

Fifteen Functions to Supercharge Your SAS® Code

Joshua M. Horstman, Nested Loop Consulting, Indianapolis, IN

ABSTRACT

The number of functions included in SAS® software has exploded in recent versions, but many of the most amazing and useful functions remain relatively unknown. This paper will discuss such functions and provide examples of their use. Both new and experienced SAS programmers should find something new to add to their toolboxes.

INTRODUCTION

As of version 9.4, the Base SAS product includes a comprehensive library of over 500 functions and call routines. Together, these provide an immense wealth of functionality to the SAS programmer. The functions cover a variety of topics including string manipulation, dates and times, math and statistics, type conversions, information about data sets and variables, logic and program control, and much more.

This paper will describe just a tiny fraction of those functions, with a focus on functions the author has found useful in a broad range of situations. The functions here are grouped into three categories: string manipulation functions, logic and program control functions, and functions for handling missing values.

STRING MANIPULATION FUNCTIONS

1. CATX – ENHANCED STRING CONCATENATION

There are many ways to concatenate strings in SAS. Before SAS 9, most programmers relied on the double vertical bar operator (||) to perform this task. This operator was often used in combination with the TRIM and LEFT functions to eliminate leading and trailing blanks from the strings being concatenated.

SAS 9 brought us the CAT family of functions, which as of 2017 includes CAT, CATT, CATS, CATX, and CATQ. The CATX function is of particular interest because it provides some very convenient extra functionality. Specifically, it will strip both leading and trailing blanks from each of the strings to be concatenated and then insert a specified delimiter between each string during the concatenation process.

The delimiter must be supplied as the first argument to the CATX function. All remaining arguments are used to specify the strings to be concatenated. As a bonus, the CATX function will ignore any empty strings. This avoids the unwanted duplicate delimiters that can occur when concatenating missing strings using other methods.

Syntax: CATX(*delimiter*, *item-1* <, ... *item-n*>)

FUNCTION CALL	VALUE RETURNED
CATX(' ', ' ', 'Lions', 'Tigers', 'Bears')	Lions, Tigers, Bears
CATX(' - ', ' a ', ' ', 'b ')	a - b

Table 1. CATX Examples

There are some additional advantages of using the CAT family of functions instead of the concatenation operator. These include the ability to use OF notation for variable lists and automatic type conversion without extraneous notes to the SAS log. More information about the CATX function and the other CAT functions can be found in Horstman (2016).

2. ANYALPHA – DETECTING THE PRESENCE OF ALPHABETIC CHARACTERS

The ANYALPHA function can be used to search a string for any alphabetic character – that is, any lowercase or uppercase letter. When one or more alphabetic characters are found, the ANYALPHA function returns the position in the string of the first such character. If none are found, a zero is returned.

The ANYALPHA function accepts up to two parameters. The first parameter is the string to be searched. The second parameter is optional and specifies where in the string to begin searching as well as the direction in which the search should proceed. The absolute value of this parameter indicates the starting position in the string and its sign indicates the direction (positive for right, negative for left).

Syntax: ANYALPHA(*string* <,*start*>)

If one is concerned only with detecting the presence of alphabetic characters but not with their actual positions in the string, the ANYALPHA function can easily be used in conditional logic without the need to compare its return value to another value. Because SAS considers any nonzero value to be logically false, one can simply write something like this:

```
if anyalpha(myvar) then <do something>;
```

The ANYALPHA function may be particularly useful in situations where numeric values are being stored in a character variable. In such cases, one might use the function to verify that no alphabetic characters have inadvertently been included in a variable the programming logic expects to contain only nonalphabetic characters.

FUNCTION CALL	VALUE RETURNED
ANYALPHA ("123ABC")	4
ANYALPHA ("123456")	0
ANYALPHA ("R2D2" , 2)	3
ANYALPHA ("C3PO" , -2)	1

Table 2. ANYALPHA Examples

In addition to the ANYALPHA function, SAS provides an extensive collection of “ANY” functions that can be used to search for various categories of characters within a string. These include ANYDIGIT, ANYLOWER, ANYUPPER, ANYALNUM (any alphanumeric character), and several others.

3. NOTALPHA – DETECTING THE ABSENCE OF ALPHABETIC CHARACTERS

The NOTALPHA function is just the opposite of the ANYALPHA function. Instead of searching the string for any alphabetic character, the NOTALPHA function searches for anything that is not an alphabetic character. It operates in a manner similar to that of the ANYALPHA function. It returns the position in the string of the first nonalphabetic character found, or a zero if there are no such characters. Also, like the ANYALPHA function, the NOTALPHA function can accept an optional second parameter which controls the starting position and direction of the search.

Syntax: NOTALPHA(*string* <,*start*>)

FUNCTION CALL	VALUE RETURNED
NOTALPHA ("123ABC")	1
NOTALPHA ("ABCDEF")	0
NOTALPHA ("R2D2" , 3)	4
NOTALPHA ("C3PO" , -3)	2

Table 3. NOTALPHA Examples

Just like the ANYALPHA function is part of a larger family of “ANY” functions, there is also an entire family of “NOT” functions like NOTALPHA. Other functions include NOTDIGIT, NOTSPACE, NOTPUNCT, and many more.

4. TRANSLATE – REPLACING CHARACTERS IN A STRING

The TRANSLATE function is an extremely useful function that can replace specific characters in a string with other characters. The syntax essentially boils down to providing the function with pairs of characters, where each pair consists of a target character and a replacement character. The function then operates on a character-by-character basis, replacing any occurrence of a target character in the source string with its corresponding replacement character.

Syntax: TRANSLATE(*source*, *to-1*, *from-1* <, ...*to-n*, *from-n*>)

There are three required arguments. The first is the source string. The second and third arguments are the replacement and target characters, in that order. The second and third arguments can be a single character or a string of multiple characters. Note that if multiple characters are specified, they are handled on a pairwise basis. For example, the first target character specified will be replaced with the first replacement character specified, the second target character will be replaced with the second replacement character, and so on.

You may also specify additional pairs of replacement and target characters in additional optional arguments. The function behaves the same whether the character pairs are spread out across separate pairs of arguments or combined into fewer arguments. If there are more target characters than replacement characters, the additional target characters are replaced with blanks. However, if there are more replacement characters than target characters, the extra replacement characters are simply ignored.

FUNCTION CALL	VALUE RETURNED
TRANSLATE ("ABCDEF" , "D" , "A")	DBCDEF
TRANSLATE ("ABCDEF" , "DEF" , "ABC")	DEFDEF
TRANSLATE ("ABCDEF" , "D" , "A" , "EF" , "BC")	DEFDEF
TRANSLATE ("ABCDEF" , "DE" , "ABC")	DE DEF
TRANSLATE ("ABCDEF" , "DEF" , "AB")	DECDEF

Table 4. TRANSLATE Examples

5. TRANWRD – REPLACING WORDS IN A STRING

Sometimes you need to replace an entire group of characters in a string with another group. The TRANSLATE function is not suitable for this task because it only looks at individual characters, not groups of characters. For replacing groups of characters, use the TRANWRD function instead.

Syntax: TRANWRD(*source*, *target*, *replacement*)

The TRANWRD function accepts exactly three arguments: a source string, a target, and a replacement. The target and replacement can each be a single character or a group of characters, either expressed as a constant or contained within a variable or other expression. Observe that the target comes before the replacement, which is exactly opposite of the unintuitive order used in the TRANSLATE function. Note also that if the replacement string is empty, the TRANWRD function will replace the target with a single space.

FUNCTION CALL	VALUE RETURNED
TRANWRD ("Ham and Eggs", "Ham", "Bacon")	Bacon and Eggs

Table 5. TRANWRD Examples

6. SCAN – PARSING WORDS FROM A STRING

The SCAN function is another incredibly useful tool. SCAN will parse a string using the delimiter(s) of your choice and allow you to extract an individual portion based on its ordinal position. For example, if the string consists of words separated by spaces, you could use SCAN to obtain the first word, the last word, or any word in between.

Syntax: SCAN(*string*, *count* <, *character-list* <, *modifier*>>)

Only the first two parameters are required. The first is the string to be parsed, and the second is a number indicating which word you want (e.g. 1 for the first word, 2 for the second word, etc.). If you specify a negative number for the second argument, SCAN will count words from right to left instead of left to right (e.g. -1 for the last word, -2 for the penultimate word, etc.)

By default, the SCAN function treats spaces and a variety of punctuation marks and other special characters as delimiters. However, with the optional third argument, you can take complete control of which characters are treated as delimiters.

The SCAN function also has an optional fourth argument which can accept various modifiers that change the behavior of the function. There are modifiers that alter the list of delimiters, modifiers that control case sensitivity, modifiers that determine how multiple consecutive delimiters are handled, and many more. An exhaustive listing of the numerous modifiers available is beyond the scope of this paper. Refer to the SAS 9.4 Functions and CALL Routines reference manual for more information (SAS 2016).

FUNCTION CALL	VALUE RETURNED
SCAN ("Bacon and Eggs", 1)	Bacon
SCAN ("Bacon and Eggs", -1)	Eggs
SCAN ("17:D4:98:C3:05:5B", 3, ":")	98

Table 6. SCAN Examples

LOGIC AND PROGRAM CONTROL FUNCTIONS

7-8. IFC/IFN – RETURN A VALUE BASED ON A CONDITION

The IFC and IFN functions provide a convenient way to package conditional logic right into a function call. The function accepts a logical expression as its first parameter. If the logical expression evaluates to true, it returns the value of the second argument. If the logical expression evaluates to false, it returns the value of the third argument. If the logical expression is missing, it can return the value of the optional fourth argument.

Syntax: IFC(logical-expression, value-returned-when-true, value-returned-when-false
<, value-returned-when-missing>)

Syntax: IFN(logical-expression, value-returned-when-true, value-returned-when-false
<, value-returned-when-missing>)

The difference between IFC and IFN is the type of data on which they operate. IFC returns a character value and expects the second, third, and fourth arguments to be character values. IFN, on the other hand, returns a numeric value and thus expects numeric arguments.

One reason IFC and IFN are so useful is because they often make it possible to reduce complex conditional logic to a single statement. For example, in a variable assignment, the IFC or IFN function can be placed on the right-hand side of the equals side to allow for the value being assigned to be determined by some bit of conditional logic. This usage can be extended further by nesting calls to IFC or IFN within one another. For a more thorough discussion of these concepts, refer to Horstman (2017).

FUNCTION CALL	VALUE RETURNED
IFN (5>3 , 1 , 2)	1
IFC (6>9 , "A" , "B")	B
IFC (1>5 , "X" , IFC (1>0 , "Y" , "Z"))	Y

Table 7. IFC and IFN Examples

9-10. WHICHC/WHICHN – SEARCH A LIST OF ARGUMENTS

The WHICHC and WHICHN functions are used to search through a list of arguments and return the index of the first one that matches a given reference value. The reference value is supplied as the first argument and can be followed by any number of additional arguments.

Syntax: WHICHC(string, value-1 <, value-2, ...>)

Syntax: WHICHN(string, value-1 <, value-2, ...>)

The index begins counting with the second argument, so if the second argument matches the value given in the first argument, then the function returns a 1. Similarly, if the third argument matches the value given in the first argument, then the function returns a 2. If no matching argument is found, the function returns a zero.

As we saw previously with IFC and IFN, the difference between WHICHC and WHICHN is the type of data on which they operate. WHICHC expects the reference value and all subsequent arguments to be character data, while WHICHN works only with numeric data. In both cases, the index value returned is numeric.

The examples shown below use constant numeric and character string values. In most applications, however, these functions would be called using variables or other complex expressions as some or all of the arguments. The WHICHC and WHICHN functions were not added to the SAS software until version 9.2.

FUNCTION CALL	VALUE RETURNED
<code>whichn(10*10,31,100,365)</code>	2
<code>whichc("SFO","OAK","SJC","SFO")</code>	3
<code>whichc("APPLE","ORANGE","BANANA")</code>	0

Table 8. WHICHC and WHICHN Examples

11-12. CHOOSEC/CHOOSEN – SELECT FROM A LIST OF ARGUMENTS

The CHOOSEC and CHOOSEN functions can be thought of as functionally opposite to the WHICHC and WHICHN functions just discussed. Rather than searching an argument list and returning the index of a matching value, they accept an index and return the corresponding argument from the list.

Syntax: `CHOOSEC(index-expression, selection-1 <, ...selection-n>)`

Syntax: `CHOOSEN(index-expression, selection-1 <, ...selection-n>)`

The first argument is the index value. Any number of subsequent arguments can be supplied and will constitute the list of arguments from which the indexed value will be returned. As with WHICHC and WHICHN, the index starts counting with the second argument to the function. Thus, if a 1 is passed as the first argument to the function, the function will return the value of its second argument, and so on.

If the index value is negative, the function will count backwards from the end of the argument list. If the index value is larger than the number of arguments in the list, the function will return a missing value and an “Invalid argument” note will be written to the SAS log.

The difference between CHOOSEC and CHOOSEN is the type of data on which they operate. CHOOSEC selects from among character arguments and will return a character value. In contrast, CHOOSEN expects numeric arguments and will return a numeric value. The index value is always numeric for either function.

FUNCTION CALL	VALUE RETURNED
<code>choosen(3,1966,1987,1993,1995,2001)</code>	1993
<code>choosec(2,"TOS","TNG","DS9","VOY","ENT")</code>	TNG
<code>choosec(-1,"TOS","TNG","DS9","VOY","ENT")</code>	ENT

Table 9. CHOOSEC and CHOOSEN Examples

FUNCTIONS FOR HANDLING MISSING VALUES

13-14. CMISS/NMISS – COUNT HOW MANY VALUES ARE MISSING

Data validation is an important part of the programming process. In many cases, this process requires checking for missing data. The CMISS and NMISS functions provide a quick way to simultaneously test multiple variables for missingness.

Syntax: CMISS(argument-1 <, argument-2,...>)

Syntax: NMISS(argument-1 <, argument-2,...>)

Either function can accept any number of parameters. The function will return the number of parameters that evaluate to missing. If no parameters are missing, a value of zero will be returned.

As we have seen with many previous pairs of functions, the difference between CMISS and NMISS is the type of data on which they operate. NMISS will convert all arguments to numeric data before assessing missingness. CMISS, on the other hand, will accept both character and numeric data and will perform no conversions.

In many cases, one does not wish to know how many values are missing but simply wishes to confirm that none are missing before proceeding. In such a situation, the CMISS or NMISS function can easily be used in conditional logic without the need to compare its return value to another value. Because SAS considers any nonzero value to be logically false, one can simply write something like this

```
if not nmiss(var1,var2,var3) then <do something with var1, var2, and var3>;
```

FUNCTION CALL	VALUE RETURNED
nmiss(1,2,3,4,5)	0
nmiss(1,2,.,4,.)	2
nmiss(1,"2",3)	0
nmiss(1,"2",".")	1
cmiss("KIRK","SPOCK",".","")	1
cmiss("NCC-",1701,"-A")	0

Table 10. CMISS and NMISS Examples

15. COALESCE/COALESCEC – RETURN THE FIRST NONMISSING VALUE

There may be times you want to perform some processing based on the value of a certain variable, but only if it is not missing. If it is missing, you may wish to use a different value as an alternative. The COALESCE and COALESCEC functions offer a simple way to implement such logic.

Each function can take any number of arguments and will simply return the first one that is not missing. If all the arguments are missing, then of course a missing value will be returned.

Syntax: COALESCE(argument-1<..., argument-n>)

Syntax: COALESCEC(argument-1<..., argument-n>)

The difference between COALESCE and COALESCEC is the type of data on which they operate. However, unlike the last several pairs of functions we looked at which used an “N” to designate a numeric function and a “C” to indicate a function for character data, the numeric function in this pair does not use an “N.” The numeric version of the function, which is simply called COALESCE, accepts only numeric arguments and returns a numeric value. The COALESCEC function is used with character arguments and returns a character value.

Suppose, for example, we wish to print a phone directory. Suppose further that we want to print only cell phone numbers, but we will print the business phone number of anyone for whom we have no cell phone number, and we will print the home phone number in those cases where we have neither of the other numbers available. We might use something like this:

```
phonenum = coalescec(cell_phone,work_phone,home_phone);
```

FUNCTION CALL	VALUE RETURNED
coalesce(1,2,3)	1
coalesce(.,.,3)	3
coalescec("","XYZ","ABC")	XYZ

Table 11. COALESCE and COALESCEC Examples

CONCLUSION

SAS software offers many useful functions that can save programming time by performing commonly needed tasks. The functions discussed in the paper are just the beginning, but they are a great start. Those wishing to increase their SAS programming expertise would do well to familiarize themselves with these and many of the other functions. A toolbox filled with highly versatile tools like this will equip the SAS programmer to skillfully tackle challenging programming situations.

REFERENCES

- Horstman, Joshua M. “Let the CAT Out of the Bag: String Concatenation in SAS® 9.” SAS Global Forum 2016, paper 11666-2016.
<http://support.sas.com/resources/papers/proceedings16/11666-2016.pdf>
- Horstman, Joshua M. “Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS® Code.” SAS Global Forum 2017, paper 326-2017.
<http://support.sas.com/resources/papers/proceedings17/0326-2017.pdf>
- SAS Institute, Inc. *SAS® 9.4 Functions and CALL Routines: Reference, Fifth Edition*. Cary, NC: SAS Institute Inc., 2016.
<http://support.sas.com/documentation/cdl/en/lefunctionsref/69762/PDF/default/lefunctionsref.pdf>

CONTACT INFORMATION

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

Joshua M. Horstman
 Nested Loop Consulting
 317-721-1009
josh@nestedloopconsulting.com