

UPDATING SOME TIPS AND TECHNIQUES TO VERSION 7

Deb Cassidy, Cardinal Distribution, Inc., Dublin, OH

ABSTRACT

The SAS® Applications Guide, 1987 Edition stated that it is “a collection of examples that illustrate how you can use base SAS software to solve a variety of data entry, analysis, management and report generation problems.” It was written as a Version 5 and 6 update for the 1980 edition. Now that another 12 years have passed, and we’re using Version 7 or 8 with rumors of Versions 9 and 10, are the examples still valid? Do they run? Are there alternative ways to do the same thing that are more efficient? This presentation will walk through some of the examples that are covered as well as cover some other options that are relevant as you head into the next century.

This paper is based on Version 7 on a Windows operating system with Base, SAS/STAT, SAS/GRAPH, SAS/FSP, SAS/AF, SAS/ASSIST, SAS/EIS, SAS/ACCESS to PC file formats and SAS/ACCESS to ODBC. You may notice differences on other operating systems or with a different suite of products installed.

One thing that has changed in Version 7 is the terminology. SAS has changed terminology to be more consistent with other languages. Variables are fields, observations are records and data sets are tables. While I still use data set in this paper, I typically used the terms fields and records unless the point was easier to make using the old terminology.

CHAPTER 1 **READING DATA**

The SAS System provides you with a way to read data that is in almost any form that you can imagine. As I reviewed the guide, one of the first things I noticed was that many of the examples used a CARDS statement and instream data. Personally, I’ve rarely had that situation except

when creating “fake” data for writing papers for SAS conferences! The proliferation of pc software packages and databases available on all platforms has certainly changed your sources of data. Just look at the “SAS Access to ____” product line. The Version 7 menu gives you an IMPORT option. My choices were dBase files, several different versions of Excel and Lotus spreadsheets, Microsoft Access table, delimited file, comma separated values, and tab delimited files. This certainly makes life easier but you still need to know how to write an INPUT statement to cover the “odd” cases.

If you happen to be able to have your data within your program stream, your code will look something like this.

```
Data new;  
  input input-specifications;  
  *optional code to do something  
  with your data;  
  cards;  
  data lines  
  ;  
run;
```

There is one situation where many people have tried to use instream data only to discover it doesn’t work. You can not put the above code in a macro. You must read your data outside of the macro and then use the newly-created SAS data set in your macro.

Most people will find their data stored somewhere outside of their program. The guide didn’t really cover how to write an INFILE statement. There are two general forms:

```
INFILE file-specification  
  <options> <host-options>;
```

The *file-specification* can either be the name of the external file or it may be a *fileref* and used in conjunction with the FILENAME statement shown below.

```
FILENAME fileref <device-type>
'external file name' <host-
option-list>;
```

I prefer the former style when I have a short file name and few options while the latter usually provides better readability for longer file names or when using some of the file options.

The *external file name* gives the location of the file and is written in the style appropriate for your operating system. For example, a file on your hard-drive on your pc could be 'c:\my SAS files\sample.txt' while a file on a Unix system might be 'my SAS files/sample.txt'. The quotes are required. The blanks are permissible if they are permitted on your operating system.

If you are using a *fileref*, some words are reserved - either by the SAS System or by your operating system. For example, CARDS is reserved by SAS while CON, NUL, PRN, LPT0-LPT9 and COM1-COM9 are reserved under Windows. You also can't use some of these words for data set names.

There are too many options to cover in this paper but some of the ones you'll find helpful in Version 7 include options for reading Dynamic Data Exchange and Pipes. One question that does appear on SAS-L frequently is how to read comma-separated data that also has quotes around the text fields as well as missing fields. You simply need to use the delimiter and DSD options as shown below.

```
infile 'c:\my file.csv'
      dlm=', ' dsd;
```

The delimiter option tells SAS what the delimiter value is - in this case it is a comma. The DSD option actually serves two purposes. The options takes care of two adjacent delimiters and also removes the quotes from your text fields. Your delimiter doesn't have to be a comma. Tabs are also common but if you try to use the TAB key

on your keyboard you'll discover it doesn't work. You need to use the hexadecimal representation of TAB so your code would look like:

```
infile 'c:\my file.csv'
      dlm='09'x dsd;
```

Now, what was covered in the guide that is still handy? Let's say your data looks like the following:

```
CANTON OHIO 44701
CHICAGO ILLINOIS 60007
CYPRESS FLORIDA 32432
N.Y. N.Y. 10012
```

All fields are separated by at least one blank, there are no embedded blanks within a field, all fields have a value, all fields are either standard character or numeric values, and character values are no longer than eight characters.

You can simply write your input statement as

```
input city $ state $ zip;
```

The \$ indicates that field is character.

Now what happens if you have
WORTHINGTON OHIO 43085
added to your data.

The city is longer than eight characters.
Someone may think you just change it to

```
input city $12. state $ zip;
```

Well, it doesn't work. The guide had a slightly different example and just showed that you had the wrong values. When I used the above example in Version 7, I also had lots of information in the log. The data set ended up as

CITY	STATE	ZIP
CANTON OHIO	44701	.
CYPRESS FLOR	IDA	32432
N.Y. N.Y. 10	012	.

The log tells me I have invalid data for ZIP and that "SAS went to a new line when INPUT

statement reached past the end of a line.” What happened? Your input statement said to read the first 12 characters as the city. Given your data, you really want to read the first 12 characters OR up to the first blank - whichever comes first. The guide shows a very easy fix to your situation. Just use the colon format modifier. The colon indicates that the system should use the informat specified instead of assuming the length is 8. All the other assumptions still apply. Thus, your input statement becomes

```
input city : $12. state $ zip;
```

I also accidentally typed the \$12. without the decimal place. While the log gave me the same messages, the new data set wasn’t even close to what I wanted. Even using the colon modifier couldn’t help this situation so remember to always include the decimal when your are specifying an informat in the input statement. There are other circumstances where the decimal is not required such as on a LENGTH statement.

What if my next city is
LONG BEACH CALIFORNIA 90745
which has both the city and state that are longer than eight characters. You also have the problem that LONG BEACH contains a blank. Well, solving the state problem just requires using another colon and informat for the state field. Another format modifier is the ampersand &. This modifier will solve the embedded blank problem - IF you have at least 2 blanks separating the field and the next field. That is, you have to have at least 2 blanks between the city and state on each record. But what happens if you don’t? The guide didn’t cover this case but there is a way around this problem.

```
data one;
  infile
    'c:\my sas files\example.txt'
  missover;
input city :$12. state : $12.
      field3 : $12. field4;
length zip 8.;
if field4=' ' then zip=field3;
else do;
  zip=field4;
```

```
city=trim(city) || ' ' ||
state;
state=field3;
end;
drop field3 field4;
run;
```

The input statement creates a data set which looks like this:

CITY	STATE	FIELD3	FIELD4
CANTON	OHIO	44701	.
CHICAGO	ILLINOIS	60007	.
CYPRESS	FLORIDA	32432	.
N.Y.	N.Y.	10012	.
WORTHINGTON	OHIO	43085	.
LONG BEACH	CALIFORNIA	99999	.

The key is to use the MISSOVER option to read 4 fields even though some records only have 3 fields. (NOTE: In some cases, you may also need to use other options such as PAD to get this to work. PAD puts in blanks to make all the records appear to be the same length.) Once you have all your values, it’s just a manner of getting them shifted to be in the correct place. The length statement forces ZIP to be numeric since you had to read FIELD3 as character because of CALIFORNIA.

Someone will probably wonder what you do if you have to have Sioux Falls South Dakota in your data. The code will become even more complex but it can be done in a similar manner. You’ll need to be looking for five fields. You will also need to write some logic which checks for each state that has two words in the name, fixes the state and then process what is left to get the city. I won’t cover it here since I need to leave something for you to delve into in more detail.

What if your data was in nice, neat columns? You can use column input. This has several advantages over list input. You can specify which fields to read so you can skip fields you don’t need which will make your program more efficient. You can select any subset of data you want to read and read it in any order. Blank fields are treated as missing values. The guide also mentions that character values can be up to

200 characters long and can contain embedded blanks - Version 7 has increased the limit to 32,767 characters. (Of course, you might have a little problem viewing and printing a single field that long.)

One of the tips mentioned in the guide is the fact that you can put in decimals even if the data doesn't have them. For example,

```
input response 1-5 .2;
```

will assume there are two decimal places so a value of 11205 in your data becomes 112.05. If the value already has a decimal, the actual value will override the assumptions. Just make sure you type this as ".2" rather than "2." The latter means you want a 2-byte numeric value but you've already told the system the value was 5 bytes long. Don't worry about getting wrong data because the system will stop because of the error. This isn't one of the typographical errors where the system thought you had legitimate code.

Another example which still applies in Version 7 is the use of the trailing @ and trailing @@. Although the guide mentions the trailing @@, it did not include an example. The @@ basically says to repeat the input statement until you run out of data even if the data are not on the same line. For example, let's say someone used other software to do their data-entry and then saved the data as a text file. Catch is, this person didn't follow any rules in doing the data entry so you ended up with a file that looks like:

```
CANTON OHIO 44701 CHICAGO ILLINOIS 60007
CYPRESS FLORIDA 32432
N.Y. N.Y. 10012
WORTHINGTON
        OHIO 43085
```

Although it looks very disorganized, there is a pattern of city-state-zip so you can read this with

```
input city : $12. state : $12.
zip @@;
```

While some of you are thinking this never happens, there are those of us who have been

unfortunate enough to get data like this. If your example also contains Long Beach California 94075, you can even write code to work around that issue - however, it's not covered here because how to do it could be an entire paper by itself!

The guide shows the following example for using a trailing @. You have address data on students. If the student lives in the state, you also have a county provided. You want to create two data sets, one for in-state students and one for out-of-state students. The following code shows you how using the trailing @ to "hold" a data line until you do additional processing on it.

```
data instate(drop=state)
    outstate(keep=name street
    city state);
infile student;
input @47 state $2. @;
if state='NC' then do;
    input @1 name $12.
        @14 street $18.
        @34 city $11.
        @54 county $9.;
    output instate;
end;
else do;
    input @1 name $12.
        @14 street $18.
        @34 city $11.;
    output instate;
end;
run;
```

The code says to read just the state field. If it has the required state, then read the rest of the fields on the same line that are appropriate. Otherwise, read the fields that are appropriate for out-of-state. Write the fields out to the appropriate data sets.

Now, I've used this type of coding many times when my input lines are totally different depending upon the value in the identifying field. I saw another way that was much shorter for this example.

```
data instate(drop=state)
    outstate(drop=county);
```

```
infile students missover;
input @1 name $12.
      @14 street $18.
      @34 city $11.
      @47 state $2.
      @54 county $9.;
if state='NC' then
  output insate;
else output outstate;
run;
```

While this code has much less typing, it wasn't always more efficient from a processing standpoint. In this particular case, the results seemed to be inconsistent and definitely needed more testing with larger files to determine which one was truly more efficient.

This section has simply covered some of the highlights of reading data. The guide covers several more examples and there are many that can be handled in Version 7 that were not covered. The key point to remember is that there is a way to read data in almost any form.

CHAPTER 2

RECODING VARIABLES

The second chapter was entitled "Recoding Variables" which simply means to create new variables or modify existing ones. The example covered in the manual has age values ranging from 1 to 49. You want to group values by decade so ages less than 10 are in one group, 10-19 are in another, and so on. The actual data also has missing values and a value of 51 which is out of the range you were expecting.

The manual shows several different ways to accomplish your groupings. Methods 1-7 were provided in the guide. I also tested another method using formats as shown in Method 8. While they all run, there are some efficiency differences and you can also get different results!

```
**Method 1;
data group1;
  set one;
  if age>= 0 and age<10 then agegroup=0;
else if age>=10 and age<20 then agegroup=10;
else if age>=20 and age<30 then agegroup=20;
```

```
else if age>=30 and age<40 then agegroup=30;
else if age>=40 and age<50 then agegroup=40;
run;
```

```
**Method 2;
data group2;
  set one;
  select;
    when (age>= 0 and age<10) agegroup=0;
    when (age>=10 and age<20) agegroup=10;
    when (age>=20 and age<30) agegroup=20;
    when (age>=30 and age<40) agegroup=30;
    when (age>=40 and age<50) agegroup=40;
  otherwise;
end;
run;
```

```
**Method 3;
data group3;
  set one;
  if 0<=age<10 then agegroup=0;
else if 10<=age<20 then agegroup=10;
else if 20<=age<30 then agegroup=20;
else if 30<=age<40 then agegroup=30;
else if 40<=age<50 then agegroup=40;
```

```
run;
```

```
**Method 4;
data group4;
  set one;
  agegroup=40;
  if age<40 then agegroup=30;
  if age<30 then agegroup=20;
  if age<20 then agegroup=10;
  if age<10 then agegroup= 0;
run;
```

```
**Method 5;
data group5;
  set one;
  agegroup=10*(10<=age<20)+20*(20<=age<30)+
    30*(30<=age<40)+40*(40<=age<50);
run;
```

```
**Method 6;
data group6;
  set one;
  agegroup=10*(age>=10)+10*(age>=20)
    +10*(age>=30)+10*(age>=40);
run;
```

```
**Method 7;
data group7;
  set one;
  agegroup=int(age/10)*10;
run;
```

```
**Method 8;
proc format;
  value age
    0-09=0
  10-19=10
  20-29=20
  30-39=30
  40-49=40
;
run;
```

```
data group8;
  set one;
  agegroup=put(age,age.);
run;
```

Table 1 at the end of the paper shows the CPU time required for different data sets. The times for the first two data sets are in seconds and the time for the largest data set are in minutes. The last column shows the results obtained for running the smallest data set AFTER the largest data set in the same SAS session.

Notice that the SELECT statement was fastest on the largest data set yet was 4th and 6th on the small data sets. Method 4 using descending checks actually came in first in the smaller data set even though every record had to be processed for each IF statement. One of the advantages to using IF-THEN-ELSE logic is the fact that once a condition is met, the record does not need to be processed again.

Since I thought the results were odd, I ran the smallest data set again after I had finished the largest one. I did not close my SAS session before I did that. Notice I have different results. I also closed my SAS session and ran it again to get slightly different results. Many factors can influence your results including what else you may have running on your machine at the same time or what else you have already run in your SAS session. You should run efficiency testing multiple times to more accurately determine which method is best. In fact, based on posting to SAS-L, there are cases where the format method is most efficient even though it always took longer in my examples!

What about the different results I mentioned earlier? Remember you had missing values and an age of 51 in your data. Here are the results you get for each method. Determining which one is “correct” really depends upon what you were trying to accomplish with your program.

Method	Result for missing value	Result for 51
1	missing	missing
2	missing	missing
3	missing	missing
4	0	40
5	0	0

6	0	40
7	missing	50
8	missing	51

CHAPTER 3 RESHAPING YOUR DATA

This chapter started with three ways to subset your data. Two other methods are available which are often more efficient.

The first method is to use a subsetting IF statement as shown.

```
data males;
  set class;
  if gender='M';
run;
```

The second method uses a DELETE statement.

```
data males;
  set class;
  if gender^='M' then delete;
run;
```

The third method uses an OUTPUT statement.

```
data males;
  set class;
  if gender='M' then output;
run;
```

The fourth method is to use a WHERE clause on the input data set.

```
data males;
  set class(where=(gender='M'));
run;
```

The fifth method is similar but uses the WHERE statement as a separate statement;

```
data males;
  set class;
  where gender='M';
run;
```

Do you get the same results for each method? The way these examples were written you do but

you could have easily ended up with wrong results in Method 2. Many people would find it easier to write

```
if gender='F' then delete;
```

This works fine if you only have specific, known values in your data. But what happens if you have other values which may simply be a result of data entry errors? Once again, you should know your data and watch your logic.

Which should you use? Of the first three, you would typically want to use whichever is easiest to specify given your data. For example, if you variable had 20 possible values and you wanted to eliminate one, it would be easier to use the DELETE. However, if you just wanted to keep one, it would be easier to use a subsetting IF. The WHERE options are often more efficient especially if the data set is indexes. WHERE also permits you to use special operators such as “contains”. WHERE can not be used if you also use the OBS= option to process only part of your records.

A main reason for wanting to use an OUTPUT statement is when you actually want to create several different data sets such as:

```
data males females unknown;
  set class;
  if gender='M' then output
    males;
else if gender='F' then output
  females;
else output unknown;
run;
```

I prefer to always explicitly code for the “unknown” case so I know I’ve accounted for all my records.

If you do prefer using the SELECT statement, you can also handle it here.

```
data males females unknown;
  set class;
  select (gender);
  when ('M') output males;
```

```
  when ('F') output females;
  otherwise output unknown;
end;
run;
```

What if you want a record to be in more than one data set that you are creating? Simple - just use a separate IF-OUTPUT statement for each. Just make sure you do NOT use an ELSE in this case. Without the ELSE, each IF will be independent of the others so multiple IF conditions can be met. You can not use a SELECT in this case since this method stops processing a record once it has been assigned.

The guide also discusses subsetting variables. This is extremely useful if you have a number of variables in your data set that you really do not need for your job.

You can use DROP and KEEP statements or DROP= and KEEP= data set options. Here are examples.

```
data one;
  set class;
  drop height weight;
run;
```

```
data two;
  set class
  (drop=height weight);
run;
```

```
data three
  (keep=height weight);
  set class;
run;
```

DROP means to drop the variables specified and keep everything else while KEEP is the opposite. Some companies have strict rules that you must use KEEP to explicitly list the variables you are keeping. If there are no rules, most people typically use whichever choice means typing fewer variable names.

Should you use the option on the input data set, on the output data set or as a separate statement? If you use the option on the input data set, the

variables will NOT be available for processing within the data step. This improves efficiency but may not be feasible given your logic. Using the option on the output data set or as a separate statement means the variables are not dropped until the output data set is created.

RENAME can be used in a similar way to give a variable a new name. If you use the option on the input data set, you need to use the new name within your data step. If you use the option on the output data set or as a separate statement, the RENAME doesn't take effect until the new data set is created so your data step code must use the old name.

The guide also covers creating a repeated measures data set. It provides two examples.

```
data more;
  set tests;
  score=test1;
test=1;
  output;
score=test2;
test=2;
  output;
score=test3;
test=3;
  output;
keep name test score;
run;
```

This works fine but what if you have 10 or even 20 tests. The code certainly becomes quite lengthy. Fortunately, there is an easier way to write the code.

```
data more;
  array alltest(30)test1-test30;
  set tests;
  do test=1 to 30;
    score=alltest(test);
  output;
end;
drop test1-test30;
run;
```

Another useful example provided in the guide actually combines the above examples. What if you want to create two data sets from the same

original but you want different variables? Just use the following:

```
data htwt(keep=name height weight)
  agegender(keep=name age gender);
  set class;
  if height ne . then output htwt;
  if gender ne ' ' then output
    agegender;
run;
```

By the way, did I mention that Version 7 now permits data set names longer than 8 characters? A data set name of "AGEGENDER" would not have been acceptable in the older versions.

Combining Data Sets

To concatenate data sets is to stack them on top of each other. Let's say you started with separate male and female data sets and you wanted to create one data set with everyone. The code would be

```
data all;
  set males females;
run;
```

The guide says you can append up to fifty data sets this way. I successfully did 1,500 data sets in Version 7 and the actual limit is higher.

Another method not covered by the manual is the use of PROC APPEND. This procedure simply stacks the data sets together. In many cases, this will be more efficient since the system does not have to read the data set. However, if you need to do any additional processing to your data set you will need to use a data step.

You can also interleave data sets using a data step. If you data sets are already sorted and you wish to maintain that order without having to do another sort. Just using the following

```
data both;
  set males females;
  by age;
run;
```

Perhaps you need to pair up records in two data sets. That is, the first record in data set 1 is

paired with the first record in data set 2, the second with the second, and so forth.

The guide offered two ways to do this. In the example, two data sets have the same variables so a rename was used.

```
Data matchup;  
  set males;  
  set females  
    (rename=(name=f_name  
              age=f_age));  
run;
```

The other method uses a MERGE statement.

```
data matchup2;  
  merge males  
        females  
        (rename=(name=f_name  
                  age=f_age));  
run;
```

Notice there is no BY statement with this merge. There is a catch. You do not get the same results when there are a different number of records in the two data sets. When you use the two set statements, processing will stop when the end of the smallest data set has been reached. In the second method, processing will stop when the end of the largest data set has been reached. You will end up with missing values for the variables in the smaller data set for any records that didn't match up.

A test with small data sets of 20,000 records showed the first method used less CPU time.

SUMMARY

This paper covers some of the examples covered in the SAS® Applications Guide, 1987 Edition. These examples are still valid in Version 7 but you also have additional methods of doing the same thing. There are also other methods not covered here due to space. Although each version brings enhancements, it doesn't always totally replace the old way of writing code. It also shows that you can write code to handle just about any situation you might encounter.

BIBLIOGRAPHY & TRADEMARK

SAS Institute Inc. SAS® Applications Guide, 1987 Edition. Cary, NC: SAS Institute Inc., 1987.

SAS is a registered trademark of SAS Institute., Inc. in the USA and other countries.

CONTACT INFORMATION

Deb Cassidy
Cardinal Distribution
7000 Cardinal Place
Dublin, OH 43016
614-757-7136
deb.cassidy@cardhealth.com

TABLE 1 - CPU Time for Recoding Variables Example

METHOD	14,533 records	Rank	1,453,333 records	Rank	14,533,333 records	Rank	14,533 records AFTER #3
1	.17	6 (tie)	11.48	1	2:58.34	6	.22
2	.16	4 (tie)	12.40	6	2:46.30	1	.22
3	.16	4 (tie)	11.75	2(tie)	2:50.31	2	.15
4	.10	1	12.08	4	2:56.75	5	.17
5	.11	2	12.74	7	3:03.50	7	.22
6	.17	6 (tie)	11.75	2(tie)	2:50.49	3	.16
7	.15	3	12.25	5	2:54.17	4	.21
8	.59	8	14.92	8	3:04.50	8	2.24