

SESUG Paper BB-24-2017

Quality Control in SAS®: Checking Input, Work, and Output

Aaron Brown, South Carolina Department of Education

ABSTRACT

Part of being a responsible SAS® programmer is checking that one's programs work as intended. Errors and bugs can come from several causes. This paper goes through some tools for checking input data, one's work as one works on the data, and output data. It is unlikely that all of these tools will be applicable for every program, but the author hopes that this paper will give tools that other programmers can use as quality control resources. The author also hopes this paper will emphasize the need for quality control and checking assumptions. These tools utilize the COMPARE, CONTENTS, FREQ, MEANS, PRINT, and SQL procedures.

INTRODUCTION

This paper is divided into three sections. The first section discusses how to do quality control checks on data that is handed to the programmer (here called input data) and mainly deals with confirming any assumptions about the data. The second section discusses how to do quality control on one's own work within one's own programs. The third section discusses how to do quality control on any output the programmer may be generating, particularly with regards to generating data sets or files such as CSVs that will be input data for another programmer.

Due to the vast differences in data files and the multitude of tools within SAS, this paper obviously will not cover every quality control step that can or should be done. This paper contains an assortment of tools, but please consider what other tools may be necessary to truly check one's input, work, and output. In general, the author's suggestion is that, if there is something that the programmer can check, the programmer should check it.

For checking input data, this paper discusses confirming data elements, confirming valid values, confirming internal consistency, and confirming uniqueness of variables that should be unique.

For checking one's work, this paper discusses checking updating or creating variables, checking merges, and checking duplicate resolution. (See the References section at the end of this paper for some other papers on quality control that discuss other tools.)

For checking one's output data, this paper discusses doing a final check of all fields and visually checking the final output. Note that this paper does not discuss subjects like checking the appropriateness or correctness of output such as charts or graphs, but one could probably modify the tools mentioned here to provide output which one could check charts or graphs against.

Generally, when I use these tools, it is with datasets containing 20,000 to 300,000 records. For ease of discussion, we will use the sample dataset in Table 1 as our input data for the majority of the tools discussed here and assume it is already in our WORK library.

FName	LName	IDNum	Gender	Inventory	Rating
Jane	Doe	1001	F	5000	High
Jake	Long	1001	M	5000	High
John	Smith	0002	m	.	Unknown

Table 1. Contents of the MyData Data Set**VERIFYING INPUT DATA: CONFIRMING DATA ELEMENTS**

If you are fortunate enough to be given input data as a SAS data set, you can easily check the nature of the variables by using the CONTENTS procedure. (Use the VARNUM option if you want the variables listed in the order they appear in the data set instead of in alphabetical order.)

```
PROC CONTENTS DATA=mydata VARNUM; RUN;
```

Variables in Creation Order		
#Variable	Type	Len
1FName	Char	20
2LName	Char	20
3IDNum	Char	4
4Gender	Char	1
5Inventory	Num	8
6Rating	Char	10

Output 1. PROC CONTENTS: List of Variables using the VARNUM Option

The output displays the variable type (Char for character, or Num for numeric) of each variable. Sometimes a variable may appear to be numeric or character, but it is already stored as the other type, which can lead to errors or bugs later on in the program. You may need to program some work-around in a DATA step to render a character variable as numeric or vice-versa.

The PROC CONTENTS output also lists each variable's length. If a character variable has shorter length than anticipated, data may be truncated during later procedures or DATA steps. For example, our name variables are set to a length of 20, so any new names that were added to the data set would be truncated to 20 characters.

Also, attempting to merge two data sets by a variable that has different lengths can sometimes cause unexpected issues. For example, if we later tried to merge *mydata* to another data set by IDNum, but the other data set had an IDNum with a length of 10, we may experience unexpected outcomes. (Fortunately, SAS will print a warning to the log when this occurs.) In my experience, the likelihood of a mismatch due to this issue is minimal, but it is safer to manipulate the variables in a DATA step so that they have the same length.

If, however, the data is not provided as a SAS data set, more work is likely required. After reading the input data into SAS, check that the created data set has the correct number of observations (rows) and variables (columns). This information is also provided by PROC CONTENTS' output, in the upper right-hand corner, as shown in Output 2.

The CONTENTS Procedure

Data Set Name	WORK.MYDATA	Observations	3
Member Type	DATA	Variables	6

Output 2. PROC CONTENTS: Number of Observations (rows) and Variables (columns)

Depending on the method you used to import the data into SAS, you may find that the variables have default lengths which may or may not be appropriate for how you plan to use the data. You may also discover that some variables have been given an unexpected variable type. Two of the most common things I have experienced is 1) a numeric field being read in as character due to missing values, and 2) character values that appear numeric being read in as numeric.

As an example of the first, if we were given a csv of *mydata* which had a blank (not a period) for the missing Inventory value, PROC IMPORT would likely read-in that variable as a character field. A programmer using this input data would need to either modify the csv before reading in the data, change how they import the data, or fix the variable type after importing.

As an example of the second, if we used PROC IMPORT to import a csv of *mydata*, the IDNum would likely be read in as a numeric variable. This would also cause the leading zeroes to be dropped (e.g., 0002 read as 2), so you would need to add them back in during a DATA step (after doing the needed steps to re-render IDNum as a character variable).

If you know the length of the ID number, a do-while loop is useful for adding in leading zeroes, as shown below.

```
Do While (LENGTHN(IDNum) < 4); IDNum='0' || STRIP(IDNum); End;
```

VERIFYING INPUT DATA: CONFIRMING VALID VALUES

Generally, when we are given a data set, we have assumptions about what are valid values in each variable. It is important to check these assumptions, lest invalid values or unexpected blanks lead to errors or bugs.

The FREQ procedure is one of the easiest tools for checking what is in each variable, and can be used to check each variable in a data set. The MISSING option can be used to explicitly list the number of records with missing or blank values. For example:

```
PROC FREQ DATA=mydata;
  Tables
    LName FName IDNum Gender Inventory
  /MISSING;
RUN;
```

However, in many data sets, there are variables that change for almost every record, and the data set is too large to want to print each observation's name and ID number. Such would also apply for most dates. An easy way to spot-check using PROC FREQ is to create a counter variable in the data set and look at every *N*th record. For example, if we had 30,000 records in *mydata*, we may want to view every 3,000th record to spot-check for valid names and ID numbers. We could then also look at each record's gender and inventory, as did in the earlier example.

```
DATA mydata; SET mydata;
  ctr=_n_;
RUN;
PROC FREQ DATA=mydata (WHERE=(MOD(ctr,3000)=0)); *mod means modulo;
  Tables LName FName IDNum/MISSING;
RUN;
PROC FREQ DATA=mydata;
  Tables Gender Inventory/MISSING;
RUN;
```

Another way to check for valid values is to create flags within a DATA step. For example, if we knew all IDNum values should have a length of 4, code such as the following could identify and print any values with errors. We could also check if Gender contains any unexpected values.¹

```
DATA mydata; SET mydata;
  IF LENGTHN(IDNum) NE 4 THEN errflag=1;
  If Gender NOT IN ('M','F') then errflag=2;
RUN;
PROC PRINT DATA=mydata; WHERE errflag NE .; RUN;
```

Using either of these methods, we can identify that one record has an invalid value for Gender. Output 3 shows the frequency table and Output 4 shows the error flag printout.

Gender	Frequency	Percent	Cumulative Frequency	Cumulative Percent
F	1	33.33	1	33.33
M	1	33.33	2	66.67
m	1	33.33	3	100.00

Output 3. PROC FREQ: Frequency Table for the Gender Variable

Obs	LName	FName	IDNum	Gender	Rating	Inventory	ctr	errflag
3	Smith	John	0002	m	Unknown	.	3	2

Output 4: PROC PRINT: Error Feedback Due to Invalid Gender Value

Depending on the nature of your input data, you can know what to look for in a PROC FREQ printout or what data quality issues to flag. Some, such as a Gender value being lower-case, can easily be fixed (e.g., `Gender=UPCASE(Gender)`), while others may require you to investigate the discrepancies or ask for corrected input data.

Note that these checks can also be useful double-checks that all variables were read-in properly. If you find a variable with all blanks but you know it had values in the input data, one likely possibility is that character data was misread as numeric and rendered all-blank.

VERIFYING INPUT DATA: CONFIRMING INTERNAL CONSISTENCY

Sometimes you expect input data to follow certain rules about internal consistency. Using our example, let's say we know that any observation with an Inventory of 4000 or higher should have a Rating of 'High', any with an Inventory below 4000 should have a Rating of 'Low', and any with a missing Inventory value should have a rating of 'Unknown'.

¹ The FIND and ANYALPHA can also be useful to flag invalid values.

If we know this, we should check that it is true. If we find any discrepancies, we should contact the person who provided the data in order to get it corrected.

The tools to check this sort of internal consistency are the same as above: we could check for internal consistency and flag those that fail the check or we could use a frequency table via PROC FREQ. For PROC FREQ, use an asterisk to get the frequency of Rating by Inventory; the LIST option can also be used to provide cleaner output. For brevity, only the PROC FREQ method is used below, in Output 5. From this output, we can quickly check and see that the Rating and Inventory levels are consistent.

```
PROC FREQ DATA=mydata;
    TABLES Rating*Inventory/LIST MISSING;
RUN;
```

Rating	Inventory	Frequency	Percent	Cumulative Frequency	Cumulative Percent
High	5000	2	66.67	2	66.67
Unknown	.	1	33.33	3	100.00

Output 5. PROC FREQ: Frequency Table Rating by Inventory

Another type of internal consistency one may need to check is consistent counts between multiple input data sets. The *mydata* set we have been using is not a good choice for this example, but please imagine two data sets related to the number of students in a school. One contains the number of students by ethnicity/race, with a variable called Count containing the total number of students in each category. Another contains the number of students by gender, similarly with a Count variable. If you know that both should add up to the same number, then check that it does.

Sample code one could use is:

```
PROC MEANS DATA=race SUM;
    VAR Count;
    TITLE 'Race/Ethnicity Counts';
RUN;
PROC MEANS DATA=gender SUM;
    VAR Count;
    TITLE 'Gender Counts';
RUN;
```

If the PROC MEANS' outputs do not show the same number, then you likely need to ask whoever prepared your input data to correct it.

Note: you may find the MAXDEC or NONOBS options give you cleaner output, depending on exactly what you are counting. For more advanced uses of PROC MEANS, see the SAS documentation on the CLASS, OUTPUT, and TYPES statements for useful tricks when creating data sets. You can also run PROC COMPARE on the output from two PROC MEANS statements, in order to check that they yield equal counts or statistics.

VERIFYING INPUT DATA: CONFIRMING UNIQUENESS OF UNIQUE FIELDS

If a variable should be unique for all observations in your input data, you should check that it indeed is unique. In our sample data, the IDNum is supposed to be unique for each observation. There are several ways to check uniqueness, but one of the fastest and easiest is to use the SQL procedure.

The query below selects all records in *mydata* with duplicate IDNum values. It does this by creating a subquery listing all IDNum values that appear two or more times, then selecting records with that IDNum. Any errors will be identified in the printout.

```
PROC SQL;
    SELECT * FROM mydata
    WHERE IDNum IN
        (SELECT IDNum FROM mydata GROUP BY IDNum HAVING Count(IDNum)>=2);
QUIT;
```

Using *mydata*, we see two instances where the IDNum was not unique, as shown in Output 6.

LName	FName	IDNum	Gender	Rating	Inventory
Doe	Jane	1001	F	High	5000
Long	Jake	1001	M	High	5000

Output 6. PROC SQL: List of Non-Unique IDNum Values

Multiple occurrences of a supposedly unique value could mean one of several things. It may be a duplicate record. It may simply be a typo that someone mistyped when preparing your input. In our example, a safe guess would be that one of the two records has the wrong ID number, since they appear to obviously be different people.

Note: as an alternative to PROC SQL, you could run PROC FREQ on IDNum and output the results that have IDNum values with a frequency of two or higher. This code only lists the IDNum values, though, not the full, original records.

```
PROC FREQ DATA=mydata NOPRINT;
    TABLES IDNum/OUT=temp (WHERE= (COUNT>=2) );
RUN;
```

VERIFYING INPUT DATA: FINAL THOUGHTS

In general while verifying input data, how you proceed when you find a data issue should depend on your workplace. In most instances, the safest and easiest method would be to ask whoever provided the input to send it back to you correct. If that is not a viable or timely option, you may need to clean the data yourself. Please be certain that you are aware of any data cleaning rules used by your workplace, so that you don't introduce new issues into the data while giving it a semblance of being clean or consistent.

Also, before you contact whoever provided you the data or sound an alert about errors, double-check that the errors are truly there. Make sure you read the data into SAS accurately. It is embarrassing to tell somebody that they gave you bad data, only to realize the errors you flagged were due to bugs in your error-checking code. One recommendation is to always open the data (be it a SAS data set, an Excel file, or a text file) and look at it prior to loading it into your SAS session; make sure you see the errors in the original before you sound the alarm.

Lastly, insofar as it is possible, know your data. If you have the time, take the time to learn what are reasonable values and frequencies, so that you are able to identify when something is amiss.

CHECKING ONE'S WORK: CHECKING UPDATED VALUES AND CREATED VARIABLES

If part of a program's data manipulations include updating values or creating new variables, it is wise to check that the values of the variable were rendered correctly. It is easy to have a variable wind up with all-blanks or unexpected values due to a character type mismatch, a faulty IF-ELSE statement, or other coding issues, even if no warnings or errors are printed to the SAS log.

For example, let's say we want to create a new variable, called NewID, which is the old IDNum plus the person's initials. We also want to change missing Inventory counts to zeroes. To be extra cautious, we can create an intermediary variable called NewInventory to check the conversion was done correctly, then rename it to Inventory. (Such is probably overkill in the example in this paper, but may be useful for very complicated updates.)

Below is code that can accomplish this. See Output 7 for a printout that lets us confirm no errors occurred.

```
DATA mydata; SET mydata;
    NewID=IDNum||SUBSTR(FName,1,1)||SUBSTR(LName,1,1);
    IF Inventory=. THEN NewInventory=0; Else NewInventory=Inventory;
RUN;
PROC FREQ DATA=mydata;
    Tables NewID*IDNum*FName*LName Inventory*NewInventory/LIST MISSING;
RUN;
DATA mydata;
    SET mydata(DROP=Inventory); *drop the old variable;
    RENAME NewInventory=Inventory; *rename new variable with the original name;
RUN;
```

NewID	IDNum	Fname	LName	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0002JS	0002	John	Smith	1	33.33	1	33.33
1001JD	1001	Jane	Doe	1	33.33	2	66.67
1001JL	1001	Jake	Long	1	33.33	3	100.00

Inventory	NewInventory	Frequency	Percent	Cumulative Frequency	Cumulative Percent
.	0	1	33.33	1	33.33
5000	5000	2	66.67	3	100.00

Output 7. PROC FREQ: Creating NewID and Updating Inventory

If your dataset is too large to reasonably view all the records for certain variables that are updated or created, use the WHERE clause with modulo, as shown earlier in this paper, to spot-check.

CHECKING ONE'S WORK: CHECKING MERGES

The MERGE keyword is a useful tool to combine multiple data sets into one via the DATA step. Merges can also be performed via PROC SQL, as a JOIN. There is a risk that a merge may not be performed as expected. A possible reason is that the BY variables do not have matching values, leading to mismatches. Another concern is if the merge was expected to be one-to-one or one-to-many but was actually many-to-many, potentially leading to duplicates in the final data set. (See Section 8 of Dr. Imelda Go's paper "Mastering the Basics: Preventing Problems by Understanding How SAS® Works" for details on many-to-many risks and how to avoid them.)

One method to check that a merge was performed successfully is to check the number of records in the data sets used to create the merged data set, then compare those counts to the number of records and where the records came from in the new data set. I will first show how to do this in a DATA step, then using PROC SQL. (Note that, for the DATA step method, data sets must be sorted by the BY variables; such is not required for PROC SQL.)

Let's say we are told to merge *mydata* to the dataset called *newdata*, shown in Table 2, and told that IDNum is the primary key to match by.

IDNum	Salary
1001JD	30000
1001JL	25000
0002JS	50000
0003KZ	20000

Table 2. Contents of the *NewData* Data Set

While looking at the data set – since we remember to check out input – we can see that the variable called IDNum in *newdata* actually corresponds to NewID in *mydata*. We can compensate for this in our program. The following is a way to use the DATA step:

1. Sort the original data sets by PROC SORT.
2. Create a counter variable in each original data set, then print the maximum value via PROC MEANS
3. Merge the data sets, creating a field to state what data sets contributed. (In the code below, I use *mflag*, short for merge flag.)
4. Use PROC FREQ to print a frequency table. Compare the cumulative frequency to the number of records in the original data set to see if you the final number of records is reasonable, given the original number. (If you see more records in the final data set than were in either source data set, you may have a many-to-many merge creating duplicate records.)

```
*#1: sort the data sets and fix ID name in newdata;
PROC SORT data=mydata; BY NewID; RUN;
PROC SORT data=newdata(RENAME=IDNum=NewID) out=newdata2; BY NewID; RUN;
```

```
*#2: create counters (called ctr);
DATA mydata; SET mydata; ctr=_n_; RUN;
DATA newdata2; SET newdata2; ctr=_n_; RUN;
PROC MEANS data=mydata max maxdec=0; var ctr; Title 'mydata'; RUN;
PROC MEANS data=newdata2 max maxdec=0; var ctr; Title 'newdata'; RUN;
```

```

*#3: merge the data sets;
DATA merged;
    MERGE mydata(IN=in1) newdata2(IN=in2);
    BY NewID;

    LENGTH mflag $7.;
    IF in1 and in2 THEN mflag='Both';
    ELSE IF in1 THEN mflag='mydata';
    ELSE IF in2 THEN mflag='newdata';
RUN;

*#4: check the merge;
PROC FREQ data=merged;
    Tables mflag/missing;
    Title 'Results of Merge';

```

```
RUN;
```

As shown in Output 8, we can see that the original data sets had 3 and 4 records, respectively, and that the new data set has 4 records (3 with a match between both data sets and 1 with just *newdata*'s data). Such looks reasonable, so we can say we checked against errors during the merge.

mydata The MEANS Procedure																			
<table border="1"> <tr> <th colspan="5">Analysis Variable : ctr</th></tr> <tr> <th colspan="5">Maximum</th></tr> <tr> <td colspan="5">3</td></tr> </table>					Analysis Variable : ctr					Maximum					3				
Analysis Variable : ctr																			
Maximum																			
3																			
Page Break																			
newdata The MEANS Procedure																			
<table border="1"> <tr> <th colspan="5">Analysis Variable : ctr</th></tr> <tr> <th colspan="5">Maximum</th></tr> <tr> <td colspan="5">4</td></tr> </table>					Analysis Variable : ctr					Maximum					4				
Analysis Variable : ctr																			
Maximum																			
4																			
Page Break																			
Results of Merge The FREQ Procedure																			
<table border="1"> <thead> <tr> <th>mflag</th><th>Frequency</th><th>Percent</th><th>Cumulative Frequency</th><th>Cumulative Percent</th></tr> </thead> <tbody> <tr> <td>Both</td><td>3</td><td>75.00</td><td>3</td><td>75.00</td></tr> <tr> <td>newdata</td><td>1</td><td>25.00</td><td>4</td><td>100.00</td></tr> </tbody> </table>					mflag	Frequency	Percent	Cumulative Frequency	Cumulative Percent	Both	3	75.00	3	75.00	newdata	1	25.00	4	100.00
mflag	Frequency	Percent	Cumulative Frequency	Cumulative Percent															
Both	3	75.00	3	75.00															
newdata	1	25.00	4	100.00															

Output 8. PROC MEANS and FREQ: Checking Before and After a DATA Step Merge

We can also do similar check when doing joins in PROC SQL. PROC SQL has an advantage in that we do not need to sort the data sets or be concerned if the BY variables have different names. A downside to PROC SQL is that you have to either explicitly list all the variables you want from a data set or use an asterisk to denote all variables.

Below is a method to create a new data set called *mergeSQL*, using the IS NOT MISSING code in SQL to flag which data sets provided values. (An alias was created for *newdata*'s IDNum so that it can be maintained when there is not a match.)

```
PROC SQL;
    CREATE TABLE mergeSQL AS
    SELECT mydata.*, newdata.Salary, newdata.IDNum AS IDNum2,
           mydata.IDNum IS NOT MISSING AS in1,
           newdata.IDNum IS NOT MISSING AS in2
    FROM mydata FULL JOIN newdata
    ON mydata.NewID=newdata.IDNum
    ;
Quit;
PROC FREQ data=mergeSQL;
    Tables in1*in2/LIST MISSING;
    Title 'Results of Merge';
Run;
```

The in1 and in2 variables essentially take the place of the merge flag. A 0 means the data set did not contribute to the final table, and a 1 means the data set did contribute to the final table. From the PROC FREQ output in Output 9, we can see the same result, albeit in a different way, as we did when using the DATA step.

Results of Merge

The FREQ Procedure

in1	in2	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	1	1	25.00	1	25.00
1	1	3	75.00	4	100.00

Output 9. PROC FREQ on a SQL-Created Table: Checking After a SQL Merge

Note: I use the FULL JOIN code to include records from both data sets, but generally a LEFT JOIN or RIGHT JOIN is all that is really required for a given task.

CHECKING ONE'S WORK: CHECKING RESOLUTION OF DUPLICATES

SAS gives us a very easy way to resolve duplicates via PROC SORT. It is also fairly easy to resolve duplicates by using the First and Last variables in a DATA step when using BY variables. However, checking the resolution of duplicates – by which I primarily mean checking that the duplicates being removed were truly duplicates and not false positives excluded as duplicates by the programming logic – can require more work. I will first discuss using PROC SORT and then discuss using the DATA step.

With PROC SORT, you can retain just one copy of each BY variable by using the NODUPKEY option. (See the References for a paper that goes into other PROC SORT options.) The OUT and DUPOUT options create two new data sets: one with just one copy of each BY variable, and one with the discarded duplicates.

```
PROC SORT data=mydata OUT=nodups DUPOUT=dups NODUPKEY;
    BY IDNum;
RUN;
```

A quick, although manual, check is to open the resultant data sets and compare them to one another and the source, to make sure that the duplicates are truly duplicates. By running this code on *mydata* and then looking at the data sets, we see that there is a duplicate on the IDNum '1001' but that it appears to be a different person (Jake Long verses Jane Doe). The program did exactly what was asked of it, but the duplicate resolution appears to be faulty. (Whether this means the input data needs to be cleaned, you need to add or change your BY variables, or the duplicate should truly be dropped likely depends on the task at hand.)

The PROC SORT method requires manually checking the data sets and thus is very prone to human error and the limitations of spot-checking. Using a DATA step allows a more programmatic approach. First, we sort the data set by the BY variables. Then, we SET the data set BY the by variables and use the FIRST. And LAST. keywords to

determine which records are unique and which have duplicates. (This syntax gets more complicated if you have multiple BY variables, but the essential idea is the same.) Then, use PROC PRINT to display all the duplicates.

```
PROC SORT data=mydata; BY IDNum; RUN;
DATA mydata;
    SET mydata;
    BY IDNum;

    IF FIRST.IDNum and LAST.IDNum THEN dupflag=.;
    ELSE dupflag=1;
RUN;
PROC PRINT data=mydata;
    WHERE dupflag NE .;
    TITLE 'Duplicate Records';
RUN;
```

If your dataset has a very large number of duplicates, requiring you to just spot-check, this alternate PROC PRINT may be more useful. This example checks the first 1000 duplicates.

```
PROC PRINT data=mydata(WHERE=(dupflag NE .) obs=1000);
    TITLE 'Duplicate Records';
RUN;
```

Output 10 shows the output (the same for either PROC PRINT in this example) to show how easy one can compare the duplicates to check if they are true duplicates or not.

Duplicate Records

Obs	LName	Fname	IDNum	Gender	Rating	ctr	NewID	Inventory	dupflag
2	Doe	Jane	1001	F	High	2	1001JD	5000	1
3	Long	Jake	1001	M	High	3	1001JL	5000	1

Output 10. PROC PRINT: Checking Records Flagged as Duplicates

This printout can be used to check the duplicate records and determine what changes, if any, should be made to duplicate resolution prior to removing a duplicate. Alternatively, you may discover simply that your ID numbers are not that meaningful and each record is actually a unique record and all should be retained.

CHECKING ONE'S WORK: FINAL THOUGHTS

The general guideline in this paper is to check whatever changes your program makes to confirm that they made the expected changes. This can be tedious, but it can also catch very problematic errors at their inception.

In a previous paper, *A Five-Step Quality Control Method: Checking for Unintended Changes to SAS® Datasets*, I have discussed how to the COMPARE procedure to check for changes in variables, values in variables, and number of records while modifying a data set. Please refer to it for additional tools not discussed in this paper.

CHECKING ONE'S OUTPUT: FINAL CHECK OF ALL FIELDS

Before releasing a final data file for a report or someone else's use, it is wise to check all the fields that will be included in the data file. This is true whether you are saving a permanent SAS data set, using PROC EXPORT to create an external file, or using a DATA step to write an external file. If there are inter-related fields, it is also wise to check them with a cross-frequency.

The code to accomplish this is essentially the same as how one would use PROC FREQ to check one's input. For example, if we wanted to create a data file with each employee's name, NewID, Inventory, Rating, and Salary from our *merged* data set, the following PROC FREQs could be used to check them.

```
PROC FREQ data=merged;
    TABLES
        FName LName NewID Inventory*Rating Salary
    /LIST MISSING;
RUN;
```

Upon review of this PROC FREQ, we would be reminded that one record did not have a match in *mydata* and thus is blank in all of these fields except NewID and Salary. Thus, we may decide to delete it from our output.

Remember to use modulo if there are so many records that spot-checking is required for fields like names, ID numbers, or dates.

CHECKING ONE'S OUTPUT: OPEN THE OUTPUT

This check is not a programmatic one, but one to be done manually. Whatever your output is – SAS data set, CSV, Excel, etc. – open it and check that it looks like what you expect it to look like. This can include checking the header row, the number of records, the order of records, and what values are missing. You can also check for things such as that you truly did create a comma-delimited file if such was your intention. (The simple omission of the DSD keyword in the DATA step can lead to an embarrassing error that the SAS log likely will not notice.)

CHECKING ONE'S OUTPUT: READ THE DATA BACK INTO SAS AND COMPARE

A more exhaustive method of checking one's output is to read it back into SAS and then use PROC COMPARE to compare it to the data set used to create it. I recommend importing via a DATA step to have the cleanest comparison, although other methods (e.g., PROC IMPORT) also work. See the paper on PROC COMPARE in the References for guidelines on reading and, more importantly, not misreading PROC COMPARE output.

Here is an example of writing a CSV file, then reading it back into SAS to compare it. To cut down on repetitive code, I recommend putting the file location and the put/input statement code into macros. Note that, if you were generating a real report, you may want to leave off certain variables (e.g., the old IDNum and dupflag) and you would likely want to create a header row. In the below, we include all variables and the header row code is commented out.

```
%let filelocation='C:\SESUG\sample file.csv';
%macro putStatement;
    NewID :$6. IDNum :$4.
    LName :$20.
    FName :$20.
    Gender :$1.
    Rating :$10.
    Inventory :8.
    ctr :1. dupflag :1.
%mend;
DATA _null_;
    FILE &filelocation DSD /*LRECL=*/;
    SET mydata;
    /*IF n =1 THEN PUT "header row goes here";*/
    PUT %putStatement;
RUN;
DATA for_compare;
    INFILE &filelocation DSD MISSOVER PAD /*FIRSTOBS=2 LRECL=*/;
    INPUT %putStatement;
RUN;
PROC COMPARE DATA=mydata COMPARE=for_compare LISTALL; RUN;
```

The first DATA step creates the CSV file. (You may need to use the LRECL option if your record length is large.) The second DATA step reads the CSV back into SAS, as the data set *for_compare*. (Again, use LRECL if needed. Add FIRSTOBS=2 if you have a header row you need to avoid reading in.) Finally, PROC COMPARE executes the comparison. Since *for_compare* should correspond record-to-record with *mydata*, we do not need an ID statement in PROC COMPARE.

Note that sometimes PROC COMPARE might find a difference that is not an error. For example, when running the above, we see that the NewID variable has different lengths in the different data sets.

Listing of Common Variables with Differing Attributes

Variable	Dataset	Type	Length
NewID	WORK.MYDATA	Char	44
	WORK.FOR_COMPARE	Char	6

Output 11. PROC COMPARE: Differences in Variable Length

In this instance, we can ignore the difference because it does not impact the actual data. If you wanted to remove this difference to get a 'clean run', you could either use a LENGTH statement when originally creating NewID to limit it to six characters or change the %putStatement macro so that it reads in NewID as a 44-character variable.

CHECKING ONE'S OUTPUT: FINAL THOUGHTS

It behooves a programmer to ensure that the product he or she creates is high quality. By using these and any other applicable methods or tools, a programmer shows due diligence regarding the creation of erroneous files.

CONCLUSION

This paper has gone over a set of tools that a SAS programmer can use to help identify errors and bugs and to ensure quality output. By checking our input, we can identify and fix false assumptions as well as dirty data. By checking our work, we can be confident that our program operated as expected and that no new errors or dirt were introduced. By checking our output, we can be confident that what we provide to others is quality work that represents both our own programming ability and our office or workplace in a positive light. Quality control can be tedious and time-consuming, but it is well worth the time and resources that can be wasted by errors going unnoticed.

REFERENCES

- Brown, Aaron (2016), "A Five-Step Quality Control Method: Checking for Unintended Changes to SAS[®] Datasets," 2016 SouthEastern SAS User's Group, Bethesda, MD, USA.
- Go, Imelda C. (2012), "Mastering the Basics: Preventing Problems by Understanding How SAS[®] Works", 2012 SouthEastern SAS User's Group, Durham, NC, USA.
- Horstman, Joshua and Roger Muller (2016), "Don't Get Blindsided by PROC COMPARE," 2016 Western Users of SAS Software, San Francisco, CA, USA.
- KelseyBassett, Britta (2006), "The SORT Procedure: Beyond the Basics," 2006 SUGI, San Francisco, CA, USA.
- Lafler, Kirk Paul (2012), "Exploring DATA Step Merges and PROC SQL Joins," Proceedings of the SAS Global Forum 2012 Conference, Software Intelligence Corporation, Spring Valley, CA, and The Bedford Group, USA.

ACKNOWLEDGMENTS

The author thanks the SESUG chair and staff for this conference and the opportunity to present. He also thanks Dr. Imelda C. Go for the code for using modulo with PROC FREQ, the idea of reading output back into SAS, and for a general mindfulness towards quality control.

CONTACT INFORMATION

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

Aaron R. Brown
 South Carolina Department of Education
 1429 Senate Street, Columbia, SC 29201
 Work Phone: 803-734-8858
 E-mail: ARBrown@ed.sc.gov

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.