# PRESTO! CHANGEO! MAKING DATA TRANSFORMATIONS LOOK LIKE MAGIC

Patricia Hettinger, SAS Professional, Oakbrook Terrace, IL

## ABSTRACT

If you need to do data transformations such as from numeric to character, character to numeric, etc., here is a discussion of some useful SAS functions. The paper will also discussion automation methods including table driven transformations.

## INTRODUCTION

There are many times during one's work day where data must be transformed in some way or other. Some transformations are simple. Others may involve so many changes that the coder is in danger of writer's cramp or carpel tunnel syndrome. Some you can't figure out how to do at all. Don't despair – SAS has many tools to make creating new variables or updating old ones simpler. You may have been reading some of the documentation but can't find examples you might use in real life. In this paper, we will examine some of SAS's built in functions, informats and formats, table look-ups and even creating your own functions to alter your variables in a manner that give predictable, repeatable results. Better yet, some of these are easy to pass to others on your team needing to do the same kind of transformation. Best of all, no special tools required, just base SAS.

## IDENTIFY THE PROBLEM

We want to do a state code look-up to replace state abbreviations with the actual state name. We could do it like many do with an IF-THEN-ELSE construct like this:

```
If state_abbrev eq 'AL' then state_name='Alabama';
If state_abbrev eq 'AK' then state_name='Alaska';
If state_abbrev eq 'AZ' then state_name='Arizona';
```

And so on.

That's a lot of typing and remembering to put in all of those semi-colons. Ah, but you're far along enough on your SAS journey to try writing a format instead. Then you could use the new format throughout your session without having to code it again while the IF-THEN-ELSE fabrication is only effective in the data step. So you decide to code a new format with proc format. Let's call the format $state_name and make it resolve to at least 20 positions so that DC, 'District of Columbia' and other longer state names will not be truncated.

```
Proc format;
Value $state_name min=20;
'AL' = 'Alabama'
'AK' = 'Alaska
'AZ' = 'Arizona'
```

And so on.

This is still a lot of typing and prone to error. How might we improve this? Better still, how might we avoid having EVERYONE code this again and again when state abbreviations haven't changed for at least the last ten years?

## SAS TO THE RESCUE

It would certainly help to develop this logic only once. In the case of our state abbreviation, SAS already did. There are actually a number of built-in functions that return various state information from the state abbreviation code. One is STNAME which will return the name of the state for any state abbreviation in upper case. STNAMEL will return the name of the state in lower case. It wasn't necessary to code this ourselves AT ALL. If there's one thing you should take away from this discussion, it is to check the literature first to see if SAS already has what you need and if so, how it is used. To assign the state abbreviation to another variable based on the contents of variable *state_code*, you would simply code:

```
state_name=stnamel(state_code);
```

Although not as complete, there are several other SAS functions that should become familiar to you in manipulating text. Maybe you have to take out unwanted characters from a name field or you need to split up a character field and put it back together again. Or you inherited some code and want a better idea of why that particular function is there. These are a few the author has used and has seen many times in others' code.

COMPRESS – removes blanks (default) or a list of characters from a string. The syntax is COMPRESS(<source> ,< chars>, < modifiers> ) where <source> is the variable you want to process, <chars> individual characters you might want to remove and <modifiers> which are whole categories of characters. The two modifiers used in the code samples are 'd' for all numerals and 'a' for all letters. If you don't put in the chars or the modifiers then just spaces will be removed. The examples below before and after for a) removing all blanks, b) removing all numbers and c) removing a particular character, 'X'

COMPRESS to remove blanks:
compress('CLINT ON') - After: 'CLINTON'

COMPRESS to remove all numbers:
compress('JOHNSON2017','d'): - After: 'JOHNSON'

COMPRESS to remove all letters:
compress('JOHNSON2017','a'): - After: '2017'

COMPRESS to remove the letter X:
compress('XL'): - After: 'L'

TRIM – Remove trailing blanks. Syntax is TRIM(<source>)
trim('JILL STEIN ') - After: 'JILL STEIN'

LEFT – align the expression to the left. Syntax is LEFT(<source>)
left(' KIRK') - After: 'KIRK '

RIGHT – align the expression to the right. Syntax is RIGHT(<source>)
right('KIRK ') - After: ' KIRK '

TRANSLATE – replace characters in a string with other characters. Syntax is TRANSLATE(<source>, <to>, <from>). Remember the character you want to translate to must come first. The arguments are also case sensitive.
translate('TRUMP','R','O','U','W','M','E','P','R') is still TRUMP (wrong argument order)
translate('TRUMP','o','r','w','u','e','m','r','p') is still TRUMP (lower case not found)
tranlsate('TRUMP','O','R','W','U','E','M','R','P') is now TOWER (correct translation)

SUBSTR – take a portion of the source value. One of the most used. The syntax is SUBSTR(<source>, <beginning position>, <length>)
substr('DUCKWORTH',1,4) - After: 'DUCK'

UPCASE, LOWCASE and PROPCASE. Puts your character expression in upper case, lower case or proper case respectively.
lowcase('Oakbrook Terrace') – After: oakbrook terrace
upcase('Oakbrook Terrace') – After: OAKBROOK TERRACE
propcase('OAKBROOK TERRACE') – After: Oakbrook Terrace

These are just some of SAS's functions. One worth understanding is SCAN. SCAN takes the nth word from an expression. SAS decides what a word is based on the default delimiters or one you specify yourself. The ASCII code set default delimiters are blank ! $ % & ( ) * + , – . / ; < |. EBCDIC default delimiters are blank ! $ % & ( ) * + , – . / ; < ¬ | ¢. This function can be used to split character strings and rearrange them. Say we want to reformat a name field that has the incoming format of LAST NAME, FIRST NAME into FIRST NAME LAST NAME (no comma). We would include any middle names or initial names in the first name field. Some of the name fields have a '/' (forward slash) at the end and some do not. This is one way we could put a name like HETTINGER, PATRICIA A into PATRICIA A HETTINGER, a form more suitable for mailing. Not only will we use the SCAN function but also UPCASE, LEFT and TRIM.

```
name_temp=scan(name_field,1,'/'); /*See Note 1*/
lname = scan(nametemp,1,','); /*See Note 2*/
fname= scan(nametemp,2,','); /*See Note 3*/
lname=upcase(left(lname)); /*See Note 4*/
fname=upcase(left(fname)); /*See Note 4*/
outname_field = trim(fname)||' '||trim(lname);/*See Note 5*/
```

Note 1: Scan the incoming field for a forward slash - this will populate the name_temp variable with the entire name whether there is a forward slash at the end or not
Note 2: Scan the name_temp variable for the first delimiter. Here we specified a comma. This goes into the lname variable.
Note 3: Scan the name_temp variable for the second delimiter, also a comma. There isn't a second comma so we will take everything after the first comma and put it in the fname variable
Note 4: Format both fname and lname to be upper case and left justified
Note 5: Trim trailing spaces for fname and lname and concatenate with a space in between

Coding Example 1

## THE PUT AND INPUT FUNCTIONS WITH FORMATS AND INFORMATS

Many people are confused by informats and formats. So many of them seem identical – what is the difference? An informat may be considered a pattern for reading input into SAS data while a format may be considered as a pattern for displaying that SAS data. For transforming SAS data, you will probably use the PUT function more than INPUT. Both of these are resolved at compilation. There are related functions that are resolved at run time but these will not be discussed here.

If your new variable is of the same type as the old variable, you may find the new variable isn't really converted to the formatted value is instead just displaying as the new value. This tends to happen more with numeric variables being decoded to character. When you print it out, it looks like it changed but it really didn't. Proc CONTENTS will show that it is still a numeric variable. If you want to transform a numeric variable to a character, you should describe a new variable first as character with an ATTRIB or LENGTH statement. Then the translation will happen as expected.

A built-in informat you should know is ANYDTDTE. This is fairly recent in the SAS universe. If you want to create a date from character input, ANYDTDTE will translate practically any character date into a SAS date. Before the ANYDTDTE informat, we had to know how many positions the incoming date was, whether it was broken up by dashes or slashes like 11/5/2016 and the order the date was in: month-day-year, year-month-day or even year-day-month. Although this makes reading dates easier, there are two things of which to be aware. If your date doesn't have the century in it (all too common, even sixteen years after Y2K), you need to pay attention to the YEARCUTOFF option. The default is 1920 which means a value of 0-20 would be considered the 21th century while a larger value would be considered the 20th century. Dates without the century have another problem. What do we do with a date like 01/01/02? Is it 01JAN2002 (month-day-year), 02JAN2001 (year-month-day) or even 01FEB2001 (year-day-month)? The DATESTYLE option sets the date order for you, with default of MDY (month-day-year). With this setting, the program would select January 1, 2002. Either the YEARCUTOFF or DATESTYLE options can be changed as needed. Considering that we have only four years to 2020, there probably will be a need. If your incoming dates are actually numeric, quite common with legacy systems, a put statement converting the numeric value to an alphanumeric will usually be effective as in this example where your numeric date might be 010303. You do, however, have to be careful with leading zeros – if you don't provide the Zn format, the ANYDTDTE informat will interpret this as a Julian date

```
sasdate=input(put(yournumeric_date,8.),$anydtdate8.); = 30OCT2010 –  303 day of 2010  (Julian 10303)
sasdate=input(put(yournumeric_date,Z8.),$anydtdate8.); = 03JAN2003  - MDY (01/03/03)
```

It works the other way around too. If you find yourself needing to convert a SAS date into a text or numeric value readable by other systems as dates, you have your choice of many date formats. ANYDTDTE is not one of them as it is an informat only. Informats tell you how to read incoming data not how to output them so asking SAS to decide how to display a date is a bit beyond its scope. An example of a format that can't be used as input is Zn which specifies a number be shown with leading zeros. If you are inputting a number, leading zeros are irrelevant. Unless you're trying to decipher a date!

You've read the documentation and can't find any SAS formats or informats that do what you need done. Do we go back to IF-THEN-ELSE? Which brings us to ---

## YOUR VERY OWN FORMATS

We can create our own formats by using proc FORMAT as we did in the state abbreviation example prior. Let's make one that SAS doesn't have like a county name look-up. If you go to the United States Postal Service website, a zip code lookup will return the county. And there are some websites that have a list of counties by zip code. We could create our county list by using proc FORMAT as so:

```
proc format;
value county;
60506 = 'Kane'
60507 = 'Kane'
60548 = 'Kendall'
```

And so on…

Not really practical – Illinois has thousands of zip codes that are updated every now and then. Wouldn't it be great if someone would maintain a list for us? And wouldn't it be even greater if we could just read this list into a format? Well, we can. There are several websites with this information. One good source is zipcodestogo.com where the data was copied for this lookup. We have unique keys in zip code – in Illinois at least, no zip code is associated with more than one county so loading the list into proc FORMAT is fairly simple. The list was copied into a spreadsheet and then transferred to SAS using the Import tool in SAS Enterprise Guide. If you don't have SAS Enterprise guide, you could use proc IMPORT in other windowing environments or even save the data in a delimited text file and read in with the data step. After import, we saved it to permanent library DCTABLE and called the member COUNTY_CODE_LIST.

The structure was:

COUNTY_NAME – Format $43.
ZIP_CODE5 – Format 5.

The zip_code5 could have been imported as either numeric or alphanumeric. The source we want to look up is numeric so we'll have our lookup key be numeric as well.

So how do we create a format from this list? First let's make sure there really are no duplicate zip codes:

```
proc sort data=dctable.county_code_list nodupkey out=ccds dupout=duprecs;
by zip_code5 ;
run;
```

Now condition the data:

```
data ctrl;  /*See Note 1*/
 length label $ 43;  /*See Note 2*/
 set ccds(rename=(zip_code5=start )) end=last; /*See Note 3*/
                label=county_name; /*See Note 4*/
retain fmtname 'county_name' type 'n'; /*See Note 5*/
 output;  /*See Note 6*/
if last then do; - /*See Note 7*/
                 hlo='O';
label='999'; /*See Note 7*/
output;
end;
run;
```

Note 1: Output for format load.
Note 2: Set the minimum length of the format so that even the longest county names display properly.
Note 3: We are using zip_code5 as the key so we will tell SAS to populate the START variable with zip code by renaming the zip_code5 to START.
Set the variable 'last' to the end of the input dataset because we will need to do something when
all of the records have been read.
Note 4: county_name is used to give the resulting value.
Note 5: county_name is our format name and 'n' indicates numeric. If we had imported our zip_code5
variable as character, we'd use 'c' instead.
Note 6: Output the conditioned data set.
Note 7: We'll put something in that assigns a default value for anything not in the list. We'll set it to a high numeric to avoid missing values for those of you disliking missing values.

Coding Example 2

Now that we have the data fixed up, a simple proc format puts the new county_name format in the work library. The cntlin option tells us to load the ctrl dataset we just created. We could save the format to a permanent library as well and use the FMTSEARCH system to show use where to search. We'll keep it in a work library for now:

```
proc format library=work cntlin=ctrl;
run;
```

Once a format is built, the simplest way to test it is to run the original table through, assign new variable names and then use a procedure like proc compare to validate the format. Below is an example of using the new format:

```
data convert_county_codes;
attrib new_county_name format=$43.; -
set ccds; - our sorted source table for the format
new_county_name=put(state_abbrev,county_name.);  d
run;
```

Using Example 2

Proc compare compares the new variable with that in the original list. We will compare county_name with new_county_name:

```
proc compare base=ccds compare=convert_county_codes;
var
county_name
;
with
new_county_name
;
run;
```

Proc compare output shows us the new fields are identical to the old, confirming they were populated correctly. Here's some of our output:

```
The COMPARE Procedure
Comparison of WORK.CCDS with WORK.CONVERT_COUNTY_CODES
(Method=EXACT)

Data Set Summary

Dataset                          Created              Modified   NVar    NObs

WORK.CCDS                        12OCT16:21:12:54     3          64
WORK.CONVERT_COUNTY_CODES        12OCT16:21:12:55     5          64

Variables Summary

Number of Variables in Common: 1.
Number of Variables in WORK.CONVERT_COUNTY_CODES but not in WORK.STATECDS: 2.
Number of VAR Statement Variables:1.
Number of WITH Statement Variables: 1.
Observation Summary

Observation      Base  Compare

First Obs        1       1
Last  Obs        1587  1587

Number of Observations in Common: 1587.
Total Number of Observations Read from WORK.CCDS: 1587.
Total Number of Observations Read from WORK.CONVERT_COUNTY_CODES: 1587.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 1587.

NOTE: No unequal values were found. All values compared are exactly equal.
```

This works well for a one-key look up. But what if you have two or more keys? Do we need to do an IF-THEN-CONSTRUCT after all, even creating some more formats? The next section describes another method available with base SAS, defining and using hash tables.

## HANDLING MULTIPLE KEYS WITH A HASH TABLE

For this example, we want to look up sales tax rates. Downloading a table from the state of Illinois, we know we need the state abbreviation code, the city and the county. We'd need this for every city in Illinois and the other 49 states. We'd also have an entry for FOREIGN COUNTRY (FC). So if we coded it, our code would look something like this:

```
if state_abbrev eq 'IL' and city_name='AURORA' and county_name='DUPAGE' then sales_tax_rate=0.0825;
if state_abbrev eq 'IL' and city_name='AURORA' and  county_name='KANE' then sales_tax_rate=0.0825. ;
```

Or assuming we have the zip code, we'd use our county_name format:

```
If state_abbrev eq 'IL' and city_name='AURORA' and put(zip_code5,county_name.) in ('DUPAGE','KANE') then
sales_tax_rate=0.0825;
```

But Illinois has hundreds of cities and counties. Typing them all in doesn't make any more sense than typing in all the zip codes previously. The state of Illinois regularly publishes such rates. Could we load them into a SAS dataset somewhere and use it to look these up? As we already saw, we could create a number of formats. But a method worth seriously considering is a hash table lookup. A hash table is loaded into memory with the keys you need for the lookup. Such a table is much faster than using a join or merge. We can easily download a list of cities, counties and sales taxes off the internet. The data for the next coding example came from SalesTaxHandbook.com.

The table structure is:

CITY_NAME – Format $20
COUNTY_NAME – Format $11
LOCATION_CODE – Format $10
SALES_TAX_RATE – Format Z6.4
STATE_ABBREV – Format $2

To find the correct sales tax and location code, you need to know the state, the city and the county. Aurora, IL, one of the largest cities in Illinois is located in four counties. Thus there are four entries in the table for Aurora:

| STATE_ABBREV | CITY_NAME | COUNTY_NAME | SALES_TAX_RATE | LOCATION_CODE |
|---|---|---|---|---|
| IL | AURORA | DUPAGE | .0825 | 022-0042-2 |
| IL | AURORA | KANE | .0825 | 045-0002-4 |
| IL | AURORA | KENDALL | .0850 | 047-0021-2 |
| IL | AURORA | WILL | .0825 | 099-0069-1 |

The list provided by the state of Illinois had 1595 entries in it as of October 2016. It also changes fairly often. Obviously coding this is not practical. Creating and updating a hash table to look these up is much easier once we understand the basic setup. We again loaded a SAS table in the DCTABLE library and called it SALES_TAX_TABLE. This will be source for the resulting hash table DCTABLE.SALES_TAX_LOOKUP.

Here's the setup. It looks much like Visual Basic or .Net with its declarations and methods:

```
data _null_;
        if 0 then set dctable.sales_tax_table; /*See Note 1*/
        if _n_ = 1 then
                        do;
                                declare Hash matchstate (ordered:'a'); /*See Note 2*/
                                matchstate.DefineKey ('STATE_ABBREV','COUNTY_NAME','CITY_NAME'); /*See Note 3*/
                                matchstate.DefineData ('STATE_ABBREV'); /*See Note 4*/
                                matchstate.DefineData ('COUNTY_NAME'); /*See Note 4*/
                                matchstate.DefineData ('CITY_NAME'); /*See Note 4*/
                                matchstate.DefineData ('SALES_TAX_RATE'); /*See Note 4*/
                                matchstate.DefineData ('LOCATION_CODE'); /*See Note 4*/
                                matchstate.DefineDone (); /*See Note 5*/
                        end;
                set dctable.sales_tax_table end=eof; /*See Note 6*/
                matchstate.add (); /*See Note 7*/
                if eof then
        matchstate.output(dataset:'dctable.sales_tax_lookup');/*See Note 8*/ run;
```

Note 1: Load SAS dataset with sales tax rates into hash table, the sales_tax_table in our dctable library is the input.
Note 2: Declare the hash table and its sort order, 'a' for ascending.
Note 3: define the keys for the hash table. There are three: STATE_ABBREV, COUNTY_NAME and CITY_NAME. The DefineKey method handles this.
Note 4: Now define the data which includes the keys. DefineData defines five data elements: STATE_ABBREV, COUNTY_NAME, CITY_NAME, SALES_TAX_RATE and LOCATION_CODE.
Note 5: The DefineDone method tells SAS that the key and data definition is done.
Note 6: Set the input dataset again and set the END internal variable indicating last observation to EOF.
Note 7: The add method will add this hash table to the collection
Note 8: When the data items are done loading, output the hash table to a permanent library. If you don't save the table to a permanent library, you'll have to set up the hash table every time you need it.

Coding Example 3

A possible downside to this method is having to redefine the table again when we use it in a data step. Here's how we'd do that, testing the original sales tax table as we did the county_name format before.

```
data outfile;
        attrib out_tax_rate format=z6.4 out_loc_code format=$10.; /*See Note 1*/

        if 0 then
                set dctable.sales_tax_table; /*See Note 2*/

        if _n_ = 1 then
                do; /*See Note 3*/
                        declare Hash Matchstate (dataset:'dctable.sales_tax_lookup');
                        matchstate.DefineKey ('STATE_ABBREV','COUNTY_NAME','CITY_NAME');
                        matchstate.DefineData ('STATE_ABBREV');
                        matchstate.DefineData ('COUNTY_NAME');
                        matchstate.DefineData ('CITY_NAME');
                        matchstate.DefineData ('SALES_TAX_RATE');
                        matchstate.DefineData ('LOCATION_CODE');
                        matchstate.DefineDone ();
                end;

        set dctable.sales_tax_table; /*See Note 4*/

        if matchstate.find() = 0 then
                do; /*See Note 5*/
                        out_tax_rate=SALES_TAX_RATE;
                        out_loc_code=LOCATION_CODE;
                end;
run;
```

Note 1: attrib statement makes sure our new variables have the desired format.
Note 2: dctable.sales_tax_table is the table we used to load the hash table.
Note 3: Need to describe the hash table again even though we told SAS where the hash table is located (dctable.sales_tax_look).
Note 4: Set original sales tax table again because that will be our input into the look-up
Note 5: Find method lets us look for the matching keys. If the return code was 0, the lookup was successful and we output the tax rate and location code

Using Example 3

Using proc compare to contrast the original sales tax and location with the ones developed by matching on city, state and county gives us:

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 1595.

NOTE: No unequal values were found. All values compared are exactly equal.

Variables with All Equal Values

| Variable | Type | Len | Compare | Len |
|---|---|---|---|---|
| SALES_TAX_RATE | NUM | 8 | out_tax_rate | 8 |
| LOCATION_CODE | CHAR | 10 | OUT_LOC_CODE | 10 |

This is all very good. But this isn't exactly simple. The end user needs to code its definition even if is located in a permanent library. Starting with SAS 9.3, there is a way to hide the details of a hash table lookup and write a function or subroutine that would be called whenever you need it. Just like SUBSTR, LENGTH, COMPRESS or the other dozens of built in SAS functions.

## DEVELOPING YOUR OWN FUNCTIONS

Proc FCMP lets you create your own reusable procedures that let you provide parameters and get desired output. You can write functions or subroutines. The difference between functions and subroutines is that a function returns a value while a subroutine does not. One thing subroutines can do that functions can't is actually update the variables coming into it. We will write a couple of functions and subroutines demonstrating the difference.

Here's how we could write a function to return the full name in the desired format. This builds on the SCAN routine covered earlier. Instead of providing the code which has a way of getting misused, we can write a function and call it instead.

```
proc fcmp outlib=dctable.functions.transform;/*See Note 1*/
function nameReformat(namefield $, delimiter $) $50; /*See Note 2*/
name_temp=scan(namefield,1,'/'); /*See Note 3*/
lname = scan(nametemp,1,delimiter);
fname= scan(nametemp,2,delimiter);
lname=upcase(left(lname));
lname=upcase(left(fname));
outname_field = trim(fname)||' '||trim(lname); /*See Note 4*/
return(trim(fname)||' '||lname); /*See Note 4*/
endsub; /*See Note 5*/
```

Note 1: Need a three level library – last two levels created if not present already but first level must be allocated so we will put these in the DCTABLE library we allocated already
Note 2: We must specify 'function' and give it a name. Then we name the incoming parameters for use within the function.
The incoming parameters are referenced by position, not name. Note the $ after namefield and delimiter. These character parameters must be specified as such or you'll get a numeric transformation error at run time. After the parameters, the type and length of the returned character value is shown as $ 50. This is to make sure the function knows it is to return character data.
Note 3: Same logic is used to parse out the incoming name as we saw before. The only difference is that we have the desired delimiter as a parameter instead of the hard-coded comma
Note 4: We created a new variable within the function, outname_field but returning this on a UNIX platform didn't give good results. Instead the concatenation operation between fname and lname is returned.
Note 5: Functions and subroutines must end with the ENDSUB statement.

Coding Example 4

We've created our NameReformat. Now how do we make it available for future use? Use the handy system option APPEND to add the new function library with the first two levels to the CMPLIB option: options append=(cmplib= *dctable.functions).* Proc options will show the functions library was added to the CMPLIB option, in this case coming behind sys.funcs24 and sys.funcs.

*CMPLIB=(*sys.funcs24 sys.funcs dctable.functions)

We would use our nameReformat function in a data step just like any built-in SAS function. And just like any other SAS function, we have to provide the parameters the function is expecting:

```
Data name_parse;
Set namelist;
Candidate_full_name=nameReformat(candidate_name,',');
Run;
```

Using Example 4

This function returns just one value. In this case, the full name in the format of FIRST MIDDLE LAST. If we wanted to return just the first name or just the last name, we'd need two functions in order to accomplish this. A way around this is to write a subroutine that would actually update the variables. Example 5 demonstrates how we would do this.

This subroutine updates the first name, last name and full name like this:

```
proc fcmp outlib=dctable.functions.transform; /*See Note 1*/
subroutine  namSplit(namefield $, delimiter $,out_first_name $, out_last_name $, out_full_name $);/*See Note 2*/
outargs out_first_name , out_last_name, out_full_name );/*See Note 3*/
attrib out_first_name out_last_name format=$26. Out_full_name format=$50.;/*See Note 4*/
name_temp=Scan(namefield,1,'/'); /*See Note 5*/
lname = scan(nametemp,1,delimiter); /*see Note 5*/
fname= scan(nametemp,2,'delimiter); /*See Note 5*/

out_last_name=upcase(left(lname)); /*See Note 6*/
out_first_name=upcase(left(fname)); /*See Note 6*/
out_full_name = trim(out_first_name)||' '||trim(out_last_name); /*See Note 6*/
endsub;
```

Note 1: Need a three level library for subroutines too, especially if you want to keep it in a permanent location.
Note 2: Specify 'subroutine' and give it a name. It will be expecting five parameters, the incoming name, the delimiter, first name, last name and reformatted full name. All of these are character so you must put the $ sign after each one
Note 3: The Outargs statement lists the variables we want to update, in this case first name (out_first_name) , last name(out_last_name) and reformatted full name (out_full_name).
Note 4: The attrib statement makes sure the returned variables are formatted with enough positions
Note 5: We do the name splitting as before
Note 6: the last name, first name and output full name are formatted. Unlike the previous function NameReformat, this code worked in a UNIX environment.

Coding Example 5

To use this in a data step, you'd have to create the first, last and full name variables as character before you use them in the CALL statement. If you do not, they will be created on the fly as numeric causing the subroutine call to fail. The subroutine is called like any other SAS subroutine: CALL NAMESPLIT

```
data name_parse;
set namelist;
attrib out_first_name out_last_name format=$26. Out_full_name format=$50.;
call namesplit(name_field,',',out_first_name, out_last_name,out_full_name);
run;
```

Using Example 5

As stated before, since SAS 9.3, we can do a hash table lookup within a function. This has the huge advantage of not having to define the hash table again in the data step. In order to do this, your hash table should have been saved to a permanent location. A sales tax lookup hash table was set up in the DCTABLE library. Example 6 shows how you would use it in a function

```
proc fcmp outlib=dctable.functions.lookups; /*See Note 1*/
        function sales_tax(instate_abbrev $,incounty_name $,
                incity_name $); /*See Note 2*/
                attrib sales_tax_rate format=z6.4; /*See Note 3*/
                /*See Note 4*/
                declare hash matchstate(dataset:"dctable.sales_tax_lookup");
                rc=matchstate.DefineKey ('STATE_ABBREV','COUNTY_NAME','CITY_NAME');
                rc=matchstate.DefineData ('STATE_ABBREV');
                rc=matchstate.DefineData ('COUNTY_NAME');
                rc=matchstate.DefineData ('CITY_NAME');
                rc=matchstate.DefineData ('SALES_TAX_RATE');
                rc=matchstate.DefineDone ();
                /*See Note 5*/
                STATE_ABBREV = instate_abbrev;
                COUNTY_NAME=incounty_name;
                CITY_NAME=incity_name;
                rc = matchstate.find(); /*See Note 6*/
                        IF RC = 0 THEN
                        DO;
                                return(SALES_TAX_RATE);
                        END;
                else return(999.99); /*See Note 6*/
        endsub; /*See Note 7*/
run;
```

Note 1: We'll put this function with the lookups, the third qualifier.
Note 2: State that it is a function, name it sales_tax. We expect three parameters, state, county and city.
These are have the $ sign after the parameter name signifying character. There is no $ after the parameter list because we are returning a numeric value.
Note 3: Format the outgoing variable as 0.0000.
Note 4: Describe the hash table here - remember it was already set up
Note 5: Set the keys to the incoming parameters and do a lookup. Assign the RC variable to the return code. 0 is a good return code so return the appropriate sales rate.
Note 6: If not found on the table, return 999.99 or you could return 'missing' if you prefer.
Note 7: Still need the 'ENDSUB' ending statement.

Coding Example 6

The next function does the same for the location code. Since the location code is alphanumeric 10, we put a $10 after the parameter list. If you do not specify either incoming parameters or the outgoing value as character when they are, SAS will attempt to use them as numeric. This will result in an error for those coming in a character and missing values for those coming back. This is much like the SAS assigns new variables in a data step. The default format for any new variable is numeric if not otherwise specified.

```
proc fcmp outlib= dctable.functions.lookups;
        function sales_loc(instate_abbrev $,incounty_name $,
        incity_name $) $10; /*See Note 1*/

        declare hash matchstate(dataset:"dctable.sales_tax_lookup");
        rc=matchstate.DefineKey ('STATE_ABBREV','COUNTY_NAME','CITY_NAME');
        rc=matchstate.DefineData ('STATE_ABBREV');
        rc=matchstate.DefineData ('COUNTY_NAME');
        rc=matchstate.DefineData ('CITY_NAME');
        rc=matchstate.DefineData ('LOCATION_CODE'); /*See Note 2*/
        rc=matchstate.DefineDone ();

        STATE_ABBREV = instate_abbrev;
        COUNTY_NAME=incounty_name;
        CITY_NAME=incity_name;
        rc = matchstate.find();
                IF RC = 0 THEN
                DO;
                        return(LOCATION_CODE); /*See Note 3*/
                END;
        else return('999-999-9');
endsub;
run;
```

Note 1: Return a 10 position location code so put in $ 10 after the parameter list.
Note 2: Since we don't care about the sales tax, don't define it. Define location_code instead
Note 3: If the keys are found on the table, return the location code. Otherwise return '999-999-9' or missing if preferred.

Coding Example 7

To use these functions in a data step, we will again run the original table through them

```
data testfunc;
attrib out_sales_tax_rate format=z6.4 out_loc_code format=$10.;
set dctable.SALES_TAX_TABLE;
out_sales_tax_rate=sales_tax(state_abbrev ,county_name,city_name );
out_loc_code=sales_loc(state_abbrev ,county_name,city_name );
run;
```

Using Examples 6 and 7

Instead of calling two functions, could we write a subroutine to update both at the same time? We certainly could. Just include them both In the OUTARGS statement as in example 8.

```
proc fcmp outlib= dctable.functions.lookups;
        subroutine state_lookup_sub(instate_abbrev $,incounty_name $,
        incity_name $,out_sales_tax_rate,out_location_code $);/*See Note 1*/
        outargs out_sales_tax_rate, out_location_code; /*See Note 2*/
        /*See Note 3*/
        declare hash matchstate(dataset:"dctable.sales_tax_lookup");
        rc=matchstate.DefineKey ('STATE_ABBREV','COUNTY_NAME','CITY_NAME');
        rc=matchstate.DefineData ('STATE_ABBREV');
        rc=matchstate.DefineData ('COUNTY_NAME');
        rc=matchstate.DefineData ('CITY_NAME');
        rc=matchstate.DefineData ('SALES_TAX_RATE');
        rc=matchstate.DefineData ('LOCATION_CODE');
        rc=matchstate.DefineDone ();

        STATE_ABBREV = instate_abbrev;
        COUNTY_NAME=incounty_name;
        CITY_NAME=incity_name;
        rc = matchstate.find();

        IF RC = 0 THEN /*See Note 4*/
                DO;
                        out_sales_tax_rate=SALES_TAX_RATE;
                        out_location_code=LOCATION_CODE;
                END;
        else do; /*See Note 5*/
        out_sales_tax_rate=999.99;
        out_location_code='999-999-9';
        end;
endsub;
run;
```

Note 1: Create subroutine state_lookup_sub with five parameters state, county, city, output tax rate and output location code. The output tax rate does not have a $ sign after it because it is numeric.
Note 2: Output two variables which will be updated by this subroutine: out_sales_tax and out_location_code
Note 3: Define the hash table which was set up already. We will do a full definition this time since we are outputting two variables.
Note 4: Return code will be 0 when the record is found on the table so update the output sales tax and location code.
Note 5: Otherwise set the output sales tax rate to 999.99 and location code to '999-999-9'. You might prefer missing values.

Coding Example 8

Using the subroutine in a data step with the original table as input for testing. This is a little more complicated:

```
data testroutine;
attrib out_sales_tax_rate format=6.4 out_location_code format=$10.; /*See Note 1*/
set dctable.sales_tax_table;
call state_lookup_sub(state_abbrev ,county_name, city_name, out_sales_tax_rate, out_location_code);
/*See Note 2*/
run;
```

Note 1: Variables do not have to exist before trying to use the subroutine but if they do not, SAS will create them as numeric, a real problem for the location code
Note 2: Call the state_lookup_sub routine with all of the parms including the ones to be updated.

Using Example 8

Using the hash routine in either of these hides the actual table lookup, precept of object-oriented programing. You can pass this function off to others. They would just need to allocate the library and append the function catalog using the append option. This could easily be done in your autoexec.

## CONCLUSION

Now that you've seen some of the things you can do with SAS, be encouraged to try your own transformation routines. Next time a team member asks for some code, hand them a format, function or subroutine instead. They really will think it's magic!

## ACKNOWLEDGEMENTS

Thank you, Joe and Paul Butkovich for reviewing this paper and presentation

## CONTACT INFORMATION

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

Patricia Hettinger
Oakbrook Terrace, IL 60523
Phone: 331-462-2142, cell 630-309-3431
Email: patricia_hettinger@att.net