PharmaSUG 2015 - Paper TT08

A Collection of Items from a Programmers' Notebook

David Franklin, Real-World & Late Phase Research, Quintiles, Cambridge, MA
Cecilia Mauldin, Independent Consultant, Chapel Hill, NC

ABSTRACT

Every programmer should have a notebook with notes and pieces of code that are of interest to them. Some will be categorized as personal, notes only of interest to them, but most are of interest to a broader range of SAS® programmers. This paper brings together ten of the most useful tips from the notebooks of two SAS programmers with nearly fifty years of combined experience that will be useful to beginner and experienced SAS programmer alike. Topics include advice for using the SAS Editor; combing SAS data sets; reviewing SAS Logs; creating Excel files from SAS data sets; creating SAS data sets from Excel files; creating editor macros in Enterprise guide; splitting the info from &sysinfo; keep only the character, numeric or printable characters in a strip of text; creating empty data sets; to that little bit of code that calculates the first, middle or end date of month. Bring along your notebook and take away an idea or two.

INTRODUCTION

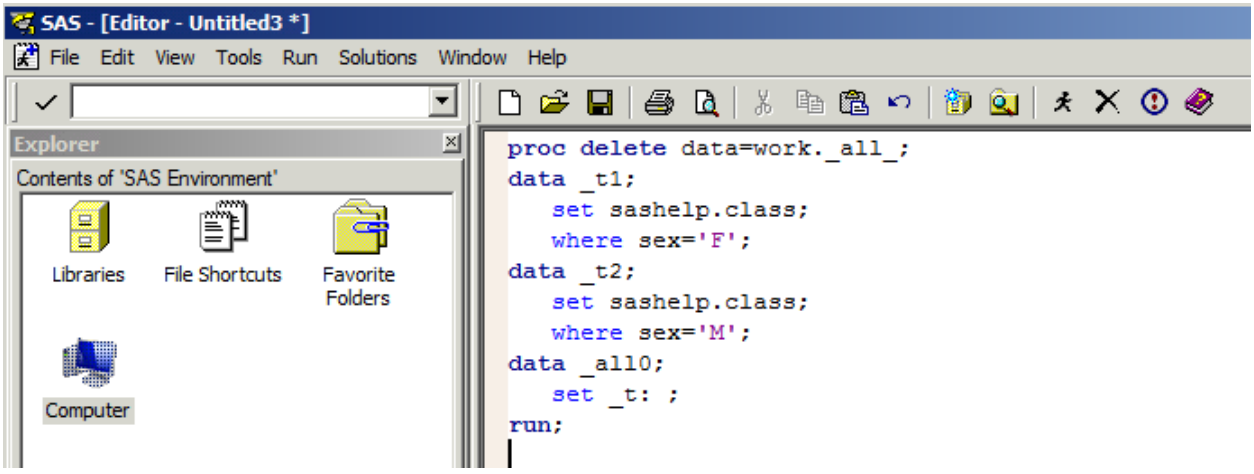
Every SAS programmer has that piece of code or interesting programming note that is put on a scrap of paper or tries to remember it in their head. Unfortunately, using this approach, that note or programming code is lost within hours -- to avoid this, the note should be saved in a notebook, electronic or paper.

With too many years of SAS experience, two programmers have kept such notebooks for many years and share, in this paper, ten of the most referenced of items for SAS programmers to also share and use.

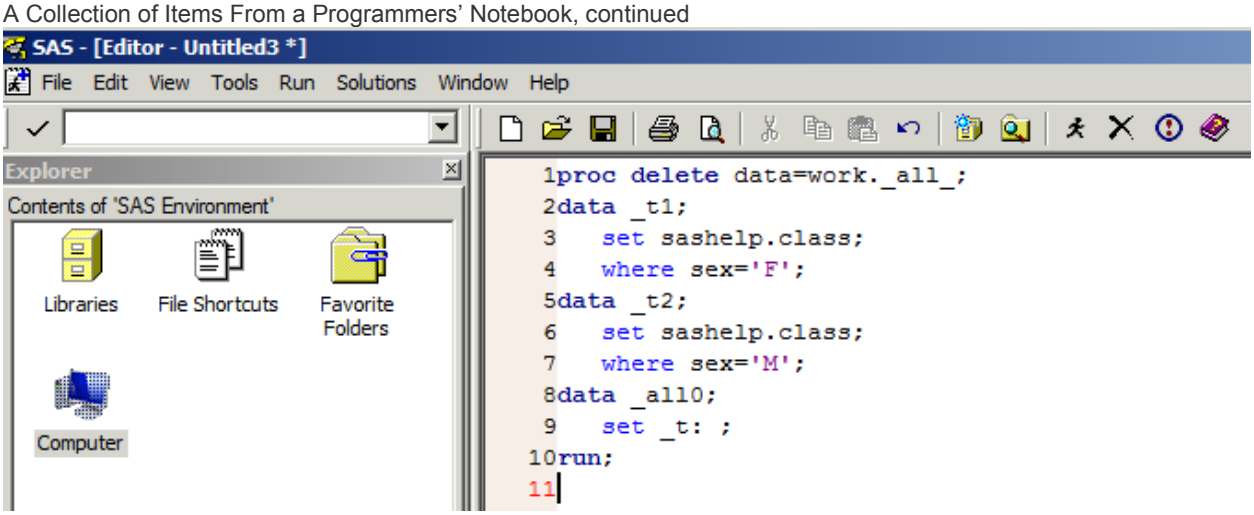
THE SAS EDITOR

Enterprise Guide has only the Enhanced Editor, but in the Windowing environment there are two default editors in interactive mode available within SAS, the Enhanced Editor and the 'Old Style' editor, known as the DMS Editor. While the Enhanced Editor is sometime better than the DMS Editor, it is sometimes a lot quicker to edit in the latter without having to reach across and edit with a mouse.

One of the key pieces of information a programmer likes to know when editing a program is what line number are on and this is switched on by the NUMS command on the Command Line, as shown below:



and results in the following display in the program window:



Key commands with editing text are within the Windows environment are much the same as elsewhere within Windows, but a few other useful commands are:

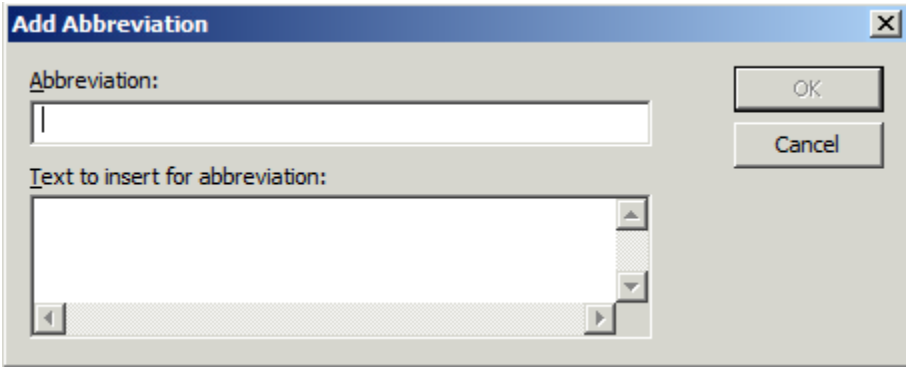
F11	Command Bar (not available in Enterprise Guide)
:I*n* or :IA*n*	Insert *n* lines after the current line
:IB*n*	Insert *n* lines before the current line
:D*n*	Remove *n* lines starting at the current line
:R*n m*	Repeat *m* lines from the current line, *n* times

A few useful key combinations are:

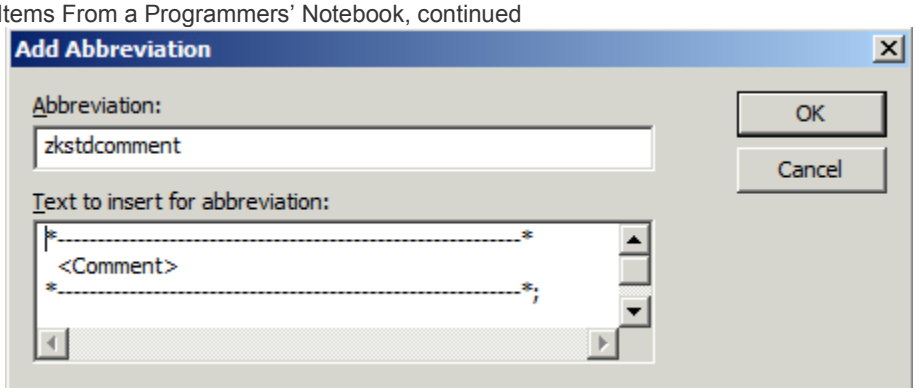
Ctrl +]	Find matching parenthesis or bracket
Ctrl + G	Goto Line
Ctrl + Shift + U	Put highlighted text into uppercase
Ctrl + Shift + L	Put highlighted text into lowercase

Commenting code, especially multiple lines of code or text, can be a fraught with issues, notably forgetting to start a line with the '/*' and/or end with '*/' -- an easier way to do it is by highlighting the text and pressing the **Ctrl** key and '/' together. To un-comment code, hit the **Ctrl + Shift + /** keys together.

Frequently we write standard pieces of code and we usually do not like having to write it over and over again, so in the editor we can do that though making an abbreviation. To do this action, hit the keys **Ctrl + Shift + A** and a window similar to that below appears:



were you would enter a set of text for an abbreviation, then the text that it is to use in the lower box. The following is a setting for a standard comment that we often use:



and after being entered into the SAS Editor by pressing OK, results in wherever the word 'ZKSTDCOMMENT' being replaced with the line

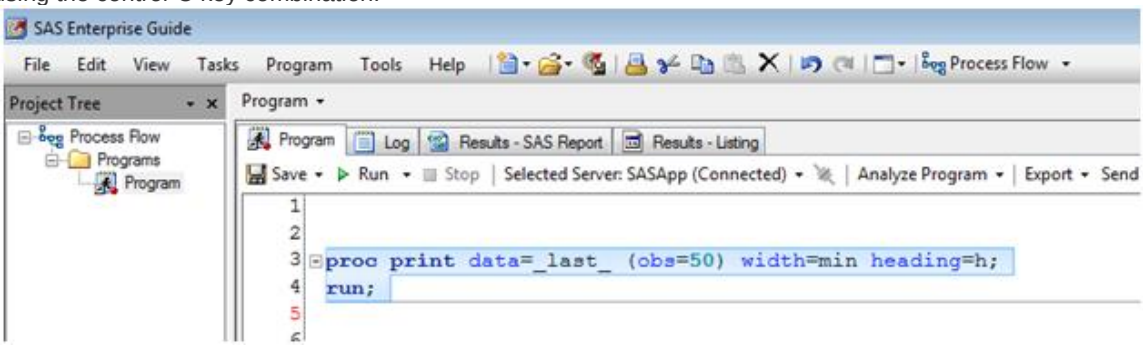
```
*------------------------------------------------------------*
  <Comment>
*------------------------------------------------------------*;
```

One word of note, that while it is possible to type in any text for an abbreviation, it is wise to stay away from SAS keywords due to possible confusion between SAS knowing it is a keyword you type in versus a defined abbreviation.

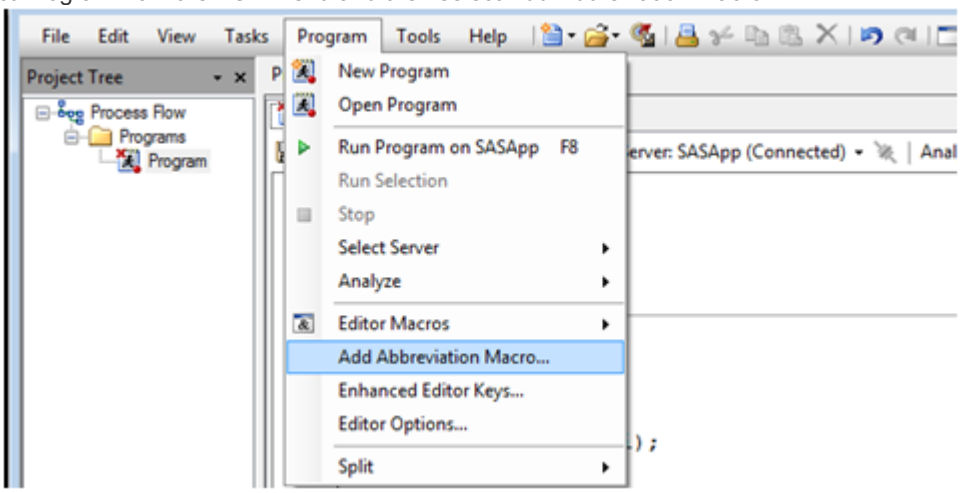
CREATING EDITOR MACROS IN ENTERPRISE GUIDE

Do you have pieces of code that you type many times like headers of a program or pieces of code for which you want reminders? In the example, the piece of code I type several times is PROC PRINT and the name I want to use is 'PRI'.

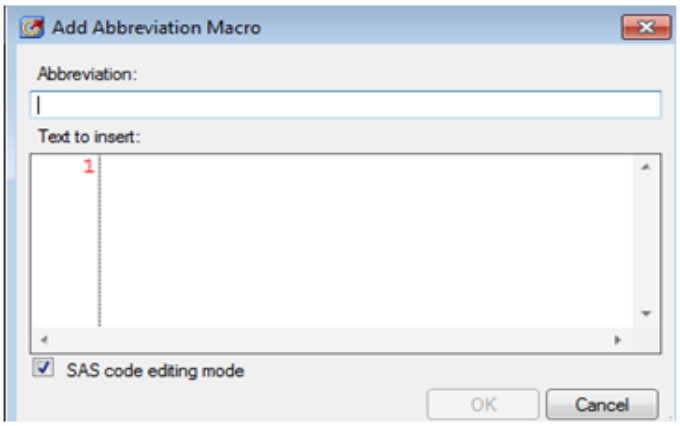
1. Type in the Enterprise Guide editor the piece of code you want to store and copy it to the Clipboard memory using the control-C key combination.



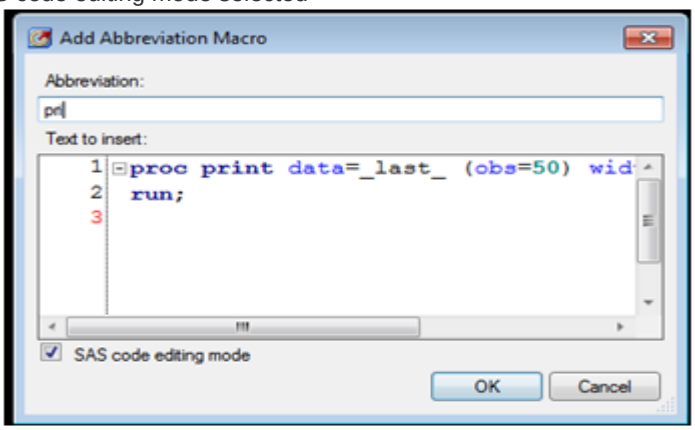
2. Select Program from the TOP menu and then select Add Abbreviation Macro…



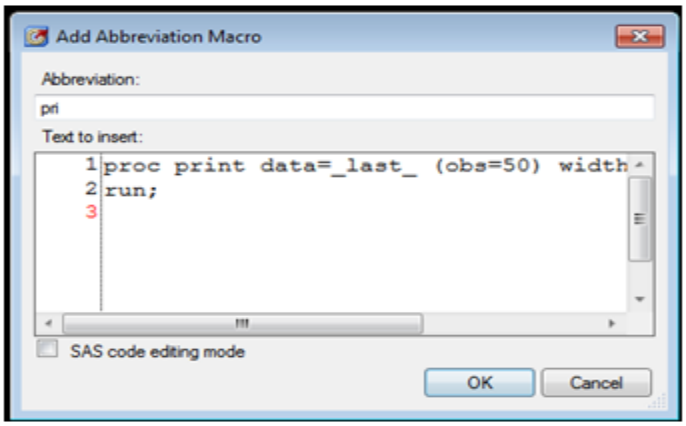
A window will pop-up:



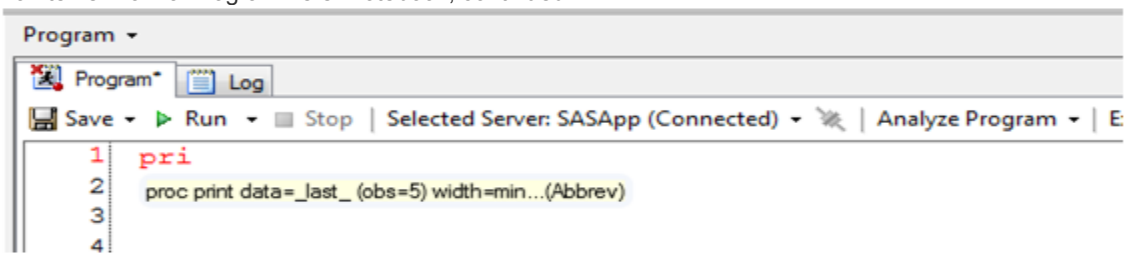
3. Type the name of your editor macro in the Abbreviation line, in this example, the name of the editor macro is PRI, and copy the saved code with Control-C to the box in the Text to Insert box. Your box will look like if you keep the SAS code editing mode selected



or like this if the SAS code editing mode box is not selected

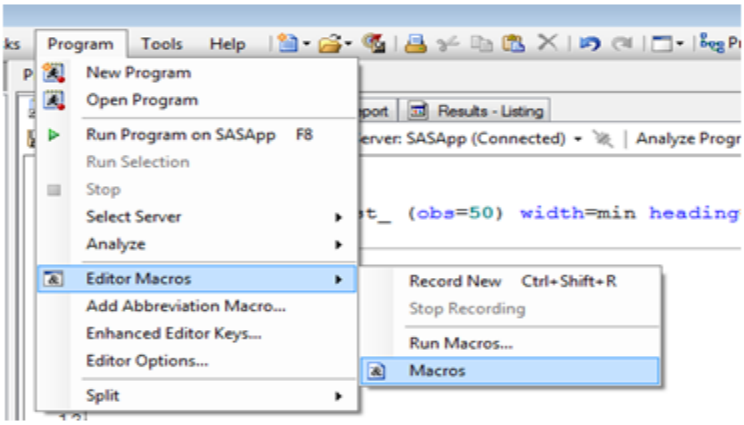


4. After both fields are entered, you can press the OK key
5. You are done, the next time you type the text used in the abbreviation, pri in this example, you will get the option of getting it, press return if you want it or keep if you don't want to use it.

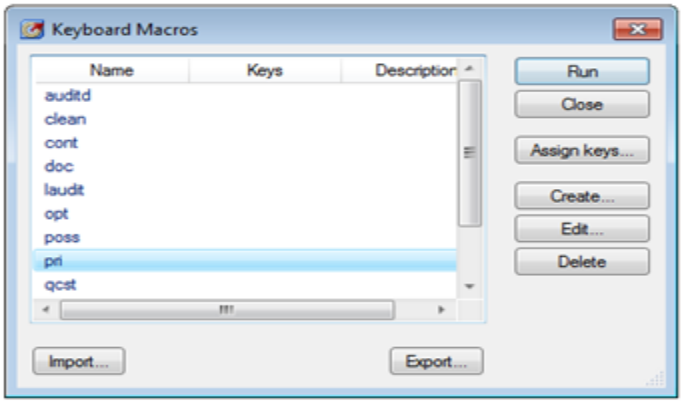


If you need to modify an existing macro, the steps are

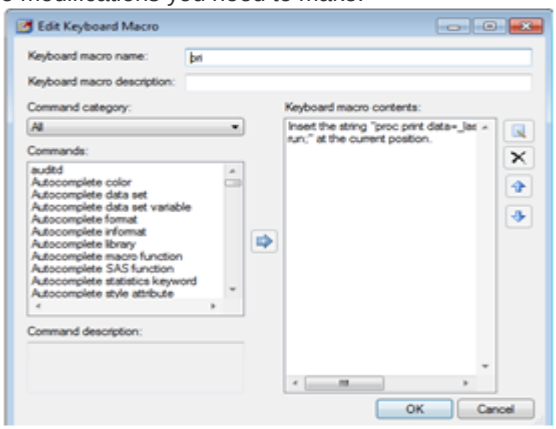
1. Select Program from the TOP menu and then select Editor Macros and Macros



2. A list with your existing macros you have will appear, select the macro you want to edit.



3. Select Edit and make the modifications you need to make.



COMBINING DATA SETS

There are two basic ways SAS data sets are combined, one being where two or more data sets are being merged by a common variable or list of variables, and a second an append where one set of data is added to a second second data set.

A large number of papers, including one titled "Countdown of the Top 10 Ways to Merge Data" written by the author and presented at PharmaSUG 2010, give numerous ways of merging data -- this topic itself could fill an entire paper, so refer to outside sources on this.

A similar situation is with appending data where there numerous papers and documentation on the subject.

But there is one method that is not well known, and it refers to where you have one data set with one observation and one or more variables that are not in the data set you want to combine, for example a number representing a total to be put into a data set with summary information -- to do this you need two set statements, as the following illustrates:

```
1    proc delete data=work._all_;

2    data _incidence;
3    length area $10;
4    infile cards;
5    input area $ count;
6    cards;

NOTE: The data set WORK._INCIDENCE has 4 observations and 2 variables.

11   ;
12   run;
13   proc summary data=_incidence;
14      var count;
15      output out=_total (drop=_type_ _freq_) sum=total;
16   run;

NOTE: There were 4 observations read from the data set WORK._INCIDENCE.
NOTE: The data set WORK._TOTAL has 1 observations and 1 variables.

17   data  all0;
18   if _n_=1 then set _total;
19   set _incidence;
20   run;

NOTE: There were 1 observations read from the data set WORK._TOTAL.
NOTE: There were 4 observations read from the data set WORK. INCIDENCE.
NOTE: The data set WORK._ALL0 has 4 observations and 3 variables.

21   proc print data=_all0;
22   run;

NOTE: There were 4 observations read from the data set WORK._ALL0.
```

which result in the following output:

```
Obs    total    area     count

 1       75     NORTH      20
 2       75     SOUTH      22
 3       75     EAST       18
 4       75     WEST       15
```

The example above demonstrates how to merge a single observation with no common variable into a data set with multiple observations.

Another useful tip when appending multiple data sets with the same prefix, it is possible to put just put the first frew characters of the data set name, followed by the ':', as the following code demonstrates when appending data sets starting with '_t':

```
23   proc delete data=work._all_;
24   data _t1;
25      set sashelp.class;
```

```
26      where sex='F';

NOTE: There were 9 observations read from the data set SASHELP.CLASS.
      WHERE sex='F';
NOTE: The data set WORK._T1 has 9 observations and 5 variables.

27   data _t2;
28      set sashelp.class;
29      where sex='M';

NOTE: There were 10 observations read from the data set SASHELP.CLASS.
      WHERE sex='M';
NOTE: The data set WORK._T2 has 10 observations and 5 variables.

DATA _all0;
    SET t_:  ①
    INDSNAME=in_dataset; ②
    origin=in_dataset;   ③
run;
```

① Any data set that starts with the letters t_ will be added without having to explicitly write the data set name(s).
② The reserved word INDSNAME stores the name of the data set from which the current row is read in the temporary variable in_dataset.
③ Because the variable in_dataset is temporary, it is necessary to create a variable origin if to preserve the value of in_dataset.

```
NOTE: There were 9 observations read from the data set WORK._T1.
NOTE: There were 10 observations read from the data set WORK._T2.
NOTE: The data set WORK._ALL0 has 19 observations and 5 variables.
```

It must be noted from the example above that the data sets _T1 and _T2 have the variable attributes the same and where two or more data sets are appended in this manner it may be necessary to amend variable lengths, types and formats before using the SET statement with the ':'.

REVIEWING SAS LOGS

The SAS LOG is an important file to review. But how does someone review it efficiently for issues, particularly if it is hundreds of pages in length? Now one can even look at a SAS LOG that long and find all messages referring to ERROR or WARNING messages efficiently.

First though some options need to be set from their default so as to make any output useful, and by that means show the code, including macros, that was used to generate an output. Some of the SAS options worth considering are:

mlogic	causes the macro processor to trace its execution and to write the trace information to the SAS log
mlogicnest	enables the macro nesting information to be displayed in the MLOGIC output in the SAS log
mprint	displays the SAS statements that are generated by macro execution
mprintnest	enables the macro nesting information to be displayed in the MPRINT output in the SAS log
symbolgen	displays the results of resolving macro variable references

To avoid some additional issues, it is often advantageous to have set

mergenoby=	Specifies the type of message that is issued when MERGE processing occurs without an associated BY statement. Valid values are NOWARN (specifies that no warning message is issued, this is default), WARN (specifies that a warning message is issued), and ERROR (specifies that an error message is issued)
msglevel=	Specifies the level of detail in messages that are written to the SAS log. Valid values are N

(specifies to print notes, warnings, CEDA message, and error messages only, default) and I (specifies to print additional notes pertaining to index usage, merge processing, sort utilities, and Hadoop MapReduce jobs along with standard notes, warnings, CEDA messages, and error messages).

source2 specifies to write to the SAS log secondary source statements from files that have been included by %INCLUDE statements.

There are certainly more options that can be set but these are the core that the authors find useful.

Now the LOG checker itself. Some have LOG checkers that are already written, some can even be written in other languages including VBA, but for any such tool there are two features that are useful:

Search for text strings, at a minimum ERROR and WARNING messages

Be able to read multiple files in a single directory

A code fragment that is useful to fulfill these requirements is given below:

```
*Define location of LOG Files to process;
%let logdir=%str(c:\pharmasug2015\TABLES);

*Bring in ALL SAS LOG(s);
data _alllogs0;
   attrib  txt length=$256 informat=$char256. format=$char256.
          _fn length=$256
          _myinfile length=$256
          _ln length=8
           myinline length=8;
   infile "&logdir.\T*.log"  /*Here use t*.log for all table LOGS
                              but can be anything*/
          lrecl=256 filename=_myinfile line=_myinline length=len;
   input _txt $varying256. len;
   _fn=scan(strip(_myinfile),-1,'\');
    ln= n ;
proc sort data=_alllogs0;
   by _fn _ln;
run;

*Output findings;
title1 "QC of SAS LOG(s)";
data  null ;
   retain _k 0; *Internal counter;
   file PRINT;
   set _alllogs0 end=eof;
   by  fn  ln;
   if first._fn then do;
      put // @1 '***** LOG FILE: ' _fn /;
      _k=0;
   end;
   if index(_txt,'ERROR:') or
      index(_txt,'WARNING:') then do;
      _k+1;
      put _txt;
   end;
   if last._fn and _k=0 then put @1 '** NO ISSUES FOUND **';
   if eof then put / '/*EOF*/';
run;
```

The INFILE statement in the data step is key to what SAS LOGs are read into the LOG Checker -- in the code given, these are all LOG files with starting with 'T' in the directory specified by the LOGDIR macro variable. In the last data step, the only text that is searched for is 'ERROR:' and 'WARNING:' but this can be set to any text your group needs. Note that if a SAS LOG has no issues found then a message is output to that effect in the output file.

CREATING EXCEL FILES FROM SAS DATA SETS

There are a number of ways an Excel file can be created from a SAS data set, but the three methods here are the ones that the authors find useful.

First is the simple CSV file using a SAS data step. The following code will take take the CLASS data set and put it into a CSV file, delimited by the character '~':

```
data _null_;
   file 'c:\pharmasug2015\class.csv' dsd delimiter="~" lrecl=1024;
```

```
    set sashelp.class;
    put (_all_)(+0);
run;
```

A CSV file can be simply read into Excel and is an standard which can be read my almost all data related systems. The disadvantage here is that it is not strictly Excel and usually takes some user knowledge to read this type of file in.

The second method shown here is PROC EXPORT which can create CSV files, and Excel files if the SAS/ACCESS Interface to PC Files is installed. The basic syntax of the PROC EXPORT for either CSV or EXCEL files are

```
PROC EXPORT DATA=<libref.>SAS data set
            OUTFILE="filename"
            <DBMS=identifier>
            <REPLACE>;
```

where

<libref.>SAS data set	identifies the input SAS data set. data set options, including KEEP, DROP and WHERE, may also be used.
OUTFILE=	specifies the complete path and filename or a fileref for the output PC file, spreadsheet, or delimited external file. A fileref is a SAS name that is associated with the physical location of a file. To assign a fileref, use the FILENAME statement.
DBMS=	These include CSV (comma separated files), DLM (delimited files where the character is specified), TAB (tab delimited files), and XLSX (Excel files)
REPLACE	overwrites an existing file. If you do not specify REPLACE, the EXPORT procedure does not overwrite an existing file.

There are other DBMS values that relate to the Excel type file, e.g. XLS, but in recent times SAS documentation has suggested strongly that the XLSX value be used.

Some useful options for different types of file are:

DELIMITER=	specifies the delimiter to separate columns of data in the output file. You can specify the delimiter as a single character or as a hexadecimal value. This option can only be used in DBMS=DLM files.
NEWFILE=	when exporting a SAS data set to an existing Excel file (XLS or XLSX file), specifies whether to delete the Excel file and load the data to a sheet in a new Excel file. Default is NO.
PUTNAMES	determines whether to write SAS variable names as column headings to the first row of the exported data file. Default is NO.
SHEET=	identifies a particular worksheet in an Excel workbook.

To create the CSV file from SASHELP.CLASS using PROC EXPORT, you could write something like

```
proc export data=sashelp.class
            outfile='c:\pharmasug2015\class.csv'
            dbms=CSV
            replace;
run;
```

that would result in each value separated by a ',', or if we want our original file created in the data step above using the delimited '~' we could write something like

```
proc export data=sashelp.class
            outfile='c:\pharmasug2015\class'
            dbms=dlm;
            delimiter='~';
run;
```

For EXCEL files, using DBMS=XLSX, would be

```
proc export data=sashelp.class
            dbms=xlsx
            outfile='c:\pharmasug2015\class.xlsx'
            replace;
run;
```

One useful option when using PROC EXPORT is that it is possible to create separate sheets in the one

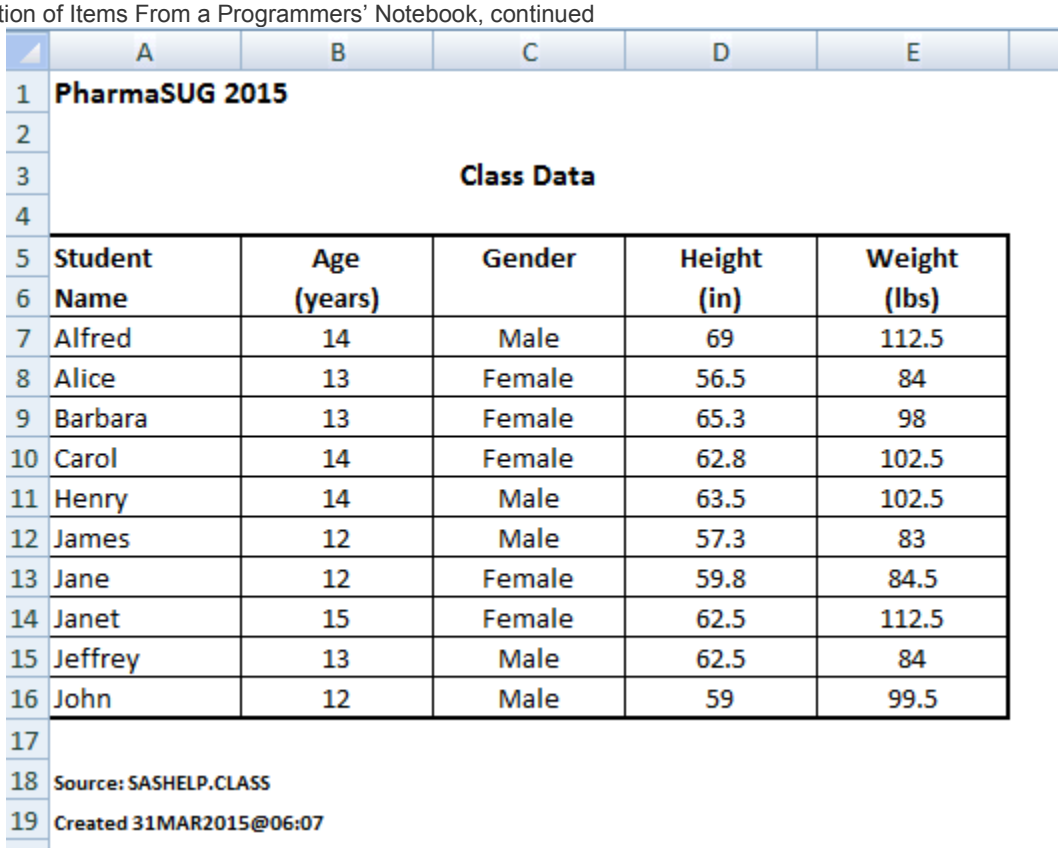
Workbook, as the following example shows where we create the sheets CLASS and BASEBALL in the Workbook StatsPaper.xlsx

```
proc export data=sashelp.class
            dbms=xlsx
            outfile='c:\pharmasug2015\StatsPaper.xlsx'
            replace;
   sheet='Class';
run;
proc export data=sashelp.baseball
            dbms=xlsx
            outfile='c:\pharmasug2015\StatsPaper.xlsx';
   sheet='Baseball';
run;
```

While the data step and PROC EXPORT methods are good, sometimes we need something more where a sheet that has titles and footnotes, this is where the use of the MSOFFICE2K in ODS, as the following code demonstrates:

```
proc format;
   value $gender
      'M'='Male'
      'F'='Female';
run;
ods escapechar='^';
ods msoffice2k file='c:\pharmasug2015\class.xls' style=minimal;
title1 j=l h=3 "PharmaSUG 2015";
title2 j=c h=3 "Class Data";
footnote1 j=l h=1 "Source: SASHELP.CLASS ^n Created &sysdate9.@&systime.";
proc report data=sashelp.class (obs=10) nowindows headline headskip split='|' missing;
   columns name age sex height weight;
   define name    /group order=internal 'Student|Name' style={cellwidth=1in just=l};
   define age     /display 'Age|(years)' style={cellwidth=1in just=c};
   define sex     /display 'Gender' format=$gender. style={cellwidth=1in just=c};
   define height  /display 'Height|(in)' format=8.1 style={cellwidth=1in just=c};
   define weight  /display 'Weight|(lbs)' format=8.1 style={cellwidth=1in just=c};
   quit;
run;
ods _all_ close;
ods listing;
run;
```

which results in the output given below:



PharmaSUG 2015

Class Data

Student Name	Age (years)	Gender	Height (in)	Weight (lbs)
Alfred	14	Male	69	112.5
Alice	13	Female	56.5	84
Barbara	13	Female	65.3	98
Carol	14	Female	62.8	102.5
Henry	14	Male	63.5	102.5
James	12	Male	57.3	83
Jane	12	Female	59.8	84.5
Janet	15	Female	62.5	112.5
Jeffrey	13	Male	62.5	84
John	12	Male	59	99.5

Source: SASHELP.CLASS

Created 31MAR2015@06:07

Notice that the title and footnote statements were used in the generation of the Excel file, but the HEIGHT and WEIGHT formats were not used. Another item to note is that the separate TITLE statements created a blank line in between the two titles, while the '^n' used in the footnote created the second section of text immediately on the next row.

There are other methods, including tagsets.excelxp, that have appeared in other forums, but this is beyond the scope of the paper.

CREATING SAS DATA SETS FROM EXCEL FILES

Before PROC IMPORT, the way to import data from an Excel file was to create a CSV version of the file and then import it using a data step. In a large number of cases PROC IMPORT has replaced that, and where you have the SAS/ACCESS Interface to PC Files is installed, it is possible to do it directly from the Excel sheet without need for conversion to CSV.

The basic syntax of the PROC IMPORT for either CSV or EXCEL files are

```
PROC IMPORT OUT=<libref.>SAS data set
            DATAFILE="filename"
            <DBMS=identifier>
            <REPLACE>;
```

where

<libref.>SAS data set	identifies the output SAS data set
DATAFILE=	specifies the complete path and filename or a fileref for the input PC file, spreadsheet, or delimited external file. A fileref is a SAS name that is associated with the physical location of a file. To assign a fileref, use the FILENAME statement.
DBMS=	These include CSV (comma separated files), DLM (delimited files where the character is specified), TAB (tab delimited files), and XLS/XLSX (Excel files)
REPLACE	overwrites an existing file. If you do not specify REPLACE, the IMPORT procedure does not overwrite an existing file.

Some useful options for different types of file are:

DATAROW=n	specifies the row number where the IMPORT procedure starts reading data. Default is 1 if

GETNAMES=NO, 2 if GETNAMES=YES.

DELIMITER='char'	specifies the delimiter, either a single character or hexadecimal value, that separates the columns of data.
GETNAMES=	specifies whether the call is to generate SAS variable names from the data values in the first row of the import file. Default is YES.
GUESSINGROWS=n	specifies the number of rows that the IMPORT procedure is to scan to determine the appropriate data type for the columns. Default is 20.
MIXED=	assigns a SAS character type for the column and converts all numeric data values to character data values when mixed data types are found. Default is NO.
RANGE=	subsets a worksheet by identifying the rectangular set of cells to import. This can be either a 'range-name' or 'absolute-range'
SHEET=	identifies a particular worksheet in an Excel workbook. Use SHEET= when you want to import the entire worksheet.
TEXTSIZE=	specifies the SAS maximum variable length that is allowed while importing data from Microsoft Excel text columns. Any TEXT data in Excel whose length exceeds this value is truncated when it is imported into SAS. Valid values are 1 to 32767.

In the section regarding Creating Excel Files From SAS Data Sets, we created a CSV file of SASHELP.CLASS, and to import it again into a SAS data set the following code could be used:

```
proc import datafile="c:\pharmasug2015\class.csv"
            dbms=csv
            out=class
            replace;
run;
```

To import a '~' separated file, the DELIMITER statement is used, as shown in the example below:

```
PROC IMPORT OUT=class
            DATAFILE="c:\pharmasug2015\class.txt"
            DBMS=DLM
            REPLACE;
   DELIMITER='~';
RUN;
```

To import an Excel file with a sheet called CLASS it is possible to use the code

```
proc import datafile="c:\pharmasug2015\prdsale.xlsx"
            dbms=xlsx
            out=work.prdsale
            replace;
   sheet="CLASS";
run;
```

The topic of importing an Excel file into a SAS data set is large and beyond the scope of this paper. Please refer to relevant documentation for specifics.

SPLIT THE INFO FROM &SYSINFO

The COMPARE procedure reports 16 possible discrepancies between two data sets. One of the outputs from Proc Compare is the code stored in the automatic macro variable &SYSINFO, this macro variable stores the 16 possible discrepancies. One of the advantages of the macro variable is that it is possible to store the value from the variable in a data set and report the information of more than one comparison in a single report. The purpose of this section is to show a method to split the possible values.

From the BASE SAS 9.4 Procedures guide, page 322 and 323, the 16 discrepancies reported by PROC COMPARE are:

Macro Return Codes

Bit	Condition	Code	Hex	Description
1	DSLABEL	1	0001X	Data set labels differ
2	DSTYPE	2	0002X	Data set types differ
3	INFORMAT	4	0004X	Variable has different informat
4	FORMAT	8	0008X	Variable has different format
5	LENGTH	16	0010X	Variable has different length
6	LABEL	32	0020X	Variable has different label
7	BASEOBS	64	0040X	Base data set has observation not in comparison
8	COMPOBS	128	0080X	Comparison data set has observation not in base
9	BASEBY	256	0100X	Base data set has BY group not in comparison
10	COMPBY	512	0200X	Comparison data set has BY group not in base
11	BASEVAR	1024	0400X	Base data set has variable not in comparison
12	COMPVAR	2048	0800X	Comparison data set has variable not in base
13	VALUE	4096	1000X	A value comparison was unequal
14	TYPE	8192	2000X	Conflicting variable types
15	BYVAR	16384	4000X	BY variables do not match
16	ERROR	32768	8000X	Fatal error: comparison not done

The table talks about 'bits' so the SYSINFO value is actually a string of 1's and 0's in decimal form so we have to change the value to a string, and where there is a '1' that issue exists in the data set. If the SYSINFO value is '0' then the data sets compare exactly.

The secret for actually decoding the return code is convert it to a binary format, and read it from right to left, and this can be done easily using the BINARY16. format. For example, there is a return code of 4097 which when run through a data step:

```
43   data _null_;
44      x=4097;
45      y=put(x,binary16.);
46   put x= y=;
47   run;
x=4097 y=0001000000000001
```

so, from this we can see the first byte on the extreme right is '1' so the labels of the data set differ, and a '1' at the 13th position (4th form the left) which means a value comparison is unequal.

Taking a look at at value 2049 we get a 'bit' equivalent of

```
x=2049 y=0000100000000001
```

so, from this we can see the first byte on the extreme left is '1' so the labels of the data set differ, and a '1' at the 12th position (5th form the left) which means the comparison data set has variable not in base.

In the examples we have only looked at two differences, but up-to 16 different messages can be retrieved.

Let's crank this up a notch, where we can use it practically, lets create a macro!

```
%macro compare_ds(
  baseds= /*Base Data set*/
 ,compareds= /*Compare Data set*/
 ,byord= /*Variable(s) to get same order (optional)*/
                );
```

```
    *Sort our two incoming data sets;
    %if (z&byord ne z) %then %do;
      proc sort data=&baseds;
        by &byord;
      proc sort data=&compareds;
        by &byord;
      run;
    %end;

    *Do compare;
    proc compare base=&baseds compare=&compareds;
    run;

    *Capture result;
    %let retcode=&sysinfo;

    *Put messages out to SAS LOG;
    data _null_;

     *Initialize;
     retain  sysinfoval &retcode;
     put "******************************************";
     put " PROC COMPARE SUMMARY RESULT";
     put "******************************************";
     put;
     put "Result of Comparison between data sets "
          "%upcase(&baseds) and %upcase(&compareds) "
         "are:";

     *Set up messages;
     array compmsg {16} $ 40 _temporary_ (
        "Data set labels differ",
        "Data set types differ",
       "Variable has different informat",
       "Variable has different format",
       "Variable has different length",
        "OLD data set has observation not in NEW",
        "NEW data set has observation not in OLD",
       "OLD data set has BY group not in NEW",
        "NEW data set has BY group not in OLD",
        "OLD data set has variable not in NEW",
        "NEW data set has variable not in OLD",
        "A value comparison was unequal",
       "Conflicting variable types",
        "BY variables do not match",
        "Fatal error: comparison not done"
        );

     *Put return code into bit code, reversing so
      first is on left;
     _bitcode=reverse(put(_sysinfoval,binary16.));

     *If return code is 0 then compare is identical;
     if index(_bitcode,'1')=0 then
       put @3 "Data sets are identical";

      *Compare not identical, put out messages;
      else do;
        put @3 "The following issue(s) are found:";
        do i=1 to 16;
            if substr(_bitcode,i,1)='1' then
         put @5 '-' compmsg(i);
        end;
      end;

        put;
        put @1 '***** END OF COMPARE *****';

      run;
   %mend compare_ds;
```

Now let's run this macro, using the SASHELP.CLASS data set and changing the data set label, adding a new variable and changing an existing variable:

```
data class (label='2015 Class Data');
   set sashelp.class;
   age=round(age+rand("Uniform"));
   ht_cm=height*2.54;
run;
```

then calling the macro in the following call:

```
%compare_ds(baseds=sashelp.class,compareds=class);
```

which creates the following in the SAS LOG:

```
*******************************************
 PROC COMPARE SUMMARY RESULT
*******************************************

Result of Comparison between data sets BASEDAT and COMPAREDAT are:
  The following issue(s) are found:
     -Data set labels differ
     -NEW data set has variable not in OLD
     -A value comparison was unequal

***** END OF COMPARE *****
```

This output is much more concise than trying to interpret pages of listing output.

The summary is in the SAS LOG but if you wanted to look at the individual issues you can look at the listing from the PROC COMPARE.

KEEP ONLY THE CHARACTER, NUMERIC, OR PRINTABLE CHARACTERS

The compress function not only removes spaces from character variables, it can keep or remove characters based in their keeps the kind of characters that you want. You can keep or remove only the kind of text that can be converted to an integer variable; you can keep only the text that cannot be converted to an integer variable or only the text that is printable, this is not a unique list of possible scenarios. This is an example of how you can keep only the characters that you need, remember that the first 32 of the 256 ASCII (American Standard Code for Information) characters are not printable.

The compress function takes up to 3 character parameters, the first one is the source, it can be a character variable or text, the second parameter has special characters that will be removed or kept, and the third parameter has the conditions of to keep or remove characters. If 'K' is indicated, the COMPRESS function will keep characters indicated, if there is not 'K', the compress function will remove the desired characters.

```
data forcomp(drop=i);
  length text $50;
  INFILE DATALINES MISSOVER LENGTH=LEN;
  i=_n_;
  nonprint=BYTE(i);
  printable=BYTE(i+180);
  INPUT text $varying200. len;
     onlyalphaNspace=  compress(text,       ' ','ka'); ①
     onlyalphaNospace= compress(text,         ,'ka'); ②
     noletters=       compress(text, ' ','a');        ③
     onlynumbersNdot=  compress(text,       '.','kd'); ④
     onlyalphaNnumbers=compress(text,        ,'kn'); ⑤
     printablef=      compress(nonprint,    ,'kw');   ⑥
     printablef2=     compress(printable,   ,'kw');   ⑦
     printablet=      compress(text,    ,'kw');       ⑧
 datalines;
the number in the house is 58,
the house is located in mile 39.5 of I-40 NC,
the telephone number is 123=1234-1234.
;;;;
run;
```

①The example is keeping the alpha characters and the spaces, the third parameter includes a 'k', so the reference is

about keeping characters, the third parameter also includes an 'a', so alpha characters will be kept. Because there is a space between the quotes of the second parameter, spaces will be kept.

② The example is similar to the previous one, but the is no second parameter, so spaces are not kept.

	text	onlyalphaNspace	onlyalphaNospace
1	the number in the house is 58,	the number in the house is	thenumberinthehouseis
2	the house is located in mile 39.5...	the house is located in mile of I NC	thehouseislocatedinmileofINC
3	the telephone number is 123=12...	the telephone number is	thetelephonenumberis

③ The example removes any alpha character.

④ The example KEEPS decimals and because the second parameter has a dot, the decimal place is kept. This is the case you would use to convert the converted variable to an integer variable.

	text	noletters	onlynumbersNdot
1	the number in the house is 58,	58,	58
2	the house is located in mile 39.5...	39.5-40,	39.540
3	the telephone number is 123=12...	123=1234-1234.	12312341234.

⑤ This example keeps alpha and digits, other symbols are not kept.

	text	onlyalphaNnumbers
1	the number in the house is 58,	thenumberinthehouseis58
2	the house is located in mile 39.5...	thehouseislocatedinmile395ofI40NC
3	the telephone number is 123=12...	thetelephonenumberis12312341234

⑥⑦⑧ In these 3 examples show what is left when Non-printable characters are removed, this is useful when importing data. The values of nonprintable exist, they don't just don't print. The values of printablef, would be empty if it weren't for the quote function.

	nonprint	printable	printablef	printablef2	printablet
1		µ	""	µ	the number in the house is 58,
2		¶	""	¶	the house is located in mile 39.5 of I-40 NC,
3		·	""	·	the telephone number is 123=1234-1234.

CREATING EMPTY DATA SETS

One way to create empty data sets is using the call missing routine, that assigns missing values to the numeric and character variables in the argument list. If more than one variable is used as an argument, a comma should separate the variables.

```
DATA comparecase;
  LENGTH table $8;
  FORMAT cdate datetime15.;
  CALL MISSING(cdate,code,table);
RUN;
```

```
NOTE: The data set WORK.COMPARECASE has 1 observations and 3 variables.
```

Notice that there were no errors or notes in the log about uninitialized variables, the only problem is that a row was created and we want an empty data set, so we will add delete;

```
DATA comparecase;
  LENGTH table $8;
  CALL MISSING(cdate,code,table);
  DELETE;
RUN;
```

```
NOTE: The data set WORK.COMPARECASE has 0 observations and 3 variables.
```

and reviewing what data we have, the tool to use is PROC CONTENTS as shown below:

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

with the output of

```
Variables in Creation Order

#    Variable    Type       Len     Format

1   table       Char        8
2   cdate       Num         8       DATETIME15.
3   code        Num         8
```

When creating an empty data set, it is important to set the type of a variable at the beginning of the data set the length, to establish if the variable is character or numeric, if not all the attributes of character variables, should be set before the missing routine is called, otherwise, the default type for a variable is n SAS is numeric. The Programming data vector has the variables specified in the attributes section first, then the variables are created in the order of the call routine. Here is an example where the variable table was defined as character after the call missing routine was called.

```
16          DATA comparecase;
17          FORMAT cdate DATETIME15.;
18          CALL MISSING(cdate,code,table);
19          LENGTH table $8;
ERROR: Character length cannot be used with numeric variable table.
20          DELETE;
21          RUN;
NOTE: The SAS System stopped processing this step because of errors.
```

One way to create empty data sets is using the call missing routine, that assigns missing values to the numeric and character variables in the argument list. If more than one variable is used as an argument, a comma should separate the variables.

```
DATA comparecase;
  LENGTH table $8;
  FORMAT cdate datetime15.;
  CALL MISSING(cdate,code,table);
RUN;

NOTE: The data set WORK.COMPARECASE has 1 observations and 3 variables.
```

Notice that there were no errors or notes in the log about uninitialized variables, the only problem is that a row was created and we want an empty data set, so we will add delete;

```
DATA comparecase;
  LENGTH table $8;
  CALL MISSING(cdate,code,table);
  DELETE;
RUN;

NOTE: The data set WORK.COMPARECASE has 0 observations and 3 variables.
```

and again using PROC CONTENTS we have

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

with the output of

```
Variables in Creation Order

#    Variable    Type       Len     Format

1   table       Char        8
2   cdate       Num         8       DATETIME15.
3   code        Num         8
```

When creating an empty data set, the length, if not all the attributes of character variables should be set before the missing routine is called, otherwise, SAS will think that the variables in the argument are numeric. The Programming data vector has the variables specified in the attributes section first, then the variables are created in the order of the call routine. Here is an example where the variable table was defined as character after the call missing routine was called.

```
16          data x;
17          format cdate datetime15.;
18          call missing(cdate,code,table);
19          length table $8;
ERROR: Character length cannot be used with numeric variable table.
20          delete;
21          run;

NOTE: The SAS System stopped processing this step because of errors.
```

CALCULATE THE BEGINNING, MIDDLE OR END OF THE MONTH

Dates of events are things we handle every day, but they are not always complete dates -- some will have day missing, some will have day and month missing, and there will be some cases where the date is completely missing. Then rules start to be applied, usually standards applied in-house. The first of the month, first of the year, date of first treatment, are easy to apply, but what if you needed the middle or the end of the month.

In a number of older papers, you will see macros that do such things as find out what month it is, add one to it, get the date of the first of that month, and then subtract one day, all to find the last day of the month. All of this has been, thankfully, replaced with the INTNX fuction, the syntax of which is shown below in one of its forms:

Return_Date=INTNX(*interval*,*date*,*increment*,*alignment*);

where

interval	this is usually values such as WEEK (each Sunday), SEMIMONTH (the first and sixteenth of each month), MONTH (the first of each month), QTR (the first of January, April, July, and October), SEMIYEAR (the first of January and July) and YEAR (the first of January)
date	SAS date, time, or datetime as a starting reference point
increment	number of intervals to shift the value of date
alignment	optional parameter aligns the beginning, middle, end or same of the period ('B', 'M', 'E', 'S"), if the fourth parameter is not used, 'B' is implied.

```
DATA missday;
  LENGTH partial $20;
  FORMAT last middle same nofourth IS8601DA.;
  INFILE DATALINES MISSOVER LENGTH=LEN;

  INPUT partial $;
  IF LENGTHN(partial)=7       THEN  DO;
          last=INTNX('MONTH',  ①
          INPUT(STRIP(partial)||'-01',IS8601DA.), , ②
          0,     ③
          'E'); ④
    middle=INTNX('MONTH',INPUT(STRIP(partial)||'-01',IS8601DA.),0,'M');
    same  =INTNX('MONTH',INPUT(STRIP(partial)||'-01',IS8601DA.),0,'S');
    nofourth=INTNX('MONTH',INPUT(STRIP(partial)||'-01',IS8601DA.),0);
  END;
  ELSE IF LENGTHN(partial)>=10  THEN  DO;
         last=INPUT(SUBSTR(partial,1,10),IS8601DA.);** No change;
         middle=INTNX('MONTH',INPUT(SUBSTR(partial,1,10), IS8601DA.),0,'E');
         middle=INTNX('MONTH',INPUT(SUBSTR(partial,1,10), IS8601DA.),0,'M');
         nofourth=INTNX('MONTH',INPUT(SUBSTR(partial,1,10), IS8601DA.),0,'S');
  END;
CARDS4;
  2014-02
  2012-02
  2000-02
  1900-02
  2015-10
  2015-06-15
```

① The interval or unit of time that is going to be added.
② A numeric date variable to which you will add an interval.
③ Number of times that you are going to add the interval, in this example is zero, but it can be a negative or positive or negative number.
④ Alignment parameter.

```
;;;;
run;
```

The resulting data set

	partial	last	middle	same	nofourth
1	2014-02	2014-02-28	2014-02-14	2014-02-01	2014-02-01
2	2012-02	2012-02-29	2012-02-15	2012-02-01	2012-02-01
3	2000-02	2000-02-29	2000-02-15	2000-02-01	2000-02-01
4	1900-02	1900-02-28	1900-02-14	1900-02-01	1900-02-01
5	2015-10	2015-10-31	2015-10-16	2015-10-01	2015-10-01
6	2015-06-13	2015-06-13	2015-06-15	2015-06-13	2015-06-01

The character variable PARTIAL has the original value. The numeric variable LAST has the imputed last date of that month. The numeric variable MIDDLE has the imputed middle date of the month. The variable SAME, has the NUMERIC value of the numeric date, because I chose to add '-01' the months dates where day was missing are displaying 01 as the day of the month, but the last observation with a day 13, has the 13 as the day of the month, the variable NOFOURTH is the variable where the first date of the month is imputed, notice that in the observation when day was not missing, the 13 is changed for 01.

Typically we would find the beginning of a month, where we have a given numeric date but we are told that the precision is only to month and year value, by using a call similar to

```
Return_Date=INTNX('MONTH',date,0,'BEGINNING');
```

or find the middle of the month by

```
Return_Date=INTNX('MONTH',date,0,'MIDDLE');
```

of the end of the month by

```
Return_Date=INTNX('MONTH',date,0,'END');
```

The examples here have been using date values, but the INTNX funtion will work equally well with datetime or time values.

CONCLUSION

Presented in this paper are ten frequently referenced items of interest for the SAS Programmer from the notebooks of two programmers. We hope that you find

REFERENCES

http://support.sas.com/resources/papers/proceedings12/063-2012.pdf SAS Institute

Base SAS 9.4 Procedures Guide, Second Edition.

http://support.sas.com/kb/45/011.html

CONTACT INFORMATION

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

Name: David Franklin
Enterprise: Real-World & Late Phase Research, Quintiles
Email: david.franklin@Quintiles.com

Name: Cecilia Mauldin
Enterprise: Independent Consultant
Email: mamadenina@hotmail.com