SESUG Paper 060-2021

Management of Metadata and Documentation When Your Data Base Structure is Fluid: What to do if Your Data Dictionary has a Varying Number of Variables

Louise S. Hadden, Abt Associates Inc.

ABSTRACT

A data dictionary for a file based on Electronic Medical Records (EMR) contains variables which represent an unknown number of COVID-19 tests for an unknown number of infants – there is no way to know in advance how many iterations of the COVID test variable will exist in the actual data file from medical entities. In addition, variables in this file may exist for three different groups (pregnant women, postpartum women, and infants), with PR, PP and IN prefixes, respectively. This presentation demonstrates how to ingest data dictionaries to collect metadata allowing identification of groups for processing, and drive label (and value label) description creation for iterated (and other) labels using SAS functions, PROC FORMAT, and data step processing, as well as other utilities.

INTRODUCTION

Documentation is a key product of any programming task. We use data dictionaries to drive much of our processing, using the metadata contained therein to drive the provision of variable and value labels, assign variable prefixes and split out subsets of files, as well as drive the creation of format assignment statements and reporting for file subsets. Data dictionaries are provided to study sites, who return extracted electronic medical record (EMR) data to us for processing.

There are two separate data dictionaries used for this project, for person level and visit level variables. The data dictionaries in use for the study have multiple tables and thousands of variables. Individual variables have up to two # signs (iteration flags), appear in a single row in the data dictionaries, and may occur more than 60 times. Variables to the right of the screenshot shown below indicate whether the variable is present for pregnant women, postpartum women, and/or infants. There are six data files delivered from each site – 2 data dictionaries x 3 populations. We need variable and value labels for each of the six data files, so we need to build label statements, a format library, and format assignments. We receive data on an ongoing basis (monthly), so the solution needs to be as efficient and data driven as possible. Iterated variables, which appear with pound signs in the data dictionaries, appear with numbers in the actual data and we need a programmatic answer to match the variables in the data dictionaries to the variables in the data files.

Solutions presented include ingesting data dictionaries to collect metadata allowing identification of groups for processing and drive processing with information derived from data dictionaries.

DATA DICTIONARY READ IN

We read in the separate tabs in the workbook in the data dictionary and collect information to be used for variable names, labels and value label, as well as other information used solely for data processing. Note the red # signs, the variable description which will be turned into a label, and the variable values which will be turned into value labels (a format) in Figure 1 shown below.

Challenges for read in include: multiple tabs in each Excel workbook: different starting rows in each tab, requiring reading in specific ranges; clean up of special characters (tabs, carriage returns), disaggregation of some fields, making sure errors encountered in read in are addressed; and the need to collect information on two levels (variable information vs value label information).

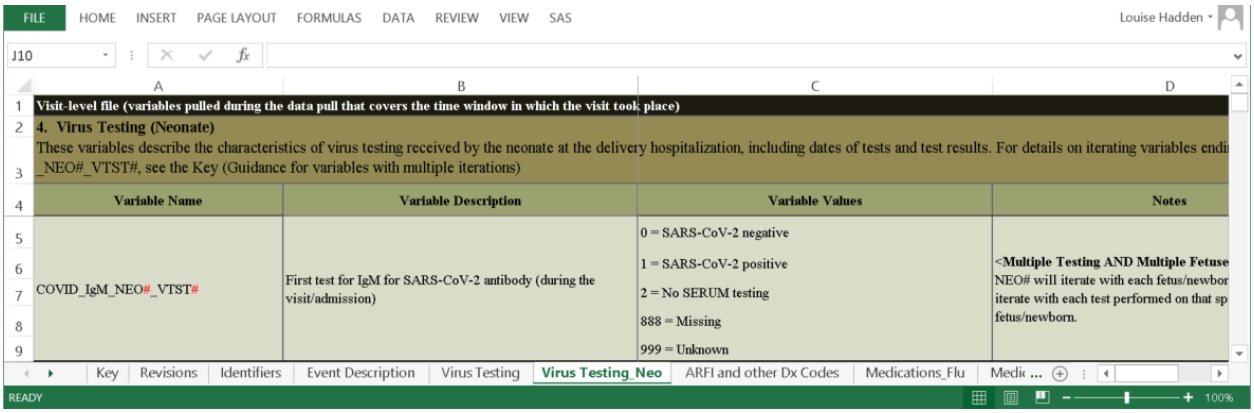


Figure 1. Sample Data Dictionary

Since multiple tabs are read in with the same structure on each tab, we take advantage of macro processing to read in our file metadata. We use ranges in our PROC IMPORT, storing valuable metadata about our incoming data. Note that it is possible to have two level range names, similar to library names, by preceding the range with the tabname and separated by a dot.

```
************************************************************;
*** Import Personal Data Dictionary one tab at a time    ***;
************************************************************;

%macro imptabs(tabn=1, tabnm=identifiers, intab=Identifiers, startrow=10, endcol=H);

       proc import dbms=xlsx out = temp datafile = " \file.xlsx" replace;
              RANGE="&intab.$A&startrow.:&endcol.999";
              getnames=YES;
run;

. . .
```

First, the LENGTH function is used to calculate the length of “variable”. The length of variable names is limited to 32 columns, and the name an of iterated variable may exceed the limits. The data dictionary is a living document, and if any overlong variables that exist once prefixes and iterated counts are added to the base, the spelling is adjusted. Identification variables are exempt from prefixes. When a file is first processed, variables, with the exception of ID variables, have prefixes added, using the CATS function. Additionally, label strings are created by concatenating prefixes and existing variable descriptions using the CATX function.

```
data labels&tabn.;
         length label labelstr $ 300 variable_type $ 8;
         set &tabnm (keep=variable_: pw_preg pw_pp inf
         where=(variable_name ne '' and variable_description ne ''));
              label=catx(": ","&tabnm.",variable_description);
              labelstr=cats(variable_name,'="',label,'"');

              variable_length=length(variable_name);
              length_flag=(variable_length+7 GT 32);
              label variable_length="Length of Variable"
             length_flag="Current Variable Length + 7 exceeds 32";
```

In preparation for iteration, we use the INDEXC function to find the location (or existence) of # signs. If we wanted to look for a string (as we do later on) we can use the INDEX function. We use COUNTC to count how many times an iteration flag occurs in a variable name. Multiple iterations of a variable can occur if, for example, multiple neonates in a single pregnancy have multiple virus tests.

```
            /* find out the # of iterations within a variable name */
            iteration_flag=(indexc(variable_name,'#') gt 0);
            iteration_count=countc(variable_name,'#');

            label iteration_flag="Binary: Variable iterations"
            iteration_count="# of iteration points within variable name";
run;

%mend;

%imptabs(tabn=1, tabnm=Identifiers, intab=Identifiers, startrow=4, endcol=H);
```

ITERATION

It is relatively simple to replace a single iterator, in this case, a #, in a variable name. It is more complicated to replace two or more iterators, especially if you do not know how many iterations there are. SAS functions process one variable transformation at a time – that is, they stop after completing a single operation on a string. After confirming the existence of a # sign in a variable and finding its position using the INDEX function, we then use the SUBSTR function to replace the # using a do loop, outputting additional label records for each iteration.

As noted above, we use the COUNTC function to discover how many #s exist in a variable name. You can use functions to discover the number of iterations needed as well in the actual data – including the REVERSE and ANYNUM functions – in the actual data. Additionally, the iteration numbers are added to the label strings using CAT functions. Multiple supplemental label records are created until no more # signs appear in the variable names.

We have thousands of variables, and multiple occurrences of iteration and the need to replace (via the SUBSTR function) items of different lengths. We quickly realized we would need to employ macro loops to handle the different requirements for a number of situations (number of iterations, the “base” of the variables needing to be iterated, one or two iteration symbols, and substr length.) Sample code for a simple loop and more complex loop follow below.

Simple loop

```
%macro do_list1(maxiter=1,suffix=neo);

%do i=1 %to &maxiter;

data iter&suffix.1_&i (drop=loc);
    length variable $ 50 labelstr $ 300;
    set formats0 (where=(count(variable,"#")=1 and
index(variable,"IDENTIFIER#")>0));

    *get the first indexed # location;
    loc=index(variable,"#");

    substr(variable,loc,1)="&i";
    labelstr=catt(labelstr," #&i");

run;

%END;

%MEND DO_LIST1;
```

Complex loop

```
%macro do_list2(maxiter=20,suffix=vtst);
```

```
%if &maxiter le 9 %then %do i=1 %to &maxiter;

data iter&suffix.1_&i (drop=loc);
    length variable $ 50 labelstr $ 300;
    set formats0 (where=(count(variable,"#")=1 and
index(variable,"VTST#")>0));

    *get the first indexed # location;
    loc=index(variable,"#");

    substr(variable,loc,1)="&i";
    labelstr=catt(labelstr," #&i");

run;

proc print data=iter&suffix.1_&i (obs=5) noobs;
    var variable labelstr;
run;

%END;

%if &maxiter gt 9 %then %do;

%do i=1 %to 9;

data iter&suffix.1_&i (drop=loc);
    length variable $ 50 labelstr $ 300;
    set formats0 (where=(count(variable,"#")=1 and
index(variable,"VTST#")>0));

    *get the first indexed # location;
    loc=index(variable,"#");

    substr(variable,loc,1)="&i";
    labelstr=catt(labelstr," #&i");

run;

proc print data=iter&suffix.1_&i (obs=5) noobs;
    var variable labelstr;
run;

%END;

%do i=10 %to &maxiter;

data iter&suffix.1_&i (drop=loc);
    length variable $ 50 labelstr $ 300;
    set formats0 (where=(count(variable,"#")=1 and
index(variable,"VTST#")>0));

    *get the first indexed # location;
    loc=index(variable,"#");

    substr(variable,loc,2)="&i";
    labelstr=catt(labelstr," #&i");
```

```
run;

proc print data=iter&suffix.1_&i (obs=5) noobs;
    var variable labelstr;
run;

%END;

%END;

%MEND DO_LIST2;
```

PRACTICAL APPLICATIONS

VARIABLE LABELS

The iteration techniques discussed above are employed in several different scenarios: data quality checks, creating variable labels, creating format assignment statements, driving range checks, and producing missingness reports. Below follow snippets of code to create a data driven variable label statement. label strings are created by concatenating prefixes and existing variable descriptions using the CATX function, The CATX function is used to add information as a prefix to the label, such as the month of data collection or the tab the variable came from in the data dictionary. The labelstr variable is a sentence that applies a variable label to a variable. When put out to a flat file, it can be included to label variables in a data set.

Assign a filename for the label statement:

```
filename label1 ".\&short._Labels.txt";
```

Create iterations of variables with # signs using macro loops described above:

```
%do_list1(maxiter=3,suffix=id);
%do_list2(maxiter=4,suffix=vtst); . . .
```

Add iterated records created by the do loops together:

```
data expand_labels;
    set iterid: itervtst: . . ._ ;
run;
```

Add iterated records to the records that did not require iteration:

```
data labels;
    length variable $ 32;
    set labels0 (where=(index(variable,"#")=0))
        expand_labels (where=(index(variable,"#")=0))
        ;
run;
```

Output the label statement:

```
data tolabel;
    retain VARIABLE_CATEGORY VARIABLE LABELSTR
           VARIABLE_TYPE VARIABLE_LENGTH
           PW_PREG PW_PP INF ITERATION_COUNT INLABELS INPOS NUM ;
    file label1 lrecl=400;
    set matchtest ( keep= VARIABLE_CATEGORY VARIABLE LABELSTR
                          VARIABLE_TYPE
```

```
                            VARIABLE_LENGTH PW_PREG PW_PP INF
                            PRIORITY_VARIABLE
                            MISSING_NOT_OK ITERATION_COUNT
                            INLABELS INPOS NUM DD_ORDER);
    by NUM;
    STATEMENT=compbl(cats(variable,'="',labelstr,'"'));
    if inlabels=1 and inpos=1 then put statement;
run;
```

Include the label statement:

```
filename label1 ".\&short._Labels.txt";
filename retain1 ".\&short._retain.txt";
run;

data &outfi. (label="Labeled &short");
    retain
        %include retain1;
    ;
    set &infi.;
    label
        %include label1;
    ;
run;
```

Figure 2 is a snippet of the text file included to produce variable labels.

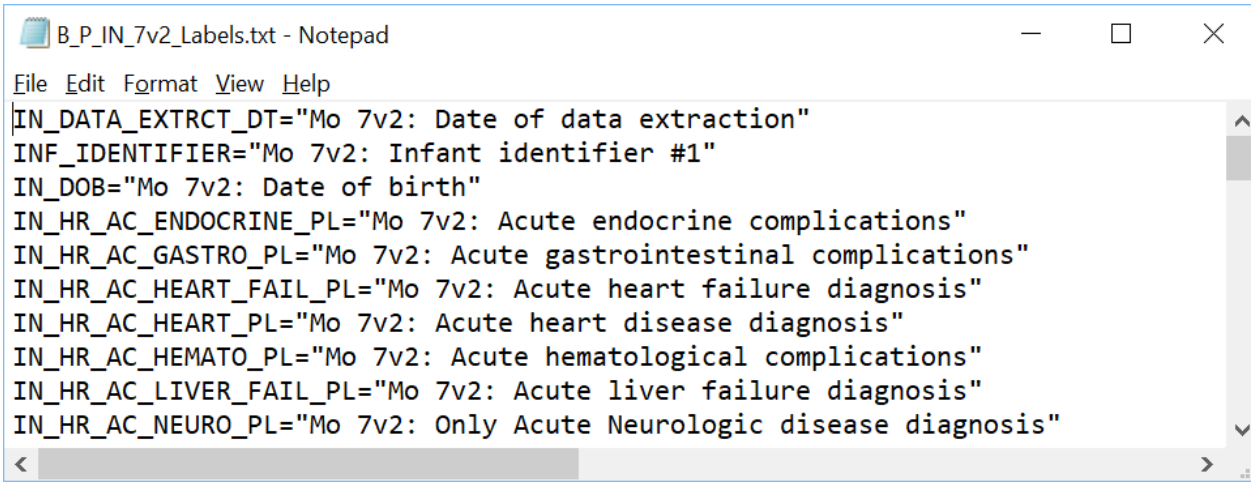
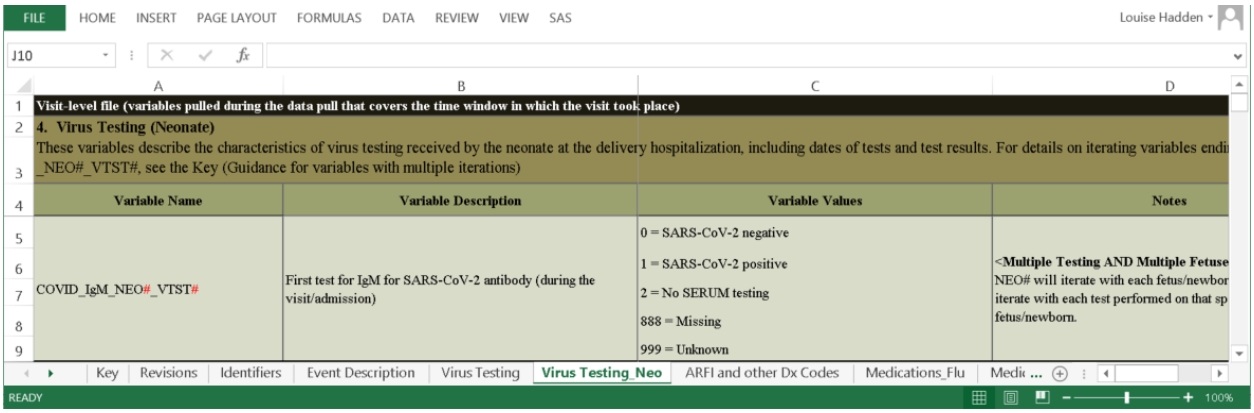


Figure 2. Sample Label Statement Text File

VALUE LABELS (FORMATS)

Let's review the data dictionary screenshot again, focusing on the variable values column and rows. Note that the variable name, variable description and notes fields are single rows, but the value labels are separate rows. Originally, the value labels were on single rows in the Excel data dictionaries but that made parsing the variable values field incredibly difficult so we redesigned the data dictionary to have a separate row for each variable value. When we read in the rows shown, we get 5 rows of data. Only the first line of data contains variable name, variable description, and notes, while each value label is on a separate row. To produce variable labels, we remove rows in blanks in the variable name field as we saw above, for value labels, we need to fill in those missing cells programmatically.




```
************************************************************;
*** Import Personal Data Dictionary one tab at a time    ***;
************************************************************;

%macro imptabs(tabn=1, tabnm=identifiers, intab=Identifiers, startrow=10, endcol=H);

proc import dbms=xlsx out = temp datafile = " \file.xlsx" replace;
               RANGE="&intab.$A&startrow.:&endcol.999";
               getnames=YES;
run;

data labels&tabn.;
    length variable_name $ 32 variable_values_edited varlabel $ 300 start $ 8
           variable_type $ 8 ;
    set &tabnm (keep=variable_: pw_preg pw_pp inf
        where=(variable_values ne '' or variable_name ne ''));

/* replace special characters such as tabs with blank and remove extraneous blanks */
    variable_values_edited=translate(variable_values,' ','09'x);
    variable_values_edited=translate(variable_values_edited,' ','0A'x);
    variable_values_edited=translate(variable_values_edited,' ','0D'x);
    variable_values_edited=compbl(variable_values_edited);

    /* create start and label variables for a start on building formats */
    if variable_type not in('ID','DATE') then do;
        start=scan(variable_values_edited,1,"=");
        varlabel=scan(variable_values_edited,2,"=");
    end;
```

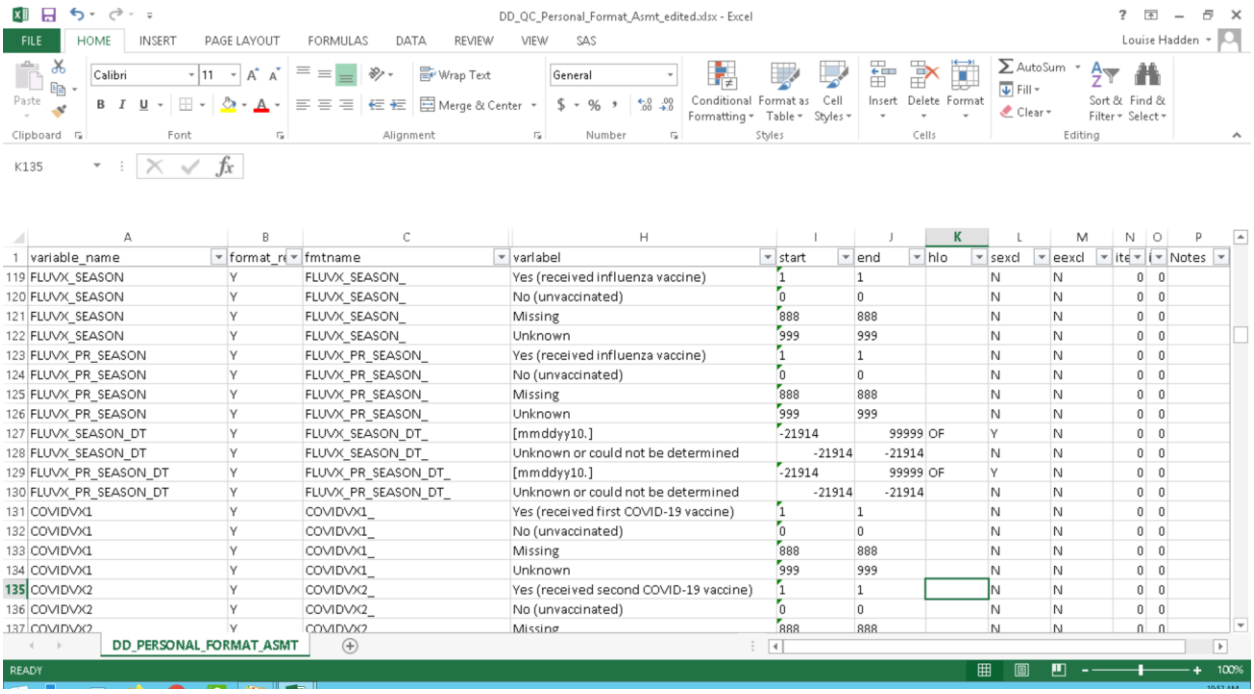
This is a similar read in as used for variable labels above, but there are key differences. For example, we are keeping records with either the variable name present OR variable value labels present. The reason for this is that some variables do not have formats assigned, or they have a blank line where a range should be specified, so if there is a variable name only, we keep the record. This will be addressed in a manual review step later, and if required, the data dictionary corrected.

In the code snippet below, we fill in the missing rows with the retain statement, and create a format name, among other things. We then export the temporary data set to a spreadsheet for manual checks and adjustments.

```
    retain _variable_type;

    if not missing(variable_type) then _variable_type=variable_type;
    else variable_type=_variable_type;
    drop _variable_type;
    formatstr=variable_values_edited;

    if variable_name ne '' then fmtname=cats(variable_name,'_');
```



	A	B	C	H	I	J	K	L	M	N	O	P
1	variable_name	format_re	fmtname	varlabel	start	end	hlo	sexcl	eexcl	ite	i	Notes
119	FLUVX_SEASON	Y	FLUVX_SEASON_	Yes (received influenza vaccine)	1	1		N	N	0	0	
120	FLUVX_SEASON	Y	FLUVX_SEASON_	No (unvaccinated)	0	0		N	N	0	0	
121	FLUVX_SEASON	Y	FLUVX_SEASON_	Missing	888	888		N	N	0	0	
122	FLUVX_SEASON	Y	FLUVX_SEASON_	Unknown	999	999		N	N	0	0	
123	FLUVX_PR_SEASON	Y	FLUVX_PR_SEASON_	Yes (received influenza vaccine)	1	1		N	N	0	0	
124	FLUVX_PR_SEASON	Y	FLUVX_PR_SEASON_	No (unvaccinated)	0	0		N	N	0	0	
125	FLUVX_PR_SEASON	Y	FLUVX_PR_SEASON_	Missing	888	888		N	N	0	0	
126	FLUVX_PR_SEASON	Y	FLUVX_PR_SEASON_	Unknown	999	999		N	N	0	0	
127	FLUVX_SEASON_DT	Y	FLUVX_SEASON_DT_	[mmddyy10.]	-21914	99999	OF	Y	N	0	0	
128	FLUVX_SEASON_DT	Y	FLUVX_SEASON_DT_	Unknown or could not be determined	-21914	-21914		N	N	0	0	
129	FLUVX_PR_SEASON_DT	Y	FLUVX_PR_SEASON_DT_	[mmddyy10.]	-21914	99999	OF	Y	N	0	0	
130	FLUVX_PR_SEASON_DT	Y	FLUVX_PR_SEASON_DT_	Unknown or could not be determined	-21914	-21914		N	N	0	0	
131	COVIDVX1	Y	COVIDVX1_	Yes (received first COVID-19 vaccine)	1	1		N	N	0	0	
132	COVIDVX1	Y	COVIDVX1_	No (unvaccinated)	0	0		N	N	0	0	
133	COVIDVX1	Y	COVIDVX1_	Missing	888	888		N	N	0	0	
134	COVIDVX1	Y	COVIDVX1_	Unknown	999	999		N	N	0	0	
135	COVIDVX2	Y	COVIDVX2_	Yes (received second COVID-19 vaccine)	1	1		N	N	0	0	
136	COVIDVX2	Y	COVIDVX2_	No (unvaccinated)	0	0		N	N	0	0	
137	COVIDVX2	Y	COVIDVX2_	Missing	888	888		N	N	0	0	

The screenshot above shows some (but not all) of the columns used to create complex formats from a data set. This spreadsheet output is reviewed carefully, any corrections made, and then it is imported into a SAS data set for use in creating formats and assignment statements.

BUILDING A FORMAT LIBRARY PROGRAMMATICALLY

We are keeping variable name in our SAS data set derived from the spreadsheet above, so that we can create format statements as well as a format catalog. For expediency, we name the format name with the variable name with a trailing underscore, stripping the iterator pound signs. To build a library, you need the following three fields:

FMTNAME - format name

LABEL - value label

START - start of a range or value

Additional fields that are used in our processing are:

Variable_name – used to build format assignment statements

Format_required – some variables do not require a format

END – end of a range

HLO – specialized formats – high, low, other

SEXCL (exclude the start of the range)

EEXCL (exclude the end of the range)

We have some complex formats for dates, ranges and nested formats which require END, HLO, SEXCL and EEXCL.

HOW DO FORMATS WORK?

The best way to figure out how formats work is to analyze them. The client for this project wanted to assign special date values for missing values. We create a small data set with the special dates and explore to see how this complex format looks in various forms. The same technique can be used to look

at ranges. What we are trying to achieve is an input data set which looks like what SAS expects under all conditions.

```
data temp;
    d1='01jan1900'd; d2='01jan1960'd; d3=today(); d4='01jan1940'd;
run;

proc print data=temp;
run;

proc print data=temp;
format d1 d2 d3 d4 mmddyy10.;
run;

proc format fmtlib;
     value foo '01jan1900'd='Invalid'
               '01jan1940'd='Still in'
               '01jan1960'd='SAS zero'
               other=[mmddyy10.];
run;

proc print data=temp;
format d1 d2 d3 d4 foo.;
run;

proc format cntlout=foo2;
run;

proc print data=foo2;
run;
```

Below we see the number representation of the special dates, followed by their formatted version (SAS data format), followed by our user-defined format.

```
Obs      d1      d2      d3       d4

  1     -21914     0    22475    -7305

Obs      d1            d2            d3            d4

  1     01/01/1900    01/01/1960    07/14/2021    01/01/1940

Obs      d1          d2           d3           d4

  1     Invalid    SAS zero    07/14/2021    Still in
```

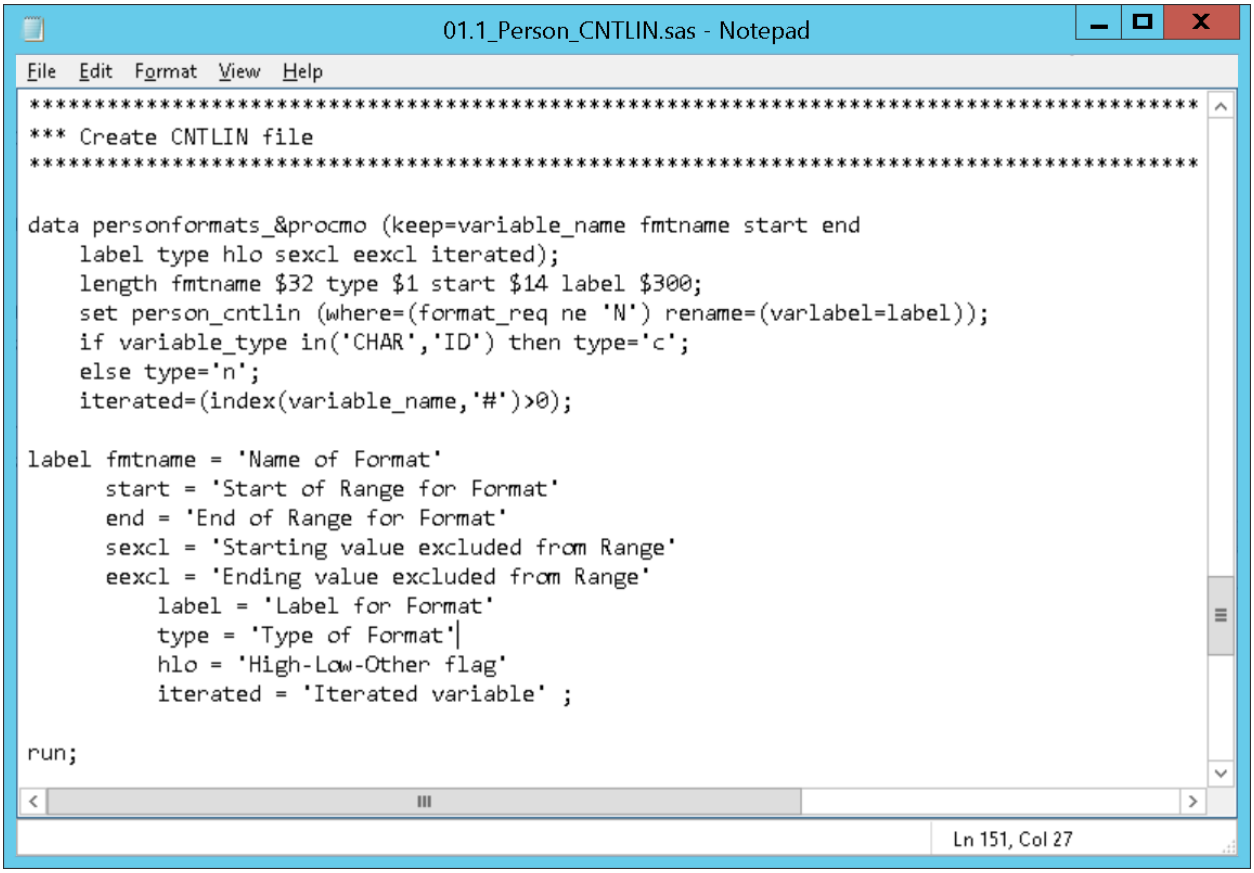
Below follows the result of PROC FMTLIB, showing how SAS represents the special date format in printed form. Note the other – this is a nested format indicating that any “other” dates should appear in MMDDYY10. Format. You can use any SAS-supplied or user created format as long as the program has access to where the format is stored.

```
-----------------------------------------------------------------------------
|       FORMAT NAME: FOO      LENGTH:   10   NUMBER OF VALUES:    4        |
|   MIN LENGTH:   1  MAX LENGTH:  40  DEFAULT LENGTH:  10  FUZZ: STD       |
|---------------------------------------------------------------------------|
|START           |END             |LABEL  (VER. V7|V8   14JUL2021:13:08:59)|
|----------------+----------------+-------------------------------------------|
|          -21914|          -21914|Invalid                                    |
|           -7305|           -7305|Still in                                   |
|               0|               0|SAS zero                                   |
|**OTHER**       |**OTHER**       |[MMDDYY10.]                                |
-----------------------------------------------------------------------------
```

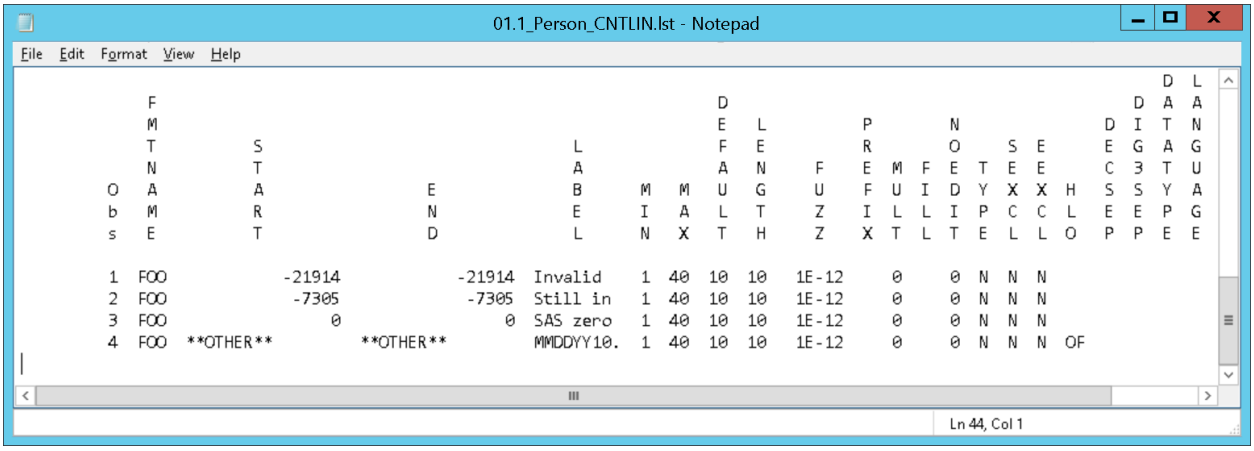
PROC FORMAT CNTLOUT produces a SAS data set from a format catalog file, which produces yet another vision of the same format. It is this version that we need to reproduce in order to use metadata to create format catalogs.

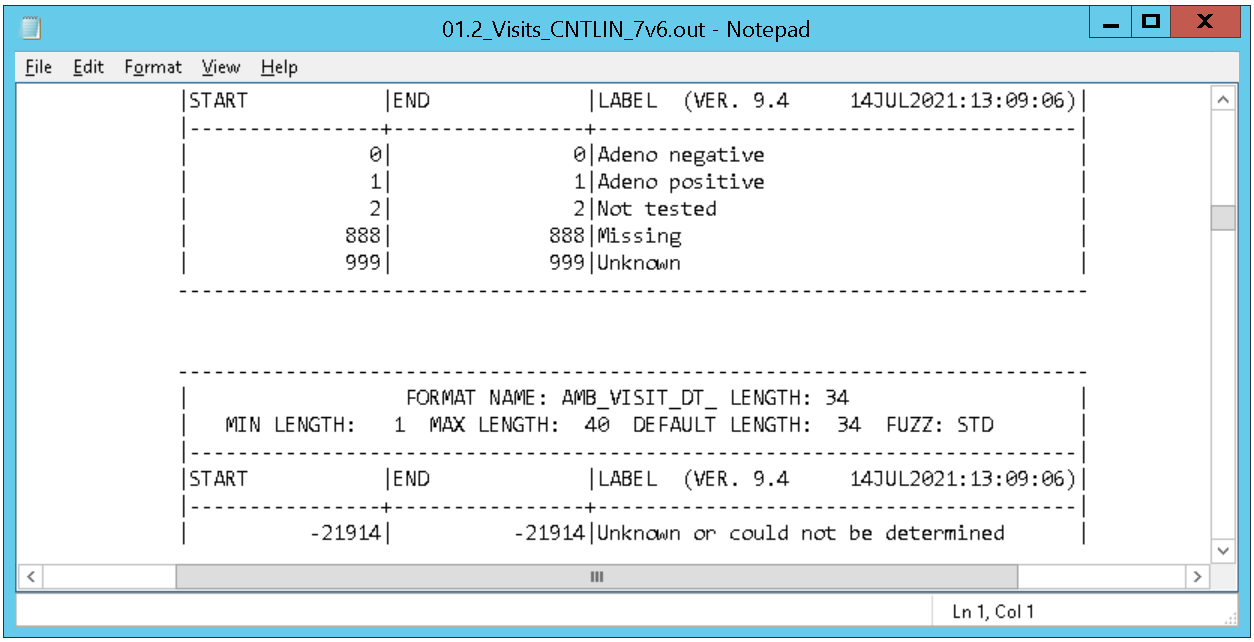
USING PROC FORMAT CNTLIN

Here we create the input to PROC FORMAT CNTLIN from our data dictionary import file, making adjustments to conform to SAS' requirements CNTLIN data sets.




```
proc format library=library.personformats cntlin=personformats_&procmo fmtlib ;
run;
```

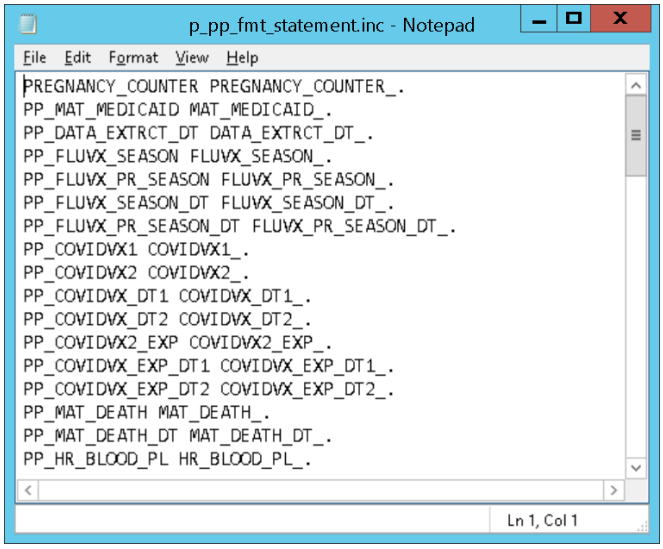




CREATING A FORMAT ASSIGNMENT STATEMENT PROGRAMMATICALLY

```
data fmtstm;
    length fmtstm $ 80 fmtdot $ 33 variable_name $ 32;
    file ".\&outfi._fmt_statement.inc" lrecl=80;
    set temp (where=(indata ne 0));
    fmtdot=cats(fmtname,'.');
    fmtstm=catx(' ',variable_name,fmtdot);
    put fmtstm;
run;
```

We use the same data set used to create the format catalogs / SAS data sets containing formats to generate format assignment statements. Note the use of SAS functions to add the dot following the format name, as this does not exist in the SAS data set. This format statements can be included in programs to analyze and characterize the data files. Below follows a screenshot of the format assignment file.



CONCLUSION

The same process of iteration and concatenation based on metadata elements is used to create macro calls to create a range report and a “missingness” report. We hope you’ll have some fun iterations with functions with your metadata as well!

Sample Range Check Report

	VARNUM	ANALVAR	LABEL	TYPE	LEN	V
2	1	STUDY_ID	Mo 7v2: Participant ID	Char	10	S
3	2	SITE	Mo 7v2: Sub-site or region	Char	32	T
4	3	INF_IDENTIFIER1	Mo 7v2: Infant identifier #1	Char	10	S
5	4	INF_IDENTIFIER2	Mo 7v2: Infant identifier #2	Char	10	S
6	5	INF_IDENTIFIER3	Mo 7v2: Infant identifier #3	Char	10	S
7	6	PREGNANCY_COUNTER	Mo 7v2: Counter indicating which pregnancy this is during the study	Num	8	1

Sample Missingness Report

Variable Name	Variable Description	# of Variable values	Missing Value Levels	Missing Value Levels
PR_ASSISTED_REP	Mo 7v2: Was the pregnancy a result of Assisted Reproduction?	3	1	2
PR_DATA_EXTRCT_DT	Mo 7v2: Date of data extraction	1	0	1
PR_FLUVX_SEASON	Mo 7v2: Current season influenza vaccination (August 1st 2020 to May 31st, 2021)	4	0	4
PR_FLUVX_PR_SEASON	Mo 7v2: Prior season influenza vaccination (August 1st 2019 to May 31st, 2020)	4	0	4
PR_FLUVX_SEASON_DT	Mo 7v2: Current season influenza vaccination date (August 1st 2020 to May 31st, 2021)	132	0	132
PR_FLUVX_PR_SEASON_DT	Mo 7v2: Prior season influenza vaccination date (August 1st 2019 to May 31st, 2020)	232	0	232
PR_COVIDVX1	Mo 7v2: First COVID-19 vaccination (if vaccine available)?	1	1	0
PR_COVIDVX2	Mo 7v2: Second COVID-19 vaccination (if vaccine available)?	3	1	2

ACKNOWLEDGEMENTS

This type of complex programming and processing is a team sport. I could not have created and implemented these techniques on my own. A very heartfelt thank you to team members Mary Juergens, Jenna Spirt, Nickolas Ferguson, Michael Duckworth and Peiyi Zhang.

CONTACT INFORMATION

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

Louise S. Hadden
Abt Associates Inc.
Louise_hadden@abtassoc.com

Any brand and product names are trademarks of their respective companies.