SAS3480-2019

What's New in FCMP for SAS 9.4 and SAS Viya

Michael Whitcher, Aaron Mays, Bill McNeill, Andrew Henrick, and Stacey Christian,

SAS Institute Inc.

ABSTRACT

Join us as we explore new features and functionality in the SAS® function compiler (FCMP). Integration with Python, support for running analytic scoring containers (ASTORE), and a new FCMP action set are the main topics we will cover. Learn how to leverage your existing investment in Python by calling Python functions from an FCMP function. Get the most from your ASTORE by porting it from SAS® Viya® to SAS® 9.4 (TS1M6), and then run it from an FCMP function called from the SAS DATA Step. Learn how to port your favorite user-written functions and subroutines to SAS® Cloud Analytic Services, and then use them within a computed column or another action. This paper will show you the tips and tricks you need to integrate your existing FCMP code with these new SAS technologies. Included are several examples to quickly get you started.

INTRODUCTION

One of the biggest challenges for those who implement computer languages is keeping the language rich with features and relevant to today's programming community. There are many aspects to keeping any computer language relevant: an easy to use syntax, a rich run time environment, integration with external systems, and performance. The pages to follow will describe in detail three new features of FCMP that provide seamless integration between different systems and languages using an easy and natural syntax. But before getting into the details, let's clarify how each new feature makes SAS relevant in today's ever changing computer world.

With the growing acceptance of open-source software, many companies are investing resources in new languages. What happens when you suddenly have a major investment in functions or model libraries in another language, like Python; but you already have a major investment in SAS? We believe that you should be able to use these systems together in an intuitive and seamless fashion. This paper highlights a new FCMP Python object that integrates Python functions with PROC FCMP and DATA Step code; allowing you to run your Python code seamlessly in SAS.

Second, as many users start to take advantage of new techniques in SAS Viya (such as artificial intelligence and machine learning models for scoring), they find themselves wanting to bring those models back into their 'tried and true' V9.4 production processes. SAS Viya introduced the concept of an analytical store (ASTORE), which containerizes the state from a predictive model so that it can be easily transported to any host platform. Why should this be limited to the SAS Viya system? Why not use the ASTORE you created in SAS Viya within your V9.4 programs as well? In this paper we show how to move an ASTORE created in SAS Viya to SAS V9.4 (TS1M6) and use it from within PROC FCMP and the DATA Step.

Similarly, wouldn't it be nice if there were a simple way to load the FCMP function libraries you developed in V9.4 into Cloud Analytic Services (CAS) and run them there? In the final section, this paper introduces the new FCMP action set that allows you to load your SAS

V9.4 FCMP functions and subroutines into SAS Viya thus enabling you to use the same set of programs in both run-time environments.

PROC FCMP INTEGRATION WITH PYTHON

The Python language was first conceived in the late 1980s, about the time when SAS was shipping its version 6 series of releases. Both language systems have gone through much evolution and expansion, and now it is possible for code written in PROC FCMP to call functions written in Python. And since FCMP functions and subroutines can be called from the DATA step, the DATA Step can indirectly call Python functions using FCMP starting in SAS v9.4 (TS1M6).

The FCMP procedure supports Python version 2.7 and above, so users can integrate with the version of their choice. Python version 3.6 and above is recommended. The examples in this paper were written using Anaconda for Python version 3.7.1 for 64-bit Windows.

SETUP

SAS® Micro Analytic Services (MAS) provides the Python support that PROC FCMP leverages to integrate Python. See the documentation, entitled *SAS® Micro Analytic Service 2.5M1: Programming and Administration Guide*, for installation and configuration information. Currently, the integration is supported on both 64-bit UNIX and 64-bit Windows by setting two environment variables prior to launching the SAS system and running PROC FCMP.

The first environment variable, called *MAS_M2PATH*, tells MAS where to find the *mas2py.py* communication program. This program comes with your installation of MAS and is the program that is started when MAS launches the Python interpreter to service function calls.

The second environment variable, called *MAS_PYPATH*, tells MAS which Python interpreter on your computer you would like to use. Since a SAS user might have multiple Python interpreters installed on their computer, use *MAS_PYPATH* to tell MAS and PROC FCMP which Python interpreter should be invoked. Putting these two variables together, here is the setup.

For UNIX:

```
export MAS_M2PATH=/your/sas/installation/path/mas2py.py
export MAS_PYPATH=/your/python-dir/python
```

For Windows:

```
set MAS_M2PATH=c:\your\sas\installation\path\mas2py.py
set MAS_PYPATH=c:\your\python-dir\python.exe
```

The Python interpreter is launched using the same credentials as the authenticated SAS user. Users can import and call any of the Python modules or libraries, which come with their Python system. SAS numeric and character data types can be passed as input arguments. And certain Python data types can be returned through the functions. Let's now look at an example in greater detail.

DECLARING A PYTHON OBJECT

A PROC FCMP based Python object is an in-memory container, which allows you to write and store Python function source code, publish the function source code to a Python interpreter, and then call the Python functions defined within the published Python module. The syntax of the declare statement for a Python object follows the generalized form for all FCMP objects.

General Declaration 1:

```
declare object object-name(python<("module-name")>);
```

Where the *object-name is the FCMP symbol name, and "module-name"* is an optional name for the Python module once it is published. Every Python module must have a name, so if the *"module-name"* is omitted, the FCMP symbol name is used as the default Python module name. The Python module name must be enclosed in quotation marks and enclosed in parentheses.

Example 1: FCMP Python Object with Module Name

```
declare object optionPricing(python("BlackScholes");
```

In Example 1 an FCMP Python object named *optionPricing* is created. The object's Python module name is "BlackScholes". Yet when a function in this module is called, the module name is not listed in the call. It is derived from the object declaration. As you'll see, the Python module name will be prepended to the function name when the function call is made to the Python interpreter. The notion of modules comes from the Python language itself. Functions are contained within a module, and in general Python functions are called as *module.function(args)*. So, the FCMP declaration for Python objects takes this into consideration by allowing you to specify the module name.

Example 2: FCMP Python Object Using a Default Module Name

```
declare object py(python);
```

In this declaration an FCMP Python object named, *py*, is created. No module name is specified, so the symbol name, *py*, is also used as the module name.

Multiple Python objects can be declared within a single PROC FCMP step, and once published, Python functions within one module can call Python functions in a second module.

WRITING PYTHON FUNCTIONS

Once the FCMP Python object has been declared, the next step is to write the Python source code that goes into it. The Python source code can be placed into the FCMP Python object by either reading the code from a file, using the new FCMP *submit into* statement, or using the APPEND method. Let's look at all three techniques.

Load Python Source Code from a File

The FCMP Python object has two methods available for reading Python source code from a file and storing it into the object. Once stored in the object, the source code can then be published.

The first is the INFILE method. This method takes a single argument string literal containing the file path location to your Python source code. The method returns a numeric return code. A return value of zero means the method call was successful. Otherwise, an error occurred. Error messages are written to the SAS log, and all return values for the FCMP Python object currently operate this same way. *Note: Future releases might assign specific errors to nonzero return codes, so at this time be aware not to assume the value for any error. Instead, check the SAS log for the error messages.*

Example 3: Parse Time - Read in a Python Program

```
rc = py.infile("c:\PythonSource\BlackScholes.py");
```

The INFILE method reads in the source during the FCMP parse phase. This allows you to populate the object with source code before the first observation of data is read. Therefore, a string literal argument is required. By comparison the RTINFILE method defers the reading of the Python source code until run time. This allows you to use observation data to determine how to finish writing the source code for your Python object. The argument might be a character string variable or a character string literal.

Example 4: Run time − Read in a Python Program

```
blackscholes = "c:\PythonSource\BlackScholes.py";
rc = py.rtinfile(blackscholes);
```

Typically, larger Python modules can be written using your favorite Python editor, saved to disk and included using the INFILE or RTINFILE methods. In this way care is taken to ensure that the proper white spacing Python requires is maintained.

Using SUBMIT INTO

The SAS language parser is white space agnostic. This means the general structure of SAS language statements might have one or more spaces in between tokens, and SAS is still able to interpret and execute the program. The Python language is different. Spaces are significant, and they are used to delineate the begin and end of functions, loops, and conditionals. To allow the SAS system to capture Python source code and not modify the whitespace, the FCMP SUBMIT INTO statement was created.

General Declaration 2:

```
submit into object-name;
<Python source code>
endsubmit;
```

The object-name is the FCMP symbol name for a previously declared Python object. This is the object that will hold the submitted Python source code. Every SUBMIT INTO must be paired with an ENDSUBMIT. The ENDSUBMIT statement terminates the code submission block of embedded Python source code and stores the code into the Python object. The ENDSUBMIT statement must be on a line by itself, and it must have no other leading tokens preceding it on the line, but leading white space is ok. Tokens that come after the ENDSUBMIT statement terminator, that is, the semi-colon, and the newline character are ignored. Everything between the SUBMIT INTO and the ENDSUBMIT lines are stored verbatim, white spaces and newlines included, into the Python object without any modification. Therefore, Python programmers can supply the necessary Python spacing necessary to successfully publish the Python source code to the Python interpreter. Putting these two statements together, we can write our first embedded Python function.

Example 5: Submit Into Statement

```
submit into py;
def pricing(reason, price, discount):
  "Output: newprice, mark"
  if reason.upper() == "MARKDOWN":
    newprice = price *(1-(discount+0.05))
    mark="MARKDOWN"
  else:
    newprice = price *(1-0.05)
    mark="STANDARD"
  return newprice, mark
endsubmit;
```

Referring to Example 5, this simple Python function will take an additional 5% off whatever discount rate is passed into the function when the "MARKDOWN" string is passed in as the reason. Otherwise, a "STANDARD" 5% discount is used. Notice the spacing of the Python source code, as required by the language. Also notice the line after the function declaration, which begins with "Output: ". Why is that line present?

> Python source code longer than 256 characters must be split into two lines using the Python line continuation character, which is a backslash '\'.

The "Output: " line is required by FCMP because Python performs a very late binding of its variables to their data type. So late is this binding that it is done as the Python interpreter is executing the lines of code. By comparison, the SAS FCMP language performs data type checking and symbol binding at compile time, before program execution and before the first observation is read. To support the integration between the two different systems, the Python programmer must provide a hint that allows MAS to know at a minimum the variable names for the data being returned from the function. This is only needed for the Python functions that are called from FCMP. other functions within the Python module, for example, helper functions not directly called from FCMP, can have any function signature supported by the Python language.

All data is returned from Python to MAS and FCMP as a Python tuple. This is true whether one or more values are being returned. In Example 5, two data values are returned; the first is numeric, the second is a character string. In general, the FCMP language does not support multiple return values, so how is this accomplished? In the section on calling your Python function we will show you how.

SUBMIT INTO Using a Path

The SUBMIT INTO statement provides a convenient way in which to embed Python source code into your SAS FCMP code. The source for the two different languages is then maintained together. When using the PROC FCMP OUTLIB= option, the SUBMIT INTO statement and the Python source are written into SAS function data set. You can see this by viewing the contents of the SAS function data set using the SAS® Explorer window. The Python *pricing* function used in Example 5 is small. What if you have a lot more lines of code, and you still want the Python source code to be placed into the SAS function data set alongside the FCMP language source? Use the SUBMIT INTO statement and supply a file path.

Example 6: Submit Into Using a File Path

```
submit into py("c:\PythonSource\BlackScholes.py ");
```

This variation of the SUBMIT INTO statement is still performed at compile time, but instead of reading lines from the SAS program, the file path is opened and read into the Python object just as if they had been specified in Example 5. An ENDSUBMIT should not be used, as this form of SUBMIT INTO is a complete statement by itself. Furthermore, the lines read from the file will be stored in the SAS function data set. Why is this distinction between SUBMIT INTO using a path important? Why not just use the INFILE method instead? To facilitate the movement of source code between SAS platforms.

More than ever, PROC FCMP source code is run on a grid environment. Examples include the HPRISK grid as well as the SAS® Cloud Analytic Server platform. Later in this paper we will show you how to use FCMPACT action set to create SAS function libraries in the SAS Cloud Analytic Server and run them. To summarize, both forms of SUBMIT INTO allow you to not only write code into your Python object, but they facilitate the transfer of your FCMP embedded Python code to other SAS platforms.

APPEND One Line at a Time

While the SUBMIT INTO statement allows you to write a block of Python code at a time, the APPEND method allows you to write a single line of Python source code into the Python object. APPEND is a run time method, and when used in conjunction with the CLEAR method, you can write, publish, call, clear, and rewrite a Python object repeatedly. Suppose you wanted your Python function to perform certain calculations based on changes in the observation data itself? Using the APPEND method, along with the RTINFILE method is how to accomplish it. Rewriting Example 5 in this way would look like what you see in Example 7.

Example 7: Append Method

```
rc = py.append('def pricing(reason, price, discount):');
rc = py.append('  "Output: newprice, mark"');
rc = py.append('  if reason.upper() == "MARKDOWN":');
rc = py.append('    newprice = price *(1-(discount+0.05))');
rc = py.append('    mark="MARKDOWN"');
rc = py.append('  else:');
```

```
rc = py.append('    newprice = price *(1-0.05)');
rc = py.append('    mark="STANDARD"');
rc = py.append('  return newprice, mark');
```

Again, the *rc* value is a return code, which you will want to check, but omitted here for brevity. The APPEND method follows standard SAS language quoting rules to allow you to write custom Python code a line at a time. As you become familiar with these new methods, you will develop a sense of what will work best for your needs. Now that we have written our Python function, let us move ahead and publish the source code to the Python interpreter.

PUBLISH Your Python Module

Publishing your Python object is the act of taking the source code, which has been stored in your Python object and submitting it to the Python interpreter. The Python interpreter will perform syntax checking, returning any syntax errors to the SAS log. Once published, the Python functions within your module can be called repeatedly from one observation to the next. This makes PUBLISH a run time method, like RTINFILE and APPEND. However, PUBLISH is only performed once for a Python object. Subsequent calls to PUBLISH are ignored until the Python object is cleared using the CLEAR method.

From the last section we learned how to write Python source code into your Python object. In addition,, multiple SUBMIT INTO statements and calls to INFILE or RTINFILE might be used to build your Python module sections at a time. Subsequent sections of code are appended to previous sections. Once your Python code creation is complete, publishing the source code pushes it to the Python interpreter. You then can call the Python functions within the module.

Example 8: Publishing a Python Module

```
rc = py.publish();
if (rc) then do;
   put "Publish of Python code failed.";
   return -1;
end;
```

Just as with other FCMP Python object methods, a return value of rc =0 means success.

CALL Your Python Function

This is the exciting part. Now that your Python object has been written and published, you are able to call it from within PROC FCMP or the SAS DATA Step. Assume for this section the Python object contains the source code from Example 5. Here is how you would call the *pricing* function.

Example 9: Calling a Python Function

```
length mark $10;
newprice = .;

reason   = "MarkDown";
```

```
price    = 9.99;
discount = 0.05;
rc = py.call("pricing", reason, price, discount);

newprice = py.results["newprice"];
mark     = py.results["mark"];
put newprice= mark=;
```

Recall that the pricing function returns two data values, *newprice* and *mark*. These values are returned to FCMP in a Python object member variable, named *results*. The r*esults* member variable is an FCMP dictionary object that can hold the results for different name/value pairs being returned from the tuple in the Python function. Now you can see how the "Output: newprice, mark" line in Example 5 is used. The names of the arguments returned from the function become member names in the *results* dictionary. The actual names of the variables used in the return statement are ignored. Instead, the first name in the "Output: " list is associated with the value for the first variable in the return statement. The actual Python variable name in the return statement might be named differently. The second variable value in the Python return statement maps to the second name in the "Output: " list, and so on.

Using dictionaries, FCMP can obtain an arbitrary number of Python values from any Python function. An output value from one Python call will overwrite any previous value of the same name in the *results* member dictionary. Comingling of output variables with different names from two or more Python function calls in the same *results* dictionary is possible and can make your programming more efficient. But for times when this is not desired, the dictionary CLEAR method might be used prior to making your next Python function call.

Example 10: Clearing Your Results

```
rc = py.results.clear();
```

Keep in mind that clearing the results dictionary before each Python function call does incur a performance cost. For times when your program is repeatedly calling the same Python function, clearing the results dictionary is not necessary. New values coming from the Python function will overwrite any previous value with the same "Output: " list name. For more information about using FCMP dictionaries, see the SAS Global Forum paper (SAS418-2017), entitled Dictionaries: Referencing a New PROC FCMP Data Type.

Mapping Data Types

Type conversion between the two languages is also augmented. Python supports all the SAS data types, for example, numeric and fixed character, but Python has certain additional data types SAS does not support, for example, Python object. Furthermore, certain returned Python types are coerced into a SAS data type. Table 1 shows how the mapping between types is made.

Table 1: Data type mapping between Python and FCMP

Python Data Type (returned)	SAS Data Type (converted to)
NoneType	Double

Python Data Type (returned)	SAS Data Type (converted to)
String	Fixed Characters
String Array	Fixed Character Array
Integer	Double
Integer Array	Double Array
Long	Double
Long Array	Double Array
Float	Double
Float Array	Double Array
Boolean	Double
Boolean Array	Double Array
Date	Date (Double)
Date Array	Date (Double) Array
Time	Time (Double)
Time Array	Time (Double) Array
DateTime	DateTime (Double)
DateTime Array	DateTime (Double) Array
Any other data type (for example, objects)	Unsupported

Treatment of missing values between FCMP and Python is something that deserves special attention. Python is a dynamically typed language. This means a Python variable of any type (for example, string, integer, or float) can be changed to have the *NoneType* type, which has a value of *None*, to indicate when a variable has no value.

This is like the SAS missing value, but there are differences. The Python NoneType is not data type specific; the SAS missing is data type specific. That is, a SAS numeric missing is commonly a period, '.', and a SAS character missing is a space, ' '. In Python there is only the one value, *None*. For this reason, all None values being returned from Python are mapped to the SAS numeric missing '.'. Therefore, when examining the results dictionary for a Python function that can return a missing value, use the dictionary DESCRIBE method to check the type. If you expect a character type and you find that it has been switched to a numeric missing, this is why.

CLEAR Method

Many users of the new Python object will follow a workflow where the programmer will write and publish the Python source code once, and then use it through each observation of the input and output data sets being processed. However, what if the problem you are trying to solve needs a more dynamic approach? That is, what if based on the value(s) of your data you needed to rewrite your Python object to consider a new calculation or analytic technique? The Python object CLEAR method allows you to accomplish this work flow.

Example 11: Clear Python Source Code from the Object

```
call py.clear();
```

By calling the CLEAR method, the Python object is cleaned of all knowledge of any previous Python source code, any Python dictionary results, and made ready to rewrite and publish the object anew. Only the name of the Python module, as defined in the DECLARE statement, remains. A Python object might be cleared and reused as often as needed.

> The CLEAR method does not have a return code. Therefore, the syntax is as a CALL subroutine.

A Complete Example

The Black-Scholes options pricing model is standard study for anyone in the financial sector. It is easily implemented in PROC FCMP, yet with the popularity of today's open-source community, many finance graduates use Python for their modeling needs. Because review and compliance within the banks and brokerage houses are strict, once a model has been validated in one language it can be burdensome to move the code to another language. This is where integration between languages truly shines. For comparison, here is the call option function when written in PROC FCMP.

Example 12: Black-Scholes Call Options Pricing Written in FCMP

```
proc fcmp outlib=work.bseopm.bs;
/* Black Scholes European Options Pricing Method for Call */
/* Dividend=0                                             */
/* S = Current Stock Price                                */
/* X = Option's Strike Price                              */
/* r = Risk Free Interest Rate                            */
/* T = Time to expiration (in days)                       */
/* v = Current price volatility                           */
function bseopmCall(s, x, r, T, v);
   if (x ne 0 and x ne . and v ne 0 and v ne .) then do;
      d1 = (log(s/x) + ((r + (v**2)/2) * T)) / (v * sqrt(T));
      d2 = d1 - (v * sqrt(T));
      C = (s * probnorm(d1)) - (x * exp( (-r) * T) * probnorm(d2) );
   end;
   else
      C = .;
return(C);
endfunc;
quit;
```

By comparison, here is what the same function would look like when written in Anaconda Python 3.7.

Example 13: Black-Scholes Call Options Pricing Written in Python and Integrated in FCMP

```
proc fcmp outlib=work.bseopm.pybs;

/* Black Scholes European Options Pricing Method */
function pyBlackScholesCallOption(curPrice, strikePrice, expireTime,
                                  priceVolatility, rfRate);

declare object py(python("BSEOPMCallOption"));
optionPrice = .;  rc = 0;
```

```
/*-----------------------------------------------------------------------
  Black-Scholes Calculation
  -----------------------------------------------------------------------*/
submit into py;
from scipy import stats
import math
def internal_black_scholes_call(stockPrice, strikePrice, timeRemaining,\
                                volatility, rate):
    if ((strikePrice != 0) and (volatility != 0)):
       d1 = (math.log(stockPrice/strikePrice) + (rate + (volatility**2)\
            / 2) * timeRemaining) / (volatility*math.sqrt(timeRemaining))
       d2 = d1 - (volatility * math.sqrt(timeRemaining))
       callPrice = (stockPrice * stats.norm.cdf(d1)) - \
                   (strikePrice * math.exp( (-rate) * timeRemaining) \
                 * stats.norm.cdf(d2))
    else:
       callPrice=0
    return (callPrice)
def black_scholes_call(stockPrice, strikePrice, timeRemaining, volatility,\
                       rate):
    "Output: optprice"
    optPrice = internal_black_scholes_call(stockPrice, strikePrice,\
                                           timeRemaining, volatility, rate)
    callPrice = float(optPrice)
    return (callPrice,)
endsubmit;

/* Publish the function to the Python interpreter         */
rc = py.publish();
if (rc) then do;
   put "Publish of Python code failed.";
   return(0);
   end;

/* Call Python function black_scholes */
rc = py.call("black_scholes_call", curPrice, strikePrice,
             expireTime, priceVolatility, rfRate);
if (rc) then do;
   put "Calling of Python code failed.";
   return(0);
   end;

optionPrice = py.results["optprice"];
return(optionPrice);
endfunc;
quit;
```

Which version is easier? That depends on the skill set of the modeler who is writing the code, the scope of the overall project, and the corporate culture of the organization. Either way, PROC FCMP is there to support you. For more information about the Python object and other built-in language objects, see the SAS documentation, entitled PROC FCMP and DATA Step Component Objects.

PROC FCMP INTEGRATION WITH ASTORE

Analytic containers are a recent technology that continue to grow in popularity. The premise behind them is that the logic created from training a model can be stored into a specialized file or table, called an ASTORE. This ASTORE is then decoupled from the code, which created it yet can be recalled and referenced by any future code that wishes to use it. This programming technique is available in SAS Viya and in SAS V9.4M5.

The way the technology works is that an action set in SAS Viya, like Forrest, FactMac, SVM, Text Mining, or Machine Learning, and so on is run to create the ASTORE container using a training data set. Once created the ASTORE can be referenced from other actions, for example, Fetch, FCMPACT, or as a CAS table computed column. The ASTORE can also be downloaded to the SAS V9 platform and used in PROC FCMP. Since PROC FCMP functions and subroutines can be called from the DATA Step, ASTORE integration is also extended to the DATA Step. Let us walk through the steps that allow you to take advantage of the technology.

DECLARING AN ASTORE OBJECT

The PROC FCMP ASTORE object is another built-in object supported on 64-bit UNIX and 64-bit Windows machines. Like other FCMP objects, it uses the DECLARE statement to create the object.

Example 14:

```
declare object myscore(astore);
```

Referring to Example 14, an FCMP object variable, named *myscore*, of type ASTORE is created. Once created, there are methods that can be called that allow you to perform operations using the analytic container. The first method you will want to call after declaring the object is the SCORE method.

SCORE METHOD

The SCORE method associates the location of the ASTORE container with the *myscore* variable. When using SAS Viya, the location is a CASLIB and TABLENAME, such as in Example 15.

Example 15: Specifying Your Scoring Model in CAS

```
call myscore.score("CASUSER", "_va_model208");
```

When using SAS V9, the location is a file path to the local disk or network, as in Example 16.

Example 16: Specifying Your Scoring Model in SAS V9.4M6

```
call myscore.score("C:\path\models\_va_model208");
```

In both cases, the contents of the trained analytical container are read into memory and placed into the FCMP object. Every ASTORE object has one or more input variables and one or more output variables. These input and output variables are automatically mapped to FCMP variables, making them accessible within the FCMP program. This is similar to the way input variables coming from a SAS data set are mapped to FCMP variables. The variables from a SAS data set can be referenced and modified in memory during program execution for each observation. In the same way, the SCORE method maps variables, often coming from the input SAS data set, to the inputs of the analytic container and calls the analytic container to perform the scoring algorithm for the observation. The result variables from the analytic container are returned to FCMP and placed into FCMP program variables. The FCMP programmer is then able to reference or change the FCMP variable the same as any other variable. Scoring is performed once for each observation per ASTORE object. Multiple ASTORE objects can be created and used in a single FCMP program, giving the user the power to evaluate complex modeling scenarios with one pass of the input data set.

This is great, but how do I discover the input and output variables for an analytic container if it was created by someone else? Use the DESCRIBE method.

DESCRIBE METHOD

In the database world a describe method is used to obtain meta information for a table or a view. In the same way, the ASTORE DESCRIBE method is used to obtain meta information for the input and output variables in the analytic container. The syntax is straightforward.

Example 17: Describe Input and Output Columns in the Scoring Model

```
PROC FCMP data=mydata.hmeq out=astore_fcmp_out;
  declare object myscore(astore);
  call myscore.score("C:\path\models\_va_model208");

  /* Use the DESCRIBE method to discover input and output variables */
  call myscore.describe();
quit;
```

DESCRIBE is a compile time method. Therefore, it prints the information to the SAS log before the first input observation is read or before program execution begins.

TIP: Use the (obs=1) data set option to limit the initial number of input data set observations when describing the input and output variables in the ASTORE.

Here is sample output for the model described in Example 17.

Output 1:

```
NOTE: Score Input Variables
NOTE:
NOTE:    CLAGE  Numeric
NOTE:    CLNO  Numeric
NOTE:    DEBTINC  Numeric
NOTE:    DELINQ  Numeric
NOTE:    NINQ  Numeric
NOTE:    VALUE  Numeric
NOTE:
```

```
NOTE: Score Output Variables
NOTE:
NOTE:    _P_  Numeric
NOTE:    P__EVENT_0  Numeric
NOTE:    P__EVENT_1  Numeric
NOTE:    I__EVENT_  Character
NOTE:    _WARN_  Character
```

Using the DESCRIBE statement in PROC ASTORE is a second way to discover the input and output variables in an analytic container.

SETOPTION METHOD

Some ASTORES can accept an option that will augment some portion of their algorithm. This can be to include additional output variables, or a change to how values are computed. It is beyond the scope of this document to describe all such options. Consult the SAS documentation for the actions sets you are using, but here is now you can set an option through the FCMP ASTORE interface.

Example 18: Setting Options in the Scoring Model

```
call myscore.setoption('RPCA_PROJECTION_TYPE', 1);
```

USING ASTORES WITH THE DATA STEP

Using an analytic container from the DATA Step is easy once you create an FCMP function. Rewriting Example 17 to wrap the scoring into an FCMP function we are now able to call the function to perform the scoring. From Output 1 we know ASTORE, *_va_model208*, returns a certain probability value, _P_. Here is how to obtain _P_ using the DATA Step.

Example 19: Calling ASTORE from the DATA Step

```
/* Using ASTORE scoring from an FCMP function */
proc fcmp outlib=work.score.funcs;
  function astore(clage, clno, debtinc, delinq, ninq, value);
     declare object myscore(astore);
     call myscore.score("C:\path\models\_va_model208");
     return(_P_);
  endfunc;
quit;

options cmplib=work.score;

data astore_ds_out;
  set mydata.hmeq;
  ds_p = astore(clage, clno, debtinc, delinq, ninq, value);
run;
```

Note that you have complete flexibility when writing the FCMP function. It can return to the DATA Step as many or few variables from the analytic container as you like. The function can also preprocess or post process the results based on certain rules you define each time an observation is scored.

CREATING AND USING AN ASTORE ON SAS VIYA

The best way to describe how to use the FCMP ASTORE object with SAS Viya is with an example. Let us assume you would like to analyze some data for classification and further regression analysis. The 'tkaasvm' action set in SAS Viya can be used for this purpose, and the following example will create the ASTORE analytic container, _va_model208, used in the examples above.

Example 20: Create an ASTORE from SVM

```
/* Run Support Vector Machine (SVM) action from VDMML */
proc cas;
    action builtins.loadactionset / actionSet='tkaasvm';
    action tkaasvm.svmtrain result=r /
  c=1.0,
  code={comment=false,fmtWdth=15,lineSize=200},
  includeMissing=false,
  maxiter=25,
  noscale=false,
  savestate={caslib="CASUSER",name="_va_model208"},
  table={
     caslib="CASUSER",compOnDemand="false",
     compPgm="
_va_calculated_208_1=round('BAD'n,1.0);
if (('_va_calculated_208_1'n = 0.0))then do;
   _va_calculated_208_11= 0.0;
end;
else do;
   _va_calculated_208_11= 1.0;
end;
_EVENT_=_va_calculated_208_11;
_va_FILTER_=(NOT(MISSING('_va_calculated_208_1'n))
        AND NOT(MISSING('CLAGE'n))
        AND NOT(MISSING('CLNO'n))
        AND NOT(MISSING('DEBTINC'n))
        AND NOT(MISSING('DELINQ'n))
        AND NOT(MISSING('NINQ'n))
        AND NOT(MISSING('VALUE'n)));
_va_calculated_208_14=NOT(('_va_FILTER_'n = 0.0));",
     compVars={"_va_calculated_208_1","_va_calculated_208_11",
               "_EVENT_","_va_FILTER_","_va_calculated_208_14"},
     name="HMEQ",onDemand="false",where="NOT('_va_calculated_208_14'n =
0)"},
  emtarget={name="_EVENT_",options={levelType="BINARY"}},
  tolerance=1.0E-6,
  var={"CLAGE","CLNO","DEBTINC","DELINQ","NINQ","VALUE"}
  outputTables={names={nobs="NObs",modelinfo="ModelInfo"}, replace="TRUE"};
quit;
```

The SAVESTATE statement in Example 20 directs SVM to create the ASTORE and save it to CASLIB="CASUSER" and table name="_va_model_208". We then can reference the ASTORE in other actions, such as this TABLE.fetch action.

Example 21: Scoring Data While Displaying a Table

```
proc cas;
  loadactionset "table";
  table.fetch
    format=false
    maxRows=1
    sasTypes=TRUE
    table = {
      compOnDemand=TRUE
      caslib="CASUSER"
      name="hmeq"
      compPgm="
declare object myscore(astore);
call myscore.score('CASUSER','_va_model208');"
      singlePass=TRUE
      compVars={"_P_", "P__EVENT_0" , "P__EVENT_1" , "I__EVENT_" ,"_WARN_"}
    };
quit;
```

Notice the DECLARE and CALL statements in the middle of Example 21. This is FCMP code. All actions in SAS Viya that support the TABLE statement also support a way to create computed columns using the *compPgm=* option, and the FCMP language is what is used to create the columns. This is a powerful way to perform custom programming to any action. In this example, we are scoring our data as we display it in the results table output.

To download the ASTORE from SAS Viya into SAS V9 we again use PROC ASTORE. With Example 22 you have everything you need to get started creating ASTOREs and using them in SAS Viya and SAS V9. The RSTORE= option refers to the ASTORE located in a CAS server. The STORE= option refers to the destination on the local file system or network where ASTORE is copied.

Example 22: Moving an ASTORE from CAS to SAS V9.4

```
proc astore;
download rstore=sascas1._va_model208
          store="C:\path\models\_va_model208";
run;
```

Not only are ASTOREs a powerful technology component, but their integration into FCMP opens new opportunities for use in SAS Viya via computed columns and FCMPACT, and in SAS V9 for the DATA Step.

THE FCMPACT ACTION SET

SAS® Viya® is a new and exciting technology evolution that allows you to scale your analytic system as you scale your business. Data volumes no longer conveniently fit entirely on a single SMP machine. Furthermore, in today's global markets you want to analyze all your data, not just a subset, and custom programming that captures the "secret sauce" of your organization remains at the heart of what you do. In this section, we introduce the FCMPACT action set. FCMPACT allows you to create custom functions and subroutines within SAS Viya the way PROC FCMP does this for SAS V9. Moreover, function and

subroutine libraries defined as SAS V9 data sets can be copied and used in SAS Viya. See the SAS documentation for PROC COPY for more details.

There are four main actions in the FCMPACT action set. They are:

- addRoutines − add FCMP functions or subroutines to a CAS table
- runProgram − execute FCMP code
- loadFcmpTable − load a single FCMP function table into memory
- loadFcmpLibs − load all FCMP tables in the session CMP library memory

Let us look at examples for each one.

ADDROUTINES ACTION

Again, using an example let us walk through how to store and call functions and subroutines from FCMPACT within the CAS server. Using PROC CAS, Example 23 uses the *addRoutine* action to place two subroutines, named *math1* and *hometown*, into the CASUSER.SUBTRN CAS table.

Example 23: Store FCMP Subroutines into CAS

```
proc cas;
session mysession;

/* Load the FCMP action set in the usual way */
loadactionset "fcmpact";

/* Add two FCMP subroutines to CASUSER.SUBRTN */
action addRoutines /
routineCode = "
  subroutine math1 (a, b, c);
    outargs b,c;
    b = a;
    c = a + b*b;
  endsub;
  subroutine hometown(city $, state $, returnstr $);
    outargs returnstr;
    returnstr = 'My hometown is: ' || ktrim(city) || ', ' || ktrim(state);
  endsub;"
package = "pkg"
saveTable=1
funcTable = {caslib="CASUSER" name="SUBRTN" replace=1};
quit;
```

After the FCMPACT action set is loaded, the *addRoutines* action is called with the *routineCode=* option to pass a quoted string containing the FCMP language source code to the CAS server. The *funcTable=* option specifies the name of the output table to store the functions. From using PROC FCMP, you know that functions and subroutines are stored within packages, so the *package=* option allows you to specify the package name. Setting saveTable=1 forces the funcTable to be saved to disk. Otherwise, the table will remain in memory within the CAS server until you explicitly promote the table to storage.

SETTING CMPLIB IN SAS VIYA

With the CASUSER.SUBRTN function table created, Example 24 shows how to set the CAS-based CMPLIB session option so that future actions can have reference to the functions and subroutines. The CMPLIB session option is like the SAS V9 CMPLIB system option shown in Example 19.

Example 24: Setting CMPLIB in CAS

```
proc cas;
  /* Replace/Update the value of the CAS session option 'cmplib' */
  action sessionProp.setSessOpt / cmplib="CASUSER.SUBRTN";

  /* What is the current value of the session option 'cmplib'?   */
  action sessionProp.getSessOpt / name="cmplib";
quit;
```

Multiple function tables can be specified in CMPLIB=, allowing FCMP language execution to find your functions and subroutines across multiple packages in multiple tables on the CAS server.

RUNPROGRAM METHOD

With the function and subroutine tables ready, Example 25 uses the *runProgram* action to execute FCMP code in the CAS server.

Example 25: Run an FCMP Program in CAS

```
proc cas;
  /* Run an FCMP program in CAS, calling our SUBRTN library */
  action runProgram /
    routineCode = "
      length residence $60;
      call math1(x, y, result);
      call hometown(city, state, residence);"

  /* Program is called on each row of input data */
  inputData={caslib="CASUSER" name="cityandnum"}

  /* Program results are written to the output table */
  outputData={caslib="CASUSER" name="outdta" replace=1};
quit;
```

The *routineCode*= option is a quoted string that contains the FCMP code to be executed. The program is executed once for each row of data in the input table specified by the *inputData*= option. Output variables are written to the output table specified in the *outputData*= option. If the output table already exists, it must first be dropped prior to running the action.

FCMP programs on SAS Viya can be simple or complex. You can define (and call) multiple functions and subroutines. You can also call SAS intrinsic and user-defined formats and functions that are available in the CAS server.

LOADFCMPTABLE AND LOADFCMPLIBS METHODS

Tables in SAS Viya must be loaded into memory prior to using them in actions. This includes tables containing FCMP functions and subroutines. To load a single table containing FCMP functions and subroutines, use the *loadFcmpTable* action. To load all tables referenced in the CMPLIB session option use the *loadFcmpLibs* action. Example 26 shows the syntax for each.

Example 26: Load an FCMP Function Library into Memory on CAS

```
action loadFcmpTable /
  table="mymath"
  caslib="casuser"
  replace = 1;

action loadFcmpLibs / replace = 1;
```

The *replace=* option causes the action to first drop any in-memory copies of the table(s) before loading a new version from disk. The tables are replicated in full on each node of the grid. This is a difference over a typical table load. For FCMP to find the functions and subroutines, the tables must not be partitioned. A complete copy of the table must exist on the controller node as well as each of the CAS worker nodes for the *runProgram* action to find the function or subroutine.

CONCLUSION

The use of objects within FCMP is expanding. ASTORE and PYTHON objects are the latest to be introduced, but as of this writing other objects are also on the R&D roadmap. You can expect more in the future, and the reason is simple. FCMP objects are a straightforward way to provide rich integration with foreign languages and other technology using a simple syntax and general interface. The possibilities are endless.

Another thing that is clear is that data volumes will continue to increase in the future. The only debate is at what rate? This means SAS® Viya® will continue to play a major role in shaping the way we use analytics to solve tomorrow's problems, as well as those of today.

The language found in PROC FCMP is present on all SAS platforms. It is the language of choice when creating user-defined functions and subroutines, computed columns in SAS Viya, and custom programming of functions in machine learning. This tight integration of a single analytics' centric language across this much technology gives you a powerful tool. It is our hope that this paper has helped you get to know FCMP better.

REFERENCES

Henrick, Andrew, and Mike Whitcher and Karen Croft. 2017. *SAS418-2017, Dictionaries: Referencing a New PROC FCMP Data Type*. SAS Global Forum 2017, Orlando, FL. Available at https://support.sas.com/resources/papers/proceedings17/SAS0418-2017.pdf

McNeill, Bill, and Andrew Henrick and Michael Whitcher and Aaron Mays. *SAS2125-2018, FCMP: A Powerful SAS Procedure You Should Be Using*. SAS Global Forum 2018. Denver, CO. Available at https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2125-2018.pdf

www.w3schools.com. *Python Tutorial*. Available at https://www.w3schools.com/python/

Hull, John C. 2018. *Options, Futures, and Other Derivatives*. 10th ed. New York, NY: Pearson

SAS Institute Inc. SAS® Micro Analytic Service 2.5M1: Programming and Administrative Guide. 2018. Cary, NC. Available at https://go.documentation.sas.com/?docsetId=masag&docsetTarget=titlepage.htm&docsetVersion=2.5M1&locale=en

SAS Institute Inc. Base SAS® 9.4 Procedures Guide, Seventh Edition: FCMP Procedure. 2018. Cary, NC. Available at https://go.documentation.sas.com/?docsetId=proc&docsetTarget=p10b4qouzgi6sqn154ipglazix2q.htm&docsetVersion=9.4&locale=en

SAS Institute Inc. Base SAS® 9.4 Procedures Guide, Seventh Edition: PROC FCMP and DATA Step Component Objects. 2018. Cary, NC. Available at https://go.documentation.sas.com/?docsetId=proc&docsetVersion=9.4&docsetTarget=p1ohgsqb6msp9fn1aj7yp5zq5ibl.htm&locale=en

ACKNOWLEDGMENTS

The authors of this paper would like to thank the following people: Chris Widman, Shannon Clark, David Duling, James Carroll, Stephen Vincent, Ted Dyer, Charles Shorb, Chris Johns, Shameka Coleman, and Daniel Underwood. All worked many hours with the authors of this paper under a tight release schedule to bring the SAS user community the new features presented. And just as importantly are the friendships that developed while accomplishing something exciting in the area of computer language development.

CONTACT INFORMATION

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

Michael Whitcher
SAS Institute
(919) 531-7936
mike.whitcher@sas.com