Paper 2584-2018

Automating Code Generation to Reduce Errors and Effort

Steve Cavill, Infoclarity, Melbourne, Australia

ABSTRACT

This paper presents a series of solutions to automate code generation. This enables programmers to reduce effort and likelihood of errors by using some tools in SAS® specifically for that purpose.

In simple terms programmers can just write a macro and call the macro with variable parameters. This method still requires code editing.

An alternative is to store the code variations in an external data set and have SAS read that data set and generate code automatically. This paper discusses techniques used to do this including CALL EXECUTE, CALL SYMPUTX and creating macro variables with PROC SQL INTO: clause.

This technique has additional side benefits including reduced source code change control, simpler tracking of changes, and even the ability for the code changes to be managed by non-programmers

This paper assumes reasonable knowledge of creating and using macro variables.

INTRODUCTION

A common task for many SAS programmers is to write repetitive code with minor variations. Writing the code manually is usually error prone and makes code hard to debug. Most experienced programmers have learned how to use macro calls to reduce the repetitive code, which goes a long way towards reducing the amount of coding and the possibility of errors. This method still requires code editing to create or change the macro calls.

A further step to automate code generation is possible, which removes the requirement to edit code altogether. The code variations can be stored in a data set, and some standard Base SAS code techniques can use the data set to create the repetitive code calls. The data set can be stored in any format that SAS can read, including SAS data sets, RDBMS tables (e.g. Oracle, SQL Server), or simple structures like Excel files and CSV files.

This paper shows three techniques to take the contents of a data set and generate the SAS code to be executed. The methods shown are CALL EXECUTE, CALL SYMPUTX and creating macro variables with PROC SQL INTO: clause. Some pro's and con's of each method are discussed but in the end the choice often comes down to individual programmer preference.

Simple code examples are used to illustrate the techniques and the paper finishes with a real world example from the NSW Bureau of Crime Statistics where the author recently worked as a SAS programmer.

This paper assumes reasonable knowledge of creating and using macro variables.

CREATING SAS CODE FROM A DATA SET

For the purpose of demonstration, we will use this sample code:

```
Data test1;
 person='Donald';
run;
Proc print data=test1; run;

Data test2;
 name='Barack';
run;
Proc print data=test2; run;
```

Our sample dataset looks like this. We have just stored the parts of the code that vary, rather than all the code:

Dsname	Person
Test1	Donald
Test2	Barack

Output 1: Data set "calls"

METHOD 1 – CALL EXECUTE

Call execute is a statement that is placed in a data step. It takes as an argument a character expression that is code to be executed.

We can place any code in the call execute argument. E.g. this is a simple invocation of call execute:

```
data _null_;
call execute ('proc print data=sashelp.cars;run;');
run;
```

In our case we want to make some of the code variable:

```
data _null_;
set calls;
call execute('Data '||dsname||'; person="'||Person||'";run;Proc print
data='||dsname||';run;');
run;
```

We can make this code a bit more readable by creating a SAS variable to contain the code to be called and using SAS functions to build the variable. You can imagine how unreadable that code would become if it was more than the few lines we are using to demonstrate. We can also keep the data set we create containing the calls which can greatly simplify debugging:

```
data calls;
set calls;
call=catx(';'
          ,catx(' ','Data',dsname)
          ,cats('person=',quote(Person))
          ,'run'
          ,cats('Proc print data=',dsname)
          ,'run;')
          ;

call execute(call);
run;
proc print data=calls;run;
```

Obs	dsname	Person	call
1	test1	Donald	Data test1;person="Donald";run;Proc print data=test1;run;
2	test2	Barack	Data test2;person="Barack";run;Proc print data=test2;run;

Output 2: Data set calls with generated code

You can see from the log the SAS code is executed AFTER the data step containing the call execute statement has completed:

```
63         data calls;
64         set calls;
65         call=catx(';'
66                   ,catx(' ','Data',dsname)
67                   ,cats('person=',quote(Person))
68                   ,'run'
69                   ,cats('Proc print data=',dsname)
70                   ,'run;')
71                   ;
72
73         call execute(call);
74         run;

NOTE: There were 2 observations read from the data set WORK.CALLS.
NOTE: The data set WORK.CALLS has 2 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: CALL EXECUTE generated line.
1         + Data test1;person="Donald";run;

NOTE: The data set WORK.TEST1 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

1         +                                Proc print data=test1;run;

NOTE: There were 1 observations read from the data set WORK.TEST1.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds


2         + Data test2;person="Barack";run;

NOTE: The data set WORK.TEST2 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

2         +                                Proc print data=test2;run;


NOTE: There were 1 observations read from the data set WORK.TEST2.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

75
76         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
89
```

Output 3. SAS log from call execute

You will also note in the log above that the formatting of the SAS code in the log can make it hard to read. In addition, if there are errors in the SAS code it can be very difficult to determine which line contains the error, as the code appears in an unformatted block – as illustrated below:

```
NOTE: CALL EXECUTE generated line.
 1         + Data test1;person="Donald";run;

 NOTE: The data set WORK.TEST1 has 1 observations and 1 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.00 seconds
       cpu time            0.00 seconds

 1         +                               Proc print data=test1;format
_char_ $10 _numeric_ 10.2;keep dsname person call;label
       person="the persons first name";label dsname="name of the data set
where the call information is stored ";run;

 22: LINE and COLUMN cannot be determined.
 NOTE 242-205: NOSPOOL is on. Rerunning with OPTION SPOOL might allow
recovery of the LINE and COLUMN where the error has occurred.
 ERROR 22-322: Syntax error, expecting one of the following: a name, a
format name, ;, _ALL_, _CHARACTER_, _CHAR_, _NUMERIC_.
 200: LINE and COLUMN cannot be determined.
 NOTE: NOSPOOL is on. Rerunning with OPTION SPOOL might allow recovery of
the LINE and COLUMN where the error has occurred.
 ERROR 200-322: The symbol is not recognized and will be ignored.
 22: LINE and COLUMN cannot be determined.
 NOTE 242-205: NOSPOOL is on. Rerunning with OPTION SPOOL might allow
recovery of the LINE and COLUMN where the error has occurred.
 ERROR 22-322: Expecting ;.
 202: LINE and COLUMN cannot be determined.
 NOTE: NOSPOOL is on. Rerunning with OPTION SPOOL might allow recovery of
the LINE and COLUMN where the error has occurred.
 ERROR 202-322: The option or parameter is not recognized and will be
ignored.
 ERROR: You are trying to use the numeric format F with the character
variable person in data set WORK.TEST1.

 NOTE: The SAS System stopped processing this step because of errors.
 NOTE: PROCEDURE PRINT used (Total process time):
       real time           0.00 seconds
       cpu time            0.00 seconds
```

Output 4. SAS log from call execute with errors

CALL EXECUTE PRO'S AND CON'S

Call execute is easy to implement and does not require any knowledge of SAS Macro. The next two techniques rely on a knowledge of macro.

Call execute does have one feature that can make it difficult to use and this is when you are using it to call macros. Macro code called by call execute is processed immediately, whereas non-macro code is executed after the data step containing the call execute has completed. This can be a problem if the macro code relies on the SAS code having already executed.

Here is a simple macro %nobs which counts the observations in a data set

```
%macro nobs(ds=,nobs=nlobs);
   %local dsid rc;
   %let dsid=%sysfunc(open(&ds));
   %sysfunc(attrn(&dsid,&nobs));
   %let rc=%sysfunc(close(&dsid));
%mend;
```

In this simple example, the %put statement runs BEFORE the data step, so the information provided by the %nobs macro is incorrect. The correct observation count is 3, but the %put displays 1, because that was in the test data set before the %test macro was called. This kind of timing issue can be VERY hard to debug!

```
%macro test(obscount);
data test;
do i= 1 to &obscount ;output;end;
run;
%put this put statement runs BEFORE the data step: test obs= %nobs(ds=test)
;
run;
%mend;

data test;
i=1;
run;

data _null_;
call execute ('%test(3)');
run;
```

SAS log:

```
74
 75         data _null_;
 76         call execute ('%test(3)');
 77         run;

 this put statement runs BEFORE the data step: test obs= 1;
 NOTE: DATA statement used (Total process time):
       real time           0.00 seconds
       cpu time            0.00 seconds


 NOTE: CALL EXECUTE generated line.
 1         + data test; do i= 1 to 3 ;output;end; run;

 NOTE: The data set WORK.TEST has 3 observations and 1 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.00 seconds
       cpu time            0.00 seconds
```

Output 5: Timing issues with call execute

METHOD 2 – CALL SYMPUTX/SYMPUT

CALL SYMPUTX is preferable to call execute when macro code is involved as it avoids the call execute timing issues with macro calls described above.

Creating macro variables with CALL SYMPUTX is very similar to the process described above for call execute. Call symputx is placed in a data step. The big difference is the generated code must be called by the programmer, it is not automatically executed. Call symputx was added in SAS Version 9. It has some additional features over SYMPUT that are not discussed here. You may find occurrences of symput in code written prior to SAS Version 9.

Symputx syntax:

Call Symputx (Macro Variable, Character Value)

Where Macro Variable is a character expression containing the name of the macro variable, and Character Value is a character expression containing the value you want to assign to the macro variable.

It is very similar in concept to %Let Macro Variable=Character Value, except the values of both the macro variable and its contents are defined at the time of running the data step rather than hardcoded like %let..

Example

Using the same data set as the previous example, we use a data step to create the required macro variables. We want one macro variable per SAS program that we intend to run, in our example that is two macro variables. We will number the variables consecutively which makes it easy to refer to them later.

```
data calls;
set calls;
length call $1000;
call=catx(';'
          ,catx(' ','Data',dsname)
          ,cats('person=',quote(Person))
          ,'run'
          ,cats('Proc print data=',dsname)
          ,'run;'
          )
          ;

call symputx(cats('call',_n_),call);
run;
%put _user_;
```

The code is almost identical to the call execute example. We replaced call execute with call symputx. Call symputx is called once per input observation, so at the end of the data step we have two macro variables, Each macro variable contains a complete SAS program we want to execute. The names of the variables are the word "call" with a suffix being the observation number in the input data set. Cats('call',_n_) is how we name those variables consecutively, that is, call1 and call2.

At the end of the code, I have placed a %put _user_; statement to see the macro variables created.

```
77          %put _user_;
 GLOBAL CALL1 Data test1;person="Donald";run;Proc print data=test1;run;
 GLOBAL CALL2 Data test2;person="Barack";run;Proc print data=test2;run;
```

To call the code we need a simple macro %do loop to invoke each of the macro variables. To use the %do loop we also need to know how many macro variables we created. We can do this by simply counting the number of rows in the input dataset. We can use the %nobs macro (described above) again.

```
%let callcount=%nobs(ds=calls);
```

You can also use the nobs= option on the set statement and use symputx in the data step to put the value of nobs in a macro variable. Both of those methods assume you are using all the rows in the data set. If that's not the case, use a counter in the data step (e.g. counter+1;) and then use symputx at the end to put the final value of counter into a macro variable.

Here is the program to call the macro variables and execute the desired SAS programs:

```
options mprint;
%macro runthecalls;
    %do i = 1 %to &callcount ;
       &&call&i ;
    %end ;
%mend runthecalls;
%runthecalls
```

Note we use the mprint option. This is because we are calling SAS code from within a macro, and without mprint we would not see any of the generated SAS code in the log.

`&&call&i` is how we invoke the sequence of macro variables. This piece of code is resolved twice (because of the double ampersand). In the first pass the double ampersand is resolved to a single ampersand. This allows the &i to resolve first. When i=1 this is the resolution process:

```
&&call&i => &call1 => data test1;….etc
```

A common mistake for new macro programmers is to code `&call&i` in the above situation, which won't work, because then SAS will look for a macro variable called &call, which doesn't exist. Worse, it may actually exist but does not contain anything we actually want and produce code that won't run correctly.

Here is the log of the program above:

```
62          options mprint;
63          %macro runthecalls;
64          %do i = 1 %to &callcount ;
65             &&call&i ;
66          %end ;
67          %mend runthecalls;
68          %runthecalls
MPRINT(RUNTHECALLS):   Data test1;
MPRINT(RUNTHECALLS):  person="Donald";
MPRINT(RUNTHECALLS):  run;

NOTE: The data set WORK.TEST1 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


MPRINT(RUNTHECALLS):  Proc print data=test1;
MPRINT(RUNTHECALLS):  run;

NOTE: There were 1 observations read from the data set WORK.TEST1.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds


MPRINT(RUNTHECALLS):   ;
MPRINT(RUNTHECALLS):   Data test2;
MPRINT(RUNTHECALLS):  person="Barack";
MPRINT(RUNTHECALLS):  run;
```

```
NOTE: The data set WORK.TEST2 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


MPRINT(RUNTHECALLS):  Proc print data=test2;
MPRINT(RUNTHECALLS):  run;

NOTE: There were 1 observations read from the data set WORK.TEST2.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

MPRINT(RUNTHECALLS):   ;
```

Output 6: Macro to call created macro variables

Note the log is a lot more readable than the log from the call execute version of the solution.

CALL SYMPUTX PRO'S AND CON'S

The process to create macro variables and then use them in a loop requires a good understanding of SAS macro, which some may regard as a con, but I think it's a strong argument in favour of using the technique! The macro variable approach avoids the timing issue identified with call execute above and the resulting SAS log is certainly easier to read.

METHOD 3 – PROC SQL INTO: CLAUSE

Proc SQL provides a very simple method to create macro variables from a data set. The feature is the INTO: clause of the SELECT statement. There are three forms of SELECT … INTO

1. Select a single row into a single macro variable:
```
proc sql;
select height into :height
 from sashelp.class
 where name='John'
 ;
 %put >>>&height<<<;
```
   writes >>>   59<<< to the SAS log. As 59 is the height of the user named John in the sashelp.class data set.
   This method is not usually very useful because if the query returns more than one row, the value stored in the macro variable is the value of the first row returned, which is not often what you want.

2. Select multiple rows into a single macro variable:
```
proc sql;
select name into :names separated by ','
 from sashelp.class
 ;
 %put >>>&names<<<;
```
```
>>>Alfred,Alice,Barbara,Carol,Henry,James,Jane,Janet,Jeffrey,John,Joyce,Jud
y,
Louise,Mary,Philip,Robert,Ronald,Thomas,William<<<
```
Note the "separated by" clause. You can specify any separator you want. All the values from that column are written to the macro variable, using the specified separator.

3. Select multiple rows into multiple macro variables:
```
proc sql ;
    select
       name into :name1 - :name9999
    from
       sashelp.class
  ;
    %let namecount = &sqlobs ;
 %put _user_ ;
GLOBAL NAME1 Alfred
GLOBAL NAME10 John
GLOBAL NAME11 Joyce
GLOBAL NAME12 Judy
GLOBAL NAME13 Louise
GLOBAL NAME14 Mary
GLOBAL NAME15 Philip
GLOBAL NAME16 Robert
GLOBAL NAME17 Ronald
GLOBAL NAME18 Thomas
GLOBAL NAME19 William
GLOBAL NAME2 Alice
GLOBAL NAME3 Barbara
GLOBAL NAME4 Carol
GLOBAL NAME5 Henry
GLOBAL NAME6 James
GLOBAL NAME7 Jane
```

```
     GLOBAL NAME8 Janet
     GLOBAL NAME9 Jeffrey
     GLOBAL NAMECOUNT 19
```

This creates a macro variable for every value returned by the select. The easiest way to see the effect is from the log above. Note that although we specified an upper bound of 9999 (name9999), we only created 19 macro variables, which corresponds to the number of rows returned by the query. Note also the use of the automatic macro variable &sqlobs to count the number of rows returned by the query.

This third method is the method we will use for our example.

All of the methods above can be used to return multiple columns as well. The syntax for that is to just specify multiple columns in the query and a corresponding number of macro variables, separated by commas. E.g.:

```
proc sql ;
   select
      name,age into :name1 - :name9999,:age1-:age9999
   from
      sashelp.class
 ;
```

For our example the proc SQL code to create the macro variables is:

```
proc sql ;
   select
      catx(';'
          ,catx(' ','Data',dsname)
          ,cats('person=',quote(Person))
          ,'run'
          ,cats('Proc print data=',dsname)
          ,'run;'
          )
   into :call1-:call9999
   from
      calls ;
   %let callcount = &sqlobs ;
```

Note the similarity to the data step and call symputx. The macro variables created are exactly the same as the macro variables created by call symputx, thus the method to execute the desired SAS programs is the same:

```
options mprint;
%macro runthecalls;
    %do i = 1 %to &callcount ;
       &&call&i ;
    %end ;
%mend runthecalls;
%runthecalls
```

There's an even shorter way to invoke the code using the proc SQL method, by taking advantage of the **"separated by"** clause.

```
proc sql ;
   select
      catx(';'
          ,catx(' ','Data',dsname)
          ,cats('person=',quote(Person))
          ,'run'
          ,cats('Proc print data=',dsname)
          ,'run;'
          )
   into :allthecalls separated by ';'
   from
      calls ;

&allthecalls
```

The &allthecalls macro variable contains all the generated code, so to run the code just reference the macro variable in your SAS code, as shown. This method does not require a macro loop, so the code is shorter, but can appear a bit obscure so use it sparingly!

In the examples above I have been using %put _user_ to show the values of the macro variables of interest. This of course shows all the macro variables in the session so it can be hard to read if you have many. If you want to show just one macro variable, you need to use a macro quoting function, otherwise SAS will try to run the code after the first semicolon. E.g:

```
%put %quote(&allthecalls);
```

PROC SQL INTO: PRO'S AND CON'S

Creating macro variables with proc SQL is very simple syntax. The ability to create multiple macro variables in one clause and the ability to concatenate multiple values into one macro variable are quite powerful features and require less code than a corresponding data step and call symputx.

On the other hand, the data step provides much more flexibility in data manipulation if your data needs a lot of logical processing to achieve the desired result.

The choice of data step/call symputx vs proc SQL into: will typically depend on the input data and programmer preference.

AN EXAMPLE APPLICATION

At New South Wales Bureau of Crime Statistics and Research, we process hundreds of thousands of records of reported crimes and court appearances every quarter. We use the process described above to produce reports of the overall numbers as a data check that an obvious data processing error has occurred. For example, if the number of court appearances increases or decreases by more than 10% from month to month, (which is an unlikely outcome) that is possibly an indication of a data processing error. Being alerted to that possibility allows us to check the data in more detail before publishing erroneous statistics.

DATA SET TO PRODUCE REPORTS:

All the tests are stored in a data set. We used Excel as it is easy to edit and accessible by users without SAS. We used a source management system to keep track of changes.

(Partial listing)

test number	system	input dataset	dataset filter	current library
1	COURTS	disposal	casetypedisposal in ('LC','TRIAL','SENTENCE')	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(
2	COURTS	disposal	casetypedisposal in ('LC','TRIAL','SENTENCE')	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(
3	COURTS	disposal	casetypedisposal in ('LC','TRIAL','SENTENCE')	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(
4	COURTS	disposal	casetypedisposal in ('LC','TRIAL','SENTENCE')	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(
17	COURTS	disposaloffence		&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(
18	COURTS	disposalpenalty		&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMONTH.\(

test number	compare library	file type	proc tabulate style summary table	formatted values
1	same	sas	finalisationjd*(bailcode all),finalisationdate	bailcode $bailcode2L.
2	same	sas	finalisationjd*(bailcode all),finalisationdate	bailcode $bailcode2L.
3	same	sas	Finalisationjd*casetypedisposal,finalisationdate	
4	same	sas	Finalisationjd*casetypedisposal,finalisationdate	
17	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_LAST_RUNMONTH.\OL	sas	Finalisationjd*outcomecode,finalisationdate	Outcomecode $outcomecode2L.
18	&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_LAST_RUNMONTH.\OL	sas	Finalisationjd*penaltycode,finalisationdate	penaltycode $penaltycode2L.

test number	period date	period size months	period offset months	period count	tolerance lower percent	tolerance upper percent	max absolute diff
1	finalisationdate	3	3	4	-10	20	20
2	finalisationdate	3	12	4	-10	20	20
3	finalisationdate	1	3	12	-10	20	20
4	finalisationdate	1	12	12	-10	20	20
17	finalisationdate	1	0	12	98	102	20
18	finalisationdate	3	0	4	98	102	20

PROGRAM TO PROCESS THE DATA CHECKS DATA SET:

This program produces a data set containing calls to the ONEDataCheck macro, which runs tabulates and compares the outputs of the tabulates. I have not included the source code of ONEDataCheck as it is not relevant to this paper. (partial listing)

```
%macro AutoChecks(system=, xlparamfile=) ;
%* ;
%* Get the macro calls from the excel file. ;
%* ;
%LibnameExcel(libref=tests, path=&xlparamfile., precopy=Y) ;

data tests (keep=call) ;
   set  tests.tests ;
   length call $1000 ;

   if test_number ne . ;
   if upcase(system) = "&system." ;
```

```
      if compare_library = 'same' then
         compare_library = current_library ;

      %* Handle commas in macro parameter ;
      proc_tabulate_style_summary_tabl =
       tranwrd(proc_tabulate_style_summary_tabl,',','%str(,)') ;

      call = catx(','
                  ,cats('testid=', system)
                  ,cats('reportsuffix=', test_number)
                  ,cats('currentlibrary=', current_library)
                  ,cats('currentdataset=', input_dataset)
                  ,cats('comparelibrary=', compare_library)
                  ,cats('comparedataset=', input_dataset)
                  ,cats('tablestatement=', proc_tabulate_style_summary_tabl)
                  ,cats('perioddate=', period_date)
                  ,cats('periodendmth=', data_run_month)
                  ,cats('periodsize=', period_size_months)
                  ,cats('periodcount=', period_count)
                  ,cats('periodoffset=', period_offset_months)
                  ,cats('formatstatement=', formatted_values)
                  ,cats('datasetfilter=', dataset_filter)
                  ,cats('lowerpct=', tolerance_lower_percent)
                  ,cats('upperpct=', tolerance_upper_percent)
                  ,cats('alloweddiff=', max_absolute_diff)
                  ,cats('outputpath=', report_path)
                  ) ;
      call = cats('%ONEDataCheck(', call, ')') ;

      output ;
   run ;

   %* ;
   %* Load each call into a macro variable and count the number of calls. ;
   %* ;
   proc sql ;
      select
         call into :call1-:call9999
      from
         tests ;
      %let callcount = &sqlobs ;
   quit ;

   options mprint ;

   %* ;
   %* Submit each call to the One Click data checking macro. ;
   %* ;
   %do i = 1 %to &callcount ;
      &&call&i ;
   %end ;

   libname tests clear ;
   %mend AutoChecks ;
```

LOG OF EXECUTED CODE:

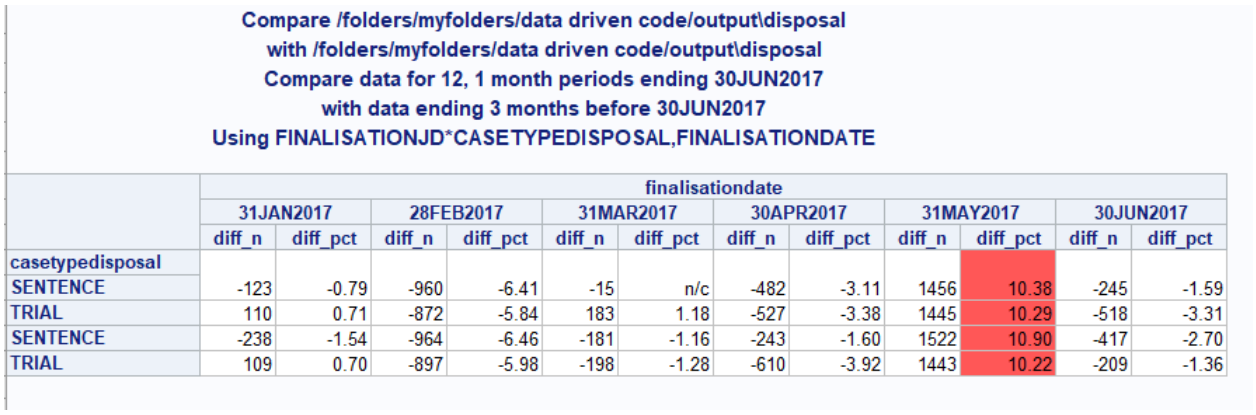
This shows the log simply consists of repeated calls to the ONEDataCheck macro with various parameters which reflect the data in the data checks data set shown above.

(partial listing)

```
%ONEDataCheck(testid=COURTS,reportsuffix=1,currentlibrary=work,currentdatas
et=disposal,comparelibrary=work,comparedataset=disposal,tablestatement=fina
lisationjd*(bailcode
all),finalisationdate,perioddate=finalisationdate,periodendmth=201706,perio
dsize=3 ,periodcount=4,periodoffset=3,formatstatement=bail code
$bailcode2L.,datasetfilter=casetypedisposal in
('LC','TRIAL','SENTENCE'),lowerpct=-10
,upperpct=5,alloweddiff=20,outputpath=&COMMON_NETWORKPATH.\COURTS\DATA\&COM
MON_RUNMONTH.\DETAILS) ;

%ONEDataCheck(testid=COURTS,reportsuffix=2,currentlibrary=&COMMON_LOCALPATH
.\COURTS\DATA\&COMMON_RUNMONTH.\OUTPUTS\MONTHLY\&COMMON_RUNMONTH.,currentda
taset=disposal,comparelibrary=&COMMON_LOCALPATH.\COURTS\DATA\&COMMON_RUNMON
TH.\OUTPUTS\MONTHLY\&COMMON_RUNMONTH.,c
omparedataset=disposal,tablestatement=finalisationjd*(bailcode
all),finalisationdate,perioddate=finalisationdate,periodendmth=&COMMON_RUNM
ONTH,periodsize=3,periodcount=4,periodoffset=12,formatsta tement=bailcode
$bailcode2L.,datasetfilter=casetypedisposal in
('LC','TRIAL','SENTENCE'),lowerpct=-
10,upperpct=5,alloweddiff=20,outputpath=&COMMON_NETWORKPATH.\COURTS\DATA\&C
OMMON_RUNMONTH.\DETAI LS)

%ONEDataCheck(testid=COURTS,reportsuffix=4,currentlibrary=&COMMON_LOCALPATH
.\COURTS\DATA\&COMMON_RUNMONTH.\OUTPUTS\MONTHLY\&COMMON_R
UNMONTH.,currentdataset=disposal,comparelibrary=&COMMON_LOCALPATH.\COURTS\D
ATA\&COMMON_RUNMONTH.\OUTPUTS\MONTHLY\&COMMON_RUNMONTH.,c
omparedataset=disposal,tablestatement=Finalisationjd*casetypedisposal,final
isationdate,perioddate=finalisationdate,periodendmth=&COM
MON_RUNMONTH,periodsize=1,periodcount=12,periodoffset=12,formatstatement=,d
atasetfilter=casetypedisposal in  ('LC','TRIAL','SENTENCE'),lowerpct=-
10,upperpct=20,alloweddiff=20,outputpath=&COMMON_NETWORKPATH.\COURTS\DATA\&
COMMON_RUNMONTH.\DETA ILS)

%ONEDataCheck(testid=COURTS,reportsuffix=17,currentlibrary=&COMMON_LOCALPAT
H.\COURTS\DATA\&COMMON_RUNMONTH.\OUTPUTS\MONTHLY\&COMMON_
RUNMONTH.,currentdataset=disposaloffence,comparelibrary=&COMMON_LOCALPATH.\
COURTS\DATA\&COMMON_LAST_RUNMONTH.\OUTPUTS\MONTHLY\&COMMO
N_LAST_RUNMONTH.,comparedataset=disposaloffence,tablestatement=Finalisation
jd*outcomecode,finalisationdate,perioddate=finalisationda
te,periodendmth=&COMMON_RUNMONTH,periodsize=1,periodcount=12,periodoffset=0
,formatstatement=Outcomecode
$outcomecode2L.,datasetfilter=,lowerpct=98,upperpct=102,alloweddiff=20,outp
utpath=&COMMON_NETWORKPATH.\COURTS\DATA\&COMMON_RUNMONTH. \DETAILS) ;
```

SAMPLE REPORT OUTPUT:

The ONEDataCheck macro produces a report similar to the one below. The highlighted column shows the data in May 2017 was more than 10% higher than the data 3 months prior (Feb 2017), which will trigger a more thorough data check as that amount of variance is suspicious.



Compare /folders/myfolders/data driven code/output\disposal
with /folders/myfolders/data driven code/output\disposal
Compare data for 12, 1 month periods ending 30JUN2017
with data ending 3 months before 30JUN2017
Using FINALISATIONJD*CASETYPEDISPOSAL,FINALISATIONDATE

	finalisationdate											
	31JAN2017		28FEB2017		31MAR2017		30APR2017		31MAY2017		30JUN2017	
	diff_n	diff_pct	diff_n	diff_pct	diff_n	diff_pct	diff_n	diff_pct	diff_n	diff_pct	diff_n	diff_pct
casetypedisposal												
SENTENCE	-123	-0.79	-960	-6.41	-15	n/c	-482	-3.11	1456	10.38	-245	-1.59
TRIAL	110	0.71	-872	-5.84	183	1.18	-527	-3.38	1445	10.29	-518	-3.31
SENTENCE	-238	-1.54	-964	-6.46	-181	-1.16	-243	-1.60	1522	10.90	-417	-2.70
TRIAL	109	0.70	-897	-5.98	-198	-1.28	-610	-3.92	1443	10.22	-209	-1.36

CONCLUSION

Generating code from data is quite easy with data step and/or proc SQL. The methods described above can significantly cut down on programmer time in coding and debugging. Externalising the variable parts of a program to a data set can assist with change management and also makes it possible for the task of modifying code to be given to non-programmers.

CONTACT INFORMATION

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

Steve Cavill
Infoclarity
Steve.cavill@infoclarity.com.au
www.infoclarity.com.au

SAS Community Page: This and other presentations can be found at my SAS Community presentations page
http://www.sascommunity.org/wiki/Presentations:Stevecavill_Papers_and_Presentations