PhUSE 2009

Paper TU06

Accessing SAS® metadata and using it to help us develop data-driven SAS programs

Iain Humphreys, PRA International, Reading, UK

ABSTRACT

There is an extensive range of metadata available to the SAS programmer: information not only about all of our SAS file entities (e.g. libraries, datasets, variables, etc.), but also the settings associated with our current SAS session, such as system options, titles, and macro variable values.

We can exploit this information to help us write programs which are more flexible, require less maintenance, and are less error prone.

This tutorial firstly introduces some of the metadata available to us; it then explains how the programmer can access this information, using SAS features such as Proc SQL, the Macro language, and SAS Component Language (SCL) functions. Examples are presented which illustrate both the principles and benefits of adopting a data-driven approach to some common programming tasks.

INTRODUCTION

Metadata is data dictionary information, i.e. data about data. We are all familiar with the CONTENTS procedure in SAS, which lists information about our SAS datasets, including the following:

- Dataset name
- Date of creation
- Number of observations and number of variables
- Whether the dataset is sorted, compressed, indexed
- Information for each variable
    - Position (sequential number) in the dataset
    - Variable name
    - Type
    - Length
    - Format and Informat
    - Label

Most of us are also aware that this information can be output to a SAS dataset, from where it can be accessed by the programmer and used to drive processing, for example:

- finding all datasets that contain a certain variable
- testing whether a dataset has any observations, and making subsequent actions conditional on this
- checking that variable type and length are consistent before merging or appending datasets, etc.

However, there is much more information available to the programmer. This information is held in data dictionary tables, and allows us to retrieve information about the libraries, datasets, external files, system options, titles, and macro variables associated with the current SAS session.

This information can help us to write more flexible code. For example, we may wish to carry out some sort of data conversion (e.g. increase the length of a character variable) in every dataset where a variable exists. One approach might be to look in all of our datasets to find out where the variable is present, and then type in all of the relevant dataset names into a program which carries out the required conversion.

However, this approach is laborious, error-prone, and requires maintenance (e.g. if new datasets are subsequently added to our library that may also require conversion).

By making use of **metadata**, we can write syntax in which, in this example, we do not need to hard-code the dataset names: the program can find out for itself which datasets require conversion, and process each one accordingly. This is a **data-driven** program.

METADATA AVAILABLE TO THE SAS PROGRAMMER

DICTIONARY TABLES

Dictionary tables are special read-only PROC SQL tables. SAS automatically issues the DICTIONARY libref, but these tables are only accessible through PROC SQL.

However, SAS also provides views based on these dictionary tables, in the SASHELP library, and these can be accessed by SAS procedures and in the data step.

Table 1 below lists some of the dictionary tables available and their equivalent SASHELP view:

Dictionary Table Name	Information contained	Corresponding SASHELP view
DICTIONARY.CATALOGS	SAS catalogs and their entries	SASHELP.VCATALG
DICTIONARY.COLUMNS	SAS dataset variables and attributes	SASHELP.VCOLUMN
DICTIONARY.DICTIONARIES	SAS dictionary tables and columns	SASHELP.VDCTNRY
DICTIONARY.EXTFILES	Filerefs and the path to the external file location	SASHELP.VEXTFL
DICTIONARY.FORMATS	Format libraries, catalogs, names and types	SASHELP.VFORMAT
DICTIONARY.GOPTIONS	SAS/Graph options and current settings	SASHELP.VGOPT
DICTIONARY.INDEXES	SAS dataset indexes	SASHELP.VINDEX
DICTIONARY.LIBNAMES	Librefs and paths	SASHELP.VLIBNAM
DICTIONARY.MACROS	Macro variables and values	SASHELP.VMACRO
DICTIONARY.MEMBERS	SAS files, including tables, catalogs, views, item stores	SASHELP.VMEMBER
DICTIONARY.OPTIONS	System options and current settings	SASHELP.VOPTION
DICTIONARY.STYLES	Style templates and item stores	SASHELP.VSTYLE
DICTIONARY.TABLES	Tables (SAS data files and views)	SASHELP.VTABLE
DICTIONARY.TITLES	Title and footnote lines	SASHELP.VTITLE
DICTIONARY.VIEWS	SAS data views	SASHELP.VVIEW

Table 1: SAS data dictionary tables

ACCESSING DICTIONARY TABLES

We can use a PROC SQL query to access this information. For example, to see all currently active filerefs:

```
proc sql;
    select * from dictionary.extfiles;
quit;
```

This will list all columns (FILEREF and PATH in this example) and all rows in the table. To see which columns are available in each dictionary table, we can use the DESCRIBE TABLE statement, for example:

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

The table definition is written to the log. The definition for the dictionary table COLUMNS is displayed in the following panel; the definitions of the other dictionary tables listed in Table 1 are displayed in the appendix.

```
NOTE: SQL table DICTIONARY.COLUMNS was created like:

create table DICTIONARY.COLUMNS
  (
   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?'
  );
```

Dictionary tables are created at the start of each SAS session and dynamically maintained throughout by SAS.

Dictionary tables can be large, so once we have the definition available, we can specify individual columns on the SELECT statement, and add a subsetting WHERE clause, in order to obtain the specific information of interest, as in the example that follows:

SQL syntax:

```
proc sql;
    select memname from dictionary.columns
    where upcase(libname)='DERIVED' and upcase(name)='SID1A';
quit;
```

Output:

```
Member Name
--------------------------------
A_AEV
A_CMP
A_DAR
A_DYS
A_IDENT
A_SCR
A_UPD23
...
```

Equivalent data step syntax, using SASHELP view:

```
proc print data=sashelp.vcolumn noobs;
    var memname;
    where upcase(libname)='DERIVED' and upcase(name)='SID1A';
run;
```

Note: accessing dictionary information through SASHELP views is generally slower than using Proc SQL.

SOME RELEVANT PROGRAMMING TECHNIQUES FOR WORKING WITH METADATA

PROC SQL AND ITS INTERFACE TO THE SAS/MACRO FACILITY

Proc SQL is able to store values to macro variables, via the INTO clause of a SELECT statement.
General form:

PROC SQL;
  SELECT *col* <,*col*>…
  INTO :*macro-variable-specification*
    <, **:***macro-variable-specification*>…
  FROM *dataset*;
QUIT;

where *macro-variable-specification* is either

:*macro-variable* <**SEPARATED BY** '*character*' <**NOTRIM**>>;

or

:*macro-variable-1* - **:***macro-variable-n* <**NOTRIM**>;

If the query produces more than one row of output, the macro variable(s) will contain only the value(s) from the first row. If the query has no rows in its output, the macro variable is not modified, or, if it did not already exist, is not created. The automatic variable **&SQLOBS** contains the number of rows produced by the query.

The different forms of the syntax are illustrated in the following panels.

To create a macro variable based on values in the first row of a query result, just provide a single name for the macro variable (for each column selected), for example:

```
proc sql noprint;
     select patid, age
     into :pat, :age
     from derived.demo;

     select count(patid)
     into :npat
     from derived.demo;
quit;
```

To create one macro variable per row of a query result, code a range of macro variable names, with the hyphen (or keywords THROUGH, THRU), for example:

```
proc sql noprint;
     select distinct centre
     into :cen1 - :cen99
     from derived.demo;
quit;
```

This produces sequential macro variables from multiple rows. In this example, room is created for 99 macro variables, but only those required are actually created (one per centre).

To concatenate the values from one column into one macro variable, code the SEPARATED BY option with a delimiting character (e.g. space, comma, etc.), for example:

```
proc sql noprint;
     select distinct centre
     into :cens separated by ' '
     from derived.demo
     where highrec = 1;
quit;
```

This produces a single macro variable, containing a list of values (in this example, high recruiting centre numbers), each separated by a blank space.

CREATING A LIST OF ITEMS FOR PROCESSING

The ability to create 'lists' of items (either in the form of a single macro variable, or as a series of sequentially named macro variables, as in the last two examples above) gives us a powerful programming construct. Since we know from &SQLOBS how many items we have in the list, we can write SAS macro language loops to work through the list and process each item separately.

So, continuing on from the above example, we could produce a separate randomization listing for each high recruiting centre, by doing the following (within a macro):

```
%do i = 1 %to &sqlobs;
    %let cen = %scan(&cens,&i);
    proc print data = rand;
        where centre = &cen;
        title1 "Randomization listing for centre &cen";
    run;
%end;
```

Or, if unique laboratory parameter codes have been stored into a sequence of macro variables, e.g. &param1-&param99, we could use this list to produce a separate plot for each parameter:

```
%do i = 1 %to &sqlobs;
    data plot;
        set derived.lab (where=(parcode = "&&param&i"));
    ...
    run;
    proc gplot data = plot;
    ...
%end;
```

Therefore, if there is a subsequent change to our data (e.g. a centre recruits sufficient subjects to become a 'high recruiter', or further lab parameters are added) we do not have to edit our program. There is no hard coding of which centres are high recruiting, or which lab parameters are plotted – the programs are **data-driven**.

CALL EXECUTE

This is a data step routine, of the general form:

CALL EXECUTE(*argument*);

where *argument* is a quoted string. The SAS statements contained within the string are executed immediately after the data step containing the CALL EXECUTE, for example:

```
data _null_;
    call execute('proc print; run;');
run;
```

The Proc Print is executed immediately after the data step.

We can make a program data-driven by building the CALL EXECUTE argument based upon values in a dataset. For example, if we have a dataset LFT_PATS, containing patients with elevated liver function tests (LFTs), we can write dynamic SAS statements to fetch adverse event (AE) data specifically for these patients. The SAS log displays the statements generated by the CALL EXECUTE.

```
data _null_;
    set lft_pats end = lastone;
    if _n_ = 1 then do;
        call execute('data ae_lft;');
        call execute('set ae;');
        call execute('where patid in (');
    end;
    call execute(quote(patid));
    if lastone then do;
        call execute(');');
        call execute('run;');
    end;
run;
```

Log:

```
NOTE: CALL EXECUTE generated line.
1   + data ae_lft;
2   + set ae;
3   + where patid in (
4   + "1234"
5   + "2345"
6   + "7890"
7   + "8901"
8   + );
9   + run;
```

The code that is executed has been built according to our data, i.e. data-driven.

- This is often analogous to writing SAS statements to an external file, which is then %INCLUDE'd back into our program and executed.
- Since CALL EXECUTE can be executed conditionally, it can often be used where we would otherwise need %IF… (i.e. it can sometimes overcome the need to write a macro).
- The argument could be a macro call, for example:

```
data _null_;
    set demo;
    call execute('%profile(pno = '||patid||');');
run;
```

The macro %PROFILE is executed for every row in the DEMO dataset, with patient number passed as a parameter.

%SYSFUNC FUNCTIONS

The %SYSFUNC function allows access from within a SAS macro to most data step functions and several SAS Component Language (SCL) functions (SAS file I/O and SAS external file functions). This simplifies many common macro tasks, and gives us ready access to data about datasets, variables, external files, etc. In other words, they represent another means of accessing SAS **metadata**.

Functions available to %SYSFUNC

- Data step functions – all except the following:

DIF	DIM	HBOUND	INPUT	LAG
LBOUND	PUT	RESOLVE	SYMGET	

INPUTC / INPUTN and PUTC / PUTN are available instead of INPUT and PUT respectively.

- SCL functions: some useful SCL function are shown in Table 2 (not an exhaustive list):

Function name	Function action, Example
EXIST	Verifies the existence of a SAS member (dataset, catalog), returning a 1 or 0: `%if %sysfunc(exist(work.sae)) %then…`
OPEN	Opens a SAS dataset and returns a value (id). Many of the subsequent functions in this table use the id as an argument (and not the dataset name): `%let dsid = %sysfunc(open(crt.d_ae));`
CLOSE	Closes the dataset given by the id, and returns a value (0 if successful). Any dataset opened with the OPEN function should be closed with the CLOSE function: `%let rc = %sysfunc(close(&dsid));`
DSNAME	Returns the dataset name associated with a dataset id: `%let dsname = %sysfunc(dsname(&dsid));`
ATTRC	Returns the value of a character attribute for the dataset associated with the id: `%let att = %sysfunc(attrc(&dsid,`*`attrib`*`));` where *attrib* is (amongst others) sortedby - sort order if dataset is sorted (otherwise blank) label - dataset label mem - dataset name
ATTRN	Returns the value of a numeric attribute for the dataset associated with the id: `%let att = %sysfunc(attrn(&dsid,`*`attrib`*`));` where *attrib* is (amongst others) nobs - number of observations nvars - number of variables crdte - date created (SAS date time format)
VARNUM	Returns the number (position) of a variable in a SAS dataset. The following VARxxxx functions in this table use this number as an argument (and not the variable name). `%let vnum = %sysfunc(varnum(&dsid,`*`varname`*`));` where *varname* is the variable name.

VARFMT, VARINFMT	Returns the format (or informat) assigned to a SAS dataset variable:
	`%let fmt = %sysfunc(varfmt(&dsid,&vnum));`
	`Or %let fmt = %sysfunc(varfmt(&dsid,`
	`%sysfunc(varnum`
	`(&dsid,varname))));`

VARLABEL	Returns the label assigned to a SAS dataset variable: `%let lbl = %sysfunc(varlabel(&dsid,&vnum));`
VARLEN	Returns the length of a SAS dataset variable: `%let len = %sysfunc(varlen(&dsid,&vnum));`
VARNAME	Returns the name of a SAS dataset variable: `%let name = %sysfunc(varname(&dsid,&vnum));`
VARTYPE	Returns the type of a SAS dataset variable, C for character and N for numeric: `%let type = %sysfunc(vartype(&dsid,&vnum));`

Table 2 - Useful SAS SCL functions

Note: these functions are also available in the data step, in which case character strings should be enclosed in quotes as usual, for example:

```
data _null_;
    dsid = open('derived.ae');
    vn   = varnum(dsid,'patno');
    rc   = close(dsid);
    put dsid= vn= rc=;
run;
```

Other similar SAS functions exist (and can also be called in a macro via %SYSFUNC), for working with:

- Catalogs and catalog entries (CEXIST)
- External files (FILEEXIST, FOPEN, FCLOSE, etc.)
- Directories (DOPEN, DNUM, etc.)

APPLYING THE PROGRAMMING TECHNIQUES TO METADATA

The examples in following panels each illustrate how one or more of the programming techniques described in the previous section can be used in conjunction with SAS dictionary data to create data-driven programs. SASHELP views could be used instead of dictionary tables, for those preferring to work with the data step rather than Proc SQL. However, regardless of the method used to access the metadata, it is often its use within a macro that gives us increased flexibility and allows our data manipulations and reporting activities to be data-driven.

Example Application 1: Performing an operation (rename) on all variables in a dataset

General principles: Proc SQL is used to obtain a list from the data dictionary tables of the variable names in a given dataset. The SELECT…INTO…SEPARATED BY clause stores this list to a single macro variable. The automatic macro variable &SQLOBS tells us the number of items on the list, and this information is then used in a macro loop to perform the operation (in this example a rename) on each variable in turn.

Syntax:

```
* Make working copy of dataset;
    data aev;
        set derived.a_aev;
    run;
* Fetch list of variable names;
    proc sql noprint;
        select distinct name into :vnames separated by ' '
        from dictionary.columns
        where upcase(libname) = 'WORK' and upcase(memname) = 'AEV';
    quit;
* Macro to process list;
    %macro ren;
            proc datasets lib = work nolist nodetails;
              modify aev;
        %do i = 1 %to &sqlobs;
              rename %scan(&vnames,&i) = %upcase(StudyA_%scan(&vnames,&i));
        %end;
            run; quit;
    %mend;
    %ren;
```

Log (partial):

```
    NOTE: Renaming variable ACNTAK1N to STUDYA_ACNTAK1N.
    NOTE: Renaming variable ACNTAK2N to STUDYA_ACNTAK2N.
    NOTE: Renaming variable ACNTAK3N to STUDYA_ACNTAK3N.
    NOTE: Renaming variable ACNTAK4N to STUDYA_ACNTAK4N.
    ...
```

Example Application 2: Performing an operation on all variables of a particular type within a library

General principles: Proc SQL is used to obtain from the dictionary tables a list of datasets containing, in this example, character date variables. For each dataset on this list, a second list is obtained of the names of variables. Each of these will undergo a conversion to a numeric date. Both lists are in the form of a set of sequentially-numbered macro variables, plus a count of the number of items on the list. Macro loops are used to process these two lists, one nested within the other.

Syntax:

```
* Access dictionary data, creating a subset of relevant dictionary information;
  proc sql noprint;
       create table datevars as
       select memname, name, label, type
       from dictionary.columns
       where upcase(label) like '%DATE%'
         and upcase(libname)  = 'OUTDATA'
         and upcase(memname) ne 'TITLES'
         and upcase(memname) ne 'FORMATS'
         and upcase(type)     = 'CHAR'
         and upcase(name)    ne 'DCMDATE'
       order by memname,name;
     * Fetch number of datasets to process;
       select count(distinct memname) into :nset
       from datevars;
     * Create first list (dataset names);
       select distinct memname into :set1 - :set%left(%trim(&nset))
       from datevars;
  quit;

* Macro to process first list, creating and processing second list for each item ;
  %macro convert;
       %put NOTE: There are %trim(&nset) dataset(s) to process...;
       %* Loop to process first list;
       %do i = 1 %to &nset;
            %put NOTE: ...Processing dataset &&set&i.....;
                proc sql noprint;
                  select count(distinct name) into :ndat
                  from datevars
                  where upcase(memname) = "&&set&i";
            %* Create second list (date variable names);
                  select distinct name into :dat1 - :dat%left(%trim(&ndat))
                  from datevars
                  where upcase(memname) = "&&set&i";
                quit;
            %put NOTE: ...There are %trim(&ndat) date(s) to convert on &&set&i.....;

                data &&set&i;
                  set outdata.&&set&i;
            %* Loop to process 2nd list;
            %do j = 1 %to &ndat;
                %put NOTE: ......Converting date &&dat&j;
                  if length(&&dat&j)= 8 then &&dat&j..N = input(&&dat&j,yymmdd8.);
                  else if not missing(&&dat&j) then put
                  "CHECK: partial date found on dataset &&set&i - " PATID= &&dat&j=;
            %end;
                run;
       %end;
  %mend;
  %convert;
```

A new numeric version of each date is created, with a suffix of 'N' added to the variable name. Character dates of length 8 are assumed to be complete; incomplete dates are not converted and a message is issued to the log, shown overleaf.

Example Application 2 / continued

Log (partial):

```
NOTE: There are 6 dataset(s) to process...

NOTE: ...Processing dataset ADEV...
NOTE: ...There are 2 date(s) to convert on dataset ADEV...
NOTE: ......Converting date DTONSE
NOTE: ......Converting date DTSTOP
CHECK: partial date found on dataset ADEV - patid=010201 003 DTONSE=200311
CHECK: partial date found on dataset ADEV - patid=010201 003 DTSTOP=200311
...
...
NOTE: ...Processing dataset DEMO...
NOTE: ...There are 3 date(s) to convert on dataset DEMO...
NOTE: ......Converting date CSTDT
NOTE: ......Converting date DOB
NOTE: ......Converting date EAASDT
```

Example Application 3: Saving and restoring current title and footnote settings

General principles: Proc SQL is used to fetch the current title and footnote settings from the dictionary tables, and store these in a dataset. CALL EXECUTE can then be used to restore them later if required.

Syntax:

```
* Capture current titles and footnotes;
    proc sql;
        create table mytitles as
        select number,
               text,
               case
                 when type = 'T' then 'TITLE'
                 else 'FOOTNOTE'
               end as linetype
        from dictionary.titles;
    quit;

* Statements which alter or delete current titles and footnotes;
    ...;

* Restore original titles and footnotes;
    data _null_;
        set mytitles;
        call execute(compress(linetype||put(number,2.))||' '||trim(text)||';');
    run;
```

Log (partial):

```
NOTE: CALL EXECUTE generated line.
1    + TITLE1 Pharma Co.                    FIRST DRAFT                        20AUG2009;
2    + TITLE2 ABC-XYZ/123-456                                               Page ^{pageof};
3    + FOOTNOTE1 Program: t_demo.sas  Output: t_demo_all.rtf;
```

Example Application 4: Deleting global macro variables

General principles: Proc SQL is used to fetch global macro variable names from the dictionary tables, and store them in a dataset. This dataset can be subsequently read, and CALL EXECUTE used to issue calls to %SYMDEL to delete each macro variable (without having to code their individual names). This can be useful for clearing up at the end of a macro.

Syntax:

```
* Store global macro variables containing safety population counts to dataset;
    proc sql;
        create table gmacs as
        select distinct name from dictionary.macros
        where upcase(scope) = 'GLOBAL' and upcase(name) like 'SAF%';
    quit;
* Delete safety population macro variables;
    data _null_;
        set gmacs;
        call execute('%symdel '||trim(left(name))||';');
    run;
```

Example Application 5: Renaming variables in a dataset

General principles: %SYSFUNC is used within a macro to call SCL functions which access dataset information. This information (in this example the names of the dataset variables) is stored to a series of sequentially numbered macro variables. In turn this permits us to process these variables in a macro loop, renaming each one, without having to code its individual name. This could be used to add a suffix / prefix to the name, or, as in this example, convert the name to uppercase where necessary.

Syntax:

```
    %macro rename;
        %let dsid  = %sysfunc(open(outdata.adev));
        %let nvars = %sysfunc(attrn(&dsid,nvars));
        %do i = 1 %to &nvars;
            %let oldname&i = %sysfunc(varname(&dsid,&i));
            %let newname&i = %upcase(&&oldname&i);
        %end;
        %let rc = %sysfunc(close(&dsid));
    %* Only rename where new name is different to old name;
        proc datasets lib = outdata nolist nodetails;
            modify adev;
            rename
            %do i = 1 %to &nvars;
                %if &&oldname&i ne &&newname&i %then &&oldname&i = &&newname&i;
            %end;
            ;
        run; quit;
    %mend rename;
    %rename;
```

Log (partial):

```
NOTE: Renaming variable protocol to PROTOCOL.
NOTE: Renaming variable centre to CENTRE.
NOTE: Renaming variable patient to PATIENT.
NOTE: Renaming variable patid to PATID.

NOTE: MODIFY was successful for OUTDATA.ADEV.DATA.
```

Example Application 6: Building a list of all variables in a dataset

General principles: %SYSFUNC is used within a macro to call SCL functions which access dataset information. This time the information is used to build a list of variable names in a single macro variable, which is returned by calling the macro. The macro could subsequently be called, for example, in a KEEP or DROP statement.

Syntax:

```
%macro varlist(dsn);
    %let dsid = %sysfunc(open(&dsn));
    %let nvars = %sysfunc(attrn(&dsid,nvars));
    %let varlist=;
    %do i = 1 %to &nvars;
        %let varlist = &varlist %sysfunc(varname(&dsid,&i));
    %end;
    %let rc = %sysfunc(close(&dsid));
    &varlist
%mend varlist;

data unmatch_a (keep = %varlist(derived.a))
     unmatch_b (keep = %varlist(derived.b));
    merge derived.a (in = a)
          derived.b (in = b);
    by subjid;
    if a and not b then output unmatch_a;
    else if b and not a then output unmatch_b;
run;
```

The example shows the trapping of mismatched rows into separate datasets. In each case we only want to keep the original variables, and not those coming from the other dataset in the merge (which will have missing values in the output dataset anyway). The example shows how the macro call removes the need for us to determine all of the relevant variable names, and code them in a lengthy KEEP statement.

Example Application 7: Testing for dataset existence and number of observations; using this metadata to control program flow

General principles: %SYSFUNC is used within a macro to call SCL functions which access dataset information. This information is used to control the actions of, in this example, a reporting program. This permits the program to be run without errors when the expected dataset is empty, or even if it does not exist at all.

Syntax:

```
%macro genrpt(dsn=);
    %if %sysfunc(exist(&dsn)) = 1 %then %do;
        %let dsid = %sysfunc(open(&dsn));
        %let nobs = %sysfunc(attrn(&dsid,nobs));
        %if &nobs GT 0 %then %do;
            %put NOTE: &nobs obs will be reported from &dsn..;
            *<Usual reporting syntax here>;
        %end;
        %else %do;
            %put NOTE: 0 obs in &dsn - Producing placeholder report. ;
            *<Alternative reporting syntax here>;
        %end;
        %let rc = %sysfunc(close(&dsid));
    %end;
    %else %do;
        %put WARNING: Dataset &dsn does not exist.;
    %end;
%mend genrpt;
%genrpt(dsn=sae);
```

Example Application 8: Saving and restoring current SAS option settings

General principles: %SYSFUNC is used within a macro to call the GETOPTION function, storing the specified option setting to a macro variable. If subsequent statements alter the option setting, we can revert to the original setting by referencing the macro variable in an OPTIONS statement.

Syntax:

```
* Store current option settings to symbols;
    %let mprint = %sysfunc(getoption(mprint,keyword));
    %let pgsize = %sysfunc(getoption(ps,keyword));

* Statements that may alter options;
    ...;

* Restore original option settings;
    options &mprint &pgsize;
```

CONCLUSION

This paper has demonstrated how metadata, used in conjunction with a few relevant programming techniques, can yield efficiency gains for the SAS programmer. Much SAS code has been written in the pharmaceutical arena with a "write once, use once" mentality, and the wheel then reinvented with similar code written on the next project.

By adopting more of an application software developer's mindset, we can raise our game and exploit the metadata at our disposal to write more flexible, data-driven programs. This enhanced flexibility benefits us in several ways:

- there is less need to "hard-code" the names of SAS entities such as datasets and variables in our programs, and, in some instances, to even know their names;
- there is less chance of making an error while editing programs, especially when dealing with large numbers of entities, and therefore less re-work required;
- there is less maintenance required, since changes to our data libraries (e.g. the addition of a new variable, or the removal of a dataset) are reflected automatically in the dictionary information that now drives our programs;
- there is improved re-usability, through the creation of code that can be applied to common programming tasks across many projects.

Many of the concepts and SAS components described here have been with us for a number of years. However, aided by the emergence of data standards such as CIDSC, and with increasing pressure to produce our deliverables faster, the data-driven approach to programming offers us a useful tool in our quest to accelerate clinical development.

SOURCES OF INFORMATION AND RECOMMENDED READING

SAS Online Documentation (http://support.sas.com/onlinedoc/913)
SAS User Group Conference Proceedings (various) (http://support.sas.com/events/sasglobalforum/previous)
Phil Mason, In the Know…SAS Programming Tips & Techniques from Around the Globe, SAS Publishing (1996)
Rick Aster, Professional SAS Programming Shortcuts, Breakfast (2002)

CONTACT INFORMATION

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

Iain Humphreys
PRA International
Pacific House, Imperial Way
Reading, RG2 0TD, UK
Email: HumphreysIain@praintl.com
Web: www.praintl.com

APPENDIX

SAS DATA DICTIONARY TABLE DEFINITIONS

```
DICTIONARY.CATALOGS
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   objname char(32) label='Object Name',
   objtype char(8) label='Object Type',
   objdesc char(256) label='Object Description',
   created num format=DATETIME informat=DATETIME label='Date Created',
   modified num format=DATETIME informat=DATETIME label='Date Modified',
   alias char(32) label='Object Alias',
   level num label='Library Concatenation Level'
```

```
DICTIONARY.COLUMNS
   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?'
```

```
DICTIONARY.DICTIONARIES
   memname char(32) label='Member Name',
   memlabel char(256) label='Dataset Label',
   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'
```

```
DICTIONARY.EXTFILES
   fileref char(8) label='Fileref',
   xpath char(1024) label='Path Name',
   xengine char(8) label='Engine Name'
```

```
DICTIONARY.FORMATS
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   path char(1024) label='Path Name',
   objname char(32) label='Object Name',
   fmtname char(32) label='Format Name',
   fmttype char(1) label='Format Type',
   source char(1) label='Format Source',
   minw num label='Minimum Width',
   mind num label='Minimum Decimal Width',
   maxw num label='Maximum Width',
   maxd num label='Maximum Decimal Width',
   defw num label='Default Width',
   defd num label='Default Decimal Width'
```

```
DICTIONARY.GOPTIONS
   optname char(32) label='Option Name',
   opttype char(8) label='Option type',
   setting char(1024) label='Option Setting',
   optdesc char(160) label='Option Description',
   level char(8) label='Option Location',
   group char(32) label='Option Group'
```

```
DICTIONARY.INDEXES
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   name char(32) label='Column Name',
   idxusage char(9) label='Column Index Type',
   indxname char(32) label='Index Name',
   indxpos num label='Position of Column in Concatenated Key',
   nomiss char(3) label='Nomiss Option',
   unique char(3) label='Unique Option'
```

```
DICTIONARY.LIBNAMES
   libname char(8) label='Library Name',
   engine char(8) label='Engine Name',
   path char(1024) label='Path Name',
   level num label='Library Concatenation Level',
   fileformat char(8) label='Default File Format',
   readonly char(3) label='Read-only?',
   sequential char(3) label='Sequential?',
   sysdesc char(1024) label='System Information Description',
   sysname char(1024) label='System Information Name',
   sysvalue char(1024) label='System Information Value'
```

```
DICTIONARY.MACROS
   scope char(32) label='Macro Scope',
   name char(32) label='Macro Variable Name',
   offset num label='Offset into Macro Variable',
   value char(200) label='Macro Variable Value'
```

```
DICTIONARY.MEMBERS
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   dbms_memtype char(32) label='DBMS Member Type',
   engine char(8) label='Engine Name',
   index char(32) label='Indexes',
   path char(1024) label='Path Name'
```

```
DICTIONARY.OPTIONS
   optname char(32) label='Option Name',
   opttype char(8) label='Option type',
   setting char(1024) label='Option Setting',
   optdesc char(160) label='Option Description',
   level char(8) label='Option Location',
   group char(32) label='Option Group'
```

```
DICTIONARY.STYLES
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   style char(32) label='Style Name',
   crdate num format=DATETIME informat=DATETIME label='Date Created'
```

```
DICTIONARY.TABLES
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   dbms_memtype char(32) label='DBMS Member Type',
   memlabel char(256) label='Dataset Label',
   typemem char(8) label='Dataset Type',
   crdate num format=DATETIME informat=DATETIME label='Date Created',
   modate num format=DATETIME informat=DATETIME label='Date Modified',
   nobs num label='Number of Physical Observations',
   obslen num label='Observation Length',
   nvar num label='Number of Variables',
   protect char(3) label='Type of Password Protection',
   compress char(8) label='Compression Routine',
   encrypt char(8) label='Encryption',
   npage num label='Number of Pages',
   filesize num label='Size of File',
   pcompress num label='Percent Compression',
   reuse char(3) label='Reuse Space',
   bufsize num label='Bufsize',
   delobs num label='Number of Deleted Observations',
   nlobs num label='Number of Logical Observations',
   maxvar num label='Longest variable name',
   maxlabel num label='Longest label',
   maxgen num label='Maximum number of generations',
   gen num label='Generation number',
   attr char(3) label='Dataset Attributes',
   indxtype char(9) label='Type of Indexes',
   datarep char(32) label='Data Representation',
   sortname char(8) label='Name of Collating Sequence',
   sorttype char(4) label='Sorting Type',
   sortchar char(8) label='Charset Sorted By',
   reqvector char(24) format=$HEX48 informat=$HEX48 label='Requirements Vector',
   datarepname char(170) label='Data Representation Name',
   encoding char(256) label='Data Encoding',
   audit char(3) label='Audit Trail Active?',
   audit_before char(3) label='Audit Before Image?',
   audit_admin char(3) label='Audit Admin Image?',
   audit_error char(3) label='Audit Error Image?',
   audit_data char(3) label='Audit Data Image?'
```

```
DICTIONARY.TITLES
   type char(1) label='Title Location',
   number num label='Title Number',
   text char(256) label='Title Text'
```

```
DICTIONARY.VIEWS
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   engine char(8) label='Engine Name'
```