MWSUG 2017 - Paper SA12

Writing Code With Your Data:
Basics of Data-Driven Programming Techniques

Joe Matise, NORC at the University of Chicago

ABSTRACT

In this paper aimed at SAS® programmers who have limited experience with data step programming, we discuss the basics of Data-Driven Programming, first by defining Data-Driven Programming, and then by showing several easy to learn techniques to get a novice or intermediate programmer started using Data-Driven Programming in their own work. We discuss using PROC SQL SELECT INTO to push information into macro variables; PROC CONTENTS and the dictionary tables to query metadata; using an external file to drive logic; and generating and applying formats and labels automatically.

Prior to reading this paper, programmers should be familiar with the basics of the data step; should be able to import data from external files; basic understanding of formats and variable labels; and should be aware of both what a macro variable is and what a macro is. Knowledge of macro programming is not a prerequisite for understanding this paper's concepts.

INTRODUCTION

When I interview a SAS programmer for a programming-intensive position, there is one feature I look for over all others: whether that programmer is familiar with data-driven programming techniques. A programmer who is not familiar with data-driven programming techniques will have a difficult time working in a programming-intensive position, in my opinion, because they will spend significantly more time writing the same programs; their programs will be difficult to maintain and understand; and they will make significantly more mistakes in their code, compared to someone who is familiar with data-driven programming techniques.

At its core, data-driven programming is the concept of writing code that in some fashion uses information stored either in a dataset or in the metadata of a dataset. In writing a data-driven program, a programmer will identify logic that depends on information either stored in a dataset, or that *could* be stored in a dataset. Rather than including that information in their program, the programmer will instruct the program to read those data and generate lines of code based on the data.

Writing a program this way gives several significant advantages.

- A data-driven program will typically require less maintenance over time, as it can often be written such that no modifications to the program are needed for multiple iterations; only the data will change each time it is run.
- A data-driven program is typically much shorter, meaning fewer opportunities to make typos or logic errors, and simpler maintenance when it is needed. It will also often take less time to write.
- A data-driven program will be easier to understand for others, particularly for non-programmers. Rather than using complex logic or embedding large amounts of information in the code, the code can be short and easily read, and the data driving it can be stored in an accessible fashion.
- Data-driven programs based on data stored in an accessible fashion can often be employed by non-programmers without a programmer's intervention at all. The non-programmer can make changes to the data as needed for each iteration, and then simply run or schedule the program.

Data-driven programming is not difficult to master, once a programmer is aware of the concept. Several techniques are commonly used for writing a data-driven program, and are quite simple to implement even for a novice programmer. Think of the techniques as tools in a toolbox; just as a novice woodworker can easily operate a hammer or screwdriver to attach two pieces of wood together, a novice SAS programmer can pull one or more of these techniques out of their toolbox to write code that is maintainable and understandable.

DEFINING DATA-DRIVEN PROGRAMMING

Data-driven programming is the practice of removing information from code that can be defined as data and instead storing that information in a dataset. The programming code then exists to read in the dataset(s) and convert the data into executable code.

DATA-DRIVEN PROGRAM EXAMPLE

In this example, we will use the dataset SASHELP.PRDSAL2 which is included with your SAS installation. The assignment will be to produce reports of sales versus projections for different regions in the company.

INITIAL PROGRAMMING APPROACH

The first step to writing a data driven program is to write a shell of the program as you would without the data-driven elements. This gives a good starting point, and also allows the programmer to analyze the written program for locations to convert to data-driven code.

Before we summarize, we first need to add a new variable to the dataset. There is not a single regional director for each country/state combination; some of the country/state combinations are grouped together (specifically, Mexico has only one regional director; Canada has two – one for Ontario and Quebec, one for the rest of Canada; and the United States has four, one for each Census division (West, South, Midwest, Northeast). For that, we'll create a new classification variable. We'll also create a nice format to print the region names out. While you're looking at the example program, see if you can spot the easy places we can employ data-driven programming:

```
data prdsal2_regions;
  set sashelp.prdsal2;
  select(country);
    when('Mexico') region=1;
    when('Canada') do;
      if state in ('Ontario','Quebec') then region=2;
      else region=3;
    end;
    when ('U.S.A.') do;
      if state in ('New York') then region=4;
      else if state in ('Illinois') then region=5;
      else if state in ('North Carolina','Texas','Florida') then region=6;
      else region=7;
    end;
    otherwise;
  end;
run;

proc format;
  value regionf
  1='Mexico'
  2='Canada East'
  3='Canada West'
  4='U.S.A. East'
  5='U.S.A. Midwest'
  6='U.S.A. South'
  7='U.S.A. West'
  Other=' '
  ;
quit;
```

Then, we will go with a fairly simple PROC REPORT. This will serve as the shell of our report:

```
options nobyline;
title "Report of Sales for Mexico – attn: Sandra Jimenez, VP Sales, Mexico
City";
proc report data=prdsal2_regions nowd spanrows;
  where region=1;
  by region;
  format region REGIONF.;
  columns prodtype year month actual predict net_projections;
  define prodtype/group;
  define month/group order=data;
  define year/group;
  define actual/analysis sum;
  define predict/analysis sum;
  define net_projections/computed;
  compute net_projections;
    net_projections = actual.sum - predict.sum;
  endcomp;
  break after year/summarize page;
run;
```

This gives us a good starting point – and at a glance has several places we can easily replace code with data-driven programming. Did you see them?

DATA-DRIVEN PROGRAMMING TECHNIQUES

In this paper we are going to cover five major techniques that are helpful in constructing a data-driven program. None require advanced SAS programming skills; however, knowledge of a few specific areas can be helpful in employing these techniques.

BASIC CONCEPTS

Some basic concepts to review before delving in to the data driven techniques

- Macro Variables
- Basic macros (simple parameter-fill macros, no %if/%then etc.)

If you're unfamiliar with either of these, you may find it useful to review a basic SAS Macro tutorial prior to continuing, although many of these techniques can be employed without knowledge of macros.

USING AN EXTERNAL CONTROL FILE TO DEFINE RELATIONSHIPS OR DRIVE LOGIC

One simple method of data-driven programming, and perhaps the most powerful, is to use an external file to define relationships and/or to drive programming logic in your code. Many of our programs will rely on some relationship being defined in our data. One example here is the relationship between regions and country/state. While it is fairly straightforward to use the select loop to define region, it is both inefficient and poor technique to define a data relationship in your code. Instead, create a table that defines this relationship, and import that table – then you can perform a simple merge. Odds are, you already *have* a table defining this relationship – your client or manager probably sent you the list of which country/state goes with which region in an Excel file (or email, which could be pasted into Excel).

In this case, the first datastep is quite easily converted into a simple merge:

```
proc import file="C:\Users\matise-joe\Documents\SESUG2016 BB229 Data.xlsx"
            out=states dbms=xlsx replace;
sheet="States";
run;

proc sort data=states;
  by country state;
run;

proc sort data=sashelp.prdsal2 out=prdsal2;
  by country state;
run;

data prdsal2_regions;
  merge prdsal2 states;
  by country state;
run;
```

Now, if the regions change – nothing changes in the code. You simply drop in the new spreadsheet from the client and the program works as before.

DATA-DRIVEN FORMATS AND LABELS

Now, let's take a look at the PROC FORMAT code. Odds are that same region-country crosswalk has a listing of the regions and their names, right? You might need to summarize this yourself from the full crosswalk, but either way that PROC FORMAT is pretty easy to source from it. Here we will assume there is a separate tab – as that tab will be useful later on, anyway.

```
data for_format;
  set mydata.regions;
  start   = region;
  label   = name;
  fmtname = 'REGIONF';
  output;
  if _n_=1 then do;
    hlo='o';
    label=' ';
    output;
  end;
run;
proc format cntlin=for_format;
quit;
```

That will do nicely. It also includes an extra line – the line with hlo='o' – which will protect us from mismatches (this creates the "OTHER" line that appears in the format). If this were a particular concern, we could change the label for this group to "ERROR" or similar.

PULLING DATA INTO MACRO VARIABLES USING PROC SQL: SELECT INTO

Before we move to the next section (the report), we need to cover a more basic technique – or specifically, one approach to a more basic technique. That is, the technique of taking data and generating actual SAS code statements from it. There are several methods for doing this, no one method particularly better than any other in general; as such, we will only cover one method here. For a discussion of the different methods and their strengths/weaknesses, see "List Processing Basics" by Art Carpenter and Ron Fehd as below.

For this kind of work, we like to use the PROC SQL SELECT INTO method with SEPARATED BY, as it generates a single macro variable which includes all of the code we need to call for a particular use. Note a significant limitation of this method: it can generate at most 65534 characters (the limit of a macro variable), and will simply truncate at that point if it goes over, so if you have a very large dataset that you are creating code from, you may need to consider alternate options.

To use this, you do not need any particular knowledge of SQL itself; the calls are always the same:

```
proc sql;
  select name
    into :namelist separated by ' '
    from sashelp.class;
quit;

%put &=namelist;
```

This generates a macro variable &namelist which contains each row's `name` value concatenated, with a space delimiter. Not very helpful as it is, unless you want to perform something like

```
if name in (&namelist.) then output;
```

However, what if you wanted to make a report for each name, like we do? It seems like it would be simple to create a macro that took `name` as an argument, and then call that macro once per name value. That's quite possible now. Imagine a macro `%report`:

```
proc sql;
  select cats('%report(name=',name,')')
    into :namelist separated by ' '
    from sashelp.class;
quit;

%put %superq(namelist);
```

That macro would then be executed once per name simply by including on the next line the macro variable, unquoted:

```
&namelist.
```

Note the use of the `cats` function, which concatenates arguments (here, the call of the macro, the variable `name` from the dataset, and the ending parenthesis). If you look in the output window, you'll see what those concatenations look like. You can add NOPRINT to the PROC SQL statement if you want to suppress that output, but I like to keep it on as it makes it easy to validate my code.

USING DATA TO CONTROL PROGRAM LOGIC VIA MACROS

Here, then, we simply construct a simple macro that accomplishes our needs, and call it from our region dataset. In this case, we have two parameters to send: the name of the region, and the name of the recipient of the report. (We assume here that by-value processing will not work as we want separate reports for each recipient. In general, when possible, by-group processing is superior to this method, but sometimes it's simply not possible.)

The macro itself will look like this:

```
%macro run_region_report(region=, to=);
title "Report of Sales for #byval(region) - attn: &to.";

proc report data=prdsal2_regions nowd spanrows;
  where region=&region.;
  by region;
  format region REGIONF.;
  columns prodtype year month actual predict net_projections;
```

```
  define prodtype/group;
  define month/group order=data;
  define year/group;
  define actual/analysis sum;
  define predict/analysis sum;
  define net_projections/computed;
  compute net_projections;
    net_projections = actual.sum - predict.sum;
  endcomp;
  break after year/summarize page;
run;
%mend run_region_report;
```

And then the call, using PROC SQL, will look like this:

```
proc sql;
  select cats('%run_region_report(region=',region,',to=%nrbquote(',director,'-
',title,'))')
    into :report_list separated by ' '
    from mydata.regions
  ;
quit;

&report_list.
```

Note the %nrbquote in the TO parameter; that is helpful when passing essentially free text (as names will be), as pesky things like apostrophes can break your code otherwise.

SOME OTHER TECHNIQUES

Now that we've got our report nicely data-driven, we can move on to some other techniques that will help in other circumstances.

For example, what if our client decided they did not like the structure of this report? Instead of eight report sections, one per year/product type, they want *one* report with three columns per year and product type. But – guess what, they still want that pesky calculated column, so we can't use *ACROSS* variables without calculating it outside of the proc or hardcoding column locations – or can we? Let's see how we could do this in a single proc step instead.

In order to do this in one step and use *ACROSS* variables, we need to use statements like this to calculate the net projection results:

```
compute net_projections;
  _C4_  = _C2_  - _C3_;
  _C7_  = _C5_  - _C6_;
  _C10_ = _C8_  - _C9_;
  _C13_ = _C11_ - _C12_;
endcomp;
```

This is where SAS reports end up causing all sorts of problems for future maintenance: those columns are not named in a way that is directly related to the actual variable names, but rather for the column number in the final display. What our job is, then, is to create those lists by determining how many columns there will be.

USING ODS OUTPUT TO OBTAIN VALUES FROM PROCS

Most Procs have OUTPUT= statements, or OUT= options, or similar methods of obtaining information in dataset form from the Proc. However, some Procs have more information that is available through the ODS OUTPUT facility.

In this case, we can use the NLEVELS table from PROC FREQ to find out how many Year values we have in our dataset, which will determine how many compute statements we need. That does not have an OUT= option, as it is a statement on the Proc itself (not a Table statement):

```
proc freq data=prdsal2_regions nlevels;
  tables year/noprint;
run;
```

This gives us the value (4) that we need, but it does not store it in a dataset. To do that, we need to first find out what the name of the table is, which we will use ODS TRACE to learn:

```
ods trace on;
proc freq data=prdsal2_regions nlevels;
  tables year/noprint;
run;
ods trace off;
```

ODS TRACE tells us the name is NLEVELS, so we plug that into ODS OUTPUT:

```
ods output nlevels=levels_year;
proc freq data=prdsal2_regions nlevels;
  tables year/noprint;
run;
```

And now we have the NLEVELS value in a dataset. We can then use it to drive a macro, like so:

```
%let start_column = 2;  *the first column for the across variables;

%macro create_columns(column=);
  %local col1 col2;
  %let col1 = %eval(&column.-2);
  %let col2 = %eval(&column.-1);
  _C&column._ = _C&col1._ - _C&col2._;
%mend create_columns;

data _null_;
  set levels_year;
  do _i = 0 to nlevels - 1;
    call symputx(cats('ccol',_i),
                 cats('%create_columns(column=',_i*3+&start_column. + 2,')'));
  end;
run;
```

Now we have several macro variables which we need to put into our program. While we could do this with a macro loop, there's an easier way.

THE DICTIONARY TABLES

SAS provides you a set of tables known as the dictionary tables, which provide access to a wide range of metadata related to your SAS session. Most commonly used are *dictionary.tables* and *dictionary.columns*, which provide access to the list of tables in the current session (in any library) and the list of columns in those tables, respectively, along with their respective properties. They are roughly the tabular versions of information available in *PROC DATASETS* and *PROC CONTENTS*, respectively.

In our case, the table *dictionary.macros* is most helpful. That contains a row for each defined macro variable in the current session. Since we know that our macro variables all begin with *ccol*, and that in our current session no other macro variable does (be sure of this, or use *symdel* to ensure it!), we can pull the macro calls into a single macro variable with a simple call:

```
proc sql;
  select value
    into :list_ccol separated by ' '
    from dictionary.macros
    where name like 'CCOL%'
;
quit;
```

This creates a new macro variable that contains all of our calls to *%create_columns*. Then it is used like so:

```
proc report data=prdsal2_regions nowd spanrows;
  where region=&region.;
  by region prodtype;
  format region REGIONF.;
  columns month prodtype, year, (actual predict net_projections);
  define prodtype/across;
  define month/group order=data;
  define year/across;
  define actual/analysis sum;
  define predict/analysis sum;
  define net_projections/computed format=dollar10.2;
  compute net_projections;
     &list_ccol.;
  endcomp;
  rbreak after/summarize page;
run;
```

PULLING IT ALL TOGETHER: A DATA-DRIVEN PROGRAM IN ACTION

We can now put these elements together and produce a single data-driven program, which will be easy to maintain and straightforward to use for other projects. Simply wrap the last section in a macro, and call it like we did before (in Using Data To Control Program Logic Via Macros). The full program is available at https://github.com/snoopy369/MWSUG-2017.

CONCLUSION

Data driven programming gives a programmer the ability to write programs that are easily understandable, highly reusable, and low maintenance when run multiple times. The programmer is not only able to reduce the number of lines of code necessary to achieve the objective, but is often able to make use of already existing documents to drive their programs. Data driven programming is one of the most powerful techniques available to serious SAS programmers, and a vital tool in any programmer's toolbox.

REFERENCES AND RECOMMENDED READING

Henderson, Donald et al, 1992. "Building Data-Driven Applications Using the SAS' Applications System: Selected Techniques". *Proceedings of SUGI '92*. Cary, North Carolina; SAS Institute, Inc. Available at http://www.sascommunity.org/sugi/SUGI92/Sugi-92-42%20Henderson%20Johnson%20Rabb%20Septoff%20Shusterman%20Smith%20Thornton%20VanDusen%20Yuan.pdf

Fehd, Ronald and Art Carpenter, 2007. "List Processing Basics: Creating and Using Lists of Macro Variables". *Proceedings of SAS Global Forum 2007.* Cary, North Carolina; SAS Institute, Inc. Available at http://www2.sas.com/proceedings/forum2007/113-2007.pdf

Carpenter, Art. 2012. *Carpenter's Guide to Innovative SAS Techniques.* Cary, NC; SAS Institute, Inc.

SAS Institute Staff, 2009. "Accessing SAS System Information by Using Dictionary Tables". SAS 9.2 SQL Procedure User's Guide. Cary, North Carolina; SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/sqlproc/62086/HTML/default/viewer.htm#a001385596.htm

ACKNOWLEDGMENTS

The author would like to in particular acknowledge the denizens of the SAS-L Listserv (https://listserv.uga.edu/cgi-bin/wa?A0=SAS-L) for the hundreds of posts over the years by many

distinguished members that have helped form the author's understanding of the topic of data-driven programming.

CONTACT INFORMATION

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

Joe Matise
NORC at the University of Chicago
55 E Monroe
Chicago, IL 60603
Matise.joe@gmail.com