Abstract

The SAS System may have had its roots in allowing statisticians to do data analysis, but that's now only one of the areas in which it excels. For most of us non-statisticians, our principal reason for using SAS software is the ease and flexibility it provides for simple, non-statistical data manipulation, aggregation, subsetting, and transformation.

This beginning tutorial will examine a few common ways of working with some basic SAS System procedures which produce output data sets that can then be manipulated further. Detailed but generally applicable examples will show you how to work more effectively and quickly with data manipulation procedures such as the FREQ, SUMMARY, MEANS, TRANSPOSE, CONTENTS, SORT, and UNIVARIATE procedures, using these as "building blocks" in your total SAS program.

I. Introduction

The Case of the Missing Mortar

There are many pitfalls for people beginning to work with the SAS System. Efficiency considerations. Missing values. Implicit arrays. Automatic temporary variables. Date manipulation. Formats. Macros. Thousands of pages of manuals. "Mindset" conversion from third-generation languages. (One wag suggested that the term "fourth generation language" was coined when people realized that it takes four generations to get an application up and running). When not to use defaults. When to reject multiple regression as an analytic method.

The pitfall that is the subject of this tutorial, however, is the relative scarcity of guidance available on actual SAS programming in a holistic sense. For the most part, the SAS manuals, especially since they are intended more as a general reference than as a programming instruction tool, focus on one procedure or one type of DATA step statement at a time. Other than the excellent and recommended SAS Applications Guide, the manuals tend not to discuss the issues that enter into fitting procedure and data step chunks together into a viable larger program. RELATIVELY few examples of multistep programs are provided in most SAS manuals. There are a lot of bricks out there in those manuals, but at times it's pretty hard to find the mortar to slap them together.

The cause of this lack stems, to some degree, from the history of the SAS System, which began its rich life as a batch-oriented statistical analysis tool, but which has now grown into a general purpose toolbox capable of virtually any kind of data manipulation. The original SAS System was designed with the ad hoc data analyst in mind, not really the programmer: PROC FREQ, for example, was primarily intended (and used) as a reporting mechanism, not as a programming tool or data manipulator in its own right. And besides PROC TRANSPOSE, that statement is true of the other procedures I will discuss: all were originally oriented towards providing output rather than providing a pure means of data manipulation. Over recent years, though, and increasingly since the advent of Version 5, the SAS System has vastly extended the capabilities of these and other procedures, making them truly useful programming tools. Systems are now being developed with SAS software that really have little or nothing to do with statistical analysis per se.

So in my view, one of the largest pitfalls for the SAS newcomer lies simply in learning to integrate, quickly and effortlessly, the most basic of the building blocks that the SAS System provides: data steps and simple procedures. Most beginning and intermediate SAS programmers have a reasonable amount of difficulty arriving at an effective mix of data steps and procedures. SAS makes it possible to arrive at solutions in extremely roundabout ways, making debugging considerably harder and programs usually longer than they have to be. This syndrome has been known to provide concrete examples to the firmly held belief of many mainframe systems administrators that SAS is generally inefficient.

It's easy, I think, for the SAS newcomer to be overwhelmed by the sheer quantity of the powerful tools available in the SAS System, and to experience some frustration at trying to get a drink from the fire hose. This tutorial will thus focus on the handful of procedures in the SAS System that are truly indispensable weapons in any SAS programmer's arsenal. It will demonstrate, moreover, how to use those procedures in a programming sense, rather than just in a report writing mode.

The Main Bricks of the SAS System

In previous SUGI papers (see the Bibliography), I have presented information about other basic SAS programming tools and issues, such as formats, macros, general efficiency concerns, and so on. This time, I will focus on the use, in programming, of the FREQ, SUMMARY, MEANS, TRANSPOSE, SORT, CONTENTS, and UNIVARIATE procedures. Each of these procedures, of course, can be (and most have indeed been) the subject of entire SUGI presentations; the focus here, however, will not be on detailing every last option of a given procedure, but rather on demonstrating how to plug these procedures together, like Legos or Tinker Toys, into your overall SAS program.

Most of what is done in most SAS programs has little or nothing to do with statistical analysis, and very much to do with data manipulation, simple aggregation, reshaping. The handful of procedures discussed in this tutorial, along with DATA step programming, represent the bulk of what most SAS applications need to fish from the vast sea of the SAS System. June Genis (1989) discovered, based on an analysis of SAS usage at a large university installation, that three quarters of the CPU time consumed in the many thousands of SAS jobs she examined was spent in DATA steps and in PROC SORT. From these results, she argued that it is "inappropriate . . . to categorize the SAS system as
a "statistical analysis system," at least as it is used at (her) installation."

Another very important reason for attaining fluency with these procedures (and one possible reason for the heavy slant in Genis' data towards use of the DATA step) is to avoid inadvertently reinventing them using contorted DATA steps. Occasionally, for specialized applications, it is true that a well-written DATA step can execute more quickly (since it serves more limited purposes) than the corresponding SAS procedure. PROC TRANSPOSE is perhaps the clearest example of that. However, using a bottled procedure instead of manually programming a solution has other advantages that outweigh the issues of execution speed: it guarantees (generally) predictable, bug-free results, easily followable and maintainable by other experienced SAS programmers. The SAS procedure call is also nearly always more concise than the DATA step.

Overall programming philosophy and practice, then, is what will be stressed here. We'll talk fairly fleetingly about each of the "main bricks" of the SAS System, but will concentrate on ways to build things with them. For further details on any particular PROC (for example, details on the variables contained in the output data sets generated by a PROC), consult the appropriate friendly SAS manual. Moreover, there are a number of data reshaping tools in SAS, most notably intricacies of the SET, MERGE, UPDATE, and BY statements, that will not be discussed here due to lack of space.

Before discussing each of the "bricks," though, we must first cover some rudimentary "mortar" concepts which have to be clearly understood before the bricks themselves can be used meaningfully. There are two kinds of "mortar" concepts that will be discussed:

- Creating output data sets from SAS procedures;
- Using simple match-merges to combine these output data sets with other data sets.

II. Output Data Sets from PROCs

This may seem obvious, but the principal "mortar" required when programming in SAS is the SAS data set itself. Virtually any SAS procedure, especially as of Version 6, can channel its results into a SAS data set, which may henceforth be manipulated in the same manner as any other SAS data set. Programmers who worked with SAS prior to Version 3 were known to go to great lengths to funnel the printed SAS procedure output to a flat file, which they would then read in to a SAS data set via a conventional SAS data step. This method was tricky and error-prone, especially since the precise layout produced by the procedure often changed depending on the data or, more certainly, on the version of SAS being run.

Now, however, creating an output data set from a given procedure is usually just a matter of consulting the manual to determine the correct syntax. Using that data set requires simply that you know its structure, which is often fairly fixed and is, of course, also documented in the reference manual. Although there are still some procedures that do not yet feature the ability to create an output data set, SAS Institute Inc. seems to be committed to adding that ability to virtually every procedure in the SAS System.

The most common, and perhaps most generally useful, example of an output data set created by a SAS procedure is the output of PROC FREQ. Given the appropriate options, FREQ will place its results in a SAS data set, which contains the variable(s) in the relevant TABLES request, as well as two new variables, COUNT and PERCENT. The output SAS data set is sorted (normally) in order of the values of the TABLES variable(s), which greatly facilitates merging. Most often, the output SAS data set (the result of a count or an aggregation) is merged in with another primary data set, and weights or percentages may then be calculated. Before showing examples of this kind of "mortar," we must first discuss the confusing variety of ways of producing an output data set from a given procedure.

Syntactic inconsistencies

Unfortunately, output data sets are not created in a syntactically consistent way across all SAS procedures. To some degree, the reasons for this inconsistency are historical, but it also stems from the need for greater amounts of detail to be specified for the procedure's output.

For some procedures, you will use an OUT= option contained in the same clause (i.e., before a semi-colon) as the PROC call itself. For example,

```sas
PROC SORT DATA=PERM.RECORDS OUT=RECORDS;
BY DATE;
RUN;
```

**FIGURE 1**

Other "brick" procedures that use the OUT= syntax in this manner are PROC TRANSPOSE and PROC CONTENTS.

With most of the "brick" procedures (no, unfortunately not all), however, you will create the output data set by having a separate OUTPUT clause, detailing specific options. This is generally true of those procedures that perform data aggregation, such as PROC MEANS, PROC SUMMARY, and PROC UNIVARIATE. These procedures allow a great deal of user specification as to what will be contained in the output data set. Usually, the OUTPUT clause specifies the name of the output data set, followed by one or more keywords that detail which statistics are desired in the new data set, as well as the variable names for these statistics. For example,

```sas
PROC SUMMARY DATA=PERM.RECORDS OUT=OUTCOST;
VAR COST;
CLASS TYPE;
RUN;
```

**FIGURE 2**

Other "brick" procedures that use the OUT= syntax in this manner are PROC TRANSPOSE and PROC CONTENTS.

156
Finally, PROC FREQ is something of a syntactic anomaly. It creates an output data set via an OUT= option, but that option appears on the TABLES clause following a slash, rather than on the PROC FREQ clause itself. Since only one data set can be created from a single PROC FREQ, and since a PROC FREQ can have multiple TABLES clauses (for different printed output tables), it is necessary for the OUT= clause to be specifically tied to a single TABLES clause. For example:

```
PROC FREQ DATA=PERM.RECORDS;
  TABLES ARTIST / OUT=ARTISTS;
RUN;
```

Finally, the IN= variables are mere temporary variables assigned by SAS at run time, and are not placed in the output data set. They take on a value of either 1 or 0, depending on whether the current observation's value of the BY variable(s) was in fact found in the corresponding input data set. In our example, it is certainly probable that we will have customer numbers that are not in our current ORDERS data set. We should, however, never have orders that contain a customer number not present in our customer information data set. Using the IN= variables, it would be possible to check for that error condition (i.e., with a statement such as IF IN_ORDER AND NOT IN_CUST) and take appropriate action such as creating a separate output data set of exceptions, etc.

Using Match-Merging for Mortar

Now let's look at an example that deals with an output data set, in this case the output from PROC FREQ.

PROC FREQ is probably the most important SAS System tool that creates a simple output data set, but it often receives too little attention from new SAS programmers. Again, most programmers become familiar with PROC FREQ chiefly as a reporting tool, one which can help them determine the nature of their data. But the very counts that are produced in the PROC FREQ output can be placed in a SAS data set, where they can be merged in with other data. The output data set from PROC FREQ will contain each of the variables listed in the relevant TABLES statement, plus the newly generated variables named COUNT and PERCENT. Note that Cumulative Frequency and Cumulative Percentage, although reported without the IN= variables, are often the most useful statistics in a data set, in this case the output from PROC UNIVARIATE, PROC SUMMARY, or other SAS tools, simply won't cut it here.

Note that both the data sets must be in the correct sort order, according to the variables on which you want to merge. The IF statement is required so that only those observations are kept whose customer numbers appear in both data sets. Although it is certainly not a requirement that you list the data set variables in the KEEP= option on the input or output data sets, it is highly recommended, so that you can see at a glance what is being combined with what. In most cases, following this practice will also provide you with fewer bugs and with faster execution time.

III. Simple Match-Merging

Besides mastering these basic building "bricks" of the SAS System, it is vital that the newcomer to SAS come rapidly to grips with basic ways of combining SAS data sets. Most importantly, the concatenating SET, the interleaving SET, and the simple match-merge (using IN= variables) are indispensable tools that must be understood. I have argued elsewhere (Kretzman, 1987) that all SAS programmers, whatever their level, should regularly reread the sections in the SAS manual that explain SET, MERGE, UPDATE, and BY, since the nuances in these tools are easy to forget for even the most experienced.

For this tutorial, let us briefly review the simple match-merge, which is the kind of data set combination most applicable to the rest of our subject. The following discussion is intentionally simplified, and is not intended to be anything close to a comprehensive treatment of even this one type of merge.

Simple match-merging typically splices together two data sets, using a BY statement with one or more variables. One of these data sets typically has more than one observation in each BY group, whereas the other data set has only one observation in each BY group. For example, a data set of sales orders (the inescapably cliched example) might contain many records for each customer number, since each customer may have placed more than one order. The customer information data set, on the other hand, would list each customer number only once, with the customer name, mailing address, and other information present as variables. When writing reports, you will want to link the orders with the customer information so that you have access to all the customer-specific data.

So, a simple match-merge on these two data sets would look something like this:

```
DATA ALLINFOKEEP=CUSTNUM CUSTNAME CUSTADR;
  MERGE ORDERS (KEEP=IN_ORDER KEEP=CUSTDATE PRODUCT QUANTITY)
  CUSTINFO(KEEP=CUSTNUM)
  KEEP=CUSTNUM CUSTNAME CUSTADR)
  BY CUSTNUM;
  IF IN_ORDER AND IN_CUST;
RUN;
```

FIGURE 4

```
DATA ALLINFOKEEP=CUSTNUM CUSTNAME CUSTADR;
  MERGE ORDERS (KEEP=IN_ORDER KEEP=CUSTDATE PRODUCT QUANTITY)
  CUSTINFO(KEEP=CUSTNUM)
  KEEP=CUSTNUM CUSTNAME CUSTADR)
  BY CUSTNUM;
  IF IN_ORDER AND IN_CUST;
RUN;
```

FIGURE 4
because the required information (both prescription counts and doctor counts for a given region) isn't available without some data manipulation.

Here, we need to use one of our basic bricks, plus a little mortar of a match-merge. PROC FREQ can easily count, using the master DOCTORS data set, the number of doctors in each region. But we're not interested in seeing the printed output of that FREQ! In fact, we don't really care much about seeing its results at all; we're interested just in using the results as the denominator for our average calculation, dividing total regional prescriptions by total regional doctors. So following the PROC FREQ that puts its results in a temporary output data set, we match-merge against the SCRIPTS data set and perform the necessary calculation. Here's the code:

```sas
PROC FREQ DATA=DOCTORS; TABLES REGION / OUT=REG_DOCS NOPRINT; RUN;
DATA REG_AVG(KEEP=REGION AVGSCRIP); MERGE SCRIPTS (KEEP=SCRIP) REG_DOCS(KEEP=DOCS) BY REGION;
ATTRIB AVGSCRIP LENGTH=4 LABEL='AVERAGE PRESCRIPTIONS PER DR';
 IF SCRIP; * keep even if region not in doctor file;
AVGSCRIP = SCRIP / COUNT; * compute average;
RUN;
```

The first step in the code above produces one observation for each region present in the master DOCTORS data set. Note that the NOPRINT option on the TABLES statement turns off the printed output that would normally result from a PROC FREQ. Note also that by default, the resulting SAS data set produced by PROC FREQ is sorted by REGION. Then, the second step uses a match-merge to put together the two data sets and compute the "prescriptions per doctor" average. One step omitted above is the potential PROC SORT required to get the SCRIPTS data set into REGION order, which might not be its natural sort order.

In this case, the subsetting IF in the DATA step intentionally keeps the even if region not in doctor file. If that happened, the AVGSCRIP variable would be missing, since there would be a missing value for the variable COUNT, and missing values in SAS calculations tend to propagate.

IV. The PROC SUMMARY Brick

In a similar manner, you can use the output data set from virtually any SAS procedure, merging it in by the appropriate BY variables with your other data.

Another fairly common need of SAS programmers is to obtain, in a data set, percentages of the total for a given variable. Of course, PROC TABULATE will produce those in printed form, but does not produce an output data set. If you need to use those percentages, to manipulate or subset your data, you need to use other bricks. In this case, using an output data set from PROC SUMMARY, combined with a match-merge against the original data, gives you the information you need to produce your percentages.

As an example, let's consider a simple data set of county information, CNTYINFO, containing four variables, STATE, COUNTY, POP, and LANDAREA. Our task is to place in a SAS data set any county that represents greater than ten percent or more of its state's population or greater than five percent of its state's land area.

Breaking this task down, we need to get more information into the CNTYINFO data set: POPPCT (percentage of state population) and LANDPCT (percentage of state land area). To do that, we will need to compute the total population and total land area within each state. PROC SUMMARY can do that easily, and if we place its results in an output data set, we can match-merge with CNTYINFO and compute the percentages we need. Then, subsetting the data by those percentages is an easy matter.

Here's the code:

```sas
PROC SUMMARY DATA=CNTYINFO KEEP=STATE POP LANDAREA NWAY; CLASS STATE; VAR POP LANDAREA; OUTPUT OUT=STATEINF(KEEP=TOTALPOP TOTALAREA) SUM=TOTALPOP TOTALAREA; RUN;
PROC SORT DATA=CNTYINFO; BY STATE; RUN;
DATA TOPGUYS(KEEP=STATE COUNTY POP LANDAREA POPPCT LANDPCT); MERGE CNTYINFO(KEEP=STATE CNTY POP LANDAREA) STATEINF(KEEP=TOTALPOP TOTALAREA) BY STATE;
ATTRIB POPPCT LENGTH=4 LABEL='Percent of State Population'; ATTRIB LANOPCT LENGTH=4 LABEL='Percent of State Land Area';
POPPCT = (POP / TOTALPOP) * 100;
LANDPCT = (LANDAREA / TOTALAREA) * 100;
IF POPPCT > 10 OR LANDPCT > 5;
RUN;
```

The results of a PROC SUMMARY with the NWAY option are all set in the sort order of the STATEINF variables, so no sorting of the STATEINF data set is needed. This example, more than the previous one, shows a fairly classic match-merge, where the primary data set (CNTYINFO) has many observations in it (i.e., one per county across all of America) and the other data set has a smaller number of observations in it (i.e., one per state). The "match-merge"
literally matches the two data sets by state. In this case, there is no need for IN= variables, since the smaller data set is a product of the first, and it is not possible for it to contain a state that isn’t present in the larger data set (or, of course, vice versa).

V. Other Aggregation Tools

In the same manner, any of the basic aggregation tools (FREQ, SUMMARY, MEANS, and UNIVARIATE) can be used to generate statistics on a class of your data, creating an output data set that is then merged in with your primary data.

One common question of SAS newcomers has to do with the differences among all these procedures. The answer to that question has changed over recent years. As of Version 6, PROC SUMMARY and PROC MEANS are now essentially aliases for one another, identical in capability, with the only difference being their default on printed output. Both of them permit a CLASS statement to substitute for the BY statement required by PROC UNIVARIATE, for example. With no need for a BY statement in the PROC itself, then there is no need to SORT the data set prior to calling the PROC. For programming (rather than reporting) purposes, I recommend using PROC SUMMARY, since you won’t have to worry about turning off the print default as with PROC MEANS.

PROC FREQ, on the other hand, is simply a counting procedure, with only a few rudimentary other capabilities such as the ability to apply weights to each observation. It performs no real aggregation, other than applying a format (which may collapse values) to the variables in the TABLES statement.

PROC UNIVARIATE, in terms of programming (rather than output) usefulness, should be fairly unnecessary in most circumstances, unless you need one of the many statistics it produces that are unavailable through any other PROC. For example, PROC UNIVARIATE can produce the median and mode for a variable.

In general, in many years of SAS programming for relatively non-statistical applications, I have been able to rely almost exclusively on PROC FREQ and PROC SUMMARY as my basic aggregation tools besides the DATA step. I recommend becoming absolutely familiar with those two.

VI. A Brick for Reshaping Your Data

PROC TRANSPOSE is arguably a very different procedure from any other in the SAS system. Its sole purpose is radical data reshaping. Its output is nearly always substantially different in form from its input, and can be quite hard to predict for the newcomer. Because of this, it may be the least understood basic procedure in the entire SAS System, and the one that is most often "reinvented" in the form of DATA steps.

Basically, the purpose of PROC TRANSPOSE is to take a SAS data set and make the variables into rows and the rows into variables. If your starting data set contains three observations, one each for Tom, Dick, and Harry, with the three variables being HEIGHT, WEIGHT, and AGE:

<table>
<thead>
<tr>
<th>NAME</th>
<th>HEIGHT</th>
<th>WEIGHT</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOM</td>
<td>73</td>
<td>192</td>
<td>33</td>
</tr>
<tr>
<td>DICK</td>
<td>68</td>
<td>260</td>
<td>42</td>
</tr>
<tr>
<td>HARRY</td>
<td>64</td>
<td>150</td>
<td>29</td>
</tr>
</tbody>
</table>

FIGURE 7

PROC TRANSPOSE will easily turn that data set around so that there are three rows, one each for height, weight, and age, with the three variables being TOM, DICK, and HARRY:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TOM</th>
<th>DICK</th>
<th>HARRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEIGHT</td>
<td>73</td>
<td>68</td>
<td>64</td>
</tr>
<tr>
<td>WEIGHT</td>
<td>192</td>
<td>200</td>
<td>130</td>
</tr>
<tr>
<td>AGE</td>
<td>33</td>
<td>42</td>
<td>29</td>
</tr>
</tbody>
</table>

FIGURE 8

The trouble is, everyone can understand that explanation, but it gets rapidly more complicated than that. What if, for example, you want each individual’s social security number preserved somehow in the transposition, as the variables T_SSN, D_SSN, and H_SSN? It’s not always obvious in a given circumstance how to program TRANSPOSE to do something like this. Sometimes, to get the result you need, you have to transpose a data set twice, with a short DATA step in between to do some minor tweaking. Most experienced SAS programmers I know need to experiment, for more complex situations, in order to follow exactly what TRANSPOSE is likely to do. But once you’ve figured it out for a given situation, the solution is typically far more concise and clean than it would be when manually coded in a DATA step.

So don’t be afraid to experiment, in general with the SAS System, of course, but especially with the radical data reshaping tool of PROC TRANSPOSE.

An example of PROC TRANSPOSE

Let’s look in greater detail at one more simple example, but one which is slightly more complicated than our example of Tom, Dick and Harry.

Imagine that we have a data set that contains information on state quarterly employment. It has the variables STATE, YEAR, QTR_1, QTR_2, QTR_3, and QTR_4. We need to “unwind” this into a more vertical form, so that we can graph it in various ways. Graphing procedures require this kind of "unwound" data, where every observation is marked by each of its categorical variables, in this case STATE, YEAR, and QUARTER. Here’s a small subset of the purely invented data:

<table>
<thead>
<tr>
<th>STATE</th>
<th>YEAR</th>
<th>QTR_1</th>
<th>QTR_2</th>
<th>QTR_3</th>
<th>QTR_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOM</td>
<td>2021</td>
<td>73</td>
<td>68</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>DICK</td>
<td>2021</td>
<td>192</td>
<td>200</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>HARRY</td>
<td>2021</td>
<td>33</td>
<td>42</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 9

159
A common way of solving this problem is to write a data step with an OUTPUT statement inside a DO loop. The DO loop marches across the four quarters for each input observation and outputs each quarterly figure to a separate observation in the output data set.

```
data loyalty_t(KEEP=STATE YEAR QTR EMPILOYMT);
  set loyalty(lc:keep=STATE YEAR QTR_1 QTR_2 QTR_3 QTR_4);
  do QTR = 1 to 4;
    EMPILOYMT = 'EMPILOY' + QTR - 1;
    output;
  end;
run;
```

I call this action "unwinding": making an essentially horizontal data set a vertical one. The equivalent operation, "winding," goes the other way, making the vertical data set horizontal, with all four quarters for a year in one observation as above. Typically, an "unwinding" DATA step is not terribly complicated. Its counterpart, however, which is a DATA step to "wind" the vertical data back up again in horizontal form, requires an array for each of the variables desired to be transposed, as well as RETAIN statements, FIRST.var processing, careful coding for exception handling, and so on. Especially for "winding" purposes, then, it often pays to experiment with PROC TRANSPOSE to see if it can do what you need done.

Successful experimentation will yield you this, using PROC TRANSPOSE:

```
data employ(KEEP=STATE YEAR QTR EMPILOYMT);
  set employ(keep=STATE YEAR QTR_1 QTR_2 QTR_3 QTR_4);
  array EMPLOYMT[4] QTR_1-QTR_4;
  do QTR = 1 to 4;
    EMPLOYMT = 'EMPILOY' + QTR - 1;
    output;
  end;
run;
```

PROC TRANSPOSE has a number of other options which I will not deal with here. In any case, if you experiment with it enough in a given situation, you will usually find that it has the power to turn most vertical data sets into horizontal ones, and vice versa, with a minimum of programming. I suggest you reference the excellent SUGI paper given on PROC TRANSPOSE (Passino, 1989) for more information and examples.

VII. Dueling Bricks: a Case Study

The SAS System is indeed a many-splendored thing, with quite a few ways of doing just about anything. The Institute itself has been known to misjudge just how to apply "mortar" in its own publications: the First Quarter, 1989 issue of SAS Communications dealt with the following problem:

"You want to count the number of unique occurrences of one value within each occurrence of another value."

The recommended solutions involved a SORT, then a DATA step using FIRST.var and LAST.var constructions. They also showed a solution that involved a SORT, a DATA step using FIRST.var, then a PROC SUMMARY. Finally, they showed a third solution, basically a variant on their second solution, incorporating some Version 6 features not previously available. All of these solutions work well, of course, and all are useful to examine in terms of how they show SAS procedures and DATA steps working together.

The original data in this example is as follows:

```
original Data
CAT PERSON CODE
CBT AJS 112
CBT HMM 112
CBT HHF 120
INS AJS 100
INS AJS 101
INS HMM 100
INS HMM 101
INS HHF 104
INS HHF 105
INS HHF 104
SMR JER 201
SMR JER 380
SMR JER 400
SMR LJS 201
SMR LJS 400
SMR LJS 201
TKT LJS 300
TKT LJS 302
TKT HHF 501
TKT HHF 300
VID AJS 202
VID AJS 235
VID HHF 235
VID HHF 202
VID HHF 202
```

In this case, PROC TRANSPOSE solves the unwinding problem a good deal more generically and smoothly than even the simple "unwinding" DATA step shown in Figure 10. Here's the code that produced the above data:

```
proc transpose data=employ
  out=employ_t(rename=(col1=EMPILOYMT));
  by state year;
run;
```
I will not go through each of their solutions in detail. (Please refer to the issue of SAS Communications if you have further interest). They are, however, reproduced below. Please note that I have preserved the style shown in the original examples.

/* First solution proposed in SAS Communications */
PROC SORT DATA=PURCHASE OUT=TEMP;
   BY CAT PERSON;
DATA COUNT;
   SET TEMP;
   BY CAT PERSON;
   IF FIRST.CAT THEN NUMBER = 0;
   IF FIRST.PERSON THEN NUMBER + 1;
   KEEP NUMBER;
PROC PRINT DATA=COUNT;
   TITLE 'Version 5 data step solution';
RUN;

/* Second solution proposed in SAS Communications */
PROC SORT DATA=PURCHASE OUT=TEMP;
   BY CAT PERSON;
DATA TEMP;
   SET TEMP;
   BY CAT PERSON;
   IF FIRST.PERSON THEN OUTPUT;
PROC SUMMARY DATA=TEMP NWAY;
   CLASS CAT;
   OUTPUT OUT=COUNT;
PROC PRINT DATA=COUNT NOOBS;
   TITLE 'Version 5 PROC SUMMARY solution';
RUN;

/* Third solution proposed in SAS Communications */
PROC SUMMARY DATA=TEMP NODUPKEY;
   CLASS CAT;
   TITLE 'Version 5 PROC SUMMARY solution';
RUN;

/* First breakdown by Category and Person */
PROC FREQ DATA=PURCHASE;
   TABLES CAT * PERSON / OUT=CATPERS NODUPKEY;
RUN;

/* Now breakdown by Category to get unique persons */
PROC FREQ DATA=CATPERS;
   TABLES CAT;
   TITLE 'Breakdown of Category by Unique Person';
RUN;

VIII. Using PROC CONTENTS

PROC CONTENTS is a different tool from the other "bricks" in this tutorial. Rather than dealing with the values contained in the data sets under analysis, it deals with variable names, labels, positions, and so on. This procedure is a metadata procedure: it deals with data about data. Hence the example we will discuss is different from the previous examples, although we still make use of an output data set that we then "feed back" into a subsequent data step, in a similar manner to what we have done in earlier examples. This example, moreover, uses the relatively advanced technique of using SAS to write a portion of a SAS program. If you think about it, it makes sense that a metadata (generating data about data) procedure such as this one would then facilitate the custom construction, programatically, of SAS routines.

In this case, our problem is as follows:

Stock data, organized by company symbol and date, reside in a collection of weekly data sets, named according to date with the convention yymndd, where yymndd represents the date of the Friday of that week. We want to construct a single data set from all the data sets available, but the names, as well as the total number of data sets, will by definition not be constant between runs of our program.

The key to solving this problem rests with discovering a way to obtain a list of current data sets that match the desired naming convention. PROC CONTENTS provides just such a capability, by facilitating the creation of an output data set. This output data set, by default when using the _ALL_ construct (as shown below in Figure 18), will contain one row for every variable that is in any data set in the PERM library. Our goal is simply to have a list of the member names in the PERM library, so we make sure to KEEP only the output data set variable which contains the member name (MEMNAME). Since each data set in the PERM library has a number of variables, each value of MEMNAME will, at first, appear in our output data set more than once. We therefore need to eliminate duplicate values of the MEMNAME variable, plus eliminate any values that do not conform to the naming convention.

The code below checks, for simplicity's sake, simply to see whether the data set name begins with a W. The result of this code fragment is a SAS data set, ALL, that contains a list of SAS data set names fitting the convention. Thanks to the NODUPKEY option on PROC SORT, each name appears only once in the final ALL data set. (The NODUPLICATE option in Version 5 would produce the same results, since there is only one variable in the data set).

/* Get all permanent library data sets into a data set */
PROC CONTENTS DATA=W* ADDMEM ALL_
   NODUPKEY;
RUN;

/* Now pluck out only the ones matching our convention */
PROC SORT DATA=ALL NODUPKEY;
   BY MEMNAME;
   WHERE MEMNAME =: 'W';
RUN;
Now we have our desired list of unique SAS data set names in the current PERM data base, all fitting our naming convention. (See Figure 19). We can now use that list to write a small chunk of SAS code that we will promptly call and execute via the SAS %INCLUDE statement.

Our desired code fragment looks something like this:

```sas
DATA ALLSETS:
   SET DATA.U900216
   DATA.U900223
   DATA.U900302
   DATA.U900309
   DATA.U900316
;
FIGURE 19
```

Since our data set ALL contains the list of data set names, it is a simple matter to write a short program to create this SAS code, writing it to a temporary flat file. (Note, in the code below, that specifying the output file is an operating-system-dependent construction).

* The DATA _NULL_ step produces flat file output only:
 data _null_
   file 'terr,,.out':
   set all(keep=memname)
      end=eof:
   if n = 1 then do;
      put 'DATA ALLSETS:';
      put 013 'SET' iii;
   end:
   put
      'DATA.' memname;
   if eof then
      put
         'Gl7 '; * finish off the SET statement;
   run;

FIGURE 20

Note that the code written by the DATA step (as shown in Figure 19) does not include a RUN statement. That means, for example, that following the %INCLUDE statement, it would be possible to have subsetting IFs or any other processing. The DATA _NULL_ step merely constructs the appropriate SET statements as a "front end" to the DATA ALLSETS step.

There are, as always, some caveats to this approach. For example, depending on the operating system in use, limits exist as to the number of data sets that may be listed in a SET statement. Also, the idea of "code writing code" tends to scare people, although it is an immensely valuable technique with the SAS System, which lacks real user-written procedures with passable parameters. In this case, short of hard-coding the data sets in the SET statement, there is no other way to solve the problem other than taking this basic approach. (Of course, you certainly could use the macro facility to create the code, rather than writing it out to a flat file. I have argued in other SUGI papers against overuse of that technique).

### IX. Conclusion

This beginning tutorial has merely touched on the surface of some very basic programming techniques for the SAS System. The reader is encouraged to explore these particular procedures and techniques further. A bibliography is provided below of useful papers and books pertaining to this subject. Happy bricklaying!

Questions and comments are very welcome. Please contact the author at:

Peter Kretzman
BR Associates, Inc.
2323 Eastlake Ave. E.
Seattle, WA 98102
Voice: (206) 328-0100
FAX: (206) 324-8886

The author may also be contacted at the Sea-SAS electronic bulletin board service (BBS), by setting your modem to N/8/1 (2400 baud or lower) and dialing:

Sea-SAS: (206) 324-9116
Hours: 7 a.m. to 4 p.m. Pacific time

Finally, the author can be contacted via Internet, using the address kretzman@uwavm.acs.washington.edu.

### X. Bibliography


NOTE:

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA. Lego is a trademark of Lego, AS.