Paper 208-2009

Testing Your SAS Code Against Known Data

Sharon Schiro, University of North Carolina, Chapel Hill, NC

ABSTRACT

As SAS® code becomes more complex and is run on large databases, how do we know if our results are correct? It is easy to assume that, if our code runs, it must be perfect. However, it is important to test SAS code against known data to ensure that you are getting the results you expect.

This paper will describe a simple mechanism for testing SAS code that can be expanded easily. Methods for generating the test database can cover quite a range—from the very simple spreadsheet to more complex random-number generation. These data are then imported into SAS, the code to be tested is run against the sample database, and the output is compared to expected results. This technique is a simple, but oft-forgotten method of ensuring that what you see is what you expect.

INTRODUCTION

Good software development practice involves writing a specification for the software, designing software to meet the specification, implementing the design as code, and testing the code. The development of SAS code to run queries is often done in a fast-paced, need-it-yesterday environment. In this type of environment, formal specifications and testing often get left out, which can lead to unexpected results.

Why is testing important? Large databases may contain millions of records. Seemingly small errors in the analysis code can lead to large differences in the output. Also, since the data sets are large, “truth” is often unknown, so results are assumed to be correct. This paper provides a simple technique for testing SAS code to ensure that your code does what you expect it to do.

HOW IS TESTING DONE?

Software is often developed in units. Each unit represents a step in the process of going from point A to point B. When all units are combined, the input to point A results in the output at point B. When an analysis is complex and contains many steps, the analysis software may contain many units. Testing of the units independently allows the source of errors to be more readily identified and fixed. Once each unit is functioning correctly, the units may be tested in combination.

For each unit and for the software as a whole, the formal software specification contains documentation of the functional requirements. This functional specification links inputs to software behaviors and outputs. These links between inputs and outputs provide the basis for testing the software. Each link requires at least one test case to ensure that the specification is met.

The steps involved in testing include using the specification document to (1) define the units to be tested, (2) identify the inputs and expected outputs for each unit, and for the application as a whole, and (3) create a database that contains inputs and expected outputs. The SAS code then is run against the test database (step 4), and (5) the actual output of the SAS code is compared to the expected output. Each step is described in detail below. The SAS code example used was developed as part of an injury surveillance project.. The code generates a value for two fields, intent and manner, based on the value of the ecode field (an indicator of the cause of the injury). Since only one unit will be tested in this example, step 1 is not described below.

STEP 2: IDENTIFYING INPUTS AND EXPECTED OUTPUTS

The functional specification for the unit to be tested links one input variable (*ECode*) to two output variables (*intent* and *manner*). A partial listing of the relationships between the input and output variables is defined in the table

below. Thus, this table from the functional specification provides the information needed for testing: the inputs, expected outputs, and the relationships between them.

ECode	Manner	Intent
E820.0-E820.9	Cut/pierce	Unintentional
E956	Cut/pierce	Self-inflicted
E966	Cut/pierce	Assault
E986	Cut/pierce	Undetermined
E830.0-.9, E832.0-.9, E910.0-.9	Drowning	Unintentional
E961, E968.0, E968..3, E979.3	Fire/burn	Assault

This specification was implemented as a macro in SAS:

```
%macro Matrix (ilibref=, olibref=, inset=, outset=);
data CauseOfInjury;
set InjuryData;

*Intent;
if 'E800' le substr(ECode,1,4) le 'E869' or 'E880' le substr(ECode,1,4) le 'E929'
then intent = 'Unintentional';

else if 'E950' le substr(ECode,1,4) le 'E959' then intent = 'Self-Inflicted';

else if 'E960' le substr(ECode,1,4) le 'E969' or substr(ECode,1,4) = 'E979' or
ECode = 'E999.1' then intent = 'Assault';

else if 'E980' le substr(ECode,1,4) le 'E989' then intent ='Undetermined';

else if 'E970' le substr(ECode,1,4) le 'E978' or 'E990' le substr(ECode,1,4) le
'E998' or ECode = 'E999.0' then intent = 'Other';

else if 'E870' le substr(ECode,1,4) le 'E879' or 'E930' le substr(ECode,1,4) le
'E949' then intent = 'Other';

*Manner;
if substr(ECode,1,4) in ('E920' 'E956' 'E966' 'E986' 'E974') then manner =
'Cut/pierce';

else if substr(ECode,1,4) in ('E830' 'E832' 'E910' 'E954' 'E964' 'E984') then
manner = 'Drowning';

else if 'E880' le substr(ECode,1,4) le 'E886' or substr(ECode,1,4) in ('E888'
'E957' 'E987') or ECode in ('E968.1') then manner = 'Fall';

else if 'E890' le substr(ECode,1,4) le 'E899' or substr(ECode,1,4) in ('E924') or
ECode in ('E958.1' 'E958.2' 'E958.7' 'E968.0'  'E968.3' 'E979.3' 'E988.1' 'E988.2'
'E988.7') or substr(ECode,1,4) in ('E961') then manner = 'Fire/Burn';

run;
%mend Matrix;
```

STEP 3: CREATING THE TEST DATABASE

The purpose of the testing is to ensure that the results match those defined by the specification document, so the test database should be developed from your specification document, not from the SAS code. The database required for testing this macro is a simple database in that it includes just three datapoints. The three datapoints in the database are the input variable (*ECode)* and the expected output variables. In the example below, the database was created in Excel. Note that the two expected output datapoints (manner, intent) were named with a “t” (for test) as a prefix to the name. In a later step, we will compare the output variables of the macro (manner and intent) to the expected values (tManner, tIntent). In the interest of space, the full test database is not copied in to this paper. The actual test database contains at least one record for each possible value of *ECode*. Because the if-then statements in the macro contain large ranges of ECode values, and many ECode values can result in the same output, the number of records in the actual test database is quite large.

ECode	tManner	tIntent
E920	Cut/pierce	Unintentional
E920.5	Cut/pierce	Unintentional
E956	Cut/pierce	Self-Inflicted
E964	Drowning	Assault
E966	Cut/pierce	Assault
E974	Cut/pierce	Other
E986	Cut/pierce	Undetermined

The test database is imported in to SAS as a SAS data set.

```
PROC IMPORT OUT=TestDB
     DATAFILE= "C:\MySAS\TestDatabase\TestDB.xls" DBMS=EXCEL REPLACE;
     SHEET="Sheet1$";
     GETNAMES=YES;
     MIXED=NO;
     SCANTEXT=YES;
     USEDATE=YES;
     SCANTIME=YES;
RUN;
```

If the SAS macro had contained more than one input variable, then the test database would need to consider all possible combinations of the three input variables. For example, if a macro has three input variables (Alpha, Beta, and Gamma), each of which has 2 possible values (0, 1), then the test database would need to have one record for each of the following input variable value combinations.

Alpha	Beta	Gamma
1	1	1
1	1	0
1	0	1
1	0	0
0	0	0
0	1	1
0	1	0
0	0	1

As the number of variables, input or output, increase, and the number of values for each of these variables increase, then the size of the test database also increases.

STEP 4: RUN SAS CODE AGAINST TEST DATABASE

In this step, the macro is run against the test database. The output data set is called TestMatrix.

```
%Matrix (ilibref = work, olibref = work, inset=TestDB, outset=TestMatrix);
```

The *TestMatrix* data set contains the input variable, *ECode*, the output variables generated by the SAS code (*intent, manner),* and the expected output variables (*tIntent, tManner)*.

STEP 5: COMPARE ACTUAL OUTPUT TO EXPECTED VALUES

Comparison of the actual values for the output variables against the expected values is made easier by having both sets of output variables in the output data set. The comparison can be done by simply running a proc freq on the output variables, looking for the records where the expected values and the output values are not the same.

```
PROC freq data=TestMatrix;
  where tManner ne Manner;
  tables ECode * Manner / norow nocol nopercent;
  title 'Manner comparison';
run;
```

```
PROC freq data=TestMatrix;
  where tIntent ne Intent;
  tables ECode * Intent / norow nocol nopercent;
  title 'Intent comparison';
run;
```

If the *intent* or *manner* values do not match the expected values, then the code should be inspected to identify why *ECode* values are not being mapped correctly. If changes are made to this unit of code, then steps 4 and 5 should be repeated. If an *ECode* value is missing from the test data set, then return to step 3 and correct the test data set.

CONCLUSION

Testing software to be sure that the results are correct is crucial in any environment. It is a step often missed, due to the pressure to get work done quickly, but the impact of not doing testing can be significant. The process demonstrated in this paper only covered one unit with a very simple structure (single input variable). Testing of complex software with multiple units and multiple input variables would be more involved and time-consuming. However, as the complexity of the code increases, so does the probability of coding errors.

As software units are added for testing, the test database can be expanded to accommodate the new fields that are needed. At the end of all of the unit testing, the resulting test database will allow the entire software package to be tested, since the test database will contain all required fields.

By running these simple tests, you can increase your confidence that your code not only runs without errors or warnings, but also that your results match what is expected from your specification document. .

CONTACT INFORMATION

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

Name: Sharon Schiro
Enterprise: University of North Carolina
Address: CB# 7228, Dept of Surgery
City, State ZIP: Chapel Hill, NC 27599-7228
Work Phone: 919-966-6263
Fax: 919-966-0369
E-mail: Sharon_Schiro@med.unc.edu
Web: n/a