Paper 3238-2019

A Complete Introduction to SASPy and Jupyter Notebooks

Jason Phillips, PhD, The University of Alabama

ABSTRACT

Thanks to the welcome introduction and support of an official SASPy module over the past couple of years, it is now a trivial task to incorporate SAS® into new workflows by leveraging the simple yet presentationally elegant Jupyter Notebook coding and publication environment, along with the broader Python data science ecosystem that comes with it. This paper and presentation begins with an overview of Jupyter Notebooks for the uninitiated, then proceeds to explain the essential concepts in SASPy that enable communicating seamlessly with a SAS session from Python code. Included along the way is an examination of Python DataFrames and their practical relationship to SAS data sets, as well as the unique advantages offered by bringing your SAS work into the Notebook workspace and into productive unity with the broad appeal of Python's syntax.

INTRODUCTION

The past several years have seen the introduction of a number of new pathways for integrating SAS® technologies and platforms with other languages and tools that are familiar to open-source data scientists, particularly with respect to the programming language Python. Yet given the growing array of new libraries and components that are now potentially relevant to the SAS analyst who wishes to integrate with Python (Jupyter, SASPy, SWAT, Pipefitter), it is perfectly forgivable to find oneself unclear as to the possibilities available, or uncertain of the practical starting points. This paper aims to provide an introduction to the first line of integrations that are likely to have the broadest immediate audience and benefit. These are, primarily, Jupyter notebooks and SASPy, which together offer a complete foundation from which to begin taking advantage of many new patterns that Python integration can bring to SAS developers of any level.

This paper begins with a brief overview of the new platforms (Jupyter Notebook) and packages or modules (SAS Kernel for Jupyter, SASPy) that represent the primary entry points with which one needs to be familiar. Along the way, a simple briefing on the utility of Python and the distinct benefits of Jupyter will be provided for those to whom these remain foreign terms. After the walkthrough of these technologies, a few additional, practical benefits to the connection between Python and SAS will follow in the concluding remarks, as well as a nudge towards further libraries that make up the current landscape of SAS and Python.

BACKGROUND: PYTHON AND JUPYTER NOTEBOOKS

While they share a background foundation, the two primary tools to which this paper will draw attention (SASPy for writing Python code to interact with SAS, and Jupyter Notebook as a programming interface) are not strictly coextensive. In fact, you might only decide to directly utilize one or the other. A bit of clarity on the nature of Python and the purpose of Jupyter Notebook is therefore in order.

PYTHON

A lengthy introduction to the Python language would be beyond the scope of this paper, yet a sense of its present position within data science will prove useful to the content that follows. Of the many prominent programming languages that are widely employed in communities of scientific or statistical computing and research, Python's distinct contributions may best be characterized in terms of its clarity of syntax, its broad utility for general purpose computing or application development alongside data work, and its well-developed pathways to efficient mathematical computation (the latter particularly by way of several popular packages like NumPy and Pandas).

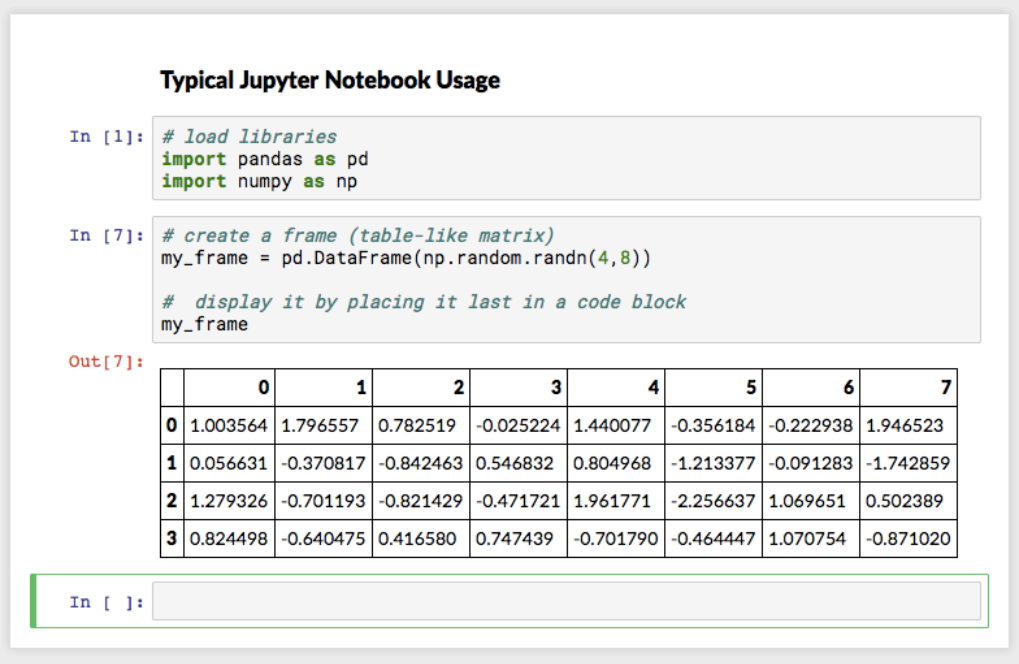
A notable area in which Python continues to find favor is machine learning and neural networks. To understand why that is the case, it bears recognizing that many of the popular tools for this area of functionality are not strictly implemented in Python code alone; often they merely take advantage of the clean syntax of the language in order to open an accessible interface to a framework that runs in lower-level code. The package Tensorflow, to take a prominent example, allows for Python to create computational graphs for machine learning that are ultimately executed using highly efficient low-level code written for optimization on GPUs or distributed systems. This pattern, in which Python acts as a developer-focused interface to code that will be executed at a lower level, is also the manner in which one will use a tool like SASPy to execute work in a local or remote SAS environment. To employ Python as a bridging language is therefore well in line with its other major uses in data science.

JUPYTER NOTEBOOKS

Jupyter Notebook (formerly known as iPython Notebook) offers an integrated environment for interactive programming, which simply means that the user can write and execute code within a single interface, as well as display many kinds of output directly inline with blocks of code. In this way, the coding, execution, and final report occupy a single cohesive "notebook" that can be distributed for others to view or to download and execute in their local environment. Jupyter is web-based in the sense that its user-facing application runs within a standard web browser, yet its most typical usage is entirely local to a single machine, which it orchestrates by creating a background process on your workstation that communicates with the web front-end via a local port. (Shared deployments for running the background process on a remote server do however exist, and further information on these developments can be found at the Jupyter Project homepage, linked below under Recommended Reading.)

While the Jupyter project grew out of Python and uses it under the hood, its notebooks can use a number of different languages, including R, Scala, Java, and even Base SAS®. In order to use additional languages you must install a "kernel" that executes your code in another process, with Python acting as a bridge. Most kernels additionally provide syntax highlighting or other editing features like inline documentation and auto-completion.

Figure 1 displays a typical notebook, with a few blocks of code and output.



Typical Jupyter Notebook Usage

In [1]:
```
# load libraries
import pandas as pd
import numpy as np
```

In [7]:
```
# create a frame (table-like matrix)
my_frame = pd.DataFrame(np.random.randn(4,8))

#  display it by placing it last in a code block
my_frame
```

Out[7]:

	0	1	2	3	4	5	6	7
0	1.003564	1.796557	0.782519	-0.025224	1.440077	-0.356184	-0.222938	1.946523
1	0.056631	-0.370817	-0.842463	0.546832	0.804968	-1.213377	-0.091283	-1.742859
2	1.279326	-0.701193	-0.821429	-0.471721	1.961771	-2.256637	1.069651	0.502389
3	0.824498	-0.640475	0.416580	0.747439	-0.701790	-0.464447	1.070754	-0.871020

In [ ]:

Figure 1: **A minimal Jupyter notebook running Python**

The title header in this example was written into the notebook using Markdown (a popular markup language for writing rich text), a feature incorporated directly into Jupyter notebooks (along with LaTeX support) that can easily be further leveraged for writing well-presented documentation alongside code blocks. Often this may be used for writing a paper-length treatment of a technique, with code examples and output interspersed throughout a body of lengthier prose sections. The title is followed in the screenshot by a couple blocks of code (in Python, in this case) and an output block, which displays the results of the prior cell. Interactive visualizations are also possible within the notebook.

A notebook can be quickly published to the web on platforms like Github as a single, self-contained file, which will display the code and output in a static form for online reading; interested parties can then download the notebook file for execution or further development on their local machine. As a consequence of this built-in capability to serve both as a complete coding environment and as a medium for teaching or sharing knowledge, Jupyter notebooks are popular in a number of online communities working with data science.

INTEGRATIONS: SASPY AND SAS KERNEL FOR JUPYTER

With an understanding of the role of Python and Jupyter Notebook in hand, the next step is to examine the two primary integrations by which you will be interacting with these tools. The SAS kernel for Jupyter makes it possible to write and execute SAS programs within a Jupyter Notebook, gaining in effect a new interface for SAS programming while requiring no alteration in code. SASPy, on the other hand, enables you to write Python code that effectively controls a SAS session, opening up the capabilities of SAS to be controlled entirely via code written in Python, and automatically handling conversion between Python and SAS data structures.

SAS KERNEL FOR JUPYTER

SAS kernel for Jupyter is a Python package developed by the SAS Institute that enables SAS to be used as a kernel for Jupyter notebooks. It works by connecting the Jupyter environment to an interactive SAS session. Note that this package does not contain a SAS installation, and depends upon having a licensed and installed SAS instance available. However, it can be configured to connect to SAS in several ways:

- A local instance of SAS running on the same machine;
- A remote instance of SAS running on Unix that is accessible by SSH;
- SAS® Viya by way of the Compute Service.

Attaching the SAS kernel to a Jupyter notebook means that the entire present notebook (which corresponds to a single program or script) will accept only SAS code blocks; this contrasts with the option to write code that alternates between uses SAS and Python, which the SASPy package addresses (see the section following this one).

Some SAS users may have first encountered the SAS kernel for Jupyter within SAS® University Edition, where it fits the educational goal of the learning edition by providing a more familiar interface to those who are likely to have already seen notebook-like interfaces in other data science contexts. Writing and coding within a notebook that is using the SAS kernel should likewise be a painless adjustment for existing users of the Base SAS programming language, with the primary change being that the log and output are both displayed inline between code blocks. See Figure 2 for a simple example.

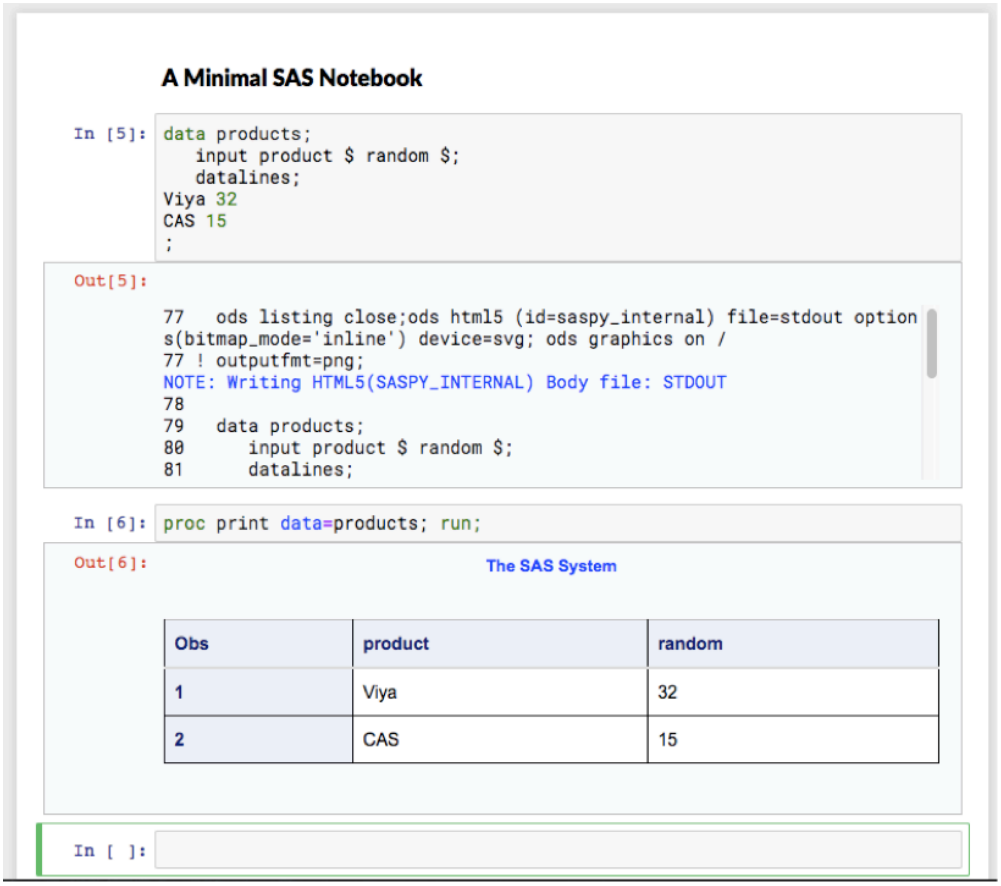


Figure 2: A minimal Jupyter notebook using the SAS kernel

The notebook in this screenshot follows the same flow as the Python notebook shown earlier, yet with ordinary SAS code written in the code blocks, and both the output and the log from a connected SAS session showing up after each section of code. This connected sequence of the log, output, and code illustrates the principal difference between using the SAS Windows application for developing code and using the Jupyter Notebook: here, all your work is executed and displayed inline (along with any accompanying write-up, if desired), so that the resulting page that is produced can act as a total report of the project's code, log, and output—and is distributable as a single file, or can be published to the web in a format that perfectly contains a snapshot of your work.

The ability to run SAS as a kernel within a Jupyter notebook opens up many of the key advantages of the notebook platform, particularly if sharing code and publishing to the web is desired for educational purposes. However, one achieves an even wider range of potential integrations by using the SASPy package directly to facilitate systematic communication and data sharing between the Python language and SAS.

SASPY

SASPy is a Python package that provides part of the underlying communication between Jupyter and SAS when using the SAS kernel; however, it can also be used directly (within Python code apart from the Jupyter environment, or within Jupyter notebooks that are written in Python), and has significant capabilities beyond the essential function of relaying SAS code and output.

At its core, SASPy is capable of creating a SAS session and sending code to it for execution, as well as returning the output (logs and other output) to the controlling Python script. Yet it is also a powerful generator of SAS code, which means that it offers methods, objects, and syntax for use directly in idiomatic Python that it can then automatically convert to the appropriate SAS language statements for execution. In most cases, SAS procedures or steps are mapped directly to Python methods as a one-to-one equivalent.

To understand where SASPy fits between Python and SAS, consider Figure 3.

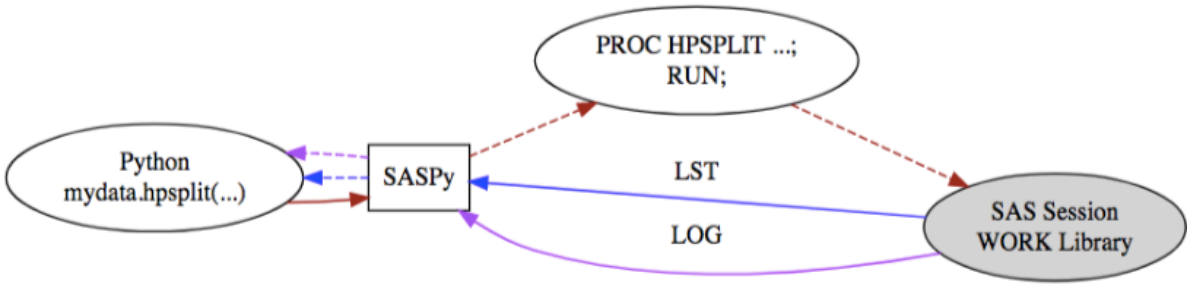


Figure 3: From Python Code to SAS and Back

The red arrows show how a Python method ("hpsplit") called on a dataset reference ("mydata") is understood by SASPy, which generates and sends corresponding SAS code (HPSPLIT Procedure) to a controlled SAS session, and then acts as a middleman to receive both the log and main listing output for reflection to the user.

SASPy achieves this integration atop shared dataset references which act as interfaces between the two environments. On the Python side, this works by way of the popular Python package Pandas, which offers a useful data and matrix abstraction called a DataFrame. A DataFrame can be created in Python and sent to SAS, after which Python retains a reference to the remote dataset for calling methods on it; additionally, you can retreive any dataset in the SAS session as a local DataFrame on the Python side.

The quickest way to gain an understanding of the typical workflows enabled by SASPy is to illustrate, in a trivial example, the full round-trip that a dataset can take between the Python language (with data stored in a Pandas DataFrame) and SAS (manipulating a corresponding data set stored in an active SAS session). Note that the lines below beginning with a hash ("#") are comments:

```
# initial library imports: SASPy and Pandas

import pandas
import saspy

# connect to a SAS session
# 'my_server' here references a connection name specified in
# the local SASPy configuration; see SASPy documentation

sas = saspy.SASsession(cfgname='my_server')

# let's say that python generates or loads some data;
# here, it loads data from a local csv using Pandas

my_dataset = pandas.read_csv("./my_local_data.csv")

# now `my_dataset` references a pandas DataFrame local to Python;
# we can send it to a SAS session using one step

# df2d stands for dataframe2sasdata
my_sas_dataset = sas.df2sd(my_dataset)

# `my_sas_dataset` now references a data set stored in the SAS session;
# yet we can manipulate or explore it using Python methods.
# Here, a simple sort is applied, which will execute in SAS

my_sas_dataset.sort('group')

# and now we can pull the altered data set back into a local DataFrame
# for further use in Python

# sd2df stands for sasdata2dataframe
my_sorted_set = sas.sd2df(my_sas_dataset.table)
```

A couple additional notes on the code above will sharpen attention to the fundamental elements:

1. The paired methods **df2sd** and **sd2df** are the principle means by which a data set may be transferred between SAS and Python. On each destination, the data types, column names, and other basic elements will be retained; some additional metadata unique to SAS data sets may however be dropped on the Python side.
2. When a data set is located on the SAS side, the Python variable (**my_sas_dataset** above) acts as a reference with various attached methods. Here, when Python code invokes the **sort** method on this reference, SASPy tells the SAS session to run the corresponding procedure on the linked data set (in this case, the SORT procedure).

The above code need not be run within Jupyter, although combining SASPy with Jupyter grants convenient display capabilities, allowing for tables or visualizations to be output immediately, as the brief notebook excerpt in Figure 4 illustrates by invoking a histogram.

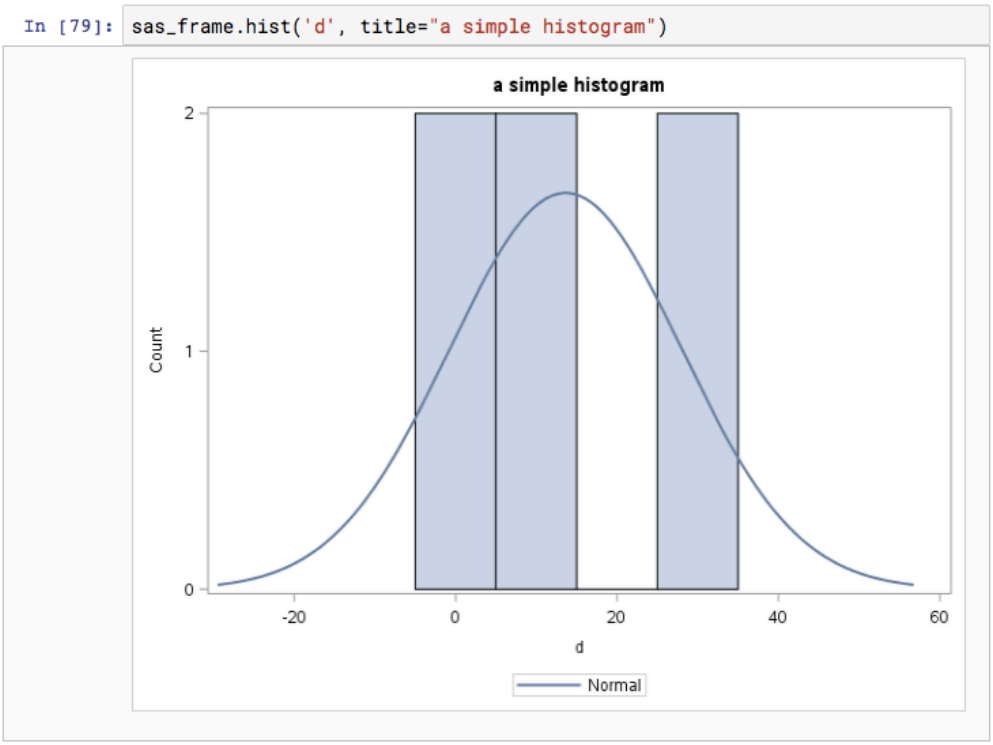


Figure 4: A Histogram within Jupyter

Both of the example methods invoked thus far (the **sort** method for invoking PROC SORT, and the **hist** method for invoking the SGPLOT procedure) are relatively trivial, yet SASPy includes support for a sizeable range of procedures, particularly in data modeling. Many of the more sophisticated methods for modeling or machine learning depend upon having a valid license for the appropriate product, and will warn the user if the corresponding product and procedures are unavailable in the connected SAS instance.

In the case of the more complex procedures available through its API, the ability to learn the specific SAS code that SASPy generates is enormously helpful both for debugging your work and for instructing Python users in deeper knowledge of SAS syntax. By invoking the **teach_me_SAS** method on your session, you can tell SASPy not to execute the code and instead to merely print it:

```
# switch into teaching mode

sas.teach_me_SAS(True)

# now display the code for our histogram

sas_frame.hist('d', title="a simple histogram")
```

The above usage outputs the SAS code that would have been generated and sent to your SAS session:

```
proc sgplot data=WORK._df;
   histogram d / scale=count;
   title "a simple histogram";
   density d;
run;
```

This capability exposes the underlying communication between Python and SAS and allows for users to more easily transition mentally between the simpler commands of the SASPy API and the more complete SAS statements used. However, it also underscores the reality that the integration between SASPy and SAS is entirely handled by offering a clean set of commands and methods in the former that can invoke code in the latter; a very different style of interaction between Python and SAS is offered by the following packages, which may play a crucial role for those who which to leverage the power of Viya in the future. These are included for your reference, and to illustrate that a number of incremental possibilities exist as you begin to move forward with Python and SAS.

ADDITIONAL PYTHON INTEGRATIONS: SWAT AND PIPERFITTER

The SAS Scripting Wrapper for Analytics Transfer (SWAT) is not the central focus of this introductory paper on Jupyter, yet its placement in the current set of options for Python integration is worthy of consideration here. SWAT breaks from the architectural model given above in which Python acts as a generator of SAS code orchestrating its throughput to an ordinary SAS session; in contrast, SWAT represents a first-tier method for executing workflows of analysis actions in SAS® Cloud Analytic Services (CAS) in SAS® Viya. The syntax, however, once again builds atop integration with Pandas DataFrames, which means that general techniques and abstractions learned from using SASPy to integrate with Python will translate well; more will be said on this connection in the final considerations below.

One final library connecting Python to SAS merits attention in that it integrates, at a higher workflow level, operations that might use either SASPy or SWAT. Pipefitter allows for the efficient implementation of pipelines of data transformations and analysis, passing a data set through a series of declared stages without needing to generate or manage many temporary data sets in between each task, eliminating much of the additional code and overhead in favor of a simple, understandable, and repeatable set of stacked steps. Once again, the existence of this library may not be immediately pertinent at the introductory level of using SASPy, yet its existence speaks to the long-term strategic advantages of adapting work to take advantages of these new paradigms.

ADVANTAGES AND FURTHER CONSIDERATIONS

Bridging SAS and Python has the immediately apparent advantage of bringing in a new set of tools in a second language and programming community that you can now more easily weave into your data manipulation or analyses. It also creates an excellent learning opportunity for those new to SAS and for easily publishing your work alongside the results in a single readable format online.

There are further advantages to incorporating these new tools that are perhaps less immediately evident, yet equally significant. Tapping into Python also means opening up your hiring positions and internal operations to a new pool of potential data science colleagues, who may already be accustomed to languages like Python or R yet face an initial barrier or resistance when asked to adapt their knowledge entirely to a new platform. Given the concentration of certain research interests like machine learning in the Python community, this can represent a significant strategic advantage. Furthermore, the open-source nature of SASPy (not only that its source code in Python is freely available, but that you can personally contribute to it, pending acceptance) can be a considerable motivating factor for certain professionals who may have an open-source background and enjoy helping to shape their own tools. The author of this paper has contributed significant functionality to SASPy that is now included in every release.

The other major value proposition at stake concerns the future payoff of adopting Python integrations now in light of the positioning of SWAT (and secondary tools like Pipefitter) at

the cutting edge of the SAS product ecosystem. SWAT leverages similar syntax to SASPy and integrates with DataFrames in a homologous manner, which means that workflows you build today—even on a small scale with a single instance of SAS and SASPy in a Jupyter notebook—may prove to be excellent starting points towards later deploying powerful cloud-based pipelines of analysis in Viya. The prospect of a single style of programming that retains its essential techniques from small, local experimentation to final production deployments on a powerfully distributed scale is well worth the time investment now.

CONCLUSION

SAS now offers an incremental set of integrations for Python, through which your work can progress in stages that will each offer immediate value. By adopting Jupyter notebooks merely as a new container for SAS development and work, you gain an excellent tool for sharing, learning, and publication. By integrating more tightly with Python libraries and syntax through leveraging SASPy directly, you can freely invoke methods and workflows bridging the advantages of the Python language and SAS products. Ultimately, the prospect that awaits is to further incorporate this work into higher order tools like Pipefitter, and to deploy cloud-based analytics by invoking SWAT under Viya. In this sense, the integration of Python and SAS ranges from a useful educational tool to a powerful part of your production suite of data processing and analysis.

RECOMMENDED READING

- Project Jupyter Home
  http://jupyter.org/
- SAS Software Github Page
  https://github.com/sassoftware
- SAS Kernel for Jupyter Documentation
  https://sassoftware.github.io/sas_kernel/
- SASPy Documentation
  https://sassoftware.github.io/saspy/
- SAS Scripting Wrapper for Analytics Transfer (SWAT) Documentation
  https://sassoftware.github.io/python-swat/
- SAS Pipefitter Documentation
  https://sassoftware.github.io/python-pipefitter/
- SAS Viya Documentation
  http://support.sas.com/documentation/onlinedoc/viya/

CONTACT INFORMATION

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

Jason Phillips, PhD
The University of Alabama
jphillips@ua.edu
https://github.com/jasonphillips