

HOW TO "RETAIN" YOUR SANITY: SOME ANNOTATED EXAMPLES

Ronald Cody , Ed.D.
Dept. of Environmental and Community Medicine
Robert Wood Johnson Medical School

Ray Pass, Ph.D.
Columbia-Presbyterian Medical Center

One of the best ways to learn features of the SAS(r) system is to examine examples of programs where these features are used. This tutorial session will present a collection of examples focusing on the use of the RETAIN statement, but including others as well. By carefully explaining how these programs work, you should grasp aspects of the RETAIN statement that are traditionally difficult to learn and use. This paper is an excerpt from a larger collection of annotated examples currently being prepared. So, here goes!

Examples Using the RETAIN statement:

RETAIN is one of the more powerful and underused statements in the SAS System DATA step. The examples that follow should help you understand how this statement works and the subtle problems that can occur if you are not careful in its use.

Prior to the reading of each new record of data in a file building DATA step, SAS normally initializes each variable with a missing value. A RETAIN statement can be used however, to instruct SAS not to assign a missing value to a variable, but rather to "remember" its value from the previous observation. This value may then, of course, be changed within the DATA step in many ways.

Example RETAIN.1

Features: RETAIN, sum

We want a program to generate incremental subject numbers. The following program almost does the job.

```
*-----;
* INCORRECT PROGRAM;
DATA EXAMPLE;
  SUBJECT = SUBJECT + 1;
  INPUT SCORE1 SCORE2;
```

```
CARDS;
5 6
7 8
RUN;
*-----;
PROC PRINT;
RUN;
*-----;
```

The resulting data set would contain two observations:

OBS	SUBJECT	SCORE1	SCORE2
1	.	5	6
2	.	7	8

The reason this program didn't perform as desired is that for each iteration of the DATA step, SUBJECT is initialized to missing (.). When we add 1 to a missing value, the result is also missing. To correct this problem, we can use a RETAIN statement to initialize SUBJECT to zero and to prevent its being set to missing each time the DATA step executes. If we place the line:

```
RETAIN SUBJECT 0;
```

immediately after the first line of the program, the program will execute as desired.

```
*-----;
DATA EXAMPLE;
  RETAIN SUBJECT 0;
  SUBJECT = SUBJECT + 1;
  INPUT SCORE1 SCORE2;
CARDS;
5 6
7 8
RUN;
*-----;
```

```
PROC PRINT;
RUN;
*-----;
```

The resulting data set now looks like the following:

OBS	SUBJECT	SCORE1	SCORE2
1	1	5	6
2	2	7	8

We chose this simple example to demonstrate the use of RETAIN. We could have achieved the same result by using a 'sum' statement to replace the following lines:

```
RETAIN SUBJECT 0;
SUBJECT = SUBJECT + 1;
```

The 'sum' statement is one of those statements in the SAS System that you discover after you've been writing code for a while, and you sit back and say, "Hmmm. I wonder what other goodies there are that I don't know about." Here is the alternate code:

```
*-----;
DATA EXAMPLE;
  SUBJECT + 1;
  INPUT SCORE1 SCORE2;
CARDS;
5 6
7 8
RUN;
*-----;
PROC PRINT;
RUN;
*-----;
```

Notice there is no equal (=) sign in the "SUBJECT + 1;" statement. This 'sum' statement does a few things. It automatically initializes SUBJECT to zero for the first observation, and then implicitly RETAINS the previous value for each new observation before performing its function (in this case, adding 1 to the RETAINED value).

Example RETAIN.2

Features: RETAIN, _N_ (observation counter), DO group, DELETE, ARRAY, SUM function, KEEP

A simple example of test-scoring code is a good way to demonstrate a variety of SAS DATA step statements

in addition to RETAIN. (You may also want to actually score a test sometime!)

Suppose we have a data file in which the first record is the answer key and subsequent records contain student answers. The task is to read the answer key data in from the first record, read the student answers in from all the other records, calculate scores for each student answer by comparing key data to answer data and then derive summary scores for each student. Here is the code to accomplish this.

```
*-----;
DATA RESULTS;
  ARRAY KEY[5] $ KEY1-KEY5;
  * ANSWER KEY;
  ARRAY ANS[5] $ ANS1-ANS5;
  * STUDENT RESPONSES;
  ARRAY SCORE[5] SCORE1-SCORE5;
  * ITEM SCORES, 1=CORRECT, 0=WRONG;
  RETAIN KEY1-KEY5;
  IF _N_ = 1 THEN DO;
    INPUT (KEY1-KEY5)($1.);
    DELETE;
  END;
  ELSE DO;
    INPUT (ANS1-ANS5)($1.);
    DO I = 1 TO 5;
      SCORE[I] = (KEY[I] = ANS[I]);
    END;
    RAW = SUM (OF SCORE1-SCORE5);
    PERCENT = 100*RAW/5;
  END;
  KEEP RAW PERCENT;
CARDS;
ABCDE <----- Answer key
ABABA <----- Student answers
ABCD A <----- " "
RUN;
*-----;
PROC PRINT;
RUN;
*-----;
```

There is a lot going on here. The method we use is to first read in values for the KEY1-KEY5 variables (contained in the first record) and make them available for the comparison work which needs to be done for each succeeding student response record. We accomplish this by RETAINing the KEY1- KEY5 values into every record. We then read in the raw data for each student (ANS1-ANS5 = responses to individual

questions), compute a score for each question (SCORE1-SCORE5) and then derive two final scores for each student (RAW = number of questions answered correctly; PERCENT = percentage answered correctly.) Now, how do we treat the first record containing the answer key data, differently from the remaining records which contain student responses? The SAS System supplies the tools free of charge. For each iteration of a DATA step, the automatically created SAS internal variable `_N_` is incremented by 1. For the purposes of this example, we can think of `_N_` as an observation counter. When we are set to read in the first data record (the answer key in this example), `_N_` equals 1. The "IF ... DO; ... ELSE DO; ... END;" logic instructs SAS to perform different actions depending on the current value of `_N_`. When `_N_` equals 1, the first "DO ... END;" section of code (this is called a 'DO group') is executed and the values for KEY1-KEY5 are read in. When `_N_` is not equal to 1, other actions are taken. Values are read in for ANS1-ANS5 instead of KEY1-KEY5, and SCORE1-SCORE5 values are created from the RETAINED KEY1-KEY5 variables. But what about that DELETE statement in the first DO group? At this point you might be asking, "Why read in a line of data and then DELETE it?" Good question! Although each record of data is "read in" to SAS, it is not actually "written out" to the data set being created until the end of the DATA step. The DELETE statement instructs the program not to write out the first observation (`_N_` = 1). We really don't need it because it contains key data only, with no student answer data. Don't worry about losing the key data, however. Since the first record values for KEY1-KEY5 are RETAINED into all observations, they are always available when needed.

And now for that strange looking 'assignment' statement, "SCORE[I] = (KEY[I] = ANS[I]);". This is a complex statement. The parentheses are optional, and were put in for clarity. (The use of parentheses is a procedure that we highly recommend for beginners.) The rightmost equal sign could have been written as EQ since it is a logical operator, and not part of the 'assignment' statement syntax. The left equal sign however, must be written as is. The statement is actually saying, "Set the value of SCORE[I] equal to the result of performing the logical operation 'KEY[I] = ANS[I]'". The result of any logical operation is either true (1) or false (0). SCORE[I] is therefore set to 1 if KEY[I] is equal to ANS[I], and 0 if it is not. If this doesn't "sit well" with you, a more traditional (albeit less elegant) coding would be:

```
IF KEY[I] = ANS[I] THEN SCORE[I] = 1;
ELSE SCORE[I] = 0;
```

While we're on a tangent (this is actually a tangent to a tangent and if we keep going, we're bound to come full circle), here is another example which combines a logical expression with an arithmetic one. Suppose we are reading in birth date data in MMDDYY format, and have a separate variable (CENTURY) which has values of 1800 or 1900 denoting the century of birth. To compute a person's age (as of January 1, 1992), we could write:

```
AGE = ('01JAN92'D - DOB)/365.25 +
      100*(CENTURY EQ 1800);
```

where DOB is date of birth stored as a SAS date. Getting back to our tangent, if CENTURY = 1800, then we add 100*(1) to the age; if not, then we add 100*(0).

OK, back to the example at hand. Once we have scored each item, the raw score on the test (RAW) is simply the sum of the number of correct responses. Since all SCORE values are either 1 or 0, RAW is simply the sum of all SCOREs. We use the SUM function here to compute this sum. This is equivalent to:

```
RAW = SCORE1 + SCORE2 + SCORE3 +
      SCORE4 + SCORE5;
```

but much simpler to write. Also, while not relevant in this example, the SUM function (as well as the MEAN function) returns the sum (or mean) of the NON-MISSING arguments only. The last step is to calculate the PERCENT score. This is simply 100 times the raw score divided by the number of items on the test. The final tidbit in this already too long example is the KEEP statement. This is a basic storage efficiency technique which we recommend using in all instances. It instructs SAS to keep only the RAW and PERCENT variables in the RESULTS data set we built. All the KEY, ANS, and SCORE variables which were needed in intermediary steps are not kept around for posterity, and do not take up valuable space. The resulting data set looks like this:

OBS	RAW	PERCENT
1	2	40
2	4	80

Example RETAIN.3

Features: RETAIN, MOD function, LAG function

This example shows one of the pitfalls you can fall into when using RETAIN. In this example, we have recorded a subject's date of birth (DOB), and weight for each of three dates. However, the date of birth is recorded only in the first record. We want a data set that contains the subject's DOB in each observation. Here is the data file we wish to process:

FILE HTWT

SUBJ	DOB	WEIGHT
1	10/21/50	155
1		158
1		162
2	11/01/46	102
2		108
2		105
3	.	200
3	.	202
3	.	198

Note: DOB is missing for subject 3

Here is a program that has all good intentions, but doesn't give us exactly what we want:

```
*-----;
DATA NOWORK;
  RETAIN OLD_DOB;
  INFILE 'HTWT';
  INPUT @1 SUBJ $2.
        @7 DOB MMDDYY8.
        @17 WEIGHT 3;
  IF DOB NE . THEN OLD_DOB = DOB;
  * SET OLD_DOB TO DOB
  SO WE CAN REMEMBER IT;
  ELSE DOB = OLD_DOB;
  FORMAT DOB MMDDYY8.;
  DROP OLD_DOB;
CARDS;
.
.
.
RUN;
*-----;
PROC PRINT;
RUN;
*-----;
```

Here is the output:

OBS	SUBJ	DOB	WEIGHT
1	1	10/21/50	155
2	1	10/21/50	158
3	1	10/21/50	162
4	2	11/01/46	102
5	2	11/01/46	108
6	2	11/01/46	105
7	3	11/01/46	200
8	3	11/01/46	202
9	3	11/01/46	198

The program is supposed to work by first reading in DOB. If DOB is not missing (a new subject), then OLD_DOB is created and set to DOB. If DOB is missing (another record for the previous subject), then DOB is set to the RETAINED value of OLD_DOB. What went wrong? Since DOB was missing entirely for subject 3, the RETAINED value of OLD_DOB for subject number 2 was used. To prevent this from happening, we cannot merely rely on the presence of a value for DOB. Instead, we need to set the variable OLD_DOB equal to DOB each time we read data from what we know to be a new subject. How do we tell when we have reached a new subject? In this example, we always have three records per subject. We can therefore replace the "IF DOB NE ..." statement with the following:

```
IF MOD(_N_,3) = 1 THEN OLD_DOB = DOB;
```

This will work since the value of MOD(_N_,3) (this is read, _N_ modulo 3) will be 1 when _N_ = 1,4,7,etc. (i.e. every third record). A more general approach, one that will work with any number of records per subject, even unequal numbers for different subjects, would be to compare the current value of SUBJ to the previous value. The LAG function returns the value of its argument the last time it (the LAG function) was executed. If we execute the LAG function for every observation, its value is the value of the variable from the previous observation. The statement:

```
IF SUBJ NE LAG(SUBJ) THEN
  OLD_DOB = DOB;
```

will set OLD_DOB to DOB each time a new subject is encountered (even if DOB is missing).

Here is the code that works:

```
*-----;
DATA WORK;
```

```
RETAIN OLD_DOB;
INFILE 'HTWT';
INPUT @1 SUBJ $2.
      @7 DOB MMDDYY8.
      @17 WEIGHT 3.;
```

*CHECK IF WE HAVE A NEW SUBJECT
NUMBER. IF SO, SET OLD_DOB TO
THE DOB VALUE FOR THE NEW SUBJECT;

```
IF SUBJ NE LAG(SUBJ) THEN
  OLD_DOB = DOB;
  ELSE DOB = OLD_DOB;
  FORMAT DOB MMDDYY8.;
  DROP OLD_DOB;
RUN;
CARDS;
.
.
.
RUN;
*-----;
PROC PRINT;
RUN;
*-----;
```

Here is the correct output:

OBS	SUBJ	DOB	WEIGHT
1	1	10/21/50	155
2	1	10/21/50	158
3	1	10/21/50	162
4	2	11/01/46	102
5	2	11/01/46	108
6	2	11/01/46	105
7	3	.	200
8	3	.	202
9	3	.	198

One last comment about this example. We used the DROP statement here to drop OLD_DOB from the data set we built.

Example RETAIN.4

Features: RETAIN, Trailing @@, INPUT function, IN statement

The techniques used in this example can be applied to many situations in which you have unstructured data and you have a mixture of character and numeric data.

We have subjects assigned to groups A, B, and C. We record scores for these subjects and arrange the data like this:

```
C 303 102 150 B 202 C 300 B 450
400 399 420 A 289 280 278
```

As you can see, we follow a GROUP code by one or more scores. Our goal is to create a data set that looks like this:

GROUP	SCORE
C	303
C	102
C	150
B	202
C	300
B	450
B	400
B	399
B	420
A	289
A	280
A	278

Here is the program:

```
DATA UNSTRUCT;
  RETAIN GROUP;
  INPUT DUMMY $ @@;
  IF DUMMY IN ('A','B','C') THEN DO;
    GROUP = DUMMY;
    DELETE;
  END;
  ELSE SCORE = INPUT (DUMMY,5.);
  DROP DUMMY;
CARDS;
C 303 102 150 B 202 C 300 B 450
400 399 420 A 289 280 278
RUN;
```

The primary approach to this problem is to read all data values as character since we have a mixture of character and numeric. (We should point out that version 6.07 allows INFORMATS which can read both character and numeric and would provide a viable alternative to the code presented here.) We then test if we have read one of our GROUP codes (A,B, or C). If so, we set GROUP = DUMMY and, since GROUP is RETAINED, it will remain at the current value for subsequent observations until it is replaced by another value. When we read in a GROUP code, we DELETE

the observation since we only want to RETAIN the value at that time--we haven't yet read in a SCORE. When we read in a SCORE, we use the INPUT function to convert the character value to a numeric value. The GROUP and SCORE values are then added to the data set. Of course, we remember to drop DUMMY.

These are just a few annotated examples displaying some of the power inherent in a few of the statements available in the SAS system data step building process, focusing on the RETAIN statement.