Paper SM16

# Table Numbers Presentation Made Easy with Custom Functions

**Sunil Kumar Pusarla, Omeros Corporation, Seattle, WA, USA**

## ABSTRACT

SAS software includes a vast array of functions but using them may not always yield the desired solution. Since version 9.2, SAS Institute has provided users with the flexibility to create their own functions, offering a more straightforward approach. Although many production macros exist, some might believe that PROC FCMP and user-defined functions have limited utility. However, combining simple formats or standard functions with PROC FCMP can produce refined and compact solutions for everyday tasks. Our custom user-defined functions are designed to be user-friendly, significantly reducing code complexity. This paper will discuss a few such user-defined functions that facilitate the elegant presentation of numbers in tables.

## OVERVIEW

Numerous papers have explored the use of PROC FCMP for defining functions, CALL routines, and subroutines, emphasizing their applications and benefits. This paper focuses on the practical integration of PROC FCMP with few SAS utilities to enhance the presentation of numerical data, particularly in tabular outputs, rather than providing an exhaustive review of its capabilities.

SAS statistical procedures generate results using default numeric formats such as BESTw. However, customized presentations—such as n (%), mean (SD), range, or confidence limits (CLMs)—are often required for reporting across various use cases, including regulatory submissions, conference abstracts, and press releases. Achieving these formats typically requires converting numeric values into character values and concatenating them accordingly.

While SAS provides a comprehensive suite of built-in functions that allow for nesting and the definition of keyboard macros for frequently used operations, user-defined functions remain crucial. Built-in functions often necessitate repetitive use across multiple panels, parameters followed by post-processing steps. As program specifications evolve, modifications must be made to accommodate variations for different analytical needs, which can be tedious and error-prone when applied manually across multiple instances, particularly under tight deadlines.

User-defined functions created with PROC FCMP offer a structured and efficient solution by streamlining data formatting, ensuring consistency, reducing manual effort, and improving adaptability in dynamic

specifications. When combined with standard SAS functions and PROC FORMAT, these functions help programmers minimize coding redundancy and seamlessly adjust to evolving decimal precision requirements. By embedding these functions within macros or using them as keyboard shortcuts, programmers can enhance flexibility while mitigating the risk of errors, such as missing updates in few sections of the code.

Additionally, function libraries can be managed at various levels—study, project, or enterprise—and referenced using the CMPLIB system option. This structured approach ensures consistency across deliverables while enhancing efficiency and maintainability in statistical programming workflows.

## METHODOLOGY

In tables and figures (e.g., via XAXISTABLE or annotations), character results, derived by concatenating two or more numeric values, often need to be displayed alongside numeric values (such as *n* or *median*). Since a single column cannot contain both numeric and character data, numeric values must first be converted to character format using the PUT function with the appropriate format.

As mentioned above, different statistical outputs require varying decimal precision. Instead of using multiple processing statements, programmers can optimize their coding efforts by leveraging formats and the PUTN function, which allows dynamic numeric formatting at runtime. These elements can be integrated into user-defined functions, as demonstrated in *Box 1*. This approach mitigates concerns related to frequently changing specifications while ensuring consistency and efficiency in output formatting.

**Box 1**

| Creating required format based on specification | Using this format and SAS functions while creating user defined functions using PROC FCMP |
|---|---|
| **proc format**;<br>value $statfmt<br>'N' = **6.**<br>'STD' = **6.3**<br>'Mean' = **6.2**<br>'Median' = **6.1**<br>'Minimum' = **6.1**<br>'Maximum' = **6.1**<br>'Percent' = **6.1**<br>'UCLM',<br>'LCLM' = **8.4**;<br>**run**; | /* To assign the format of Percent with 1 decimal from the $statfmt created */<br><br>y1 = strip (putn (xxx, put ('Percent', $statfmt.)));<br><br>/* To assign the format of Mean with 2 decimals & STD with 3 decimals from the $statfmt created */<br><br>x1 = strip (putn (x, put( 'Mean', $statfmt.)));<br>y1 = strip (putn (y, put ('STD', $statfmt.))); |

**USE CASES**

When presenting the number of subjects or events along with their percentages in a table, the conventional approach requires multiple steps as seen in Box 2:

- Converting the numeric count to a character value using the PUT function with the appropriate format.
- Removing leading and trailing blanks.
- Concatenating an open parenthesis (.
- Calculating the percentage, converting it to a character value with the desired decimal precision, and remove leading and trailing blanks.
- Appending the above with a % sign followed by a closing parenthesis ).

This process must be repeated for each treatment group across the table, often requiring manual duplication or copy-pasting of code. A major challenge arises when modifications to decimal precision are requested after the outputs are produced. Identifying every instance where decimal formatting is applied (e.g., .1 or .2) and updating it accordingly can be error-prone, requiring multiple trial-and-error executions. This not only increases the risk of misrepresentation but also adds to the overall inefficiency of the process.

A user-defined function elegantly handles such formatting complexities with minimal effort. For instance, by creating a study- or project-level format with the required decimal precision for percentages ($statfmt. as demonstrated in *Box 1*), all programs within the study can automatically reference this standardized format (via autoexec).

If a specific table or figure or parameter requires a different decimal precision for percentages, PROC FORMAT can be called within that program with the modified specification. Placing this modification at an easily identifiable location—typically at the beginning of the program, following the header section—ensures seamless future adjustments.

**Example 1: Presenting Statistical component combinations – Mean and STD**

As seen in the *code snippet 1* below*,* the PUT function applies the $statfmt format to Mean/STD, while the PUTN function applies PUT('Mean/STD', $statfmt) to the x variable at runtime. Here, the variables Mean and STD are assigned their respective decimal values as defined in the standardized format -

$statfmt. (*Box 1*). While PUT function can apply only a single format (e.g., 6.2) to every observation, PUTN allows different formats for different observations within the same column.

*Code snippet 1* also illustrates the post-processing that occurs, which needs to be implemented only once when creating the function. These lines demonstrate how the assigned format (x1) and the BESTw. format (x2) are used to calculate values. The assigned format pads the decimal, whereas the BESTw. format does not pad or round off values. For instance (*lines 11 to 13 in the code snippet 1*), if the mean value of the x variable is 12 (a whole number), the assigned format 6.3 will pad it with three zeros, resulting in x1 = 12.000, whereas x2 = 12 because BESTw. does not apply padding. The shorter of the two values is assigned i.e., x1 = x2 (12) which is more meaningful in this context.

**Code Snippet 1**

```
1   libname myflib 'C:\...\FCMP';
2
3   proc fcmp outlib = myflib.myfunc.report;
4
5   /* Concatenating the mean and standard deviation */
6
7        function MeanStd(x,y) $40;
8        length want x1 y1 x2 y2 $100;
9
10       if (n(x) eq 1) then do;
11         x1 = strip(putn(x,put('Mean', $statfmt.)));
12         x2 = strip(put(x,best.));
13         if (length(x2) lt length(x1)) then x1 = x2;
14       end;
15
16       if (n(y) eq 1) then do;
17         y1 = strip(putn(y,put('STD', $statfmt.)));
18         y2 = strip(put(y,best.));
19         if (length(y2) lt length(y1)) then y1 = y2;
20       end;
21
22       if (n(x,y) eq 2) then want = cats(x1,'(',y1,')');
23       if ((n(x) eq 1) and (n(y) ne 1)) then want = x1;
24       if f (n(x,y) lt 1) then  want = '';
25       want = tranwrd(want,'(',' (');
26       return(want);
27       endsub;
28  run;
```

Using this approach streamlines the code required to produce the desired result by reducing the need for multiple lines of code by consolidating numeric-to-character conversion, concatenation, and formatting into a single function call (as seen on left and right sides of *Box 2* and *Box 3*).

**Box 2**

| Without using the user defined function | Using user defined function |
|---|---|
| if n (placebo) eq 1 then<br>  col1 = strip (put  (placebo, 4.)) ||<br>  ' (' \|\| strip (put ((placebo / &col1.)*100, 6.1)) \|\| '%)';<br>  else col1 = '0';<br><br>if (col1 eq '100.0%') then col1 = '100%';<br>  else if (col1 eq '0(0.0%)') then col1 = '0'; | col1 = *npct* (placebo, &col1); |

Additionally, when only a single observation is available for a particular visit, post-processing becomes necessary. However, utilizing user-defined functions eliminates this need and allows for generating results in various formats, as demonstrated in *Box 3*.

**Box 3**

| Without using the user-defined function | Using user-defined function |
|---|---|
| length meanc stdc meansd $40;<br>if n (mean) eq 1 then meanc = strip (put (mean, 8.2));<br>if n (std) eq 1 then stdc = strip (put (std, 8.3));<br>meansd = strip (meanc) \|\| ' (' \|\| strip (stdc) \|\| ')';<br><br>/* Post-processing steps required */<br>meansd = tranwrd (meansd, ' ()', '');<br>meansd = scan (meansd,1,'.'); | col2 = *meanstd* (mean, std);<br><br>/* Can yield results with different formats */<br><br>Visit 13:  25.72 (3.693)<br>Visit 14:  26.12 (3.8)<br>Visit 15:  25 (4.382)<br>Visit 16:  23.1 (3.42)<br>Visit 17:  23 |

**Example 2: Presenting 'n (%)'**

If the value of Percentage for a variable is 100, it is displayed as 100.00 based on the specified format ($statfmt.) of 6.2 for 'Percent'. By incorporating *lines 27 and 28 of the code snippet 2*, the length of the value y1 is calculated. If the length of y1 exceeds three characters (e.g., 100.00 or 100.0), functions such as SUBSTR can be used to ensure the value is displayed as 100. Furthermore, defensive coding strategies can be integrated early in the code development process to account for anticipated scenarios and conditions (ex: *lines 22 to 25 of the code snippet 2*).

Thus, by leveraging commonly used SAS functions such as N, STRIP, PUT, LENGTH, SUBSTR, and TRANWRD, along with obscured functions like PUTN and PROC FORMAT, the output can be refined for consistency in appearance and formatting. Additionally, user-defined functions account for missing values and ensure that missing values within a combination result do not require additional post-processing.

**Code snippet 2**

```
1   libname myflib 'C:\...\FCMP';
2
3   proc fcmp outlib = myflib.myfunc.report;
4
5   /* Concatenating the n with percent (after creating it) */
6
7        function npct(x,y) $40;
8        length want x1 y1 y2 $100;
9
10       if (n(x) eq 1) then x1 = strip(put(x,best.));
11         else              x1 = '0';
12       if ((n(x,y) eq 2) and
13          (y ne 0))   then Pct = (x/y)*100;
14       if n(pct) eq 1 then y1  = strip(putn(Pct,
15                                           put('Percent',
16                                           $statfmt.)
17                                           ));
18
19       if n(pct) eq 1 then y2 = strip(put(pct, best.));
20       if (length(y2) lt length(y1)) then y1 = y2;
21
22       if ((n(x) ne 1) or
23           (x eq 0) or
24           (y eq 0) or
25           (n(y) ne 1)) then  y1 = '';
26
27       if ((length(y1) ge 3) and
28           (substr(y1,1,3) eq '100')) then y1 = '100';
29
30       want = cats(x1,'(',y1,'%)');
31       if (y1 eq '') then want = '0';
32       want = tranwrd(want,'(',' (');
33       return(want);
34  endsub;
35  run;
```

## CONCLUSION

In statistical programming, evolving data and changing specifications often necessitate frequent modifications. Manual updates increase the risk of errors and inconsistencies, underscoring the need for flexible and robust coding practices. User-defined functions offer a structured solution by simplifying code, reducing worktime, enhancing adaptability, and ensuring accuracy. By leveraging SAS functions and formats in custom function development, programmers can significantly reduce coding effort while

maintaining precision and consistency in an environment where the data and specifications are ever-changing.

**The opinions expressed in this paper are those of the author and do not represent the views of Omeros Corporation.**

## REFERENCES

SAS Documentation:
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/pgmsashome/titlepage.html

Pusarla, Sunil Kumar, Hamilton, Paul. 2018. "Techniques for Global, Adaptable, and Efficient Programming." PharmaSUG.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged.

**Author Name:** Sunil Kumar Pusarla
**Company:** Omeros Corporation
**Address:** 201 Elliott Avenue West, Seattle, WA 98119
**Work Phone:** 206-676-5000
**E-mail:** skumar@omeros.com