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.

<table>
<thead>
<tr>
<th>Status</th>
<th>Sex</th>
<th>Age at Start</th>
<th>Weight Status</th>
<th>Smoking Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dead</td>
<td>23</td>
<td>Overweight</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>2</td>
<td>Dead</td>
<td>41</td>
<td>Overweight</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>3</td>
<td>Alive</td>
<td>57</td>
<td>Overweight</td>
<td>Moderate (6-15)</td>
</tr>
<tr>
<td>4</td>
<td>Alive</td>
<td>39</td>
<td>Overweight</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>5</td>
<td>Alive</td>
<td>42</td>
<td>Overweight</td>
<td>Heavy (16-25)</td>
</tr>
<tr>
<td>6</td>
<td>Alive</td>
<td>58</td>
<td>Overweight</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>7</td>
<td>Alive</td>
<td>36</td>
<td>Overweight</td>
<td>Moderate (6-15)</td>
</tr>
<tr>
<td>8</td>
<td>Dead</td>
<td>53</td>
<td>Normal</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>9</td>
<td>Alive</td>
<td>35</td>
<td>Overweight</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>10</td>
<td>Dead</td>
<td>52</td>
<td>Normal</td>
<td>Light (1-5)</td>
</tr>
</tbody>
</table>
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.

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.

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).

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”.

Table of AgeAtStart by Sex

<table>
<thead>
<tr>
<th>AgeAtStart(Age at Start)</th>
<th>Female</th>
<th>Male</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millenials</td>
<td>806</td>
<td>578</td>
<td>1384</td>
</tr>
<tr>
<td>Generation X</td>
<td>1198</td>
<td>306</td>
<td>1504</td>
</tr>
<tr>
<td>Baby Boomers</td>
<td>669</td>
<td>722</td>
<td>1391</td>
</tr>
<tr>
<td>Traditionalist</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>2673</td>
<td>2391</td>
<td>5064</td>
</tr>
</tbody>
</table>
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.

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

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 as an option in the CLASS statement. Then we apply a format to our CLASS variable.

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

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.

```sql
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.

If we want 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.

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.

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

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;
```

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.

<table>
<thead>
<tr>
<th>Smoking Status</th>
<th>N Obs</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy (16-25)</td>
<td>443</td>
<td>443</td>
</tr>
<tr>
<td>Light (1-5)</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td>Moderate (6-15)</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td>Non-smoker</td>
<td>891</td>
<td>891</td>
</tr>
<tr>
<td>Very Heavy (&gt; 25)</td>
<td>237</td>
<td>237</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Smoking Status</th>
<th>N Obs</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-smoker</td>
<td>891</td>
<td>891</td>
</tr>
<tr>
<td>Light (1-5)</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td>Moderate (6-15)</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td>Heavy (16-25)</td>
<td>443</td>
<td>443</td>
</tr>
<tr>
<td>Very Heavy (&gt; 25)</td>
<td>237</td>
<td>237</td>
</tr>
</tbody>
</table>
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 so that we can output the dataset _FMTVALS containing the formatted values.

<table>
<thead>
<tr>
<th>Format name</th>
<th>Format value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATION</td>
<td>Generation I</td>
</tr>
<tr>
<td>GENERATION</td>
<td>Millennials</td>
</tr>
<tr>
<td>GENERATION</td>
<td>Generation X</td>
</tr>
<tr>
<td>GENERATION</td>
<td>Baby Boomers</td>
</tr>
<tr>
<td>GENERATION</td>
<td>Traditionalist</td>
</tr>
</tbody>
</table>

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. This will serve as our class variable in the PROC MEANS, below. We also specify the CLASSDATA= option and point it to _FMTVALS.

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

![](image)

<table>
<thead>
<tr>
<th>Analysis Variable : AgeAtDeath Age at Death</th>
<th>Label</th>
<th>N Obs</th>
<th>N Mean</th>
<th>Std Dev</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation I</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Millennials</td>
<td>232</td>
<td>233</td>
<td>57.5137339</td>
<td>7.7362711</td>
<td>36.0000000</td>
<td>80.0000000</td>
</tr>
<tr>
<td>Generation X</td>
<td>705</td>
<td>705</td>
<td>66.2245763</td>
<td>8.3590986</td>
<td>42.0000000</td>
<td>81.0000000</td>
</tr>
<tr>
<td>Baby Boomers</td>
<td>1050</td>
<td>1050</td>
<td>76.5144285</td>
<td>8.2400479</td>
<td>54.0000000</td>
<td>90.0000000</td>
</tr>
<tr>
<td>Traditionalist</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
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.

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. 1 By specifying NUMBER in the PROC SQL call, 2 we can get the row numbers output. We must use ODS OUTPUT to obtain the dataset ROWNUMBER that contains this value. 3 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.

<table>
<thead>
<tr>
<th>Row</th>
<th>Format value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-smoker</td>
</tr>
<tr>
<td>2</td>
<td>Light [1-5]</td>
</tr>
<tr>
<td>3</td>
<td>Moderate [6-15]</td>
</tr>
<tr>
<td>4</td>
<td>Heavy [16-25]</td>
</tr>
<tr>
<td>5</td>
<td>Very Heavy (&gt;25)</td>
</tr>
</tbody>
</table>

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, 4 grouped by SMOKING STATUS, 5 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). 6 We want to set missing values to ‘NA’ for the mean and standard deviation, which we accomplish by using the IFC function. 7 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. 8 Finally, we order the dataset by ROW, the variable that represents the logical order for the values of smoking. 9 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.
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,

```sas
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.

```sas
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.
```

<p>| Table of Sex by Smoking_Status |
|-------------------------------|---|---|---|---|
| Smoking_Status(Smoking Status) | Male |  |  |  |  |</p>
<table>
<thead>
<tr>
<th>Sex</th>
<th>Light (1-5)</th>
<th>Non_smoker</th>
<th>Tobacco Exec</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>20.00</td>
<td>60.00</td>
<td>20.00</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>20.00</td>
<td>60.00</td>
<td>20.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>20.00</td>
<td>60.00</td>
<td>20.00</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>
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:

```sas
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.

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.

```sas
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.

```sas
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;
  ods output ttests=ttests_trad;
  proc ttest data=heart
    (where=(put(AgeAtStart, generation.)='Traditionalist')) plots=none;
    class sex;
    var AgeAtStart;
    run;
%end;
%else %do;
  %put Nothing to do because there are only &levels levels.;
%end;
%mend;
options mprint;
%conditional;
```

We can place it into a %DO loop 1, which will only be processed if the conditions are met. Otherwise, we can send a message to the log. 2

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.

<table>
<thead>
<tr>
<th>label</th>
<th><em>TYPE</em></th>
<th>n</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>max</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>1991</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0: ± (0)</td>
</tr>
<tr>
<td>Generation I</td>
<td>1</td>
<td>1</td>
<td>233</td>
<td>57.67</td>
<td>36</td>
<td>69</td>
<td>57.6±7.8 (36.0 - 69.0)</td>
</tr>
<tr>
<td>Millennials</td>
<td>1</td>
<td>1</td>
<td>233</td>
<td>7.78</td>
<td>36</td>
<td>69</td>
<td>7.78±0.8 (69.0 - 81.0)</td>
</tr>
<tr>
<td>Generation X</td>
<td>1</td>
<td>1</td>
<td>233</td>
<td>7.86</td>
<td>36</td>
<td>69</td>
<td>7.86±0.8 (69.0 - 81.0)</td>
</tr>
<tr>
<td>Baby Boomers</td>
<td>1</td>
<td>1</td>
<td>233</td>
<td>76.3</td>
<td>54</td>
<td>93</td>
<td>76.3±8.2 (54.0 - 93.0)</td>
</tr>
<tr>
<td>Traditionalist</td>
<td>1</td>
<td>1</td>
<td>233</td>
<td>8.2</td>
<td>54</td>
<td>93</td>
<td>8.2±0.8 (54.0 - 93.0)</td>
</tr>
</tbody>
</table>

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

```sas
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”:

<table>
<thead>
<tr>
<th>label</th>
<th>n</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>max</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation I</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0: ± (0)</td>
</tr>
<tr>
<td>Millennials</td>
<td>233</td>
<td>57.67</td>
<td>36</td>
<td>69</td>
<td>69</td>
<td>57.6±7.8 (36.0 - 69.0)</td>
</tr>
<tr>
<td>Generation X</td>
<td>708</td>
<td>8.35</td>
<td>42</td>
<td>81</td>
<td>81</td>
<td>8.35±0.8 (81.0 - 81.0)</td>
</tr>
<tr>
<td>Baby Boomers</td>
<td>1050</td>
<td>76.3</td>
<td>54</td>
<td>93</td>
<td>93</td>
<td>76.3±8.2 (54.0 - 93.0)</td>
</tr>
<tr>
<td>Traditionalist</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0: ± (0)</td>
</tr>
</tbody>
</table>

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. 1 When N=1, the standard deviation will be undefined, so we will set it to “NA”. 2 And, in the case

```sas
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, ': ',
                           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*/
  else if n=1 then string =cat(n, ': ',
                              strip(put(mean,8.1)), ' ± NA ',
                              strip(put(min,8.1)), '- NA ',
                              strip(put(max,8.1)), ')');
  /*no records in the group*/
  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. PROC TABULATE is used in this simple example to get counts for each generational group, and then the results are set at the bottom of a dataset called STACKED. Furthermore, a global macro variable, &CALLS, is incremented each time the macro is called.

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

<table>
<thead>
<tr>
<th>Age at Start</th>
<th>N</th>
<th>subgroup</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Generation I</td>
<td>0</td>
<td>All Subjects</td>
<td>1</td>
</tr>
<tr>
<td>2 Millennials</td>
<td>1484</td>
<td>All Subjects</td>
<td>1</td>
</tr>
<tr>
<td>3 Generation X</td>
<td>2134</td>
<td>All Subjects</td>
<td>1</td>
</tr>
<tr>
<td>4 Baby Boomers</td>
<td>1591</td>
<td>All Subjects</td>
<td>1</td>
</tr>
<tr>
<td>5 Traditionalist</td>
<td>5</td>
<td>All Subjects</td>
<td>1</td>
</tr>
<tr>
<td>6 Generation I</td>
<td>0</td>
<td>Patients Who Have Died</td>
<td>2</td>
</tr>
<tr>
<td>7 Millennials</td>
<td>233</td>
<td>Patients Who Have Died</td>
<td>2</td>
</tr>
<tr>
<td>8 Generation X</td>
<td>708</td>
<td>Patients Who Have Died</td>
<td>2</td>
</tr>
<tr>
<td>9 Baby Boomers</td>
<td>1050</td>
<td>Patients Who Have Died</td>
<td>2</td>
</tr>
<tr>
<td>10 Traditionalist</td>
<td>0</td>
<td>Patients Who Have Died</td>
<td>2</td>
</tr>
</tbody>
</table>

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;

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

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

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:

1. &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=);
proc tabulate data=&dsn
   (where=(&where and not missing(&var))) out=_tryout;
   class &var sex/preloadfmt;
   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;
   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
   from &dsn
   where &where and not missing(sex);
quit;
%if %eval(&levels >= 2) and %eval(&sex = 2)%then %do;
   proc freq data=&dsn (where=(&where and not missing(sex)));
   tables sex*&var / chisq;
   output out = _chi (keep=P_PCHI) pchi;
run;
data _tryout2; merge _tryout2 _chi;
   merge _tryout2 _chi;
run;
%end;
data stacked (where=(not missing(factor)));
   format factor $200.;
   set stacked _tryout2 (in=innew);
   format titlewhere $200.;
   if innew then titlewhere="&titlewhere";
   if stat_male=''. (O./) then stat_male='0';
   if stat_female=''. (O./) then stat_female='0';
pval=ifc(not missing(P_PCHI), put(P_PCHI, pvalue6.4), '---');
run;
proc datasets lib=work memtype=data nolist;
   delete _;
quit;
%mend sum1;
using %DO loop processing. We obtain counts for each categorization using PROC TABULATE. The use of PRELOADFMT © 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 © 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). © We use a %DO loop © to check that the analysis variables have the appropriate number of levels, and if they do, then the statistical test is performed and © 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. © 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. © 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).

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 in Lake Forest, California. She has been using SAS for over 20 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 has been using SAS since 1979 and is consultant and founder of Software Intelligence Corporation. He is a SAS Certified Professional, provider of IT consulting services, educator to SAS users around the world, UCSD Extension professor, mentor, and sasCommunity.org emeritus Advisory Board member. As the author of six books including Google® Search Complete (Odyssey Press. 2014) and PROC SQL: Beyond the Basics Using SAS, Second Edition (SAS Press. 2013); Kirk has written more than five hundred papers and articles; been an Invited speaker and trainer at five hundred-plus SAS International, regional, special-interest, local, and in-house user group conferences and 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

ACKNOWLEDGMENTS
The authors would like to thank Sandra Minjoe and Eric Larson 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


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


TRADEMARK INFORMATION
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.