Paper 4839-2020

Read Before You Read: Reading, Rewriting & Re-Reading Difficult Delimited Data in a Data Step

Michael Chu, TD Bank

ABSTRACT

Loading delimited data within the DATA step can get interesting/frustrating quickly if you have quirky data. The abrupt appearance of a delimiter in an unquoted character field shifts all following fields to the right; the random removal of one shifts fields to the left. Left unchecked, the line will not get processed correctly, as SAS® attempts to read each separated field using its neighbour's INFORMAT.

Thankfully, there is potentially a way to spot issues like these, namely via the "INPUT @" statement. What's more, it may also be possible to correct them on-the-fly by directly modifying the "_INFILE_" automatic variable. This additional coding can be injected into the existing DATA step code such that the original INPUT statement(s) can continue to function properly even when faced with the difficult delimited data.

This paper provides an in-depth exploration of the approach outlined above. Readers can immediately test out this concept using the supplied code. Other potential workarounds are also touched upon. After digesting this information, readers will possess another method to ingest raw data elegantly into a SAS dataset.

INTRODUCTION

The delimited file format ought to be a reliable choice for sharing data in an error-free manner. As a plain text file, it is easy to parse, with each record typically written out as a single line, and a chosen character, the delimiter, that separates data fields within the record. This use of a delimiter is the file format's strength and weakness: it works great when the file creator follows the basic rules about how to generate them, with each line/record of data having the same number of delimiters, and therefore fields. Things can go downhill quickly once this is no longer the case.

Consider the situation of delimiters that exist as part of the data, for example a CSV with a field "Name" that stores the surname followed by the given name and separated by a comma, as in: Smith, John. When the program comes across data like this, it recognizes that comma as a delimiter and splits the data at that point: "Name" is simply "Smith", its neighbouring field is "John" and every single following field gets shifted one to the right. To counter this, we can wrap text fields in quotation marks; this lets the reader know that any delimiter character found within should be treated as data and not a field separator. Unfortunately, not all report generators use this standard convention, which makes for a lot of frustrated SAS users left with lots of bad data.

Another situation involves data files that are missing delimiter(s) in some records. Unlikely as that sounds, it is possible. Take for example a concatenation of feed files, where one source system decided a field was unnecessary and removed it entirely. No matter what the cause, the result is similar to the first situation: all subsequent fields are shifted one over and read in using their neighbour's INFORMAT. The key difference here is that adding quotation marks does not help; we don't need to mask the presence of a delimiter character, after all.

Thankfully, there is potentially a way to spot issues like these from within the DATA step that ingests the file. By adding the "INPUT @" statement, we can make SAS read a line into memory without attempting to parse it. We can follow that with an inspection of the "_INFILE_" automatic variable, which lets us see the entire line – and if there are any problems with it. Furthermore, this same variable can be modified, and any changes made to it are reflected in the remaining INPUT statements of the DATA step. In other words, the combination of these two elements gives us the ability to detect and correct delimiter issues on-the-fly within our normal DATA steps.

In the rest of this paper, we will provide complete examples of each of the situations described above, and how this technique of combining "INPUT @" with "_INFILE_" can potentially resolve them.

WHAT CAN GO WRONG WITH DELIMITED DATA

A common issue with delimited data files is the presence of the delimiter character in an unquoted text field. As discussed in the introduction, the field will be split at that character: the left half gets assigned to the text field and the right half gets assigned to the following field, if there is one. The pipe-delimited file in Figure 1 below demonstrates this; the offending pipe is circled in red, and the fields are colour-coded for your convenience.

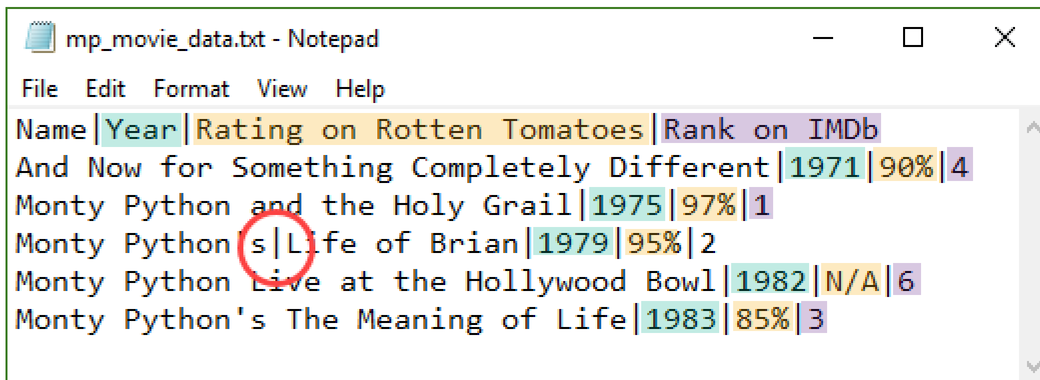


Figure 1: a pipe-delimited file of movie data containing a pipe within an unquoted field

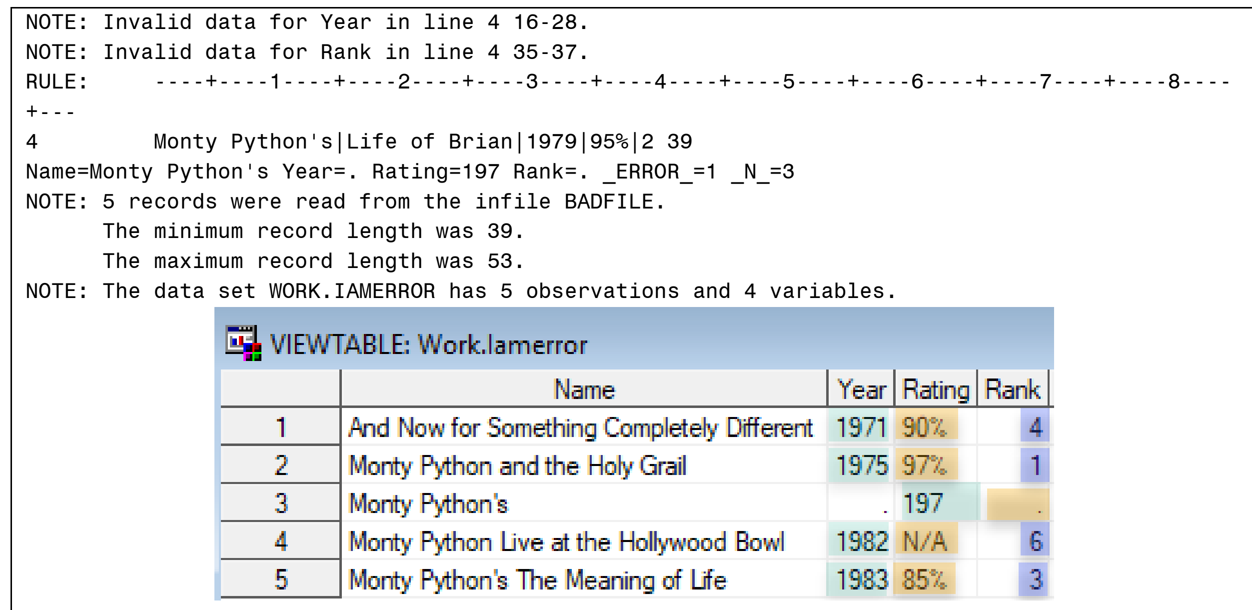
Looking at the file, we recognize that the first pipe on line 4 should not be treated as a delimiter, yet this is precisely what will happen. Consider the simple DATA step in Figure 2 below, which would execute error-free if not for this extra pipe. Since the "Name" field is unquoted, there is no benefit to adding the DSD option to the INFILE statement:

```
FILENAME BADFILE 'C:\TEMP\mp_movie_data.txt';
data iamerror;
  infile BADFILE dlm='|' firstobs=2;
  format Name $50. Year 4. Rating $3. Rank 1. ;
  input  Name      Year   Rating     Rank;
run;
```

Figure 2: a DATA step to import the pipe-delimited text file from Figure 1

Submitting this code will generate errors as expected, as SAS tries to load the parsed data into the (incorrect) neighbouring fields. No errors are generated for the character variable

"Rating", but it certainly still counts as bad data. Figure 3 below shows the SAS log on submitting the code, along with the resulting dataset:



```
NOTE: Invalid data for Year in line 4 16-28.
NOTE: Invalid data for Rank in line 4 35-37.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----
+---
4         Monty Python's|Life of Brian|1979|95%|2 39
Name=Monty Python's Year=. Rating=197 Rank=. _ERROR_=1 _N_=3
NOTE: 5 records were read from the infile BADFILE.
      The minimum record length was 39.
      The maximum record length was 53.
NOTE: The data set WORK.IAMERROR has 5 observations and 4 variables.
```

VIEWTABLE: Work.Iamerror

	Name	Year	Rating	Rank
1	And Now for Something Completely Different	1971	90%	4
2	Monty Python and the Holy Grail	1975	97%	1
3	Monty Python's	.	197	.
4	Monty Python Live at the Hollywood Bowl	1982	N/A	6
5	Monty Python's The Meaning of Life	1983	85%	3

Figure 3: the SAS log after submitting the DATA step from Figure 2, and the resulting dataset

WHAT CHOICE DO I HAVE?

When tasked with importing difficult delimited files such as the one above, you have a few options to try. If you know who creates the file, you could ask them to enclose all character fields in quotation marks - thus allowing you to use the DSD flag for the INFILE statement. But let's suppose that route doesn't pan out. Another option is to pre-process it outside of SAS; as a plain text file, you could simply open the file in a text editor to make the required correction. This is a reasonable option for a one-time ad hoc job, but if the file is a periodic report that you will be importing regularly, a manual correction step loses its appeal.

Pre-processing is still a viable choice if you can script it, and there are plenty of tools available to do so in a pre-processing step (e.g. PowerShell, Python, AWK, Perl, sed). No matter what tool you consider, one thing remains true: you must design an algorithm to detect the problematic delimiter character and correct it. For example, if the file contains a single record with an unquoted delimiter, you can specify a rule that targets that specific record. Going back to our movie data from Figure 1 above, some sample pseudo-code might look like this:

1) Read one line from original text file
2) If the line starts with "Monty Python's|Life of Brian" then
   a) Replace the start of line with "Monty Python's Life of Brian"
3) Write line to corrected output text file

While this sounds like a decent way to go, consider the following disadvantages of pre-processing your data. The first is maintenance: you now have an external script to run and keep updated; if your colleagues are unfamiliar with the selected tool then it may become your sole responsibility. The second is the additional time and computer resources required

to execute the step, e.g. the CPU cycles spent performing the fix, the extra disk space required to store the corrected copy.

But there is a better (SAS) way that has the advantages of a scripted pre-processing step and can be performed on-the-fly within the DATA step that ingests the delimited file.

THE TRICK: "INPUT @" AND "_INFILE_"

You may already be familiar with the "trailing @" for INPUT statements. In a nutshell, by adding the "@" to the end of your INPUT statement, you prevent SAS from moving on to the next line of the input file. This allows you to do things like read in the record type of the current line, then decide what INPUT statement to use for the remaining fields of that record. It is a decidedly powerful feature when ingesting data files, and it is capable of more.

A "null INPUT statement" is an INPUT statement that has no arguments. It loads the record/line into memory without trying to parse it, so it never generates any errors even when bringing in bad data. By adding the "trailing @" we instruct SAS to load the record without trying to parse it, and to stay on that record so we can read it later using our normal INPUT statement. The first part of our trick is simply this null INPUT statement variation, as shown in Figure 4 below:

```
   input @;   /* Bring the record into memory */
```

Figure 4: a variation on the null INPUT statement; the first half to the trick

The second part of our trick is to inspect the line that was loaded in and modify it if it contains an unquoted delimiter character. The way to do that is via an automatic SAS variable called "_INFILE_", which is so critical to this technique it deserves mention within Figure 5 below:

```
   _INFILE_   /* The SAS variable that lets us implement this technique */
```

Figure 5: the very important _INFILE_ automatic variable. Not a gratuitous Figure at all

Within the DATA step, once the "INPUT @" statement is executed, the entire line that was read is accessible within this _INFILE_ variable. It works like a normal character variable, meaning you can apply any string functions you want to inspect the line. Additionally, this also means that when you determine a correction is required, you can simply modify the _INFILE_ variable to make that change.

What makes this all work is the fact that any changes made to the _INFILE_ variable are reflected in the remaining INPUT statements of the DATA step. In other words, injecting these two pieces of code into your DATA step fixes the delimiter error on-the-fly and allows the original INPUT statement to run without error.

Figure 6 below is a modified copy of the DATA step from Figure 2. We inject 3 lines that implement the pseudo-code from the previous section, which tests if the line begins with "Monty Python's|Life of Brian" and fixes it if so:

```
FILENAME BADFILE 'C:\TEMP\mp_movie_data.txt';
data targeted_fix;
  infile BADFILE dlm='|' firstobs=2;
  format Name $50. Year 4. Rating $3. Rank 1. ;
  input @;
  if substr(_INFILE_, 1, 28) eq "Monty Python's|Life of Brian" then
    _INFILE_ = "Monty Python's Life of Brian" || substr(_INFILE_, 29);
  input  Name      Year    Rating     Rank;
run;
```

Figure 6: implementing a targeted fix for the sample data

FROM TRICK TO TECHNIQUE

The sample code above does the trick but is only good for fixing a handful of problematic lines. By generalizing the inspection and modification of the _INFILE_ variable, we can potentially create a DATA step that removes all erroneous delimiters from the record lines. Instead of looking for specific records to fix, we can instead look for the effect an extra delimiter would have on the rest of the line. And instead of replacing with an entire chunk of the line, we can opt to replace or remove just that delimiter.

The implementation will differ from file to file, but the approach for inspection is as follows:

1) Identify the closest mandatory field that is to the right of the problematic character field and has a limited number of expected values
2) Gather up all the possible values for this mandatory field, or define some rules about those values
3) Inject code into the DATA step to:
   a) Bring a record/line into memory
   b) Pull out the characters that SAS would try to assign to the mandatory field
   c) If this string has a suitable value, then do nothing
   d) Otherwise, fix the record by removing/replacing one of the prior delimiters

To demonstrate this, we will return to our movie data file from Figure 1. The problematic character field is the first field, "Name". Its immediate neighbour, "Year", appears to meet our needs. Let's explore this:

1) Field #2, "Year", is always populated and appears to store years, which limits its possible values
2) We can define a rule for the expected data of field #2 as a number with exactly 4 digits
3) The code we will inject between the INFILE and existing INPUT statements:
   a) "INPUT @", to bring the line into memory
   b) Pull out the 4 characters that follow the first delimiter seen in the _INFILE_ variable
   c) Test if all 4 characters are numeric
   d) If not, replace the first delimiter with a space

Figure 7 below shows the full DATA step that implements the process described above:

```
FILENAME BADFILE 'C:\TEMP\mp_movie_data.txt';
data r4d4 (drop=DLM1at field2);
  infile BADFILE dlm='|' firstobs=2;
  format Name $50. Year 4. Rating $3. Rank 1. ;
  * 3a) Bring in the record so we can perform the "bad value" test;
  input @;
  * 3b) Find first delimiter, then grab next 4 characters;
  DLM1at = find(_INFILE_, '|');
  length field2 $4;
  field2 = substr(_INFILE_, DLM1at + 1, 4);
  * 3c) Test if all 4 characters are numeric;
  if lengthn(compress(field2, '1234567890')) ne 0 then do;
    * 3d) If not, replace the first delimiter with a space;
    _INFILE_ = substr(_INFILE_, 1, dlm1at - 1) || ' ' ||
               substr(_INFILE_, dlm1at + 1);
  end;
  input   Name      Year    Rating     Rank;
run;
```

Figure 7: implementing a generalized inspection & modification for the sample data from Figure 1

APPLYING THIS TRICK TO HANDLE MISSING DELIMITERS

Missing delimiters is the other half of this problem space. When SAS reads a record with a missing delimiter, fields starting at that point get shifted one to the left and read in using their neighbour's INFORMAT. Consider the tilde-delimited file in Figure 8 below, in which the second field "Mode" is missing from 3 of the 5 records:

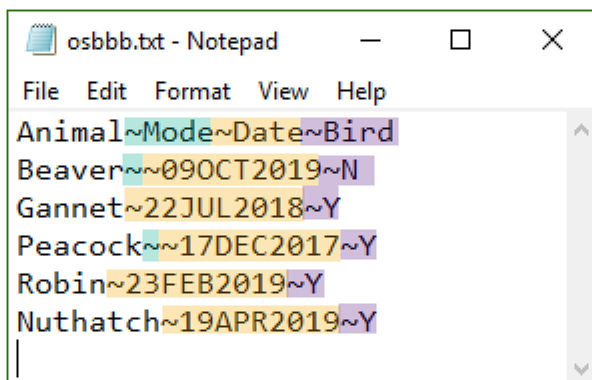


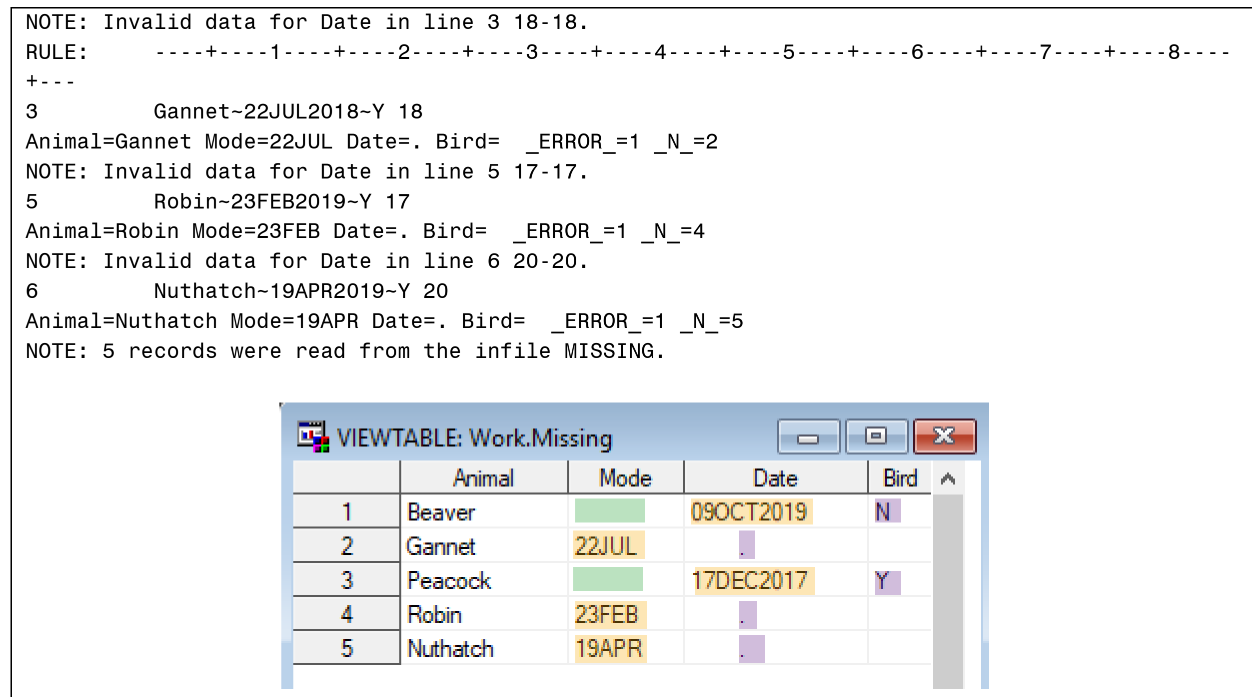
Figure 8: a delimited data file that is missing a delimiter in some of its records

As can be seen, the "Mode" field is not populated even when present but is required nonetheless by our file's defined structure. A missing delimiter adds another complication: since the record is considered short by one field, SAS will read in the next line to grab the remaining required fields. That is, unless an option like TRUNCOVER is provided. The DATA step in Figure 9 below uses the DSD option to properly handle the empty "Mode" field, and TRUNCOVER to prevent SAS from reading the next line in error:

```
FILENAME MISSING 'C:\TEMP\osbbb.txt';
data missing;
  infile MISSING dlm='~' DSD firstobs=2 truncover;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  input  Animal      Mode      Date:DATE9.  Bird;
run;
```

Figure 9: a DATA step to import the tilde-delimited text file from Figure 8

Submitting this code will generate errors as expected, as SAS tries to load the parsed data into the incorrect neighbouring fields – left-shifted this time. Figure 10 below shows the SAS log on submitting the code, along with the resulting dataset:



```
NOTE: Invalid data for Date in line 3 18-18.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----
+---
3         Gannet~22JUL2018~Y 18
Animal=Gannet Mode=22JUL Date=. Bird=  _ERROR_=1 _N_=2
NOTE: Invalid data for Date in line 5 17-17.
5         Robin~23FEB2019~Y 17
Animal=Robin Mode=23FEB Date=. Bird=  _ERROR_=1 _N_=4
NOTE: Invalid data for Date in line 6 20-20.
6         Nuthatch~19APR2019~Y 20
Animal=Nuthatch Mode=19APR Date=. Bird=  _ERROR_=1 _N_=5
NOTE: 5 records were read from the infile MISSING.
```

VIEWTABLE: Work.Missing

	Animal	Mode	Date	Bird
1	Beaver		09OCT2019	N
2	Gannet	22JUL	.	
3	Peacock		17DEC2017	Y
4	Robin	23FEB	.	
5	Nuthatch	19APR	.	

Figure 10: the SAS log after submitting the DATA step from Figure 9, and the resulting dataset

We can deal with these missing delimiters by applying the same technique discussed above for handling unquoted delimiters. Our inspection follows the same basic steps:

1) Identify the closest mandatory field that is to the right of the sometimes-missing delimiter and has a limited number of expected values
2) Gather up all the possible values for this mandatory field, or define some rules about those values
3) Inject code into the DATA step to:
   a) Bring a record/line into memory
   b) Pull out the characters that SAS would try to assign to the mandatory field
   c) If this string has a suitable value, then do nothing

d) Otherwise, fix the record by inserting a delimiter

A quick look back at the text file from Figure 8 shows that every field besides "Mode" is always populated. We could use its immediate neighbour, "Date", but let's select the last field instead because the rules are easier to define:

1) Field #4, "Bird", is always populated and only has 2 unique values
2) We can define a rule for the expected data of field #4 as a single character that is either "Y" or "N"
3) The code we will inject between the INFILE and existing INPUT statements:
   a) "INPUT @", to bring the line into memory
   b) Retrieve the fourth "word" from the _INFILE_ variable, split on the tilde character
   c) Test if it is "Y" or "N"
   d) If it is neither, insert a delimiter after the first one

Figure 11 below shows the full DATA step that implements the process described above:

```
FILENAME MISSING 'C:\TEMP\osbbb.txt';
data fixed (drop=DLM1AT field4);
  infile MISSING dlm='~' DSD firstobs=2;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  * 3a) Bring in the record so we can perform the "bad value" test;
  input @;
  * 3b) Get fourth "word", then pull out the first 2 characters;
  length field4 $1;
  field4 = scan(_INFILE_, 4, '~', 'M');
  * 3c) Test if it is Y or N;
  if field4 not in ('Y', 'N') then do;
    * 3d) If not, insert a delimiter after the first delimiter;
    DLM1AT = FIND(_INFILE_, '~');
    _INFILE_ = SUBSTR(_INFILE_, 1, DLM1AT) ||
               '~' || SUBSTR(_INFILE_, DLM1AT + 1);
  end;
  input  Animal      Mode      Date:DATE9.  Bird;
run;
```

Figure 11: implementing a generalized inspection & modification for the sample data from Figure 8

MORE THAN ONE WAY TO SKIN A CAT

An alternate approach is to look at the structure of the line as opposed to the values contained in a specific field. We can define a set of rules or patterns that dictate how a line should look, then modify any lines that do not fit the patterns. Our rules for this text file from Figure 8 are as follows:

1) The line begins with a string that is up to 8 characters long, with no character being a tilde (field "Animal")
2) It is immediately followed by a tilde (the delimiter separating "Animal" from "Mode")
3) Since "Mode" is blank, the very next character is another tilde (the delimiter separating "Mode" from "Date")
4) The line continues with 9-character string, with no character being a tilde (field "Date")
5) It is immediately followed by a tilde (the delimiter separating "Date" from "Bird")
6) The line ends with a single character (field "Bird")

For simplicity's sake, let us assume that the text file only has problems with the "Mode" field, meaning we only need to test lines using the first three rules listed above. The key test is rule #3: what is the character following the first tilde? If it is not another tilde, then the "Mode" delimiter is missing and must be inserted. This sort of pattern testing lends itself well to regular expressions, which we can harness using the PRX functions (Perl Regular eXpression). Figure 12 below shows a DATA step that uses the PRXCHANGE function to perform the search & replace work:

```
FILENAME MISSING 'C:\TEMP\osbbb.txt';
data everythings_better_with_regex;
  infile MISSING dlm='~' DSD firstobs=2;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  * Bring in the record so we can check the line structure;
  input @;
  * This single function call performs both the test and fix;
  _INFILE_ = PRXCHANGE('s/^([^~]{1,8}~)([^~])/\1~\2/', 1, _INFILE_);
  input  Animal      Mode      Date:DATE9.  Bird;
run;
```

Figure 12: a variation that works by inspecting the line structure using patterns

AND THAT'S NOT ALL

The trick we have discussed is powerful, and not limited to fixing issues with delimited data. There are applications for fixed width data as well. Imagine you have an established process that reads numerous fields from a fixed width file, and one day the file creator decides to inject 100 spaces at the start of every record. You could fix the problem by adding 100 to every column pointer of the INPUT statement. Or you could load the line in using "INPUT @", apply the SUBSTR function to the _INFILE_ variable to remove those 100 spaces and then keep your original INPUT statement intact. If the INPUT statement is part of an included macro that is not within your control to modify, this might be the most effective way to deal with this file.

CONCLUSION

In this paper, we discussed the delimited file format and described the problems that can occur when a record has too many or too few delimiters. We introduced a variation of the NULL input statement, "INPUT @", which we can use to load a record into memory without attempting to parse it. We explained how the "_INFILE_" automatic variable can be inspected and modified to correct a record if and only if a change is required. And we

showed how these two elements can be injected into a DATA step to allow difficult delimited data to be ingested elegantly into a SAS dataset.

DISCLAIMERS

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of TD Bank.

REFERENCES

Mullin, Charley. 2011. Finding Your Way Through the Wilderness: Moving Data from Text Files to SAS® Data Files.
Available: http://support.sas.com/resources/papers/proceedings11/256-2011.pdf

Schreier, Howard. 2001. Now _INFILE_ is an Automatic Variable – So What?
Available: https://www.lexjansen.com/nesug/nesug01/cc/cc4018bw.pdf

Windham, K. Matthew. 2014. Introduction to Regular Expressions in SAS(R). Cary, NC: SAS Institute Inc.

SAS® 9.4 DATA Step Statements: Reference. INPUT Statement.
https://documentation.sas.com/?docsetId=lestmtsref&docsetTarget=n0oaql83drile0n141pdacojq97s.htm&docsetVersion=9.4

SAS® 9.4 DATA Step Statements: Reference. INFILE Statement.
https://documentation.sas.com/?docsetId=lestmtsref&docsetTarget=n1rill4udj0tfun1fvce3j401plo.htm&docsetVersion=9.4#p07t1i9htxlzidn10a5812h0a4l5

ACKNOWLEDGMENTS

The author would like to thank TD Bank management for supporting his participation in SAS Global Forum 2020.

Additionally, the author would like to thank Lionel Teed and Michael A. Raithel for their encouragement and support in writing this paper.

CONTACT INFORMATION

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

Michael Chu
michael.chu@td.com

APPENDICES

APPENDIX A - COMPLETE CODE FOR UNQUOTED DELIMITERS

```
FILENAME BADFILE 'C:\TEMP\mp_movie_data.txt';

/***************************************************************************
Creating the sample text file with an unquoted delimiter in the first field
***************************************************************************/
data _NULL_;
  file BADFILE;
```

```
  put "Name|Year|Rating on Rotten Tomatoes|Rank on IMDb";
  put "And Now for Something Completely Different|1971|90%|4";
  put "Monty Python and the Holy Grail|1975|97%|1";
  put "Monty Python's|Life of Brian|1979|95%|2";
  *                  ^-- injected delimiter;
  put "Monty Python Live at the Hollywood Bowl|1982|N/A|6";
  put "Monty Python's The Meaning of Life|1983|85%|3";
run;

/****************************************************************************
A DATA step that will fail to read record #3 correctly
****************************************************************************/
data iamerror;
  infile BADFILE dlm='|' firstobs=2;
  format Name $50. Year 4. Rating $3. Rank 1. ;
  input  Name      Year    Rating     Rank;
run;

/****************************************************************************
Apply our trick of using "INPUT @" & "_INFILE_" automatic variable
****************************************************************************/
data r4d4 (drop=DLM1at field2);
  infile BADFILE dlm='|' firstobs=2;
  format Name $50. Year 4. Rating $3. Rank 1. ;
  input @;
  DLM1at = find(_INFILE_, '|');
  length field2 $4;
  field2 = substr(_INFILE_, DLM1at + 1, 4);
  if lengthn(compress(field2, '1234567890')) ne 0 then do;
    _INFILE_ = substr(_INFILE_, 1, dlm1at - 1) || ' ' ||
               substr(_INFILE_, dlm1at + 1);
  end;
  input  Name      Year    Rating     Rank;
run;
```

APPENDIX B – COMPLETE CODE FOR MISSING DELIMITERS

```
FILENAME MISSING 'C:\TEMP\osbbb.txt';

/****************************************************************************
Creating the sample text file with a missing delimiter in a few records
****************************************************************************/
data _NULL_;
  file MISSING;
  put 'Animal~Mode~Date~Bird';
  put 'Beaver~~09OCT2019~N';
  put 'Gannet~22JUL2018~Y';
  *           ^-- missing delimiter;
  put 'Peacock~~17DEC2017~Y';
  put 'Robin~23FEB2019~Y';
  *          ^-- missing delimiter;
  put 'Nuthatch~19APR2019~Y';
  *             ^-- missing delimiter;
run;

/****************************************************************************
A DATA step that will fail to read records #2, 4 and 5 correctly
****************************************************************************/
```

```
data missing;
  infile MISSING dlm='~' DSD firstobs=2 truncover;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  input  Animal      Mode      Date:DATE9.  Bird;
run;

/*****************************************************************************
Apply our trick of using "INPUT @" & "_INFILE_" automatic variable
First using normal string functions, then using regular expressions
*****************************************************************************/
data fixed (drop=DLM1AT field4);
  infile MISSING dlm='~' DSD firstobs=2;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  input @;
  length field4 $1;
  field4 = scan(_INFILE_, 4, '~', 'M');
  if field4 not in ('Y', 'N') then do;
    DLM1AT = FIND(_INFILE_, '~');
     _INFILE_ = SUBSTR(_INFILE_, 1, DLM1AT) ||
                '~' || SUBSTR(_INFILE_, DLM1AT + 1);
   end;
  input  Animal      Mode      Date:DATE9.  Bird;
run;

data everythings_better_with_regex;
  infile MISSING dlm='~' DSD firstobs=2;
  format Animal $8.  Mode $5.  Date DATE9.  Bird $1.;
  input @;
  _INFILE_ = PRXCHANGE('s/^([^~]{1,8}~)([^~])/\1~\2/', 1, _INFILE_);
  input  Animal      Mode      Date:DATE9.  Bird;
run;
```

APPENDIX C – BENCHMARKING NOTES

Adding additional processing instructions into a DATA step comes with an increase in time and resources. Some simple benchmarking tests using large delimited files of over 8GB showed an 18% increase to the real time spent within the DATA step and an 11% increase to the memory consumption.