PharmaSUG 2020 - Paper AP-251

How to Achieve More with Less Code

Timothy J. Harrington, Navitas Data Sciences, Inc.;

ABSTRACT

One of the goals of a SAS® Programmer should be to achieve a required result with the minimal amount of code. The two reasons for "code complexity" are: one, too many lines of code, and two: code which is unduly difficult to decipher, for example a large number of nested operations, pairs of parentheses, operators, and symbols packed closely together. This paper describes methods for reducing the amount and complexity of SAS code, and for avoiding repetition of code. Included are examples of SAS code using v9.2 or later functions such as IFN, IFC, CHOOSE, WHICH, and LAG. This content and discussion is primarily intended for beginner and intermediate SAS users.

INTRODUCTION

This paper is primarily intended for beginner and intermediate SAS programmers and demonstrates methods of making the best use of the SAS system and minimizing code complexity to achieve a given result. A sequence of examples are shown of coding situations where the amount and structure of SAS source code is simplified and made easier to interpret without the need for large amounts of comment text. (Comments should explain the purpose of the code rather than its logical function). Such code minimization also often results in reduced compile and execution run time and lower resource consumption. Note: Some of the examples discussed may include functions which are not backwardly compatible with earlier versions of SAS.

MAKING SELECTIONS

MAKING SELECTIONS USING THE SCAN FUNCTION

The two main methods of conditional execution are the IF THEN ELSE construct and the SELECT construct. The following example sets a flag SEXIND based on the applicability of certain clinical tests regarding patient gender. Most tests apply to both male and female patients so SEXIND is set to 'B'. However, a few tests apply only to female patients, where SEXIND is set to 'F' and a few tests apply only to male patients, where SEXIND is set to 'M'.

The two SAS constructs which could be used to set SEXIND based on the test (LBTEST) are (1) the IF THEN ELSE construct and (2) the SELECT clause:

```
if lbtest in ('Pregnancy Test','Mammogram') then do;
  sexind='F';
end;
else if lbtest='Sperm Count' then do;
  sexind='M';
end;
else if lbtest ne ' ' then do;
  sexind='B';
end;
else do;
  sexind=' ';
end;

select (lbtest);
  when ('Pregnancy Test','Mammogram') sexind='F';
  when ('Sperm Count') sexind='M';
  when (' ') sexind=' ';
  otherwise sexind='B';
end;
```

Another method of selecting a category indication is to use the SCAN function, which returns the *n*th 'word' of a text string using one or more optional delimiters, the syntax of a SCAN function call is:

```
scan(<text string>,n,<delimiter list>);
```

If no delimiters are specified, the space is used as the 'word' separator. The following example returns the second word, 'SAS', in the text 'Senior SAS Programmer':

```
scan("Senior SAS Programmer",2);
```

Note: if n is negative the 'words' are counted from the end instead of the beginning of the string.

A simple example for selecting a value is setting a Yes/No flag, in this case BFLAG is set to 'Y' when VALN=BLVALN or to 'N' otherwise. (Note: This also sets BLFLAG to 'Y' if both VALN and BLVALN are missing):

```
blflag=scan('N Y',1+(valn=blvaln));
```

To set a numeric flag to 0 or 1 the Boolean expression alone will suffice:

```
blflagn=(valn=blvaln);
```

Using the above example of assigning SEXIND, the values 'B', 'F', or 'M' are based on the LBTEST text. A consideration should be made for the possibility of LBTEST being blank (or any other 'null' value). For this reason a check has to be performed to ensure SEXIND is not set to 'B' when LBTEST is missing. This is where the delimiter option helps.

```
sexind=scan('!B! !F!M',1+(lbtest=' ')+2*(lbtest in
  ('Pregnancy Test','Mammogram'))+3*(lbtest='Sperm Count'),'!');
```

The first argument is the string of values which can apply to SEXIND, each of these, including a blank space, are separated by a delimiter '!'. Which of the four values is assigned is determined by the result of the second argument. If LBTEST is 'Pregnancy Test' or 'Mammogram' the argument passed to the SCAN function becomes 1+0+2+0=3, which selects the third letter in the first argument, which is the letter 'F'. Similarly, if LBTEST is 'Sperm Count' the second argument becomes 1+0+0+3=4, which selects the fourth letter, 'M'. If LBTEST is not any of these gender specific texts and is not blank the second argument resolves to 1+0+0+0=1, which selects the 'B'. When LBTEST is blank the second argument becomes 1+1+0+0=2, which selects the second delimited item, the space. Note: The Boolean expressions must be mutually exclusive, for example checking for LBTEST ne ' ' instead of when blank would result in this condition but also the ‘F’ and ‘M’ conditions being true at the same time. Also, setting all literal strings to upper case and using the UPCASE function in situations like this is good programming practice.

Another situation where the delimiter is needed is where a text result can have two or more words. In the following example the '!' is being used as the delimiter because there are multiple words (delimited by spaces) in the first two possible values for PRESP, selected based on the numeric value ANSWER:

```
ptresp=scan('No Response!Do not know!No!Yes',1+(answer=0)+2*(answer=1)+
  3*(answer=2),'!');
```

CONDITIONAL SELECTION USING THE IFC AND IFN FUNCTIONS

Another method of using less code and achieving faster run time performance is to use IFC. The IFC function returns a character result based on a specified condition and whether that condition is true or false:

```
<character result>=IFC(<condition>, <value if condition is true>,
   <value if condition is false>, <value if the condition resolves to
   missing>);
```

The first argument is the condition, the second argument is the result value to return when the condition is true, the third argument is the result to return when the condition is false, and the optional fourth argument is the value to return when the result of the condition is missing (which is not always possible).

The following code assigns SEXIND 'F' if LBTEST is 'Pregnancy test' or 'Mammogram', otherwise SEXAPP is set to 'B':

```
sexind=ifc(lbtest in ('Pregnancy Test','Mammogram'),'F','B');
```

However, if LBTEST is 'Sperm Count' or is blank, SEXAPP will still be 'B'. To handle these situations IFC functions are nested:

```
sexind=ifc(lbtest=' ',' ',
       ifc(lbtest in ('Pregnancy Test','Mammogram'),'F',
       ifc(lbtest='Sperm Count','M','B')));
```

Here, the outermost IFC call checks to see if LBTEST is blank, is so SEXIND is returned as the 'true' value, which is a space. If LBTEST is not blank the third 'false' argument is returned, this is itself the result of the next level of IFC call, which is the result of checking to see if LBTEST is 'Pregnancy Test' or 'Mammogram', this returns 'F' when true, otherwise the returned value is the result of the innermost IFC call, which is the LBTEST='Sperm Count' test. If this is true 'M' is passed back, otherwise 'B'.

For returning a numeric instead of character result there is a corresponding function IFN, which has the same syntax. The following example sets the numeric flag BLFLAGN to 1 when the variables VALN and BVALN are equal, or to 0 when they are not equal and one or both are non-missing. If both VALN and BLVALN are missing the NMISS function returns 2 and BLFLAGN is set to missing:

```
blflagn=ifn(nmiss(bvaln,valn)=2,.,ifn(valn=blvaln,1,0));
```

SELECTING VALUES AND RANGES USING THE CHOOSE AND WHICH FUNCTIONS

There are four more useful SAS functions, CHOOSE and WHICH (numeric based), and CHOOSEC and WHICHC (character based). CHOOSEC returns a character result based on a numeric input argument.

```
CHOOSEC(<numeric value or expression>,<string1>,<string2>,.....<stringn>);
```

If the first argument resolves to 1 the function returns the value in *string1*, if the first argument resolves to 2 the function returns the value in *string2*, and so on. The SEXIND determination discussed above, including if LBTEST is missing, could be made using this code:

```
sexind=choosec(1+(lbtest=' ')+
   2*(lbtest in ('Pregnancy Test','Mammogram'))+
   3*(lbtest='Sperm Count'),'B',' ','F','M');
```

The '1+' at the start of the first argument ensures that when all conditions are false a zero is not returned. The second argument of the WHICHC call (in this case the ‘B’) must therefore be the 'fall through' situation, when LBTEST is neither 'Pregnancy Test', 'Mammogram', 'Sperm Count' or ' '. Once again all conditions must be mutually exclusive.

Another example is selecting the text for an adverse event (AE) severity grade based on the numeric value of the grade, AEGRDN, in the code below:

```
aesevtxt=choosec(sum(aegrdn,1),
 'NO AE','MILD','MODERATE','SEVERE','LIFE THREATENING','FATAL');
```

Note: If the value of the first argument does not correspond any of the other arguments a blank value is returned. Since 0 is not a position of an argument, when AEGRDN is zero (or missing) the SUM function is used to add 1 (SUM treats missing values as zero), ie: when there is no AE grade, the first argument 'NO AE' is selected, otherwise when AEGRDN is 1, the second argument 'MILD' is selected, and so on.

Just as with IFC and IFN there is a corresponding CHOOSEN function that returns a selected numeric value from a series of numeric arguments, instead of character strings.

The WHICHN function returns the argument number (position) where a specified value is in a series of numeric arguments

```
WHICHN(<numeric value or expression>,<value 1>,<value 2>,...<valuen>);
```

Here, if the first argument is equal to *value1* the function returns a 1, if the first argument is equal to *value 2* the function returns a 2 and so on. If the first argument does not match any of the other argument values, the function returns zero. If there is a match to two or more of the list of values, the position of the first match is returned. Since ranges are mutually exclusive this function can be used to assign a range identifier based on where a result is in relation to specified boundary values.

An example is assigning a toxicity grade based on the concentration of a substance in a PK sample. In this example the lowest detectable amount of the analyte is contained in the numeric variable LTR, the measured amount (or concentration) of the analyte is RESULT. If RESULT is not LTR but less than LORANG the toxicity grade, TOXGRADE is 0. If RESULT is greater than or equal to LORANG but less than HIRANG TOXGRADE is 1. If RESULT is greater or equal to HIRANG but less than HIRANG*2 TOXGRADE is 2. If RESULT is greater or equal to HIRANG*2 but less than HIRANG*3 TOXGRADE is 3. If RESULT is greater than or equal to HIRANG*3 TOXGRADE is 4.

Coding this using an IF THEN ELSE structure would be, lengthy, complex, and difficult to read and follow. However, each range set is mutually exclusive, so a simpler solution is to just use Boolean expressions such as:

```
toxgrade=(ltr <= result<= hirang) +
  2*(hirang < result <= hirang*1.5) +
  3*(hirang*1.5 < result <= hirang*3) +
  4*(result > hirang*3);
```

Here the toxicity grade is assigned according to which range statement is true (TOXGRADE is zero if RESULT<LTR), however, even this code is difficult to follow and intermediate ranges have to be typed twice, increasing the risk of programmer error. The next solution involves more lines of code but is much simpler top follow, it uses the SORTN function to order the range boundary values and then a WHICHN call to determine where a result value is in relation to the boundaries.

In the example below the range boundary values are stored in an array RANGES as:

```
array ranges{*} x1 x2 x3 x4 x5;
```

where x1 is the lowest and x4 is the highest, x5 will be used to store the result being tested.

First, RESULTN is checked to see if it is a valid result, not missing, negative, or zero, if this is the case TOXGRADE is set to missing, otherwise a fuzz factor is added to RESULTN and this value of RESULTN is copied to x5. This fuzz factor is needed to handle the rare but possible case where the source RESULTN is exactly equal to any of the boundary values (x1-x4). FUZZ must be a number of less than the precision of the data, for example if the precision of RESULTN is three decimal places, FUZZ would have to be 0.0001 or less. In the case where RESULTN is being tested for being greater than the lower boundary and less than or equal to the upper boundary (as in the above example using HIRANG) fuzz should be subtracted instead of added.

The CALL SORTN function now sorts the array elements in ascending numerical order. (The lowest boundary value, X1 in this case, could be missing since SORTN handles missing values) In this example x5 holds the RESULTN value and by including x5 in the sort this value is sequenced with the other, boundary values x1 through x4. Now x5 is between its closest boundaries, so its position in the array now corresponds to the applicable range.

The WHICHN function has RESULTN as its first argument so this value is searched for in the array RANGES. Since x5 is equal to RESULTN the position of x5 is returned. Subtracting 1 gives the corresponding toxicity grade, 0 through 4.

```
if resultn>0 then do;
  fuzz=0.0000001;
  resultn=resultn-fuzz;
  x5=resultn;
  call sortn(of ranges{*});
  toxgrade=whichn(resultn,of ranges{*})-1;
end;
else do;
  toxgrade=.;
end;
```

The next example is used for assigning a 'high/low' flag to a lab result ('VL'='Very Low', 'L'='Low', 'H' is 'High', 'VH' is 'Very High' and blank is in 'normal' range.

Here the character string HL_FLAG is used to store this flag result, this is performed by using the SCAN function on the WHICHN numeric result. There is no subtraction from this result since all values must correspond to a position in the first argument to the SCAN function. The '!' is the delimiter for the SCAN, since HL_FLAG is to be blank when RESULTN is 'normal'.

```
length hl_flag $5;
array ranges {*} x1 x2 x3 x4 x5;

if resultn>0 then do;
  fuzz=0.0000001;
  resultn=resultn-fuzz;
  x5=resultn;
  call sortn(of ranges{*});
  hl_flag=scan('!VL!L! !H!VH!',whichn(resultn,of ranges{*}),'!');
end;
else do;
  hl_flag='ERROR';
end;
```

ACCUMULATING GROUP COUNTS AND TOTALS

The following code is reading a dataset, EX01, where each observation contains a dose amount given to a specified patient. The DATA step accumulates the number of non-missing and non-zero doses and the cumulative total amount of dose given to each patient. The variables which are to contain the number of doses and the dose amount are N_DOSES and CUM_DOSE respectively. The RETAIN statement prevents these two variables from being reset to missing at the start of each DATA step iteration. The observations have been sorted by the patient ID USUBJID and the date and time, as a SAS internal datetime EXSTDT, the dose was taken. The variables N_DOSES and CUM_DOSE are both initialized to zero at the first observation for a given patient. If EXDOSE is not missing and greater than zero (in SAS missing is considered as 'less than' negative numbers) N_DOSES is incremented and the dose amount, EXDOSE, is added to CUM_DOSE. If the observation just processed is the last for that patient, it is written to the output data set PATDOSES.

```
data patdoses(keep=usubjid n_doses cum_dose);
  set ex01;
  by usubjid exstdt;
  retain n_doses cum_dose;
  if first.usubjid then do;
    n_doses=0;
    cum_dose=0;
  end;
  if exdose>0 then do;
    n_doses=n_doses+1;
    cum_dose=cum_dose+exdose;
  end;
  if last.usubjid;
run;
```

A more efficient method is to use Boolean expressions instead of IF THEN constructs.

```
data patdoses(keep=usubjid n_doses cum_dose);
  set ex01;
  by usubjid exstdtc;
  retain n_doses cum_dose 0;
  n_doses=n_doses*(first.usubjid=0)+(exdose>0);
  cum_dose=cum_dose*(first.usubjid=0)+sum(exdose,0);
  if last.usubjid;
run;
```

In this code the zero in the RETAIN statement initializes N_DOSES and CUM_DOSE to zero. At the start of a new patient FIRST.USUBJID is 1, hence (FIRST.USUBJID=0) is false (0), multiplying this by N_DOSES sets N_DOSES back to 0 for each new USUBJID. In subsequent observations for the same patient (FIRST>USUBJID=0) is true (1), hence the current value of N_DOSES is multiplied by 1, which leaves it unchanged. Now if there is a non-missing and non-zero dose (EXDOSE is greater than zero) the expression (EXDOSE>0) is true (1), hence N_DOSES has 1 added to it (is incremented). If EXDOSE is zero or missing (or a negative number), (EXDOSE>0) is false (0) so nothing is added to the existing value of N_DOSES. The same logic applies to CUM_DOSE except here the requirement is to add the current dose amount to the total dose accumulated for the patient so far. The SUM function considers a missing value as zero, so if EXDOSE is missing or zero the amount added to CUM_DOSE is 0. If EXDOSE is non-missing and non-zero the SUM function adds zero to it, not affecting the value of EXDOSE, which is now added to CUM_DOSE. (A negative EXDOSE will result in subtraction taking place). A point to note here is the 0 in the RETAIN statement, this is needed to initialize N_DOSES and CUM_DOSES to zero, if this zero were omitted the initial values of N_DOSES and CUM_DOSE would both be missing and all of their calculated values for the whole dataset would be missing because the SAS system propagates missing values in arithmetic expressions. Similar reasoning applies to the SUM function in the calculation of CUM_DOSE. When a result of missing is generated the message 'Missing values were generated at line xx number of times yy" is written to the log file.

REFERENCING A PRIOR OBSERVATION USING THE LAG FUNCTION

There are many situations in a SAS DATA step where prior values need to be carried forward. An example is how a value changes from its baseline over a series of visits. In order to carry forward a value three things are required: (1) The observations must be sorted in the applicable order, (2) the value being carried forward has to be retained, and (3) where carrying forward is within a specified BY group (e.g. the patient ID) care must be taken not to 'roll over' values from the prior group.

In a DATA step the RETAIN statement specifies to retain the prior value of the given variable instead of setting it to missing at the start of each DATA step iteration. The example shown below carries forward

the values of a patient's measured weight from the prior two visits. To do this the observations must be sorted by patient and visit date order, variables must be defined to store (retain) the weight values for each of the preceding two visits, and these retained variables must be initialized (set to missing) each time a new patient is encountered.

This code extracts all the patient weight values from the dataset VTLS01 where there is a non-missing visit date, renames the variable holding the weight value to WTKG, and sorts in chronological order by visit date.

```
proc sort data=vtls01(keep=usubjid vsdate vstest vsstresn
  where=(upcase(vstest)='WEIGHT (KG)' and vsdate ne .))
  out=wt01(rename=(vsstresn=wtkg));
   by usubjid vsdate;
run;
```

The next step is to define two variables (WTCF and WTCF2) to hold the retained weight values. WTCHG is the change in patient weight since the prior visit and WTCH2 is the change in weight since two visits prior. Both of these variables must be initialized as missing each time a new patient is encountered (FIRST.USUBJID=1), since there are no prior values before the first visit.

If the observation is not the first for the patient WTCF is set to WTKG. WTCF2 must remain as missing at the second visit (observation) for the patient since there was no visit two weeks prior. Hence, before storing WTKG in WTCF a check is performed on WTCF to see if it is still missing, WTCF2 is only set to WTCF when WTCF is non-missing (there was a weight recorded for the patient two visits ago).

```
data wt02;
  set wt01;
  by usubjid vsdate;
  retain wtcf wtcf2 0;
  if first.usubjid then do;
    wtcf=.;
    wtcf2=.;
  end;
  else do;
    if wtcf2 ne . then do;
      wtchg2=wtkg-wtcf2;
    end;
    if wtcf ne . then do;
      wtchg=wtkg-wtcf;
      wtcf2=wtcf;
    end;
    wtcf=wtkg;
  end;
run;
```

A way to greatly simplify this process is to use the SAS provided LAGn and DIFFn functions. LAGn automatically retains the nth prior value of the variable which is its argument. The default value of n is 1, hence in this example LAG2 must be specified to retain the value of WTKG from two observations prior. DIFFn returns the difference between the current value of the variable passed as the argument and the value n observations prior. (A positive difference is an increase and a negative difference is a decrease).

```
data wt02;
  set wt01;
  by usubjid vsdate;
  if first.usubjid=0 then do;
    wtcf=lag(wtkg);
    wtchg=diff(wtkg);
  end;
  if lag2(usubjid)=usubjid then do;
    wtcf=lag2(wtkg);
    wtchg2=diff2(wtkg);
  end;
run;
```

Taking this a stage further the IFN function can be used to replace the if then else code. In the example below LAG(WTKG) and LAG2(WTKG) contain the values of WTKG for the prior visit and visit two observations prior. The IFN function is used to check for the first observation for a new patient, if the patient is not the first WTCF is set to the prior value of WTKG, otherwise, WTCF is set to missing. The IFN call for WTCF2 uses the LAG2 function to check if the patient ID two observations prior is the same as for the current observation, if it is WTCF2 is set to WTKG two observation prior, otherwise WTCF2 is set to missing. The same logic is applied to the change in weight variables WTCHG and WTCHG2 except the differences DIFF and DIFF2 are being referenced.

```
data wt02;
  set wt01;
  by usubjid vsdate;
  wtcf=ifn(first.usubjid=0,lag(wtkg),.);
  wtchg=ifn(first.usubjid=0,diff(wtkg),.);
  wtcf2=ifn(lag2(usubjid)=usubjid,lag2(wtkg),.);
  wtchg2=ifn(lag2(usubjid)=usubjid,diff2(wtkg),.);
run;
```

Note: There is no ‘LEAD’ function in SAS, which corresponds to LAG but references the next instead of the prior observation.

IDENTIFYING CHARACTER STRING CONTENT

FINDING A TYPE OF CHARACTER USING ANY AND NOT

There are many situations where there is a need to determine if a certain data type, such as a characteror numeric value, is present, or absent, in a character string. SAS provides two applicable functions, ANY and NOT.

One such example is checking to see if the character version of a date is a valid date in the required format. If the required date format is is8601dt (e.g. 2020-02-17) there should be no alphabetic characters. A way to test for alphabetic characters could be to use INDEXC:

```
if indexc(upcase(aedtc),'ABCDEFGHIJKLMNOPQRSTUVWXYZ')>0 then do;
  put 'ERROR: Invalid date ' aedtc= ;
end;
```

AEDTC is the date stored as a character string. The UPCASE function is needed to identify any lower case alphabetic characters.

A simpler method is to use the ANYALPHA function:

```
if anyalpha(aedtc)>0 then do;
  put 'ERROR: Invalid date ' aedtc= ;
end;
```

The ANYALPHA function returns the position of the first occurrence of an alphabetic character (in upper or lower case). A zero is returned if no alphabetic characters are found.

Other available related functions are:

ANYALNUM - any character which is 'a' - 'z', 'A' - 'Z', or '0' -' 9'

ANYUPPER - upper case 'A' - 'Z'

ANYLOWER - lower case 'a' - 'z'

ANYDIGIT - digits '0' - '9'

ANYSPACE – blanks

ANYPUNC - punctuation characters such as '?', '!', '/', ' .'

Each of these functions actually has two arguments, the character string being searched is the first argument, the second is the position in the string to begin the search, which defaults to 1, the start of the string. For example:

ANYPUNC(sample_id,10)

returns the position of any punctuation symbol at or after the tenth position in the string. Note: The REVERSE function can be used on the string argument to find the last occurrence, when the result must then be subtracted from the string length plus 1. (REVERSE(TRIM(<string>)) allows for trailing spaces.)

To check for the absence of particular character types NOT is used instead of ANY, for example

NOTDIGIT(sample_id)

returns the position of the first occurrence of any character which is not a digit '0'-'9', this includes spaces.

THE SUBSTRN FUNCTION

There is a ‘sister’ function of SUBSTR, SUBSTRN. When using the SUBSTR function there is often a need to make sure an attempt is not made read beyond the end of the argument string. For example, if a SAS datetime is stored in a character string EVNTDTC of length $25 usingn the is8601dt. format (e.g. '2020-02-18T11:28:00') and there is a need to extract the time part into a string HHMMSS, this code could be used:

```
hhmmss=substr(evntdtc,12);
```

This reads in all the characters after the 'T', that is the time as hh:mm:ss. The problem is an occurrence of EVNTDTC may be, for whatever reason, lacking the time part, in which case a 'Invalid second argument to SUBSTR function' ERROR or WARNING will occur. Note: This happens if an attempt is made to reference a point beyond the actual length of the string argument, not just beyond the declared length, ie: within any training blanks.

One solution is to ‘protect’ the SUBSTR call by checking the length of the string

```
if length(evntdtc)>10 then do;  /* Format is is8601dt. */
  hhmmss=substr(evntdt,12);
end;
```

However, there is a function SUBSTRN which allows for this

```
hhmmss=substrn(evntdt,12);
```

This call returns the time part if there is one, but returns blanks if there is no time part, without an ERROR or WARNING being issued.

DETERMINING HOW MANY 'WORDS' ARE IN A CHARACTER STRING

In SAS there is function COUNTW, which returns the number of 'words' in a character string. This avoids the problem of referencing beyond the end of the string. For example, the code below counts the number of adverse events (AEs) which are classified as a skin rash.

The character string AELIST contains a sequence of distinct AE texts delimited by commas. The COUNTW function returns the total number of such comma delimited 'words'. (Some AE descriptions contain multiple words separated by spaces or punctuation symbols such as a dash or a slash). The DO END loop executes that number of times, the SCAN function extracts each *i*th AE text. If the *i*th text is one of those listed in the parentheses following the IN keyword 1 is added to the accumulating total RASHES, otherwise zero is added. The SUM function is used to set rashes to zero at the start of the search, when *i* is 1. (All SAS variables when first created are set to missing).

```
do i=1 to countw(aelist,',');
  rashes=sum(rashes,0)*(i ge 2)+(scan(aelist,i,',') in ('RASH',
 'RASH MACULOPAPULAR','HERPES ZOSTER'));
end;
```

The COUNTW call prevents the SCAN function from attempting to reference a 'word' beyond the end of AELIST, which would generate an 'Invalid Second Argument to SCAN' ERROR or WARNING. This also guarantees a termination of the DO END loop. If AELIST is empty COUNTW(AELIST) will be zero and the loop will not execute. Note: The comma must be specified as a delimiter in both the COUNTW and the SCAN functions. The default delimiter for both of these functions is the space.

STRING CONCATENATION

In SAS, character strings can be joined together using '||' but this can be complex due to the presence of leading or trailing spaces. For example, to create a unique patient identifier, USUBJID, based on the character variables STUDY_ID and CENTERID, and a unique numeric patient identifier PATNUM, all linked by dashes, the following code could be used:

```
usubjid=trim(left(study_id))||'-'||trim(left(centerid))||'-'||
  left(put(patnum,8.));
```

If STUDY_ID is 'CDT1008', CENTERID is '422670', and PATNUM is 100117 USUBJID would become 'CDT1008-422670-100117'

The LEFT function is needed to remove leading spaces and the TRIM function to remove trailing spaces and the concatenation characters '-' have to be placed as literals in quotation marks and be joined using '||'.

A simpler way to do this is with the CAT (concatenation) functions CAT, CATT, CATS, and CATX.

CAT is the simplest, just joining 'as is' without removing any of the leading or trailing spaces
CATT removes the trailing spaces, the same as with the TRIM function
CATS is the equivalent of the example above because it removes both the leading and the trailing spaces
CATX performs the same function as CATS but also inserts a specified delimiter between the strings

Using the above example, the same resulting value of USUBJID is achieved with

```
CATX('-',study_id,center_id,put(putnum,8.));
```

The first argument is the concatenation delimiter, the default being one space. Any number of strings may be concatenated when using any of the CAT functions, thus significantly reducing the amount of code..

Furthermore, the CAT functions can also be used with character arrays, so a large number of text elements are joined using this one call:

```
CATX('<delimiter>',of <character array name> {*});
```

IDENTIFYING THE ATTRIBUTES OF A VARIABLE

Run time attributes of a SAS dataset column are available in the SAS VHELP and the DICTIONARY.COLUMNS libraries but SAS provides several run time 'VHELP' attribute functions, these include:

VLENGTH - the declared length of a character string, the text length plus trailing spaces

VLABEL - the label associated with the variable, returns blank if no label

VTYPE - the data type 'C' if character' or 'N' if numeric

VFORMAT - the format associated with a variable (including a user defined format)

VINFORMAT - the informat associated with a variable

VVALUE - the formatted value of the content of a variable (e.g. formatted text of a SAS internal numeric date)

VNAME - the name of the variable matching an array element number

Each of these functions has a single argument, which is the variable name, or an array element number accordingly. The returned result is character, except for VLENGTH, which is numeric. Returned character results are always in upper case, except for VLABEL, which returns the actual text of the label.

Example 1: Initializing a variable stored in a macro variable COL1 to missing when that variable could be character or numeric:

```
if vtype(&col1)='C' then do;
  col1=' ';
end;
else if vtype(&col1)='N' then do;
  col1=.;
end;
```

Example 2: Storing a date as a SAS numeric date read from a character string when the format of the date could be either of DATE9. or IS8601DA.

```
if vformat(evntdt)='DATE9.' then do;
  evntdtc=put(evntdt,date9.);
end;
else if vformat(evntdt)='IS8601DA.' then do;
  evntdtc=put(evntdt,is8601da.);
end;
```

Example 3: Returning the names of the variables which are the array elements in the RANGES array described earlier

```
do i=1 to dim(ranges);
  varname=vname(ranges{i});
  put 'Var' i varname;
end;
```

The resulting output is:

```
Var 1 X1
Var 2 X2
Var 3 X3
Var 4 X4
Var 5 X5
```

DETERMINING SAS OPTIONS AT RUNTIME

In SAS there is a GETOPTION keyword which returns the current SAS OPTIONS settings during the execution of a DATA step or PROCedure. The single argument to this function is the name of the option in quotation marks, for example, these calls return the current centering, LINESIZE and PAGESIZE settings:

```
alignmnt=getoption('center');
linlngth=getoption('linesize');
linperpg=getoption('pagesize');
```

The variable ALIGNMNT is a character string containing either 'LEFT', 'CENTER', or 'RIGHT' accordingly. The variables LINLNGTH and LINPERPG are numeric and contain the line size and number of lines per page respectively.

CONCLUSION

This paper has described several methods of reducing the amount of code needed to achieve certain results in a variety of programming situations. The aim has been to inform SAS programmers of more efficient coding methods and to make SAS code more portable and maintainable. This paper has also introduced several functions that are useful for performing certain tasks which would otherwise require more complex code and be less runtime and resource efficient.

ACKNOWLEDGMENTS

Navitas Data Sciences, Inc.

Pharmasug Paper Selection Committee

RECOMMENDED READING

- *SUGI 2012 Paper 257-2012: Leave Your Bad Code Behind: 50 Ways to Make Your SAS® Code Execute More Efficiently. William E Benjamin Jr, Owl Computer Consultancy, LLC*

  http://support.sas.com/resources/papers/proceedings12/257-2012.pdf

- *IFC and IFN Functions: Alternatives to Simple DATA Step IF-THEN-ELSE, SELECT-END Code and PROC SQL CASE Statements. Thomas E. Billings, Union Bank, San Francisco, California*

  https://www.lexjansen.com/wuss/2012/28.pdf

- *WUSS 2017 Fifteen Functions to Supercharge Your SAS®Code. Joshua M. Horstman, Nested Loop Consulting, Indianapolis, IN*

  https://www.lexjansen.com/wuss/2017/92_Final_Paper_PDF.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please feel free to contact the author at:

Timothy J. Harrington
Professional Services,
Navitas Data Sciences, Inc, 1610 medical Drive, Suite 300, Pottstown, PA 19464
610-970-2333

timothy.harrington@navitaslifesciences.com

www.navitaslifesciences.com