Paper 3162-2015

Best Practices: Subset Without Getting Upset

Mary F. O. Rosenbloom, Edwards Lifesciences LLC, Irvine, California
Kirk Paul Lafler, Software Intelligence Corporation, Spring Valley, California

ABSTRACT

You've worked for weeks or even months to produce an analysis suite for a project, and at the last moment, someone wants a subgroup analysis and they inform you that they need it yesterday. This should be easy to do, right? So often, the programs that we write fall apart when we use them on subsets of the original data. This paper takes a look at some of the best practice techniques that can be built into a program at the beginning, so that users can subset on the fly without losing categories or creating errors in statistical tests. We review techniques for creating tables and corresponding titles with by-group processing so that minimal code needs to be modified when more groups are created, and we provide a link to sample code and sample data that can be used to get started with this process.

KEYWORDS

SAS, subset, subsetting, by-group, by-group processing, subgroup analysis, best practice, SQL, SAS SQL, PROC SQL, joins, LEFT join, Coalesce

INTRODUCTION

As programmers, we spend a great deal of time and effort to ensure that our code is correct, functioning appropriately, and is robust enough to handle a growing database where new data issues may appear after program validation. However, when we are asked to run our programs on a subset of that data, especially a very small subset, too often we discover that our bulletproof programs aren't so bulletproof after all. Subgroup analysis is often requested on the fly after initial results have come out.

Researchers may want to dig deeper to understand the root cause of a study result. When time is fleeting, discovering warnings, errors, and summary data containing periods instead of summary statistics is not what we want to happen. Programs will then need to be modified or even overhauled, and then usually validated, and all of this takes time that we probably don't have, is stressful, and it makes us look bad. This paper identifies some of the major pitfalls that can happen when data are subset, and discusses best practice techniques that can be implemented the first time the program is written so that we can subset without getting upset.

THE SAMPLE DATA

Most of the examples in this paper use the built-in SAS dataset SASHELP.HEART, so that the reader can work through the examples as well as reading the paper. This data originates from the Framingham Heart Study and contains vital status, gender, age at start, weight and smoking status, among other factors. Selected columns are shown here.

	Status	Sex	Age at Start	Weight Status	Smoking Status
1	Dead	Female	29	Overweight	Non-smoker
2	Dead	Female	41	Overweight	Non-smoker
3	Alive	Female	57	Overweight	Moderate (6-15)
4	Alive	Female	39	Overweight	Non-smoker
5	Alive	Male	42	Overweight	Heavy (16-25)
6	Alive	Female	58	Overweight	Non-smoker
7	Alive	Female	36	Overweight	Moderate (6-15)
8	Dead	Male	53	Normal	Non-smoker
9	Alive	Male	35	Overweight	Non-smoker
10	Dead	Male	52	Normal	Light (1-5)

SUBSETTING CAN CAUSE DROPPED VALUES IN THE OUTPUT

When data are subset, most likely some of the data levels present in the main data are lost. We usually would like to represent all of the data levels in the summary, showing zero counts for the levels that nobody has. Working with the SASHELP.HEART dataset, we used PROC SQL to add a few more data points. These five fictitious subjects – all male – are the only subjects in the study that are 70 or older at the study start.

```
proc sql;
create table heart as
select status, sex, ageatstart, height, weight, diastolic, systolic, mrw,
   smoking, cholesterol, chol_status, bp_status, weight_status,
   smoking_status
from sashelp.heart;

insert into heart
values('Alive', 'Male', 80, 60, 160, 85, 130, 118, 0, 185, 'Desirable',
   'Normal', 'Normal', 'Non-smoker')
values('Alive', 'Male', 82, 61, 185, 86, 160, 120, 0, 186, 'Borderlin',
   'Normal', 'Normal', 'Light (1-5)')
values('Alive', 'Male', 79, 62, 190, 87, 150, 121, 0, 187, 'Desirable',
   'Normal', 'Normal', 'Non-smoker')
values('Alive', 'Male', 90, 63, 175, 88, 130, 130, 0, 188, 'Borderlin',
   'Normal', 'Normal', 'Non-smoker')
values('Alive', 'Male', 89, 64, 170, 89, 120, 135, 0, 189, 'Desirable',
   'Normal', 'Normal', 'Tobacco Exec');
quit;
```

At SAS Global Forum 2014, Jason Dorsey of The Center for Generational Kinetics gave a colorful presentation on intergenerational working issues. He classified generations into five groups: “Generation I”, “Millenials”, “Generation X”, “Baby Boomers”, and “Traditionalist”. Using those definitions, we can create a format for our age variable.

```
proc format cntlout=_fmtvals;
      value generation
             1-18   = 'Generation I'
            19-37   = 'Millenials'
            38-49   = 'Generation X'
            50-68   = 'Baby Boomers'
            69-high = 'Traditionalist'
            ;
run;
```

Now, when we summarize gender by age group, we see that the following generations are present in our population (remember that we added a few observations, above).

```
proc freq data=heart;
      format AgeAtStart generation.;
      tables sex*AgeAtStart/missing;
run;
```

Table of AgeAtStart by Sex

	Sex		
AgeAtStart(Age at Start)	Female	Male	Total
Millenials	806	678	1484
Generation X	1198	936	2134
Baby Boomers	869	722	1591
Traditionalist	0	5	5
Total	2873	2341	5214

The youngest generation are the “Millenials”. We have no subjects from “Generation I” and just five “Traditionalists” (the fake records that we added, above). There are plenty of patients from “Generation X” and “Baby Boomers”.

Perhaps now we want to dig a little deeper and examine the relationship between generation and gender for the patients who have died. We can add a subsetting WHERE clause to our PROC FREQ call.

```
proc freq data=heart;
      where status='Dead';
      format AgeAtStart generation.;
      tables sex*AgeAtStart/
  missing norow nocol nopercent
out=_dead;
run;
```

Table of AgeAtStart by Sex			
	Sex		
AgeAtStart(Age at Start)	Female	Male	Total
Millenials	100	133	233
Generation X	308	400	708
Baby Boomers	488	562	1050
Total	896	1095	1991

Now, we only get summary statistics for “Millenials”, “Generation X”, and “Baby Boomers”. Instead, we would like to generate summary statistics for all five generational groups whenever the data are summarized, whether the levels are present or not.

USE PRELOADFMT TO ENSURE THAT ALL THE DATA LEVELS ARE PRESENT

There are several ways to ensure that all of the data levels are present in the output. Some are more labor intensive than others. Our goal is to set up a system the first time that the code is written so that when the data are subset, little-to-no additional programming is required, with the exception of requesting the new subgroup analysis. Using the PRELOADFMT option is a powerful way to do this.

Using PROC TABULATE, we can employ PREFLOADFMT to ensure that all of the levels in the format definition for GENERATION are present in the output. The code is straightforward. We specify PRELOADFMT ❶ as an option in the CLASS statement. Then we apply a format to our CLASS variable. ❷

```
proc sort data=heart out=heart_sort;
   by sex;
run;
proc tabulate data=heart_sort
      out=_dead_all_level;
   by sex;
   class AgeAtStart/preloadfmt;❶
   format AgeAtStart generation.;❷
   table AgeAtStart='Generation',(n)/
      printmiss misstext='0';
run;
```

Sex	Age at Start	N
Female	Generation I	0
Female	Millenials	806
Female	Generation X	1198
Female	Baby Boomers	869
Female	Traditionalist	0
Male	Generation I	0
Male	Millenials	678
Male	Generation X	936
Male	Baby Boomers	722
Male	Traditionalist	5

The figure on the right shows that “Traditionalists” are summarized for females, even though the count is zero, unlike with PROC FREQ. PRELOADFMT is available in PROC MEANS, PROC REPORT and PROC SUMMARY, too, but not in PROC FREQ. Examining the output dataset, we see that all the levels are present, although we’ll need to do a little more work to get the data into the polished format for the output. More on this later.

What if there are values present in the data that are not part of the format? Let’s see what happens when we summarize smoking status. We have created a very unglamorous format for smoking status so that we can employ PRELOADFMT. We won’t be modifying any of the values for SMOKING_STATUS, but we create the format to ensure that all values are represented.

```
proc format cntlout=_fmtvals;
   value $ smoking_status
       'Non-smoker' = 'Non-smoker'
       'Light(1-5)' = 'Light (1-5)'
       'Moderate (6-15)' = 'Moderate (6-15)'
       'Heavy (16-25)' = 'Heavy (16-25)'
       'Very Heavy (> 25)' = 'Very Heavy (> 25)'
       ;
run;
```

When we summarize the modified heart data, we see that not only are all of the levels present, but there is an additional level – "Tobacco Executive"! So, we are able to summarize the data, ensuring that we get a record for each formatted value, plus any unexpected values.

```
proc sort data=heart out=heart_sort;
       by smoking_status;
run;
proc tabulate data=heart_sort
              out=_smoke_all_level;
       by smoking_status;
       class AgeAtStart/preloadfmt;
       format AgeAtStart generation.;
       table AgeAtStart='Generation',(n)
              /printmiss misstext='0';
run;
```

Smoking Status=Tobacco Exec

	N
Generation	
Generation I	0
Millenials	0
Generation X	0
Baby Boomers	0
Traditionalist	1

If we want to ensure that "Tobacco Executive" is part of the summary for further analysis, we will need to add it to the PROC FORMAT code. If we want to only summarize the values in the format, the EXCLUSIVE option is available, and is placed in the CLASS statement. This will result in the summarization of all class levels contained in the format, but none other, so "Tobacco Executive" will be excluded.

```
proc sort data=heart out=heart_sort;
       by sex;
run;
proc tabulate data=heart_sort  out=_smoke_all_fomat;
       by sex;
       class smoking_status/preloadfmt exclusive;
       format smoking_status $smoking_status.;
       table smoking_status='Smoking
Status',(n)/printmiss misstext='0';
run;
```

We can use PRELOADFMT with continuous data, too. In the next example, we summarize age at death among those who have died, based on their generational group. Without PRELOADFMT, we only get summary statistics for the three generations present in the data.

```
proc means data=sashelp.heart
   (where=(status='Dead'));
 class AgeAtStart;
 format AgeAtStart generation.;
 var AgeAtDeath;
run;
```

Analysis Variable : AgeAtDeath Age at Death						
Age at Start	N Obs	N	Mean	Std Dev	Minimum	Maximum
Millenials	233	233	57.6137339	7.7862711	36.0000000	69.0000000
Generation X	708	708	66.2245763	8.3590986	42.0000000	81.0000000
Baby Boomers	1050	1050	76.3114286	8.2460473	54.0000000	93.0000000

With PRELOADFMT, we create summary statistics for all of the groups.

```
proc means data=sashelp.heart
      (where=(status='Dead')) completetypes;
   class AgeAtStart/preloadfmt;
   format AgeAtStart generation.;
   var AgeAtDeath;
run;
```

Age at Start	N Obs	N
Generation I	0	0
Millenials	233	233
Generation X	708	708
Baby Boomers	1050	1050
Traditionalist	0	0

Notice also, that the underlying age values help to dictate the order of the generational groups in the output. This is a nice bonus. However, when we perform a similar summary by smoking status, the output is in alphabetical order (Morris, 2011).

```
proc means data=sashelp.heart
      (where=(status='Dead')) completetypes;
   class smoking_status/preloadfmt ;
   format smoking_status $smoking_status.;
   var AgeAtDeath;
run;
```

Smoking Status	N Obs	N
Heavy (16-25)	443	443
Light (1-5)	187	187
Moderate (6-15)	213	213
Non-smoker	891	891
Very Heavy (> 25)	237	237

This is because formats are stored in alphabetical order. But, there is a remedy for this. First, we must specify the NOTSORTED option in our PROC FORMAT call. We create a new format, SMOKING_STATUS_NS, to illustrate this.

```
proc format cntlout=_fmtvals;
   value $ smoking_status_ns (notsorted)
       'Non-smoker' = 'Non-smoker'
       'Light (1-5)' = 'Light (1-5)'
       'Moderate (6-15)' = 'Moderate (6-15)'
       'Heavy (16-25)' = 'Heavy (16-25)'
       'Very Heavy (> 25)' = 'Very Heavy (> 25)'
       ;
run;
```

Then, we use ORDER=DATA in PROC MEANS (Li, Hua, Li, and Lan, 2011).

```
proc means data=sashelp.heart
      (where=(status='Dead')) completetypes;
   class smoking_status/preloadfmt order=data;
   format smoking_status $smoking_status_ns.;
   var AgeAtDeath;
run;
```

Smoking Status	N Obs	N
Non-smoker	891	891
Light (1-5)	187	187
Moderate (6-15)	213	213
Heavy (16-25)	443	443
Very Heavy (> 25)	237	237

The result is a summary presented in a meaningful order rather than alphabetically. For a detailed description of PRELOADFMT, EXCLUSIVE, COMPLETETYPES, and other useful options for these procedures, see Carpenter, 2012.

USING A CLASSDATA DATASET

So far we have seen that by taking the time to set up formats for the data, that we can avoid dropping categories from our summaries when we subset. But there are other ways to ensure that all categories are present. One way is to simply create a dataset with all possible levels for a variable, and then to merge it with the summary statistics that we create in PROC MEANS or PROC FREQ or some other procedure. In fact, a more elegant option is available in several of the procedures (PROC TABULATE, PROC MEANS, and PROC SUMMARY), and allows us to specify a dataset containing all possible categories without merging that dataset into the present one. Moreover, since we have already taken the time to create formats for our variables, we can use the CNTLOUT option in PROC FORMAT to output the values to a dataset, which we can subset and then use for this purpose.

Let's take a look at the process that would be used to summarize age at death by generation. In the PROC FORMAT call, we specify the CNTLOUT= option ❶ so that we can output the dataset _FMTVALS containing the formatted values.

Format name	Format value label
GENERATION	Generation I
GENERATION	Millenials
GENERATION	Generation X
GENERATION	Baby Boomers
GENERATION	Traditionalist

It takes a little bit of work to ensure that the variables are formatted to the right length but we are able to use the results to provide the CLASSDATA dataset. Next, we create the variable LABEL, which is the formatted age data. ❷ This will serve as our class variable ❸ in the PROC MEANS, below. We also specify the CLASSDATA= option and point it to _FMTVALS. ❹

The resulting output (shown below) contains all five generations of subjects (Carpenter, 2012).

```
/*set up some formats*/
proc format
      cntlout=_fmtvals (keep=label FMTNAME);❶
   value generation
           1-18   = 'Generation I'
          19-37   = 'Millenials'
          38-49   = 'Generation X'
          50-68   = 'Baby Boomers'
          69-high = 'Traditionalist'
   ;
run;

/*make sure it is character 17*/
data _fmtvals;
   length label $ 17;
   set _fmtvals;
run;

/*create formatted variable*/
data _heartfmt;
       set sashelp.heart;
       label=put(AgeAtStart, generation17.);❷
run;

/*get summary for all five age groups*/
proc means data=_heartfmt
   (where=(status='Dead'))
   classdata=_fmtvals  ❹
      (where=(FMTNAME='GENERATION'));
   class label / order=data; ❸
   var AgeAtDeath;
run;
```

Analysis Variable : AgeAtDeath Age at Death						
label	N Obs	N	Mean	Std Dev	Minimum	Maximum
Generation I	0	0	.	.	.	.
Millenials	233	233	57.6137339	7.7862711	36.0000000	69.0000000
Generation X	708	708	66.2245763	8.3590986	42.0000000	81.0000000
Baby Boomers	1050	1050	76.3114286	8.2460473	54.0000000	93.0000000
Traditionalist	0	0	.	.	.	.

USING PROC SQL TO MIMIC PRELOADFMT

PROC SQL is a powerful and flexible alternative to using procedures. Let's look at an example where we summarize weight by smoking status. We will use the modified HEART data that has fake observations added to it, and we will also use the unsorted smoking status format $SMOKING_STATUS_NS. We want to order the output in the logical order specified in the format definition.

```
ods output sql_results=rownumber; ❸
proc sql number; ❷
   select label
   from _fmtvals❶
   where UPCASE(fmtname) ='SMOKING_STATUS_NS';
quit;

proc sql;
   create table try1 as
   select
       mean(weight) as mean_wt,❹
       std(weight) as std_wt,
       count(weight) as n,
       smoking_status as label
   from heart
   where not missing(smoking_status)
   group by smoking_status; ❺

   /*join with formats*/
    create table try2 as
    select
      ifc(missing(mean_wt), 'NA', ❼
          put(mean_wt, 5.1)) as mean_wt,
      ifc(missing(std_wt), 'NA',
          put(std_wt,5.1)) as std_wt,
      n,
      coalesce(a.label,g.label) as final,❽
      row
    from try1    g
          FULL JOIN ❻
          rownumber       a
       on g.label = a.label
    order by row; ❾
quit;
```

First, we will use the output dataset _FMTVALS, which is produced by PROC FORMAT, above. We subset this to the records from the format $SMOKING_STATUS_NS, which is stored without sorting, so it maintains the logical order in which it was defined rather than alphabetical order. ❶ By specifying NUMBER in the PROC SQL call, ❷ we can get the row numbers output. We must use ODS OUTPUT to obtain the dataset ROWNUMBER that contains this value. ❸ It will also contain the other variable(s) in our SELECT statement. The dataset ROWNUMBER contains the unsorted format values and their row number, which we can use to order the values later.

Row	Format value label
1	Non-smoker
2	Light (1-5)
3	Moderate (6-15)
4	Heavy (16-25)
5	Very Heavy (> 25)

This technique for obtaining row numbers with PROC SQL was developed by Jiangtang Hu (see references). Use of the MONOTONIC function for this purpose is not advisable, since it can produce incorrect results. Next we return to the modified HEART dataset. We calculate the mean, standard deviation, and count for weight, ❹ grouped by SMOKING STATUS, ❺ and store it in the dataset TRY1. Finally, we perform a FULL JOIN of the ROWNUMBER data and the TRY1 data, on the values for smoking status (contained in the LABEL variable on both datasets). ❻ We want to set missing values to 'NA' for the mean and standard deviation, which we accomplish by using the IFC function. ❼ We COALESCE the LABEL variables from the two datasets, which has the effect of assigning the first non-missing value between the two variables to FINAL.❽ Finally, we order the dataset by ROW, the variable that represents the logical order for the values of smoking. ❾ The resulting dataset contains all the values from our format, plus the summary for 'Tobacco Exec', a value found in the data but not found in our format definition. We keep this value in the summary because we performed a FULL JOIN. Since it does not have a value for ROW, it sorts to the top. We would like to thank Richard C Carson for suggesting the JOIN of the PROC FORMAT output dataset to the data itself to ensure that all formatted values are represented.

mean_wt	std_wt	n	final	Row
170.0	NA	1	Tobacco Exec	.
153.8	29.3	2502	Non-smoker	1
146.8	27.5	579	Light (1-5)	2
144.6	27.0	575	Moderate (6-15)	3
154.8	28.5	1046	Heavy (16-25)	4
164.1	27.5	469	Very Heavy (> 25)	5

STATISTICAL TESTS CAN FAIL WITH SPARSE DATA

So far we have explored several ways to ensure that all levels of the data are represented. Let's turn our attention now to statistical testing. Many of the statistical tests that we commonly perform will break down when data are sparse. Recall that when we modified the SASHELP.HEART dataset, we added five fictitious male subjects, all of whom comprise the only "Traditionalist" generation for this dataset. When we attempt to compare smoking status between genders among this generation,

```
proc freq data=heart;
      where put(AgeAtStart, generation.)='Traditionalist';
      tables sex*smoking_status/chisq;
      output out = _gender_smoke pchi;
run;
```

we get WARNINGs in our log:

```
NOTE: Writing HTML Body file: sashtml.htm
NOTE: No statistics are computed for Sex * Smoking_Status since Sex
has less than 2 nonmissing levels.
WARNING: No OUTPUT data set is produced because no statistics can be
computed for this table, which has a row or column
         variable with less than 2 nonmissing levels.
WARNING: Data set WORK._GENDER_SMOKE was not replaced because new file
is incomplete.
NOTE: There were 5 observations read from the data set WORK.HEART.
      WHERE PUT(AgeAtStart, GENERATION14.)='Traditionalist';
NOTE: PROCEDURE FREQ used (Total process time):
      real time           1.64 seconds
      cpu time            1.31 seconds
```

This is because there are no females in the "Traditionalist" generation. When we try to perform a t-test comparing the unformatted age between genders within the "Traditionalist" group, we get an ERROR regarding the CLASS statement, and a WARNING regarding the generation of ODS output.

Table of Sex by Smoking_Status				
	Smoking_Status(Smoking Status)			
Sex	Light (1-5)	Non-smoker	Tobacco Exec	Total
Male	1	3	1	5
	20.00	60.00	20.00	100.00
	20.00	60.00	20.00	
	100.00	100.00	100.00	
Total	1	3	1	5
	20.00	60.00	20.00	100.00

```
ods output ttests=_ttests_trad;
proc ttest data=heart
      (where=(put(AgeAtStart, generation.)
       ='Traditionalist'))
       plots=none;
   class sex;
   var AgeAtStart;
run;
```

```
ERROR: The CLASS variable does not have two levels.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE TTEST used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

WARNING: Output 'ttests' was not created.  Make sure that the output object name, label,
or path is spelled correctly.  Also, verify that the appropriate procedure options are
used to produce the requested output object.  For example, verify that the NOPRINT
option is not used.
```

Similar messages are placed in the log when we try to perform a K-S test with PROC NPAR1WAY. This scenario has happened to at least one of the authors of this paper, and it can be very alarming. Let's look at two ways to work around this.

USING THE NLEVELS OPTION WITH PROC FREQ

PROC FREQ provides one handy solution for us with the NLEVELS option. In the examples above where we wanted to compare two groups, the tests failed because the variable acting as the class variable had fewer than two levels. We can use the NLEVELS option in PROC FREQ to determine the number of levels in the data. The following code can be used to evaluate the levels of all of the character data in the SASHELP.HEART dataset:

```
proc freq data=sashelp.heart nlevels;
      tables _character_ / noprint;
run;
```

The output includes the number of levels, the number of missing levels, and the number of non-missing levels for each categorical variable in the dataset.

Number of Variable Levels

Variable	Label	Levels	Missing Levels	Nonmissing Levels
Status		2	0	2
DeathCause	Cause of Death	6	1	5
Sex		2	0	2
Chol_Status	Cholesterol Status	4	1	3
BP_Status	Blood Pressure Status	3	0	3
Weight_Status	Weight Status	4	1	3
Smoking_Status	Smoking Status	6	1	5

Let's return to the example where we wanted to use a t-test to compare starting ages between genders within the "Traditionalist" generation. Prior to requesting this test, we could have checked the levels of the CLASS variable, SEX.

```
ods output nlevels=Levels;
proc freq data=heart
      (where=(put(AgeAtStart, generation.)='Traditionalist'))  nlevels;
   tables sex / noprint;
run;
```

This creates a dataset called LEVELS, which tells us that in this subgroup, SEX only has one value. We can store this information in a macro variable.

Number of Variable Levels

Variable	Levels
Sex	1

```
data _null_;
      set levels;
      call symputx('levels', NLevels);
run;
%put There are &levels levels for the variable SEX;
```

We might then use this within a macro to conditionally request the statistical test.

```
%macro conditional();
%if %eval(&levels ge 2) %then %do;❶
   ods output ttests=_ttests_trad;
   proc ttest data=heart
       (where=(put(AgeAtStart, generation.)='Traditionalist')) plots=none;
   class sex;
   var AgeAtStart;
   run;
%end;❶
   %else %do;
       %put Nothing to do because there are only &levels levels.;❷
   %end;
%mend;
options mprint;
%conditional
```

We can place it into a %DO loop ❶, which will only be processed if the conditions are met. Otherwise, we can send a message to the log. ❷

This results in the following message in the log:

```
Nothing to do because there are only 1 levels.
```

Of important note, here, is that the data which creates the NLEVELS output must also be subset the same way as the data to be analyzed. The macro, above, is an extremely simplified example that serves to illustrate the technique of conditional processing. Later, we will present a more sophisticated macro that ties together several of the techniques that we are discussing.

CONDITIONAL STATISTICAL TESTING WITH PROC SQL

PROC SQL provides a streamlined way to both count the levels of the class variable and store that count in a macro variable.

```
proc sql;
       select count(distinct sex) into :levels
       from heart
       where put(AgeAtStart, generation.)='Traditionalist';
quit;
%put &levels has &levels levels;
```

We could put this into a macro similar to the one shown above. We will see more of this later. Again, it is important to use the same subsetting WHERE conditions that will be used if the statistical test is requested.

SET UP SUMMARY STRINGS TO HANDLE MISSING DATA AND ZERO VALUES

After calculating summary statistics, it is often necessary to concatenate the values into strings which contain some formatting. Next, we'll return to the CLASSDATA example where we summarized age at death by generational group. Now, we add an output statement, and request the mean, standard deviation, min and max.

```
proc means data=_heartfmt (where=(status='Dead'))
             classdata=_fmtvals (where=(fmtname='GENERATION'));
       class label / order=data;
       var AgeAtDeath;
       output out=_out1 n=n mean=mean std=std min=min max=max;
run;
```

Part of the output dataset is shown here. You can see that there are some missing values for the classes where N=0.

	label	_TYPE_	n	mean	std	min	max
1		0	1991	70.536413862	10.559406247	36	93
2	Generation I	1	0	.	.	.	.
3	Millenials	1	233	57.613733906	7.7862710885	36	69
4	Generation X	1	708	66.224576271	8.3590986071	42	81
5	Baby Boomers	1	1050	76.311428571	8.2460472581	54	93
6	Traditionalist	1	0	.	.	.	.

We might then build strings to contain a formatted version of these summary statistics.

```
data _string;
      set _out1;
      attrib string label='Summary' format=$30.;
      string=cat(n, ': ',
            strip(put(mean,8.1)), ' ± ',
            strip(put(std,8.1)), ' (',
            strip(put(min,8.1)), ' - ',
            strip(put(max,8.1)), ')'
      );
run;
```

But when we look at the _STRING dataset, we can easily see that the summary strings are wrong for “Generation I” and “Traditionalist”:

	label	n	mean	std	min	max	string
1	Generation I	0	.	.	.	.	0: . ± . (. - .)
2	Millenials	233	57.613733906	7.7862710885	36	69	233: 57.6 ± 7.8 (36.0 - 69.0)
3	Generation X	708	66.224576271	8.3590986071	42	81	708: 66.2 ± 8.4 (42.0 - 81.0)
4	Baby Boomers	1050	76.311428571	8.2460472581	54	93	1050: 76.3 ± 8.2 (54.0 - 93.0)
5	Traditionalist	0	.	.	.	.	0: . ± . (. - .)

So, we need to do a little more work to manage these cases. Additionally, we will add a clause to deal with the case when a group contains only one observation, which will cause the standard deviation to be undefined. The following code does the trick. We can use the value of N for each record to tell us how to build the string. For cases where N is greater than 1, we expect all of the summary statistics to be defined. ❶ When N=1, the standard deviation will be undefined, so we will set it to “NA”. ❷ And, in the case

```
data _string;
   set _out1 (where=(_type_=1));
   attrib string label='Summary' format=$100.;
   /*more than one record in the group*/
   if n gt 1 then string=cat(n, ': ', ❶
      strip(put(mean,8.1)), ' ± ',
      strip(put(std,8.1)), ' (',
      strip(put(min,8.1)), ' - ',
      strip(put(max,8.1)), ')'
   );
      /*only one record in the group so STD not defined*/❷
      else if n=1 then string =cat(n, ': ',
         strip(put(mean,8.1)), ' ± NA ', ' (',
         strip(put(min,8.1)), ' - ',
         strip(put(max,8.1)), ')'
      );
         /*no records in the group*/❸
         else if n=0 then string ='0: NA ± NA (NA - NA)';
run;
```

where N=0, we will create a string indicating this, using a professional display in place of the string of missing values. We can employ similar code for summaries of categorical variables, and for p-values. We will see this later in the paper.

STACK THE RESULTS AND REPORT USING BY GROUP PROCESSING

Often, when subgroup analysis is requested, it is requested in addition to the overall analysis rather than in place of it. When this is the case, we would like to minimize the coding that must be done to summarize and report the subgroup. One way to do this is to use a macro to create the summary data, and then stack it together. Then, we can use BY group processing within a procedure such as PROC REPORT or PROC PRINT to display the data and to create custom titles.

Let's take a look at one such macro. The STACK macro takes two parameters, &WHERE, the WHERE statement, and &SUBGROUP, which is a descriptive character string for the subgroup. ❶ PROC TABULATE is used in this simple example to get counts for each generational group, ❷ and then the results are set at the bottom of a dataset called STACKED. ❸ Furthermore, a global macro variable, &CALLS, is incremented each time the macro is called. ❹

```
%macro stack(where=, subgroup=);❶
/*number of calls to the macro*/
%let calls=%eval(&calls+1);❹

/*calculate summary statistics*/❷
proc tabulate data=heart_sort  out=_current;
where &where;
class AgeAtStart/preloadfmt;
format AgeAtStart generation.;
table AgeAtStart='Generation',(n)/printmiss
misstext='0';
run;

/*stack onto the results dataset*/
data stacked;
      set stacked _current(in=inc);❸
      format subgroup $30.;
      if inc then do;
            subgroup="&subgroup";
            /*create a sorting variable*/
            calls=&calls;
      end;
      if not missing(subgroup);
run;
%mend stack;
```

To run the macro, we must first initialize the stacking dataset, ❺ initialize the global macro variable &CALLS to zero, ❻ and then call the macro. For simplicity in this example, we set missing values to print as zero in the OPTIONS statement. ❼ Two calls are shown, one for all patients and one for patients who died. ❽

```
/*set missing to zero for simplicity in this example*/
options missing='0'; ❼

/*initialize the stacking dataset*/
data stacked;❺
run;

/*initialize the variable to count the number of calls to the
macro*/
%let calls=0; ❻

/*call the macro for each group*/ ❽
%stack(where=not missing(AgeAtStart), subgroup=All Subjects)
%stack(where=status='Dead', subgroup=Patients Who Have Died)
```

Relevant columns in the output data are shown here:

	Age at Start	N	subgroup	calls
1	Generation I	0	All Subjects	1
2	Millenials	1484	All Subjects	1
3	Generation X	2134	All Subjects	1
4	Baby Boomers	1591	All Subjects	1
5	Traditionalist	5	All Subjects	1
6	Generation I	0	Patients Who Have Died	2
7	Millenials	233	Patients Who Have Died	2
8	Generation X	708	Patients Who Have Died	2
9	Baby Boomers	1050	Patients Who Have Died	2
10	Traditionalist	0	Patients Who Have Died	2

Now that we have this lovely stacked output, with CALLS, a variable that can be used to order a by-group, we can display the results with one call to PROC REPORT (or PROC PRINT, etc).

```
options nodate nonumber papersize=letter
orientation=portrait nobyline;
ods rtf file="C:\WUSS\by group.rtf"
   style=Journal bodytitle;

title1 j=c "Table #byval1.. Summary for #byval2";

proc report data=stacked (drop=_:) nowd;
  by calls subgroup;
run;
ods rtf close;
```

Table 1. Summary for All Subjects

Age at Start	N	subgroup	calls
Generation I	0	All Subjects	1
Millenials	1484	All Subjects	1
Generation X	2134	All Subjects	1
Baby Boomers	1591	All Subjects	1
Traditionalist	5	All Subjects	1

The PROC REPORT is run with a BY statement which specifies CALLS and SUBGROUP. In the TITLE statement, we can get the value of CALLS with “#BYVAL1” and the value of SUBGROUP with “#BYVAL2”. One of the resulting tables is shown at the right. For a nice explanation of BY group processing of output, see Carpenter (1997).

PUTTING IT ALL TOGETHER

We have discussed several techniques, and have written the macro %SUM1 in an attempt to tie the concepts together. This macro can be used to summarize categorical data by gender for the SASHELP.HEART data. This is a simplified macro that could be generalized in many ways, but for our purposes, serves to illustrate how many of the techniques mentioned might be used together.

This macro takes several parameters: ❶ &DSN holds the input dataset name, &WHERE, which must be non-missing, selects a subgroup within a WHERE statement in several locations. We can specify a string representing the subgroup to be used in titles with &TITLEWHERE. The analysis variable name is stored in &VAR, while the corresponding label is specified with &LABEL, and the format with &FORMAT. We have limited this macro to analysis on character variables, but one could easily incorporate analysis for a continuous variable and then stack the results all within this macro,

```
%macro sum1(dsn=, where=, titlewhere=, var=, label=, format=);❶
proc tabulate data=&dsn
   (where=(&where and not missing(&var))) out=_tryout;
     class &var sex/preloadfmt;❷
     format sex $gender. &var &format..;
     table &var='Factor'*sex='Gender',(n)/printmiss misstext='0';
run;

proc transpose data=_tryout out=_ttryout;
   by &var;
   id sex;
   var n;
run;

proc sql;❸
   create table _tryout2 as
   select &var as factor format=$200. label='Factor',
      cat("&label", ' % (n/N)') as group format=$200.,
      catt(strip(put(max(0,male)/sum(male),percent8.1)),
             ' (', max(0,male),'/',sum(male), ')')
             as stat_male format=$200. label='Male',
      catt(strip(put(max(0,female)/sum(female),percent8.1)),
             ' (', max(0,female),'/',sum(female), ')')
             as stat_female format=$200. label='Female'
       from _ttryout;

   select count(distinct &var), count(distinct sex)
      into :levels, :sex ❹
   from &dsn
   where &where and not missing(sex);
quit;
%put the variable &var has &levels levels;

%if %eval(&levels >= 2) and %eval(&sex = 2)%then %do; ❺
   proc freq data=&dsn (where=(&where and not missing(sex)));
      tables sex*&var /  chisq;
      output out = _chi (keep=P_PCHI) pchi;
   run;
   data _tryout2; ❻
      merge _tryout2 _chi;
   run;
%end;

data stacked (where=(not missing(factor)));❼
   format factor $200.;
   set stacked _tryout2 (in=innew);

   format titlewhere $200.;
   if innew then titlewhere="&titlewhere";

   if stat_male='. (0/.)' then stat_male='0'; ❽
   if stat_female='. (0/.)' then stat_female='0';

   pval=ifc(not missing(P_PCHI), put(P_PCHI, pvalue6.4), '---');
run;

proc datasets lib=work memtype=data nolist;❾
  delete _:;
quit;
%mend sum1;
```

using %DO loop processing. We obtain counts for each categorization using PROC TABULATE. The use of PRELOADFMT ❷ ensures that all values specified in the corresponding FORMAT definition will be summarized, even if they have zero counts. We transpose these results in order to create a column for each gender. Next, we use PROC SQL ❸ to calculate the remaining summary statistics and place them into strings of concatenated values. The MAX function is used to set missing numerator counts to zero. Next, we count the number of distinct values for SEX and for the summary variable &VAR, making sure to subset with the same WHERE statement that will be used to calculate the summary statistics for this subgroup (and for the statistical test, if appropriate). ❹ We use a %DO loop ❺ to check that the analysis variables have the appropriate number of levels, and if they do, then the statistical test is performed and ❻ merged with the other summary statistics. This type of merge (without a BY statement) forces the data together, and the p-value will join with the first row of data on the other dataset. When we call this macro, we will first initialize a dataset called STACKED that we will use to stack the results. We will see an example of that shortly. In this section of the macro, however, we use the STACKED dataset to append the summary statistics to the previous data, if any. ❼ We create the variable TITLEWHERE from the value of the macro variable with the same name, and then use a few more lines to account for zero counts, or undefined p-values, transforming these values into polished ones. ❽ Finally, we delete all of the temporary datasets, which we have set to have names beginning with an underscore. This is a best practice housekeeping technique. (Rosenbloom and Lafler, 2012)

To call this macro, we just need to initialize the stacking dataset first. In this example, we call the macro for the overall data. We will summarize the variable AGE_GROUP (which is created in the data step for HEART2 – the dataset HEART is based on SASHELP.CLASS with records added – found in the beginning of this paper), SMOKING_STATUS, and STATUS (alive or dead). However, we may be asked to subset on the fly. We might choose to summarize for the subgroup where STATUS='Dead', or AGE_GROUP='Traditionalist'. We will have an error and warning-free log, and professional-looking values in the output. Since the data are stacked in a meaningful way, we can set up BY group summarization of the output, which will automatically summarize the new subgroups in their own tables. You can see, however, that the macro calls are somewhat repetitive, and long. We have only shown three calls here, but one might want to call this macro for every categorical variable on the HEART2 dataset. There are ways to automate these calls. While this is outside the scope of this paper, the process for automating calls to the %SUM1 macro is discussed in Rosenbloom and Carpenter (2014).

```
data heart2;
       set heart;
       age_group=put(ageatstart, generation.);
run;

data stacked;
run;

/*overall data*/
%sum1(dsn=heart2, where=not missing(status),
       titlewhere=All Records, var=age_group,
       label=Generation, format=$generation)

%sum1(dsn=heart2, where=not missing(status),
       titlewhere=All Records, var=smoking_status,
       label= Smoking Status, format=$smoking_status)

%sum1(dsn=heart2, where=not missing(status),
       titlewhere=All Records, var=status,
       label=Status, format=$status)
```

FINAL THOUGHTS

We have explored several ways that one can get into trouble when data become sparse due to subsetting, and we have suggested several ways to address these scenarios before they happen. We have seen that taking the time to set up and assign formats to variables is a good investment, since it allows us to take advantage of several procedure options that address sparse data. Since formats can be created with datasets, it may be beneficial to create some formats for a study dynamically using all values found in the largest subset of the data. This would ensure that all possible subgroups are always summarized when the appropriate options are used. We have also seen that PROC SQL provides many useful tools for dealing with sparse data, such as checking the number of non-missing levels and assigning that value to a macro variable. Many of the examples that we have shown have been simplified and generalized for discussion purposes. Hopefully this will inspire the reader to set up these safety nets in your programs so that you can subset without getting upset.

ABOUT THE AUTHORS

Mary Rosenbloom is a statistical programmer at Edwards Lifesciences in Irvine, California. She has been using SAS for over 15 years, and is especially interested in using macros to generate data-driven code, documenting best practice techniques, DDE, and program validation methods.

Kirk Paul Lafler is consultant and founder of Software Intelligence Corporation and has been programming in SAS since 1979. He is a SAS Certified Professional, sasCommunity emeritus advisory board member, application developer, data scientist, and provider of IT consulting services and training to SAS users around the world. As the author of six books including *Google Search Complete!* (Odyssey Press. 2014), *PROC SQL: Beyond the Basics Using SAS, Second Edition* (SAS Institute. 2013) and *PROC SQL: Beyond the Basics Using SAS* (SAS Institute. 2004), he has written more than five hundred papers and articles, been an Invited speaker at four hundred-plus SAS International, regional, special-interest, local, and in-house user group conferences/meetings, and is the recipient of 23 "Best" contributed paper, hands-on workshop (HOW), and poster awards.

AUTHOR CONTACT

Let's continue the discussion! Contact the authors, obtain the source data and sample code for this paper via Mary's author index on sasCommunity.org:

sasCommunity.org/discuss

ACKNOWLEDEMNENTS

The authors would like to thank Ethan Miller and Frank Ferriola for leading a great conference. Mary would like to thank Kirk for his friendship and mentoring, and for being a very fun person to collaborate with.

REFERENCES

Carpenter, Art, 2012, Carpenters Guide to Innovative SAS® Techniques, SAS Press, SAS Institute Inc., Cary NC.

Carpenter, Arthur L., 1997, "Better Titles: Using the #BYVAR and #BYVAL Title Options", presented at the 5th Western Users of SAS Software, WUSS, meetings (October, 1997) and published in the conference proceedings. Also presented at the 23rd SAS User's Group International, SUGI, meetings (March, 1998) and published in the Proceedings of the Twenty-Third Annual SUGI Conference, 1998.

DeFoor, Jimmy (2006). *"PROC SQL: A Primer for SAS® Programmers,"* Proceedings of the SAS Users Group International (SUGI) 2006 Conference.

Dorsey, Jason. The Gen Y Guy®, Best-Selling Author, and Keynote Speaker at the 2014 SAS Global Forum (SGF) Conference.

Hu, Jiangtang. Life Sciences Consultant at d-Wise Technologies in Raleigh, NC.

Lafler, Kirk Paul (2013). *PROC SQL: Beyond the Basics Using SAS, Second Edition*, SAS Institute Inc., Cary, NC, USA.

Lafler, Kirk Paul (2004). *PROC SQL: Beyond the Basics Using SAS*, SAS Institute Inc., Cary, NC, USA.

Lavery, Russ and Alejandro Jaramillo (2008). *"An Animated Guide: Using the Put and INput Functions,"* Proceedings of the NorthEast SAS Users Group (NESUG) 2008 Conference.

Li, Suwen et al. (2011), "*Utilizing PRELOADFMT Option with User-defined Formats to Create Summary Tables,"* Proceedings of the 2011 SAS Global Forum.

Morris, Christina (2011), "*Using Multi-label Formats to Create Subtotals in PROC TABULATE,*" Proceedings of the 2011 SAS Global Forum.

Rosenbloom, Mary and Lafler, Kirk (2012), "*Best Practices: Clean House to Avoid Hangovers",* Proceedings of the 2012 SAS Global Forum.

Rosenbloom, Mary and Carpenter, Art (2014), "*Are You a Control Freak? Control Your Programs – Don't Let Them Control You!"* Proceedings of the Western Users of SAS Software (WUSS) 2014 Conference.

Williams, Christianna S. (2006). *"Those Sneaky SAS® Functions: Beware of Unexpected Handling of Missing Data and Variable Lists,"* Proceedings of the NorthEast SAS Users Group (NESUG) 2006 Conference.

TRADEMARK INFORMATION