Cody's Data Cleaning Techniques

Using SAS®

Second Edition

Ron Cody

SAS® Press Series

Table of Contents

5 Looking for Duplicates and "n" Observations per Subject

6 Working with Multiple Files

7 Double Entry and Verification (PROC COMPARE)

8 Some PROC SQL Solutions to Data Cleaning

9 Correcting Errors

10 Creating Integrity Constraints and Audit Trails

11 DataFlux and dfPower Studio

Appendix Listing of Raw Data Files and SAS Programs

Index

1 Checking Values of Character Variables

Introduction

There are some basic operations that need to be routinely performed when dealing with character data values. You may have a character variable that can take on only certain allowable values, such as 'M' and 'F' for gender. You may also have a character variable that can take on numerous values but the values must fit a certain pattern, such as a single letter followed by two or three digits. This chapter shows you several ways that you can use SAS software to perform validity checks on character variables.

Using PROC FREQ to List Values

This section demonstrates how to use PROC FREQ to check for invalid values of a character variable. In order to test the programs you develop, use the raw data file PATIENTS.TXT, listed in the Appendix. You can use this data file and, in later sections, a SAS data set created from this raw data file for many of the examples in this text.

You can download all the programs and data files used in this book from the SAS Web site: http://support.sas.com/publishing. Click the link for SAS Press Companion Sites and select *Cody's Data Cleaning Techniques Using SAS, Second Edition*. Finally, click the link for Example Code and Data and you can download a text file containing all of the programs, macros, and text files used in this book.

Description of the Raw Data File PATIENTS.TXT

The raw data file PATIENTS.TXT contains both character and numeric variables from a typical clinical trial. A number of data errors were included in the file so that you can test the data cleaning programs that are developed in this text. Programs, data files, SAS data sets, and macros used in this book are stored in the folder C:\BOOKS\CLEAN. For example, the file PATIENTS.TXT is located in a folder (directory) called C:\BOOKS\CLEAN. You will need to modify the INFILE and LIBNAME statements to fit your own operating environment.

Here is the layout for the data file PATIENTS.TXT.

Variable Name	Description	Starting Column	Length	Variable Type	Valid Values
Patno	Patient Number	1	3	Character	Numerals only
Gender	Gender	4	1	Character	'M' or 'F'
Visit	Visit Date	5	10	MMDDYY10.	Any valid date
HR	Heart Rate	15	3	Numeric	Between 40 and 100
SBP	Systolic Blood Pressure	18	3	Numeric	Between 80 and 200
DBP	Diastolic Blood Pressure	21	3	Numeric	Between 60 and 120
Dx	Diagnosis Code	24	3	Character	1 to 3 digit numeral
AE	Adverse Event	27	1	Character	'0' or '1'

There are several character variables that should have a limited number of valid values. For this exercise, you expect values of Gender to be 'F' or 'M', values of Dx the numerals 1 through 999, and values of AE (adverse events) to be '0' or '1'. A very simple approach to identifying invalid character values in this file is to use PROC FREQ to list all the unique values of these variables. Of course, once invalid values are identified using this technique, other means will have to be employed to locate specific records (or patient numbers) containing the invalid values.

Use the program PATIENTS.SAS (shown next) to create the SAS data set PATIENTS from the raw data file PATIENTS.TXT (which can be downloaded from the SAS Web site or found listed in the Appendix). This program is followed with the appropriate PROC FREQ statements to list the unique values (and their frequencies) for the variables Gender, Dx, and AE.

Program 1-1 Writing a Program to Create the Data Set PATIENTS

```
*-----------------------------------------------------------*
|PROGRAM NAME: PATIENTS.SAS in C:\BOOKS\CLEAN               |
|PURPOSE: To create a SAS data set called PATIENTS          |
*-----------------------------------------------------------*;
libname clean "c:\books\clean";

data clean.patients;
   infile "c:\books\clean\patients.txt" truncover /* take care of problems
                                                     with short records */;

   input @1  Patno    $3.
         @4  Gender   $1.
         @5  Visit    mmddyy10.
         @15 Hr       3.
         @18 SBP      3.
         @21 DBP      3.
         @24 Dx       $3.
         @27 AE       $1.;

   LABEL Patno   = "Patient Number"
         Gender  = "Gender"
         Visit   = "Visit Date"
         HR      = "Heart Rate"
         SBP     = "Systolic Blood Pressure"
         DBP     = "Diastolic Blood Pressure"
         Dx      = "Diagnosis Code"
         AE      = "Adverse Event?";
    format visit mmddyy10.;
run;
```

The DATA step is straightforward. Notice the TRUNCOVER option in the INFILE statement. This will seem foreign to most mainframe users. If you do not use this option and you have short records, SAS will, by default, go to the next record to read data. The TRUNCOVER option prevents this from happening. The TRUNCOVER option is also useful when you are using list input (delimited data values). In this case, if you have more variables on the INPUT statement than there are in a single record on the data file, SAS will supply a missing value for all the remaining variables. One final note about INFILE options: If you have long record lengths (greater than 256 on PCs and UNIX platforms) you need to use the LRECL= option to change the default logical record length.

Next, you want to use PROC FREQ to list all the unique values for your character variables. To simplify the output from PROC FREQ, use the NOCUM (no cumulative statistics) and NOPERCENT (no percentages) TABLES options because you only want frequency counts for each of the unique character values. (Note: Sometimes the percent and cumulative statistics can be useful—the choice is yours.) The PROC statements are shown in Program 1-2.

Program 1-2 Using PROC FREQ to List All the Unique Values for Character Variables

```
title "Frequency Counts for Selected Character Variables";
proc freq data=clean.patients;
   tables Gender Dx AE / nocum nopercent;
run;
```

Here is the output from running Program 1-2.

```
Frequency Counts for Selected Character Variables

The FREQ Procedure

      Gender

Gender    Frequency
-------------------
2                 1
F                12
M                14
X                 1
f                 2

Frequency Missing = 1


Diagnosis Code

Dx    Frequency
---------------
1             7
2             2
3             3
4             3
5             3
6             1
7             2
X             2

Frequency Missing = 8
```

(*continued*)

```
Adverse Event?

AE    Frequency
---------------
0           19
1           10
A            1

Frequency Missing = 1
```

Let's focus in on the frequency listing for the variable Gender. If valid values for Gender are 'F', 'M', and missing, this output would point out several data errors. The values '2' and 'X' both occur once. Depending on the situation, the lowercase value 'f' may or may not be considered an error. If lowercase values were entered into the file by mistake, but the value (aside from the case) was correct, you could change all lowercase values to uppercase with the UPCASE function. More on that later. The invalid Dx code of 'X' and the adverse event of 'A' are also easily identified. At this point, it is necessary to run additional programs to identify the location of these errors. Running PROC FREQ is still a useful first step in identifying errors of these types, and it is also useful as a last step, after the data have been cleaned, to ensure that all the errors have been identified and corrected.

For those users who like shortcuts, here is another way to have PROC FREQ select the same set of variables in the example above, without having to list them all.

Program 1-3 Using the Keyword _CHARACTER_ in the TABLES Statement

```
title "Frequency Counts for Selected Character Variables";
proc freq data=clean.patients(drop=Patno);
   tables _character_ / nocum nopercent;
run;
```

The keyword _CHARACTER_ in this example is equivalent to naming all the character variables in the CLEAN.PATIENTS data set. Since you don't want the variable Patno included in this list, you use the DROP= data set option to remove it from the list.

Using a DATA Step to Check for Invalid Values

Your next task is to use a DATA step to identify invalid data values and to determine where they occur in the raw data file (by listing the patient number).

This time, DATA step processing is used to identify invalid character values for selected variables. As before, you will check Gender, Dx, and AE. Several different methods are used to identify these values.

First, you can write a simple DATA step that reports invalid data values by using PUT statements in a DATA _NULL_ step. Here is the program.

Program 1-4 Using a DATA _NULL_ Step to Detect Invalid Character Data

```
title "Listing of invalid patient numbers and data values";
data _null_;
   set clean.patients;
   file print; ***send output to the output window;
   ***check Gender;
   if Gender not in ('F' 'M' ' ') then put Patno= Gender=;
   ***check Dx;
   if verify(trim(Dx),'0123456789') and not missing(Dx)
      then put Patno= Dx=;
   /***********************************************
   SAS 9 alternative:
   if notdigit(trim(Dx)) and not missing(Dx)
      then put Patno= Dx=;
   ************************************************/

   ***check AE;
   if AE not in ('0' '1' ' ') then put Patno= AE=;
run;
```

Before discussing the output, let's spend a moment looking over the program. First, notice the use of the DATA _NULL_ statement. Because the only purpose of this program is to identify invalid data values and print them out, there is no need to create a SAS data set. The reserved data set name _NULL_ tells SAS not to create a data set. This is a major efficiency technique. In this program, you avoid using all the resources to create a data set when one isn't needed.

The FILE PRINT statement causes the results of any subsequent PUT statements to be sent to the Output window (or output device). Without this statement, the results of the PUT statements would be sent to the SAS Log. Gender and AE are checked by using the IN operator. The statement

```
if X in ('A' 'B' 'C') then . . .;
```

is equivalent to

```
if X = 'A' or X = 'B' or X = 'C' then . . .;
```

That is, if X is equal to any of the values in the list following the IN operator, the expression is evaluated as true. You want an error message printed when the value of Gender is not one of the acceptable values ('F', 'M', or missing). Therefore, place a NOT in front of the whole expression, triggering the error report for invalid values of Gender or AE. You can separate the values in the list by spaces or commas. An equivalent statement to the one above is:

```
if X in ('A','B','C') then . . .;
```

There are several alternative ways that the gender checking statement can be written. The method above uses the IN operator.

A straightforward alternative to the IN operator is

```
if not (Gender eq 'F' or Gender eq 'M' or Gender = ' ') then
put Patno= Gender=;
```

Another possibility is

```
if Gender ne 'F' and Gender ne 'M' and Gender ne ' ' then
put Patno= Gender=;
```

While all of these statements checking for Gender and AE produce the same result, the IN operator is probably the easiest to write, especially if there are a large number of possible values to check. Always be sure to consider whether you want to identify missing values as invalid or not. In the statements above, you are allowing missing values as valid codes. If you want to flag missing values as errors, do not include a missing value in the list of valid codes.

If you want to allow lowercase M's and F's as valid values, you can add the single line

```
Gender = upcase(Gender);
```

immediately before the line that checks for invalid gender codes. As you can probably guess, the UPCASE function changes all lowercase letters to uppercase letters.

If you know from the start that you may have both upper- and lowercase values in your raw data file, you could use the $UPCASE informat to convert all lowercase values to uppercase. For example, to read all Gender values in uppercase, you could use:

```
@4 Gender $upcase1.
```

to replace the line that reads Gender values in Program 1-1.

A statement similar to the gender checking statement is used to test the adverse events.

There are so many valid values for Dx (any numeral from 1 to 999) that the approach you used for Gender and AE would be inefficient (and wear you out typing) if you used it to check for invalid Dx codes. The VERIFY function is one of the many possible ways you can check to see if there is a value other than the numerals 0 to 9 as a Dx value. The next section describes the VERIFY function along with several other functions.

Describing the VERIFY, TRIM, MISSING, and NOTDIGIT Functions

The verify function takes the form:

```
verify(character_variable,verify_string)
```

where *verify_string* is a character value (either the name of a character variable or a series of values placed in single or double quotes). The VERIFY function returns the first position in the *character_variable* that contains a character that is not in the *verify_string*. If the *character_variable* does not contain any invalid values, the VERIFY function returns a 0. To make this clearer, let's look at some examples of the VERIFY function.

Suppose you have a variable called ID that is stored in five bytes and is supposed to contain only the letters X, Y, Z, and digits 0 through 5. For example, valid values for ID would be X1234 or 34Z5X. You could use the VERIFY function to see if the variable ID contained any characters other than X, Y, Z and the digits 0 through 5 like this:

```
Position = verify(ID,'XYZ012345');
```

Suppose you had an ID value of X12B44. The value of Position in the line above would be 4, the position of the first invalid character in ID (the letter B). If no invalid characters are found, the VERIFY function returns a 0. Therefore, you can write an expression like the following to list invalid values of ID:

```
if verify(ID,'XYZ012345') then put "Invalid value of ID:" ID;
```

This may look strange to you. You might prefer the statement:

```
if verify(ID,'XYZ012345') gt 0 then put "Invalid value of ID:" ID;
```

However, these two statements are equivalent. Any numerical value in SAS other than 0 or missing is considered TRUE. You usually think of true and false values as 1 or 0—and that is what SAS returns to you when it evaluates an expression. However, it is often convenient to use values other than 1 to represent TRUE. When SAS evaluates the VERIFY function in either of the two statements above, it returns a 4 (the position of the first invalid character in the ID). Since 4 is neither 0 or missing, SAS interprets it as TRUE and the PUT statement is executed.

There is one more potential problem when using the VERIFY function. Suppose you had an ID equal to 'X123'. What would the expression

```
verify(ID,'XYZ012345')
```

return? You might think the answer is 0 since you only see valid characters in the ID (X, 1, 2, and 3). However, the expression above returns a 5! Why? Because that is the position of the first trailing blank. Since ID is stored in 5 bytes, any ID with fewer than 5 characters will contain trailing blanks—and blanks, even though they are sometimes hard to see, are still considered characters to be tested by the VERIFY function.

To avoid problems with trailing blanks, you can use the TRIM function to remove any trailing blanks before the VERIFY function operates. Therefore, the expression

```
verify(trim(ID),'XYZ012345')
```

will return a 0 for all valid values of ID, even if they are shorter than 5 characters.

There is one more problem to solve. That is, the expression above will return a 1 for a missing value of ID. (Think of character missing values as blanks). The MISSING function is a useful way to test for missing values. It returns a value of TRUE if its argument contains a missing value and a value of FALSE otherwise. And, this function can take character or numeric arguments! The MISSING function has become one of this author's favorites. It makes your SAS programs much more readable. For example, take the line in Program 1-4 that uses the MISSING function:

```
if verify(trim(Dx),'0123456789') and not missing(Dx)
   then put Patno= Dx=;
```

Without the MISSING function, this line would read:

```
if verify(trim(Dx),'0123456789') and Dx ne ' '
   then put Patno= Dx=;
```

If you start using the MISSING function in your SAS programs, you will begin to see statements like the one above as clumsy or even ugly.

You are now ready to understand the VERIFY function that checked for invalid Dx codes. The verify string contained the characters (numerals) 0 through 9. Thus, if the Dx code contains any character other than 0 through 9, it returns the position of this offending character, which would have to be a 1, 2, or 3 (Dx is three bytes in length), and the error message would be printed. Output from Program 1-4 is shown below:

```
Listing of invalid patient numbers and data values
Patno=002 Dx=X
Patno=003 gender=X
Patno=004 AE=A
Patno=010 gender=f
Patno=013 gender=2
Patno=002 Dx=X
Patno=023 gender=f
```

Note that patient 002 appears twice in this output. This occurs because there is a duplicate observation for patient 002 (in addition to several other purposely included errors), so that the data set can be used for examples later in this book, such as the detection of duplicate ID's and duplicate observations.

If you have SAS 9 or higher, you can use the NOTDIGIT function.

```
notdigit(character_value)
```

is equivalent to

```
verify(character_value,'0123456789')
```

That is, the NOTDIGIT function returns the first position in *character_value* that is not a digit. The NOTDIGIT function treats trailing blanks the same way that the VERIFY function does, so if you have character strings of varying lengths, you may want to use the TRIM function to remove trailing blanks.

Using the NOTDIGIT function, you could replace the VERIFY function in Program 1-4 like this:

```
if notdigit(trim(Dx)) and not missing(Dx)
   then put Patno= Dx=;
```

Suppose you want to check for valid patient numbers (Patno) in a similar manner. However, you want to flag missing values as errors (every patient must have a valid ID). The following statement:

```
if notdigit(trim(Patno)) then put "Invalid ID for PATNO=" Patno;
```

will work in the same way as your check for invalid Dx codes except that missing values will now be listed as errors.

Using PROC PRINT with a WHERE Statement to List Invalid Values

There are several alternative ways to identify the ID's containing invalid data. As with most of the topics in this book, you will see several ways of accomplishing the same task. Why? One reason is that some techniques are better suited to an application. Another reason is to teach some additional SAS programming techniques. Finally, under different circumstances, some techniques may be more efficient than others.

One very easy alternative way to list the subjects with invalid data is to use PROC PRINT followed by a WHERE statement. Just as you used an IF statement in a DATA step in the previous section, you can use a WHERE statement in a similar manner with PROC PRINT and avoid having to write a DATA step altogether. For example, to list the ID's with invalid GENDER values, you could write a program like the one shown in Program 1-5.

Program 1-5 Using PROC PRINT to List Invalid Character Values

```
title "Listing of invalid gender values";
proc print data=clean.patients;
   where Gender not in ('M' 'F' ' ');
   id Patno;
   var Gender;
run;
```

It's easy to forget that WHERE statements can be used within SAS procedures. SAS programmers who have been at it for a long time (like the author) often write a short DATA step first and use PUT statements or create a temporary SAS data set and follow it with a PROC PRINT. The program above is both shorter and more efficient than a DATA step followed by a PROC PRINT. However, the WHERE statement does require that all variables already exist in the data set being processed. DATA _NULL_ steps, however, tend to be fairly efficient and are a reasonable alternative as well as the more flexible approach.

The output from Program 1-5 follows.

```
Listing of invalid gender values

Patno    gender

 003       X
 010       f
 013       2
 023       f
```

This program can be extended to list invalid values for all the character variables. You simply add the other invalid conditions to the WHERE statement as shown in Program 1-6.

Program 1-6 Using PROC PRINT to List Invalid Character Data for Several Variables

```
title "Listing of invalid character values";
proc print data=clean.patients;
   where Gender not in ('M' 'F' ' ')           or
         notdigit(trim(Dx)) and not missing(Dx) or
         AE not in ('0' '1' ' ');
   id Patno;
   var Gender Dx AE;
run;
```

The resulting output is shown next.

```
Listing of invalid character values

Patno    Gender    Dx    AE

 002       F       X     0
 003       X       3     1
 004       F       5     A
 010       f       1     0
 013       2       1
 002       F       X     0
 023       f             0
```

Notice that this output is not as informative as the one produced by the DATA _NULL_ step in Program 1-4. It lists all the patient numbers, genders, Dx codes, and adverse events even when only one of the variables has an error (patient 002, for example). So, there is a trade-off—the simpler program produces slightly less desirable output. We could get philosophical and extend this concept to life in general, but that's for some other book.

Using Formats to Check for Invalid Values

Another way to check for invalid values of a character variable from raw data is to use user-defined formats. There are several possibilities here. One, you can create a format that leaves all valid character values as is and formats all invalid values to a single error code. Let's start out with a program that simply assigns formats to the character variables and uses PROC FREQ to list the number of valid and invalid codes. Following that, you will extend the program by using a DATA step to identify which ID's have invalid values. Program 1-7 uses formats to convert all invalid data values to a single value.

Program 1-7 Using a User-Defined Format and PROC FREQ to List Invalid Data Values

```
proc format;
   value $gender 'F','M' = 'Valid'
                 ' '     = 'Missing'
                 other   = 'Miscoded';
```

```
   value $ae '0','1' = 'Valid'
             ' '     = 'Missing'
              other  = 'Miscoded';
run;

title "Using formats to identify invalid values";
proc freq data=clean.patients;
   format Gender $gender.
          AE     $ae.;
   tables Gender AE/ nocum nopercent missing;
run;
```

For the variables GENDER and AE, which have specific valid values, you list each of the valid values in the range to the left of the equal sign in the VALUE statement. Format each of these values with the value 'Valid'.

You may choose to combine the missing value with the valid values if that is appropriate, or you may want to keep track of missing values separately as was done here. Finally, any value other than the valid values or a missing value will be formatted as 'Miscoded'. All that is left is to run PROC FREQ to count the number of 'Valid', 'Missing', and 'Miscoded' values. The TABLES option MISSING causes the missing values to be listed in the body of the PROC FREQ output. (Important note: When you use the MISSING TABLES option with PROC FREQ and you are outputting percentages, the percentages are computed by dividing the number of a particular value by the total number of observations, missing or non-missing.) Here is the output from PROC FREQ.

```
Using formats to identify invalid values

The FREQ Procedure

       Gender

Gender      Frequency
_____________________
Missing            1
Miscoded           4
Valid             26


   Adverse Event?

AE          Frequency
_____________________
Missing            1
Valid             29
Miscoded           1
```

This output isn't particularly useful. It doesn't tell you which observations (patient numbers) contain missing or invalid values. Let's modify the program by adding a DATA step, so that ID's with invalid character values are listed.

Program 1-8 Using a User-Defined Format and a DATA Step to List Invalid Data Values

```
proc format;
   value $gender 'F','M' = 'Valid'
                 ' '     = 'Missing'
                 other   = 'Miscoded';

   value $ae '0','1' = 'Valid'
             ' '     = 'Missing'
              other  = 'Miscoded';
run;

title "Listing of invalid patient numbers and data values";
data _null_;
   set clean.patients(keep=Patno Gender AE);
   file print; ***Send output to the output window;
```

```
   if put(Gender,$gender.) = 'Miscoded' then put Patno= Gender=;
   if put(AE,$ae.) = 'Miscoded' then put Patno= AE=;
run;
```

The "heart" of this program is the PUT function. To review, the PUT function is similar to the INPUT function. It takes the following form:

```
character_variable = put(variable, format)
```

where *character_variable* is a character variable that contains the value of the variable listed as the first argument to the function, formatted by the *format* listed as the second argument to the function. The result of a PUT function is always a character variable, and the function is frequently used to perform numeric-to-character conversions. In Program 1-8, the first argument of the PUT function is a character variable you want to test and the second argument is the corresponding character format. The result of the PUT function for any invalid data values would be the value 'Miscoded'.

Here is the output from Program 1-8.

```
Listing of invalid patient numbers and data values
Patno=003 gender=X
Patno=004 AE=A
Patno=010 gender=f
Patno=013 gender=2
Patno=023 gender=f
```

Using Informats to Remove Invalid Values

PROC FORMAT is also used to create informats. Remember that formats are used to control how variables look in output or how they are classified by such procedures as PROC FREQ. Informats modify the value of variables as they are read from the raw data, or they can be used with an INPUT function to create new variables in the DATA step. User-defined informats are created in much the same way as user-defined formats. Instead of a VALUE statement that creates formats, an INVALUE statement is used to create informats. The only difference between the two is that informat names can only be 31 characters in length. (Note: For those curious readers, the reason is that informats and formats are both stored in the same catalog and an "@" is placed before informats to distinguish them from formats.) The following is a program that changes invalid values for GENDER and AE to missing values by using a user-defined informat.

Program 1-9 Using a User-Defined Informat to Set Invalid Data Values to Missing

```
*-----------------------------------------------------------------*
| Purpose: To create a SAS data set called PATIENTS2              |
|          and set any invalid values for Gender and AE to        |
|          missing, using a user-defined informat                 |
*----------------------------------------------------------------*;
libname clean "c:\books\clean";

proc format;
   invalue $gen    'F','M' = _same_
                   other   = ' ';
   invalue $ae    '0','1' = _same_
                   other  = ' ';
run;

data clean.patients_filtered;
   infile "c:\books\clean\patients.txt" pad;
   input @1  Patno    $3.
         @4  Gender   $gen1.
         @27 AE       $ae1.;

   label Patno   = "Patient Number"
         Gender  = "Gender"
         AE      = "adverse event?";
run;

title "Listing of data set PATIENTS_FILTERED";
proc print data=clean.patients_filtered;
   var Patno Gender AE;
run;
```

Notice the INVALUE statements in the PROC FORMAT above. The keyword _SAME_ is a SAS reserved value that does what its name implies—it leaves any of the values listed in the range specification unchanged. The keyword OTHER in the subsequent line refers to any values not matching one of the previous ranges. Notice also that the informats in the INPUT statement use the user-defined informat name followed by the number of columns to be read, the same method that is used with predefined SAS informats.

Output from the PROC PRINT is shown next.

```
Listing of data set PATIENTS_FILTERED

Obs    Patno    Gender    AE

  1     001       M       0
  2     002       F       0
  3     003               1
  4     004       F
  5     XX5       M       0
  6     006               1
  7     007       M       0
  8               M       0
  9     008       F       0
 10     009       M       1
 11     010               0
 12     011       M       1
 13     012       M       0
 14     013
 15     014       M       1
 16     002       F       0
 17     003       M       0
 18     015       F       1
 19     017       F       0
 20     019       M       0
 21     123       M       0
 22     321       F       1
 23     020       F       0
 24     022       M       1
 25     023               0
 26     024       F       0
 27     025       M       1
 28     027       F       0
 29     028       F       0
 30     029       M       1
 31     006       F       0
```

Notice that invalid values for GENDER and AE are now missing values, including the two lowercase 'f's (patient numbers 010 and 023).

Let's add one more feature to this program. By using the keyword UPCASE in the informat specification, you can automatically convert the values being read to uppercase before the ranges are checked. Here are the PROC FORMAT statements, rewritten to use this option.

```
proc format;
   invalue $gen (upcase)  'F' = 'F'
                          'M' = 'M'
                        other = ' ';
   invalue $ae '0','1' = _same_
                other  = ' ';
run;
```

The UPCASE option is placed in parenthesis following the informat name. Notice some other changes as well. You cannot use the keyword _SAME_ anymore because the value is changed to uppercase for comparison purposes, but the _SAME_ specification would leave the original lowercase value unchanged. By specifying each value individually, the lowercase 'f' (the only lowercase GENDER value) would match the range 'F' and be assigned the value of an uppercase 'F'.

The output of this data set is identical to the output for Program 1-9 except the value of GENDER for patients 010 and 023 is an uppercase 'F'.

If you want to preserve the original value of the variable, you can use a user-defined informat with an INPUT function instead of an INPUT statement. You can use this method to check a raw data file or a SAS data set. Program 1-10 reads the SAS data set CLEAN.PATIENTS and uses user-defined informats to detect errors.

Program 1-10   Using a User-Defined Informat with the INPUT Function

```
proc format;
   invalue $gender 'F','M' = _same_
                   other  = 'Error';
   invalue $ae     '0','1' = _same_
                   other   = 'Error';
run;

data _null_;
   file print;
   set clean.patients;
```

```
   if input (Gender,$gender.) = 'Error' then
      put @1 "Error for Gender for patient:" Patno" value is " Gender;
   if input (AE,$ae.) = 'Error' then
      put @1 "Error for AE for patient:" Patno" value is " AE;
run;
```

The advantage of this program over Program 1-9 is that the original values of the variables are not lost.

Output from Program 1-10 is shown below:

```
Listing of invalid character values
Error for Gender for patient:003  value is X
Error for AE for patient:004  value is A
Error for Gender for patient:006  value is
Error for Gender for patient:010  value is f
Error for Gender for patient:013  value is 2
Error for AE for patient:013  value is
Error for Gender for patient:023  value is f
```

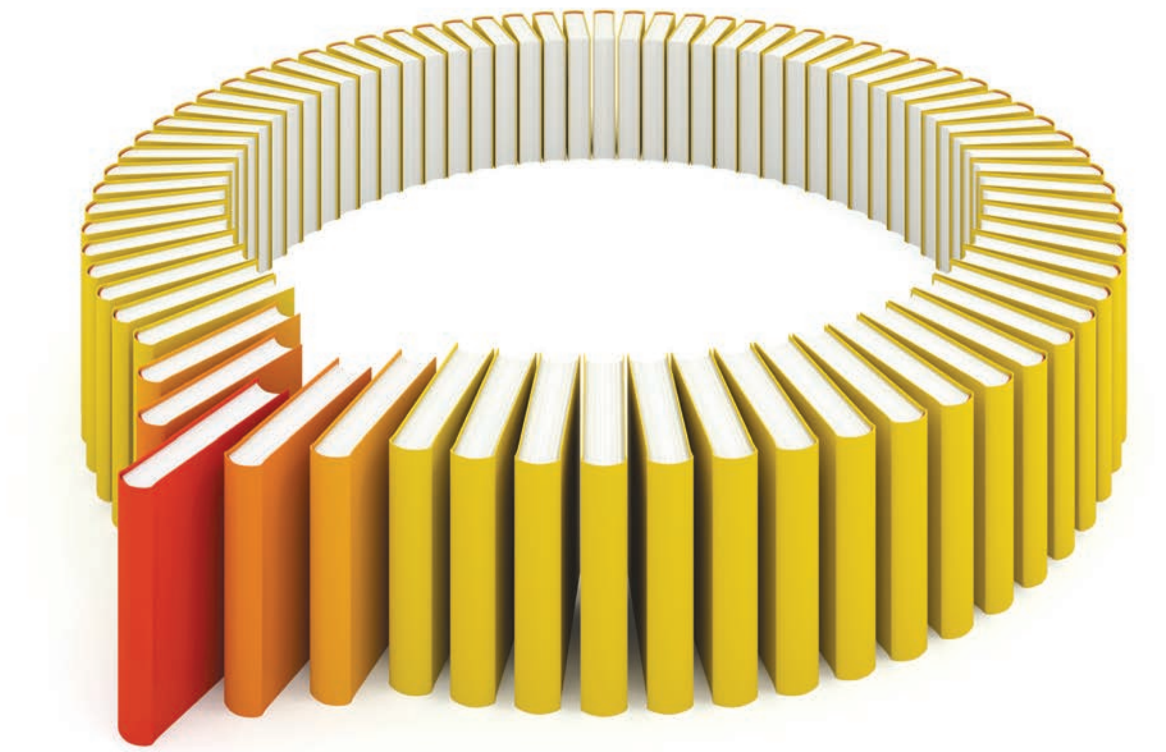
Index

N

V

W

Symbols



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

sas
THE POWER TO KNOW.