PharmaSUG 2019 - Paper AP-114

# The Art of Defensive Programming: Coping with Unseen Data

Philip R Holland, Holland Numerics Limited, UK

## ABSTRACT

This paper discusses how you cope with the following data scenario: the input data set is defined in so far as the variable names and lengths are fixed, but the content of each variable is in some way uncertain. How do you write a SAS program that can cope appropriately with data uncertainty?

"*Code doesn't care who wrote it, but it does matter how well it was written!*" -- Gavin Winpenny (2017)

## READ FIRST, CONVERT LATER

External data could contain a mixture of character and numeric values in any data item. Therefore, best practice would be to read all external data into SAS as character values, and then check these values, and convert where necessary, once they are in SAS variables. Creating pairs of character and numeric variables for each input column will allow every input value to be saved in an appropriate variable type.

This best practice will prevent unexpected character values "hidden" in a column of mostly numeric values being converted automatically by SAS into a missing value, and, therefore, lost. This can easily happen when using PROC IMPORT to read an external file, SAS will guess what data type a column contains based on the first few records. The number of "guessing rows" can be increased using the GUESSINGROWS= parameter, which defaults to 20, but this can also add a new risk that the numeric variables created could change to character variables in a later run. It is much better to force the variable type to character by using the MIXED=YES parameter, when reading from an Excel sheet or a database table, or appending the text file to another text file containing all character columns.

The following sample code demonstrates the risks when reading external files:

Text file **a.txt**:

```
1, ABC, 123, DEFGHI, 456789
2, DEF, 456, JKLMNO, 789012
3, GHI, 789, PQRSTU, 345678
```

This program will import **a.txt** from the current folder using the first 3 rows to determine the value type and length:

```
PROC IMPORT FILE = 'a.txt' OUT = work.a DBMS = CSV REPLACE;
  GUESSINGROWS = 3;
  GETNAMES = NO;
RUN;
```

Data set **work.a**:

```
#  Variable  Type  Len  Format  Informat

1  VAR1      Num     8  BEST12. BEST32.
2  VAR2      Char    3  $3.     $3.
3  VAR3      Num     8  BEST12. BEST32.
4  VAR4      Char    6  $6.     $6.
5  VAR5      Num     8  BEST12. BEST32.
```

```
Obs  VAR1  VAR2  VAR3  VAR4     VAR5

  1     1  ABC    123  DEFGHI 456789
  2     2  DEF    456  JKLMNO 789012
  3     3  GHI    789  PQRSTU 345678
```

Text file **b.txt**:

```
1, ABC, 123, DEFGHI, 456789
2, DEF, 456, JKLMNO, 789012
3, GHI, 789, PQRSTU, 345678
4, JKLM, 2AB, VWXYZ01, 345678A
```

This program code will import **b.txt** from the current folder using the first 3 rows to determine the value type and length:

```
PROC IMPORT FILE = 'b.txt' OUT = work.b DBMS = CSV REPLACE;
  GUESSINGROWS = 3;
  GETNAMES = NO;
RUN;
```

Data set **work.b**:

```
#  Variable  Type  Len  Format  Informat

1  VAR1      Num     8  BEST12. BEST32.
2  VAR2      Char    3  $3.     $3.
3  VAR3      Num     8  BEST12. BEST32.
4  VAR4      Char    6  $6.     $6.
5  VAR5      Num     8  BEST12. BEST32.
```

```
Obs  VAR1  VAR2  VAR3  VAR4     VAR5

  1     1  ABC    123  DEFGHI 456789
  2     2  DEF    456  JKLMNO 789012
  3     3  GHI    789  PQRSTU 345678
  4     4  JKL      .  VWXYZ0       .
```

Note that the 4th records has missing values for **var3** and **var5**. This is due to the “guessing rows” being insufficient to accurately predict the more appropriate data type of Char in that column.

Text file **##header.txt**:

```
ID, 3LETTERS, 3NUMBERS, 6LETTERS, 6NUMBERS
```

This program code will import every file ending in ".txt" in alphanumeric order from the current folder, in this case **##header.txt**, **a.txt** and **b.txt**, using the first 3 rows to determine the value type and length:

```
PROC IMPORT FILE = '*.txt' OUT = work.all DBMS = CSV REPLACE;
  GUESSINGROWS = 3;
  GETNAMES = NO;
RUN;
```

Data set **work.all**:

```
#  Variable  Type  Len  Format  Informat

1  VAR1      Char    2  $2.     $2.
2  VAR2      Char    8  $8.     $8.
3  VAR3      Char    8  $8.     $8.
4  VAR4      Char    8  $8.     $8.
5  VAR5      Char    8  $8.     $8.
```

```
Obs  VAR1  VAR2     VAR3     VAR4     VAR5

  1  ID    3LETTERS 3NUMBERS 6LETTERS 6NUMBERS
  2  1     ABC      123      DEFGHI   456789
  3  2     DEF      456      JKLMNO   789012
  4  3     GHI      789      PQRSTU   345678
  5  1     ABC      123      DEFGHI   456789
  6  2     DEF      456      JKLMNO   789012
  7  3     GHI      789      PQRSTU   345678
  8  4     JKLM     2AB      VWXYZ01  345678A
```

In this case the “header” row will act as the defining values, determining data type **and** variable length for each data column.

The reason this relatively simple technique works in SAS is that SAS data sets can only contain variables for either character or numeric data values, so there is then only a need to test for missing or out of range values. You just have to remember to remove the “header” record before processing the data further!

## TESTING FOR MISSING VALUES

SAS includes several functions to test for missing values. MISSING() only takes a single argument, which can be character or numeric, and returns 0 if the argument is not missing, but 1 if it is missing. NMISS() can be used to count how many of its multiple arguments are missing, but the arguments must all be numeric, and character values will be converted to numeric values first. CMISS() can be used to count how many of its multiple arguments are missing, but the argument can numeric, character, or both.

The following sample code demonstrates how these 3 functions work:

```
DATA _NULL_;
  a = 49;
  b = .;
  c = 'text';
  d = ' ';
  e = '99';
  miss_a = MISSING(a);
  miss_b = MISSING(b);
  miss_c = MISSING(c);
  miss_d = MISSING(d);
  nmiss_atoe = NMISS(a, b, c, d, e);
  cmiss_atoe = CMISS(a, b, c, d, e);
  PUT a= b= c= d= e=;
  PUT miss_a= miss_b= miss_c= miss_d=;
  PUT nmiss_atoe= cmiss_atoe=;
RUN;
```

The output from this Data Step is:

```
a=49 b=. c=text d= e=99
miss_a=0 miss_b=1 miss_c=0 miss_d=1
nmiss_atoe=3 cmiss_atoe=2
```

Note that **nmiss_atoe** is 3, and not 2, because **c** is first converted by NMISS() from 'text' to a missing numeric value, which is shown in the SAS Log as:

```
NOTE: Invalid numeric data, c='text'
```

## IF AND SELECT

Never use IF without ELSE, but better still to use SELECT .. WHEN .. OTHERWISE, which forces a fallback alternative.

In PROC SQL conditional selection is done using a CASE clause, which, like the SELECT statements in the Data Step, includes a mandatory fallback alternative. However, the ELSE statement is only optionally used with IF in the Data Step, which means that, if an expression does not match the condition, it could be missed altogether. Therefore, using a SELECT statement forces a fallback option to be considered and coded, even if it is to do nothing.

```
DATA testing1 (KEEP = name group);
  SET sashelp.class;
  IF name =: "A" THEN group = 1;
  ELSE IF name =: "J" THEN group = 2;
  ELSE IF name =: "P" THEN group = 3;
  ELSE IF name =: "R" THEN group = 4;
RUN;

DATA testing2 (KEEP = name group);
  SET sashelp.class;
  SELECT;
    WHEN (name =: "A") group = 1;
    WHEN (name =: "J") group = 2;
    WHEN (name =: "P") group = 3;
    WHEN (name =: "R") group = 4;
    OTHERWISE PUT 'WAR' 'NING: ' name 'has a missing group';
  END;
RUN;

PROC SQL;
  CREATE TABLE testing3 AS
    SELECT name
          ,(CASE
            WHEN SUBSTR(name, 1, 1) = "A" THEN 1
            WHEN SUBSTR(name, 1, 1) = "J" THEN 2
            WHEN SUBSTR(name, 1, 1) = "P" THEN 3
            WHEN SUBSTR(name, 1, 1) = "R" THEN 4
            ELSE .
            END) AS group
    FROM sashelp.class
  ;
QUIT;
```

Data sets **testing1**, **testing2** and **testing3** will all contain the following records:

```
Obs  Name    group

  1  Alfred      1
  2  Alice       1
  3  Barbara     .
  4  Carol       .
  5  Henry       .
  6  James       2
  7  Jane        2
  8  Janet       2
  9  Jeffrey     2
 10  John        2
 11  Joyce       2
 12  Judy        2
 13  Louise      .
 14  Mary        .
 15  Philip      3
 16  Robert      4
 17  Ronald      4
 18  Thomas      .
 19  William     .
```

However, the **testing2** Data Step will also generate the following messages in the Log:

```
WARNING: Barbara has a missing group
WARNING: Carol has a missing group
WARNING: Henry has a missing group
WARNING: Louise has a missing group
WARNING: Mary has a missing group
WARNING: Thomas has a missing group
WARNING: William has a missing group
```

Note that the split of 'WARNING' in the PUT statement above in **testing2** prevents any syntax formatting software from falsely flagging the program statement in the Log. Equally 'ERROR' should also be split for the same reason.

## MISSING AND INVALID DATA WARNINGS

Always add "missing data" or "invalid data" warning messages to every value test. Change warnings to errors if data is considered critical, so they can't be ignored.

The **testing2** example above demonstrates how to test for values that are not pre-specified, but it is also possible to test values against ranges, as in the following example:

```
DATA range1 (KEEP = name age agegroup);
  SET sashelp.class;
  SELECT;
    WHEN (age = .) PUT 'ER' 'ROR: ' name
                       'has a missing age';
    WHEN (age < 12) PUT 'WAR' 'NING: ' name
                        'is younger than 12';
    WHEN (12 <= age < 13) agegroup = '12';
    WHEN (13 <= age < 14) agegroup = '13';
    WHEN (14 <= age < 15) agegroup = '14';
    WHEN (15 <= age < 16) agegroup = '15';
    OTHERWISE PUT 'WAR' 'NING: ' name 'is older than 15';
  END;
RUN;
```

Data set **range1** will contain the following records:

```
Obs  Name    Age  agegroup

  1  Alfred   14  14
  2  Alice    13  13
  3  Barbara  13  13
  4  Carol    14  14
  5  Henry    14  14
  6  James    12  12
  7  Jane     12  12
  8  Janet    15  15
  9  Jeffrey  13  13
 10  John     12  12
 11  Joyce    11
 12  Judy     14  14
 13  Louise   12  12
 14  Mary     15  15
 15  Philip   16
 16  Robert   12  12
 17  Ronald   15  15
 18  Thomas   11
 19  William  15  15
```

The **range1** Data Step will also generate the following messages in the Log:

```
WARNING: Joyce is younger than 12
WARNING: Philip is older than 15
WARNING: Thomas is younger than 12
```

In this case there are no missing age values, so no ERROR messages appear in the Log.

## FORMATS

A similar test can be achieved using formats by adding an extra value to every format. For example:

```
OTHER = "value not in format"
```

The following SAS code uses a format with an OTHER value:

```
PROC FORMAT;
  VALUE testage
    12 = '12'
    13 = '13'
    14 = '14'
    15 = '15'
    OTHER = "value not in format testage"
    ;
RUN;

DATA range2 (KEEP = name age agegroup);
  SET sashelp.class;
  LENGTH agegroup $50;
  agegroup = PUT(age, testage.);
RUN;
```

Data set **range2** will contain the following records:

```
Obs  Name    Age  agegroup

  1  Alfred   14  14
  2  Alice    13  13
  3  Barbara  13  13
  4  Carol    14  14
  5  Henry    14  14
  6  James    12  12
  7  Jane     12  12
  8  Janet    15  15
  9  Jeffrey  13  13
 10  John     12  12
 11  Joyce    11  value not in format testage
 12  Judy     14  14
 13  Louise   12  12
 14  Mary     15  15
 15  Philip   16  value not in format testage
 16  Robert   12  12
 17  Ronald   15  15
 18  Thomas   11  value not in format testage
 19  William  15  15
```

## NEVER HARD-CODE VALUES

It is never good maintenance practice to hard-code values into your programs, but much better to use lookup data sets or SAS formats to specify values and ranges instead, as these can be updated once to maintain consistency across multiple programs without the need to update each program individually. However, each program that uses the common data must use it in the same way, otherwise an update for an issue in one program could adversely impact the others.

## CHECK FOR UNIQUE KEY VARIABLES USED IN MERGE

If merging data by key variables assuming unique keys, then it is prudent to test whether there are multiple occurrences of these keys to avoid generating extra records:

```
SELECT key1, key2, key3
FROM dsn1
GROUP BY key1, key2, key3
HAVING COUNT(*) > 1
;
```

For example, does **sashelp.class** have multiple occurrences of sex + age + 1st letter of name combinations?

```
PROC SQL;
  SELECT sex
        ,age
        ,SUBSTR(name, 1, 1) AS first_letter
        ,COUNT(*) AS count
  FROM   sashelp.class
  GROUP BY
         sex
        ,age
        ,CALCULATED first_letter
  HAVING COUNT(*) > 1
  ;
QUIT;

%PUT Duplicate count = &sqlobs.;
```

There are 2 occurrences of **sex=M**, **age=12** and **first_letter=J**, so the key variables are not unique:

```
Sex  Age  first_letter  count

M     12  J                2
```

The automatically populated **&sqlobs** macro variable will contain the count of duplicate key combinations, and the log will include the following:

```
Duplicate count = 1
```

Replacing the calculated **first_letter** with **name** will result in no duplicate keys, and the log will include the following lines, because the **&sqlobs** macro variable will still be populated:

```
NOTE: No rows were selected.

Duplicate count = 0
```

## CONCLUSIONS

- When reading external data always read first into character variables, then convert into related numeric variables to avoid losing data.
- Test for any missing values with CMISS(), so you don't need to know whether the variables are character or numeric.
- Use a SELECT statements in Data Steps and CASE clauses in PROC SQL because they both force you to code a fallback alternative when testing the data. However, the SELECT statement allows the use of PUT statements to send messages to the Log too.
- Use lookup data sets or (in)formats instead of hard-coding values, so there is a common source of values for all related programs.
- If data sets are being merged by unique key variables, then test each data set for unique combinations of these keys variable first. The **&sqlobs** macro variable will still be populated, even if there are no duplicates.

## REFERENCES


- Philip R Holland, "*How Do You Use Look-up Tables?*", PhUSE, Brussels, Belgium, 2013.
- Philip R Holland, "*What is Efficient SAS Coding?*", PhUSE, Dublin, Eire, 2006.


## CONTACT INFORMATION

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


Philip R Holland
Holland Numerics Limited
phil@hollandnumerics.org.uk
blog.hollandnumerics.org.uk