148-2012
Go Beyond The Wizard With Data-Driven Programming

Renato G. Villacorte
Fairbank, Maslin, Maullin, Metz & Associates, Santa Monica, California

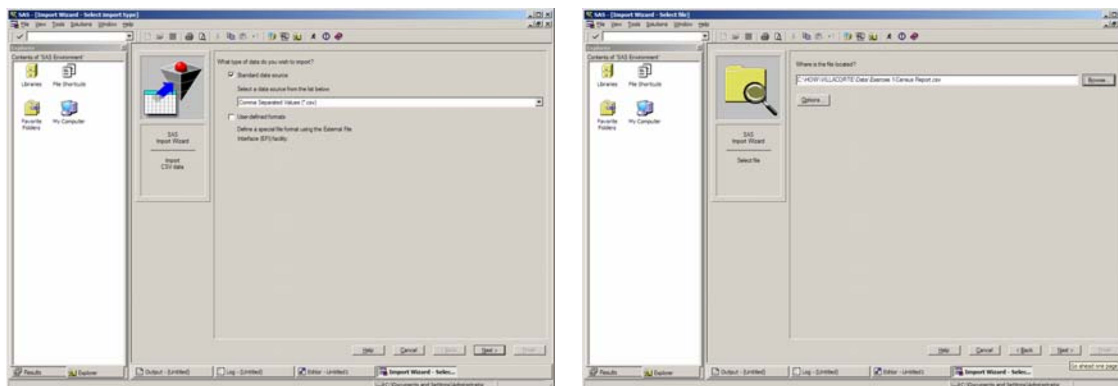
ABSTRACT
Programming techniques that take advantage of Data-Driven Programming will be demonstrated for Novice and Intermediate Users of Base SAS®. Most users already take advantage of Data-Driven Programming with wizards. Wizards harvest information through an interface and then write and execute programs based on those parameters. As programmers' skills evolve, they may want to edit the product of a wizard in order to confront a variety of problems that a wizard could not anticipate. Going a step further, programmers can author their own wizards that make their own work easier. The workshop will include demonstrations in working with built-in SAS® wizards, developing simple Data-Driven Programs, and the use of parameter-gathering techniques using Enhanced Editor, simple macro language, and ODS.

INTRODUCTION
Tools like the Import Wizard are useful in simplifying tasks and are also often used to generate executable programs tailored to our project. Looking closely, the Wizard is really a program that writes another program and then executes it. Creating a Wizard is the practice of Data-Driven Programming. This paper will demonstrate the utility of an Import Wizard and the detailed product of its program. We will also present methods of taking advantage of existing data-driven programming and the tools needed to create your own. All of these methods are available in BASE SAS and are appropriate training for the Novice and Intermediate programmer.

WHAT IS DATA-DRIVEN PROGRAMMING?
Data-driven programming describes the practice of writing programs that write other programs. A data-driven program starts with a pre-written block of code, collects parameters to insert into the block, and then writes out a complete program that can be executed on its own. The most sophisticated data-driven programs attempt to author the most complex blocks of codes with minimal user input. These more sophisticated programs gather additional parameters through procedure output or results generated through the Output Delivery System (ODS). Since the syntax of a program can depend on the output of analytical procedures, the final code that results can be considered "driven" by the data.

WIZARDS
Wizards are the most familiar examples of a data-driven program's front end. In our SAS example, the Data Import Wizard collects information about the file to be imported into a SAS dataset.

IMPORT WIZARD
In the Import Wizard's first two screens, it asks the user for the type of delimiter found in the target file and its location on the hard drive.



The next two screens ask which library will be used to create the new dataset and if the resulting program file will be saved. Once executed, the Import Wizard inserts the collected parameters in a PROC IMPORT template with the following code:

```
PROC IMPORT OUT= RESULTS.IMPORT1
            DATAFILE= "c:\HOW\VILLACORTE\Data\Exercise 1-Census Report.csv"
            DBMS=CSV REPLACE;
     GETNAMES=YES;
     DATAROW=2;
RUN;
```

The Import Wizard automatically submits the PROC IMPORT and if the log displays that the file has been imported successfully, the dataset will be found in the destination library.

PROC IMPORT IS JUST ANOTHER WIZARD

PROC IMPORT is a fine example of data-driven programming. As you can see, the only parameters entered into the PROC IMPORT template were those collected with the Import Wizard's window interfaces. You would expect that an import program would need to know more about the data it will be importing. The PROC actually reads the first several lines of data and collects variable names, variable types, and an estimate of data length. Then, PROC IMPORT writes DATA STEP syntax with the data-driven parameters. To discover the actual code that is generated, submit the PROC IMPORT code and then RECALL the submitted code. Unlike most other PROCs, the recalled code is actually the more basic code that conducts the import. Portions of the recalled code are displayed below:

```
data RESULTS.IMPORT3                             ;
%let _EFIERR_ = 0; /* set the ERROR detection macro variable */
infile 'c:\HOW\VILLACORTE\Data\Exercise 1-Census Report.csv'
   delimiter = ',' MISSOVER DSD lrecl=32767 firstobs=2 ;
   informat _ $12. ;
   informat VAR2 $9. ;
   informat VAR3 $22. ;
   informat S01 $16. ;
   informat VAR5 $16. ;
   informat VAR6 $16. ;
   informat VAR7 $16. ;
   informat VAR8 $16. ;
   informat VAR9 $16. ;
   informat VAR10 $27. ;
   format _ $12. ;
   format VAR2 $9. ;
   format VAR3 $22. ;
   format S01 $16. ;
   format VAR5 $16. ;
   format VAR6 $16. ;
   format VAR7 $16. ;
   format VAR8 $16. ;
   format VAR9 $16. ;
   format VAR10 $27. ;
input
            _ $
            VAR2 $
            VAR3 $
            S01 $
            VAR5 $
            VAR6 $
            VAR7 $
            VAR8 $
            VAR9 $
            VAR10 $
;
if _ERROR_ then call symputx('_EFIERR_',1);  /* set ERROR detection macro variable */
run;
```

The resulting code can also be viewed as inserted parameters into the proper place of a template. PROC IMPORT creates code in several distinct blocks. There is a header for the import routine that consists of the DATA and INFILE statements. The parameters in these lines are exactly those collected by the import wizard which includes delimiter, filename, and destination dataset. The footer of the import routine merely consists of some error detecting code and the run statement that terminates the import routine. (Note: The error trapping statements in the header and footer are included in this exhibit but are beyond the scope of this paper.) The body, made up of three distinct code blocks, is the result of the data-driven programming within PROC IMPORT. PROC IMPORT derived variable names and attributes by looking at the first rows of the file and making decisions about the expected formats. Then, it created matching sets of INFORMAT and FORMAT statements. Lastly, it created an INPUT statement followed by a variable listing.

WIZARDS ARE JUST A STARTING POINT

As you can see, the PROC IMPORT parameter gathering code makes some odd decisions about variable names. If we look at the actual file we are importing, the decisions start to make sense.

	A	B	C	D	E	F	G	H	I	J
1				S01	S01	S01	S01	S01	S01	S01
2				Total popu	Total popu	Total popu	Total popu	Total popu	Total popu	Total popu
3										
4										
5										
6				SUBHD01(	SUBHD01(	SUBHD01(	SUBHD02(	SUBHD02(	SUBHD02(	HD03
7	GEO.id	GEO.id2	GEO.displ	Number	Number	Number	Percent	Percent	Percent	Males per
8	Id	Id2	Geograph	Both sexe:	Male	Female	Both sexe:	Male	Female	
9	0100000US		United Sta	3.09E+08	1.52E+08	1.57E+08	100	100	100	96.7

The sample data file we are importing is a worksheet downloaded from the US Census website. Notice that this is not a plainly formatted data file with the usual first row of variable names. Instead, PROC IMPORT sees the first row beginning with three blank columns and seven iterations of “S01.” PROC IMPORT judges most of the data within these columns as unsuitable and reverts to default variable naming. The first column is labeled as “_” and the two following blank columns are assigned VAR2 and VAR3. The fourth column contains an acceptable variable name, “S01,” but the columns that follow are assigned VAR5-VAR10. The decisions made by PROC IMPORT will result in a successful import, but it is likely to be only the starting point in the processing of this data. Edits need to be made to the DATA step coding in order to successfully label and retrieve the numerical information contained in the worksheet.

HOW TO WRITE A SIMPLE DATA-DRIVEN PROGRAM

The ability to write a program that writes another program is a valuable tool for any programmer analyst. Designing and writing these types of programs will require a strong sense of logic and a working knowledge of macro programming. However, these projects become easier when tasks are broken up into small and manageable blocks. First, we will start with a clean, working program. Then, we will replace the parameters with macro variables. Lastly, we will use a DATA statement to write the program into a file. In the following example, we will create a simple PROC REPORT program that analyzes a portion of census data. The parameters we expect to use to drive the program consist of a dataset name and two variables for analysis.

START WITH A WORKING PROGRAM

This simple PROC REPORT will calculate the sum of two variables within a dataset. To create and test a working program, we used the RESULTS.POPULATION4 and two if its variables, Men and Women.

```
/* Simple Proc Report */
Proc Report data=RESULTS.Population4 nowindows;

Column Men Women Total;

Define Men /Analysis sum width=15 format=comma15.;
Define Women /Analysis sum width=15 format=comma15.;
Define Total /Computed width=15 format=comma15.;

Compute Total;
 Total=Sum(Men.Sum,Women.Sum);
endcomp;

run;
```

ASSIGN MACRO VARIABLES AND SUBSTITUTE WITHIN PROGRAM

Data-driven programs collect parameters and store them in macro variables. The macro variables are inserted into the working code and will resolve into the correct parameters when the program is written to a file. The following set of code demonstrates the assignment of the macro variables and their placement within the program.

```
/* Assign Macro Variables */
%Let DSet=Results.Population4;
%Let Variable_1=Men;
%Let Variable_2=Women;

/* Substitute Literal Variable Names with Macro Variables */
Proc Report data=&DSet. nowindows;

Column &Variable_1. &Variable_2. Total;

Define &Variable_1. /Analysis sum width=15 format=comma15.;
Define &Variable_2. /Analysis sum width=15 format=comma15.;
Define Total /Computed width=15 format=comma15.;

Compute Total;
 Total=Sum(&Variable_1..Sum,&Variable_2..Sum);
endcomp;

run;
```

In an actual data-driven program, the %LET portion of this code would be replaced with a parameter gathering technique. These techniques can include using a Windows interface like the Import Wizard or enter them directly into a %MACRO statement as a macro parameter. More advanced techniques take advantage of PROC output or the OUTPUT DELIVERY SYSTEM to derive parameter values.

WRITE OUT THE PROGRAM

The last step in our simple example is to write out the program to a file. As you may know, SAS programs are simple text files with a *.SAS extension. First, we will assign a filename and use the DATA _NULL_ method to write plain text into the file. Then, we can start each line of our sample code with a PUT statement and open double-quote. We will terminate each line with a closing double-quote and semi-colon.

```
/* Assign Filename */
Filename Meta1 'C:\HOW\Villacorte\Results\Meta1.sas';

/* DATA step to write plain text into a file */
Data _null_;  file Meta1;

/* Put Statements and Quotation Marks */
Put "Proc Report data=&DSet. nowindows;                              ";

Put "Column &Variable_1. &Variable_2. Total;                         ";

Put "Define &Variable_1. /Analysis sum width=15 format=comma15.;     ";
Put "Define &Variable_2. /Analysis sum width=15 format=comma15.;     ";
Put "Define Total /Computed width=15 format=comma15.;                ";

Put "Compute Total;                                                  ";
Put " Total=Sum(&Variable_1..Sum,&Variable_2..Sum);                  ";
Put "endcomp;                                                        ";

Put "Run;                                                            ";

/* RUN statement to terminate the code*/
run;
```

The Enhanced Editor is very useful in this stage because of its context sensitive formatting. Notice that the text within the quotes is colored differently than the executable code outside of the quotes. Unbalanced quotes are very likely to be noticed since the color scheme will look less orderly. Finally, the RUN statement is appended to finish off the DATA STEP and terminate the file export.

SIMPLE BUILDING BLOCKS

This seemingly simplistic example can serve as the foundation for more complex code. A more sophisticated example may use datasets to store either parameters or program statements or both. In that case, we would need to set up header and footer code blocks to surround a body of code. The body would come from individual rows as the DATA statement steps through a dataset. The previous PROC IMPORT example illustrates this technique. In that case, the variable names are placed in a dataset and can be written out as single lines of the INFORMAT, FORMAT, and INPUT sections of the body. While the idea of writing this complex code may seem overwhelming, it becomes a simple matter when you break down each programming event into simple blocks of code.

PARAMETER GATHERING TECHNIQUES

Applying Parameter Gathering Techniques is the next step in developing data-driven programs. Ideally, the user should only enter the minimum amount of information and the program would determine the rest. In this section, we will demonstrate a few simple ways to get information about datasets or the data within them.

PROCEDURE OUTPUT DATASETS

Many SAS® PROCS have an option to generate a dataset containing the results of the procedure. PROC CONTENTS is one example and can provide a great deal of information about the target dataset.

```
/* Proc Contents to get variable attributes */
Proc Contents
       Data=Results.Population4
       Out=Results.VariableSpecs1;
run;
```

The OUT= option of this PROC creates a dataset containing forty variables that describe the attributes of each variable in the Population4 data. This can be useful for data-driven programs that intend to write code for single variables in a dataset. Referring back to our IMPORT example, the listings of each variable in the body sections (INFORMAT, FORMAT, and INPUT) are derived from another dataset containing the individual variable names. PROC CONTENTS is one technique of gathering this type of variable listing.

```
/* Add KEEP Statement to filter */
Proc Contents
       Data=Results.Population4
       Out=Results.VariableSpecs2 (KEEP=Varnum Name Label )
       ;
run;
```

Another advantage to OUT= options is that they allow the use of certain dataset keywords like KEEP=. In this modification of the code, we can trim down the forty variables created by PROC CONTENTS into the select few that we may need.

ANALYSIS PROCEDURES

Another useful set of parameters to have for data-driven programming are the results of analysis PROCs like PROC FREQ. In this example, the OUT= dataset contains the tabulated frequencies for values within the SEX variable.

```
/* Standard Procedure Output */
Proc Freq Data=Data.Class;
       Table Sex /Out=Results.GenderTabulation;
run;
```

The resulting GenderTabulation dataset contains a row for each valid variable value, its corresponding count, and percentage of the total valid cases. This dataset can be used for generating program code for individual variable values or making programming decisions based on the statistics found in the analyses. Perhaps one such application would create a more detailed analysis program on only the 5 most frequent cases found in a dataset.

STORE DATASET VALUES INTO A MACRO VARIABLE

Often, the program is designed to just hunt down one data point and store it into a macro variable for data-driven programming. This can be achieved with the SYMPUT function.

```
/* Load a record value into a Macro Variable */
Data _NULL_;
        Set Data.Class;
        Call Symput ("NewMacroVariableName",Name);
run;
```

In this example, the value of NAME within the CLASS dataset is assigned to a macro variable named "NewMacroVariableName." Subsequent to this code, the value can be inserted into the code by resolving the macro variable "&NewMacroVariableName." (It is important to note that this example is overly simplistic. If the Class dataset has multiple rows, it is the value of the last row that will ultimately get assigned to the macro variable.)

GATHERING FROM ODS OUPUT

There are several other PROCS within SAS that do not have the OUT= option but do output some attractive data for our data-driven programming. For instance, PROC LOGISTIC creates several tables of statistics and model estimates as a result of its analyses. The Output Delivery System actually creates a table for each set of results it plans to output. To retrieve these tables, we must determine the name of the ODS Object and then specify which objects to retain as datasets.

ODS TRACE

We can turn ODS TRACE ON to get a log listing of every ODS object created in the PROCs to follow. This will continue until we submit ODS TRACE OFF.

```
ODS TRACE ON;
Proc Logistic
        Data=Data.Class;
        Model SEX=Age Height;
run;
ODS TRACE OFF;
```

A portion of the resulting log is exhibited below.

```
Output Added:
-------------
Name:        ParameterEstimates
Label:       Parameter Estimates
Template:    Stat.Logistic.ParameterEstimates
Path:        Logistic.ParameterEstimates
-------------

Output Added:
-------------
Name:        Association
Label:       Association Statistics
Template:    Stat.Logistic.Association
Path:        Logistic.Association
-------------
```

ODS OUTPUT

The Name: entry is what we need to retrieve the output datasets. By using the ODS OUTPUT statement, we can save these objects as permanent datasets.

```
ODS OUTPUT
        Parameterestimates=Results.Ex8_Paramset
        Association=Results.Ex8_AssocStats;
```

The next time PROC LOGISTIC is submitted, the ODS objects Parameterestimates and Association will be saved in the Results.Ex8_Paramset and Results.Ex8_AssocStats datasets, respectively. Once we have our dataset saved, it is a simple matter to retrieve any of the available statistics or model estimates for the data-driven program.

CONCLUSION

Data-driven programming is a powerful way to regenerate useful programs that are tailored to a specific project need. Most of us are already accustomed to working with data-driven programs dressed up as Wizards. The Hands-On Workshop designed in support of this paper will train the user to go beyond the design of existing Wizards when our project becomes more complex. It will also demonstrate the tools used to create data-driven programs and techniques that are necessary to gather parameters.

ACKNOWLEDGMENTS

In addition to my wife and children who support me in all endeavors, there are several people who have helped me complete this paper and presentation. I am ever grateful to Eliot Roth, the mentor and friend who introduced me to SAS. My colleagues at FM3 inspire me to improve my programming and development skills. Kimberly LeBouton's invitations to speak at LABSUG have led to my deeper involvement in the SAS community. Lastly, I am grateful to Maribeth Johnson and Ann Stephan for inviting me to present this paper at SGF 2012.

CONTACT INFORMATION

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

Renato G. Villacorte
Senior Vice-President, Fairbank, Maslin, Maullin, Metz & Associates
2425 Colorado Ave, Suite 180, Santa Monica, CA 90404
Work Phone: 310-828-1183
Fax: 310-453-6562
E-mail: renato@fm3research.com
Web: www.fm3research.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.