Paper AD04

SASPY in a Flask

Jame Pilkington, AstraZeneca, Barcelona, Spain

ABSTRACT

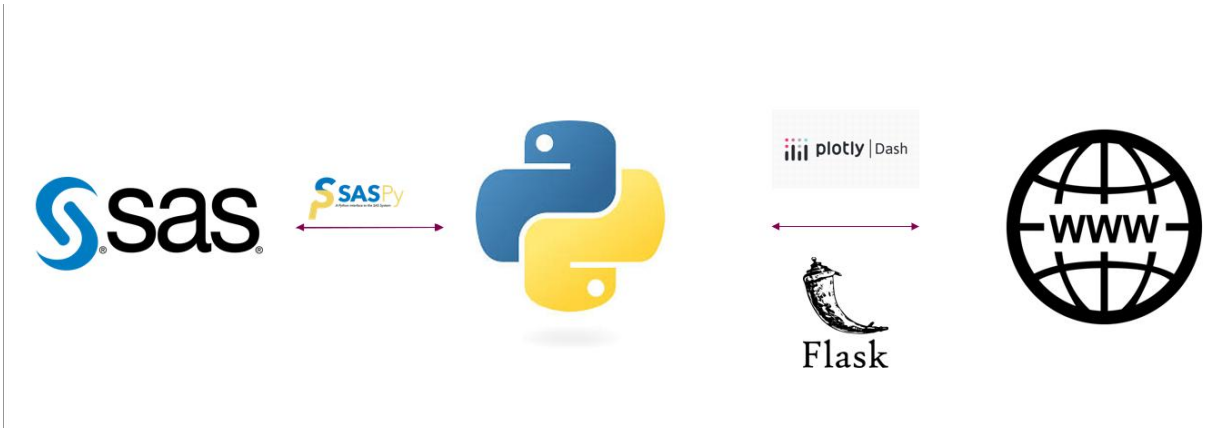
Python is equipped with a suite of robust tools that facilitate the creation and deployment of web applications. Building upon prior research that has already demonstrated the interoperability between Python and SAS using SASPy, this presentation aims to demonstrate the development of a Python-based web application, using a package called Dash, capable of hosting dynamic SAS outputs. These outputs can be interactively updated through the user interface of the web application. The paper will be divided into three sections: an introduction to SASPy, an overview of the fundamental concepts of a Dash web application, and a live demonstration. This paper will feature a graphical output that exemplifies the potential modifications, alongside the corresponding SAS code generated on the backend.

INTRODUCTION

The pharmaceutical industry is experiencing rapid and sustained growth in data volume and complexity, driven by innovations in clinical trial design and the increasing adoption of AI-enabled analytical methodologies. While established tools remain reliable, they can fall short in providing the interactivity and flexibility required for modern exploratory and operational needs. This paper introduces an approach that integrates the SAS and Python ecosystems to enable dynamic, interactive dashboards while preserving regulatory compliance and alignment with established workflows.

Given SAS's central role in regulatory programming, the extensive expertise within the field, and the substantial body of existing, validated code, it is essential to consider how SAS can continue to be leveraged as analytical techniques evolve. By combining this legacy of rigor and reliability with contemporary, open-source capabilities, we outline a pathway that retains the strengths of SAS-based workflows and extends them with Python-driven interactivity, thereby enhancing analytical agility without compromising compliance or reproducibility.

THE SASPY WEBAPP FRAMEWORK



The Goal

The objective is to develop a SAS program whose outputs can be rendered within a Python-based web application. This implementation will rely on Base SAS, SASPy, Python, and a lightweight web framework (e.g., Flask or Dash) to provide a reproducible and maintainable integration layer between analytical computation and presentation.

The web interface is intended to introduce controlled interactivity to the SAS workflow, enabling users to specify analytical parameters at runtime and thereby tailor execution to their needs without modifying core code. Parameter choices will be constrained to validated, enumerated options to preserve methodological integrity, prevent syntactic errors, and ensure consistent behavior across sessions and environments.

Within this architecture, SAS serves as the computational backend, executing data management tasks and validated statistical procedures, while Python functions as the presentation and orchestration layer. SASPy provides the bridge between these components, allowing Python to submit SAS programs, capture logs, and retrieve outputs in a structured manner suitable for web delivery.

This division of responsibilities leverages the familiarity and consistency of SAS for core analytics together with the interactivity and flexibility of Python for user engagement and dissemination. The resulting system supports transparent, reproducible analyses whose outputs are readily consumable in web contexts, facilitating both scholarly communication and iterative exploration.

SAS

A SAS program for this workflow should be structured so that key analytical choices can be changed without editing the core logic. The program needs clearly identified parameters—such as input datasets, variable selections, model specifications, and filtering criteria—that determine its behavior. Making these elements explicit supports reproducibility, documentation, and safe reuse across studies and environments, especially when non-expert users will drive execution through a Python interface.

Parameterization in SAS is most effectively handled with macros. At the top of the program, define macro variables (for example, %let dataset=...; %let vars=...;) or lightweight macro wrappers around common tasks, so that downstream DATA steps and PROCs reference these macro variables rather than hard-coded values. This convention localizes all user-adjustable settings in a single, readable block, leaving the analytical code intact and easier to review, test, and version.

Python then acts as the external controller by updating those macro values before submission. Using an f-string, Python can interpolate validated user choices into the macro definitions, producing a complete SAS program where only the header block changes across runs. To preserve safety and clarity, constrain inputs to enumerated, whitelisted options and ensure any identifiers or text are properly quoted to avoid syntactic errors or unintended code paths. This separation keeps the SAS program deterministic while enabling flexible runtime configuration.

Finally, the program should be prepared to produce web-friendly outputs through SAS's Output Delivery System. By targeting the default ODS HTML destination and organizing tables, listings, and graphics under well-labeled titles, the results are immediately suitable for embedding or serving in web applications. Combined with systematic log capture, this approach yields outputs that are easy to render, audit, and archive, aligning with transparent, reproducible academic workflows.

SASPY

SASPy is an open-source Python package from the SAS Institute that bridges Python and SAS by allowing a Python program to establish a session with a local or remote SAS server and execute SAS code. It supports multiple connectivity backends—such as local sessions, IOM/COM to enterprise servers, and SSH-based sessions—configured through a reproducible sascfg.py profile that specifies authentication, encodings, and session options. This integration lets researchers leverage SAS's validated statistical procedures and enterprise data handling while orchestrating analyses from Python, which is often the preferred environment for scripting, parameter management, testing, and automation in academic workflows.

From a Python session, SASPy exposes a high-level API to submit SAS code, invoke many SAS PROCs through Pythonic wrappers, and move data between pandas DataFrames and SAS libraries. The central mechanism used in this paper is the .submit() method, which takes SAS code as a string, executes it in the connected SAS session, and returns structured access to both the SAS log and generated outputs. Because SASPy can display the exact SAS statements it runs, it supports transparency and method documentation; learners can see the native SAS syntax, and authors can report the precise code used for analyses, aiding reproducibility and peer review.

A practical strength of SASPy is the ability to parameterize SAS programs with Python f-strings. By interpolating Python variables into SAS code passed to .submit(), Python effectively acts as a lightweight macro processor: dataset names, variable lists, model specifications, and filters can be assembled at runtime without manual editing of SAS files. Since interpolated values become executable SAS statements, user inputs should be constrained to whitelisted, validated options to avoid requiring SAS expertise and to prevent invalid or unsafe code paths. This separation of concerns—Python handling configuration, control flow, and input validation; SAS executing the analytical workload—supports clear governance and repeatability across environments.

SASPy also facilitates inspection and dissemination of results. Logs can be captured systematically to check errors, warnings, and notes, providing an auditable record of execution and aiding diagnosis of data or convergence issues. Outputs produced through SAS's Output Delivery System are readily consumable; the default ODS destination is HTML, which can be embedded directly in web applications, dashboards, or electronic notebooks.

FLASK/DASH

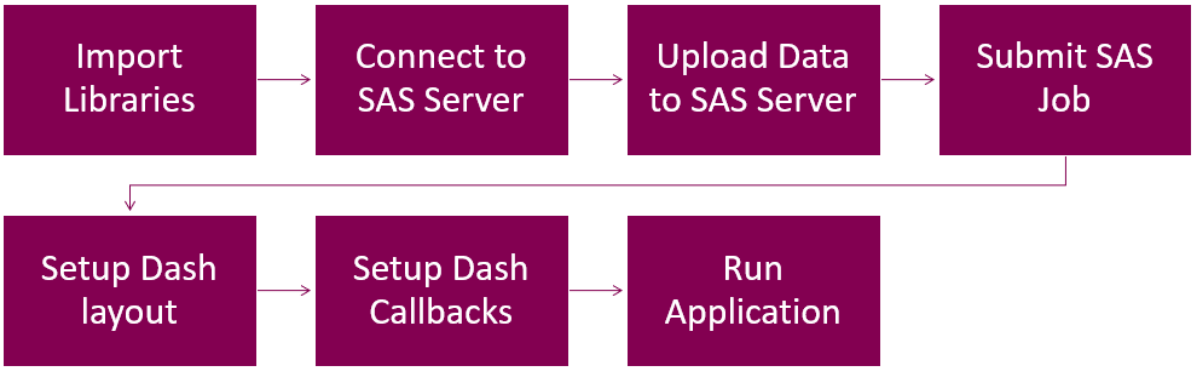
Both Flask and Dash are Python-based frameworks used to build web applications, share the same underlying server foundations (Dash runs on top of Flask), and integrate naturally with the broader Python data and scientific ecosystem. They encourage modular design, and can be combined within a single project—Dash for interactive UI components and Flask for custom endpoints or services.

Flask is a lightweight, modular web framework for Python that follows a microframework philosophy: it provides the essentials—URL routing, and HTML templating through Jinja2—while leaving databases, authentication, and other features to optional extensions. Its simplicity, explicitness, and Pythonic design make it popular for building REST APIs, microservices, and custom web applications ranging from small prototypes to production systems.

Dash is a high-level Python framework for building interactive, data-driven web applications, particularly dashboards and analytic interfaces, developed by Plotly. It sits atop Flask for the server layer, React for the front end, and Plotly for charting, letting you define the entire app—layout, components, callbacks, and state—in pure Python without writing JavaScript. Dash provides a declarative component model (graphs, tables, controls), reactive callbacks for interactivity, and a deployment story compatible with standard Flask servers, making it well-suited for rapid development of data visualization tools in scientific, business intelligence, and operational analytics contexts.

Comparing the two, Flask is a general-purpose web framework that gives you maximum flexibility to design your routes, templates, and architecture, while Dash is specialized for building interactive dashboards on top of Flask with a batteries-included component system and front-end integration. Use Flask when you need a custom web app or API with fine-grained control over stack choices and behavior; use Dash when your primary goal is to create rich, interactive visualizations and dashboards quickly in Python without front-end coding. In practice, Dash can be viewed as an opinionated layer that leverages Flask, so teams sometimes combine them—serving standalone Flask endpoints alongside Dash apps within the same project to meet broader application needs.

VISUALISATION TOOL EXAMPLE



The process begins with importing libraries. In Python, this step brings into scope the modules that provide the application's core capabilities: Dash (and Plotly) for building the web interface and rendering charts, saspy for establishing a controlled connection to SAS and executing SAS programs, and standard Python utilities such as pandas for tabular data handling, os or pathlib for paths and configuration, and possibly logging for observability. Dash supplies a declarative model to define layouts and callback functions, while Plotly underpins interactive, browser-rendered figures. Saspy exposes a high-level Python API that manages SAS sessions, submits data and code, and moves datasets between Python and SAS, typically through secure connections configured via a SAS profile.

Next, the application connects to the SAS server. With saspy, this involves creating a SAS session bound to a configuration profile specifying connection method (for example, IOM, STDIO, or HTTP/Compute), host, port, and authentication. The session encapsulates state on the SAS side, including librefs, macro variables, and work libraries. Robust applications validate the session upon startup, handle credentials securely, and implement error handling and reconnection logic. This step is crucial because all subsequent computational stages rely on SAS resources being reachable and responsive. In many deployments, the SAS server provides managed libraries, access controls, and compute queues; saspy respects these controls while giving Python code a simple submit mechanism and data exchange functions.

The application then uploads data to the SAS server. Depending on the source, data may originate from local files, cloud storage, databases accessed by Python, or be provided interactively. Saspy supports transferring pandas DataFrames into SAS datasets using functions like df2sd, placing them into a designated libref or the WORK library. Alternatively, if SAS has direct access to the source data (for instance, through LIBNAME engines or database connectors), the application can issue SAS steps to read the data in place, minimizing network transfer and maintaining governance. The upload step should normalize column types and names, apply any required privacy or compliance rules, and confirm successful creation of SAS datasets. Where volumes are large, developers often prefer server-side ingestion to avoid Python memory bottlenecks and to leverage SAS's optimized IO.

With data resident in SAS, the application submits SAS jobs to compute results aligned to the current user-selected filters. Saspy's submit or run functions accept SAS code as strings; the code can include PROC steps, DATA steps, macros, and ODS outputs. Filters chosen in the UI—such as date ranges, product categories, or geographic segments—are passed from Dash callbacks into parameterized SAS code, often via macro variables or conditional WHERE clauses. The SAS job produces either summarized datasets, analytical outputs (for example, statistical estimates), or graphics-ready tables that Dash will consume. Careful design isolates the computational kernel into reusable SAS programs or macros, enabling reliable, repeatable generation of outputs each time filters change. The application should capture logs and return errors to the front end gracefully, with clear messages if inputs are invalid or if data are unavailable.

After establishing the computational pipeline, the Dash layout is set up. Dash describes user interfaces as trees of components, including controls such as dropdowns, sliders, date pickers, and input fields, alongside visual containers such as graphs, tables, and tabs. The layout defines the structure and appearance: where filters are placed, how selections are grouped, and where results are displayed. Each visual element is an interactive React component in the browser, while the Python side declares the component properties and initial state. The application can include headings and explanatory text, responsive grid design, and accessibility considerations. The layout stage also determines which component properties will be targets for callbacks, making them reactive to changes in user input.
The heart of interactivity resides in Dash callbacks. A callback is a Python function decorated with inputs and outputs that ties user actions to updates in the interface. In this application, when users change a filter, the corresponding callback gathers selected values, constructs or selects appropriate SAS code, and submits the job through saspy. Once SAS finishes, the callback retrieves results—either by converting SAS datasets into pandas DataFrames or by reading chart-friendly structures—and then builds or updates Plotly figures. Dash's reactive model ensures that each filter change triggers a fresh computation and a new visualization, so users can iteratively explore the data. To maintain responsiveness, long-running SAS jobs can be optimized with pre-aggregations, indexed datasets, or caching strategies keyed by filter states. Concurrent users are handled via stateless callbacks and per-request SAS interactions or session pooling, depending on infrastructure. Error handling and timeouts protect the user experience, and logs provide traceability between UI events and SAS submissions.

Finally, the application is run. In development mode, the Dash server starts locally, serving the UI and orchestrating callbacks. In production, the app is hosted behind a WSGI server and a suitable proxy, with secure connections to the SAS server configured via saspy profiles. This stage includes environment configuration, loading secrets securely, and enforcing authorization if needed. Once running, the system enters a stable loop: users manipulate filters, Dash triggers callbacks, saspy submits SAS code, SAS computes aggregates or transformations, results flow back into Python, and Dash re-renders interactive visuals. Because callbacks are purely reactive, new visualizations are continually generated as users select different filters, enabling rapid, on-demand exploration with SAS performing the underlying analytics and Dash providing a fluid front end.

Across these stages, saspy provides the bridge between Python and SAS, translating Python function calls into SAS session commands and returning logs and data in usable forms, while Dash delivers a declarative, component-based approach to building a dynamic web interface driven by user events.

THE CODE

To facilitate reproducibility and transparent dissemination of the computational workflow, the complete source code implementing the procedure described above is provided in a Google Colab notebook (referenced as "the Colab notebook" hereafter). Google Colab is a cloud-hosted Python environment that provisions a standardized runtime. For the purposes of this paper, Colab serves as a controlled execution environment analogous to a lightweight container. This setup minimizes variability arising from local configuration differences (for example, operating system, dependency versions, and hardware), thereby supporting consistent execution across users and sessions.

CONCLUSION

This framework offers an experimental approach to modernizing legacy SAS programs. While illustrated through a visualization use case, its application can extend more broadly to enhancing user interaction with existing workflows. In particular, it could streamline complex macro invocations—such as those with distinct parameter sets for SDTM versus ADaM inputs—by encapsulating options within a web application that employs dropdowns and context-aware controls. Such an interface can reduce reliance on lengthy documentation and lower the cognitive load associated with intricate parameterization.

A key consideration is the licensing model for SAS. If execution requires additional licenses for each end user, the approach may become prohibitively expensive at scale. Under such constraints, use cases that primarily benefit users already covered by existing SAS licenses are likely to be more cost-effective.

As data volumes increase and analytical methods continue to evolve, this framework can serve as a complementary tool in the statistical programming toolkit. Empowering statistical programmers to select the most appropriate technology for a given task is essential to maximizing efficiency and impact.

REFERENCES:

https://colab.research.google.com/drive/1u8V2sW_NHPT_FvqYpKz5SsSOkBUnuxsz?usp=sharing

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)
Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Jamie Pilkington
Company: AstraZeneca
Address: Avinguda de Roma, 81, L'Eixample, 08029 Barcelona
Work Phone: +34646726716
Email:Jamie.pilkington1@astrazeneca.net