SESUG Paper 218-2023

Getting Started with PROC DS2

James Blum, University of North Carolina Wilmington;
Jonathan Duggins, North Carolina State University

Abstract

This workshop is designed to give DATA step programmers foundational information to develop programs in PROC DS2. Starting with several common tasks given as DATA step program examples, the workshop goes through transitioning the code examples to PROC DS2 code step-by-step. As part of the process, various similarities and differences between the two steps are noted, and pros and cons of using each are discussed. Suggested topics for study for building on the PROC DS2 concepts presented are also provided, along with reference material to aid in further study.

Introduction

Introduced with SAS® 9.4, the DS2 procedure expands on the power of the DATA step, adding in SQL support including ANSI SQL data types, user-defined methods and packages, and other language extensions. Of course, these modifications come at the expense of learning new syntax rules. In the examples that follow, common tasks from DATA step programming are transitioned to PROC DS2 to illustrate many of the potentiall pitfalls you may encounter when making such a transition. Indeed, much of this workshop is about how things go wrong in rather unexpected ways. However, as we look at some of the challenges, some benefits to transitioning to PROC DS2 are also covered.

Basic Program Structure in PROC DS2

The DS2 procedure has some basic features that differentiate it from many other procedures. It ends with a QUIT statement, it supports RUN-group processing, and it uses methods. There are three fundamental RUN-groups permitted in PROC DS2: DATA, PACKAGE, and THREAD. Most of this paper focuses on the DATA RUN-group, starting with the first example. There are three system methods available: INIT, RUN, and TERM; here we focus primarily on RUN. User-defined methods are also permitted, and we take a look at those near the end of the workshop. Program 1 gives a simple comparison of the DATA step and PROC DS2 using a variation on the most mind-numbing coding example in history.

Program 1: Hello World Adapted to the DATA Step and PROC DS2

```
data _null_;
 Say='I am from the DATA step';
 put say;
run;❶

proc ds2;❷
 data _null_;❸
  method run();❹
   Say='I am from DS2';
   put say;
  end;❹
 enddata;❸
 run;❺
quit;❷
```

❶ An obviously simple DATA step that puts the value of Say to the log.

❷ DS2 is a procedure, which is closed with a QUIT statement.

❸ The DATA statement begins this DS2 program, and names any output tables generated (which can be no table, as here). It closes with the ENDDATA statement.

❹ All executable code lies within methods in DS2. DS2 uses system methods (like RUN), and also supports user-defined methods. Methods close with an END statement.

❺ The RUN statement seems redundant here, but as Program 2 shows, it plays an important role.

As expected, the DATA step puts a single line into the SAS log with the value of the variable Say (along with notes on execution). The DS2 program, in addition to execution notes, also writes the value for its version of Say but, perhaps surprisingly, also generates a warning.

Log 1: PROC DS2 Declaration Warning

```
I am from DS2
WARNING: Line ##: No DECLARE for assigned-to variable say; creating it
as a global variable of type char(13).
```

The ability of PROC DS2 to work with multiple data platforms and data types makes DS2 much more particular about variable declaration and scope than the DATA step. We will explore this further in later examples but, before that, consider the following example on RUN-group processing.

Program 2: The RUN Statement in PROC DS2

```
proc ds2;
 data _null_;
   method run();
     Say='I am from DS2';
     put say;
   end;
 enddata;
 data _null_;
   method run();
     Say='Me too!';
     put say;
   end;
 enddata;
 run;
quit;
```

Log 2: The RUN Statement in PROC DS2

```
ERROR: Compilation error.
ERROR: Parse encountered DATA when expecting end of input.
ERROR: Line ##: Parse failed:  >>> data <<<  _null_;
```

The RUN statement is not an optional step boundary in this setting and the attempt to invoke a second DATA RUN-group fails (place a RUN statement after the first ENDDATA statement and the code functions). The ENDDATA statements are not required here, but including them is considered a good programming practice.

Starting Tasks: Read in a Data Set, Construct a New Variable

To begin with DATA step operations we are all familiar with, compute the EPA combined MPG for the Cars data distributed in the SASHelp library. Typical DATA step code is shown in the first step of Program 3 followed by an attempt to do the same with PROC DS2;

Program 3: Computing a New Variable from the Cars Data

```
data Combo;
 set sashelp.cars;
 ComboMPG=0.55*mpg_city+0.45*mpg_highway;
run;❶

proc ds2;
 data cars;
   method run();
     set sashelp.cars;
     ComboMPG=0.55*mpg_city+0.45*mpg_highway;
   end;❷
 enddata;
 run;
quit;
```

❶ A basic DATA step computation using the EPA formula for combined MPG.

❷ The same statements are used inside the RUN method in a DS2 DATA step.

The RUN method, used in a simple form in the previous section, actually performs much like the implicit loop and implicit output structure of the typical DATA step. It also supports many common DATA step statements like SET. In general, any execution-time statement from the DATA step that is permitted with PROC DS2 must occur inside a method. Unfortunately, this seemingly simple program produces the set of errors shown in Log 3.

Log 3: Computing a New Variable from the Cars Data

```
ERROR: Compilation error.
ERROR: BASE driver, schema name SASHELP was not found for this connection
ERROR: Table "SASHELP.CARS" does not exist or cannot be accessed
ERROR: Line ##: Unable to prepare SELECT statement for table cars
(rc=0x80fff802U).
```

This error set seems to imply that the SASHelp.Cars data set is not present—while it may not be present on some computers, it most certainly was on the machine where this code was initially executed. SASHelp is a concatenated library, made up of several directories. PROC DS2 ***does not support*** connections to these types of libraries. Fortunately, a copy of the cars dataset has been supplied with the data for this workshop. Program 4 assumes the library DS2HOW has been assigned to the folder where the workshop datasets are stored.

Program 4: Computing a New Variable from the Cars Data–Try 2

```
proc ds2;
 data cars;
   method run();
     set DS2HOW.cars;
     ComboMPG=0.55*mpg_city+0.45*mpg_highway;
   end;
 enddata;
 run;
quit;
```

While this still gives a type and scope declaration warning message in the log (not shown), it does create the data set expected. Program 5 uses the DECLARE statement on ComboMPG to try to remove that warning.

Program 5: Specific Declaration of a Variable and its Type

```
proc ds2;
 data cars(overwrite=yes)❶;
   method run();
     declare double ComboMPG;❷
     set DS2HOW.cars;
     ComboMPG=0.55*mpg_city+0.45*mpg_highway;
   end;
 enddata;
 run;
quit;
```

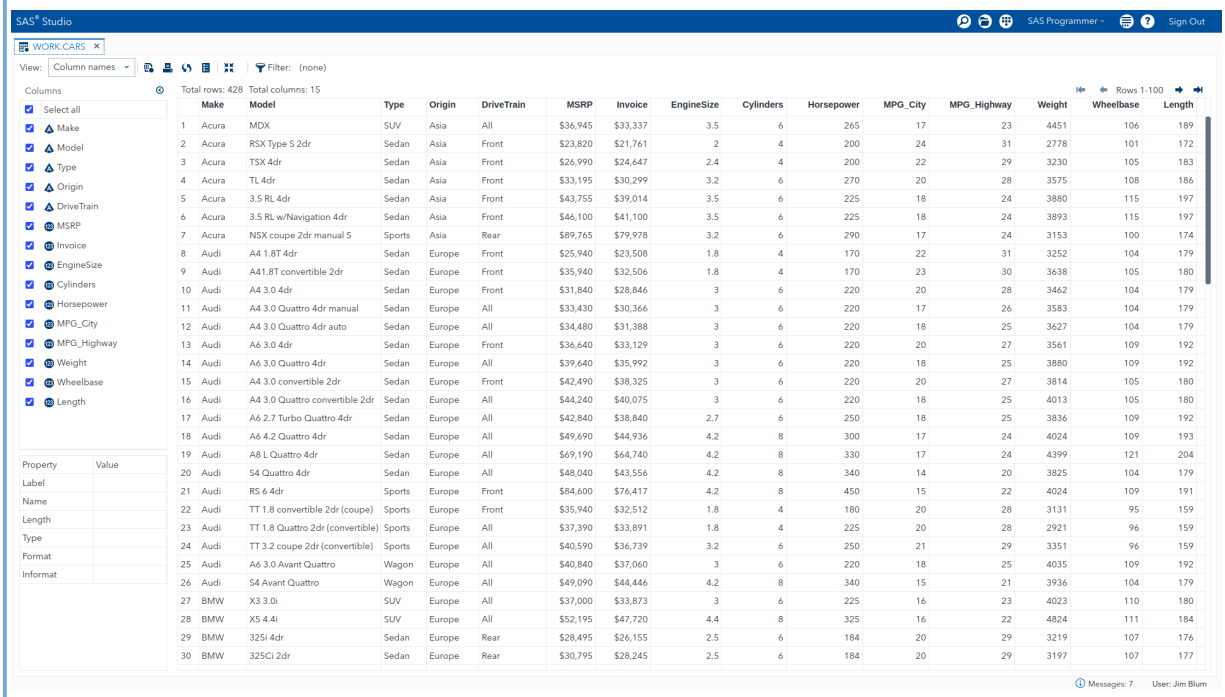
❶ OVERWRITE is a data set (table) option, and the default value is NO–do not overwrite the table if it exists, even if it's in the Work library.

❷ We use the DECLARE statement to make a specific type declaration–potential types are discussed later.

The OVERWRITE=YES option is necessary here, if you do not use it, errors are produced in the log and no execution occurs. Given the data sources it is designed to work with, DS2 is protective about potential overwrites.

Now, if you open the Cars data set from the Work library (see Data View 5), or run PROC CONTENTS on it—you see that the ComboMPG variable is not present. PROC DS2 allows variables to have either global and local scope, with local variables being local to the method in which they are declared. In this instance, ComboMPG is local to the RUN method. The net effect is then similar to dropping the variable in the DATA step. Operationally, ComboMPG is known only to the RUN method, is destroyed when that method ends, and is not part of the output data set (in fact, it is not part of the PDV).

Data View 5: Specific Declaration of a Variable and its Type



	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	Weight	Wheelbase	Length
1	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6	265	17	23	4451	106	189
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2	4	200	24	31	2778	101	172
3	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4	200	22	29	3230	105	183
4	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6	270	20	28	3575	108	186
5	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6	225	18	24	3880	115	197
6	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41,100	3.5	6	225	18	24	3893	115	197
7	Acura	NSX coupe 2dr manual S	Sports	Asia	Rear	$89,765	$79,978	3.2	6	290	17	24	3153	100	174
8	Audi	A4 1.8T 4dr	Sedan	Europe	Front	$25,940	$23,508	1.8	4	170	22	31	3252	104	179
9	Audi	A41.8T convertible 2dr	Sedan	Europe	Front	$35,940	$32,506	1.8	4	170	23	30	3638	105	180
10	Audi	A4 3.0 4dr	Sedan	Europe	Front	$31,840	$28,846	3	6	220	20	28	3462	104	179
11	Audi	A4 3.0 Quattro 4dr manual	Sedan	Europe	All	$33,430	$30,366	3	6	220	17	26	3583	104	179
12	Audi	A4 3.0 Quattro 4dr auto	Sedan	Europe	All	$34,480	$31,388	3	6	220	18	25	3627	104	179
13	Audi	A6 3.0 4dr	Sedan	Europe	Front	$36,640	$33,129	3	6	220	20	27	3561	109	192
14	Audi	A6 3.0 Quattro 4dr	Sedan	Europe	All	$39,640	$35,992	3	6	220	18	25	3880	109	192
15	Audi	A4 3.0 convertible 2dr	Sedan	Europe	Front	$42,490	$38,325	3	6	220	20	27	3814	105	180
16	Audi	A4 3.0 Quattro convertible 2dr	Sedan	Europe	All	$44,240	$40,075	3	6	220	18	25	4013	105	180
17	Audi	A6 2.7 Turbo Quattro 4dr	Sedan	Europe	All	$42,840	$38,840	2.7	6	250	18	25	3836	109	192
18	Audi	A6 4.2 Quattro 4dr	Sedan	Europe	All	$49,690	$44,936	4.2	8	300	17	24	4024	109	193
19	Audi	A8 L Quattro 4dr	Sedan	Europe	All	$69,190	$64,740	4.2	8	330	17	24	4399	121	204
20	Audi	S4 Quattro 4dr	Sedan	Europe	All	$48,040	$43,556	4.2	8	340	14	20	3825	104	179
21	Audi	RS 6 4dr	Sports	Europe	Front	$84,600	$76,417	4.2	8	450	15	22	4024	109	191
22	Audi	TT 1.8 convertible 2dr (coupe)	Sports	Europe	Front	$35,940	$32,512	1.8	4	180	20	28	3131	95	159
23	Audi	TT 1.8 Quattro 2dr (convertible)	Sports	Europe	All	$37,390	$33,891	1.8	4	225	20	28	2921	96	159
24	Audi	TT 3.2 coupe 2dr (convertible)	Sports	Europe	All	$40,590	$36,739	3.2	6	250	21	29	3351	96	159
25	Audi	A6 3.0 Avant Quattro	Wagon	Europe	All	$40,840	$37,060	3	6	220	18	25	4035	109	192
26	Audi	S4 Avant Quattro	Wagon	Europe	All	$49,090	$44,446	4.2	8	340	15	21	3936	104	179
27	BMW	X3 3.0i	SUV	Europe	All	$37,000	$33,873	3	6	225	16	23	4023	110	180
28	BMW	X5 4.4i	SUV	Europe	All	$52,195	$47,720	4.4	8	325	16	22	4824	111	184
29	BMW	325i 4dr	Sedan	Europe	Rear	$28,495	$26,155	2.5	6	184	20	29	3219	107	176
30	BMW	325Ci 2dr	Sedan	Europe	Rear	$30,795	$28,245	2.5	6	184	20	29	3197	107	177

Variables derived from tables listed in a SET statement (or MERGE) are automatically assigned global scope. Looking back at the warnings generated for undeclared variables, it is clear that these were given a global scope as well. To make the declaration for ComboMPG have global scope, it must be moved outside any method, as shown in Program 6

Program 6: Global Declaration of a Variable and its Type

```
proc ds2;
 data cars(overwrite=yes);
   declare double ComboMPG;
   method run();
     set DS2HOW.cars;
     ComboMPG=0.55*mpg_city+0.45*mpg_highway;
   end;
 enddata;
 run;
quit;
```

Now we get a clean log with no warnings (or errors) and ComboMPG appears in the output data set. If you wish to be even more particular about variable declaration (or less so), PROC DS2 provides options to control how undeclared variable references are handled. In the PROC DS2 statement, the SCOND option can be set to NONE, NOTE, WARNING (the default), or ERROR. For each of the first three, the program compiles and executes normally, with the corresponding information type for messages on undeclared variables sent to the log. ERROR causes compilation to fail and the program does not execute. The SAS system option DS2SCOND has the same possible settings and produces the same behavior. Program 7 revisits and earlier attempt without declarations; however, this time the SCOND setting of ERROR stops execution of PROC DS2.

Program 7: Setting Behavior for Undeclared Variables

```
proc ds2 scond=error;
 data cars(overwrite=yes);
   method run();
     set DS2HOW.cars;
     ComboMPG=0.55*mpg_city+0.45*mpg_highway;
   end;
 enddata;
 run;
quit;
```

Log 7: Setting Behavior for Undeclared Variables

```
ERROR: Compilation error.
ERROR: Line ##: No DECLARE for assigned-to variable combompg; creating
it as a global variable of type double.
```

The second error statement in Log 7 is inaccurate, ComboMPG (and the data set) are not created, the compilation error stops execution also. This message is a recast of the warning you get under default conditions, it does not reflect the actual behavior of PROC DS2 under these conditions.

SET, MERGE, and Other Data Assembly Tools

SET and MERGE in PROC DS2

In the DATA step, we can use the SET statement to concatenate multiple data sets, and the same is true in PROC DS2. In the data given with this workshop, a split of the Cars data across origin is given in AsiaCars, EurCars, and USCars; Program 8 puts them back together.

Program 8: Concatenating Data Sets in PROC DS2

```
proc ds2;
 data carsBuild(overwrite=yes);
  declare character(6) Origin;❶
  method run();
   set DS2HOW.AsiaCars(in=InAsia)
       DS2HOW.EurCars(in=InEur)
       DS2HOW.USCars;❷
   if InAsia then Origin = 'Asia';
    else if InEur then Origin = 'Europe';
     else Origin = 'USA';❸
  end;
 enddata;
 run;
quit;
```

❶ The split data sets do not include the Origin variable, so it is created during execution. It is declared as having the Character type, with a length of 6.

❷ The SET statement looks much like it would in a DATA step, including the use of IN= variables.

❸ The IF-THEN-ELSE structure also has the same syntax as could be used in the DATA step.

If no length is given for the character variable in a DECLARE statement like ❶, the default length is 8. If the DECLARE statement is omitted, the code still executes as SCOND is at the default of Warning. What is the length assigned to Origin in that case? Hint: it is the same as it would be if you put the code in the RUN method into the DATA step.

The cars data set is also split across its variable set: CarDims (which includes dimensions, engine specs, gas mileage, along with make, model, and type information), CarPrices (price info with make and model also), and CarOrigins (including only make and origin). The one-to-one match merge shown in Program 9 of the price and dimension information in PROC DS2 looks like that of the DATA step, but there are some key differences.

Program 9: One to One Match Merge in PROC DS2

```
proc ds2;
 data carMerge(overwrite=yes);
  method run();
   merge DS2HOW.CarDims DS2HOW.CarPrices;
   by make model drivetrain;
  end;
 enddata;
 run;
quit;
```

In the DATA step match merge, all data sets listed in the MERGE statement must be sorted in the manner specified in the BY statement. However, if you check the CarDims and CarPrices data, you see that they are not sorted on this key set. In a sense, the BY statement in DS2 is an indication to create these by groupings, rather than an expectation they are already sorted into groups. We will revisit this in a later example.

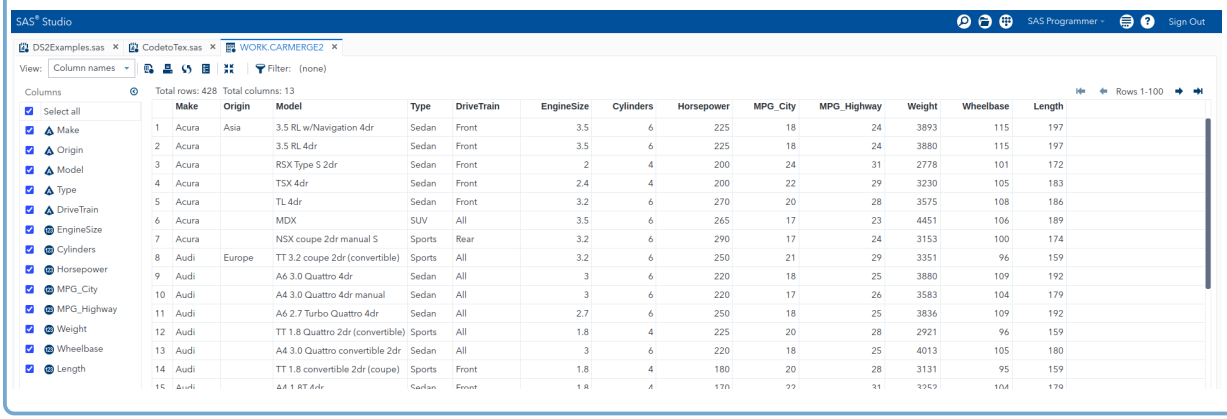
Joining the origin data to either the prices or the dimensions is a one-to-many match merge, Program 10 gives the code you would expect to run to merge CarOrigins to CarDims.

Program 10: One to Many Match Merge in PROC DS2

```
proc ds2;
 data carMerge2(overwrite=yes);
  method run();
   merge DS2HOW.CarOrigins DS2HOW.CarDims;
   by make;
  end;
 enddata;
 run;
quit;
```

As in Program 9, the data sets being merged are not sorted in the manner specified in the BY statement, but the matching of rows is correct. Inspecting the data shows that the there is another difference between this merge in DS2 as opposed to the DATA step. The information that is the "one", Origin, does not populate its value across the "many" rows for all matches on the Make variable, as it would in a DATA step merge. If you have SAS 9.4M6 or later, you can use the RETAIN option in the MERGE statement (after a /) to fix this issue. If you do not have M6 or higher, fear not, there is actually a more interesting way to do this, and other data assembly, in PROC DS2.

Data View 10: One to Many Match Merge in PROC DS2



	Make	Origin	Model	Type	DriveTrain	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	Weight	Wheelbase	Length
1	Acura	Asia	3.5 RL w/Navigation 4dr	Sedan	Front	3.5	6	225	18	24	3893	115	197
2	Acura		3.5 RL 4dr	Sedan	Front	3.5	6	225	18	24	3880	115	197
3	Acura		RSX Type S 2dr	Sedan	Front	2	4	200	24	31	2778	101	172
4	Acura		TSX 4dr	Sedan	Front	2.4	4	200	22	29	3230	105	183
5	Acura		TL 4dr	Sedan	Front	3.2	6	270	20	28	3575	108	186
6	Acura		MDX	SUV	All	3.5	6	265	17	23	4451	106	189
7	Acura		NSX coupe 2dr manual S	Sports	Rear	3.2	6	290	17	24	3153	100	174
8	Audi	Europe	TT 3.2 coupe 2dr (convertible)	Sports	All	3.2	6	250	21	29	3351	96	159
9	Audi		A6 3.0 Quattro 4dr	Sedan	All	3	6	220	18	25	3880	109	192
10	Audi		A4 3.0 Quattro 4dr manual	Sedan	All	3	6	220	17	26	3583	104	179
11	Audi		A6 2.7 Turbo Quattro 4dr	Sedan	All	2.7	6	250	18	25	3836	109	192
12	Audi		TT 1.8 Quattro 2dr (convertible)	Sports	All	1.8	4	225	20	28	2921	96	159
13	Audi		A4 3.0 Quattro convertible 2dr	Sedan	All	3	6	220	18	25	4013	105	180
14	Audi		TT 1.8 convertible 2dr (coupe)	Sports	Front	1.8	4	180	20	28	3131	95	159

Embedded SQL in the SET Statement in PROC DS2

It is possible to embed SQL SELECT statements inside the SET statement, Program 11 puts together the price and dimension information by performing a join inside the SET statement.

Program 11: Embedding a SELECT Statement into the SET Statement

```
proc ds2;
 data carMerge3(overwrite=yes);
  method run();
   set ❶{
         select❷ Origin, Dim.*
          from DS2HOW.CarOrigins as Orig
               inner join
              DS2HOW.CarDims as Dim
               on orig.make =❸ dim.make
         };
  end;
 enddata;
 run;
quit;
```

❶ Braces in the set statement are used as a container for an embedded SQL query.

❷ The SELECT statement is permitted, along with any clauses legal within it, though there is at least one caveat noted in relation to Program 12.

❸ The SAS mneumonic *eq* cannot be used here, the embedded SQL here corresponds to PROC FEDSQL, not PROC SQL.

To expand a bit on ❸, the SQL implementation in PROC DS2 adheres to the ANSI 1999 standard, just as PROC FEDSQL does. Of course, this is done as a precaution, given the variety of data sources DS2 is designed to work with directly. So, SAS-specific syntax legal in PROC SQL will not be legal in the embedded SQL in DS2.

Program 12 packs a fair number of DATA step and SQL concepts into a single RUN method inside PROC DS2, illustrating some of the power and flexibility you gain by using DS2. There is quite a bit to unpack there, so look at it and the call outs carefully.

Program 12: Combining Multiple SQL and DATA Step Concepts

```
proc ds2;
 data AboveAverage(overwrite=yes);
  declare double CityDiff CityPctDiff HwyDiff HwyPctDiff;❶
  method run();
   set {select a.*, CityMean, HwyMean
        from DS2HOW.cars as a
         inner join
             ❷(select type,
                    mean(mpg_city) as CityMean,
                    mean(mpg_highway) as HwyMean
               from DS2HOW.cars
               group by type)❸ as b
         on a.type = b.type
         order by msrp❹
       };
   if mpg_city ge CityMean and mpg_highway ge HwyMean;❺
   CityDiff = mpg_city - CityMean;
   CityPctDiff = CityDiff/CityMean;
   HwyDiff = mpg_highway - HwyMean;
   HwyPctDiff = HwyDiff/HwyMean;
  end;
 enddata;
 run;
quit;
```

❶ We can declare several variables of the same type in a single DECLARE statement.

❷ This SELECT statement includes a join where one table is actually an inline view.

❸ The inline view uses summary functions–make sure you use function names that are ANSI standard compliant–PROC SQL allows some that are not. It also contains a GROUP BY clause.

❹ An ORDER BY clause is included in the main SELECT statement, but this is not the only way (or the most reliable) to sort the resulting data set during DS2 execution.

❺ A subsetting IF is employed to limit the results to cars that are above average on both city and highway MPG.

The embedded query is quite complex, but as it is all built from a SELECT statement, it is perfectly legal. Personally, I like how SQL handles joins, permits inline views and subqueries, and allows grouping and ordering. I am not a particular fan of how new columns are defined, especially if they require conditional logic–for that I prefer DATA step syntax. In DS2, you can work with both. The ORDER BY clause discussed in ❹ can also be achieved with the DS2 BY statement, and the subsetting done by the IF statement noted in ❺ could have been achieved with a WHERE clause in the SQL statement. So, opportunities to mix SQL structures and DATA step structures abound.

As a further note on ❹, in regards to reliability of ORDER BY in this context, the SAS 9.4 DS2 Programmer's Guide states (p. 227): "Some environments might preserve the order imposed by the ORDER BY clause, but others do not." The Programmer's Guide goes on to give a better way to get the ordering in PROC DS2, which is part of the many items illustrated in Program 13.

Program 13: Combining More SQL and DATA Step Concepts

```
proc ds2;
 data Cars
      TypeMPGSummary(keep=(Type CityMeanMPG HwyMeanMPG))
      / overwrite=yes;❶
  declare integer count;❷
  declare double CityMPGTot HwyMPGTot CityMeanMPG HwyMeanMPG;
  method run();
   set {select 'Asia' as origin, *
          from DS2HOW.AsiaCars
         union corresponding
        select 'Europe' as origin, *
          from DS2HOW.EurCars
         union corresponding
        select 'USA' as origin, *
          from DS2HOW.USCars
        };❸
  by type descending msrp;❹
  output Cars;❺

  if first.type then do;❻
    count=0;
    CityMPGTot=0;
    HwyMPGTot=0;
  end;

  count+1;
  CityMPGTot+MPG_City;
  HwyMPGTot+MPG_Highway;❼

  if last.type then do;❽
    CityMeanMPG = CityMPGTot/count;
    HwyMeanMPG = HwyMPGTot/count;
    output TypeMPGSummary;
  end;
 end;
 enddata;
 run;
quit;
```

❶ As with the DATA step, mulitple data sets can be listed in the DATA statement in DS2. The KEEP option has slightly different syntax than in the DATA step (and other locations). The OVERWRITE option is given as a statement option to apply it to both data sets.

❷ While INTEGER is a legal type declaration, and a reasonable one for count, check Log 13 to see its actual type.

❸ This is more work than the concatenation done in Program 8, but it illustrates that a wide variety of SELECT statement logic can be embedded in the SET statement.

❹ Obviously the embedded query does not have this sort order, but it does not matter. The BY statement in DS2 does not refer to the incoming data rows' expected order, it is an instruction for how to order them.

❺ We direct every record to the Cars data set using the OUTPUT statement with a target.

❻ Even though BY operates differently in DS2, one way in which it is the same is in the creation of *first.*

and *last.* variables.

❼ Sum statements are permitted using the same syntax we would use in the DATA step.

❽ To enter this DO group, we condition on a *last.* variable, and inside we target a specific data set for output.

Programs 12 and 13 give a taste of what is possible when you are permitted to combine SQL and DATA step syntax and processing, along with a few extensions (like the BY statement) that are unique to DS2. You can also use many DATA step functions, build your own functions (methods), and build packages from those. Some examples of these concepts are covered in the next section.

Functions, User-Defined Methods, and Packages

DATA Step Functions and DS2 Functions

A data set named Employees is also provided with the files given for this workshop, and for this data the following modifications are to be made:

1. Create a retirement eligibility flag for any employees at least 65 years of age, or at least 60 years of age with at least 30 years of service.
2. Compute an updated salary base on a 2% raise for level 1 employees, 1.5% for level 2 employees, 1% for level 3, and 1.75% for all others. If the job has a level, it is stored as a digit in the third character of the JobCode variable.

A first attempt at this based on DATA step principles is given in Program 14.

Program 14: Computing Retirement Eligibility and Raises

```
proc ds2;
 data emps(overwrite=yes);
  method run();
   set DS2HOW.employees;

   Age=yrdif(DateOfBirth,Today());
   Service=yrdif(DateOfHire,Today());❶

   if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
    else RetEligible='N';

   Level=input(substr(JobCode,3,1),1.);❷

   select(Level);
    when(1) salary=1.02*salary;
    when(2) salary=1.015*salary;
    when(3) salary=1.01*salary;
    otherwise salary=1.0175*salary;
   end;❸
 end;
 enddata;
run;
quit;
```

❶ The YRDIF function is used to create two variables for the employee age and years of service.

❷ We create a numeric value for job level–this does not need to be created as a separate variable, of course, nor does it need to be numeric. It is done this way for diagnostic purposes.

❸ The SELECT group is available, the syntax and logic matches what you expect from the DATA step

Log 14: Computing Retirement Eligibility and Raises

```
ERROR: Compilation error.
ERROR: Compilation error.
ERROR: Parse encountered INPUT when expecting one of:
       identifier constant expression.
ERROR: Line ##: Parse failed: Level= >>> input <<< (substr(JobCode,3,1)
```

Unfortunately, as Log 14 shows, compilation errors occur, and execution does not. The parse failure with "input" is a reflection of the fact that INPUT is not one of the many DATA step functions available in DS2. Program 15 fixes that issue by using INPUTN in place of INPUT.

Program 15: Using INPUTN in Place of INPUT

```
proc ds2;
 data emps(overwrite=yes);
  method run();
   set DS2HOW.employees;

   Age=yrdif(DateOfBirth,Today());
   Service=yrdif(DateOfHire,Today());

   if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
    else RetEligible='N';

   Level=inputn(substr(JobCode,3,1),1.);

   select(Level);
    when(1) salary=1.02*salary;
    when(2) salary=1.015*salary;
    when(3) salary=1.01*salary;
    otherwise salary=1.0175*salary;
   end;
 end;
 enddata;
run;
quit;
```

Replacing INPUT with INPUTN allows execution to commence, even though Log 15 notes a compilation error, and reveals yet another problem. The invalid conversion messages come from the calculation of Age and Service using the YRDIF function on DateOfBirth and DateOfHire.

Log 15: Using INPUTN in Place of INPUT

```
ERROR: Compilation error.
ERROR: Line ##: Invalid conversion for date or time type.
ERROR: Line ##: Invalid conversion for date or time type.
```

DateOfBirth and DateOfHire originate from a SAS data set where they are stored as numeric, double precision values. However, since they have a date format applied, DS2 interprets them as having the date type. This may seem strange, but remember, DS2 is designed to operate across multiple data platforms, where dates are often stored with a date type, so this implicit conversion is another form of protection.

YRDIF only accepts double precision inputs, so the TO_DOUBLE function is added in Program 16 to fix the issue.

Program 16: Dates in DS2 Versus DATA Step

```
proc ds2;
 data emps(overwrite=yes);
  method run();
   set DS2HOW.employees;

   Age=yrdif(to_double(DateOfBirth),Today());
   Service=yrdif(to_double(DateOfHire),Today());

   if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
    else RetEligible='N';

   Level=inputn(substr(JobCode,3,1),1.);

   select(Level);
    when(1) salary=1.02*salary;
    when(2) salary=1.015*salary;
    when(3) salary=1.01*salary;
    otherwise salary=1.0175*salary;
   end;
 end;
 enddata;
run;
quit;
```

Log 16: Dates in DS2 Versus DATA Step

```
WARNING: Line ##: No DECLARE for assigned-to variable age; creating it as
 a global variable of type double.
WARNING: Line ##: No DECLARE for assigned-to variable service; creating it
 as a global variable of type double.
WARNING: Line ##: No DECLARE for assigned-to variable reteligible; creating
 it as a global variable of type char(1).
WARNING: Line ##: No DECLARE for assigned-to variable level; creating it as
 a global variable of type double.❶
NOTE: BASE driver, creation of a DATE column has been requested, but is not
 supported by the BASE driver. A DOUBLE PRECISION column has been created
 instead. A format has been associated with each column.❷
ERROR: Unexpected error detected in function inputn.❸
```

❶ We still have several warnings about undeclared variables to clean up.

❷ DateOfBirth and DateOfHire were given the date type at read in, but they cannot be written to a SAS data set as that type, so another implicit type conversion occurs.

❸ Several of these errors appear, only one is shown. Every time INPUTN encounters a non-digit input this error is thrown. The value returned is missing, so the net effect is the same as using INPUT, but we would prefer to clean up these messages.

Program 17 adds in fixes for the issues shown in Log 16, and notes some other issues of importance.

Program 17: Retirement and Raise Calculations, Cleaned Up

```
proc ds2;
 data emps(overwrite=yes);
  declare char(1) RetEligible;❶
  method run();
   declare double Age Service;
   declare integer Level;❷
   set DS2HOW.employees;

   Age=yrdif(to_double(DateOfBirth),Today());
   Service=yrdif(to_double(DateOfHire),Today());

   if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
    else RetEligible='N';

   if AnyDigit(Reverse(Level)) eq 1
    then Level=inputn(substr(JobCode,3,1),1.);
    else Level = .;❸

   select(Level);
    when(1) salary=1.02*salary;
    when(2) salary=1.015*salary;
    when(3) salary=1.01*salary;
    otherwise salary=1.0175*salary;
   end;
 end;
 enddata;
run;
quit;
```

❶ The RetEligible flag variable is declared prior to the RUN method, making it global and placing it into the PDV and output data set.

❷ Age, Service, and Level are all declared inside the RUN method, making them local to the RUN method and not part of the output data set.

❸ We use the ANYDIGIT and REVERSE functions to determine whether or not to pull a digit off the end. In the ELSE statement, we did not use CALL MISSING to set missing values because we cannot–no CALL routines are supported in DS2.

Program 18 performs a variation on the process in Program 17. Variables for the raise amount and updated salary are added, and included in the output data set, with formats and labels. The FORMAT and LABEL statements are not legal anywhere in PROC DS2, so they must be assigned in a different manner. A HAVING clause in a DECLARE statement can be used to make these assignments. In this case, it requires separate declare statements for Raise and NewSalary as they have different label attributes. The remainder of Program 18 is largely similar to Program 17.

Program 18: Retirement and Raise Calculations, Extended

```
proc ds2;
 data empsB(overwrite=yes);
  declare char(1) RetEligible;
  declare double Raise having format dollar12.2;
  declare double NewSalary having format dollar12.2 label 'Updated Salary';
  method run();
   declare double age service;
   declare integer level;
   set DS2HOW.employees;

    Age=yrdif(to_double(DateOfBirth),Today());
    Service=yrdif(to_double(DateOfHire),Today());

    if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
      else RetEligible='N';

    if AnyDigit(Reverse(Level)) eq 1 then
          Level=inputn(substr(JobCode,3,1),1.);
     else Level = .;

    select(level);
     when(1) Raise=.02*salary;
     when(2) Raise=.015*salary;
     when(3) Raise=.01*salary;
     otherwise Raise=.0175*salary;
    end;
    NewSalary=Salary+Raise;
 end;
 enddata;
run;
quit;
```

User-Defined Methods

Some of you may be users of PROC FCMP for defining functions and/or call routines, and PROC DS2 can use these (though it is not covered here). However, it is also possible to define methods and packages within DS2 to provide such functionality. Program 19 revisits Program 18, re-defining some of the computations via methods.

Program 19: Creating a Method

```
proc ds2;
 data empsC(overwrite=yes);
  declare char(1) RetEligible;
  declare double Raise having format dollar12.2;
  declare double NewSalary having format dollar12.2 label 'Updated Salary';

  method retire(double age, double serve) returns char;❶
   declare char(1) Retire;❷
   if age ge 65 or (age ge 60 and serve ge 30) then Retire='Y';
    else Retire='N';
   return Retire;❸
  end;

  method Raise(double salary, integer group, double rate1, double rate2,
               double rate3, double rate0) returns double;
   select(group);
    when(1) Raise=rate1*salary;
    when(2) Raise=rate2*salary;
    when(3) Raise=rate3*salary;
    otherwise Raise=rate0*salary;
   end;
   return Raise;
  end;❹

  method run();
   declare double age service;
   declare integer level;
   set DS2HOW.employees;

   Age=yrdif(to_double(DateOfBirth),Today());
   Service=yrdif(to_double(DateOfHire),Today());
   RetEligible=Retire(Age,Service);❺

   if AnyDigit(Reverse(Level)) eq 1
     then Level=inputn(substr(JobCode,3,1),1.);
     else Level=.;
   Raise=Raise(Salary,Level,0.02,0.015,0.01,0.0175);❺
   NewSalary=Salary+Raise;
 end;
 enddata;
run;
quit;
```

❶ User-defined methods are defined with a METHOD statement of the form: METHOD *method-name* (*parameter-list*) RETURNS *type*. METHODS are always defined in blocks and terminate with an END statement.

❷ This DECLARE statement is important–if it is not present, the variable Retire is established as global and is placed into the PDV and output data set.

❸ The RETURN statement returns the value of the indicated variable as the return from the method. Its type should match the one stated in the METHOD statement.

❹ The computation for raise is also defined as a method, with several parameters, and no DECLARE statement for the returned variable.

❺ Each method is called at an appropriate point in the code in the same manner as a function is called.

The parameter list for a method is a comma separated list of names, each of which must be preceded by its type. The named parameters are variables local to the method, and the values passed can be variables or expressions that have the appropriate type, or constants that fit the type. Other variables created in the method must be explicitly declared inside the method if they are to be local to that method. If not, they are either given global scope, or an error occurs, depending on your SCOND setting. If you remove the DECLARE statement noted in ❷, you get both a Retire and a RetEligible variable in the PDV and in the output data set.

The fact that the Raise method returns a variable called Raise and there is also a global declaration for a variable named Raise, probably looks a bit strange, and it is perhaps not the best practice–but it is there for a reason. To test your understanding of variable scope, try each of the following and explain the result.

1. Remove Raise= from the expression noted in ❺.
2. Remove the global declaration for Raise in line 4 of Program 19.

There are certainly times when you will want a method to return and/or update an existing variable, and a direct method is given to do so. In Program 20, we revert to the problem of simply updating Salary based on the raise rules. Since Salary is a parameter passed to the method, and it is intended to be updated, it is taken as an IN_OUT parameter.

Program 20: In-Out Parameters

```
proc ds2;
 data empsD(overwrite=yes);
  declare char(1) RetEligible;

  method retire(double age, double serve) returns char;
   declare char(1) Retire;
   if age ge 65 or (age ge 60 and serve ge 30) then Retire='Y';
    else Retire='N';
   return Retire;
  end;

  method SalaryBump(in_out double salary, integer group, double rate1,
                    double rate2, double rate3, double rate0);❶
   select(group);
    when(1) salary=(1+rate1)*salary;
    when(2) salary=(1+rate2)*salary;
    when(3) salary=(1+rate3)*salary;
    otherwise salary=(1+rate0)*salary;
   end;
  end;

  method run();
   declare double age service;
   declare integer level;
   set DS2HOW.employees;

   Age=yrdif(to_double(DateOfBirth),Today());
   Service=yrdif(to_double(DateOfHire),Today());
   RetEligible=Retire(Age,Service);

   if AnyDigit(Reverse(Level)) eq 1
     then Level=inputn(substr(JobCode,3,1),1.);
     else Level=.;
   SalaryBump(Salary,Level,0.02,0.015,0.01,0.0175);❷
 end;
 enddata;
run;
quit;
```

❶ The IN_OUT designation precedes the type declaration for the parameter in the list.

❷ The SalaryBump method is called directly, no assignment expression is used.

Methods can have multiple IN_OUT parameters, and they may also return other values (as of 9.4M5).

Now, there was probably no reason to write these computations as methods. It is actually extra code, and the extra effort is only reasonable if you plan to use these functions repeatedly (a similar rationale for using PROC FCMP to define functions or macros to define dynamic code). If you find yourself in this situation, there is one more step you will want to take, collecting your related methods into packages. The next section generalizes these methods and puts them into a package.

Defining Packages

Packages are collections of methods whose rules are stored in a data set, which can (should) be placed in a permanent library. They can then be loaded into any DS2 routine–and PROC FEDSQL can also access

them. Variations on the Retire and SalaryBump methods are constructed as part of a package, and Program 21 shows the package definition through the methods for raise. The full code for the package is located in the Appendix

Program 21: A Package Definition

```
proc ds2;
 package DS2HOW.EmployeeStuff / overwrite=yes;❶

 method EmployeeStuff();❷
  put;
  put 'Methods Available in EmployeeStuff:';
  put ' Retire';
  put ' Raise';
  put;
 end;
 method retire();❷
  put 'RETIRE Method';
  put 'General Syntax: Retire(Age, AgeLim <,Serve, AgeServLim, ServLim>)';
  put ' Age is double, employee age';
  put ' AgeLim is double, cutoff for age-based retirement eligibility';
  put ' Serve is optional years of service parameter, also requires:';
  put ' AgeServLim: age cutoff when service is included together with...';
  put ' ServLim: Years of service required in conjuction with age';
 end;
 method retire(double age, double agelim, double serve, double ageservlim,
               double servlim) returns char;❸
  declare char(1) Retire;
  if age ge agelim or (age ge ageservlim and serve ge servlim)
    then Retire='Y'; else Retire='N';
    return Retire;
  end;
 method retire(double age, double agelim) returns char;❸
  declare char(1) Retire;
  if age ge agelim then Retire='Y'; else Retire='N';
  return Retire;
 end;
 ❹
 endpackage;
run;
quit;
```

❶ The package definition establishes a package name and a library for storage. Like data sets, the default is to not overwrite a package. Unlike data sets, the option must be given as a statement option, after the /, and cannot be given in the form of a data set option.

❷ Any method can be defined with a null parameter set, as can the package itself. We use the convention that a null reference is set up to print information to the log about the package or method.

❸ The Retire method is defined three times in this package, which is legal as long as its **type signature** is different in each definition. This is a practice known as method stacking.

❹ There are also methods for SalaryBump in the package definition, but they are not displayed here. The full package definition is given in the Appendix.

The type signature of a method is an ordered list of the parameter types. So, methods with the same name have a different type signature when they have a different number of parameters. If they have the same number of parameters, at least one parameter position must have a different type. Either of these are

sufficient for the list of parameters passed to correctly map to a specific instance of the requested method. One note, the CHAR and NCHAR types are ***not*** sufficiently different to give different type signatures when used in the same position–hopefully you do not run into that situation.

Once the package is defined, it can be used in any subsequent DS2 routine. Program 22 shows how to do this–as an extra step, you can restart your SAS session before running this code to show that the methods are available at the start of the session without any code submission (other than a library assignment, perhaps).

Program 22: Using the Package

```
proc ds2;
 data Retire NonRetire / overwrite=yes;
 declare package DS2HOW.EmployeeStuff ES();❶

 method init();❷
  ES.Retire();
  ES.SalaryBump();❸
 end;

 method run();
  declare double age;
  declare double service;
  declare integer level;
  set DS2HOW.employees;

  Age=yrdif(to_double(DateOfBirth),Today());
  Service=yrdif(to_double(DateOfHire),Today());

  if AnyDigit(Reverse(Level)) eq 1
   then Level=inputn(substr(JobCode,3,1),1.);
   else Level=.;

  ES.SalaryBump(Salary,Level,0.02,0.015,0.01,0.0175);❹

  if ES.retire(age,65,service,60,30)❺ eq 'Y' then output retire;
   else output NonRetire;
 end;
 enddata;
run;
quit;
```

❶ To access a package, we use a DECLARE statement, pointing to the library and package name, and giving a name to the package, ES here. The empty argument ensures the information about the EmployeeStuff package is printed to the log when the package is delcared.

❷ This is our first look at the SAS-supplied INIT method. As INIT is short for intialization, this package executes before the RUN method (no matter the order in the code).

❸ The null argument references here print the information for each method to the log before execution of the RUN method starts.

❹ The SalaryBump method uses Salary as an In_Out parameter, so this reference updates salary based on the parameters passed, no assignment is needed.

❺ The retirement flag is created only for evalutating the condition, just as any other function might be used. Methods do not have to be used in assignment statements/expressions.

Conclusion

Transitioning from the DATA step to PROC DS2 is not simple, and seemingly minor issues can frustrate you along the way. But the value and power of PROC DS2 is high if you can exploit it, so hopefully some of the concepts discussed here will help you make the transition. The books by Mark Jordan and Peter Eberhardt are highly recommended if you plan to go down this path, along with a multitude of SAS Blogs on DS2 and related concepts.

Recommended Reading

- SAS® 9.4 DS2 Programmer's Guide
- Mastering the SAS® DS2 Procedure, Mark Jordan, SAS Institute, Cary, NC, 2018
- The DS2 Procedure: SAS Programming Methods at Work, Peter Eberhardt, SAS Institute, Cary, NC, 2016

Contact Information

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

James Blum, University of North Carolina Wilmington
blumj@uncw.edu
http://people.uncw.edu/blumj

Full Employee Package Definition

```
proc ds2;
 package DS2HOW.EmployeeStuff / overwrite=yes;

 method EmployeeStuff();
  put;
  put 'Methods Available in EmployeeStuff:';
  put ' SalaryBump';
  put ' Raise';
  put;
 end;
 method retire();
  put 'RETIRE Method';
  put 'General Syntax: Retire(Age, AgeLim <,Serve, AgeServLim, ServLim>)';
  put ' Age is double, employee age';
  put ' AgeLim is double, cutoff for age-based retirement eligibility';
  put ' Serve is optional years of service parameter, also requires:';
  put ' AgeServLim: age cutoff when service is included together with...';
  put ' ServLim: Years of service required in conjuction with age';
 end;
 method retire(double age, double agelim, double serve, double ageservlim,
               double servlim) returns char;
  declare char(1) Retire;
  if age ge agelim or (age ge ageservlim and serve ge servlim)
    then Retire='Y'; else Retire='N';
    return Retire;
  end;
```

```
 method retire(double age, double agelim) returns char;
  declare char(1) Retire;
  if age ge agelim then Retire='Y'; else Retire='N';
  return Retire;
 end;

 method SalaryBump();
  put;
  put 'SalaryBump Method';
  put 'General Syntax: SalaryBump(Salary,Group,List of Rate Parameter)
       or SalaryBump(Salary,rate)';
  put ' Salary is double for input and is updated for output';
  put ' Group is integer';
  put ' Rates are double, number of parameters is one more than
       levels of group available--';
  put ' Levels of group can be 1, 1 and 2, or 1 and 2 and 3';
  put ' Rates are given in sequence for each level plus another
       for all other categories';
  put 'Without a group, a single rate is given to apply to all records';
  put ;
 end;
 method SalaryBump(in_out double salary, integer group, double rate1,
              double rate2, double rate3, double rate0);
  select(group);
   when(1) Salary=(1+rate1)*salary;
   when(2) Salary=(1+rate2)*salary;
   when(3) Salary=(1+rate3)*salary;
   otherwise Salary=(1+rate0)*salary;
  end;
 end;
 method SalaryBump(in_out double salary, integer group, double rate1,
              double rate2, double rate0);
  select(group);
   when(1) Salary=(1+rate1)*salary;
   when(2) Salary=(1+rate2)*salary;
   otherwise Salary=(1+rate0)*salary;
  end;
 end;
 method SalaryBump(in_out double salary, integer group, double rate1,
              double rate0);
  select(group);
   when(1) Salary=(1+rate1)*salary;
   otherwise Salary=(1+rate0)*salary;
  end;
 end;
 method SalaryBump(in_out double salary, double rate0);
  Salary=(1+rate0)*salary;
 end;

 endpackage;
run;
quit;
```