

Summarizing and Reorganizing Data

Jimmy DeFoor, Citi Card, Irving, Texas

The Basics

Summarizing data and reorganizing data are usually parts of every SAS program. The order can vary, of course, and, each may occur multiple times in the same program. Below is a simple example of a data step that both summarizes and reorganizes:

```
Data ZipSales
  Drop Sales Totsales Zipcode;
  Set Sales
    (where=(Zipcode in ('76116',
                        '76126',
                        '76108')));

  Retain Totsales 0;
  By Product Zipcode;
  If first.zipcode then
    Totsales = 0;
  /* Summarize sales */
  Totsales + sales;
  /* Reorganize rows into columns */
  If last.zipcode then
    do;
      If      zipcode eq '76116' then
        _76116 = Totsales;
      Else if zipcode eq '76108' then
        _76108 = Totsales;
      Else if zipcode eq '76126' then
        _76126 = Totsales;
      Output;
    end;
Run;
*;
```

This code summarizes sales by product and zip code and then reorganizes (transposes) the data so that it shows zip code sales as columns instead of rows.

Product	_76116	_76126	_76108
Paint	109,000	213,000	13,000
Brushes	3,000	7,000	500

Notice, however, that the order of the data isn't either in ascending order or in the order of the IF statements. That's because SAS creates additional variables as they are encountered in the data. Variable _76116 came first, so it was created before the other zip code variables. If that isn't the order that is wanted, the only way to change the order is with a Length statement.

```
Length Product $12
        _76108      8
        _76116      8
        _76126      8
        ;
Set Sales;
```

The Length statement must also precede the Set statement. If it follows the Set statement, it will set the order of only the variables not from the Set statement.

In this example, the length of the fields were set according to their length in the Sales data set, which was found with a Proc Contents. The values for the zip code fields were set according to the length of Totsales because it is those values they will contain. The resulting data set will have this order.

Product	_76108	_76116	_76126
Paint	13,000	109,000	213,000
Brushes	500	3,000	7,000

Since we knew the zip codes that would be in the data, specifying them in a Length statement was straightforward. But, what if we didn't know the zip codes that would be in the reorganized data set? Then, we would have to generate the Length statement from the data while the program is running. The next section explains how to do so.

Automating the Variable Order

First, create a data set with only one occurrence of each zip code that will be in the reorganized data set. A Proc Sort with *nodupkey* specified will do the trick.

```
Proc sort data = Sales(keep=Zipcode)
          nodupkey
          out = Unique;
  by Zipcode;
Run;
```

Then, write code that will create a Length statement. Precede the code with a Filename statement.

```
Filename length 'C:\temp\length.sas';
```

Then code a Data _Null_ step that will create the Length statement as if it were text in a file.

```
Data _null_;
  File length;
  Set Unique end=eof;
  By Zipcode;
  If _n_ = 1 then
    put 'Length Product $12 ' ;
  If first.zipcode then
    do;
      Field = ' '||zipcode;
      Put Field ' 8 ' ;
    end;
  If eof then
    Put ';' ;
Run;
```

This code will generate the following statements in length.sas, without the indentations.

```
Length Product $12
      76108      8
      76116      8
      76126      8
;
```

Next, add an Include reference just before the Set statement in the SAS data step shown at the beginning of this paper, exactly in the place where the Length statement should go.

```
Data ZipSales(drop=Sales Zipcode);
  %inc length;
  Retain TotSales 0;
  Set Sales;
```

While compiling the data step, SAS will retrieve the code in length.sas and build the pdv (program data vector) based upon the length statement. If the option source2 is turned on, SAS will show the included code in the log as if it were part of the data step.

Should multiple summary reports have to be created with the same column, the same process could be used to set the order of the variables in all data sets without having to sort the data again. For example, the process of creating the order of the zip code variables in the Product report could also be used in creating the order of the zip code variables for each of the following sort combinations:

```
By Product Media Zipcode;
By Media Zipcode;
By Zipcode;
```

The only difference in the process for each sort would be the variables specified in the 'static' portion of the Length statement that is generated when _N_ = 1 is valid. The first sort would have the variables Product and Media in the static portion of the Length statement. The last would not have any variables in the static portion.

If you find this technique interesting, but don't understand its function, think about the discomfort of viewing data in the wrong order in FS Browse, Proc Tabulate, or Proc Print. Setting a length statement usually fixes that problem for those procedures.

But what about Data _Null_ reports? Using a Length statement to set the order of the variables in the pdv won't do the trick. Instead, Put statements must also be generated using the same technique as used with the Length statement. Here are examples for writing both the headings and the data values.

Create Headings

```
Filename heading 'C:\temp\heading.sas';
Data _null_;
  File heading;
  Set Unique end=eof;
  By Zipcode;
  If _n_ = 1 then
    put "Put 'Product' +6" ;
  If first.zipcode then
    do;
      Field = ' '||zipcode;
      Put "" Field +(-1) "" +5" ;
    end;
  If eof then
    Put ';' ;
Run;
```

Write Variables

```
Filename vars 'C:\temp\vars.sas';
Data _null_;
  File vars;
  Set Unique end=eof;
  By Zipcode;
  If _n_ = 1 then
    put 'Put Product $12.0 +1' ;
  If first.zipcode then
    do;
      Field = ' '||zipcode;
      Put Field ' comma9.0 +1' ;
    end;
  If eof then
    Put ';' ;
Run;
```

Add to a Data _Null_ Report

```
Data _Null_;
  Title1 'Product Sales by Zip code';
  File Print heading=H;
  Set Sales;
  /* write sales values by product */
  %inc vars;
  /* write headings for each new page*/
  h:
    %inc heading;
    Put; /* blank line */
Run;
```

When the code is executed, the Data _Null_ step will look like this after the Includes are performed.

```
Data _Null_;
  Title1 'Product Sales by Zip code';
  File Print heading=H;
  Set Sales;
  /* write sales values by product */
  Put Product      $12.0 +1
      _76108 comma9.0 +1
      _76116 comma9.0 +1
      _76126 comma9.0 +1
  ;
  /* write headings for each new page*/
  h:
    Put 'Product' +6
      ' _76108' +5
      ' _76116' +5
      ' _76126' +5
    ;
    Put; /* blank line */
Run;
```

Add to a Data _Null_ to Create a CSV

The same approach can also be used to create the csv file that can be imported by spreadsheet software. The Data Set with the generated code would be:

```
Filename csv 'C:\output\out.csv';
Data _Null_;
  File csv;
  Set Sales;
  /* included code */
  Put Product \,
      _76108 \,
      _76116 \,
      _76126 \,
  ;
Run;
```

The code to create the Put statement would look like this:

```
Filename writecsv 'C:\temp\csv.sas';
Data _null_;
  File writecsv;
  If _n_ = 1 then
    put "Put Product \, ' " ;
  If first.zipcode then
    do;
      Field = \_||zipcode;
      Put Field \, ' ' ;
    end;
  If eof then
    Put \, ' ' ;
Run;
```

Using the technique on Put statements is more difficult than Length statements because of the need to space the fields and headings evenly. Still, with interactive SAS and writing the code to the output window, the correct code can usually be created fairly quickly.

This is a good technique for using interactive SAS to write Data Steps that generate code.

- Write the code you wish to create in the Notepad window and leave it there as a model.
- Stack the model code as shown to the left, with the static portion of the code on the first line. Place each dynamically created variable on a line by itself. Put the ending semi-colon on the last line.
- Wrap enough Data Step code around the model code to test it effectively. (You can skip this step if you believe it's easier to test the code after it is generated.)
- Write the code-generating Data Step, using the model code as a guide. Execute the code, but send the generated code to the list window by using a File Print statement.
- Compare the resulting code to the model code and modify the code-generating Data Step until they match.
- Replace the File Print with the true file destination and execute the code again. Examine the file to see whether the code was generated correctly.
- Implement the %inc reference into the correct location of the Data Step that will use the code. Now, test.

Add a Total to Each Line

Once you've mastered creating Put statements, you can easily add the code that generates a total for each line with a Sum function

```
Filename Rowtotal 'C:\temp\total.sas';
Data _null_;
  File Rowtotal;
  Set Unique end=eof;
  By Zipcode;
  If _n_ = 1 then
    put 'Total = Sum(of ' ;
  If first.zipcode then
    do;
      Field = '_'||zipcode;
      Put Field ;
    end;
  If eof then
    Put ');' ;
Run;
```

The %inc for Rowtotal could be placed in the any Data Step that has the new zip code fields, including the Data _Null_ that generates a report or a csv. If placed in the Data _Null_, it would go above the %inc that writes the data to the csv or report.

```
Data _Null_;
  Title1 'Product Sales by Zip code';
  File Print heading=H;
  Set Sales;
  /* sum zip code sales to a total */
  %inc rowtotal;
  /* write sales values by product */
  %inc vars;
  /* write headings for each new page*/
  h:
  %inc heading;
  Put; /* blank line */
Run;
```

Automating the Reorganization

Reorganizing (transposing) the data can also be automated in the same way as the Length and Put statements are generated. Use the same data set created from the Proc Sort that generated unique set of zip codes. Then write the code that generates the 'If Then Else' statements that were in the first Data Step example.

```
If      zipcode eq '76116' then
  _76116 = Totsales;
Else if zipcode eq '76108' then
  _76108 = Totsales;
Else if zipcode eq '76126' then
  _76126 = Totsales;
```

This code should generate the 'If Then Else' statements. Notice that Else is added only after the first line of data.

```
Filename reorg 'C:\temp\reorg.sas';
Data _null_;
  File reorg;
  Set Unique end=eof;
  By Zipcode;
  /* add an underscore to zipcode */
  Field = '_'||Zipcode;
  /* write else on all but the */
  /* first If statement.      */
  If _n_ gt 1 then
    put 'Else ';
  Put 'If Zipcode eq '
      '"' Zipcode +(-1) '"' ' then '
      Field ' = Totsales;' ;
Run;
```

Then, add the %inc reorg to your code in place of the transposing If statements.

```
Data ZipSales (Drop=Sales Zipcode);
  %inc length;
  Set AllSales;
  %inc reorg;
  Output;
Run;
```

Simple, right? Of course, not! And that's why SAS created Proc Transpose.

Using Proc Transpose

Using Proc Transpose often eliminates the need to generate code that will reorganize data, particularly when the reorganization involves only simple transpose. Here are the statements that would transpose the Sales data by zip code into the same column structure that we created in SAS data step above.

```
Proc Transpose data = Sales;
  /*write a new line for each product*/
  By Product;
  /* use zip codes for the columns */
  Id Zipcode;
  /* put sales values under columns */
  Var Sales;
  Output out= ZipSales;
Run;
```

I will discuss only a few features of Proc Transpose. The rest are well described in SAS documentation.

The By statement causes Proc Transpose to establish a new row of output for every change in the value of the By variable. In this case, when a new product grouping begins, SAS adds

new entries in the columns that it is creating. If it encounters an entry for a new column, it adds that column after the others that it has already created.

The names of the columns are defined by the values of the variable specified in the ID statement, which is Zipcode in this case. SAS uses the actual value of the variable for the column if it begins with an alpha character. But if it begins with a numeric character, it adds an underscore in front of the value (*just like we did in the code we generated*). It then places values under each column from the variable specified in the Var statement, which is Sales.

The resulting data set is exactly like the one created from the first data step in the paper, *provided that the data for each product and zip code combination has been summarized to a unique value.*

Product	76116	76126	76108
Paint	109,000	213,000	13,000
Brushes	3,000	7,000	500

If the data had not been summarized to a unique value for each product and zip code, Proc Transpose would have failed with an error message regarding duplicate values. SAS forces a failure because it wants the user to know that duplicate values exist and to decide the course of action that should be taken. If the user doesn't want to summarize the data, then the Let option can be employed to output the last occurrence of each grouping. This is exactly the same result of the first Data Step had the data not been summarized.

Since Proc Transpose cannot summarize the data itself, the summarization must be executed before the transpose. That allows the use of Proc Summary, which has additional benefits over summing the data in a Data Step.

Using Proc Summary

Proc Summary is now only another name for Proc Means. To understand its full functionality, read the Proc Means section of your Procedures Guide. The two differ only in that Summary has noprint as its default.

One major advantage of Proc Summary over a Data Step is that it can be used on either sorted or unsorted data. Here is the basic syntax of the sorted approach.

```
Proc Summary data = Sales;
  By Product Zipcode;
  Var Sales;
  Output out=TotSales sum=Totsales;
Run;
```

The purpose of each statement is obvious, with the exception of the Sum= option of the Output Out statement. That option specifies both the measure that is to be calculated and the variable name that will contain the measure. If a name is not specified - if it is left blank - the measure will be put into the same variable name as specified in the Var statement.

Proc Summary also generates two automatic variables, _Type_ and _Freq_. The variable _Type_ contains a number that describes the level of the summarization. More about this will be discussed later, but the value of _Type_ is zero when the summarization is the highest level possible. For summarization through a By statement only, the _Type_ value is always zero.

The _Freq_ variable contains the count of the number of rows that were used to calculate the total that was written to the Sum= variable. For example, if 12 accounts in zip code 76116 had purchased a brush, the value of _Freq_ would be 12 for that zip code.

The other method of calculating a measure is to use the Class Statement instead of or with a By Statement. The Class statement that would replace the By statement above is:

```
Class Product Zipcode;
```

When a Class statement is used, Proc Summary sets up summary fields for *each value of each variable* in the Class statement. If the number is not large, say 1000 or less, then the Proc Summary can be very quick because it is done in memory. But if it is large, say 100,000 or more, then the Proc Summary may have to page some of the summary fields off of disk storage. In that case, the processing time can take longer than sorting the data first and then summing with a By statement.

But Class statements have more value than speed. They also allow multi-level summaries to be created at the same time as single summary of the data. In this case, they would allow summaries to be created by not only Product and Zipcode, but also by Zipcode alone. In particular, this enables the building of a reports or csv file that would have a summary by

zipcode that preceded a detail listing of sales by product and zipcode.

Generating these summaries is simple since SAS added the Types statement. The Zipcode and Product totals and the Zipcode totals would be created this way.

```
Class Product Zipcode;
Types          Zipcode
              Product*Zipcode;
```

The resulting data set would look like this:

Product	Zipcode	Sales	_Type_
	76108	112,000	1
	76116	220,000	1
	76126	13,500	1
Brushes	76108	3,000	3
Brushes	76116	7,000	3
Brushes	76126	500	3
Paint	76108	109,000	3
Paint	76116	213,000	3
Paint	76126	13,000	3

As stated earlier, the Types variable has values related to the level of summarization. The numbers are calculated according to the order and number of variables in the Class statement. Knowing how to calculate those numbers is only necessary, however, if any un-summarized values of the Class variables are missing or blank.¹ Otherwise, the summed values can be identified by the presence of a missing value for numeric class variables and blanks for character class variables.

With this knowledge, user formats can be written that would replace the missing and blank values with words such as Total or All.

```
Proc Format;
  Value $total
    ' ' = 'Total';
  Value total
    . = 'Total';
Run;
```

A format reference would then cause 'Total' to be displayed when the summary line was shown in FS Browse or written to a report.

```
Format Product total12.0;
```

Product	Zipcode	Sales	_Type_
Total	76108	112,000	1
Total	76116	220,000	1
Total	76126	13,500	1

Furthermore, the existence of the missing or blank values in the By variables does not affect Proc Transpose's ability to transpose the data correctly. It will place the summary line at the top of the transposed data set; and, when displayed, the format would still show the word Total for that line.

Product	76108	76116	76126
Total	112,000	220,000	13,500
Brushes	3,000	7,000	500
Paint	109,000	213,000	13,000

Finally, Proc Summary will also create many other measures besides the sum. They include the median, mean, max, min, P10, P25, P75, and P90. The syntax would be:

```
Output Out = Totsales sum=Totsales 3
  mean=Avgsales median=Medsales
  max=Maxsales min=Minsales
  P10=P10sales P25=P25sales;
```

In Summary

- Proc Summary is an excellent method of summarizing data.
 - By statements for sorted data.
 - Class statements for unsorted data and for the creation of multi-level summaries.
 - Types statements specify the summaries that are needed.
 - User formats replace blanks and missing values for summary lines with Total or All.
- Proc Transpose easily reorganizes data from a vertical to horizontal format.
 - By variable statements set the rows of the new data set.
 - Var statements specify the variables that will be transposed.
 - Id variables set the columns of the new data set.
- Length statements, Sum functions, and Put statements generated from the unique values of the transposed variable can be used to automate the process of creating reports and csv files from the transposed data.

References

SAS Institute Inc., Advanced SAS Programming Techniques and Efficiencies, Course Notes, 1999.

SAS Institute Inc., SAS Language Reference Concepts, Version 8. (see <http://www.sas.com/service/library/onlinedoc/>)

SAS Institute Inc., SAS Procedures Guide, Version 8. (see <http://www.sas.com/service/library/onlinedoc/>)

Theilbar, Melinda, SAS Institute, Inc., Cary, NC, 'How to Transpose a Data Set. The Easy Way and the Hard Way'. (see <http://www.sas.com/service/techtips/bitsandbytes/transpose.html>)

Author Information

Jimmy DeFoor
Citi Card, Irving, Texas
Email: jimmy_defoor@afcc.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Missing or Blank Values

Unless you specify the missing option, Proc Summary does not create a separate line of the sum of the missing or blank values in a Class variable; but it does add those values to summary lines for the dimension. Thus, the measures generated for the higher-level summaries are accurate; they just don't equal the sum of the lower-level values displayed in the output data set for in that dimension.

If this isn't clear, return to the summary table on the previous page that display the `_Type_` variables. Subtract four Sales events worth \$100 from the Brushes row of 76126 and mentally assign them to four events that had a blank Product description. The value of that row is now \$100. Keep all other lines exactly the same. Do the sales for Brushes in the three zip codes now sum to the \$13,500 total for all Brushes?

Adding the missing option makes Proc Summary show the measures for the blank or missing values, but it also makes it impossible to select the summary lines for a dimension without knowing the value of the `_Type_` variable for each dimension. It's much simpler to eliminate the issue by assigning another value to the missing or blank values before they are summed. Changing missing values to zero (0) and blank values to 'blank' will often work for more tasks. If they won't, the `_Type_` values must be used to separate the levels.

`_Type_` values are calculated according to the order and number of the Class Variables.

Class Media Dept Product Zipcode;

To calculate the binary form of the `_Type_` variable, first determine which summaries are wanted. Review the Types statement.

```
Types                               Zipcode
                                Product*Zipcode
                                Dept*Product*Zipcode
                                Media*Dept*Product*Zipcode;
```

Next, recreate the summaries in comment form with zeroes where there are blanks and ones where there are variable names. These are the binary forms of the `_Type_` values associated with the requested summaries.

```
/* Types      0      0      0      1
              0      0      1      1
              0      1      1      1
              1      1      1      1 */
```

Now, use these values with bit testing and If statements to assign each `_Type_` to an order number that shows the level of summarization.

```
If      _Type_ = '0001'b then
  Order = 1;
Else if _Type_ = '0011'b then
  Order = 2;
Else if _Type_ = '0111'b then
  Order = 3;
Else if _Type_ = '1111'b then
  Order = 4;
```

The order numbers can then be used to select and sort the data for reporting purposes. It can also be used with user-built formats to indicate the level of the summaries.

```
Proc Format;
  Value Order
    1 = 'Zip code'
    2 = 'Product and Zip code'
    3 = 'Dept, Product, and Zip code'
    4 = 'Media, Dept, Product, and Zip';
Run;
```