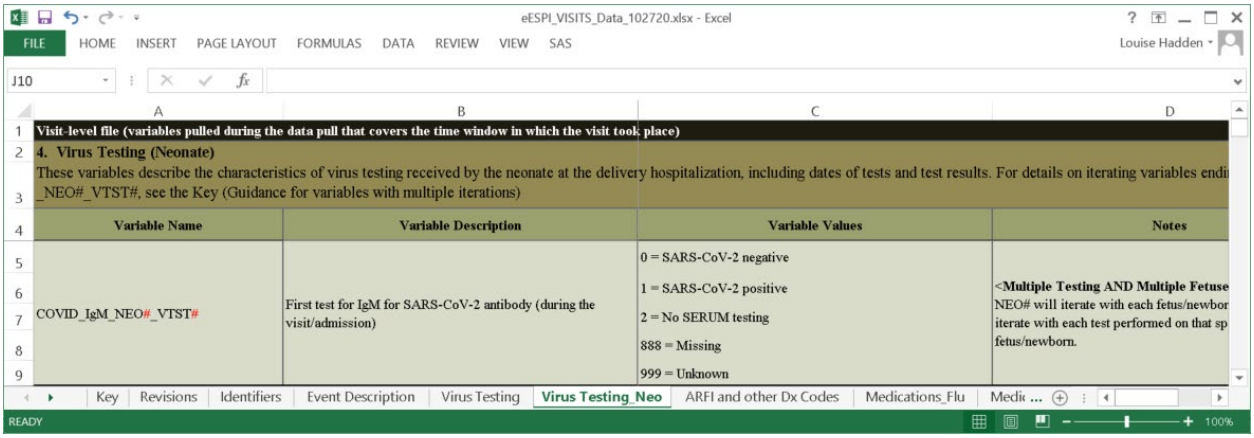
Weekly Tip for Nov. 3, 2020

Your data dictionary contains some variables that can iterate - for example, the unknown # of COVID tests for an unknown # of infants. How do you manage documentation?

A data dictionary for a file based on Electronic Medical Records (EMR) contains variables which represent an unknown number of COVID tests for an unknown number of infants – there is no way to know in advance how many iterations of this variable will exist in the actual data file. 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 Tuesday Tip demonstrates how to process such variables in a data dictionary to drive label (and value label) description creation for iterated (and other) labels using SAS functions, as well as other utilities.



Using PROC CONTENTS and ODS OUTPUT on an imported data dictionary (example shown above) and a data file from a health care entity, the position ODS OUTPUT object is created, and the column variable is standardized using the UPCASE function.

```
*************************************************************;
*** 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 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";

              /* 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);
```

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.

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. SAS functions process one variable transformation at a time – that is, they stop after completing a single operation on a string. We look for # in a variable and flag using the INDEX function. The INDEX function returns the position of the first pound sign. We then use the SUBSTR function to replace the # using a do loop, outputting additional label records for each iteration. Please see the example below. We use the COUNTC function to discover how many #s exist in a variable name. In practice, this is done in a multidimensional array with a dimension for each # sign in a variable, with the maximum number of possible iterations. 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 supplement label files are created until no more # appear in the variable names.

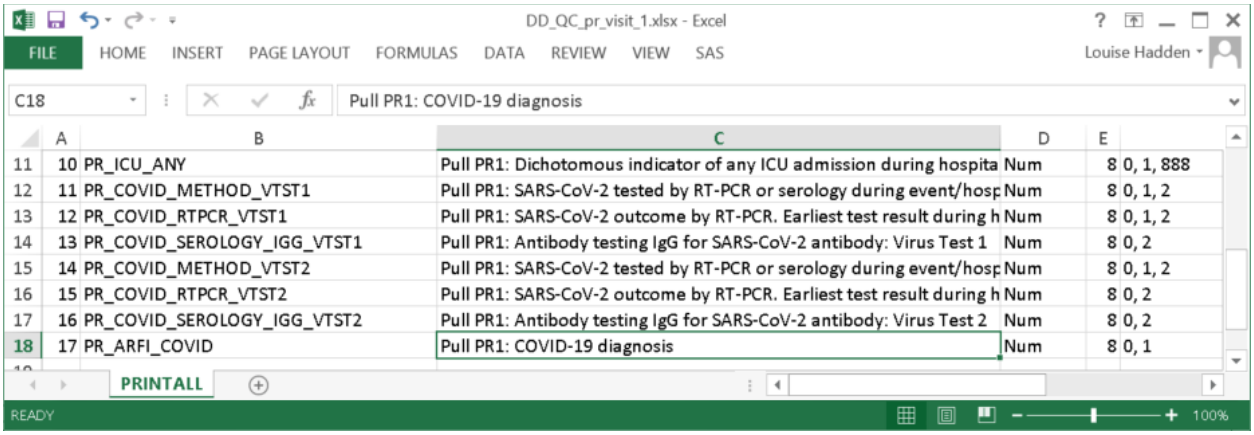
```
data expand_labels;
              length variable $ 32;
              set labels0 (where=(index(variable,'#')>0));

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

              substr(variable,loc,1)="1";
              output;
              substr(variable,loc,1)="2";
              output;
run;
```

Multiple supplement label files are created until no more # appear in the variable names. The files are concatenated and sorted to create a file of labels for potentially thousands of variables, which is matched to the variables in the metadata for actual files. This file can be included when processing an actual data file.

```
data tolabel;
              file label1 lrecl=300;
```

```
            set matchtest (where=(inlabels=1 and inpos=1));
            put labelstr;
run;
```

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



	A	B	C	D	E	
11	10	PR_ICU_ANY	Pull PR1: Dichotomous indicator of any ICU admission during hospita	Num	8	0, 1, 888
12	11	PR_COVID_METHOD_VTST1	Pull PR1: SARS-CoV-2 tested by RT-PCR or serology during event/hosp	Num	8	0, 1, 2
13	12	PR_COVID_RTPCR_VTST1	Pull PR1: SARS-CoV-2 outcome by RT-PCR. Earliest test result during h	Num	8	0, 1, 2
14	13	PR_COVID_SEROLOGY_IGG_VTST1	Pull PR1: Antibody testing IgG for SARS-CoV-2 antibody: Virus Test 1	Num	8	0, 2
15	14	PR_COVID_METHOD_VTST2	Pull PR1: SARS-CoV-2 tested by RT-PCR or serology during event/hosp	Num	8	0, 1, 2
16	15	PR_COVID_RTPCR_VTST2	Pull PR1: SARS-CoV-2 outcome by RT-PCR. Earliest test result during h	Num	8	0, 2
17	16	PR_COVID_SEROLOGY_IGG_VTST2	Pull PR1: Antibody testing IgG for SARS-CoV-2 antibody: Virus Test 2	Num	8	0, 2
18	17	PR_ARFI_COVID	Pull PR1: COVID-19 diagnosis	Num	8	0, 1

This week's tip was contributed by Louise Hadden. Louise Hadden is a Lead Programmer Analyst in the Health and Environment Division at Abt Associates Inc., specializing in analysis and reporting.

Weekly Tip for Nov. 17, 2020

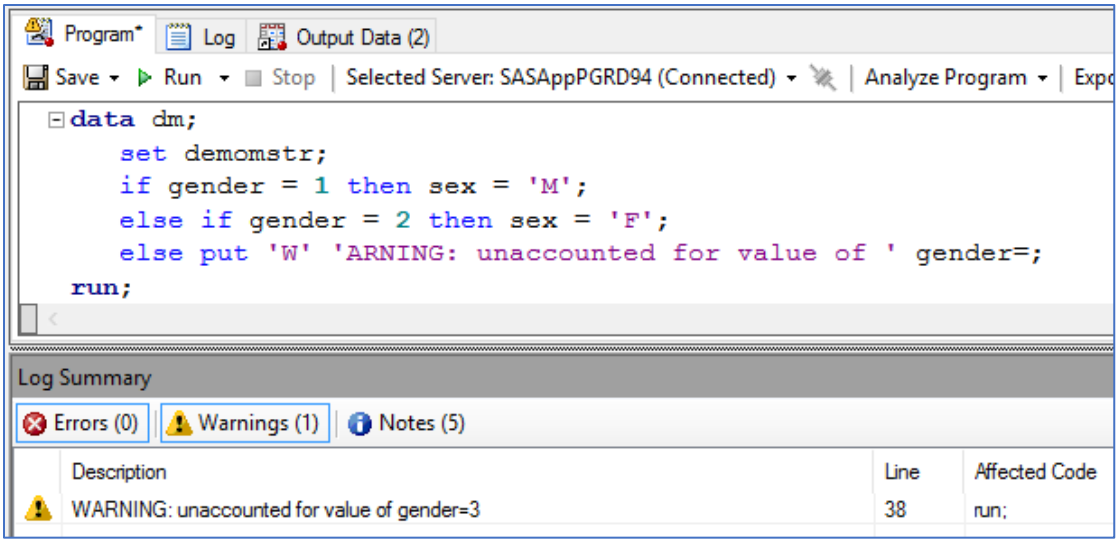
We all make the occasional assumption about variable values when we code. This tip describes a useful technique for checking and documenting your assumptions when you make them.

Consider the simple task of converting a numeric GENDER variable to a character SEX variable.

```
data dm;
   set demomstr;
   if gender = 1 then sex = 'M';
   else if gender = 2 then sex = 'F';
run;
```

The main shortcoming of this code is that it is assuming that 1 and 2 are the only possible values of GENDER. What happens if you have a GENDER value of 3, or 99, or missing? How do you defend against all possible unexpected values of GENDER without having to write 27,463 additional ELSE statements?

The technique is called "custom warnings". It involves writing a mere line of code which, if triggered, will result in a WARNING being written to your log.



If the incoming data set has only values of 1 and 2 for GENDER, then you'll have a clean log. But if any other value occurs (in this case 3), you'll get a WARNING in your log to alert you to the fact that action needs to be taken.

Notice that the word WARNING is broken up into two pieces in the ELSE statement. This is done so that the word WARNING will **not** appear in your log unless the PUT is triggered.

A variation of this technique can also be applied to the macro language. Imagine you have a macro parameter &POS that must be non-missing. You can put in a check at the top of your macro.

```
%if &pos eq %str() %then
   %put %str(W)ARNING: POS is a required parameter;
```

Notice that the details in the macro language are slightly different. Instead of splitting up the word WARNING using quotation marks, we instead split it up using a macro quoting function.

A collection of over a dozen additional applications of custom warnings, collected from my coworkers, is available on GitHub.

https://github.com/srosanba/sas-custom-warnings/wiki

This week's tip was contributed by Shane Rosanbalm. Shane is a Senior Biostatistician at Rho, Inc., with a particular interest in SAS macro efficiencies, as well as graphical presentations of data.

Weekly Tip for Nov. 24, 2020

What simple tips can you use to improve the readability of your SAS output?

The SAS system option shown below returns a line instead of a page break between pages of output. It makes the review of your LST file much more efficient, and I literally use this every time I'm looking at SAS

interactively. This is the single most-used tip I have ever encountered, courtesy of Joy Smith a number of years ago.

```
************************************************************;
OPTIONS FORMDLIM = '_';
************************************************************;
```

For more information on the FORMDLIM system option, see the SAS documentation

FORMDLIM= System Option

Another favorite is the WIDTH=MIN option in PROC PRINT, to condense your output for better readability. I use this enough that I've long since created a little macro that I call "Venita Print", or %VPR. I typically put this at the beginning of all my programs as well.

```
 %MACRO vpr (dset, obs);
    %if "&obs"="" %then %do;
        Title "--- &dset. ---";
        proc print data=&dset (obs=&obs) width=min;
        run;
        Title;
    %end;
    %else %do;
        Title "--- &dset. ---";
        proc print data=&dset width=min;
        run;
        Title;
    %end;
%MEND vpr;
```

For more information on the WIDTH= PROC PRINT OPTION, see the SAS

WIDTH= Option for PRINT Procedure

Happy SASsing!

This week's tip was contributed by Venita DePuy. Venita is the owner of Bowden Analytics and can be contacted at bowden.analytics@gmail.com.

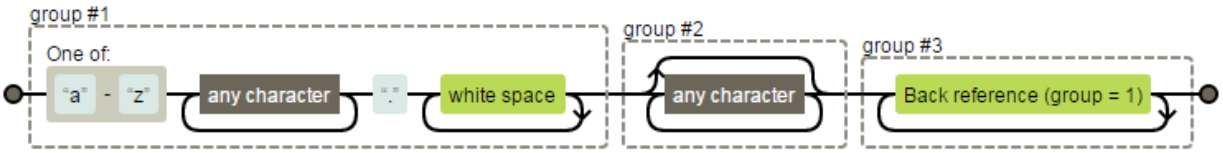
Weekly Tip for Dec. 8, 2020

Removing Duplicate Texts in a String

Sometimes we need to remove duplicate texts in a string. From my point of view, using traditional SAS functions such as SCAN and SUBSTR to solve this issue might be confusing and laborious. Here is a regular expression solution.

```
data _null_;
       infile cards truncover;
       input STRING $32767.;
       REX1=prxparse('s/([a-z].+?\.\s+)(.*?)(\1+)/\2\3/i');
       REX2=prxparse('/([a-z].+?\.\s+)(.*?)(\1+)/i');
       do i=1 to 100;
       STRING=prxchange(REX1, -1, compbl(STRING));
       if not prxmatch(REX2, compbl(STRING)) then leave;
       end;
       put STRING=;
cards;
a. The cow jumps over the moon. b. The chicken crossed the road. c. The quick brown
fox jumped over the lazy dog. a. The cow jumps over the moon.
b. The chicken crossed the road. a. The cow jumps over the moon. b. The chicken
crossed the road. c. The quick brown fox jumped over the lazy dog.
a. The cow jumps over the moon. a. The cow jumps over the moon. b. The chicken
crossed the road. b. The chicken crossed the road. c. The quick brown fox jumped
over the lazy dog. c. The quick brown fox jumped over the lazy dog.
a. The cows jump over the moon. a. The cows jump over the moon. b. The chickens
crossed the road. b. The chickens crossed the road. c. The quick brown foxes jumped
over the lazy dog. c. The quick brown foxes jumped over the lazy dog.
a. The cow jumps over the moon. b. The chicken crossed the road.  c. The quick
brown fox jumped over the lazy dog. a. The cow jumps over the moon.  b. The chicken
crossed the road. c. The quick brown fox jumped over the lazy dog.
;
run;
```

Regular expression visualization by Regexper:

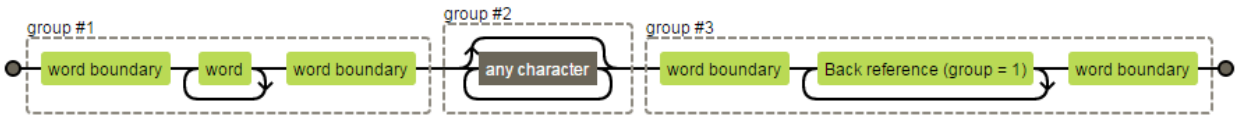


Here's a brief explanation of the regular expression. "[a-z]" matched a single lower-case letter. ".+?" matches any characters as few times as possible. "." matches exactly a period character. "\s+" matches exactly a space as many times as possible. ".*?" matches any characters as few times as possible. "\1+" matches the first capturing group as many times as possible.

Note that if the repeated time value is greater than 100, you need to increase the stopping value in DO loop accordingly. I think this scenario rarely happens. If you want to remove duplicate words instead of sentences, you need to adjust the expression. For example:

```
data _null_;
    STRING='cow chicken fox cow chicken fox cows chickens foxes';
    REX1=prxparse('s/(\b\w+\b)(.*?)(\b\1+\b)/\2\3/i');
    REX2=prxparse('/(\b\w+\b)(.*?)(\b\1+\b)/i');
    do i=1 to 100;
        STRING=prxchange(REX1, -1, compbl(STRING));
        if not prxmatch(REX2, compbl(STRING)) then leave;
    end;
    put STRING=;
run;
```

Regular expression visualization by Regexper:



"\b" matches a word boundary. "\w+" matches any word character (equal to [a-zA-Z0-9_]) as many times as possible.

This week's tip was contributed by Allen Zeng / 曾宪华. Xianhua Zeng is a principal statistical programmer at PAREXEL with 7+ years of experience in clinical trials industry. More information on this topic may be viewed at Removing Duplicate Texts in a String.

Weekly Tip for Dec. 15, 2020

Compare & Warn

Sick and tired of having to look at your output to determine whether or not your PROC COMPARE ran cleanly!? Unburden thyself, dear reader.

SAS® generates an automatic macro variable called SYSINFO after PROC COMPARE has run. This macro variable contains information about the results obtained by the procedure. The macro %CompareWarn (https://github.com/srosanba/sas-comparewarn) has been created to interrogate SYSINFO and write messages to the log about any issues that were detected.

The code is quite simple; you put the macro call immediately after your PROC COMPARE.

```
proc compare base=derive.adsl compare=verify.v_adsl;
run;


%CompareWarn();
```

If the compare is clean, nothing happens. But if the compare has any issues, you will see log messages like the following:

```
WARNING: %CompareWarn detects: BASE data set has observation not in COMPARE
```

Or maybe:

```
WARNING: %CompareWarn detects: A value comparison was unequal
```

The main benefit in using the %CompareWarn macro is that it leverages the capabilities of modern SAS editors such as Enterprise Guide and Studio. Their color-coded Log Summary reports let you know that something is amiss in your compare even before you check the output.

If you work in an environment where log checking programs are used to make sure there are no ERROR or WARNING messages in your logs, this technique will also help to make sure there are no missed issues with broken compares as part of that log checking process.

The %CompareWarn macro is available for download on GitHub:

https://github.com/srosanba/sas-comparewarn

This week's tip was contributed by Shane Rosanbalm. Shane is a Senior Biostatistician at Rho, Inc., with a particular interest in SAS macro efficiencies, as well as graphical presentations of data.

Weekly Tip for Dec. 22, 2020

Get Fuzzy! Comparing Variables with Character and Numeric SAS Functions

SAS® users often need to compare values of variables and assess the matches. SAS has provided a number of tools which can perform "fuzzy matching". Among these tools are "wild card" searches in a where statement using the LIKE (alias ?) and CONTAINS operators; string searches in an if statement using operators and functions; "fuzzy" comparison functions such as COMPGED and SPEDIS; PROC FCMP; PROC GEOCODE, and data-driven control tables enabling user-defined formats to standardize data. Our Tuesday Tip today introduces, by example, "fuzzy matching" tools and techniques, with an example of both character and numeric fuzzies. Additional information may be obtained in our paper, "Quick, Call the "FUZZ": Using Fuzzy Logic", at https://www.pharmasug.org/proceedings/2020/AP/PharmaSUG-2020-AP-007.pdf.

Character strings, especially strings describing names and addresses, are notoriously dirty and prone to spacing, length and punctuation issues. Any real-world comparison of character strings or selection based on character strings needs to be both flexible and configurable, i.e., the degree of "sameness" needs to be quantifiable. SAS provides several character functions that allow you to make a fuzzy comparison: COMPARE, COMPGED, COMPLEV, SOUNDEX and SPEDIS. Each of these functions use a different fuzzy algorithm and can be used in conjunction with one another to achieve a (subjectively) optimal match. Use of these functions produces inexact results by definition, and results must be reviewed carefully.

We illustrate an example of character fuzzies below, using the COMPGED function supplied by SAS. The COMPGED function determines how close the two arguments are in regard to matching (i.e., it determines the generalized edit distance between the two values).

Syntax: COMPGED(string-1, string-2 <, cutoff > <, modifier(s)>)

The cutoff is a numeric value that is returned if the generalized edit distance is greater than the cutoff value. The modifiers that can be used are the same as those used for COMPARE, described in the following table.

Modifier	Description
i or I	Indicates that casing should be ignored for both strings.
l or L	Indicates that leading blanks should be removed from both strings before comparison.
n or N	Indicates that quotation marks from either argument should be removed and that casing should be ignored.
: (colon)	Indicates that the longer string should be truncated to the length of the shorter string. If th is not specified, then the shorter string is padded with blank spaces to the length of the longer string.

More information on the generalized edit distance can be found in the SAS documentation Functions and CALL Routines.

Below we demonstrate how the generalized edit distance can be calculated based on differences in the two strings.

```
data roster_compged;
      set newroster;
      FNCOMP = compged(FIRSTNAME, FIRSTNAME2);
      LNCOMP = compged(LASTNAME, LASTNAME2);
      FNCOMP_I = compged(FIRSTNAME, FIRSTNAME2, 'i');
      LNCOMP_I = compged(LASTNAME, LASTNAME2, 'i');
run;
```

FIRSTNAME	LASTNAME	FIRSTNAME2	LASTNAME2	FNCOMP	FNCOMP_I	LNCOMP	LNCOMP_I
Jan	Write	Jan	Wright	20	20	210	210
Lucy	Smyth	Lucy	Smith	0	0	100	100
Kris	Johnson	Chris	Johnson	300	300	0	0
Chris	Jones	Chris	Jones	0	0	0	0
Tyler	ones	Tyler	Jones	0	0	200	200

Illustration of COMPGED function

We see that for the first name the generalized edit distance without modifiers and with the modifier to ignore casing both yield a value of 20. This is because for FIRSTNAME2 there are two blank spaces that precede the value 'Jan'. The unit cost for each blank space is 10 units, therefore, the total unit cost is 20 units. For the last name we have to replace 'gh' with 'te' and truncate the value. The unit cost to replace a character is 100 units per character and the unit cost to truncate the output string is 10 units per character truncated. Since we had to replace two characters and truncate one character, the total unit cost is 210. Limits or cutoffs can be implemented as well.

The technique demonstrated above, one of many designed for use with character strings, is highly useful, but what if we need to compare a numeric value? There are various options available for determining if a numeric value is equivalent to another numeric value. In some cases, the values will be exactly equal, and no additional comparison is needed. However, there are some cases where the values are not quite equal but are equal 'enough' so that if the values were rounded or truncated or a 'fuzz' factor is added then the values would be considered equal. Depending on the type of 'fuzz' factor you wish to consider will determine which function should be best utilized. Enter the FUZZ function! The FUZZ function either returns the nearest

integer (i.e., rounds up or down based on normal rounding rules) if the return value is within 1E-12 of the argument. If the return value is not within 1E-12 of the argument, then the FUZZ function returns the argument.

```
data fuzz_score;
      set roster;
      format S_FUZZ 20.14;
      S_FUZZ = fuzz(SCORE);
run;
```

FIRSTNAME	LASTNAME	GENDER	SCORE	S_FUZZ
Jan	Write	F	1.00000000000012	1.00000000000000
Lucy	Smyth	F	1.00000000000121	1.00000000000121
Kris	Johnson	F	1.00324325660746	1.00324325660746
Chris	Jones	M	0.00000000000121	0.00000000000121
Tracey	Smith	F	0.00000000000012	0.00000000000000

Illustration of FUZZ function

For the cell highlighted in pink, note that the return value was within 1E-12 and therefore, the value was rounded accordingly.

We hope we have demonstrated that SAS has provided a myriad of tools to utilize for "fuzzy" matching by showing just two of many techniques. Selection of records with where statements (conditions and special operators) and if statements (:_ operator and functions); standardizing of records using fuzzy matching techniques including user defined formats, functions, and PROC GEOCODE (address information); and PROC FCMP (a user-defined function to clean addresses) are all discussed in the full paper referenced above. We hope you've gained some appreciation for the "fuzz" and you'll get "fuzzy" along with us!

This week's tip was contributed by Richann Watson and Louise Hadden.

Richann is an independent consultant. She is a statistical programmer and CDISC consultant with DataRich Consulting. She is an active member of the ADaM team and co-leads the ADaM oncology team.

Louise Hadden is a Lead Programmer Analyst in the Health and Environment Division at Abt Associates Inc., specializing in analysis and reporting.

Weekly Tip for Jan. 05, 2021

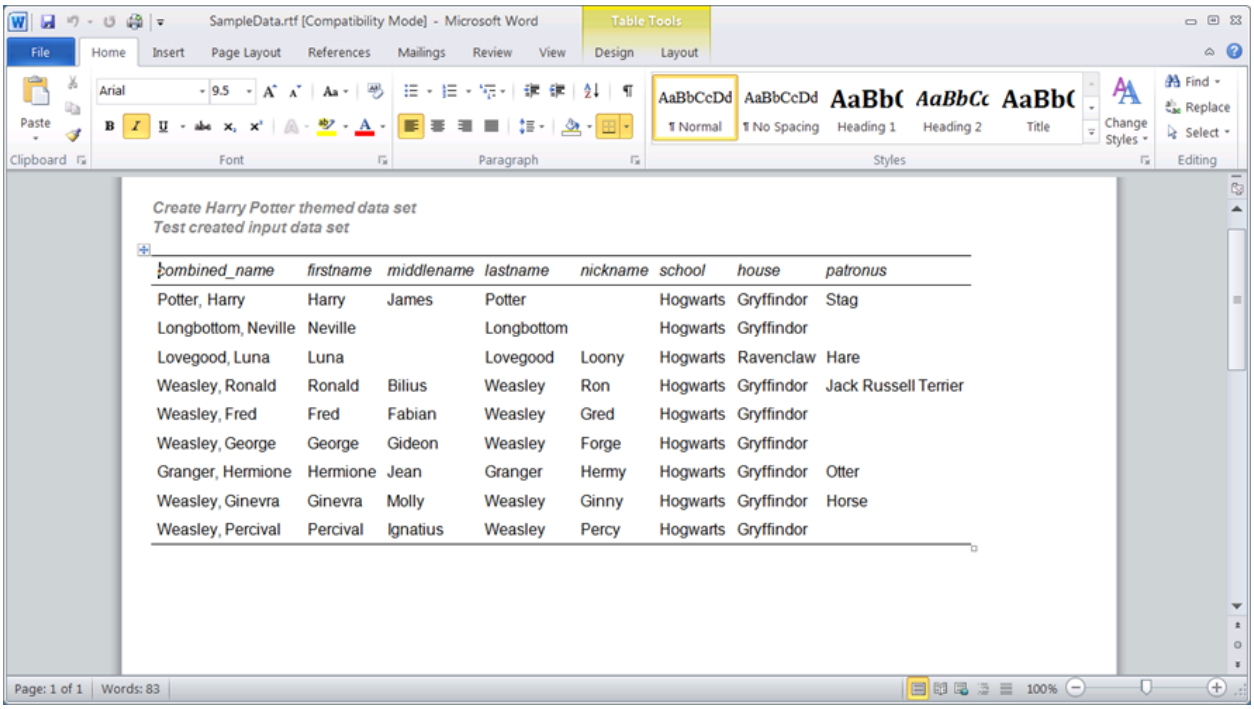
Purrfectly Fabulous Feline Functions

SAS has a wide variety of useful functions and call routines that are constantly being added to and enhanced. Some of my favorites are the fabulous feline functions (CAT, CATS, CATT, and CATX) and three new CALL routines (CALL CATS, CALL CATT, and CALL CATX) that replace clunky and verbose concatenation routines.

```
LENGTH stcounty $ 5 citystzip $ 60;
stcounty=TRIM(LEFT(PUT(state,Z2.)))||TRIM(LEFT(PUT(county,Z3.)));
```

```
citystzip=TRIM(LEFT(city))||','||TRIM(LEFT(statecode))||','||
          TRIM(LEFT(zip));
```

The customary syntax seen above consists of two functions (TRIM and LEFT) paired with the concatenation operator (|| .) TRIM removes trailing blanks, while LEFT left aligns text values. LEFT does not remove leading blanks. It simply moves leading blanks to the end of the string, which is why you usually see LEFT used in conjunction with (and inside) the TRIM function for string concatenation. Over the years, SAS has enhanced the function (pardon the pun) of TRIM and LEFT. A newer function TRIMN also strips trailing blanks: the difference between TRIM and TRIMN is the end result in the case of the argument being missing. TRIM results in a single blank (length 1) for a missing argument while TRIMN results in a null (length 0.) This might come in handy, for example, when concatenating first, middle and last names, in which middle names are frequently missing. Another newer function, STRIP, is the equivalent of TRIMN(LEFT(varx)) but is obviously more convenient. STRIP removes both leading and trailing blanks. The CAT CALLS and functions build on the original and derivative functions to offer a complete array of concatenation options - with a lot less code.



Given the data set shown above, we can use CAT functions (or CAT calls) to create strings combining two or more variables.

The CAT Function

```
data temp;
  set dd.harrypotter;
  fullname=cat(firstname, middlename, lastname);
run;
```

Result of using the CAT function:

```
fullname
```

```
Harry      James      Potter
```

Note the spaces between the three words. The CAT function operates just like the concatenation bars, and simply strings variables together regardless of trailing blanks. Don't forget to use a length statement (I used $ 40) - the default length of the resulting string when using many character functions is 200. On the other hand, be careful - if you don't leave sufficient length for the resulting string, you will get a warning about your buffer and the string will either be truncated or blank depending on your system.

The CATS Function

```
data temp;
  set dd.harrypotter;
  fullname=cats(firstname, middlename, lastname);
run;
```

Result of using the CATS function:

```
fullname
HarryJamesPotter
```

With the CATS function, ALL blanks are STRIPPED from the resulting string.

The CATT Function

```
data temp;
  set dd.harrypotter;
  fullname=catt(firstname, middlename, lastname);
run;
```

Result of using the CATT function:

```
fullname
HarryJamesPotter
```

With the CATT function, TRAILING blanks are removed from the resulting string.

The CATX Function

```
data temp;
  length fullname $ 80;
  set dd.harrypotter;
  fullname=catx(' ',firstname, middlename, lastname);
run;
```

Result of using the CATX function:

```
fullname
```

```
Harry James Potter
```

With the CATX function, EXTRA characters are added to the resulting string.

The CATQ Function

```
data temp;
  length fullname $ 120;
  set dd.harrypotter;
  fullname=catq('CT',
  combined_name, school, house);
run;
```

Result of using the CATQ function:

```
fullname
"Potter, Harry",Hogwarts,Gryffindor
```

With the powerful CATQ function, a number of modifiers are available to customize the resulting string, including surrounding portions with quotation marks.

CALL CATS, CALL CATT and CALL CATX all behave more or less like their corresponding functions described above, except that the resulting character variable is included in the CALL. Arguments can be numeric instead of character; numerics are converted to character using BESTw. Format. Results for CAT functions and CAT CALLs are always character, and should not be reserved SAS variables. Items to be concatenated can be character or numeric. If an input item is numeric, it is converted to BESTw. format automatically and leading/trailing blanks are omitted. If an input item is blank and a delimiter is specified, CAT functions and CAT CALLs remove the delimiter for the blank variable.

It is purrfectly clear that the CAT functions and call routines are valuable additions to the SAS® programmer's toolbox. Try them out, and you'll become a CAT lover too! Please see the full paper for more information and a number of examples at https://www.pharmasug.org/proceedings/2018/EP/PharmaSUG-2018-EP13.pdf. A poster and presentation are available upon request.

This week's tip was contributed by Louise Hadden. Louise Hadden is a Lead Programmer Analyst in the Health and Environment Division at Abt Associates Inc., specializing in analysis and reporting.

Weekly Tip for Jan. 26, 2021

Like, Learn to Love SAS® Like

How do I like SAS®? Let me count the ways.... There are numerous instances where LIKE or LIKE statements can be used in SAS - and all of them are useful. This tip will walk through such uses of LIKE as: searches and joins with that smooth LIKE operator (and the NOT LIKE operator); the SOUNDS LIKE operator; using the LIKE condition to perform pattern-matching and create variables in PROC SQL; and PROC SQL CREATE TABLE LIKE.

Smooth Operators

SAS operators are used to perform a number of functions: arithmetic calculations, comparing or selecting variable values, or logical operations. Operators are loosely grouped as "prefix" (for example a sign before a variable) or "infix" which generally perform an operation BETWEEN two variables. Arithmetic operations using SAS operators may include exponentiation (**), multiplication (*), and addition (+), among others. Comparison operators may include greater than (>, GT) and equals (=, EQ), among others. Logical, or Boolean, operators include such operands as || or !!, AND, and OR, and serve the purpose of grouping SAS operations. Some operations that are performed by SAS operators have been formalized in functions. A good example of this is the concatenation operators (|| and !!) and the more powerful CAT functions which perform similar, but not identical, operations. Like operators are most frequently utilized in the data step and PROC SQL via a data step. There is a category of SAS operators that act as comparison operators under special circumstances, generally in where statements in PROC SQL and the data step (and DS2) and subsetting if statements in the data step. These operators include the LIKE operator and the SOUNDS like operator, as well as the CONTAINS and the SAME-AND operators.

Character operators are frequently used for "pattern matching", that is, evaluating whether a variable value equals / does not equal / sounds like a specified value or pattern. The LIKE operator is a case sensitive character operator that employs two special "wildcard" characters to specify a pattern: the percent sign (%) indicates any number of characters in a pattern, while the underscore (_) indicates the presence of a single character per underscore in a pattern. The LIKE operator is akin to the GREP utility available on Unix / Linux systems in terms of its ability to search strings.

Table 1: Like Operator	
Like "Tr__"	Returns Trys and Troy
Like "Tro%"	Returns Troy and Troye
Like "Try%t"	Returns Tyrst
Like "_r%"	Returns Bryce, Troix, Trys, Troy, Troye and Tryst
Like "Tr___"	Returns Troix, Troye and Tryst

The LIKE operator, described above, searches the actual spelling of operands to make a comparison. ***The SOUNDS LIKE operator*** uses phonetic values to determine whether character strings match a given pattern. As with the LIKE operator, the SOUNDS LIKE operator is useful for when there are misspellings and similar sounding names in strings to be compared. The SOUNDS LIKE operator is denoted with a short cut '-*'. SOUNDS LIKE is based on SAS's SOUNDEX algorithm. Strings are encoded by retaining the original first column, stripping all letters that are or act as vowels (A, E, H, I, O, U, W, Y), and then assigning numbers to

groups: 1 includes B, F, P, and V; 2 includes C, G, J, K, Q, S, X, Z; 3 includes D and T; 4 includes L; 5 includes M and N; and 6 includes R. “Tristn” therefore becomes T6235, as does Tristan, Tristen, Tristian, and Tristin.

Table 2: Sounds Like Operator	
Trystan	T6235
Trysten	T6235
Trystian	T6235
Trystin	T6235
Trystn	T6235
Troye	T6

Joins with the LIKE operator

It is possible to select records with the LIKE operator in PROC SQL with a where statement, including with joins. For example, the code below selects records from the SASHELP.ZIPCODE file that are in the state of Massachusetts and are for a city that begins with “SPR”.

```
proc sql;
   create table temp1 as
   select a.city,
          a.countynm,
          a.city2,
          a.statename,
          a.statename2
   from sashelp.zipcode as a
   where upcase(a.city) like 'SPR%' and pcase(a.statename)='MASSACHUSETTS';
quit;
```

The test print of table TEMP1 shows only cases for Springfield, Massachusetts.

CITY	*COUNTYNM*	*CITY2*	*STATENAME*	*STATENAME2*
Springfield	Hampden	SPRINGFIELD	Massachusetts	MASSACHUSETTS
Springfield	Hampden	SPRINGFIELD	Massachusetts	MASSACHUSETTS
Springfield	Hampden	SPRINGFIELD	Massachusetts	MASSACHUSETTS

The code below joins SASHELP.ZIPCODE and a copy of the same file with a renamed key column (city → geocity), again selecting records for the join that are in the state of Massachusetts and are for a city that begins with “SPR”.

```
proc sql;
   create table temp2 as
   select a.City,
          b.geocity,
          a.countynm,
          a.statename,
          b.statecode,
          a.x,
          a.y
   from sashelp.zipcode as a, zipcode2 as b
   where a.city = b.geocity and upcase(a.city) like 'SPR%' and
         b.statecode = 'MA';
quit;
```

The test print of table TEMP2 shows only cases for Springfield, Massachusetts with additional variables from the joined file.

CITY	*geocity*	*COUNTYNM*	*STATENAME*	*STATECODE*	*X*	*Y*
Springfield	Springfield	Hampden	Massachusetts	MA	-72.589584	42.099922
Springfield	Springfield	Hampden	Massachusetts	MA	-72.589584	42.099922
Springfield	Springfield	Hampden	Massachusetts	MA	-72.538580	42.116054

The LIKE condition

The LIKE operator is sometimes referred to as a "condition", generally in reference to character comparisons where the prefix of a string is specified in a search. LIKE "conditions" are restricted to the data step, as the colon modifier is not supported in PROC SQL. The syntax for the LIKE "condition" is:

```
  where firstname=: 'Tr';
```

This statement would select all first names in Table 1 above. To accomplish the same goal in PROC SQL, the LIKE operator can be used with a trailing % in a where statement.

PROC SQL CASE LIKE

The LIKE operator in PROC SQL can be used in conjunction with the CASE WHEN statement to create variables. The code snippets below feature using the LIKE operator with OR operators and CASE WHEN to create both a categorical variable (DXTYPE) and binaries (CANCERDX, ARTHDX, BCKPNDX, RHEUMDX, NONMSTDX, and OVERDOSE) from selected diagnosis codes with varying prefix lengths. *(Please note that the list of diagnosis codes for each category or binary is incomplete – this code snippet is for demonstration purposes only.)*

```
PROC SQL;
    CREATE TABLE out1 AS
    SELECT DYAD_OPIOID_ID,
           analelig,
           DIAG,
           CASE
           WHEN DIAG LIKE '14%'
           OR diag  LIKE 'D45%'
           THEN 'CANCERDX'
             WHEN DIAG LIKE '720%'
           OR diag  LIKE 'M6788%'
           THEN 'BCKPNDX'
           WHEN DIAG LIKE '725%'
           OR diag  LIKE 'R2989%' THEN 'RHEUMDX'
             WHEN DIAG LIKE '3382%'
           OR diag  LIKE 'T50%' THEN 'OVERDOSE'
            ELSE ''
            END AS DXTYPE,
            CASE
            WHEN DIAG LIKE '3382%'
            OR diag like 'G89%'
            THEN 1
            ELSE 0
           END AS NONMSTDX, . . .
```

PROC SQL CREATE TABLE LIKE

There are several methods of creating an empty data set. PROC SQL CREATE TABLE LIKE is one of the most efficient ones when creating a shell from an existing data set, as it automatically uses the metadata from the existing SAS data set. The CREATE TABLE LIKE method is NOT benign and will overwrite an existing data set. Therefore, the program uses conditional logic and %SYSFUNC to determine if the file to be written already exists, and does not overwrite the file, instead producing an error note in the log.

```
%if %sysfunc(exist(annual.allratings&fileyear._long))=0 %then %do;
  PROC SQL;
    CREATE TABLE annual.allratings&fileyear._long
          LIKE prevann.allratings&prevyear._long;
    DESCRIBE TABLE annual.allratings&fileyear._long;
  QUIT;
%end;

%else %do;
   %put ERROR: Data Set ALLRATINGS&FILEYEAR._LONG already exists!;
 run;
%end;
```

SAS provides practitioners with several useful techniques using LIKE statements: the smooth LIKE OPERATOR / CONDITION in both the data step and PROC SQL, CASE WHEN LIKE in PROC SQL, and CREATE TABLE LIKE in PROC SQL. There are definitely reasons to like LIKE in SAS programming. 

This week's tip was contributed by Louise Hadden. Louise Hadden is a Lead Programmer Analyst in the Health and Environment Division at Abt Associates Inc., specializing in analysis and reporting.

Weekly Tip for Feb. 2, 2021

SAS® formats are a ubiquitous part of every SAS programmer's toolbox. Read on for some useful tips on how to deal with embedded formats that aren't included in a data submission, and how to gracefully ensure that your data deliveries do not cause similar problems for users!

SAS® practitioners are frequently called up to format variables in SAS data sets they have received or created for various use cases. Analysts and other end users desire the convenient categorization, transformative nature, and attractive appearance that SAS formats can lend to variables for reports and further analytic and data set construction purposes. SAS formats can be created in SAS workspace, and can be stored permanently in SAS catalogs, a specially purposed container for SAS files. SAS formats created in workspace are ephemeral, and only exist for the duration of a SAS session. SAS formats stored in a SAS catalog are notoriously difficult to transfer across platforms, SAS versions and "bit" versions (32-bit vs 64-bit). Recipients of SAS data sets with "embedded formats" and/or SAS catalogs originating from incompatible systems find themselves in a quandary – SAS reports errors when it can't find a compatible catalog (IF a catalog accompanies a data set) for user defined formats permanently associated with variables in a SAS data set. SAS catalogs are also very difficult to update, document and manipulate. We propose some straightforward SAS solutions for the creation, transfer and use of SAS formats.

The bane of every SAS programmer's existence are mystery data sets: those data sets with unknown ancestry and little to no information regarding the contents of the data set, delivered without the comfort of well-written documentation and background information such as a survey instrument. Most of us, when faced with such a "mystery box", resort to PROC CONTENTS, a test print, maybe some freqs and means. If you are lucky, you gain useful information from this strategy. If you are REALLY lucky, you can proceed with your analytic and/or file building tasks. All too often, it's a worst-case scenario: you may not have noticed the presence of user-defined formats in the PROC CONTENTS output in your testing, and/or you went straight to a test print or frequencies. The next thing you know you see the dreaded error message in your log:

```
ERROR: The format $YNDKF. was not found or could not be loaded.
```

Reality comes crashing in. Sometimes SAS data sets are delivered with embedded formats, and the expected format catalog is either not supplied, is incompatible due to system/platform/bit differences, is outdated, is corrupted, and/or is incomplete.

```
OPTIONS nofmterr;
```

This little SAS system option is a quick band aid that will allow you to open and process a data set with embedded formats that you do not have a catalog for, but you will not have the benefit of the formats or view the data as it was intended to be.

```
OPTIONS FMTSEARCH=(lib.survey1 lib.survey2 lib2.survey3);
```

At other times, the problem may be the location of the missing library. Using the fmtsearch option allows you to load format catalogs in the order that they are mentioned. Note that the librefs are tied to the libname statements in your program.

But what do you do if you STILL don't have a format catalog for your data set?

If the data set that has been delivered without being properly attired is from a known source that has been contracted to provide the data, request that incoming data sets be delivered without embedded formats and with well documented programs to (a) create a collection of formats and (b) associate those formats with variables in the delivered data set(s).

If you are unable to obtain better documentation and/or your format catalog, not all is lost. Strip embedded formats with the option NOFMTERR described above and a data step by simply reformatting variables with an "empty" format statement.

```
DATA WANT;
   SET HAVE;
   FORMAT _ALL_;
RUN;
```

PROC DATASETS, ATTRIB statements or PROC SQL also work well to (a) create a collection of formats and (b) associate those formats with variables in the delivered data set(s). Documentation of the data set(s) certainly helps in this effort.

Strip embedded formats and (re) build formats and format associations using techniques described in detail below.

Now we have associated formats in our file - what else should we do?

Create transportable, modifiable formats to associate with variables in data sets of in procedures.

Build a SAS data set to house elements of a format catalog that can be transformed into catalogs via PROC FORMAT CNTLIN.

```
PROC FORMAT LIBRARY = LIBRARY.CHILD_DAY CNTLIN=PRVTYPID; RUN;
```

Build a SAS data set from an existing SAS format catalog via PROC FORMAT CNTLOUT that can be used for PROC FORMAT CNTLIN.

Just as you can use PROC FORMAT CNTLIN to read a SAS data set into a temporary or permanent format catalog, you can also get information OUT of a format catalog. When you (or someone else) create a format

catalog, even in workspace, a SAS data set is generated with an inordinate number of columns, of which only five or six are commonly used, FMTNAME, TYPE, START, END, LABEL and HLO (only three are required – FMTNAME, START and LABEL).

```
PROC FORMAT = LIBRARY.CHILD_DAY;
   CNTLOUT = CNTLOUT;
RUN;
QUIT;
```

WYO – write your own format!

```
PROC FORMAT = LIBRARY.CHILD_DAY;
   VALUE YESNODK 1 = "Yes"
                 2 = "No"
                 8 = "Don't Know";
RUN;
```

Additionally, you can use an include statement that contains PROC FORMAT statements to read in formats within your SAS program, and then apply them to a work catalog so that they can be used. If you can locate an electronic survey instrument or existing documentation, it's possible to process these files to create a format creation program.

When building SAS format catalogs, use identifiable two-part format catalog names i.e., survey1.sas7bcat, survey2.sas7bcat instead of formats.sas7bcat. To save a format or formats in a permanent format catalog, use PROC FORMAT to create a format by either defining value statements or by using PROC FORMAT's CNTLIN statement. Note the use of a specific format catalog name instead of the default formats.sas7bcat, which makes it easier to identify the specific formats needed for a task. In addition, if you are creating several different types of catalog entries for a project, make sure to distinguish between format, macro, and template catalogs either by name or by locating them in separate folders to avoid overwriting your entries.

The below sample shows how to modify a format by adding a description, and the PROC CATALOG listing shows the entry name, type, dates and description.

```
PROC CATALOG CATALOG = LIBRARY.CHILD_DAY;
   MODIFY PRVTYPID FORMAT (DESCRIPTION = "Categories for Provider Type ID");
   CONTENTS;
RUN;
QUIT;
```

The PROC CATALOG CONTENTS statement lists the contents of the format catalog. Note that PROC CATALOG is the ONLY way to add a description to a format, as PROC FORMAT does not have a LABEL or DESCRIPTION option.

```
Contents of Catalog LIBRARY.CHILD_DAY
# Name        Type    Create Date  Modified Date       Description
1 PRVTYPID    FORMAT  07/21/2019   07/21/2019          Categories for Provider Type ID
```

For more detail on FORMAT type entries, use PROC FMTLIB. Note that PROC CATALOG provides a different set of information from PROC FORMAT FMTLIB, and that both outputs are useful.

```
PROC FORMAT CATALOG = LIBRARY.CHILD_DAY FMTLIB;
RUN;
```

```
                  FORMAT NAME: PRVTYPID LENGTH: 24
MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH: 24 FUZZ: STD
START           |END           |LABEL (VER. 9.4 24MAR2019:13:35:29)
----------------+--------------+----------------------------------
CCC             |CCC           |Child Care Center
HS              |HS            |Healthy Start
AR              |AR            |At Risk
OSH             |OSH           |Outside of School Hours
```

A very powerful technique is to build a SAS data set instead of a SAS format catalog to house your formats.

```
* 1: map provnum to fips county code;

data ctrl_prov_fips;
length label $ 3 BASEVAR $ 32;
    set prov_xwalk2 (keep = provnum county_fips
    rename=(provnum=start county_fips=label)) end=last;
    retain fmtname '$FIPSCTY' type 'c' BASEVAR 'PROVNUM';
    output;
    if last then do;
    hlo='O';
    label='***';
    output;
    end;
    keep fmtname type start label hlo basevar;
run;

proc sort data = ctrl_prov_fips ;
    by fmtname start;
run;

* Put the formats together, setting length long enough to accommodate
 the longest label;

data allgeogformats_&fileyear.&filedate.;
   length fmtname $8 type $1 start $14 label $255;
   set ctrl_prov_fips
         ctrl_prov_cntynm
         ctrl_prov_cbsa
         ctrl_prov_urban
         ctrl_prov_regionc
         ctrl_prov_fips_st;
     by fmtname start;
     label fmtname = 'Name of Format'
           start = 'Start of Range for Format'
           label = 'Label for Format'
           type = 'Type of Format';
run;
```

You can then use this data set for reporting and/or to create a format statement, as shown below.

```
proc format library = work
    cntlin = allgeogformats_&fileyear.&filedate.;
run;
```

We hope you can use some of these tips to declare your independence from the format catalog!

This week's tip was contributed by Nancy McGarry. Nancy is a Lead Programmer Analyst in the Social and Economic Policy Division at Abt Associates Inc., specializing in nutritional analysis and reporting.