Documentation

This Python script automates handling and comparing CSV log files stored locally. It renames files to include their modification date/time prefix, groups files by log type, and enables comparing two versions of the same log to find added, removed, or changed rows. It is designed to be beginner-friendly, modular, and easy to maintain or extend.

Table of contents

1. Environment and Setup

Python Version: Compatible with Python 3.x
Dependencies: Only standard libraries (os, csv, pathlib, datetime, re, collections)
Operating System: Cross-platform (tested on Windows, should work on Linux/macOS)

2. Modules and Imports

os: Access file system details such as file modification times.
csv: Read CSV files easily into dictionaries.
pathlib.Path: Cross-platform way to manipulate filesystem paths.
datetime.datetime: Convert timestamps into readable dates and times.
re: Use regular expressions to parse filename patterns.

Provides regular expression (regex) capabilities for pattern matching in strings. Here, it is used to parse date/time and log type from filenames that follow a specific naming pattern.

Regex breakdown:

- `^ - start of string`
- `(\d{4}-\d{2}-\d{2}_\d{2}-\d{2}) - captures date and time in YYYY-MM-DD_HH-MM format`

- `_ - underscore separating date/time from log type`
- `(.+) - captures any characters after underscore until '.csv' (the log type)`
- `.csv$ - ends with ".csv"`

collections.defaultdict: Efficiently group logs by type without manual initialization.

3. Constants and Configuration

```
LOGS_FOLDER = Path("logs")
DELIMITER = ";"
```

LOGS_FOLDER is where your logs should be placed.
DELIMITER matches your CSV format (";" by default).

4. Functions

1. `extract_prefix_and_type(filename)`
   Purpose:
   Extracts the date/time prefix and log type from a given filename.
   Parameters:
   filename (str): The name of the file (e.g. "2024-06-01_15-30_firewall.csv").
   Returns:
   Tuple (date_time, log_type) where both are strings.
   Returns (None, None) if filename doesn't match expected pattern.
   Details:
   Uses regex r"^(\d{4}-\d{2}-\d{2}_\d{2}-\d{2})_(.+)\.csv$" to parse:
   - Date/time: "YYYY-MM-DD_HH-MM"
   - Log type: remainder after date/time, before .csv

2. `list_log_files_by_type()`

   Purpose:
   Scans the LOGS_FOLDER subdirectories for CSV files, grouping them by log type based on subfolder names, and sorts each group by date/time descending.
   Returns:
   Dictionary mapping log types to lists of (date_time, Path) tuples.
   Details:
   Each subfolder name is treated as a log type.
   Gathers CSV files within these subfolders.
   Sorts versions by date/time descending, easy access to latest versions.

3. `load_csv_to_dict(file_path, key_column='id')`

   Purpose:
   Loads a CSV file into a dictionary indexed by the specified `key_column`.

Parameters:
`file_path` (`Path`): Path to the CSV file.
`key_column` (str): The column to use as a unique key (default `'id'`).
Return:
A dictionary: { key_value: {column_name: value, ...}, ... }
Details:
Reads CSV using csv.DictReader with specified delimiter.
Each row becomes a nested dict, keyed by the value in the key_column.

4. `compare_dicts(old_data, new_data)`
   Purpose:
   Compares two dictionaries of CSV rows to identify additions, removals, and changes.
   Parameters:
   old_data (dict): Data from the older CSV version.
   new_data (dict): Data from the newer CSV version.
   Returns:
   Tuple (added, removed, changed) where:
   added = set of keys present only in new_data.
   removed = set of keys present only in old_data.
   changed = dict mapping keys to column-wise value differences.
   Details:
   Compares row-by-row for keys present in both.
   For changed rows, records columns where values differ.
5. `print_comparison_results(added, removed, changed, old_data, new_data)`
   Purpose:
   Prints a readable summary of added, removed, and changed rows with full data.
   Parameters:
   added (set): Keys of rows added in new data.
   removed (set): Keys of rows removed since old data.
   changed (dict): Keys mapped to changed column details.
   old_data (dict): Older CSV data.
   new_data (dict): Newer CSV data.
   Details:
   Prints counts and details of added, removed, and changed rows.
   Shows full old and new CSV lines (joined with delimiter) for context.
   Highlights specific changed columns and their old → new values.

6. `get_modification_date(file_path)`

   Purpose:
   Returns the last modification date/time of a file.
   Parameters:
   file_path (Path): The file to check.
   Returns:
   datetime object representing the file's last modification timestamp.

7. `file_starts_with_date(filename)`
   Purpose:
   Checks if a filename already starts with a date/time prefix to avoid double-renaming.
   Parameters:
   filename (str): Filename to check.
   Returns:
   Boolean: True if filename starts with pattern "YYYY-MM-DD_HH-MM_", else False.

8. `rename_files_with_date_prefix(folder)`

   Purpose:
   Renames files in folder to prepend their modification date/time if missing.
   Parameters:
   folder (Path): Directory containing files to rename.
   Details:
   Skips files that already have a date prefix.
   Uses modification time formatted as "YYYY-MM-DD_HH-MM".
   If renaming causes name collision, appends incrementing counter suffix.
   Prints messages when skipping or renaming files.

9. `organize_files_by_log_type(folder)`

   Purpose:
   Sorts files into subfolders based on log type.
   Parameters:
   folder (Path): Directory containing files to rename.
   Details:
   Creates subfolders (with parents=True for nested dirs).
   Moves files into their respective log-type subfolders.
   Handles file name collisions by adding counters.
   Includes exception handling to avoid crashes on move errors.

10. `main()`

    Purpose:
    Drives the entire program workflow.
    Workflow:
    1. Run renaming to prepend date/time prefix if missing.
    2. Organize files into subfolders by log type.
    3. Scan subfolders for available log types.
    4. List subfolder names (log types) for user to choose.
    5. List versions inside chosen subfolder for user to pick two.
    6. Load selected versions and compare.

7. Print results.
8. Wrap all above in a loop to allow repeated runs until user chooses to exit (e.g., by typing 'q' or pressing CTRL+C).

Error Handling:
Handles invalid selections gracefully.
Prints appropriate messages when no files found or input is invalid.
Catches KeyboardInterrupt to print a friendly exit message.

5. Workflow

1. On execution, the script scans the logs/ folder for all .csv files.
2. Checks if filenames start with a date/time prefix (YYYY-MM-DD_HH-MM_). If missing, it renames files to prepend their last modified timestamp.
3. Extracts log type from filenames and groups files accordingly.
4. Creates subfolders named after each log type (if they don't exist) and moves files into them.
5. Prompts the user to select a log type from available subfolders.
6. Displays versions inside the selected subfolder, sorted newest to oldest.
7. Prompts user to pick two versions for comparison.
8. Loads both CSV files into dictionaries keyed by the unique column (id by default).
9. Compares the dictionaries to find added, removed, and changed rows.
10. Prints a clear, detailed summary of differences.
11. Repeats the process until the user types 'q' or presses CTRL+C.

6. How to Use

Place your CSV logs in `logs/` folder.
Run the script: `python test-version1.py`
Follow prompts to pick log type and versions.
View output directly in the console.

7. Extending and Customization

Change CSV delimiter by editing the DELIMITER constant.
Change key column by passing different key_column to load_csv_to_dict().
Update filename regex in extract_prefix_and_type() and file_starts_with_date() for different naming schemes.
Add functionality to save comparison output to file.
Add logging or exception handling for robustness.

8. Troubleshooting and Notes

Ensure CSV files have a consistent structure with the key column present.
File modification time depends on your OS and may change if files are copied or edited.
Regex patterns are strict; filenames must conform to expected format.

9. User Interaction and Terminal Usage

Pressing CTRL+C in the terminal raises a KeyboardInterrupt and immediately terminates the program.

To safely copy text from the terminal on Windows without stopping the program:

> Right-click the terminal title bar → Edit → Mark, select text, then right-click → Copy.

In some terminals (e.g., VS Code, Git Bash), use CTRL+Shift+C or right-click → Copy.