Metadata: SAS Tools and Techniques

SAS provides a variety of tools for discovering and using metadata about SAS data sets, variables, and other SAS objects (formats, macro variables, options, etc.). These tools include

- DATA step functions
- %SYSFUNC for calling any DATA step function in a macro context
- The dictionary tables

Because of the utility of %SYSFUNC in taking advantage of the other SAS tools discussed in this chapter, this chapter will also provide valuable extensions to your SAS macro knowledge.

Speaking of SAS macros, I would say that being able to take advantage of the tools described in this chapter is second only to comfort with the macro facility in making you a capable writer of flexible and smart SAS applications.

DATA step functions for determining metadata about variables in the current data set

Within a DATA step, the following functions (among others) can be used to find out characteristics of variables in the current data set:

VTYPE	returns the type (N or C) of the variable
VLABEL	returns the label of the variable (also see CALL LABEL)
VLENGTH	returns the storage length of the variable (as opposed to the actual length)

Each of these functions takes a variable name as an argument.

If you add an X to the end of any of these function names (for example, VTYPEX), the argument can be an expression that yields a variable name.

Example 1: Current data set - determine a variable's type, conditionally apply a format

This first piece of code uses the VTYPE function to find a variable's type.

```
DATA cars;
     SET bios511.cars2011;
     reltype = VTYPE(reliability);
     PUT reltype= ;
     STOP;   * try the code with and without this statement;
RUN;
```

Why might this be of interest? For example, suppose we would like our program to be able to automatically decide whether a numeric or character format should be applied to a specific variable. We could use the value returned by VTYPE (N or C) in conjunction with some macro code to conditionally apply an appropriate format. The code below is somewhat artificial but shows the concept.

```
%macro applycondfmt;
DATA cars;
    SET bios511.cars2011;
    reltype=VTYPE(reliability);
    CALL SYMPUTX("mreltype",reltype);
    STOP;
RUN;
PROC PRINT DATA=bios511.cars2011(OBS=8);
    VAR make model reliability;
    %IF &mreltype=N %THEN
        FORMAT reliability WORDS30.;;
RUN;
%mend;
%applycondfmt
```

Obs	Make	Model	Reliability
1	Acura	MDX	two
2	Acura	RDX	one
3	Acura	RL	one
4	Acura	TL	one
5	Acura	TSX	three
6	Acura	ZDX	missing
7	Audi	A3	missing
8	Audi	A3	four

Example 2: Current data set - use VLABEL to transfer and modify a variable label

This more realistic example uses the VLABEL function to store the label of an existing variable and modify it to form the label of a derived variable.

In data set CARS2011, variable CurbWeight provides car weight in pounds. We would like to derive a new variable CurbWeightKG that provides car weight in kilograms, and we would like the variable label to automatically reflect that. We know that the label of the current variable includes the word *pounds*, and we want to substitute *kilograms*.

```
DATA cars;
    SET bios511.cars2011;
    CurbWeightKG=CurbWeight/2.2046;
    LENGTH newlabel $256;
    newlabel=TRANWRD(VLABEL(CurbWeight),"pounds","kilograms");
    CALL SYMPUTX("CWKGlabel",newlabel);
    DROP newlabel;
RUN;

DATA cars2;
    SET cars;
    LABEL CurbWeightKG="&CWKGlabel";
RUN;

PROC PRINT DATA=cars2(OBS=7) LABEL;
    VAR Make Model CurbWeight CurbWeightKG;
RUN;
```

Obs	Make	Model	Curb weight in pounds	Curb weight in kilograms
1	Acura	MDX	4595	2084.28
2	Acura	RDX	4015	1821.19
3	Acura	RL	4085	1852.94
4	Acura	TL	3705	1680.58
5	Acura	TSX	3440	1560.37
6	Acura	ZDX	4410	2000.36
7	Audi	A3	3318	1505.03

```
/* more efficient than creating a new data set just to change a variable label */
PROC DATASETS LIB=work;
    MODIFY cars;
        LABEL CurbWeightKG="&CWKGlabel";
RUN; QUIT;
```

DATA step functions for determining metadata about data sets and variables other than the current data set

The functions in the section above could be applied only to variables in the current data set. What if we want to be able to find out about data sets other than the current one? For example, we might want to be able to determine whether a certain data set exists.

Example 3: The EXIST function to see whether a data set exists

The EXIST function returns a 1 if the data set named as the argument exists (is known to the current SAS session) and 0 if it does not exist.

```
DATA _null_;
    rc = EXIST("sashelp.class");
    PUT rc= ;
    STOP;
RUN;
```

In a macro context, we can use the EXIST function to execute code conditionally based on a data set's existence. We'll cover more about %SYSFUNC shortly, but basically it allows you to call any DATA step function in a macro context rather than a DATA step. This is truly WOW functionality in terms of the power it can add to your SAS programming.

```
%macro doit(ds=);
    %IF %SYSFUNC(EXIST(&ds))=1 %THEN %DO;
        PROC PRINT DATA=&ds;
        RUN;
    %END;

    %ELSE %DO;
        DATA _null_;
            FILE PRINT;
            PUT "Data set &ds does not exist.";
        RUN;
    %END;
%mend;

* options mprint mlogic symbolgen;
%doit(ds=sashelp.class)
%doit(ds=class)
```

Example 4: Open data set – determine data set attributes, transfer a data set label

Once we know that a data set exists, we might want to find out other things about it or the variables it contains. We wrap these calls with OPEN and CLOSE function calls to open and close the data set respectively.

dsid = OPEN(“data set”);

call the functions, referencing the dsid as necessary

rc = CLOSE(dsid);

To determine the numeric and character attributes of a data set, we can use the ATTRN and ATTRC functions respectively. Numeric attributes include number of observations, number of variables, and creation date, while character attributes include the data set’s name, library, label, and sort variables. The DSNAME function can be used to find out the complete name of the open data set.

NumObs=ATTRN(dsid,’NOBS’);

NumVars=ATTRN(dsid,’NVARS’);

DSLabel = ATTRC(dsid,’LABEL’);

DSLib = ATTRC(dsid,’LIB’);

DSMemName = ATTRC(dsid,’MEM’);

DSCompleteName = DSNAME(dsid);

```
DATA _null_;
    dsid=OPEN('sashelp.class');
    DSLib=ATTRC(dsid,'LIB');
    DSName=ATTRC(dsid,'MEM');
    name=DSNAME(dsid);
    NumObs=ATTRN(dsid,'NOBS');
    PUT DSLib= DSNAME= name= NumObs= ;
    rc=CLOSE(dsid);
RUN;
```

The PUT statement results in the following being printed in the SAS log:

```
DSLib=SASHELP DSName=CLASS name=SASHELP.CLASS.DATA NumObs=19
```

A nice and real application of these functions might be to transfer a data set's label to a new data set, which doesn't happen automatically with use of a SET statement.

```
DATA _null_;
    dsid=OPEN('bios511.cars2011');
    LENGTH dslabel $256;
    dslabel=ATTRC(dsid,'LABEL');
    rc = CLOSE(dsid);
    CALL SYMPUTX("dslabel",dslabel);
RUN;

DATA cars(LABEL="&dslabel");
    SET bios511.cars2011;
RUN;
```

Example 5: Open data set – determine variable characteristics such as type

We can also determine characteristics of the variables in the "open" data set, much like the VTYPE, VLABEL, and VLENGTH functions mentioned in the previous section. These functions include VARTYPE, VARLABEL, VARLEN, and VARFMT, and the dsid of the data set of interest is passed in as the initial argument. The second parameter to these functions is a variable number, which can be obtained with the VARNUM function.

In practice, the primary value of these functions is providing information for future decision-making based on variable characteristics. Typically, we need the information returned by the function to be in a macro variable to be useful for that decision-making.

```
DATA _null_;
    dsid=OPEN('sashelp.class');
    SexType=VARTYPE(dsid,VARNUM(dsid,'sex'));
    PUT sextype= ;
    CALL SYMPUTX('sexvartype',sextype);
    rc = CLOSE(dsid);
RUN;
%PUT &sexvartype;
```

SAS log after submitting this code:

```
19   data _null_;
20       dsid=open('sashelp.class');
21       SexType=vartype(dsid,varnum(dsid,'sex'));
22       put sextype= ;
23       call symputx('sexvartype',sextype);
24       rc = close(dsid);
25   run;

SexType=C
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.00 seconds

26   %put &sexvartype;
C
```

Example 6: The CEXIST function to see whether a user-defined format exists

Another useful "existence" function is CEXIST. The C here stands for "catalog", where a catalog is an object that SAS uses to store many different types of items. Items stored in catalogs are called "entries", and catalog entries have a four-level name: libref.catalogname.entryname.entrytype.

Usually you don't need to know about catalogs and catalog entries, but occasionally the knowledge can be useful. The CEXIST function can be used to check whether a particular catalog entry exists; it returns a 1 if the entry exists, 0 if the entry does not exist. The only type of catalog entry that you have probably encountered in your SAS work so far is a user-defined format, and in all likelihood you did not even know that user-defined formats were stored in catalogs! But yes, they are.

So the point of these few paragraphs is the following: What if you want to know whether a certain user-defined format exists? You can use the CEXIST function to find out, assuming you know what the format would be named. By default, formats defined in the current SAS session are stored in the WORK.FORMATS catalog, and they have the entrytype FORMAT (for a numeric format) or FORMATC (for a character format).

So if you wanted to find out whether a numeric format named AGEGRP had been defined in the current SAS session, you could use code like this, and of course you could write the return code to a macro variable for later use in your program.

```
DATA _null_;
    rc = CEXIST('WORK.FORMATS.AGEGRP.FORMAT');
    PUT rc= ;
    CALL SYMPUTX("isAgeGrpformat",rc);
RUN;
```

A limitation: You might think of structuring your CEXIST code in this way:

```
if CEXIST('WORK.FORMATS.AGEGRP.FORMAT') then x = PUT(AGE,AGEGRP.);
```

but don't do it. Why not? If the AGEGRP format does not exist, then you get a semantic error for the PUT part of the statement at COMPILE time, while CEXIST is evaluated at the EXECUTION stage.

How many observations are in my data set?

Now we have all the tools needed for several methods of learning how many observations are in a certain data set. If you have an application where this would be helpful, choose the method that makes the most sense to you or fits best with other methods you are using in your program.

Method 1: DATA step loop counting technique as in BIOS 511

```
DATA count1;
    SET sashelp.class END=eof;
    RETAIN nobs 0;
    nobs=nobs+1;
    IF eof THEN CALL SYMPUTX("nobs1",nobs);
RUN;
%PUT &nobs1;
```

Method 2: Using the NOBS option on a SET statement

This is an older, traditional method of finding out how many observations are in a data set, where you can think of NOBS= as being in the same family of options as END= . The STOP; statement can be used to keep all of the DATA step loops after the first from executing – they aren't needed for NOBS to return an accurate value. STOP; isn't absolutely needed here, but it adds efficiency to your program.

```
DATA count2;
    IF 0 THEN SET sashelp.class NOBS=nobs;
    CALL SYMPUTX("nobs2",nobs);
    /*  equivalent to   CALL SYMPUT("nobs2",PUT(nobs,BEST.));   */
    STOP;
RUN;
%PUT &nobs2;
```

Method 3: Using the ATTRN function

```
DATA _null_;
    dsid=OPEN("sashelp.class");
    nobs=ATTRN(dsid,'NOBS');
    CALL SYMPUTX("nobs3",nobs);
    rc = CLOSE(dsid);
RUN;
%PUT &nobs3;
```

We can generalize this code nicely with a macro.

```
%macro countnobs(ds= );
    DATA _null_;
```

```
        %IF %SYSFUNC(EXIST(&ds))=1 %THEN %DO;
            dsid=OPEN("&ds");
            nobs=ATTRN(dsid,'NOBS');
            CALL SYMPUTX("nobs3",nobs);
            rc = CLOSE(dsid);
        %END;
    RUN;
    %PUT &nobs3;
%mend;
%countnobs(ds=sashelp.class)
```

Method 4: Using PROC SQL and the automatic SQLOBS macro variable

Macro variable SQLOBS is automatically created when you run a query and contains the number of rows returned by the query. &SQLOBS is available even after you quit SQL and will reflect the last SELECT. But see the note below and be very careful when relying on SQLOBS.

```
PROC SQL NOPRINT;
    CREATE TABLE class AS SELECT * FROM sashelp.class;
%PUT &sqlobs;

PROC SQL;
    SELECT * FROM sashelp.class;
%PUT &sqlobs;
QUIT;
```

Note: Sometimes if you use the NOPRINT option and do a SELECT without a CREATE TABLE, &SQLOBS will contain the value 1 rather than the record count you expect. So be very careful when using &SQLOBS, and always check your results. For more information, see http://analytics.ncsu.edu/sesug/2012/BB-03.pdf (John Bentley, Leveraging SQL Return Codes When Querying Relational Databases).

Method 5: Using PROC SQL and COUNT(*)

Of course, PROC SQL's COUNT(*) will return the number of rows in the current data set.

```
PROC SQL NOPRINT;
    SELECT COUNT(*) INTO :nobs4 FROM sashelp.class;
QUIT;
%put &nobs4;
```

Note: It is possible to have a data set with 0 observations. Method 1 is the only method among the above that will not correctly return a 0 in that situation.

More about %SYSFUNC

The %SYSFUNC macro function can be used anywhere in your SAS code to call a DATA step function. As I said earlier, WOW! The steps for success with %SYSFUNC are the following:

1. Write the function call as usual – as you would write it in a DATA step.
2. Wrap the call with %SYSFUNC( ).
3. Remove the quotes around any arguments – they aren't needed in the macro context.

For example, in a DATA step you would write

rc = EXIST('sashelp.class');

With %SYSFUNC, you would write

rc = %SYSFUNC(EXIST(sashelp.class));

The value returned by a function called through %SYSFUNC is the same as the value returned in a DATA step call.

A nice use of %SYSFUNC would be to improve the date presented in our standard footnote. Recall that &SYSDATE returns values such as 23OCT13.

```
footnote "Job run on %sysfunc(TODAY(),worddate18.)";
proc print data=sashelp.class; run;
```

The footnote here would be

```
Job run on October 23, 2013
```

Another interesting use of %SYSFUNC is to store yet another kind of metadata, which is data about options in your current SAS session. The GETOPTION function can be used to find out such information – you simply pass in an option name and the current value is returned. So the value returned by function call GETOPTION(LINESIZE) is the current value of the LINESIZE option in your SAS session.

In this example, say that we want to temporarily make the linesize smaller or larger than what's being used for the rest of the SAS session. After our special step, we want to return the linesize value to what it was before (you could say that we were cleaning up after ourselves).

```
* sysfunc with another type of metadata - about your SAS session (options
settings);
%let oldlinesize=%sysfunc(getoption(linesize));
%put &oldlinesize;
options linesize=64;
proc print data=bios511.cars2011(obs=10);
run;
options linesize=&oldlinesize;
```

The SAS system option MISSING is another one that we might want to control temporarily (for a particular display) but then restore to its previous value, as done in the second of the long PROC REPORT examples.

The next example shows using %SYSFUNC to call some character functions in a macro context rather than in a DATA step.

Notice that each function call needs its own %SYSFUNC wrapper. Also note that this code could definitely be improved, as shown by the results!

```
%macro indx(letter=);
      %if %sysfunc(index(%sysfunc(upcase(Kathy Roggenkamp)),&letter))>0 %then
%put Name contains the character &letter;
      %else %put Name does not contain the character &letter;
%mend;
%indx(letter=A)
%indx(letter=B)
%indx(letter=a)
```

Results:

```
275  %indx(letter=A)
Name contains the character A
276  %indx(letter=B)
Name does not contain the character B
277  %indx(letter=a)
Name does not contain the character a
```

<u>The Dictionary Tables</u>

The functions and techniques described above provide access to specific pieces of SAS metadata, like peeking into the header portion of a SAS data set to find out one attribute at a time. What are called the SAS dictionary tables give you access to everything – all SAS metadata – at once.

So what are the dictionary tables? I would describe them as a constantly-updated database of tables (you can think of them as data sets) that contain metadata about all SAS objects in your current SAS environment, whether you are in a SAS windowing session or doing a batch run: there are tables about all libraries, data sets (called tables in this context), variables (called columns in this context), formats, macro variables, etc.

You can access the dictionary tables in two ways. Typically programmers use PROC SQL to run queries on members of the DICTIONARY library (that's right, a 10-letter libref!); those members include DICTIONARY.COLUMNS and DICTIONARY.TABLES. The DICTIONARY library contents can ONLY be accessed with PROC SQL. Alternatively, without SQL you can use a set of views that live in the SASHELP library, where they are all the objects whose names start with V: SASHELP.VCOLUMN and SASHELP.VMEMBER, for example.

Here's a look at SASHELP.VMEMBER, which contains the same information as DICTIONARY.TABLES. This is simply PROC CONTENTS and PROC PRINT output.

SASHELP.VMEMBER (like DICTIONARY.TABLES)

The CONTENTS Procedure

Data Set Name	SASHELP.VMEMBER	**Observations**	.
Member Type	VIEW	**Variables**	7
Engine	SQLVIEW	**Indexes**	0
Created	Tuesday, May 24, 2011 02:49:16 PM	**Observation Length**	1115
Last Modified	Tuesday, May 24, 2011 02:49:16 PM	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	Default		
Encoding	Default		

Variables in Creation Order

#	Variable	Type	Len	Flags	Label
1	libname	Char	8	P--	Library Name
2	memname	Char	32	P--	Member Name
3	memtype	Char	8	P--	Member Type
4	dbms_memtype	Char	32	P--	DBMS Member Type
5	engine	Char	8	P--	Engine Name
6	index	Char	3	P--	Indexes
7	path	Char	1024	P--	Pathname

SASHELP.VMEMBER (like DICTIONARY.TABLES) – First 10 observations

Obs	libname	memname	memtype	dbms_memtype	engine	index	path
1	BIOS511	BASEBALL	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
2	BIOS511	BIRTHS	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
3	BIOS511	BP_CUTPOINTS	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
4	BIOS511	CANDY	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
5	BIOS511	CARS2011	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
6	BIOS511	CLASS	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
7	BIOS511	CLIN	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
8	BIOS511	CLIN3NEW	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
9	BIOS511	CLIN4	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA
10	BIOS511	CLIN4NEW	DATA		V9	no	C:\Users\Kathy\Dropbox\BIOS511\DATA

And here's a look at SASHELP.VCOLUMNS, which contains the same information as DICTIONARY.COLUMNS.

SASHELP.VCOLUMN (like DICTIONARY.COLUMNS)

The CONTENTS Procedure

Data Set Name	SASHELP.VCOLUMN	**Observations**	.
Member Type	VIEW	**Variables**	18
Engine	SQLVIEW	**Indexes**	0
Created	Tuesday, May 24, 2011 02:48:26 PM	**Observation Length**	523
Last Modified	Tuesday, May 24, 2011 02:48:26 PM	**Deleted Observations**	0

Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	Default		
Encoding	Default		

Variables in Creation Order

#	Variable	Type	Len	Flags	Label
1	libname	Char	8	P--	Library Name
2	memname	Char	32	P--	Member Name
3	memtype	Char	8	P--	Member Type
4	name	Char	32	P--	Column Name
5	type	Char	4	P--	Column Type
6	length	Num	8	P--	Column Length
7	npos	Num	8	P--	Column Position
8	varnum	Num	8	P--	Column Number in Table
9	label	Char	256	P--	Column Label
10	format	Char	49	P--	Column Format
11	informat	Char	49	P--	Column Informat
12	idxusage	Char	9	P--	Column Index Type
13	sortedby	Num	8	P--	Order in Key Sequence
14	xtype	Char	12	P--	Extended Type
15	notnull	Char	3	P--	Not NULL?
16	precision	Num	8	P--	Precision
17	scale	Num	8	P--	Scale
18	transcode	Char	3	P--	Transcoded?

SASHELP.VCOLUMN (like DICTIONARY.COLUMNS) - First 10 observations (partial)

Obs	libname	memname	memtype	name	type	length	npos	varnum	label	format
1	BIOS511	BASEBALL	DATA	name	char	18	136	1	Player's Name	
2	BIOS511	BASEBALL	DATA	team	char	12	154	2	Team at the end of 1986	
3	BIOS511	BASEBALL	DATA	no_atbat	num	8	0	3	Times at Bat in 1986	
4	BIOS511	BASEBALL	DATA	no_hits	num	8	8	4	Hits in 1986	

Obs	libname	memname	memtype	name	type	length	npos	varnum	label	format
5	BIOS511	BASEBALL	DATA	no_home	num	8	16	5	Home Runs in 1986	
6	BIOS511	BASEBALL	DATA	no_runs	num	8	24	6	Runs in 1986	
7	BIOS511	BASEBALL	DATA	no_rbi	num	8	32	7	RBIs in 1986	
8	BIOS511	BASEBALL	DATA	no_bb	num	8	40	8	Walks in 1986	
9	BIOS511	BASEBALL	DATA	yr_major	num	8	48	9	Years in the Major Leagues	
10	BIOS511	BASEBALL	DATA	cr_atbat	num	8	56	10	Career times at bat	

Note that all LIBNAME and MEMNAME values in the dictionary tables are UPPER-CASED, so if you are using WHERE clauses involving those, you must specify upper-cased values.

For a compact PROC CONTENTS-like list of variables, remember this code:

```
proc sql;
      describe table dictionary.columns;
quit;
```

This produces the following in the SAS log:

```
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   name char(32) label='Column Name',
   type char(4) label='Column Type',
   length num label='Column Length',
   npos num label='Column Position',
   varnum num label='Column Number in Table',
   label char(256) label='Column Label',
   format char(49) label='Column Format',
   informat char(49) label='Column Informat',
   idxusage char(9) label='Column Index Type',
   sortedby num label='Order in Key Sequence',
   xtype char(12) label='Extended Type',
   notnull char(3) label='Not NULL?',
   precision num label='Precision',
   scale num label='Scale',
   transcode char(3) label='Transcoded?'
```

Since the V data sets and dictionary tables are equivalent, the following produce basically the same output – lists of all variables in the CARS2011 data set, with all of their characteristics:

```
proc print data=sashelp.vcolumn;
      where libname='BIOS511' and memname='CARS2011';
run;

proc sql;
      select * from dictionary.columns
```

```
            where libname='BIOS511' and memname='CARS2011';
quit;
```

In these notes we'll query the dictionary tables rather than using the SASHELP.V… views, since most real SAS applications use the dictionary tables.

If you looked at Example 13 in the PROC SQL notes, you've already seen an example of using the dictionary tables to make a variable list.

Example 1: Make a macro variable that contains a list of all variables in a data set

```
proc sql noprint;
      select name into :varlist separated by ' '
            from dictionary.columns
            where libname='BIOS511' and memname='CARS2011';
quit;
%put &varlist;
```

```
Make Model Type BaseMSRP CityMPG HwyMPG Country Seating CurbWeight Reliability Satisfaction
OwnerCost5Years
```

This example shows that you can use an INTO clause to write to a macro variable, the name of which should be preceded by a colon (:). You can also specify a separator. In the code above, if you wanted to separate the variable names with commas rather than blanks, you could have written

```
      separated by ','
```

Example 2: Make a macro variable that contains a customized list of all character variables in a data set

The following code produces a list of all character variables in BIOS511.CARS2011, omitting variable Model.

```
proc sql noprint;
      select name into :charlist separated by ' '
            from dictionary.columns
            where libname='BIOS511' and memname='CARS2011' and
                  type='char' and upcase(name)^='MODEL';
quit;
%put &charlist;
```

```
Make Type Country
```

Example 3: Make a macro variable that contains a list of all date variables in a data set

This code works as long as data set creators have followed good practice and associated a reasonable date format, such as DATE9. or MMDDYY10., with every date variable. The following code produces a list of all date variables in METS.SAEA_669.

```
proc sql noprint;
    select name into :datelist separated by ' '
        from dictionary.columns
        where libname='METS' and memname='SAEA_669' and
              type='num' and
              (index(format,'DATE')>0 or index(format,'MMDDYY')>0);
quit;
%put &datelist;
```

```
SAEA0B  SAEA1  SAEA6  SAEA7  SAEA13A  SAEA14C  SAEA15A
```

Example 4: Making a copy of a data set with the variables in alphabetical order

```
proc sql noprint;
      select name into :alphalist separated by ','
            from dictionary.columns
            where libname='BIOS511' and memname='CARS2011'
            order by name;

      create table alphacars as
            select &alphalist
            from bios511.cars2011;
quit;

title 'First 8 obs of copy of cars2011 with variables in alphabetic order';
proc print data=alphacars(obs=8); run;
```

First 10 obs of copy of cars2011 with variables in alphabetic order

Obs	BaseMSRP	CityMPG	Country	CurbWeight	HwyMPG	Make	Model	OwnerCost5Years	Reliability	Satisfaction	Seating	Type
1	42930	16	Japan	4595	21	Acura	MDX	5	2	2	7	SUV
2	32895	19	Japan	4015	24	Acura	RDX	4	1	3	5	SUV
3	47200	17	Japan	4085	24	Acura	RL	5	1	3	5	Sedan
4	35605	20	Japan	3705	29	Acura	TL	3	1	2	5	Sedan
5	29610	22	Japan	3440	31	Acura	TSX	2	3	2	5	Sedan
6	46020	16	Japan	4410	23	Acura	ZDX	4	.	.	5	SUV
7	30250	30	Germany	3318	42	Audi	A3	.	.	.	5	Diesel Wagon
8	27270	21	Germany	3305	30	Audi	A3	2	4	.	5	Wagon

Example 5: Print the first five observations of every data set in a library

In this case, we'll check out the WORK library.

```
%macro print5;
      proc sql noprint;
            select memname into :dsns separated by ' '
                  from dictionary.tables
                  where libname="WORK";
      quit;

      %put &sqlobs data sets in WORK are &dsns;

      %if &sqlobs=0 %then %do;
            %put No data sets!;
            %goto bottom;
      %end;

      %let i=1;
      %do %until (%scan(&dsns,&i)= );
            %let dsn = %scan(&dsns,&i);

            proc print data=work.&dsn(obs=5);
                  title "First 5 observations of &dsn";
            run;

            %let i = %eval(&i+1);
      %end;

      %bottom: ;
%mend;

%print5
```

In the SAS log, this message is written:

```
3 data sets in WORK are ALPHACARS CARS CARS2
```

In the SAS output, we get this:

First 5 observations of ALPHACARS

Obs	BaseMSRP	CityMPG	Country	CurbWeight	HwyMPG	Make	Model	OwnerCost5Years	Reliability	Satisfaction	Seating	Type
1	42930	16	Japan	4595	21	Acura	MDX	5	2	2	7	SUV
2	32895	19	Japan	4015	24	Acura	RDX	4	1	3	5	SUV
3	47200	17	Japan	4085	24	Acura	RL	5	1	3	5	Sedan
4	35605	20	Japan	3705	29	Acura	TL	3	1	2	5	Sedan
5	29610	22	Japan	3440	31	Acura	TSX	2	3	2	5	Sedan

First 5 observations of CARS

Obs	Make	Model	Type	BaseMSRP	CityMPG	HwyMPG	Country	Seating	CurbWeight	Reliability	Satisfaction	OwnerCost5Years	CurbWeightKG
1	Acura	MDX	SUV	42930	16	21	Japan	7	4595	2	2	5	10130.14
2	Acura	RDX	SUV	32895	19	24	Japan	5	4015	1	3	4	8851.47
3	Acura	RL	Sedan	47200	17	24	Japan	5	4085	1	3	5	9005.79
4	Acura	TL	Sedan	35605	20	29	Japan	5	3705	1	2	3	8168.04
5	Acura	TSX	Sedan	29610	22	31	Japan	5	3440	3	2	2	7583.82

First 5 observations of CARS2

Obs	Make	Model	Type	BaseMSRP	CityMPG	HwyMPG	Country	Seating	CurbWeight	Reliability	Satisfaction	OwnerCost5Years	CurbWeightKG
1	Acura	MDX	SUV	42930	16	21	Japan	7	4595	2	2	5	10130.14
2	Acura	RDX	SUV	32895	19	24	Japan	5	4015	1	3	4	8851.47
3	Acura	RL	Sedan	47200	17	24	Japan	5	4085	1	3	5	9005.79
4	Acura	TL	Sedan	35605	20	29	Japan	5	3705	1	2	3	8168.04
5	Acura	TSX	Sedan	29610	22	31	Japan	5	3440	3	2	2	7583.82

Example 6: Put the label of a data set into a title

```
* putting a data set label into a title;
%macro printit(libname=, memname=);
proc sql noprint;
      select memlabel into :label
            from dictionary.tables
            where libname="&libname" and
                  memname="&memname" and
                  memtype in ('DATA','VIEW');
quit;

proc print data=&libname..&memname;
      title "Data set &libname..&memname - &label";
run;
%mend printit;

%printit(libname=BIOS511, memname=CARS2011)
```

Data set BIOS511.CARS2011 - 2011 cars available in the US - most data from Consumer Reports

Obs	Make	Model	Type	BaseMSRP	CityMPG	HwyMPG	Country	Seating	CurbWeight	Reliability	Satisfaction	OwnerCost5Years
1	Acura	MDX	SUV	42930	16	21	Japan	7	4595	2	2	5
2	Acura	RDX	SUV	32895	19	24	Japan	5	4015	1	3	4
3	Acura	RL	Sedan	47200	17	24	Japan	5	4085	1	3	5
4	Acura	TL	Sedan	35605	20	29	Japan	5	3705	1	2	3
5	Acura	TSX	Sedan	29610	22	31	Japan	5	3440	3	2	2
6	Acura	ZDX	SUV	46020	16	23	Japan	5	4410	.	.	4
7	Audi	A3	Diesel Wagon	30250	30	42	Germany	5	3318	.	.	.

Example 7: Reconstructing the LENGTH statement for a data set

Sometimes you would like to rearrange the order of variables in a data set. A LENGTH statement is a good way to do this reordering. However, writing a LENGTH statement yourself can be tedious and error-prone, especially if there are a lot of variables. Instead, we can have SAS construct a LENGTH statement, and then we can edit that for the order we want.

```
* reconstruct the LENGTH statement for a data set;
%macro makelen(libname=,dsn=);

      %let libname=%upcase(&libname);
      %let dsn=%upcase(&dsn);

      proc format;
            value $typ 'char'='$'
                       other =' ';
      run;

      proc sql noprint;
            select catx(' ',name,put(type,$typ1.),put(length,5.))
                  into :lenvar separated by ' '
                  from dictionary.columns
                  where libname="&libname" and memname="&dsn"
                  ;
      quit;

      %put LENGTH statement for &libname..&dsn is &lenvar;

%mend makelen;

%makelen(libname=bios511,dsn=cars2011)
```

```
LENGTH statement for BIOS511.CARS2011 is Make $ 13 Model $ 17 Type $ 18 BaseMSRP 8 CityMPG 8
HwyMPG 8 Country $ 13 Seating 8 CurbWeight 8 Reliability 8 Satisfaction 8 OwnerCost5Years 8
```

Note: In general, using a SELECT clause within a PROC SQL CREATE TABLE statement is the easiest and safest way that I know of to control the order of variables in a data set. Two non-SQL ways to control variable order are with a LENGTH statement, as described here, where the LENGTH statement would typically be positioned between the DATA and SET statements, or with a RETAIN statement. However, each of these methods has perils (RETAIN can have profound consequences if you misuse it, LENGTH requires you to supply lots of information), making SQL SELECT appealing.

Recommended references:

Abolafia, Jeff. What Would I Do Without PRO SQL and the Macro Language
http://www2.sas.com/proceedings/sugi30/031-30.pdf

Davis, Michael. You Could Look It Up: An introduction to SASHELP Dictionary Views
http://www2.sas.com/proceedings/sugi26/p017-26.pdf

DiIorio, Frank and Abolafia, Jeff. Dictionary Tables and Views: Essential Tools for Serious Applications http://www2.sas.com/proceedings/sugi29/237-29.pdf

Moriak, Chris. %SYSFUNC: A Macro Variable Can't Function Without It
http://www.nesug.org/proceedings/nesug02/ps/ps003.pdf

Worden, Jeanina. Skinny to Fat: In Search of the "Ideal" Dataset (illustrates using techniques of this chapter to solve a problem that arises with PROC TRANSPOSE)
http://www2.sas.com/proceedings/forum2007/162-2007.pdf

SAS note http://support.sas.com/kb/48/674.html (Sample 48674) provides a great example of the usefulness of the dictionary tables in meeting a specific need, here adding a suffix or prefix to each of a group of variable names.