Advanced Techniques to Build and Manage User-Defined SAS® FORMAT Catalogs

Jack Shoemaker, Oxford Specialty Management, Norwalk, CT

SAS supplies a wealth of formats for displaying character and numeric data. When these do not suffice, you can use PROC FORMAT to create user-defined formats which are stored as members of a SAS catalog named FORMATS by default. For example, the following code fragment will create a member called LOB with an entry type of FORMATC in the catalog WORK.FORMATS.

```
proc format;
  value $lob
    'CO' = 'Commercial'
    'MC' = 'Medicare'
    'SF' = 'Self-Funded'
    ;
  run;
```

All formats defined as above will reside in the WORK.FORMATS catalog. Like all objects in the SAS WORK area, the FORMATS catalog disappears once the SAS session terminates. You can create a permanent user-defined format, that is, one which persists after the current SAS session terminates by using the LIBRARY= option to direct SAS to use a catalog other than WORK.FORMATS. For example, the following code fragment will store the catalog member LOB.FORMATC in LIBRARY.FORMATS instead of WORK.FORMATS.

```
proc format library = library;
  value $lob
    'CO' = 'Commercial'
    'MC' = 'Medicare'
    'SF' = 'Self-Funded'
    ;
  run;
```

The preceding code assumes that the aggregate storage location LIBRARY has been defined to SAS using a LIBNAME statement. In fact, you can instruct SAS to create the FORMATS catalog in any place you choose. For example, assuming the directory specified inside the single quotes exists, the following code fragment will create the format catalog in MYLIB instead of LIBRARY.

```
libname mylib '/opt/sas/mylib';

proc format library = mylib;
  value $lob
    'CO' = 'Commercial'
    'MC' = 'Medicare'
    'SF' = 'Self-Funded'
    ;
  run;
```

Maintaining and searching catalogs

When you use a format either to display data in a PROC step or as a parameter to the PUT(), PUTC(), or PUTN() functions, SAS searches the SAS-supplied formats, WORK.FORMATS, and LIBRARY.FORMATS in that order, to find the format definition. If you choose to store user-defined formats in a catalog other than WORK.FORMATS or LIBRARY.FORMATS, you must tell SAS where to look by using the FMTSEARCH= system option. For example, the following code fragment will tell SAS to look in MYLIB.FORMATS after searching WORK.FORMATS and LIBRARY.FORMATS.

```
options fmtsearch = ( mylib );
```

In fact, you may use the FMTSEARCH= system option to change the default search order. For example, the statement below will cause SAS to look in MYLIB.FORMATS before searching LIBRARY.FORMATS and WORK.FORMATS.

```
options fmtsearch =
  ( mylib library work );
```

Why would you want to do this? One good reason is for testing. Let's assume that you already have a format definition for $LOB. In LIBRARY.FORMATS. You want to modify, or enhance, this definition. But, before you commit the new $LOB definition to LIBRARY.FORMATS, you wish to test its use in an application or two. SAS stops searching for format definitions as soon as it finds one. So, using the example above, your test version of the $LOB definition will be used instead of the one already in place in LIBRARY.FORMATS. Other applications which use the formats in LIBRARY.FORMATS will not be affected by your testing and maintenance of the $LOB definition.

Except for testing purposes, it's a terrible idea to have the same format defined in more than one format catalog. But, why have a separate format catalog at all? That is, since SAS searches LIBRARY.FORMATS automatically, why not place all user-defined formats in this special catalog and save yourself the bother of coding FMTSEARCH= statements?

It's a good question which partly comes down to style. Consider how folks set up their Windoze desktop. Generally speaking there are two dominant approaches: those who like to have many documents

open simultaneously, much like piles of papers on a real desk; and, those who tend to have one or two applications open at once opening and closing files and documents as necessary. Members of the first group can't believe that members of the second group can actually wait for specific applications to open and close; members of the second group can't believe that members of the first can actually work among such clutter. Neither group is right or wrong. It's only a matter of personal style and preference.

The question of multiple format catalogs is analogous. Placing all user-defined formats in one catalog means there's only one catalog to maintain. And, if this catalog is named LIBRARY.FORMATS, SAS will search it automatically. On the other hand, many user-defined formats are specific to a subject area or project. That is, they do not have broad applicability. Placing all user-defined formats in LIBRARY.FORMATS means that all applications will search the entire catalog of user-defined formats even though many of these formats are of no interest. Another knock against the one format approach is that the format catalog tends to become quite large which poses its own set of back-up and storage allocation problems.

The good news is that SAS allows you to manage your format catalogs as you wish. You can begin by using LIBRARY.FORMATS exclusively. If you have large, project-specific formats to define and store, you may place those definitions in a separate catalog and use a FMTSEARCH= statement in the project-specific application which uses those formats. Generally speaking user-defined formats which are small or frequently used belong in LIBRARY.FORMATS. User-defined formats which are large and infrequently used are good candidates for separate format catalogs.

Finally, format catalogs need not be named FORMATS. The LIBRARY= option on PROC FORMAT accepts one-level LIBNAMES, or two-level catalog names. For example, the following code fragment will store $LOB in a format catalog called LIBRARY.MYPROJ.

```
proc format library = library.myproj;
  value $lob
    'CO' = 'Commercial'
    'MC' = 'Medicare'
    'SF' = 'Self-Funded'
    ;
  run;
```

The corresponding FMTSEARCH= system option would now refer to this two-level catalog name.

```
options fmtsearch = ( library.myproj );
```

Listing formats

Since the format catalog is just a SAS catalog as any other, you can use PROC CATALOG to interrogate, manage, and modify members. A format catalog contains two entry types: FORMATN and FORMATC. Numeric formats have an entry type of FORMATN while character formats have an entry type of FORMATC. You can display the contents of a format catalog (or any catalog for that matter) by using the CONTENTS statement as follows:

```
proc catalog c = library.formats;
  contents;
  run;
```

This will produce a listing similar to this:

```
Contents of Catalog LIBRARY.FORMATS

# Name Type    Description

1 LOB  FORMATC FORMAT:MAXLEN=2,2,11
```

You receive a slightly enhanced display if you include the STAT keyword on the CONTENTS statement. For example:

```
proc catalog c = library.formats;
  contents stat;
  run;
```

The resulting output will include information about the size of each member to the right of the description field. One caveat: you'll want to have a LINESIZE setting of at least 121 to make this render properly. Otherwise the STAT information will wrap around and confuse the display. You can also use PROC CATALOG to modify the member description which contains information about the maximum lengths of the START, END, and LABEL fields by default. For example, the following code fragment will change the description of the $LOB format to "Nicer Description".

```
proc catalog c = library.formats;
  modify lob.formatc
    ( description = 'Nicer Description' );
  run;
```

Note that you need to specify both the member name and the entry type when you invoke the MODIFY statement. Similarly, you can use the COPY statement to copy a member from one catalog to another. For example, this code fragment will copy $LOB from LIBRARY.FORMATS to MYLIB.FORMATS.

```
proc catalog c = library.formats;
  copy out = mylib.formats;
  select lob.formatc;
  run;
```

The COPY statement also includes a MOVE option which will delete the entry from the source catalog once the copy is complete. For example, this code

fragment will move $LOB from LIBRARY.FORMATS to MYLIB.FORMATS.

```
proc catalog c = library.formats;
  copy out = mylib.formats move;
  select lob.formatc;
  run;
```

Associating formats

Note that members of a format catalog are uniquely identified by the full four-level name. That is, LIBRARY.FORMATS.AGE.FORMATN and LIBRARY.FORMATS.AGE.FORMATC refer to separate member entries - one a numeric format and one a character format. This is useful if you have a numeric format which maps a continuous numeric value to set of code values. You can then define a character format of the same name which decodes the coded values. Consider the following example which maps age into four levels 1, 2, 3, and 4.

```
proc format;
  value age
    0 -< 15 = '1'
   15 -< 45 = '2'
   45 -< 65 = '3'
   65 - high = '4'
   ;
  value $age
    '1' = '1:  0 to 14'
    '2' = '2: 15 to 44'
    '3' = '3: 45 to 64'
    '4' = '4: 65 +++++'
    ;
```

You might use the first numeric format inside the PUT() function in a data step to create a one-character data-step variable called AGEGROUP; send the resulting data set through PROC SUMMARY using AGEGROUP as one of the classification variables; and, use the second character format to display the results of the reduction using PROC PRINT.

```
data agegroup;
  set in;
  length agegroup $ 1;
  agegroup = put( age, age. );
  run;

proc summary data = agegroup nway missing;
  class agegroup;
  output out = reduced;
  run;

proc print data = reduced;
  format agegroup $age.;
  run;
```

Of course you could get similar results by supplying a modified format for AGE directly in the PROC SUMMARY step. The point is that the numeric format AGE and the character format $AGE are two separate members in the format catalog. You could call the character format $AGEGRP or something similar, but by taking advantage of the uniqueness of numeric and character formats you can imply a connection between the two formats - a poor man's meta data, if you will.

Creating your own date formats

The SAS system supplies a rich set of formats to display date values. SAS stores dates and times as numbers. For dates, the date values are the number of days since 01JAN1960 which has a date value of zero. New Year's Eve 1959, 31DEC1959 has a value of -1 and 02JAN1960 has a value of 1. The day of this presentation, 06OCT1998 is 14,158. The SAS-supplied DATE9. format displays dates in ddMMMyyyy, Y2K-compliant format as shown in this paragraph.

Membership eligibility rosters often contain a pair of dates which define the spell of eligibility for the record. For example, the membership eligibility roster for a health insurer might contain and effective from and through date pair to indicate the period of eligibility for the characteristics contained on the rest of the data record. The active record will have an open, or missing, through date. Let's say that the process which extracts this information form the on-line transaction processing system converts open through dates to the special missing value .A. If we were to use the DATE9. format to display these data you might see a sequence like this.

```
THROUGH
01MAR1998
A
30SEP1998
```

Based on the discussion in the previous paragraph, you know that the A value refers to an active or open record. If you wish to have this value displayed as 'Active' instead of 'A' you may concatenate a definition for .A with the SAS-supplied DATE9. format as shown in this code fragment.

```
proc format;
  value mydate
    .A = 'Active'
   other = [DATE9.]
  ;
  run;
```

Using the user-defined MYDATE format to display the same series of dates will produce these results.

```
THROUGH
01MAR1998
```

```
Active
30SEP1998
```

You may also use format concatenation to temporarily override or enhance an existing user-defined format. For example, assume that there exists a permanent, user-defined format called $LOB defined as follows.

```
proc format library = library;
  value $lob
    'CO' = 'Commercial'
    'MC' = 'Medicare'
    'SF' = 'Self-Funded'
    other = 'Unknown'
    ;
  run;
```

Now assume that one of the OTHER values that keeps showing up is MD which is the coded value for Medicaid. Eventually this value will makes its way into the permanent format library. In the meantime, you would like to display MD as 'Medicaid' rather than 'Unknown'. You can accomplish this by concatenating the definition for MD with the extant definition for $LOB as shown in this code fragment.

```
proc format;
  value $mylob
    'MD' = 'Medicaid'
   other = [$LOB12.]
    ;
  run;
```

You may also use format concatenation to modify an existing definition. For example, what if SF does not mean 'Self-Funded' but rather means 'Single Family'? You can correct this mistake as shown in the code fragment below.

```
proc format;
  value $mylob
    'SF' = 'Single Family'
   other = [$LOB12.]
    ;
  run;
```

This does not result in an OVERLAPPING RANGE error message because there really aren't any overlapping ranges. The user-defined format $MYLOB associates the coded value 'SF' with the literal phrase 'Singe Family'. All other values use the $LOB format. That the $LOB format also contains a mapping for 'SF' matters not.

Building CNTLIN data sets

The examples presented so far use the simple four-item $LOB format which is easy enough to code and maintain by hand. Often maps and translators are much larger and difficult to maintain by hand. Fortunately you can build a user-defined format from a SAS data set known as the CNTLIN data set. The CNTLIN data set uses specially named fields to build the user-defined format. There are about a score of these fields, fortunately only four are required: FMTNAME, START, TYPE, and LABEL. These fields contain the format name, the start value, the format type, and the label value respectively. If you have built a SAS data set called CNTLIN which contains these specially defined names, you load the format using PROC FORMAT as follows.

```
proc format cntlin = cntlin;
```

You are free to use whatever name you like for the CNTLIN data set; however, sticking with CNTLIN as the data set name provides a simple method for documenting what's going on. If you'd like to see all the specially named fields recognized by PROC FORMAT, use the CNTLOUT= option to create a CNTLOUT data set. Then run a PROC CONTENTS or PROC PRINT on the CNTLOUT data set. That exercise is left to the reader. The focus here is to build CNTLIN data sets from existing SAS data sets.

Assume that the two-character LOB codes shown in the examples above are aggregations of a more refined concept known as PRODUCT. Furthermore, assume that you have a SAS data set called PRODUCT which has two fields - PRODUCT and LOB which maps PRODUCT to LOB. The code fragment shown below creates a CNTLIN data set which will create a user-defined format called PRODMAP to map PRODUCT into LOB aggregations.

```
data cntlin(
  keep = fmtname type hlo start label );
  retain fmtname 'PRODMAP' type 'C';
  set product end = lastrec;
  start = product;
  label = lob;
  output;
  if lastrec then do;
    hlo = 'O';
    label = 'Unknown';
    output;
  end;
  run;
```

Note that this data set has an additional field beyond the required four called HLO. This field is used to indicate one of the special range values HIGH, LOW, or OTHER. A value of O in this field indicates the OTHER range value. Since we are creating a single user-define format from the PRODUCT data set, the values for FMTNAME and TYPE will be constant for the entire data set. These fields are given the values of PRODMAP and C, respectively, on the RETAIN statement. The value of C for TYPE means that PRODMAP is a character format. The only other valid value is N which indicates a numeric format. The values for START and LABEL are set by way

of simple assignment statements. Alternatively, you could use the RENAME= data set option on the SET statement as shown here.

```
set product( rename =
  ( product = start lob = label ) )
  end = lastrec;
```

The use of the RENAME= option versus the assignment statement is largely a matter of personal style. Many would argue that the RENAME= method is confusing and hard to understand. Others would counter that the assignment statements are a waste given that nothing is happening other than renaming of the variables.

The END= option on the SET statement creates a temporary data-step variable which has a value of 0, or false, for each observation in the incoming data set save the last observation where the variable takes on a value of 1, or true. In this example we have called this temporary variable LASTREC although any valid SAS variable name would do. Having this variable available allows us to test for the end of the PRODUCT data set when we want to generate an assignment for the special range OTHER. The assignment of O to the special variable HLO indicates that this assignment has the special range OTHER.

Creating multiple formats

The preceding CNTLIN data step serves as a general model for converting an existing SAS data set into a CNTLIN= data set for use by PROC FORMAT. You can also produce multiple user-defined formats from a single data set. Suppose you have a SAS data set called ZIPINFO which contains three fields: ZIP Code, state code, and FIPS state-county code. You would like to have one format which maps ZIP code to state code and a second format which maps ZIP code to FIPS state-county code. One strategy would be to follow the example above and create two CNTLIN data sets. Alternatively, you could create one CNTLIN data set with both format definitions as follows.

```
data cntlin(
  keep = fmtname type start label );
  retain type 'C';
  set zipinfo end = lastrec;
  start = zipcode;
  fmtname = 'ZIPST';
  label = statecd;
  output;
  fmtname = 'ZIPFIPS';
  label = fipscd;
  output;
  run;

proc sort data = cntlin;
  by fmtname start;
  run;
```

Note the PROC SORT step immediately following the CNTLIN data step. This is needed so all of the format definitions are grouped together physically. Otherwise, SAS will create many single-entry formats with each change in the value of FMTNAME. Although not shown in this example, you could create a definition for the special range OTHER following the example above. A key difference is that there will need be two OUTPUT statements in the DO-group - one for each format definition.

Creating tokenized labels

As the number of user-defined formats you would like to create from an existing SAS data set increases, the method shown above may become overly cluttered. As an alternative, you can create a tokenized label statement which contains all the information you want mapped to the key value delimited by some rarely used character like #. For example, ZIP code 06854 belongs in state CT with a FIPS state county code of 09001. To put both the state and county information in a single LABEL variable you would create an assignment statement as follows.

```
label = statecd || '#' || fipscd;
```

For our example, label would contain the literal 'CT#09001'. To use this type of format definition in a data step, you need to break out the tokens as necessary as shown in this code fragment.

```
data example;
  set in;
  length statecd $ 2 fipscd $ 5;
  info = put( zip, $zipinfo. );
  statecd = scan( info, 1, '#' );
  fipscd = scan( info, 2, '#' );
  run;
```

The preceding example assumes that the user-defined format $ZIPINFO exists with labels defined as above. The SCAN() data-step function is used to pull out specific tokens, or sub-fields, from the composite label. You can use any character as a token delimiter; however, you need to choose a character which will not appear in one of the tokens. Alternatively, you would create a fixed-field label value and use the SUBSTR() function to extract the desired tokens. In this example, state code will always be two characters and FIPS code will always be five characters, so a seven-character label would do. Once again, the choice is largely a matter of style.

Outlier identification

So far our CNTLIN example have been single-keyed. That is, a single value placed in the START variable is mapped to a label. You may also specify ranges of values in a CNTLIN data step using the special field name called END. To see how this might work consider the following example.

You have a data step which has upper and lower bounds by county for an analytic variable COST. You wish to create a summary display which shows average COST by county for both he raw, nominal values as well as the trimmed values - that is, values within the lower and upper bounds. You could accomplish this task by merging the data set containing the upper and lower bounds to the raw data. Provided the raw data are already sorted by county, this approach is simple enough to apply. However, if the data are not sorted properly and the raw data happen to be many millions of records, PROC FORMAT affords an alternate solution which does not require sorting of the raw data. The first step is to create a CNTLIN= data set from the bounds data set which defines a separate format for each county with a value of 1 between the upper and lower bound and 0 otherwise.

```
data cntlin( keep =
  fmtname type start end hlo label );
  retain type 'N';
  set bounds;
  fmtname = '_' || fipscd || '_';
  start = lower;
  end = upper;
  label = '1';
  output;
  hlo = 'O';
  label = '0';
  output;
  run;
```

The preceding example assumes that the BOUNDS data set has three variables - FIPSCD, LOWER, and UPPER, which represent the county code, lower and upper bounds respectively. Since user-defined formats may not start or end with a number, we create the FMTNAME by adding an underscore character to the beginning and end of the FIPS state-county code. Values with the lower and upper bounds are given a value of 1, other values get a 0. Note that this is a numeric format because we will use these formats to compare individual values of COST - a numeric variable - in the raw data set to a lower and upper bound.

Next we load the user-defined formats using the CNTLIN= option on PROC FORMAT. This will create as many user-defined formats as there are observations in BOUNDS.

```
proc format cntlin = cntlin;
```

Now we pass the raw data through a data step to create a new variable called TRIMCOST which has the value of COST for any value within bounds and missing otherwise.

```
data trimmed( keep =
  fipscd cost trimcost );
  set raw;
  fmtname = '_' || fipscd || '_';
  if putn( cost, fmtname ) = '1'
    then trimcost = cost;
  run;
```

The PUTN() data-step function is similar to the PUT() function except that it accepts as its second parameter a step-step variable which contains the value of a user-defined format. If the value of COST is between the lower and upper bounds, the previously defined format will return a value of '1' and we assign the value of COST to TRIMCOST. If this is not the case, TRIMCOST will retain its initial value of missing.

The final step is to pass TRIMMED through PROC SUMMARY to obtain the desired parametric statistics.

```
proc summary data = trimmed
  nway missing;
  class fipscd;
  var cost trimcost;
  output out = reduced
    n = n_cost n_trim mean=;
  run;
```

The data set REDUCED will contain the MEAN and N values for COST and TRIMCOST. The difference between N_COST and N_TRIM is the number of outliers removed from the raw mean to create the trimmed mean. Note that this was all done without a single SORT or MERGE which on a multi-million observation data set will yield tremendous savings of computational resources.

This concludes a somewhat rambling survey of advanced PROC FORMAT techniques. The party line about SAS is that while it doesn't do everything better than some other product with a specific niche - graphics, statistics, etc. - it does do everything it does pretty well. In terms of data manipulation the tools SAS provides are unsurpassed and PROC FORMAT is a gem of a tool without rival in any other application programming languages. I sincerely hope that this paper has given you some food for thought.

The author welcomes comments and criticisms.

Jack N Shoemaker
Oxford Specialty Management
800 Connecticut Avenue
Norwalk, CT 06854

203 851 2650 JShoemak@oxhp.com