Paper 2822-2018

A Basic Introduction to SASPy and Jupyter Notebooks

Jason Phillips, PhD, The University of Alabama

ABSTRACT

With the recent introduction of the official SASPy package, it is now trivial to incorporate SAS® into new workflows 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 provides an overview of Jupyter Notebooks for the uninitiated, along with the initial steps and options for working with your SAS instance using SASPy. Included along the way are the general principles of passing data between Python's DataFrames and SAS data sets, as well as the unique advantages SAS brings to the Notebook workspace and Python ecosystem.

INTRODUCTION

In the past several years, a number of new possibilities have emerged for integrating SAS® into tools that are widely used in the Python corner of data science. Yet given the number of potentially overlapping components involved (Jupyter, SASPy, SWAT, Pipefitter), there is potential for confusion regarding the practical starting points and advantages of coordinating Python with SAS products and platforms. This paper aims to provide an introduction to those integrations that are likely to have the broadest immediate audience and benefit. These are primarily Jupyter notebooks and SASPy, which together offer an excellent starting point towards taking advantage of many benefits that Python integration can bring to SAS workflows and projects.

This paper begins with a brief overview of the new platforms (Jupyter Notebook) and packages or modules (SAS Kernel for Jupyter, SASPy, SWAT) that represent the primary entry points with which one needs to be familiar in order to begin taking advantage of these integrations. After the general overview, a few more focused benefits to the connection between Python and SAS will follow in the concluding remarks, as well as a consideration of further libraries that make up the current landscape of SAS and Python.

OVERVIEW OF FRAMEWORKS AND COMPONENTS

PYTHON

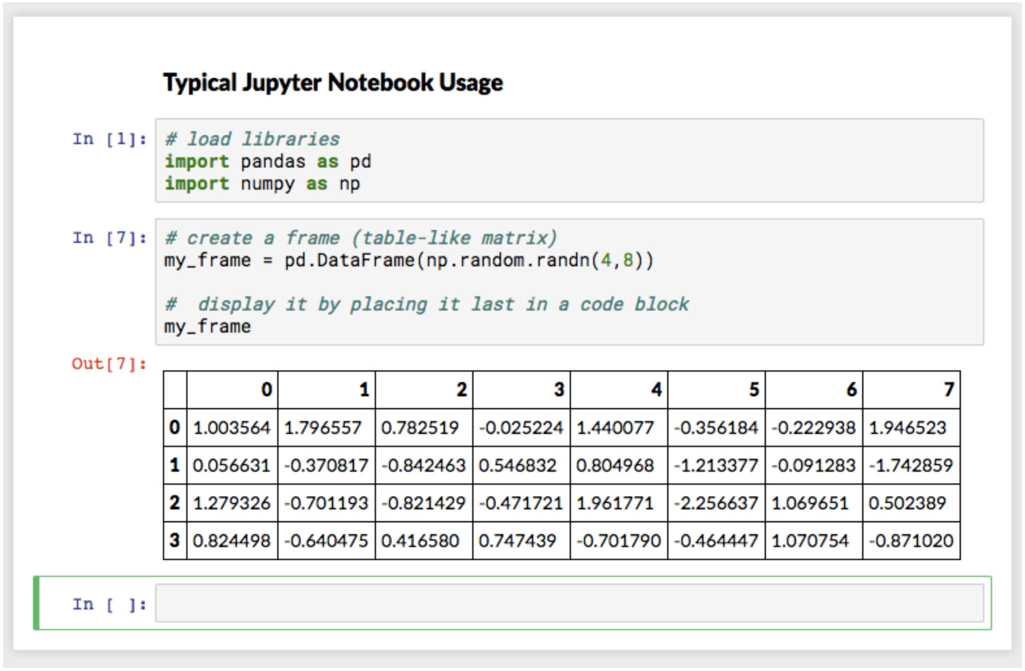
A lengthy introduction to the Python language would be superfluous, yet some sense of its present position within data science will prove useful in the content that follows. Of the many different languages that are widely employed in varying contexts and communities of scientific or statistical computing and research, Python’s distinct contributions may best be characterized in terms of its clarity of syntax and its well-developed pathways to efficient mathematical computation (the latter particularly by way of the ubiquitous package NumPy).

A notable area in which Python continues to be used heavily is machine learning and neural networks. However, it bears recognizing that many of the popular tools for this area of functionality are not strictly implemented in Python code alone, and often 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. To employ Python as a developer-focused interface to code that will be executed by SAS or on SAS cloud-based platforms is well in line with other major uses of the language 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. 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 Jupyter orchestrates by creating a background Python 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.)

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 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.

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.

While the Jupyter project grew out of Python, 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.

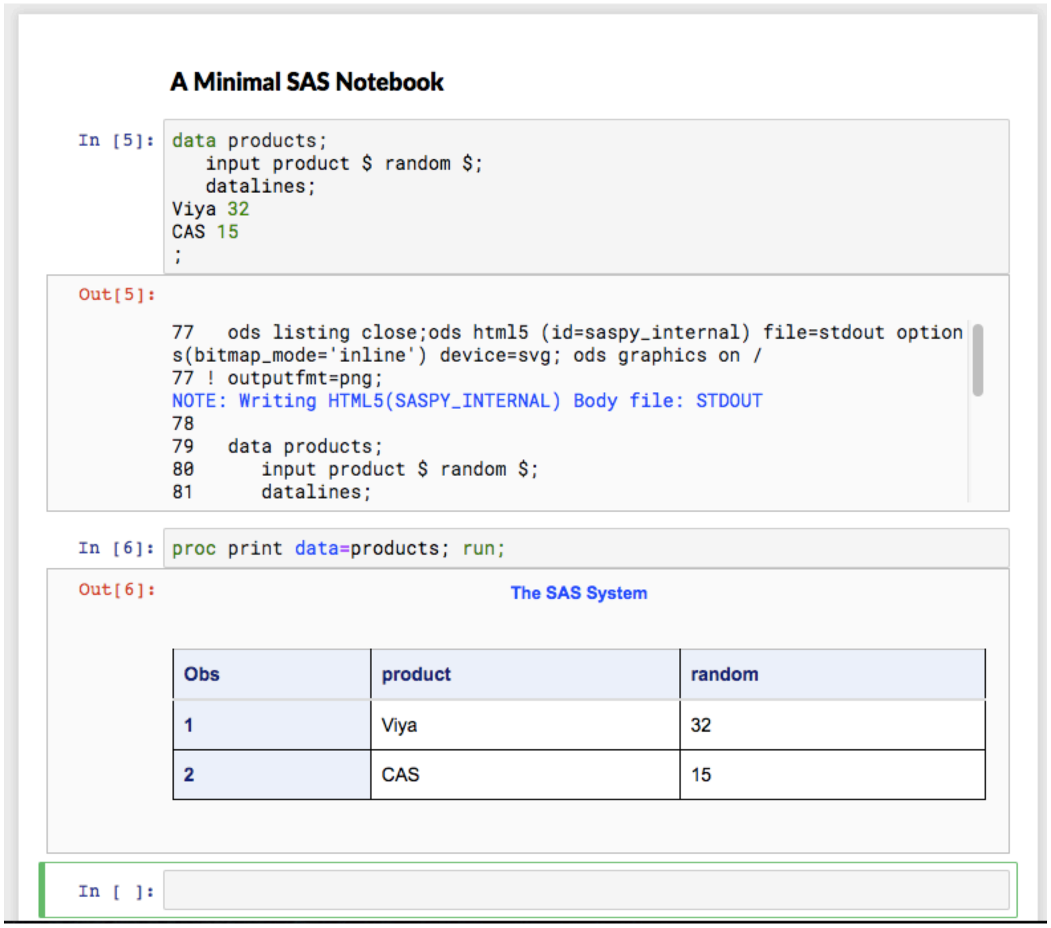
SAS KERNEL FOR JUPYTER

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.

Switching to the SAS kernel within 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 below).

Many 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.



A Minimal SAS Notebook

```
In [5]: data products;
           input product $ random $;
           datalines;
        Viya 32
        CAS 15
        ;
```

```
Out[5]:

        77   ods listing close;ods html5 (id=saspy_internal) file=stdout option
        s(bitmap_mode='inline') device=svg; ods graphics on /
        77 ! outputfmt=png;
        NOTE: Writing HTML5(SASPY_INTERNAL) Body file: STDOUT
        78
        79   data products;
        80      input product $ random $;
        81      datalines;
```

```
In [6]: proc print data=products; run;
```

Out[6]:

The SAS System

Obs	product	random
1	Viya	32
2	CAS	15

```
In [ ]:
```

Figure 2: A minimal Jupyter notebook using the SAS kernel

The ability to run SAS as a kernel within a Jupyter notebook opens up many of the key advantages of the notebook platform, particularly where sharing code and publishing to the web is desired for educational purposes. However, one achieves a 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 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 in Python that it can automatically convert to the appropriate SAS language statements for execution. It achieves this integration largely by creating references that act as interfaces between the two environments, using the popular Python package Pandas (which offers a useful data and matrix abstraction called a DataFrame) for coordinating datasets.

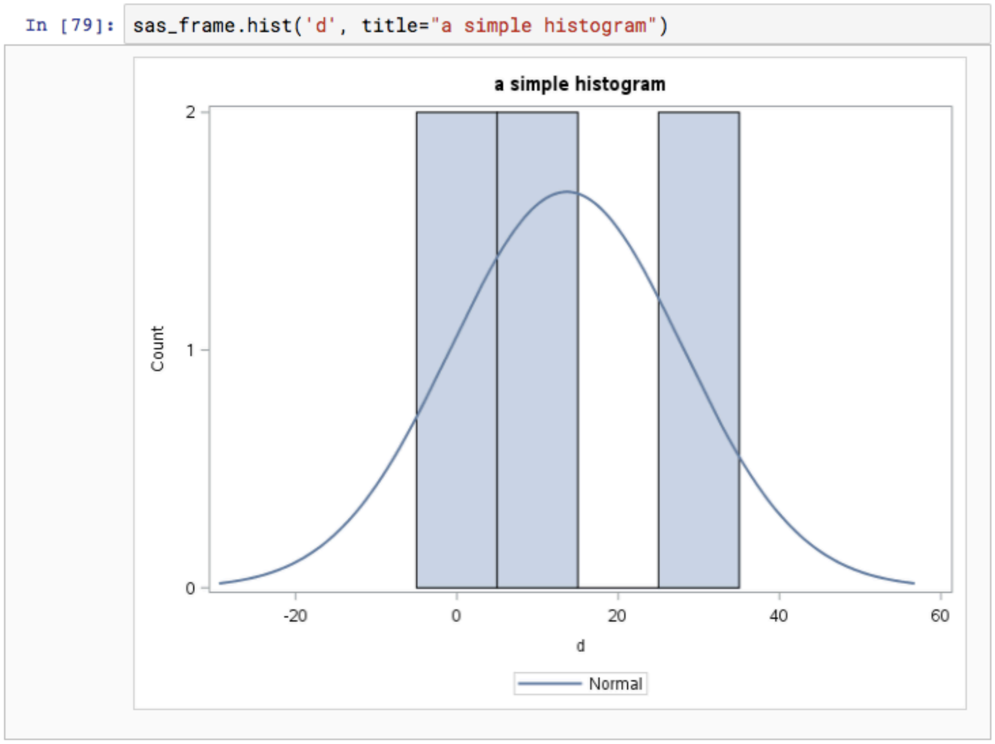
The quickest way to gain an understanding of the typical workflows enabled by SASPy is to illustrate, in a trivial example, the full round-trip that a dataset can take between the Python language (with data stored in a Pandas DataFrame) and SAS (manipulating a corresponding data set stored in an active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

```
# initial library imports

import pandas
import saspy

# connect to a SAS session
# 'my_server' here references a connection name specified
# in the local SASPy configuration; see installation 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 local Python methods.
# here, a simple sort is applied

my_sas_dataset.sort('group')

# and now we can pull the altered data set back into a Python DataFrame
#    (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. 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 3 illustrates by invoking a histogram.



```
In [79]: sas_frame.hist('d', title="a simple histogram")
```

Figure 3: 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 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.

SAS SCRIPTING WRAPPER FOR ANALYTICS TRANSFER (SWAT)

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.

SAS PIPEFITTER

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 your output 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 your positions and operations up to a new pool of potential data science colleagues who are already 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.

The other major value proposition at stake concerns the future payoff of adopting Python integrations early 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