Quality Control As Data Analysis
Paul Gorrell, Social & Scientific Systems, Inc., Silver Spring, MD

ABSTRACT

Good quality control programs are the opposite of good marketing; i.e. you want the smallest flaw, even if it's on a single record, to announce its presence loud and clear. We will take advantage of the fact that SAS® offers a wide variety of tools for data analysis and we will look at quality control [QC] as one particular form of data analysis. From this perspective, effective QC becomes a question of applying data-analysis techniques through the use of SAS PROCs, formats, and LOG messages. For example, valid data analysis would never be based on the first 50 observations of a 100,000 record file. Yet many SAS programmers print 50 observations to 'check' on variable values, or the results of a merge. Although there may be times this is necessary (or, at least, comforting), it is rarely (if ever) sufficient. QC, like data analysis, should be based on logic and pattern verification, as well as on properties of the entire output data set (or an identified sub-group). I will emphasize the importance of knowing what patterns the data SHOULD exhibit, and how to use SAS PROCs and LOG messages to test for them.

INTRODUCTION

The title of this paper points to two important aspects of quality control with SAS programming. The first is that checking to see that the data set created by a SAS program has the intended properties is a form of data analysis. The second is that the SAS programmer is in the fortunate position of having a variety of excellent data-analysis tools at hand.

I'm going to distinguish between 3 types of quality control [QC].

1. a. INPUT QC
   b. PROGRAM QC
   c. OUTPUT QC

Input QC is the process of learning about the input data to your program. It's important to distinguish between properties that the data is supposed to have, and the properties it actually has. It is also important for the SAS programmer to have a good, general, sense of what the data means and how it will be used. This knowledge allows the programmer to integrate more-intelligent QC measures into the programs than would be possible by blindly following specs.

Program QC refers to writing programs so that quality-control information is available both from intermediate stages of the program as well as in the final output. For example, merges are crucial areas of a program requiring QC. I will give examples where the merge DATA step can be written to show important information about the merge.

Output QC is what most programmers are thinking about when QC is mentioned. That is, Output QC is looking at the output data sets to make sure that they have the intended properties. There are various aspects of this. There's the basic level of making sure that 2+2=4. There are also relationships between variables that are determined by external factors, e.g. IF MONTH = 'SEPT' THEN (1 <= DAY <= 30). There are variable values and relationships to check that are specific to the particular data set being input or generated, e.g. number of children in the family was recorded only if they were living at home.

I will discuss a number of specific SAS programming examples, but the goal of this paper is to emphasize as much as possible the general principles which lead to more-effective quality control. Throughout I will focus on a common situation for a SAS programmer: you have an input SAS data set (or, sets) and you have specs to modify this data set in some way and output a new data set. Of course there are numerous other situations (e.g. reading in flat files, outputting to Excel spreadsheets, etc.) that require QC. I hope that I am able to communicate my basic approach in such a way that the programmer can readily apply it to situations not covered here.

One thread that will run through this discussion is what we might call content-based programming. By that I mean that it is well worth the time for a SAS programmer to learn as much as possible about the project the programs are a part of, the type of people who will be working with the data, and the general goals which motivate the particular specs. A lot of time and money can be saved if the programmer is in a position to catch content-based problems before they are delivered to the client. For example, I work a lot with health-care survey data and it is fairly easy to figure out that males should not have valid responses to pregnancy questions. It is less obvious that some questions (e.g. "Have you fully recovered?") are only asked of people with certain types of medical conditions.

An added benefit to content-based programming is that it makes the work more interesting. The more interested I am, the more alert and engaged I am with the data and the programs. So it's efficient in the long run for the programmer to take some time before ever writing a line of code to find out as much as possible about the data, and to plan ahead with QC in mind.

INPUT QC

A lot of old cliches are exactly right when it comes to Input QC ("A stitch in time saves nine", "Penny wise but pound foolish", etc.). I'll add two more:

2. a. Don't overlook the obvious.
   b. Verify what you're told.

The word verify is going to come up quite often in this paper because a lot of basic QC comes down to verifying that what you think is true, or have been told is true, actually is true. One of the great enemies of good QC is time pressure, and this can start right at the beginning by not taking the time to take a good look at the input data.
SAS provides a wonderful tool for initially examining input data sets: the CONTENTS procedure. It is absolutely essential to run a PROC CONTENTS to see, at least, the following:

(3) a. Number of Variables
   b. Number of Observations
   c. Sorted?
   d. Engine (V8?)
   e. Variable Name
   f. Variable Type (Numeric/Character?)
   g. Variable Length
   h. Variable Labels
   i. Variable Formats

Initially it's important to know if this general information agrees with the documentation you have about the file. If it doesn't then you need to straighten this out before going any further (perhaps you have the wrong file, one that's similar to the right file).

If the general information (e.g. number of variables and observations) appears to be as it should be, you can look at specifics such as variable names and data types. Whether a variable is numeric or character is not always predictable from its name. An ID variable that consists only of numbers might be either numeric or character. A flag variable whose values are only 0 or 1 might be numeric or character. It's obviously important when you start writing code to know what type of variables you have.

When working with numeric data it's important to know the variables' length. The default is 8 bytes, but this is often reduced to save disk space. If you are going to be modifying the values of numeric data, it's especially important to know variable length. Here's a simple program for taking a first look.

(4) PROC CONTENTS DATA= DSN_IN OUT= CHECK1
    (KEEP= NAME TYPE LENGTH LABEL);
RUN;

PROC FREQ DATA= CHECK1;
   TABLES LENGTH;
   WHERE TYPE = 1;
RUN;

The output data set CHECK1 will have 4 variables: NAME (= the names of the variables on the DSN_IN data set); TYPE (1 if numeric, 2 if character); LENGTH and LABEL.

The PROC FREQ outputs a table showing the frequency of numeric variables (WHERE TYPE = 1) by LENGTH. There are two reasons to care about numeric length: (i) if it is less than 8 bytes and you will be increasing any values, you need to make sure that the length specification is sufficient for the updated values; (ii) if you are concerned about saving disk space, you may want to consider reducing numeric length for some variables. For more-detailed discussion of this topic, see my paper on SAS numeric data in these Proceedings.

The situation is similar for character data. Suppose you have a character variable RESPONSE that has a LENGTH of 3 on the input data set and 3 values: "YES", "NO" and "DK" (where "DK" indicates "Don't Know"). You're asked to edit this variable as in (5):

(5) IF RESPONSE = 'DK'
    THEN RESPONSE = 'MAYBE';

Because LENGTH is a variable attribute that, by default, is inherited by one data set from another, the LENGTH of 3 will cause the new value to be "MAY" instead of "MAYBE". This is one of many situations where a little time spent reading before typing can save a lot of time later ("Anyone remember which program we edited that RESPONSE variable in?").

It's important to become familiar with variable labels because they are often important sources of information about the intended content and use of the variables. Sometimes labels offer abbreviated explanations of particular values ("1 IF NUMERIC, 2 IF CHARACTER"). If your editing adds to, or changes, these values, you will need to modify the label as well.

Even the best specs aren't perfect and the programmer who has taken (or, been given) the time to understand the context of the data and the specs, can often spot additional modifications that may be needed. Often these take the form, "If you change X then you should also change Y." This is another potential time saver—and one that is often appreciated by the client writing the specs.

Formats are also important for a number of reasons. If numeric variables are associated with formats, this may affect the operation of particular PROCS. It's good to verify that associated formats should be kept, and to communicate any potential implications.

If formats indicate value ranges, it's good to verify that these ranges are meaningful for the data you have. For example, if the format values are \([-1 = '-1']\\), \([0 = '0']\\) and \([1-100 = '1-100']\\) but the actual range of positive values on the input data set is 40-60, you should check whether or not this format range should be changed. For this type of case, if you only ran a formatted FREQ, you would never see any problem because the range of actual values is within the format range. The output would only show a certain number of records with each of the format values.

Another important piece of information to know about an input data set is which variable(s) uniquely identify observations, i.e. the key variable (often, but not always, this is the SORTED BY variable). If you are told that PERSON_ID uniquely identifies rows, then you should run the following check:

(6) PROC SORT DATA= DSN_IN OUT= CHECK2
    NODUPKEY;
    BY PERSON_ID;
RUN;

If PERSON_ID really is the key variable, then the LOG will show that 0 observations have been deleted.

Don't overlook the obvious takes many forms. For example if you have MONTH and DAY variables, you could include conditionals such as (7) as part of a DATA step.
You can also use format ranges to test for valid and invalid values for particular variables. Part of the importance of knowing what properties the data SHOULD have concerns how various types of non-response values are coded. For example, what is the value for a MONTH variable if this information is unknown. For many surveys, responses such as ‘Don’t Know’ or refusals are coded as specific minus values (-8, -7, etc.) or out-of-range values (e.g. 99). Knowing what these are is essential before any code is written that will edit or re-code the input data. See Ron Cody’s book on data cleaning for many useful approaches to checking the values of input variables.

Here’s an example of the verify what you are told theme: suppose that you have a person-level data set (key=PERSON_ID) and each person is a member of a family, identified by FAMILY_ID. Both variables are character, with PERSON_ID having LENGTH 4 and FAMILY_ID LENGTH 2. PERSON_ID is composed of FAMILY_ID plus a two-character sequence number. The data set would look like (8).

(8) | FAMILY_ID | PERSON_ID |
---|---|---|
01 | 0101 |
01 | 0102 |
01 | 0103 |
02 | 0201 |
02 | 0202 |
03 | 0301 |

One generalization is that the first two characters of PERSON_ID = FAMILY_ID. You can verify this as follows:

(9) IF FAMILY_ID NE SUBSTR(PERSON_ID,1,2)
    THEN DO;
    PUT ‘***** ID ERROR *****’ ;
    PUT FAMILY_ID= PERSON_ID= ;
    END;

This will print a message to the LOG for each record where FAMILY_ID is not the first two characters of PERSON_ID. If you suspect you have a lot of bad data and don’t want to fill up your LOG, you can initially set a flag variable (e.g. ID_ERR) that is assigned a value of 1 if the conditional evaluates as true. Then you can run a FREQ on ID_ERR or use it to create an output data set.

This type of approach is much better than dumping 50 or so observations to ‘check’ on variable values or the relationships between variables. I will discuss this further in the section on Output QC, but it is relevant here as well. Printing a small number of records is an amazingly common practice among SAS programmers. It is often requested by clients. It is useful for getting a kind of ‘at-a-glance’ initial sense of the data. It may even make you aware of a problem that would never have occurred to you to check on. But it is never a sufficient QC check for the simple reason that, if you print 50 records, there may be a problem with records 51 through 100,000.

Here’s where ‘QC as data analysis’ comes in most clearly. Good QC, like good data analysis, allows you to draw conclusions about the data. From a QC perspective, you want to be able to say, with confidence, that the first two characters of each PERSON_ID on DSN_IN = FAMILY_ID. The only way to put yourself in position to say that is to examine all the records. The only reasonable way to do this is with a DATA or PROC step. I recently worked with an input file with 4 different ID variables. The file has 323,536 records. On each record, a substring of each of the ID variables is supposed to have the same 3-digit person identifier. This turned out to be true for all but one record. But it was important to catch that one error and correct it.

**PROGRAM QC**

The point of Program QC is to write DATA steps (e.g. merges) and include PROCs in such a way that you are outputting information that is necessary for evaluating the correctness of the program and its output.

There are a couple of SAS options that I would recommend either as system or program options. The first is ERRORABEND. This will cause SAS to abend for errors where it would otherwise (i.e. if NOERRORABEND is set) write a LOG ERROR note, set OBS=0 and go into syntax-checking mode. I rarely find syntax checking after the initial error very useful, and it is much easier for me to focus on correcting the one error that caused SAS to abend. The LOG of any running program should be free of ERROR messages, even those associated with errors that “don’t matter.” Chapter 2 of Michele Burlew’s book on debugging SAS programs has good information on interpreting and correcting Notes, Warning and Error messages in SAS LOGs.

Another option I would recommend, not available in V6, is MERGENOBY=ERROR. If you fail to include a BY statement when merging data sets, SAS will write the following message to the LOG.

(10) ERROR: No BY statement was specified for a MERGE statement.

This is a very useful option which I have in my AUTOEXEC file. I never intentionally merge data sets without a BY statement (even one-to-one merges) so =ERROR works best for me. You can also set MERGENOBY=WARN to get a warning rather than an error message.

In (11) I illustrate a merge DATA step where temporary variables containing information about the merge are written to the LOG.

(11) DATA THREE;
    MERGE ONE (IN= A) TWO (IN= B)
    END= ITSOVER;
    BY ID;
    IF A AND B
    THEN MTCH+1;
    ELSE IF A
THEN JUSTA+1;
ELSE IF B
  THEN JUSTB+1;
IF ITSOVER
  THEN PUT MTCH= JUSTA= JUSTB= ;
DROP MTCH JUSTA JUSTB;
RUN;

END= creates a temporary variable indicating the end of the file with the largest number of observations. Here this variable is named ITSOVER and the LOG message is only written when the last observation is processed. At this point the value of the variable MTCH equals the number of observations where ONE and TWO match BY ID. The value of JUSTA is the number of non-matches in ONE. The value of JUSTB is the number of non-matches in TWO. Although it’s often the case that the number of OBS in the output data set indicates the number of matches, if a subsequent problem is noticed, then knowing the number of non-matches can be important.

When possible I try to have enough information to be able to make predictions about the outcome of a merge. Is it a one-to-one merge where all records should match? Is it a one-to-many match? I can’t always predict what the exact values of MTCH, JUSTA and JUSTB should be—but I can usually make some predictions, even it’s simply that there should be very few non-matches.

Let’s take a different type of example, one where Input and Program QC will help you write the correct code without trial and error. Suppose you have the spec in (12).

(12) Create a variable AGECAT. If a person is under 18 then AGECAT = CHILD. If the person is 18 or older, then AGECAT = 'ADULT'.

Without having looked at the data, it might be tempting to code this as:

(13) IF AGE < 18
  THEN AGECAT = 'CHILD';
ELSE AGECAT = 'ADULT';

But as part of Input QC, you noticed that there were both missing values and zero values. The conditional in (13) will code both of these values as ‘CHILD.’ This may be want is intended for 0, but certainly not for missing values. Another way to catch this type of value is to always include a QC crosstab after re-coding. This crosstab should have the basic form BEFORE*AFTER. Of course the trick to such a crosstab is how to format the BEFORE variable. For something like AGE (with perhaps 100 different values) it wouldn’t be too annoying to leave it unformatted. But that isn’t often a realistic option.

Here you could run a PROC to get the MIN and MAX values (perhaps using a WHERE AGE GT 0 clause) and then format AGE as MISSING, O, 1-17 and 18-95 (assuming that 95 was the maximum age on the data set). Then you could modify (13) as in (14). Of course, once you know to ask, it may be a simple check to find out if 0 values should be coded as ‘CHILD.’ If so then (14) can be simplified to include 0 in the CHILD range.

(14) IF (1 <= AGE <= 17)
  THEN AGECAT = 'CHILD';
ELSE IF (18 <= AGE <= 95)
  THEN AGECAT = 'ADULT';
ELSE IF AGE = 0
  THEN AGECAT = 'ZERO VALUE';
ELSE IF AGE = .
  THEN AGECAT = 'MISSING';
ELSE AGECAT = '??????';

The final clause here points to another part of good programming: conditionals should always exhaust the logical possibilities. The final ELSE gives you control over the unexpected. You will have a clear indicator ("??????") that there are AGE values you haven’t accounted for.

Often the best you can do for quality control is to test that two different methods lead to the same result. SAS is great for this because there’s usually more than one way to do any particular operation. Consider an example where you have a person-level data set with three expenditure variables: OOP (for out of pocket); PRV (for private); and PUB (for public). The spec is to create a person-level total expenditure variable TOTEXP.

Here’s a way of building good QC into the DATA step (assume there’s a person-level ID variable).

(15) DATA TWO;
    SET ONE END= ITSOVER;
    TOTEXP = (OOP+PRV+PUB);
    SUMTOTXP+TOTEXP;
    TOTCHK = SUM(OOP,PRV,PUB);
    SUMTOTCK+TOTCHK;
    SUMOOP+OOP;
    SUMPRV+PRV;
    SUMPUB+PUB;
    IF TOTEXP NE TOTCHK
      THEN PUT ID= TOTEXP= TOTCHK= ;
    IF ITSOVER
      THEN
        DO;
          IF SUMTOTXP NE SUMTOTCK
            THEN PUT ID= SUMTOTXP= SUMTOTCK= ;
          ELSE PUT 'SUM TOTALS OK';
          IF SUMTOTXP NE
            SUM(SUMOOP,SUMPRV,SUMPUB)
            THEN PUT ID= SUMTOTXP= SUMOOP= SUMPRV= SUMPUB= ;
          ELSE PUT 'SUMTOTXP OK' ;
        END;
    RUN;

On the record (i.e. person) level, this DATA step creates a total expenditure variable TOTEXP. As a check on this we create another total expenditure variable TOTCHK with the sum function. They should be equal and we test for this on each record (IF TOTEXP NE TOTCHK...). As an added check, we sum these totals and their components (OOP, PRV and PUB) for the full data set—and then put in conditionals to make sure
that the sums of the totals equal the totals of the sums (i.e. everything adds up the way it should).

Clearly there is more that could be built in here, and other ways (e.g., using summary PROCs) to check on these variables. But, even though this DATA step is more complicated than it would be without the QC, it is easier than coming back later and trying to figure out why something's not adding up.

The extra assignment statements and conditionals in this DATA step put you in a position where you can say, with confidence, that for each record the TOTEXP variable accurately sums the OOP, PRV and PUB variables. Also, if program LOGs are part of what you deliver to the client, you now have specific program output that you can point to to back up your assertions about the programs and the data.

Another common area of SAS programming is creating a new file or variable that's at a different level (e.g. person, family, etc.) than the input file. As an example, assume that you have a person-level file with family and person ID variables, plus an age variable. The spec is in (16).

(16) Create a family-level variable FAM_AGE that has a value of 1 if at least one member of the family is 65 or older. Otherwise FAM_AGE = 0.

A common way to do this in SAS is shown in (17). I haven't included the QC parts of the merge because (17) is illustrating common practice.

(17) PROC SORT DATA= ONE;  
   BY FAMILY_ID;  
   RUN;  
   
   DATA TWO (KEEP= FAMILY_ID FAM_AGE);  
   SET ONE;  
   BY FAMILY_ID;  
   RETAIN FAM_AGE;  
   IF FIRST.FAMILY_ID  
      THEN FAM_AGE = 0;  
   IF AGE >= 65  
      THEN FAM_AGE = 1;  
   IF LAST.FAMILY_ID;  
   RUN;  
   
   PROC SORT DATA= TWO;  
   BY FAMILY_ID;  
   RUN;  
   
   DATA THREE;  
   MERGE ONE TWO;  
   BY FAMILY_ID;  
   RUN;

In this DATA step the data set ONE is read in twice BY FAMILY_ID. That is, a family group of records is read in from ONE (IN= A), and the FAM_AGE variable is created based on the AGE variable. This variable is RETAINed and its value on the last record for this family 'spills over' into the records for this same family when ONE is read in again BY FAMILY_ID. The trick here is this second pass through the family group doesn't trigger the THEN FAM_AGE = 0 conditional because the value of FAMILY_ID doesn't change so it's only records on the IN= A data set where FIRST.FAMILY_ID = 1. If you want to see this concretely, create a small data set of 3 or 4 families and place (19) just before the RUN statement.

(19) PUT FAMILY_ID= AGE= FAM_AGE=  
     FIRST.FAMILY_ID= LAST.FAMILY_ID= ;

The DATA step in (18) isn't directly generating any QC output. But it is creating an output data set that will allow for OUTPUT QC to yield some results, as we'll see in the next section.

The general structure, somewhat simplified, of many SAS programs is to input a data set, create a series of temporary data sets, and output a permanent data set. We can illustrate this in (20).

(20) I \rightarrow T_1 \ldots \rightarrow \ldots \rightarrow T_n \rightarrow O

It's a good rule of thumb to generate QC output (either as LOG messages or PROC output) for each temporary data set the program generates. That way, if you have to work your way back from some problematic output to its source, you have a series of snapshots to guide the search.

**OUTPUT QC**

Let's begin this section with the last example from the previous section. We've created a family-level variable FAM_AGE on a person-level file. How do we check the correctness of our code?

First question is: Given the specs, what SHOULD be true? It SHOULD be the case that all records on the file where AGE is 65 or greater have a FAM_AGE value of 1. That is, if you're 65 or older you must be in a family that has a member who is 65 or older. So, if we run a crosstab FAM_AGE*AGE (where AGE is formatted as 'UNDER 65' and '65 OR OLDER'), we should find that for all records where AGE='65 OR OLDER', FAM_AGE = 1.
But what can we say about the records where AGE is less than 65? Not much directly. This is where the DATA step in (17) comes in. We now have two DATA sets that should be identical, even though the family-level variable FAM_AGE was created in two different ways. This is a time to use PROC COMPARE. We can use it in its simplest form, as in (21).

(21) PROC COMPARE BASE=THREE
    COMPARE=CHKFAM;
    RUN;

The PROC COMPARE output should say that there are no observations with variables of unequal value. Note that this is also an added check on the merge in (17). That is, data set THREE should be equal to ONE plus the new FAM_AGE variable. Data set CHKFAM created in (18) also should be identical to ONE plus the FAM_AGE variable. So, if (17) and (18) produce identical output, and the crosstab shows that all persons 65 or older are in families where FAM_AGE = 1, then this seems like a pretty strong indication that we're outputting the intended result. Note that it's not proof. We've used the same conditionals in (17) and (18) to create the variables, so if we're repeating an error in our logic that it isn't shown by the crosstab, it may still slip through.

This points to another aspect of trying to exhaust the possibilities, not just when writing conditionals, but also when testing output. I have seen programs where, having recognized that we can only predict the FAM_AGE values for those 65 and older, programmers will run a PROC FREQ for FAM_AGE and include the clause WHERE AGE >= 65. This is risky because then we can't see the numbers for those under 65. It might be that the crosstab shows that FAM_AGE = 1 for ALL records on the file. This is almost certainly a programming error (if it's true, why create the variable?). The WHERE clause restricts the information that is output, creating the possibility of missing a large problem.

This brings us back to content-based programming. What is the population on the data set? Is it families with at least one person receiving Medicare? A random sample of families in the US? Obviously this will affect our expectations. When creating variables at a certain level, it's useful to run FREQUENCIES at that level, even if not required by the specs. So run a FREQUENCIES for FAM_AGE on the family-level file TWO created in (17). Is the output reasonable given the population?

One last point with this example. If you wanted to directly check that the output file THREE is identical to ONE except for the addition of the FAM_AGE variable, you can compare them by simply dropping the new variable, as in (22).

(22) PROC COMPARE
    BASE=THREE (DROP=FAM_AGE)
    COMPARE=ONE;
    RUN;

This allows you to check that there haven't been any 'inadvertent' changes along the way. As mentioned in the section on PROGRAM QC, a typical program usually generates a number of temporary data sets before a permanent data set is created. In addition to including QC LOG messages and PROC output for the temporary data sets, it is important to repeat any relevant QC measures with the final output. Not all will be possible because the output file might be different in character (e.g. a person-level rather than a family-level file) from some of the temporary data sets, but what is possible should be done. After all, it's the final output that's important.

One all-too-common way to check the final output is to print 50 OBS or so of a large file to check that everything looks the way it should. A lot of programmers do this, and a lot of clients request it, because it gives a concrete sense of how the variable values look. But it is not a sufficient QC measure.

Sometimes there's a series of record-level edits that really are just a list of individual changes to a file. In this case PROC OUTPUT, which is usually designed to capture patterns in the data, is of little use. Here's one case where the temptation to dump 50 OBS is hard to resist. If we take the 'data-analysis' perspective, one problem with dumping 50 OBS is that you usually see the following:

(23) PROC PRINT DATA=THREE (OBS=50);
    RUN;

Here the OBS= 50 tells SAS to print the first 50 observations in the data set. Usually there is no particular reason to print the first 50 observations, it's just convenient. If there really is no alternative than examining a small number of observations to check a large number of record-level edits (i.e. there is no pattern to be tested), I would recommend, since it's now very easy in SAS, to take a simple random sample [SRS], as shown in (24). This is really just a form of data-entry checking.

(24) PROC SURVEYSELECT DATA=THREE
    METHOD=SRS
    SAMPSIZE=50
    OUT=CHECK3;
    RUN;

    PROC PRINT DATA=CHECK3;
    RUN;

You can change SAMPSIZE to be whatever number of records you would normally dump and then print the output. This method allows you to avoid basing conclusions on variable values that may be particular to the first 50 or so observations in a data set. It's not perfect, but it's better than the alternative.

In many ways OUTPUT QC mirrors INPUT QC in that you want to have a very detailed picture of both input and output data sets. In the INPUT QC section I stressed the essential role of PROC CONTENTS. This is equally important with OUTPUT QC. The list of data set properties in (3), repeated here as (25) should form a minimal output checklist.

(25) a. Number of Variables
    b. Number of Observations
    c. Sorted?
    d. Engine (V8?)
    e. Variable Name
    f. Variable Type (Numeric/Character?)
    g. Variable Length
    h. Variable Labels
    i. Variable Formats
Are the number of variables and observations correct? Should the data set be sorted? Do all the variables have labels and/or formats? I would also add to this list a question about compression. SAS V8 offers good options for compression that are worth considering (COMPRESS= CHAR | BINARY). It is also useful to use the POSITION or VARNUM option to clearly see the position of the variables on the data set. This is often a good way to pick up any stray variables (e.g. I or X from a DO loop) that may have been accidentally left on the data set. These variables will tend to be positioned near the end.

CONCLUSION

The basic principle of QC is to know your data. Good programming requires that you know your input data, that you know the relevant properties of intermediate data sets, and that you know the details of your permanent output. An important element of this is to distinguish between the values that variables actually have, and the values that they should have. As much as possible this should be done for all records on all the files you are working with.

SAS programmers are fortunate to have a variety of data-analysis tools at their disposal. The minimal added cost of including QC PROCs or conditional LOG messages in SAS programs is usually more than offset by the benefit of preventing the generation of unintended output.

In the section on INPUT QC, I said that time pressure is one of the great enemies of good QC. It’s clear that in the real world you can’t always run every check that you’d like. But there’s an awful lot that can be done fairly efficiently. In the same way that there are patterns in data we want to capture with our programs, there are patterns to QC that often recur on projects we are working on. As with input data before beginning a program, it’s worthwhile at the beginning of a project to anticipate QC needs and build them into program specs. Often this will allow a number of QC steps to be ‘macrotized’ for a greater saving of project resources by being applicable to a number of different programs.

In general, the net cost in time for good QC is probably minimal. Many of us know from painful experience the feeling that comes when someone asks, right after the final data delivery, “Did you know that those expenditure values we summed had –1 values for missing data?”

REFERENCES


ACKNOWLEDGMENTS

When writing papers such as this, one feels more like an editor than an author. Part of being a good programmer is being a magnet for all the tips and tricks that you can pick up from old programs, old programmers, young programmers, listservs, users’ groups and hallway conversations. This paper is filled with things I’ve picked up along the way and much of it (I hope) is familiar to many readers. Thanks to the SAS-programming community, especially my colleagues at Social & Scientific Systems.

CONTACT INFORMATION

Paul Gorrell
Social & Scientific Systems, Inc.
8757 Georgia Avenue, 12th Floor
Silver Spring, MD 20910

Email: pgorrell@s-3.com
Telephone: 301-628-3237 (Office)
301-628-3000 (Main)
FAX: 301-628-3201