**Overview B2-1**

This documentation provides an explanation of the Python script that performs automated testing and validation on a dataset using a modular approach. The script is designed to run checks and validations for different scenarios and generate logs for both failed and passed cases. This document aims to help users understand the purpose, structure, and functionality of the script.

**2. Script Overview**

The script is structured in a modular manner, with different functions responsible for specific checks and validations. The main components of the script include:

* Scenario-specific check functions (**run\_all\_checks\_S1**, **run\_all\_checks\_S2**, ...)
* Utility functions for performing individual checks.
* A scenario handler function (**senario\_handler**) to determine which checks to execute based on the scenario and dataset characteristics.
* Logging mechanisms to record both failed and passed checks.
* Output to Excel files for further analysis and reporting

**3. Usage**

To use the script, follow these steps:

1. Import the required libraries.
2. Ensure all files are in the same folder as the program.
3. Set the desired **scenario** based on the testing scenario you want to run.
4. Run the script.

**4. Function Descriptions**

The script includes several key functions:

* **run\_all\_checks\_S1**, **run\_all\_checks\_S2**, ...: These functions encompass a set of checks corresponding to different scenarios. They call specific utility functions based on the scenario.
* Scenario-specific utility functions: These functions implement individual checks for each scenario, such as dwelling checks, connection checks, and NRT (Non-Revenue Time) checks.
* **senario\_handler**: This function determines which set of checks to run based on the specified scenario. It also handles the decision to use a fallback version of checks if the dataset size is insufficient.
* Logging functions: The script generates logs for both compliance and non-compliance cases, which are stored in separate data frames.

High-level overview of checks:

1. **check\_if\_selected\_category\_dwells\_on\_station\_based\_on\_icon\_S3\_S4**: This function checks if a selected category (e.g., "Cross Icon" or "Table") dwells on a specific station based on an icon. It takes parameters such as the DataFrame of timetable data, the icon to check, the expected icon state, the station name, a time threshold, and the scenario. It logs whether the check passed or failed.
2. **check\_last\_value\_in\_range\_S3\_S4**: This function validates the last value within a range (e.g., "Arrival Time" or "Departure Time") for a specific direction (e.g., "Inbound" or "Outbound"). It takes the DataFrame, the value type to check, the direction, and a time threshold. It logs whether the last value is within the specified range.
3. **max\_runtime\_for\_train\_S3\_S4**: This function checks the maximum runtime for a train between two specific stations. It takes the DataFrame, the departure station, arrival station, a maximum runtime threshold, and the scenario. It logs whether the train's runtime is within the specified limit.
4. **connection\_check\_for\_dwell\_S3\_S4**: This function checks if connections between stations adhere to a specified dwell time. It takes the dwell time threshold and the scenario as parameters. It logs whether connections meet the dwell time criteria.
5. **connection\_time\_check\_for\_NRT\_S3\_S4**: This function checks if connection times for non-revenue trains (NRT) between stations meet the specified criteria. It takes the minimum and maximum connection times, and the scenario as parameters. It logs whether the NRT connection times are within the specified limits.
6. **nrt\_check\_S3\_S4**: This function validates the presence of non-revenue trains based on criteria such as icon type, direction, and bound. It takes parameters like the criteria dictionary, icon type, icon state, direction, bound, and the scenario. It logs whether non-revenue trains meet the specified criteria.

The S3\_S4 are for scenario the terminal function (one that creates output) will have this at the end of each name.

**5. Log Generation**

The script generates logs to track the results of the validation process. Logs are categorized into two data frames:

* **non\_compliance\_df**: Contains logs for failed checks.
* **compliance\_df**: Contains logs for passed checks.

These logs provide a clear overview of the dataset's compliance with the predefined criteria.

**6. Output**

The script outputs two Excel files:

* **Compliance Log and Non Compliance Log.xlsx**: Contains the compliance and non-compliance logs in separate sheets.
* Console output: Displays the failed cases and passed cases logs.

**7. Conclusion**

The automated testing script provides a systematic approach to validate a dataset against predefined criteria. By using scenarios and modular functions, the script can easily adapt to different testing requirements. The generated logs and Excel files offer valuable insights into the dataset's quality and help identify areas that need improvement.

**Helpful Tips for Using the Automated Testing Script**

1. **Understanding Scenarios:** Before running the script, make sure you understand the different testing scenarios available (S1, S2, S3, S4). Each scenario corresponds to a specific set of checks designed for different types of data validation.
2. **Scenario Selection:** Choose the appropriate scenario based on the nature of your dataset and the validation checks you want to run. Consider both the content of your dataset (the timetable) weekday, weekend, .1,.2. Split timetable if it contains both weekday or weekend.
3. **Handling Incomplete Data:** If your connection table is incomplete (less than 20 lines of data), the script will automatically run a fallback version of checks. Only the checks that can be made without the connection table.
4. **Reviewing Logs:** After running the script, review the generated logs. The "Compliance Log" sheet will show the cases that passed the checks, while the "Non Compliance Log" sheet will display the cases that failed.
5. **Excel Output:** The script generates an Excel file named **Compliance Log and Non Compliance Log.xlsx** containing the logs. This file can be valuable for reporting and analysis. This file should appear in the same folder as where the program is.
6. **Customization:** Feel free to customize the script to match your specific validation requirements. You can add, modify, or remove checks based on need.
7. **Script Performance:** The script's runtime can vary based on the size of your dataset and the number of checks. Be patient when running the script on larger datasets. It should take 5-20 seconds to run.
8. **Error Handling:** The script includes basic error handling, but it's a good practice to ensure your dataset and input parameters are correctly configured to avoid unexpected errors. This is mostly when the check criteria is changed within the code.
9. **Continuous Improvement:** As your dataset and validation requirements evolve, consider updating and refining the script to ensure it remains effective and aligned with your goals.

**1. Architecture:**

The program follows a modular and structured architecture to enhance maintainability and scalability. It consists of the following components:

* **Main Script (Rev\_1.py):** This script acts as the entry point of the program. It orchestrates the execution of various validation checks based on the provided input data.
* **Utility Functions:** The program includes a set of utility functions responsible for performing individual validation checks. These functions are organized in a logical manner to ensure clarity and ease of debugging.
* **Input Files:** The program requires two main input files: **Inputexcel.xlsx** (Excel file containing data to validate) and **timetable.txt** (text file containing timetable information).
* **Output File:** The program generates an Excel file named **Compliance Log and Non Compliance Log b2-1 version.xlsx**. This file contains logs of checks performed and their outcomes.

**2. Technical Details:**

* **Libraries and Modules:** The program leverages several Python libraries and modules to achieve its functionality, such as Pandas for data manipulation, openpyxl for Excel file handling, and time for performance tracking.
* **Data Parsing and Processing:** The program reads data from the input Excel file and timetable text file. It processes the data into data structures for validation checks.
* **Validation Checks:** The utility functions in the program execute validation checks based on predefined criteria. These checks range from verifying data integrity to enforcing specific conditions.
* **Logging:** The program generates detailed logs for each validation check, recording whether the checks passed or failed. These logs are organized into two categories: compliance and non-compliance.
* **Excel File Generation:** The program generates an Excel output file to provide a structured view of the validation check results. It uses the Pandas library to create Excel sheets for compliance and non-compliance logs.

**3. Benefits and Use Cases:**

* **Modular Approach:** The use of modular utility functions promotes code reusability and simplifies maintenance and debugging.
* **Scalability:** The program can easily accommodate additional validation checks by adding new utility functions to the architecture.
* **Customizable Criteria:** The program can be tailored to different scenarios by modifying the validation criteria within the utility functions.
* **Automation:** The program's automated validation checks save time and effort by swiftly identifying compliance issues within large datasets.
* **Data Integrity:** By enforcing strict validation, the program enhances data integrity and reduces the risk of errors in downstream processes.

**Next Steps and Improvements**

While the current version of the program successfully performs data validation checks, there are always opportunities for enhancement and refinement. Here are some potential next steps and improvements that could be considered:

1. **User-Friendly Interface:** Create a user-friendly graphical interface (GUI) that allows users to interact with the program without needing to run it through the command line. This could include options for selecting input files, specifying scenarios, and viewing the validation results.
2. **Error Handling and Reporting:** Enhance error handling mechanisms to provide more informative error messages when unexpected issues arise. This will help users troubleshoot problems more effectively.
3. **Automated Testing:** Develop a suite of automated tests to validate the correctness of the program's functionality. Unit tests can be created for each utility function and integration tests for the overall program flow. A skeleton has been started and function validation is implemented, the automation of these tests are not. (Functions return True when done)
4. **Logging Enhancements:** Improve the logging system to include timestamps, error details, and severity levels. This will provide a more comprehensive view of the validation process and aid in debugging.
5. **Customizable Rules dashboard:** Implement a rules dashboard in excel that allows users to define their own validation rules and criteria. This would provide greater flexibility and adaptability to different use cases.
6. **Parallel Processing:** Explore the possibility of parallelizing the validation checks to improve performance, especially when dealing with large datasets.
7. **Code Refactoring:** Continuously refactor the codebase to enhance readability, maintainability, and adherence to coding best practices. This will make the program more accessible to other developers and future maintainers.
8. **Interactive Reports:** Generate interactive reports that visualize the validation results using graphs and charts. This can provide a clearer overview of compliance and non-compliance trends.

**Installation and Execution Guide**

**Important:** Before running the program, ensure that all required modules are downloaded. You can find a comprehensive list of required modules in the **requirements.txt** file.

**Steps:**

1. **Open Command Prompt:**
   * Press Win + R.
   * Type "cmd" and press Enter.
2. **Navigate to Project Directory:**
   * Use the **cd** command to navigate to the directory where your **requirements.txt** file is located.
   * Example: **cd path\to\your\project**
3. **Install Packages:**
   * Run the following command to install the packages listed in the **requirements.txt** file:

Copy code:

pip install -r requirements.txt

1. **Run the Program:**
   * Once all required packages are installed, you can execute the program.
   * You can either open the program in vscode and run or
   * Use the following command to run the main script (**Rev\_1.py**):

Copy code

python Rev\_1.py

**Files Needed:**

* **Requirements.txt**: Contains a list of required modules and their versions.
* **Rev\_1.py**: The main Python script that performs data validation checks.
* **Inputexcel.xlsx**: Excel file containing input data.
* **timetable.txt**: Text file containing timetable information.

**Output:**

The program generates an Excel file named **Compliance Log and Non Compliance Log b2-1 version.xlsx**. This file will contain two sheets: "Compliance Log" and "Non Compliance Log", each displaying logs of checks that passed or failed.

**Note:**

* All input files (**Inputexcel.xlsx** and **timetable.txt**) must be located in the same folder as the Python script.
* The output Excel file (**Compliance Log and Non Compliance Log b2-1 version.xlsx**) will also appear in the same folder.

By following these steps, you can easily set up and run the program to perform validation checks on your input data.