PharmaSUG 2024 - Paper AP-135

LAST CALL to Get Tipsy with SAS®: Tips for Using CALL Subroutines

Lisa Mendez, PhD, Catalyst Flex;
Richann Jean Watson, DataRich Consulting

ABSTRACT

This paper provides an overview of six SAS CALL subroutines that are frequently used by SAS® programmers but are less well-known than SAS functions. The six CALL subroutines are CALL MISSING, CALL SYMPUTX, CALL SCAN, CALL SORTC/SORTN, CALL PRXCHANGE, and CALL EXECUTE.

Instead of using multiple IF-THEN statements, the CALL MISSING subroutine can be used to quickly set multiple variables of various data types to missing. CALL SYMPUTX creates a macro variable that is either local or global in scope. CALL SCAN looks for the nth word in a string. CALL SORTC/SORTN is used to sort a list of values within a variable. CALL PRXCHANGE can redact text, and CALL EXECUTE lets SAS write your code based on the data.

This paper will explain how those six CALL subroutines work in practice and how they can be used to improve your SAS programming skills.

INTRODUCTION

Using CALL subroutines in SAS Programs provides a powerful way to extend the functionality of your SAS code. There are benefits to learning CALL subroutines but learning to use them properly will ensure your programs are robust and reusable. They enhance functionality, efficiency, and interoperability. By calling external subroutines, you can avoid reinventing the wheel. Instead of writing custom code within SAS, you can leverage existing solutions to perform specific tasks more efficiently.

One thing you need to ensure is that any external CALL subroutines you use are compatible with your SAS environment. Different versions of SAS may have varying levels of support for specific CALL subroutines.

CALL MISSING

A common SAS subroutine that sets one or more variable values (numeric or character) to missing is the CALL MISSING subroutine. Note: A missing value for a numeric variable is denoted by a period (.) while a missing value for a character variable is denoted by a blank space. You may ask, ‘why should I use the CALL MISSING subroutine to initialize variables to missing?’ Here are few reasons why:

1. Prevents unplanned assignments of values due to a value being carried forward from a previous DATA step or from a previous record.
2. Ensures a clean slate when dealing with imputation or derivations for numeric calculations.
3. Removes the ‘uninitialized value’ message in the log.

The syntax for CALL MISSING is

CALL MISSING(*variable-name-1* <, *variable-name-2,* ...>);

Before we look at sample code, let’s take a partial look at the SASHELP.BASEBALL data set (Sample Data 1).

	Name	Team	Salary
1	Allanson, Andy	Cleveland	.
2	Ashby, Alan	Houston	475
3	Davis, Alan	Seattle	480
4	Dawson, Andre	Montreal	500
5	Galarraga, And...	Montreal	91.5
6	Griffin, Alfredo	Oakland	750
7	Newman, Al	Montreal	70
8	Salazar, Argenis	Kansas City	100
9	Thomas, Andres	Atlanta	75
10	Thornton, Andre	Cleveland	1100

Sample Data 1: SASHELP.BASEBALL Subset

In this data set you will see that the first value of SALARY is missing (.) for Allanson, Andy, but the other values are present. Let's use the CALL MISSING subroutine to set all the salary values to missing as seen in SAS Program 1. Note that SALARY is numeric, yet with CALL MISSING, you can mix and match character types if the character type is already defined within the program data vector (PDV). This is illustrated in a subsequent example. In this case the PDV is populated with the data type with the SET statement.

The sample code will look something like this:

```
data work.baseball;
  set sashelp.baseball (keep = name team salary);

  /* Initialize salary variables */
  call missing(salary);

  /* Rest of your data processing steps */
run;
```

SAS Program 1: Sample Code to Initialize Numeric and Character Variables in a Data Set

After the code is run, Sample Data 2 illustrates the data set with the salary variable set to missing.

	Name	Team	Salary
1	Allanson, Andy	Cleveland	.
2	Ashby, Alan	Houston	.
3	Davis, Alan	Seattle	.
4	Dawson, Andre	Montreal	.
5	Galarraga, And...	Montreal	.
6	Griffin, Alfredo	Oakland	.
7	Newman, Al	Montreal	.
8	Salazar, Argenis	Kansas City	.
9	Thomas, Andres	Atlanta	.
10	Thornton, Andre	Cleveland	.

Sample Data 2: Subset of SASHELP.BASEBALL with Values Set to Missing

Another example would be a data set with lab results for patients; however, some of the values are not as expected. After further investigation, we find that the values are incorrect and we need to remove them (or set them to missing), but only for a select number of patients and for only a few variables with the variables being a mix of numeric and character variables.

Sample Data 3 shows a small data set named cholesterol, with patients and their cholesterol results.

	subjid	labid	cholesterol	cholesterol_stat
1	XX-US-123s45678-2001	LabCorp 1234	181	desireable
2	XX-US-123s45678-2034	LabCorp 1234	250	high
3	XX-US-123s45678-2314	LabCorp 1289	552	high
4	XX-US-123s45678-2076	LabCorp 1234	281	high
5	XX-US-123s45678-2098	LabCorp 1234	196	desireable
6	XX-US-123s45678-2065	LabCorp 1234	276	high
7	XX-US-123s45678-2390	LabCorp 1289	611	high
8	XX-US-123s45678-2086	LabCorp 1234	284	high
9	XX-US-123s45678-2011	LabCorp 1234	225	boderline
10	XX-US-123s45678-2045	LabCorp 1234	221	borderline

Sample Data 3: Sample of Cholesterol Results for a Patient

SAS Program 2 demonstrates the code we can use to set only a few variables to missing for some subjects as seen in Sample Data 4. In SAS Program 2, CHOLESTEROL is a numeric variable and CHOLESTEROL_STAT is a character variable. As with the previous example, the PDV is populated with the SET statement and so the data types for CHOLESTEROL and CHOLESTEROL_STAT are assigned.

```
data cholesterol2;
  set cholesterol;

  /* specify specific patients */
  if subjid in ('XX-US-123s45678-2314', 'XX-US-123s45678-2390') then
      call missing (cholesterol, cholesterol_stat); /* set only a few variables to missing */
run;
```

SAS Program 2: Sample Code to Set Only Specific Variables for Specific Records to Missing

	subjid	labid	cholesterol	cholesterol_stat
1	XX-US-123s45678-2001	LabCorp 1234	181	desireable
2	XX-US-123s45678-2034	LabCorp 1234	250	high
3	XX-US-123s45678-2314	LabCorp 1289	.	
4	XX-US-123s45678-2076	LabCorp 1234	281	high
5	XX-US-123s45678-2098	LabCorp 1234	196	desireable
6	XX-US-123s45678-2065	LabCorp 1234	276	high
7	XX-US-123s45678-2390	LabCorp 1289	.	
8	XX-US-123s45678-2086	LabCorp 1234	284	high
9	XX-US-123s45678-2011	LabCorp 1234	225	boderline
10	XX-US-123s45678-2045	LabCorp 1234	221	borderline

Sample Data 4: Sample of Cholesterol Results for a Patient with Specific Records Set to Missing

Using the CALL MISSING subroutine allows us to eliminate the need for arrays, loops, series of IF-THEN-ELSE, etc. in our code and set any variables in the argument to missing. As you can see in the second example, you can set multiple variables to missing in one CALL, and you can set both character and numeric variables to missing.

Before you use the CALL MISSING subroutine you can check to see if a variable has a missing value by using the MISSING function, which will return a value of one (1) if the argument is missing, or a value of zero (0) otherwise.

CALL SYMPUTX

There are several different ways to specify a macro variable: %LET, SQL procedure, positional parameter in a macro definition, key parameter in a macro definition.

In addition, to these common techniques for defining a macro variable, there are two CALL subroutines that can be used to assign a value to a macro variable within a DATA step. They both have two required arguments, the name of the macro variable and the value to be assigned.

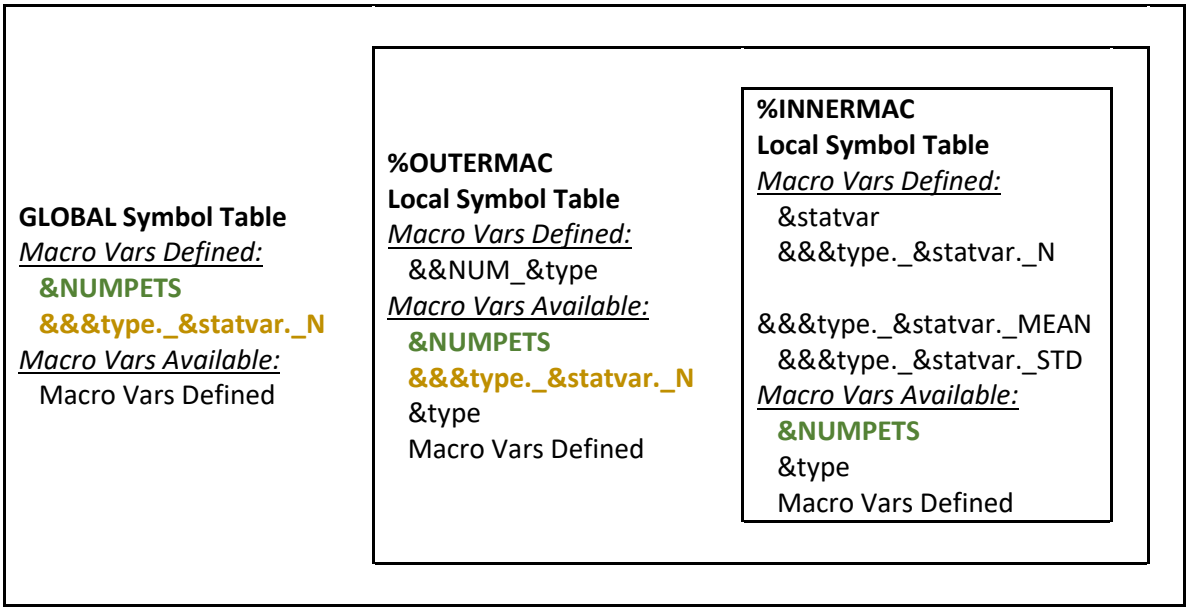
CALL SYMPUT(*macro-variable*, *value*);

CALL SYMPUTX(*macro-variable*, *value*<, *symbol-table*>);

While these may appear the same, they actually are different with respect to scope of the macro variable that is being created. In order to understand this difference, it is important to understand the symbol tables that a macro variable can be associated with. There is the global symbol table and local symbol table, with the local symbol table having different levels depending on the depth of nesting of macros (Display 1).

Macro variables within the global symbol table are available for access for the duration of the SAS session and can be referenced anywhere in the program (SAS Institute Inc., n.d.). However, macro variables within a local symbol table are available for access within the scope of the macro. Thus, local macro variables have no meaning outside the macro and if referenced outside the scope for which the macro is defined, you will get the following WARNING message.

```
WARNING: Apparent symbolic reference <<macro-variable>> not resolved.
```



Display 1: Illustration of Scope of Macro Variables (Watson & Hadden, 2022)

Now that we have an understanding of symbol tables, we can investigate the differences between CALL SYMPUT and CALL SYMPUTX.

CALL SYMPUT stores the macro variable within the most local table symbol if available. In order words, if you define a macro variable using CALL SYMPUT within %INNERMAC, then the macro variable can only be accessed within %INNERMAC. It cannot be referenced within %OUTERMAC or anywhere else in the program. In addition, when a numeric value is stored in a macro variable, it is converted to character and a message is written to the log.

CALL SYMPUTX allows you to specify which symbol table the macro variable is to be stored in. The options are 'G' for global, 'L', for the most local or 'F' for the local symbol table if it already exists. Otherwise, it is stored in the most local symbol table. SAS Program 3 illustrates the use of CALL SYMPUT and CALL SYMPUTX.

```
data class;
   set SASHELP.CLASS end = eof;
   if eof then do;
      call symput('numstudents', _n_);      ❶
      call symputx('laststudent', NAME, 'f');      ❷
   end;
run;
%put _user_;

%macro class;
   proc sort data = class;
      by descending AGE;
   run;
   data agecnt;
      set class;
         by descending AGE;
         retain cnt;
         if first.age then cnt = 0;
         cnt + 1;
         call symput(cats('age', AGE), cnt);
         call symputx(cats('dage', AGE), cnt);
         call symputx(cats('fage', AGE), cnt, 'f');      ❸
         call symputx(cats('fage', AGE), cnt, 'l');
         call symputx(cats('xage', AGE), cnt, 'g');
       run;
       %put _user_;
%mend class;

%class
%put _user_;    ❹
```

SAS Program 3: Sample Code to Illustrate CALL SYMPUT vs CALL SYMPUTX

❶ CALL SYMPUT is storing the number of students, a numeric value, as a macro variable. Notice that it produces a message in the log regarding conversion of a numeric value to character (yellow portioned highlighted in SAS Log 1). Note that when the numeric value is converted to a character value it is **not** left justified. By default, since there is no local symbol table, the macro variable NUMSTUDENTS is placed in the global symbol table and is available for reference throughout the SAS session.

❷ CALL SYMPUTX is storing the name of the last student in the file as a global macro variable regardless of what symbol table is specified. This is because when the third argument is not one of the options or the indicated symbol table cannot be found, it defaults to using the same symbol table that CALL SYMPUT uses. In this case it would use the global symbol table and therefore is available for reference throughout the SAS session.

❸ Within the CLASS macro definition, we illustrate the difference between CALL SYMPUT and SYMPUTX. The portion in SAS Log 1 that is in the turquoise box shows the local symbol for the %CLASS as well as the global symbol table. CALL SYMPUT within the macro definition utilizes the local symbol table if it exists and CALL SYMPUTX will use the symbol table used by CALL SYMPUT if a symbol table is not specified. At the creation of the macro variable AGE16, the local symbol table, CLASS, did not exist yet, therefore, AGE16 is stored as a global macro variable and since CALL

SYMPUTX without the third argument uses the same symbol table as CALL SYMPUT it will store DAGE16 in the global table as well (see turquoise highlight). For the other AGE and DAGE macro variables, these are stored in the CLASS symbol table because at the time of their creation the local symbol table had been created. The FAGE macro variables are created using the 'F' option in CALL SYMPUTX, therefore it will store the macro variables in the most local symbol table, in this case, CLASS. XAGE uses the 'G' option for CALL SYMPUTX and therefore the macro variables are stored in the global symbol table and can be used outside of the macro.

❹ To show that the macro variables that were stored in the local symbol table are no longer available, we can use %PUT _USER_ to list all the user-defined macro variables in the log. The portion in SAS Log 1 in the pink box shows the macro variables that are available for reference, which are all the global macro variables.

```
1    data class;
2       set SASHELP.CLASS end = eof;
3       if eof then do;
4          call symput('numstudents', _n_);
5          call symputx('laststudent', NAME, 'f');
6       end;
7    run;

NOTE: Numeric values have been converted to character values at the places given by:
      (Line):(Column).
      4:34
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.CLASS has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.00 seconds

8    %put _user_;
GLOBAL LASTSTUDENT William
GLOBAL NUMSTUDENTS          19
9
10   %macro class;
11      proc sort data = class;
12         by descending AGE;
13      run;
14      data agecnt;
15         set class end = eof;
16         by descending AGE;
17         retain cnt;
18         if first.age then cnt = 0;
19         cnt + 1;
20         call symput(cats('age', AGE), cnt);
21         call symputx(cats('dage', AGE), cnt);
22         call symputx(cats('fage', AGE), cnt, 'f');
23         call symputx(cats('fage', AGE), cnt, 'l');
24         call symputx(cats('xage', AGE), cnt, 'g');
25         if eof then call symputx('laststudent', NAME, 'f');
26       run;
27       %put _user_;
28   %mend class;
29
30   %class

NOTE: There were 19 observations read from the data set WORK.CLASS.
NOTE: The data set WORK.CLASS has 19 observations and 5 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds


NOTE: Numeric values have been converted to character values at the places given by:
      (Line):(Column).
      1:230
NOTE: There were 19 observations read from the data set WORK.CLASS.
NOTE: The data set WORK.AGECNT has 19 observations and 6 variables.
NOTE: DATA statement used (Total process time):
```

```
      real time             0.01 seconds
      cpu time              0.00 seconds

CLASS AGE11             2
CLASS AGE12             5
CLASS AGE13             3
CLASS AGE14             4
CLASS AGE15             4
CLASS DAGE11 2
CLASS DAGE12 5
CLASS DAGE13 3
CLASS DAGE14 4
CLASS DAGE15 4
CLASS FAGE11 2
CLASS FAGE12 5
CLASS FAGE13 3
CLASS FAGE14 4
CLASS FAGE15 4
CLASS FAGE16 1
GLOBAL AGE16            1
GLOBAL DAGE16 1
GLOBAL LASTSTUDENT Thomas
GLOBAL NUMSTUDENTS           19
GLOBAL XAGE11 2
GLOBAL XAGE12 5
GLOBAL XAGE13 3
GLOBAL XAGE14 4
GLOBAL XAGE15 4
GLOBAL XAGE16 1
31
32   %put _user_;
GLOBAL AGE16            1
GLOBAL DAGE16 1
GLOBAL LASTSTUDENT Thomas
GLOBAL NUMSTUDENTS           19
GLOBAL XAGE11 2
GLOBAL XAGE12 5
GLOBAL XAGE13 3
GLOBAL XAGE14 4
GLOBAL XAGE15 4
GLOBAL XAGE16 1
```

SAS Log 1: Log for SAS Program 3: Sample Code to Illustrate CALL SYMPUT vs CALL SYMPUTX

CALL SCAN

The CALL SCAN subroutine can play a crucial role in SAS programs. The SCAN function within SAS is a potent tool, engineered for breaking a string down into distinct words. This function allows us to set our own delimiters, which are then used to separate words or characters within a string. Common examples of delimiters include spaces, commas, asterisks, and slashes. Depending on which environment you are in, will determine the default delimiters.

- ASCII: blank ! $ % & ( ) * + , - . / ; < ^ ¦
- EBCDIC: blank ! $ % & ( ) * + , - . / ; < ¬ | ¢ ¦

Note that for ASCII environments if there is no ^, then ~ is also used as a default delimiter.

Before we go on, let's be clear about what a delimiter is and what a word is when using the CALL SCAN subroutine. Per the SAS Help Center, a delimiter is any character that is used to separate words. For example, a space (SAS Institute Inc., 2024). In the CALL SCAN subroutine, a word refers to a substring that has all the following characteristics:

- Is bounded on the left by a delimiter or the beginning of a string
- Is bounded on the right by a delimiter or the end of the string
- Contains no delimiters. (SAS Institute Inc., 2024)

Let's look at the syntax of the CALL SCAN routine:

CALL SCAN(<*string*>, *count, position, length* <, <*character list*> <, <*modifier(s)* >>>);

The count, position, and length arguments are mandatory, but string, character list, and modifiers are optional. Let's break down the arguments a bit, paying particular attention to the **count** argument.

String:	The string argument specifies a character constant (e.g. "T"), a variable, or expression.
Count:	The count argument is a nonzero numeric constant (i.e., positive or negative, e.g. "5"), variable that contains a numeric value, or expression that resolves to an integer value. The count specifies the number of the word in the character string (the nth word) that you want the subroutine to select. Let's review the count argument a bit. If the count is positive, then CALL SCAN counts words from LEFT (at the ***beginning*** of the string) to RIGHT in the character string. If the count is negative, then it counts words from RIGHT to LEFT in the character string from the ***end*** of the string.
Position:	The position is a numeric variable that holds the position of the first character of the specified word. Note that if count exceeds the number of words in the string, the value that is returned for the *position* argument is zero (0). If count is zero (0) or missing (.) the value that is returned for the *position* argument is missing (.).
Length:	The length is a numeric variable that holds the length of the word that is returned. Note that if count exceeds the number of words in the string, then the value that is returned for the *length* argument is zero (0). If count is zero (0) or missing (.) the value that is returned for the *length* argument is missing (.).
Character List:	The character list specifies an *optional* character constant, variable, or expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. There are a couple of rules. 1) By default, all characters in character-list are used as delimiters, and 2) if you specify the K modifier in the modifier argument, all characters that are not in character-list are used as delimiters (see modifiers below).
Modifiers:	Modifiers specify a character constant, variable, or expression in which each non-blank character modifies the action of the subroutine. We won't delve too deeply into modifiers in this paper, but you can find a list of modifiers in the SAS Help Center. However, we will mention the 'M' modifier, which indicates that multiple consecutive delimiters are treated as words with a length of zero and that delimiters at the beginning or end of a string are also treated as words with length zero.

(SAS Institute Inc., 2024)

Let's look at a couple of examples where we will use the CALL SCAN routine to find the first and last word of a character string. We will be using the SASHELP.WEBMSG data set; however, we will only be showing a partial data set (observations 60-69 in the original data set, Sample Data 5). The character strings in the TEXT variable have many characters, including special characters.

TEXT
the SAS System. The service you chose is release
%1$. Please choose another service.
<HTML>
<HEAD><TITLE>Application Error</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#8
<H1>Application Error</H1>
<P>System Message: %1$
System return code: %2$ </P>
<HR>
<ADDRESS>Please notify <A HREF="mailto:%1$">%2$</A>

Sample Data 5: Variable TEXT from a Modified SASHELP.WEBMSG Data Set

To begin, let's create a data set that will create the variables TEXT, POSITION and LENGTH and store the first word of the character string in the TEXT variable using the CALL SCAN subroutine (SAS Program 4 and Sample Data 6).

```
data webmsg2;
   set webmsg;

      /* obtain the first word from the variable named Text for all observations */
      call scan(text, 1, position, length);
      /* create a variable for the word returned from the CALL SCAN routine */
      First_Word = substrn(text, position, length);

run;
```

SAS Program 4: Sample Code to Illustrate CALL SCAN Subroutine

TEXT	position	length	First_Word
the SAS System. The service you chose is release	1	3	the
%1$. Please choose another service.	2	1	1
<HTML>	2	5	HTML>
<HEAD><TITLE>Application Error</TITLE></HEAD>	2	5	HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#800080" ALINK="#FF0000">	2	4	BODY
<H1>Application Error</H1>	2	14	H1>Application
<P>System Message: %1$ 	2	8	P>System
System return code: %2$ </P>	1	6	System
<HR>	2	3	HR>
<ADDRESS>Please notify <A HREF="mailto:%1$">%2$</A>	2	14	ADDRESS>Please

Sample Data 6: Sample Data from SAS Program 4: Sample Code to Illustrate CALL SCAN Subroutine

Remember that leading and trailing delimiters are ignored because the M modifier is not used. The reason many of the observations have a position of 2 is because the word is preceded by a '<', which SAS sees as a delimiter. Also notice that the number '1' is counted as a word, as numbers are not considered delimiters.

Now let's add in code to look for the last word of the string. Note that we can change the names of "position" and "length" to differentiate between the first word and the last word. One thing we want to stress is that when you use a negative number for the count (see code highlighted in yellow), SAS begins at the END of the string, and counts words from right to left. This is different than some other SAS functions, which may scan right to left starting at the absolute value of argument, so be careful when interpreting results. Because we are using -1, SAS will begin at the end of the string, and return the last word in the string. SAS Program 5 demonstrates the use of CALL SCAN to extract the first and last words of a string as seen in Sample Data 7.

```
data webmsg2;
   set webmsg;

      /* obtain the first word from the variable named Text for all observations */
      call scan(text, 1, first_pos, first_length);
      /* create a variable for the word returned from the CALL SCAN routine */
      First_Word = substrn(text, first_pos, first_length);

      /* obtain the last word from the variable named Text for all observations */
      call scan(text, -1, Last_Pos, Last_Length);
      /* create a variable for the word returned from the CALL SCAN routine */
      Last_Word = substrn(text, Last_Pos, Last_Length);

run;
```

SAS Program 5: Sample Code to Illustrate Extracts First and Last Word Using CALL SCAN

	first_pos	first_length	First_Word	Last_Pos	Last_Length	Last_Word
1	1	3	the	42	7	release
2	2	1	1	28	7	service
3	2	5	HTML>	2	5	HTML>
4	2	5	HEAD>	41	5	HEAD>
5	2	4	BODY	71	16	ALINK="#FF0000">
6	2	14	H1>Application	24	3	H1>
7	2	8	P>System	25	3	BR>
8	1	6	System	27	2	P>
9	2	3	HR>	2	3	HR>
10	2	14	ADDRESS>Please	50	2	A>

Sample Data 7: Sample Data for SAS Program 5: Sample Code to Illustrate Extracts First and Last Word Using CALL SCAN

CALL SCAN facilitates efficient word extraction and manipulation within SAS programs, which makes it one of the most vital pieces of code to know and have in your toolbox.

CALL SORTC / SORTN

We all have sorted data at the record level, but how about sorting data within a variable? Sorting values within a variable can be accomplished using the CALL SORTC (for character) and CALL SORTN (for numeric) values.

CALL SORTC(variable-1, <, …, variable-n>)

CALL SORTN(variable-1, <, …, variable-n>)

Like most anything else, before we can use it, we need to understand how it works. To do so, we need to understand the different collating sequences that can be used for comparing character values in the SORT procedure. PROC SORT either uses EBCDIC or ASCII collating sequences depending on the environment (SAS Institute Inc., 2024).

Collating Sequence for EBCDIC

blank . < (+ | & ! $ *) ; ¬ - / , % _ > ? : # @ ' = "
a b c d e f g h i j k l m n o p q r ~ s t u v w x y z
{ A B C D E F G H I } J K L M N O P Q R \S T
U V W X Y Z
0 1 2 3 4 5 6 7 8 9

Collating Sequence for ASCII

blank ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ∘ _
a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~

Notice that for EBCDIC lower case is sorted before upper case, while upper case is sorted before lower case in ASCII. In addition, the numbers for ASCII are sorted prior to the alphabet while they are sorted after the alphabet for EBCDIC. Furthermore, the various symbols are sorted differently. Of particular interest is the underscore, which is sorted after the upper-case letters in ASCII, but before all letters in EBCDIC.

Let's look at an example of how data would be sorted using PROC SORT. In Sample Data 8, the MYPEEPS2 data set is sorted using the default collation sequence, ASCII, for the environment in which this was run. If the desire is to sort using ASCII, there are other ways to ensure that ASCII collating sequence is used as shown in alternative ❷ and ❸ in Sample Data 8. Thus, in the particular environment in which the example was run, all three yield (❶, ❷, ❸) the same results. In Sample Data 9, we illustrate the sort order of the data if we use EBCDIC collating sequence. You can see that _AllPeeps is the last record in Sample Data 8 but it is the first record in Sample Data 9.

NAME
Richann
Louise
Troy
Josh
Lisa
_AllPeeps

```
❶ proc sort data = mypeeps2;
     by name;
  run;

❷ proc sort ASCII data = mypeeps2;
     by name;
  run;

❸ options SORTSEQ = ASCII;
  proc sort data = mypeeps2;
     by name;
  run;
```

NAME
Josh
Lisa
Louise
Richann
Troy
_AllPeeps

Sample Data 8: Sample Data Sorted Using Default Collating Sequence

NAME
Richann
Louise
Troy
Josh
Lisa
_AllPeeps

```
❶ proc sort EBCDIC data = mypeeps2;
     by name;
  run;

❷ options SORTSEQ = EBCDIC;
  proc sort data = mypeeps2;
     by name;
  run;
```

NAME
_AllPeeps
Josh
Lisa
Louise
Richann
Troy

Sample Data 9: Sample Data Sorted Using Specified Collating Sequence, EBCDIC

Now that we have that knowledge in our toolkit, let's turn to sorting within a variable. To accomplish this, we need to determine the number of items in our list (Sample Data 10) so that we can parse it out and save each element as an array element. Note that Sample Data 9 has tokens in no particular order. The goal is to put them in alphabeticaly order. In SAS Program 6, we use CALL SYMPUTX to store the number of elements in the list and that is used to build the array. The values for each array element prior to the CALL SORTC statement is shown in Sample Data 11. Once CALL SORTC is executed and MYPEEPS is reassigned, the values are sorted as seen in Sample Data 12. The idea for this came from Paul Dorfman's "Sorting Arrays Using Hash Object".

MYPEEPS
Richann Louise Troy Josh Lisa _AllPeeps

Sample Data 10: Sample Data Illustrating Sorting Values in a Variable

```
data _null_;
  set mypeeps;
  call symputx('numpeeps', countw(mypeeps));
run;

%put &=numpeeps;

data sortedpeeps (drop = peeps:);
  set mypeeps;
  array peeps(&numpeeps) $10;
  call missing(of peeps{*});
  do i = 1 to countw(mypeeps);
   peeps{i} = scan(mypeeps, i);
  end;
  call sortc (of peeps{*});
  mypeeps = catx(' ', of peeps{*});
run;
```

SAS Program 6: Sample Code to Sort Values in a Variable

MYPEEPS	PEEPS1	PEEPS2	PEEPS3	PEEPS4	PEEPS5	PEEPS6
Richann Louise Troy Josh Lisa _AllPeeps	Richann	Louise	Troy	Josh	Lisa	_AllPeeps

Sample Data 11: Sample Data Prior to Sorting

MYPEEPS	PEEPS1	PEEPS2	PEEPS3	PEEPS4	PEEPS5	PEEPS6
Lisa Josh Louise Richann Louise Troy _AllPeeps	Josh	Lisa	Louise	Richann	Troy	_AllPeeps

Sample Data 12: Sample Data After Sorting

Some things to note is that CALL SORTC uses ASCII. Even if you set the SORTSEQ option to EBCDIC on the OPTIONS statement, it will still use ASCII.

CALL EXECUTE

There are times when implementing data-driven techniques help with speeding up the programming process by allowing the data determine what code is written. For example, if we want to look for the maximum date for each subject in all the data sets in a given library, how would we achieve this? The brute force way would be to manually code logic to look at each data set and date variable combination as shown in SAS Program 7. This would require us to know every single data set and date variable combination and manually code that, but what if some combinations are removed or added. We would have to go back into the program and make this manual update.

```
%macro maxdt(dsn = , dtc = );
  data &dsn;
   set SDTMDATA.&dsn;
   format __&dtc. date9.;
   if length(&dtc) >= 10 then __&dtc = input(substr(&dtc, 1, 10), e8601da.);
   keep USUBJID __&dtc;
  run;

  proc sort data = &dsn;
   by USUBJID __&dtc;
  run;

  data __max_&dtc._&dsn;
   set &dsn;
   by USUBJID __&dtc;
   if last.USUBJID;
  run;
%mend maxdt;

%maxdt(dsn = AE, dtc = AESTDTC)
%maxdt(dsn = AE, dtc = AEENDTC)
…
%maxdt(dsn = VS, dtc = VSDTC)
```

SAS Program 7: Sample Code to Look for Maximum Date – Brute Force

An alternative approach is to use a data-driven technique that utilizes CALL EXECUTE subroutine. CALL EXECUTE subroutine takes one argument. It resolves the argument and then executes the argument at the next step boundary (e.g., run; quit;). CALL EXECUTE allows us to develop code that writes code based on the data.

CALL EXECUTE(*argument*);

The argument can be a character string enclosed in quotation marks or a variable (not enclosed in quotation marks) or a combination of both. Keep in mind that if you need to mask certain macro facility characters (e.g., %, &), those should be enclosed in single quotes. The use of double quotes within CALL EXECUTE will resolve to a macro invocation and execute immediately, which may not be the desired result.

Going back to our example of looking for the maximum date for each subject in each data set, we need to programmatically identify all the data set and date variable combinations. This can be easily achieved using DICTIONARY.COLUMNS (see SAS Program 8 and Sample Data 13).

```
proc sql noprint;
  create table alldts as
  select MEMNAME, NAME
  from DICTIONARY.COLUMNS
  where LIBNAME = 'SDTM' and NAME ? 'DTC';
quit;
```

SAS Program 8: Sample Code to Identify All Data Sets and Date Variable Combinations

MEMNAME	NAME
AE	AESTDTC
AE	AEENDTC
CE	CEDTC
CE	CESTDTC
CE	CEENDTC
CM	CMDTC
CM	CMSTDTC
CM	CMENDTC
DA	DADTC
DM	RFSTDTC
DM	RFENDTC
DM	RFXSTDTC

MEMNAME	NAME
DM	RFXENDTC
DM	RFICDTC
DM	RFPENDTC
DM	DTHDTC
DM	BRTHDTC
DS	DSSTDTC
DV	DVSTDTC
EC	ECSTDTC
EC	ECENDTC
EX	EXSTDTC
EX	EXENDTC
HO	HOSTDTC

MEMNAME	NAME
HO	HOENDTC
LB	LBDTC
MB	MBDTC
MH	MHDTC
MH	MHSTDTC
MH	MHENDTC
RP	RPDTC
SU	SUDTC
SU	SUSTDTC
SU	SUENDTC
VS	VSDTC

Sample Data 13: Sample List of All Data Sets and Date Variable Combinations

Now that we have identified all the possible combinations, we can use CALL EXECUTE to build the macro call statements. Note CALL EXECUTE is going to write all the %MAXDT call statements that were manually programmed in SAS Program 7 (portion highlighted in turquoise). CALL EXECUTE is called within a DATA step (SAS Program 9).

```
data _null_;
  set alldts;  ❶
  call execute (catx(' ', '%maxdt(dsn =', MEMNAME, ', dtc =', NAME, ')'));  ❷
run;
```

SAS Program 9: Sample Code to Illustrate CALL EXECUTE

❶ The program utilizes the data from the data set that contains the list of all possible data set and date variable combinations.

❷ CALL EXECUTE takes one argument. The argument can be a string within quotes, a variable or a combination of a string and variable. If using a combination of a string and variable, this can be done either by creating a new variable prior to the CALL EXECUTE statement and using the new variable or by building the statement within CALL EXECUTE by nesting functions. For this example, we opted to build the argument directly within CALL EXECUTE by nesting the CATX function. Notice that the macro call portion '**%maxdt(dsn =**' is in single quotes in order to mask the %. It is then followed by the variable name that contains the data set name (**MEMNAME**) and this is not enclosed in quotes. We continue to build the rest of the macro call by indicating the key word parameter in quotes, '**, dtc =**' and using the variable name that contains the date variable (NAME) not enclosed in quotes. Lastly, we close the macro call statement with the closing parenthesis. This is building all the %MAXDT call statements (e.g., %maxdt(dsn = AE, dtc = AESTDTC) %maxdt(dsn = AE, dtc = AEENDTC)).

Once we get to the step boundary (e.g., run;), all the %MAXDT statements are executed and the code that is written is shown in the log.

```
1  data _null_;
2    set alldts;
3    call execute (catx(' ', '%maxdt(dsn =', MEMNAME, ', dtc =', NAME, ')'));
4  run;

NOTE: There were 35 observations read from the data set WORK.ALLDTS.
NOTE: DATA statement used (Total process time):
   real time      0.00 seconds
   cpu time       0.01 seconds

NOTE: CALL EXECUTE generated line.
1  + data AE;     set SDTMDATA.AE;     format __AESTDTC date9.;     if length(AESTDTC) >= 10 then __AESTDTC =
input(substr(AESTDTC, 1, 10), e8601da.);     keep USUBJID __AESTDTC;  run;
```

```
NOTE: There were 411 observations read from the data set SDTMDATA.AE.
NOTE: The data set WORK.AE has 411 observations and 2 variables.
NOTE: Compressing data set WORK.AE increased size by 100.00 percent.
   Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: DATA statement used (Total process time):
   real time      0.00 seconds
   cpu time       0.00 seconds

1  +                                                                                                                    proc
sort data = AE;    by USUBJID __AESTDTC;  run;

NOTE: There were 411 observations read from the data set WORK.AE.
NOTE: SAS sort was used.
NOTE: The data set WORK.AE has 411 observations and 2 variables.
NOTE: Compressing data set WORK.AE increased size by 100.00 percent.
   Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: PROCEDURE SORT used (Total process time):
   real time      0.00 seconds
   cpu time       0.00 seconds

2  + data __max_AESTDTC_AE;    set AE;    by USUBJID __AESTDTC;    if last.USUBJID;  run;

NOTE: There were 411 observations read from the data set WORK.AE.
NOTE: The data set WORK.__MAX_AESTDTC_AE has 85 observations and 2 variables.
NOTE: Compressing data set WORK.__MAX_AESTDTC_AE increased size by 100.00 percent.
   Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: DATA statement used (Total process time):
   real time      0.00 seconds
   cpu time       0.00 seconds

<< repeated for each MEMNAME and NAME combination >>

69 + data VS;    set SDTMDATA.VS;    format __VSDTC date9.;    if length(VSDTC) >= 10 then __VSDTC =
input(substr(VSDTC, 1, 10), e8601da.);    keep USUBJID __VSDTC;  run;

NOTE: There were 6845 observations read from the data set SDTMDATA.VS.
NOTE: The data set WORK.VS has 6845 observations and 2 variables.
NOTE: Compressing data set WORK.VS increased size by 50.00 percent.
   Compressed is 6 pages; un-compressed would require 4 pages.
NOTE: DATA statement used (Total process time):
   real time      0.01 seconds
   cpu time       0.01 seconds

69 +                                                                                                              proc sort
data = VS;    by USUBJID __VSDTC;  run;

NOTE: There were 6845 observations read from the data set WORK.VS.
NOTE: SAS sort was used.
NOTE: The data set WORK.VS has 6845 observations and 2 variables.
NOTE: Compressing data set WORK.VS increased size by 50.00 percent.
   Compressed is 6 pages; un-compressed would require 4 pages.
NOTE: PROCEDURE SORT used (Total process time):
   real time      0.01 seconds
   cpu time       0.00 seconds

69 + data
70 + __max_VSDTC_VS;    set VS;    by USUBJID __VSDTC;    if last.USUBJID;  run;

NOTE: There were 6845 observations read from the data set WORK.VS.
NOTE: The data set WORK.__MAX_VSDTC_VS has 111 observations and 2 variables.
NOTE: Compressing data set WORK.__MAX_VSDTC_VS increased size by 100.00 percent.
   Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: DATA statement used (Total process time):
   real time      0.00 seconds
   cpu time       0.00 seconds
```

SAS Log 2: Log for SAS Program 9: Sample Code to Illustrate CALL EXECUTE

CALL PRXCHANGE

SAS has a variety of tools that help us to find search strings within another string or to change a specific set of characters or string within another string. But what if you don't have an actual string to search for that needs to be replaced. Instead, you have a pattern that needs to be redacted.

SAS utilizes Perl-regular expression (PRX) functions and CALL subroutines to help find these patterns within a text string. With PRX functions and CALL subroutines you can search for these patterns, and when a match is found you can modify it by swapping order or replacing the text. There are five PRX

functions and five CALL subroutines, but for the purpose of this paper we are only going to look at the PRXCHANGE subroutine.

CALL PRXCHANGE(*regular-expression-id*, *times*, *old-string* <, *new-string* <, *result-length* <, *truncation-value* <, *number-of-changes*> > > >);

CALL PRXCHANGE is used in conjunction with PRXPARSE function.

regular-expression-id = PRXPARSE(*perl-regular-expression*)

Because the syntax for PRX functions and subroutines are a bit clunky, we need to explain some metacharacters. Note only the metacharacters used in this example are listed in Table 1.

Metacharacter	Description
s/	Substitution operator
( )	Indicates a grouping
\	Escape character, overrides the next metacharacter
n	Matches capture buffer *n*
\d	Matches a digit (0 – 9)
\D	Matches any character that is not a digit
\w	Matches any word character or alphanumeric character, including the underscore
{*n*}	Matches *n* times
+	Matches the preceding subexpression one or more times

Table 1: List of Some PRX Metacharacters

We use the subject profile description in Sample Data 14 to illustrate the use of PRXPARSE and CALL PRXCHANGE. The goal is to mask the subject's name, the subject ID and the date of birth in every instance it is found. In addition, we want to change the order in which the condition is specified so that it reads more like a sentence.

DESCR
Smith, Jane: Overactive/Condition. Subject **001-001** date of birth **04JUL1976** exhibits high energy and inability to sleep. In addition subject **001-001** demonstrates high functioning even with the lack of sleep. **Smith, Jane** is like the energizer bunny. Recorded 06JAN2022.
Doe, John: Grumpy/Condition. Subject **001-002** date of birth **24MAY1972** has moderate energy and needs adequate sleep for full functioning. Subject **001-002** has a tendency towards grumpiness when not provided with enough sleep. **Smith, John** can be like a grumpy old bear. Recorded 06JAN2022.

Sample Data 14: Sample Data to Illustrate Redaction with CALL PRXCHANGE

To achieve the desired result can be a daunting task, but with PRXCHANGE this can be easily achieved. Well somewhat easy! It does take some time to figure out the PRX syntax. However, once you get over the hurdle of learning the PRX syntax, it can help with masking things like phone numbers or removing diagnostic codes from a patient chart or dictionary codes from coded terms in clinical data.

Using the data in Sample Data 14, we execute the code in SAS Program 10 to redact the information. The pattern that is going to be changed is stored in a regular-expression-id using the PRXPARSE function. In all four regid# statements, **s/** is used to indicate that we want to perform a substitution. The end of the pattern that is being searched for is also marked by a / and following is the string that is used for the substitution. The end of the substitution is also marked by a /.

```
data patprofile_masked;
  set patprofile;
  regid1 = prxparse('s/\d{3}-\d{3}/XXXXXX/');  ❶
  regid2 = prxparse('s/birth \d{2}\D{3}\d{4}/birth DDMONYYY/');  ❷
  regid3 = prxparse('s/(\w+), (\w+)/FNAME LNAME/');  ❸
  regid4 = prxparse('s/(\w+)\/(\w+)/$2 is $1/');  ❹
  call prxchange(regid1, -1, DESCR);
  call prxchange(regid2, -1, DESCR);
  call prxchange(regid3, -1, DESCR);  ❺
  call prxchange(regid4, -1, DESCR);
 run;
```

SAS Program 10: Sample Code to Redact Subject Information

❶ \d indicates we want to match on a digit. The {3} immediately following indicates we want to be 'greedy' and match on the digit 3 times. The hyphen is a non-metacharacter so it will be matched as a hyphen. We then have \d{3} to indicate we want 3 more digits. Thus, the pattern that is being searched for is 3 digits followed by a hyphen followed by 3 more digits (e.g., ###-###). Once the search pattern is identified, then the substitution value is specified which in this case it is 'XXXXXX'.

❷ birth is a string that will be searched for as is. It is followed by a space and then \d{2} which indicates we want 2 digits. Immediately following the 2 digits is \D{3} which indicates we want to match on 3 non-digits. That is then followed by 4 more digits (i.e., \d{4}). Thus, the pattern we are looking for is 'birth ##XXX####'. This pattern is then replaced with the indicated text 'birth DDMONYYYY'. You may be wondering why 'birth' was part of the search string if it is not being replaced with anything. It was added to the search string in case there are other dates, such as the recorded date, that should not be masked.

❸ (\w+) indicates that we want to match a group of characters. In this search pattern, we want a group of characters that is followed by a column and then followed by another group of characters. This is replaced with the text 'FNAME LNAME'.

❹ The fourth regular-expression-id is like the third in that it is looking for a group of characters separated by another group of characters, but instead of being separated by a comma it is separated by a /. Since / is a metacharacter that is used to indicate the start and end of a pattern, we need to use the override metacharacter \ to mask it. Thus, \/ is needed to indicate / is part of the search string. The search string is then replaced by a pattern that is based on the buffers. In the search pattern we identified two capture buffers. The first capture buffer was the group of characters prior to the /, while the second capture buffer is the group of characters after the /. According to the replacement pattern we are placing the group of characters in the second capture buffer first, followed by the string ' is ', and then followed by the group of characters in the first capture buffer.

❺ Now that all the search/substitution patterns have been assigned to regular-expression-id, that can be used within CALL PRXCHANGE to redact the data. One of the arguments within CALL PRXCHANGE is the number of times the substitution is to be made. If the value is -1, then the substitution will repeat until the end of the source value.

The result of the data (Sample Data 15) shows that the subject's name, ID and date of birth have been replaced with generic text and the condition is swapped so that it reads like a complete sentence.

DESCR
FNAME LNAME: Condition is Overactive. Subject XXXXXX date of birth DDMONYYY exhibits high energy and inability to sleep. In addition XXXXXX demonstrates high functioning even with the lack of sleep. FNAME LNAME is like the energizer bunny. Recorded 06JAN2022.
FNAME LNAME: Condition is Grumpy. Subject XXXXXX date of birth DDMONYYYY has moderate energy and needs adequate sleep for full functioning. Subject XXXXXX has a tendency towards grumpiness when not provided with enough sleep. FNAME LNAME can be like a grumpy old bear. Recorded 06JAN2022.

Sample Data 15: Sample Data of Redacted Subject Data using CALL PRXCHANGE

CONCLUSION

Let's conclude our exploration of some SAS CALL subroutines. These powerful tools enhance the capabilities of SAS programming by providing specialized functionality. CALL MISSING assigns missing values to specified character or numeric variables. It's particularly useful when you need to set both character and numeric variables to missing values. CALL SYMPUTX dynamically creates or modifies macro variables within a DATA step. It's handy for automating tasks and customizing program behavior. CALL SCAN extracts words from a character string based on specified delimiters. Useful for parsing text data and extracting relevant information. CALL SORTC/SORTN can sort character or numeric arrays. These routines allow you to efficiently organize data for subsequent analysis. CALL PRXCHANGE performs pattern-matching replacements using Perl regular expressions. It's great for data cleaning and transformation. CALL EXECUTE dynamically generates and executes SAS code within a DATA step. This is valuable for creating flexible and adaptive programs.

CALL subroutines can also be used for those routines that might not be readily available in SAS. For example, you can leverage advanced statistical algorithms from R or perform complex data manipulations using Python. CALL subroutines facilitate seamless communication between SAS and other programming languages. This interoperability allows you to combine the strengths of different tools within a single program. And always be cautious when executing external code within SAS. Validate the source of the routine and ensure it doesn't compromise the security of your data or system.

Leveraging CALL subroutines empowers SAS programmers to handle complex tasks, manipulate data effectively, and streamline their workflows. Remember to choose the appropriate subroutine based on your specific requirements and enjoy the efficiency they bring to your SAS programs!

ACKNOWLEDGMENTS

The authors want to thank Lex Jansen for continuing to publish SAS Proceedings from 1976 to present. The website searches 36,515 papers (as of the date of writing this paper) and is a wealth of information for SAS Users. https://www.lexjansen.com.

REFERENCES

Amy Alabaster, M. A. (n.d.). Cracking Cryptic Doctors' Notes with SAS® PRX Functions. SAS Global Forum. Retrieved Feb 2024, from https://support.sas.com/resources/papers/proceedings20/4638-2020.pdf

Campbell, J. (2012). Perl Regular Expressions in SAS® 9.1+ - Practical Applications. San Francisco, CA: PharmaSUG. Retrieved from https://www.pharmasug.org/proceedings/2012/TA/PharmaSUG-2012-TA08.pdf

Dorfman, P. (2018). Sorting Arrays Using Hash Object. St. Pete's Beach, FL: SESUG. Retrieved from https://analytics.ncsu.edu/sesug/2018/SESUG2018_Paper-288_Final_PDF.pdf

Kunwar, P. S. (2019). Quick Tips and Tricks: Perl Regular Expressions in SAS®. Dallas, TX: SAS Global Forum. Retrieved from https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/4005-2019.pdf

Mullins, L. (2019). Using the PRXCHANGE Function to Remove Dictionary Code Values from the Coded Text Terms. Philadelphia, PA: PharmaSUG. Retrieved from https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-315.pdf

SAS Institute Inc. (2023, Dec 11). *CALL SYMPUT Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/p09y28i2d1kn8qn1p1icxchz37p3.htm

SAS Institute Inc. (2023, Dec 11). *CALL SYMPUTX Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/p1fa0ay5pzr9yun1mvqxv8ipzd4d.htm

SAS Institute Inc. (2024, Jan 18). *CALL EXECUTE Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p1blnvlvciwgs9n0zcilud6d6ei9.htm

SAS Institute Inc. (2024, Feb 28). *CALL MISSING Routine*. Retrieved Mar 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p1iq436yh8838rn1ud38om45n99k.htm

SAS Institute Inc. (2024, Jan 18). *CALL PRXCHANGE Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/n0frf578x6vno8n1w26b6qn2wlt5.htm

SAS Institute Inc. (2024, Feb 28). *CALL SCAN Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/n0ecxfx00bn8i4n1vhh8up24ha6x.htm#p1j4nd5cxbbj4qn1nmiw4bzsfd92

SAS Institute Inc. (2024, Jan 18). *CALL SORTC Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/n00aheqlqf6qdln1og24z2hcsfd6.htm

SAS Institute Inc. (2024, Jan 18). *CALL SORTN Routine*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p0exzteatvicqkn1bzyzs0qaecy1.htm

SAS Institute Inc. (2024, March 24). *Functions and CALL Routines*. Retrieved from SAS Help Center: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/n0ecxfx00bn8i4n1vhh8up24ha6x.htm

SAS Institute Inc. (2024, Jan 18). *Pattern Matching Using Perl Regular Expressions (PRX)*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/n13as9vjfj7aokn1syvfyrpaj7z5.htm

SAS Institute Inc. (2024, Jan 18). *SAS® 9.4 Functions and CALL Routines: Reference, Fifth Edition*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/titlepage.htm

SAS Institute Inc. (2024, Jan). *SORT Procedure*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n1c1pczk6ogkbsn168zmf4k1izrh.htm#:~:text=The%20main%20features%20of%20the%20ASCII%20sequence%20are%20that%20digits,character%20that%20you%20can%20display.

SAS Institute Inc. (2024, Jan 18). *Tables of Perl Regular Expression (PRX) Metacharacters*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p0s9ilagexmjl8n1u7e1t1jfnzlk.htm

SAS Institute Inc. (2024, Jan 18). *Using Perl Regular Expressions in the DATA Step*. Retrieved Feb 2024, from SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p1vz3ljudbd756n19502acxazevk.htm

SAS Institute Inc. (n.d.). *CALL SYMPUT Routine*. Retrieved Feb 2024, from SAS® Visual Data Mining and Machine Learning 8.1: https://documentation.sas.com/doc/da/vdmmlcdc/8.1/mcrolref/p09y28i2d1kn8qn1p1icxchz37p3.htm

SAS Institute Inc. (n.d.). *Scopes of Macro Variables*. Retrieved from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/mcrolref/p1b76sxg9dbcyrn1l5age5j5nvgw.htm

Watson, R. J., & Hadden, L. S. (2022). *Functions (and More!) on CALL!* Retrieved from https://tp1210.p3cdn1.secureserver.net/wp-content/uploads/2023/01/Functions_and_More_on_CALL.pdf

Zach. (2023, May 1). *How to Use the MISSING Function in SAS (with Examples)*. Retrieved from Statology.org: https://www.statology.org/sas-missing-function/

CONTACT INFORMATION

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

Lisa Mendez
Catalyst Flex
lisa.mendez@catalystcr.com

Richann Jean Watson
DataRich Consulting
richann.watson@datarichconsulting.com