Paper SAS4141-2020

The History and Evolution of SASPy, Including an Overview of What It Can Do and How to Use It

Tom Weber, SAS Institute Inc.

ABSTRACT

SASPy is one of the most popular SAS® open-source packages, available on the SAS® GitHub site (https://github.com/sassoftware/saspy) and also installable via Python Package Index (PyPI) and Anaconda. But, how did it come to be? How and why did SAS create an open-source Python interface to your existing SAS® 9.4 systems, and without even needing any updates to your existing SAS servers for it to be able to connect to them? Often, having an understanding of the design criteria and constraints of a software package helps you better understand and use it, so you can be more productive in what you are really trying to accomplish. This session blends in this insight while giving an overview of SASPy, including the various ways it can connect to your different SAS installations, how to configure it for each one, a walk-through of the core functionality, and a look at some of the more advanced features. The goal is that you end up with a much better understanding of how to use SASPy to accomplish even better combined Python and SAS workflows for your projects.

HISTORY AND EVOLUTION

SASPy originated with one unassuming question; "Hey Tom, do you know how to make Python talk to SAS?" Jared Dean (the other SASPy co-author) had been tasked with getting a 'notebook' to interface to SAS. These notebooks were the new thing that were becoming popular with data scientists, students, and other programmers. Jared had narrowed it down to Jupyter (IPython) Notebook to start with, and because it was implemented in Python, he needed an interface from Python to SAS.

I hadn't written any Python before, although I had just recently prototyped embedding Python into a C-code system, so I prototyped my idea of an interface in C. How backwards that now feels, because I have since used Python to very quickly prototype some C interfaces I've written! The first constraint I needed to deal with was that I needed a Python interface to the existing production SAS systems in the field. That means code couldn't be added to the SAS side in order to build a client-server interface. Because I'm more of a 'line mode' guy than a 'clicky-pointy' interface guy, I got the idea of connecting to SAS in line mode (-nodms), in the same way I do when running SAS in a Linux terminal from the command prompt.

```
tom64-5> sas -nodms

NOTE: Copyright (c) 2016 by SAS Institute Inc., Cary, NC, USA.

NOTE: SAS (r) Proprietary Software 9.4 (TS04.01M6D11062019)

      Licensed to SAS Institute Inc., Site 1.
[…]
NOTE: SAS initialization used:
```

```
      real time           1.31 seconds

      cpu time            0.17 seconds

  1? data a; x=1; run;

NOTE: The data set WORK.A has 1 observations and 1 variables.

NOTE: DATA statement used (Total process time):

      real time           0.19 seconds

      cpu time            0.02 seconds

  2? proc print;run;

          The SAS System             1

              14:24 Thursday, November 7, 2019

          Obs    x

          1     1

NOTE: There were 1 observations read from the data set WORK.A.

NOTE: PROCEDURE PRINT used (Total process time):

      real time           0.57 seconds

      cpu time            0.21 seconds

  3?
```

I figured that if the Python process created a subprocess running SAS and connected up standard input (STDIN), standard output (STDOUT), and standard error (STDERR) in the same way as the terminal itself, then I could submit code by writing to STDIN, get the LOG by reading STDERR, and get the List and Results by reading STDOUT. Simple! SAS won't have any idea it's being talked to from Python, and it won't require any extra code on the SAS side to make this work.

I took the C program that I used for this and turned it into a C-Python module (extending python with C code), which I could then 'import' into Python. With three simple functions [submit(), getlog(), getlst()], I started learning Python enough to start building out the beginnings of what is now SASPy. I wrote Python functions to start calling this module, and Python was submitting SAS code and getting the output back. Soon after, I learned that Python itself has lower level operating system (OS) interfaces (such as fork, exec, pipes, and sockets), and I replaced the C-Python input-output interface module with Python source code! This was the real deal: no C code, no compile, no link edit, no shipping executables. This was an actual open-source interface to SAS! At this point I had a proof of concept.

Also worth mentioning is that, at this point, I had this simple interface working with both Python2 and Python3. As I continued, I found that some of the lower-level interfaces didn't work as well and that support for some functionality wasn't there in Python2 as it was in

Python3. I eventually gave up on trying to keep Python2 support and just committed to Python3 for SASPy.

The next constraint I imposed was to use only the Python standard library, as opposed to having any requirements on other third-party packages to implement functionality. I've held true to that principle, and I have no requirements upon anything that's not in the standard Python library. That's not to say other things can't be used. Pandas, for instance, is integrated in SASPy so that you can get tabular results as DataFrames and you can move data between SAS and Python using DataFrames. But SASPy isn't dependent upon Pandas. You can do almost everything in SASPy without having Pandas installed. Another example is IPython, which is needed in Jupyter Notebooks but not required nor needed outside of that.

Because Python is an object-oriented language, the first architectural decisions came next. A SASsession object, which represents the SAS session that Python is connected to, was the first thing needed. The next immediately obvious object was a SASdata object. This object would have methods that you would expect for interacting with a data set, while the more global methods would be on the SASsession object. Once I had these two objects and a handful of methods on each, the need to transfer data between SAS and Python was then next necessary requirement in order to have a real working prototype. Without data transfer, you really couldn't interface Python and SAS together in a single workflow; it would still be an either-or thing. sd2df() and df2sd() were written next. Once these were functioning, we were in business; we had a prototype! This was June 2015.

At this point I went back to Jared and showed him what I had come up with. The next thing was to add the ability to do analytics. Jared is the data scientist, so he figured we would want analytic objects and an analytic results object. The analytic methods would be the SAS procedures and the results would be the various tables, plots, graphs, and so on, that the procedures produce. Because this was for a web-based notebook, ODS HTML5 output would render these results perfectly. ODS HTML5 was the next thing to be able to return from SAS. Jared got the analytic methods, results, and ODS going while I continued filling out the SASsession and SASdata objects and working on the STDIO access method.

After we had these 4 components working, the next step in the evolution was adding support for STDIO over secure shell (SSH), so that Python and SAS didn't need to be on the same machine. Also, because everything I'd done so far was only on Linux, I looked into trying to get this working on Windows. Unfortunately, support for running SAS in -stdio mode was removed many SAS versions ago, so my STDIO access method would not work with SAS installed on Windows. At this point we were only on Linux, although we did have support for running in Jupyter (HTML results), in a terminal (text results), and in a batch Python script (batch mode), with a local or remote SAS install.

Next, Jared worked on creating a Jupyter kernel to run SAS language code - the SAS Kernel. This allowed the SAS language to be executed in a Jupyter Notebook. It imported and ran SASPy under the covers, using the submit() method to run the SAS code, and it returned the results (LST) or the LOG if there were no results or there were errors. Now you could interact with SAS from either a Python language notebook, or a SAS language notebook!
It was at this point that we were allowed to create our first open-source projects on GitHub with these two pieces…almost, anyway. The SAS Kernel was approved, and it was dependent on SASPy, but SASPy itself was a bit ahead of its time and was not approved to go out. So, I stripped SASPy down to have only the submit() method. I could then also put SASPy out on GitHub (July 2015) so that the SAS Kernel could use it. Eventually, SASPy (the real SASPy) was approved and V2.1.0 was published to the SASPy GitHub site, with full functionality (March 2017).

We continued implementing new functionality and fixing any issues we encountered, but the missing support for running from a Windows client was the next big problem to solve. Our Integrated Object Model (IOM) client-server interface was the obvious choice, and that would open up a much wider range of SAS deployments that we could support. But, IOM has no Python client interface. It has a Component Object Model (COM) interface for Windows, and a Java interface that runs on any client platform. It also has a C interface, but that would preclude having a source-only package. So, I decided on the Java IOM client (I publish the Java source code in the repository). Now had to figure out how to interface Python to Java, because that didn't already exist, not as I needed or wanted it anyway. I created my own interface, and after that was accomplished we had two access methods that allowed for running from any client OS and for connecting (local or remote) to virtually any SAS session, including SAS® Grid Computing and any other deployment based on the workspace server.

The HTTP access method is the third one I've written. It is for using the Compute Service in SAS® Viya®, which is a restful interface (HTTP) that provides access to the SAS 9.4 code in a SAS Viya deployment (referred to as SPRE). The Compute Service in SAS Viya is logically like what the IOM client is in the SAS 9 world. Interestingly, I wrote the HTTP access method before IOM. But because the Compute Service didn't go out until much later, I ended up writing and delivering IOM first.

There is one more access method named COM. This access method was completely user-written and submitted to SASPy as a pull request (PR) through the SASPy GitHub site! It uses the IOM COM (Windows only) client. This and many other contributions really shows that SASPy is an open-source project. An access method is no trivial amount of code. Not only do Jared and I implement SASPy, but we also incorporate community-written functionality and fixes. And much of the functionality I've implemented has been based on requests from users. So even if they didn't write it, the community has shaped the content of SASPy in many ways.

CONFIGURATION

Having a number of different ways to connect to a large variety of SAS deployments, configured in many different ways, imposes a configuration. I created a file that is an importable module that made accessing the configuration information easy, and even allowed for some cool extensibility. The sascfg.py file started out pretty simple. In the beginning, about all I needed was the path to the SAS start-up script, and that was it! The main component of the configuration file is a configuration definition. This is just a Python dictionary that contains key-value pairs of options. Because there can be more than one SAS deployment to connect to, or more than one set of options and values to use for different connections, having more than one configuration definition was necessary. Se we use SAS_config_names, which is a list of the configuration definition dictionary names. Add in a SAS_config_options list, for the occasional global option, and you're set.

There are a number of common configuration options that can be used for any access method. Beyond those, each access method has its specific options. These are all documented on the GitHub site: https://sassoftware.github.io/saspy/install.html#configuration. When I got past the first STDIO access method, my sascfg.py file quickly started to grow. But with this simple structure, I can easily add and support any number of configuration definitions and also manipulate things at runtime. Any of the options in the configuration definitions can be

specified or overridden at run time on the SASsession() method. It's really quite convenient. At the time of this writing, this is the list of configuration definitions:

```
SAS_config_names = ['default', 'SASgrid', 'http', 'httptest', 'ssh',
                    'sshtun', 'sshrtun', 'httpfred', 'grid',
                    'tdi', 'tdilat','iomj',
                    'iomc', 'iomjwin', 'winiomj', 'winiomjwin', 'winlocal',
                    'gridiom', 'wingridiom',
                    'zos', 'zos2', 'winzos', 'winzos2', 'sshtest', 'sshloc',
                    'sdssas', 'saskr', 'vb010',
                    'vb015', 'pune', 'notpune', 'iomkr', 'httpviya',
                    'kevin', 'issue176', 'httpjason', 'itviya', 'carrie1',
                    'console',
                    'iomcom','iomcom_loc', 'iom4', 'm5bug', 'spre34'
                   ]
#SAS_config_names = ['sdssas'] # STDIO
#SAS_config_names = ['iomj']   # IOM
#SAS_config_names = ['itviya'] # HTTP
```

You don't want to see all those dictionaries! Whenever I need to run unit tests, which connect to many sessions and would prompt for which configuration definition to use, I just uncomment one of the SAS_config_names= lines with only one value (the commented out lines above) for the specific access method I want to run the test against, and it's all good. The rest of the file still has everything in it, but because only one of the configurations is listed in SAS_config_names, SASPy just uses that configuration without prompting, and without having to include the cfgname= on the SASsession() in the unit tests.

The autocfg.py file is another cool feature of this configuration file. It generates a configuration file for a local install on Windows.See https://github.com/sassoftware/saspy-examples/blob/master/SAS_contrib/autocfg.ipynb. It uses one of the extensibility tricks, by being an imported module, to get around the requirement of having to have the path to the sspiauth.dll file in your Windows System path (which is an IOM requirement). Instead, it just generates Python code to add that path to the local environment variable for the Python processes. It's added dynamically and found correctly, with no extra work needed, to add it to the Windows System path. This happens when the configuration file is imported, which is when SASPy is imported.

So why do I keep saying sascfg.py when we all know it's sascfg_personal.py? Well, on day one it was sascfg.py, but the new configuration file from the repository overwrote my version with all of my configurations the first time I made a change to the sascfg.py file in the GitHub repository and then installed that version! So, we clearly needed a different file to really use, while keeping the one in the repository to show the examples of the various

ways to configure the different access methods. So sascfg_personal.py was born, and since then I've never lost any of my configurations.

That brings me to another design constraint I've always had, which is having no breaking changes. We always support backwards compatibility. So far, I've held to that and have added only new features or enhanced things in a way that doesn't change existing behavior. In the years SASPy has existed, and through the sheer volume of enhancements over that time, having never needed to change anything in a way that would break previous functionality, I think speaks to the simple, clean, and solid architecture that we came up in the very beginning.

And, with no breaking changes, sascfg.py is still the last resort to use, if no _personal version exists. In fact, 'default' in sascfg.py is what is still used in SAS® University Edition! That's backwards compatibility.

CHOOSING AN ACCESS METHOD

How do you know how to configure SASPy? Well, the first thing is to know is what SAS instance you are trying to connect to. Next is where are you connecting to it from - what client are you running SASPy on? These two answers will dictate which access method you will use and thus what your configuration definition will contain.

1) What kind of SAS deployment, and where?
    a. Stand-alone SAS 9 install
        i. On Linux
            1. Client Linux
                a. STDIO - over SSH if not the same machine
            2. Client Windows
                a. Can't get there from here
        ii. On Windows
            1. Client Linux
                a. Can't get there from here
            2. Client Windows
                a. IOM or COM - on same machine. Can't get there if different machines
    b. Workspace server (this is SAS 9, and deployment on any platform is fine)
        i. Client Linux
            1. IOM - local or remote
        ii. Client Windows
            1. IOM or COM – local or remote
    c. SAS Viya install
        i. On Linux
            1. Client Linux
                a. HTTP – must have compute service configured and running
                b. STDIO - over SSH if not the same machine
            2. Client Windows
                a. HTTP – must have compute service configured and running
        ii. On Windows
            1. HTTP – must have compute service configured and running

Now you can go to the access method specific configuration page in the documentation to see what you'll need for your configuration definition: https://sassoftware.github.io/saspy/install.html#configuration. There are also example configuration definitions in the example configuration file, sascfg.py, showing each of these different cases.

USAGE

After you have SASPy configured, you can start using it. There are a couple of helpful things that you can do to understand what's going on and help you diagnose any problems. You should start by reading the installation and configuration documentation, which will eliminate 90% of any problems you might run into. If you still have a problem running, read the Troubleshooting section of the documentation: https://sassoftware.github.io/saspy/troubleshooting.html. This section shows you many of the errors that you might experience if you have something wrong with your configuration and tells you how to fix it. If you can't find the answer there, open a new issue on the GitHub site (https://github.com/sassoftware/saspy/issues) and I'll help you figure out what's wrong and get you up and running. You can even search the issues to see if that's already been addressed.

These are the first things you should do even if you aren't having problems. First, submit the various SASPy objects to see information about them. For instance, after importing SASPy, submit 'saspy' and you will see the actual path that this module is being loaded from.

```
>>> import saspy

>>> saspy

<module 'saspy' from '/opt/tom/github/saspy/saspy/__init__.py'>

>>>
```

Next, you can verify which sascfg_personal.py file will be used, and which ones exist that could be used. Of course, you can also explicitly specify one by using the cfgfile= option on SASsession() method if you want.

```
>>> saspy.SAScfg

'/opt/tom/github/saspy/saspy/sascfg_personal.py'

>>> saspy.list_configs()

['/opt/tom/github/saspy/saspy/sascfg_personal.py',

'/opt/tom/github/saspy/saspy/sascfg_personal.py',

'/usr/sastpw/.config/saspy/sascfg_personal.py']

>>>
```

Now you know where SASPy is being run from and which configuration file it will use by default. You won't believe how many times this simple exercise has explained why things

weren't working as expected! Now create your SASsession and submit that object to see about it.

```
>>> sas = saspy.SASsession()
Please enter the name of the SAS Config you wish to run. Available Configs
are: ['default',

'SASgrid', 'http', 'httptest', 'ssh', 'sshtun', 'sshrtun', 'httpfred',
'grid', 'tdi', 'tdilat', 'iomj', 'iomc', 'iomjwin', 'winiomj', 'winiomjwin',
'winlocal', 'gridiom', 'wingridiom', 'zos', 'zos2', 'winzos', 'winzos2',
'sshtest', 'sshloc', 'sdssas', 'saskr', 'vb010', 'vb015', 'pune', 'notpune',
'iomkr', 'httpviya', 'kevin', 'issue176', 'httpjason', 'itviya', 'carrie1',
'console', 'iomcom', 'iomcom_loc', 'iom4', 'm5bug', 'spre34'] sdssas
SAS Connection established. Subprocess id is 25365
No encoding value provided. Will try to determine the correct encoding.
Setting encoding to iso8859_15 based upon the SAS session encoding value of
latin9.
>>> sas
Access Method         = STDIO
SAS Config name       = sdssas
SAS Config file       = /opt/tom/github/saspy/saspy/sascfg_personal.py
WORK Path             = /sastmp/SAS_work2DFA00006332_tom64-5/
SAS Version           = 9.04.01M6D11072018
SASPy Version         = 3.1.7
Teach me SAS          = False
Batch                 = False
Results               = Pandas
SAS Session Encoding  = latin9
Python Encoding value = iso8859_15
SAS process Pid value = 25394
>>>
```

This information can be very helpful, especially when trying to diagnose any issues. You can do the same with SASdata objects. It will show what table they refer to what, if any, data set options are associated with them, and how any results will be returned.

```
>>> cars = sas.sasdata('cars','sashelp')

>>> cars

Libref  = sashelp

Table   = cars

Dsopts  = {}

Results = Pandas

>>>
```

Get in the habit of submitting these objects when they are created, and you will always know what is going on in your program. The next things to make use of are the log methods. If you run something and don't get what you expect, look at the LOG (the SASLOG)! There are two methods for this, saslog(), which shows the whole log at that point in time, and lastlog(), which shows the part of the log from the last code that was submitted. lastlog() is convenient when you want to see only last thing submitted. As an example, you might have run cars.head() and nothing came out. Why? What happened? Submit print(sas.lastlog()) and see if there was an error. This way, you don't have to scroll to the end of a possibly long log to see thet last step. However, you can always see the session log via: print(sas.saslog()).

SPECIFIC FEATURES

Finally, let's go over a few other topics worth mentioning.

RESULTS

The results= parameter is an option for specifying how you want to get tabular results back to Python from SAS. The choices are as Pandas DataFrames, as HTML, or as text. Graphs, plots, and other non-tabular results are returned only as HTML. Results can be set globally on the SASsession object. It can be set independently on each SASdata object, although it will default to whatever the session is set to. You can also set it on the submit() method, although Pandas is not an option here, because there's no telling what kinds of output might be produced from any code you submit. Only HTML or text are supported for the submit method.

DATA SET OPTIONS (DSOPTS=)

The SAS language allows you to provide some options when processing a SAS data set. These data set options allow you to subset, augment, or otherwise transform the data for that specific step. The SASdata object has an attribute where you can specify many of these options so that they are applied by methods that act on the object. This attribute is 'dsopts', and it is a Python dictionary. The supported options are where, keep, drop, obs, firstobs, and format. You can set, change, or remove any of these on the fly, but whatever is in the dsopts when you interact with the SASdata object will be applied within that method.

DISPLAY

Because SASPy was originally developed with the idea of using it in Jupyter Notebooks, the way it rendered HTML was specific to the IPython module's methods for rendering. As other notebook platforms have been developed and users have asked me to support them, I've reworked the way HTML rendering is done in SASPy, so it's a swappable interface. SASPy now supports three different notebook interfaces, each having a different way to display HTML output. Jupyter is supported, of course, but Zeppelin and most recently Databricks are also supported. You use the 'display' configuration option to specify which platform you're running on. The default, of course, is Jupyter.
There are two helper methods in SASPy to allow you to render HTML results that are returned to you programmatically (rather than being rendered for you), such as from the submit method or when you are in batch mode. These helper methods are DISPLAY() and HTML(). Depending on the notebook, you use sas.DISPLAY(sas.HTML(results['LST'])) or just sas.HTML(results['LST']) after submitting results = sas.submit(code).

SUBMIT

SASPy has many methods that allow you to interact with SAS without even knowing any SAS programming code. However, there is no way to provide a comprehensive SAS language interface in Python. So, there needs to be a way for you to submit your own hand-written code to be processed. That's the submit() method. The submit method executes the code that you provide and returns a Python dictionary containing the LOG and the LST (whatever results were produced). The LOG can simply be printed to display its contents, while the LST is in an HTML document (by default), so rendering it is notebook specific (as described in the previous section).

There are now two other versions of the submit method, submitLOG() and submitLST(), which display the LOG or LST outright for you, so you don't get a dictionary back and have to do it yourself. The displayLST() method has an extra method= parameter that allows you to fine tune it to meet your needs. By default, (method='listonly') renders the LST, including 'nothing' if there was no output. 'listorlog' renders the LST if there is output, otherwise it renders the LOG. 'listandlog' renders the LST followed by the LOG, and 'logandlist' renders the LOG followed by the LST.

DATA TRANSFER

As I mentioned, even before calling the code a prototype, I needed to be able to transfer data back and forth between SAS and Python. The sasdata2dataframe (sd2df) and dataframe2sasdata (df2sd) methods were written to do that. There have been several enhancements to these methods since the first implementation. There is still only one df2sd, although it has a few parameters for special case data, such as support for embedded newline characters in character columns. However, the sd2df method has evolved more. The first major enhancement came early on, when trying to pull over large data sets. The original implementation streamed the data over and cached it in a Python list of rows that was then used as input to create the DataFrame. That worked great for data that wasn't too big, but not for large data. 'Chunking' that algorithm helped address that for medium-size data, but it still didn't scale due to object churn and memory thrashing in Python.

The CSV (comma-separated values) version was created so that the data could be cached on disk and then turned into a DataFrame using a much faster Pandas routine to read the file. This version used PROC Export from SAS to create a .csv file that was then imported with pandas.read_csv(). This addressed the performance issue, but it still had some issues with specific types of data. You can't correctly parse a comma-separated file when you have

embedded commas in character columns. There are options to change the delimiter, but not options to handle other data issues.

Due to CSV limitations, the DISK version was most recently implemented. It's a hybrid of the two. It uses the original algorithm, which has more control over how the data is streamed over and delimited, in order to handle special characters like embedded delimiters, NULLs, New Lines, and other things. The DISK version writes the data to a disk file that is then used to create the DataFrame, as in the CSV version, instead of trying to generate the DataFrame in memory as it reads the data. This method has the performance closer to the CSV version with the special data support of the original MEMORY version.

ANALYTICS

An interface to SAS wouldn't be complete without analytic methods! SASPy has a number of analytic objects, which correspond you SAS products. These products, like SAS/STAT®, each have a number of procedures that perform various analytic functions. So, the methods for each of these analytic objects correspond to a procedure. Each procedure produces some number of results, such as various tables, graphs, plots, and so on. The Analytic Results object is what these methods return and is what 'contains' these various output results: *e.g.* result = stat.reg(model, data). You can render each different output simply by submitting the result.NAME, where NAME is the result name (hint: submit dir(result) to see what results there are), or you can render all of them by submitting result.ALL().

RUNTIME ENVIRONMENTS

When SASPy was just a prototype, I knew there were three different environments in which it needed to be able to run. Jupyter was one, but that wasn't what I used to develop it. I ran it from the command line in a Linux shell, Which I call line mode. That environment is where SASPy was really developed and where text results came from. The third environment was, of course, in a Python batch script. Python has been used as a scripting language by systems developers since long before the term 'data scientist' was uttered for the first time. Allowing SASPy to be used effectively in a batch script and in line mode, rather than only in a notebook, was another original constraint.

When set to batch mode, by using set_batch(True), all the methods that would otherwise render their output now return that output as a Python object that you can interact with programmatically. This is similar to how the submit method returns a dictionary. In this way, you can programmatically process even HTML plots and graphs that are returned by analytic methods. You can write out HTML results to files, which then render correctly just by opening them, since they are complete HTML documents.

FILESYSTEM INTERACTION

It wasn't until a user opened an issue asking if SASPy could navigate the filesystem on the SAS server side, to which I answered "No, it doesn't. But that sounds kinda cool, so hang on …", that I wrote some methods for doing this. dirlist() takes a path and returns a list containing the files and directories (in that path. Directories end in a '/' or '\' so that you can tell they are directories. fileinfo() gets the OS information about the specified file, which varies with the OS. With just these two methods, you can programmatically navigate and interact with the filesystem. I then implemented upload and download methods so that you could do binary transfer of files between the client and server sides. This opened a lot of options for doing some really cool things!

CONCLUSION

Creating SASPy has been a very interesting endeavor, to say the least. I hope that getting a bit of backstory on SASPy helps give you some understanding of it, which then helps you use it more effectively. There is no magic in software, so when something that you're using seems or feels like magic, it's time to look behind the curtain and get a better understanding. Even though this program is open source, not everyone can gain that insight from reading the code, so I hope this paper helps in some way.

RESOURCES

Here are some helpful links to SASPy resources.
SASPy Repository: https://github.com/sassoftware/saspy

SASPy Documentation: https://sassoftware.github.io/saspy/index.html

SASPy Examples: https://github.com/sassoftware/saspy-examples

To request help with SASPy, just open an issue on the SASPy GitHub site here:

https://github.com/sassoftware/saspy/issues

Demo Notebook: https://github.com/sassoftware/saspy-examples/blob/master/SAS_contrib/SGF_2020_Demo1.ipynb

CONTACT INFORMATION

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

Tom Weber
SAS Institute Inc.
Tom.Weber@sas.com