

Func 4

Creates a new frame to hold control buttons (like Start/Stop monitoring).

Positioned below the path input section.



```
    )
    self.path_entry.pack(side="left", fill="x", expand=True, padx=(10, 10), pady=10)

    self.browse_btn = ctk.CTkButton(
        path_input_frame,
        text="📁 Browse",
        width=100,
        height=40,
        command=self.browse_directory
    )
    self.browse_btn.pack(side="right", padx=(0, 10), pady=10)

    # Control buttons frame
    control_frame = ctk.CTkFrame(self.main_frame)
    control_frame.pack(fill="x", pady=(0, 20))

    self.start_btn = ctk.CTkButton(
        control_frame,
        text="▶ START MONITORING",
        width=200,
        height=50,
        font=ctk.CTkFont(size=16, weight="bold"),
        fg_color="#28a745",
        hover_color="#218838",
        command=self.start_monitoring
    )
```

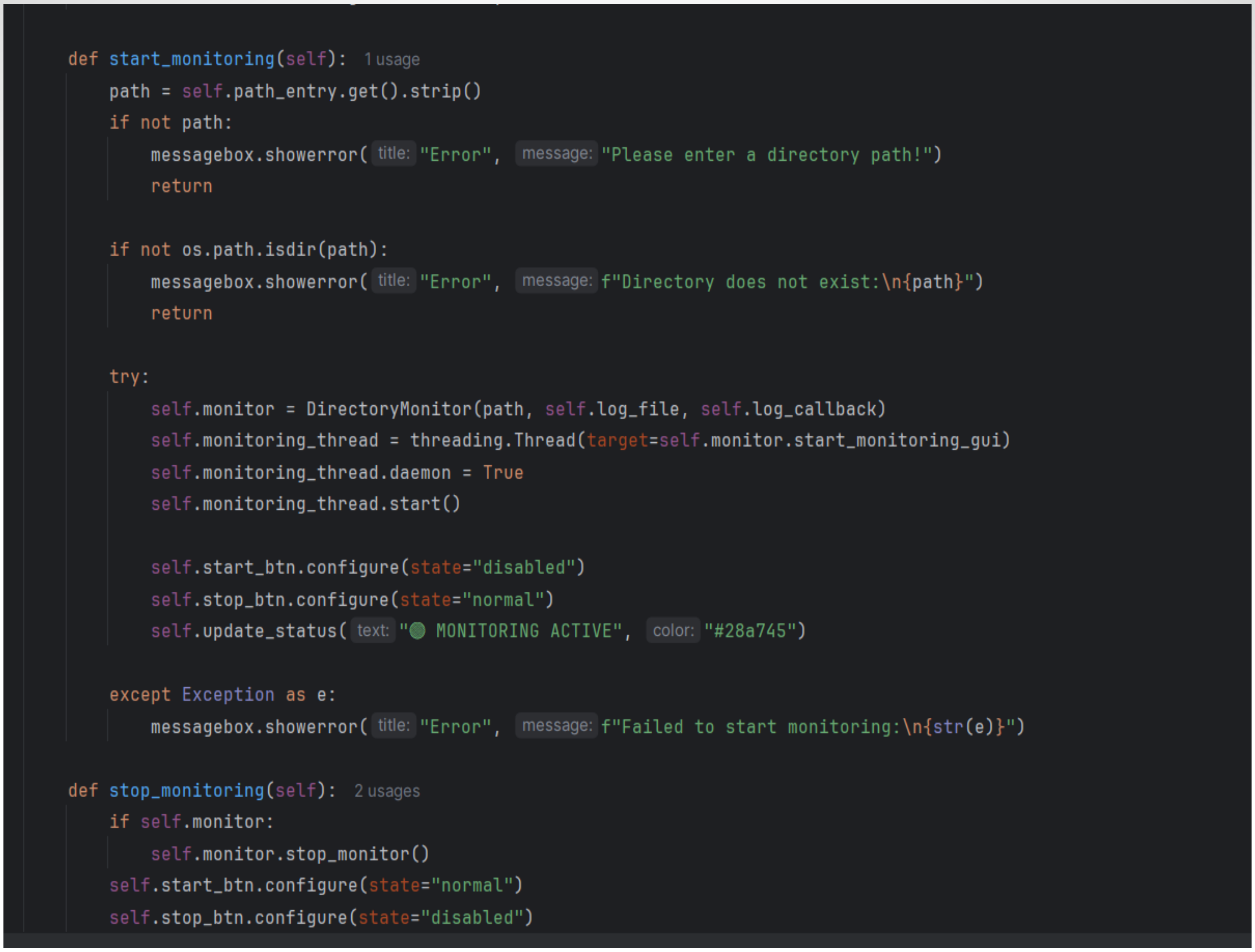
Validation checks

- If empty → show error.
- If not a valid directory → show error.

 Creates a DirectoryMonitor object and runs it in a background thread so the GUI stays responsive.

 Update button states and status

- Disable **Start** button.
- Enable **Stop** button.
- Update status label to show monitoring is active.
- **Completing `stop_monitoring(self)`**
- You need to stop the monitoring thread and reset the GUI state. A typical implementation looks like this:



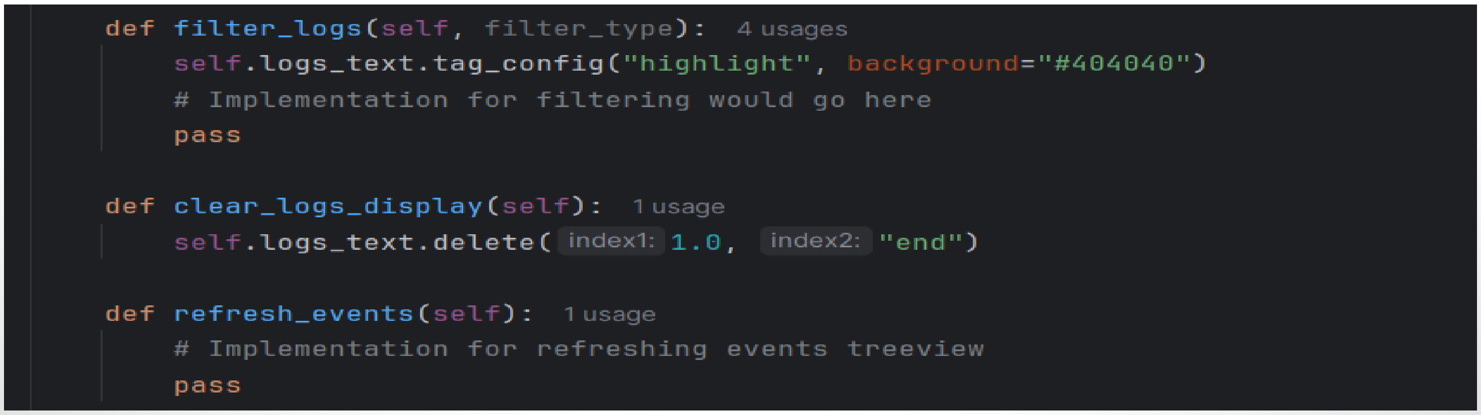
```
def start_monitoring(self):  1 usage
    path = self.path_entry.get().strip()
    if not path:
        messagebox.showerror( title: "Error",  message: "Please enter a directory path!")
        return

    if not os.path.isdir(path):
        messagebox.showerror( title: "Error",  message: f"Directory does not exist:\n{path}")
        return

    try:
        self.monitor = DirectoryMonitor(path, self.log_file, self.log_callback)
        self.monitoring_thread = threading.Thread(target=self.monitor.start_monitoring_gui)
        self.monitoring_thread.daemon = True
        self.monitoring_thread.start()

        self.start_btn.configure(state="disabled")
        self.stop_btn.configure(state="normal")
        self.update_status( text: "🟢 MONITORING ACTIVE",  color: "#28a745")

    except Exception as e:
        messagebox.showerror( title: "Error",  message: f"Failed to start monitoring:\n{str(e)}")

def stop_monitoring(self):  2 usages
    if self.monitor:
        self.monitor.stop_monitor()
    self.start_btn.configure(state="normal")
    self.stop_btn.configure(state="disabled")
```

1. filter_logs(self, filter_type)

- **Current state:** You've set up a text tag called "highlight" with a dark background (#404040), but the actual filtering logic is missing.
- **Purpose:** This function should allow the user to filter log entries based on a type (e.g., "created", "modified", "deleted").

2. clear_logs_display(self)

- **Current state:** Already implemented.
- **Purpose:** Clears the text widget (self.logs_text) completely.
- **Usage:** Called before reloading logs or when the user presses a "Clear Logs" button.



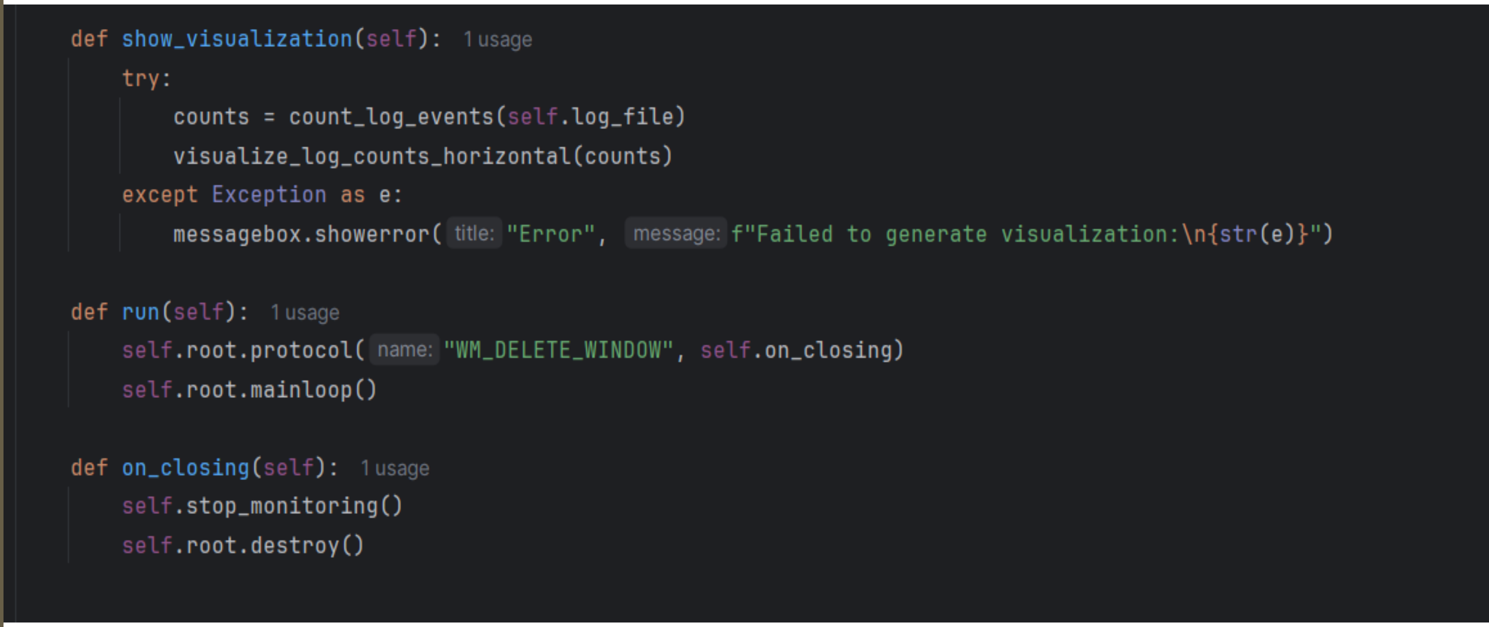
```
def filter_logs(self, filter_type):  4 usages
    self.logs_text.tag_config("highlight", background="#404040")
    # Implementation for filtering would go here
    pass

def clear_logs_display(self):  1 usage
    self.logs_text.delete( index1: 1.0,  index2: "end")

def refresh_events(self):  1 usage
    # Implementation for refreshing events treeview
    pass
```

Helper Functions

show_menu_items(pattern_f, pattern_d, name_f, log_file='log.txt') This function displays a submenu for viewing specific types of log entries (created, deleted, renamed, modified)

Calls another helper function visualize_log_counts_horizontal.

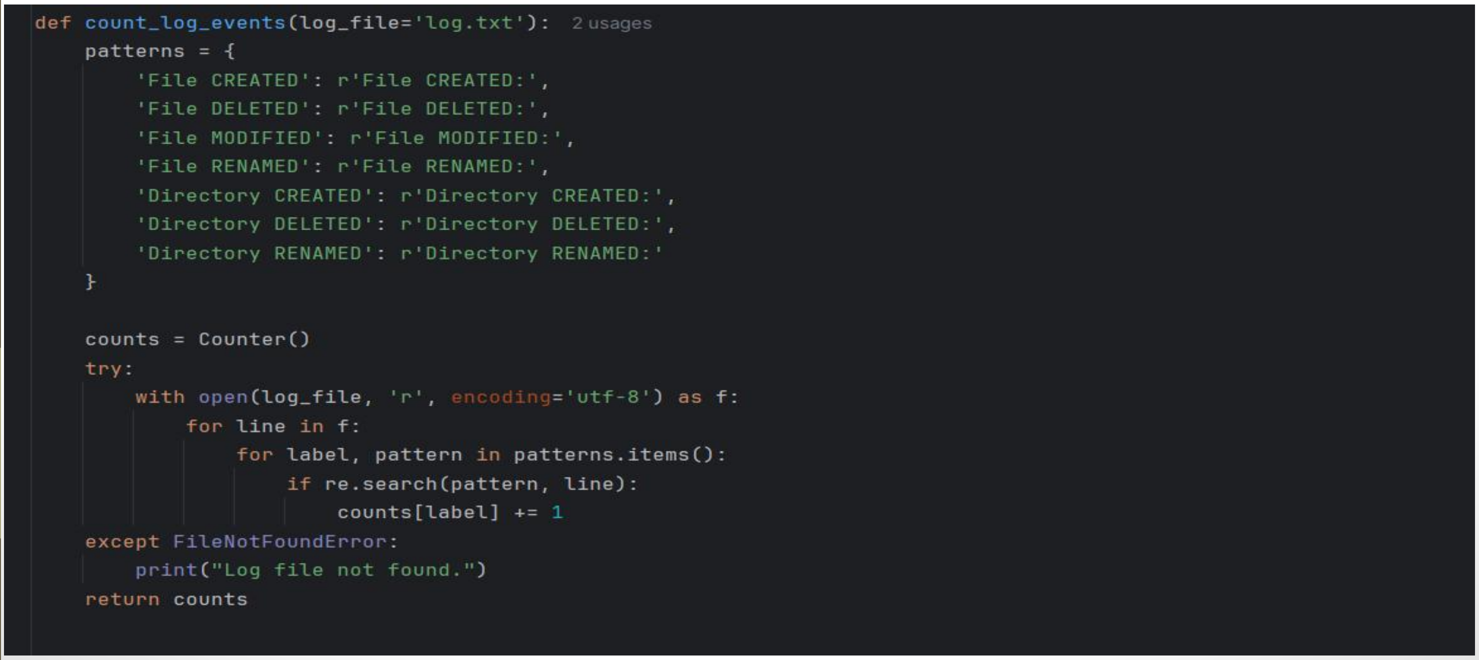
This probably uses a plotting library (like matplotlib) to display a **horizontal bar chart** showing how many times each type of event occurred



```
def show_visualization(self):  1 usage
    try:
        counts = count_log_events(self.log_file)
        visualize_log_counts_horizontal(counts)
    except Exception as e:
        messagebox.showerror( title: "Error",  message: f"Failed to generate visualization:\n{str(e)}")

def run(self):  1 usage
    self.root.protocol( name: "WM_DELETE_WINDOW", self.on_closing)
    self.root.mainloop()

def on_closing(self):  1 usage
    self.stop_monitoring()
    self.root.destroy()
```

This function count_log_events

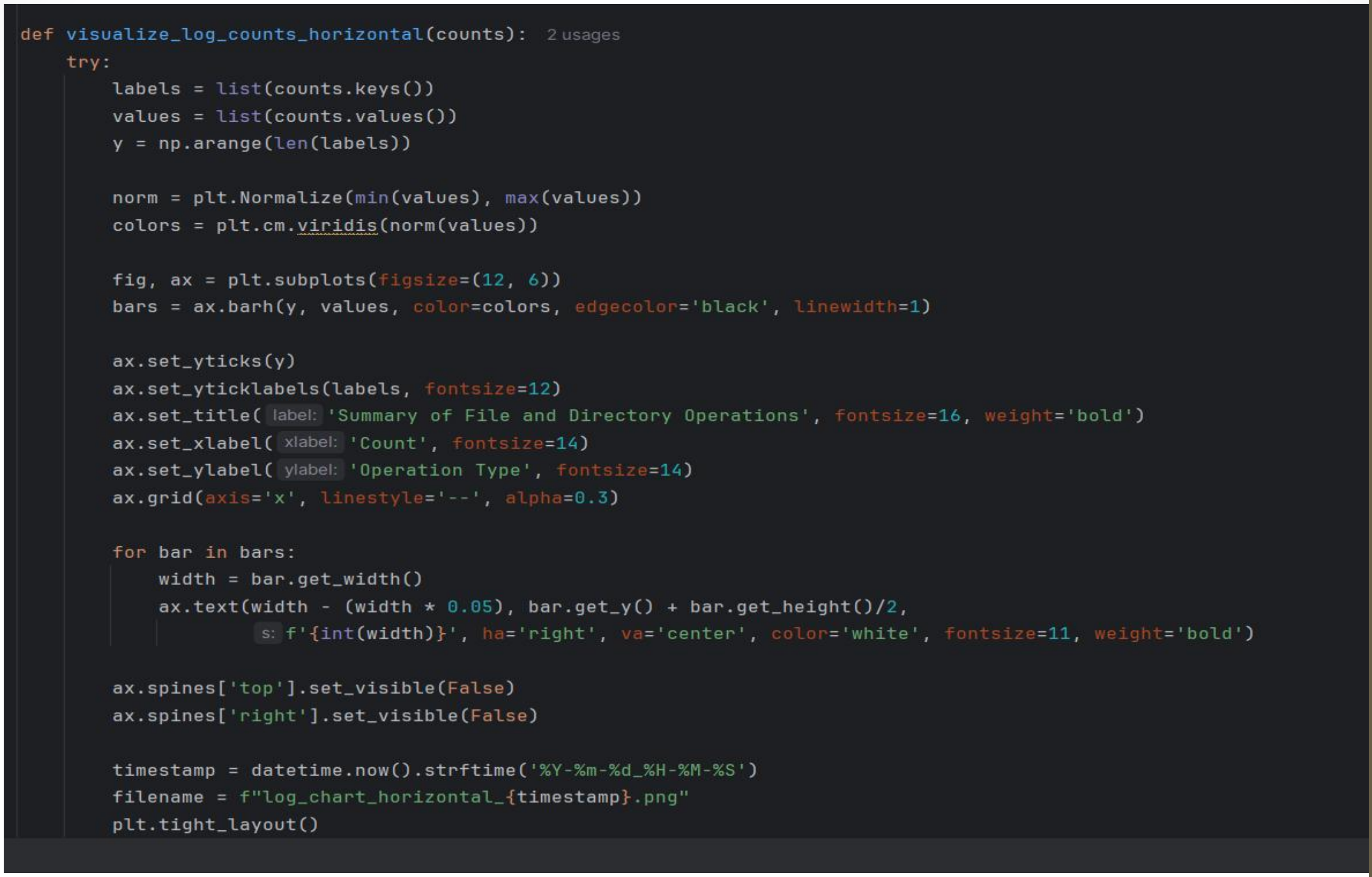
is a helper utility that scans your log file and counts how many times each type of file or directory event occurred. Let me explain it step by step:



```
def count_log_events(log_file='log.txt'):  2 usages
    patterns = {
        'File CREATED': r'File CREATED:',
        'File DELETED': r'File DELETED:',
        'File MODIFIED': r'File MODIFIED:',
        'File RENAMED': r'File RENAMED:',
        'Directory CREATED': r'Directory CREATED:',
        'Directory DELETED': r'Directory DELETED:',
        'Directory RENAMED': r'Directory RENAMED:'
    }

    counts = Counter()
    try:
        with open(log_file, 'r', encoding='utf-8') as f:
            for line in f:
                for label, pattern in patterns.items():
                    if re.search(pattern, line):
                        counts[label] += 1
    except FileNotFoundError:
        print("Log file not found.")
    return counts
```

This function visualize_log_counts_horizontal

(counts) is the companion to your count_log_events function. It takes the event counts and produces a **horizontal bar chart** using matplotlib. Let’s break it down:



```python
def visualize_log_counts_horizontal(counts):  2 usages
    try:
        labels = list(counts.keys())
        values = list(counts.values())
        y = np.arange(len(labels))

        norm = plt.Normalize(min(values), max(values))
        colors = plt.cm.viridis(norm(values))

        fig, ax = plt.subplots(figsize=(12, 6))
        bars = ax.barh(y, values, color=colors, edgecolor='black', linewidth=1)

        ax.set_yticks(y)
        ax.set_yticklabels(labels, fontsize=12)
        ax.set_title( label: 'Summary of File and Directory Operations', fontsize=16, weight='bold')
        ax.set_xlabel( xlabel: 'Count', fontsize=14)
        ax.set_ylabel( ylabel: 'Operation Type', fontsize=14)
        ax.grid(axis='x', linestyle='--', alpha=0.3)

        for bar in bars:
            width = bar.get_width()
            ax.text(width - (width * 0.05), bar.get_y() + bar.get_height()/2,
                    s: f'{int(width)}', ha='right', va='center', color='white', fontsize=11, weight='bold')

        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)

        timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
        filename = f"log_chart_horizontal_{timestamp}.png"
        plt.tight_layout()
```

Control Flow and Main Execution Logic

The main() function serves as the entry point of the program. It prompts the user to enter the path of the directory to be monitored. Once the path is provided, the program initializes an instance of the DirectoryMonitor class, which is responsible for scanning the directory, detecting changes, and logging events. Upon initialization, the program attempts to load the previous state from a JSON file named monitor_states.json . This file contains metadata about files and directories from the last scan, allowing the program to compare and detect changes such as creation, deletion, modification, or renaming. After loading the state, the main_menu() function is called. This function displays a terminal-based menu that allows the user to: * 1 Start monitoring in a separate thread. * -1 Stop monitoring. * 2, 3, 4, 5 View created, deleted, renamed, and modified items. * 6 Visualize log summary. * 0 Exit the program safely. It uses try-except blocks to handle invalid inputs and unexpected errors gracefully.

Threading Integration.

To ensure the interface remains responsive while monitoring is active, the program uses Python’s threading module. When the user selects "Start Monitoring," the start_monitoring() method is executed in a background thread. This allows real time monitoring without blocking the menu interface, making the tool suitable for continuous observation.

Summary of Execution Flow .

1.User Input: Directory path via terinal.

2. Initialization: DirectoryMonitor object created.

3. State Loading: Previous state loaded from . monitor_states.json .

4. Monitoring: Changes detected and logged.

5. Menu Interaction: User navigates options to view or visualize logs.

6. Threading: Monitoring runs in the background for real-time updates.

7. Error Handling: All critical operations wrapped in try-except .

Input and Output Handling.

The tool handles input and output in multiple ways to ensure ease of use and provide comprehensive information to the user.

Terminal Input.

The user enters the full directory path via the terminal. After submission, the program verifies the validity of the path. If the path is incorrect, an error message is displayed in the terminal. Input is handled through a command-line prompt, and the user selects from options (-) to interact with the tool.

File Input .

The program reads the current state of the directory from a file named monitor_states.json . This file contains metadata such as file hashes, sizes, and modification timestamps, allowing the tool to detect changes.

Terminal Output .

The program displays real-time messages in the terminal, including: * Monitoring status. * Detected changes. * Error messages. * Operation summaries. Each operation type (created, deleted, renamed, modified) is clearly shown to the user.

File Output

All detected operations are saved to a log file named log.txt . The log includes timestamps and operation types, making it easy to analyze changes later. The state file monitor_states.json is also updated after each scan to reflect the latest directory structure. """

Error Handling.

The tool incorporates robust error handling mechanisms to ensure its stability and reliability. The following common error types are handled:

FileNotFoundError.

This error occurs when the user enters a directory path that does not exist or is inaccessible. The program displays a clear error message and safely stops execution. This error may also occur if log.txt or monitor_states.json is missing.

PermissionError and IOError .

These errors occur when the program cannot access a file due to permission restrictions or file corruption. The tool logs the issue and continues execution without crashing """

JSONDecodeError

This error occurs when the monitor_states.json file is present but contains invalid JSON. The program logs the error and initializes an empty state to continue monitoring safely.

ValueError

This error occurs when the user enters an invalid option in the menu (e.g., letters instead of numbers). The program prompts the user to enter a valid number and does not exit.

General Exception

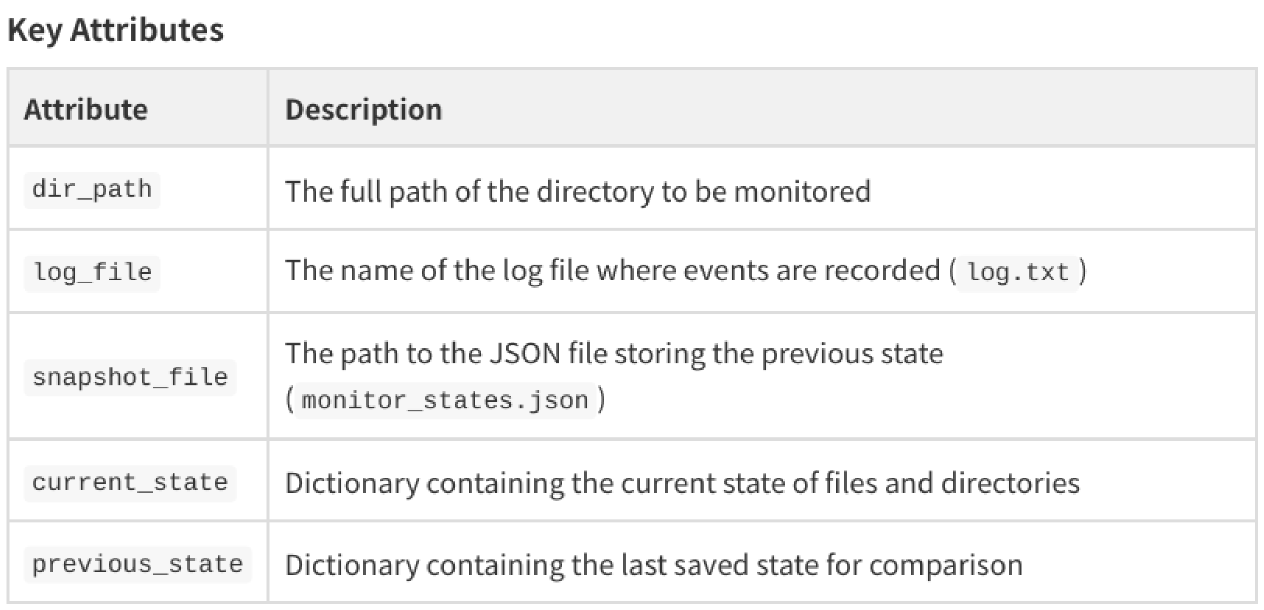
This catches any unexpected errors during runtime in main() or main_menu() . The program displays a generic error message and continues safely.

Object-Oriented Programming (OOP) Design

This monitoring tool is built using the principles of Object-Oriented Programming (OOP), which provides a structured and scalable approach to software development. The program is centered around a single class named DirectoryMonitor , which encapsulates all the core logic and functionality required for monitoring file system changes. By using OOP, the tool benefits from: * **Encapsulation**: All attributes and methods are grouped within the class, making the code organized and modular. * **Reusability**: The class can be reused in other projects or extended with additional features. * **Maintainability**: The structure allows for easy updates and debugging. * **Modeling**: The class models real-world behavior by representing a directory as an object with state and behavior.

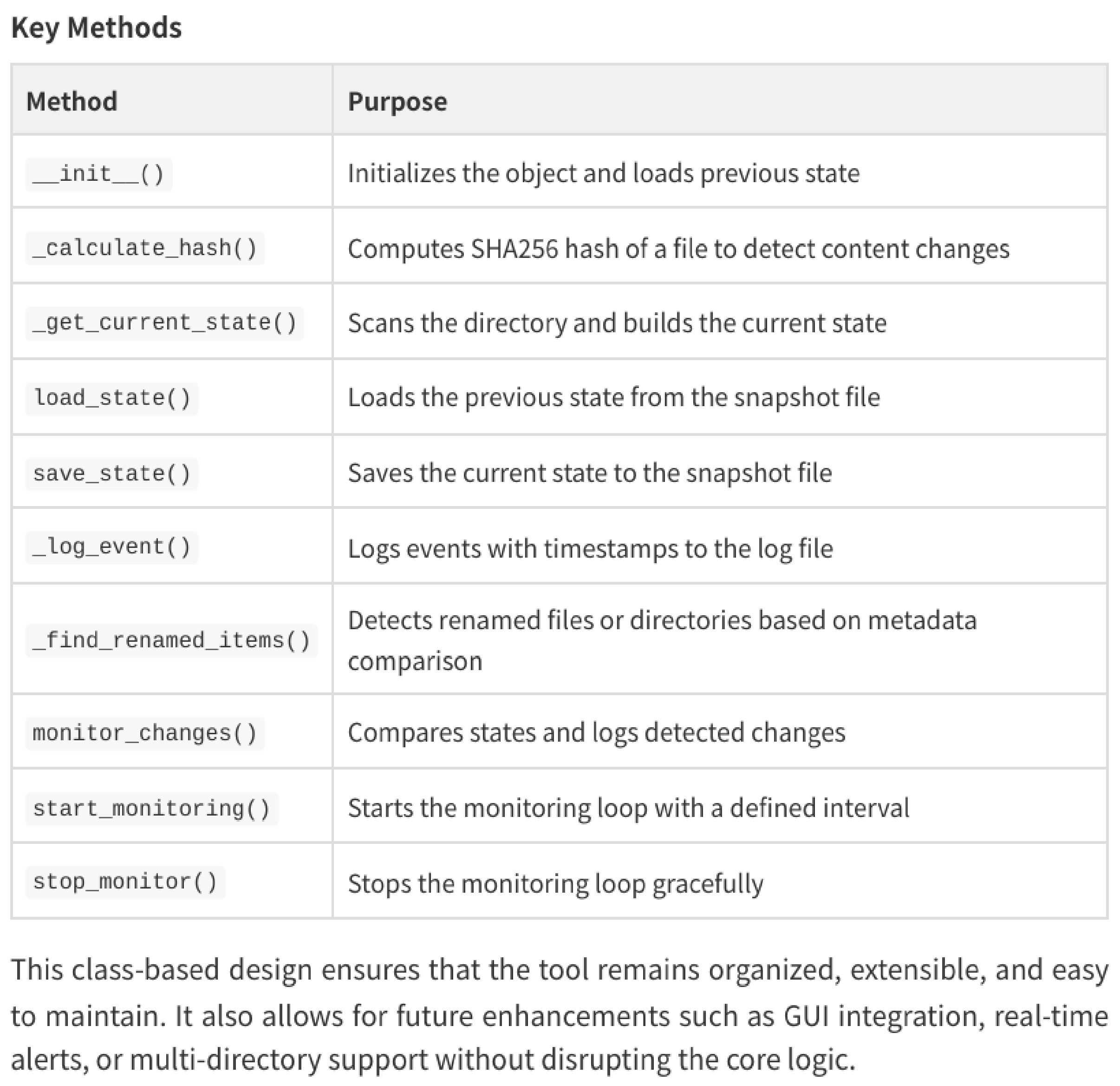
DirectoryMonitor Class

This class is responsible for: * Scanning the target directory. * Comparing current and previous states. * Detecting changes (created, deleted, modified, renamed). * Logging events. * Saving and loading state data



Key Attributes

Attribute	Description
`dir_path`	The full path of the directory to be monitored
`log_file`	The name of the log file where events are recorded (`log.txt`)
`snapshot_file`	The path to the JSON file storing the previous state (`monitor_states.json`)
`current_state`	Dictionary containing the current state of files and directories
`previous_state`	Dictionary containing the last saved state for comparison



Key Methods

Method	Purpose
`__init__()`	Initializes the object and loads previous state
`_calculate_hash()`	Computes SHA256 hash of a file to detect content changes
`_get_current_state()`	Scans the directory and builds the current state
`load_state()`	Loads the previous state from the snapshot file
`save_state()`	Saves the current state to the snapshot file
`_log_event()`	Logs events with timestamps to the log file
`_find_renamed_items()`	Detects renamed files or directories based on metadata comparison
`monitor_changes()`	Compares states and logs detected changes
`start_monitoring()`	Starts the monitoring loop with a defined interval
`stop_monitor()`	Stops the monitoring loop gracefully

This class-based design ensures that the tool remains organized, extensible, and easy to maintain. It also allows for future enhancements such as GUI integration, real-time alerts, or multi-directory support without disrupting the core logic.

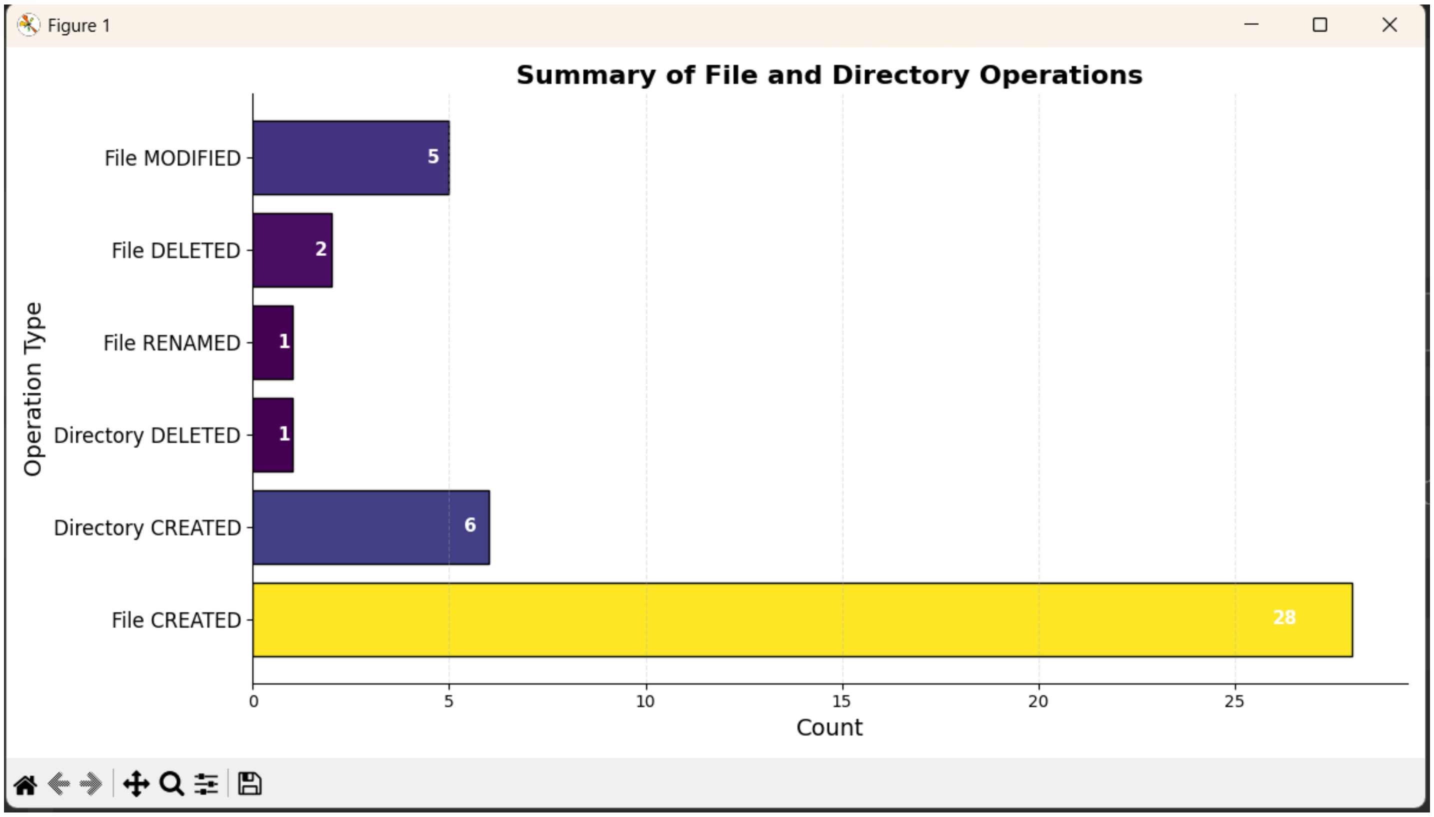
Network and Web Features (if applicable)

This tool currently does not include any network or web features. However, it can be extended in the future to support remote monitoring or integration with other systems over the network, as discussed in the "Future Enhancements and Recommendations" section.

Output Visualization

The tool includes a visualization module using matplotlib to provide a visual summary of file and directory operations. A horizontal bar chart is generated,

illustrating the frequency of different operations (creation, deletion, modification, renaming) and is automatically saved as a PNG image



Future Enhancements and Recommendations

To improve the tool's usability, scalability, and integration with modern systems, the following enhancements are recommended:

Graphical User Interface (GUI)

The tool can be upgraded from a command-line interface to a graphical user interface, making it more user-friendly and accessible. This can be achieved using frameworks such as Tkinter or PyQt. A GUI would allow users to interact with the tool through buttons, menus, and visual panels, reducing the need for manual_command-line input.

Network Integration

The tool can be extended to support remote monitoring using technologies like Sockets or RPC (Remote Procedure Call). This would enable centralized monitoring across multiple devices, allowing administrators to track changes from a single dashboard.

SIEM Integration (Security Information and Event Management) .

To support enterprise-level security, the tool can be integrated with SIEM platforms. This would allow: * Real-time alerts. * Centralized log collection. * Automated incident response. Such integration would make the tool suitable for professional cybersecurity environments.

Behavioral Analysis .

Future versions could include anomaly detection by analyzing behavioral patterns in f ile operations. For example: * Sudden spikes in deletions. * Frequent renaming of sensitive files. * Unusual access times. This would help identify suspicious activity and potential threats.

Cross-Platform Support .

The tool can be adapted to run on multiple operating systems, including: * Windows. * macOS. * Linux. This ensures broader usability and deployment flexibility.

Packaging and Distribution .

To simplify installation and usage, the tool can be packaged as a standalone application or installer. This would make it easier to distribute and deploy in various environments.

Full Recursive Scanning .

The tool can be enhanced to support deep recursive scanning of directories, with configurable depth and filters. This would allow comprehensive monitoring of complex directory structures

State File Optimization and Encryption .

The monitor_states.json file can be optimized for faster read/write operations and better performance during large-scale scans. Encryption can also be added for increased security.

Conclusion .

This project demonstrates the design and implementation of a file and directory monitoring tool using Python. It reflects a strong understanding of programming fundamentals, error handling, and system architecture. The tool was built with a focus on: * **Modularity** through object-oriented design. * **Reliability** via robust error handling. * **Scalability** for future enhancements. * **Responsiveness** using threading. * **Clarity** through terminal and file-based outputs. The DirectoryMonitor class serves as the core of the system, encapsulating all logic related to scanning, comparing, logging, and saving state. The program handles exceptions gracefully and provides clear feedback to the user, making it suitable for real-world use. This project has strengthened my skills in structured programming, system design, and practical cybersecurity applications. It also opened the door to future improvements such as GUI integration, network support, and SIEM compatibility

References .

Python Official Documentation >> https://docs.python.org

Real Python Tutorials >> https://realpython.com

Stack Overflow >> https://stackoverflow.com

GitHub – Code Examples and BesPractices>>https://github.com