PharmaSUG 2023 - Paper SD-313

Be a Lazy Validator – Let Your Code Do the Work!

Cara Lacson, Ray de la Rosa, and Carol Matthews

Advance Research Associates, Santa Clara, CA

ABSTRACT

Regardless of the study phase or indication, programs written to produce output that supports clinical trials submitted to regulatory agencies have one thing in common – they need to be validated. While code and output can be independently validated by a number of methods, the primary responsibility for ensuring that *any* code produces the correct result resides with the programmer who wrote that code. Both production and validation programmers are responsible for ensuring that their own code is valid before comparing the results. While many feel that this is a time-consuming process, there are many ways to quickly add code that effectively checks for data and logic issues so the act of validating your own code can be done efficiently. This paper will discuss a number of simple techniques to make your SAS® and R (R Core Team, 2021) programs self-validating so future code runs with updated data can be checked quickly and effectively.

INTRODUCTION

Validation is a word used throughout the programming space within the pharmaceutical industry. Whether you're discussing the software used to collect clinical data or the programs used to generate tables to report that data, validation is the critical piece of the process that gives everyone a level of comfort that what is being reflected in the data or output accurately reflects the source of the information. When statistical analysis is used to glean meaning from data that is collected, validation is also responsible for ensuring that analytical requirements are met, and that methodology is applied correctly.

Validation includes many levels:

- Data – does the data contain what we expect, and does it behave as we expect? (e.g., are the values in the pregnancy status variable YES, NO or N/A, and are all values N/A when the patient sex is MALE?)
- Syntax – is the code doing what it is intended to do? (e.g., are all character dates correctly converted to a numeric value?)
- Result – does the result of applying logic to the data make sense? (e.g., after calculating percentages for the number of males and females, do those percentages add to 100%?)

Whether creating SDTM datasets or complicated efficacy tables, programmers are responsible for ensuring that the data and the specifications provided to them fit together appropriately. This means that programmers cannot just "follow the specifications" to the letter – they need to ensure that the source data meets both the explicit *and* implicit requirements of those specifications and that the result of applying the specifications makes sense.

While programmers are not responsible for redoing the data cleaning process done by Data Management, programmers *do* need to ensure that any critical assumptions about the data, as related to the application of specifications, are met. This can take many forms, from what values appear in the data to how different data domains will combine to allow for specific analyses. If these key assumptions are not met, then even if subsequent code is applied correctly, the result will not be correct.

While everyone wants validated output, there is always pressure for time and cost. Even though it takes time to validate summary tables, it takes significantly *more* time if errors are found once the tables are included in the study report. In those cases, not only does the programmer need to go back to correct the issue, but the medical writer then needs to include the updated table in the report, revise the text explaining the data presented, then reviewers need to re-review. In addition, now confidence in *all* programmed output has been shaken, often causing everyone to slow down and second guess *everything*. So, how do programmers know when to include checks that ensure critical parts of the

program are correct while not redoing work that was already done? What kinds of checks are worth including?

CONSIDER YOUR DATA

Whether combining data or applying complex analysis logic, it's important to understand the state of the data that you're working with so you know if adding a few extra checks will have impact. For example, if you are reanalyzing data from a completed clinical trial to support a new publication, you would *not* include checks to verify that critical variables only include expected values because the data is clean and final. However, if you're working with a "dirty" database extract from an ongoing trial, you *would* consider adding those checks – especially if you have seen unexpected values in previous versions of the database.

As a general rule, if you're working with interim data and know that you'll need to run your programs on two or more future versions of that data, you'll want to include at least some checks in your code to ensure that critical assumptions made in the specifications are met. While you wouldn't expect to check every data point, you would check any data that is critical to study analysis. Many times, data issues become "obvious" as you work through your program – either causing syntax errors or undesirable messages in your log. Usually, these are the places where you'd want to add code to self-validate that assumptions are met in future runs of the code when new data is received.

CONSIDER THE LOGIC

Many statistical analyses have very specific assumptions about the data to ensure the resulting statistics are valid. These assumptions may be included in the specifications or may cause log errors when not met. If there are complex combinations of data that are needed to perform an analysis (think multiple nested if-then-do statements), you would want to include code that checks if any data fell outside of that set of logic. As with the data, if you encounter issues where logic rules are incomplete for addressing all scenarios that appear in the data or statistical procedures give warnings about assumptions not being met, these are often cases where you'd want to add code to verify that your code is handling all data correctly.

CONSIDER THE FUTURE

Realistically, code is often run by individuals other than the person who authored the code. In many cases, it's the project lead running all production code after new data is received. Other times the program author may be out of the office or busy on another project. By building in critical logic and data checks, program authors can save others the time it takes to review code, logs and resulting output while still ensuring that key assumptions are met. When done judiciously, the code itself will alert anyone running the program when assumptions no longer being met.

Once you realize the value in building critical validation checks into your code, the next step is understanding some of the techniques to add those checks effectively. The remainder of this paper will discuss some of the most effective techniques available in Base SAS and R which allow programmers to build self-validating code.

CHECKS FOR EXPECTED DATA VALUES

While programmers are not expected to duplicate all of the effort made by data managers to review and clean data, it is still important to ensure that data values meet underlying expectations that authors had when writing programming or analysis specifications. This applies to both categorical and continuous data values, although efficient review of continuous data can be more challenging. One effective way to check continuous data is by using available methodologies to group or categorize data in a way that highlights extreme or unexpected values. Applying these methods can condense validation output to only meaningful information that needs review.

Dates are a type of continuous data that need to be handled delicately when they are integral to an analysis. For example, it is more critical to check for unexpected adverse event start and end dates than medical history dates which may only be displayed in data listings. In cases like adverse events, it is

important to check for any unexpected date values to make sure you are handling these dates (and the data that will be analyzed based on those dates) appropriately, and/or for data cleaning if required.

Both SAS and R have ways to group data without changing the underlying values so programmers can easily verify that both the data and their programs are performing as expected.

EXPECTED RANGE CHECKS

In many cases, the continuous data that is collected in clinical trials has expected value parameters. For example, vital signs have definite ranges that are consistent with human life – values outside of those ranges are obviously incorrect. For this type of data, it is possible to create a general library of these ranges that can be used by all programmers to effectively check for unexpected values in the data with little time or effort.

SAS: Use Formats

Base SAS includes the ability to define custom formats which can be applied to data for a variety of uses. While formats are commonly used for decoding categorical variables (e.g., Y=Yes, N=No), they can also be used to group continuous variables in meaningful ways that facilitate efficient verification that data expectations are met. The code below illustrates one example of how to use SAS formats to check for values outside of expected ranges:

```
proc format ;
   value tempc
      35-<40  = 'ok'
      0-<35   = 'TOO LOW'
      40-high = 'TOO HIGH' ;
run ;

proc print data=vs (where=(vstestcd eq 'TEMP' and
                           put(vsstresn,tempc.) ne 'ok')) ;
   title 'TEMPERATURE VALUES THAT ARE TOO HIGH OR TOO LOW' ;
run ;
```

R: Subset Records with Unexpected Data

R lacks the formats that SAS has, but we can still subset and display records that have unexpected values or values that fall outside of a certain range:

```
# SHOW RECORDS THAT HAVE A LOW OR HIGH VALUE OF vsstresn
low <- 35
high <- 40

vs[vs$vstestcd == "TEMP" & vs$vsstresn < low,]  # LOW TEMPERATURE RECORDS
vs[vs$vstestcd == "TEMP" & vs$vsstresn > high,] # HIGH TEMPERATURE RECORDS
```

DATE CHECKS

Numeric dates are typically stored as the number of days from a fixed reference point, which makes them simple continuous numeric variables that are presented in a human-readable format via a given software's method of rendering those numbers meaningful (e.g., via a SAS date format). This can make checking for expected values challenging, but not impossible.

SAS: Use Formats

Formats can also be used to condense dates into appropriate ranges, and if a value is outside the expected date range, it will be more obvious that there is a potential issue. The tricky part with date formats is determining the underlying SAS date value to include in the format. The code below illustrates how to use SAS formats to check for unexpected date values:

```
proc format ;
   value chkdates
                  . = '<<MISSING>>'
        low - -21915 = 'Pre-1900?!'
      -21914-  14609 = '1900s'
      14610 -  18262 = '2000s'
      18263 -  21914 = '2010s'
      21915 -  22280 = '2020'
      22281 -  22645 = '2021'
      22646 -  23010 = '2022'
      23011 - %sysfunc(today()) = '2023'
      %sysfunc(today()) - HIGH  = 'FUTURE?!?' ;

run ;

proc print data=vs (where=(put(input(vsdtc,YYMMDD10.),chkdates.) eq
                                        'FUTURE?!?')) ;
   title 'FUTURE DATES???' ;
run ;
```

R: Use Date Functions

To check whether a date is in the future, we could use the `Sys.Date()` function which returns the current date. Below we get the records in `vs` where `vsdtc` is greater than today's date and, thus, in the future or dates occurring before 2000:

```
# RECORDS WITH A FUTURE DATE FOR vsdtc OR A DATE PRIOR TO 2000
vs[as.Date(vs$vsdtc) > Sys.Date() | as.Date(vs$vsdtc) < as.Date("2000-01-
01"),]
```

EXPECTED VALUE CHECKS

SAS: Use Formats

In addition to continuous variables, SAS formats can also be used to check for unexpected values in categorical data. The code below illustrates how to use a SAS format such that *only* unexpected values are reported in your validation output:

```
proc format ;
   value $chkyn
      'Y', 'N' = 'expected'
      other    = '<<NOT EXPECTED>>' ;
run ;

proc print data=vs (where=(put(vsfl,$chkyn.) eq '<<NOT EXPECTED>>')) ;
   title 'UNEXPECTED VALUES IN VSFL' ;
run ;
```

R: Use the %in% Operator

The `%in%` operator can be used to check whether a certain value is expected based on its presence in a group of pre-specified values. In the example below, variable `expected` contains these pre-specified values. We want to see those records with a value of `vsfl` that we are not expecting:

```
# ANY VALUE OF vsfl THAT IS NOT "Y" OR "N" IS UNEXPECTED
expected <- c("Y", "N")       # EXPECTED VALUES
vs[!(vs$vsfl %in% expected),] # RECORDS WITH UNEXPECTED VALUES FOR vsfl
```

Please see SAS Example 1 in the Appendix 1 for the complete SAS program and corresponding output and R Example 1 in the Appendix 2 for the complete R program and corresponding output.

CHECKS FOR COMBINING DATA

Combining data by either appending or merging datasets can present its own challenges when there are records in one dataset but not in the other, especially when this is not expected. For example, if adverse event data contains subjects that are not in your subject-level analysis dataset, this could be an underlying data issue that needs to be addressed. At minimum, it is best practice to build checks into your code that identify such potential issues, so you can be aware of them and handle them appropriately.

SAS: USE IN= OPTION IN DATA STEPS

When combining datasets using a `merge` statement within a `data` step, you can use the `in=` option to check that data combinations work as expected. The `in=` option helps identify which records originate from which dataset, and because it is a simple way to add an extra layer of data checking, we recommend incorporating it for most data merges within a `data` step.

In the example below, records originating from the work dataset `one` will be assigned a temporary variable `in1` with a value of 1, while records originating from the work dataset `two` will be assigned a temporary variable `in2` with a value of 1. Here, these variables `in1` and `in2` are only available for use within this data step. Note that we recommend naming these variables '`inX`' for better readability and to give better understanding to what these variables mean.

The code below illustrates how to use the `in=` option to write records that are not in both incoming datasets to another dataset `check` for checking purposes:

```
data new check ;
   merge one (in=in1)
         two (in=in2) ;
   by usubjid ;

   if in1 and in2 then output new ;
   else                output check ;
run ;
```

From there we can print the resulting work dataset `check` to see and confirm all of the records in question:

```
proc print data=check ;
   title 'RECORDS NOT IN BOTH DATASETS' ;
run ;
```

Please see SAS Example 2 in Appendix 1 for the complete example and corresponding output.

R: ADD VARIABLES

Before merging `one` and `two` together, we must first add a variable called `in1` to `one` and `in2` to `two` in order to implement the same technique described above in the SAS example. Both of these variables will be set to 1 for all records in their corresponding dataset:

```
one$in1 <- 1
two$in2 <- 1
```

We then merge `one` and `two` together to get `one_or_two` which contains all records appearing in either `one` or `two`. After merging, `in1` will be 1 if the record is coming from `one` or `NA` if the record is not. Likewise, the same can be said for `in2` with respect to `two`:

```
one_or_two <- merge(one, two, by=c("usubjid"), all=TRUE)
```

From there we can create the dataset `new` which contains only the records that are present in both `one` and `two` by subsetting `one_or_two`. We use the `is.na()` function to check whether a variable is `NA` or not:

```
new <- one_or_two[!is.na(one_or_two$in1) & !is.na(one_or_two$in2),]
```

But even more importantly, we can also get the records that are not present in both `one` and `two`. We can save these records in `check` and later review them:

```
check <- one_or_two[is.na(one_or_two$in1) | is.na(one_or_two$in2),]
```

`new` and `check` are both subsets of `one_or_two` and are mutually exclusive with each other.

Please see R Example 2 in Appendix 2 for the complete example and corresponding output.

CHECKS FOR UNEXPECTED DUPLICATES/MULTIPLE RECORDS PER ID VARIABLES

It is often critical, especially when dealing with messy data, to determine if there are any erroneous duplicate records in your data. Sometimes two or more records may not be exact duplicates (having the same value on every variable), but they may have multiple records within certain ID variables such as subject ID and visit, and it is important to know that such records exist in order to handle them appropriately.

SAS: USE SORT OPTIONS

One way to check for unexpected duplicates/multiple records per select ID variables is by using a `proc sort` statement with both the `nodupkey` and `dupout=` options. The `nodupkey` option checks for multiple records with the same values in the variables listed in the `by` statement, and pairing this option with the `dupout`= option writes these multiple records to the defined work dataset. This is illustrated in the code below, where the multiple records identified using the `nodupkey` option are output to the work dataset `checkme`, and the remaining records are output to the work dataset `vitals1`:

```
proc sort data=vitals out=vitals1 nodupkey dupout=checkme ;
   by usubjid visit_id visit_name ;
run ;

proc print data=checkme ;
   title "UNEXPECTED DATA: MULTIPLE RECORDS WITHIN SUBJECT AND VISIT" ;
run ;
```

Another similar approach is to use a `proc sort` statement with the `noduprec` option. Unlike `nodupkey`, the `noduprec` option checks for multiple records with the same values in all of the variables within the specified dataset. The `dupout`= option can also be used here as well:

```
proc sort data=vitals out=vitals2 noduprec dupout=checkme2 ;
   by usubjid visit_id visit_name ;
run ;

proc print data=checkme2 ;
   title "UNEXPECTED DATA: FULL DUPLICATE RECORDS" ;
run ;
```

Please see SAS Example 3 in Appendix 1 for the complete example and corresponding output.

R: USE THE DUPLICATED() FUNCTION

In R we can make use of the `duplicated()` function to get the duplicate records in the `vitals` dataset and store them in `checkme` for review. In the example below, duplicates are based solely on the values of `usubjid`, `visit_id`, and `visit_name` regardless of the values of any other variables in the `vitals` dataset:

```
checkme <- vitals[duplicated(vitals[, c("usubjid", "visit_id",
"visit_name")]),]
```

Please see R Example 3 in Appendix 2 for the complete example and corresponding output.

LOGIC CHECKS

In some cases, the use of logic checks can help to identify problems in your data. Generally, if a certain condition is true, then an action is applied, such as assigning a value to new variable. Otherwise, if another condition is true, then a different action is applied, and so on. Any unexpected condition could, for example, trigger a warning message or output to a separate check dataset to help you identify the problem.

SAS: USE IF/THEN OR SELECT CONSTRUCTS IN DATA STEPS

In SAS, a series of `if/then` or `select` statements can help identify unexpected values. In addition, if the series of logic checks detects an expected value, you can use the `put` statement to print a warning message to the SAS log in addition to the values of key variables.

The code below demonstrates `if/then` logic:

```
data new ;
   set old ;
        if var1 eq 1 and var2 eq 1 then newvar = 1 ;
   else if var1 eq 2 and var2 eq 1 then newvar = 2 ;
   else put 'WAR' 'NING: UNEXPECTED COMBINATION OF VAR1 AND VAR2: ' subjid=
var1= var2= ;
run ;
```

Similarly, the code below demonstrates `select` logic:

```
data new ;
   set old ;
   select (condition) ;
      when ('Y')   newvar = 1 ;
      when ('N')   newvar = 2 ;
      otherwise put 'WAR' 'NING: UNEXPECTED VALUE FOR CONDITION: ' subjid=
condition= ;
   end ;
run ;
```

Note that in both of the above examples, instead of writing the word 'WARNING' in the SAS code, we split the word into 'WAR' and 'NING'. This prevents the word 'WARNING' from being written to your SAS log except in the case of a true warning, where the resulting SAS log will look like this:

```
WARNING: UNEXPECTED COMBINATION OF VAR1 AND VAR2: subjid=1002 var1=1 var2=2
```

Putting the word 'WARNING' in your SAS log allows you to quickly search through your log for all of the instances of 'WARNING', which enables you to confirm the results and take an alternative action if necessary.

Please see SAS Example 4 in Appendix 1 for the complete example and corresponding SAS log.

R: USE IF/ELSE IF LOGIC

As in the case of the SAS code above, the R code below goes through all the values of `condition` and assigns `newvar` some value. The algorithm outputs the corresponding value of `usubjid` and `condition` during run time whenever we encounter an unexpected value for `newvar`:

```
newvar <- as.integer()
counter <- 1
for (ii in OLD$condition)
{
   if (ii == 1){
      newvar = c(newvar, 1)
   } else if (ii == 2) {
      newvar = c(newvar, 2)
   } else {
      newvar = c(newvar, NA)
      print(paste("WARNING: UNEXPECTED VALUE FOR CONDITION: USUBJID =",
OLD$usubjid[counter], ", CONDITION:", ii))
   }
   counter <- counter + 1
}
NEW <- cbind(OLD, newvar)
```

Please see R Example 4 in Appendix 2 for the complete example and corresponding output.

CONCLUSION

Incorporating self-validating techniques into your code will almost always save time in the long run, as you can identify potential issues and handle them appropriately from the outset. It also gives you greater confidence that your programs are working as you expect and helps prevent extra back-and-forth time between you and the validator. In addition, robust programming code is designed to predict unexpected values that may not be present in the current data but may be present in future data transfers/updates; this helps other programmers who may not have written the code initially to be able to rerun code and check for issues without having an intimate knowledge of the data.

Ideally, the methods you choose to self-validate your code should depend on how you can most efficiently and accurately check through the "checking" output that your program produces. For example, if it is more effective for you to search through your program log for "WARNING" messages, then you may want to consider choosing this method to write warning messages to your log rather than "print" problematic records to an output file.

Ultimately, validation is critical in clinical trials as we are working with data that represents *people,* and what we are producing helps protect the safety of patients and contributes to the development of potentially life-changing/life-saving treatments. Self-validating code adds an extra layer of confidence that is handled data appropriately, logic is applied appropriately, and unexpected data issues are not overlooked.

REFERENCES

Matthews, Carol I; Shilling, Brian C. 2008. *Validating Clinical Trial Data Reporting with SAS®*. Cary, NC: SAS Institute, Inc.

RECOMMENDED READING

- *Base SAS® Procedures Guide*
- *SAS® For Dummies®*

CONTACT INFORMATION

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

Cara Lacson
Advance Research Associates, Inc.
clacson@advanceresearch.com

Ray de la Rosa
Advance Research Associates, Inc.
rdlrosa@advanceresearch.com

Carol Matthews
Advance Research Associates, Inc.
cmatthews@advanceresearch.com

APPENDIX 1: SAS EXAMPLES

SAS EXAMPLE 1

```
** ASSIGN FORMATS **;

proc format ;
   value chkdates
                    . = '<<MISSING>>'
        low - -21915 = 'Pre-1900?!'
      -21914-  14609 = '1900s'
      14610 -  18262 = '2000s'
      18263 -  21914 = '2010s'
      21915 -  22280 = '2020'
      22281 -  22645 = '2021'
      22646 -  23010 = '2022'
      23011 - %sysfunc(today()) = '2023'
      %sysfunc(today()) - HIGH  = 'FUTURE?!?' ;

   value tempc
      35-<40  = 'ok'
      0-<35   = 'TOO LOW'
      40-high = 'TOO HIGH' ;

   value $chkyn
      'Y', 'N' = 'expected'
      other    = '<<NOT EXPECTED>>' ;
run ;

** INPUT VITAL SIGNS DATA **;

data vs ;
   length birthdtc vsdtc $10 vstestcd $4 vsfl $1 ;
   input subjid birthdtc $ vstestcd $ vsdtc $ vsstresn vsfl $ ;

cards ;
1001 1973-11-08 TEMP 2023-03-27 35.2 Y
1002 1985-10-24 TEMP 2033-03-24 34.5 Y
2001 1945-05-28 TEMP 2023-03-27 39.0 U
2002 1968-01-27 TEMP 2023-03-16 40 N
;
run ;

** PRINT TEMPERATURE VALUES THAT ARE OUT OF NORMAL RANGE **;

proc print data=vs (where=(vstestcd eq 'TEMP' and
                           put(vsstresn,tempc.) ne 'ok')) ;
   title 'TEMPERATURE VALUES THAT ARE TOO HIGH OR TOO LOW' ;
run ;

** PRINT FUTURE DATES **;

proc print data=vs (where=(put(input(vsdtc,YYMMDD10.),chkdates.) eq
                                    'FUTURE?!?')) ;
   title 'FUTURE DATES???' ;
run ;
```

```
** PRINT UNEXPECTED VALUES IN VSFL **;

proc print data=vs (where=(put(vsfl,$chkyn.) eq '<<NOT EXPECTED>>')) ;
   title 'UNEXPECTED VALUES IN VSFL' ;
run ;
```

SAS Example 1 Output from PROC PRINT

```
TEMPERATURE VALUES THAT ARE TOO HIGH OR TOO LOW

Obs     birthdtc       vsdtc       vstestcd    vsfl    subjid    vsstresn

2     1985-10-24    2033-03-24      TEMP       Y       1002       34.5
4     1968-01-27    2023-03-16      TEMP       N       2002       40.0

FUTURE DATES???

Obs     birthdtc       vsdtc       vstestcd    vsfl    subjid    vsstresn

2     1985-10-24    2033-03-24      TEMP       Y       1002       34.5

UNEXPECTED VALUES IN VSFL

Obs     birthdtc       vsdtc       vstestcd    vsfl    subjid    vsstresn

3     1945-05-28    2023-03-27      TEMP       U       2001        39
```

SAS EXAMPLE 2

```
** INPUT SUBJECT-LEVEL DATA **;

data one ;
   input subjid ;

cards ;
1001
1002
2001
run ;

** INPUT VITAL SIGNS DATA **;

data two ;
   length vsdtc $10 vstestcd $4 vsfl $1 ;
   input subjid vstestcd $ vsdtc $ vsstresn vsfl $ ;

cards ;
1001 TEMP 2023-03-27 35.2 Y
1002 TEMP 2023-03-24 34.5 Y
2001 TEMP 2023-03-27 39.0 U
2002 TEMP 2023-03-16 40 N
run ;

** MERGE DATA **;

data new check ;
   merge one (in=in1)
```

```
        two (in=in2) ;
   by subjid ;

   if in1 and in2 then output new ;
   else                output check ;
run ;

proc print data=check ;
   title 'RECORDS NOT IN BOTH DATASETS' ;
run ;
```

SAS Example 2 Output from PROC PRINT

```
RECORDS NOT IN BOTH DATASETS

Obs    subjid      vsdtc       vstestcd    vsfl    vsstresn

1      2002     2023-03-16      TEMP       N         40
```

SAS EXAMPLE 3

```
** INPUT VITAL SIGNS DATA **;

data vitals ;
   length visit_id $4 visit_name $8 vsdtc $10 vstestcd $4 vsfl $1 ;
   input subjid visit_id $visit_name $ vstestcd $ vsdtc $ vsstresn vsfl $ ;

cards ;
1001 VIS1 Baseline TEMP 2023-03-27 35.2 Y
1001 VIS1 Week1    TEMP 2023-04-04 35.3 Y
1002 VIS1 Baseline TEMP 2023-03-24 34.5 Y
1002 VIS1 Baseline TEMP 2023-03-25 34.6 Y
1002 VIS1 Week1    TEMP 2023-04-02 34.7 Y
2001 VIS1 Baseline TEMP 2023-03-27 39.0 U
2001 VIS1 Week1    TEMP 2023-04-04 39.1 U
2002 VIS1 Baseline TEMP 2023-03-16 40 N
2002 VIS1 Baseline TEMP 2023-03-16 40 N
run ;

** CHECK FOR MULTIPLE RECORDS WITH SAME SUBJECT, VISIT ID, VISIT NAME  **;

proc sort data=vitals out=vitals1 nodupkey dupout=checkme ;
   by subjid visit_id visit_name ;
run ;

proc print data=checkme ;
   title "UNEXPECTED DATA: MULTIPLE RECORDS WITHIN SUBJECT AND VISIT" ;
run ;

** CHECK FOR FULL DUPLICATE RECORDS **;

proc sort data=vitals out=vitals2 noduprec dupout=checkme2 ;
   by subjid visit_id visit_name ;
run ;

proc print data=checkme2 ;
   title "UNEXPECTED DATA: FULL DUPLICATE RECORDS" ;
```

```
run ;
```

SAS Example 3 Output from PROC PRINT

```
UNEXPECTED DATA: MULTIPLE RECORDS WITHIN SUBJECT AND VISIT

                  visit_
Obs   visit_id     name       vsdtc      vstestcd   vsfl   subjid  vsstresn

 1      VIS1     Baseline   2023-03-25     TEMP      Y      1002      34.6
 2      VIS1     Baseline   2023-03-16     TEMP      N      2002      40.0

UNEXPECTED DATA: FULL DUPLICATE RECORDS

                  visit_
Obs   visit_id     name       vsdtc      vstestcd   vsfl   subjid  vsstresn

 1      VIS1     Baseline   2023-03-16     TEMP      N      2002       40
```

SAS EXAMPLE 4

```
** INPUT DATA **;

data old ;
   input subjid var1 var2 condition $ ;

cards ;
1001 1 1 Y
1002 1 2 Y
2001 2 1 N
2002 2 1 U
run ;

** ADD DATA LOGIC **;

data new ;
   set old ;
        if var1 eq 1 and var2 eq 1 then newvar = 1 ;
   else if var1 eq 2 and var2 eq 1 then newvar = 2 ;
   else put 'WAR' 'NING: UNEXPECTED COMBINATION OF VAR1 AND VAR2: ' subjid=
var1= var2= ;
run ;

data new ;
   set old ;
   select (condition) ;
      when ('Y')   newvar = 1 ;
      when ('N')   newvar = 2 ;
      otherwise put 'WAR' 'NING: UNEXPECTED VALUE FOR CONDITION: ' subjid=
condition= ;
   end ;
run ;
```

SAS Example 4 Log Results

```
1          ** INPUT DATA **;
2
3          data old ;
4             input subjid var1 var2 condition $ ;
5
6          cards ;

NOTE: The data set WORK.OLD has 4 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


11         run ;
12
13         ** ADD DATA LOGIC **;
14
15         data new ;
16            set old ;
17                 if var1 eq 1 and var2 eq 1 then newvar = 1 ;
18            else if var1 eq 2 and var2 eq 1 then newvar = 2 ;
19            else put 'WAR' 'NING: UNEXPECTED COMBINATION OF VAR1 AND
VAR2: ' subjid= var1= var2=
19       ! ;
20         run ;

WARNING: UNEXPECTED COMBINATION OF VAR1 AND VAR2: subjid=1002 var1=1 var2=2
NOTE: There were 4 observations read from the data set WORK.OLD.
NOTE: The data set WORK.NEW has 4 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


21
22         data new ;
23            set old ;
24            select (condition) ;
25               when ('Y')   newvar = 1 ;
26               when ('N')   newvar = 2 ;
27               otherwise put 'WAR' 'NING: UNEXPECTED VALUE FOR CONDITION:
' subjid= condition= ;
28            end ;
29         run ;

WARNING: UNEXPECTED VALUE FOR CONDITION: subjid=2002 condition=U
NOTE: There were 4 observations read from the data set WORK.OLD.
NOTE: The data set WORK.NEW has 4 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

APPENDIX 2: R EXAMPLES

R EXAMPLE 1

```
subjid <- c("001", "002", "003", "004")
vstestcd <- "TEMP"
vsdtc <- c("2010-10-10", "2099-09-09", "2024-07-31", "2012-07-18")
vsstresn <- c(10, 37, 35, 102)
vsfl <- c("Y", "Y", "", "N")
vs <- data.frame(subjid, vstestcd, vsdtc, vsstresn, vsfl)
vs # SHOW RECORDS IN vs
  subjid vstestcd      vsdtc vsstresn vsfl
1    001     TEMP 1990-10-10       10    Y
2    002     TEMP 2099-09-09       37    Y
3    003     TEMP 2024-07-31       35
4    004     TEMP 2012-07-18      102    N

# SHOW RECORDS THAT HAVE A LOW OR HIGH VALUE OF vsstresn
low <- 35
high <- 40

vs[vs$vstestcd == "TEMP" & vs$vsstresn < low,]  # LOW TEMPERATURE RECORDS
  subjid vstestcd      vsdtc vsstresn vsfl
1    001     TEMP 1990-10-10       10    Y

vs[vs$vstestcd == "TEMP" & vs$vsstresn > high,] # HIGH TEMPERATURE RECORDS
  subjid vstestcd      vsdtc vsstresn vsfl
4    004     TEMP 2012-07-18      102    N

# RECORDS WITH A FUTURE DATE FOR vsdtc OR A DATE PRIOR TO 2000
vs[as.Date(vs$vsdtc) > Sys.Date() | as.Date(vs$vsdtc) < as.Date("2000-01-
01"),]
  subjid vstestcd      vsdtc vsstresn vsfl
1    001     TEMP 1990-10-10       10    Y
2    002     TEMP 2099-09-09       37    Y
3    003     TEMP 2024-07-31       35

# ANY VALUE OF vsfl THAT IS NOT "Y" OR "N" IS UNEXPECTED
expected <- c("Y", "N")       # EXPECTED VALUES
vs[!(vs$vsfl %in% expected),] # RECORDS WITH UNEXPECTED VALUES FOR vsfl
  subjid vstestcd      vsdtc vsstresn vsfl
3    003     TEMP 2024-07-31       35
```

R EXAMPLE 2

```
usubjid1 <- c("001-001", "001-002", "001-003")
procdt <- c("2021-10-12", "2019-07-24", "2022-01-01")
one <- data.frame(usubjid=usubjid1, procdt)

usubjid2 <- c("001-002", "001-003", "001-004")
discdt <- c("2019-12-31", "2023-01-15", NA)
two <- data.frame(usubjid=usubjid2, discdt)

one$in1 <- 1
two$in2 <- 1

one
```

```
  usubjid     procdt in1
1 001-001 2021-10-12   1
2 001-002 2019-07-24   1
3 001-003 2022-01-01   1

two
  usubjid     discdt in2
1 001-002 2019-12-31   1
2 001-003 2023-01-15   1
3 001-004       <NA>   1

one_or_two <- merge(one, two, by=c("usubjid"), all=TRUE)
one_or_two
  usubjid     procdt in1     discdt in2
1 001-001 2021-10-12   1       <NA>  NA
2 001-002 2019-07-24   1 2019-12-31   1
3 001-003 2022-01-01   1 2023-01-15   1
4 001-004       <NA>  NA       <NA>   1

new <- one_or_two[!is.na(one_or_two$in1) & !is.na(one_or_two$in2),]
new
  usubjid     procdt in1     discdt in2
2 001-002 2019-07-24   1 2019-12-31   1
3 001-003 2022-01-01   1 2023-01-15   1

check <- one_or_two[is.na(one_or_two$in1) | is.na(one_or_two$in2),]
check
  usubjid     procdt in1 discdt in2
1 001-001 2021-10-12   1   <NA>  NA
4 001-004       <NA>  NA   <NA>   1
```

R EXAMPLE 3

```
subjects <- c("001-001","001-001","001-001","001-003")
visit_id <- c(1, 1, 2, 1)
visit_name <- c("D1", "D1", "D7", "D1")
wt <- c(147, 151, 150, 123)
vitals <- data.frame(usubjid=subjects, visit_id, visit_name, wt)
vitals
  usubjid visit_id visit_name  wt
1 001-001        1         D1 147
2 001-001        1         D1 151
3 001-001        2         D7 150
4 001-003        1         D1 123

checkme <- vitals[duplicated(vitals[, c("usubjid", "visit_id",
"visit_name")]),]
checkme
  usubjid visit_id visit_name  wt
2 001-001        1         D1 151
```

R EXAMPLE 4

```
OLD <- data.frame(usubjid=c("001-006", "001-007", "001-008", "001-009"),
condition=c(1, 0, 2, 9))
OLD
```

```
  usubjid condition
1 001-006         1
2 001-007         0
3 001-008         2
4 001-009         9
```

```
newvar <- as.integer()
counter <- 1
for (ii in OLD$condition)
{
   if (ii == 1){
      newvar = c(newvar, 1)
   } else if (ii == 2) {
      newvar = c(newvar, 2)
   } else {
      newvar = c(newvar, NA)
      print(paste("WARNING: UNEXPECTED VALUE FOR CONDITION: USUBJID =",
OLD$usubjid[counter], ", CONDITION:", ii))
   }
   counter <- counter + 1
}
```

```
[1] "WARNING: UNEXPECTED VALUE FOR CONDITION: USUBJID = 001-007 ,
CONDITION: 0"
[1] "WARNING: UNEXPECTED VALUE FOR CONDITION: USUBJID = 001-009 ,
CONDITION: 9"
```

```
NEW <- cbind(OLD, newvar)
NEW
```

```
  usubjid condition newvar
1 001-006         1      1
2 001-007         0     NA
3 001-008         2      2
4 001-009         9     NA
```