Paper 3548-2019

Integrating Python and Base SAS®

Venu Gopal Lolla, Oklahoma State University

ABSTRACT

This paper presents a new framework for integrating Python into Base SAS® on the Microsoft Windows platform. Previous attempts at invoking Python functionality through Base SAS involved the use of a Java object in the DATA step. This paper aims to present a more comprehensive framework that includes support for the following: transferring data and parameters from SAS to Python; transferring data from Python to SAS; transferring figures generated in Python (using Matplotlib) back to SAS reports; and transferring contents from standard output and error streams from the Python process back to SAS reports. Although the components required to invoke Python from Base SAS have been available for a while, the lack of ease-of-use could prove to be a barrier to widespread use. The proposed framework aims to improve the usability of Python functionality through Base SAS by making the process more user-friendly. The framework is implemented through helper SAS macros, helper Python scripts, and helper binaries.

INTRODUCTION

Integration pathways between SAS and popular open source analytics software such as R and Python allow data scientists and data analysts to use the best features of both worlds. A solid integration pathway between SAS and R was introduced through SAS/IML® in 2009 (SAS Institute Inc., 2009). However, Python does not seem to enjoy similar first-class integration support yet.

A white paper published in 2015 (Hall, Myneni, and Zhang, 2015) provides a practical way to achieve this using the Base SAS Java Object. The approach presented is not even specific to (or restricted to) Python; it can be used to integrate with R or any other program. Perhaps due to the broad range of programs that can be targeted using the approach, it does not address certain specific usability issues that could deliver an improved integration experience to SAS users.

This paper attempts to: enumerate various features/options that are needed to provide a comprehensive SAS-Python integration experience; propose an architecture to support such features; and provide an implementation of the proposed architecture. An open-source implementation is made available under the permissive MIT License (for commercial use, modification, distribution, and private use) at: https://github.com/svglolla/sasnpy.

A point to note: while SAS-Python integration is referred to in this paper multiple times, it focuses on the case of trying to use/invoke Python capabilities from SAS and excludes the case of trying to use SAS from within Python. Ample development seems to be occurring and available through SAS-sponsored open-source GitHub repositories (SAS Institute Inc., 2015) in the context of using SAS functionality (and feature set) through Python.

DESIRABLE FEATURES

Several features are needed to provide a complete SAS-Python integration solution. An integration pathway that supports the following feature subset is likely to cover a significant number of use cases arising in the SAS-Python integration context:

- Ability to invoke Python scripts: the ability to invoke Python scripts from SAS is the first and perhaps the only strictly necessary requirement.
- Ability to transfer data from SAS to Python and Python to SAS: the ability to transfer data between SAS and Python semi-automatically is one of the first steps that could enhance the usability of the integration significantly. Data to be transferred could be: (a) tables representing datasets used by (or produced during) analyses; (b) scalar values that could be used to govern specific behaviors in the Python script; (c) binary payloads representing data that might not readily fit into a tabular form (for example, PMML models or network graph representations).
- Ability to capture output and error streams from Python: the ability to automatically capture output and error streams and inject them into SAS logs, outputs, or reports will make information that is potentially not present in data returned from Python available to the user without the user having to process/search log files.
- Ability to retrieve plots generated in Python: the ability to automatically capture plots and inject them into SAS reports is likely to make information (and insights) that is potentially not readily available in the data returned from Python available to the user.
- Ability to set/modify various parameters that modify the behavior of the integration component such as Python installation to use, turning on (or off) plot capture, whether or not to keep the Python program alive, etc. Whenever there is a design decision to be made and no clear winner emerges, it is probably desirable that the user be able to switch between various options elegantly without having to modify the code of the integration component.

It is also desirable that the integration component is light-weight and minimally intrusive in both the SAS and Python environments.

While the feature subset is enumerated in the context of SAS-Python integration, they are likely to apply to other similar integration use-cases and should probably be considered during the development of similar integration components.

ARCHITECTURE OVERVIEW

While it is possible to provide a platform-independent mechanism that works on various operating systems, the remainder of this paper restricts its scope to the Windows platform.

The general idea driving the design of the architecture for the SAS-Python integration solution proposed in this paper is to hide from the end-user, as much as possible, the technical details relating to the integration and let the user focus on using the integration instead. In other words, the integration solution should allow the end-user to focus on using the integration as opposed to having to exercise (or learn) the nitty-gritty technical details about the integration itself. The user should be allowed to focus on their task (for example, perform network analytics computations using the igraph library on data from SAS) instead of worrying about how to capture the output stream from Python.

Python provides an ability to specify the script to run in a file form as a command line argument; it is also possible to provide the script through an input stream. The ability to invoke Python (or any other program) through SAS will satisfy the first requirement discussed in the previous section. SAS provides multiple mechanisms to achieve this: the SYSTASK statement, the X statement, the CALL SYSTEM routine, and the %SYSEXEC macro. However, when the other requirements from the previous section are considered, the idea of a separate component managing interactions with the Python program begins to gain traction. This component will be in-charge of creating temporary folders, managing temporary files, managing the Python program (process) lifetime, capturing the output and error streams, etc.

The details of using the aforementioned component can be encapsulated into a SAS helper script file that the end-user can use. This script will surface macros to the end-user that can be used as entry points into the integration pipeline. This script file will also be responsible for insulating the end-user from the details relating to data transfer between SAS and Python and injecting output and error streams from the Python process into SAS reports.

A helper Python script (module) will be responsible for insulating the end-user from the details relating to capturing plots, converting data between SAS datasets to pandas data frames (or other Python-native representations), installing required Python packages if necessary etc.

The SAS-Python integration solution described in this paper is delivered as single a component (referred to as "SASnPy integration component"; .ZIP file) that can be downloaded and used. The component will have three major subcomponents: a helper SAS script file (.SAS file), a core integration binary component (.DLL file), and a helper Python script file (.PY file); see Figure 1. The component will also contain other files (text files serving as templates etc.) that might be used by any of the three major subcomponents.

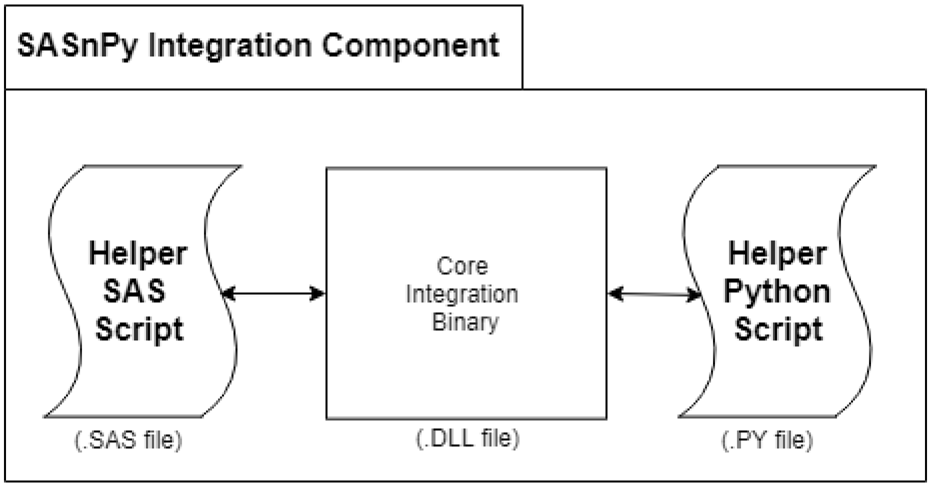

Figure 1: Sub-components of the SAS-Python integration component

The SAS helper file will be included into the user's SAS program (%INCLUDE) and will be executing in the SAS session along with the user's SAS script. The integration binary component will be loaded into the SAS session upon initialization and will instantiate a Python process. The helper Python script and the user's Python script will be running in the Python process instantiated by the integration binary. Figure 2 illustrates the activity sequence involved in executing a Python script from the user's SAS script using the SASnPy integration component. Separate sequences are involved in other activities such as transferring data between SAS and Python.

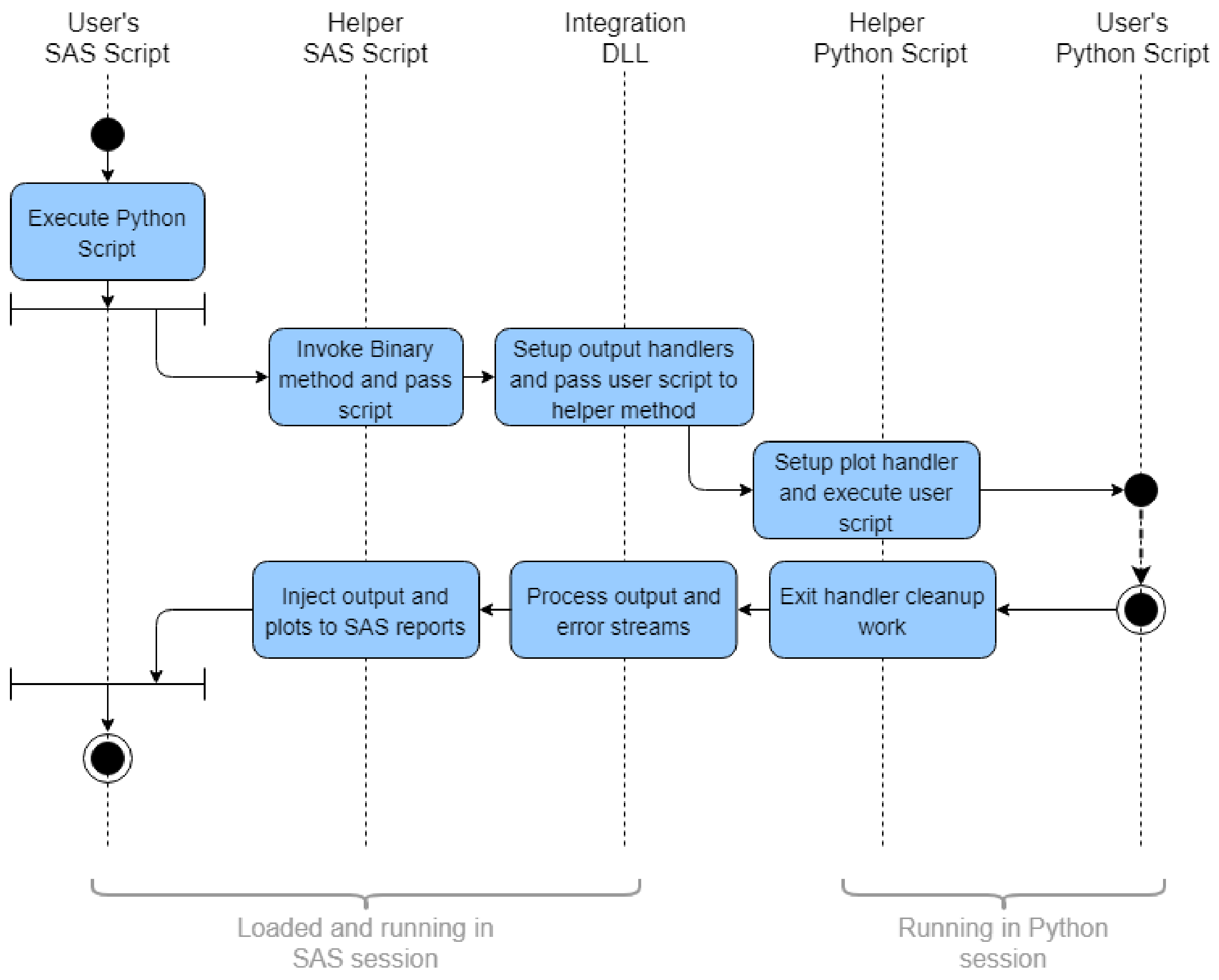
User's SAS Script
Helper SAS Script
Integration DLL
Helper Python Script
User's Python Script
Execute Python Script
Invoke Binary method and pass script
Setup output handlers and pass user script to helper method
Setup plot handler and execute user script
Inject output and plots to SAS reports
Process output and error streams
Exit handler cleanup work
Loaded and running in SAS session
Running in Python session

Figure 2: Example activity diagram for the execution of a Python script from SAS.

IMPLEMENTATION OVERVIEW

The implementation, source code, and related information are available under the MIT license at https://github.com/svglolla/sasnpy. This section will provide an overview of some of the key implementation details.

CORE INTEGRATION COMPONENT (SASNPY.DLL)

The core integration binary component (sasnpy.dll) is developed as a .NET class library in C#. The community edition of Visual Studio 2017 was used to develop this component. The project uses a set of compile-time libraries through the UnmanagedExports nuget (Giesecke, 2012) to expose the various methods in the component as C-style methods for use in SAS.

Some of the key methods exposed by this component are as follows:

- *PyStartSession*(): initialize a Python process and load helper modules.
- *PyEndSession*(): shutdown an open Python session, close the process, and conduct clean-up activities.
- *PySetPath*(): used to set the Python installation to be used to execute the user's Python script.
- *PyExecuteScript*(): used to execute the user's Python script. This method is also responsible for capturing (and processing) output and error streams from the Python process and making them available for use by the SAS helper script.
- *PySessionTempLocation*(): used to get the temporary folder location where intermediate files are created for the consumption of various sub-components.
- *PySetInputTable*(): used to induce information relating to a SAS dataset into the Python session.
- *PySetInputScalar*(): used to induce a scalar value into the Python session.
- *PyGetOutputTable*(): used to get information relating to a dataset in Python into the SAS session.
- *PyGetOutputScalar*(): used to get the value of a scalar object in the Python session into the SAS session.

The end-user does not have to be aware of these methods or their usage. These methods will be invoked almost exclusively through the SAS helper script file sub-component.

PYTHON HELPER SCRIPT (SASNPY.PY)

The .NET component instantiates a Python process and then loads the Python helper script file as a module. The Python helper script/module exports various methods to manage the execution of the user's Python script.

This script defines various helper methods that are used by the .NET component. These methods are used to set input tables, set input values, get output tables, get output values, and to run user scripts. Data tables sent from the SAS session are converted to pandas data frames if the pandas library is installed; if pandas is not installed, data tables are represented as a list of lists. If matplotlib is available, the script loads the package and then reroutes the pyplot.show() from matplotlib to automatically capture any plots created by the user script.

Some of the key methods exported by this script/module are as follows:

- *execute_script*(): used to execute user's Python code in the current session.
- *reroute_pyplot_show*(): used to reroute the pyplot.show() from matplotlib to *pyplot_show_handler*().
- *pyplot_show_handler*(): save a plot generated by user code to the disk instead of showing it on the screen.
- *entry_handler*(): used to conduct initialization tasks during module load.
- *exit_handler*(): used to conduct clean-up tasks during session end.
- *set_input_table*(): used to induce a data table from SAS into the global namespace.
- *set_input_scalar*(): used to induce a scalar value from SAS into the global namespace.

- *get_output_table*(): used to route a table-like object (for example, a pandas data frame) from Python back into the SAS session.
- *get_output_scalar*(): used to route a scalar value from the Python back into the SAS session.

The end-user does not have to be aware of these methods or their usage. These methods will be invoked almost exclusively through the .NET binary component.

SAS HELPER SCRIPT (SASNPY.SAS)

The SAS helper script defines various SAS macros that make using various features of the SAS-Python integration component easy to use. The macros encapsulate various details relating to loading the .NET binary component and insulate the user from the same. These macros are to be used directly in the user's SAS script and hence the user should familiarize themselves with the usage of these macros. The user will have to include (using %INCLUDE) this script file in their SAS script to use the SAS-Python integration component.

Some of the key macros exported by this script are as follows:

- *%PyInitialize*(): after including the helper script file in the user's SAS script, this is the first macro that is to be used. This macro uses the PROTO procedure to prototype the various methods exported by the .NET component for further use. This macro also uses FCMP procedure to define various functions used by other macros defined in the SAS helper script.
- *%PySetPath*(): used to specify the path of the Python installation to use for executing the user's Python script.
- *%PyStartSession*(): used to initiate a Python session through the integration component.
- *%PyEndSession*(): used to terminate a Python session currently in use through the integration component.
- *%PySetInputTable*(): used to induce a SAS dataset into the Python session.
- *%PySetInputScalar*(): use to induce a scalar value into the Python session.
- *%PyGetOutputTable*(): used to bring back a table-like object from the Python session back into SAS.
- *%PyGetOutputScalar*(): used to bring back a scalar value from the Python session back into SAS.
- *%PyExecuteScript*(): used to send Python code to the integration component for execution in the Python session.
- *%PySetOption*(): used to set/modify the values for various options that govern the behavior of the SAS-Python integration component.

USING THE IMPLEMENTATION

The following code snippets provide an example of how the user’s SAS and Python scripts could use the SAS-Python integration component. More examples can be found at https://github.com/svglolla/sasnpy.

To use the component, the SAS-Python integration component zip file matching the bit-ness (32-bit or 64-bit) of the SAS installation has to be downloaded and extracted. For the purpose of the following example, assume that the files have been extracted to "C:/sasnpy". Also assume that the Python installation to use is located at "C:/Python3.6/Python.exe" and the user's Python script is located at "C:/scripts/test.py".

Example SAS script file:

```
/* Include SASnPy helper script */
%include "C:/SASnPy/sasnpy.sas";

/* Initialize/define functions required for SAS-Python Integration */
%PyInitialize("C:/SASnPy");

/* Set Python installation to use */
%PySetPath("C:/Python3.6/Python.exe");

data _null_;

   /* Start Python session */
   %PyStartSession();

   /* Send data tables to Python */
   %PySetInputTable("air_ds", sashelp.air);
   %PySetInputTable("comet_ds", sashelp.comet);

   /* Send scalar data to Python */
   %PySetInputScalar("max_iter", 42);
   %PySetInputScalar("my_name", "sasnpy");

   /* Execute script */
   %PyExecute("C:/scripts/test.py", result);

   /* Get data tables from Python */
   %PyGetOutputTable("py_ds1", work.pyd1);
   %PyGetOutputTable("py_ds2", work.pyd2);

   /* Get scalar data from Python */
   %PyGetOutputScalar("py_res1", abc);
   %PyGetOutputScalar("py_res2", xyz);

   put "abc = " abc;
   put "xyz = " xyz;

   /* End Python session */
   %PyEndSession();

run;

proc print data=work.pyd1; run;
proc print data=work.pyd2; run;
```

Example Python script file:

```python
import pandas as pd

# Induced from SAS
# air_ds: pandas data frame
# comet_ds: pandas data frame
# max_iter: numeric scalar
# my_name: string scalar

air_ds.describe()
comet_ds.describe()
for i in range(max_iter):
    print(i)

print("My name is " + my_name)

data1 = {'Name': ['Alpha', 'Bravo', 'Charlie'],
        'Age': [22, 35, 29]}

data2 = {'Name': ['Alpha', 'Bravo', 'Charlie'],
        'Code': ['AG', 'BT', 'CQ']}

py_ds1 = pd.DataFrame.from_dict(data1)
py_ds2 = pd.DataFrame.from_dict(data2)
py_res1 = 12.34
py_res2 = "This works"

# Available for retrieval from SAS
# py_ds1, py_ds2: data frames
# py_res1, py_res2: scalars
```

CONCLUSIONS & FUTURE WORK

A new framework for integrating Python into Base SAS® on the Microsoft Windows platform has been presented. An enumeration of desirable features for SAS-Python integration and an architecture to achieve those features have been presented. An open-source implementation of the architecture (under the MIT license) can be accessed at https://github.com/svglolla/sasnpy. The implementation allows end users to focus on using the SAS-Python integration instead of having to focus on the technical details of the integration itself. Technical details are abstracted away from the user and encapsulated into sub-components in the implementation that can be used via convenient macros.

While the new framework presents a light-weight and easy-to-use SAS-Python integration pathway, there is still plenty of scope for improvement. Following are some points to be considered for future work:

- The current implementation is limited to the Microsoft Windows platform; the proposed architecture could be implemented in a platform-independent manner to remove the limitation.
- The current implementation does not provide support for the transfer of binary objects between SAS and Python; adding this feature will allow users to exchange non-tabular data objects (such as model representations) easily.

- The current implementation does not support the use of non-local Python engines; adding this feature will allow SAS-Python integration to take advantage of remote Pythons installations (possibly as services) that offer better performance.
- The current implementation does not support asynchronous script execution or multiple concurrent Python sessions; adding this feature will allow SAS-Python integration to allow the user to execute large Python jobs without blocking the execution of the user's SAS scripts when warranted.
- The current implementation does not provide any fault-tolerance; if a Python process shuts down unexpectedly or is stopped, the job/script submitted to the Python process is lost. Adding this feature to the implementation will improve the reliability of the integration.
- Further abstraction (and parametrization) of the implementation is possible in a manner that will allow the use of the implementation for engines other than Python.

REFERENCES

SAS Institute Inc. 2009. "Enhancements in SAS/IML® 9.22 Software." Accessed March 23, 2019. https://support.sas.com/rnd/app/iml/IML922.html

Hall, Patrick, Myneni, Radhika and Zhang, Ruiwen. 2015. "Open Source Integration using the Base SAS Java Object." Accessed March 23, 2019. https://github.com/sassoftware/enlighten-integration/blob/master/SAS_Base_OpenSrcIntegration/SAS_Base_OpenSrcIntegration.pdf

SAS Institute Inc. 2015. "SAS Software: Open Source from SAS Software." Accessed March 23, 2019. https://github.com/sassoftware

Giesecke, Robert. 2012. "UnmanagedExports 1.2.7: Unmanaged Exports (DllExport for .Net)." Accessed March 23, 2019. https://www.nuget.org/packages/UnmanagedExports

CONTACT INFORMATION

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

Venu Gopal Lolla
Oklahoma State University
venu.lolla@okstate.edu