Implementing Look-up Tables in SAS

You are sending a thank you note to your grandmother in Texas, and on the envelope you represent the state by TX. An unknown person calls you on the phone and you see that the area code is 803, and you wonder what part of the US has an 803 area code. For all types of information, geographic and otherwise, we commonly need to translate from one representation of information to another.

Computers are useful in performing these translations because they can easily hold lists or tables of code/value pairs for translating between one form and another. If you researched "look-up tables" in a general computer science context, you would see discussions of speed and efficiency as reasons for look-up table implementation. In the diverse world of SAS applications, sometimes our goal would be program speed (manipulate codes rather than longer values) and sometimes it would be display clarity (substitute a value for a code) and sometimes it would be ease of future program maintenance by our co-workers. Given your knowledge of SAS as a rich toolkit for data manipulation, you might imagine that SAS offers many ways to do table look-up, and that is in fact the case.

Potential SAS application examples:

- Examine ICD-9 or ICD-10 codes on hospital discharge records to identify cases of diabetes (look-up table links ICD codes with diseases)
- Identify medical events that occurred on a holiday (look-up table links dates with holiday names)
- For an annual climate summary table, replace weather station codes with geographic place names (look-up table links codes such as PBZ with place names such as Pittsburgh)

Using the ADMITS and HOSPITALS data sets that were used in the PROC SQL course notes, this chapter illustrates several ways of implementing and using look-up tables in SAS. The goal in each of the chapter's examples is to add the hospital name to each admission record, when what is on each admission record now is the hospital code (variable Hosp). In this situation, the HOSPITALS data set can be considered a look-up table since it contains the hospital code/hospital name pairs (Hosp_ID/HospName).

A valuable reference paper on look-up tables and SAS is Jane Stroupe and Linda Jolley's *Using Table Lookup Techniques Efficiently*

(http://www2.sas.com/proceedings/forum2008/095-2008.pdf), and I highly recommend you take a look at that paper. [Two notes on techniques used in that paper: (1) A colon (:) in an INPUT statement enables you to use list input but also specify an informat. (2) The DSD option on an INFILE statement specifies that when data values are enclosed in quotation marks, delimiters within the value are treated as character data. Quotation marks are removed from the values after they are read in.]

A separate document posted for this unit summarizes strengths and weaknesses of various methods of implementing look-up tables in SAS to help you choose between them for future applications.

In an associated exercise, you will practice using several of these methods for a METS task.

Some of these look-up table techniques are simply a new application of, or a different way of looking at, a SAS technique you already know. Others involve new syntax that will be briefly introduced as necessary below.

Scenario

The hospital admissions data set ADMITS contains the hospital code of the hospital where the patient was treated, with the code stored in variable Hosp. Data set HOSPITALS contains information about most hospitals where a patient might have been admitted, including the hospital code (variable Hosp_ID) and hospital name (variable HospName). You need to prepare a data set in which the appropriate hospital name has been added to each hospital admission record.

In the examples below, the ADMITS data set is referred to as the original or admissions data, and the HOSPITALS data set is referred to as the table data. Note that the table data set HOSPITALS contains additional variables in addition to the code/value pairs (Hosp_ID/HospName), and we need to be careful to omit those other variables (Town, NBeds, and Type) in some of the examples.

Data step MERGE

A data step MERGE can be a great way of implementing a table look-up. You already know all about this technique, and there will be no surprises in the code below. The key is merging the original data (ADMITS) with the table data (HOSPITALS) by the code variable, keeping only the code variable and the value variable from the table data (to avoid bringing in information that we don't need and that might cause trouble). Also important here is using an IN flag on the ADMITS data set and keeping all records from there, whether or not they have a match in the table data (in fact you'll see from the listing below that hospital code 9 appears in the original data but not in the table data).

```
* DATA step MERGE;

PROC SORT DATA=ex.admits(RENAME=(hosp=hosp_id))
          OUT=admits;
    BY hosp_id;
RUN;

DATA merge_lu;
    MERGE admits(IN=inmain) ex.hospitals(KEEP=hosp_id hospname);
    BY hosp_id;
    IF inmain;
RUN;

PROC SORT DATA=merge_lu(RENAME=(hosp_id=Hosp));
    BY pt_id admdate;
RUN;
```

ADMITS data set (selected variables, first 15 records)

Obs	PT_ID	AdmDate	DisDate	Hosp	PrimDX
1	001	07FEB2007	08FEB2007	1	410.0
2	001	12APR2007	25APR2007	1	428.2
3	001	10SEP2007	19SEP2007	2	813.90
4	001	06JUN2008	12JUN2008	2	428.4
5	003	15MAR2007	15MAR2007	3	431
6	004	18JUN2007	24JUN2007	2	434.1
7	005	19JAN2007	22JAN2007	1	411.81
8	005	10MAR2007	18MAR2007	1	410.9
9	005	10APR2007	14APR2007	2	411.0

Obs	PT_ID	AdmDate	DisDate	Hosp	PrimDX
10	007	28JUL2007	10AUG2007	2	155.0
11	007	08SEP2007	15SEP2007	2	155.0
12	008	01OCT2007	15OCT2007	3	820.8
13	008	26NOV2007	28NOV2007	3	V54.8
14	008	10DEC2007	12DEC2007	9	V54.8
15	009	15DEC2007	04JAN2008	2	410.1

Same admissions records, now with hospital name added

Obs	PT_ID	AdmDate	DisDate	Hosp	PrimDX	HospName
1	001	07FEB2007	08FEB2007	1	410.0	University Hospital
2	001	12APR2007	25APR2007	1	428.2	University Hospital
3	001	10SEP2007	19SEP2007	2	813.90	Our Lady of Peace
4	001	06JUN2008	12JUN2008	2	428.4	Our Lady of Peace
5	003	15MAR2007	15MAR2007	3	431	Veteran's Administration
6	004	18JUN2007	24JUN2007	2	434.1	Our Lady of Peace
7	005	19JAN2007	22JAN2007	1	411.81	University Hospital
8	005	10MAR2007	18MAR2007	1	410.9	University Hospital
9	005	10APR2007	14APR2007	2	411.0	Our Lady of Peace
10	007	28JUL2007	10AUG2007	2	155.0	Our Lady of Peace
11	007	08SEP2007	15SEP2007	2	155.0	Our Lady of Peace
12	008	01OCT2007	15OCT2007	3	820.8	Veteran's Administration
13	008	26NOV2007	28NOV2007	3	V54.8	Veteran's Administration
14	008	10DEC2007	12DEC2007	9	V54.8	
15	009	15DEC2007	04JAN2008	2	410.1	Our Lady of Peace

SQL Join

Since a SQL join has a lot in common with a DATA step MERGE, it makes sense that a join can also be used for table look-up. Since we just covered SQL joins, the code below should seem familiar to you. Two code sequences are included since the first one, the inner join, does not quite take care of things adequately. Recall that an inner join produces the intersection of the two data sets, and some Hosp codes in the original data do not have a corresponding Hosp_ID in the table data (why we needed to use an IN flag in the MERGE section above). The RIGHT JOIN done in the second code sequence takes care of this important detail, as it says to keep all records in the second or right-most data set (ADMITS) whether or not they have a match in the first.

```
* SQL inner join;

PROC SQL;
    CREATE TABLE innerjoin_lu(drop=hosp_id) AS
        SELECT *
            FROM ex.admits AS A,
                 ex.hospitals(drop=town nbeds type) AS H
            WHERE a.hosp=h.hosp_id;
QUIT;


* the inner join above excludes admits to a hospital not in hospitals table;
* do right join to keep all records that are in ex.admits;
PROC SQL;
    CREATE TABLE rightjoin_lu(drop=hosp_id) AS
        SELECT *
            FROM ex.hospitals(drop=town nbeds type) AS H
                RIGHT JOIN
                 ex.admits AS A
            ON a.hosp=h.hosp_id
            ORDER BY pt_id, admdate;
QUIT;
```

If you run both code sequences, you'll see that the first results in a new data set with 23 rows, while the second gives the correct 25 rows. Each has the appropriate 11 variables (the 10 original variables in ADMITS plus HospName).

IF/THEN (or CASE in SQL)

A series of IF/THEN statements (or the SQL SELECT and CASE equivalent) is an obvious way to program code/value pairs. This technique does not take advantage of the fact that we already have a data set that contains the table data – we have to type in the information.

```
* IF/THEN;

DATA ifthen_lu;
    SET ex.admits;

    LENGTH HospName $30;

    IF hosp=1 THEN hospname='University Hospital';
    IF hosp=2 THEN hospname='Our Lady of Peace';
    IF hosp=3 THEN hospname="Veteran's Administration";
    IF hosp=4 THEN hospname='Community Hospital';
    IF hosp=5 THEN hospname='City Hospital';
    IF hosp=6 THEN hospname="Children's Hospital";

RUN;



* SQL with CASE;

PROC SQL;

    CREATE TABLE sqlcase_lu AS
        SELECT *,
                CASE
                    WHEN hosp=1 THEN 'University Hospital'
                    WHEN hosp=2 THEN 'Our Lady of Peace'
                    WHEN hosp=3 THEN "Veteran's Administration"
                    WHEN hosp=4 THEN 'Community Hospital'
                    WHEN hosp=5 THEN 'City Hospital'
                    WHEN hosp=6 THEN "Children's Hospital"
                    ELSE ' '
                END AS HospName LENGTH=30
            FROM ex.admits;

QUIT;
```

Using a format and PUT – manual format definition

Of course you are familiar with creating a user-defined format, and using the PUT function with a format to create a new character variable was covered in BIOS 511. As with the IF/THEN technique, this table look-up method involves typing in the code/value pairs and does not take advantage of the fact that they already exist in our table data set.

```
* format - manual;

PROC FORMAT;
    VALUE hnm 1='University Hospital'
              2='Our Lady of Peace'
              3="Veteran's Administration"
              4='Community Hospital'
              5='City Hospital'
              6="Children's Hospital"
          other=' ';
RUN;

DATA fmtmanual_lu;
    SET ex.admits;
    LENGTH HospName $30;
    HospName=PUT(hosp,hnm.);
RUN;
```

Using a format and PUT – defining the format with a data set

Fortunately, we can create a format from a SAS data set, overcoming the weakness of the format-based technique above. Once we do that, we can use the PUT function as in the previous section.

A format definition data set must contain variables FMTNAME, START, and LABEL. The value of character variable FMTNAME should be the name of the format to be created. If the format will be applied to a character variable, the value of variable FMTNAME should begin with $. The values of variable START should be the range values (these would typically be your codes). The value of character variable LABEL should be the string value that you want to correspond to the START value on the same observation.

Once we have a data set in the right form, we can use PROC FORMAT's CNTLIN option to read in the data set and create a format named with the value of the FMTNAME variable. Then that format is available for use just like any other format.

In this example, we start by making data set FMT from the table data HOSPITALS. We use a RETAIN statement to add a FMTNAME variable to every record with the value HND, which will be our format name. We rename Hosp_ID to START and HospName to LABEL and drop the remaining variables, making what should be a good CNTLIN data set. Here's data set that results from running the FMT DATA step below:

Obs	start	label	fmtname
1	1	University Hospital	hnd
2	2	Our Lady of Peace	hnd
3	3	Veteran's Administration	hnd
4	4	Community Hospital	hnd
5	5	City Hospital	hnd
6	6	Children's Hospital	hnd

In general, all we would need to do to create the new HND format suitable for use in PUTting Hosp values to make HospName would be this:

```
PROC FORMAT CNTLIN=fmt;
RUN;
```

We need the more complicated PROC FORMAT code below because our PUT statement also needs to be able to handle Hosp codes of 9, which aren't in the HND format. So we make a new format HNAME that is basically HND with a length of 30 (HND30.), along with value 9 that should be translated to blank.

After that, we can PUT Hosp with the HNAME format to make HospName.

```
* format - from data set;

DATA fmt;
    SET ex.hospitals(RENAME=(hosp_id=START hospname=LABEL));
    RETAIN FMTNAME 'hnd';
    DROP town nbeds type;
RUN;

PROC FORMAT CNTLIN=fmt;
    VALUE hname     9=' '
                other=[hnd30.];
RUN;

DATA fmtds_lu;
    SET ex.admits;
    LENGTH HospName $30;
    HospName=PUT(hosp,hname.);
RUN;
```

Using an array – manual definition

Arrays: Always everyone's favorite SAS topic! But they are useful in many situations to simplify your code. In this table look-up context, use of an array has the same weakness as using IF/THEN or creating a format manually: defining the array involves typing in information and does not take advantage of the fact that the code/value pairs already exist in our table data set.

The code below should look familiar to you from BIOS 511. The HN array is created as _TEMPORARY_ since the values aren't variables and we don't want to create variables – it is simply useful to have the strings in an array so that we can call them with an index (which is the corresponding code value).

```
* array - manual;

DATA arraymanual_lu;
    SET ex.admits;

    LENGTH HospName $30;

    ARRAY hn {6} $30 _TEMPORARY_ ('University Hospital',
                                  'Our Lady of Peace',
                                  "Veteran's Administration",
                                  'Community Hospital',
                                  'City Hospital',
                                  "Children's Hospital");

    DO i=1 TO 6;
        IF hosp=i THEN HospName=hn{i};
    END;

    DROP i;
RUN;
```

Using an array – creating the array from a data set

Fortunately, we can create an array from a data set.

I won't explain all aspects of the code below, but it works beautifully. However, if you are inclined to define and use an array in this way, you might as well use a hash object, as described in the next section.

```
* array - from data set;

DATA arrayds_lu;

    LENGTH HName $30;

    IF _n_=1 THEN DO i=1 TO numobs;
        SET ex.hospitals(KEEP=hosp_id hospname) NOBS=numobs;
        ARRAY hnames{1:6} $30 _TEMPORARY_;
        hnames(hosp_id)=hospname;
    END;

    SET ex.admits;

    IF hosp=9 THEN HName=' ';    /* special case since code 9 not in HOSPITALS */
    ELSE HName=hnames{hosp};

    DROP i hosp_id hospname;
RUN;
```

Using a hash object

Hash objects are a relatively new SAS feature explicitly intended to be used for look-up purposes using keys. Notice that the code below looks very similar to the code above for filling an array based on a data set; a hash object is ALWAYS based on a data set.

A hash object is defined in a DATA step with code that looks object-oriented. During the first iteration of the DATA step loop (IF _N_=1), the hash object is created as follows.

- In an initial DECLARE statement, you name the hash object and tell SAS what data set contains the table to be used to fill it. Here, the hash name is HN, and it is to be based on the HOSPITALS data set.
- The HN.DEFINEKEY call says that the keys of the HN object should be the values of the Hosp_ID variable (our code variable).
- The HN.DEFINEDATA call says that the values to associate with those keys are the values of the HospName variable (our value variable).
- HN.DEFINEDONE() and CALL MISSING(codevar,valuevar) simply tell SAS that you are finished defining the hash object and take care of some clean up.

Later on in the DATA step, as it is looping through records of the ADMITS data set, HN.FIND(KEY: Hosp) tells SAS to look for the current Hosp variable value among the keys of the hash object HN. If that code value is found, then the HospName variable is filled in with the corresponding value from the hash.

Note that the initial LENGTH statement is needed to make sure that HospName in the new data set is a character variable of the appropriate length. If Hosp_ID had been character, a LENGTH statement for it would have been needed as well.

```
* hash;

DATA hash_lu;

    LENGTH HospName $30;

    IF _n_=1 THEN DO;
        DECLARE HASH hn(DATASET:'ex.hospitals');
        hn.DEFINEKEY('Hosp_ID');
        hn.DEFINEDATA('HospName');
        hn.DEFINEDONE();
        CALL MISSING(Hosp_ID,HospName);
    END;

    SET ex.admits;

    rc=hn.FIND(KEY:Hosp);
```

```
    DROP rc hosp_id;
RUN;
```