MWSUG 2018 - Paper SP-049

# Data-Driven Data Analysis

Jack N Shoemaker, Greensboro, NC

## ABSTRACT

When confronted with a new data channel, the modern data scientist or analyst will employ sophisticated data visualization tools like Visual Analytics to size up the data. Not all users have access to these tools and must rely on more pedestrian code-based approaches. This paper explores techniques using Base SAS to provide data-driven data analysis to help size up data absent the more modern tools. These techniques leverage the details about data available from the dictionary subsystem. Knowing the names, formats, and data types of the data allows one to derive great insight into the content of the data stream.

## INTRODUCTION

The general approach to an analytic task has three phases: size-up the data; beat the data into submission; and, finally, run the analysis. The overarching goal of the modern tools is to get as quickly as possible to the final phase by streamlining or eliminating the first two phases. Ideally, the data scientist or analyst need only point to the primary data to get her work going. That is, the modern tools are woke to the content of data which allows these tools to preset the analyst with the proper starting point without the need of specifying a lot of details about the data first. It is in this spirit that we will explore a handful of techniques that size-up the data without requiring any more than a pointer to the data in the form of a SAS data object like a dataset or table.

## SAMPLE DATA

For this exercise we will use a public data set, namely, drug (NDC) information published daily by the Food and Drug Administration. Assuming your SAS can reach the internet, the following code will grab the latest ZIP archive from FDA:

```
%let FDA = www.accessdata.fda.gov/cder;

proc http
  method = "get"
  url = "http://&FDA./ndctext.zip"
  out = filename
  ;
run;
```

Although a full discussion of grabbing data over the internet is beyond the scope of this paper, the full program that grabs and parses the files contained in the FDA ZIP archive may be found here:

https://github.com/jackshoemaker/FDA.git

## DATA-DISCOVERY EXAMPLES

We will use a data sets called 'FDA.PACKAGE' and 'FDA.PRODUCT' which are created by the SAS program available from the GitHub link above.

Here are four data-discovery displays based on this public data source.

### FIELD-LENGTH REPORT

The purpose of the field-length report is to show the maximum field widths based on the actual contents of the columns. This comes in handy in at least two scenarios. In the first scenario you may be confronted with a delimited file. You know the names of the columns, but don't know the exact format. As a result, on the first pass through the data you may just read everything as $CHAR200., or some other unreasonably large value. Once you have the data in a SAS data set, you can use the field-length report to turn the column sizes to match the actual contents of the fields. The second scenario is similar. You may be provided data in a DBMS table that has made no effort to size the columns appropriately. In other words, all of the columns are VARCHAR(200) or some other unreasonably large value. You can use the details of the field-length report to size the working SAS data sets appropriately. Output 1 shows the results of the field length report run against the FDA.PACKAGE data set.

```
The MEANS Procedure

Variable Label                             N      Minimum          Maximum
------------------------------------------------------------------------------
_len_1   PRODUCTNDC - [15]            233914            9               10
_len_2   NDCPACKAGECODE - [20]        233914           12               12
_len_3   NDC_EXCLUDE_FLAG - [1]       233914            1                1
_len_4   SAMPLE_PACKAGE - [1]         233914            1                1
_len_5   productIDs - [150]           233914           46               95
_len_6   PackageDescriptions - [1600] 233914           27              779
------------------------------------------------------------------------------
```

**Output 1. Output from field-length report run against FDA.PACKAGE.**

The results here are not as dramatic as they could be as the FDA.PACKAGE data set has already had some sizing performed on it. But, as a user you may wish to re-size, or perhaps discard, the sixteen-hundred-byte column listed as ‘PackageDescriptions’. A fuller explanation on how this display is generated follows below.

## MISSING VALUE REPORT

The purpose of the missing-value report is to provide a sense on how complete the data are and to get a sense of cardinality or number of levels in each column. This report can help you know what to expect from your programs that use these data. For example, although you may be excited to use the column called DEASCHEDULE, you should not be surprised that the rule will only fire about 5% of the time. That may be fine. If, on the other hand, you were expecting all rows to have a DEASCHEDULE value, you will be disappointed with the results. The report can also confirm the uniqueness of a column. For example, we know from the report below that PRODUCTID is unique because there are 121,608 different values for 121,608 rows of data. Output 2 shows the results of the field length report run against the FDA.PRODUCT data set.

```
COLUMN NAME                    TYPE N_MISSING %_MISSING N_PRESENT %_PRESENT  Levels
APPLICATIONNUMBER              CHAR    17,038  14.0106    104,570   85.989   10,629
DEASCHEDULE                    CHAR   116,700  95.9641      4,908    4.036        4
DOSAGEFORMNAME                 CHAR         0   0.0000    121,608  100.000      135
ENDMARKETINGDATE               NUM    119,022  97.8735      2,586    2.127      671
LABELERNAME                    CHAR         0   0.0000    121,608  100.000    8,813
LISTING_RECORD_CERTIFIED_THRU  NUM      2,588   2.1281    119,020   97.872        3
MARKETINGCATEGORYNAME          CHAR         0   0.0000    121,608  100.000       11
NDC_EXCLUDE_FLAG               CHAR         0   0.0000    121,608  100.000        2
NONPROPRIETARYNAME             CHAR         0   0.0000    121,608  100.000   21,444
PRODUCTID                      CHAR         0   0.0000    121,608  100.000  121,608
PRODUCTNDC                     CHAR         0   0.0000    121,608  100.000  119,960
PRODUCTTYPENAME                CHAR         0   0.0000    121,608  100.000        7
PROPRIETARYNAME                CHAR         0   0.0000    121,608  100.000   44,965
PROPRIETARYNAMESUFFIX          CHAR   104,710  86.1045     16,898   13.895    7,335
ROUTENAME                      CHAR     2,027   1.6668    119,581   98.333      202
STARTMARKETINGDATE             NUM          5   0.0041    121,603   99.996    7,833
```

**Output 2. Output from missing-value report run against FDA.PRODUCT.**

The SAS program that produces this display borrows heavily from original work by Mike Zdeb. You can find a fuller description of Mike’s missing-value routines here:

https://www.lexjansen.com/nesug/nesug11/ds/ds12.pdf

## SEVEN-NUMBER SUMMARY

The seven-number summary is a data analysis technique first proposed by John Tukey. In professor Tukey's construction, the seven numbers are the count, the mean, the minimum and maximum values, the median value, and the two quartile values. This analysis applies only to numeric columns. The intent is to provide a sense of the shape of the distribution of the data values in a simple tabular form. Output 3 shows the seven-number summary for the numeric columns in our sample data set. Note that since the meta data for this sample data set tells us that these numeric columns are dates, we display the values as dates.

```
              ..... Parametric ....  ............ Rank Statistics ................
Column Name      N Obs     Average     Minimum      Median    75th Pct     Maximum
ENDMARKETIN      2,586  2020-02-03  2018-06-19  2019-04-30  2020-06-30  3031-02-09
LISTING_REC    119,020  2018-12-26  2017-12-31  2018-12-31  2018-12-31  2019-12-31
STARTMARKET    121,603  2008-10-09  1900-01-01  2012-08-01  2015-09-01  2018-06-18
```

**Output 3. Seven-number summary for FDA.PRODUCT.**

Readers that are of the Python world may notice the similarity between the seven-number summary above and that which is available from the describe method. For example, using the open-source SASpy package, we can load a panda called 'fda' from our sample data set and execute the describe method as follows:

```
import saspy
sas = saspy.SASsession(cfgname='winlocal')
fda = sas.sasdata("PRODUCT","FDA")
fda.describe()
```

Output 4 shows the results of running the chunk of Python code above. Note that some of the display has been truncated to fit into the box below.

```
Variable                      N       Median  Mean   Min     P25    P50    P75    Max
STARTMARKETINGDATE            121603  19206   17814  -21914  17561  19206  20332  21353
ENDMARKETINGDATE              2586    21669   21948   21354  21488  21669  22096  391214
LISTING_RECORD_CERTIFIED_THRU 119020  21549   21544   21184  21549  21549  21549  21914
```

**Output 4. Results of the describe method executed on FDA.PRODUCT.**

Of course, the Python describe method knows nothing of date formats, so the results show the internal numeric values instead of the formatted values. Using either technique we see a suspect maximum value for ENDMARKETINGDATE and perhaps a suspect minimum value for STARTMARKETINGDATE.

## FIELD-CONTENT VERIFICATION

The seven-number summary does a good job for numeric columns but will not work for character data. To size up the character data we will employ a technique known as field-content verification (FCV) which shows the contents of a column in descending frequency order of occurrence. In other words, you get a sense of the most frequent values in a column as well as structure or format of that columns. For example, the FCV display will show you whether a column contains all capitalized values or whether a column has formatting punctuation. The FCV display also will show you unexpected values, that is, noise that is on the data channel and not mentioned in any sort of documentation. Output 5 shows the result of applying the FCV routine to FDA.PRODUCT. Note that the FCV routine produces a separate display for each character column in the data set. We show an example from just one column below.

```
Field Under Analysis: PRODUCTTYPENAME [SPL Document Type]
Total rows in [FDA.PRODUCT]:      121,608
                                                 Cumulative
Field Contents              Occurs   Percent         Rows   Percent
HUMAN OTC DRUG              63,101     51.9%       63,101     51.9%
HUMAN PRESCRIPTION DRUG     56,066     46.1%      119,167     98.0%
NON-STANDARDIZED ALLERGENIC  1,843      1.5%      121,010     99.5%
PLASMA DERIVATIVE              337      0.3%      121,347     99.8%
STANDARDIZED ALLERGENIC        140      0.1%      121,487     99.9%
VACCINE                        113      0.1%      121,600    100.0%
CELLULAR THERAPY                 8      0.0%      121,608    100.0%
```

**Output 5. FCV display for column PRODUCTTYPENAME in FDA.PRODUCT.**

Note that the SAS routine that produces the display above picks up the LABEL attribute from the SAS data set and displays this in square brackets next to the column name. This display tells us that 98% of the drugs in the FDA.PRODUCT data set are 'HUMAN' drugs split almost evenly between OTC and PRESCRIPTION.

## HOW IT'S DONE

The examples above use data sets with a small number of columns. That is, it would not be too terrible a burden to run the PROC MEANS and PROC FREQ procedures 'by hand' by supplying explicit lists of column names to these procedures. But in the spirit of data-driven data discovery, we want to specify just the data-set name and then allow SAS to use the various meta data and column-name short cuts to dynamically generate the SAS statements which will produce the displays above. Not surprisingly, that code is wrapped in a set of SAS macros which we will explicate below. But first, here is the SAS code used to produce the displays above.

```
%FLR(DSN=FDA.PACKAGE);
%miss_report(DSN=FDA.PRODUCT);
%NumerFCV(DSN=FDA.PRODUCT);
%AlphaFCV(DSN=FDA.PRODUCT);
```

That the calling signature of the four SAS macros listed above is the same is by design. In fact, in real-life use, these SAS macros may themselves be placed in a higher-level wrapper script so that a SAS macro call like this will execute all four routines for both data sets.

```
%Inspect(LIB=FDA,_TABLES=PACKAGE PRODUCT);
```

The details of the SAS 'Inspect' wrapper are beyond the scope of this paper but should not be too hard for the reader to imagine how to implement.

The overarching strategy for these SAS macros is as follows:

- Require minimal specification
- Create display for given object
- Get list of objects from meta data
- Use macro to loop over objects

### %FLR

To create the field-length report we need to create a new set of data-step variables that contain the length of the trimmed contents of each column. That is,

```
column_width = length( trim( column_name ) );
```

Then we will use PROC MEANS to show us the minimum and maximum values for the computed column_width data-step variable. We want to do this for every character column in the data set. Furthermore, since the meta data for the SAS data set will tell us the current size of the column, we will display that as well. We can retrieve the NAME and LENGTH attributes for the columns in the SAS data set under consideration from the dictionary table called DICTIONARY.COLUMNS which is available in the PRCO SQL subsystem. The following PROC SQL statement will store the column names and lengths as space-delimited strings into SAS macro symbols &V and &L respectively. We will also capture the number of character columns into the SAS macro symbol &N.

```
proc sql noprint;
    select name into :V separated by ' ' from dictionary.columns
        where libname = "FDA" & memname = "PACKAGE" & type = 'CHAR';
    select length into :L separated by ' ' from dictionary.columns
        where libname = "FDA" & memname = "PACKAGE" & type = 'CHAR';
    select left( put( count( * ), 5. ) ) into :N from dictionary.columns
        where libname = "FDA" & memname = "PACKAGE" & type = 'CHAR';
quit;
```

We will store the column width into a new set of data-step variables in the form '_len_*NNN*', where *NNN* is a sequential number starting at 1 and ending with the value of &N. as captured in the PROC SQL statement above. Note that the naming convention, '_len_*NNN*' is completely arbitrary and, of course, will not work if your data set happens to have existing character columns named '_len_*NNN*'. Given that we have created a new set of data step variables to capture the column widths, we will use the label attribute to store the original column name and length. Since PROC MEANS displays column labels by default, this will provide the cross referencing we need. Here is the data step that will create a new data set that contains the column widths for all character columns in the data set under analysis.

Now that we have the list of character columns and their corresponding lengths, we can write a simple data step using classic array processing to compute the column width for each character column in the data set under analysis. Here is that data step:

```
data inspect( keep = _len_: );
    set FDA.PRODUCT;
    array v{*} &V.;
    array l{*} _len_1 - _len_&N.;
    label %LenLab;
    do i = 1 to dim( v );
        l{i} = length( trim( v{i} ) );
        end;
run;
```

Now, given the data set inspect, we produce the field-length report using a simple call to PROC MEANS as follows:

```
title1 "Character Field Length Report for [FDA.PRODUCT]";
proc means data = inspect n min max maxdec=0;
    var _len_:;
run;
```

The inspect data set uses a SAS macro called %LenLab to generate label statements in the form:

```
_len_1 = 'Original Column Name [column width]'
```

Discussion of the macro technique used to generate a complete set of label statements is beyond the scope of this paper; however, you can find the definition of the %LenLab macro in the FLR.sas file in the GitHub repository: https://github.com/jackshoemaker/FCV.git

## %MISS_REPORT

The work horse of Zdeb's missing-value routines is PROC FREQ. We will use the following variable-name shorthands to reference the columns in the data set under analysis. Those character shortcuts are:

- _ALL_ is all columns in data set
- _NUMERIC_ is all numeric columns
- _CHARACTER_ is all character columns

To get counts of missing and non-missing values, we want to run one-way frequencies for all the columns in the data set. For example:

```
proc freq data = FDA.PRODUCT;
table _all_ / missing;
run;
```

The use of the special shorthand _ALL_ allows us to specify all the column names in FDA.PRODUCT without having to list all the columns explicitly. Although the use of the MISSING option on the TABLE statement will produce a count of missing values, the code above will also produce counts for each distinct non-missing value. For the missing-value report, we wish to collapse the counts to just two values: missing and non-missing. We can achieve this by using a user-defined format and then associating that user-defined format with the columns in the data set under analysis. Since user-defined formats are specific to column type, we will need two user-defined formats – one for character columns and one for numeric columns. Here is the definition of user-defined formats:

```
proc format;
    value nm    . = '0' other = '1';
    value $ch ' ' = '0' other = '1’;
run;
```

When applied to numeric columns, the NM format will code missing values as ‘0’ and everything else as ‘1’. Similarly, the $CH format will code missing values as ‘0’ and everything else as ‘1’. With these user-defined formats in place, we can modify the original PROC FREQ code to apply these user-defined formats to the numeric and character columns in the data set under analysis as follows:

```
proc freq data = FDA.PRODUCT;
table _all_ / missing;
format _numeric_ nm. _character_ $ch.;
run;
```

The code shown above will create a one-way frequency display for each column in the data set under analysis. That output will be sent to whatever the active output destination is at the time of execution. To create our report, we need to direct the output to a data set, so we can use reporting procedures like PROC REPORT or PROC PRINT to display the results. To accomplish this feat, we will use ODS to direct the PROC FREQ output to a data set as follows:

```
ods listing close;
ods output onewayfreqs = tables( keep = table f_: frequency percent );

proc freq data = FDA.PRODUCT;
table _all_ / missing;
format _numeric_ nm. _character_ $ch.;
run;

ods output close;
ods listing;
```

Since we don’t care about the normal PROC FREQ output, we first turn off, or close, the default LISTING destination with the first ODS statement. The second ODS statement directs SAS to place the one-way frequency results into a data set called tables. With the ODS destination now set, we run PROC FREQ to compute and capture the counts of missing and non-missing values. When done executing PROC FREQ, we then close the OUTPUT destination and turn back on the LISTING destination.

The tables data set created by the ODS capture above is not quite in condition for display and reporting. Transforming the PROC FREQ output into a data set suitable for printing requires some data-step programming which the thrust of this paper is not and is best described in Zdeb’s original paper as referenced earlier. You can find the full source for the %MISS_REPORT macro in the GitHub repository: https://github.com/jackshoemaker/FCV.git

## %NUMERFCV

The %NumerFCV creates the seven-number summary display. The heavy work is done by PROC UNIVARIATE because when this routine was originally created, that was the only SAS procedure that provided rank statistics as output. In other words, the Q1, MEDIAN, and Q3 rank statistics were not available in PROC MEANS as they are today. One can achieve results similar to the Python describe method directly from PROC MEANS by employing the _NUMERIC_ shorthand as follows:

```
proc means data = FDA.PRODUCT
    n mean min q1 median q3 max;
var _numeric_;
run;
```

The main advantage of using the %NumerFCV macro instead is that date and date-time columns are displayed using a date format which aids in readability assuming you don’t convert SAS serial dates to calendar notation in your head. In keeping with the theme of this paper, we’ll use the dictionary tables to establish column type based on the format attribute of the column and then take appropriate action based on the column type. The format attribute is available in the DICTIONARY.COLUMNS table that is part of the PROC SQL subsystem. This PROC SQL statement will create a data set called NUMER that contains the name, label, and format attributes for all the numeric columns in the data set under analysis:

```
      proc sql;
          create table numer as
              select name, label, format
              from dictionary.columns
              where upcase( type ) = 'NUM' &
              upcase( libname ) = 'FDA' &
              upcase( memname ) = 'PRODUCT'
              ;
      quit;
```

The next step is to load these attributes into macro variable arrays just as we did with the field-length routine. In the %FLR macro we did this in the PROC SQL statement directly and did not create an intermediate data set. One advantage of using the data-step approach is that you can test for existence and respond accordingly. For example, if the data set under analysis has no numeric columns, then you don't have anything to do. Accordingly, the macro should stop processing statements at this point. You can accomplish this by examining the &SYSNOBS automatic macro symbol and forcing a return if the value is 0:

```
  %if &SYSNOBS. = 0 %then %return;
```

If there are numeric columns in the data set under analysis, we load the attributes into macro variable arrays using the following data step:

```
  data _null_;
      set numer end = lastrec;
      call symputx( catt( '_V', _n_ ), trim( name ) );
      if not( missing( label ) )
          then call symputx( catt( '_L', _n_ ), trim( label ) );
      else call symputx( catt( '_L', _n_ ), trim( name ) );
      call symputx( catt( '_F', _n_ ), trim( format ) );
      if lastrec then call symputx( '_N', left( put( _n_, 5. ) ) );
      run;
```

After this data step executes, we have a set of macro symbols in the form &_V1, &_V2, …; &_L1, &_L2, …; &_F1, &_F2, …; and &_N which holds the count of the number of numeric columns in the data set under analysis. Now we run PROC UNIVARIATE for each numeric column inside a %DO loop as follows:

```
  %do i = 1 %to &_N;
      proc univariate data = FDA.PRODUCT noprint;
          var &&_V&i;
          output out = stats
              n = nobs sum = tot mean = avg
              min = min q1 = q1 median = median q3 = q3 max = max;
          run;

      data stats;
          length name $ 32 label $ 80 format $ 32;
          set stats;
          name = "&&_V&i";
          label = "&&_L&i";
          format = "&&_F&i";
          run;

      proc append base = report data = stats;
      run;
   %end;
```

Each call to PROC UNIVARIATE creates a data set call STATS. The next data step adds the columns name, label, and format to the STATS data set. As a last step in the macro loop, the STATS data set for a single column is appended to master REPORTS data set which will create the seven-number summary displays. In order to display dates and date-time values using date formats, we need to split the master REPORTS data set into three data sets based on the format attribute as follows:

```
data report datevar dtvar;
    set report;
    if prxmatch( '/TIME|MDY/', format ) > 0 then output dtvar;
    else if prxmatch( '/DATE|YY|QQ|MON/', format ) > 0 then output datevar;
    else output report;
    run;
```

Note than any one or two of these data sets may have 0 observations. For example, the DTVAR data set will not have any observations if there aren't any date-time columns in the data set under analysis. More precisely, the DTVAR data set will have 0 observations if none of the formats match this regular expression: /TIME/MDY/. That regular expression does not capture all possible date-time format designations; however, as a practical matter, this simple regex has proven resilient. Similarly, the regular expression used to find date columns is not comprehensive but has been shown to cover most scenarios.

At this point, all that is left to do is display the results using PROC REPORT as follows:

```
proc report data = report missing;
    columns name
        ( '. Parametric Statistics .' nobs tot avg )
        ( '. Rank Statistics .' min q1 median q3 max );
    define name / order format=$32. 'Column Name';
    define nobs / display format=comma11. 'N Obs';
    define tot / display format=&NUMFMT. 'Total';
    define avg / display format=&NUMFMT. 'Average';
    define min / display format=&NUMFMT. 'Minimum';
    define q1 / display format=&NUMFMT. '25th Pct';
    define median / display format=&NUMFMT. 'Median';
    define q3 / display format=&NUMFMT. '75th Pct';
    define max / display format=&NUMFMT. 'Maximum';
    title2 "Numeric Field Content Analysis:  FDA.PACKAGE";
    run;
```

You will find the entire %NumerFCV macro in this GitHub repository:

https://github.com/jackshoemaker/FCV.git

## %ALPHAFCV

The %AlphaFCV creates the descending frequency report for character columns. Although PROC MEANS provides a mechanism to do a quick-and-dirty seven-number summary directly without the need for any macro gymnastics, there is no parallel mechanism to produce the FCV displays. We start by obtaining a list of the character column names from DICTIONARY.COLUMNS and loading the name and label attributes into macro variable arrays as follows:

```
proc sql;
    create table alpha as
    select name, label
    from dictionary.columns
    where upcase( type ) = 'CHAR' &
        upcase( libname ) = 'FDA' &
        upcase( memname ) = 'PRODUCT'
        ;
    quit;

%if &SYSNOBS. = 0 %then %return;
```

```
data _null_;
    set alpha end = lastrec;
    call symputx( catt( '_V', _n_ ), trim( name ) );
    if not( missing( label ) | ( trim( label ) = trim( name ) ) ) then
        call symputx( catt( '_L', _n_ ), catt( name, ' [', label, ']' ) );
    else call symputx( catt( '_L', _n_ ), trim( name ) );
    if lastrec then call symputx( '_N', left( put( _n_, 5. ) ) );
    run;
```

For each character column we want to create a data set with two columns: VALUE which will contain the unique values in the column under inspection and ROWS which will be a row count (or frequency) of the occurrence of those values. We use a PROC SQL statement to do this as follows:

```
proc sql;
    create table peek as
    select &&_V&i as value, count( * ) as rows
    from FDA.PRODUCT
    group by &&_V&i
    ;
    quit;
```

The next step is to collate the PEEK data set in descending order of occurrence and discard all but the first 40 rows. The value of 40 is arbitrary and set as a macro parameter on %AlphaFCV allowing the user to change this value as needed. The goal of the FCV routine is to get a quick (one-page) peek at the contents of a character column. Expanding on our example from about, if PRODUCTID is a character variable and unique, then the PEEK data set created by the PROC SQL statement above will yield a data set with all the observations in FDA.PRODUCT. Displaying that entire data set would have little value. If you look at just the top forty values and all of those have an occurrence of 1, you have all you are going to get from this routine, namely, that the column is unique and looks like the sample of 40 values shows you. To achieve these results, we sort the PEEK data set in descending order and send the result through a simple data step to cull all but the top 40 values and compute percentages and cumulative values as follows:

```
proc sort data = peek;
    by descending rows value;
    run;

data peek;
    set peek;
    pct = rows / &DENOM.;
    cumpct + pct;
    cumrows + rows;
    if _n_ > &TOP. then stop;
    run;
```

All that is left to do at this point is to display the PEEK data set using PROC REPORT as follows:

```
proc report data = peek missing;
    columns value rows pct ( 'Cumulative' cumrows cumpct );
    define value / format=&COLFMT. 'Field Contents';
    define rows / format=&NUMFMT. 'Occurs';
    define pct / format=&PCTFMT. 'Percent';
    define cumrows / format=&NUMFMT. 'Rows';
    define cumpct / format=&PCTFMT. 'Percent';
    title2 "Field Under Analysis: &&_L&i.";
    run;
```

The steps above are repeated for each character column inside a %DO macro loop using the macro variable arrays. You will find the entire %AlphaFCV macro in this GitHub repository:

https://github.com/jackshoemaker/FCV.git

## CONCLUSION

Over the past three score decades, information-processing technology has become more powerful and ubiquitous. This has changed not only how quickly we can turn around data-analysis tasks but also how we think about approaching those tasks in the first place. I doubt an Economics graduate student in the Fifties was giving too much thought to a heat map showing GDP per person by county across the United States. What has remained constant for the data scientist over time is the mantra, “Know thy data.” This paper explored four very basic techniques to help a user size up and understand the contents of a given data set or DBMS table. To make these techniques agnostic and comprehensive in terms of variable scope, the routines are implemented as SAS macros that first interrogate the meta data to generate variable lists and then employ macro looping techniques to transcend those lists. This paper focused on the former of those two parts of the macro routines, namely, the interrogation and use of the available meta data. A reader interested in seeing the full macro implementation may find those macros in the GitHub repository referenced in the contact information below.

Strikingly, the main work horses for these routines are SAS procedures that have been part of SAS since inception, or close to it: FREQ, MEANS, UNIVARIATE, and of course, DATA – the data step being a SAS procedure in a fashion. That is testament to the long-term and long-lasting utility of those old procedures that have always been there to help a data scientist know her data.

## CONTACT INFORMATION

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

Jack N Shoemaker
Greensboro, NC
Jack.Shoemaker@gmail.com

Full macro source code for the topics discussed in this paper may be found in the GitHub repository: https://github.com/jackshoemaker/FCV.git