Paper 255-29

An Introduction to SAS Function-ality

Deb Cassidy, Dublin, OH

ABSTRACT

A major advantage of SAS® over some other languages is the vast number of built-in functions that make it very easy to do a task. In SAS 9, there are over 450 such functions. This presentation will include an overview of the structure of functions, some examples of commonly used functions and some pitfalls that often trip a new user. You'll also learn where to go to learn even more about the available functions and what they do. So, if you want to learn a quick way to add fields that might have missing values, extract part of a text field or just get an idea of what else you can do with functions, then this presentation is for you.

INTRODUCTION

A SAS function is a built-in method to perform a computation or system manipulation and return a value. A CALL routine alters variable values or performs other system functions. CALL routines are similar to functions, but differ from functions in that you cannot use them in assignment statements. There are over 450 functions and call routines in 26 categories. This paper will focus on a sample of functions to give you an overview of how to use them and what they can do. Some of the functions I've included are probably more complex that a new SAS user might use but I wanted to show them because you just might inherit code written by a more experienced user.

The general format of a function is simply:

```
myvalue=FUNCTION_NAME(required_argument1, …, required_argumentN,optional_argument1,…,optional_argumentN);
```

However, the number of required and optional arguments varies. Some functions simply have one required argument. Others have one required and one or more optional arguments. Functions that are more complex may have several required arguments. In most cases, the order of the arguments is important. Some people may think that if the code runs, then they had the arguments in the right order but that isn't always the case. Some functions will give different results if you accidentally swap the order of the arguments.

CATEGORIES

The categories of functions were modified somewhat in SAS 9. The current categories are:

Category	# of Functions	# of Call Routines
Application Response Measurement	6	
Array	3	
Bitwise Logical Operations	6	
Character	74	6
Character String Matching	5	9
Currency Conversion	1	
Date and Time	25	
DBCS (Double-byte character set)	22	
Descriptive Statistics	23	
External Files	34	
External Routines	4	2
Financial	23	
Hyperbolic	3	
Macro	2	3
Mathematical	30	2
Probability	18	
Quantile	7	
Random Number	12	12
SAS File I/O	29	
Special	25	4
State and ZIP Code	10	
Trigonometric	6	
Truncation	11	
Variable Control		3

Variable Information	30	1
Web Tools	4	
TOTAL COUNT	413	42

APPLICATION RESPONSE MEASUREMENT

These functions are used for monitoring performance. SAS recommends using the application response macros instead of the functions. These functions are not really for beginners.

ARRAY

An array is a "group" of variables that can be referenced by a single name instead of each individual variable name. The functions tell you things about the size of your array. These functions are also not really for beginners.

BITWISE LOGICAL OPERATIONS

These functions deal with the bits in each byte of data. These functions are not really for beginners either.

CHARACTER

Finally, we get to a category with some functions that are appropriate for beginners. Of course, with 80 functions in this category, you can bet that some are more challenging.

Let's start with something real simple. You have a field with names but the data entry person was not consistent and was a poor typist. Hence, the names look like:

```
JOE
John
pETE
RoberT
```

You'd like them to be consistent.

```
first_name=UPCASE(first_name);
```

The results give you:

```
JOE
JOHN
PETE
ROBERT
```

There is also a corresponding **LOWCASE** function to change characters to lowercase. What if you want the first character uppercase and the rest lowercase? Here's one way to do this.

```
first_name=CAT(UPCASE(SUBSTR(first_name,1,1)), LOWCASE(SUBSTR(first_name,2)));
```

If you didn't follow this, then let's break it into pieces. Perhaps this is easier to follow. I'm simply using "**pETE**" as my example.

```
first_letter=SUBSTR(first_name,1,1);
rest_letters=SUBSTR(first_name,2);
first_letter=UPCASE(first_letter);
rest_letters=LOWCASE(rest_letters);
first_name=CAT(first_letter,rest_letters);
```

You split the first letter off from the name and uppercase it. You split off the rest of the letters and lowercase them. You use the **CAT** function to concatenate them back together. That's easy to follow but it doesn't work quite as planned. If you actually submit the above code, you get the following warning (note that the number of characters and line/column numbers will be specific to your program):

```
WARNING: In a call to the CAT function, the buffer allocated for the result was not long enough to contain the
concatenation of all the arguments. The correct result would contain 8 characters, but the actual result may either be
truncated to 4 character(s) or be completely blank, depending on the calling environment. The following note
indicates the left-most argument that caused truncation.
NOTE: Argument 2 to function CAT at line 44 column 12 is invalid.
first_name=  First_letter=p Rest_letters=ETE _ERROR_=1 _N_=1
```

By default, the length of a field created from **SUBSTR** is the length of the original field. First_name had a length of 4 in my example name. Therefore, the length of first_letter is 4 and the length of rest_letters is also 4. Thus, it was trying to put 8 bytes into a 4-byte field when you did the concatenation.

How do you fix this? One method I tried was to create a new field so I used

first_name2=CAT(first_letter,rest_letters);

While I didn't get an error message, my result was blanks in first_name2. I'm not sure I can explain why. Although the default length for **SUBSTR** is the length of the original field, the default length for many other character functions such as **CAT** is 200. That means the length of first_name2 is 200 so it was definitely long enough.

Have you guessed that the solution to our problem might have to do with controlling the length? You can set the length of a new field with a length statement.

length field_name2 $10;
(Note, $10. also works. The decimal used to be required in early versions of SAS but is now optional.)

Now you'll discover
first_name2=CAT(first_letter,rest_letters);

Results in **P   ete** . Oops, the code is not quite right yet. Remember, the length of first_letter is 4, which means it is a single letter followed by 3 blanks. By default, **CAT** does not eliminate the blanks. You actually have multiple ways to resolve this problem.

Method 1 – specify the length for all new fields and replace the original field with the new data.

length first_letter $1 rest_letters $3.
first_name=CAT(first_letter,rest_letters);

Method 2 – use a concatenate function that trims trailing blanks.

first_name=CATT(first_letter,rest_letters);

Some of you might have already seen the **TRIM** function and be thinking it could be used here as in:

first_letter=TRIM(first_letter);
rest_letters=TRIM(rest_letters);
first_name=CAT(first_letter,rest_letters);

Unfortunately, this doesn't work. Although you used the trim function, you put the results back into the longer field so the trailing blanks came back. You could use:

first_name=CAT(TRIM(first_letter),rest_letters);

You don't need the **TRIM** on the rest_letters field because the trailing blank on the rest_letters gets truncated. First_letter occupied one spot which left only 3 for rest_letters.

Now to really confuse you, why didn't we get the same error message as we did above since the total of the trimmed first_letter and rest_letters was longer than 4?. Simple, the error message above occurred because the first item placed in the concatenation used all the available space and SAS could not even attempt to include the second item. In this last example, there was space for at least part of the second item so the concatenation was successful. The "extra" characters in the field were actually truncated. Since it was a blank in my example, it wasn't obvious.

The moral of this story – character functions are not hard to use but you must always watch the default length for the function, the length of your original fields, and leading/trailing blanks. The second moral of the story – combining multiple functions into one statement may process differently than using several statements. Some people may argue that one is easier to read than the other is. However, not everyone will agree which method is easier. That is one of the things that you or your company will have to decide. My personal opinion, combining several functions or using them separately really depends upon the functionality you are trying to accomplish. Readability for you and others should be a major factor in deciding whether to combine multiple functions.

We've been talking about blanks so here are a couple more functions that are useful in dealing with blanks.

my_field="This has a bunch of   blanks.";
my_field=COMPRESS(my_field," ");
my_field=COMPBL(my_field);

The **COMPRESS** function will remove all occurrences of the value(s) you specified. The results in this example would be **Thishasabunchofblanks.**

The **COMPBL** function removes multiple blanks. The results in this example would be
This has a bunch of blanks.

The multiple blanks have been removed and left as only one blank. The single blanks were not affected.

How many "S"s are in the character string "**From Sea to Shining Sea**"? To check, use the **COUNTC** function

s_count=COUNTC("From Sea to Shining Sea is how the line goes","S");

You'll find S_count=**3**. Did you expect **5**? What happens if you typed the following instead?

s_count=COUNTC("From Sea to Shining Sea is how the line goes ","s");

You'll discover than s_count is **2** in this case. Why? **COUNTC** is case-sensitive. Of course, there is a way to get around this.

You could use:
s_count=COUNTC("From Sea to Shining Sea is how the line goes","sS");

However, that could get cumbersome. A better method is to use the optional modifier "i" to indicate you want case to be ignored. The modifier can be upper or lower case.

Of course, some of you are wondering why you would ever do this. Perhaps you have a product code and the occurrence of certain letters in the code means something. Chances are, you'll want to apply this function to a field rather than to specific text. The statement would then be:

s_count=COUNTC(my_variable,"S",i);

This will find all the "S" and "s" for each record.

CHARACTER STRING MATCHING
There are some other presentations at SUGI 29 which cover this topic. Look in the proceedings for functions such as **RXPARSE** and **RXMATCH**. Just as the name implies, these functions are used for comparing two or more strings of data.

CURRENCY CONVERSION
How many Swedish Krona are there in an Irish pound?

swedish=EUROCURR(1, 'IEP','SEK');

The answer is **11.892252566**. Note this isn't the exchange rate which changes all the time. Think of it as converting pennies to dollars in the United States – 100 pennies is always 1 dollar. There are 26 currencies for those of you needing to convert European monies.

DATE AND TIME
Date and time offer a different type of conversion. There have been many SUGI papers over the years that cover just this category of functions. I'm covering just a few.

First, a SAS date is the number of days since January 1, 1960.

Date	**SAS date value**
January 1, 1960	0
January 2, 1960	1
December 31, 1960	365
May 10, 2004	16201

A SAS time is the number of seconds since midnight. A SAS datetime value is the number of seconds since midnight January 1, 1960.

Some people get confused between functions and formats when it comes to dates and times. The formats control the look of the field. For example, a format of mmddyy8. will print the SAS date 16201 as 05/10/04 while the format weekdate. writes Monday, May 10, 2004 but the value is still stored as 16201.

If your date field is from a non-SAS source, the first thing you need to do is convert it to a SAS date. There are many ways to do this and the correct way depends upon how your date is stored by default. Here is one example. Your data comes in with separate year, month and day fields. You'd like it stored as a SAS date so you can do calculations such as determining the number of days between a start and end date. Your code would look like:

```
start_date=MDY(start_month,start_day,start_year);
end_date=MDY(end_month,end_day,end_year);
```

Now, how do you find the number of days between start and end? If you just need the number of days, then all you do is subtract..

```
days=end_date  - start_date;
```

If your start date is **May 1, 2004** and your end date **is May 10, 2004** then you'll have SAS date values of **16192** and **16201**. The result is **9**. If any of you think it should be 10, then you wish to count the start date as well and you'll need to add 1 to the calculation.

The **TODAY** function is very useful so a report always uses the current day no matter when it is run. For example, you want to show invoices and how many days past due they are.

```
days_past_due=TODAY() –invoice_date;
```

Notice that this function does not really have an argument but you do need to have the parentheses as though there were arguments.

One of the things many people want to change is the way the run date appears on the report. They don't like the date and time as it appears by default in the upper right corner of the output. Let's assume you want it to appear in the title instead. The following code will create a macro variable .

```
data _null_;
  today_date=TODAY();
CALL SYMPUT('rundate',PUT(today_date),mmddyy8.);
run;
```

This will then let you use the macro variable such as
title "This report was run &rundate";

Of course, you need to know something about macros. Notice that this also used the **PUT** function which is in the Special category we'll cover later. **CALL SYMPUT** is also one the CALL features under the MACRO category.

INTCK is used to return the number of interval boundaries. For example, this paper was due for the proceedings on **Feb 2, 2004**. How many months have passed since then? The example uses X as the due date and Y as **May 10, 2004**.

```
num_months= INTCK ('months',x,y);
```

The answer is **3** – March, April and May.
If I requested "years" instead, the answer would be 0 since no year boundary has been crossed.

This function also has options which allows even greater flexibility such as calculating biennual (2-year) periods or starting your calculations based on a specific day of the week.

What would happen if you weren't paying attention and wrote your statement as shown below?

```
num_months= INTCK ('months',y,x);
```

Instead of 3 you will get **-3** since the “from” occurs after the “to” so you have to go backwords and cover May, April, and March boundaries.

Some people confuse **INTCK** with **INTNX** since they are so close in spelling. **INTNX** is used to increment a date by a specified amount. I have probably used this the most in running monthly reports when I wanted data for the previous month. For example, if I run a monthly report today for last calendar month, I could use:

lastmonth= INTNX ('month',today(),-1);

The -1 says to back up one month so the result based on the conference presentation day of **May 10** will be **April 1, 2004**. By default, the value will be the first day of the month. You can also specify it the middle or the end of the month as follows.

firstday= INTNX ('month',x,-1,'B'); *result: April 1, 2004;
middle= INTNX ('month',x,-1,'M'); *result: April 15, 2004;
lastday= INTNX ('month',x,-1,'E'); *result: April 30, 2004;

Notice that the end of April is the 30 and not the 31st. The function adjusts for the number of days in the month.

Sometimes you need to extract a portion of the date. For example, you have your SAS dates but you just need to show the year.

my_year=YEAR(my_date);

If you have a datetime field, you can also split out the date and time separately as shown.

my_date=DATEPART(my_datetime_field);
my_time=TIMEPART(my_datetime_field);

DBCS (Double-byte character set)
This is another category that may not be as commonly used so I won’t cover it except to say it exists.

DESCRIPTIVE STATISTICS
Many of the descriptive functions have the same name as values you can obtain in procedures. The difference is that procedures work across rows. Functions work across multiple fields within a row.

A simple example is to find the minimum value among several fields.
my_min=MIN(field1,field2,field3);

What if one of those is missing? You get the smallest NON-missing value.

Do you need to know if there are any missing values? Just use:

missing_values=MISSING(field1,field2,field3);

This function simply returns 0 if there aren't any or 1 if there are missing values. If you need to know how many missing values you have then use

num_missing=NMISS(field1,field2,field3);

You can also find the number of non-missing values with

non_missing=N(field1,field2,field3);

Another useful function is **SUM**.

total=SUM(field1,field2,field3);

Is anyone wondering why you wouldn’t just use

```
total=field1+field2+field3;
```

First, how do you want missing values handled? The **SUM** function returns the sum of non-missing values. If you choose addition, you will get a missing value for the result if any of the fields are missing. Which one is appropriate depends upon your needs. However, there is an advantage to use the **SUM** function even if you want the results to be missing. If you have more than a couple fields, you can often use shortcuts in writing the field names

```
if MISSING(of field1-field20)=0 then total=SUM(of field1-field20);
else total=.;
```

If your fields are not numbered sequentially but are stored in the program data vector together then you can use:

```
total=SUM(of fielda--zfield);
```

Just make sure you remember the “of” and the double dashes or your code will run but you won’t get your intended results. Mean is another function where the function will calculate differently than the writing out the formula if you have missing values.

A function that is new in SAS 9 is **LARGEST**. If **A=1, B=3,C=5** and **D=7** what do you think LARGE2 will be?

```
large2=LARGEST(2,A,B,C,D);
```

This function returns the X largest value where X is the first argument. In this example, the result is **5**. What happens if you asked for the 5th largest but only had 4 values? You get a missing value as the result. You also get a missing value if there are missing values in your fields and you don’t have enough non-missing values to determine the result. There is also a **SMALLEST** function.

EXTERNAL FILES

There are a number of functions for working with external files. The one that many people like to use is **FILEEXIST**. This lets you know if an external file already exists. Many times, if the file already exists, you may not want to write over it. The function returns a value of 1 if the files exists and a value of 0 otherwise. A simple example is:

```
 x= FILEEXIST('a:\functions.doc');
```

Notice that the file I checked for was a Word document and not a SAS dataset. How you write the name of the file will vary by operating system.

The **PATHNAME** function returns the physical name of a SAS data library or external file. For example,

```
 libname mylib 'f:\dcassidy\sugi';
   data one;
   length mypath $50;
   mypath= PATHNAME('mylib');
 run;
```

The length statement was included because the default length for the **PATHNAME** function is 200 bytes which is can present a challenge when you want to display the field.

EXTERNAL ROUTINES

These can be used when working with things such as custom DLL's. Some of these functions are also for use in IML. Once again , probably not for the typical beginner but useful to know they exist if you do need to work with external processes.

FINANCIAL

There are several functions to do financial calculations such as depreciation or net present value in a variety of ways. The one that might be useful to even non-financial people is **MORT.** If you bought a **$200,000** house and wanted to pay for it over **30** years with a current interest rate of **5%**, you'd learn the monthly payments would be **$1,073.64**.

```
A=200000;         *initial amount;
```

```
P=.;            *payment;
R=.05/12;       *interest rate expressed as a fraction;
N=360;          * number of months;

payment= MORT (a,p,r,n);
```

What if you are trying to figure out the amount you can pay for a house if you know the interest rate and the maximum payment you want to make? **MORT** is a function where there are always the same number of required arguments. However, you leave one of the arguments as missing and that is the one that will be calculated.

```
A=.;            *initial amount;
P=750;          *payment;
R=.05/12;       *interest rate expressed as a fraction;
N=360;          *number of months;

amount= MORT(a,p,r,n);
```

HYPERBOLIC

There are 3 functions in this category, You can compute hyperbolic sine, hyperbolic cosine and hyperbolic tangent.

MACRO

This category currently has 3 call routines and 2 functions. You've seen one use of **CALL SYMPUT** in the section on date functions. **RESOLVE** can be used with **CALL SYMPUT** to use the macro variable you just created in the same datastep. There are many SUGI papers on macros that can help you delve into these functions.

MATHEMATICAL

There are many mathematical functions ranging from common functions such as absolute value to the lesser known calculations such as the bessel function.

Those of you who do basic math might need the absolute value as in:

```
x=ABS(fielda);
```

Do you need to know the remainder?

```
remainder=MOD(field,10);
```

If field=104, the remainder will be 4. Note that you used a comma to separate your arguments and not a / to represent division.

MODZ is a function to avoid unexpected floating-point results. There are several functions with a corresponding ‘z’ function. If your data goes out to several decimal places, you might want to investigate these functions. The results can be dependent upon the operating system.

Many people have encountered an issue when they used code as simple as:

```
y=a/b;
```

How could something so simple cause problems? Easy, if you did not know you had records with a value of 0 for B and did not adjust the code accordingly, you get a message like the following telling you where the divison by zero occurred.

```
NOTE: Division by zero detected at line 79 column 7.
```

In most cases, it shows up way too many times in the log. The old way to get rid of these messages was to use the following logic:

```
   if b ne 0 then y=a/b;
else y=.;
```

The much simpler way is

y=DIVIDE(a,b);

This happens to be a function that is new in SAS 9.1 rather than new to 9.0 so be sure you aren't using an older version when you try it.

PROBABILITY

Several functions are available to return the probability from a variety of distributions.

QUANTILE

You can also return the quantile for several distributions.

RANDOM NUMBER

Did you think there was only one way to get a random number? There are actually many ways depending upon the distribution that you assume your data has. Of course, some will argue that there is no such thing as a random number. A commonly used function is **RANUNI** that returns a random variate from a uniform distribution.

z=RANUNI(seed);

But what to use for your seed? Some people will use the time from the computer when the function executes. Just remember to make sure your seed is an integer. It is often a wise practice to write this to the log so you know what the seed was. If you use the same seed again, do you get the same value? If you create two variables within the same datastep with the same seed, you do not get the same results. However, if you run the datastep again and use the same seed, you do.

SAS FILE I/O

In the External Files category, I mentioned the **FILEEXIST** function. The are similar functions for catalogs and data library members. **CEXIST** will check for the catalog or catalog entry while **EXIST** will check for the data library member. You need to understand the difference so you use the correct function.

format_exists= CEXIST ("work.formats");
data_exists= EXIST("work.my_dataset");
program_exists= CEXIST("mylib.mycat.somecode.program");

Notice that one **CEXIST** function has a 2-part name and the other uses a 4-part. Catalogs are identified by 1 or 2-part names while catalog entries such as programs are identified as 3 or 4-part names. Remember, you don't need to specify "work" if it is a temporary catalog or dataset you are using.

A note of caution on these functions if you use them within a macro such as in the macro function **%SYSFUNC**. While you need the quotes in the above examples, you will not use the quotes within **%SYSFUNC**. This is just one example that fits the rule that says "There is always an exception to every rule."

There are many other functions that can be used with SAS files. Some of these functions such as **LIBNAME** and **LIBREF** have names that correspond to statements. You want to use the function if you are checking that something exists or if you need to return a value to be used later. You want to use the statement to set the value in the first place.

SPECIAL

This category has all the functions that do not fit in any other category. They cover a wide variety of things such as putting the system to "sleep" or changing memory values. The latter is something that I think should only be done by very experienced users who also understand the operating system and how it handles memory. Although I've used SAS for over 20 years, I don't feel qualified to use **PEEK** and **POKE** to play with memory values! However, **SLEEP** is a very useful function especially for anyone who needs to communicate with the operating system. While the **SLEEP** function itself is very easy to use, the situations in which you might need it are more complicated. Look up SUGI papers about DDE to see an example of using **SLEEP**.

A special function which is used by many is the **LAG** function. However, it is a contender for the most misunderstood function. Many people think **LAG** is the way to get a value from the previous record. In many cases, it does work that way. However, what it truly does is to return a value from a queue. If your processing does not write every record to the queue, you may get unexpected results. So if you have a need to get data from a previous record, please read up on the details of the **LAG** function before trying to do it. You may also want to understand the RETAIN statement since sometimes it is more appropriate. Once again, there are entire SUGI papers devoted to the topic.

Another function in the special category is **PUT**. It was shown earlier in conjunction with an example in the date and time category.

STATE AND ZIP CODE

Quick – AL is the postal code for which state? If you did not know immediately and had to run through all the states starting with "A", you might find some of the state and zip code functions very helpful.

```
mystate=STNAME("AL");
mystate=STNAMEL("AL");
```

The first statement above code will convert the 2-character postal code to the state name of **ALABAMA**. The second statement will convert it to **Alabama**. You just need to decide if you want all uppercase or mixed case. There are many other related functions such as **FIPNAME** that will convert your Federal Information Processing Standards (FIPS) codes to state names and vice versa. Anyone who has created maps with SAS/GRAPH is familiar with FIPS codes because the map datasets use them to identify states and counties.

```
state=ZIPNAME(43017);
```

The above gives you the state for the zipcode value. However, zip codes are always being added so they may not be found. If you have an invalid zipcode, you will receive the following message in the log:

```
NOTE: Invalid argument to function ZIPNAME at line 59 column 7.
```

(Of course, your line and column number will be specific to your program!)

TRIGONOMETRIC

Do you need to calculate the mileage between two points? If so, you'll be using trigonometric functions such as **COS** to do it. **COS** gives you the cosine. There are also functions for sine, tangent, and the inverse of these three. I'll leave it up to you to find the formula for such things as calculating mileage since there are several formulas depending upon what assumptions you wish to make.

TRUNCATION

The most common form of truncation is probably rounding. Most of us were taught rounding in our early school years so you would think it was very simple. It is as long as you pay attention to what you really want to do as in the following case.

A poster on SAS-L wondered why the following wasn't giving him the correct result when he wanted to round to 2 decimal places.

```
x=ROUND(myfield,2);
```

If myfield=**145.334** the result is **146**. Why? Because he specified to round to the "nearest 2" rather than 2 decimals. To round to 2 decimal places, you specify

```
x=ROUND(myfield,.01);
```

The result will be **146.33**.

What is the result when myfield=**145.345**? If you said **145.35** you are right. If you said **145.34** you are also right. What? Different countries and even different industries following different rules when it comes to handling the "5" in rounding. SAS has now provided you with a function for each method. **ROUND** will give you **145.35** while **ROUNDE** will give you **145.34**. Some will refer to the **ROUNDE** method as the European method.

Rounding isn't the only form of truncation. If your value is **123.45**, you'll get the following results with different functions.

ORIGINAL VALUE	**123.45**	**123**	**-123.45**
STATEMENT			
C=CEIL(MYFIELD);	**124**	**123**	**-123**
F=FLOOR(MYFIELD);	**123**	**123**	**-124**
I=INT(MYFIELD)	**123**	**123**	**-123**

CEIL returns the smallest integer equal to or greater than your value. **FLOOR** returns the smallest integer equal to or less than your value. **INT** returns the integer portion of your value. At first glance, someone might think that **FLOOR** and **INT** would always return the same value. However, if you look at the example with the negative number, you'll see that INT matches **FLOOR** for positive values but it matches **CEIL** for negative values.

If you are working with data that has a number of decimal places, you'll want to check out the differences between these functions and their related zero fuzz functions.

VARIABLE CONTROL

This category has 3 CALL routines that deal with such things as assigning variable labels/names or linking variables. Such functions might be used in macros or arrays where you are creating or using variable names or labels based on other logic in your code. Once again, this is a category of functions that probably won't be used by everyone but is essential to those who do have a need for it.

VARIABLE INFORMATION

These functions return information about variables such as whether a given variable is character or numeric, whether it has a format assigned or if it had an informat. One of the uses that I might have for these functions also requires an introductory knowledge of macros. If I want my title to use the label of a variable in my dataset, I could do the following:

```
data _null_;
  set mydataset;
  CALL SYMPUT("mylabel",VLABEL(myvar));
run;
  title "Report for: &mylabel”;
```

This uses the **VLABEL** function to get the label and it uses **CALL SYMPUT** to create a macro variable with that value. Of course, in real code, I would probably have a bit more complicated logic to determine which variable would be reported.

WEB TOOLS

The last category provides functions for encoding or decoding strings for use in HTML. These functions are **HTMLDECODE**, **HTMLENCODE**, **URLDECODE** and **URLENCODE**. These functions are fairly easy to use but you need to understand some things about HTML and the special characters to understand the results. If you need to work with HTML and URL, you should read the specifics in SAS Help.

LEARNING MORE

There are many ways to learn more about functions.

- HELP in the SAS System provides information and examples. How you access help at your site may vary so contact your SAS administrator if you need assistance.
- You can search the SUGI proceedings for the last several years. You can access these at http://support.sas.com/usergroups/sugi/proceedings/index.html. A user also created a more flexible search engine but I could not connect at the time I wrote this paper so I did not include the URL since it may have changed. Some of the regional groups also have their proceedings online.
- You can also post questions to SAS-L, an electronic newsgroup. There are several ways to post questions as well as search the archives of SAS-L. To learn more, search the SUGI proceedings first to find the option that is best for your situation.

SUMMARY

There are many functions included with the SAS System that eliminate the need to write complex code. Some are very easy to use and others are more complicated. The key things to understand in using any function are:

-required versus optional arguments
-order of the arguments
-handling of missing values
-when you need to quote arguments
-using text as an argument versus a field name as an argument
-whether the arguments are case-sensitive
-differences between similar functions and/or statements
-defaults for the results (length defaults or value defaults)
-combining functions versus separate statements
-exceptions to anything that I've stated in this paper!

When you are learning a new function, you should test the function on a small sample of data to make sure you understand how it works before you use it with a large amount of data. You should ALWAYS read the log to see if something unexpected happened. This is true even when you have lots of experience with a function because there may be an unknown data issue. If you are an experienced user, you will also want to review functions in a new version because there may be a simpler way to accomplish the same task.

REFERENCES

SAS Institute Inc. 2003. SAS OnlineDoc® 9.1. SAS: Cary, NC. http://support.sas.com/91doc/docMainpage.jsp
The above reference is for SAS 9.1. Documentation should also be available at your site.

TRADEMARKS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. I can be contacted at:

Deb Cassidy
Debc_sugstuff@usa.com
(Note: Some people have other e-mail addresses for me but I would prefer questions to be directed to this address.)