Paper 054-2010

SAS 1-liners

Stephen Hunt, ICON Clinical Research, Redwood City, CA

ABSTRACT

In virtually all programming languages, code efficiency is a perpetual, though occasionally implicit, aspiration. Terms used to describe programs or sections of programs as 'elegant', 'slick' or 'smooth' insinuate an uncertain boundary between art and science in programming. Code that performs correctly is one thing. Code that performs correctly *and* manages to do so in a highly-efficient or unique manner 'ups the ante' by promoting interest and the sharing of ideas in a way that 'mundane' code cannot. Those enamoured by code efficiency often find themselves unable to resist approaching a programming problem by, paradoxically, using as little programming as is feasible. To that end, this paper seeks to challenge both the beginning and experienced programmer alike to look for new ways to accomplish routine SAS programming tasks as stylistically as is only humanly possible.

INTRODUCTION

SAS courses and texts on programming efficiency highlight at least 2 aspects: Code efficiency and processing efficiency. Although these two are frequently complimentary of one another, this paper will focus on the former, even if in some cases this appears in opposition to the latter. The concepts and examples in this paper generally emphasize one criteria above all else in accomplishing a programming task: The fewest number of keystrokes possible to complete the programming requirement. This isn't at all however meant to suggest that such simplifying criteria is optimal in all circumstances. Programmer discretion is advised.

DYNAMIC DENOMINATORS

Conditional clauses likely represent the single greatest inefficiency area in programming. Efficient programming requires avoiding if/then/else statements at all cost. Our first example involves determining denominators as macro-variables to be used in summary tables. A 'condition-less' technique is as follows:

```
proc freq data=sasdata.adsl;
   table trt1pn / out=trttots;
run;

data _null_;
   set trttots;

   if trt1pn>.z then call symput('n'||compress(trt1pn),compress(count));
run;
```

In the above symput call, it doesn't matter how many levels of trt1pn exist, as macro variables for n1, n2, n*y*… will all be created driven by the available treatment levels in the data.

Once you're ready to use your treatment count macro variables to actually calculate percentages, a corresponding one-line statement deploying a 'resolve' function can return your totals to you just as quickly and cleanly:

```
proc freq data=sasdata.adsl noprint;
   table trt1pn*agegrp1 / out=agecats;
run;

data agecats;
   set agecats;

   if trt1pn>.z then denom=resolve('&n'||compress(trt1pn));
   percent=count/denom*100;
run;
```

Of course, if our denominator is used exclusively to create a percentage, the additional variable 'denom' above is superfluous; thus, an even better approach would be…

```
   if trt1pn>.z then percent=count/resolve('&n'||compress(trt1pn))*100;
```

'DUMMY' TREATMENT CODE ASSIGNMENTS

Most of the time in clinical trials, programmers are 'blinded' to the actual treatment to be given to subjects. This 'blinding' persists during most of the tasks of TLG (table, listing, and graph) development. As such, it's useful to create artificial treatment values to be used prior to unblinding. This can be accomplished relatively simply with the following:

```
trt1pn=ceil(ranuni(0)*x); *** WHERE X CREATES X LEVELS OF TRT1PN.
```

The only adjustment to the above necessary to create different treatment levels is the multiplier (x) itself (e.g., *2 for 2 levels, *3 for 3 levels, etc… Note that the above distribution provides approximately equal (50% each for 2, 33% each for 3, etc..) dummy-treatment assignments. For studies with non-equal randomization schemes, variations on the above will be required. For example, the code below distributes 3 groups at approximately 25%/50%/25%.

```
trt1pn=ceil(ranuni(0)*2)+ceil(ranuni(0)*2)-1;
```

REVERSING OR RE-ARRANGING VALUES

One of my favorite types of 1-liner conditional replacements involves reverse-coding. On many clinical questionaires, some individual results are often required to contribute to a total scored in the opposite direction from which they are collected (theoretically, this strategy helps to ensure internal validity through the voluminous questionaire documents). As such, a common value reversal might look like this:

4 → 1
3 → 2
2 → 3
1 → 4

In applying this to programming, not only is a set of if-then statements a deplorable waste of keystrokes, if used carelessly, it can produce results that are just plain wrong. Consider the following, which will **not** succeed in the task of reverse coding:

```
if var=4 then var=1;
if var=3 then var=2;
if var=2 then var=3;
if var=1 then var=4;
```

Of course, most programmers wouldn't be foolish enough to leave an 'else' off the last 3 lines, but why not go with a simple one-liner instead:

```
if var>.z then var=abs(5-var);
```

In this instance, the data-step may be at a disadvantage relative to PROC SQL (something I'm never happy to admit), because the number of levels of a variable can't be determIned in a single line dynamically outside of SQL (at least not to my knowledge), thus the above '5' would require adjustment to '4' (i.e., number of levels + 1) for a question containing only 3 levels that requires reverse coding, etc... Fortunately, taking it further with SQL's 'distinct' operator can allow the programmer to ignore the number of levels (carefully, to some extent… more below):

```
proc sql noprint;
   create table d2 as select abs((count(distinct var)+1)-var) as var from d1;
quit;
```

The shortcoming of the 'dynamic' SQL approach is that if any 1 level is entirely missing, the appropriate number of levels won't be passed into the count(distinct) statement. Thus, adding in an additional safeguard of the maximum value of the levels *can* help out (assuming the maximum level isn't one that is all missing, and that your levels are all ordered, fixed-interval integers, the later of which is usual, the former of which is certainly not guaranteed):

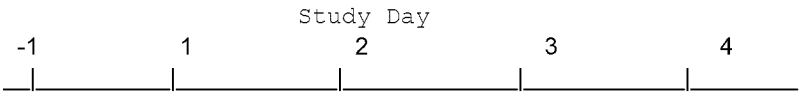
```
proc sql noprint;
   create table d2 as select abs(ceil((max(var)+count(distinct var))/2+1)-var) as
var from d1;
quit;
```

Of course, code 'switching' can involve more intricate re-assemblies than mere reversal. In some cases, repeated DMC deliverables require 'shuffling' of partially-unblinded (e.g., A, B, C, D) treatment codes in a different form for each deliverable. For example:

```
*** RE-CODING TREATMENT IN THE FOLLOWING MANNER FOR JUN2009 DMC:(4=1, 3=2, 1=3,
2=4).;
newtrt=abs(trt1pn-((trt1pn<3)*-2)-((trt1pn>2)*5));
```

STUDY 'DAY' CALCULATION

One of the most common calculations used across all types of programming is determining a relative 'day' based on 2 date fields. In clinical trials, the initial 'study day' is generally considered to begin at either randomization or dosing, thus assessments made prior to this starting point require a slight variation in the calculation in order to preserve the typical 'no day 0' concept:



When it comes to calculating such, some programmers opt to bother with evalulating whether a visit date occurred prior to or on/after randomization:

```
if visdt>=randdt>.z then stydy=visdt-randdt+1;
else stydy=visdt-randdt;
```

However, this is unnecessary, since a 1-liner will suffice:

```
if visdt>.z & randdt>.z then stydy=visdt-randdt+(visdt>=randdt);
```

USING FORMATS INSTEAD OF CONDITIONAL CLAUSES

This is more a general suggestion than a specific one-liner, but still applies because it can contribute significantly to trimming keystrokes from your code. Applying formats instead of conditionals allows for code re-use with minimal overhead (e.g., a single format or informat used in either multiple programs or multiple sections of the same program can be controlled in a single location). In the following example, an informat is set up with various age category ranges corresponding to aribrary numeric categories (1 – 5).

```
proc format library=work;
   invalue incat
   0 -7  = 1
   8 -18 = 2
   19-35 = 3
   36-65 = 4
   65-high=5;
run;

data adsl;
   set adsl;

   age_cat=input(age,incat.);
run;
```

TRANSPOSING MULTIPLE VARIABLES IN A SINGLE PROC STEP

In numerous situations, transposition on more than 1 variable is necessary in order to derive header and sub-header columns for summaries in a particular table. Many programmers opt for either 2 separate PROC TRANSPOSE steps or 1 transpose and an additional data step with a merge in order to accomplish this. However, it's actually easiest to accomplish this in a dataset plus a single transpose. In the following example, the work dataset SCORE contains binary treatment and 5-level race category variables to be presented as column headers when summarizing a total score.

```
data score;
   set score;

   trt_race=(10*trt1pn)+race;
```

```
run;

proc sort data=demog;
   by trt_race;
run;

proc univariate data=demog noprint;
   by trt_race;
   var score;
   output out=univout mean=mean_score;
run;

proc transpose data=univout out=univout;
   id trt_race;
   var mean_score;
run;
```

This results in columns of _11, _12, _13, _14, _15, _21, _22, _23, _24, & _25 containing mean values for SCORE for each treatment/race category. Naturally, this technique stays the same for 3, 4, or however many columnar variables/levels are required (and that you can fit on a page). Any new field values added into the single transposition variable merely need to be included such that the components can still be distinguished [e.g., trt_sex_race= (100*trt1pn) + (10*gender) + race].

DATETIME CALCULATION

When creating date/time variables, if the time component is sporatically missing, the entire date/time field will end up missing for that record. Obviously this is undesirable, but there's no need to waste keystrokes with an if/then statement to evalulate for presence of time:

```
labdttm=dhms(labdate,max(0,hour(labtime)),max(0,minute(labtime)),0);
```

IDENTIFYING SPECIAL CHARACTERS

On occasion, characters from sets not recognized by SAS make their way into SAS datasets (especially in cases where the source is Excel/CSV files). One specific example is line-feed characters. When SAS reads these, it attaches them to the last variable/column in each row. Since many special characters won't show up 'visually' in printed renderings of the data, programmers must rely on code to figure out when one of these fields appears. The following code searches through all character fields in a dataset and removes special characters when/if they're encountered:

```
data random_csv(drop=i);
   set random_csv;

   %macro repair;
      %do i=1 %to 255;
         if indexc(byte(&i),':;=1234567890-~!@#$%^&*()_+}{[]\|/><.,
                           ABCDEFGHIJKLMNOPQRSTUVWXYZ
                           abcdefghijklmnopqrstuvwxyz ')=0 then do;
            if chars(i)^=compress(chars(i),byte(&i)) then do;
               put "WAR" "NING: Special Character Found at byte=&i for variable "
                   chars(i)= "Fixing!";
               chars(i)=compress(chars(i),byte(&i));
            end;
         end;
      %end;
   %mend repair;

   array chars (*) _character_;

   do i=1 to dim(chars);
      %repair;
   end;
run
```

DATASET SOURCE FLAG IDENTIFICATION

When merging or setting together several similar datasets, it's often useful to create a flag using an (in=) variable.

One way to create a single 'source' variable from multiple 'in' variables is as follows:

```
data one;
   do i=1 to 6;
      output;
   end;
run;

data two;
   do i=7 to 12,16;
      output;
   end;
run;

data three;
   do i=13 to 15;
      output;
   end;
run;

data all;
   set one(in=in_1)
       two(in=in_2)
       three(in=in_3);

   source=sum((in_1=1),(in_2=1)*2,(in_3=1)*3);
run;
```

In the example above, the 'source' variable is derived from the sum of three evaluation statements, with each one 'weighted' with the appropriate multiplier to reflect the appropriate dataset source.

DATE IMPUTATIONS

Occasionally, date imputation rules require imputing to the end of the month when the day is unknown and the end of the year when the both day & month are unknown. This one's actually a 2-liner, but given that the year imputation always goes to 31Dec, maybe we won't count that one:

```
data test;
   date="00/000/2008"; output;
   date="00/DEC/1996"; output;
   date="00/JAN/1996"; output;
   date="00/FEB/2000"; output;
   date="00/FEB/1999"; output;
run;

data test;
   set test;

   if index(date,'00/000/')>0 then do;
      dateI=tranwrd(date,'00/000/','31/DEC/');
   end;
   else if index(date,'00/')>0 then do;
      dateI=upcase(put(day(input("01"||
      trim(left(put(month(input(compress(tranwrd(date,'00/','01/'),'/'),date7.))+1-
                              ((index(upcase(date),'DEC')>0)*12),z2.)))||
      trim(left(put(year (input(compress(tranwrd(date,'00/','01/'),'/'),date7.))+
                               (index(upcase(date),'DEC')>0),4.))),ddmmyy8.)-
1),z2.)||
            substr(trim(left(date)),3));

      put "NOTE: Imputing start date day to beginning of month! " date= dateI=;
   end;
run;
```

Results in…

```
    date           dateI

 00/000/2008    31/DEC/2008
```

```
00/DEC/1996    31/DEC/1996
00/JAN/1996    31/JAN/1996
00/FEB/2000    29/FEB/2000
00/FEB/1999    28/FEB/1999
```

Essentially, what the above code accomplishes is creating an actual SAS date derived as the beginning of the month (day 01) for the month that follows the month given in the partial date being imputed. Once this date has been created, subtracting 1 from the SAS date value always returns the last day of the preceeding month (the month of the partial date) without having to evaluate which month has how many days or whether a leap year (and thus 29Feb) is involved.

CONVERTING LETTERS TO NUMBERS

And finally, for entertainment value only (i.e., I can't recall where I used this, but I'm sure it was no more than once), you can convert any letter in the alphabet into it's ranked number (1 – 26) with a simple 1-liner:

```
data test;
   do letter='A','B','C','D','E','F','Z';
      output;
   end;
run;

data test;
   set test;


letternum=indexc(tranwrd('ABCDEFGHIJKLMNOPQRSTUVWXYZ',trim(left(upcase(letter))),'*'
),'*');
run;
```

The trick in this example involves 2 steps: The first is replacement of the letter value within the entire alphabetic string with a star '*' (though any non-alphabetic character will suffice). Afterwards, indexc is used to identify the location within the alphabetic string of this arbitrary '*' character that has been used to replace the particular letter being derived as a number.

CONCLUSION

When it comes to 'code-slinging', less is indeed more. Finding a programming solution using as few keystrokes as possible isn't necessarily something that will earn you a bigger bonus or a promotion. But rather, much like a particularly good Chess move or Sudoku game, the inherent satisfaction is truly reward in itself.

ACKNOWLEDGMENTS

The author would like to thank the myriad of talented programmers that he's had to good fortune to work with over the years who have introduced him to numerous 'slick' coding concepts & strategies. Thanks especially to Jackie Lane for the encouragement and suggestions and to David Carr for the helpful review and comments.

CONTACT INFORMATION

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

Stephen Hunt
ICON Clinical Research
Suite 500, 303 Twin Dolphin Rd.
Redwood City, CA 94065
Email: stephen.hunt@iconplc.com