Task summary

The app represents a data management program that reads JSON data from a file and represents it in Excel format in 3 data layers.

How to compile: python3 data_manager.py
Packages to install: pandas, openpyxl, dotenv
Input: the input file path can be set in ".env" configuration file. By default: "input.json" file in the current directory.
The input file is expected to contain rows representing JSON objects.
Output: the path can be set in the ".env" configuration file. By default: "output.xlsx".

Layer 1
This part represents "Raw" sub-layer of "Staging" data layer. The data is stored "as-is", without flattening.

1.1 Pandas DataFrame "df" representing data is created

1.2 The DataFrame is exported to Excel ("layer1" sheet)

Note:

In the current implementation the output data is *replaced* with each new start of a program.

Layer 2
Data *standardization* starts. Now the data is cleansed: duplicate rows are deleted.
Two rows are considered duplicate if the values of subset ["@timestamp", "agent", "application"] are equal.

2.1 As there are nested objects in data, and we need to remove duplicated rows using data from fields inside the nested objects, the data first needs to be flattened (*df_normalized* – the DataFrame after flattening)

2.2 In the raw data some of the agent.name fields include lowercase "x" and some – uppercase "X". I decided that subsets that differ only by case in this field should be considered equal, and because of that transformed values to one (upper) case before removing duplicates

2.3 Then duplicates are removed and the result is written to the file ("layer2" sheet)

Additional considerations:

a) I decided to keep all the data, even if one of the key fields (for example, "@timestamp") is Null. It's done to prevent data loss. Other possible strategies: *drop* the data row if any of the "key" column data is missing; *inputing*: replace the missing data when possible (for example, the application name can be deduced from the log file pathname)

b) I suppose that "application" and "@timestamp" data elements are created automatically, so the format is already correct. If needed, on this step the format of the data can be checked (for example, that the string representing the timestamp is really a timestamp and, for example, that this timestamp is before the current point in time). It can be done with Pandas conversion functions ('to_datetime', ...)

c) Also during this step, the names of the columns can be changed to more verbose ones that will be more understandable to a data consumer. But in order to do it, more information about the problem being solved is needed.

d) I've decided to keep all data columns. Depending on the needs of data consumers, some of the columns with low-level information can be removed.

Layer 3
The "layer3" sheet contains a summary with aggregated data: total number of data rows, number of data rows after removal, and number of rows with specific "process.Severity" and "application".

The data is divided: grouped by logging level ("processed.Severity").
In the example data there are entries only of "INFO" and "WARN" logging levels, but data of any other level will also be processed.

3.1 The flattened data is grouped by "processed.Severity"
3.2 The program iterates through groups
3.3 Group data is sorted – primary by timestamp, secondary – by application
3.4 The group data is exported to a separate labeled Excel sheet

Considerations:

a) I decided to separate data of different logging levels into separate labeled Excel sheets, to allow easier analysis. Depending on the logging level, this data will probably be analyzed by different people (or by the same people in different points in time).

b) I sorted the data in each table, from most recent to least recent, in alphabetical order for the same timestamp. It can be done in different logic depending on the requirements.

Additional considerations:

a) Some of the "processed.message" data elements contain log data that can be parsed in the future.