PharmaSUG 2018 - Paper AD-12

Why SAS Programmers Should Learn Python Too

Michael Stackhouse, Covance, Inc.

ABSTRACT

Day to day work can often require simple, yet repetitive tasks. All companies have tedious processes that may involve moving and renaming files, making redundant edits to code, and more. Often, one might also want to find or summarize information scattered amongst many different files as well. While SAS programmers are very familiar with the power of SAS, the power of some lesser known but readily available tools may go unnoticed.

The programming language Python has many capabilities that can easily automate and facilitate these tasks. Additionally, many companies have adopted Linux and UNIX as their OS of choice. With a small amount of background, and an understanding of Linux/UNIX command line syntax, powerful tools can be developed to eliminate these tedious activities. Better yet, these tools can be written to “talk back” to the user, making them more user friendly for those averse to venturing from their SAS editor. This paper will explore the benefits of how Python can be adopted into a SAS programming world.

INTRODUCTION

SAS is a great language, and it does what it’s designed to do very well, but other languages are better catered to different tasks. One language in particular that can serve a number of different purposes is Python. This is for a number of reasons.

First off, it’s easy. Python has a very spoken-word style of syntax that is both intuitive to use and easy to learn. This makes it a very attractive language for newcomers, but also an attractive language to someone like a SAS programmer. SAS syntax is very specific to its task. The data step loop is a novel way to iterate over the rows of a dataset and makes data processing very simple, and the variety of procedures in SAS are very powerful. But when you look at tasks better suited for a general purpose programming language, things can get complicated.

Second, Python is powerful. The base package of Python itself comes packed with a variety of tools that make complex tasks extremely simple. Python data types are also very flexible, handling many things automatically that the programmer would need to specify in other languages. Furthermore, Python has an incredible amount of third party libraries available to install, expanding the language well beyond the base package and giving you the tools you need to get your job done quickly.

Another powerful tool that many SAS programmers have at the tip of their fingers is either Linux or UNIX. Many companies run SAS on a Linux/Unix OS, and SAS programmers often have access to the command line. Linux/UNIX has numerous commands available to the user to handle a wide variety of tasks. The problem becomes knowing the tools at your disposal and the syntax to use them.

There is a great deal to learn about these topics, and the internet has endless repositories about both, but the scope of this paper is to explore how a little bit of Python can make your day to day tasks both simpler and more efficient. To accomplish this, we will explore how Python can be applied to:

1. Enhance the command line
2. Easily gather information
3. Update numerous files at one time

Something to keep in mind as you read this paper is that the possibilities are virtually endless. A number of examples are presented, but these can be expanded or altered to suit your needs. My intention is to demonstrate that with a little bit of background and little bit of code, you can have a very large impact on your team’s or your company’s daily work.

ENHANCING THE COMMAND LINE

Many SAS programmers are intimidated by the command line. It is not your natural interface for using a computer today. You are left without a mouse, and you navigate and explore using different commands. This leaves many people uncomfortable – and yes, the learning curve can be a bit steep. But at the sake of comfort, these command-line averse programmers are missing out on a number of powerful tools.

The first example of how Python can be applied takes a somewhat obscure approach in that Python does not do the footwork. Here instead, a convenient capability of Python is applied – "talking back". The INPUT function (RAW_INPUT prior to Python 3) reaches back out to the user to supply the input to your variable:

```
>>> my_var = raw_input("Hello! What's your name? ")
Hello! What's your name? Mike
>>> print my_var
Mike
```

By making user input so simplistic, you can craft scripts that "talk back" to you and ask you informative questions. Generally, this is a useful feature of Python, but one way this can be applied is to help build command line syntax. This circumvents the requirement to memorize lists of options and the proper order of parameters, as the Python script can build it for you.

For example, on Linux/UNIX systems – the command *grep* is a very powerful search tool. It can scan all the files you specify and quickly give you any line of the file that matches your search pattern. For this example, we consider a shortcut version of the *grep* command called *fgrep*. Its functionality is largely the same, but it searches for exact text rather than apply regular expressions, which would greatly raise the complexity of this script.

Consider the following SAS program in your directory:

```
/*************************************************************************
test.sas

Inputs  : sdtm.dm, sdtm.suppdm, sdtm.fa, sdtm.suppfa, sdtm.vs, sdtm.lb,
Outputs : ADAM.adsl

*************************************************************************/
```

Input 1. Contents of test.sas

The following script is named *smart_search.py*. After executing the command, the user follows the prompts (user input and results in bold):

```
What files do you want to check? >>> test.sas

#########################################################
# List of options                                       #
# i -> Ignore case                                      #
# v -> Search instead for NON-matching lines            #
# c -> Return instead a match of a count of matching    #
#      lines for each input file                        #
# l -> Print file names with a match instead            #
#      of matching lines                                #
# n -> Add the line number of matching lines to output  #
#########################################################


String to search for >>> Inputs
List all options desired (hit enter for none) >>> i
```

```
Search for substring? (Y/N) >>> n
Command executed: fgrep -H -i 'Inputs' test.sas

test.sas:Inputs  : sdtm.dm, sdtm.suppdm, sdtm.fa, sdtm.suppfa, sdtm.vs,
sdtm.lb,
```

Now consider we have a second file – *test2.sas*. This script would also allow you to search for an unlimited number of substrings. For example – if you want to search all program headers for the *Inputs* line and search specifically for the text *sdtm.ae*:

```
/*************************************************************************
test2.sas

Inputs  : ADAM.adsl, sdtm.ae, sdtm.suppae
Outputs : ADAM.adae

*************************************************************************/
```

Input 2. Contents of test2.sas

Applying the search on all SAS programs in the directory (user input and results in bold):

```
What files do you want to check? >>> *.sas

#########################################################
# List of options                                       #
# i -> Ignore case                                      #
# v -> Search instead for NON-matching lines            #
# c -> Return instead a match of a count of matching    #
#      lines for each input file                        #
# l -> Print file names with a match instead            #
#      of matching lines                                #
# n -> Add the line number of matching lines to output  #
#########################################################

String to search for >>> Inputs
List all options desired (hit enter for none) >>> i
Search for substring? (Y/N) >>> Y
Substring to search for >>> sdtm.ae
List all options desired (hit enter for none) >>>
Search for another substring? (Y/N) >>> n
Command executed: fgrep -H -i 'Inputs' *.sas | fgrep  'sdtm.ae'

test2.sas:Inputs  : ADAM.adsl, sdtm.ae, sdtm.suppae
```

Beyond these examples, this Python script is capable of giving you matching file names, the count of matching lines, non-matching lines, adding line numbers to the output, and more. *Grep* has many other powerful options the could be incorporated to expand the script as well.

EASILY GATHERING INFORMATION

GATHERING UPDATES

Sometimes there also may be situations where there is not a command quite right for your situation. In these cases, it is also easy to craft your own scripts in Python to get the information that you need. Consider the situation where you have a directory of SAS programs of many different types – SDTM,

ADaM, tables, listings, figures. At certain times, you might want to be able to easily pull out programs updated after a certain date – or even a particular time of day on that date.

The following Python script, *check_date.py*, reports back a group of programs (or all programs) updated after a user specified date and time (user input in bold):

```
Tables, listings, figures, ADaM, SDTM, or all files? adam

Please enter the date time of comparison. Use the format: MMM DD YYYY HH:NN
(i.e. Apr 25 2017 15:23)
Comparison Date? Jun 23 2017 09:00

Development code updated files:
adeff.sas  Mon Jun 26 20:32:51 2017
adlb.sas   Tue Jul 11 19:39:18 2017
adeg.sas   Mon Jun 26 18:10:55 2017

QC code updated files:
qadeff.sas  Fri Jun 23 20:09:03 2017
qadlb.sas   Wed Jul 12 17:04:50 2017
qadmo.sas   Wed Jul 12 17:33:26 2017
qadeg.sas   Fri Jun 23 17:05:55 2017
```

Notice that by only specifying the group of programs, the script was able to identify all of the SAS files desired. By crafting your own script, you can specify your company's program naming conventions to enable you to find the appropriate programs very easily.

LOOKING FOR ISSUES

To take things a step further, consider tools tailored to tasks that you have in mind. One rather specific situation that you may want to examine is which files in your directory containing non-ASCII characters and the location of those characters. Without specific software, this is rather difficult to do with certain file types – so for this example we will only examine flat text files. That being said, this is still a useful tool when looking through CSV files or SAS programs. For example, if you are transferring programs between different platforms, then the source file encoding may need to be taken into consideration and non-ASCII characters could interfere. Additionally, CSV files are a fairly common data type to encounter, and this script would be capable of identifying all characters within the file.

This script, a*scii_check.py*, also has two methods of being executed: following the user prompts similar to the previous example, and from the command line directly.

Executing the plain command with no options from the command line, you are prompted for the following (user input bolded):

```
Non-ascii Character Check
List the files you want to scan, separated by spaces
You can also use Linux wildcards like '?' and '*'
What files do you want to check? *.*
Print to terminal? Answering N will create a CSV file to review
(Y or N) >>> Y
File Name   Issue Found
test.txt    Non-ascii character detected at Line: 1 Column: 0.
test.txt    Non-ascii character detected at Line: 1 Column: 1.
test.txt    Non-ascii character detected at Line: 7 Column: 0.
test.txt    Non-ascii character detected at Line: 7 Column: 1.
test.txt    Non-ascii character detected at Line: 17 Column: 0.
test.txt    Non-ascii character detected at Line: 17 Column: 1.
test.xml    Non-ascii character detected at Line: 1 Column: 0.
test.xml    Non-ascii character detected at Line: 1 Column: 1.
```

```
test.xml    Non-ascii character detected at Line: 7 Column: 0.
test.xml    Non-ascii character detected at Line: 7 Column: 1.
test.xml    Non-ascii character detected at Line: 17 Column: 0.
test.xml    Non-ascii character detected at Line: 17 Column: 1.
```

Note the last question – *Print to Terminal?* Answering N to this option instead outputs a CSV file with the same information. If the user needs to share the data, or expects a large deal of findings, this option is convenient in that it allows the information to be shared or filtered in a spreadsheet program.

The second option is to execute this command with all the necessary options directly from the command line. To get the same information as above, this would look as follows:

```
python ascii_check.py -print *.*
```

On Linux/Unix you could also set an alias to abbreviate the command further, and then use something similar to the following:

```
ascii_check -print *.*
```

This example may be rather specific, but the functionality of the script is interchangeable. The purpose of this example is to demonstrate how these scripts can have separate methods of interfacing – prompts for less experienced users and shortcuts for more advanced. With a little effort, these scripts can become much more flexible on the front end, encouraging your users to get the most benefit.

UPDATING NUMEROUS FILES AT ONCE

How about those tedious situations where the *same exact* update is required in all of your programs? To make matters worse, what if the programs reside in numerous subdirectories? Opening each program individually would be quite time consuming depending on the scope of the update. Luckily, there is an easier way.

Python has a few features that make an update like this a simple task. First and foremost, text processing is very easy in Python. With only a few lines of code, you can open a file, edit it, and save it back to the drive. Additionally, Python has numerous functions that make string operations an easy task as well. For example, globally replacing a substring within a string can be done as follows:

```
>>> "Hello! My name is Mike.".replace("Hello","Goodbye")
'Goodbye! My name is Mike.'
```

Once you get a text file into a string, it is just as simple.

The next convenience that Python has to offer is the simplicity of gathering the file names from a directory, or even subdirectories. Thanks to the power of the Python *os* module, you can get a list of all files within your directory and all subdirectories with one line of code:

```
os.walk(<directory>, topdown=False)
```

Combine these things together, add some conditional statements, and in less than twenty lines of code you can global replace any string you need in all the SAS programs in your directory (and subdirectories if required).

To demonstrate the basic functionality, consider the following input file:

```
data x;
    set y;
    studyid = "MIRACLE-DRUG";
run;
```

Input 3. Contents of test2.sas

The script could prompt you as (user input in bold):

```
From text? >>> MIRACLE-DRUG
To text? >>> MIRA-DRUG
```

The resulting output file would be:

```
data x;
    set y;
    studyid = "MIRA-DRUG";
run;
```

Input 4. Contents of test2.sas

Caution should always be taken when global replacing across an entire file, let alone numerous files at one time, but if you are confident that the replace will work appropriately, this can save an immense amount of time and patience.

AUTOMATE PACKAGE CREATION

This last example is fairly specific, but demonstrates how numerous strategies discussed in this paper can be combined. Now, consider a situation in which your deliverable is your program files rather than your datasets. In your partnership, you are developing SAS programs on your personal system, but need to migrate them to the client system where they must be executable.

The situation presents a few challenges:

- On your system, development and validation programs reside in the same folder with different names. On the client system, they are stored in separate folders with identical names.
- The client system requires an %INCLUDE line at the top of the file. Your system does not, so you comment it out.
- Some macros and macro variables must be renamed between systems.
- Some LIBNAMEs must be renamed between systems.

To handle this manually could potentially take hours. Each program file would need open and edited for the necessary changes. Additionally, making these changes manually opens the door to typos and mistakes that you may not see until you start executing programs. But the situation is consistent and repetitive, which makes it a perfect candidate for automation.

Step by step, here is how Python can be applied:

1. Establish what package needs to be created (i.e. SDTM, ADaM, Tables, Listings, Figures)
2. Create subdirectories to hold the program files for editing.
3. Copy the programs into the subdirectories, separating development and validation in the process.
4. Rename all validation programs to match their development side counterparts.
5. Execute global replaces on all of the program files to uncomment the %INCLUDE line, rename the necessary macros, macro variables, and LIBNAMEs.
6. Create zip files for easy delivery to the client system.

While this is a very specific situation, think about your own processes. What do you spend a lot of time doing? What aspects of the processes are redundant? What wastes time? Do you see a pattern? These are the things that open the door to automation. The more complex the situation and process, the more effort it will take, but even if it takes 40 hours to write a fully capable script, consider how much time can be saved. In this example, a delivery of 250 table programs could take hours to update on the client system. Furthermore, this upload may need to happen numerous times – potentially even weekly. The Python script can process the files in seconds.

CONCLUSION

While a SAS programmer's primary task is writing SAS code, Python can still bring a lot to the table. Python can wrap Linux/UNIX command line syntax in a user friendly interface, inviting less advanced Linux/UNIX users to take advantage of the power that some commands have to offer. Without too much code, Python can gather information from different files and directories for you, and customize it to your needs. It can crawl across your directories and make redundant updates to files that would otherwise be time consuming and tedious. In more complicated situations, Python can be used to automate repetitive tasks that are still somewhat complex, and save you hours of time. Best of all, Python is a language that you do not need to master before you start reaping the benefits. With a little effort to learn the basics, you will have the tools you need to start making a difference for yourself and the people you work with.

For any and all Python scripts discussed in this paper, please feel free to reach out and I will provide you with the Python code and some notes for installation.

CONTACT INFORMATION

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

Michael Stackhouse
Covance, Inc.
Michael.Stackhouse@Chiltern.com