Taming External Data: Lessons Learned in Importing External Data into a SAS Database

Kathryn Valdes, SRI International, Menlo Park, CA

Abstract

In social science research, we have met and mastered data in many forms. Occasionally, we have the opportunity to specify file formats, but most often we receive data that is someone else's structure and design. Files may arrive in ASCII, Microsoft Excel®, Microsoft Access®, FileMaker Pro®, dBASE®, or other file formats. Often, we are handed data collection instruments, such as forms and questionnaires, and must design a file structure to fit the forms. The ultimate goal is to build a SAS database for analysis that we can use with relative ease. Trial and error have taught us some hard lessons.

The following includes some macros that are routinely used in our research group. In this paper, I specifically address file formats on the PC; our current SAS license is SAS® System for Windows, Release 6.12.

Introduction

Most of our studies gather data from multiple sources. An important task in putting together data from external sources is to build a database in such a way that it is easy to use and manipulate. The aim is to have a group of files that can be merged, summarized, and analyzed with relative ease. If receipt of data is an ongoing process, we want to automate data construction tasks as much as possible. It is important to have a database that we understand and know, one where we can get to work with a simple contents procedure by our side. In short, we want to structure the data to fit our needs.

Our sources of data are paper or electronic. Our paper sources are generally questionnaires or data collection forms. Electronic formats are either ASCII flat files or files that need to be imported from an external PC database format.

Plan for Ease of Use

A thorough discussion of database design and structures is not possible in this space, but basically the concept is to keep it simple. Often, our databases consist of multiple files, one for each data source, and summary files of composite data. For example, student high school transcript data can have multiple records for each student, and a summary file can contain a single record per student with information such as cumulative GPA. Or a single record of school data may link to multiple students' records. The summary file may contain composite variables calculated from several sources. Even when using a summary or composite file, it is helpful to have the individual source files. It can get confusing when everything is thrown into one large ungainly file. Having separate files classified and stored by data source also makes keeping track of data elements easier.

Naming the variables is not always an option, but when it is we use a naming convention for variables based on the data collection instruments. We assign a letter for the data source—such as "T" for teacher questionnaire—and all variables from that source begin with that letter. If there are section letters in the instrument, the next letter in the name is the section letter. Finally, we use the question or item number. By setting up our database to correspond to the instrument numbering, we can put the instruments into a binder and have a quick and easy codebook to work with.

It is useful to create a system of identifiers to easily link different levels and sources of data. For example, we may have information about a student, a student's classroom and teacher, a student's school, a district, and a state. Each level would be nested and have a unique identifier: state ID,

district ID, school ID, teacher ID, and student ID. In addition to separate identifiers, I like to have a single identifier with a user-defined picture format that separates each part with hyphens. For example, a teacher's ID would appear as "04-115-09-28" (state-district-school-teacher), and a student in the teacher's classroom would appear as "04-115-09-28-54" (state-district-school-teacher-student). A single ID is useful for such tasks as sorting and printing information from the files.

When the data is sorted by the complete ID, it can be quickly merged by all levels. If the unit of analysis is student, for example, it is simple to store a single record for each state in a state-level file and merge information to the students' observations by state. The first sort level for the student is state if we have nested the ID numbers, and no additional sorting is necessary.

The key is to have a system set up with identifiers that give us the power to merge, summarize, or subset data as needed. In addition, creating files that consist of logical categories of data, such as source instruments, is a good foundation for a usable database. Finally, using meaningful and easy to recognize variable names makes working with the database less of a chore.

ASCII Data Macros

Much of our data begins life in hard-copy form. The amount and complexity of data entry dictates whether we send it to a subcontractor or enter it in-house. Our subcontractor provides us with column-specific ASCII data, or a flat file. We typically receive hard-copy instruments in batches. Mail questionnaires, for example, are always returned to us in dribs and drabs. We prepare them for data entry in batches and in return get an ASCII file for each batch from our subcontractor. The files are incrementally numbered. When we get a collection of batches, we load the data into SAS. And since repetitive processing cries out for macros, we have come up with a few.

At the head of each program, I routinely have macro variables for the library location, input and output files, and program name. In addition to these, for loading programs I add a few macro variables that specify the incremental program run number, the root name and location of the ASCII files, and the first and last batch numbers of the ASCII files.

```
%let program = svc rec ld.sas ;
   /* program name */
%let libloc = d:\neils\data\Svc Rec
   data;
   /* library location */
%let runnbr = 5 ;
   /* increment before ea progrm run */
%let fileout = sasdb.svcld&runnbr ;
   /* output file */
%let strtfil = 11 ;
   /* first ASCII file for this run */
%let endfil = 17 ;
   /* last ASCII file for this run */
%let frqfil = &libloc\nsvc ;
   /* root ASCII data files */
libname sasdb "&libloc" ;
   /* SAS data library location*/
libname library "&libloc" ;
   /* SAS format library location */
```

Using the macro variables defined above, a macro to specify the input ASCII data files creates my "filename" statements for me.

```
%macro specinp ;
   /* specify input data files */

****  define output files for reading ;
  %do I = &strtfil %to &endfil ;
    filename in&i "&&frqfil.&i..dat" ;
  %end ;
%mend specinp ;
```

The result is:

```
FILENAME IN11 "d:\neils\data\Svc Rec
  data\nsvc11.dat" ;
FILENAME IN12 "d:\neils\data\Svc Rec
  data\nsvc12.dat" ;
FILENAME IN13 "d:\neils\data\Svc Rec
  data\nsvc13.dat" ;
FILENAME IN14 "d:\neils\data\Svc Rec
  data\nsvc14.dat" ;
FILENAME IN15 "d:\neils\data\Svc Rec
  data\nsvc15.dat" ;
FILENAME IN16 "d:\neils\data\Svc Rec
  data\nsvc16.dat" ;
FILENAME IN17 "d:\neils\data\Svc Rec
  data\nsvc17.dat" ;
```

The next macro, GETIT, called from a datastep, loads each file.

```
%macro getit ;
  %do I = &strtfil %to (%eval(&endfil -
  1) ; ;
    %do j = (%eval(&i + 1)) %to
  (%eval(&i + 1)) ;
      infile in&i missover eof=read&j ;
        %ldit ;   /* the input stmnt */
        file = &I ;
        end ;
        return ;
      read&j:
    %end ;
  %end ;
  infile in&i missover ;
     %ldit ;
     file = &I ;
%mend getit ;
```

In this example, the macro loop is subscripted to start at 11 and end at 16 (see macro variable definitions for STRTFIL and ENDFIL above). An indexed macro loop creates a macro variable that can be used in the macro code. The first time through, macro variable &I is 11 and the macro variable &J is 12. The file containing the 11th batch of data is referenced in the infile statement. The "eof =read&j" on the infile statement creates a boolean value. While the EOF is false, the return statement causes the code between the infile statement and the return to be executed, i.e., it continues loading data from that file. EOF is true when the end of the file is reached and execution jumps to the label statement "read&j". A label statement is essentially a "go to" statement (i.e., jump out of the loop and start up again at that specified point). At the label, execution returns to the top of the macro loop until the 16th batch of data is read. The 17th file is read outside of the macro loop (&I is 17 at the conclusion of the macro loop) because it needs to be read without a return statement. There is also an assignment statement that assigns the batch number to the variable "file" to track the source of the data.

The first time through the macro loop, the result is:

```
INFILE IN11 MISSOVER EOF=READ12 ;
FILE = 11 ;
  [%ldit]  /* input statement */
RETURN ;
READ12: ;
```

The final time through the loop:

```
INFILE IN16 MISSOVER EOF=READ17 ;
FILE = 16 ;
  [%ldit]
RETURN ;
READ17: INFILE IN17 MISSOVER ;
FILE = 17 ;
  [%ldit]
```

When we have slightly different input statements for different sources of data, we can call separate input statements by putting them in separate macros. Below is an example with a file type "A" and a file type "B" requiring different input statements. A parameter is passed to the macro that designates which type of data is to be read.

```
%macro getit(ab) ;
    %do I  = 1 %to (%eval(&&nfiles&ab -
  1) ; ;
      %do j  = (%eval(&i + 1)) %to
  (%eval(&i + 1)) ;
          infile in&&ab.&i missover
  eof =read&&ab.&j ;
        %ldit&ab ;
        file  = "&&ab.&I" ;
        return ;
      read&&ab.&j:
     %end ;
    %end ;
    infile in&&ab.&i missover
  eof=end&ab;
     %ldit&ab ;
     file  = "&&ab.&I" ;
     return ;
    end&ab:
%mend getit ;
```

The macro LDIT can be a simple input statement or can be complex. For example, we had a data collection instrument that asked a set of questions about services provided in several different locations. We set up the data entry instructions so that there were a variable number of records for each observation, depending upon the number of locations the respondent received services. We specified that each record begin with a letter that indicated the service location. The macro LDIT conditionally called macros for each record type. This example of conditional loading used the trailing "@" feature in an input statement:

```
%macro ldit;
Input
  [input statements for first line of
   data]
  / @1 (rectype) ($1.) @ ;  /* goes to
  next line and reads type of record */
If rectype = 'B' then do ;
  %ldb ;                    /*
  conditionally does next input
  statement */
end ;
%mend ldit ;
```

Note: A single trailing "@" holds the data pointer on a data line for a single observation and will be released with the next SAS observation or input statement. A double trailing "@@" holds the pointer on the data line across SAS observations until released with the next input statement. Without the trailing "@", the next input statement would cause the pointer to move to the next line of data.

The macro GETIT loops through multiple files while hiding complex input statements. As a matter of routine (and well-advised caution), we print out five observations after loading the data and compare the new SAS data with the hard-copy instruments to verify that the input statement reads the data as expected.

PC Databases

We have received data in PC formats such as Microsoft Access, FileMaker, dBASE, and Excel, as well as simple delimited files. Excel can read many of the formats. I default to importing data into Excel, saving it as an Excel Version 5, and importing it directly into SAS using the "SAS/ACCESS Software for PC File Formats" module. Looking at the data in a spreadsheet gives me a clue as to what my data is (or is not). When given the choice, I request Excel files. Excel is my personal preference because it easy to review the data before processing it and—this is key—I have a pretty good idea of what will happen when it is imported into SAS.

The danger is in not knowing what has happened in imports and exports. For example, at one time, when dBASE files were imported into SAS, all missing data were recoded to a value of "0". We had to

come up with a dummy value for values that were actually "0" and recode them once the data was in SAS. This, fortunately, is no longer the case, but the lesson remains: know what happens when data is exported and imported. It is not always what you expect.

We use FileMaker extensively. When a database is set up for sharing, it can be hosted on one computer and accessed by multiple users in our network. Using a system of passwords, we can allow different levels of access, from complete power to create and destroy to read only. FileMaker can be set up so that it is easy to use for people who have limited computer experience.. We can write and use scripts, create calculated fields, use the relational power for portals (listing many to one), and create artistic layouts. By trial and error, I have found that the best way to get data into and out of FileMaker is a merge file (extension type ".MER"). A merge file is comma delimited with the first record containing the field names. For calculated fields, FileMaker puts out the current value in an export. For example, a field that calculates current age from the current date and the date of birth changes on a daily basis. The merge file would contain the value on the day the file was exported from FileMaker.

I load the comma-delimited file into Excel using the Import wizard. I specify "delimited" in Step 1 and "delimiters: comma" in Step 2. I can click finish and accept all data columns in the general format (default), or go to Step 3 and field-by-field specify general, text, or date format, or omit a column. Data columns in Excel can be formatted, but those formats are not carried over when the data is imported into SAS. For numeric data in which I want to retain leading zeros—a ZIP code, for example—I specify it as text. If I know that all of my ZIP codes are in a 5-digit format, I can import the data into Excel and SAS as a numeric and apply a SAS Z5. or user-defined picture format; or I can bring it in as text and forget about further formatting. If I have combinations of 5-digit and 9-digit ZIP codes, it is prudent to specify a text format.

Another big problem, for now, is the 8-character limitation in SAS for variable names. Field names in FileMaker can be whole sentences, as can column headers in Excel. If no changes are made, SAS will take the first 8 characters of the name. If field names are "Service Type 1" and "Service Type 2", SAS will give an error message that the name appears twice (SERVICE_ is the resulting name for both). The next logical step is to create 8-character names by inserting a row below the original names and creating new ones.

Once the names are reduced to 8 characters, the format becomes another issue. SAS defaults to the format in the first record. If the first occurrence of a variable is missing, SAS makes it a text variable. This is particularly amusing when it is added to an existing data file that has the variable defined as numeric. Routinely I add a row below the new field names with dummy values, "XXX" for character, "999" for numeric, and "7/7/77" for dates. I eliminate the dummy record after the data is converted.

Within the original Excel workbook, I create a copy of the worksheet excluding the initial names but including the new 8-character names and the dummy record with the formats. If I am going to receive multiple files in this same format, I will keep a header record in a separate Excel file to copy and paste it into the new files.

I have to admit to laziness, but I import data using the menu and recall the code to put it into any programs that I may need to run again. The following is code recalled from an interactive session using the SAS menu

driven-option for an Excel workbook with multiple worksheets:

```
PROC ACCESS DBMS=EXCEL;
CREATE WORK._IMEX_.ACCESS;
PATH='D:\VIRTUALH\spring
  99\Data\stdnt99.xls';
WORKSHEET='surveyS99';
GETNAMES YES;
SCANTYPE=YES;
CREATE WORK._IMEX_.VIEW;
SELECT ALL;
RUN;
DATA WORK.stdnt99;
SET WORK._IMEX_;
RUN;
PROC DATASETS LIBRARY=WORK
  MEMTYPE=ACCESS NOLIST;
DELETE _IMEX_;
QUIT;
PROC DATASETS LIBRARY=WORK MEMTYPE=VIEW
  NOLIST;
DELETE _IMEX_;
QUIT;
```

There are many types of files that can be brought into SAS directly or indirectly. The challenges are to be aware of how values and formats may change in the import/export process, as well as to be aware of limitations and variations between database packages, such as the maximum allowable characters in a string variable, maximum number of variables allowed, and how dates are calculated. In short, review the imported file and be certain that the contents of the SAS file reflect the original.

Conclusion

This is at best an overview of converting data from external sources to a SAS database and designing a database structure for ease of use. To summarize lessons learned, set the database up so that it is easy to manipulate and understand. If there is repetitive processing, take advantage of macros. Whatever the data source, examine the resulting SAS data carefully. Expect the unexpected; know what happens in the conversions. Even if everything is automated with macros and it is the *nth* time the data have arrived, review newly converted data before adding it to the master database.

The real trick to taming the data is to know where it has been, where it is going, and what will happen when it gets there.