PharmaSUG 2026 - Paper ET-392

Running Python from SAS: A Practical Comparison of Available Approaches

David Ward, Triam Ltd

INTRODUCTION

Python has become one of the most widely used languages in modern data science and analytics. Its large ecosystem of libraries provides capabilities ranging from machine learning and natural language processing to advanced data visualization and web services. Because of this expanding ecosystem, many organizations that rely heavily on SAS are increasingly interested in incorporating Python into their analytical workflows.

For SAS programmers, however, the question is rarely whether Python is useful. The more practical question is **how Python can be incorporated into existing SAS programs without disrupting established workflows**. In many production environments—particularly in regulated industries such as pharmaceuticals—SAS programs represent years of development, validation, and operational investment. Rewriting these systems entirely in another language is often impractical.

Instead, many teams pursue a hybrid approach in which Python code is executed from within SAS programs when its functionality is needed. This allows programmers to continue leveraging the strengths of the SAS language while accessing Python libraries or external services that may not be easily available within SAS alone.

A common example arises in organizations that maintain mature SAS-based data pipelines but wish to leverage modern machine learning libraries available in Python. For instance, a clinical data processing workflow may prepare and validate datasets using SAS programs that have been maintained for many years. Rather than rewriting the entire workflow, the SAS program can invoke Python to train or apply a machine learning model using libraries such as scikit-learn or XGBoost. The results can then be returned to SAS for reporting, validation, and regulatory submission. In this type of hybrid architecture, SAS continues to provide the reliable data processing framework while Python contributes specialized analytical capabilities.

The ability to integrate Python into established SAS workflows without rewriting existing systems has therefore become an increasingly important capability. Fortunately, the SAS language provides several mechanisms that make this type of integration possible. Depending on the SAS platform and environment, Python code can be executed in several ways, including:

- Embedded calls to Python functions from within SAS programs
- Direct integration through SAS procedures designed to invoke Python
- External execution of Python scripts through operating system commands
- Network-based service calls using REST APIs

Each of these approaches enables Python integration but differs significantly in architecture, performance characteristics, portability, and operational complexity. Some methods execute Python within the SAS runtime itself, while others rely on external processes or services. As a result, choosing the correct approach requires understanding not only how each technique works but also when it is appropriate.

This paper will provide a practical comparison of the most common approaches for executing Python code from SAS. The techniques discussed include PROC FCMP, PROC PYTHON (in both SAS/Viya and Siemens SLC environments), system calls to external Python processes, and REST-based service integration. In addition, several Python packages that facilitate SAS interoperability will be briefly introduced.

The goal of this paper is not to advocate for a single solution, but rather to provide SAS programmers with a clear understanding of the available options and the trade-offs associated with each. By understanding how these approaches differ, programmers can make informed decisions about how to integrate Python into their own SAS applications.

KEY TAKEAWAYS

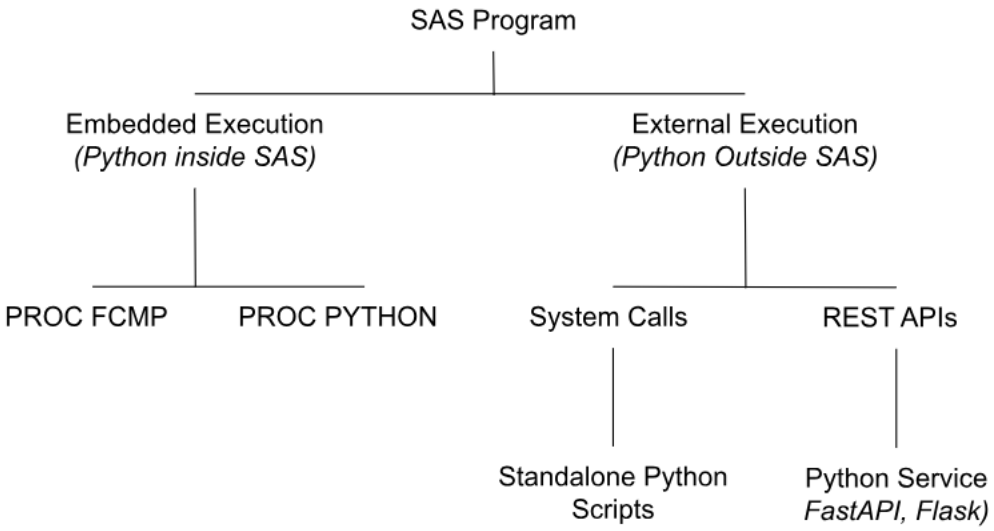
This paper provides a practical overview of several techniques that allow SAS programs to execute Python code. After reading this paper, you should be able to:

- Understand the **architectural differences** between embedded Python execution and external Python execution.
- Identify when to use **PROC FCMP** versus **PROC PYTHON** for integrating Python logic within SAS programs.
- Execute Python scripts from SAS using **system calls** when a fully independent Python environment is required.
- Invoke Python-based services using **REST APIs** and tools such as PROC HTTP.
- Select an appropriate integration strategy based on **platform availability, environment constraints, and workflow requirements**.

OVERVIEW OF PYTHON EXECUTION MODELS IN SAS

The techniques discussed in this paper fall into two architectural categories. Figure 1 summarizes these approaches.

Figure 1. Python Execution from SAS



IN-PROCESS EXECUTION

In the in-process model, Python code is executed directly within the same runtime environment as SAS. In other words, the SAS session itself hosts the Python interpreter. When a SAS program invokes Python code using this approach, the Python instructions run as part of the same process that is executing the SAS program.

This architecture provides several advantages. Because both languages operate within the same process, data can often be exchanged directly between SAS and Python objects. For example, a SAS data set might be converted into a pandas DataFrame, manipulated in Python, and then returned to SAS without the need to write intermediate files to disk.

This tight integration can make Python execution appear almost seamless from the perspective of the SAS programmer. Python functions can behave much like native SAS procedures or functions, and results can be returned immediately to the calling SAS program.

However, this approach also introduces constraints. Because Python is embedded within the SAS runtime environment, the available Python installation and libraries are often controlled by the SAS platform itself. In enterprise environments, this may limit the ability of individual programmers to install or manage additional Python packages.

EXTERNAL EXECUTION

In the external execution model, Python runs in a separate process from SAS. Rather than executing Python code directly inside the SAS runtime, the SAS program launches an external Python interpreter using operating system commands or service calls.

This approach provides greater flexibility in managing the Python environment. Developers may use independent Python installations, virtual environments, or containerized environments without affecting the SAS installation.

The trade-off is that communication between SAS and Python becomes more explicit. Data must typically be transferred through files, pipes, or network protocols. Error handling and logging must also be managed across process boundaries.

Despite these additional considerations, external execution can be a powerful approach, particularly when Python is used to access external services, perform specialized computations, or interact with systems that are not directly accessible from SAS.

TRANSPARENT VS. OPAQUE INTEGRATION

These approaches can also be viewed in terms of transparency. Embedded techniques behave like native SAS functionality, while external techniques invoke Python processes or services outside the SAS runtime.

PROC FCMP (SAS 9.4M6+)

One of the earliest mechanisms provided by SAS for integrating Python functionality into SAS programs is through the use of **PROC FCMP**. PROC FCMP is primarily known as a facility for creating user-defined functions and subroutines that can be called from DATA steps, procedures, or other SAS programs. Beginning with SAS 9.4M6, PROC FCMP was extended to allow Python functions to be invoked from within these user-defined routines.

This approach provides a relatively natural integration point between SAS and Python because it allows Python code to be wrapped inside a function that behaves like a native SAS function. Once defined, the function can be called repeatedly from SAS code without requiring the programmer to manage the Python execution environment directly each time.

HOW PYTHON IS INVOKED

When Python is used through PROC FCMP, the SAS runtime loads a Python interpreter and executes the specified Python code within that context. The programmer defines a function in PROC FCMP and specifies that the implementation should be handled by Python. Inputs to the SAS function are passed to the Python code, and the result is returned to SAS.

Example 1. Calling Python using PROC FCMP

```
proc fcmp outlib=work.funcs.python;
  function py_square(x);
     declare object py(python);
     rc = py.rtinitt();                /* initialize Python */
     rc = py.submit("def square(x):
                        return x*x");
     rc = py.call("square", x, result);
     return(result);
  endsub;
run;

options cmplib=work.funcs;

data example;
  x = 5;
  y = py_square(x);
run;
```

In this example, a Python function named square is defined and executed through PROC FCMP. The SAS function py_square acts as a wrapper around the Python implementation, allowing it to be called from a DATA step like any other SAS function.

Although this example is intentionally simple, the same technique can be used to access more complex Python functionality, including calls to external libraries.

STRENGTHS OF THE PROC FCMP APPROACH

Using PROC FCMP to integrate Python into SAS programs has several advantages.

First, it provides **tight integration with SAS code**. Because the Python logic is wrapped inside a SAS function, it can be invoked directly from DATA steps or procedures without requiring external scripts or separate processes.

Second, this approach supports **clean encapsulation of functionality**. Python code can be packaged inside reusable SAS functions, which can simplify the organization of hybrid SAS/Python programs.

Third, because the Python interpreter is invoked from within the SAS runtime, **data exchange can be handled efficiently**, particularly for scalar values and small objects.

Finally, this technique works within the familiar SAS programming model. Programmers who are comfortable writing FCMP functions can often adopt this approach with minimal conceptual overhead.

LIMITATIONS

Despite these advantages, PROC FCMP is not a universal solution for Python integration.

One limitation is that the interface between SAS and Python is relatively narrow. Passing simple values such as numbers and character strings works well, but working with large data sets or complex data structures can be more cumbersome. In many cases, intermediate files or explicit data conversion steps may still be required.

Another consideration is **environment management**. Because Python is invoked from the SAS runtime environment, the available Python installation and packages may be determined by system configuration rather than by individual developers. This can make it more difficult to use specialized Python libraries unless the environment is carefully managed.

Finally, while PROC FCMP can invoke Python functions effectively, it is generally **not designed for large-scale Python workflows** or extensive data manipulation within Python itself. In those cases, other techniques—particularly those designed specifically for Python execution—may be more appropriate.

APPROPRIATE USE CASES

PROC FCMP is most effective when Python functionality is needed in relatively small, well-defined units of work. Typical use cases include:

- Calling specialized Python libraries for calculations
- Performing transformations easier to express in Python
- Wrapping Python algorithms as reusable SAS functions
- Integrating small Python utilities into larger SAS workflows

In these scenarios, PROC FCMP can provide a convenient bridge between the two languages while preserving the structure and execution model of the SAS program.

In the next section, we will examine **PROC PYTHON**, which represents a more direct and modern mechanism for integrating Python execution into SAS environments. Unlike the PROC FCMP approach, PROC PYTHON is designed specifically for running Python code within SAS sessions and provides deeper integration with Python data structures and libraries.

PROC PYTHON (SAS/VIYA AND SIEMENS SLC)

While PROC FCMP provides a mechanism for invoking Python functions from SAS, more recent developments in the SAS ecosystem have introduced a more direct approach to integrating Python into SAS programs. **PROC PYTHON** was designed specifically to allow Python code to be executed within a SAS session, providing a more natural environment for Python development while still operating within the context of a SAS program.

PROC PYTHON is available in modern SAS environments such as SAS/Viya and in compatible implementations such as Siemens SAS Language Compiler. Unlike PROC FCMP, which embeds Python functionality within SAS-defined functions, PROC PYTHON allows programmers to write Python code directly within a SAS program block.

This approach significantly lowers the barrier to incorporating Python logic into SAS workflows, particularly for programmers who already have experience working in Python.

EXECUTION MODEL

When PROC PYTHON is invoked, the SAS session initializes a Python interpreter and executes the code contained within the procedure block. The code written inside PROC PYTHON is interpreted entirely as Python rather than SAS.

Example 2. Executing Python code with PROC PYTHON

```
proc python;
submit;
import math

x = 5
y = math.sqrt(x)

print("Square root:", y)
endsubmit;
run;
```

In this example, the statements between submit and endsubmit are interpreted as Python code. The Python interpreter executes the script and returns output to the SAS log.

From the programmer's perspective, this approach behaves much like embedding a small Python script inside a SAS program. The SAS session manages the lifecycle of the Python interpreter, while the Python code runs using the standard Python syntax and libraries.

DATA EXCHANGE BETWEEN SAS AND PYTHON

One of the most important features of PROC PYTHON is its ability to exchange data between SAS and Python environments. In many workflows, SAS remains responsible for data ingestion, transformation, and governance, while Python is used for specialized computation or advanced analytics.

Depending on the environment, data can be transferred between SAS and Python objects through built-in interfaces that convert SAS data sets into Python structures such as pandas DataFrames. This allows Python libraries to operate on SAS data and return results back to the SAS program.

A simplified conceptual workflow might look like the following:

1. SAS reads or prepares a data set.
2. The data set is passed into Python.
3. Python performs calculations using libraries such as pandas or NumPy.
4. Results are returned to SAS for further processing.

Although the details vary depending on platform configuration, this integration allows SAS programmers to access modern Python libraries without leaving the SAS programming environment.

STRENGTHS OF PROC PYTHON

PROC PYTHON offers several advantages compared with FCMP.

First, it provides a **native Python programming environment** within SAS. Instead of wrapping Python code inside SAS functions, programmers can write Python code directly using standard Python syntax.

Second, PROC PYTHON enables **clear separation between SAS and Python logic**. SAS code can handle data preparation and orchestration, while Python code can focus on specialized computations.

Finally, because the syntax is less cumbersome, programmers who may not be as familiar with Python or proc FCMP should be able to more rapidly implement their solutions.

LIMITATIONS

Despite its advantages, PROC PYTHON also introduces several considerations.

One practical concern is **platform availability**. PROC PYTHON is not available in SAS 9.4 and is primarily supported in SAS Viya environments, though similar functionality is available in Siemens SLC. Siemens' procedure differs slightly but accomplishes essentially the same functionality. Organizations using SAS 9.4 will need to add either SAS/Viya or Siemens SLC in order to use this technique.

Another consideration involves **environment management**. As with FCMP, the Python environment is typically managed by system administrators and may be difficult for individual users to modify.

In addition, while PROC PYTHON is well suited for executing Python scripts, it may not always be the best solution for integrating with external Python applications or distributed systems. In those cases, approaches that launch external Python processes or interact with network services may provide greater flexibility.

BEST-FIT SCENARIOS

PROC PYTHON is particularly useful in situations where SAS programmers want to integrate Python analytics directly into existing SAS workflows. Examples include:

- Applying machine learning models using Python libraries
- Performing specialized statistical or numerical computations
- Using Python visualization tools
- Integrating Python data-processing utilities into SAS pipelines

In these situations, PROC PYTHON provides a clean and modern bridge between SAS and Python.

The next section will examine techniques that take a different architectural approach—methods that execute Python **outside** the SAS runtime environment using operating system commands or service interfaces. These techniques sacrifice some integration convenience but often provide greater flexibility and control over the Python environment.

OPAQUE TECHNIQUES: SYSTEM CALLS AND REST APIS

Another category of integration techniques takes a fundamentally different approach. Rather than embedding Python inside the SAS runtime, these methods execute Python **externally**, using the SAS program primarily as an orchestrator. SAS launches Python processes or communicates with Python-based services, allowing the two environments to interact without sharing the same execution context.

These techniques can be described as **opaque integration methods**. From the perspective of the SAS program, Python execution occurs outside the SAS runtime, and the internal behavior of the Python code is not directly visible to SAS. While this model requires more explicit coordination between the two environments, it offers important advantages in flexibility and environment control.

Two of the most common opaque techniques are **operating system system calls** and **REST-based service interfaces**.

SYSTEM CALLS

One of the simplest ways to execute Python from SAS is to launch a Python script using an operating system command. The SAS language provides several mechanisms for executing external commands, including the X statement, the SYSTEM() function, and the more structured SYSTASK statement.

In this model, SAS simply invokes the Python interpreter as an external program and passes the name of a Python script to execute. The Python code runs independently of the SAS session, and any communication between the two environments typically occurs through files or command-line parameters.

Example 3. Calling a Python script from SAS using pipes

```
filename cmd pipe "python myscript.py input.csv output.csv";

data _null_;
  infile cmd;
  input;
  put _infile_;
run;
```

In this example, SAS launches a Python script that reads an input file and produces an output file. Once the Python process completes, SAS can read the results and continue processing.

More advanced implementations often use command-line parameters to control the behavior of the Python script. For example, a SAS program might generate input files, call Python to perform specialized processing, and then read the resulting output files back into SAS data sets.

One advantage of this approach is **complete independence of the Python environment**. The Python interpreter, its libraries, and its configuration can be managed entirely outside the SAS system. This allows developers to use virtual environments, containers, or specialized Python installations without affecting the SAS environment.

However, this flexibility comes with several trade-offs. Data exchange must be handled explicitly through files or other mechanisms, and error handling between SAS and Python can require additional coding. In addition, performance may be affected if large data sets must be written to disk and read again after Python processing.

Despite these limitations, system calls remain one of the most flexible and widely available techniques for integrating Python with SAS programs.

REST APIS

Another increasingly common approach is to interact with Python code through **REST-based web services**. In this model, Python code runs as a server-side application—often implemented using frameworks such as FastAPI or Flask—and SAS communicates with that service through HTTP requests.

Rather than executing a local Python script, the SAS program sends data to a network endpoint, receives a response, and continues processing based on the returned results.

A simplified workflow might proceed as follows:

1. A Python application exposes a REST endpoint.
2. The SAS program sends an HTTP request containing input data.
3. The Python service processes the request.
4. The service returns a response to SAS, typically in JSON format.

Consider the following Python API

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/square/{x}")
def square(x: int):
    return {"input": x, "result": x * x}
```

SAS can interact with REST services using tools such as PROC HTTP, allowing it to send requests and capture responses. Here is SAS code that calls this API, passing in a number and receiving a JSON response which can then be read as a data set using the json library engine.

```
filename resp temp;
proc http
    url="http://127.0.0.1:8000/square/5"
    method="GET"
    out=resp;
run;

libname r json fileref=resp;
proc print data=r.root;
run;
```

JSON Response:

```
{
  "input": 5,
  "result": 25
}
```

This architecture provides several advantages. First, it allows Python code to operate as a **standalone service**, potentially serving requests from many clients simultaneously. Second, it supports modern software architectures in which analytics are deployed as reusable microservices.

Because the Python service runs independently of SAS, it can be deployed on different machines, scaled independently, or integrated with other applications.

The primary disadvantage of the REST approach is that it introduces additional architectural complexity. Deploying and maintaining a web service requires infrastructure, monitoring, and security considerations. In addition, network communication introduces latency that may not be acceptable for very high-frequency calls.

ADVANTAGES AND TRADE-OFFS

Opaque integration techniques provide several important advantages:

- Complete independence of the Python environment
- Flexibility to use modern Python frameworks and deployment strategies
- Compatibility with distributed systems and cloud architectures

However, these techniques also introduce additional complexity:

- Data must be exchanged through files or network protocols
- Error handling must be managed across process boundaries
- Execution may be slower if large data transfers are required

For many organizations, these methods are particularly attractive when Python functionality already exists as a standalone application or service, or when Python development is managed by a separate engineering team.

In the next section, we will briefly examine several **Python libraries that facilitate interoperability with SAS**, including packages designed to access SAS data sets and expose Python functionality in ways that integrate more naturally with SAS workflows.

IMPORTANT PYTHON PACKAGES

The focus of this paper is on techniques that allow **SAS programs to execute Python code**. However, when building hybrid analytical systems it is useful to understand, at least at a high level, how **Python programs interact with SAS environments**. In many organizations, integration occurs in both directions: SAS workflows may invoke Python for specialized processing, while Python-based analytics pipelines may also access SAS data or submit SAS jobs.

One widely used tool for this purpose is **SASPy** from the SAS Institute. SASPy provides a Python interface to a running SAS session, allowing Python programs to submit SAS code, retrieve results, and exchange data between SAS data sets and Python structures such as pandas DataFrames. In this architecture, Python acts as a client while SAS remains the execution engine. Communication between the two environments can occur through several connection mechanisms, including local STDIO connections, IOM connections to remote SAS servers, or HTTP-based interfaces in newer platforms.

In situations where a SAS session is not available, several Python libraries can read and write SAS data sets directly. Libraries such as **pyreadstat**, **sas7bdat**, and the pandas.read_sas() interface allow .sas7bdat files to be loaded into Python objects such as pandas DataFrames.

COMPARATIVE SUMMARY

The techniques presented in this paper offer several ways to execute Python code from within SAS programs. Each method has distinct architectural characteristics, operational requirements, and appropriate use cases.

As shown in Figure 1, these approaches fall into two categories: embedded and external execution. Embedded techniques, such as **PROC FCMP** and **PROC PYTHON**, run Python code within the SAS runtime environment and therefore provide tighter integration with SAS programs. External techniques, such as **system calls** and **REST-based services**, execute Python independently and interact with SAS through files or network communication.

PROC FCMP provides one of the earliest mechanisms for integrating Python into SAS programs. By wrapping Python logic inside user-defined functions, it allows Python functionality to be invoked from DATA steps or procedures in a way that resembles native SAS functions. This approach works well for relatively small units of computation but can become cumbersome when working with larger Python workflows or complex data structures.

PROC PYTHON represents a more modern integration mechanism available in newer SAS environments and Siemens SLC. It allows programmers to write Python code directly inside SAS programs and provides more natural access to Python libraries. When available, PROC PYTHON typically offers the most seamless experience for integrating Python logic into SAS workflows.

External techniques provide greater flexibility but require more explicit coordination between SAS and Python environments. System calls allow SAS programs to launch Python scripts as independent processes, while REST APIs expose Python functionality as network services that SAS programs can invoke using HTTP requests. These approaches are particularly useful when Python code already exists as standalone scripts or shared services.

In practice, no single method is universally superior. The best choice depends on factors such as the SAS platform being used, the complexity of the Python functionality required, and organizational constraints around software deployment and governance.

Table 1 provides a high-level comparison of the techniques discussed in this paper.

Method	Execution Model	Integration Level	Best Use Cases
PROC FCMP	Embedded	Moderate	Small Python functions within SAS programs
PROC PYTHON	Embedded	High	Integrated analytics and Python library usage
System Calls	External	Low	Running standalone Python scripts
REST APIs	External	Low–Moderate	Service-oriented architectures and shared analytics

‘

By understanding these trade-offs, SAS programmers can make informed decisions about how to incorporate Python capabilities into their existing workflows while maintaining the reliability and governance expected in production SAS environments.

CONCLUSION AND KEY TAKEAWAYS

As Python continues to grow in importance for analytics and data science, many SAS programmers are looking for practical ways to incorporate Python capabilities into existing SAS workflows. Fortunately, several techniques are available for executing Python code from within SAS programs, each with different strengths and trade-offs.

Embedded approaches such as **PROC FCMP** and **PROC PYTHON** provide the most direct integration with SAS code, while external techniques such as **system calls** and **REST APIs** offer greater flexibility and independence from the SAS runtime environment.

The most appropriate method depends on the SAS platform, the complexity of the Python functionality required, and operational constraints within the organization

If you need…	Recommended Approach
Tight SAS integration	PROC PYTHON
Small reusable functions	PROC FCMP
Full control of Python environment	System Calls
Shared analytics services	REST APIs

By understanding these options, SAS programmers can incorporate Python capabilities into their workflows while maintaining reliable and manageable analytical systems.

CONTACT INFORMATION

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

David Ward
Triam Ltd
david.ward@triamltd.com
https://triamltd.com/