Log Monitoring Application:

This document provides a detailed, line-by-line explanation of the provided Python script for processing a CSV-based job log. The code follows a **Test-Driven Development** structure, with clear functions for parsing, processing, and reporting.

---

1. Overview

The script is a **Job Log Processor** designed to parse a CSV log file containing job events, calculate their durations, and identify common logging anomalies like long-running or mismatched jobs. It uses Python's built-in `logging` module to provide clear feedback and the `datetime` module for precise time calculations.

---

2. Requirements

The script is built to satisfy the following requirements:

- A job must have both a **START** and **END** event to be considered complete.
- **WARNING** if a job duration exceeds **5 minutes**.
- **ERROR** if a job duration exceeds **10 minutes**.
- Log jobs that started but never ended as "no-ending jobs."
- Log **END** events without a **START** as "orphan jobs."

---

3. Code with Explanations

`import` statements and Logging Configuration:

- `import csv`: Imports the `csv` module to handle CSV file reading.
- `from datetime import datetime, timedelta`: Imports `datetime` for timestamps and `timedelta` for time differences.
- `import logging`: Imports the `logging` module for structured message logging.
- `logging.basicConfig(...)`: Configures the basic settings for the logging system.
- `level=logging.INFO`: Sets the logging level to `INFO`, meaning all messages with a severity of `INFO` and higher (like `WARNING`, `ERROR`) will be displayed.
- `format='%(levelname)s: %(message)s'`: Defines the format for log messages, showing the log level (e.g., 'WARNING') followed by the message.

Thresholds:

- `WARNING_THRESHOLD = timedelta(minutes=5)`: Defines a constant `timedelta` object representing a 5-minute duration for a warning threshold.
- `ERROR_THRESHOLD = timedelta(minutes=10)`: Defines a constant `timedelta` object representing a

10-minute duration for an error threshold.

parse_log_file(file_path) function:

- def parse_log_file(file_path):: Defines a function to parse the log file, taking the file path as an argument.
- job_events = []: Initializes an empty list to store the parsed job events.
- with open(file_path, 'r') as file:: Opens the specified file in read mode ('r') and ensures it is automatically closed using a with statement.
- reader = csv.reader(file): Creates a CSV reader object to iterate over rows in the file.
- for row in reader:: Loops through each row in the CSV file.
- if len(row) != 4:: Checks if the current row has exactly 4 fields.
- continue: Skips the current iteration if the row is malformed.
- timestamp_str, description, status, pid = map(str.strip, row): Unpacks the row fields into variables and removes any leading/trailing whitespace.
- timestamp = datetime.strptime(timestamp_str, '%H:%M:%S'): Converts the timestamp string into a datetime object using the specified time format.
- job_events.append(...): Appends a new dictionary to the job_events list.
- 'timestamp': timestamp: Stores the datetime object.
- 'description': description: Stores the job description.
- 'status': status: Stores the job status (e.g., 'START' or 'END').
- 'pid': pid: Stores the process ID.
- return job_events: Returns the list containing all the parsed job events.

process_jobs(events) function:

- def process_jobs(events):: Defines a function to process the list of events, taking the list as an argument.
- active_jobs = {}: Initializes a dictionary to track jobs that have started but not yet ended, using the PID as the key.
- completed_jobs = []: Initializes a list to store jobs that have both a START and an END event.
- no_ending_jobs = []: Initializes a list for jobs that started but never ended.
- end_without_start_jobs = []: Initializes a list for END events that have no matching START.
- for event in events:: Loops through each event dictionary in the input list.
- pid = event['pid']: Extracts the process ID from the current event.
- if event['status'] == 'START':: Checks if the event status is 'START'.
- active_jobs[pid] = event: Adds the event to the active_jobs dictionary, keyed by PID.
- elif event['status'] == 'END':: Checks if the event status is 'END'.
- if pid in active_jobs:: Checks if a START event for this PID has been recorded.
- start_event = active_jobs.pop(pid): Removes the start event from active_jobs and gets its value.
- duration = event['timestamp'] - start_event['timestamp']: Calculates the duration by subtracting the start time from the end time.
- completed_jobs.append(...): Appends a new dictionary of job information to the completed_jobs list.
- 'pid': pid: Includes the process ID.
- 'description': start_event['description']: Includes the job description from the start event.
- 'start_time': start_event['timestamp']: Includes the job start time.

- 'end_time': event['timestamp']: Includes the job end time.
- 'duration': duration: Includes the calculated duration.
- if duration > ERROR_THRESHOLD:: Checks if the duration exceeds the 10-minute error threshold.
- logging.error(...): Logs an error message.
- elif duration > WARNING_THRESHOLD:: Checks if the duration exceeds the 5-minute warning threshold.
- logging.warning(...): Logs a warning message.
- else:: This block runs if an END event is found without a corresponding START in active_jobs.
- logging.warning(...): Logs a warning for an orphan END event.
- end_without_start_jobs.append(...): Appends the orphan END event information to a separate list.
- latest_timestamp = max(...) if events else datetime.now(): Finds the latest timestamp in the log file, or uses the current time if the list is empty.
- for pid, job in active_jobs.items():: Loops through all remaining jobs in the active_jobs dictionary.
- duration = latest_timestamp - job['timestamp']: Calculates the duration of the unfinished job based on the latest log timestamp.
- no_ending_jobs.append(...): Appends the unfinished job information to the no_ending_jobs list.
- if duration > ERROR_THRESHOLD:: Checks if the duration of the unfinished job exceeds the error threshold.
- logging.error(...): Logs an error message for an unfinished job exceeding the time limit.
- elif duration > WARNING_THRESHOLD:: Checks if the duration exceeds the warning threshold.
- logging.warning(...): Logs a warning for an unfinished job exceeding the time limit.
- return completed_jobs, no_ending_jobs, end_without_start_jobs: Returns the three lists containing the categorized jobs.

generate_report(...) function:

- def generate_report(...): Defines a function to print a summary report.
- print(...): Prints a header for the completed jobs section.
- for job in completed_jobs:: Loops through the list of completed jobs.
- print(f"..."): Prints the details of each completed job.
- if no_ending_jobs:: Checks if there are any jobs with no ending event.
- print(...): Prints a header for the no-ending jobs section.
- for job in no_ending_jobs:: Loops through the list of no-ending jobs.
- print(f"..."): Prints the details of each no-ending job.
- if end_without_start_jobs:: Checks if there are any orphan END events.
- print(...): Prints a header for the orphan events section.
- for job in end_without_start_jobs:: Loops through the list of orphan events.
- print(f"..."): Prints the details of each orphan event.

Main Entry Point (if __name__ == "__main__":):

- if __name__ == "__main__":: This block of code runs only when the script is executed directly.
- log_file = "logs.log.csv": Sets the variable for the log file path.
- events = parse_log_file(log_file): Calls the parsing function and stores the result.
- completed_jobs, no_ending_jobs, end_without_start_jobs = process_jobs(events): Calls the processing function and unpacks the three returned lists into separate variables.

- `generate_report(completed_jobs, no_ending_jobs, end_without_start_jobs)`: Calls the report generation function with the processed lists.

4. Conclusion

The Log Monitoring Application script, developed with a Test-Driven Development approach, provides a robust and reliable solution for analyzing job logs. By systematically parsing log events, the script accurately identifies job statuses, calculates durations, and flags anomalies such as orphan events and no-ending jobs. The use of Python's logging module ensures that warnings and errors for jobs exceeding predefined time thresholds are clearly documented, aiding in proactive system maintenance and debugging. This structured methodology guarantees that the tool is not only functional but also maintainable and easily adaptable to future requirements, making it a valuable asset for monitoring and managing automated processes.