Paper 159-2023

# SAS® Log Parsing Made Easy with Python

Erin O'Dea, NORC at the University of Chicago

## ABSTRACT

The intention of this paper is to provide an educational experience for SAS Users who are interested in learning more about using Python in their work. Many SAS users will regularly run the exact same program daily, weekly, or monthly - sometimes they will run 50 or 100 SAS programs on a regular basis. Each of these programs might output a log, and many of those logs may have an error or warning that the user expects to see.

While best practice in coding is to completely remove these warnings or errors, there are many situations where it would require a significant amount of re-writing to remove them. SAS users may have inherited this code or may be under time constraints in their work.

Python has the ability to read log outputs, and with some clever engineering, can compare those outputs to previous runs and output a list of new warnings or errors. This cuts down substantially on the time it takes to read and interpret SAS log outputs. Without needing to install additional programs or libraries, it is possible (and relatively easy) to parse hundreds of SAS logs within seconds using Python.

## INTRODUCTION

This paper is intended for users with a basic understanding of Python as a programming language but is broken down into easy steps and explained piece by piece. Because of the incredible necessity for proper indentation in Python, the entire program for this technique is located at the end of the paper. This can be adjusted for many kinds of log parsing, but the main intention is to compare one set of SAS log outputs to another set of SAS log outputs. This paper will follow the process from the very beginning of outputting a log to the very end with your own text file and formatted output based on the log parser.

## MAKING A BETTER LOG IN SAS

One of the most important advantages you can give yourself when working with a significant number of files is proper organization. This starts at the most basic level in SAS: log outputs. While SAS will automatically provide you a log inside of whatever tool you use to run it, saving the log programmatically in a specified location is vital to this strategy, and is also useful for reference for anyone who may be running the code regularly. An easy way of doing this within SAS is with the proc printto function at the beginning of the program:

```
proc printto log="C:\Users\SASUser\Example.log" new;  run;
```

To end a proc printto function you simply need to close out at the end of the file:

```
proc printto;run;
```

This function can be made more useful for programs that are run regularly by use of SAS's built in date function. Sysfunc allows you to access these and place them in a macro variable, which you can add on to the end of your log name. Make sure to add a period to the end of your macro variable so that SAS can interpret the ".log" as the type of file:

```
%let todaysDate = %sysfunc(today(), yymmddn8.);
proc printto log="C:\Users\SASUser\Example._&todaysDate..log"
new;  run;
```

This ensures that each day the program is run, you will have a separate and comparable log in the same folder location. This can be modified—for example, if a program is run multiple times per day, you can utilize the built-in time function from SAS to add to your log name. If a program is run for a test space and

then run for a production space, you can add _test or _prod. Anything makes sense for the regularity of the program or purpose of the run is worth adding to the log name or log location, as this will be extremely important when trying to match up logs in Python.

## READING LOGS WITH PYTHON

### FINDING THE RIGHT LOGS

This technique uses basic Python and does not require any additional libraries outside of what comes with a standard Python installation. However, you do need to access the os library, which gives the user access to system commands and file structures. This comes installed with Python, but still needs to be imported at the top of the file for use. You can use the os function listdir to look at a file path and find everything inside like so:

```
import os
files = os.listdir("C:\Users\SASUser\ExamplePath\LogLocation")
```

This will put a list of every file name into an array called “files” which you can then loop through to determine which files you need to compare for the run. This is where using the date functions can be extremely helpful. If you name every log using date functions, you can look for log names that contain that specific date to tell the program these files should be taken into consideration. You can also look for a previous run date to see which logs to compare the new logs to:

```
newfiles = []
oldfiles = []
for file in files:
    if “10312023” in file:
          newfiles.append(file)

    elif “09242023” in file:
          oldfiles.append(file)
    newfiles.sort()
    oldfiles.sort()
```

Now you have two arrays of files from the same path, one that includes all files with the most recent run date, and the other that includes all files from the previous run date. This can be changed based on the naming conventions that the user has decided to use for their logs, but the effect is the same. You have two sets of the same logs from two different runs. You can use the built-in Python sort method or any sorting method the user prefers to make sure that the files are aligned in both arrays so that you can directly compare one log from the previous run to one log in the new run.

Once the logs are collected and organized, you’ll want to start another loop. You can use an iterative variable to keep track of where you are within your array and access both arrays with the variable as you loop through. Using Python’s length function len, you can run the loop until the program reaches the final file in the array, and with Python’s formatted literal strings you can combine the file path and name into one variable to keep the program concise:

```
i = 0
path = “C:\Users\SASUser\ExamplePath\LogLocation”
while i < len(newfiles):
    fileNew = f"{path}\{newfiles[i]}"
    fileOld = f"{path}\{oldfiles[i]}"
```

### READING THE LOGS

Within the loop you’ve just created, you are ready to begin using Python to read these logs. Python has built-in functions to read files, and there are two that will be used by this technique: first, you’ll open the

log using Python's open method, and second, you'll read the lines using Python's readlines function. The open method exists on any file object in Python and accepts two parameters:

1. The file path with the file name you want to look at.
2. The "mode" of opening – here we use 'r' for read.

You can use these functions on both the old log file and the new log file, and after execution you will have two new arrays of strings—one that contains each line in the old log, and one that contains each line in the new log. You also want to create some empty arrays store the warnings and errors that the program might find in each log:

```
newFile = open(fileNew, 'r')

newLines = newFile.readlines()

oldFile = open(fileOld, 'r')

oldLines = oldFile.readlines()

warnNew = []
errNew = []
warnOld = []
errOld = []
```

## FINDING WARNINGS AND ERRORS

Now that you have your strings, you'll want to create yet another loop to look at each line. SAS Logs have WARNING and ERROR as the first part of the line output. That means that when you are looking for either warnings or errors in your array of strings, you can expect them to be at the very beginning of the string. Python has another built-in function called startswith that will look there specifically to avoid any potential code that may have "warning" or "error" as a standard part of the input stack. You can use this function as you look at each line, and if it meets your criteria, you can go ahead and add it to your warning and error holding arrays. Stripping the lines with Python's strip function is a way of keeping your arrays clean of any empty space. You will repeat this process on the old log file as well:

```
for line in newLines:
      if line.startswith("WARNING"):
            warnNew.append(line.strip())
      elif line.startswith("ERROR"):
            errNew.append({line.strip())
for line in oldLines:
      if line.startswith("WARNING"):
            warnOld.append(line.strip())
      elif line.startswith("ERROR"):
            errOld.append(line.strip())
```

Once this is executed, you will have every error and warning from your SAS log in their own arrays. In other words, you have found all the errors and warnings present in your first logs for both the old and new runs, and now you can go about comparing the lists.

## WRITING LOGS WITH PYTHON

### CREATING A LOG FILE IN PYTHON

If you are using this technique just for a few logs, or if you expect very few warnings or errors, you can simply print the relevant information into the Python terminal with a print statement. This is straightforward, but not as effective for reference as creating a new document. Outputting a text file is fairly simple with Python, and it ensures that you programmatically save the contents for future reference and comparison. To do this, you first need to create and open a new text file. This only needs to be done once at the beginning of the program and can stay open until all the loops are complete. We will use the same open function that we used to read the original logs, but in this case the "mode" will be "a" for append. This allows Python of create a brand-new file, or if the file exists, append information to the end of the file. We can also give the log file a variable nickname to keep our program concise:

```
logpath = "C:\Users\SASUser\ExamplePath\LogLocation\FinalLog"
with open(f"{logpath}/LogCheck_10312023.txt", 'a+') as mylog:
```

Everything that comes after the log open statement should be indented. What we are telling Python is to create and open a text file, then perform all the actions in the rest of the program. We can write to this log at any time within the program itself without needing to re-open the file. This is the most efficient way of accessing it because it doesn't require opening and closing the file multiple times, and it will cut down execution time.

### WRITING TO A LOG FILE IN PYTHON

Once you have a log file created and opened, you can write to it very simply. What you've created is called a file object in Python, and a file object has a method called write. This works just like a function, but is tied to a specific file object and comes directly after the object name like so:

```
mylog.write("Any text you want!")
```

This will write whatever argument you give it directly into your text file. Because the file was opened with the "a+" mode, it will write to the very end of any existing text within the file. Its worth noting that each time you write again to the log, it will pick up in the exact place that the last thing written ends. Because of this, to create a more readable file it is best practice to end any statement you write in with a new line break. This can be achieved in Python by using Python's newline character \n:

```
mylog.write("Any text you want! \n")
```

## COMPARING LOGS AND FINAL OUTPUT

### COMPARING WARNING AND ERROR ARRAYS

The final step of this technique is to compare each warning and error located in the arrays you have gathered to determine whether these are new issues or things you have seen in previous runs. Once you have determined that a warning or error is brand new, you'll want to output it to your new log file. You'll also want to log which program created the warning or error as some of them may not be obvious from the auto-generated error and warning statements that SAS creates. This can be done by using another formatted string literal. You simply need to loop through the array of new warnings or errors and ask Python if those warnings or errors exist in the corresponding array of older warnings and errors.

```
for warn in warnNew:
      if warn not in warnOld:
            mylog.write(f"New Warning Found in {newfiles[i]}: /n {warn}")
for err in errNew:
      if err not in errOld:
            mylog.write(f"New Error Found in {newfiles[i]}: /n {err}")
```

### RESETTING FOR NEXT FILE

At the end of the comparison, you'll want to reset the arrays to empty arrays to prepare for the next round of log files you are searching and comparing. You'll also want to add one to your iterative variable to move on to the next set of files. **This is very important to avoid an endless loop in your program**:

```
warnNew=[]
warnOld=[]
errNew=[]
errOld=[]
i += 1
```

### ENDING THE LOG

Once all these loops have concluded, you've now created a new text file with all the new warnings or errors that are in the logs for the new run. The final step is best practice, and that is to close the text file. This is easily accomplished using one last file object method: close. This method does exactly what it says on the tin and does not require any arguments:

```
mylog.close()
```

## FULL PROGRAM WITH PROPER INDENTATION FOR PYTHON:

```
import os

files = os.listdir("C:\Users\SASUser\ExamplePath\LogLocation")
path = "C:\Users\SASUser\ExamplePath\LogLocation"
logpath = "C:\Users\SASUser\ExamplePath\LogLocation\FinalLog"

newfiles = []
oldfiles = []

for file in files:
    if "10312023" in file:
        newfiles.append(file)
    elif "09242023" in file:
        oldfiles.append(file)
newfiles.sort()
oldfiles.sort()

with open(f"{logpath}/LogCheck_10312023.txt", 'a+') as mylog:
    i = 0
    while i < len(newfiles):
        fileNew = f"{path}\{newfiles[i]}"
```

```
        fileOld = f"{path}\{oldfiles[i]}"
        newFile = open(fileNew, 'r')
        newLines = newFile.readlines()
        oldFile = open(fileOld, 'r')
        oldLines = oldFile.readlines()
        warnNew = []
        errNew = []
        warnOld = []
        errOld = []
        for line in newLines:
            if line.startswith("WARNING"):
                warnNew.append(line.strip())
            elif line.startswith("ERROR"):
                errNew.append(line.strip())
        for line in oldLines:
            if line.startswith("WARNING"):
                warnOld.append(line.strip())
            elif line.startswith("ERROR"):
                errOld.append(line.strip())
        for warn in warnNew:
            if warn not in warnOld:
                mylog.write(f"New Warning Found in {newfiles[i]}: \n {warn}
\n")
        for err in errNew:
            if err not in errOld:
                mylog.write(f"New Error Found in {newfiles[i]}: \n {err} \n")
        warnNew=[]
        warnOld=[]
        errNew=[]
        errOld=[]
        i += 1

mylog.close()
```

## CONCLUSION

Many aspects of this technique are applicable to all kinds of SAS log parsing, but these steps are the true building blocks to developing a Python program that can do hours of work for you in seconds of runtime. This script can be broken out into functions that can assist in all kinds of search functionality, and the only limit is your imagination and organization.

## CONTACT INFORMATION

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

Erin O'Dea
NORC at the University of Chicago
odea-erin@norc.org