Paper 3780-2016

DS2 with Both Hands on the Wheel

Peter Eberhardt, Fernwood Consulting Group Inc., Toronto, ON

Xue Yao, Winnipeg Regional Health Authority, Winnipeg, MB

ABSTRACT

The DATA Step has served SAS® programmers well over the years, and although it is handy, the new, exciting, and powerful DS2 is a significant alternative to the DATA Step by introducing an object-oriented programming environment. It enables users to effectively manipulate complex data and efficiently manage the programming through additional data types, programming structure elements, user-defined methods, and shareable packages, as well as threaded execution. This tutorial is developed based on our experiences with getting started with DS2 and learning to use it to access, manage, and share data in a scalable and standards-based way. It facilitates SAS users of all levels to easily get started with DS2 and understand its basic functionality by practicing the features of DS2.

INTRODUCTION

DS2 is a new programming language that is a powerful tool for advanced data manipulation. DS2 is based on object-oriented concepts wherein objects are instances of classes and methods are basic program execution units. It is designed to be simpler to develop programs as well as easier to understand programs to ease maintenance down the road thus, lending itself to more robust programs. As a part of Base SAS, the PROC DS2 enables you to submit DS2 language statements; as a part of SAS High-Performance Analytics environment, the PROC HPDS2 executes DS2 language statements.

This paper is example based, using code samples to highlight some aspects of DS2. For more general introduction to DS2 see Eberhardt 2014.

DS2 OVERVIEW

DS2 has many features. Since this can be overwhelming to the SAS programmer getting started with DS2 we will cover some of the features of DS2 we think are critical to know. As your progress with your DS2 programming you will find there are more features of which you can take advantage. Here we will review four components: Methods, DATA program, Package, and Thread.

CLARITY

In DS2, you are expected to be clear with each identifier that you are using. An identifier is one or more tokens, or symbols, that name programming language entities, such as variables, labels, method names, package names and arrays, as well as data source objects, such as table names and column names. In DS2, almost all identifiers are to be declared using DECLARE statement. A DECLARE statement is allowed only at the top of the programming block in which it is used, otherwise, a compilation error occurs.

Variables are defined using the DECLARE statement. Variables declared within a method are local to that method while variables declared outside all methods are global; this makes them available to all methods. To explicitly declare variable use the DECLARE statement, The DECLARE statement takes the form:

```
DECLARE dataType varName HAVING label ‘label text’ FORMAT sasfmt.
```

For example:

```
DECLARE DOUBLE feeAgePrem65   HAVING label  'Age Premium (65+)' format comma6.2;
DECLARE DECIMAL(10,2)  feeAgePrem65X   HAVING label  'Age Premium (65+) * services';
DECLARE INTEGER i j; /* for DO loops */
```

A best practice is to always run your DS2 programs using variable declaration strict mode (DS2SCOND=ERROR) to enforce the explicit declaration of all program variables.

Here we declare a character string named ***str*** within the ***method init()*** block;

```
proc ds2;
data _null_;
  method init();
    dcl varchar(16) str;   /* declare the variable */
    str = 'Hello World!';
    put str;
  end;
enddata;
run;
quit;
```

In DS2, you are expected to be clear with each indentifier that you are using. An identifier is one or more tokens, or symbols, that name programming language entities, such as variables, labels, method names, package names and arrays, as well as data source objects, such as table names and column names. In DS2, almost all identifiers are to be declared using DECLARE statement. A DECLARE statement is allowed only at the top of the programming block in which it is used, otherwise, a compilation error occurs. A best practice is to always run your DS2 programs using variable declaration strict mode (DS2SCOND=ERROR) to enforce the explicit declaration of all program variables.

SCOPE

In programming, ***scope*** is used to mean where a variable is “visible”; put another way, where its value can be accessed. If you write SAS macro functions you will be familiar with ***local*** (%local) and ***global*** (%global) macro variables. A variable declared within a macro function as:

```
%local macVar
```

is only available in the macro function; when the function completes the variables declared as %local are no longer available. On the other hand, macro variables defined as:

```
%global macVar;
```

are available to PROCs, macros (variables or functions) and DATA steps during the entire SAS session. What sometimes will confuse a SAS programmer learning macro programming is the ability to have two macro variables with the same name but with different scope – one global and one local. When a macro variable is defined local in a macro function, its value will take precedence over a global macro variable with the same name ***while the macro function is executing***. When the macro function completes execution, then the global macro variable is once again available.

The DATA step has no concept of scope; all variables are accessible from in any part of the DATA step (of course if the variable has yet to be assigned a value, accessing the variable will result in a missing value). We could say the variables are global to the DATA step.

In DS2 scope an attribute of identifiers where identifiers refer to program entities:

- method names
- functions
- data names
- labels
- program variables

Although we use program variables as examples, all identifiers are subject to scoping rules.

Scope describes where in a program a variable can be accessed. Global variables have global scope and are accessible from anywhere in the program while local variables have local scope, that is, are accessible only from within the block in which the variable was declared while that block is executing. Each variable in any given scope must have a unique name, but variables in different scopes can have the same name. When scopes are nested, if a variable in an outer scope has the same name as a variable in an inner scope, the variable within the outer scope is hidden by the variable within the inner scope. For example, in the following program two different variables share the same name, x. Global variables ***x and y*** (declared outside all method blocks) have global scope, and local variable ***x*** (declared in INIT) has local scope. Within the local scope of method INIT, local variable ***x*** hides global variable ***x*** therefore the assignment statement assigns 5 to local variable ***x***; since ***y*** is a global variable it is accessible in all methods.

```
proc ds2;
data _null_;
declare int x;  /* global x in global scope */
declare int y;  /* global y in global scope */
method init();
  declare int x;  /* local x in local scope */
  x = 5;        /* local x assigned 5 */
  y = 6;        /* global y assigned 6 */
```

```
   put '0. in init() ' x= y=;
end;
method run();
    put '1. in run()  ' x= y=;
      x = 1;
    put '2. in run()  ' x= y=;

end;
method term();
end;
enddata;
run;
quit;
```

the method INIT first runs, assigns a value (5) to the local variable x and prints x to the log. After method INIT runs, method RUN executes. When we first print the value of ***x*** to the log in method RUN we see it is no longer 5 (actually has no value); the value of 5 belonged to the variable ***x*** which was local to method INIT. Since INIT is no longer active, the variable with the value 5 is no longer available. The LOG:

```
0. in init()  x=5 y=6
1. in run()   x= y=6
2. in run()   x=1 y=6
```

When considering the scope of a variable, a good rule is to keep variables local as much as possible.

```
proc ds2;

data;

dcl char(24) abc abc2;  /* global abc abc2 in global scope */

method init();

      dcl char(8) a b c; /* local a b c in local scope */

      a = repeat('a',5);

      b = repeat('b',6);

      c = repeat('c',7);

      abc = a || b || c;

      abc2 = trim(a) || trim(b) || c;

end;
```

```
enddata;

run;

quit;
```

Variables for output must have global scope to data program. If the variables ***abc*** and ***abc2*** are declared within ***method init()***, there will be a compilation error: No columns for output rowset.

MODULARITY AND ENCAPSULATION

A programming block defines a section of a DS2 program that encapsulates variables and code. Programming blocks enable the modularity and encapsulation of DS2. A programming block contains the modular and reusable codes to perform specific tasks, which can shorten development time and standardize often-repeated or business-specific programming tasks. Layers of programming blocks enable encapsulation and abstraction of behavior, which enhances readability and understandability and other programmers.

In addition, a programming block also defines the scope of identifiers within that block that is where the identifiers can be accessed. An identifier declared in the outermost programming block having global scope. An identifier declared in any nested block having scope that is local to that block.

Block	Delimiters	Scope
Data program	DATA…ENDDATA	Variables that are declared at the top of this program block have global scope within the data program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in data program are included in the program data vector (PDV). **Note:** Global variables exist for the duration of the data program.
Method	METHOD…END	A method is a subblock of a data program, package, or thread program. Method names have global scope within the enclosing block. Variables that are declared at the top of this programming block have local scope. Local variables are not included in the PDV. **Note:** Local variables exist for the duration of the method call.

Block	Delimiters	Scope
Package	PACKAGE…ENDPACKAGE	Variables that are declared at the top of this programming block have global scope within the package. Package-scope variables are not included in the PDV of a data program that is using an instance of the package. **Note:** Package-scope global variables exist for the duration of the package instance.
Thread	THREAD…ENDTHREAD	Variables that are declared at the top of this programming block have global scope within the thread program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in a thread program are included in the thread output set. **Note:** Thread-scope global variables exist for the duration of the thread program instance, but they can be passed to the SET FROM statement in the data program.

Table 1. DS2 Programming Blocks and Scope

DS2 also has two types of arrays: temporary and variable; both of them exist only for the duration of the DS2 program. To create a temporary array, use a DECLARE statement to specify the name, data type, and number and size of the array bounds. The temporary array can be declared in local or global scope. In the following example, C is a temporary array with 5 elements. VARARRAY statement creates a variable array which is a temporary grouping of global variables. The variable array must be declared in global scope. In the following example, N is a variable array with 5 elements.

```
proc ds2;
data _null_;
vararray double N[5];
  method init();
    dcl double i;
    dcl char(2) C[5];
    do i=1 to dim(n);
      N[i]=dim(n)-I;
    end;
    C:=('MB','ON','SK','BC','QC');
    put C[*]=;
    put N[*]=;
```

```
   end;
enddata;
run;
quit;
```

To create, update and manage a dataset in DS2, the table options work like the data set options for Data Step. Table options specify actions that apply only to the tables with which they appear. Most table options can apply to either input or output tables. If a table option is associated with an input table, the action applies to the table that is being read. If the option appears in the DATA statement, SAS applies the action to the output table. In DS2, table options for output tables must appear in the DATA statement, not in any OUTPUT statements that might be present. Some table options can apply to packages and threads. Table options are either enclosed in parentheses or preceded by a forward slash, depending on which statement they are used in. The forward slash can be used for all applicable statements.

```
data prod (drop=(price sales));
package invent /overwrite=yes;
```

Table Options	Data Source	Description
BUFNO	SAS data set	Specifies the number of buffers to be allocated for processing a SAS data set.
BUFSIZE	SAS data set	Specifies the size of a permanent buffer page for an output SAS data set.
COMPRESS	SAS data set	Specifies how observations are compressed in a new output SAS data set. Note: Use with output data sets only.
ENCRYPT	SAS data set	Specifies whether to encrypt an output SAS data set. Note: Use with output data sets only.
LABEL	SAS data set	Specifies a label for a table.
LOCKTABLE	SAS data set	Places shared or exclusive locks on tables.
PW	SAS data set	Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected SAS file.
READ	SAS data set	Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.
WRITE	SAS data set	Assigns a WRITE password to a SAS file that prevents users from writing to a file, unless they enter the password.
ALTER	SAS data set	Assigns an ALTER password to a data set that prevents users from replacing or deleting the file, and enables access to a Read- and Write- protected file.

Table Options	Data Source	Description
TYPE	SAS data set	Specifies the data set type for a specially structured SAS data set.
BULKLOAD	ODBC, Hadoop, Oracle, MySQL…	Loads rows of data as one unit
DBCREATE_ TABLE_OPT S	ODBC, Hadoop, Oracle, MySQL…	Specifies DBMS-specific options to be added to the DATA statement.
DBKEY	ODBC, Hadoop, Oracle, MySQL…	Specifies a key column to optimize DBMS retrieval. Can improve performance when you are processing a join that involves a large DBMS table and a small SAS data set or DBMS table.
DBNULL	ODBC, Hadoop, Oracle, MySQL…	Indicates whether NULL is a valid value for the specified columns when a table is created.
DROP	All	For an input table, excludes the specified columns from processing; for an output table, excludes the specified columns from being written to the table.
KEEP	All	For an input table, specifies the columns to process; for an output table, specifies the columns to write to the table.
RENAME	All	Changes the name of a column
IN	All	Creates a Boolean variable (1 or 0) that indicates whether the table contributed data to the current row. Note: Use with the SET and DATA statements only. Values of IN= variables are available to program statements during the DS2 program, but the variables are not included in the table that is being created, unless they are assigned to a new variable.
OVERWRITE	All	For a table, drops the output table before the replacement output table is populated with rows; for packages and threads, drops the existing package or thread if a package or thread by the same name exists.

Table 2. DS2 Table Options

In Table 2, all the table options of DS2 are listed by their applied data sources. OVERWRITE=YES | NO specifies whether the output table is deleted before a replacement output table is created or whether a package or thread is dropped. For tables, a table is not overwritten unless the OVERWRITE=YES, but use the OVERWRITE=YES statement only with the data that is backed up or with data you can reconstruct. Because the output table is deleted first, data will be lost if a failure occurs while the output table is being written. If OVERWRITE is set to YES in a PACKAGE statement, only the package is dropped. If it's in a THREAD statement, only the thread is dropped.

DATA TYPES

The SAS data step is a powerful programming language based on two data types: numeric (double precision floating point), and fixed length character. The data type can be explicitly stated using the ***LENGTH***, ***FORMAT***, or ***ATTRIB*** statements, or it can be implicitly determined based on the result of a calculation; implicit determination is very common in SAS programs. Because there is no need to explicitly declare variables, subtle errors can creep into DATA step programs when variables are misspelled. DS2 changes this.

First, DS2 provides a rich array of data types; these data types match the data types found in ANSI standard RDBMS. In addition, there is a new ***DECLARE*** statement to define the variable. Data types available in DS2 are:

CHAR(***n***)	a fixed length character string of maximum n characters. This is the same as using LENGTH $n in the DATA step
NCHAR(***n***)	a fixed-length character string like CHAR but uses a Unicode national character set, here ***n*** is the maximum number of multi-byte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters.
VARCHAR(***n***)	a varying-length character string, where ***n*** is the maximum number of characters to store. The maximum number of characters is not required to store each value. If varchar(10) is specified and the character string is only five characters long, only five characters are stored in the column.
NVARCHAR(***n***)	a varying-length character string like VARCHAR but uses a Unicode national character set, where ***n*** is the maximum number of multi-byte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.
BIGINT	a signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
INTEGER	a signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded.
SMALLINT	a signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767.

TINYINT	a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127.
DECIMAL(***p,s***) / NUMERIC(***p,s***)	a signed, exact, fixed-point decimal number, with user specified precision (p) and scale (s). The precision and scale determines the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and right of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. Scale must be less than or equal to the precision. For example, decimal(9,2) stores decimal numbers up to nine digits, with a two-digit fixed-point fractional portion, such as 1234567.89.
DOUBLE	a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point. This is the DATA step numeric.
FLOAT(***p***)	a signed, approximate, single-precision or double precision, floating-point number. The user-specified precision determines whether the data type stores a single-precision or double-precision number. If the specified precision is equal to or greater than 25, the value is stored as a double-precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single-precision number, which is a REAL. For example, float(10) specifies to store up to ten digits, which results in a REAL data type.
REAL	a signed, approximate, single-precision, floating-point number.
BINARY(***n***)	a fixed-length binary data, where n is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
VARBINARY(***n***)	stores varying-length binary data, where n is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If varbinary(10) is specified and the binary string uses only five bytes, only five bytes are stored in the column.
DATE	a calendar date. A date literal is specified in the format yyyy-mm-dd: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as 1975-09-24.
TIME(***p***)	a time value. A time literal is specified in the format hh:mm:ss[.nnnnnnnnn]; a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as 06:30:00. When supported by a data source, the p parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.

TIMESTAMP(***p***)	stores both date and time values. A timestamp literal is specified in the format yyyy-mm-dd:hh:mm:ss[.nnnnnnnnn]: a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as 1975-09-24:06:30:00. When supported by a data source, the p parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.

We will touch on data types again later.

FUNCTIONS

A DS2 function performs a computation or system manipulation on arguments and returns a value like the Data Step. A function expression invokes a function from anywhere in a DS2 program, method, or thread. If the number of arguments and the argument data types match, the function executes. If the number of arguments do not match, an error occurs. If the number of arguments match but one of more argument data types do not match, DS2 attempts to convert the argument data types to those of the function requires. Thus if the function expression contains an argument with a data type of INTEGER and the valid data type for that argument is DOUBLE, DS2 converts that data type to DOUBLE when the function executes. Only four data types, VARBINARY and the date/time data types, DATE, TIME, and TIMESTAMP are non coercible, meaning that the function expression must contain valid data type. We will illustrate a few of the commonly used functions with examples.

We will explore a few examples about functions. %TSLIT() is the function to call macros in Base SAS. SQLEXEC can insert into, update, create, or delete rows from a table. FedSQL statement must be enclosed in single quotation marks. NULL returns 1 if the argument is null, 0 if not null. The IF function conditionally produces a value.

METHODS

Methods are the building blocks of DS2 programs, threads, and packages. A Method is a named unit consisting of related program statements; in DS2, all executable code must reside in methods. A method can be invoked multiple times by name. In its simplest a method has following form:

```
method method_name();
...
      DS2 statements
...
end;
```

To be more useful, methods can take arguments and also return a value; in this way they act the same as a SAS function. The following example shows a method that converts temperature from degrees Fahrenheit to degrees Celsius:

```
method F_to_C(DOUBLE F) returns DOUBLE;
/* convert degrees fahrenheit to degrees celsius */
return (F - 32.) * (5. / 9.)
end;
```

This method takes one argument, a DOUBLE, representing the temperature measured in Fahrenheit; it returns a DOUBLE, the equivalent temperature in Celsius. To invoke the method, you call it as you would a SAS function:

```
DECLARE float c having label ‘Celsius degrees’ format 5.1;
c = F_to_C(212.);
```

Methods are global in scope . There are two types of methods: predefined and user-defined. First, there are four predefined methods: ***INIT***, ***RUN***, ***TERM***, and ***SETPARMS***. Every DS2 program will contain, either implicitly or explicitly, the three methods ***INIT***, ***RUN*** and ***TERM***; if you do not explicitly define them the DS2 compiler will generate them. The ***INIT*** method is automatically called once at the beginning of the program and the ***TERM*** method is automatically called once at the end of the program; this provides an efficient and structured mechanism to invoke start-up and clean-up code. After the ***INIT*** method runs, the ***RUN*** is called; by default, this method will iterate once for each input row. The ***SETPARMS*** method is used in a thread to initialize parameters.

When the built-in methods ***INIT***, ***RUN***, ***TERM*** are explicitly defined, they must be defined without any parameters and without a return value; adding parameters and/or returning a value will result in a compile error. These three methods provide a more structured framework than the SAS DATA Step implicit loop concept.

As noted above, user-defined methods can take parameters and return values; each parameter must have a data type. DS2 allows for method overloading, that is using the same method name but different parameter lists (the method signature). When the method is called, DS2 determines which instance to use based on the arguments passed in. The best match will be the one for which the number of method parameters is equal to the number of arguments, and such that no other method signature has as many exact parameter type matches for the given argument list. If a best match is not found, an error occurs. The following example shows the F_to_C method with two signatures: one for a double precision argument and one for an integer:

```
method F_to_C(DOUBLE F) returns DOUBLE;
  /* convert degrees fahrenheit to degrees celsius */
  return (F - 32) * (5. / 9.);
 end;
 method F_to_C(INTEGER F) returns DOUBLE;
  /* convert degrees fahrenheit to degrees celsius */
  return (F - 32) * (5. / 9.);
 end;
```

The RETURN statement is used to return a value. And each RETURN statement that appears in the method must have an associated return expression.

Methods can also return values in through the parameter list; in this case there can be no RETURN statement in the method. To return values through the parameter list use the IN_OUT modifier:

```
method F_to_C(IN_OUT INTEGER F, IN_OUT DOUBLE C);
  /* convert degrees fahrenheit to degrees celsius */
  C = (F - 32) * (5. / 9.);
 end;
```

Methods can have the same name as a SAS function; if this is the case the SAS function is hidden. To access the SAS function you need to use the SYSTEM modifier:

```
td = SYSTEM.date();
```

PACKAGES

DS2 packages are collections of logically related methods and variables that can be shared and re-used in DS2 programs and threads; for example, you may have a set of measurement conversion functions tested and validated that can then be used through- out the organization. These packages can be thought of as a library that contains the methods. The method stored in a package can be accessed by creating an instance of the package and then using dot notation to call the method. SAS also provides packages for your use; these include: FCMP, Hash, Logger, Matrix and SQLSTMT.

- The FCMP package supports calls to FCMP functions and subroutines from within the DS2 language.
- The Hash package enables you to quickly and efficiently store, search, and retrieve data based on unique lookup keys.
- The Logger package provides a basic interface to the SAS logging facility.
- The Matrix package provides a powerful and flexible matrix programming capability.
- SQLSTMT provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

The PACKAGE statement defines a package, and the DECLARE PACKAGE statement constructs an instance of the package. The following example shows a package definition as well as use of the package:

```
1 PACKAGE convert / overwrite = yes;
method C_to_F(INTEGER C) returns DOUBLE;
  /* convert degrees fahrenheit to degrees celsius */
  return 32 + (C *  (9. / 5.));
end;
method IN_to_CM(DOUBLE inch) returns double;
  return inch * 2.54;
end;
...
ENDPACKAGE;
2 ATA measures (overwrite=YES);
```

3

4

```
declare package convert cnv();
  method init();
    type = 'C';
    do C = 0. to 100.;
       F = cnv.C_to_F(C);
       output;
end;
end;
ENDDATA;
```

1. PACKAGE convert / overwrite = yes;
   All methods between the PACKAGE/ENDPACKAGE are included. ***overwrite=yes*** will recreate the package if it exists; DS2 will not automatically overwrite packages
2. DATA measures (overwrite=YES);
   Starts the DATA program block. ***overwrite=yes*** will recreate the package if it exists; DS2 will not automatically overwrite data tables
3. DECLARE PACKAGE convert cnv();
   Instantiates an instance of the package convert. The instance is called cnv. Since this is declared outside all methods it has global scope
4. cnv.C_to_F(C);
   Calls the C_to_F() method through the package instance cnv.

A package instance can be created two ways. First, as shown above, the instance variable has parenthesis; the parenthesis tells DS2 to automatically instantiate the package instance:

```
DECLARE PACKAGE convert cnv();
```

Second, the instance variable is declared without parenthesis and later it the program it is instantiated with the _NEW_ operator.

```
DECLARE PACKAGE convert cnv;
  ...
cnv = _NEW_ [THIS] convert();
```

The scope of a package instance follows the same rules as the scope for variables; that is, if it declared within a method it is local to that method and if it is declared outside of a method it has global scope. The scope of the package instance can also be set with the _NEW_ operator. In the example above, the keyword ***THIS*** tells DS2 to make cnv global in scope. In addition to the THIS keyword, you can also supply the name of another variable and the new package instance will have the same scope as that variable.

In our examples, DS2 methods and packages are applied to the market basket analysis. Market basket analysis is frequently used by marketing professionals to reveal affinities between individual products or product groupings. It includes the measures of support, confidence, expected confidence and lift. Consider the rule A➔B. The support is the ratio of number of times both items to the total number of transactions. The confidence is the conditional probability that a randomly selected transaction will include item B given item A. The expected confidence is the probability of item B. And the lift is the ratio of the probability of item A and B occurring together to the multiple of the two individual probabilities for item A and item B.

```
  proc ds2;
1 package mba / overwrite=YES;
  dcl double freq_co_occur anl_unit_freq assoc_anl_unit_freq tot_basket_dimensions;
2   method support(double freq_co_occur, double tot_basket_dimensions) returns
             double;
      return freq_co_occur/tot_basket_dimensions;
    end;
    method confidence(double freq_co_occur, double anl_unit_freq) returns double;
      return freq_co_occur/anl_unit_freq;
    end;
    method expect_confidence(double assoc_anl_unit_freq, double
             tot_basket_dimensions)
                returns double;
      return assoc_anl_unit_freq/tot_basket_dimensions;
    end;
    method lift(double confidence, double expect_confidence) returns double;
      return confidence/expect_confidence;
    end;
3  method lift(double freq_co_occur, double anl_unit_freq, double
             assoc_anl_unit_freq, double tot_basket_dimensions) returns double;
  return freq_co_occur*tot_basket_dimensions/(anl_unit_freq*assoc_anl_unit_freq);
  end;
  endpackage;

4 data data.MBAresult2 / overwrite=YES;
  dcl double support confidence expect_confidence lift;
5 dcl package mba M();
  method run();
    set data.MBA;
    support=M.support(freq_co_occur, tot_basket_dimensions);
    confidence=M.confidence(freq_co_occur, anl_unit_freq);
    expect_confidence=M.expect_confidence(assoc_anl_unit_freq,
             tot_basket_dimensions);
    lift=M.lift(confidence, expect_confidence);
```

6
```
end;
enddata;
run;
quit;
```

1. The package named mba is defined with overwrite option.
2. The user defined method such as support is defined.
3. The method named lift is overloaded with different method signature.
4. To output the data in data program.
5. The instance of package mba is declared as M().
6. The method named lift with two input arguments.

THREAD

A DS2 program can run in two different ways: as a program and as a thread. Using threaded processing, sections of the code can run concurrently; each section that executes concurrently is said to be running in a thread. When running in a thread, input data can only come from database tables, not other threads, and output data are returned to the DS2 program that started the thread; in essence, the thread reads the data then does any necessary sub-setting and evaluations and returns these results to the calling programme. If running as a program, input data can include both rows from database tables and rows from DS2 program threads and output data can be both database tables and rows returned to the program.

A thread is defined by the THREAD statement and ended by ENDTHREAD statement. To use a thread, an instance of a thread is created by using the DECLARE THREAD statement. To execute the thread, use the SET FROM statement. In the following example, a simple thread **t** is created, then the instance of t, **t_instance,** is declared, and two threads are executed, using the SET FROM statement in the RUN method.

1. A thread is created

```
thread t;
  dcl int x;
  method init();
    dcl int i;
    do i=1 to 3;
      x=i;
      output;
    end;
   end;
endthread;
data;
   dcl thread t t_instance;
```

2. Thread instance DECLARED

```
    method run();
     set from t_instance threads=2;
     put ‘x= ’ x;
    end;
enddata;
```

3. Read from 2 threads

In this example, the thread instance t_instance is essentially launched twice, the results (printed to Results) would be:

X
1
2
3
1
2
3

From one of the threads

From the other thread

We see that each of the threads returned the three numbers (1,2,3).

If we have a very large dataset that needs to be sub-setted and returned, the use of threads can improve performance in many cases; the level to which performance is improved is dependent on the data store and the processors available. Data stores that are optimized for multi-threaded queries should show the greatest improvement.

The following example is based on sample code from the SAS web site. The example first creates a large SAS dataset, then queries the data using a different number of threads, from non-threaded to eight threads. Two datasets were generated, one with 60,000,003 and one with 600,000,003 rows and “benchmarked” (not a rigourous benchmark) on a Lenovo W540 laptop with a core i7 processor,24 gb of memory and an SSD drive use SAS 9.4 on a Windows 8.1 Pro 64 bit platform. Here we can see that even with an environment not optimized for parallel processing, significant improvements can be had through threading.

Create a dataset:

```
%let lv = 25000;
%let uv = 90000;
data data.incomes;
     call streaminit(98765);
     do j = 1 to 1E8;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'John';  citycode=1; output;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'Jane';  citycode=1; output;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'Joe';   citycode=2; output;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'Jan';   citycode=2; output;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'Josh';  citycode=3; output;
      income = floor(&lv + (&uv-&lv)* rand("Uniform"));   /* U[a,b] */
        name = 'Jill';  citycode=3; output;
        /* The three people to find during mining */
        if j = 5E5 then do;
          name = 'James'; income = 103243; citycode=1; output;
        end;
        if j = 7E5 then do;
          name = 'Joan'; income = 233923; citycode=2; output;
        end;
        if j = 8E5 then do;
          name = 'Joyce'; income = 132443; citycode=3; output;
        end;
     end;
run;
```

Access the data

1. With DATA step
2. With DS2, no thread
3. Create a thread
4. Access data through the thread

1. DATA step

```
data resultsDS;
     set data.incomes;
     accept = 0;
     if      citycode = 1 and income > 100000 then accept = 1;
     else if citycode = 2 and income > 200000 then accept = 1;
     else if citycode = 3 and income > 120000 then accept = 1;
     if accept then output;
run;
```

2. DS2, no thread

```
proc ds2;
data resultsDS2 (overwrite=yes);
  method run();
     DCL int accept i;
     set data.incomes;
     accept = 0;
     if      citycode = 1 and income > 100000 then accept = 1;
     else if citycode = 2 and income > 200000 then accept = 1;
     else if citycode = 3 and income > 120000 then accept = 1;
     if accept then output;
  end;
run;
quit;
```

3. DS2, create thread

```
proc ds2;
thread score  /overwrite=yes;
  method run();
     DCL int accept i;
     set data.incomes;
     accept = 0;
     if      citycode = 1 and income > 100000 then accept = 1;
     else if citycode = 2 and income > 200000 then accept = 1;
     else if citycode = 3 and income > 120000 then accept = 1;
     if accept then output;
  end;
endthread;
run;
run;quit;
```

4. DS2, use thread

```
proc ds2;
data results1 /overwrite=yes;
   dcl thread score score;
   method run();
      set from score threads=1;
   end;
enddata;
run;
quit;
```

DECLARE the thread instance

NOTE-> ***set from <threadname> threads=n***

See Appendix A for the table of execution times. As a comparison, with 600,000,003 rows, using DS2 with 8 threads the program ran in about 24 seconds while the equivalent DATA step ran in about 2 minutes From this we can see that although threads are most effective when used in a threaded multi-processing environment, even in a BASE SAS environment they can provide performance gains.

DS2, PROC FCMP AND USER DEFINED FUNCTIONS

Prior to SAS 9.3, if a BASE SAS programmer wanted to create a user defined function, there were two alternatives:

1. In a DATA step, use LINK/RETURN structure.
2. Use SAS macro functions.

Although both work, each has its drawbacks. The LINK/RETURN structure is written entirely in DATA step code, however if the “function” was to be used in another DATA step, the code had to be copied. This would lead to maintenance issues. Macro functions are more portable, but they can be difficult to write properly, and more difficult to maintain by programmers not proficient in SAS macro language. The introduction of PROC FCMP meant that programmers could use the familiar DATA step language to create portable functions; for more information on creating and using functions from PROC FCMP see Eberhardt 2009, 2010.

If you have a built investment in PROC FCMP, or you will be building functions for use both in DS2 and in the DATA step, you can access these functions in DS2. To do so, you use a SAS provided FCMP package. The basis syntax of the package is:

```
package fcmpCnv /
        language='fcmp'
        table='fcmp.conv'
        overwrite=YES ;
run;
```

1. Create a package

2. an FCMP package

3. Location of FCMP code

Creating FCMP Functions

```
proc fcmp outlib=fcmp.conv.base ;
```

1. Specify output library

2. Define a function

```
FUNCTION C_to_F(C) ;
  /* convert degrees fahrenheit to degrees celsius */
  return (32 + (C *  (9. / 5.)));
ENDSUB;
/* more functions */
run;
quit;
```

To access the functions in a DATA step or PROC SQL, you need to first tell SAS where the functions are stored, then access them:

```
/* tell SAS where to find your functions */
options cmplib=fcmp.conv;                    3. Location of FCMP code.
                                                2 level
data testDS_1;
    drop i;
    format inch cm 6.2;
   do inch = 0 to 100 by 0.5;
     cm = in_to_cm(inch);                    4. Invoke the function
     output;
   end;
run;
```

The equivalent DS2 code is:

```
proc ds2;                               1. Create a package
package fcmpCnv /                       2. An FCMP package
        language='fcmp'
        table='fcmp.conv'               3. Location of FCMP code
        overwrite=YES
       ;
run;

data testDS2_1 (overwrite=YES);

DECLARE double inch cm having format 6.2;

declare package fcmpCnv cnv();
```

```
  method init();
   do inch = 0 to 100 by 0.5;
      cm = cnv.in_to_cm(inch);          4. Invoke the function
      output;
   end;
  end;
enddata;

run;
quit;
```

Note that when you are invoking FCMP to create functions you specify a three level output (e.g. ***fcmp.conv.base***), but when accessing the functions through the DS2 package you only specify the two level table (e.g. ***fcmp.conv***).

Of course, if you need a function library that will be used exclusively by DS2, then you should build the functions as DS2 methods and save them in a permanent library.

DS2 AND THE HASH OBJECT

When SAS introduced the Hash Object to the DATA step, it was the beginning of an object oriented style of programming in the DATA step. For details on using the HASH object see Eberhardt 2010. When creating and using a HASH object in DS2 you have more flexibility in how you write the code, particularly in declaring the object; however, the fundamental use of the HASH has remained the same. Here we will show one way to create and use a HASH object.

In general, a HASH object should only be defined/declared one time. In a DATA step programme this was commonly done using and ***if _n_ = 1*** structure; this ensures the code is executed only one time on the first iteration of the DATA step. With DS2 we have the ***METHOD init()*** which is executed one time for each DATA programme; putting the HASH declaration in this method immediately provides clearer and more robust code.

In the DATA step, all arguments to the HASH methods are character strings; that is, if the variable ID is to be the key to the HASH, the defineKEY() method of the transaction object would look like:

```
transaction.defineKEY(‘ID’)
```

In DS2 we can also use a character string:

```
transaction.defineKEY(‘ID’)
```

We can also use the variables as arguments.

```
transaction.KEYS([ID])
```

In the following example, we will use this variable list method to create the HASH object.

To create and use the HASH object we need to use a SAS supplied HASH package. The five steps to create a HASH object are:

1. DECLARE a HASH package
   - There are three ways to declare the package (see the documentation for complete list of arguments for each type)
     - **Partial** – The key and data variables are defined by method calls. The optional parameters that provide the initialization data can be specified either in the DECLARE PACKAGE statement as shown above, in the _NEW_ operator, by method calls, or a combination of any of these. A single DEFINEDONE method call completes the definition.
     - **Complete key and data -** The key and data variables are defined in the DECLARE PACKAGE statement, which indicates that the instance should be created as completely defined. No additional initialization data can be specified with subsequent method calls.
     - **Complete key only hash-** The key and data variables are defined in the DECLARE PACKAGE statement, which indicates that the instance should be created as completely defined. No additional initialization data can be specified with subsequent method calls.
   - Our example uses the partial type
     - declare package hash h_mtdSales(**8**, '', 'A','','','');

2. Define the keys
   - If you use either of the complete types, the keys are defined as part of the package declaration. Our example uses the partial declaration do we use method calls to define the key
     - ***h_mtdSales.keys([salesID saleCat]);***

3. Define the data
   - If you use either of the complete types, the keys are defined as part of the package declaration. Our example uses the partial declaration do we use method calls to define the key
     - ***h_mtdSales.data(***[salesID saleCat mtdSales mtdCommission]***);***

4. Complete the declaration
     - ***h_mtdSales.defineDone();***

5. Add data to the HASH
   - In our example we will be adding data using the .ADD() method. It is more common to load data from a dataset into the HASH
     - ***h_mtdSales.ADD();***

The example we are using was taken from Kaufmann 2014 with some minor changes; Kaufman's example shows many good uses not only of the HASH object, but also good programming practices for DS2. In the spirit of code re-use, we are using his example; for a full description of the problem and code, see Kaufman 2014. The basic purpose of the programme is to calculate commissions at an individual sales rep level based on month to date sales of products in different categories where different categories get different commission rates. See Appendix B for the compete source code.

WHEN TO USE DS2

There are many technical reasons that would compel you to learn and use DS2, but possibly the most important reason is DS2 allows you to modular, robust, and reusable code.

Of course there are technical features in DS2 that will compel you to start implementing your solutions in DS2, most important among these are:

- precision
    - DS2 has several new data types that will allow greater precision in calculation.
- reusable methods
    - PROC FCMP brought the ability to write our own functions in DATA step language. These functions are still available to us.
    - Packages and methods extend the ability of FCMP functions into the object oriented realm,
- SQL
    - SQL code can now be submitted directly
- offload processing
    - When you have access to High-Performance Analytics technologies such as Grid Manager, SAS In Database, and SAS Embedded Process capable databases
- threading
    - if you work in threaded processing environments that are configured either as a Symmetric Multiprocessing (SMP) environment, or a Massively Parallel Processing (MPP) environment engineered specifically to support high-volume parallel processing and high-performance computing.

As noted above, even if no technical reason will compel you to move to DS2, the ability to write more robust programs should compel you.

WHAT'S MISSING

The current implementation of DS2 does not support the input statement; this means programs which read raw data and create data tables are not candidates for DS2. In our case this is a disappointment since we have several programs which create tables from raw data files and many these programs would benefit from DS2. We tried tricking DS2 by creating a data set view, but DS2 was not tricked.

DS2 has a rich variety of data types. Base SAS tables still only support fixed length character and double precision numeric variables. If your DS2 programs will be accessing RDBMS data you will be better able to match the data types in your DS2 programs. If you are using SAS data tables, you can use the new data types in your programs, but when the tables are saved the variables are converted to fixed length character or double precision numeric as appropriate. Be sure you read and understand the SAS rules for data type conversion.

MERGE. We so have the SET statement but MERGE is not available (although MERGE is listed as one of the reserved DS2 words). The basic functionality of a merger statement can be accomplished in SQL, however a common validation/cleaning process we use looks something like:

```
data inBoth inOne inTwo;
   merge  one (in=in1) two (in=in2);
   by j;
   if      in1 and in2 then do; /* more statements */ output inBoth; end;
   else if in1         then  do; /* more statements */ output inOne; end;
   else                     do; /* more statements */ output inTwo; end;
  end;
run;
```

this allows us to process the incoming data once and create several output tables.

MISSING VALUES

DS2 still supports SAS missing values. The SQL concept of null has been added to DS2 in addition to the usual SAS missing values. If you deal with missing values you will have to understand how DS2 treats SAS missing values and null values. In DS2, only the two base SAS data types (CHAR, DOUBLE) can have missing values while all the new data types can have null values. DS2 provides a NULL() function to test for a null value and a MISSING() function to test for a SAS missing value or a null value. In addition, DS2 provides two modes for processing null data: ANSI mode and SAS mode. In SAS mode missing values read from a table are handled as SAS missing values; in ANSI mode all SAS numeric missing values are converted to null. If you make use of SAS special missing values (._, .A - .Z) then you will need to process in SAS mode otherwise they will all be converted to null and you will lose information. SAS character variables are considered missing if the character string contains blanks; in ANSI mode this would not be converted to null, rather it is just left as a blank string. The reference manual has six pages dealing with the null/missing values so the description here is simplified but sufficient for most uses.

DATES

For many SAS programmers, and particularly new programmers, SAS dates are usually a source of confusion. With DS2 we now have date data types and these data types are not compatible with SAS dates. To use one of the new date data types with the SAS date functions you need to cast the date variable to a double using the to_double() function:

```
ageDays = intck('day', to_double(patientDOB), to_double(servdate) );
```

When DS2 reads a variable from a SAS dataset that has a date format, it converts it to DS2 date variable thus making it necessary to cast the variables.

CONCLUSION

This has been a quick overview of the DS2 procedure and its main components. Our experience has shown that the learning curve for DS2 is not steep, however there are two areas in which you initially need attention: data type conversion, and missing values. The use of methods and packages will make it easier to build more robust code. For programmers working in a threaded multi-processing environment and/or using SAS High-Performance Analytics technologies can take advantage of threaded processing. For many programmers there is not 'need' to move to DS2 programming, but the move will be worthwhile.

ACKNOWLEDGEMENT

We would like to acknowledge the help and encouragement of Craig Casper of Manitoba Health for his help and review of the exercise for the workshop.

REFERENCES

SAS® 9.4 DS2 Language Reference. SAS Institute Inc. 2013. Cary, NC: SAS Institute Inc.

Eberhardt, Peter "A Cup of Coffee and PROC FCMP: I Cannot Function Without Them", *Proceedings of the SAS Global Forum 2009 Conference*. Cary, NC: SAS Institute Inc.

Eberhardt, Peter "The SAS® HASH Object: Its Time to .find() Your Way Around", Proceedings of the SAS Global Forum 2010 Conference. Cary, NC: SAS Institute Inc.

Eberhardt, Peter "Functioning at an Advanced Level: PROC FCMP and PROC PROTO", *Proceedings of the SAS Global Forum 2010 Conference*. Cary, NC: SAS Institute Inc.

Dorfman, Paul and Eberhardt, Peter "Two Guys on HASH", *Proceedings of the South East SAS User 2011 Conference*.

Eberhardt, Peter and Yao Xue "I Object: SAS® Does Objects with DS2", *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc.

Kaufmann, Shaun " A Paradigm Shift: Complex Data Manipulations with DS2 and In-Memory Data Structures", *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc.

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

Peter Eberhardt
Fernwood Consulting Group
Toronto, ON, Canada
peter@fernwood.ca
www.fernwood.ca
Twitter: rkinRobin
WeChat: peterOnDroid

Xue Yao
Winnipeg Regional Health Authority
Winnipeg, MB, Canada
xueyao.statistic@gmail.com

For the example code and used in the Hands-On Workshop, email DS2@fernwood.ca

APPENDIX A – Thread Execution Results

60,000,003

	1 thread	2 threads	3 threads	4 threads	5 threads
real time	10.39	5.25	3.61	3.07	2.67
user cpu time	10.81	10.79	11.23	12.64	13.78
system cpu time seconds	0.62	0.40	0.29	0.34	0.35
memory	4081.90k	3822.40k	3876.76k	4096.17k	4403.56k
OS Memory	23536.00k	23536.00k	23536.00k	23792.00k	24304.00k

600,000,003

real time	01:44.9	52.03	36.71	31.02	27.93
user cpu time	01:50.8	01:48.8	01:55.5	02:10.1	02:26.1
system cpu time seconds	9.82	3.10	3.31	3.06	3.57
memory	3806.34k	3818.65k	3952.15k	4241.81k	4406.37k
OS Memory	23792.00k	23792.00k	24048.00k	24304.00k	24560.00k

60,000,003

	6 threads	7 threads	8 threads	No Thread	Data Step
real time	2.44	2.25	2.22	10.01	12.59
user cpu time	14.92	16.20	16.85	10.57	12.46
system cpu time seconds	0.43	0.35	0.40	0.32	0.12
memory	4882.75k	4915.43k	5221.84k	3288.87k	494.56k
OS Memory	24816.00k	25072.00k	25584.00k	23536.00k	23280.00k

600,000,003

real time	25.69	24.08	23.50	01:39.8	02:02.5
user cpu time	02:41.0	02:55.8	03:02.3	01:45.3	02:00.4
system cpu time seconds	3.70	3.25	4.01	4.67	2.09
memory	4703.03k	4986.75k	5147.62k	3287.12k	494.78k
OS Memory	24816.00k	25072.00k	25328.00k	23536.00k	23280.00k

APPENDIX B – HASH EXAMPLE

```
proc ds2;
data _null_;
*------------------------------------------------------;
* the arguments are positional:                        ;
*    1: hashexp - number of hash buckets,               ;
*    2: datasource - a table name or SQL expression   ;
*    3: ordered    - 'A', 'Y', 'D', 'N'                ;
*                 - can be spelled out eg ASCENDING  ;
*                 - if empty ('') then 'N'          ;
*    4: duplicate  - how do deal with duplicate keys  ;
*    5: suminc     - variable to hold summary count   ;
*                 - of hash package keys           ;
*    6: multidata  - indicator to allow duplicate key ;
*------------------------------------------------------;
*------------------------------------------------------;
* in the call beloe the values are                     ;
*    1: hashexp    - 8                              ;
*    2: datasource - '' not specified                  ;
*    3: ordered    - 'A' order keys in Ascending      ;
*    4: duplicate  - '' first key value is kept       ;
*    5: suminc     - '' no variable                  ;
*    6: multidata  - '' duplicate keys not allowded   ;
*------------------------------------------------------;

 declare package hash h_mtdSales(8, '', 'A','','','');

 declare float mtdSales       having format comma18.2;
 declare float mtdCommission   having format comma18.2;
 declare float commissionFactor;
 declare float commissionAmt;
 declare integer rc;

*--------------------------------------------------------;
* since the INIT() method gets called only once        ;
* it simplifies the DECLARE                            ;
*   -- no need for if _n_ = 1                          ;
*--------------------------------------------------------;
```

```
 method init();
*-------------------------------------------------------;
* the KEYS/DATA methods are used to define the key.data ;
*  unlike the DEFINEKEY()/DEFINEDATA() methods        ;
*  these methods take the actual variable            ;
*  NOT a string with the variable name               ;
*-------------------------------------------------------;

*-------------------------------------------------------;
* the KEYS/DATA methods can take a list of variabkes    ;
*-------------------------------------------------------;
  h_mtdSales.keys([salesID saleCat]);
  h_mtdSales.data([salesID saleCat mtdSales mtdCommission] );
  h_mtdSales.defineDone();
 end;

*-------------------------------------------------------;
* since we did not specify a data source in the DECLARE ;
* we will use the .find() and .add() methods          ;
*  -- first see id the salesID and saleCat are in hash  ;
*     if not, initialize data items and add to hash    ;
*     if found the PDV values are updated by the hash   ;
*-------------------------------------------------------;
method getmtdSales();
  rc = h_mtdSales.find();
  if rc <> 0 then
   do;
    mtdSales      = 0;
    commissionAmt = 0;
    mtdCommission = 0;
    h_mtdSales.add();
   end;
end;

*-------------------------------------------------------;
* we want to add up sales and commissions by ID and cat ;
*  -- the getmtdSales() method would give is the       ;
*     the current running total                      ;
* -- here we add the amounts from the data            ;
*   and update the hash using .replace()             ;
*--------------------------------------------------------;
method updatemtdSales();
  mtdSales = mtdSales + saleAmt;
  mtdCommission = mtdCommission + commissionAmt;
  h_mtdSales.replace();
end;

*--------------------------------------------------------;
* computational methods                            ;
*--------------------------------------------------------;
method getCommission_Factor();
  if saleCat = 1
  then
   do;
    if mtdSales < 700000
      then commissionFactor = 0.02;
      else commissionFactor = 0.03;
   end;

  else if saleCat = 2
  then
   do;
    if mtdSales < 890000
      then commissionFactor = 0.01;
      else commissionFactor = 0.015;
   end;
end;

 method Calculate_Commission();
  commissionAmt = saleAmt * commissionFactor;
 end;

*--------------------------------------------------------;
* the run() method simply calls each of the computation ;
* methods to do the work                           ;
* init() and term() are called automatically          ;
*--------------------------------------------------------;
```

```
 method run();
  set data.salesDEC;
  getmtdSales();
  getCommission_Factor();
  Calculate_Commission();
  updatemtdSales();
 end;

*-------------------------------------------------------;
* once all the rows have been read, write out the      ;
* the contents of the hash object to a dataset         ;
* since we need the key values in the dataset          ;
* there were also part of the data values              ;
*-------------------------------------------------------;
 method term();
   h_mtdSales.output('commissions');
 end;

 enddata;
run;
quit;
```