Paper IT-03

Quick Hits - My Favorite SAS® Tricks

Marje Fecht, Prowerk Consulting, Canada and USA

ABSTRACT

Are you time-poor and code-heavy?

It's easy to get into a rut with your SAS code and it can be time-consuming to spend your time learning and implementing improved techniques.

This presentation is designed to share quick improvements that take 5 minutes to learn and about the same time to implement. The quick hits are applicable across versions of SAS and require only BASE SAS knowledge.

Included are:
- simple macro tricks
- little known functions that get rid of messy coding
- dynamic conditional logic
- data summarization tips to reduce data and processing
- generation data sets to improve data access and rollback.

INTRODUCTION

Many SAS programmers learn SAS "by example" - using the programs they have inherited or a co-worker's examples. ***Some examples are better than others!*** Sometimes, the examples we learn from reflect coding practices from earlier SAS versions, prior to the introduction of the large function library and language extensions that exist today.

When you are tasked with producing results, there isn't always time to learn and implement new features. This paper shares quick tricks that are easy to learn and implement.

CODE REDUCTION

The SAS function library grows with each release of SAS and provides built-in functionality for accomplishing many common tasks. To help you relate the described functionality to your existing code, "before and after" examples are included.

CONCATENATION

If you join strings of data together, then you have likely used the concatenation operator (||) as well as other operators to obtain the desired results. Concatenation and managing delimiters and blanks can be frustrating and may involve a lot of steps including:

- `TRIM`
- `LEFT`
- `STRIP`
- `||`
- Adding delimiters.

The ***old way*** of joining the contents of the variables n (numeric), with a, b, c (all character) might look like:

```
old =   put(n,1.)  || ' ' || trim(a) || ' ' || trim(b) || ' ' || c;
```

Unfortunately the above code won't always produce a pleasing result, and thus could require even more complication. Consider the case when the variable **a** contains all blanks. This would result in 3 blanks in a row, in the variable **`old`** since **`trim(a)`** would reduce to a single blank and then add the additional blanks used as delimiters. Furthermore, a `LEFT` or `STRIP` function would be needed to insure that leading blanks don't cause issues.

The SAS 9 family of `CAT` functions reduces complexity when concatenating strings!

- `CAT` concatenates multiple strings in one function call
- `CATT` - same as `CAT` but also `TRIM`s
- `CATS` - same as `CAT` but also `STRIP`s leading and trailing blanks
- `CATX` - same as `CATS` but you can specify a delimiter.

Using CATX, the above example would be reduced to:

```
new = CATX ( ' ',  n , a, b, c);
```

Note:

- If any of n, a, b, c are ***blank***, CATX will not include the blanks and is smart enough to therefore not include the delimiters.
- If any arguments are numeric, the CAT functions will handle the numeric to character conversion without LOG messages.

CONDITIONAL CONCATENATION

Consider an example where you have a series of indicators that represent marketing channels used in a campaign. You want to build a single string (see channels_CATX) showing all channels used for each client. For this example, we start with 4 indicators in the data set.

ch_dm	ch_on	ch_cc	ch_st	***Desired Result*** channels_CATX
Y	Y	Y	Y	DM_ON_CC_ST
N	Y	Y	N	ON_CC
Y	N	N	Y	DM_ST
N	Y	Y	Y	ON_CC_ST

Example 1 - Solution 1:

A classic solution would be to create a variable that corresponds to each indicator with either a blank or the two letter code that is desired for the result Then concatenate the 4 new variables.

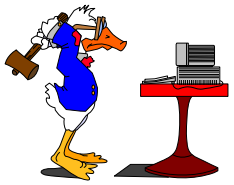
```
  /*** Create text field for each channel ***/
if ch_dm = 'Y' then dm = 'DM';
if ch_on = 'Y' then on = 'ON';
if ch_cc = 'Y' then cc = 'CC';
if ch_st = 'Y' then st = 'ST';

chann_1 = dm || '_' || on || '_' || cc || '_' || st;
```

This solution is

- wordy since it requires creating an extra set of variables
- problematic
  - extra underscores in result since delimiters are included regardless of "missing data"
  - TRIM or STRIP could be needed if variables had differing length strings.

ch_dm	ch_on	ch_cc	ch_st	chann_1
Y	Y	Y	Y	DM_ON_CC_ST
N	Y	Y	N	_ON_CC_
Y	N	N	Y	DM_ _ _ST
N	Y	Y	Y	_ON_CC_ST



Example 1 - Solution 2:
Improve upon Solution 1 by using the CATX function. A LENGTH statement is needed since CATX returns a string of length 200.

```
/*** Using CATX – ignores missing values and handles STRIP ***/
 length channels_CATX $12;
 channels_CATX = CATX ( '_' , dm , on , cc , st);
```

ch_dm	ch_on	ch_cc	ch_st	chann_1	channels_CATX
Y	Y	Y	Y	DM_ON_CC_ST	DM_ON_CC_ST
N	Y	Y	N	_ON_CC_	ON_CC
Y	N	N	Y	DM_ _ _ST	DM_ST
N	Y	Y	Y	_ON_CC_ST	ON_CC_ST

Solution 2 still uses the extra 4 variables but it behaves when some values are missing, so you don't need COMPRESS or special logic.

CONDITIONAL ASSIGNMENT OF VALUES - SIMPLE
In the above concatenation example, fields were created conditional on the values for a set of indicators. Conditional assignments are often accomplished with if-then-else logic (or SQL CASE-WHEN coding). You can consolidate sets of if-then-else code with a single function call using the IFC or IFN functions.

IFC (**IFN**) returns a character (numeric) value based on whether an expression is

- true *result of expression NOT 0 or . (missing)*
- false *result of expression is 0*
- missing *result of expression is . (missing)*

IFC (**IFN**) is coded as

```
IFC (expression-to-evaluate , true-result , false-result , missing-result)
```

Consider the statements

```
if ch_dm = 'Y' then dm = 'DM';
else dm = ' ';
```

This logic can be rewritten as

```
dm = IFC( ch_dm = 'Y' , 'DM' , ' ');
```

Note that

- the expression ch_dm = 'Y' can ONLY result in TRUE or FALSE and thus two *result* arguments are provided
- the result 'DM' is returned only when ch_dm = 'Y'
- any value of ch_dm other than 'Y' results in a blank since the expression is FALSE.

Example 1 - Solution 3:
Improve upon Concatenation Solution 2 by removing the need to create the extra 4 variables. Note that as with most function calls in SAS, `IFC` can easily be embedded within `CATX`.

```
/*** NO NEED TO CREATE EXTRA TEXT VARIABLES ***/
 length channels_IFC_CATX $12;
 channels_IFC_CATX = CATX(
                     '_'      /* note delimeter of underscore */
                   , IFC( ch_dm = 'Y' , 'DM' , ' ')
                   , IFC( ch_on = 'Y' , 'ON' , ' ')
                   , IFC( ch_cc = 'Y' , 'CC' , ' ')
                   , IFC( ch_st = 'Y' , 'ST' , ' ')
                         ) ;
```

ch_dm	ch_on	ch_cc	ch_st	channels_IFC_CATX
Y	Y	Y	Y	DM_ON_CC_ST
N	Y	Y	N	ON_CC
Y	N	N	Y	DM_ST
N	Y	Y	Y	ON_CC_ST

CONDITIONAL ASSIGNMENT OF VALUES – MORE COMPLEX
Conditionally assigning values with `IFC` / `IFN` is powerful but you need to be careful with how the function works.

Example 2: Assign course status based on course grade.
Using the value of grade, assign a status of **Pass** for grades of at least 70, **Fail** for grades that are less than 70 but not missing, and **Pending** for grades that are still outstanding (missing).

Example 2 - Solution 1

```
If grade ge 70 then status = "Pass";
  else if grade = . then status = "Pending";
     else status = "Fail";
```

This coding works fine but is wordier than necessary.

Example 2 - Solution 2
Recall that **`IFC`** returns a character value based on whether an expression is true (**not 0 or .**), false (**0**), or missing (**.**).

```
Length Status $7;     /*IFC defaults the result to length 200*/
Status = IFC( grade ge 70 ,
                "Pass" ,   /* TRUE    */
                "Fail" ,   /* FALSE   */
                "Pending"  /* Missing */
            );
```

The previous example will only result in Pass and Fail, since the logical expression will only result in True (1) or False(0).
BEWARE!! It is helpful to note that when SAS evaluates an expression, a value of ZERO denotes FALSE and any other numeric non-missing values other than zero denotes TRUE.

Example 2 - Solution 3

Correct the above logic by using a mechanism to insure that a missing value of `grade` will result in a missing value for the expression. A solution I commonly employ is to multiply the *logical expression* (`grade ge 70`) by the *variable being evaluated* (`grade`).

For example, for

```
grade * (grade ge 70)
```

the expression is

- TRUE when `grade ge 70`
  since `grade * (grade ge 70) = grade * 1 = grade` ➔ `TRUE`
- FALSE when `grade lt 70 and grade NE .`
  since `grade * (grade ge 70) = grade * 0 = 0` ➔ `FALSE`
- MISSING when `grade = .`
  since `grade * (grade ge 70) = . * 0 = .` ➔ `MISSING.`

The corrected logic is:

```
Length Status $7;
Status = IFC( grade * (grade ge 70) ,
              "Pass" ,  /* TRUE */
              "Fail" ,  /* FALSE = 0 */
              "Pending"  /* Missing */
            );
```

MORE FUNCIONS FOR YOUR TOOLKIT

Many SAS programmers write beautifully coded logic to handle functionality that already exists in the SAS Function Library. Before you embark down that path, review the SAS online documentation to see what functions are out there.

A few more of my favorite (and more obscure) functions include:

- `TRIMN` ➔ removes trailing blanks – returns null for values that are all blank
- `COUNT`, `COUNTC` ➔ counts # of occurrences of a string or Character
- `INDEX`, `INDEXC`, `INDEXW` ➔ locates position of first occurrence of string, Character, or Word
- `LENGTH` (min=1) , `LENGTHN` (min=0) ➔ position of last non-blank. `LENGTHN` returns 0 for values that are all blank
- `LARGEST` (*k , var1, var2, …)* ➔ kth largest non-missing value
- `SMALLEST` (*k , var1, var2, …)* ➔ kth smallest non-missing value
- `PROPCASE` ➔ handles upper / lower case to assist with Proper Names and Addresses, etc.

CODE GENERALIZATION

If you suffer from copy-paste syndrome or if every request seems to require a "from scratch" build, then you need to work on changing your approach!! Before you begin writing a program, think about ***what could change about this request in the future?*** If the request is *time-specific* or if it focuses on just a *subset of the available population*, chances are good that you can generalize the code so that it can be easily adapted to future requests. Think also about ***what have I written before that could be leveraged now***?

BEST PRACTICES: REUSABLE CODE

According to wikipedia,

> *In computer science and software engineering,* **reusability** *is the likelihood that a segment of source code can be used again to add new functionalities with slight or no modification.*

Reusable code modules

- reduce implementation time
- increase the likelihood that prior testing and use has eliminated bugs
- localizes code modifications when a change in implementation is required.

My programs commonly include a comment line of

```
/***** NO CHANGES BELOW HERE *****/
```

since I pass parameters to my source code and use generalized coding practices with

- User Defined Macro variables for "static" information
- Data-Driven values via metadata or functions for "dynamic" information
- Generalized location / naming / etc to enable easy changes
- File names and locations that are parameterized
- System locations, options, settings that are generalized and parameterized.

My source code remains "un-touched", unless upgrades are implemented, which then roll-out to all programs that call the source code.

CODE GENERALIZATION: LOGS AND OUTPUT

To generalize file locations and names, such as logs and output files, a typical example would

- Use macro variables to supply path and project ID information
- Specify the stage of development (Development, Test, Production)
- Use macro and SAS functions to determine the date and time and then format them for the file naming
  - Note: the N in yymmddN8. requests NO separators (- , / , etc) so that just an 8 digit string is produced
  - Note: the compress removes the **:** from the time value
- PROC PRINTTO is used to route the log so that it remains available for perusal

```
%let requestID = PROJECT974;
%let filepath  = mygroup\myprojectfiles ;
%let stage = prod;         ** could be devl, test, prod **;

... <other program logic – following parameter def'n> ...

%let dir = \&filepath.\&requestID.;
%let filename = &requestID._extract1;
%let datetime = %sysfunc(compress(%sysfunc(today(),yymmddN8.)_%sysfunc(time(),hhmm6.)
                                  , ': '));
** route Log and Listing to permanent location **;
proc printto
       log ="&dir.\logs\&stage.\&filename._&datetime..log"
       print="&dir.\output\&stage.\&filename._&datetime..lst";
run;
... < other program logic > ...
** reroute to default locations **;
proc printto; run;
```

The resulting "Versioned" file names would be

- PROJECT974_*20120507_1329.log*
- PROJECT974_*20120507_1329.lst*

CODE MODULARIZATION: SOURCE CODE AND DRIVER PROGRAMS

Once you have generalized your code to make it readily available for re-use, you need a mechanism for separating the source modules from *request-specific* input. My approach is to utilize ***driver*** programs that contain all of the parameters and other input, and that call the appropriate ***source*** modules.

```
*** Driver Program – specify parameters;
%let prestart  = 01SEP2011;
%let prestop   = 31DEC2011;
%let poststart = 01JAN2012;
%let poststop  = 30JUN2012;
%let title = "January 2012 Introduction of Claims Changes";
%let plans = ('78','7X','7R','55','85','18');
%let codes = ('015', '119', '214');

/*** NO CHANGES below here ***/
*** run standard extract and reporting programs;
%include 'ClaimsReportExtract_20110310.sas';
%include 'ClaimsReportOutput_20110113.sas';
```

One problem occurs with the above approach…
When the source module changes, you have to find ALL the drivers that call it to change the version date specified in the `%include`. This could involve searching 100's of drivers.

Solution:
Create a program that calls the latest version of the source, and call that "control program" from your drivers.

Contents of Control Program: `ClaimsReportExtract_CurrentPgm.sas`

```
*** call latest version of source program;
%include 'ClaimsReportExtract_20110310.sas';
```

Now, when source changes, your drivers always call the most current version.

```
*** Driver Program – specify correct parameters;
%let prestart  = 01SEP2011;
%let prestop   = 31DEC2011;
%let poststart = 01JAN2012;
%let poststop  = 30JUN2012;
%let title = "January 2012 Introduction of Claims Changes";
%let plans = ('78','7X','7R','55','85','18');
%let codes = ('015', '119', '214');
*** run standard reporting programs;
%include 'ClaimsReportExtract_CurrentPgm.sas';
%include 'ClaimsReportOutput_CurrentPgm.sas';
```

Helpful Hint:

- There is no limit to the # of `%include` statements you can use
- If you want the code from a `%include` to display in your log, use the **SOURCE2** option
- In addition to `%include`, reusable modules may be
  - Macros
  - Format Libraries.

CODE GENERALIZATION : DYNAMIC CODE

Reporting and analytics requests frequently revolve around lists of dates, campaigns, products, departments, etc. Thus, generalizing your code to dynamically create and accept LISTS is worth the effort. If a list can be programmatically generated, you avoid manual input and thus the introduction of typos and errors.

Example: Extract and summarize data for all marketing within a given date range and focus.

Solution 1: Create a ***generalized program*** that expects a list of all campaign codes for the date range with the focus of interest. Manually input the list of campaign codes within the dates of interest.

```
%let codes =
   2010307ABC
  ,2010337ABC
  ,2011003ABC
 ... Etc ...
;
```

Problem: Someone has to manually locate the campaign codes of interest and correctly input them for the program.

CREATE DYNAMIC LISTS

If the campaign codes follow a pattern or if additional metadata are available to assist you in generating the list, use that information programmatically.

Solution 2: Use SQL to build a macro variable (`codes`) that contains a *comma-delimited* list of all of the campaign codes

- with campaign "drop" between a specified begin and end date
- that end in 'ABC' since that identifies the campaign focus.

```
proc sql noprint;
  /** include database connection, if needed  **/
 select distinct cmpgn_code
          into :codes separated by ','
 from mktg_metadata
 where
      drop_dt between %str(%')&start%str(%')
              and     %str(%')&end%str(%')
       and substr(cmpgn_code , 8 , 3) = 'ABC'
;
%let NumCodes = &SQLOBS;     /** # of rows returned from SQL query **/
quit;
%put NumCodes = &NumCodes;
%put Campaign Codes    = &codes;
```

The resulting comma delimited list can be used

- as an IN list for subsetting
- as input to a macro loop for processing the data
- for variable and data set naming.

DYNAMIC VARIABLE NAMING

Suppose you need to summarize the latest 3 months of data, and you want monthly variables representing the totals for each month of data.

You want the variable names to reflect the month, such as

- TotAmt_**1205** represents Current Month (May2012)
- TotAmt_**1204** represents one month ago (Apr2012)
- TotAmt_**1203** represents two months ago (Mar2012)

You plan to use SQL with CASE to create the monthly amounts, and you don't want to manually intervene with the program. Instead, you want the program to determine the current month and generate the latest three months of data and names.

The SQL clause to bucket the monthly data might look like:

```
sum(case when txn_date between "&M1_beg"d and "&M1_end"d
         then txn_amt
        else 0
         end ) as Tot_Amt_&M1
```

You need macro variables for:

- Beginning Date of each month, in SAS Date format (ddMMMyy) – **M1_beg**
- Ending Date of each Month , in SAS Date format (ddMMMyy) – **M1_end**
- YYMM for each month (as a suffix for the variable names) – **M1**

INTNX – move in intervals

To dynamically generate the dates above, you need the ability to determine the current date and then move in increments of months back from today. Additionally, you need to be able to identify the first and last day of the month (without worrying about the nuances of the calendar). The INTNX function is one of the most versatile of the SAS date functions, enabling you to move forward and backward from a date and time using the interval of your choice, such as month, quarter, day, week, etc.

The **INTNX** function increments dates by intervals:

INTNX (*interval, from, n < , alignment* >) ;

- ***interval*** - interval name eg: 'MONTH', 'DAY', 'YEAR'
- ***from*** - a SAS date value (for date intervals) or datetime value (for datetime intervals)
- ***n*** - number of intervals to increment from the *from* value
- ***alignment*** - alignment of resulting SAS date, within the interval. Eg: **B**EGINNING, **M**IDDLE, **E**ND.

Example: Create 3 macro variables that contain the current and two previous months in the format: **yymm**

```
%let M0 = %sysfunc( today() , yymmN4.);

%let M1 = %sysfunc( intnx( MONTH ,
                    %sysfunc( today() ) ,
                    -1) , yymmN4.);          /** go back one month from today **/

%let M2 = %sysfunc( intnx( MONTH ,
                    %sysfunc( today() ) ,
                    -2) , yymmN4.);          /** go back two months from today **/
```

Note: when INTNX is used in %sysfunc, do not use quotes for the arguments of INTNX.

The above code produces three macro variables with values such as

- M0 = 1205
- M1 = 1204
- M2 = 1203

Example: Create 2 macro variables per month that contain the 1st and last day of the month, in SAS date format.

```
%let M2_beg = %sysfunc( intnx( MONTH
                , %sysfunc( today() ) , -2 , B)  /** return Beginning of month **/
                , date9.);
%let M2_end = %sysfunc( intnx( MONTH
                  , %sysfunc( today() ) , -2 , E)  /** return End of month **/
                  , date9.);
```

Note: Use whatever date format is appropriate for your data. For the example data, a SAS Date value is used.

If you are inclined to copy-paste and you expand this for the three months requested, the below code creates 3 macro variables per month to create the variable suffix (yymm) and the beginning and end date ranges for each month. This works but can obviously be improved upon.

```
%let M0 = %sysfunc( today() , yymmN4.);
%let M0_beg = %sysfunc( intnx( MONTH , %sysfunc( today() ) , 0 , B)
                        , date9.);
%let M0_end = %sysfunc( intnx( MONTH , %sysfunc( today() ) , 0 , E)
                        , date9.);

%let M1 = %sysfunc( intnx( MONTH     , %sysfunc( today() ) , -1)
                        , yymmN4.);
%let M1_beg = %sysfunc( intnx( MONTH , %sysfunc( today() ) , -1 , B)
                        , date9.);
%let M1_end = %sysfunc( intnx( MONTH , %sysfunc( today() ) , -1 , E)
                        , date9.);

%let M2 = %sysfunc( intnx( MONTH     , %sysfunc( today() ) , -2)
                        , yymmN4.);
%let M2_beg = %sysfunc( intnx( MONTH , %sysfunc( today() ) , -2 , B)
                        , date9.);
%let M2_end = %sysfunc( intnx( MONTH , %sysfunc( today() ) , -2 , E)
                        , date9.);
```

The 3 macro variables for each of the 3 months could be used to compute the monthly totals in code such as:

```
select
   sum(case when txn_date between
            "&M0_beg"d and "&M0_end"d
          then txn_amt else 0 end ) as Tot_Amt_&m0

  ,sum(case when txn_date between
            "&M1_beg"d and "&M1_end"d
          then txn_amt else 0 end ) as Tot_Amt_&m1

  ,sum(case when txn_date between
            "&M2_beg"d and "&M2_end"d
          then txn_amt else 0 end ) as Tot_Amt_&m2

 from amts
 where txn_date between "&M2_beg"d and "&M0_end"d
```

Helpful Hint:
The previous code assumed that SAS Date Values are needed (ddMMMyy).
Suppose your query is for a database table with

- dates stored as ***yyyy-mm-dd***
- single quotes are needed to surround date values (not **"**).

Change

- format of dates in macro variables ➔ `yymmddD10.`
- change the double quotes to single quotes: `%str(%')&M0 Beg%str(%')`

DYNAMIC CODE GENERATION

The above examples do provide ***dynamic code*** and it DOES work. But, what if we now want 12 months of results?

COPY – PASTE syndrome. . .

Notice that the code could easily be generated in a macro loop as long as MaxMonth (the maximum number of monthly computations) is defined.

Developing the appropriate code requires planning and understanding of the desired outcome.

What **Macro** variables and **SAS** variables are needed?

Macro Variable Name	Value	SAS Variable Name
M0	1205	Tot_Amt_1205
M0_Beg	1May2012	
M0_End	31May2012	
M1	1204	Tot_Amt_1204
M1_Beg	1Apr2012	
M1_End	30Apr2012	
. . .	. . .	
M11	1106	Tot_Amt_1106
M11_Beg	01Jun2011	
M11_End	30Jun2011	

Example: Create a macro to generate the Macro Variable Names and Values, using as input only today's date and the number of months desired (`MaxMonth`).

```
%let MaxMonth=11;   /** Months start at ZERO **/

%macro Monthly;
  %do Num = 0 %to &MaxMonth;
    /*** use %global if macro variables needed outside macro ***/
    %global M&Num. M&Num._beg M&Num._end;

         /** create suffix for variable names **/
    %let M&Num. = %sysfunc( intnx( MONTH ,%sysfunc( today() )
                        , -&Num.) , yymmN4.);
         /** create the date corresponding to beginning of month **/
    %let M&Num._beg = %sysfunc( intnx( MONTH , %sysfunc( today() )
                        , -&Num. , B) , date9.);

         /** create the date corresponding to end of month **/
    %let M&Num._end = %sysfunc( intnx( MONTH , %sysfunc( today() )
                        , -&Num. , E) , date9.);

  %end;

  /*** use a similar approach with SQL CASE Statements ***/
%mend Monthly;

%Monthly
%put _user_;    /** write all user defined macro variables and values to the LOG **/
```

The above example dynamically generates the macro variables needed for the extensible solution. In a similar macro %DO loop, the SQL to create the case statements could be accomplished.

SPACE MANAGEMENT - BEST PRACTICES

REUSE SPACE !!

Your use (or abuse) of SAS Work Space could crash other jobs (including yours)! If you are processing large data files, ***please*** delete intermediate data sets when they are no longer needed. For example, once joins are complete with other data, delete the intermediate data sets that comprised the join.

PROC DATASETS is handy to manage your SAS datasets including deleting files, modifying attributes, changing names, etc.

```
proc datasets lib = work
              memtype = data  details;
  /** delete old data **/
  delete ChqReport_1a
         ChqReport_1b
         ChqReport_1c ;
quit;
```

Note: Before deleting intermediate data, you should confirm there are no error conditions, etc.

MINIMIZE THE AMOUNT OF DATA YOU READ

You do not have to read data to change many data set attributes. Many SAS programmers rely on the DATA step to handle all variable attributes (labels, formats, renaming, etc). However, the DATA step reads every record which is problematic if your data include millions of records. Instead, the following PROC DATASETS example

- changes a data set name
- changes variables names
- assigns a variable format.

No data are read!

```
proc datasets lib = project memtype = data  details;
  /** rename dataset **/
  change ChqReport = ChqReport_&lastMonth ;
  /** change variable attributes in ChqExtract **/
  modify ChqExtract;
      rename txns = Transactions
             Date = Txn_Date;
       format TotalAmt Dollar12.;
quit;
```

What about SAS Data sets?

When creating permanent SAS data sets, a date-time stamp in the name may be problematic for processing from other programs and processes (eg: Excel Pivots). Instead, consider generation data sets.

Each data set in a SAS generation data group has the same member name (data set name) but has a different version number. Every time the SAS data set is updated, a new generation data set is created and the version numbers of the older versions are incremented. The DEFAULT version is called the base version, and is the most recent version of the data. What are the advantages of using generation data sets?

- The most current version of the generation data group is referenced using the base data set name. Thus, downstream programs do not need to worry about date-time stamps in the data set name.
- Older versions of the data are available for `PROC COMPARE` testing when validating the data.
- You don't have to ***remember to save*** the data before creating a new version ☺.

For further information on creating and using generation data sets, see the **genmax** and **gennum** data set options in SAS online documentation. Or, reference Lisa Eckler's SAS Global Forum Paper on Generation Data Sets: http://support.sas.com/resources/papers/proceedings12/051-2012.pdf .

CONCLUSION

Deadlines and deliverables leave little time for learning new tricks and making changes to existing programs. But there are improvements that can be made that save you time and headaches.

This paper provides techniques you may want to consider to update and improve your programs. In the long run, the changes could generalize your code and decrease your maintenance efforts.

CONTACT INFORMATION

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

Marje Fecht
Prowerk Consulting
marje.fecht@prowerk.com
www.prowerk.com