

Enhancing Your SAS® Viya® Workflows with Python: Integrating Python's Open-Source Libraries with SAS® using PROC PYTHON

Ryan Paul Lafler, Premier Analytics Consulting, LLC

Miguel Ángel Bravo Martínez Del Valle, SDSU Climate Informatics Lab

ABSTRACT

Data scientists, statistical programmers, machine learning engineers, and researchers are increasingly leveraging a growing number of open-source tools, libraries, and programming languages that can enhance and seamlessly integrate with their existing data workflows. One of these integrations, built into SAS® Viya®, is its pre-configured Python runtime integration, **PROC PYTHON**, that gives SAS programmers access to Python's open-source data science libraries for wrangling and modeling structured and unstructured data alongside the validated procedures provided in SAS. This paper demonstrates how to install and import external Python libraries into their SAS Viya sessions; generate Python scripts containing methods that can import, process, visualize, and analyze data; and execute those Python methods and scripts using SAS Viya's **PYTHON** procedure. By integrating the added functionalities of Python's libraries for data processing and modeling with SAS procedures, SAS programmers can enhance their existing data workflows with Python's open-source data solutions.

INTRODUCTION

Beginning in 2021, SAS® introduced the **PYTHON** procedure (**PROC PYTHON**) into their SAS® Viya® cloud analytics platform to integrate Python's growing community of open-source libraries and packages with their proprietary statistical programming language. This relatively new procedure integrates Python's objects, methods, libraries, and vibrant open-source community with the verified procedures developed by SAS. This integrated workflow offers advantages for both Python and SAS programmers, allowing them to co-exist in an integrated SAS Viya environment to build data processing pipelines and train supervised and unsupervised machine learning algorithms using Scikit-Learn, perform statistical modeling with **PROC GLM** and **PROC REG**, and handle a variety of structured, semi-structured, and unstructured data sources using Python libraries including Pandas, NumPy, OpenCV, Dask, and Xarray that come bundled with efficient, pre-built methods.

This application-oriented paper will demonstrate how to establish a connection between Python and SAS within SAS Viya using **PROC PYTHON** to execute Python scripts, install and import Python libraries within SAS Viya, define custom Python methods to help import and process data, convert Pandas DataFrame objects into recognizable SAS dataset objects stored in memory, and investigate those SAS datasets exported from the Python code.

This paper was written to highlight the potential integration of Python and SAS programming within the SAS Viya ecosystem. By showing this integration, organizations with existing SAS Viya workflows can experiment with, adopt, and implement scalable data processing, analytical, and predictive systems using Python's well-supported community of libraries together with the validated procedures maintained by SAS.

1. PYTHON FUNDAMENTALS

Among the most important open-source programming languages in 2024 is Python, which is an interpreted, object-oriented, and high-level programming language that features an active community of programmers and developers contributing libraries that are published on the Python Package Index (PyPi) and GitHub for anyone to download and use. As time advances, so do the languages used by organizations and businesses, which is why the integration of Python within SAS Viya offers SAS programmers new ways to extend the capabilities of SAS by incorporating open-source Python tools to assist in processing, visualizing, analyzing, and modeling data alongside the validated procedures created by SAS. Beginning with the basics, what exactly are the classes, objects, data structures, methods, and libraries in Python used for storing, accessing, manipulating, and handling data?

1.1) PYTHON CLASSES, OBJECTS, DATA STRUCTURES, METHODS, AND LIBRARIES

A *class* in Python is a template for creating *objects*, or unique instances, belonging to that class. A Python class acts as a blueprint for objects to share the same sets of attributes (options and methods) that can be customized and re-used when assigning unique

objects. In other words, a class encapsulates the data and functions into a single, cohesive, and re-usable blueprint to customize objects from. Classes encourage the creation of complex and unique data structures through abstraction and code re-usability.

Python stores objects in different forms of mutable (editable) and immutable (permanent) data structures, including:

1. **List** (indexed, mutable structure containing elements of the same or different types)
2. **Tuple** (indexed, immutable structure of elements that cannot be changed once created)
3. **Set** (unordered list of elements that cannot contain duplicate values)
4. **Dictionary** (unordered, mutable JSON-like structure that relies on key → value mappings)

Methods in Python are typically defined within a class to carry-out tasks when creating objects. For example, a method can change the attributes of an object as defined by its class and return some type of value(s) to the user. Methods are defined by using the *def* keyword and are invoked using the dot notation (e.g., *object.method()*). Methods can modify/update/alter an object's state and perform operations using the object's parameters (data) passed to it by the programmer.

A Python *library* consists of a set of modules that contain re-usable, pre-defined, sharable, and customizable methods and classes developed and released in repositories like PyPi and GitHub by third-party authors. These modules typically include classes, methods, and additional dependency libraries that can be imported into Python programs to perform specialized (and often, optimized) tasks that reduce code duplication. All the methods of a library can be imported at once, or more efficiently, specific methods can be called only when needed. Some libraries, such as Scikit-Learn and TensorFlow, load more slowly when imported in their entirety as opposed to quicker-to-load libraries like NumPy and Pandas. All libraries must go through a one-time installation in the Python environment and must then be imported into the active Python session either in their entirety or by calling specific methods on an as-needed basis. Libraries facilitate code deployment by providing programmers with pre-built tools to perform specialized tasks with optimal efficiency and minimal code duplication (eliminates "re-creating the wheel").

NumPy extends the capabilities of Python to vectorize operations across entire columns (or rows) of data, rather than iterating operations sequentially one element at-a-time. Scikit-Learn integrates well with NumPy, SciPy, and Pandas to build data processing pipelines, train ML algorithms, and deploy them using minimal code and pre-defined classes in the form of data transformers and model estimators. These libraries greatly contribute to the scalability and efficiency of Python when building data workflows, also providing programmers with standardized frameworks of pre-built methods and documentation that teams can implement cohesively.

After covering these fundamental Python concepts, let's delve a little deeper into one of the most important Python libraries in the realm of data science and how it extends and enhances the capabilities of Python's more rudimentary data structures: the Pandas ecosystem.

1.2) UNDERSTANDING THE PANDAS LIBRARY, DATAFRAME OBJECTS, AND SAS DATASETS

Pandas is an open-source Python library that provides high-performance data structures along with methods and tools for data cleaning, imputation, feature engineering, data analysis, and its own sets of feature types (called *dtypes*) that includes support for integer, float, string (*object*), categorical, datetime, and boolean types. Pandas is primarily used when working with tabular data (i.e., flat file formats such as CSV, JSON, spreadsheets) and includes methods for scraping data from the web and working with unstructured text-based data sources.

Pandas facilitates tasks including data imports, data cleaning, filtering, subsetting, aggregating, summarizing, transforming, time series management (using its vectorized datetime functions), and data exports using its ecosystem of well-defined methods. The library offers two main structures: the DataFrame (tabular data comprised of rows and columns) and a Series (a vector of data stored as either a row or column inside a DataFrame). Pandas DataFrames are created from Python dictionaries containing lists of elements where the keys of the dictionary denote the DataFrame's column names, and the dictionary's values are contained within Python lists to produce the columns of that DataFrame.

How are Pandas DataFrames and SAS datasets alike? Both share similarities in how they handle and organize data in a 2-dimensional tabular format, particularly for data analysis, querying, and manipulation tasks. Pandas DataFrames and SAS datasets both support:

- Mixed feature types including numeric, string, categorical, and datetime column types
- Encoding and representation of missing values (Pandas uses 'NaN' while SAS employs '.' to denote missing values)
- Merging, joining, stacking, and concatenation between different datasets

- Generating statistical summaries that recognize and account for missing values

Although Pandas DataFrames and SAS datasets share many similarities in their formats and structures, there are, however, several key differences with how data are retrieved, indexed, processed, and the methods in which data can be manipulated and transformed. These differences include:

- Pandas DataFrames must be loaded entirely in Python’s memory (RAM), making methods such as data chunking and lazy evaluation necessary with additional libraries like Dask and Polars. SAS datasets are stored on disk, allowing SAS to naturally (out-of-the-box) process big data concurrently without running into memory constraints or limitations.
- Methods for processing, sorting, filtering, transforming, and engineering new features are built into DataFrame objects. These operations are performed entirely in Python’s memory. Python programmers can even define, vectorize, and map their own custom functions to DataFrames. SAS datasets require the use of pre-defined procedures (PROCs) to perform similar functions.
- Pandas DataFrames support automatic and custom (multi- and hierarchical) indexing to denote rows and columns, while SAS datasets require that indices be *explicitly* created on one or more columns. Since SAS datasets read data from the disk, as opposed to Python DataFrames where the entire index is stored in memory, filtering by indices on a SAS dataset for moderate-sized data is slower than its Python counterpart.

Therefore, when using **PROC PYTHON** in SAS Viya and importing data using the Pandas library, the entire dataset will be imported into the Python session’s memory (stored for immediate access with RAM). This stands in contrast to the **DATA** step in base SAS where the data are read from the disk and chunks of the data are loaded into SAS as needed (constant I/O for read/write operations into SAS). **Figure 1** shows a comparison of how the Pandas DataFrame and SAS dataset visually appear, respectively, using the same tabular dataset source. Notice how Python automatically indexes the first observation starting at 0, and then increments by 1 from its 0-based index (this is a significant and noteworthy difference from SAS).

	PatientID	Age	Gender	Ethnicity	SocioeconomicStatus	EducationLevel	BMI	Smoking	AlcoholConsumption	PhysicalActivity	...	Itching	Qualiti
0	1	71	0	0	0	2	31.069414	1	5.128112	1.676220	...	7.556302	
1	2	34	0	0	1	3	29.692119	1	18.609552	8.377574	...	6.836766	
2	3	80	1	1	0	1	37.394822	1	11.882429	9.607401	...	2.144722	
3	4	40	0	2	0	1	31.329680	0	16.020165	0.408871	...	7.077188	
4	5	43	0	1	1	2	23.726311	0	7.944146	0.780319	...	3.553118	

KIDNEY_SAS												
Table rows: 1659 Columns: 54 of 54 Rows 1 to 200												
	Ⓜ PatientID	Ⓜ Age	Ⓜ Gender	Ⓜ Ethnicity	Ⓜ SocioeconomicStatus	Ⓜ EducationLevel	Ⓜ E					
1	1	71	0	0	0	2	31.0694					
2	2	34	0	0	1	3	29.6921					
3	3	80	1	1	0	1	37.3948					
4	4	40	0	2	0	1	31.3296					
5	5	43	0	1	1	2	23.7263					

Figure 1. Pandas DataFrame (top) and the equivalent SAS dataset (bottom) showing the first-5 observations. Notice how Python implements automatic indexing starting at 0.

1.3) DEFINING A CUSTOM PYTHON METHOD FOR IMPORTING CSV DATA SOURCES WITH PANDAS

Leveraging pre-built methods from the Pandas library, let’s define our own Python method that can accomplish the following data import tasks:

- Import any delimited file type (i.e., comma, tab, colon) into the Python session as a Pandas DataFrame
- Allow programmers to specify the input file name, exported SAS dataset name, and type of delimiter
- Elements in the first row of the delimited file are used as column header names
- Convert the Pandas DataFrame from an in-memory dataset to a SAS dataset stored on-disk
- Allow programmer to determine if data is temporarily saved or permanently saved on-disk using a *LIBNAME*

This custom method, defined in (1.3.1), is saved inside of a separate Python script file (saved with the `.py` file extension) as `import_data.py`. Python script files can contain as many user-defined methods, classes, and objects that can be imported and used directly inside of **PROC PYTHON** through the `INFILE` option and pointing it to the script file's location in SAS Drive. For simplicity, the authors recommend saving the Python script file, SAS program file, and CSV dataset to the same working folder inside of SAS Viya's SAS Drive.

Python Code (1.3.1) | Defining a Python Data Import Method in a Separate Script (.py) File

```
def import_data(self, file_name, export_name, delim=',', libname='/') :
    try :
        # Import using the Pandas library
        df = pd.read_csv(
            file_name ,
            delimiter = delim ,
            header = 'infer'
        )

        # Export pathway
        export_path = f'{libname}.{export_name}'
        # Transform into a SAS Dataset
        SAS.df2sd(df, export_path)

    except :
        raise Exception(
            f'There was an issue importing {file_name}'
        )
```

The code shown in (1.3.1) defines a method called `import_data()` that imports any delimited data source as a Pandas DataFrame and converts it to a temporary SAS dataset. **PROC PYTHON** automatically links the SAS session to the Python environment, with this connection stored in an object called `SAS`, using the **SASPy** library to do this when **PROC PYTHON** is executed. Therefore, when converting the DataFrame to a SAS dataset, the object linking the SAS and Python sessions together, called `SAS`, includes a method that does exactly that: `SAS.df2sd()`. The Pandas DataFrame is then exported as a SAS dataset that can be accessed outside of the Python environment using any of the procedures built into SAS.

The `import_data()` method shown in (1.3.1) also incorporates a `try-except` statement to assist in debugging. Should an error occur anywhere within the `try` block, then the `except` block executes and the error message is returned. If the `try` block executes successfully, then the error message in the `except` block is not returned.

By default, the `LIBNAME` option is set to store the SAS dataset in on-disk temporary storage, typically called the `WORK` directory. Any files stored inside of this temporary storage are deleted when the SAS Viya session is terminated. Programmers can persist the resulting SAS dataset to permanent storage in the SAS Drive by changing the `LIBNAME` option to a permanent folder pathway.

2. CONNECTING TO PROC PYTHON IN SAS VIYA

With the Python script file containing the import method saved and exported, let's focus on initializing a Python session within SAS Viya using **PROC PYTHON**. Let's first investigate the version of Python that has been installed in **PROC PYTHON**. Then let's examine which open-source libraries (and their versions) are installed in the **PROC PYTHON** environment.

Traditionally, the interaction between SAS and Python happens through the **SASPy** library. **PROC PYTHON** simplifies this process by coming bundled with a pre-configured (and updated) version of Python that does not need to be connected to the SAS environment, since **PROC PYTHON** is already running *within* the SAS Viya environment. This simplifies data workflows and session management between Python and SAS but limits the programmer to only using the pre-defined Python environment and installed libraries within **PROC PYTHON**. More flexibility is allowed by using the **SASPy** library which allows users of SAS 9.4 and SAS Viya to connect their own customized Python environments to their SAS sessions, albeit with more complexity, time-to-setup, maintain, and manage sessions.

2.1) CHECKING THE INSTALLED PYTHON VERSION IN SAS VIYA

Knowing the current version of Python installed within **PROC PYTHON** is helpful when handling libraries, managing their dependencies, and avoiding deprecation and conflicts between libraries. This can be achieved by importing Python's `sys` library, which is installed alongside Python natively, and allows the programmer to interact directly with the Python runtime environment installed within **PROC PYTHON**.

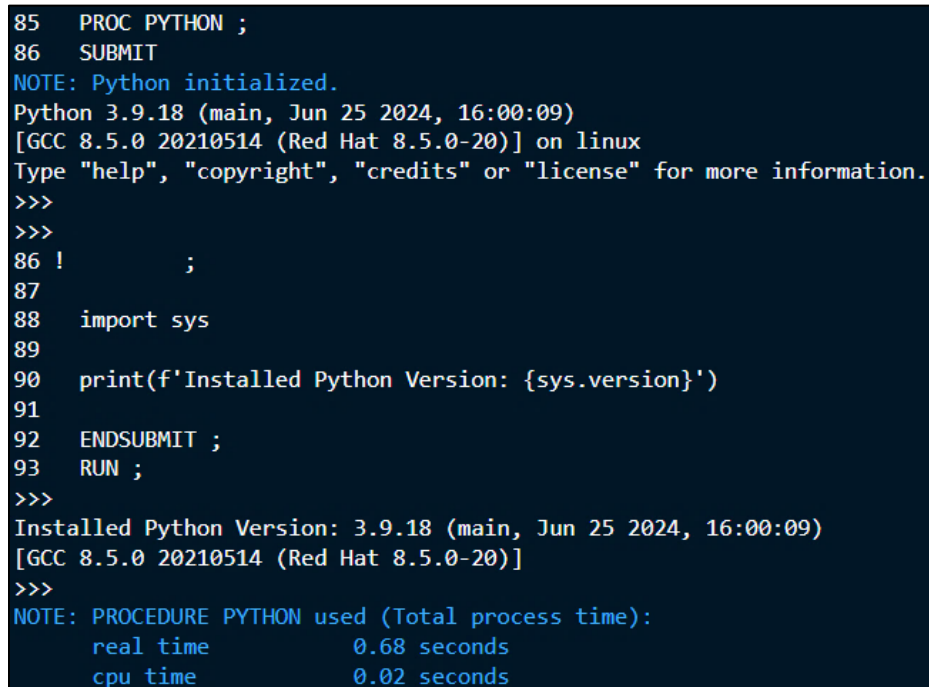
The following **PROC PYTHON** code in (2.1.1) includes a code chunk that prints the currently installed Python version natively used in the procedure. Keep in mind that the version of Python installed in **PROC PYTHON** at the publication of this paper may be different than what is shown in the resulting log output.

SAS Code (2.1.1) | Checking the Python Installation in SAS Viya

```
/* Check and verify Python Installation in SAS Viya */
PROC PYTHON ;
    SUBMIT ;

import sys
print(f'Installed Python Version: {sys.version}')

    ENDSUBMIT ;
RUN ;
```



```
85 PROC PYTHON ;
86 SUBMIT
NOTE: Python initialized.
Python 3.9.18 (main, Jun 25 2024, 16:00:09)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-20)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
86 !          ;
87
88 import sys
89
90 print(f'Installed Python Version: {sys.version}')
91
92 ENDSUBMIT ;
93 RUN ;
>>>
Installed Python Version: 3.9.18 (main, Jun 25 2024, 16:00:09)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-20)]
>>>
NOTE: PROCEDURE PYTHON used (Total process time):
      real time          0.68 seconds
      cpu time           0.02 seconds
```

SAS Log (2.1.1)

The resulting SAS log produced from running the **PROC PYTHON** code block in (2.1.1) shows the installed version of Python to be 3.9.18 on Red Hat version 8.5.0-20. When executing a **PROC PYTHON** block of code for the first time, the log will also show the Python version and the operating system (OS) that it's running on, which is the Linux distribution Red Hat.

2.2) VERIFYING INSTALLED PYTHON LIBRARIES WITHIN SAS VIYA'S PROC PYTHON

After understanding the Python runtime version within **PROC PYTHON**, it's helpful to know which libraries come pre-installed within the SAS Viya procedure, especially since the pre-configured Python environment receives occasional updates.

This is accomplished using the `sys` and `subprocess` libraries, both natively installed alongside Python, to issue `pip` terminal commands that interact with the Python runtime. The **PROC PYTHON** code shown in (2.2.1) issues a `pip` terminal command that retrieves the alphabetical listings of all the installed libraries within the Python runtime (including their specific versions) and prints the listing of libraries to the SAS log.

SAS Code (2.2.1) | Checking the Installation of Python Libraries in SAS Viya

```
/* Check installation of Python libraries in SAS Viya */
PROC PYTHON ;
    SUBMIT ;
```

```

# Import subprocess to run pip commands by accessing the Python terminal in SAS Viya
import sys
import subprocess

# Run the pip freeze command to list all installed packages
installed_packages = subprocess.check_output([sys.executable, '-m', 'pip', 'freeze'])

# Decode and print the result
print(installed_packages.decode("utf-8"))

    ENDSUBMIT ;
RUN ;

```

```

96 /* Check installation of Python libraries in system */
97 PROC PYTHON;
98     SUBMIT
NOTE: Resuming Python state from previous PROC PYTHON invocation.
98 !         ;
99
100 # Import subprocess to run pip commands
101 import subprocess
102
103 # Run the pip freeze command to list all installed packages
104 installed_packages = subprocess.check_output([sys.executable, '-m', 'pip', 'freeze'])
105
106 # Decode and print the result
107 print(installed_packages.decode("utf-8"))
108
109     ENDSUBMIT;
110 RUN;
>>>
abs1-py==2.1.0
aiohttp==3.9.5
aiosignal==1.3.1
annotated-types==0.7.0
asttokens==2.4.1
astunparse==1.6.3
async-timeout==4.0.3
attrs==23.2.0
blinker==1.8.2
SAS Log (2.2.1)

```

The SAS log from the code in (2.2.1) shows only a small subset of the installed Python libraries within **PROC PYTHON**. These include libraries that assist with processing different types of data (i.e., Pandas, GeoPandas, NumPy, SciPy, Pillow, NLTK, Regex); accessing data from the cloud and servers on the web (i.e., Filesystem Spec, AIOHTTP); visualizing data (i.e., Matplotlib, Plotly, Seaborn); performing statistical modeling and training machine learning algorithms (i.e., Statsmodels, Scikit-Learn, XGBoost); and developing deep learning architectures (i.e., TensorFlow, PyTorch, Keras, NVIDIA CUDA toolkit).

3. LEVERAGING PROC PYTHON TO BUILD PYTHON WORKFLOWS IN SAS

With the Python script file containing the custom method saved to the SAS Drive folder and PROC PYTHON coming pre-installed with the Pandas library, let's use PROC PYTHON to import a CSV dataset from within the SAS Drive using Pandas and then export the Pandas DataFrame as a SAS dataset temporarily saved to the *WORK* directory.

3.1) DEFINING REFERENCES TO THE CSV DATASET AND PYTHON SCRIPT FILE

The dataset and Python script file must first be uploaded into the SAS Drive. Following this, the next step is to define the pathways to the Python script file and the CSV dataset itself. The SAS code in (3.1.1) achieves the following:

- Creates a file reference called *SCRIPT* for the Python script that is located on the DISK using the FILENAME statement
- Declares a macro variable called *REFDATA* that stores the CSV dataset pathway to be used inside of **PROC PYTHON**

In the SAS code in (3.1.1), two items are defined: a file reference for the Python script file and a macro variable containing the pathway to the CSV dataset. By defining a file reference with the *FILENAME* statement, programmers can use the shorthand reference name *SCRIPT* instead of its full pathway when importing the Python script into **PROC PYTHON**. The *REFDATA* macro variable allows the pathway to the CSV dataset to be defined in SAS and shared with the Python session inside of **PROC PYTHON** as a string input. Code (3.1.1) shows how to implement this in SAS Viya.

SAS Code (3.1.1) | Importing Libraries, Data, and Saving to a SAS Dataset with PROC PYTHON

```
/* Python script file input */
FILENAME SCRIPT DISK '/export/viya/homes/{user-email}/casuser/import_data.py' ;

/* Define macro variable for dataset path */
%LET REFDATA = /export/viya/homes/{user-email}/casuser/Chronic_Kidney_Disease_data.csv ;
```

3.2) DEFINING THE GLOBAL LOCATIONS TO THE DATASET AND PYTHON SCRIPT FILE

Initializing a Python session inside of SAS Viya can be executed by calling **PROC PYTHON** and specifying the Python script file's pathway and location using the *INFILE* option (if necessary: only used when Python code is saved to an external file with the *.py* extension). In this example, the Python code from (1.3.1) was saved to an external Python script file, using the *.py* extension, containing a custom method for importing delimited data sources using the Pandas library and then exporting that Pandas DataFrame as a SAS dataset that can be accessed by any of SAS Viya's procedures.

When executing Python code inside of **PROC PYTHON**, programmers must specify the *SUBMIT*; and *ENDSUBMIT*; statements to mark the beginning and end of the executed Python code. The Python code is then executed sequentially, line-by-line, until the entire chunk of code either results in completion (all lines execute without errors) or termination (one or more lines of code produce an error).

The **PROC PYTHON** code shown in (3.2.1) can be summarized by the following steps:

1. Imports the Python script file,
2. imports the Pandas library into the active Python session,
3. Retrieves the macro variable string pathway pointing to the CSV dataset and stores it as a Python string in *filename*,
4. Imports the CSV dataset as a Pandas DataFrame object entirely in Python's memory,
5. Coerces the Pandas DataFrame object to a SAS dataset stored on disk in the temporary *WORK* library

SAS Code (3.2.1) | Importing CSV Data using Pandas and Exporting as a SAS Dataset

```
/* Submit Python code into PROC PYTHON */
PROC PYTHON INFILE=SCRIPT ;
    SUBMIT ;

# Python library imports
import pandas as pd

# Specify filename pathway
filename = SAS.symget('REFDATA')

# Retrieve method from Python script file to load the data
import_data(
    file_name = filename ,
    export_name = 'KIDNEY_SAS'
)

    ENDSUBMIT ;
RUN ;
```

```

>>>
135 proc printto
135! log='/opt/sas/viya/config/var/tmp/compsrv/default/edcd2a8d-f34d-4148-a968-577345d10cf0/SAS_workDB4200000202_sas-compute-server-
135! b5e7b800-42b2-4932-8634-13ba62305b81-6956/sas2py.log';run;filename sock socket 'localhost:53021' recfm=V termstr=LF
NOTE: PROCEDURE PRINTTO used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

>>>
/export/viya/homes/rplaf1er@sdsu.edu/casuser/Papers/PROC-PYTHON/Chronic_Kidney_Disease_data.csv
>>>
NOTE: PROCEDURE PYTHON used (Total process time):
      real time           0.17 seconds
      cpu time            0.03 seconds

```

SAS Log (3.2.1)

The code inside of **PROC PYTHON** can also retrieve and edit macro variables defined in the SAS session, which in the example shown in (3.2.1), contains a string macro variable defined by the **%LET** statement called **REFDATA** that contains the pathway to the location of the desired CSV file. Without having to import any additional libraries, **PROC PYTHON** allows the programmer to store the SAS macro variable as a python object, which in this case, is a Python string object called *filename*.

Once this code block is executed, a new SAS dataset (.sas7bdat file) called **KIDNEY_SAS** is created and stored (temporarily) inside of the **WORK** library and can be accessed by any of the descriptive, visualization, statistical, and modeling procedures in SAS.

4. PERSISTING THE SAS DATASET TO PERMANENT STORAGE

Currently, the resulting SAS dataset is saved to temporary storage inside of the **WORK** library. By setting a **LIBNAME** in the SAS code that points to a persistent folder in the SAS Drive, programmers can utilize the SAS dataset across SAS Viya sessions without having to import and process the original CSV data source in Pandas.

4.1) DEFINING A PERSISTENT DATA STORAGE DIRECTORY WITH LIBNAME AND LIBREF

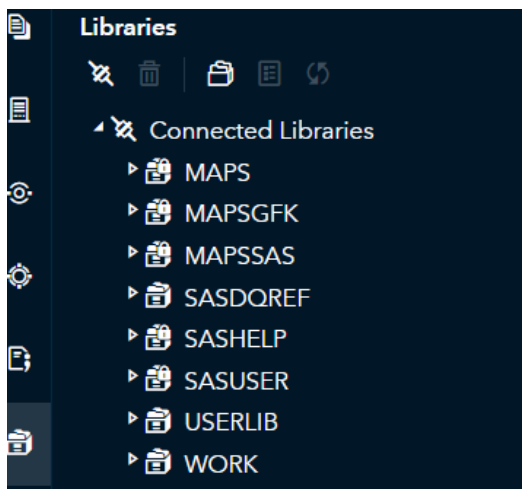
The **LIBNAME** statement allows the programmer to define a persistent directory that can be invoked using an 8-character (max) shorthand reference within SAS for convenient dataset storage. The **LIBREF** contains the reference to the dataset folder pathway while the **LIBNAME** is the shorthand library name. The **LIBNAME** statement shown in (4.1.1) defines a library named **USERLIB** that is stored in this designated folder location: `‘/export/viya/homes/{user-email}/casuser/Data’`. The `/Data` folder was created to store and persist SAS datasets between SAS Viya sessions.

SAS Code (4.1.1) | Defining a LIBNAME and LIBREF for Persistent SAS Dataset Storage

```

/* Create a persistent folder containing exported SAS Datasets */
LIBNAME USERLIB '/export/viya/homes/{user-email}/casuser/Data' ;

```



SAS Output (4.1.1)

Located inside of SAS Studio’s libraries tab, the output from (4.1.1) creates the **USERLIB** library that is added to the existing list of default (locked) libraries and the temporary **WORK** library.

4.2) USING THE LIBNAME IN PROC PYTHON TO EXPORT THE PANDAS DATAFRAME

After setting up the persistent library connected with the location in SAS Drive, let's amend the **PROC PYTHON** code to account for this by setting the *LIBNAME* option in the *import_data()* Python method to point to the *USERLIB* library and persist the exported SAS dataset to its designated reference folder. The SAS code in (4.2.1) shows how to do this.

SAS Code (4.2.1) | Using PROC PYTHON to Export and Persist the SAS Dataset Using LIBNAME

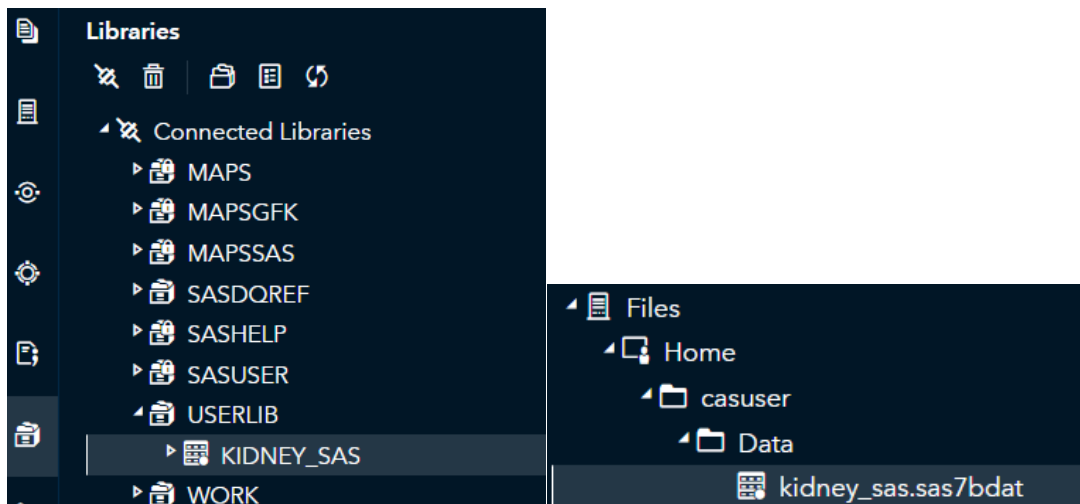
```
/* Submit Python code into PROC PYTHON */
PROC PYTHON INFILE=SCRIPT ;
    SUBMIT ;

# Python library imports
import pandas as pd

# Specify filename pathway
filename = SAS.symget('REFDATA')

# Retrieve method from Python script file to load the data
import_data(
    file_name = filename ,
    export_name = 'KIDNEY_SAS' ,
    libname = 'USERLIB' # Set the LIBNAME to the `USERLIB` library
)

    ENDSUBMIT ;
RUN ;
```



SAS Output (4.2.1)

As shown in the SAS output for (4.2.1), the original CSV dataset is successfully exported as a SAS dataset file (*.sas7bdat*) that persists between SAS Viya sessions. In summary, the original CSV dataset was imported into the **PROC PYTHON** runtime as an in-memory Pandas DataFrame and then converted into a SAS dataset that was exported and saved to the *LIBREF* location specified in the *LIBNAME* statement. This process showcases a framework for integrating the data wrangling capabilities of Python within the SAS environment to generate SAS datasets from raw data sources that are now accessible by any of the procedures in SAS workflows.

5. EXAMINING THE METADATA OF THE NEWLY CREATED SAS DATASET

Finally, let's verify the integrity of the resulting SAS dataset to ensure that the data import, conversion, and export processes performed in **PROC PYTHON** successfully transformed the raw CSV file. Let's use **PROC CONTENTS** to generate and investigate the metadata for this resulting SAS dataset.

5.1) RUNNING PROC CONTENTS TO VERIFY METADATA FOR THE SAS DATASET

The SAS code in (5.1.1) invokes **PROC CONTENTS** on the SAS dataset exported and saved to the *USERLIB* library.

SAS Code (5.1.1) | Importing CSV Data using Pandas and Exporting as a SAS Dataset

```
/* Check the contents of the created SAS Dataset */  
PROC CONTENTS DATA=USERLIB.KIDNEY_SAS ;  
    TITLE 'Metadata for the SAS Dataset' ;  
RUN ;
```

Metadata for the SAS Dataset			
The CONTENTS Procedure			
Data Set Name	USERLIB.KIDNEY_SAS	Observations	1659
Member Type	DATA	Variables	54
Engine	V9	Indexes	0
Created	10/21/2024 20:10:27	Observation Length	440
Last Modified	10/21/2024 20:10:27	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64, LINUX_POWER_64		
Encoding	utf-8 Unicode (UTF-8)		

SAS Output (5.1.1)

The **PROC CONTENTS** output shows there are a total of 1,659 observations and 54 features within this dataset. Each observation accounts for approximately 440 bytes of data on disk. Therefore, it's safe to assume that the integration of Python within SAS was successful.

CONCLUSION

PROC PYTHON is a powerful tool for leveraging a pre-configured Python runtime within the SAS Viya cloud analytics platform to import, access, process, manage, analyze, visualize, and model structured and unstructured data formats using pre-installed Python libraries. By integrating the versatility of Python with the validated procedures of SAS, programmers can greatly enhance and expand their existing SAS Viya workflows to handle more diverse types of data and define powerful methods in Python to process those datasets and export them as SAS datasets to be used anywhere within the SAS environment.

This paper investigated and discussed the types of data structures and open-source libraries available in Python for users to adopt, including the capabilities of the Pandas ecosystem and how it differs from the disk-based approach favored by SAS. An example Python and SAS integration workflow was then shown that utilized **PROC PYTHON** to import delimited data (CSV file) as a Pandas DataFrame that was then exported to a SAS dataset in temporary and persistent storage with the help of the *LIBNAME* statement. Finally, the integrity of this newly created SAS dataset was tested using **PROC CONTENTS** which successfully confirmed its integrity.

The integration of Python and SAS greatly expands the capabilities of new and existing SAS workflows to incorporate open-source solutions with the validated and well-maintained procedures in SAS.

REFERENCES

Box, J. (2023). *Running Python code inside a SAS® program*. In *Proceedings of PharmaSUG 2023 Conference* (Paper QT-165). Retrieved from <https://pharmasug.org/proceedings/2023/QT/PharmaSUG-2023-QT-165.pdf>

SAS Institute. (2024, October 17). *How PROC PYTHON Works*. SAS® Help Center. https://documentation.sas.com/doc/en/pgmsascdc/v_056/proc/p1m1pc8yl1crtkn165q2d4njnip1.htm

ACKNOWLEDGEMENTS

The authors would like to extend their appreciation to the Midwest SAS Users Group (MWSUG) 2024 Conference, the Conference Committee, the Academic Chair and the Operations Chair, and the Section Chair for accepting their paper. The authors express their gratitude to the MWSUG 2024 Conference for giving him the opportunity to present, publish, and network with professional colleagues, industry specialists, researchers, and students at this esteemed data science conference.

TRADEMARK CITATIONS

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the United States of America and other countries. ® indicates USA registration. Any other brand and product names are trademarks of their respective companies.

ABOUT THE AUTHORS

Ryan Paul Lafler is the Founder, CEO, Chief Data Scientist, and Lead Consultant at **Premier Analytics Consulting, LLC**, a data science consulting firm based in San Diego, California. Ryan is Adjunct Faculty at San Diego State University (SDSU) for the Big Data Analytics Graduate Program and the Department of Mathematics and Statistics. He also serves as a Principal Investigator (PI) and Research Scientist for the SDSU Climate Informatics Laboratory (SCIL) on funded grants and projects. Ryan's programming expertise in Python, R, SAS®, JavaScript (React.js), open-source API frameworks, and SQL has contributed to his success as a big data scientist; consultant; ML engineer; statistician; and full-stack application developer. He received his Master of Science in Big Data Analytics from San Diego State University in May 2023 following the publication of his thesis. He holds a Bachelor of Science in Statistics, a minor in Quantitative Economics, and graduated *magna cum laude* from San Diego State University. Ryan's passionate about open-source programming languages; applied Machine Learning (ML) / Deep Learning (DL) / Artificial Intelligence (AI); statistical analysis and modeling; full-stack application and interactive dashboard development; and data visualization methods.

Miguel Ángel Bravo Martínez Del Valle is a second-year Master of Science student in the Big Data Analytics Graduate Program at San Diego State University graduating in the Spring of 2025. Miguel earned his Bachelor of Science in Electronics, Robotics, and Mechatronics Engineering at the University of Málaga, Spain. He is currently a Graduate Researcher in the SDSU Climate Informatics Laboratory (SCIL) assisting in the development of full-stack climate applications; processing big climate data; training ML workflows; and programming in Python, SAS, and JavaScript (React.js). Miguel's interests include data analytics, training ML/AI workflows, and developing full-stack climate applications.

CONTACT INFORMATION

Comments, suggestions, and/or any questions are encouraged and may be sent to:

Ryan Paul Lafler, M.Sc.

Premier Analytics Consulting, LLC *and* San Diego State University
Founder, CEO, Chief Data Scientist, Lead Consultant, and Adjunct Faculty

E-mail: rplafler@premier-analytics.com

Website: www.Premier-Analytics.com

LinkedIn: www.Linkedin.com/in/RyanPaulLafler

Résumé: www.Premier-Analytics.com/ryan-paul-lafler

Miguel Ángel Bravo Martínez Del Valle

SDSU Climate Informatics Laboratory (SCIL)
Graduate Researcher and M.Sc. Big Data Analytics Student

E-mail: miguelangelbravo2000@gmail.com

Website: www.mabravo.com

LinkedIn: www.Linkedin.com/in/Miguel-Angel-Bravo/