Project Report: File-Naming Automation Tool

Section	Details
Project Topic	Develop a versatile File-Naming Automation Tool
Code Language	Python
Key Libraries	os, argparse, datetime
Primary Functionality	Date-based file renaming (from the provided code)

File Naming Conventions: How to Optimize Document Management

Submitted by – SAMEER RAJBHAR
Registration number – 25BAI11033
Date – 24th November 2025

Introduction

- The **File-Naming Automation Tool** is a command-line utility designed to simplify and standardize the process of renaming multiple files within a directory. Managing a large collection of files (e.g., photos, documents, logs) often leads to inconsistent or unhelpful naming conventions. This tool provides multiple, powerful, and automated renaming modes to enhance file organization and retrieval. The current implementation focuses on a **Date-Based Renaming** mode, using file creation or modification timestamps to create human-readable, chronologically ordered file names.

Problem Statement

- Manually renaming a large batch of files to follow a specific, consistent naming convention is **time-consuming, prone to errors**, and **inefficient**. Specifically, deriving chronological information from file metadata (like creation or modification dates) and embedding it into the file name is a complex, repetitive task. The problem is to create a robust and flexible tool that automates these file-naming tasks, especially for chronologically sensitive files like photographs.

Functional Requirements

- **R1: Date-Based Renaming:** The tool must be able to rename files using their creation time or last modification time.

- **R2: Custom Date Format:** Users must be able to specify a custom date/time format using Python's strftime directives.

- **R3: Prefix/Suffix Inclusion:** The tool must allow optional prefixes and suffixes to be added to the generated file name.

- **R4: Conflict Resolution:** The tool must automatically handle filename conflicts (e.g., if multiple files share the same timestamp) by appending an incremental counter.

- **R5: Directory Handling:** The tool must gracefully skip directories during the renaming process.

- **R6: Command-Line Interface (CLI):** All functionality must be accessible via command-line arguments using argparse.

Non-functional Requirements

- **N1: Robustness:** The tool should handle exceptions gracefully (e.g., permission errors, invalid folder paths).
- **N2: Performance:** Renaming should be fast, even for directories containing thousands of files.
- **N3: Usability:** The CLI should be intuitive, providing clear help messages and examples.
- **N4: Extensibility:** The core structure (main function with subparsers) should allow for easy addition of new renaming modes (e.g., sequential, text-replacement) in the future.

System Architecture

The system follows a simple **Monolithic Command-Line Utility Architecture**.

Components:

1. Command-Line Interface (argparse): Acts as the entry point, defining commands (seq, rep, date) and arguments (e.g., folder_path, date_format).
2. **Main Dispatcher (main function):** Parses arguments and directs the request to the appropriate renaming function based on the selected mode.
3. **Renaming Modules (date_rename):** Contains the core logic for a specific renaming task, interacting with the operating system.
4. **OS Interaction (os module):** Handles file system operations such as listing files (os.listdir), getting metadata (os.path.getctime/getmtime), path manipulation (os.path.join), and renaming (os.rename).

Design Diagrams

Use Case Diagram

- This diagram illustrates the primary user goals and how the tool fulfills them.
- **Actor:** User
- **Use Cases (for Date Mode):**
  - Set Target Folder
  - Choose Time Source (Creation/Modification)
  - Specify Date Format
  - Execute Renaming

Workflow Diagram

This illustrates the step-by-step logic within the date_rename function.

1. **Start**
2. **Input:** folder_path, date_format, prefix, suffix, use_modification_time
3. **Choose Timestamp Function:** getmtime (if -m) or getctim
4. **Loop through Files in Folder**
5. **Get Timestamp & Convert:** Convert timestamp to datetime object
6. **Format Date String:** Apply strftime(date_format).
7. **Construct New Name:** Combine prefix, date_str, suffix, and extension.
8. **Check for Conflict:** Is new_path unique?
   - **Yes:** Proceed to rename.
   - **No:** Increment counter and append to suffix, then check again.
9. **Rename File:** os.rename(old_path, new_path)
10. **End Loop**

Sequence Diagram

This focuses on the interaction between the User, the CLI, and the Renaming Logic for the Date mode.

1. **User CLI:** Execute python tool.py date ...
2. **CLI Main Dispatcher:** Return parsed args object.
3. **Main Dispatcher Date Renamer:** Call date_rename(args).
4. **Date Renamer OS:** Get file list.
5. **Date Renamer OS:** Get file time (getctime/getmtime).
6. **Date Renamer OS:** Execute os.rename().
7. **Date Renamer Main Dispatcher:** Return status.
8. **Main Dispatcher User:** Print "Renaming complete."

Design Decisions & Rationale

Decision	**Rationale**
Using argparse for CLI	Standard Python library for robust and user-friendly command-line interfaces. Supports **subparsers**, which is essential for creating multiple distinct renaming modes (e.g., date, seq, rep).
Separation of Concerns (e.g., date_rename)	Keeps the logic for each renaming mode isolated and testable. The main function simply dispatches, adhering to the Single Responsibility Principle.
File-Conflict Resolution Loop	The while os.path.exists(new_path) loop with an incrementing counter ensures **no data is lost** by overwriting existing files, even if two files share the exact same timestamp (down to the second).
Using os.path.getctime vs. os.path.getmtime	Provides flexibility to the user. **Creation time (ctime)** is often the true capture time for photos, while **modification time (mtime)** is better for documents that are edited. The -m flag allows the user to choose.

Implementation Details

The core implementation is done in **Python 3.x**. The date_rename function uses standard library features only, making the tool highly portable.

Key Logic Snippets:

1.Time Source Selection:

Python

```
timestamp_func = os.path.getmtime if use_modification_time else os.path.getctime
```

2.Timestamp Conversion and Formatting:

Python

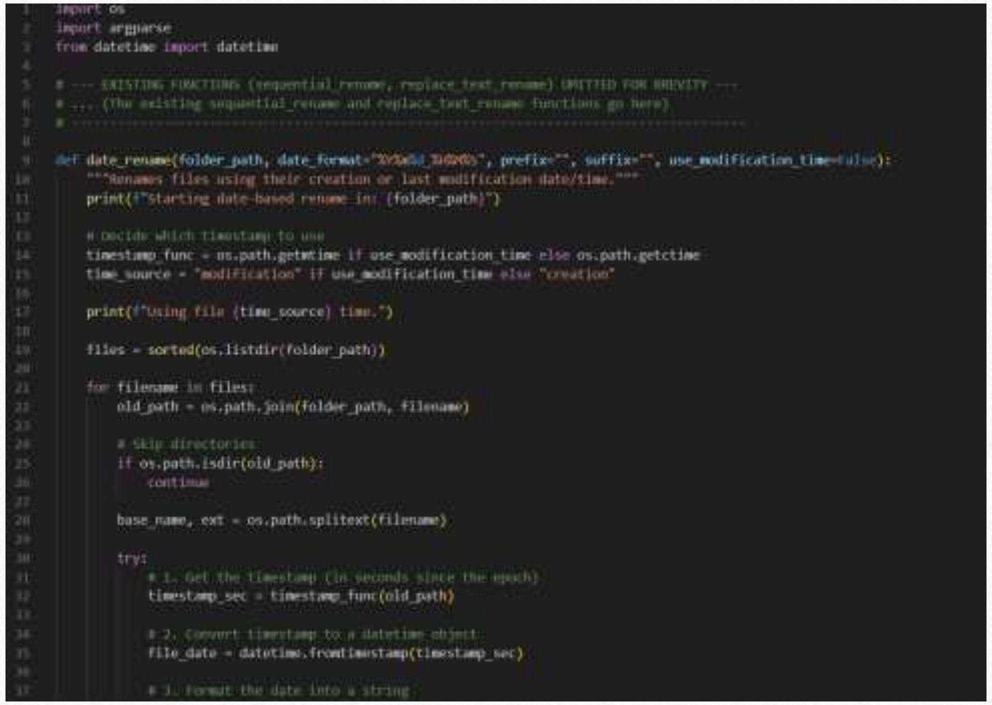
```
timestamp_sec = timestamp_func(old_path) file_date = datetime.fromtimestamp(timestamp_sec) date_str = file_date.strftime(date_format)
```

3.Filename Construction:

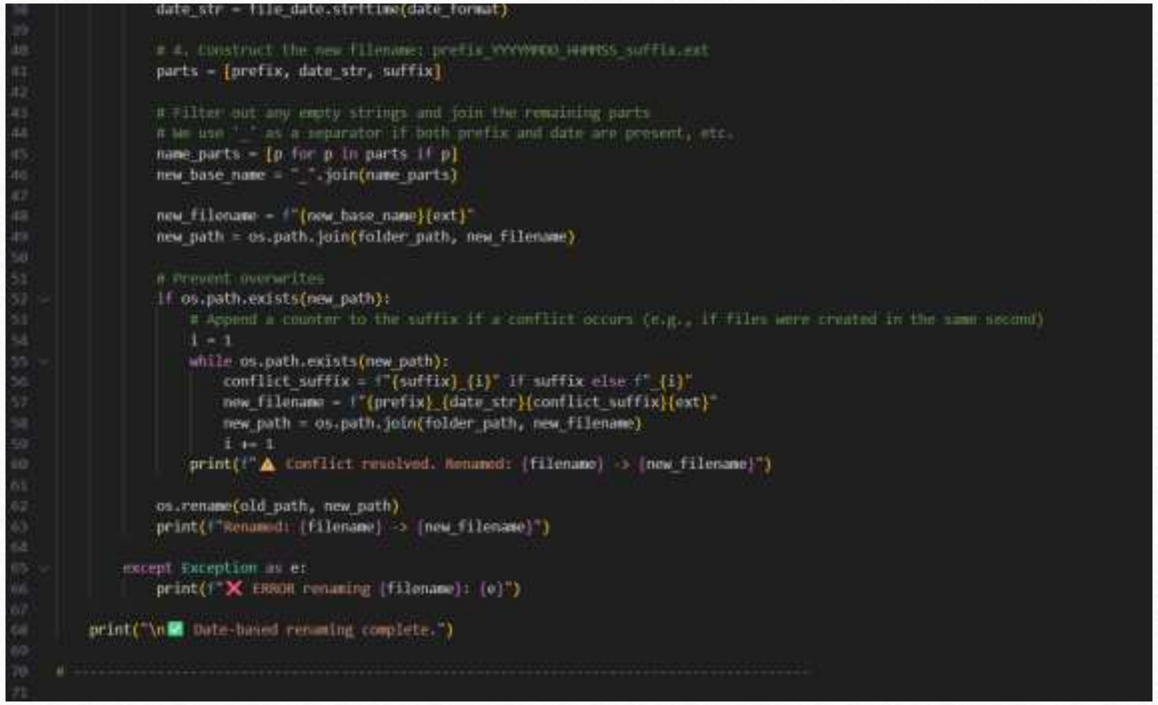
Python

```
name_parts = [p for p in [prefix, date_str, suffix] if p] new_base_name = "_".join(name_parts) new_filename = f"{new_base_name}{ext}"
```

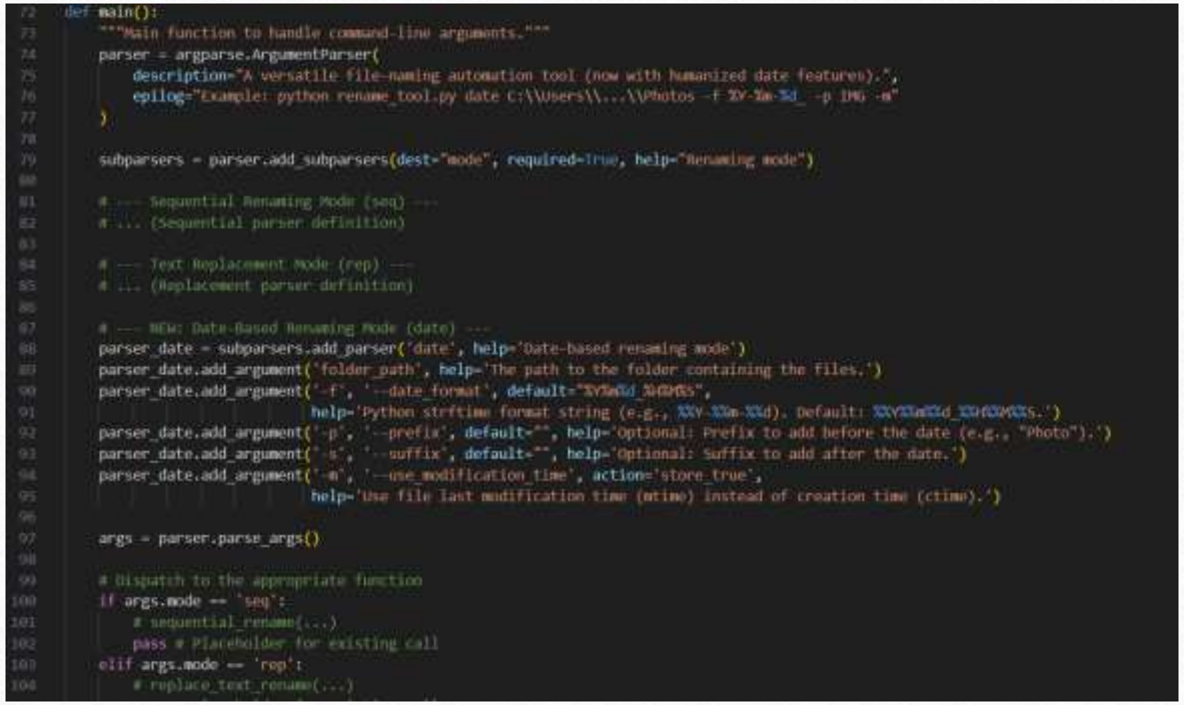
Screenshots/Results



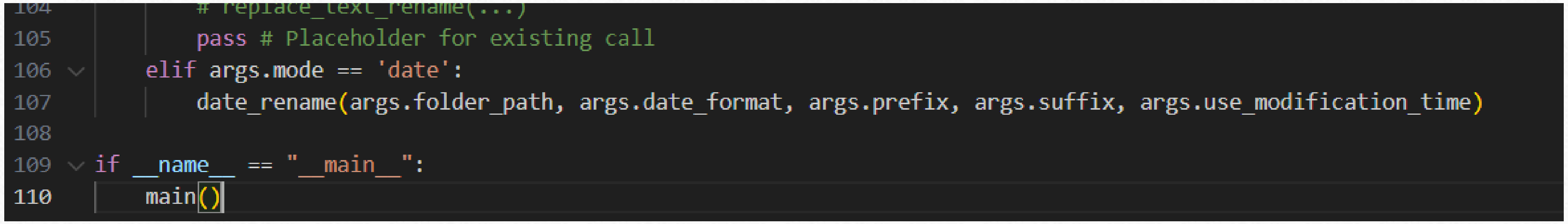
```python
import os
import argparse
from datetime import datetime

# --- EXISTING FUNCTIONS (sequential_rename, replace_text_rename) OMITTED FOR BREVITY ---
# ... (The existing sequential_rename and replace_text_rename functions go here)
# ------------------------------------------------------------------------------------

def date_rename(folder_path, date_format="%Y%m%d_%H%M%S", prefix="", suffix="", use_modification_time=False):
    """Renames files using their creation or last modification date/time."""
    print(f"Starting date-based rename in: {folder_path}")

    # Decide which timestamp to use
    timestamp_func = os.path.getmtime if use_modification_time else os.path.getctime
    time_source = "modification" if use_modification_time else "creation"

    print(f"Using file {time_source} time.")

    files = sorted(os.listdir(folder_path))

    for filename in files:
        old_path = os.path.join(folder_path, filename)

        # Skip directories
        if os.path.isdir(old_path):
            continue

        base_name, ext = os.path.splitext(filename)

        try:
            # 1. Get the timestamp (in seconds since the epoch)
            timestamp_sec = timestamp_func(old_path)

            # 2. Convert timestamp to a datetime object
            file_date = datetime.fromtimestamp(timestamp_sec)

            # 3. Format the date into a string
```



```python
            date_str = file_date.strftime(date_format)

            # 4. Construct the new filename: prefix_YYYYMMDD_HHMMSS_suffix.ext
            parts = [prefix, date_str, suffix]

            # Filter out any empty strings and join the remaining parts
            # We use '_' as a separator if both prefix and date are present, etc.
            name_parts = [p for p in parts if p]
            new_base_name = "_".join(name_parts)

            new_filename = f"{new_base_name}{ext}"
            new_path = os.path.join(folder_path, new_filename)

            # Prevent overwrites
            if os.path.exists(new_path):
                # Append a counter to the suffix if a conflict occurs (e.g., if files were created in the same second)
                i = 1
                while os.path.exists(new_path):
                    conflict_suffix = f"{suffix}_{i}" if suffix else f"_{i}"
                    new_filename = f"{prefix}_{date_str}{conflict_suffix}{ext}"
                    new_path = os.path.join(folder_path, new_filename)
                    i += 1
                print(f"⚠ Conflict resolved. Renamed: {filename} -> {new_filename}")

            os.rename(old_path, new_path)
            print(f"Renamed: {filename} -> {new_filename}")

        except Exception as e:
            print(f"❌ ERROR renaming {filename}: {e}")

    print("\n✅ Date-based renaming complete.")

# ----------------------------------------------------------------------------
```



```python
def main():
    """Main function to handle command-line arguments."""
    parser = argparse.ArgumentParser(
        description="A versatile file-naming automation tool (now with humanized date features).",
        epilog="Example: python rename_tool.py date C:\\Users\\...\\Photos -f %Y-%m-%d_ -p IMG -m"
    )

    subparsers = parser.add_subparsers(dest="mode", required=True, help="Renaming mode")

    # --- Sequential Renaming Mode (seq) ---
    # ... (Sequential parser definition)

    # --- Text Replacement Mode (rep) ---
    # ... (Replacement parser definition)

    # --- NEW: Date-Based Renaming Mode (date) ---
    parser_date = subparsers.add_parser('date', help='Date-based renaming mode')
    parser_date.add_argument('folder_path', help='The path to the folder containing the files.')
    parser_date.add_argument('-f', '--date_format', default="%Y%m%d_%H%M%S",
                             help='Python strftime format string (e.g., %%Y-%%m-%%d). Default: %%Y%%m%%d_%%H%%M%%S.')
    parser_date.add_argument('-p', '--prefix', default="", help='Optional: Prefix to add before the date (e.g., "Photo").')
    parser_date.add_argument('-s', '--suffix', default="", help='Optional: Suffix to add after the date.')
    parser_date.add_argument('-m', '--use_modification_time', action='store_true',
                             help='Use file last modification time (mtime) instead of creation time (ctime).')

    args = parser.parse_args()

    # Dispatch to the appropriate function
    if args.mode == 'seq':
        # sequential_rename(...)
        pass # Placeholder for existing call
    elif args.mode == 'rep':
        # replace_text_rename(...)
```



```
            # replace_text_rename(...)
            pass # Placeholder for existing call
        elif args.mode == 'date':
            date_rename(args.folder_path, args.date_format, args.prefix, args.suffix, args.use_modification_time)

if __name__ == "__main__":
    main()
```

Testing Approach

Due to the nature of file system operations, testing must include real-world scenarios in a **controlled test environment** (a temporary, isolated folder).

- **Unit Tests:** Verify helper functions (e.g., date formatting, path joining).
- **Integration Tests (Black Box):**
  - **Success Case:** Rename a folder of files using default format (%Y%m%d_%H%M%S).
  - **Edge Case 1 (Conflict):** Test the conflict resolution by creating two files with a dummy timestamp that are identical.
  - **Edge Case 2 (Prefix/Suffix):** Test with various combinations of prefix, suffix, and custom date formats (e.g., including spaces or hyphens).
  - **Failure Case:** Attempt to rename files in a read-only directory or with an invalid path to check for graceful error handling (try...except).

Challenges Faced

- **Handling File Conflicts:** The biggest challenge was ensuring robust conflict resolution when two files had timestamps identical down to the second. This was solved by implementing the incrementing counter in a while loop that checks for the existence of the newly constructed path.

- **Date/Time Source Consistency:** Different operating systems (especially Windows vs. Linux/macOS) treat file **creation time (ctime)** differently. Providing the -m flag to explicitly use **modification time (mtime)** was necessary to offer a consistently available and reliable alternative across platforms.

- **Argument Parsing Complexity:** Structuring argparse with subparsers required careful planning to correctly define required arguments per mode, but ultimately resulted in a much cleaner and more user-friendly interface.

Learnings & Key Takeaways

- **Importance of argparse:** Learned how to create professional-grade CLI tools using subparsers for multi-mode functionality.
- **Deep Dive into os Module:** Gained practical understanding of file metadata functions like os.path.getctime and the need to use datetime.fromtimestamp for human-readable dates.
- **Defensive Programming:** The conflict resolution logic highlighted the importance of anticipating and proactively handling rare but critical edge cases in file system operations to prevent data loss or program crashes.

Future Enhancements

•**Full Implementation of Other Modes:** Implement the planned **Sequential Renaming** and **Text Replacement** modes.

•**Recursive Renaming:** Add an option to process files not just in the current folder, but recursively in all subdirectories.

•**File Filtering:** Add options to filter files by extension (e.g., only rename .jpg files).

•**Preview Mode:** Implement a dry-run feature (--dry-run) that prints the old name $\rightarrow$ new name mapping without actually performing the renaming operation.

References

•Python 3 Official Documentation: os module.

•Python 3 Official Documentation: datetime module and strftime() directives.

- Python 3 Official Documentation: argparse **module**.