PAPER CC-06

SAS® Formats: Effective and Efficient

Harry Droogendyk,
Stratia Consulting Inc., Lynden, ON

ABSTRACT

SAS® formats, whether they be the vanilla variety supplied with the SAS system, or fancy ones you create yourself, will increase your coding and program efficiency. (In)Formats can be used effectively for data conversion, data presentation and data summarization, resulting in efficient, data-driven code that's less work to maintain. Creation and use of user-defined formats, including picture formats, are included in this paper.

INTRODUCTION

If you've been in the programming business for any length of time, you're well aware that your data is not necessarily stored as you enter it or as it is displayed when presented for your viewing. For example, open up an editor on your ASCII machine and type the word SESUG in the first record and save the file. If we display the file in hexadecimal representation, we do not see 'SESUG', rather, the five bytes appear as '53 45 53 55 47'. However, the hexadecimal representation is really only a more readable form of the binary format a computer ultimately stores and uses. How does an ASCII machine store the word SESUG? Each of the five bytes contains eight bits, a total of 40 bits, 0101001101000101010100110101010101000111 . That's just plain ugly!

Without getting too technical, matters become more complicated when we consider numbers, especially those stored in a SAS dataset. To enable SAS to store numbers of large magnitude and perform calculations that require many digits of precision to the right of the decimal point, SAS stores all numbers using **floating-point** representation. You may know this format as *scientific notation*. Values are represented as numbers between 0 and 1 times a power of 10. e.g. the number 1 in scientific notation is $.1 \times 10^1$

We can look at this and do the math and come up with an answer of 1. But, just to confuse the matter even further, SAS stores numbers in scientific notation, not in base 10 (like we humans count), but in base 2. In hexadecimal format the numeral 1 appears as 3F F0 00 00 00 00 00 00. Certainly doesn't look like 1 to me! Ahhh, aren't you glad you don't have to work with the internals?

It's evident that *something* happens to our data after we enter it, *something else* before it's displayed to us. The stuff we can easily read is modified by SAS to a format it can understand, massage and store. In order to make the data meaningful to us, SAS must re-format the data for our reading pleasure when pumping it back out. Thankfully we're not limited to SAS's idea of how things ought to be - we can use informats and formats to direct SAS *how* to input and display our data.

This paper will introduce you to SAS-supplied *and user-defined* formats and informats and how to use them to effectively and efficiently deal with data conversion and presentation tasks.

Let's begin with some definitions and examples. An **informat** is an instruction used to read data values into a SAS variable. In addition, if a variable has not yet been defined, SAS uses the informat to determine whether the variable ought to be numeric or character. The informat is also used to determine the length of character variables. A **format** is defined as an instruction that SAS uses to write data values. Formats are used to control the written appearance of data values. Both informats and formats are of the form: <**$**> **name** <**w**> **.** <**d**>

§	$	required prefix for *character* (in)formats
§	format	up to 7 characters long for character, 8 for numeric, may not **end** in a number! v9 format names may be 31 / 32 characters long
§	w	width of value to be read / written, includes commas, decimal places, dollar signs etc…
§	**.**	a period, *every* format **must have** a period, distinguishes formats from variable names
§	d	decimal places, optional, *only for numeric (in)formats*

For example:	$char20**.**	- 20 byte character format
	dollar12**.**2	- 12 byte numeric format, with 2 decimal places

HAVE YOU EVER USED FORMATS OR INFORMATS?

"Are you kidding?!", you say, "I haven't been using SAS that long, formats are *confusing*!". Well, I can say with certainty you have used formats, even if SAS was doing it behind the scenes on your behalf. Consider the following simple data step and PRINT procedure:

```
data weather;
  input date city $ degrees_celsius
;
cards;
20100117 Edmonton  -134.3456
20100204 Toronto     -2.5
20100328 Calgary      7.1
20100413 Ottawa      12.64
20100510 Lynden      17.2
run;
```

```
proc print data=weather;
run;
```

Obs	date	city	degrees_ celsius
1	20100117	Edmonton	-134.346
2	20100204	Toronto	-2.500
3	20100328	Calgary	7.100
4	20100413	Ottawa	12.640
5	20100510	Lynden	17.200

1

Since neither the data step nor PRINT procedure specified input or output formats, SAS has used default formats. How did this default behavior affect what SAS did with the weather data?

SAS read the input data and converted them to its internal format. As we discovered in the introduction, the internal representation will not look anything like the characters in our data. Rather, the value will be stored internally in a format that allows SAS to easily store and manipulate data efficiently. The PRINT procedure takes the internal values and produces output using default format specifications. It seems as though SAS did a pretty good job - the default formatted output generated by the PRINT procedure looks pretty good, with one exception. We lost the fourth decimal place in Edmonton's January temperature. PRINT defaulted to the **best.** format and decided that all we needed was three decimal places. Maybe defaults aren't good enough.

The use of default informats is only possible if the data is what SAS calls " standard numeric or character format." In other words, while numeric data may contain negative signs and decimal points, it may not contain commas or currency signs. Character data cannot contain embedded blanks. There's going to be a time when you must input and report non-standard data. What happens if we allow SAS to default in those cases?

Note the data error in the example below when the default behavior couldn't deal with the special characters in **accum_parking_revenue**. From the output created by the PUT statement, we can see that two variables were created, but **accum_parking_revenue** contains a missing value.

```
data parking;
  input city $ accum_parking_revenue ;
  put city= accum_parking_revenue=;
cards;
Edmonton $145,234.72
run;
```

Log Output:

```
NOTE: Invalid data for accum_parking_revenue in line 11 10-20.
city=Edmonton accum_parking_revenue=.
RULE:      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8--
11         Edmonton $145,234.72
city=Edmonton accum_parking_revenue=. _ERROR_=1 _N_=1
NOTE: The data set WORK.PARKING has 1 observations and 2 variables.
```

SPECIFYING (IN)FORMATS

INFORMATS

Why couldn't we simply read the non-standard numeric **accum_parking_revenue** data as character data? Doing so will preserve the currency signs and the commas so it'll still look purty on the way out. But… we cannot use character data in calculations! If we even *suspect* that we'll *ever* need the data for *any* type of calculation (the correct answer is YES!!!) , informats must be explicitly specified to properly INPUT the values.

```
data parking;
  input city $ accum_parking_revenue dollar12.2 ;
cards;
Edmonton $145,234.72
Ottawa $221,691.00
Toronto $275,876.54
Lynden $397,112.23
Hamilton $226,432.02
run;
```

As a bonus, consider the space savings in treating **accum_parking_revenue** as a number rather than a character: 8 numeric bytes vs. at least 11 bytes in character mode.

```
##    Variable                Type    Len    Pos
ffffffffffffffffffffffffffffffffffffffffffffffff
2    accum_parking_revenue    Num       8      0
1    city                     Char      8      8
```

While the following PRINT output displays the same *values* as the input data, the **accum_parking_revenue** values are not formatted as we'd expect for currency data. The informats specified on the INPUT statement ensured SAS read the data correctly, but nothing we've done so far has resulted in anything other than default output formats.

```
proc print data=parking;
run;
```

Obs	city	accum_parking_revenue
1	Edmonton	145234.72
2	Ottawa	221691.00
3	Toronto	275876.54
4	Lynden	397112.23
5	Hamilton	226432.02

FORMATS

Just as it's possible to explicitly specify informats when reading raw data, *temporary or permanent* formats can also be defined to the columns for output. Utilizing the FORMAT statement in the PRINT procedure is an example of a temporary format assignment.

```
proc print data=parking;
  format accum_parking_revenue dollar12.2; /*  Temporary format  */
run;
```

Obs	city	accum_parking_revenue
1	Edmonton	$145,234.72
2	Ottawa	$221,691.00
3	Toronto	$275,876.54
4	Lynden	$397,112.23
5	Hamilton	$226,432.02

PERMANENT (IN)FORMATS

Temporary formats are only in force for the life of the step in which they are defined. For persistent output format definitions, the format must be stored by SAS in the descriptor portion of the data set. Most often, this is done at dataset creation time via the (IN)FORMAT or ATTRIB statements. When informats are defined in this manner, it is not necessary to specify them again on the INPUT statement.

```
data parking;
  attrib   date label = 'Accum Date';
  informat accum_parking_revenue dollar12.2;
  format   accum_parking_revenue dollar12.2;
  label    accum_parking_revenue = 'Accum. Parking Revenue';
  input date city : $12.  accum_parking_revenue ;
cards;
20100117 Edmonton   $145,234.72
20100204 Edmonton   $221,691.00
20100328 Edmonton   $375,876.54
20100413 Edmonton   $597,112.23
20100510 Edmonton   $726,432.02
run;
```

Notice the *much* more interesting CONTENTS listing, now showing permanent formats, informats and labels for two fields:

```
#   Variable                 Type   Len  Pos  Format       Informat     Label
ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ
1   date                     Num      8    0                            Accum Date
2   accum_parking_revenue    Num      8    8  DOLLAR12.2   DOLLAR12.2   Accum. Parking Revenue
3   city                     Char    12   16
```

Since permanent output formats have been defined for the **accum_parking_revenue** column, there' s no need to use the (temporary) FORMAT statement in the PRINT procedure:

```
proc print data=parking label;
run;
```

```
                          Accum.
           Accum          Parking
Obs        Date           Revenue        city

 1       20100117     $145,234.72    Edmonton
 2       20100204     $221,691.00    Edmonton
 3       20100328     $375,876.54    Edmonton
 4       20100413     $597,112.23    Edmonton
 5       20100510     $726,432.02    Edmonton
```

SAS DATE / TIME VALUES

We've talked about converting data values, particularly numeric items, into SAS's internal format, a format more suited for storage and computation. There's one more very important class of values that must be highlighted: those relating to date and time.

If we were to take the data from the previous example, and calculate the average parking revenue per day, based on the **date** field values, we'd first have to calculate how many days had elapsed between observations. The year, month and day values could be parsed out of the date value and the appropriate arithmetic gymnastics performed to subtract the chunks, paying special attention to number of days / month, year rollovers, leap years etc… Of course there's a better way or it wouldn't be mentioned in a tutorial paper!

SAS has the ability to store dates *in a numeric field* as the number of elapsed days since January 1, 1960. In other words, Jan 2, 1960 has a SAS date value of 1, Dec 31, 1960 is 365, Sept 12, 2005 is 16,691. If the appropriate informats are used on INPUT, SAS will convert our readable date values to a SAS date value. Once our dates are in SAS date format, the number of days between observations is a simple subtraction between the two date values. In addition, it allows us to use a plethora of date functions and output formats to effectively process and present date values.

In an analogous fashion, SAS time formats convert times to the number of seconds since midnight. 1:00 am. is stored as 3600, 1:30 as 5400, 2:00 as 7200 etc… Again, making time calculations much simpler.

Note the **date** informat and output format specifications:

```
data parking;
  attrib   date informat = yymmdd10. format = mmddyyd10. label = 'Accum Date';
  informat accum_parking_revenue     dollar12.2;
  format   accum_parking_revenue
           avg_daily_parking_revenue dollar12.2;
  label    accum_parking_revenue     = 'Accum. Parking Revenue';
  label    avg_daily_parking_revenue = 'Avg. Daily Parking Revenue';
  input date city : $12. accum_parking_revenue ;

 avg_daily_parking_revenue =
     ( accum_parking_revenue - lag(accum_parking_revenue)) /
     ( date - lag(date)) ;

cards;
20100117 Edmonton $145,234.72
20100204 Edmonton $221,691.00
20100328 Edmonton $375,876.54
20100413 Edmonton $597,112.23
20100510 Edmonton $726,432.02
run;
```

Obs	Accum Date	Accum. Parking Revenue	Avg. Daily Parking Revenue	city
1	01-17-2010	$145,234.72	.	Edmonton
2	02-04-2010	$221,691.00	$4,247.57	Edmonton
3	03-28-2010	$375,876.54	$2,909.16	Edmonton
4	04-13-2010	$597,112.23	$13,827.23	Edmonton
5	05-10-2010	$726,432.02	$4,789.62	Edmonton

ADDITIONAL WAYS TO SPECIFY (IN)FORMATS

Thus far, we've seen a couple methods of applying input and output formats to our data:

- § INPUT / PUT statements — temporary
- § FORMAT (in data step), INFORMAT, ATTRIB statement — permanent

There are occasions when the data will *already* be in a SAS dataset, in a format not suited for our purposes. Perhaps a date field has been read and stored in its external format, e.g. 20050912, or numeric data must be presented in a different format than the permanent format defines. In those cases, it's often necessary to format the data "on the fly" or create additional variables in the format we require using INPUT and PUT *functions*.

Consider an observation containing the data below.

```
data bad_data;
   date        = '20100912';
   amt         = '$123,456.78';
   time        = '08:34';
   postal_code = 'l0r 1t0';
run;
```

Obs	date	amt	time	postal_ code
1	20100912	$123,456.78	08:34	l0r 1t0

To convert the existing SAS data, we can use the INPUT *function* to manipulate the data. The PUT *statement* displays both the new informatted values and the new formatted values, specifying formats different than the original data.

```
data good_data ( keep = new: );
  set bad_data;
```

```
  new_date = input(date,yymmdd8.);
  new_amt  = input(amt,dollar14.2);
  new_time = input(time,time5.);
  new_pc   = input(postal_code,$upcase.);

  put new_date  @30 new_date yymmdds10.  /
      new_amt   @30 new_amt  dollar13.2  /
      new_time  @30 new_time tod12.2 /
      new_pc ;
run;
```

Unformated:	Formatted:
18517	2010/09/12
123456.78	$123,456.78
30840	08:34:00.00
L0R 1T0	

INPUT and PUT statements may also be used in PROC SQL sentences to convert / reformat **bad_data** on the fly:

```
proc sql;
  select input(date,yymmdd8.)              as date format=yymmdds10.
        ,input(amt,comma12.2)              as amt  format=dollarx12.2
        ,input(time,time5.)                as time format=hour4.1
        ,put(upcase(postal_code),$20.-r)   as postal_code
    from bad_data
  ;
quit;
```

```
date          amt        time  postal_code
ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2010/09/12   $123.456,78   8.6              L0R 1T0
```

ROLL YOUR OWN FORMATS

Often you'll have very specific data conversion or presentation tasks that cannot be handled by the SAS-supplied (in)formats. For those cases it is possible to create your own informats and formats using the FORMAT procedure. User-defined formats perform the same type of operations as SAS-supplied formats, but allow a wider breadth of functionality due to the custom nature of the formats defined. Three operations are possible:

1. convert numeric values to character values
2. convert character values to numeric values
3. convert character values to other character values

Note that you cannot *directly* convert a numeric value to another numeric value using SAS or user-defined formats.

PROC FORMAT allows the creation of a number of different *styles* of informats or formats. It's possible to hard-code VALUE and INVALUE definitions, create PICTURE formats, or even create formats from values in a dataset. In addition, an existing format can be "unloaded" and used to create a dataset of the format specifications.

SIMPLE VALUE STATEMENTS

Where only a few values must be defined, the simplest method is using the VALUE and INVALUE statements. Remembering the format naming conventions outlined in the INTRODUCTION, the (in)format name must be specified and value pairs defined for each set of values. Additionally, the (in)format may be defined to a permanent library or, by default, defined to the WORK library.

For example, you may want to store a status description in place of a single-character status code. SAS beginners often code this type of conversion with a series of IF / THEN / ELSE or SELECT / WHEN statements:

```
data policy;
   length status_desc $8;
   format issue_date date9.;
   input policy_no issue_date yymmdd10. status_code $;
```

```
   select (status_code);
     when ('A')  status_desc = 'Active';
     when ('I')  status_desc = 'Inactive';
     when ('C')  status_desc = 'Closed';
     when ('P')  status_desc = 'Pending';
     otherwise;
   end;
cards;
00001 2009-04-05 A
00003 2008-12-28 C
00004 2009-09-11 P
00005 2007-01-17 I
00006 2008-07-09 Z
run;
```

In SAS-L nomenclature, this is known as “wallpaper” code (virtually identical lines of code forming a pattern). As the number of values increase, the repetition in the code increases. In addition, this could turn into a maintenance nightmare, especially if the same conversion was required in multiple programs. Far better to define a format and utilize the format in every program requiring the conversion. Later we'll talk about creating permanent formats *once,* storing them and accessing them when needed.

```
proc format ;
  value $status
   'A' = 'Active'
   'I' = 'Inactive'
   'C' = 'Closed'
   'P' = 'Pending' ;
run;

data policy;
   length status_desc $8;
   format issue_date date9.;
   input  policy_no  issue_date  yymmdd10.
status_code $;
   status_desc                          =
put(status_code,$status.);
cards;
00001 2009-04-05 A
00003 2008-12-28 C
00004 2009-09-11 P
00005 2007-01-17 I
00006 2008-07-09 Z
run;
```

```
title 'Policy Data';
proc print data = policy;
  var policy_no issue_date status_desc;
run;
```

PRINT procedure output:

Policy Data

Obs	policy_ no	issue_ date	status_ desc
1	00001	05APR2009	Active
2	00003	28DEC2008	Closed
3	00004	11SEP2009	Pending
4	00005	17JAN2007	Inactive
5	00006	09JUL2008	Z

Note the status description printed for student number 00006. When the input value is not found in the format specification, the input value is returned. In this case, the status_desc field contained ‘Z’ because the format did not specify a formatted value for ‘Z’.

VALUE RANGES STATEMENTS

Often, a range of values must be assigned to a single formatted value. In addition, input values not explicitly defined can be assigned a default, or ‘other’, output value. To convert a character value to a numeric value, the INVALUE statement is utilized to create an informat, useful in INPUT statements and INPUT functions.

```
proc format ;
  value grade
   low -<50 = 'F'   /* anything less than 50 is an 'F' */
   50 - 59  = 'D'
   60 - 69  = 'C'
   70 - 85  = 'B'
   85<-100  = 'A'
   other    = 'U'  /* any other value returns ‘U’ */
  ;

  invalue subjno (upcase)  /*UPCASE specified to make subject case-insensitive*/
   'ALGEBRA'   = 1
   'ENGLISH'   = 2
   'HISTORY'   = 3
```

```
    'COMPSCI'    = 4
    'GEOGRAPHY' = 5
    'LAW'        = 6
    'CALCULUS'   = 7
     other       = 99 ;
run;

data grade;
   input student_no subject : $12. mark  ;
   subject_no = input(subject,subjno.);
cards;
00001 Algebra 98
00001 English 50
00001 History 85
00001 CompSci 102
00003 Law .
00003 Calculus 76
00003 Chemistry 45
run;

title 'Grade Data';
proc print data = grade;
  var student_no subject_no subject mark;
  format  student_no z5.
          subject_no z2.
          mark grade.
        ;
run;
```

The data step uses the INPUT function and the `subjno.` informat to create a new numeric variable, **subject_no.** from the character subject value. The `grade.` format is employed at print time to convert the numeric grade value to a letter grade on output.

```
Grade Data
       student_    subject_
Obs       no          no       subject      mark

 1      00001         01       Algebra       A
 2      00001         02       English       D
 3      00001         03       History       B
 4      00001         04       CompSci       U
 5      00003         06       Law           U
 6      00003         07       Calculus      B
 7      00003         99       Chemistry     F
```

The values for student_no 00001 'CompSci' and 00003's 'Law' marks are assigned the 'other' value 'U' because their numeric values did not fall within the ranges defined. In the same manner, since 'Chemistry' wasn't specified in the informat definition, it too received the 'other' value of '99'.

BUILDING (IN)FORMATS FROM DATA

Using formats in the manner described above approximates the functionality and results of a table "look-up". It's often more efficient to process a table look-up using formats than to sort / merge or join via SQL. And, if your code values and descriptions are already in a SAS dataset, or in a form that's easily brought into SAS (e.g. Excel spreadsheet, text file or RDBMS table you can access via SAS) , there's no need to use the (IN)VALUE clause with it's hard-coded pairs of values. The CNTLIN option allows user-defined formats to be created from a SAS data set. For instance, if the subject name and subject number data from the previous example was stored in a dataset, the appropriate informat can be created very easily.

CNTLIN datasets must include the following fields:

fmtname	- name of format
type	- type of format, c=character, n=numeric, I=informat
start	- value to be converted
label	- converted value

```
data subjects;
  length subject $12;
  input subject subject_no;
cards;
ALGEBRA 1
ENGLISH 2
HISTORY 3
COMPSCI 4
LAW 6
CALCULUS 7
run;

data subjno_cntlin ( keep = fmtname type start label hlo );

   /*  Required fields for CNTLIN processing  */
   fmtname = 'subjno';  * format name ;
   type    = 'i';       * type = informat ;
   hlo     = ' ';       * high/low/other blank ;

   /*  Read format value pairs from dataset,  */
   /*  rename incoming variables as needed    */
   do until(eof);
      set subjects ( rename = ( subject = start subject_no = label ))
         end=eof;
      output;
   end;

   /*  Handle 'other'  */
   hlo   = 'o';
   start = ' ';
   label = 99;
   output;

   stop;
run;

proc format cntlin = subjno_cntlin;
run;
```

Sourcing the (in)format from a dataset (or data source) frees you up from maintenance that would otherwise be required when values are added, deleted or modified.

Conversely, formats can be “unloaded” to a dataset using the CNTLOUT option. Doing so permits the modification of the data underlying the user-defined format.

```
695  proc format cntlout = subject_cntlout;
696  run;

NOTE: The data set WORK.SUBJECT_CNTLOUT has 8 observations and 21 variables.
```

You will remember that when the subjno. format was created, we only used 5 variables. However the CNTLOUT operation created a SAS dataset with 21 variables! The label information on the CONTENTS listing provides more information about the data items that may be specified when creating user-defined formats. See the Online Docs for additional details.

```
      -----Alphabetic List of Variables and Attributes-----

 #    Variable    Type    Len    Pos    Label
ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ
20    DATATYPE    Char      8    101    Date/time/datetime?
18    DECSEP      Char      1     99    Decimal separator
 7    DEFAULT     Num       3    123    Default length
19    DIG3SEP     Char      1    100    Three-digit separator
16    EEXCL       Char      1     87    End exclusion
 3    END         Char      9     33    Ending value for format
```

```
12    FILL        Char      1     84    Fill character
 1    FMTNAME     Char      8     16    Format name
 9    FUZZ        Num       8      0    Fuzz value
17    HLO         Char     11     88    Additional information
 4    LABEL       Char     40     42    Format value label
21    LANGUAGE    Char      8    109    Language for date strings
 8    LENGTH      Num       3    126    Format length
 6    MAX         Num       3    120    Maximum length
 5    MIN         Num       3    117    Minimum length
11    MULT        Num       8      8    Multiplier
13    NOEDIT      Num       3    129    Is picture string noedit?
10    PREFIX      Char      2     82    Prefix characters
15    SEXCL       Char      1     86    Start exclusion
 2    START       Char      9     24    Starting value for format
14    TYPE        Char      1     85    Type of format
```

PICTURE FORMATS

We're accustomed to seeing certain data items in familiar formats. For example, phone numbers may be displayed with the area code enclosed in parenthesis and the exchange separated from the rest of the number with a dash. The user-defined formats discussed thus far *convert* data values from one form or value to another. Picture formats allow us to push **numeric** values through a template to control its presentation.

Picture formats are comprised of the digits 0 and 9 and special characters such as parenthesis, commas, decimal points and dashes that will separate the value in resulting output. In the following example we have two numeric fields, phone number and net worth. We've specified two different formats for the phone number depending on the presence or absence of a leading 1. Net worth is displayed in $millions, but we have to do something special for those with less than a million dollars.

```
data clients;
  input name $ phone net_worth;
  label name           = 'Name'
        phone          = 'Phone Number'
        net_worth      = 'Net Worth (Millions)';
cards;
Tom 4162561234 12345678
Dick 5196472472 29490000
Harry 18004558673 155678
George 9999999
run;

proc format ;
  picture phone
    2000000000-9999999999   = '999) 999-9999' ( prefix ='(')
    9999999999<-19999999999 = '9-999-999-9999'
    other                   = 'Invalid';
  picture million ( round )
    low-<1000000 = '9.0M'     (prefix='$' mult=.00001 )
    1000000-high = '0,000.0M' (prefix='$' mult=.00001 );
run;

proc print data = clients split=' ';
 format phone phone. net_worth million.;
run;
```

```
Client Data

                                    Net
                    Phone          Worth
Obs    Name         Number       (Millions)

 1     Tom      (416) 256-1234      $12.3M
 2     Dick     (519) 647-2472      $29.5M
 3     Harry    1-800-455-8673       $0.2M
```

In addition to the **prefix**, **round**, and **multiplier** options illustrated, picture formats also permits the specification of thousands separators, decimal separators and fill characters. SAS date, time and datetime data can be displayed in custom formats as well using a suite of directives to specify the presence / format of the portions of the date / time value:

```
data clients;
  input name $ start_date date9.;
  label name           = 'Name'
        start_date     = 'Start Date';

cards;
Tom 04Jan1984
Dick 27Feb2003
Harry 13Nov1994
George 11Sep2005
run;

proc format ;
  picture annodom
    other = 'Day %d of the month %B, %Y Anno Domini         ' (datatype=date);
run;

title 'Client Data';
proc print data = clients split=' ';
 format start_date annodom.;
run;
```

```
Client Data
                                    Start
Obs    Name                         Date

 1     Tom       Day 4 of the month January, 1984 Anno Domini
 2     Dick      Day 27 of the month February, 2003 Anno Domini
 3     Harry     Day 13 of the month November, 1994 Anno Domini
 4     George    Day 11 of the month September, 2005 Anno Domini
```

NESTED FORMATS

Sometimes the user-defined format is only required for a specified set of values. Rather than conditioning the application of formats by checking values of the incoming data, nested formats can be utilized. In this example, values less than a million are to be displayed using the COMMA. format, values in the millions are to display 'Millions', billions to display 'Billions', and anything higher displayed in exponential format.

```
proc format ;
  value largeno
    low-<1000000               = [comma7.]
    1000000-<1000000000        = 'Millions'
    1000000000-<1000000000000  = 'Billions'
    1000000000000-high         = [e13.2];
run;

data numbers;
   num = 123456;          output;
   num = 2345678;         output;
   num = 3456789012;      output;
   num = 4567890123456;   output;
run;
```

```
proc print data=numbers;
  format num largeno.;
run;
```

```
Nested formats

Obs    num

 1     123,456
 2     Millions
 3     Billions
 4      4.567890E+12
```

CREATING AND ACCESSING PERMANENT USER-DEFINED FORMATS

Often you'll want to create and permanently store user-defined formats. To do so, it's a matter of specifying the LIB= option on the PROC FORMAT statement.

```
libname mydata 'C:\_Stratia\Presentations\sesug_2010\data';
libname temp   'C:\temp';
```

```
proc format lib=temp;
  value largeno
    low-<1000000               = [comma7.]
    1000000-<1000000000        = 'Millions'
    1000000000-<1000000000000  = 'Billions'
    1000000000000-high         = [e13.2];
run;

data mydata.numbers;
   format num largeno.;       /*  Format permanently assigned to variable  */
   num = 123456;           output;
   num = 2345678;          output;
   num = 3456789012;       output;
   num = 4567890123456;    output;
run;

proc print data=mydata.numbers;  /*  Format permanently assigned to num  */
run;                             /*  so not specified in PRINT           */
```

However, you're not done yet, defining the library isn't enough to *access* permanent formats, you must tell SAS where to find them. By default, SAS automatically checks the WORK library and, if it exists, a libref of LIBRARY. Since our format was created in TEMP, running the PROC PRINT in the example will error because the **largeno.** format, permanently assigned to the NUM variable in the NUMBERS dataset could not be found in WORK or LIBRARY:

```
561  proc print data=mydata.numbers;
ERROR: The format LARGENO was not found or could not be loaded.
562  run;
```

To add TEMP to the format search path, the FMTSEARCH option must be correctly set.

```
options fmtsearch = ( temp work);
```

Now, when a non-SAS-supplied format is encountered in the PRINT procedure, the FORMATS catalogs in TEMP and WORK are successfully searched for the **largeno.** format.

POTENTIAL PITFALLS WITH FORMATS

It's not easy to have problems with formats, but there are a few gotchas to watch for.

MISSING FORMATS CATALOG

In the previous example, the **largeno.** format was permanently defined to the NUM variable in the MYDATA.NUMBERS dataset via the FORMATS statement in the data step. The format catalog was stored in the TEMP library, defined to C:\temp. Unfortunately, an overly exuberant hard-drive clean-up emptied the C:\temp directory, deleting the formats catalog containing **largeno.** When an attempt is made to PRINT the dataset, the error we received before issuing the FMTSEARCH options was again generated:

```
ERROR: The format LARGENO was not found or could not be loaded.
```

However because the FORMATS catalog has disappeared, it's no longer as simple as specifying the correct FMTSEARCH option to tell SAS where to find the format. It's not even possible to copy the NUMBERS data to a new data set:

```
700  data new;
701    set mydata.numbers;
702  run;

ERROR: The format LARGENO was not found or could not be loaded.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.NEW may be incomplete.  When this step was stopped there were 0
         observations and 1 variables.
```

By default, SAS errors and refuses to proceed if a format error is detected. However the **nofmterr** option is available to allow the step to continue when format errors are encountered.

```
options nofmterr;
```

Let's see what the log produced now that we've specified **nofmterr**.

```
704  options nofmterr;
705  title 'Largeno. AFTER Delete';
706  proc print data=numbers;
707  run;

NOTE: There were 4 observations read from the data set WORK.NUMBERS.
NOTE: At least one W.D format was too small for the number to be printed. The decimal may be
      shifted by the "BEST" format.

Largeno. AFTER Delete
Obs             num

 1           123456
 2          2345678
 3       3456789012
 4     4.5678901E12
```

While SAS found that the default BEST. format was inadequate for the data values, the dataset was processed despite the missing **largeno.** format.

INCORRECT FORMAT WIDTH

Way back in the introduction, when we were talking about the components of a format, we found out that the number before the period was the total allowed width. For example, the format **dollar12.2** allows 12 positions for everything, dollar sign, digits, commas, decimal point and, if we're talking the balances in government books, a negative sign.

```
balance = -1234567.89;
put balance dollar10.2;

214     balance = -1234567.89;
215     put balance dollar10.2;
-1234567.9
NOTE: At least one W.D format was too small for the number to be printed. The decimal may be
      shifted by the "BEST" format.
```

SAS has done the math and concluded you can't put 10lbs of stuff in a 5lb bag. Rather than ensuring **balance** was displayed replete with dollar signs and commas but missing significant digits, SAS defaulted to the **best.** format and pumped out as many significant digits as it could into the allotted 10 bytes.

Sometimes, despite SAS's **best.** efforts, there just isn't room. In those cases a string of asterisks is output along with the " ain't gonna fit " log message. You've seen this same type of behavior in Microsoft Excel which displays # signs when the cell width is not adequate to contain the value in it.

```
widgets = 123;
put widgets z2.;
```

```
218  data _null_;
219     widgets = 123;
220     put widgets z2.;
221  run;
**
NOTE: At least one W.D format was too small for the number to
be printed. The decimal may be shifted by the "BEST" format.
```

INFORMAT WITH DECIMALS

Many of the numbers we deal with are not integers. The fact that we have to account for decimals leaves us open to another pitfall. Consider the following data step where different amounts with varying precision are read. We've specified an informat with a maximum length of 7 (including decimal point) and two decimal digits.

```
data all_amts;
   format amt 7.2;
   input amt 7.2;
cards;
123.45
3.7
567.88787
67
11.
run;
```

```
Obs        amt

 1      123.45
 2        3.70
 3      567.89
 4        0.67
 5       11.00
```

From the PRINT output, we can see that our disparate data has been handled quite nicely for the most part. It is readily apparent that SAS took the decimal point supplied in the input data at face value and truncated or added zeroes to flesh out the **7.2** format. But, there's a problem with **67**. Note that this line of input data did **not** contain a decimal point. When using a W.D numeric informat, SAS divides the input data by 10^D (10 ** D) *unless* the input data contains a decimal point. Be consistent or your data values may end up smaller by magnitudes!

EASY LIVING VIA FORMATS

We've already seen some benefits of user-defined formats, custom formatting, data conversion and table lookup operations, but there's more. There's other gains to be realized, formats can also deliver us from some of the drudgery in our SAS life. Let's look at a few time savers available via the means of formats.

JUSTIFICATION

Typically, character data are left-justified, numeric data right-justified. There's times when you might want to left-justify a numeric value. Perhaps you're ticked at the chief accountant, so why not left-justify all the columns of numbers on that daily journal entry report? A more realistic application, creating labels for report lines. Note how the **line** value of **9** is snuggled up to the hyphen when the -L format modifier is added:

```
data _null_;
  do line = 9 to 10;
     left  = 'Line-' || put(line,2.-L);
     right = 'Line-' || put(line,2.);
     put @2 left= / right=;
  end;
run;
```

Log Output:

```
 left=Line-9
right=Line- 9
 left=Line-10
right=Line-10
```

RETENTION OF LEADING ZEROES

In most numeric data, the high-order, leading zeroes are not required on output. If we have an amount field with values ranging from 4 to 2000, we wouldn't want the lower values to be displayed with leading zeroes, e.g. 0004. However, there are times when leading zeroes are important. Consider part numbers, invoice and cheque numbers and the like:

```
a = 4;
put a @10 a z4.;
```

Log Output:

```
4          0004
```

PRESERVATION OF LEADING SPACES

By default, SAS ignores leading spaces. Well, worse than that, it'll get rid of them if it can. If leading spaces are important, you must explicitly tell SAS to keep 'em via the $CHAR. informat at INPUT time:

```
data a;
   input @1 fruit1 $10.     @11 where1 $10.
         @1 fruit2 $char10. @11 where2 $char10.;  /*  Read line again  */
```

```
cards;
12345678901234567890
     Apple   Orchard
    Orange     Grove
     Grape     Store
run;
```

Note the **fruit2** and **where2** retained the leading spaces while **fruit1** and **where1** are left-justified:

Obs	fruit1	where1	fruit2	where2
1	1234567890	1234567890	1234567890	1234567890
2	Apple	Orchard	Apple	Orchard
3	Orange	Grove	Orange	Grove
4	Grape	Store	Grape	Store

SUMMARIZE DETAIL USING FORMATTED VALUES

Base SAS comes with many useful procedures designed to make common computing tasks more efficient; more efficient both in terms of coding and processing effort. As a rule, if SAS has provided a procedure to deal with a data manipulation task, it really ought to be used over a home-grown solution. For example, the TABULATE procedure displays descriptive statistics (e.g. mean, sum) in tabular format with very little coding effort. Consider a small data set containing sales figures by date:

(Note: date field literals defined with the '*date value*'**d** syntax are *automatically* converted to internal SAS date values. ie. the DO loop iterates from 15706 to 16070, the internal values for 10Jan2003 and 31Dec2003).

```
data analysis ;
  format date   yymmddd10.
         sales  dollar12.2;
  do date = '01jan2003'd to '31dec2003'd by 11;
      sales  = ranuni(1) * 20000;
      output;
  end;
run;
```

The date field is stored in internal SAS format so the benefit of SAS date formats can be brought to bear on the field. The simplest way to summarize the sales figures by month and quarter is using the simple TABULATEs below. Note that no pre-processing of the data was required and the only differences between the TABULATE steps is the FORMAT specified for the **date** variable.

```
proc tabulate data=analysis;
   class date;
     var sales ;
   table date='Month /
Year',sales='Sales' * sum=' '
*f=dollar12.2;
  format date monyy5.;
run;
```

Note that the **monyy5.** format has been applied to the date field, thus summarizing the data by month and year.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,                     ,   Sales    ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,Month / Year         ,            ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ‰            ,
,JAN03                ,  $31,097.51,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,FEB03                ,  $43,005.57,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,MAR03                ,  $22,489.30,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
   etc…
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,OCT03                ,  $41,397.52,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,NOV03                ,  $33,759.13,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,DEC03                ,  $47,338.74,
Šƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

```
proc tabulate data=analysis;
   class date;
     var sales ;
   table date='Quarter',sales='Sales'
* sum=' ' *f=dollar12.2;
  format date yyq6.;
run;
```

Note that the **yyq6.** format has been applied to the date field, thus summarizing the sales data by year and *quarter*. Same dataset, same field, same data, but different results via simply altering the format specification.

```
„ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ…ƒƒƒƒƒƒƒƒƒƒƒ†
,                     ,   Sales    ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,Quarter              ,            ,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ‰            ,
,2003Q1               ,  $96,592.38,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2003Q2               ,  $77,143.28,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2003Q3               ,  $86,315.43,
‡ƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒˆƒƒƒƒƒƒƒƒƒƒƒ‰
,2003Q4               , $122,495.39,
Šƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒƒ‹ƒƒƒƒƒƒƒƒƒƒƒŒ
```

$CONCLUSION.

Sometimes it seems as though our working life consists of nothing but data, mountains of it. SAS is our favorite tool for scaling that mountain, reading and massaging data, dicing, summarizing, storing and finally presenting it again in a meaningful manner. Informats and formats, both SAS supplied and user-defined, are indispensable in all these activities.

REFERENCES

http://www2.sas.com/proceedings/sugi29/236-29.pdf - *Building and Using User Defined Formats*, Arthur Carpenter, SUGI 29 236-29.

SAS-L, many posts over many years. If you don't know what SAS-L is, run, don't walk to your computer and enter http://listserv.uga.edu/archives/sas-l.html in your browser address bar. Subscribe, browse, search the archives, learn.

SAS Online Docs, SAS Institute, Cary, NC

CONTACT INFORMATION

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

Harry Droogendyk
Stratia Consulting Inc.
www.stratia.ca