PharmaSUG 2024 - Paper SD- 365

ReadLog Utility: Python Based Log Tool and the First Step of a Comprehensive QC System

Zhihao (Haythem) Luo, Vertex Pharmaceuticals

ABSTRACT

Log checking is a crucial part of SAS programming quality control. To make this process smoother and more robust, I have developed a python-based log checking and summarization tool to provide a better alternative for different user scenarios. The tool provides an improved graphical user interface compared to traditional log reviewing method; it can highlight all important log messages that the user feeds into the application in batches; it creates a summary for different programming folders or even all programming folders under a reporting effort; it also serves as an input for the comprehensive QC tool that is currently in development.

This paper is a journal of how the ReadLog Tool was developed: I will talk about the thought process behind the planning of this tool, the programming logic of how this tool works, how different components were developed to achieve various functionalities, and an in-depth look at some of the Python codes used along the way. I will also introduce my current plan of a comprehensive QC tool, which will be a programming tracker that would integrate log checking, validation status, metadata and programs crosschecking, changes monitoring, etc. all within one application.

INTRODUCTION

When thinking about SAS programming, writing the SAS codes itself is only the beginning of the job: Log Reviewing and Quality Control (mainly validation with double programming) are the other two key components in SAS programing development work. Without a proper method for SAS log review, this process can be time consuming and tedious. The current process we use involves a mix of a SAS log summary macros (which would process all .log files within a folder, look for certain keywords, and output a summary for that folder) and then perform manual review of the .log files that is flagged by the macro. The current process is not very efficient, nor is it user-friendly, and would require rerun of the macro every time there is update to any log file in the folder. Therefore, I have started an initiative to develop a Python based tool to improve this process.

FROM THEORY TO PRACTICE – THE FIRST WORKING DEMO

As it was probably the first Python Application project in the statistical programming department in our company, I have broken down this project into several stages, and started development from its core function – a “log navigation” tool: the tool will replace the current process of reviewing the .log file, which uses plain text notepad and search for keyword identified by the SAS Macro. The very first version of this tool consists only the basic functionalities: it can load in a .log file, check its contents against a list of pre-defined keywords, provide a summary with highlights, and user can navigate through the log files by selecting a line in the summary panel. With this demo, I will be able to evaluate how well a Python based application would work in our SAS server, how everyone feels about this idea, and how much of an improvement compared to the traditional way of log reviewing.

The initial demo program consists of four components: a keyword file in csv format, a function to load the keyword file, functions to compare a log file with the keywords, and a graphical user interface.

INITIATING A NEW PYTHON PROJECT

When constructing a new Python project, a good habit is to first draft a pseudo structure of the program by defining key objects and functions before writing any actual codes. Table 1 is a quick list of what objects and functions we need for this program:

To do/To process	Functions	Python Objects
Keywords CSV file	• To read the keywords CSV file and store as a python object.	• A list object to store the contents from the CSV file.
Target .log file	• To create one method for .log file selection. • To load in the .log file.	• A plain text object to store the full log file loaded.
Keyword matching	• To compare with a list of exclusion keywords and skip these rows. • To process with the 4 level of keywords, and mark with the highest level.	• A list of keyword lines found with the matching keywords and attached with severity level and color.
Prepare for display	• To create a summary line for the counts of different level of messages found. • To navigate to the selected log line in the full log panel when clicked on the summary panel.	• A list of keyword lines, with proper color attached to each line for display purpose. • A single line summary.

Table 1. List of High-Level Tasks, Functions and Objects

Then we can come up with pseudo codes below:

```
import csv
import tkinter as tk
# Import all other related libraries

class LogFileAnalyzer:
    def __init__(self, master):
        # define class variables here
        self.keywords = [] # keyword loaded from Keywords CSV file
        self.text_box = tk.Text # text box for full .log file to be displayed on right panel
        self.keyword_lines = [] # selected log lines that have a match with any level of keywords found
                                from CSV file, for back-end processes
        self.keyword_listbox = tk.Listbox # a list of text with keywords to be display on the left panel
        self.summary_label = tk.Label # a single line summary for counts of different level of log
                                      messages found

    def load_keywords(self):
        # Load keywords from CSV file, run during initiation

    def open_log_file(self):
        # Open a log file and analyze it
        # This function should have a pop-up window for .log file selection
        # Check log lines against the keyword objects, and create objects for display
        # Call check_exclude and check_keywords

    def check_exclude(self, line):
        # Check if the line should be excluded based on 'Exclude' severity

    def check_keywords(self, line):
        # Check if the line matches any keyword

    def navigate_to_line(self, event):
```

```
        # Navigate to the selected line in full log panel

    def update_keyword_counts(self):
        # Update keyword counts in summary label

    def run(self):
        # Start the main event loop
        self.master.mainloop()

if __name__ == '__main__':
    root = tk.Tk()
    analyzer = LogFileAnalyzer(root)
    # Code to Check whether the program was open with a .log file

    analyzer.run()
```

Now that we have a plan, we will need to put in the pieces one by one. We will go through the development of each function step by step.

KEYWORD FILE AND LOADING FUNCTION

The keyword file is a CSV file with four columns: message level, keywords, case indicator, and color. I have defined five levels for the log messages in this file:

Critical: log messages that indicates there is an error which stops the execution of the SAS code, or log messages that indicates there is a high possibility of altered outcome.

Warning: log message that indicates there is a possibility of altered outcome.

Review: custom log messages for various purposes.

Information: log messages generated by in-house macros for review.

Exclude: keywords to exclude unintendedly flagged messages.

Here is a screenshot of a reduced keyword CSV file:

	A	B	C	D
1	severity	keyword	case_sensitive	color
2	Critical	(Critical log message that would terminate SAS execution or highly possibly affect the outcome of the program, required to be fixed)	Y	red
3	Critical	ERROR:	Y	red
4	Critical	FATAL	Y	red
5	Critical	NOTE: The SAS System stopped	Y	red
6	Critical	NOTE: DATA STEP stopped due to looping	y	red
7	Critical	NOTE: Division by zero	y	red
8	Warning	(Warning log message that has possibility of impact on output and should be fixed)	Y	darkorange4
9	Warning	is uninitialized	n	darkorange4
10	Warning	NOTE: MERGE statement has more than one	y	darkorange4
11	Warning	NOTE: Numeric values have been converted to character	y	darkorange4
12	Warning	NOTE: Character values have been converted to numeric	y	darkorange4
13	Review	(For custom queries coming from Macro or user defined checks that would require manual review efford)	Y	blue
14	Review	Review_Duplicate:	n	blue
15	Review	Review_SpecialCharacter:	n	blue
16	Review	Review_Reminder:	n	blue
17	Information	(For custom log massge that is informational only, lowest level of check)	y	green
18	Information	FYI(	y	green
19	Exclude	(any log message identify with below keyword would be excluded from summary regardless of message leve)	y	black
20	Exclude	put warning	n	black
21	Exclude	put error	n	black
22	Exclude	%put err	n	black

Figure 1. Keywords CSV file

The first step of setting up this program is to load in the keyword CSV file. To do so in Python, we would develop the `load_keywords` function to process the file and turn them into Python “dataset”. Here is the function to import the CSV file:

```
    def load_keywords(self):
        # Load keywords from the CSV file
        keywords = []
```

```
        print("Enter keyword loading")
        csv_file_path = 'c:/temp/keywords.csv'

        if os.path.exists(csv_file_path):
            with open(csv_file_path, 'r') as csv_file:
                reader = csv.reader(csv_file)
                next(reader) # Skip the first row
                for row in reader:
                    severity = row[0].strip().lower()
                    keyword = row[1].strip()
                    case_sensitive = row[2].strip().lower() == 'y'
                    color = row[3].strip()

                    keywords.append({
                        'severity': severity,
                        'keyword': keyword,
                        'case_sensitive': case_sensitive,
                        'color': color
                    })
        print(keywords)
        print("Exit keyword loading")
        return keywords
```

We first define “keywords” to be a “list” object, as well as the path to the keywords file. Next, the `os.path.exists(csv_file_path)` function would verify the existence of the file, and the `csv.reader(csv_file)` function would load in the CSV file and turn it into a “list of lists”; then we will store each line into a “list of dictionaries” object, which is an object equivalent to a table with 4 columns (severity, keyword, case_sensitive, color). We will be able to verify the imported contents with this `print(keywords)` function. This is how this list of dictionaries looks like in Python,

```
[ …,
{'severity': 'critical', 'keyword': 'ERROR:', 'case_sensitive': True, 'color': 'red'},
{'severity': 'critical', 'keyword': FATAL', 'case_sensitive': True, 'color': 'red'},
… ,
{'severity': 'warning', 'keyword': 'is uninitialized', 'case_sensitive': False, 'color': 'darkorange4'},
… ,
{'severity': 'warning', 'keyword': 'NOTE: MERGE statement has more than one', 'case_sensitive': True, 'color': 'darkorange4'},
{'severity': 'warning', 'keyword': 'NOTE: Numeric values have been converted to character', 'case_sensitive': True, 'color':
'darkorange4'},
…,
{'severity': 'review', 'keyword': 'Review_Duplicate:', 'case_sensitive': False, 'color': 'blue'},
{'severity': 'review', 'keyword': 'Review_SpecialCharacter:', 'case_sensitive': False, 'color': 'blue'},
{'severity': 'review', 'keyword': 'Review_Reminder:', 'case_sensitive': False, 'color': 'blue'},
… ]
```

With this list loaded in Python, we can write a function to load in a .log file and check the file against this keyword list we have.

LOG FILE SELECT AND PROCESSING

The next major proportion of the program is to process target .log files. We need to develop a practical way for users to feed a .log file into the application, and to process the file right after. Within this function, we will need to perform the following,

1. Create at least one method to read in a .log file,
2. Check the log file line by line to identify a match in keywords,
3. Create a summary of all the identified log lines,
4. Prepare objects/variables for final display.

First, we will define a master function for this purpose, and the very first step of this function is to ask for the path and name of the .log file:

```
    def open_log_file(self, file_path=None):
```

```
        """Open a log file and analyze it"""
        if not file_path:
            file_path = filedialog.askopenfilename(
                initialdir='.',
                title="Select Log File",
                filetypes=(("Log Files", "*.log"), ("All Files", "*.*"))
            )

        if file_path and file_path.lower().endswith('.log'):
            self.text_box.delete('1.0', tk.END)
            self.keyword_listbox.delete(0, tk.END)
            self.keyword_lines.clear()
```

The `file_path = filedialog.askopenfilename` function will open a pop-up file explorer dialog box for users to navigate and choose the .log file to process, then assign the path to `file_path` variable. After that, it will initialize the various objects holding different information pieces. Within this program, `self.text_box` is an object to hold the full texts loaded from the .log file for display; `self.keyword_listbox` is an object to hold the matched log lines for display; `self.keyword_lines` is an object to hold the matched log lines for back-end processes; all these initialization is to be done each time a new .log file is loaded, so it will clear out the information stored from the last .log file.

After initialization of the objects, we need to open the file and process it line by line.

```
            with open(file_path, 'r') as log_file:
                line_number = 1
                for line in log_file:
                    line = line.strip()
                    self.text_box.insert(tk.END, f'{line_number:08d}: {line}\n')

                    # Check if the line should be excluded based on the 'Exclude' severity
                    if self.check_exclude(line):
                        line_number += 1
                        continue

                    # Check if the line matches any keyword
                    matched_keywords = self.check_keywords(line)
                    if matched_keywords:
                        self.keyword_lines.append((line_number, line, matched_keywords))

                    line_number += 1
```

The `with open(file_path, 'r') as log_file` clause will open the selected log file and turns it into a list object, with each row of .log text being one entry in this object. We are giving each line of log a `line_number`, and we will iterate through the log file line by line with a for loop. For each line of log, we will insert them into the `self.text_box` object, then check for exclusion keywords with the `self.check_exclude` function (see Appendix A). Any log line that satisfies the exclusion function will not be processed further. If the log line gets pass the exclusion check, it will then check for the 4 level of keywords from top to bottom with the `self.check_keywords` function (see Appendix A). If a critical message is identified, it will not run for warning/review/information checks. If a match is found, this line of log will be added to the `self.keyword_lines` object.

Now that we have loaded in the log file and identified all the matching lines, we will construct the summarization panel. In order make the result more intuitive, we are adding colors to different level of log messages. Below is the function to process the log line and add them to the summary display panel.

```
                for line_number, line, matched_keywords in self.keyword_lines:
                    self.keyword_listbox.insert(tk.END, f'{line_number:08d}: {line}')

                    for matched_keyword in matched_keywords:
                        keyword = matched_keyword['keyword']
                        self.keyword_listbox.itemconfig(tk.END, {'fg': matched_keyword['color']})
                        start_index = f'{line_number}.0'
```

```
                    end_index = f'{line_number}.end'

                # Display the keyword counts in the summary label
                self.update_keyword_counts()
```

Notice that we have two objects for the keywords found (`self.keyword_listbox` and `self.keyword_lines`). They are not redundant but rather have different purposes. The `self.keyword_lines` object is to store the log lines with line number and keyword found within the line, and will be used in the back-end for other functions (navigation, highlighting, etc.). The `self.keyword_listbox` object is to be displayed in the user interface. First, we add each line of log in the `self.keyword_lines` object into the end of the `self.keyword_listbox` object. After that, modify the newly added lines with color option. This function `self.keyword_listbox.itemconfig(tk.END, {'fg': matched_keyword['color']})` will change the text color of the newly added line with the keyword's color loaded from the CSV file. After adding and coloring all the keyword lines into the display object, we will run through the `self.keyword_lines` object again to create counts of each level of log lines found in this file, with `self.update_keyword_counts()` function (see Appendix A).

At this point, we have selected and loaded in a target .log file, processed it with the keywords loaded from the keywords CSV file, and prepared all the necessary contents for display. We can start to construct the graphical user interface of the tool.

DEVELOP GRAPHICAL USER INTERFACE

There are plenty of open-source Graphical User Interface (GUI) libraries for Python, and the one that I used for this tool is called TKINTER. There are several advantages of using TKINTER: it is a built-in Python library, so you will have it once you installed Python; it is an old fashion but well-established tool so you can easily find help and examples online; it has all the basic functions to develop a simple to moderately complex user interface.

TKINTER has various widgets for you to choose from when developing GUIs. It has label, listbox, button, checkbutton, entry, frame, menu, scrollbar, etc. basically, for most of the elements you will find in a standard windows application, you can find a corresponding object in TKINTER.

To have the most precise control on the arrangement of different widgets using TKINTER, we will need to use Frame objects to split the window one by one. Figure 2 is how I would want the first demo to look like,

Figure 2. Graphical User Interface Draft Design

And here is the code to build this interface,

```
    def __init__(self, master):
        self.master = master
        self.master.title('ReadLog')
        self.master.geometry('1200x800')

        # Create a PanedWindow to hold the left and right panels
        self.paned_window = tk.PanedWindow(self.master, orient='horizontal',
                                           sashrelief=tk.RAISED, sashwidth=5)
        self.paned_window.pack(fill=tk.BOTH, expand=True)

        # Create left panel with keyword lines and summary
        self.left_frame = tk.Frame(self.paned_window)
        self.paned_window.add(self.left_frame)

        # Create right panel with log file
        self.right_frame = tk.Frame(self.paned_window)
        self.paned_window.add(self.right_frame)
```

From the above code, we first define the title and the size of this TK object. Then we add a TK object Paned Window `self.paned_window = tk.PanedWindow`. Paned Window is an object with a sliding bar in the middle of two separate panels. Within the PanedWindow, we will then create two `tk.Frame` objects to hold contents, one for the left panel and one for the right panel. If you have more elements to add into an interface, you can continue to split the frames into smaller pieces. After splitting the window into enough pieces, we can start adding the objects derived from previous sections into each place. We will begin with left panel,

```
        # Create the summary label
        self.summary_label = tk.Label(self.left_frame, anchor='w',
                                      padx=10, pady=10, font=('Helvetica', 12))
        self.summary_label.pack(side=tk.TOP, fill=tk.X)

        # Create keyword lines listbox and summary label in the left panel
        self.keyword_lines = []
        self.keyword_listbox = tk.Listbox(self.left_frame, width=60, selectbackground='lightblue')
        self.keyword_listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        self.keyword_listbox.bind('<<ListboxSelect>>', self.navigate_to_line)

        # Create a scroll bar for the keyword listbox
        scrollbar = tk.Scrollbar(self.left_frame, command=self.keyword_listbox.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.keyword_listbox.config(yscrollcommand=scrollbar.set)
```

In the above code, we create a `tk.Label` widget and a `tk.Listbox` widget, as well as attaching a `tk.Scrollbar` to the listbox widget. The listbox widget contains the log lines with matching keywords we derived from the previous function. `self.keyword_listbox.bind` command will attach a function call of `self.navigate_to_line` (see Appendix A) to the action `<<ListboxSelect>>`, which means that when a line in this listbox is selected, it will call the `self.navigate_to_line` function and perform the corresponding action defined within.

Next, we will define contents in the right frame,

```
        # Create a scrollable text widget in the right panel
        self.text_box = tk.Text(self.right_frame, wrap=tk.NONE)
        self.text_box.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Add a scrollbar to the text widget
        scrollbar = tk.Scrollbar(self.right_frame, command=self.text_box.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.text_box.config(yscrollcommand=scrollbar.set)
```

In the above code, we add a `tk.Text` widget which would hold the full text from the log file, as well as adding a scrollbar to the right. The `wrap=tk.NONE` option will prevent the text from wrapping if it goes beyond the viewable frame.

And to wrap up the tool, we will need to add in some buttons for initial function calls. There are several options within the TKINTER toolset, but I have chosen to go with `tk.Menu`,

```
        # Create a menu bar
        menu_bar = tk.Menu(self.master)
        file_menu = tk.Menu(menu_bar, tearoff=0)
        file_menu.add_command(label="Open Log File", command=self.open_log_file)
        file_menu.add_command(label="Exit", command=self.master.quit)
        menu_bar.add_cascade(label="File", menu=file_menu)
        self.master.config(menu=menu_bar)
```

This is a simple dropdown menu, with `"File"` as the menu button, and having `"Open Log File"` and `"Exit"` as the two command buttons in this menu. We can attach function call to each button by adding `command=` option.

With all the codes put together, we now have a working demo of a log file analyzer tool.

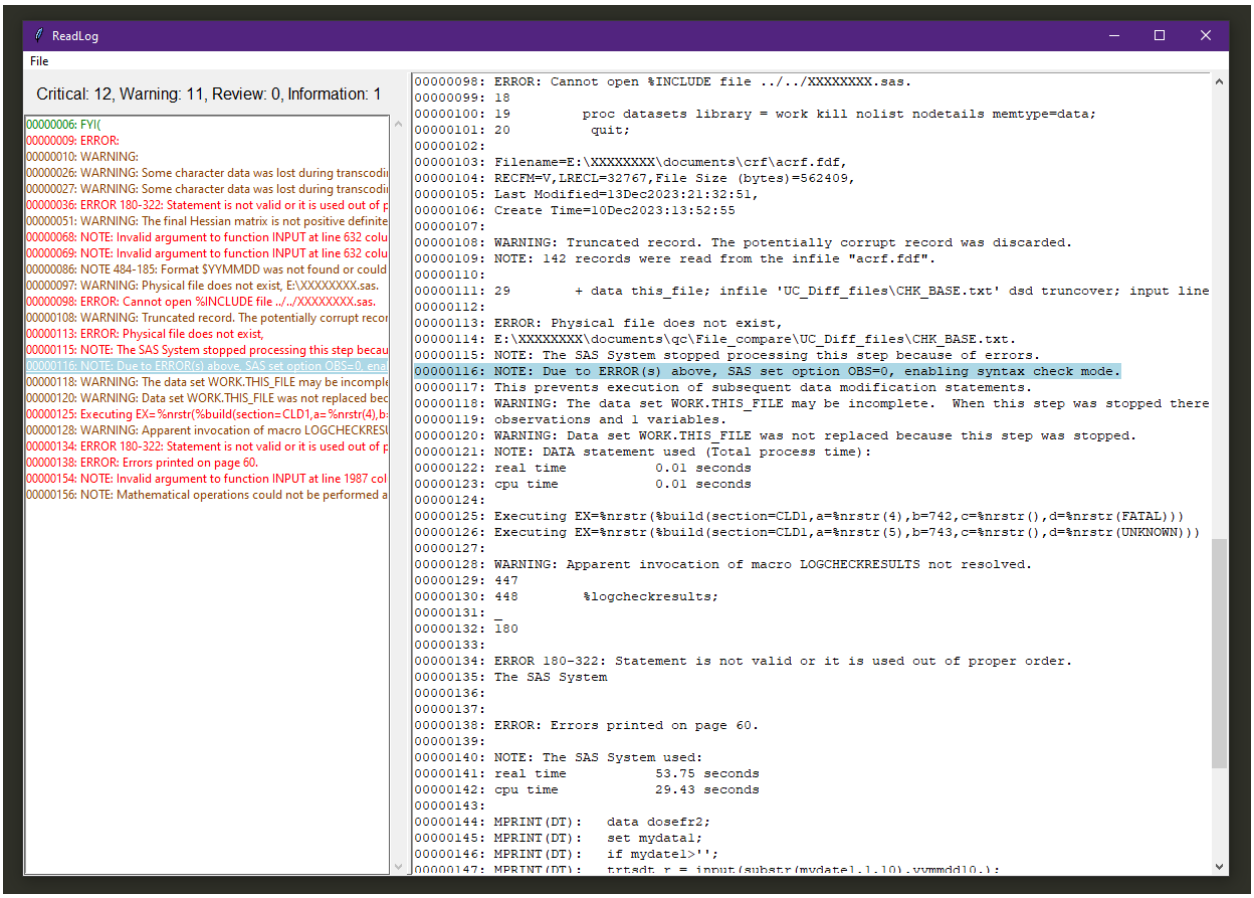

Figure 3. First Working Demo

CONTINUOUS DEVELOPMENT – IMPLEMENTATION OF VARIOUS FEATURES

In this section, we will move from entry level Python utilization to slightly advanced level. It requires more fundamental understanding and may not be as straightforward as the previous section.

The demo had proven that Python is a suitable tool for windows application development for use in our SAS server, the next step in my plan is to fulfill other missing pieces to eventually replace the existing log review process. Several additions would need to be made to achieve this goal:

1. It should be capable of processing multiple or all .log files within a folder,
2. It should generate a summary report output that would be compatible with the current summary report from SAS macro,
3. It should be executable at the end of a batch file, so the log tool would be executed and start processing all .log files within the same folder right after the last SAS program is finished in a batch run.
4. It should be a standalone application, with no dependencies on Python installation.
5. Additional quality-of-life features.

EXPAND THE GUI TO HOLD MULTIPLE FILES

To expand the functionality of the demo for opening multiple .log files at the same time, we can utilize the notebook widget within the TKINTER library. Within the TKINTER package, Themed TK (TTK) is a set of newer widgets compared to TK widgets. It gives themes to different widgets and usually provide a better appearance on different platforms. A **`ttk.Notebook`** widget can hold multiple copies of the same TK widgets in the back-end, and will bring up a specific copy to be displayed when a corresponding tab is selected. One of the excellent features of TKINTER is that not only can you split the UI by adding frames inside, but you can also add things outside to wrap around the existing widgets and frames. To utilize a **`ttk.Notebook`** widget, we will move all the UI creation codes from the above program into a separate function, and add a **`ttk.Notebook`** object into the main body of the code.

```
class LogFileAnalyzer:
    def __init__(self, master, **kwargs):

          ...
          ...
          ...

        # Create a notebook to hold the tabs
        self.notebook = ttk.Notebook(self.master, **kwargs)
        self.notebook.pack(fill=tk.BOTH, expand=True)

          ...
          ...
          ...

    def create_tab(self, file_path):
        """Create a new tab with components"""
        tab = ttk.PanedWindow(self.notebook, orient='horizontal')
        tab.pack(fill=tk.BOTH, expand=tk.TRUE)
        self.notebook.add(tab, text=os.path.basename(file_path))

          ...
          ...
          ...
```

By creating the `self.notebook` as a class variable, we can use the **`create_tab`** as an independent function to create new tab. The **`create_tab`** function will then be integrated into the **`open_log_file`**, so it will create a new tab within this notebook when a .log file is fed to the application. The rest of the code in **`create_tab`** function is mostly the same as when building the UI. But instead of putting the PanedWindow into the main TK object, we will add it into the notebook as a tab.

Besides changing the UI to hold multiple .log files, we will also need to add in functionality to take in multiple .log files. Luckily, there is already an existing solution in TKINTER.

```
    def open_log_file(self, file_path=None):
        """Open a log file and analyze it"""
        if not file_path:
            file_path = filedialog.askopenfilenames(
                initialdir='.',
                title="Select Log File",
                filetypes=(("Log Files", "*.log"), ("Text Files", "*.txt"), ("All Files", "*.*"))
            )

        if file_path:
            if isinstance(file_path, tuple):
                for fp in file_path:
                    self.open_log_file(fp)

            ...
```

By changing the function **filedialog.askopenfilename** to function **filedialog.askopenfilenames**, it allows users to select and open multiple .log files. However, changing from old function to the new one would also change the formatting of how the `file_path` variable would look like. We will need to add in codes to separate the multiple file names and paths into individual ones before processing.

The second piece of code above would verify whether the variable `file_path` is a **tuple** (similar to a list but has different property and usage). If that is true, it will then separate the value from the tuple, and for each value, call the function **open_log_file** again. **open_log_file** now becomes a recursive function: we call the **open_log_file** function within this same **open_log_file** function. And here is a diagram of how the logic works within this recursive function:

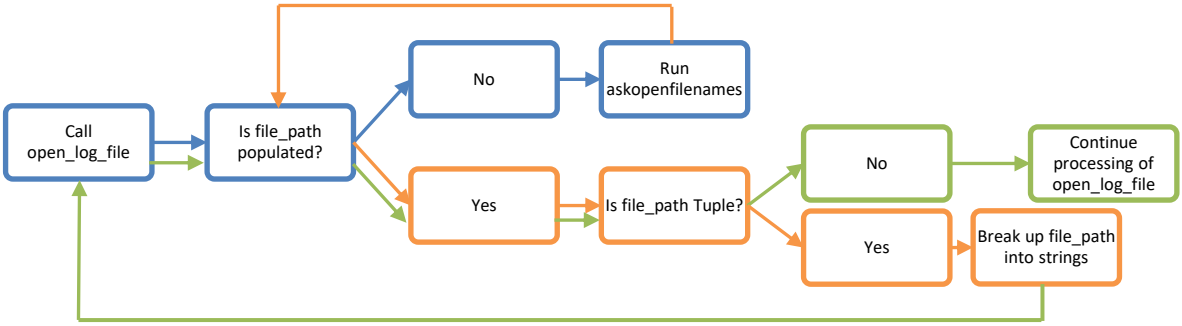

Figure 4. Path of Recursive open_log_file Call with Multiple Files Selected

At the beginning of the function call, it first verifies whether the `file_path` variable has value. In some other, scenario we can attach `file_path` value into the **open_log_file** function call (see Setting up the Application with Operation System). If `file_path` is null, the file selection dialog will pop-up (blue path). Once the `file_path` variable is populated from the dialog or it carries value from the function call itself, the code will check whether the `file_path` variable is a tuple. If it is a tuple (orange path), the code will divide the tuple into multiple single paths and recursively call the function one string at a time. This time the `file_path` variable becomes a string with single file path, it will then process the remaining code (green path). This process will run as many times as needed until each file path string is processed through the **open_log_file** function.

By adding in tab overlay as well as the option of multiple file selection, the tool is now capable of processing multiple .log files at the same time. And here is a screenshot of the second demo, with added tabs and other visual improvements:

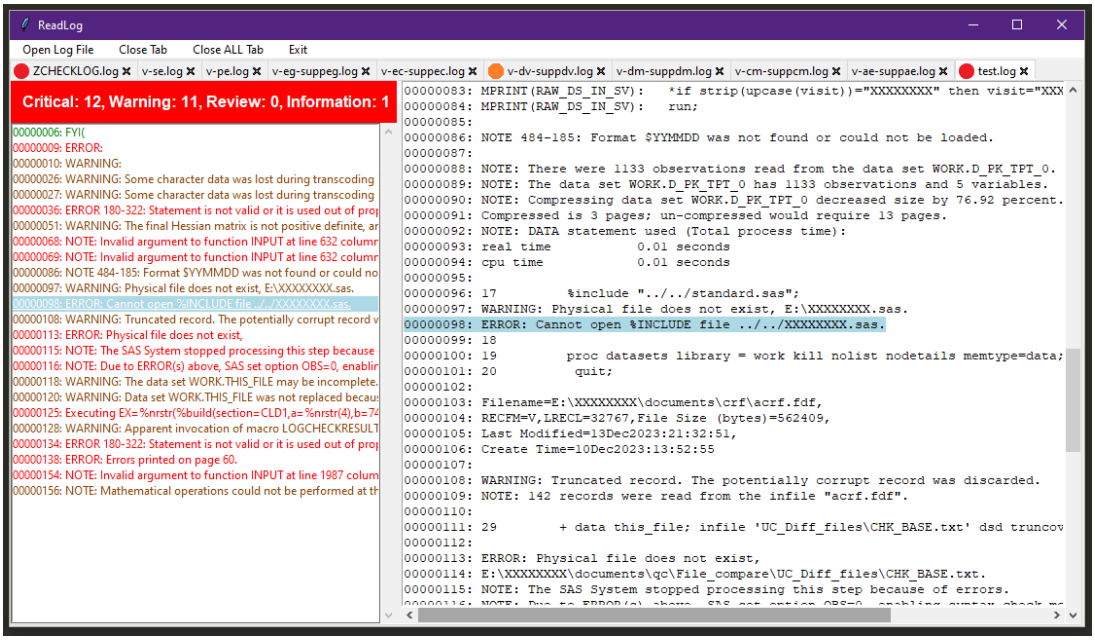

Figure 5. Second Demo with Multiple Log Files Capacity

GENERATING SUMMARY REPORT OUTPUTS

With the additional functions mentioned above, the tool is now capable of replicating 80% of the traditional method of log file review. The last piece of the puzzle would be to generate a summary report. For most of the functionalities we created above, they focus mostly on improving the user experience of individual programmers, while the summary report is designed to help the study lead or other quality control/quality assurance personnel to review all .log files in each programming folders. The summary reports residing in each programming folder should serve several purposes:

1. It provides an alternative way of reviewing when different programmers or study lead need to check on the status of the log files.
2. It provides a snapshot of the status of the log files after a batch run. It works as a supporting material for documentation, QC/QA procedures and archiving.
3. It also serves as an input for other applications.

We would want the report to focus on the most severe issues within this folder, and study leads should be able to tell whether the quality of the .log files within this folder passed all the checks, or there are still outstanding log issues to be resolved.

From the above codes, there is already an object we can use for this purpose. We have the class variable `self.keyword_lines` which holds all the identified log lines within each .log file and is ready to be processed to create such a report. This variable is holding all the important information as a list of lists object, which is very similar to SAS dataset. And we just need to create an output with a summary table followed by an explicit listing of all the issues found, sorted by descending severity.

```
    def export_result(self, file_path=None):
        # This function creates a Summary Report and save it to the location/name with popup window
        outReportClean = []
        outReportIssue = []
        outReportCombine = []
        cleanCount = 0
        issueCount = 0
```

We will first define several temporary variables to store the information we need to export. By default the report will be saved at the same location as where the first .log is processed, and it will have a default name Log_summary.txt. It will then process the stored information from `self.keyword_lines` from each

file and append them into either `outReportIssue` for log files with issues identified, or `outReportClean` for clean log files. Depends on the purpose of this report, we will write either just the `outReportIssue` variable, or both `outReportIssue` and `outReportClean` variables into the summary report. In the meantime, for each clean log or log with issues, we do +1 for the `cleanCount` or the `issueCount`. Then we can perform some file writing using the **open** file function, similar to how we read in .log files, but doing it reversely: we will combine all the texts we want to write to the file in an object of a list, use a for loop to separate this list into individual strings, and write to the file line by line. And below is an example of how the report looks like:

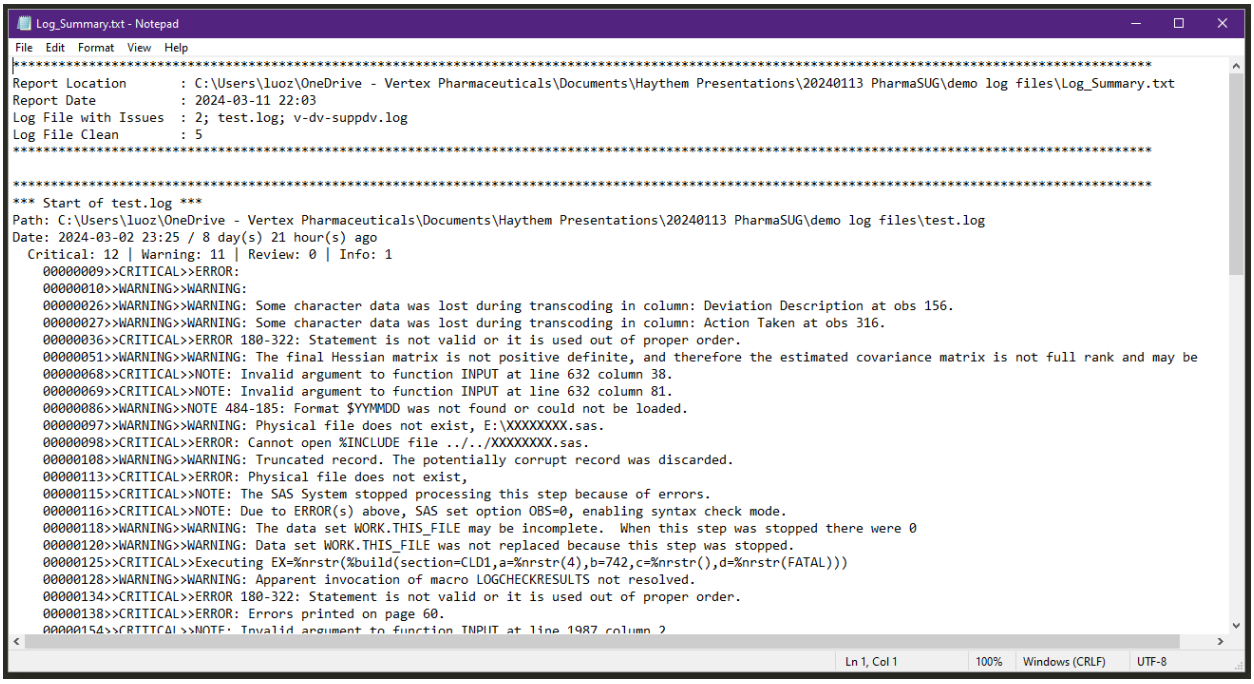

Figure 6. Contents from Summary Report

You may wonder why I chose to use a .txt file as the summary. It feels like I just defeat the purpose of moving away from plain text file and now going backwards again. But in fact, they are completely different scenarios. I have chosen to go with a .txt format because most of the time .txt file can be open immediately, and the tool is providing the summary in the first several rows of the report, so the reviewer will not need to go any further to get the information they need. Therefore, a txt file here would satisfy the requirements while having the best performance compared to other formats. The full text file, on the other hand, is a full snapshot of a summary of all the log messages identified and fully satisfied the requirements for documentation purpose as well.

SETTING UP THE APPLICATION WITH OPERATION SYSTEM

The tool now should check all the boxes I initially planned for. However, there are still plenty of aspects for improvement. With proper configuration, we can set up an OS environment link of the .log format to the tool. For instance, we can add codes for the Python program to understand this action: when the application is “launched with” a .log file, it should call the **open_log_file** function with the path and name of the attached .log file, and, on the other hand, set up a link from the Operation System to open all the .log format files with this tool. Then we can open any .log file anywhere with the tool by double clicking the .log file:

```
    if len(sys.argv) > 1:
        log_file_paths = [file_path for file_path in sys.argv[1:] if file_path.lower().endswith('.log')]
        for log_file_path in log_file_paths:
            analyzer.open_log_file(log_file_path)
```

With this part of code defined in the top-level environment, it enables the application to look for additional arguments that were fed into the execution command at launch. After linking the .log format in the operation system, when we open a .log file (i.e. C:\Temp\text.log) by double clicking, an equivalent command would be executed in the OS:

```
ReadLog.exe C:\Temp\text.log
```

The if statement with `len(sys.argv) > 1` will check whether the command itself contains more than one argument; the second row of code will then perform a .log file path verification. For each .log file path, the tool will call `open_log_file` to open and analyze it.

With the set up above, we can expand the command to let the tool takes in multiple .log files with batch files (see Appendix C).

PREPARE FOR DISTRIBUTION

Not every computer or server has Python. Technically, if someone want to run a piece of Python code, they will need to install Python as well as the corresponding libraries into their computer. But there is another way to bypass this requirement. When we need to publish a Python project to the server or distribute it to other users on their laptops, we will need to create an executable package which would include all the required libraries. In this case, we will need to download and run a Python package called Pyinstaller. It will search for all the dependent packages within Python, put them all into a single folder and convert your .py codes into an executable file (.exe in Windows). Then you can run it like all your other Windows executables or distribute it to target users. Here is an example of how to compile an executable package for ReadLog_Demo.py:

```
PS C:\Temp> pyinstaller --onefile --windowed readlog_demo.py
```

The “onefile” option would package everything into a single executable file (which is a compressed file, easier for distribution and management but slightly increase launch time). “Windowed” option would hide the command line window used by Python execution.

OTHER QUALITY-OF-LIFE IMPROVEMENTS

Drag and Drop function is probably the most anticipated addition to the tool. For individual programmers, you can now keep the application open, and drag-and-drop any .log file into the tool to open them. It can handle multiple file drops which greatly improve the user experience of the tool. There are plenty of Drag-and-Drop extension libraries that is built around TKINTER and are publicly available online.

A refresh button is attached to individual .log file panel. This button will call the `open_log_file` again to reload and re-summarize the selected .log file. It significantly improves the experience of individual programmer who is debugging single SAS program.

A status bar is also added into the interface, which would let the user know the path of the .log they have opened, the date time of the file last modified and how long ago it was modified (i.e., 2023-03-01 12:15 / 5 min(s) ago), and a simple search function.

Current version of the tool can check through all .log within a selected folder and all its subfolders. This is useful when performing a thorough check on a whole deliverable or even a whole study. However, it will also open some of the unintended .log files (i.e. backed up .log from previous run, .log from retired items, other.log files that does not require formal QC, etc.). A function to open only the intended programming folders within a reporting effort folder is under construction and will be added in the next version. This function would check through all .log files with a list of predefined programming folders under a reporting effort folder, and create a single summary report for all issues.

As of March 2024, we are using a stable version of the tool as below,

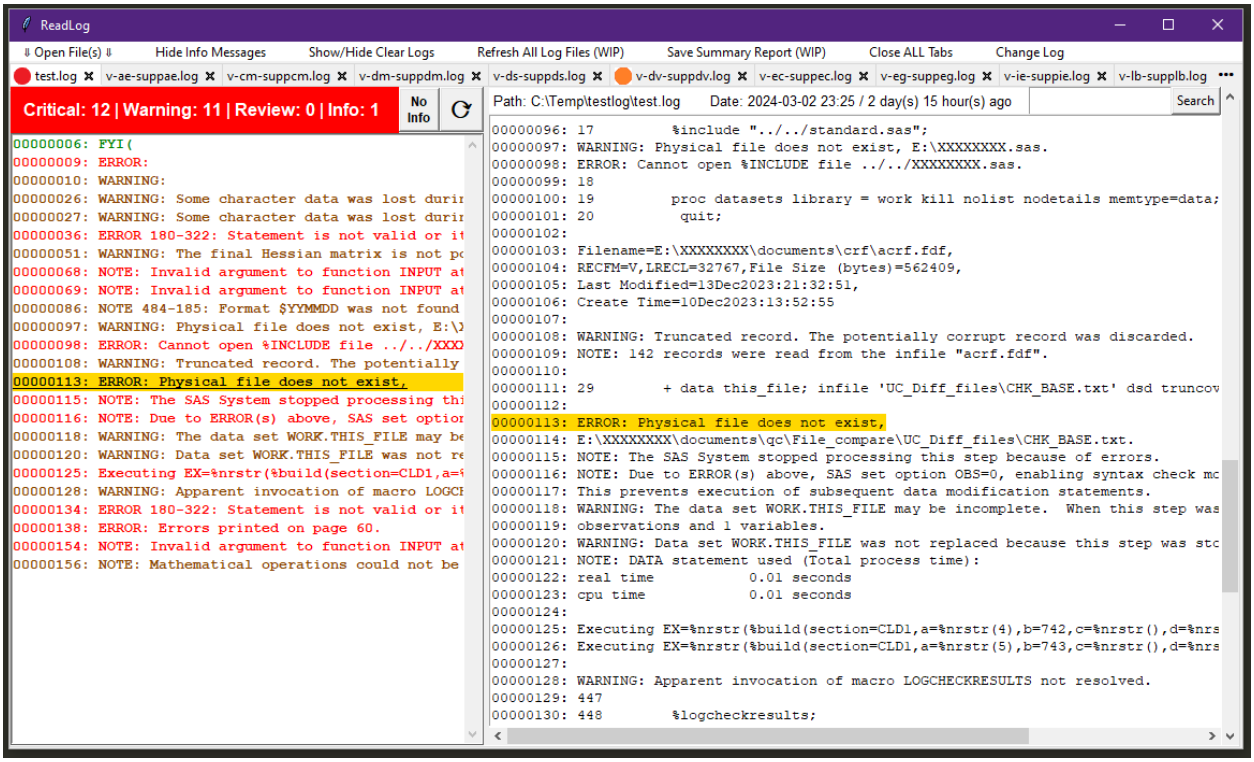

Figure 7. March 2024 Version of ReadLog Utility

The tool can now fully replace the traditional way of SAS .log reviewing process within the statistical programming department. We are currently still in the process of feedback collection and refinement of the tool, at the same time figuring out a way to QC the results to ensure the accuracy of the summarized log messages, before formally distributing the tool to all programmers.

THE END GAME – A COMPREHENSIVE QC SYSTEM

With the ReadLog Tool mostly completed, I have started planning my next project, which would automate many other QC processes we face in statistical programming works. It is still an early stage of a plan to develop a Comprehensive Quality Control System for SAS programming. The system should include components to perform the following activities,

1. SAS .log QC and documentation.
2. SAS validation results management, a way to summarize the PROC COMPARE result.
3. SAS programs cross checking among SDTM, ADaM and TFLs
    a. SDTM, ADaM and TLFs metadata changes monitoring (check if program has been updated or rerun after a change in specs or shells)
    b. SDTM, ADaM and TLFs dependencies check (ensure ADaM is rerun if corresponding SDTM dataset updated, same for TLFs with ADaM).
    c. SAS file timestamps and LOG file timestamps check (ensure program is rerun after changes were made, and QC program run after production program).
    d. Program files and metadata cross check (identify missing programs).
4. A new tool (Excel VBA, new tool developed in Python or other language) to dynamically generate a programming progress and comments tracker, and integrate all the information above into the

tool.

With the ReadLog Utility, we can export raw data as input for this comprehensive tool. A lot of the other required information can be obtained by making slight modifications to existing SAS macros.

The body part of the tracker will be dynamically generated with metadata from SDTM, ADaM and TFLs. As a programming tracker, individual programmers can enter their names and comments into each item they worked on. The status of log files, PROC COMPARE results, dependency checks and chronology checks would be obtained from prepared datasets in a centralized location for each reporting effort, and users also have the option to refresh any results by calling different connected applications or macros to run in the background.

OTHER ADAPTATIONS

Although I started this tool with the intention to smoothen the .log file review process, the tool itself is not attached to .log or any file format, nor is it useable only for SAS log review. It is in fact a "Keyword Identification and Navigation Tool" with a set of predefined keywords. If you have a proper keywords spreadsheet, you can adapt this tool to look for other keywords in documents with other formats.

For example, RTF and HTML formats are widely used in statistical programming, and they store information in human readable syntaxes (whereas DOCX is a compressed format and opening the file with text editor will give you unreadable wall of texts). If you have a list of unwanted characters, terms, or syntaxes you would want to identify in the source text of an RTF or an HTML file, this tool can be converted to perform such action.

CONCLUSION

There are a lot of area of improvements awaiting to be made in the statistical programming world. Python is a language that is comparatively easy to learn, and widely compatible with open-source libraries to help you achieve your goal. I hope my article would give you inspiration and motivate everyone to start their own Python projects soon.

RECOMMENDED READING

This is the instruction document for TKINTER library:

- *Tkinter – Python interface to Tcl/Tk (https://docs.python.org/3/library/tkinter.html, Last Accessed 3/11/2024)*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Zhihao (Haythem) Luo
Vertex Pharmaceuticals
Haythemluo1987@gmail.com

APPENDIX

APPENDIX A: FULL PYTHON CODES OF THE FIRST WORKING DEMO

Below is the full Python code of the first working demo. To execute this piece of code, you will need to install Python application from official source, copy the code fully into Python editor, and save it as a Python code file in .py format. You will also need your own Keywords.csv file to run this code properly. The file can be put at the same location as where you save your .py python code at, or you can give a fix location to the keywords.csv file by modifying this line of code `csv_file_path = './keywords.csv'` in the program.

Filename: ReadLog_Demo.py
Path: c:\temp\
Body:

```
"""
Project Name:       ReadLog Utility

Purpose:            Create an interactive GUI for log checking activities
                    Quick navigation to indicated log messages

Author:             Haythem Luo
Email:              haythemluo1987@gmail.com

Current Version:    0.1

Version history:    0.1     Initial version with log message indication and navigation

"""
import os
import csv
import sys
import tkinter as tk
from tkinter import filedialog


class LogFileAnalyzer:
    def __init__(self, master):
        self.master = master
        self.master.title('ReadLog')
        self.master.geometry('1200x800')

        # Create a PanedWindow to hold the left and right panels
        self.paned_window = tk.PanedWindow(self.master, orient='horizontal',
                                           sashrelief=tk.RAISED, sashwidth=5)
        self.paned_window.pack(fill=tk.BOTH, expand=True)

        # Create left panel with keyword lines and summary
        self.left_frame = tk.Frame(self.paned_window)
        self.left_frame.pack(fill=tk.BOTH, expand=True)
        self.paned_window.add(self.left_frame)

        # Create right panel with log file
        self.right_frame = tk.Frame(self.paned_window)
        self.right_frame.pack(fill=tk.BOTH, expand=True)
        self.paned_window.add(self.right_frame)

        # Create the summary label
        self.summary_label = tk.Label(self.left_frame, anchor='w',
                                      padx=10, pady=10, font=('Helvetica', 12))
        self.summary_label.pack(side=tk.TOP, fill=tk.X)
```

```
        # Create keyword lines listbox and summary label in the left panel
        self.keyword_lines = []
        self.keyword_listbox = tk.Listbox(self.left_frame, width=60, selectbackground='lightblue')
        self.keyword_listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        self.keyword_listbox.bind('<<ListboxSelect>>', self.navigate_to_line)

        # Create a scroll bar for the keyword listbox
        scrollbar = tk.Scrollbar(self.left_frame, command=self.keyword_listbox.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.keyword_listbox.config(yscrollcommand=scrollbar.set)

        # Create a scrollable text widget in the right panel
        self.text_box = tk.Text(self.right_frame, wrap=tk.NONE)
        self.text_box.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Add a scrollbar to the text widget
        scrollbar = tk.Scrollbar(self.right_frame, command=self.text_box.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.text_box.config(yscrollcommand=scrollbar.set)

        # Load keywords from the CSV file
        self.keywords = self.load_keywords()

        # Create a menu bar
        menu_bar = tk.Menu(self.master)
        file_menu = tk.Menu(menu_bar, tearoff=0)
        file_menu.add_command(label="Open Log File", command=self.open_log_file)
        file_menu.add_command(label="Exit", command=self.master.quit)
        menu_bar.add_cascade(label="File", menu=file_menu)
        self.master.config(menu=menu_bar)

    def load_keywords(self):
        # Load keywords from the CSV file
        keywords = []
        print("Enter keyword loading")
        csv_file_path = os.path.join(os.path.dirname(sys.executable), 'keywords.csv')
        if not os.path.isfile(csv_file_path):
            # put the keywords.csv file at the same folder
            csv_file_path = './keywords.csv'

        if os.path.exists(csv_file_path):
            with open(csv_file_path, 'r') as csv_file:
                reader = csv.reader(csv_file)
                next(reader)  # Skip the first row
                for row in reader:
                    severity = row[0].strip().lower()
                    keyword = row[1].strip()
                    case_sensitive = row[2].strip().lower() == 'y'
                    color = row[3].strip()

                    keywords.append({
                        'severity': severity,
                        'keyword': keyword,
                        'case_sensitive': case_sensitive,
                        'color': color
                    })
        print(keywords)
        print("Exit keyword loading")
        return keywords
```

```
def open_log_file(self, file_path=None):
    # Open a log file and analyze it
    if not file_path:
        file_path = filedialog.askopenfilename(
            initialdir='.',
            title="Select Log File",
            filetypes=(("Log Files", "*.log"), ("All Files", "*.*"))
        )
    # Verify file to be .log file and initialize variables
    if file_path and file_path.lower().endswith('.log'):
        self.text_box.delete('1.0', tk.END)
        self.keyword_listbox.delete(0, tk.END)
        self.keyword_lines.clear()

        # open file and write into text_box
        with open(file_path, 'r') as log_file:
            line_number = 1
            for line in log_file:
                line = line.strip()
                self.text_box.insert(tk.END, f'{line_number:08d}: {line}\n')

                # Check if the line should be excluded based on the 'Exclude' severity
                if self.check_exclude(line):
                    line_number += 1
                    continue

                # Check if the line matches any keyword
                matched_keywords = self.check_keywords(line)
                if matched_keywords:
                    self.keyword_lines.append((line_number, line, matched_keywords))

                line_number += 1

        # Display the keyword lines in the left panel
        for line_number, line, matched_keywords in self.keyword_lines:
            self.keyword_listbox.insert(tk.END, f'{line_number:08d}: {line}')

            for matched_keyword in matched_keywords:
                keyword = matched_keyword['keyword']
                self.keyword_listbox.itemconfig(tk.END, {'fg': matched_keyword['color']})
                start_index = f'{line_number}.0'
                end_index = f'{line_number}.end'

        # Display the keyword counts in the summary label
        self.update_keyword_counts()

def check_exclude(self, line):
    # Check if the line should be excluded based on 'Exclude' severity
    for keyword in self.keywords:
        if keyword['severity'] == 'exclude' and keyword['keyword'].lower() in line.lower():
            return True
    return False

def check_keywords(self, line):
    # Check if the line matches any keyword
    matched_keywords = []
    for keyword in self.keywords:
        if keyword['case_sensitive']:
            if keyword['keyword'] in line:
```

```
                    matched_keywords.append(keyword)
            else:
                if keyword['keyword'].lower() in line.lower():
                    matched_keywords.append(keyword)
        return matched_keywords

    def navigate_to_line(self, event):
        # Navigate to the selected line in the log file
        selected_index = self.keyword_listbox.curselection()
        if not selected_index:
            return

        line_number = int(self.keyword_listbox.get(selected_index[0]).split(':')[0])
        self.text_box.tag_config('highlight', background='lightblue')
        self.text_box.tag_remove('highlight', '1.0', tk.END)
        self.text_box.tag_add('highlight', f'{line_number}.0', f'{line_number}.end')
        self.text_box.see(f'{line_number}.0')

    def update_keyword_counts(self):
        # Update the keyword counts in the summary label
        keyword_counts = {keyword['severity']: 0 for keyword in self.keywords}
        for line_number, _, matched_keywords in self.keyword_lines:
            for matched_keyword in matched_keywords:
                severity = matched_keyword['severity']
                if severity != 'exclude':
                    keyword_counts[severity] += 1

        summary_text = ""
        for severity, count in keyword_counts.items():
            if severity != 'exclude':
                summary_text += f"{severity.capitalize()}: {count}, "
        summary_text = summary_text.rstrip(", ")
        print(keyword_counts)

        self.summary_label.config(text=summary_text)

    def run(self):
        # Start the main event loop
        self.master.mainloop()


if __name__ == '__main__':
    root = tk.Tk()
    analyzer = LogFileAnalyzer(root)

    # Check if the program was opened with a .log file
    import sys
    if len(sys.argv) > 1 and sys.argv[1].lower().endswith('.log'):
        log_file_path = sys.argv[1]
        analyzer.open_log_file(log_file_path)

    analyzer.run()
```

APPENDIX B: SAMPLE KEYWORDS.CSV FILE

Below is a sample Keywords.csv file. It only contains a selected list of .log issues, so you will need to add in your own keywords to make it function properly. This list of keywords below is only provided for demonstration and studying purpose. I have also added my definition for each level of log messages into the body of this file. You can remove them to slightly increase performance.

Filename: Keywords.csv
Path: c:\temp\
Body:

```
severity,keyword,case_sensitive,color
Critical,"(Critical log message that would terminate SAS execution or highly possibly affect the outcome of the program, required to be fixed)",Y,red
Critical,ERROR:,Y,red
Critical,FATAL,Y,red
Critical,NOTE: The SAS System stopped,Y,red
Critical,NOTE: DATA STEP stopped due to looping,y,red
Critical,NOTE: Division by zero,y,red
Warning,(Warning log message that has possibility of impact on output and should be fixed),Y,darkorange4
Warning,is uninitialized,n,darkorange4
Warning,WARNING:,Y,darkorange4
Warning,NOTE: MERGE statement has more than one,y,darkorange4
Warning,NOTE: Numeric values have been converted to character,y,darkorange4
Warning,NOTE: Character values have been converted to numeric,y,darkorange4
Review,(For custom queries coming from Macro or user defined checks that would require manual review efford),Y,blue
Review,Review_Duplicate:,n,blue
Review,Review_SpecialCharacter:,n,blue
Review,Review_Reminder:,n,blue
Information,"(For custom log massge that is informational only, lowest level of check)",y,green
Information,FYI(,y,green
Exclude,(any log message identify with below keyword would be excluded from summary regardless of message leve),y,black
Exclude,put warning,n,black
Exclude,put error,n,black
Exclude,%put err,n,black
```

APPENDIX C: SAMPLE BATCH FILE

Below sample batch file includes several placeholders of SAS program executions and a launch command for ReadLog Utility at the end. Before setting up the PATH environmental variable to direct .log execution to readlog.exe, you will have to call with the full path of the readlog.exe. Notice that this command is not compatible with Demo Version 0.1 (first demo).

Filename: Test_Batch_File.bat
Path: c:\temp\testlog\
Body:

```
SAS test_program_1.sas
SAS test_program_2.sas
SAS test_program_3.sas

C:\temp\readlog_demo.exe test_program_1.log test_program_2.log test_program_3.log
```

This batch file will batch run the first 3 SAS programs within the executed folder, and launch Readlog.exe to analyze their log files.