Chapter 1
A Collection of Useful Tips

How to fix unbalanced quotes

Sometimes when using lots of character strings and/or comments you may get your SAS code a little confused, ending up with messages like:

CHARACTER STRING WAS MORE THAN 200 CHARACTERS

This indicates that you may have unbalanced quotes. To close unbalanced quotes or comments, submit:

```
*'; *"; */;
```

This closes any unbalanced quotes or /* comments, and the statements are three harmless comments if no such unbalanced strings exist.

If things still don't work and if you are using macros, submit:

```
*); */; /*'*/ /*"*/; %mend;
```

This will close an unclosed macro argument list, close unbalanced quotes inside a macro (quotes and comments work differently inside macros), and terminate an unclosed macro definition.

If things still don't work, hit the ATTENTION/BREAK/INTERRUPT key. This key varies depending on your operating system and hardware being used. For instance, in Windows you can generally hold down the control key and then press the break key. Please refer to the documentation for your operating system.

If nothing happens, hit the RETURN or SUBMIT key. Do NOT hit the ATTENTION/BREAK/INTERRUPT key again, since that may kill the SAS session. You should get a message like this:

Press

Y to cancel submitted statements,
N to continue.

Carefully press Y and on some systems RETURN. This should return the tokenizer to a pristine state. If it doesn't, you have found a bug. Remember that the ultimate "fix" is to enter the ENDSAS or BYE command. This will close SAS so that you can restart it in a normal state.

Using a wildcard in variable lists

Using a wildcard in variable lists can save you a lot of typing and make your code much more generic (in some instances).

The colon can be used as a wildcard in variable lists. For example, ABC: means all variable names beginning with ABC.

```
data x(keep=a:) ;
  a1=1 ;
  a2=10 ;
  a3=100 ;
  b1=1000 ;
  b2=10000 ;
run;

The data set WORK.X has 1 observations and 3 variables.
 The DATA statement used 0.01 CPU seconds and 1435K.
```

Notice that the three variables starting with the letter ‘a’ have been kept. We can also use ‘a:’ in procedures.

```
proc print ;
  var a:;
run ;

The PROCEDURE PRINT used 0.01 CPU seconds and 1489K.
```

The colon must be at the end of the name, not embedded—AB:C is invalid. Colons can be used in most places where other abbreviated variable lists, such as ABC1-ABC99, are allowed. It cannot be used in a SUM function in place of a list of variables.

Using wildcards to read external files

To read many identically laid-out and consecutively-named files (for example, Tab1.dat, tab2.dat, and so on) in a single DATA step in Release 6.08 and higher under Windows, Windows NT, VMS or OS/2 (not MVS or CMS), you can use the FILENAME statement with a wildcard (asterisk or question mark).


```
filename in 'c:\tab*.dat' ;

data report ;
  infile in ;
  input a b c ;
run ;
```

Commenting out code containing comments

Sometimes it is necessary to comment out an entire section of code, including DATA steps, procedures and other comments (of the /* */ kind). Traditional /* */ symbols will not work to comment out the code if there are other comments of that type in the code you want to comment out.

You can make a whole section of code into a macro, which will effectively comment it out. Be careful not to comment out code that contains a macro definition; otherwise, this technique may not work as expected, since the %MEND that you add may end the other macro. For example,

```
%macro junk ; * this statement used to comment out following
code ;

* do the next data step ;
data this ;   /* this is a nice dataset */
  set that ;
  x+1 ;        /* add 1 to x */

%mend ;  this statement used to comment out previous code ;
```

If you are using the macro statements to comment out a block of code, you can make the log more readable by using the NOSOURCE option to stop your commented out code from printing to the log.

```
options nosource;
%put **;
%put ** note: excluding some code. be back soon.;
%put **;
%macro junk ; * this statement used to comment out following
code ;
```

```
* do the next data step ;
data this ;   /* this is a nice dataset */
  set that ;
  x+1 ;          /* add 1 to x */

%mend ; this statement used to comment out previous code ;
options source;
```

Another possibility is to enforce the use of the "* ;" statement for all comments in open code. Then the "/* */" form can be reserved for commenting out large blocks of code. This technique requires that programmers observe this coding standard, since if they don't, you can get undesired results.

Sizing the screen space used by SAS applications

Why?

When using SAS interactively, it is usually desirable to maximize the space available to your application. This tip explains how to do so automatically.

The SAS Application Workspace (AWS) window can be sized and its menu bar turned off when you run an application in SAS (under Windows or OS/2). This is useful to ensure that maximum screen space is available for the application. Putting the following lines into your SAS configuration file (usually CONFIG.SAS) or on the SAS command line will accomplish this.

Part of CONFIG.SAS

```
* cause SAS to use 100% of the display area
* turn off the AWS menu bar.
Part of CONFIG.SAS
-awsdef 0 0 100 100
-noawsmenu
```

Data encryption for the beginner

Why?

The data stored in some SAS data sets is sensitive and needs protection. This tip provides sufficient protection for many security requirements.

If you don't yet have Release 6.11 of the SAS System, which has a built in data encryption feature, or if you are looking for further data encryption techniques, then this tip may be for you.

The BXOR function, added to base SAS in Release 6.07, presents budding encrypters with a basic tool for encrypting numbers. The function works by returning the result of a binary exclusive OR between the 2 arguments. The example below shows how a 'key' can be used to change 'original' numbers into 'coded' numbers. Then by doing a BXOR against those 'coded' numbers with the same key, the numbers are returned to the 'original' value.

This technique truncates values to the nearest 1 (see the value '34.4' in the example program below). If using decimal places (for example, money) then you should convert numbers to INTEGERS (for example, express the value in cents).

If you want to see what is going on a little better, you can use the following format to see where the bits are set in keys and data. This will enable you to easily check the function by hand.

Format	Description	Width Range	Default Width	Alignment
BINARYw.	converts numeric values to binary	1-64	8	left

Code

```
data coded ;
 * set the value of the key ;
  retain key 1234567 ;
  format key original coded binary.;
  input original ;
 * encode the original value using the key ;
  coded=bxor(original,key);
  put key= original= coded= ;
  cards ;
1
1234567
999999
34.4
0
run ;
```

```
data decode ;
 * the value of the key must be the same,
   or else the number will not decode correctly ;
  retain key 1234567 ;
  set coded ;
 * decode the coded value using the key ;
  decoded=bxor(coded,key);
  put coded= decoded= ;
run ;
```

The following log shows the results of the first DATA step, which encodes the numbers.

```
KEY=1234567 ORIGINAL=1        CODED=1234566
KEY=1234567 ORIGINAL=1234567 CODED=0
KEY=1234567 ORIGINAL=999999  CODED=1938616
KEY=1234567 ORIGINAL=34.4    CODED=1234597
KEY=1234567 ORIGINAL=0        CODED=1234567
NOTE: The data set WORK.CODED has 5 observations and 3 vari-
ables.
NOTE: The DATA statement used 0.01 CPU seconds and 1451K.
```

The following log shows the results of the second DATA step, which decodes the numbers.

```
CODED=1234566 DECODED=1
CODED=0        DECODED=1234567
CODED=1938616 DECODED=999999
CODED=1234597 DECODED=34
CODED=1234567 DECODED=0
NOTE: The data set WORK.DECODE has 5 observations and 4 vari-
ables.
NOTE: The DATA statement used 0.01 CPU seconds and 1471K.
```

Capturing screens under Windows

Why?

There are many times when you might like to capture a screen image from SAS, such as when documenting SAS/AF applications.

Pressing ALT-PRTSCR under Windows 3.1 or Windows 95 captures the current screen and drops its graphic image into the clipboard, allowing pasting into applications such as Microsoft Word and Microsoft PowerPoint.

If there is a pop-up menu on the screen, then pressing ALT is considered to be a response to the pop-up menu, which means the pop-up menu disappears before the screen image is captured. However holding down PRTSCR and then pressing ALT will do a screen print with the pop-up menu intact.

Cautions in dealing with missing values

Here are several things to be aware of when dealing with missing values.

1. Most operators propagate missing values, but comparison operators treat them as negative infinity, so X<0 is true when X is missing.
2. Adding and/or subtracting several numbers using the plus (+) and minus (-) operators will return a missing value if any of the numbers are missing. To get around this, you should use the SUM function. The SUM function will only return a missing value if all of the values/variables being added are missing. If you always want a 0 returned, rather than a missing value, use SUM(0,var1,var2,...).
3. Many procedures (for example, SUMMARY, TABULATE, FREQ, CALENDAR, and so on) ignore missing values or at least treat them in a different way. Usually, missing values are ignored by default, and you should override the default if you want them included.
4. Some procedures have different statistics for missing and non-missing variables. For instance, PROC SUMMARY has N for the number of non-missing observations and NMISS for the number of observations with missing values. If you use PROC MEANS with a CLASS statement, you can get the NOBS statistic, which is the sum of missing and non-missing observations.


```
data _null_ ;
 * Initialise values ;
  a=. ;
  b=0 ;
  c=-7 ;
  d=99 ;
 * Try various forms of addition involving missing values ;
  add=a+b+c+d ;
  put 'Addition of missing & non-missing values :  ' add= ;
  sum=sum(a,b,c,d) ;
  put 'Sum of missing & non-missing values :  ' sum= ;
  summiss=sum(.,a) ;
  put 'Sum of missing values only :  ' summiss= ;
  sumzero=sum(0,.,a) ;
  put 'Sum of 0 and missing values :  ' sumzero= ;
 * See how the missing value compares to zero ;
  if a<0 then
    put 'Missing is less than 0' ;
  else if a>0 then
    put 'Missing is greater than 0' ;
run ;
```

In the following log, you can see the results of performing various operations on missing values. You will also notice the NOTE: about missing values being generated. This note warns you that some calculations in your code contain missing values and have, therefore, resulted in missing values. Usually you don't want this to happen, so be aware of this message. Use the (Line):(Column) to locate the places that it occurs so that you can verify that it is OK or else fix it.

```
Addition of missing & non-missing values :  ADD=.
Sum of missing & non-missing values :  SUM=92
Sum of missing values only :  SUMMISS=.
Sum of 0 and missing values :  SUMZERO=0
Missing is less than 0

NOTE: Missing values were generated as a result of performing
an operation on missing values.
Each place is given by: (Number of times) at (Line):(Column).
1 at 77:8    1 at 77:10   1 at 77:12   1 at 81:11
The DATA statement used 0.02 CPU seconds and 1420K.
```

5. Special missing values are handled in a way you may not necessarily expect. It is not the case that if you "do something" with a special missing value, it will be propagated as a special missing value. Consider the following example.

```
data temp;
  missing Z;
  input a;
  b = a + 0;
cards;
7
4
.Z
5
;

proc print ;
run ;
```

In the output you may have expected that the special missing value when added to 0 would result in the same special missing value (.Z). However, it results in a normal missing value (.). Note that the ‘Z’ shown in the output is actually the missing value ‘.Z’. SAS does not yet print it as ‘.Z’.

OBS	A	B
1	7	7
2	4	4
3	Z	.
4	5	5

NOTE: Z is a special missing value, which is different from a normal missing value.

Saving resources when the log is long

Why?

When writing a lot of information to the log in an interactive SAS program, you can be slowed down as SAS scrolls the log to display each line as it is written. This is the default behavior, which works well when you don't write much to the log. This tip will tell you how to save time and resources by altering the AUTOSCROLL setting.

Activating the LOG window (by selecting Log from the Globals menu or simply clicking on the LOG window) and setting AUTOSCROLL to 0 tells SAS not to bother scrolling the LOG window until the DATA step is finished. AUTOSCROLL can be set by using the pull down menus to choose EDIT then OPTIONS then AUTOSCROLL.

The example writes 2000 lines (that is, 2 lines per PUT statement) to the log. The first execution (130 secs) uses the default AUTOSCROLL value, whereas the second execution (27 secs) sets AUTOSCROLL to 0.


```
data _null_ ;
  set sasuser.fitness;
  do i=1 to 1000 ;
    put _all_ ;
  end ;
  stop ;
run ;
NOTE: The DATA statement used 2 minutes 10.33 seconds.

.....  same code but with AUTOSCROLL set to 0 .....
NOTE: The DATA statement used 26.42 seconds.
```

Additional SAS documentation

If you want more information about the tips covered in this section, then try reading the relevant SAS documentation.

These manuals include:

- *SAS Programming Tips: A Guide to Efficient SAS Processing*
- *SAS Companion for the OS/2 Environment, Version 6, First Edition*
- *SAS Companion for the MVS Environment, Version 6, Second Edition*
- *SAS Companion for the Microsoft Windows Environment, Version 6, First Edition*
- *SAS Companion for the Microsoft Windows Environment, Version 6, Second Edition*
- *SAS Software: Abridged Reference, Version 6, First Edition*
- *SAS Language and Procedures: Usage 2, Version 6, First Edition*