Functional Functions

Gary M McQuown, Data and Analytic Solutions Inc., Fairfax, VA
Dorothy E. Brown, Independent Consultant, Matthews, NC

Abstract:

Functions are an important aspect of data step programming that are often overlooked and under utilized. Not only can functions be used to resolve a data step dilemma; they can be mixed and matched to create efficient and precise code. The arrival of V8 includes a number of new functions, making it even more difficult to stay up to date. The following prose and examples cover many of the newly introduced functions as well as some unusual methods of using some old favorites.

Introduction:

SAS functions are among the most basic and commonly used data step tools. A SAS function performs a computation or system manipulation on arguments and returns a value. Most functions use arguments supplied by the user, but a few obtain their arguments from the operating environment. In base SAS software, you can use SAS functions in DATA step programming statements, in a WHERE expression, in macro language statements, in PROC REPORT, and in Structured Query Language (SQL). Some statistical procedures also use SAS functions. With V8, we have over fifty official new functions and some minor enhancements to a few of our old favorites. The following is a collection of documentation and code from various sources (mostly from SAS or SAS-L) intended to explain and promote interest in these functions.

Many of the functions listed are not actually "new". Most have been around on an undocumented experimental basis since late in the V6 series or emulate SCL function behavior. Regardless of their lineage or timing, they are worth learning more about.

Enhancements to PUT SCAN and QUOTE

Two of the most commonly used functions are PUT and SCAN. PUT returns a value using a specified format. It is often used to convert numeric value as a character value. With V8, we have the option of specifying the alignment of the character value returned in addition to its format. This saves us the chore of aligning the value in an additional step. In many ways, that is what functions are all about: making code more efficient, concise and convenient.

- **PUT**(*source*, *format.*)

Returns a value using a specified format

The new alignment specifications for PUT are: -L for left alignment, -C for center alignment and –R for right alignment.

```
Example:
text = "Where does it go";
put text $50. -L ;
put text $50. -C ;
put text $50. -R ;
results =
Where does it go
            Where does it go
                        Where does it go
```

- **SCAN**(*argument,n*<, *delimiters*>)

Selects a given word from a character expression

The SCAN function is used to select a given work from a character expression. It has been modified to accept a negative value directing it to read character segments starting from the end of the character string rather than from the beginning.

```
Example:
destination = “New Orleans LA”;
state = scan(destination, -1);
results =  LA .
```

- **QUOTE**(*argument*)

Adds double quotation marks to a character value

The third function to receive modifications is the QUOTE function. The QUOTE function places double quotes around a character value and now retains all trailing blanks. Previous versions of this function removed trailing blanks.

```
Example:
x='George''s';
y=quote(x);
put y;
result = "George's"
```

The following SQL code uses quote to create a list of values.

```
PROC SQL noprint;
select quote( trim( string) ) into :list separated by ', ' from
data_set;
run;
```

New Mathematical and Probability Functions

The introduction of the new mathematical functions makes it easier to compute factorials, permutation, and combinations.

- **COMB**(*n, r*)

Computes the number of combinations of **n** elements taken **r** at a time and returns a value

- **CONSTANT**(*constant*<, *parameter*>)

Computes some machine and mathematical constants and returns a value

CONSTANT allows you to pass certain mathematical values, some of which may be platform or environment specific.

```
Example:
pi = constant ('PI');
```

The following is a list of the constants that can be returned.

Constant 'Argument'

The natural base 'E'
Euler constant 'EULER'
Pi 'PI'
Exact integer 'EXACTINT' <,nbytes>
The largest double-precision number 'BIG'
The log with respect to base of BIG 'LOGBIG' <,base>
The square root of BIG 'SQRTBIG'
The smallest double-precision number 'SMALL'
The log with respect to base of SMALL 'LOGSMALL' <,base>
The square root of SMALL 'SQRTSMALL'
Machine precision constant 'MACEPS'
The log with respect to base of MACEPS 'LOGMACEPS' <,base>
The square root of MACEPS 'SQRTMACEPS'

- **DEVIANCE**(*distribution*, *variable*, *shape-parameter(s)* < ,*[EPSIV]*>)

Computes the deviance and returns a value

- **FACT**(*n*)

Computes a factorial and returns a value

- **PERM**(*n*<,*r*>)

Computes the number of permutations of **n** items taken **r** at a time and returns a value

- **PROBBNRM**(*x, y, r*)

Computes a probability from the bivariate normal distribution and returns a value

- **PROBMC**(*distribution, q, prob, df, nparms*<, *parameters*>)

Computes a probability or a quintile from various distributions for multiple comparisons of means, and returns a value

Character-String Matching Functions

The following RX functions and CALL routines provide character-string matching functionality. That is, they enable you to search for (and, optionally, to replace) patterns or characters in a string.

- **CALL RXCHANGE** (*rx, times, old-string*<, *new-string*>);

Changes one or more substrings that match a pattern

- **CALL RXFREE** (*rx*);

Frees memory allocated by other regular expression (RX) functions and CALL routines

- **CALL RXSUBSTR** (*rx, string, position, length, score*);

Finds the position, length, and score of a substring that matches a pattern

- *position*=**RXMATCH** (*rx, string*)

Finds the beginning of a substring that matches a pattern and returns a value

- *rx*=**RXPARSE**(*pattern-expression*)

Parses a pattern and returns a value

The ability to parse strings can be useful in many different ways. While some use these tools to explore, clean and modify their data, others use the same tools to automate their processes. The following example shows how a SAS log or the output from PROC CONTENTS can be processed to determine directory paths.

```
Parsing a SAS Entry Name from a Line of Text
By Jack Hamilton on SAS-L

data _null_;

length result $35.;
drop rx string;
retain rx;

if _n_ = 1 then
   rx = rxparse('` <:> <$n "." $n "." $n ".program"> <:> to =2');
infile cards end=end;
input string $char80.;
call rxchange(rx, 2, string, result);
put result=;
if end then
   call rxfree(rx);
run;

data:
0jd#abc.def.xyz.program6834efghijklmn.op.qr.program2633
123defghijklmn.op.qr.program2633
hijklmn.op.qr.programsandmore
```

```
results:
abc.def.xyz.program
hijklmn.op.qr.program
hijklmn.op.qr.program
```

Variable Information Functions

The largest category of new functions supplies variable information. The information returned ranges from whether or not the variable is in an array, is character or numeric, to the name of the format or informat associated with the variable and its label. This category is actually two complementary sets of functions that perform the same task but with different arguments. The first set begins with the letter V and requires a variable name or array reference as its argument. The second also begins with a V, but ends with an X and requires a character string as the argument. For each of the V-X functions, SAS evaluates the argument to determine the variable name.

- **VARRAY** (*name*)

Returns a value that indicates whether the specified name is an array

- **VARRAYX** (*expression*)

Returns a value that indicates whether the value of the specified argument is an array

- **VINARRAY** (*var*)

Returns a value that indicates whether the specified variable is a member of an array

- **VINARRAYX** (*expression*)

Returns a value that indicates whether the value of the specified argument is a member of an array

VARRAY, VARRAYX, VINARRAY, VINARRAYX, VTYPE, and VTYPEX make determinations and return a specific value. VARRAY and VARRAYX determine if the specified name or expression is the name of an array. VINARRAY and VINARRAYX are used to determine if a specified name or expression is a member of an array. Both VARRAY and VARRAYX return a 1 if the argument is the name of an array and a 0 if it is not, but VARRAYX requires an expression rather than a name. The same is true for VINARRAY and VINARAYX, which determine if the name or expression is a member of an array.

```
 Example:
  an_array=varray(name);
  an_array=varrayx(expression(x)) ;
  in_array=vinarray(name);
  in_array=vinarrayx(expression(x));
```

- **VTYPE** (*var*)

Returns the type (character or numeric) of the specified variable

- **VTYPEX** (*expression*)

Returns the type (character or numeric) for the value of the specified argument

VTYPE and VTYPEX are a little different in that they return the letter N if the variable is numeric and the letter C if it is a character variable.

```
Example:
v_type=vtype(name);
v_type=vtypex(expression(x));

if vtype(&varname )='N' then do;
      /* code for numeric processing */
   end;
   else do;
      /* code for character processing */
end;
```

The VLABEL, VLENGTH and VNAME function pairs are especially helpful tools when processing arrays. VNAME and VNAMEX return the name of the requested variable and VLABEL and VLABELX return any label associated with it. VLENGTH and VLENGTHX return the length at processing time.

- **VLABEL** (*var*)

Returns the label that is associated with the specified variable

- **VLABELX** (*expression*)

Returns the variable label for the value of a specified argument

- **VLENGTH** (*var*)

Returns the compile-time (allocated) size of the specified variable

- **VLENGTHX** (*expression*)

Returns the compile-time (allocated) size for the value of the specified argument

- **VNAME** (*var*)

Returns the name of the specified variable

- **VNAMEX** (*expression*)

Validates the value of the specified argument as a variable name

```
data a;
length x1-x3 $8;
label x1 = "first"
      x2 = "second"
      x3 = "third" ;
array x(3) x1-x3;
 x1 = 'abc';
 x2 = 'cde';
 x3 = '';
v_name=vname(x(1));
v_length=vlength(x(1));
x_length=length(x1) ;
v_label=vlabel(x(3));
put v_name= v_label= v_length= x_length=;
run;

results:
v_name=x1 v_label=third v_length=8 x_length=3
```

The remaining sixteen V functions return information about the format or informats associated with a variable. Because a separate function exists for formats and another for informats as well as the name or expression argument discussed earlier, we now have two sets of pared functions. Their tasks are to return the format or informat, the format or informat name, the format or informat length and the format or informat decimal value for the given variable. Those that are associated with formats begin with VFORMAT and while those associated with informats begin with VINFORMAT. As with the other V Functions, those ending with an X must receive an expression while those that do not end in an X must receive a name or array reference.

- **VFORMAT** (*var*)

Returns the format that is associated with the specified variable

- **VFORMATD** (*var*)

Returns the format decimal value that is associated with the specified variable

- **VFORMATDX** (*expression*)

Returns the format decimal value that is associated with the value of the specified argument

- **VFORMATN** (*var*)

Returns the format name that is associated with the specified variable

- **VFORMATNX** (*expression*)

Returns the format name that is associated with the value of the specified argument

- **VFORMATW** (*var*)

Returns the format width that is associated with the specified variable

- **VFORMATWX** (*expression*)

Returns the format width that is associated with the value of the specified argument

- **VFORMATX** (*expression*)

Returns the format that is associated with the value of the specified argument

- **VINFORMAT** (*var*)

Returns the informat that is associated with the specified variable

- **VINFORMATD** (*var*)

Returns the informat decimal value that is associated with the specified variable

- **VINFORMATDX** (*expression*)

Returns the informat decimal value that is associated with the value of the specified argument

- **VINFORMATN** (*var*)

Returns the informat name that is associated with the specified variable

- **VINFORMATW** (*var*)

Returns the informat width that is associated with the specified variable

- **VINFORMATNX** (*expression*)

Returns the informat name that is associated with the value of the specified argument

- **VINFORMATWX** (*expression*)

Returns the informat width that is associated with the value of the specified argument

- **VINFORMATX** (*expression*)

Returns the informat that is associated with the value of the specified argument

The following example illustrates how V functions VYPE and VFORMAT can be used to write a formatted value to another variable, while retaining the original format.

```
if vtype(name)='N' then do;
   new_var=putn(name,vformat(name));
end;
else do;
   new_var= putc(name,vformat(name));
end;
```

New Date and Time Functions

A common topic among SAS programmers is the different ways to determine and or define duration: roughly the amount of time passing between two points in time. In the past, most solutions involved the use of INTCK or INTNX, which have their strong and weak points. The new functions DATDIF and YRDIF should make the task of determining time duration easier.

- **DATDIF**(*sdate*,*edate*,*basis*)

Returns the number of days between two dates

- **JULDATE7**(*date*)

Returns a seven-digit Julian date from a SAS date value

- **YRDIF**(*sdate*,*edate*,*basis*)

Returns the difference in years between two dates

Both DATDIF and YRDIF use the arguments for start date, end date and basis. Start and End dates are very straightforward, but defining "basis" is more complicated. As per the on-line docs:

Basis identifies a character constant or variable that describes how SAS calculates the date difference. The following character strings are valid:

'30/360'
specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year. Alias: '360'
Tip: If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366. Alias: 'Actual'

'ACT/360'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

```
Example:
data _null;
   startdate = '11jul71'd;
   enddate = '11jul01'd;
   actday = datdif(startdate, enddate, 'act/act');
   days360 = datdif(startdate, enddate, '30/360');
   months = yrdif(startdate, enddate, 'act/act')*12;
   yr30 = yrdif(startdate, enddate, '30/360');
   yract = yrdif(startdate, enddate, 'act/act');
   yra_360 = yrdif(startdate, enddate, 'act/360');
   yra_365 = yrdif(startdate, enddate, 'act/365');
put  actday = days360 = months = yr30 = yract =
       yra_360 = yra_365 = ;
run;

results:
   actday=10958
   days360=10800
   months=360
   yr30=30
   yract=30
   yra_360=30.438888889
   yra_365=30.021917808
```

Missing and Error Functions

- **MISSING**(*numeric-expression* | *character-expression*)

Returns a numeric result that indicates whether the argument contains a missing value

Like several of the previously mentioned functions, the MISSING function returns an affirmative indicator of 1 if a variable contains a missing value and negative indicator of 0 if the value is non-missing. It works for both a character and numeric expressions.

- *character-variable*=**IORCMSG**()

Returns a formatted error message for _IORC_

IORCMSG returns the formatted error message associated with the most recently posted IROC code. A _IORC_ message is the value of an automatic variable created when the Modify statement or the Set statement with the KEY= option is used. This return code indicates whether the retrieval for matching observation was successful. A returned value of 0 indicates a successful execution; a -1 indicates an end-of-file error; and any other value indicates a non-match occurrence.

In the following program, observations are either rewritten or added to the updated master file that contains bank accounts and current bank balance. The program queries the _IORC_ variable and returns a formatted error message if the _IORC_ value is unexpected.

```
Example:
libname bank 'SAS-data-library';

data bank.master;
   set bank.trans;
   modify bank.master key=Accountnum;
   if (_IORC_ EQ %sysrc(_SOK)) then
      do;
         balance=balance+deposit;
         replace;
      end;
else
   if (_IORC_ = %sysrc(_DSENOM)) then
      do;
         balance=deposit;
         output;
         _error_=0;
      end;
else
   do;
      errmsg=IORCMSG();
      put 'Unknown error condition:'
      errmsg;
   end;
run;
```

Web-Based Functions

- **HTMLDECODE**(*argument*)

Decodes a string containing HTML numeric character references or HTML character entity references and returns the decoded string

- **HTMLENCODE**(*argument*)

Encodes characters using HTML character entity references and returns the encoded string

- **URLDECODE**(*argument*)

Returns a string that was decoded using the URL escape syntax

- **URLENCODE**(*argument*)

Returns a string that was encoded using the URL escape syntax

```
Example:
data _null_;
  text="This string contains characters !@#$%^& that must be
encoded";
  html=
    '<a href="/cgi-bin/broker.exe?_service=
       default&_program=test.echo.sas&text=
       '!!urlencode(text) !!'">Show encoded text</a>'
  ;
  put html;
  run;
```

which produces the following valid HTML hyperlink:

```
  <a href="/cgi-
bin/broker.exe?_service=default&_program=test.echo.sas&text
=This%20string%20contains%20characters%20%21@%23%2
4%25%5E%26%20that%20must%20be%20encoded">Show
encoded text</a>
```

Urldecode() works the other way to decode these cryptic strings e.g.

```
  data _null_;
  text="%21Hello+World%21";
  text=urldecode(text);
  put text=;
  run;
```

which produces:

```
  TEXT=!Hello World!
```

Financial Functions

With SAS being used by virtually all of the major financial institutions, some financial functions were certainly in order.

- **CONVX**(*y,f,c(1), ... ,c(k)*)
Returns the convexity for an enumerated cashflow

- **CONVXP**(*A,c,n,K,k_0,y*)
Returns the convexity for a periodic cashflow stream, such as a bond

- **DUR**(*y,f,c(1), ... ,c(k)*)
Returns the modified duration for an enumerated cashflow

- **DURP**(*A,c,n,K,k_0,y*)
Returns the modified duration for a periodic cashflow stream, such as a bond

- **PVP**(*A,c,n,K,k_0,y*)
Returns the present value for a periodic cashflow stream, such as a bond

- **YIELDP**(*A,c,n,K,k_0,p*)
Returns the yield-to-maturity for a periodic cashflow stream, such as a bond

Interesting Uses of Functions

The use of functions is often limited only by the imagination, creativity and need of the programmer. The following code shows how various functions can be combined to solve dilemmas and make life easier.

- TIP 00270 from WWW.SCONSIG.COM ****/
A COMPRESS function for Macro Variables
By Peter Crawford

```
   %macro Remove__ ( STRING, REMVECHR );
    %sysfunc( compress( &string, &REMVECHR));
   %mend Remove__ ;

/*** Sample Call to Invoke ***/
%let  string__ = %remove__( "PLA" Derivative, '"');

%put &string__ ;
PLA Derivative
```

You will need to be careful in compressing quotes (single or double) - make sure you surround your preference (quotes to be compressed) with a pair of opposite quotes (ie, '"'   or "'").

- TIP 00136 from WWW.SCONSIG.COM
To Generate Nine Variables from a Nine Length Character String
By Paul Dorfman

```
    data manyvar2(drop=addr len);
      array v(10) $1;
      addr = addr(v(1));
      len  = dim(v);
      do until (eof);
        set in end=eof;
        call poke (string, addr, len);
        output;
      end;
    run;
```

About the Authors

Gary McQuown is a SAS Quality Partner with Data and Analytic Solutions, Inc. of Fairfax VA. He has previously presented at NESUG and SESUG.

Dorothy Brown is a SAS Consultant currently on contract at Sprint Communications World Headquarters in Kansas. This is her first presentation.

Author Contact

Gary McQuown
Data and Analytic Solutions, Inc.
10502 Assembly Drive, Fairfax, VA 2200
mcquown@DASconsultants.com
www.DASconsultants.com

Dorothy Brown
819-201 Cameron Village Drive, Matthews, NC 28105

Bibliography:

William F. Heffner, "DATA Step in Version 7: What's New?" SUGI 23 Proceedings

Denise J Moorman and Deanna Warner, "Updating Data Using the Modify Statement and the KEY=Option" SAS Observations

Mike Rhoads, "Hidden Nuggets in Version 8: New Informats, Formats and Functions" SUGI 23 Proceedings

SAS Institute Inc., Changes and Enhancements to Base SAS Software Release V8.1, Cary, NC: SAS Institute,

Trademark Information