Paper 4500-2016

Transformers: Revenge of the PROCs

Faron Kincheloe, Baylor University

ABSTRACT

Sometimes data are not arranged the way you need them to be for the purpose of reporting or combining with other data. This presentation examines just such a scenario. We focus on the TRANSPOSE procedure and its ability to transform data to meet our needs. We explore this method as an alternative to using "longhand" code involving arrays and OUTPUT statements in a SAS® DATA step.

INTRODUCTION

This is a tale of two challenges. As these challenges are considered, it is a high priority to design code that can adjust to certain changes in the data structure without rewriting the code or changing hard coded values. The first challenge involves financial aid data for college students stored in a single record per student. The data set contains spaces for up to 10 financial aid funds per student. Each record has a variable containing the fund code and another for the amount of aid. Funds are categorized according to the basis or other aspect of the award such as Athletic, Academic, or Loan. Each student's record needs to be enhanced by adding the fund category for each fund based on values from a separate table that contains the fund code and its corresponding category. The program code needs to be written so that it does not need to be modified if more than 10 funds are included in the future.

A sample of the original data is shown in Table 1. Columns 4 through 10 for amounts and funds are not shown for readability.

ID	amt1	amt2	amt3	fund1	fund2	fund3
120101	5000	.	.	1101		
120102	2250	1500	500	1011	1190	1228
120103	1500	.	.	A130		
120104	1500	1500	3750	1040	A125	1186

Table 1. Original Student Financial Aid Data

Table 2 shows a sample of the desired results. Again, several columns are not shown due to space limitations.

ID	amt1	amt2	amt3	fund1	fund2	fund3	type1	type2	type3
120101	5000	.	.	1101			Foundations		
120102	2250	1500	500	1011	1190	1228	Foundations	Skillset	Athletic
120103	1500	.	.	A130			Athletic		
120104	1500	1500	3750	1040	A125	1186	Military	Athletic	Corporate

Table 2. Enhanced Student Financial Aid Data

The second challenge is in working with jobs data downloaded from the Bureau of Labor Statistics. The original format is a spreadsheet that has had only minimal modifications before being imported into a SAS® data set. The file contains a column listing the U.S. states and territories. It also contains columns for each of the most recent 13 months with the number of 1000 jobs in each state for the respective month. The column names are a combination of all or part of the month name along with the year. The goal of this challenge is to incorporate the column names into the data so there are only 4 columns for state, month, year and number of jobs. The month and year need to be numeric values to facilitate sorting. This code should also be written so that little if any maintenance is required to adapt to changes of months and years that are included in the input data set.

A sample of the original jobs data downloaded from the Bureau of Labor Statistics is shown below in Table 3. Only the columns for the first six months are shown due to space limitations:

State	Aug2014	Sept2014	Oct2014	Nov2014	Dec2014	Jan2015
Alabama	80.0	80.2	80.1	80.3	81.0	80.7
Alaska	16.9	16.9	17.3	17.8	18.1	18.3
Arizona	122.9	122.6	124.4	126.5	126.7	129.6
Arkansas	45.7	45.9	45.2	46.5	47.4	49.0

Table 3. Original Data from Bureau of Labor Statistics

A sample of the desired data set is shown in Table 4 below:

State	month	year	jobs
Alabama	8	2014	80.0
Alabama	9	2014	80.2
Alabama	10	2014	80.1
Alabama	11	2014	80.3
Alabama	12	2014	81.0
Alabama	1	2015	80.7

Table 4. Transformed Monthly Jobs Data

MEET THE CONTENDER

The SAS® DATA step is probably the first thing most programmers would consider to meet the challenges outlined above. With the use of arrays, loops, variable lists, explicit output, by variables, retain statements and a merge, the DATA step is capable of meeting these challenges with certain limitations. Since these are not the main focus of this paper, each of these components will only be introduced briefly. You are encouraged to consult SAS® Help or other documentation for a more detailed explanation. One weakness of the DATA step that is a key issue in this discussion is that the DATA step provides no straightforward access to use column names as data nor manipulate them within the DATA step.

CHALLENGE 1: ADD FUND CATEGORIES

A number of transformations will be required to meet this challenge regardless of the approach that is taken. The first transformation is to produce a list of fund codes for each student, suitable for joining with fund category data. Second, the fund category values will be combined with the list of fund codes. The third transformation is to rotate the combined data back into a table that contains only one row per student. Finally, the fund category data from the last table will be combined with the original data to ultimately produce the desired outcome. The first and third transformations are the focus of this paper and alternate methods will be compared for these two transformations.

First Transformation

The DATA step shown below creates the narrow data set of IDs and fund codes that is needed to easily combine with the fund category values. For this particular part of our challenge, the DATA step is an efficient, easy to write solution with few if any drawbacks. A detailed explanation follows the code:

```
data student_funds(keep=ID fund_code i);
   set sgf.student_aid(keep=ID fund:);

*** Define an array that can be used to process each aid fund;
   array fund{*} fund:;
```

```
*** Create a macro variable that contains the number of funds;
   call symputx('fundnum',dim(fund));

*** Use a loop to process array and output non-missing values;
   do i=1 to dim(fund);
         if not missing(fund{i}) then do;
               fund_code=fund{i};
               output;
         end;
   end;
run;
```

The data set produced by this code only needs the fund code in order to match up with the fund category. However, the student ID is required for matching back with the original data. There must also be a way to ensure that the categories stay paired with the appropriate fund. In other words, the value in type1 must correspond to the fund in fund1. For this reason, an index variable "i" is kept in the output data set. It will be used for sorting later on in the process.

The KEEP option on the SET statement restricts the number of variables that are read in to maximize efficiency. The expression "fund:" is a variable list based on a name prefix. Since all of our fund code variables are named fund1 through fund10, this type of variable list can refer to all of these variables without listing them individually. It also has the advantage of automatically accommodating additional columns as long as they follow the naming convention like fund11.

In SAS® programming, an array is not an object or data type. Rather, it is an alias to a series of variables that facilitates applying common code across the variables. Loops are commonly used to access the variables in an array using the loop counter as the array index value. The ARRAY statement shown above assigns an array named fund to all the variables whose name begins with "fund" using the variable list feature described earlier. The value inside the brackets specifies the number of elements in the array. By using an asterisk, the array can be whatever size is dictated by the number of variables in the variable list. This also accommodates additional columns without needing to change the program.

Looking ahead to a future transformation, we will need to know how many funds or fund categories to expect. A macro variable named fundnum will be created to automatically store that number. Most macro processing is done during the compile phase before any data values are read. The SYMPUTX routine creates a macro variable during the execution phase using a data value available within the DATA step. The first argument to the SYMPUTX routine is the name of the macro variable. The second argument is the value to be assigned to the macro variable. In the code shown above, the DIM function is used to return the number of elements in the fund array.

The actual work of this step is done inside the DO loop. The DIM function is used here to establish the end boundary for the loop. The loop starts at 1 and processes each member of the fund array one at a time. Most students have fewer than 10 fund values and some have only 1. The IF statement prevents empty rows from being created when the fund variables have no value. The MISSING function is efficient for this task because it can be used in an IF statement and it works with either numeric or character variables. When a fund value exists, that value is assigned to the variable "fund_code". The OUTPUT statement causes the contents of the program data vector to be written to the output data set immediately instead of waiting until the end of the DATA step. As a result, as many as 10 rows of output data could be created for each iteration of the DATA step as it processes the data provided.

A sample of the results of this transformation is shown in Table 5 below:

ID	i	fund_code
120101	1	1101
120102	1	1011
120102	2	1190
120102	3	1228
120103	1	A130
120104	1	1040
120104	2	A125
120104	3	1186

Table 5. Student Data with Fund Codes Ready for Merging

Second Transformation

The next step is to combine the fund category data with the data created in the previous transformation. This can either be done using a MERGE step or an SQL join. The fund code is the common variable for matching the two data sets. If the MERGE is used, both data sets must be sorted by fund code first. Regardless of which method is used, the resulting data must be sorted or ordered by the variables "ID" and "i" to stay in sync with the original data.

A sample of the final results from this step is shown below in Table 6:

ID	i	fund_code	Category
120101	1	1101	Foundations
120102	1	1011	Foundations
120102	2	1190	Skillset
120102	3	1228	Athletic
120103	1	A130	Athletic
120104	1	1040	Military
120104	2	A125	Athletic
120104	3	1186	Corporate

Table 6. Original Data Combined with Fund Categories

Third Transformation

The DATA step shown below reads multiple rows per student (up to 10 in this example) and produces a single row per student with each value of Category being in its own column:

```
data wide_types (keep=ID type:);
   set fund_types;
   by ID;

*** Create array to manage fund type values;
   array type{&fundnum} $ 11;
   retain type:;

*** Initialize array on first value of ID;
   if first.ID then call missing(of type:);

*** Assign category to respective fund value based on position;
   type{i}=category;
```

```
*** On the last value of ID output the record;
   if last.ID then output;
run;
```

The data set produced by this code contains the student ID and the new fund category values that are stored in a series of variables or columns whose names begin with “type”. Again a variable list is used in the keep statement for simplification.

Adding a BY statement to a DATA step provides additional processing control based on a group of IDs that have the same value. As a general rule, when BY variables are used the data must already be sorted in that order. For each BY variable (ID in this example) SAS® creates two temporary variables in the program data vector to indicate the beginning and end of the BY group. In this example the variables are “FIRST.ID” and “LAST.ID”. Using Table 6 above for demonstration purposes, on the first row both FIRST.ID and LAST.ID are TRUE because there is only one row for ID 120101. On row 2, the value of FIRST.ID is TRUE indicating the beginning of the group for ID 120102. On row 4, the value of LAST.ID is TRUE indicating the end of the group for ID 120102.

In the first transformation the ARRAY statement was used to reference a group of variables that already existed. Another feature of the ARRAY statement is that it will create the variables if they do not exist. In this DATA step the ARRAY statement creates a series of variables beginning with “type” matching the name of the array. Since these are new variables, the number of members, their type and their length must all be specified. By using the macro reference “&fundnum” we retrieve the number of members saved from the first transformation to ensure that the number of types is the same as the number of funds. The dollar sign specifies that character variables be created. One shortcoming of this method is that we must know and hard code the length of the variables since the variables are created before the DATA step reads the input data.

The default behavior is for new variables to be reset to missing at the beginning of each iteration of the DATA step. In order to achieve the desired results, the value of type1 assigned on the first row for a student needs to be carried over to the second row and so on. The RETAIN statement prevents the values in the “type” variables from being reset to missing.

However, at the beginning of each student ID group, the values in the “type” variables need to be reset to prevent the values of one student from carrying over to subsequent students. When FIRST.ID is TRUE, the MISSING routine is called to clear out any existing values in the “type” variables.

The value of “i” that was kept from the first transformation is used as the index value to determine which member of the array is assigned the value of the Category variable on that row. This is how the fund types are kept in sync with the original fund codes.

The last statement of the DATA step checks to see if the record being processed is the last of a student ID group. Only on the last record are the contents of the program data vector written to the output data set.

Table 7 below shows a sample of the data set produced by the third transformation:

ID	type1	type2	type3
120101	Foundations		
120102	Foundations	Skillset	Athletic
120103	Athletic		
120104	Military	Athletic	Corporate

Table 7. Fund Categories in One Row per Student

Fourth Transformation

The fourth and final transformation is to combine the data from Table 7 with the original data. A simple match MERGE step will accomplish this with ID as the BY variable. The results of this transformation are shown in Table 2 above in the Introduction section.

CHALLENGE 2: CREATE MONTHLY JOBS BY STATE

Even though this challenge requires only a single transformation, the DATA step fails miserably. At the heart of the challenge is the need to correlate data values with the names of columns or variables. Unlike the previous challenge, the variables in this challenge do not begin with a common prefix or adhere to a standard pattern that works with variable lists or arrays. Additionally, the variable names vary depending on when the data are downloaded from the Bureau of Labor Statistics. The example data span from August 2014 through August 2015. If the data were downloaded at the time this paper was written, it would span from February 2015 through February 2016. A programmer can look at the column names and see what values need to be assigned, but there is no inherent way for the DATA step to do this automatically.

The DATA step shown below transforms the jobs data set from a "wide" table into the desired narrow format. This example shows only enough code to process the columns for two months. A similar block of code is required for each month:

```
data monthly_jobs;
   set sgf.tabled1x;
   * Specify variables to keep;
   keep state month year jobs;

   *Output a row for each month;
   if not missing(aug2014) then do;
         month=8;
         year=2014;
         jobs=aug2014;
         output;
   end;
   if not missing(sept2014) then do;
         month=9;
         year=2014;
         jobs=sept2014;
         output;
   end;
```

This is a relatively ordinary DATA step that should require little explanation. The KEEP statement is used to ensure that the original variables are not included in the output data set. As in the previous challenge, the MISSING function is used in an IF statement to prevent rows from being created with missing data. In this challenge, the variable name, value for month, and value for year must all be hard coded. The number of jobs from our "month" variable is assigned to the new "jobs" variable. The OUTPUT statement forces a new row to be written to the output data set for each month. This code does not adapt to changes at all. Everything must be manually recoded whenever there is a change in the input data set. The results of this transformation are shown in Table 4 back in the Introduction section.

INTRODUCING THE HERO

The TRANSPOSE procedure is a powerful alternative in solving the two challenges presented above. It may be underutilized because its features and options are not the most intuitive. The general form of the TRANSPOSE procedure that is presented in SAS® training material and documentation is very "busy" and difficult to visualize as code. This paper will forego that method and will attempt to teach by example. The most simplistic description of PROC TRANSPOSE is that it turns columns into rows and rows into columns. However, one of its most notable acts is that it also turns variable names into data. Some of its prominent features are listed below:

- Rotates all numeric variables by default.
- Uses an optional VAR statement to specify which variables to rotate.
- Creates a new variable called _NAME_ containing names of rotated variables.
- Creates a new variable called _LABEL_ if rotated variables had a label.
- Uses an optional BY statement to maintain groupings of values.
- Allows a PREFIX option for specifying the prefix to name columns created by PROC TRANSPOSE.
- Uses an optional ID statement to specify a suffix to column names.

The remainder of this paper will be dedicated to showing solutions to the two challenges using the TRANSPOSE procedure.

CHALLENGE 1: ADD FUND CATEGORIES

As demonstrated previously, four transformations are required to meet this challenge. This time PROC TRANSPOSE will be used instead of the DATA step to perform the first and third transformations. The second and fourth transformations remain essentially unchanged with only minor adjustments necessary to accommodate different data types.

First Transformation

The PROC step shown below creates the data set of IDs and fund codes that can be combined with the fund category values:

```
proc transpose data=sgf.student_aid(keep=ID fund:)
               out=student_funds(rename=(col1=fund_code)
                                 where=(not missing(fund_code)))
               name=i;
   by ID;
   var fund:;
run;
```

The first four lines of code shown above are all part of the PROC TRANSPOSE statement. DATA, OUT, and NAME are procedure options. As with most procedures, the DATA option identifies the input or source data set and OUT identifies the data set that will be created by the procedure. If no OUT option is specified, the output data set will be named Data1. Since both of these options specify data sets, most of the common data set options can be used. As with the DATA step solution, only the necessary variables are read in from the input data set and a variable list is used to identify the “fund” variables.

The data set options for the output data set are a bit more complex. Neither is absolutely necessary at this time, but both address issues that must be taken care of sooner or later. New variables or columns that are created by the TRANSPOSE procedure are given the name COL with a numeric suffix beginning with 1 and incremented sequentially in the order the variables are created. In this challenge only one variable is created and it contains the value of the fund codes. Since we anticipate matching on a variable named fund_code in another data set, the RENAME= data set option is used to change the name from col1 to fund_code in this output dataset. Because there are ten funds, ten rows are created for each student regardless of the number of “fund” variables that actually have a value. As many as nine of these rows could have missing fund codes for any particular student. The WHERE= option utilizes the MISSING function to eliminate the rows with missing fund codes from the output data set.

The NAME= option provides a convenient way of renaming the _NAME_ variable that is automatically created to hold the original variable names from the rotated columns. It is not necessary to change the name to achieve the results of this challenge, but to stay consistent with the DATA step code, avoid confusion later on, and demonstrate the use of the option, we are renaming the variable to “i”.

The BY and VAR statements harness the power of the TRANSPOSE procedure to produce the desired results of this transformation. Without them, the output would consist of a single row containing a separate variable for each student ID since ID is the only numeric variable in the input data. The fund

codes are character variables and would not be rotated by default. The BY statement causes ID to remain an independent variable that is used to group the transformed data together by each ID value. The VAR statement uses a variable list to specify which columns are to be rotated. A sample of the output is shown in Table 8 below:

ID	i	fund_code
120101	fund1	1101
120102	fund1	1011
120102	fund2	1190
120102	fund3	1228
120103	fund1	A130
120104	fund1	1040
120104	fund2	A125
120104	fund3	1186

Table 8. Fund Code Data Transformed Using PROC TRANSPOSE

Second Transformation

As before, the next step is to combine the fund category data with the data created in the previous transformation. It should be noted that the index variable "i" shown in Table 8 is a character variable instead of the numeric variable as was created by the DATA step method. As a result, the index variable will not be sorted in a true numeric sequence since there are more than nine variables. Both the SORT procedure and SQL ORDER BY clause will put fund10 between fund1 and fund2 instead of putting it after fund9. This must be taken into consideration when designing the second transformation. Otherwise, the third transformation may create inaccurate data or at least create the variables in an undesirable order. One way to solve this issue is to end the transformation with the SORT procedure shown below:

```
proc sort data=fund_types sortseq=linguistic (NUMERIC_COLLATION=on);
   by id i;
run;
```

The combination of SORTSEQ and NUMERIC_COLLATION causes the SORT procedure to sort the numeric suffixes as numbers instead of treating them as characters. Except for the difference in the variable "i", the output of this transformation is identical to that shown in Table 6.

Third Transformation

The PROC step shown below reads multiple rows per student (up to 10 in this example) and produces a single row per student with each value of Category being in its own column:

```
proc transpose data=fund_types
               out=wide_types(drop=_name_ _label_)
               prefix=type_;
   by id;
   id i;
   var category;
run;
```

As before, the DATA and OUT options specify the input and output data sets respectively. The _NAME_ and _LABEL_ variables refer only to the "Category" variable from the input data set, so they are of no use in this context. The DROP data set option is used to eliminate them from the output data set.

The PREFIX option replaces the default "COL" for the naming of new variables created by the procedure. This option can be used in conjunction with the ID statement to control the variable names. If it can be guaranteed that the input data set is sorted on student ID and "i" in the proper sequence of fund1 through fund10, then the PREFIX option alone is sufficient to create the variables type_1 through type_10 since

the order would match that of the original fund codes. The ID statement ensures that the new names are connected to the original fund variables even if the order is not correct. If the ID statement is used without the PREFIX option, the fund categories will be assigned to variables named fund1 through fund10. This is unacceptable because those names are already reserved for the fund codes. One way to solve this problem is to use a RENAME on the OUT option to change the variable names from "fund" to "type". By using PREFIX and ID together, the new variables are given the names type_fund1 through type_fund10. Since the last half of the name comes from the data, it will always be correct even if it is out of order. In other words, type_fund10 will always match up with the fund code in fund10 even if the variable type_fund10 is created immediately following the variable type_fund1. (This will happen if the sorting is not done correctly in the second transformation because PROC TRANSPOSE creates the columns based on the order of the rows from the input data.)

The BY and VAR statements serve the same purpose here as they did in the previous implementation of the TRANSPOSE procedure. A sample of the output from this procedure is shown below in Table 9:

ID	type_fund1	type_fund2	type_fund3
120101	Foundations		
120102	Foundations	Skillset	Athletic
120103	Athletic		
120104	Military	Athletic	Corporate

Table 9. Fund Type Data Restructured by PROC TRANSPOSE

Fourth Transformation

With the data shown in Table 9, the fourth transformation is identical to the transformation used for the DATA step process. With the exception of the names of the new "type" variables, the results of this transformation are identical to those shown in Table 2 above in the Introduction section.

CHALLENGE 2: CREATE MONTHLY JOBS BY STATE

The TRANSPOSE procedure can do most of the work of this challenge, and do so without any hard coding of values. As previously mentioned, one of the strengths of PROC TRANSPOSE is its ability to turn variable names into data. This is particularly useful for this challenge. The PROC step shown below transforms the jobs data into a narrow format that comes close to meeting the requirements of the challenge:

```
proc transpose data=tabled1x out=monthly_jobs;
   by state;
run;
```

There is little to be said about this code since it is almost as basic as a PROC TRANSPOSE can be. Only the BY statement has been added to group the results together by the value of state. All other variables are numeric and need to be included so there is no need for a VAR statement or other options. Partial results from this step are shown in Table 10 below:

State	_NAME_	_LABEL_	COL1
Alabama	Aug2014	Aug2014	80.0
Alabama	Sept2014	Sept2014	80.2
Alabama	Oct2014	Oct2014	80.1
Alabama	Nov2014	Nov2014	80.3
Alabama	Dec2014	Dec2014	81.0
Alabama	Jan2015	Jan2015	80.7

Table 10. Jobs Data Restructured by PROC TRANSPOSE

The data set contains the necessary information. However, it is not in the required format. The month and year are still stored in a single variable as character values. This prevents any grouping, sorting, or analysis by years and months. For example, a sort on the _NAME_ variable would put all of the April values together as the first months instead of January. Unfortunately, the TRANSPOSE procedure is unable complete this challenge without some assistance from its rival, the DATA step. The good news is that the DATA step can now do the necessary conversions without hard coding any specific values. The data step below creates the final version of the data to complete the challenge:

```
data monthly_jobs2;
   set monthly (rename=(col1=jobs));
   where not missing(jobs);
   keep state jobs month year;
*** Create numeric month based on Month/Year text values;
   month=month(input(cats(substr(_name_,1,3),
                          substr(_name_,length(_name_)-3)),monyy.));
*** Parse numeric year out of Month/Year text values;
   year=input(substr(_name_,length(_name_)-3),8.);
run;
```

The primary objective of this step is to parse numeric month and year values out of the text values in the _NAME_ variable. Some other cleanup is done here that could have also been done in the TRANSPOSE step. First, the col1 variable is renamed to jobs to give it a more meaningful name and to be consistent with the alternate process. A WHERE clause is used to filter out any records that have a missing value of jobs and a KEEP statement is used to eliminate unwanted variables.

The assignment statement for the month variable seems more complex that the typical assignment statement because several functions are nested together. The conversion of these values is complicated by the fact that the values do not have a uniform length. The length varies from 7 to 8 depending on the month because sometimes the month is designated with 3 characters and sometimes it is designated with 4. Since there is an informat that will convert a value like AUG2014 into a SAS date, part of the statement cleans up the values into this form. The SUBSTR function is used twice to separate out the month and year components. Then, they are put back together with the CATS function. Since the first digit of the year is always 3 characters less than the length of the string, the LENGTH function is used as an argument to the SUBSTR function that parses out the year. The INPUT function converts the result of the CATS function into a SAS date using the MONYY informat. Finally, the MONTH function is used to convert this SAS date into the desire numeric month value.

The assignment of the year value is a shorter version of the step described above. The year component is extracted in the same manner and the INPUT function is used to convert the character year to a numeric value. This data step produces the same output as shown in Table 4 from the Introduction section except that the variables are in a different order, which is unimportant for this challenge. By using this combination of PROC TRANSPOSE and DATA step, a process has been developed that can produce the desired results without further maintenance regardless of how many or which months and years are included in the data.

CONCLUSION

The TRANSPOSE procedure is not intuitive to use, but once it has been mastered it provides a powerful tool. Its ability to convert variable names into data values can be very useful. PROC TRANSPOSE can be used for complex transformations such as those in these two challenges and can also be useful when more basic restructuring of the data is needed. As demonstrated in the examples above, PROC TRANSPOSE can often eliminate the need for complex, repetitive DATA step code. It can also eliminate the need for hard coding values and make the code very adaptable to changes in data set structures. While complex challenges such as those can be solved with a series of DATA steps, PROC TRANSPOSE can reduce the number of transformations and often requires little or no maintenance in future iterations of the program.

REFERENCES

Bureau of Labor Statistics. 2015. “State and Metro Area Employment, Hours & Earnings.” Accessed October 13, 2015. http://www.bls.gov/sae/.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Faron Kincheloe
Baylor University
(254) 710-8835
Faron_Kincheloe@baylor.edu