Paper 124-2019

‘Tis Better to Give than Receive: Considerations when Sharing Data

Melissa R. Pfeiffer, Children’s Hospital of Philadelphia, Philadelphia, Pennsylvania

ABSTRACT

Data exchanges are increasingly common and are critical in propelling advancements in science and industry. However, these exchanges are not always seamless. Issues can arise both when receiving data from others and when distributing data, although the latter is usually easier. The aim of this paper is to provide guidance for both situations. For receiving data, we will discuss the importance of trying to obtain basic information about the data, such as a data dictionary or file layout with information about the field names, types, lengths, and meanings. At a bare minimum, recipients should ask about the number of records expected to confirm what is imported into SAS. We will discuss possible issues that may occur when importing non-SAS data into SAS and solutions. Finally, we will review some checks that can be conducted to confirm that files received are structurally sound. For distributing data, we will discuss strategies for creating non-SAS data, depending on what file type the recipient needs (e.g., Excel or CSV), whether column headings should be variable names or labels, and whether values should be formatted or unformatted. To help promote our reputations for giving generously, we will talk about documentation that should accompany the data you provide, such as a data dictionary—either simple or more detailed—and variable format information, when applicable. Collaborations and data sharing should be reasons for celebration rather than trepidation; the aim of this paper is to make these processes a little easier.

INTRODUCTION

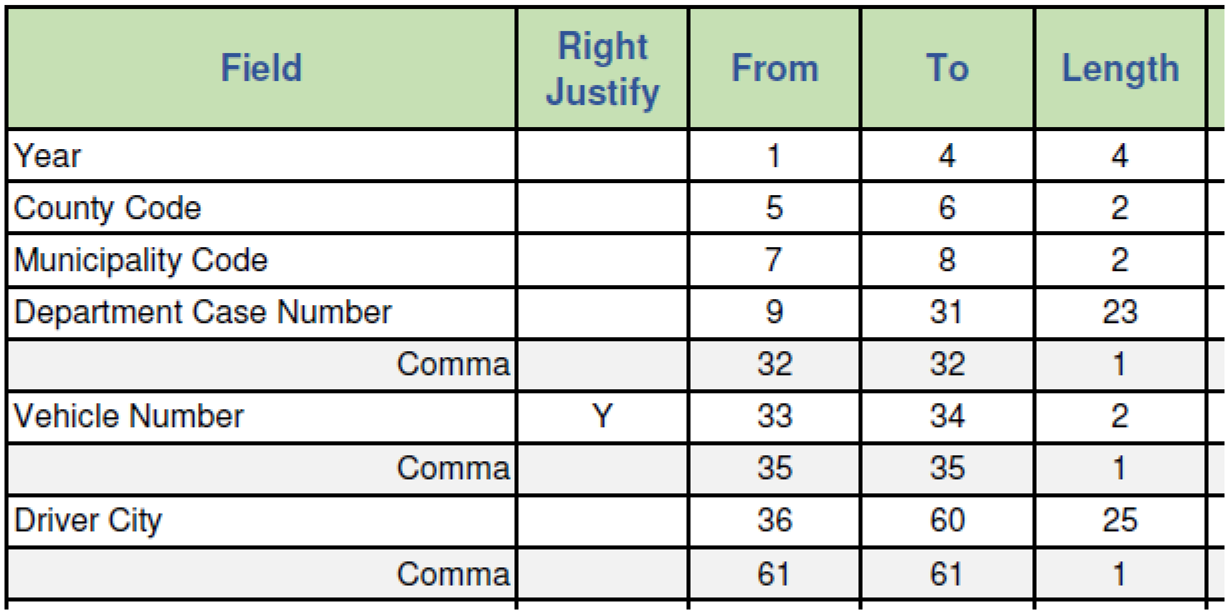
We can all benefit by sharing data—both giving data and receiving data from others. There are some strategies that can be employed to help each of us both be and create informed data users. The intention of this paper is to discuss topics to help achieve that goal, that is, to address common hurdles experienced when receiving data as well as to provide considerations to keep in mind when dispensing data. Specifically, it covers asking for and giving metadata, import and export issues, and enhancing understanding of data sets by examining your new data and providing insights into the data you release. Although the background for this paper was formed while working, generally, with administrative and research data, it can be applied to all fields and is suitable for anyone who gives or receives data.

RECEIVING DATA IN OTHER FORMATS: TRUST BUT VERIFY

Yes, that old Russian proverb of “trust but verify” still serves as very useful advice. When other people give us data, they no doubt give us as clean and accurate data as they can. But not only do we all occasionally make mistakes, but sometimes people are so familiar with their own data that they forget what it’s like to start fresh with it. For those reasons, it’s important to try to obtain as much supporting information with your new data as possible.

ASK FOR DATA DICTIONARIES

Even when the data seem relatively straightforward, it is always helpful to have a data dictionary or file layout information. A simple file layout document might just contain information about where each field begins and ends in a file with fixed fields (Display 1).



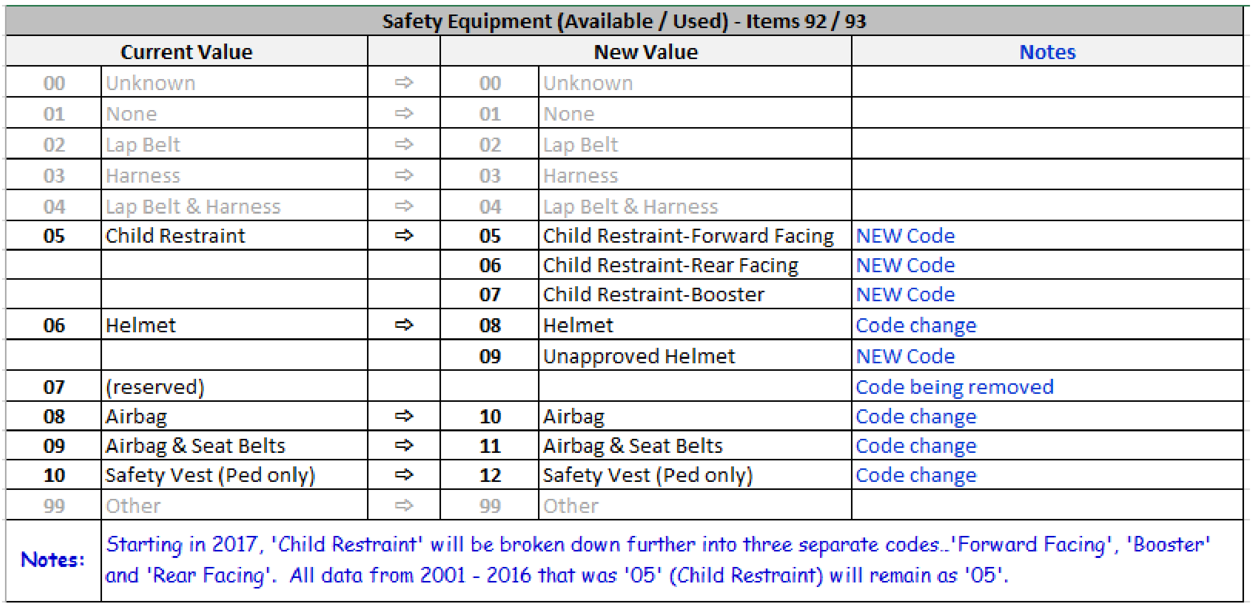
Field	Right Justify	From	To	Length
Year		1	4	4
County Code		5	6	2
Municipality Code		7	8	2
Department Case Number		9	31	23
Comma		32	32	1
Vehicle Number	Y	33	34	2
Comma		35	35	1
Driver City		36	60	25
Comma		61	61	1

Display 1. Example of a File Layout Document

A more detailed data dictionary may contain field names, types, and lengths and might include labels or variable meanings and possibly even valid values or ranges. Review this information thoroughly and be on the look-out for incorrect or outdated information. Make sure that the data you import into SAS matches the information supplied in the documentation—and follow-up with the data owners if it does not match. I recently received a file in which I was expecting the last field to be a character variable with a length of 1—and indeed it seemed to be. But the file layout provided indicated that the field was a date variable. In communications with the data owner, we confirmed that the field should be text and that the file layout had a copy and paste error. Similarly, the variable structure could change over time—what was once captured as a numeric variable may now be a character variable (or vice versa).

The data dictionary may also provide insights as to what coded values mean. For instance, if you have a coded variable for sex with numeric values of 0 and 1, that variable provides little information if you do not know what those values mean. Although you may be able to discern the meaning from the values of other variables (such as a variable for “pregnant”), it is always best to avoid guessing, no matter how educated those guesses might be.

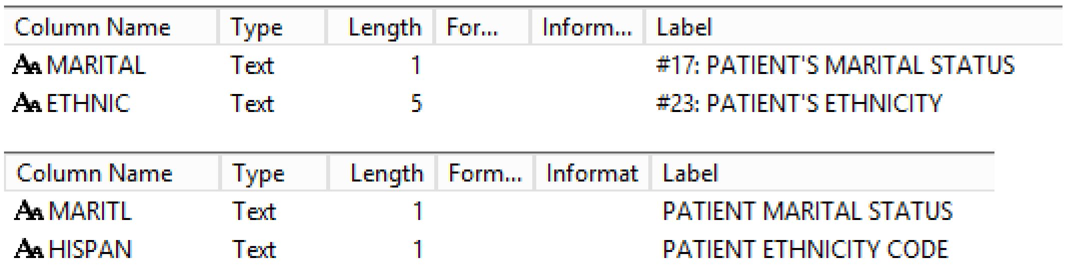
Be aware that sometimes the meaning of codes for a variable can change over time when you are dealing with longitudinal data (particularly administrative data). As an example, the New Jersey Department of Transportation (NJ DOT) recently changed their crash report with regard to safety equipment used (among other things) to allow for details about child restraint systems. Previously 05 meant “Child Restraint” and 06 meant “Helmet,” but starting in 2017 05 means “Child Restraint—Forward Facing,” 06 means “Child Restraint—Rear Facing,” and “Helmet” has been changed to 08. The publicly available copies of the New Jersey Police Crash Investigation Report form for the years 2001 – 2016 and for 2017 were instrumental in determining these different meanings and re-coding our own variable accordingly. The NJ DOT also thoughtfully provided a document describing the differences in coding with the new crash report (Display 2).



Safety Equipment (Available / Used) - Items 92 / 93					
Current Value			New Value		Notes
00	Unknown	⇨	00	Unknown	
01	None	⇨	01	None	
02	Lap Belt	⇨	02	Lap Belt	
03	Harness	⇨	03	Harness	
04	Lap Belt & Harness	⇨	04	Lap Belt & Harness	
05	Child Restraint	⇨	**05**	Child Restraint-Forward Facing	NEW Code
			06	Child Restraint-Rear Facing	NEW Code
			07	Child Restraint-Booster	NEW Code
06	Helmet	⇨	**08**	Helmet	Code change
			09	Unapproved Helmet	NEW Code
07	(reserved)				Code being removed
08	Airbag	⇨	**10**	Airbag	Code change
09	Airbag & Seat Belts	⇨	**11**	Airbag & Seat Belts	Code change
10	Safety Vest (Ped only)	⇨	**12**	Safety Vest (Ped only)	Code change
99	Other	⇨	99	Other	
Notes:	Starting in 2017, 'Child Restraint' will be broken down further into three separate codes..'Forward Facing', 'Booster' and 'Rear Facing'. All data from 2001 - 2016 that was '05' (Child Restraint) will remain as '05'.				

Display 2. Example of NJDOT Documentation of Code Change in NJ Crash Report

The structure of files can also shift; variables with the same meaning may have different names or the availability of variables may change. For example, I have received health-related data sets in which the variable for marital status was called “MARITAL” in one data set and “MARITL” in the other; similarly the variable for patient’s ethnicity was called “ETHNIC” in one data set and “HISPAN” in the other (Display 3).



Column Name	Type	Length	For...	Inform...	Label
MARITAL	Text	1			#17: PATIENT'S MARITAL STATUS
ETHNIC	Text	5			#23: PATIENT'S ETHNICITY

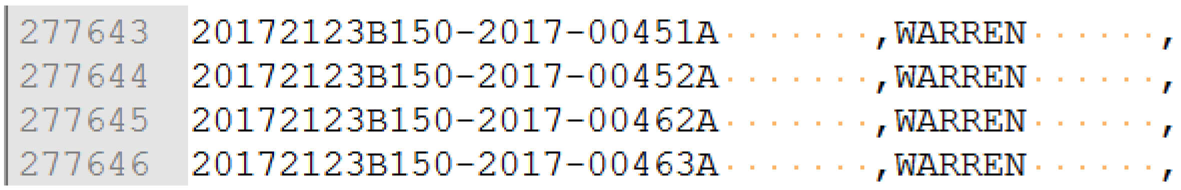
Column Name	Type	Length	Form...	Informat	Label
MARITL	Text	1			PATIENT MARITAL STATUS
HISPAN	Text	1			PATIENT ETHNICITY CODE

Display 3. Example of Variables With Same Meaning but Different Names

Be careful when you see slight modifications in variable names, however. In both of these instances the coding of the values was not the same—for marital status, values of “S” and “U” had different meanings for “MARITAL” and “MARITL” (single versus separated, unknown versus unmarried) and for ethnicity, one variable had 5-digit codes and the other single-digit codes. Other variables simply cease to be included in the later time period, which could be due to those variables no longer being collected (such as patient’s employer) or those variables no longer being released in data requests (such as social security number).

ASK FOR THE NUMBER OF RECORDS IN EACH FILE

It is advisable to always ask for the number of records that should be in the files you are receiving. This information can be particularly valuable if metadata are not provided. You can check the number of records in a file by opening that file with non-SAS software, such as Notepad++ (Display 4).



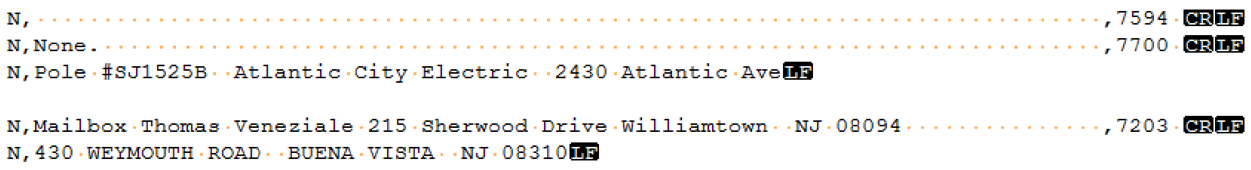
```
277643  20172123B150-2017-00451A       ,WARREN      ,
277644  20172123B150-2017-00452A       ,WARREN      ,
277645  20172123B150-2017-00462A       ,WARREN      ,
277646  20172123B150-2017-00463A       ,WARREN      ,
```

Display 4. Example of CSV File Opened in Notepad++ Showing End of File

Knowing the number of records you should import can be a helpful piece in the puzzle of confirming that all records were imported properly; or, on the contrary, it can be an indication that something is causing some of the records to be omitted. This can happen because of unexpected and problematic characters imbedded in the file.

COMMON IMPORT PROBLEMS

When it comes time to import your new non-SAS data into SAS, there are a number of issues that you may encounter. One issue that I have encountered with multiple files is embedded line feed (LF or \n) indicators. Ordinarily, the end of a line is indicated by the combination of a carriage return indicator (CR or \r) and LF (Display 5).

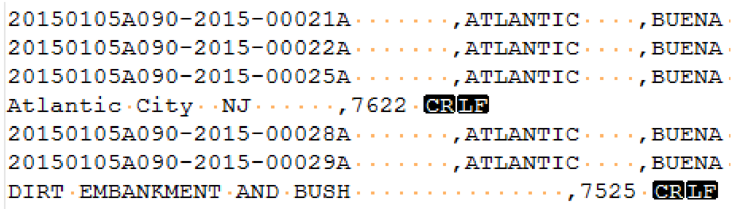


```
N,                                                                                ,7594 CRLF
N,None.                                                                           ,7700 CRLF
N,Pole #SJ1525B  Atlantic City Electric  2430 Atlantic AveLF

N,Mailbox Thomas Veneziale 215 Sherwood Drive Williamtown  NJ 08094               ,7203 CRLF
N,430 WEYMOUTH ROAD  BUENA VISTA  NJ 08310LF
```

Display 5. End of Line of CSV File Opened in Notepad++ Showing Usual End Marker and Embedded Line Feed Indicator

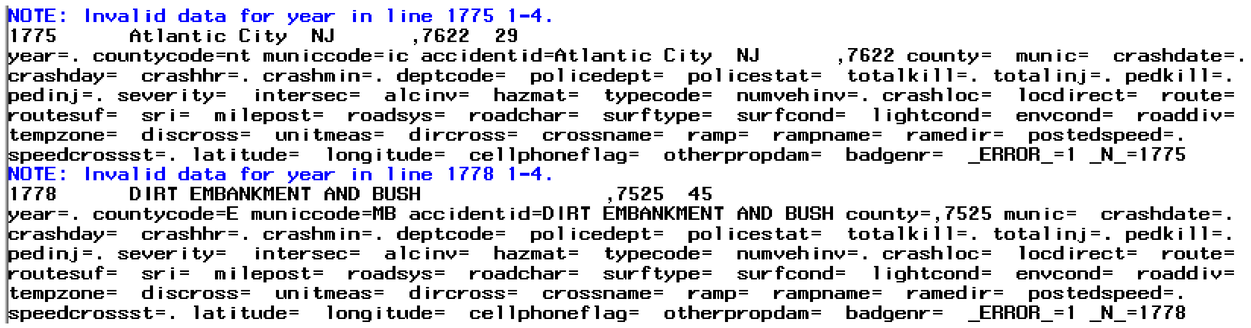
However, if there is an embedded LF indicator within a field, SAS will read that as the end of a record and attempt to read the remainder of that record into a new observation (Display 6).



```
20150105A090-2015-00021A       ,ATLANTIC    ,BUENA
20150105A090-2015-00022A       ,ATLANTIC    ,BUENA
20150105A090-2015-00025A       ,ATLANTIC    ,BUENA
Atlantic City  NJ      ,7622 CRLF
20150105A090-2015-00028A       ,ATLANTIC    ,BUENA
20150105A090-2015-00029A       ,ATLANTIC    ,BUENA
DIRT EMBANKMENT AND BUSH                ,7525 CRLF
```

Display 6. CSV File Opened in Notepad++ Showing Record Carried Over to Next Line

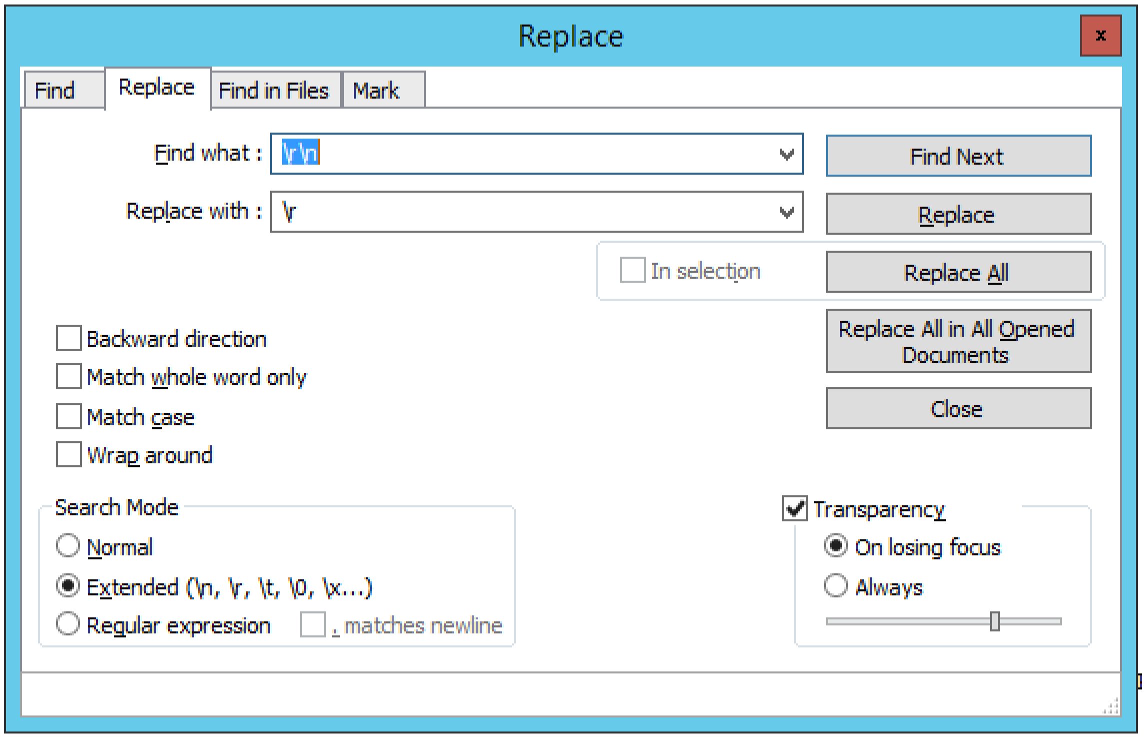
This can cause errors if the remainder of the information cannot fit into the SAS variables as prescribed and can cause problems if it does fit but does not belong in those variables (Display 7).



```
NOTE: Invalid data for year in line 1775 1-4.
1775      Atlantic City  NJ      ,7622  29
year=. countycode=nt municcode=ic accidentid=Atlantic City  NJ      ,7622 county=  munic=  crashdate=.
crashday=  crashhr=. crashmin=. deptcode=  policedept=  policestat=  totalkill=. totalinj=. pedkill=.
pedinj=. severity=  intersec=  alcinv=  hazmat=  typecode=  numvehinv=. crashloc=  locdirect=  route=
routesuf=  sri=  milepost=  roadsys=  roadchar=  surftype=  surfcond=  lightcond=  envcond=  roaddiv=
tempzone=  discross=  unitmeas=  dircross=  crossname=  ramp=  rampname=  ramedir=  postedspeed=.
speedcrossst=. latitude=  longitude=  cellphoneflag=  otherpropdam=  badgenr=  _ERROR_=1 _N_=1775
NOTE: Invalid data for year in line 1778 1-4.
1778      DIRT EMBANKMENT AND BUSH                ,7525  45
year=. countycode=E municcode=MB accidentid=DIRT EMBANKMENT AND BUSH county=,7525 munic=  crashdate=.
crashday=  crashhr=. crashmin=. deptcode=  policedept=  policestat=  totalkill=. totalinj=. pedkill=.
pedinj=. severity=  intersec=  alcinv=  hazmat=  typecode=  numvehinv=. crashloc=  locdirect=  route=
routesuf=  sri=  milepost=  roadsys=  roadchar=  surftype=  surfcond=  lightcond=  envcond=  roaddiv=
tempzone=  discross=  unitmeas=  dircross=  crossname=  ramp=  rampname=  ramedir=  postedspeed=.
speedcrossst=. latitude=  longitude=  cellphoneflag=  otherpropdam=  badgenr=  _ERROR_=1 _N_=1778
```

Display 7. SAS Log Showing Invalid Data Note Because of Record Carried Over to Second Line

Rather than searching your original file for each of these indicators and correcting them individually, I employ a three step "find and replace" process to remove all instances. When doing the find and replace in Notepad++, the "extended" search mode needs to be selected to find these special characters (Display 8).



Find what : \r\n

Replace with : \r

☐ Backward direction
☐ Match whole word only
☐ Match case
☐ Wrap around

Search Mode
○ Normal
● Extended (\n, \r, \t, \0, \x...)
○ Regular expression ☐ . matches newline

☑ Transparency
● On losing focus
○ Always

Display 8. Find and Replace in Notepad++ With Extended Search Mode Selected

Step 1: I replace all CRLF (\r\n) indicators—indicators of true line ends—with CR (\r). This takes the correct end-of-line indicators and replaces them with just a carriage return.

Step 2: I replace all LF (\n) indicators with a space. This takes the embedded LF indicators within records—the ones causing problems—and swaps them for a space. Within SAS, you can always use the COMPBL function to translate two or more consecutive blanks into a single blank if needed.

Step 3: I replace CR (\r) with CRLF (\r\n) so that all records only occupy a single line (again).

The occurrence of this issue may depend on how the original file was created. I have encountered this issue with both CSV files and TXT files. If possible, it would be better to avoid the issue in the first place than fix it once it occurs. I noticed that embedded LF indicators were generated when I created a CSV file using API functionality but did not appear when I did a direct export from the original source (REDCap secure web application). If there is more than one way to create your CSV file, it might be worth trying a different option if you find this problem. Additionally, my colleague found that this issue existed in a CSV file she received but was not present when the data were prepared as an XML file. If you can request a different file type, the problem may be avoided.

An import without any obvious errors does not mean that your data are all as they should be. If you do not receive dictionary or file layout information indicating the necessary length for character variables, it is possible that values were truncated during import because the length of the SAS variable was not long enough to hold the entire value. One check you can conduct to determine if your variables are long enough is to find the maximum length of the values for each character variable in your new data set and compare that against the assigned variable lengths. To be certain that values were not truncated, the maximum length of the values in each character variable should be smaller than the length assigned to that variable. For example, if the longest value for variable A is 10 and variable A has an assigned length of 10, information in variable A may have been truncated because only 10 characters were allowed but there were really 15. You can check this with relatively simple, flexible code in such a way that you do not need to know how many character variables are in your data set or their names. First, use PROC SQL to capture the names of all of the character variables in your data set and put them into a macro variable (here named charvars), with the variable names separated by a space. You can use another very similar SELECT statement in PROC SQL to create an additional macro variable (here named lengthvars). This macro has the same character variable names, but each variable name is preceded with "l_"—these variables will be used to hold the length of each value:

```
proc sql noprint ;
   select name
    into :charvars
       separated by ' '
    from dictionary.columns
    where libname = 'WORK' and memname = 'PATIENT_IMPORT' and type = 'char'
   ;
   select cats('l_', name)
    into :lengthvars
       separated by ' '
    from dictionary.columns
    where libname = 'WORK' and memname = 'PATIENT_IMPORT' and type = 'char'
   ;
 quit ;
```

Then use a data step to populate the "l_" variables with the actual length of the values for each record:

```
data _testlength ;
   set patient_import ;
   array orig{*} &charvars ;
   array lengths{*} &lengthvars ;
   do _i = 1 to dim(orig) ;
      lengths{_i} = lengthn(orig{_i}) ;
   end ;
run ;
```

Finally, use the SUMMARY procedure to display the maximum length of the values in each character variable:

```
proc summary data = _testlength max print ;
   var l_: ;
run ;
```

Now that you know the length of the longest value for each variable, you can check to make certain that each variable is assigned a length at least a little bit longer. If the maximum length is the same as the assigned length, values may have been truncated. There are of course other ways to achieve the same information; use the method that you are most comfortable with or try various ways as a means of expanding your coding repertoire.

An alternative to checking that the variables have lengths sufficient to avoid truncating values is to use PROC IMPORT code to read in your data, setting GUESSINGROWS to MAX:

```
proc import out = PtImport
      datafile = "[your path\filename.csv]"
      dbms = csv <replace> ;
   getnames = yes;
   datarow = 2 ;
   guessingrows = max ;
run ;
```

GUESSINGROWS indicates to SAS the number of rows that should be scanned in order to determine the appropriate data type and length. This should prevent truncation. Note, however, that setting GUESSINGROWS to MAX can adversely affect performance if the input file is large—causing an import that could have taken seconds to take minutes.

CONFIRM YOUR NEW DATA ARE STRUCTURALY SOUND

Once your new data have been safely imported into SAS, it's time to start exploring and making sure the data are reasonably close to what you expect. If you have relational tables—multiple tables that can be joined with key variables—make sure that you do not have orphan observations, or observations that should be in two data sets but only appear in one data set. Again using NJ DOT crash data as an example, there should be no situations in which there is a record in the driver data set that does not connect with a record in the crash data set. (The converse can be true—a crash record can exist with no driver record because of hit-and-run crashes.)

It may also be worth exploring whether you received all of the data you expected. For instance, do data seem to be missing for people born in a certain year? Or incomplete for certain people? I recently needed to follow up with data owners of licensing data because a subset of people were in the main license data set but did not have license-related transactions in a secondary data set. Exploration revealed that my subset only held a driver's permit, not a full driver's license. Since license-related transactions only pertain to full licenses, this group of permitted-only drivers did not have transactions and thus should not have records in the data set of transactions. However, it was reassuring to be able to find this explanation and confirm it with the data owners. Finally, do the data appear as expected or make sense? Do proportions of certain variable values make sense overall and when stratified temporally? For example, proportions of race/ethnicity values that do not match population expectations or that change over time may indicate a change in coding.

GIVING DATA TO OTHERS: FOLLOW THE GOLDEN RULE

Giving data to others can often be easier and more straight-forward than receiving data. Although there may be constraints on the way in which the file is constructed, you have the capability to make a fellow data user's life easier. Follow the "Golden Rule" and give them information you would like to receive.

WAYS TO CREATE DIFFERENT FILES

There are many possibilities for the structure of the data you are creating for someone else, such as file type (e.g., Excel, CSV); whether the column headings contain the variable name, the variable label, or are omitted (and the data begin in row 1); whether the values are formatted or unformatted; and, if creating an Excel file, whether the data are all in one sheet, multiple sheets, or separate files.

PROC EXPORT is, of course, one way to create a non-SAS data file. The export wizard can walk you through the steps so that you do not need to remember the syntax. I recommend including the optional step of that wizard and saving the PROC EXPORT code created so that a) you have a record of what was done and b) you can just re-execute that code (possibly with modifications) when you need to re-generate the file. This happens to me all too often. The basic structure of the code is very similar whether creating an Excel file or a CSV file:

```
proc export data = libref.dataset
      outfile = "[path\filename.xlsx]"
      dbms = xlsx <replace> ;
   sheet = "sheetname" ;
run ;

proc export data = libref.dataset
      outfile = "[path\filename.csv]"
      dbms = csv <replace> ;
   putnames = yes ;
run ;
```

In both of those code examples, an Excel or CSV file is created that has the variable names as the first row. If you want the label instead, simply add the LABEL option to the PROC EXPORT statement (e.g., “DBMS = XLSX/CSV LABEL”).

If you do not want a header row with either the variable names or labels (i.e., you want the data to start in row 1), then change the option on the PUTNAMES statement to “no” instead of “yes.” This statement works for both CSV and Excel. One key difference between these two export procedures is that the data created with DBMS = XLSX are unformatted, whereas the data created with DBMS = CSV use user-defined formats.

Another way to produce an Excel file with data from a SAS data set is to use the Excel libname engine. Once that is assigned, you can populate your new Excel file with data through a data step or through a procedure like PROC COPY:

```
libname makexls excel ‘[path\filename.xls]’ ;
proc copy in = work out = makexls ;
   select dsname ;
quit ;

libname makexls clear ;
```

Note that you will not be able to access your new Excel file until you submit the LIBNAME statement that clears the libref and releases SAS’s hold on the file (unless you want to open a read-only copy). It is also worth mentioning that if you make a copy of a sorted data set or you use the SORT procedure to create your Excel file, you will get a warning or note about Excel not supporting SORTEDBY operations or a sort indicator not being set. If you use PROC COPY (or PROC DATASETS) to create an Excel copy of your data, the sheet will have the data set name, whereas if you use a data step or procedure in which you can name our new data set (like PROC SORT), the sheet will have the name you supply for your new data set.

Finally, you can generate Excel or CSV files by using the Output Delivery System (ODS). The nuances and capabilities of ODS are extensive; this is meant as a brief introduction. Further exploration is encouraged! With ODS, you can output information from a SAS procedure, such as PROC PRINT or PROC REPORT, directly into a CSV or Excel file. For CSV, simply open the CSV ODS destination, submit your procedure, and then close the destination again:

```
ods csv file = ‘[path\filename.csv]’ ;
[your procedure]
ods csv close ;
```

There is also an ODS EXCEL destination, but I typically instead use the TAGSETS.EXCELXP destination to create an Excel file. As mentioned, there are many capabilities of this worth exploring, such as different style templates, embedding titles and footnotes, naming your new sheets, and controlling how many tables are in a single worksheet. One of my favorite features of this destination is the ability to create more than one sheet in a single Excel file. This can be particularly handy when the data you are distributing needs to be separated for certain subjects, such as data for cases in one sheet and data for controls in another. As discussed below, you can also use this to create a single Excel file that has one sheet with a data dictionary and another with meanings for user-defined formats. Your SAS session will produce documentation about the EXCELXP destination in the log if you submit the following statements:

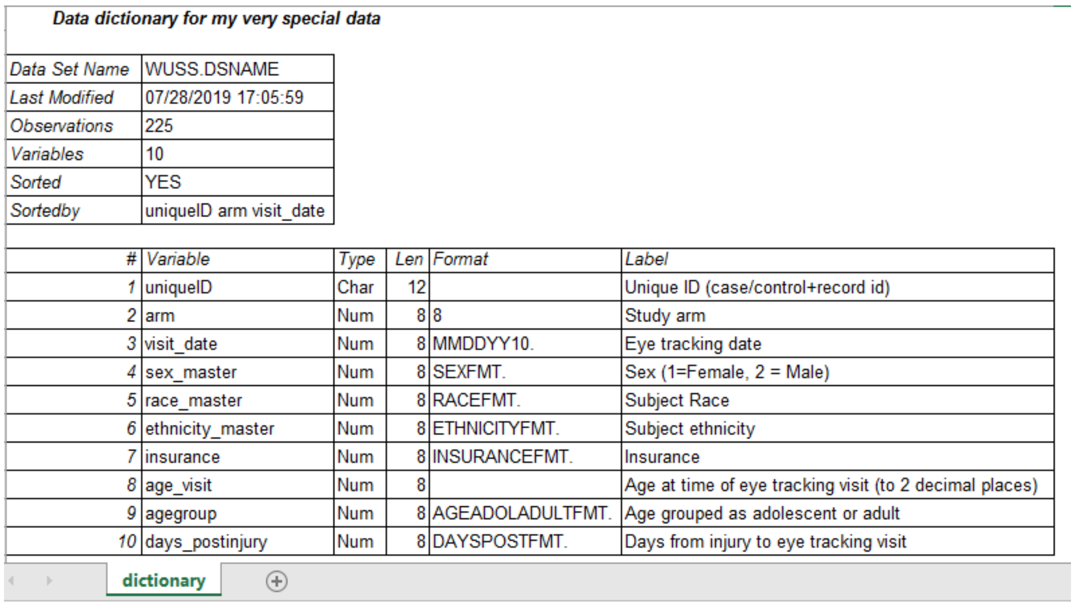
```
ods tagsets.excelxp file = ‘[path\file name.xml]’ options(doc=‘help’) ;
ods tagsets.excelxp close ;
```

SUPPLY A DATA DICTIONARY (EVEN A SIMPLE ONE)

It can be very frustrating to receive data without information about the data, such as what variables mean. Instead of passing along that kind of frustration to someone else, supply a data dictionary when you transfer data to another person. The data dictionary does not have to be extensive (after all, everyone should investigate data they receive), but it can provide a starting point to orient someone to new data and it can minimize some of the questions you receive about your data. One method is to use PROC CONTENTS in conjunction with ODS TAGSETS.EXCELXP to quickly create a simple data dictionary in Excel and then modify the output created:

```
ods tagsets.excelxp file = '[path\file name.xml]' ;
ods tagsets.excelxp options (embedded_titles = 'yes' sheet_name =
      'dictionary' sheet_interval = 'none') ;
titel1 'Data dictionary for my very special data' ;
proc contents data = dsname varnum ;
run ;
ods tagsets.excelxp close ;
```

I find that the most useful parts of this output have to do with the file name, modification date, number of observations and variables, sort order, and many of the fields related to the variables themselves. Consequently, I take a few moments to delete information (in rows and columns) that I do not need and shift similar information to be closer together. In no time at all, I have an Excel file that provides basic information to a new data user (Display 9).



Data dictionary for my very special data

Data Set Name	WUSS.DSNAME
Last Modified	07/28/2019 17:05:59
Observations	225
Variables	10
Sorted	YES
Sortedby	uniqueID arm visit_date

#	*Variable*	*Type*	*Len*	*Format*	*Label*
1	uniqueID	Char	12		Unique ID (case/control+record id)
2	arm	Num	8	8	Study arm
3	visit_date	Num	8	MMDDYY10.	Eye tracking date
4	sex_master	Num	8	SEXFMT.	Sex (1=Female, 2 = Male)
5	race_master	Num	8	RACEFMT.	Subject Race
6	ethnicity_master	Num	8	ETHNICITYFMT.	Subject ethnicity
7	insurance	Num	8	INSURANCEFMT.	Insurance
8	age_visit	Num	8		Age at time of eye tracking visit (to 2 decimal places)
9	agegroup	Num	8	AGEADOLADULTFMT.	Age grouped as adolescent or adult
10	days_postinjury	Num	8	DAYSPOSTFMT.	Days from injury to eye tracking visit

Display 9. Example of Data Dictionary in Excel Produced by Proc Contents

With a few modifications to your SAS code, you could produce output that is a bit closer to the final product. Use an ODS SELECT statement before a PROC CONTENTS to only output the data set attributes and sortedby information, omitting the enginehost and position sections of output. Then use another ODS SELECT statement before another PROC CONTENTS to request just the position information, omitting all other sections:

```
ods select attributes sortedby ;
proc contents data = dsname varnum ;
run ;
ods select position ;
proc contents data = dsname varnum ;
run;
```

If you prefer to just have one ODS SELECT statement with one PROC CONTENTS, asking for the three sections attributes, sortedby, and position (in other words, asking that enginehost be omitted), the sortedby section will be last in the output created regardless of where it appears in the ODS SELECT statement:

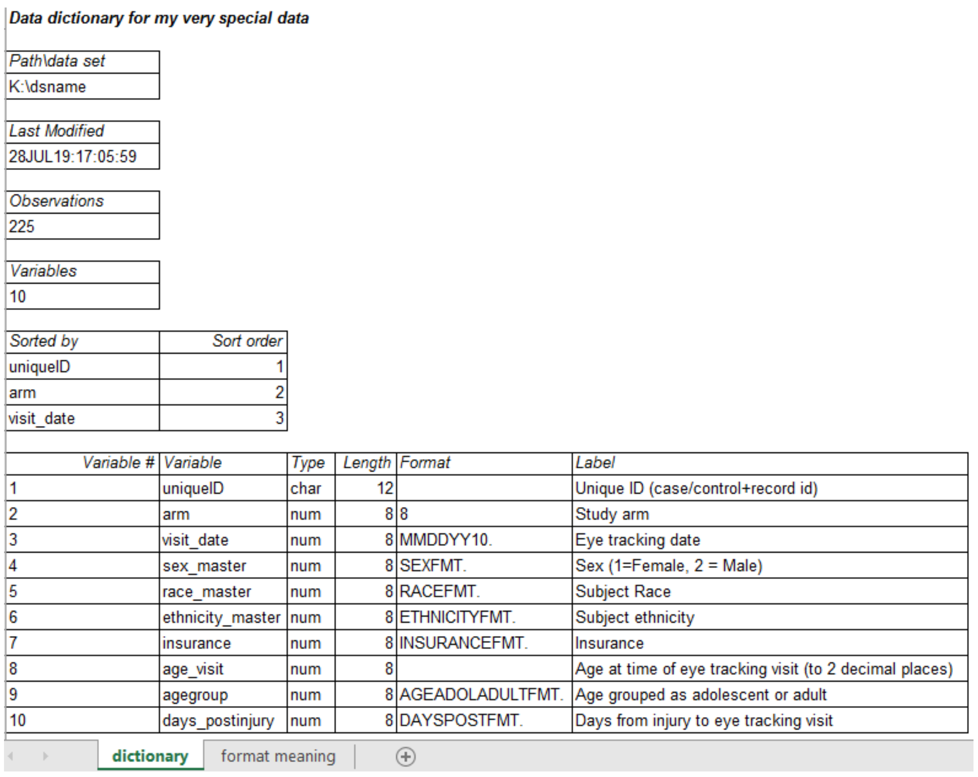
```
ods select attributes sortedby position ;
proc contents data = dsname varnum ;
run ;
```

If you forget the names of the different sections, you can submit the statement “ODS TRACE ON;” before a PROC CONTENTS—the sections produced by the procedure will be described in the log, including the name you need to use to refer to the section. Just don’t forget to turn that option back off (“ODS TRACE OFF;”) when you have the information you need.

By tapping into the dictionary tables, you could output the limited data that you need with a few PROC SQL SELECT statements (and the same ODS TAGSETS.EXCELXP statements):

```
title1 ‘Data dictionary for my very special data’ ;
proc sql ;
   select distinct catx(‘\’, path, ‘dsname’) label = ‘Path\data set’
      from dictionary.libnames
      where libname = ‘WUSS’
   ;
quit
title1 ;
proc sql ;
   select modate label = ‘Last Modified’
      from dictionary.tables
      where libname = ‘WUSS’ and memname = ‘DSNAME’
   ;
   select nobs label = ‘Observations’
      from dictionary.tables
      where libname = ‘WUSS’ and memname = ‘DSNAME’
   ;
   select nvar label = ‘Variables’
      from dictionary.tables
      where libname = ‘WUSS’ and memname = ‘DSNAME’
   ;
   select name label = ‘Sorted by’, sortedby label = ‘Sort order’
      from dictionary.tables
      where libname = ‘WUSS’ and memname = ‘DSNAME’ and sortedby > 0
      order by sortedby
   ;
   select varnum label = ‘Variable #’, name label = ‘Variable’,
         type label = ‘Type’, length label = ‘Length’,
         format label = ‘Format’, label label = ‘Label’
      from dictionary.tables
      where libname = ‘WUSS’ and memname = ‘DSNAME’
   ;
quit ;
```

With a little bit of editing in Excel, this data dictionary is produced. Since the code has two PROC SQL steps with a null TITLE statement between them instead of a single PROC SQL step, the title is not repeated before each section (Display 10).


Data dictionary for my very special data

Path\data set
K:\dsname

Last Modified
28JUL19:17:05:59

Observations
225

Variables
10

Sorted by	*Sort order*
uniqueID	1
arm	2
visit_date	3

Variable #	*Variable*	*Type*	*Length*	*Format*	*Label*
1	uniqueID	char	12		Unique ID (case/control+record id)
2	arm	num	8	8	Study arm
3	visit_date	num	8	MMDDYY10.	Eye tracking date
4	sex_master	num	8	SEXFMT.	Sex (1=Female, 2 = Male)
5	race_master	num	8	RACEFMT.	Subject Race
6	ethnicity_master	num	8	ETHNICITYFMT.	Subject ethnicity
7	insurance	num	8	INSURANCEFMT.	Insurance
8	age_visit	num	8		Age at time of eye tracking visit (to 2 decimal places)
9	agegroup	num	8	AGEADOLADULTFMT.	Age grouped as adolescent or adult
10	days_postinjury	num	8	DAYSPOSTFMT.	Days from injury to eye tracking visit

Display 10. Data Dictionary Created by Alternate Code

INCLUDE FORMAT INFORMATION

A critical component of providing information about your data is including information about the meaning behind coded values. This can be easily accomplished by providing information about your user-defined formats. Many of us have copious amounts of user-defined formats; including all of them when supplying data to a new user is both unnecessary and potentially confusing. However, you can easily whittle down the format information to just those relevant to the variables in your data set. There are, of course, many ways to produce a document with information about user-defined formats. One strategy is to capitalize on the information in the dictionary.columns table to limit the user-defined formats to those in the data set you are releasing, create a SAS data set with the format information, and then create a report based on that data.

First, use the dictionary.columns table to acquire the formats attached to your data set (here called wuss.dsname), putting those format names (without the final period) into a macro variable (here called formatlist), separated by spaces. Variables without formats should be ignored (null values), and you may want to disregard numeric and date formats:

```
proc sql noprint ;
   select distinct scan(format, 1, ‘.’)
   into :formatlist separated by ‘ ’
   from dictionary.columns
   where libname = ‘WUSS’ and memname = ‘DSNAME’
      and format not in (‘ ’, ‘MMDDYY10.’ ‘8.’)
   ;
quit ;
```

Then use proc format to create a data set (here called formatds) that has all of the user-defined formats selected from the data set “dsname” —except those excluded—as well as their start values, end values, meanings, etc:

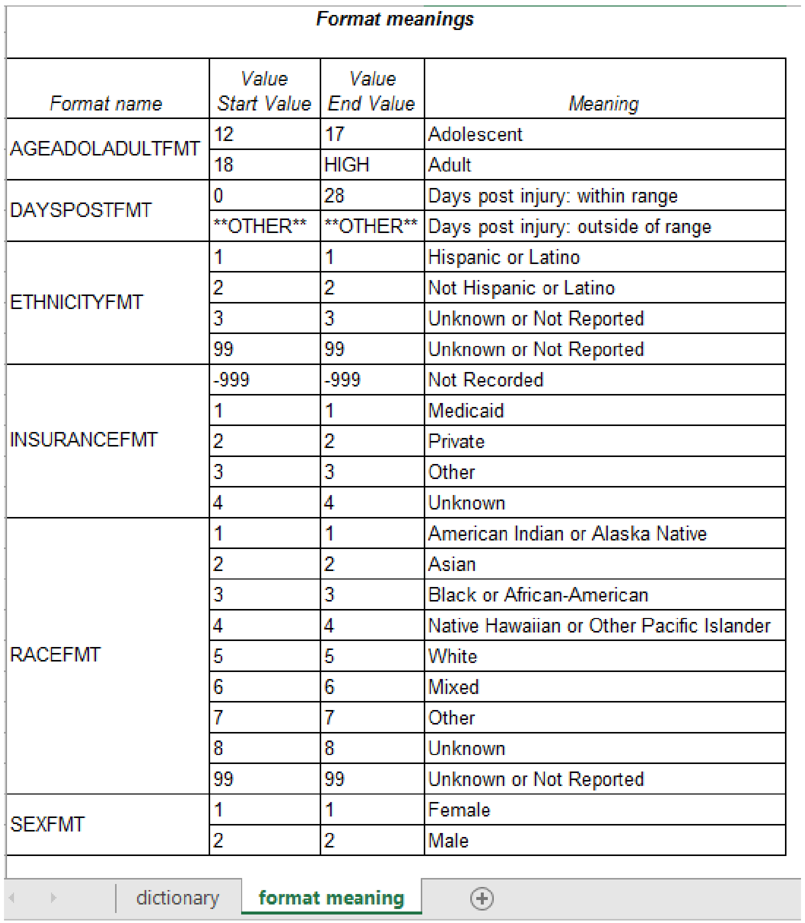
```
proc format library = work.formats cntlout = formatds ;
```

```
      select &formatlist ;
   run ;
```

The contents of that data set can then be reported out into a different sheet within the same Excel document that contains your data dictionary information:

```
   ods tagsets.excelxp options (sheet_name = 'format meaning') ;
   title1 'Format meanings' ;
   proc report data = formatds ;
      column fmtname start end label ;
      define fmtname / order ;
      define start / display 'Value/Start Value' ;
      define end / display 'Value/End Value' ;
      define label / display format = $45. 'Meaning' ;
   run ;
```

That process will generate a list of the formats associated with the variables in this specific data set (rather than your entire format library) and can provide critical insights into your data for the person receiving it (Display 11).



Format meanings

Format name	*Value Start Value*	*Value End Value*	*Meaning*
AGEADOLADULTFMT	12	17	Adolescent
	18	HIGH	Adult
DAYSPOSTFMT	0	28	Days post injury: within range
	OTHER	**OTHER**	Days post injury: outside of range
ETHNICITYFMT	1	1	Hispanic or Latino
	2	2	Not Hispanic or Latino
	3	3	Unknown or Not Reported
	99	99	Unknown or Not Reported
INSURANCEFMT	-999	-999	Not Recorded
	1	1	Medicaid
	2	2	Private
	3	3	Other
	4	4	Unknown
RACEFMT	1	1	American Indian or Alaska Native
	2	2	Asian
	3	3	Black or African-American
	4	4	Native Hawaiian or Other Pacific Islander
	5	5	White
	6	6	Mixed
	7	7	Other
	8	8	Unknown
	99	99	Unknown or Not Reported
SEXFMT	1	1	Female
	2	2	Male

Display 11. Excel Sheet With Meanings of Formats for Specific Data Set

If your formats are always for a single value (e.g., 0 = Female, 1 = Male), you can just report the start or end value, since they will be the same. However, if some of your formats are or could be a range (e.g., 13 – 17 = Adolescent), then you will want to have both the start and end values included. You will also want to use both start and end values if your formats include the keywords "high" or "low." Additionally, you may want to check on the length of the longest text for the format labels and be sure that the format you use in your DEFINE label statement is long enough.

CONCLUSION

Modern day communication takes on many forms and certainly includes information exchanges in the form of sharing data sets. We can all work to enhance our communication by doing our best to understand the data that we receive as well as trying to provide details about the data that we dispense. This paper by no means exhausts this topic—there are many other considerations as well as other ways to accomplish the activities described. Hopefully it highlights some of the issues to be aware of and provides some recommendations for how to accomplish the goal of keeping all of us informed about our data.

ACKNOWLEDGMENTS

Many thanks to Allison Curry for her continued support of my SAS geekiness; to my team of SAS enthusiasts including Meghan Carey, Fairuz Mohammed, and Rachel Myers; and especially to Kristi Metzger for providing unending insights, encouragement, and a second pair of eyes.

CONTACT INFORMATION

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

Melissa R. Pfeiffer
PfeifferM@email.chop.edu