%%Assert() your way to sleep-filled nights:  
**A one line data validation macro**

Quentin McMullen, Siemens Healthcare Diagnostics Inc., Norwood MA

**ABSTRACT**
Data are messy, and should never be trusted. This paper presents a simple SAS® macro, %Assert, which allows the user to state a Boolean expression that is expected to be true, e.g.: %assert(Age>0). If the expression is false, an error message is printed to the log. The macro may be used to verify expectations about data at any point in a DATA step. Use of the %Assert macro provides two important benefits: it provides a method for automated error detection, enhancing confidence that results are correct; and it allows the programmer to explicitly state expectations regarding the data being processed, enhancing the readability of code. Assertions are a critical tool for both defensive coding and test-driven development (Wright, 2006). Principles of macro design encountered during the development of the macro are addressed.

**INTRODUCTION**
Many programming languages have an assert statement, which provides a simple way to verify data. The programmer asserts that an assumption about the data is true, and if that assertion proves false, an error message is generated. The assertion may be any Boolean expression. Assert statements are an essential tool for defensive coding. SAS does not have an ASSERT statement, but this paper presents a macro, %Assert, which may be called in a DATA step to generate the equivalent of an ASSERT statement. The ease with which %Assert may be invoked encourages the use of defensive programming practices, and often pays off in early detection of problematic data or incorrectly coded algorithms.

**ASSERTING IN SAS**
SAS does not have an ASSERT statement. Despite that, in the DATA step, programming an assertion can be as simple as writing an IF statement. The step below computes age as a continuous variable from date of birth and consent date, without any error handling code.

```sas
data want;
   set have;
   Age=(ConsentDt-DOB)/365.25;
run;
```

The assignment statement is simple, but what are some of the things that could go wrong? Either ConsentDt or DOB could be missing (SAS will put a note to the log, and continue processing). The date of birth could be after the Consent Date (resulting in a negative age). It would be nice if you could easily check that the values of ConsentDt, DOB, and AGE are in the expected range.

One approach to checking data is to use PROC PRINT/ PROC FREQ/ PROC MEANS to generate a check table which summarizes the source variables (ConsentDT and DOB) and the derived variable (AGE). But that approach requires you to manually review the output, every time the program is run (or at least every time the program is modified, or run on new data). As an appropriately lazy programmer, how could you automate this check?

```sas
data want;
   set have;
   if NOT ( "01JAN2011"d <= ConsentDt <= today() )
      then putlog "ERROR: invalid ConsentDt";
   if NOT ( "01JAN1900"d <= DOB <= today() )
      then putlog "ERROR: invalid DOB";
   Age=(ConsentDt-DOB)/365.25;
   if NOT ( 18 <= Age <= 100 ) then putlog "ERROR: invalid age";
run;
```
Note that there are two types of assertions in the step, where the programmer states expectations regarding the data. Preconditions are asserted to verify the input data to a step or algorithm. The above step checks two preconditions: ConsentDt is between January 1, 2011 and today's date; and date of birth is between January 1, 1900 and today's date. Postconditions are asserted to verify the data output from a step or algorithm. The code checks that the derived age must be between 18 and 100. Asserting a precondition will alert you to problems in the input data, and asserting a postcondition will alert you to problems in the output data, which may result from problems in an algorithm or the input data. If an assertion is false, an error message will be printed in the log.

In addition to generating error messages when an assertion is false, these assert statements serve an important purpose even when the data are clean. They allow the programmer to communicate their expectations regarding the data. The future programmer who inherits code with assertions can immediately see that the original programmer expected ConsentDt to be no earlier than January 1, 2011 (perhaps the trial start date), and expected all participants to be at least 18 years old when they consented to participate in the trial. For this reason, it is often helpful to leave assertions in place in production code, even when data are "known" to be clean.

%ASSERT
The one line macro which is helpful in generating assertions is:

%macro assert(assertion);
  if NOT (&assertion) then
      putlog "ERROR: Assertion (%superq(assertion)) is FALSE."
  %mend assert;

The macro generates a DATA step IF statement, which tests an assertion (any valid expression), and writes an error message to the log if the assertion is false. The step computing age could be rewritten using %Assert as:

data want;
  set have;
  %assert("01Jan2011"d <= ConsentDt <= today() )
  %assert("01Jan1900"d <= DOB <= today() )
  Age=(ConsentDt-DOB)/365.25;
  %assert( 18 <= Age <= 100 )
run;

DESIGN OF %ASSERT
Each call to %Assert generates a single DATA step IF statement (not a macro %IF). The main job of the macro language is to be a SAS code generator. Most macros do little more than save you from having to type SAS code yourself. %Assert is a keystroke-saving macro. It does not provide new functionality to the SAS programmer, but by making it easier to generate an assert statement, %Assert makes it more likely that programmers will test their assumptions about data.

When designing a macro, the design of the parameterization and choice of macro name are critical considerations. Macro design should attempt to maximize both functionality and ease of use. Often these are competing considerations which must be balanced against each other. In designing for ease of use, it is helpful for the designer to rely on the assumption that the macro user will have knowledge of the SAS language. Macros that employ constructs familiar to SAS users will be easier to use.

The design of %Assert attempts to increase ease of use by limiting the number of parameters. There is just one parameter in the one-line version, and only one required parameter in the enhanced version described below. Compare that to an alternative parameterization I have seen, with three required parameters:

%Assert(leftside=, operator=, rightside=). With such a parameterization, to check that age is at least 18 would require an invocation like: %Assert(leftside=age, operator= >=, rightside=18). Having multiple parameters makes the macro more difficult to invoke, and actually limits its functionality. Passing the assertion as just a single parameter allows the user to specify any valid SAS expression, ranging from simple, %Assert(1) /*obviously true*/ , to complex, %Assert((n(height,weight,bmi)=3) and (bmi=weight/height**2) ) . The user passes a SAS expression, and SAS programmers already know how to write expressions.
Also critical to the design is that one macro can be used to define any logical assertion. An alternative approach might be to define multiple macros: %AssertEqual, %AssertInRange, %AssertNotMissing etc, each designed for a unique “type” of assertion. Such a design would also increase complexity (requiring the user to remember a group of macro names, and the designer to maintain a group of macros), without increasing functionality.

The macro places the entire expression passed by the user inside of parentheses:

```sql
if NOT (%assertion) then ...
```

The parentheses are necessary to ensure that the entire expression passed by the user is negated. If the parentheses were not added, and a user called

```sql
%assert(0 or 1)
```

The macro would generate

```sql
if NOT 0 or 1 then ...
```

which would evaluate as TRUE, because without parentheses, the order of operations evaluates this as

```sql
if (NOT 0) or (1) then ...
```

With the parentheses, the macro generates

```sql
if NOT (0 or 1) then ...
```

which correctly evaluates as FALSE.

Because the macro adds parentheses around the expression in the IF statement, it is typically not necessary for the user to include parentheses around the expression in the macro call. I typically code:

```sql
%assert ( 18<=age<=100 )
```

rather than

```sql
%assert ( (18<=age<=100) )
```

There are times when it is necessary to include the parentheses, since they can mask symbols that may confuse the macro processor. For example, when asserting an equality, if one were to code:

```sql
%assert ( gender="Female" )
```

SAS will return an error, because the macro processor thinks you are trying to specify a value of "Female" to a macro parameter named GENDER. If you place the expression in parentheses, they will hide the equals sign from the macro processor:

```sql
%assert ( (gender="Female") )
```

An alternative would be to avoid the equals sign altogether by substituting its mnemonic equivalent:

```sql
%assert ( gender eq "Female" )
```

Parentheses also mask a comma when a macro is invoked, so an assertion such as:

```sql
%assert ( gender IN ("Male","Female") )
```

will execute, without the macro processor seeing the comma and confusing it for a parameter delimiter.

On the putlog statement, the %superq() function is used to mask any characters in the assertion which might cause problems for the compiler:

```sql
putlog "ERROR: Assertion (%superq(assertion)) is FALSE."
```

For example, if the user supplies an assertion that includes double quotes:

```sql
%assert ( gender eq "Female" )
```

%superq() will mask the quote marks, preventing the compiler from seeing them as defining the end of the text string specified in the PUTLOG statement. For a discussion of macro quoting, see Whitlock (2003).

EXAMPLES

Several examples of using %Assert for different purposes are discussed below.

**Assert that all values of a variable are in the expected range**

A simple use of %Assert, as shown above, is to validate the values of a single variable:

```sql
%assert ( 18<=age<=100 )
```

Cody (2008) suggests a method for checking data ranges, using formats rather than hard-coding the limits in the DATA step code. This approach can be implemented using %Assert, and one can imagine expanding it to build a tool that would check all variables in a data set.

```sql
proc format ;
  value ageck
    18-100='Valid'
    other='Invalid'
  ;
run ;
```
data _null_; set have; %assert(put(age,ageck.)="Valid") run;

Note that the PUT function does not cause a problem for %Assert, because the assertion can be any logical expression, and in SAS an expression can be built from variables, constants, functions, and operators.

Assert the existence of a data set

data _null_; set have; %assert(exists(sashelp.shoes)) run;

The exists() function returns 1 if the data set exists, 0 if it does not. In a logical expression, SAS evaluates 0 as False. Thus calling %Assert(exists(DATA SET)) will generate an error message if the data set does not exist.

Assert that a key is unique

data _null_; set have; by id; %assert(first.id and last.id) run;

Assert that there are no missing values in a data set

data _null_; set have; %assert( cmiss(of _all_) = 0 ) run;

Assert an "if and only if" condition is true

Often there are skip patterns in survey data, and the "missingness" of some questions should depend on responses to earlier questions. For example, in a survey of cancer history, a question regarding prostate cancer is only applicable to men.

data _null_; set in; %assert( gender IN ('M','F') ) %assert( ProstateCancerHx IN (.,0,1) ) %assert( (gender='M') = (not missing(ProstateCancerHx)) ) run;

The last expression above is slightly more complex than others. It asserts that all males completed the prostate cancer history question, and that everyone who completed the prostate cancer question was male. This works because of the way that SAS evaluates expressions. In evaluating the expression:

\[(\text{gender}='M') = (\text{not missing}(\text{ProstateCancerHx}))\]

Each of the sub-expressions will return 1 (True) or 0 (False), and then those values will be compared. So when gender='M' and ProstateCancerHx=1 it resolves to:

\[1 = 1 \text{ (true)}\]

When Gender='M' and ProstateCancerHx=. it resolves to:

\[1 = 0 \text{ (false)}\]

Assert that expectations for a merge are met

When merging data sets, it is particularly important that the programmer understand the "unique-ness" of the keys in each data set, but this understanding is rarely communicated in the code. In reviewing a program, if I see:

data c; merge a b; by id; run;
questions enter my head: is this a one-to-one merge, a one-to-many merge, or a (disastrous) many-to-many merge? Should all records in work.a and work.b match, or is it okay if there are mismatches? Compare that code to below step, with assertions added.

data c ;
merge a(in=ina) b(in=inb) ;
by id ;
%assert(first.id and last.id)
%assert(ina and inb)
run ;

Adding the %Assert calls has made it clear that the programmer has designed this to be a one-to-one merge (first assertion), and that all records in work.a and work.b should match (second assertion). The %Assert calls have made the code more readable. They communicate information not contained in the main code. As an added benefit, if somehow a duplicate record appeared in one of the input data sets, or there was a mismatch between them, an error message would be sent to the log.

Macro calls do not need semicolons

Note that there is no semicolon after each call to %Assert above. SAS statements need semicolons to end them, macro calls do not. %Assert generates a full IF statement, with the required closing semicolon. If you put a semicolon after the macro call, you will generate a null statement, which is usually not problematic, but can cause errors in rare situations.

Unfortunately, the SAS syntax highlighter in the Enhanced Editor and Enterprise Guide can often be confused by macro code. With macro invocations, the coloring of the code may suggest that you add a semicolon to the end of a macro call. The coloring is a guide to the programmer, which has no impact on how code is actually compiled and executed. It can be safely ignored.

ENHANCED %ASSERT

While the one-line version of the macro is useful, the functionality can be expanded through the addition of parameters. Below is an enhanced version of the macro, followed by comments.

%macro Assert
(assertion /*(Req) Expression being asserted */
, msg= /*(Opt) Message to put to log if false */
, msglevel=1 /*(Opt) Message level, 1=error, 2= warn, 3=note */
, errors=10 /*(Opt) Max number of messages to put to log */
, errorflag= /*(Opt) Name of variable to create, flagging records where assertion is false */
, force=0 /*(Opt) Boolean: force assertion to run, even if the global parameter &debug is set to 0 */
);

/* if user set global program parameter debug=0 (off),
and assertion is not forced on by local parameter force=1, skip assertion;
%if %symglobl(debug) %then %do;
%global debug; /*Boolean: 1=debug mode on (assertions run);
%if &debug=0 and &force=0 %then %goto mexit;
%end;

%set flag for false assertion to 0;
%if %superq(errorflag) ne %str() %then %do;
if &errorflag=. then &errorflag=0;
%end;
*/
if NOT (&assertion) then do;
  /*only print as many messages as specified in &errors;
   if __msg&sysindex < &errors then do;
     putlog
       %if &msglevel=1 %then %do;
       "ERROR: "
       %end;
     %else %if &msglevel=2 %then %do;
       "WARNING: "
       %end;
     %else %if &msglevel=3 %then %do;
       "NOTE: "
       %end;
     %else %put ERROR: (%nrstr(%%)&sysmacroname) Bad parameter for msglevel=&msglevel;

     "Assertion (%superq(assertion)) is FALSE. " &msg
   ;
   __msg&sysindex ++1;
   if __msg&sysindex = &errors then putlog
     "NOTE: Max number of false assertions
     (%superq(assertion)) reached, no more messages."
   end;

   %if %superq(errorflag) ne %str() %then %do;
     &errorflag=1; %*set flag for assertion false;
   %end;
   end;
   drop __:;
%mexit:
%mend assert;

The enhanced macro has added several parameters, but note that only the first parameter, ASSERTION, is required for a basic call.

The MSG parameter allows the user to specify additional text to be printed to the log when an assertion is false. I often use this to print the values of variables involved in the assertion, as well as record identification numbers, e.g.:
%assert(age>=18,msg=(id age)(=))

May return a log message such as:
ERROR: Assertion (age>=18) is FALSE. id=3 age=5

The MSGLEVEL parameter specifies whether false assertions should be written to the log as ERRORs WARNINGs or NOTEs.

The ERRORS parameter limits the number of messages that will be written to the log. The DATA step variable __msg&sysindex is used as a counter variable to count the number of log messages written when the assertion is false. &SYSINDEX is used to ensure that every invocation of %Assert has its own counter, which is important when a single DATA step calls %Assert multiple times.

The ERRORFLAG parameter allows the user to specify the name of a data set variable that will be set to 1 when an assertion is false. When there are multiple assertions in a step, the user can choose to create independent flag variables for each assertion:
data want;
  set have;
  %assert (ConsentDt<=today() ,errorflag=BadConsentDt)
  %assert (dob<=today() ,errorflag=BadDob)
  age=(ConsentDt-dob)/365.25;
  %assert ( 18<=age<=100 ,errorflag=BadAge)
run;

or create a single flag variable which will be set to 1 if any assertion failed:

data want BadData;
  set have;
  %assert (ConsentDt<=today() ,errorflag=BadRecord)
  %assert (dob<=today() ,errorflag=BadRecord)
  age=(ConsentDt-dob)/365.25;
  %assert ( 18<=age<=100 ,errorflag=BadRecord)

  if BadRecord then output BadData;
  else output want;
run;

The FORCE parameter works along with a global parameter, DEBUG, to let a user switch assertions on or off. As mentioned above, I typically leave assertions on in my production code. I use %Assert both for debugging and for run-time validation. Others may want to use %Assert for debugging only. Once the program has been debugged, they may want to turn off the assertions, to avoid spending the extra CPU cycles. The global macro variable DEBUG is a switch a user may create in the calling program, which allows the user to indicate whether they want assertions to be generated or not. A user can turn off all assertions by setting DEBUG=0. If there are specific assertions which the user would like to be run regardless of the value of DEBUG, they may specify FORCE=1 when they call %Assert.

CONCLUSION
Logical assertions are a powerful debugging tool that provides run-time validation of both data and algorithms. The %Assert macro makes it easy for SAS programmers to generate assertions, thereby encouraging programmers to adopt the use of assertions as part of a defensive coding practice.

REFERENCES


DISCLAIMER
The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Siemens Healthcare Diagnostics, Inc.

ACKNOWLEDGEMENTS
I am grateful to Ian Whitlock for suggesting that I develop this macro, and for his comments on an early draft of this paper, and for much, much more.

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 registered trademarks or trademarks of their respective companies.
CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:

Quentin McMullen
Siemens Healthcare Diagnostics, Inc.
2 Edgewater Drive
Norwood, MA 02062
qmcmullen.pub@gmail.com