Friends are better with Everything: Using PROC FCMP Python Objects in Base SAS

Matthew T. Slaughter, Kaiser Permanente Center for Health Research, Portland, OR

Isaiah Lankham, University of California, Office of the President, Oakland, CA

ABSTRACT

Flexibly combining the strengths of SAS and Python allows programmers to choose the best tool for the job and encourages programmers working in different languages to share code and collaborate. Incorporating Python into everyday SAS code opens up SAS users to extensive libraries developed and maintained by the open-source Python community.

The Python object in PROC FCMP embeds Python functions within SAS code, passing parameters and code to the Python interpreter and returning the results to SAS. User-defined SAS functions executing Python code can be called from the DATA step or any context where SAS functions are available. User-defined call routines can also be used in DATA steps.

This paper provides an overview of the syntax of FCMP Python objects and practical examples of useful applications incorporating Python functions into SAS processes. For example, we will demonstrate incorporating Python packages into SAS code for leveraging complex API calls such as validating email addresses, geocoding street addresses, generating highly formatted excel files, and converting a YAML file into a SAS dataset.

All examples from this paper are available at https://github.com/saspy-bffs/wuss-2022-proc-fcmp-python

INTRODUCTION

Python is a general-purpose, open-source programming language originally developed for educational purposes in the 1990s [45]. Because of its straightforward syntax, Python quickly became popular as a "glue" language [46] and has gone on to be used in many prominent commercial and scientific applications. Popular websites developed as Python applications include Disqus, Dropbox, Instagram, Pinterest, Reddit, Spotify, Uber, and YouTube [4][5]. Recently, NASA launched the Ingenuity Mars Helicopter using Python for elements of the helicopter's software, including flight modeling and data processing [7].

Because of Python's ever-increasing popularity for data science [13], many SAS users have considered adding it to their toolset. Python's deep library of open-source packages and capacity for fast, in-memory analytics make it an appealing complement to traditional SAS programming. In addition, because of its large and growing userbase, incorporating Python into existing data pipelines also gives organizations the ability to hire from a deeper pool of talent, with SAS and Python programmers able to share code and cooperate while avoiding duplication of effort.

This paper describes how Python code can be embedded inside of SAS programs using PROC FCMP, along with a series of example use cases illustrating the power of Python inside SAS. For the opposite direction, embedding SAS code into Python scripts using the Python package SASPy, see [14].

A HISTORY OF USING PYTHON WITHIN SAS

Various options for utilizing Python code inside of SAS programs have been developed over the years. The simplest approach takes advantage of the operating system's ability to launch Python scripts, passing commands to the operating system from SAS using X commands [41], CALL SYSTEM [29], SYSTASK [39], or other similar method. Unfortunately, these techniques provide no convenient way to exchange values or datasets between SAS and Python. This forces the programmer to import and export manually on both ends. In addition, network administrators might also disable shell commands in some

SAS programming environments for security reasons.

Intermediary technologies can also be used to execute Python code inside of SAS programs, including Java DATA step component objects [9] and the .NET programming language inside of PROC PROTO [15]. This generally makes it easier to exchange values or datasets between SAS and Python, but it also potentially adds the unwanted complexity of Java or .NET as a dependency. Ideally, when integrating SAS and Python, the programmer should only need to be familiar with these two languages.

FCMP Python objects, released In SAS 9.4 TS1M6 [48][51], allow Python code to be used directly by SAS [40]. As long as the Python code can be structured as a function, it can be executed directly inside PROC FCMP, or it can be wrapped in a user-defined SAS function or call routine. User-defined SAS functions can be called inside DATA step, %SYSFUNC, or any other context where SAS functions are available, and they can be used to exchange values between SAS and Python by passing in any number of arguments and returning a single value. On the other hand, user-defined call routines can only be used inside DATA steps, but they are capable of returning multiple values to SAS.

Finally, SAS Viya version 2021.1.3 introduced PROC PYTHON. This procedure allows arbitrary Python code to be embedded inside of a SAS program, with entire datasets able to be passed between SAS and Python [36]. SAS code and functions can even be nested inside Python code within the procedure. However, unlike FCMP Python objects, PROC PYTHON cannot be called from inside a DATA step or another PROC. In addition, this procedure is only available with SAS Viya, while many users continue to use SAS 9.4.

As of fall 2022, this leaves SAS Viya users with two strong options for embedding Python code in a SAS program, depending on their use case. However, for the many users and sites using SAS 9.4, FCMP Python objects are objectively the best tool available.

MAKING SAS AND PYTHON TALK TO EACH OTHER

PYTHON INSTALLATION

Before calling Python from SAS, a Python distribution must be installed in the same environment where SAS will be running. For users who are new to Python, we recommend the latest version of the analytics-oriented Anaconda distribution [1]. Any version of Python 2.7 or later can be used [48], but examples in this paper were written in Python 3.9[1] and may not work in earlier versions of Python.

Because Python is an open-source language, it's common to use additional packages beyond the pre-installed standard library. Many popular third-party packages are included with Anaconda, and additional packages can be installed with a package manager. The most commonly used package manager is a command-line program called `pip`[2] [23] with this syntax:

```
pip install <space-separated list of package names>
```

Alternatively, some online tutorials recommend syntax like

```
python -m pip install <space-separated list of package names>
```

or

```
python3 -m pip install <space-separated list of package names>
```

or even

```
python3.9 -m pip install <space-separated list of package names>
```

In each of these cases, we're telling a specific Python interpreter to use its specific `pip` module to install a package. This is especially useful when multiple versions of Python have been installed and we want to

---

[1] As of this writing, the latest version of the Anaconda distribution comes with Python 3.9. However, Python 3.10 can also be downloaded directly from https://www.python.org/downloads/

[2] Perhaps apocryphal, it's commonly said that `pip` is a recursive acronym for "`pip` installs packages."

make sure we are installing a package for the right version.

Alternatively, the Anaconda distribution also provides its own package manager called `conda`[3] [2]. The basic syntax for installing packages from the command line with `conda` is essentially the same:

```
conda install <space-separated list of package names>
```

In order to use all of the examples in this paper, the following third-party packages should be installed:

- email-validator [44]: a package for checking formatting and deliverability of email addresses

- Faker [12]: a package for generating synthetic data values

- geocoder [3]: a package for querying web APIs to turn physical addresses into latitude/longitude

- pandas [20]: a package providing a tabular data object known as a DataFrame, which is essentially the Python equivalent of a SAS dataset

- PyYAML [42]: a package for reading and writing YAML files

- requests [28]: a package for making HTTP requests

- saspy [47]: a package for executing embedded SAS code inside of Python scripts, as well as reading and writing SAS datasets[4]

- XlsxWriter [17]: a package for creating highly formatted Excel files

These eight packages can be installed simultaneously from the command line, along with any other packages they depend on, as follows:

```
pip install Faker email-validator geocoder pandas PyYAML requests saspy XlsxWriter
```

The examples in this paper also use the packages `pathlib`, `random`, and `sys`, but these are already included in the Python standard library and shouldn't need to be installed separately. Similarly, if you're using the Anaconda distribution, you should already have `pandas` and `XlsxWriter` installed. Attempting to install a package that's already available will typically just upgrade it to the latest version.

ENVIRONMENT VARIABLES

Some additional configuration is also necessary to help SAS find Python. These two environment variables must be set prior to launching SAS [48]:

- MAS_M2PATH: This variable should be set to the path of the file mas2py.py included in your SAS installation. SAS uses this file to execute Python code.

- MAS_PYPATH: This variable should be set to the path of the Python executable, which is determined when Python is installed.

In Windows, you can set environment variables for your account using the Control Panel, or by using the `setx` command from the command line[5] [18]. If SAS and Anaconda are both locally installed with default settings, the command-line version might look like this:

```
setx MAS_M2PATH="C:\Program Files\SASHome\SASFoundation\9.4\tkmas\sasmisc\mas2py.py"
```

```
setx MAS_PYPATH="C:\ProgramData\Anaconda3\python.exe"
```

---

[3] `Conda` is also an environment manager, meaning it can be used to install and manage multiple instances of Python on a particular machine, with each instance allowed to be a different version of Python with different packages installed. `Pip` can be used with another environment manager (e.g., `venv`) to achieve the same effect. See [24] for more information about `pip` and Python environments in general.

[4] In this paper, we will be using Python with SASPy inside of PROC FCMP, which will allow us to return a SAS dataset. In general, Python can be used with SASPy independently of PROC FCMP. For more information, see [14].

[5] To set either of these variables when launching SAS, use the -SET option [37]. For example, in Windows you can create a shortcut to the SAS application whose target string includes a -SET option, allowing MAS_PYPATH to be set to the path of a specific Python installation using instructions similar to [38].

For more information, including setup steps for UNIX-like operating systems, see [30].

PROC FCMP: A PRIMER

PROC FCMP allows SAS programmers to define their own SAS functions and call routines [31]. Generally, user-defined functions return a single value but can be called from any context that allows SAS function calls (including a DATA step, PROC SQL, and the macro function %SYSFUNC). On the other hand, call routines can return multiple values but are generally only usable inside of a DATA step[6].

FCMP functions and call routines must be stored in a function library for later use. You may set this library with the CMPLIB option:

```
options cmplib=work.funcs;
```

For all subsequent examples in this paper, we will assume this statement has already been submitted, saving our functions in the WORK directory. To save functions for use in another program or for sharing with other programmers, you should instead create a permanent libref with a LIBNAME statement and specify it in an CMPLIB option.

SAY "HELLO" TO USER-DEFINED FUNCTIONS

To create a simple SAS function that takes a parameter and returns a value, we can use the FUNCTION statement in PROC FCMP:

```
* SAS Code producing the output shown in Figure 1;
proc fcmp outlib=work.funcs.sas; ❶
    function hello_from_sas(name $) $ 25; ❷
        Message = 'Hello, '||name; ❸
        return(Message); ❸
    endfunc; ❹
run;

data _null_; ❺
    message = hello_from_sas('WUSS');
    put message=;
run;

%put %sysfunc(hello_from_sas(WUSS)); ❻
```

When we invoke the FCMP procedure ❶, we must specify a package name (here, sas) within a function library (here, work.funcs) that was previously defined in a CMPLIB option. The FUNCTION statement ❷ marks the beginning of a function definition, giving the function name (here, hello_from_sas), parameters[7] (here, name), and the type of the returned value (here, character with length 25)[8]. The function body ❸ then defines the variable Message and uses a RETURN statement to return the value of Message. Finally, the ENDFUNC statement ❹ marks the end of the function definition. At this point, we can call our function from a DATA step ❺, or from any other context that a SAS function can be used, such as %SYSFUNC ❻. In both cases, the value of the variable Message defined in hello_from_sas is printed to the log, as shown in Figure 1.

---

[6] Or procedures which support DATA step statements, like PROC REPORT compute blocks or PROC FCMP itself.
[7] Note that our parameter is followed by a dollar sign, indicating it's of type character. If the dollar sign were left out, a numerical argument would be assumed.
[8] In general, it's only necessary to include information about the return value when we're returning a character value. If no information is included, a numerical return value is assumed.

```
43          proc fcmp outlib=work.funcs.sas;
NOTE: No CMP or C functions found in library work.funcs.
44             function hello_from_sas(name $) $ 25;
45                Message = 'Hello, '||name;
46                return(Message);
47             endfunc;
48          run;

NOTE: Function hello_from_sas saved to work.funcs.sas.
NOTE: PROCEDURE FCMP used (Total process time):
      real time              0.05 seconds
      cpu time               0.04 seconds


49
50          data _null_;
51             message = hello_from_sas('WUSS');
52             put message=;
53          run;

message=Hello, WUSS
NOTE: DATA statement used (Total process time):
      real time              0.02 seconds
      cpu time               0.03 seconds


54
55          %put %sysfunc(hello_from_sas(WUSS));
Hello, WUSS
```

Figure 1. The log generated by a hello-world SAS function example.

Because SAS functions use a RETURN statement, they can only return a single value. If we wish to instead return multiple values, we can create a call routine using the SUBROUTINE and ENDSUB statements in place of FUNCTION and ENDFUNC:

```
* SAS Code producing the output shown in Figure 2;
proc fcmp outlib=work.funcs.sas; ❶
    subroutine hello_goodbye(name $, greeting $, farewell $); ❷
        outargs greeting, farewell; ❸
        greeting = 'Hello, '||name; ❹
        farewell = name||', Goodbye!'; ❹
    endsub; ❺
run;

data _null_; ❻
    length message1 message2 $ 25; ❼
    call hello_goodbye('WUSS', message1, message2); ❽
    put message1 / message2; ❾
run;
```

As before, we invoke the FCMP procedure ❶, and we specify a package name within a function library previously defined in a CMPLIB option. We then use the SUBROUTINE statement ❷ to list the routine name and parameters, and the OUTARGS statement ❸ is used to specify which parameters can be modified to serve as return values. The call routine body sets return values ❹ before we end the call routine definition with an ENDSUB statement ❺. We then start a DATA steps ❻, where we are careful to specify variable type and length ❼ before using our new call routine ❽ to get values for message1 and message2. Finally, we print our return values on separate lines in the SAS log ❾, as shown in Figure 2.

```
67          proc fcmp outlib=work.funcs.sas;
68              subroutine hello_goodbye(name $, greeting $, farewell $);
69                  outargs greeting, farewell;
70                  greeting = 'Hello, '||name;
71                  farewell = name||', Goodbye!';
72              endsub;
73          run;

NOTE: Function hello_goodbye saved to work.funcs.sas.
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds


74
75          data _null_;
76              length message1 message2 $ 25;
77              call hello_goodbye('WUSS', message1, message2);
78              put message1 / message2;
79          run;

NOTE: Variable message1 is uninitialized.
NOTE: Variable message2 is uninitialized.
Hello, WUSS
WUSS, Goodbye!
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.01 seconds
```

Figure 2. The log generated by a hello-world call routine example.

FCMP PYTHON OBJECTS OVERVIEW

The general pattern for invoking Python from PROC FCMP is as follows:

```
proc fcmp;
    declare object <object-name>(python('<optional-module-name>')); ❶
    submit into <object-name>; ❷
        def <python-function>(<input parameters list>): ❸
            """Output: <output-key>""" ❹
            <Python statements> ❺
            return <return-value> ❻
    endsubmit; ❷
    rc=<object-name>.publish(); ❼
    rc=<object-name>.call('<python-function>',<input parameters list>); ❽
    output = <object-name>.results['<output-key>']; ❾
    <additional SAS statements> ❿
run;
```

We begin by declaring an instance of a Python object with optional module name[9]. We then use a SUBMIT block[10] ❷, within which Python syntax is used to define a function ❸, a comment[11] specifying the name of a value to return to SAS ❹, additional code ❺, and an actual value to return ❻. This is followed by a PUBLISH method ❽, which makes our Python function available, and a CALL method ❾, which invoke it and sets `output` to the `<return-value>` create by `<python-function>`. Finally, this return value `output` can be used in additional SAS statements ❿, such as printing to the log.

---

[9] In this paper, we have opted not to include module names in our examples.

[10] To submit Python code from a file instead, use `rc=<object-name>.infile('<path/to/script.py>')`

[11] A triple-quoted statement appearing as the first line in a Python function is called a docstring [8].

```
95         proc fcmp;
96             length message $ 25;
97             declare object py(python);
98
99             submit into py;
100                def hello():
101                    """Output: hello_return_value"""
102                    return 'Hello'
103            endsubmit;
104
105            rc = py.publish();
106            rc = py.call('hello');
107
108            message = py.results['hello_return_value'];
109            file log;
110            put message=;
111        run;

message=Hello
NOTE: PROCEDURE FCMP used (Total process time):
      real time            0.99 seconds
      cpu time             0.07 seconds
```

Figure 3. The log generated by a Python-driven hello-world example.

Here's a concrete example:

```
* SAS Code producing the output shown in Figure 3;
proc fcmp;
    length message $ 25;
    declare object py(python); ❶

    submit into py; ❷
        def hello(): ❸
            """Output: hello_return_value""" ❹
            return 'Hello' ❺
    endsubmit; ❷

    rc = py.publish(); ❼
    rc = py.call('hello'); ❽

    message = py.results['hello_return_value']; ❾
    file log; ❿
    put message=; ❿
run;
```

After a LENGTH statement creates a character variable, we use a DECLARE statement ❶ to define a Python object `py`, and we use a SUBMIT block ❷ to define Python function `hello` ❸ with a return value named `hello_return_value` ❹ but having the actual value `'Hello'` ❺. The contents of the SUBMIT block are executed when the PUBLISH method is called ❼, passing the definition of the Python function `hello` to the Python interpreter. The `hello` function is then called ❽, and the SAS variable `message` is set to its return value ❾. Finally, the value of `message` is printed[12] ❿, as shown in Figure 3.

This method can be used to submit any arbitrary Python code inside PROC FCMP and return the results, as long as the Python code can be structured as a function.

---

[12] A FILE statement is used to explicitly tell PROC FCMP to direct output to the log, instead of whatever its default output destination might already be set to.

USING A SAS FUNCTION AS A WRAPPER FOR PYTHON CODE

To call a Python function from outside of PROC FCMP, we need to go one step further and wrap it in a SAS function, as in this example:

```
* SAS Code producing the output shown in Figure 4;
proc fcmp outlib=work.funcs.python;
    function greetings_from_python() $ 25; ❶
        length message $ 25; ❷
        declare object py(python); ❸

        submit into py; ❸
            def greetings(): ❹
                """Output: greetings_return_value""" ❺
                import random ❻
                greeting = random.choice( ❻
                    ['Hello', "What's up", 'How do you do?'] ❻
                ) ❻
                return greeting ❼
        endsubmit; ❸

        rc = py.publish(); ❽
        rc = py.call('greetings'); ❽
        message = py.results['greetings_return_value']; ❽
        return(message); ❾
    endfunc; ❶
run;

data _null_; ❿
    message = greetings_from_python();
    put message=;
run;
```

As we saw in our very first PROC FCMP hello-world example, a FUNCTION block ❶ is used to define a SAS function named `greetings_from_python` with a character return value as follows:

- A LENGTH statement ❷ is used to define a SAS variable `message` to be used later.

- Drawing from our first Python-driven hello-world example above, we use a combination of a DECLARE statement and a SUBMIT block ❸ to define PROC FCMP Python object `py`.

- Inside of this SUBMIT block, we define a Python function named `greetings` ❹ with a return value named `greetings_return_value` ❺.

- In order to create this return value, we first import the module `random` (from Python's standard library), and we use this module's `choice` method to randomly select from the list of values provided ❻ before returning the resulting random greeting ❼.

- After the SUBMIT block, we use the same pattern as before ❽:

  o The PUBLISH method passes the Python function `greetings` to the Python interpreter.

  o The CALL method calls the `greetings` function.

  o The results of calling the `greetings` function are stored in the SAS variable `message`.

- Finally, a RETURN ❾ statement is used to set the return value of `greetings_from_python`.

Once it's been defined in the PROC FCMP step, our brand-new SAS function is ready to be used in a DATA step ❿, as show in Figure 4.

```
145        data _null_;
146            message = greetings_from_python();
147            put message=;
148        run;

message=How do you do?
NOTE: DATA statement used (Total process time):
      real time             0.99 seconds
      cpu time              0.09 seconds
```

Figure 4. The log generated by a Python-driven hello-world function example.

USING A CALL ROUTINE TO RETURN MULTIPLE VALUES FROM PYTHON

Up until this point, each of our Python functions has only returned a single value, which has allowed us to avoid thinking too closely about their docstrings (meaning the triple-quoted strings appearing as the first line in a function's definition). To have a Python function return multiple values, we can extend our basic template as follows:

```
proc fcmp;

    declare object <object-name>(python('<optional-module-name>'));

    submit into <object-name>;
        def <python-function>(<input parameters list>):
            """Output: <key-1>, <key-2>, ..., <key-n>""" ❶
            <Python statements>
            return <value-1>, <value-2>, ..., <value-n> ❷
    endsubmit;

    rc=<object-name>.publish();

    rc=<object-name>.call('<python-function>',<input parameters list>);

    <output-1> = <object-name>.results['<key-1>']; /* = <value-1> */ ❸
    <output-2> = <object-name>.results['<key-2>']; /* = <value-2> */ ❹
    ...
    <output-n> = <object-name>.results['<key-n>']; /* = <value-n> */ ❺

    <additional SAS statements> ❻
run;
```

In this template, the docstring ❶ essentially says that we are defining a list of character or numeric names for output values called <key-1>, <key-2>, ..., <key-n>, where n is a positive integer not exceeding six. In addition, we specify the actual output values in the Python return statement ❷ as the tuple[13] <value-1>, <value-2>, ..., <value-n>, where each output value can be a character value, a numeric value, an array (aka a list in Python), or a hash object (aka a dictionary in Python).

In other words, the list of names of output values needs to be the same length as the tuple of return values because we are implicitly defining key-value pairs. We can then index <object-name>.results with each key and store the corresponding value in a SAS variable ❸-❺, and we can use these SAS variables in additional SAS statements ❻.

---

[13] A Python tuple is essentially a fixed-length list [43]. Tuples can be defined as a comma-separated list of values in parentheses; e.g., here's a tuple of length four: (1,2,3,4). However, the parentheses are optional, meaning the same tuple of length four could also be defined as follows: 1,2,3,4

```
186        data _null_;
187            length greeting name $ 25;
188            call personal_greetings_from_python(greeting, name);
189            put greeting name;
190        run;

NOTE: Variable greeting is uninitialized.
NOTE: Variable name is uninitialized.
What's up, Maurice Frey
NOTE: DATA statement used (Total process time):
      real time           1.38 seconds
      cpu time            0.10 seconds
```

Figure 5. The log generated by a Python-driven hello-world call routine example.

To access multiple return values outside of PROC FCMP, we can wrap our Python function definition in a call routine, as in this example:

```
* SAS Code producing the output shown in Figure 5;
proc fcmp outlib=work.funcs.python;
    subroutine personal_greetings_from_python(greeting $, name $); ❶
        outargs greeting, name; ❷
        declare object py(python); ❸

        submit into py; ❸
            def personal_greetings():
                """Output: greeting_return_value, name_return_value""" ❹
                import random
                from faker import Faker
                greeting = random.choice(
                    ['Hello,', "What's up,", 'How do you do,']
                )
                fake = Faker()
                name = fake.name()
                return greeting, name ❺
        endsubmit; ❸

        rc = py.publish(); ❻
        rc = py.call('personal_greetings'); ❻
        greeting = py.results['greeting_return_value']; ❼
        name = py.results['name_return_value']; ❽
    endsub; ❶
run;

data _null_; ❾
    length greeting name $ 25;
    call personal_greetings_from_python(greeting, name);
    put greeting name;
run;
```

A SUBROUTINE block ❶ is used to define call routine `personal_greetings_from_python`, with OUTARGS statement ❷ specifying which parameters can be modified to serve as return values. We then define Python object `py` using a DECLARE statement and a SUBMIT block ❸. Inside this SUBMIT block, Python function `personal_greetings` is defined with docstring ❹ specifying output values named `greeting_return_value` and `name_return_value`. These output values are built using a combination of random values obtained from standard-library module `random` and third-party module `faker` [12] before being returned ❺. After publishing and calling our Python function ❻, we place the return values in SAS variables ❼-❽. Finally, we use our call routine ❾ to get the output in Figure 5.

Obs	message
1	Hello Alfred!
2	Hello Alice!
3	Hello Barbara!

Figure 6. The first few rows of `work.greetings`, which was generated by applying `data_driven_hello_from_python` to the column `name` in `sashelp.class`.

PARAMETERIZING PYTHON CODE FROM SAS

So far, we've executed static Python code and received the results in SAS. To execute a Python function in a data-driven fashion, we can pass parameters from the outer SAS function to the inner Python function. This allows us to use a SAS dataset to drive Python processing, as in the following example:

```
* SAS Code producing the output shown in Figure 6;
proc fcmp outlib=work.funcs.python;
    function data_driven_hello_from_python(name $) $ 25; ❶
        length Message $ 25; ❷
        declare object py(python); ❸

        submit into py; ❸
            def data_driven_hello(name):
                """Output: hello_return_value""" ❹
                return f'Hello {name}!' ❺
        endsubmit; ❸

        rc = py.publish(); ❻
        rc = py.call('data_driven_hello', name); ❻
        Message = py.results['hello_return_value']; ❻
        return(Message); ❼
    endfunc; ❶
run;

data greetings; ❽
    set sashelp.class; ❾
    message = data_driven_hello_from_python(name); ❾
run;

proc print data=greetings; ❿
    var message;
run;
```

A FUNCTION block ❶ is used to define function `data_driven_hello_from_python` with character parameter `name` and a character return value, and a LENGTH statement ❷ defines a SAS variable `Message` to be used later. We then define Python object `py` using a DECLARE statement and a SUBMIT block ❸. Inside this SUBMIT block, Python function `data_driven_hello` is defined with a single input parameter `name`, a single output value named `hello_return_value` ❹, and a `return` statement building this output value[14] ❺ based on the value of the input parameter `name`. After publishing, calling, and retrieving the return value of our Python function ❻, we can set the return value for the outer SAS function ❼. When called in a DATA step ❽, we can apply our new SAS function to each value in a column ❾, just like any other SAS function, with the results of this process ❿ shown in Figure 6.

---

[14] Used to format string values, an f-string will insert the value of an object in curly braces {} into a string [11].

	A	B	C	D
1	2021-22 Private School Affidavit Data - Schools with Enrollment of Six or More Students			
2	California Department of Education, Educational Data Management Division, CALPADS/CBEDS/CDS/PRIVATE SCHOOLS Data Support (07/01/2022)			
3	County	Public School District	CDS Code	School Name
4	Alameda	Alameda Unified	01611196141683	The Child Unique Montessori School/Montessori Elementary Sch
5	Alameda	Alameda Unified	01611196147607	Da Vinci Center for Gifted Children
6	Alameda	Alameda Unified	01611196154637	Advance College Academy

Figure 7. The first few rows of the file `privateschools2122.xlsx`.

PRACTICAL EXAMPLES OF EMBEDDING PYTHON IN SAS PROGRAMS

The first three example use cases below are based on a publicly available dataset from the California Department of Education[15]. We can import this dataset (and apply some light data cleaning to mailing addresses and the unique id column CDS Code) as follows:

```
filename outxlsx 'C:\WUSS22\privateschools2122.xlsx';
proc http
    url='https://www.cde.ca.gov/ds/si/ps/documents/privateschools2122.xlsx'
    method='get'
    out=outxlsx
;
run; quit;

libname schools xlsx 'C:\WUSS22\privateschools2122.xlsx';
data schools_sds(rename=(cds_code_char=cds_code));
    length cds_code_char $ 14;
    set schools.'2021-22 Private School Data$A3:0'n;
    full_address = catx(', ', street, city, state, zip);
    cds_code_char = put(cds_code, z14.);
    drop cds_code;
run;
libname schools clear;
```

To get a sense of the type of information in the file `privateschools2122.xlsx`, see Figure 7.

EXAMPLE USE CASE 1: VALIDATING EMAIL ADDRESSES

The `email_validator` package (AKA *email-validator* on PyPI [44]) provides a convenient way to check email address syntax and deliverability. It might be possible to code an email syntax checker in SAS using regular expressions and/or SAS character functions, but the RFC standards are surprisingly nuanced, per the summary in [49]. Since a Python package already exists, why not take advantage of it?

This illustrates one of the main benefits of working in Python: Because the language has a large userbase embracing its open-source ethos, it's common for developers to "give back" by publishing new packages on sites like GitHub (https://github.com/) and the Python Package Index (https://pypi.org/).

Often, these packages attempt to abstract away details so that a single function call can handle something as complex as checking email deliverability, as illustrated in this example:

```
* SAS Code producing the output shown in Figure 8;
proc fcmp outlib=work.funcs.python;
    function get_normalized_email(email $) $ 100; ❶
        length normalized_email $ 100 Exception_Encountered $ 500; ❷
        declare object py(python); ❸
        submit into py; ❸
```

---

[15] This file contains contact information for all private K–12 schools in the state of California during academic year 2021-2022. For more information, https://www.cde.ca.gov/ds/si/ps/

```
                def normalize_email(e): ❹
                    """Output: normalize_email_return_value, exception""" ❹
                    from email_validator import (
                        validate_email, EmailNotValidError
                    )
                    try: ❺
                        normalized_email = validate_email(
                            e, check_deliverability=False
                        )
                        return normalized_email.email, ' ' ❻
                    except EmailNotValidError: ❺
                        return ' ', repr(e) ❻
            endsubmit; ❸

            rc = py.publish(); ❼
            rc = py.call('normalize_email', email); ❼

            Exception_Encountered = py.results['exception']; ❽
            if not missing(Exception_Encountered) then
                put Exception_Encountered=;

            normalized_email = py.results['normalize_email_return_value']; ❽
            return(normalized_email); ❾
        endfunc; ❶
    run;

    data normalized_emails; ❿
        set schools_sds;
        normalized_email = get_normalized_email(primary_email);
    run;

    proc print data=normalized_emails; ❿
        var primary_email normalized_email;
        where primary_email NE normalized_email;
    run;
```

A FUNCTION block ❶ defines the function `get_normalized_email` with character parameter `email` and a character return value, and a LENGTH statement ❷ defines two SAS variables `normalized_email` and `Exception_Encountered` to be used later. We then define Python object `py` using a DECLARE statement and a SUBMIT block ❸. Inside this SUBMIT block, Python function `normalize_email` is defined with a single input parameter `e` (representing an email address) and two output values named `normalize_email_return_value` and `exception` ❹.

To build these output values, we use the `validate_email` function from the `email_validator` package inside of a `try`/`except` structure[16] ❺. In other words, we expect one of two things to happen: If `e` is a valid email address, then `validate_email(e)` will return a normalized version of the address. Otherwise, if `e` is not a valid email address, then `validate_email(e)` will raise an exception, meaning an attempt to stop code execution[17]. Depending on the scenario, one of the two `return` statements ❻ will be used to set the values returned by `normalize_email`.

---

[16] The code in the `try:` block is executed first. If no exceptions are raised, the `except:` block is ignored. Otherwise, if an exception is raised in the `try:` block, program execution is returned to the state it had before the `try:` block, and the code in the `except:` block is executed instead. See [26] for more information.

[17] This is a fairly common pattern in Python: Functions raising exceptions as a means of communicating with the caller, and they expect the caller to be ready to handle the exception appropriately.

Obs	Primary_Email	normalized_email
13	office@AIMmontessori.com	office@aimmontessori.com
83	Admin@LAMBO.School	Admin@lambo.school
88	JohnMuster@MentoringAcademy.org	JohnMuster@mentoringacademy.org
144	UNIBIBLICAM@GMAIL.COM	UNIBIBLICAM@gmail.com
230	info@CrestmontSchool.org	info@crestmontschool.org
357	SCBCFAM10@GMAIL.COM	SCBCFAM10@gmail.com
413	admissions@StanzaAcademy.org	admissions@stanzaacademy.org
438	no data	

Figure 8. The first few rows of `work.normalized_emails`, which was generated by applying `get_normalized_email` to the column `primary_email` in `work.schools_sds`.

Then, after publishing and calling our Python function ❼, we can retrieve its two return values ❽. The `exception` return value of `normalize_email` is put into the SAS variable `Exception_Encountered`, giving us information about whether an exception was raised by the `validate_email` function. If so, we write the invalid value to the SAS log.

The other return value of `normalize_email` is put into the SAS variable `normalized_email`. The value of `normalized_email` will either be the normalized version of an email address or an empty string, depending on whether an exception was raised by the `validate_email` function. Either way, we use `normalized_email` as the return value for `get_normalized_email` ❾.

The results of using `get_normalized_email` ❿ can be seen in Figure 8.

EXAMPLE USE CASE 2: GEOCODING

Geocoding a mailing address (i.e., mapping it to a set of physical coordinates like latitude and longitude) is often a prerequisite for any kind of geographic analysis. While SAS provides the GEOCODE procedure, this PROC requires the user to have access to an extensive collection of well-maintained, local lookup tables [32].

In order to get instant, up-to-date geocoding information, Python programmers tend to prefer querying a web API using a package like `geocoder` [3], which provides a straightforward interface for popular geocoding services like MapQuest[18]. We illustrate this in the following example, where we use a call routine in order to return separate values for latitude and longitude:

```
* SAS Code producing the output shown in Figure 9;
%let MAPQUEST_API_KEY = '[PUT YOUR MAPQUEST API KEY HERE]';

proc fcmp outlib=work.funcs.python;
    subroutine get_lat_long(address $, key $, lat, long); ❶
        outargs lat, long; ❷
        declare object py(python); ❸

        submit into py; ❸
            def geocode(a, k):
                """Output: latitude_return_value, longitude_return_value"""
                import geocoder
                g = geocoder.mapquest(a, key = k) ❹
                lat = g.latlng[0] ❹
                long = g.latlng[1] ❹
```

[18] We chose MapQuest because an API Key can quickly be obtained by completing a web form [16].

14

Obs	full_address	lat	long
1	2226 Encinal Avenue And 1400 6Th Street, Alameda, CA, 94501	37.7717	-122.264
2	2050 Lincoln Avenue, Alameda, CA, 94501	37.7701	-122.249
3	2710 Encinal Avenue, Alameda, CA, 94501	37.7594	-122.239

Figure 9. The first few rows of `work.lat_lng`, which was generated by applying `get_lat_long` to the column `full_address` in `work.schools_sds`.

```
                return lat, long  ❺
        endsubmit;  ❸

        rc = py.publish();  ❻
        rc = py.call('geocode', address, key);  ❻

        lat = py.results['latitude_return_value'];  ❼
        long = py.results['longitude_return_value'];  ❽
    endsub;
run;

data lat_lng;  ❾
    set schools_sds(obs=10);
    call get_lat_long(full_address, &MAPQUEST_API_KEY, lat, long);
run;

proc print data=lat_lng;  ❿
    var full_address lat long;
run;
```

A SUBROUTINE block ❶ is used to define call routine `get_lat_long`, with OUTARGS statement ❷ specifying which parameters can be modified to serve as return values. We then define Python object `py` using a DECLARE statement and a SUBMIT block ❸. Inside this SUBMIT block, Python function `geocode` is defined with two input parameters. The first input parameter `a` represents a full street address, and the second input parameter `k` is an API key. In addition, `geocode` has a docstring specifying output values named `latitude_return_value` and `longitude_return_value`.

In order to build these output values, we use the method `geocoder.mapquest`, supplying both the address `a` and the MapQuest API key `k` ❹. The resulting object `g` has an attribute called `latlng`, which is a list that `lat` and `long` can be extracted from[19]. We can then use `lat` and `long` as the return values of the Python function `geocode` ❺.

After publishing and calling our Python function ❻, we can place its return values in SAS variables ❼-❽. Finally, we use our call routine ❾ and print the results ❿ to get the output in Figure 9.

EXAMPLE USE CASE 3: CREATING FORMATTED EXCEL FILES FROM SAS DATASETS

Because they're an excellent tool for data exploration and reporting, SAS programmers regularly export SAS datasets and reports as Excel workbook files. While it's possible to create custom Excel files in SAS using ODS [22], various Python packages provide extensive options with a potentially more intuitive interface. This example demonstrates how to use the popular Python package `XlsxWriter` [17] to transform an entire SAS dataset into a highly formatted Excel file:

---

[19] In Python, lists are indexed starting with zero, and elements can be accessed using square-bracket notation [43]. In other words, `g.latlng[0]` is the initial element in the list, and `g.latlng[1]` is the subsequent element.

```
* SAS Code producing the output shown in Figure 10;
proc fcmp;
    length libpath path outfile $ 500; ❶
    libpath = pathname('work'); ❶
    path = catx('\',libpath,'schools_sds.sas7bdat'); ❶
    file log; ❶

    declare object py(python); ❷
    submit into py; ❷
    def format_excel(datasetpath): ❸
        """Output: output_file""" ❹
        import pandas
        import pathlib
        import xlsxwriter

        # read a SAS dataset
        schools_df = pandas.read_sas(datasetpath, encoding='latin1') ❺

        # output an Excel file
        file_path = pathlib.Path('C:\WUSS22\excel_output') ❻
        file_name = 'example_excel_export.xlsx' ❻
        sheet_name = 'Augmented CDE Data' ❻

        # setup Excel file writer
        with pandas.ExcelWriter( ❼
            pathlib.Path(file_path, file_name), engine='xlsxwriter'
        ) as writer:
            schools_df.to_excel(
                writer,
                sheet_name=sheet_name,
                index=False,
                startrow=1,
                header=False,
            )
            max_column_index = schools_df.shape[1] - 1


            # setup formatting to be applied below
            workbook = writer.book
            text_format = workbook.add_format({'num_format': '@'})
            header_format = workbook.add_format({
                'bold': True,
                'text_wrap': True,
                'valign': 'center',
                'num_format': '@',
                'fg_color': '#FFE552',  # Light Gold
                'border': 1,
            })

            # write header row values with formatting
            worksheet = writer.sheets[sheet_name]
            for col_num, value in enumerate(schools_df.columns.values):
                worksheet.write(0, col_num, value, header_format)

            # use fixed column width and use a universal text format
            worksheet.set_column(0, max_column_index, 20, text_format)
```

Figure 10. The first few rows of the Excel workbook file generated by applying the Python package `XlsxWriter` to `work.schools_sds`.

```
            # turn on filtering for top row
            worksheet.autofilter(
                0, 0, schools_df.shape[0], max_column_index
            )

            # turn on freeze panes for top row
            worksheet.freeze_panes(1, 0)
        return str(pathlib.Path(file_path, file_name)) ❽
    endsubmit; ❷

    rc = py.publish(); ❾
    rc = py.call('format_excel', path); ❾
    outfile = py.results['output_file']; ❾
    put 'Output file: ' outfile; ❿
run;
```

Unlike the previous two examples, we have embedded our Python code inside a PROC FCMP step without an outer SAS function or call routine. Instead, we use several setup steps ❶ to create some character variables, get the path for the SAS dataset `work.schools_sds`, and set the output destination to the SAS log (instead of whatever the default output destination might have already been set to).

We then define Python object `py` using a DECLARE statement and a SUBMIT block ❷. Inside this SUBMIT block, Python function `format_excel` is defined with a single input parameter representing the path on disk to a SAS dataset ❸. In addition, `format_excel` has a docstring ❹ specifying a single output value named `output_file`, which will be the path on disk to an Excel workbook file.

In order to begin the process of creating this Excel file, we use the `read_sas` method of the `pandas` package ❺, including specifying a character encoding value[20]. We then set the path, filename, and sheet name for the Excel file ❻ before using a collection of features from the `XlsxWriter` package to write values to the worksheet `Augmented CDE Data` in the file `example_excel_export.xlsx` inside of a context manager[21]. In addition, at the start of the context manager, we use the `pathlib` package to combine the file path and filename into a single path on disk to the Excel file we intend to create.

Once we've finished building the Excel file inside the context manager, we use its path on disk as the return value for `format_excel` ❽. At this point, we can publish and call our Python function ❾ before printing the path on disk to the SAS log ❿. A few rows of the resulting Excel file are shown in Figure 10.

---

[20] It's necessary to specify the encoding when reading SAS character variables into Python from a single-byte character encoding like `'latin1'`. For maximum compatibility with Python, the use of a variable-byte character encoding like `'utf-8'` may be a good idea.

[21] The `with:` statement is the start of the context manager, and everything indented underneath the `with:` statement is the body of the context manager. At the start of the context manager, the specified file is automatically opened, and this file is then automatically closed as soon as we exit the context manager. Because this saves us from needing to manually keep track of when files are opened and closed, context managers tend to be the safest way of creating files in Python. See [27] for more information.

EXAMPLE USE CASE 4: TURNING A YAML FILE INTO A SAS DATASET

YAML bills itself as "a human-friendly data serialization language" [50]. Similar to JSON [6], files written in YAML can be used to describe complex, hierarchical structures with arbitrarily nested lists of values (aka *lists*) and key-value mappings (aka *dictionaries*). Here's an example:

```
human:
   first-name: Person1
   last-name: LastName
   age: 42
   hobbies:
      - Writing SAS Code
      - Embedding Python Code inside PROC FCMP
      - Learning YAML:
         - Reading the Spec
         - Converting YAML to JSON
```

In YAML, colons are used to separate keys and their corresponding values in dictionaries, and hyphens are used to denote elements in a list. To see how much is going on in this compact structure, it's helpful to look at the corresponding JSON document[22]:

```
{
   "human": {
      "first-name": "Person1",
      "last-name": "LastName",
      "age": 42,
      "hobbies": [
         "Writing SAS Code",
         "Embedding Python Code inside PROC FCMP",
         {
            "Learning YAML": [
               "Reading the Spec",
               "Converting YAML to JSON"
            ]
         }
      ]
   }
}
```

When viewing JSON, the three most important things to remember are that curly braces {} are used to denote dictionaries, colons are used to separate keys from their corresponding values in dictionaries (just like in YAML), and square brackets [] are used to denote lists. (Conveniently, these also happen to be the exact same conventions for denoting lists and dictionaries in Python [25].)

SAS provides built-in engines for reading many common file types using PROC IMPORT [35] and LIBNAME statements [34], and INPUT statements in a DATA step can be used to read complex, non-standard file types [33]. However, as powerful of a data-processing tool as SAS may be, it's typically best suited for reading non-hierarchical data. While it's possible to read JSON to a limited degree [10], the authors are not aware of any tools for parsing YAML with SAS, other than complex DATA step programming using INPUT statements.

Python, on the other hand, has excellent tools for reading and writing both hierarchical and non-hierarchical data. To illustrate this, the following example rectangularizes a YAML file[23] by first converting it to JSON and then "normalizing" the JSON[24] before saving the result as a SAS dataset:

---

[22] Generated using https://onlineyamltools.com/convert-yaml-to-json
[23] This file contains a list of all U.S. members of congress, presidents, and vice presidents. For more information, see https://github.com/unitedstates/congress-legislators
[24] Normalizing JSON involves flattening nested structures into columns. See [21] for more information.

```sas
* SAS Code producing the output shown in Figure 11;
proc fcmp;
    length workpath outfile $ 500; ❶
    workpath = pathname('work'); ❶
    file log; ❶

    declare object py(python); ❷
    submit into py; ❷
        def import_yaml_to_sas(workpath):
            """Output: output_table"""
            import yaml
            import requests
            from pandas import json_normalize
            import sys
            setattr(sys.stdin, 'isatty', lambda: False) ❸
            from saspy import SASsession

            url = 'https://raw.githubusercontent.com/unitedstates/congress-
legislators/main/legislators-current.yaml'
            request_response = requests.get(url) ❹
            if request_response.status_code != 200: ❹
                return ' ' ❹

            legislators_list = yaml.safe_load(request_response.text) ❺
            legislators_df = json_normalize(legislators_list) ❺
            legislator_info = legislators_df[[
                'id.cspan',
                'name.first',
                'name.middle',
                'name.last',
                'bio.birthday',
                'bio.gender'
            ]] ❺

            sas = SASsession() ❻
            sas.saslib(libref='out',path=workpath) ❻
            outds = sas.dataframe2sasdata(
                df=sas.validvarname(legislator_info),
                libref='out',
                table='legislators',
                encode_errors='replace'
            ) ❻
            sas.endsas() ❻
            return outds.table ❻
    endsubmit; ❷

    rc = py.publish(); ❽
    rc = py.call('import_yaml_to_sas', workpath); ❽
    outfile = py.results['output_table']; ❾
    if not missing(outfile) then ❾
        put 'Output dataset:' outfile; ❾
run;

proc print data=legislators(obs=10); ❿
run;
```

Obs	id_cspan	name_first	name_middle	name_last	bio_birthday	bio_gender
1	5051	Sherrod		Brown	1952-11-09	M
2	26137	Maria		Cantwell	1958-10-13	F
3	4004	Benjamin	L.	Cardin	1943-10-05	M

Figure 11. The first few rows of `work.legislators`, which was generated by rectangularizing a YAML file in Python and exporting the results as a SAS dataset using the Python package SASPy.

As with the previous example, we've embedded our Python code inside a PROC FCMP step without an outer SAS function or call routine, and we use some setup steps ❶ to create character variables `workpath` and `outfile`. We set `workpath` to the path on disk of the work library[25], and we will be putting the path on disk to our output dataset into `outfile`. We also once again set the output destination to the SAS log (instead of whatever the default output destination might already be set to).

We then define Python object `py` using a DECLARE statement and a SUBMIT block ❷. Inside this SUBMIT block, Python function `import_yaml_to_sas` is defined with a single input parameter representing the path on disk to the work library. In addition, `import_yaml_to_sas` has a docstring specifying a single output value named `output_table`, which will be the path on disk to the SAS dataset we are going to create. We also import several packages at the start of `import_yaml_to_sas`, and we use the standard-library package `sys` to make it possible to use the third-party package `saspy` inside of PROC FCMP[26] ❸.

With this setup, we're now ready to download our YAML file ❹. We are also careful to check the HTTP status code to make sure the HTTP request was successful[27]. If not, `import_yaml_to_sas` will return an empty string.

Assuming our HTTP request was successful, we transform the YAML file into a JSON document using `yaml.safe_load`, we normalize the JSON into a rectangular structure called a DataFrame using `json_normalize`[28], and we subset this DataFrame to the few columns we're interested in ❺. (DataFrames are defined by the third-party `pandas` package and can be thought of as the Python equivalent of a SAS dataset.)

At this point, all of the data gymnastics are complete, and the only remaining step is to convert our DataFrame into a SAS dataset. To do this, we use the `saspy` package to create a new SAS session[29], meaning an independent connection to the SAS kernel having no shared context with our outer SAS session. Inside this new SAS session, we use Python syntax to issue SAS commands and write our DataFrame to disk as a SAS dataset (setting `encode_errors='replace'` in case there are any encoding issues) before having `import_yaml_to_sas` return the path to this SAS dataset ❻.

We are now able to publish and call our Python function ❽ before printing the path on disk to the log ❾. Also, because we have written our new SAS dataset to disk in the work library, we're able to immediately use the dataset outside of the PROC FCMP step ❿, as shown in Figure 11.

---

[25] If we use a different file path, we could save the generated SAS dataset permanently.

[26] SASPy expects to ask for information from the user interactively, which is why we need to disable user interactivity.

[27] Whenever we visit a URL, whether it's in a web browser or using a Python package like `requests`, we send an HTTP request to a server, and the server returns an HTTP response back to us. The status code of this response indicates whether the request was successful, with 200 meaning "OK" [19].

[28] We could also build a DataFrame manually by iterating over the list with a for loop. For more complex data, this can sometimes be easier than using `pandas` built-in normalization functionality.

[29] The `saspy` module is maintained by SAS institute, and it can be used more generally to embed SAS code inside any Python script, not just inside PROC FCMP [14].

CONCLUSION

FCMP Python objects are an excellent tool for embedding Python code inside of SAS programs. The ability to call Python functions from SAS opens up exciting new opportunities for programmers to combine the two languages, leveraging Python's excellent in-memory analytical capabilities and wealth of open-source packages. Individual programmers benefit from the ability to flexibly choose the best tool for the job, and organizations gain efficiency when programmers from various backgrounds and disciplines can cooperate effectively.

REFERENCES

[1] Anaconda, Inc. "Anaconda Installers." *Anaconda Distribution*. Date accessed: 15AUG2022. Available at https://www.anaconda.com/products/distribution

[2] Anaconda, Inc. "User Guide." *Conda*. Date accessed: 15AUG2022. Available at https://docs.conda.io/projects/conda/en/latest/user-guide/

[3] Carriere, Denis. "geocoder." *Python Package Index*. Date accessed: 15AUG2022. Available at https://pypi.org/project/geocoder/

[4] Django Stars. (2019) "Top Seven Apps Built with Python." *Hacker Noon*. Date accessed: 15AUG2022. Available at https://hackernoon.com/top-seven-apps-built-with-python-2cd8dfd3c00a

[5] Driscoll, Mike. (2018) *Python Interviews: Discussions with Python Experts*. Packt Publishing: Birmingham, U.K.

[6] Ecma International. "Introducing JSON." *ECMA-404 The JSON Data Interchange Standard*. Date accessed: 15AUG2022. Available at https://www.json.org/

[7] Finley, Klint. (2021) "Open Source on Mars: Community powers NASA's Ingenuity Helicopter." *The ReadME Project*. Date accessed: 15AUG2022. Available at https://github.com/readme/featured/nasa-ingenuity-helicopter

[8] Goodger, David; van Rossum, Guido. "PEP 257 – Docstring Conventions." *Python Enhancement Proposals*. Date accessed: 15AUG2022. Available at https://peps.python.org/pep-0257/

[9] Hall, Patrick; Myeni, Radhikha; Zhang, Ruiwen. "Open Source Integration Using the Base SAS Java Object." *SAS Products & Solutions*. Date accessed: 15AUG2022. Available at https://support.sas.com/rnd/app/data-mining/enterprise-miner/pdfs/SAS_Base_OpenSrcIntegration.pdf

[10] Hemedinger, Chris. "Reading data with the SAS JSON libname engine." *SAS Blogs*. Date accessed: 15AUG2022. Available at https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/

[11] Jablonski, Joanna. "Python 3's f-Strings: An Improved String Formatting Syntax (Guide)." *Real Python*. Date accessed: 15AUG2022. Available at https://realpython.com/python-f-strings/

[12] joke2k. "Faker." *Python Package Index*. Date accessed: 15AUG2022. Available at https://pypi.org/project/Faker/

[13] Krill, Paul. "Python popularity still soaring." *Inforworld*. Date accessed: 15AUG2022. Available at https://www.infoworld.com/article/3669232/python-popularity-still-soaring.html

[14] Lankham, Isaiah; Slaughter, Matthew. "Everything is better with friends: Executing SAS code in Python scripts with SASPy." *Proceedings of the Western Users of SAS Software 2019 Conference*, Seattle, WA. Date accessed: 15AUG2022. Available at https://www.lexjansen.com/wuss/2019/142_Final_Paper_PDF.pdf

[15] Lolla, Venu Gopal. "Integrating Python and Base SAS." *Proceedings of the SAS Global Forum 2019 Conference*, Dallas, TX. Date accessed: 15AUG2022. Available at https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3548-2019.pdf

[16] MapQuest, Inc. "Welcome Developers!" *MapQuest Developer*. Date accessed: 15AUG2022. Available at https://developer.mapquest.com/documentation/

[17] McNamara, John. "XlsxWriter." *Python Package Index*. Date accessed: 15AUG2022. Available at https://pypi.org/project/XlsxWriter/

[18] Microsoft Corporation. "setx." *Commands by Server Role*. Date accessed: 15AUG2022. Available at https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/setx

[19] Mozilla Corporation. " HTTP response status codes." *MDN Web Docs*. Date accessed: 15AUG2022. Available at https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[20] Pandas Development Team. "pandas." *Python Package Index.* Date accessed: 15AUG2022. Available at https://pypi.org/project/pandas/

[21] Pandas Development Team. "pandas.json_normalize." *API Reference.* Date accessed: 15AUG2022. Available at https://pandas.pydata.org/docs/reference/api/pandas.json_normalize.html

[22] Parker, Chevell. "Tips for Using the ODS Excel Destination." *SAS Blogs*. Date accessed: 15AUG2022. Available at https://blogs.sas.com/content/sgf/2017/02/20/tips-for-using-the-ods-excel-destination/

[23] Pip developers. "pip." *pip documentation*. Date accessed: 15AUG2022. Available at https://pip.pypa.io/en/stable/

[24] Python Packaging Authority. "Installing Packages." *Python Packaging Users Guide*. Date accessed: 15AUG2022. Available at https://packaging.python.org/en/latest/tutorials/installing-packages/

[25] Python Software Foundation. "Data Structures." *The Python Tutorial*. Date accessed: 15AUG2022. Available at https://docs.python.org/3/tutorial/datastructures.html

[26] Python Software Foundation. "Errors and Exceptions." *The Python Tutorial*. Date accessed: 15AUG2022. Available at https://docs.python.org/3/tutorial/errors.html

[27] Ramos, Leodanis Pozo. "Context Managers and Python's with Statement." *Real Python*. Date accessed: 15AUG2022. Available at https://realpython.com/python-with-statement/

[28] Reitz, Kenneth. "requests." *Python Package Index.* Date accessed: 15AUG2022. Available at https://pypi.org/project/requests/

[29] SAS Institute. "CALL SYSTEM Routine." *Dictionary of Functions and Call Routines*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p089n536m1spv9n1cpuo8u34hw5m.htm

[30] SAS Institute. "Configuring SAS to Run the Python Language." *Configuring SAS to Run External Languages*. Date accessed: 15AUG2022. Available at https://go.documentation.sas.com/doc/en/bicdc/9.4/biasag/n1mquxnfmfu83en1if8icqmx8cdf.htm

[31] SAS Institute. "FCMP Procedure." *Base SAS Procedures Guide*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0pio2crltpr35n1ny010zrfbvc9.htm

[32] SAS Institute. "GEOCODE Procedure." *Dictionary of SAS/GRAPH and Base SAS Mapping Procedures*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/grmapref/n02y3yabtlqatsn16gp2fo51yo7p.htm

[33] SAS Institute. "Input Statement." *DATA Step Statements*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lestmtsref/n0oaql83drile0n141pdacojq97s.htm

[34] SAS Institute. "LIBNAME Statement." *Global Statements*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lestmtsglobal/n1nk65k2vsfmxfn1wu17fntzszbp.htm

[35] SAS Institute. "IMPORT Procedure." *Base SAS Procedures Guide*. Date accessed: 15AUG2022. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n18jyszn33umngn14czw2qfw7thc.htm

[36] SAS Institute. "PYTHON Procedure." *Base SAS Procedures Guide.* Date accessed: 15AUG2022. Available at
https://go.documentation.sas.com/doc/en/pgmsascdc/v_009/proc/p1iycdzbxw2787n178ysea5ghk6l.htm

[37] SAS Institute. "SET System Option." *SAS System Options.* Date accessed: 15AUG2022. Available at
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lesysoptsref/n1evyjd7jdkqxwn1w2d50ngwg386.htm

[38] SAS Institute. "Starting from Custom Shortcuts or Program Items." *Getting Started under Windows.* Date accessed: 15AUG2022: Available at
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/hostwin/p16esisc4nrd5sn1ps5l6u8f79k6.htm#n0y3djnit3wssfn1j6qfvz0fvsjr

[39] SAS Institute. "SYSTASK Statement: Windows." *SAS Statements under Windows.* Date accessed: 15AUG2022. Available at
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/hostwin/p09xs5cudl2lfin17t5aqvpcxkuz.htm

[40] SAS Institute. "Using PROC FCMP Python Objects." *SAS Component Objects.* Date accessed: 15AUG2022. Available at
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lecompobjref/p18qp136f91aaqn1h54v3b6pkant.htm

[41] SAS Institute. "X Statement." *Dictionary of SAS Global Statements.* Date accessed: 15AUG2022. Available at
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lestmtsglobal/p11ba12uypvfazn1jk7acffuzlbl.htm

[42] Simonov, Kirill. "PyYAML." *Python Package Index.* Date accessed: 15AUG2022. Available at
https://pypi.org/project/PyYAML/

[43] Sturz, John. "Lists and Tuples in Python." *Real Python.* Date accessed: 15AUG2022. Available at
https://realpython.com/python-lists-tuples/

[44] Tauberer, Joshua. "email-validator." *Python Package Index.* Date accessed: 15AUG2022. Available at https://pypi.org/project/email-validator/

[45] Van Rossum, Guido. (1996) "Foreword." *Programming Python, 1st Edition.* O'Reilly Media: Sebastopol, CA.

[46] Van Rossum, Guido. (1998) "Glue It All Together with Python." *Position paper for the OMG-DARPA-MCC Workshop on Compositional Software Architectur*e. Date accessed: 15AUG2022. Available at
https://www.python.org/doc/essays/omg-darpa-mcc-position/

[47] Weber, Tom. "saspy." *Python Package Index.* Date accessed: 15AUG2022. Available at
https://pypi.org/project/saspy/

[48] Whitcher, Michael; Mays, Aaron; McNeill, Bill; Henrick, Andrew; Christian, Stacey. "What's New in FCMP for SAS 9.4 and SAS Viya." *Proceedings of the SAS Global Forum 2019 Conference,* Dallas, TX. Date accessed: 15AUG2022. Available at https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3480-2019.pdf

[49] Wikipedia. "Syntax." *Email address.* Date accessed: 15AUG2022. Available at
https://en.wikipedia.org/wiki/Email_address#Syntax

[50] YAML Language Development Team. "YAML Resources." *YAML: YAML Ain't Markup Language.* Date accessed: 15AUG2022. Available at https://yaml.org/

[51] Zizzi, Mike. "SAS or Python? Why not use both? Using Python functions inside SAS programs." *SAS Blogs.* Date accessed: 15AUG2022. Available at https://blogs.sas.com/content/sgf/2019/06/04/using-python-functions-inside-sas-programs/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please contact the authors as follows:

Isaiah Lankham
Senior Research Analyst
U. of California Office of the President
1111 Franklin Street
Oakland, CA 94607
Phone: +1-510-987-9776
E-mail: Isaiah.Lankham@ucop.edu

Matthew T. Slaughter
Statistical Research Analyst
Kaiser Permanente Center for Health Research
3800 N Interstate Avenue
Portland, OR 97227
Phone: +1-503-335-6319
E-mail: Matthew.T.Slaughter@kpchr.org