ABSTRACT

One of the most versatile mechanisms in the SAS data step for iterating is DO the loop in its many forms. All programmers of the SAS System need to be familiar with at very least the basics to get the most from the data step.

This paper considers several uses and forms of DO loops, from the basic DO – END iteration, through lesser known uses of iteration with dates, letters and expressions.

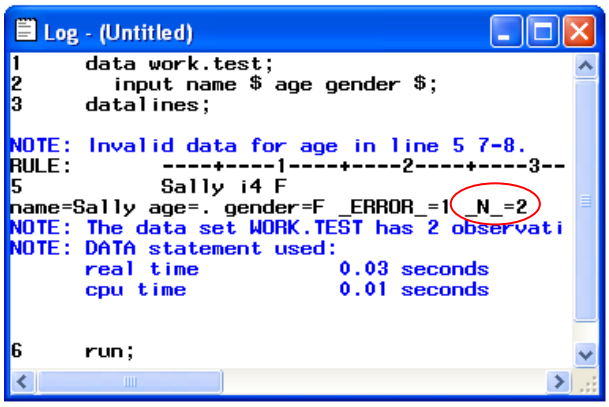
Further techniques are covered by example, including looping through elements of arrays, avoiding infinite loops and finally conditional termination of WHILE and UNTIL loops.

This paper will be of benefit to any Base SAS programmer who wishes to further their understanding of looping with the data step.

HERE WE GO…

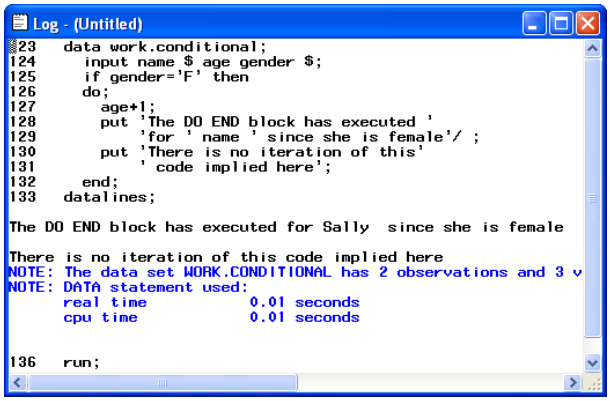
Let's begin by reaffirming what we all need to understand about how looping actually works in the data step.

Every data step that you have ever written is in itself an iterative loop. The data error message produced by SAS gives some insight into this when it shows the _N_ counter variable that is stored in the Logical Program Data Vector.



The data step iterates until a flag is encountered which signals that the data step should cease iterating. This point at which this flag is generated depends upon the code you are executing. Typically, the flag is generated when SAS has read the last observation of the input file. There is an implied return statement at the bottom (immediately prior to the run statement) of virtually every data step program that facilitates this iterating process.

DO loops should not be confused with the more common use of DO - END blocks that simply enables the execution of an entire block of code to be conditional and does not imply any iteration of the code contained in the block.



So, by introducing our own loops into our data step programs, we are actually writing loops within loops, Loopy Loo!

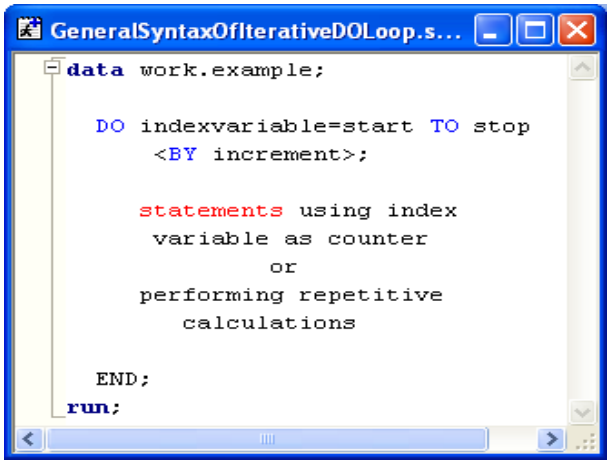
LOOPY DO

This paper is about practical uses of loops; therefore this section concentrates on practical uses of do looping.

ITERATING THROUGH SERIES

The most common use of DO loops within SAS programs makes use of the iterative style of loop that allows the loop to iterate through a series which is defined as part of the loop itself.

The general format of the iterative style of DO loop can be described in the following way:-



Where;

- The indexvariable is NOT automatically dropped from the data set.
- Start, stop and increment are:
  - Set on entry to the loop
  - Cannot be changed during the processing of the loop
  - Numbers, variables or expressions
- The BY clause is optional; the default value of increment is 1.
- Indexvariable is incremented at the bottom of the loop.
- Comparison of the values of indexvariable and stop is done at the top of the loop.

Looping by example

The first group of examples show how loops can have numeric bounds;

Simple Index Incrementation

Log - SimpleDOLoop1.log

```
11    data work.simple1;
12      DO i=1 TO 5;
13        put 'Iteration ' i;
14      END;
15    run;

Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
NOTE: The data set WORK.SIMPLE1 has 1 obser
NOTE: DATA statement used:
      real time             0.03 seconds
      cpu time              0.03 seconds
```

Using the BY in order to decrement by decimals

Log - SimpleDOLoop2.log

```
26    data work.simple2;
27      DO i=1 TO 0 BY -0.1;
28        put 'Index value is ' i;
29      END;
30    run;

Index value is 1
Index value is 0.9
Index value is 0.8
Index value is 0.7
Index value is 0.6
Index value is 0.5
Index value is 0.4
Index value is 0.3
Index value is 0.2
Index value is 0.1
Index value is 1.387779E-16
NOTE: The data set WORK.SIMPLE2 has 1 obs
NOTE: DATA statement used:
      real time             0.03 seconds
```

Or even using equations

Log - SimpleDOLoop3.log

```
31    data work.simple3;
32      A=10;
33      B=7;
34      DO i=1 TO A-B;
35        put 'Index value is ' i;
36      END;
37    run;

Index value is 1
Index value is 2
Index value is 3
NOTE: The data set WORK.SIMPLE3 has 1 obs
NOTE: DATA statement used:
      real time             0.04 seconds
      cpu time              0.03 seconds
```

But we are not constrained to just simple numbers that would be loopy! For instance, we could loop through all of the elements of an array

Log - SimpleDOLoop4.log

```
75    data work.simple4;
76      temp1=10;temp2=20;temp3=30;
77      array temps temp:;
78      DO i=1 TO dim(temps);
79      * DO i=lbound(temps) TO hbound(te
80        put 'Index value is ' i /
81            'and the corresponding '
82            'array value is ' temps{i};
83      END;
84    run;

Index value is 1
and the corresponding array value is 10
Index value is 2
and the corresponding array value is 20
Index value is 3
and the corresponding array value is 30
NOTE: The data set WORK.SIMPLE4 has 1 obs
NOTE: DATA statement used:
      real time             0.06 seconds
```

Use characters

Log - SimpleDOLoop5.log

```
166   data work.simple5;
167     DO day='Mon','Tues','Wed';
168       iteration+1;
169       put 'On iteration ' iteration
170           'the day is ' day;
171     END;
172   run;

On iteration 1 the day is Mon
On iteration 2 the day is Tue
On iteration 3 the day is Wed
NOTE: The data set WORK.SIMPLE5 has 1 obs
NOTE: DATA statement used:
      real time             0.03 seconds
```

Notice that when we use a list of constant values, there is no automatic counter variable available in the loop. If we require one in our code then we must create and increment our own. We could use a list of constant numeric values in exactly the same way.

Or even use dates or times

Log - SimpleDOLoop6.log

```
159   data work.simple6;
160     DO date='01Apr04'd TO '30Apr04'd;
161       iteration+1;
162       put 'On iteration ' iteration
163           'the date is' date worddatx
164     END;
165   run;

On iteration 1 the date is   1 Apr 2004
On iteration 2 the date is   2 Apr 2004
On iteration 3 the date is   3 Apr 2004
On iteration 4 the date is   4 Apr 2004
On iteration 5 the date is   5 Apr 2004
On iteration 6 the date is   6 Apr 2004
```

The same DO loop concepts are applicable in the Macro language also. Of course, the macro language is not constrained to the data step so DO loops in macro can be used to generate SAS code

MacroDOLoops.sas

```
* Delete all data sets located in the
  work library that are prefixed with
  _temp_;

%macro deldata;

  proc contents data=work._all_
                out=work._temp_contents
                noprint;
  run;

  proc sql;
    select distinct(memname),
           count(distinct(memname))
      into : dsnames separated by '*',
           : number
        from work._temp_contents
          where memname like '_TEMP_%';
  quit;

  %if &number>1 %then
  %do;

    proc datasets lib=work;
      delete
    %do i=1 %to &number;
      %scan(&dsnames,&i,*)
    %end;
      ;
    quit;

  %end;
%mend;
%deldata;
```

ITERATING WITH WHILE AND UNTIL

Often we may not know how many times a section of code needs to iterate when we write the code. In such circumstances, we may desire the code to execute UNTIL or WHILE a condition is true.

Both of these scenarios are catered for with the corresponding DO UNTIL and DO WHILE conditional style loops.

GeneralSyntaxOfDOWHILELoop.sas

```
data work.example;
  DO WHILE (expression);
    further SAS statements;
  END;
run;
```

Where;

- The loop continues as long as the condition is true.
- The expression is evaluated at the top of the loop so the code in the loop is not necessarily executed at all.

GeneralSyntaxOfDOUNTILLoop.sas

```
data work.example;
  DO UNTIL (expression);
    further SAS statements;
  END;
run;
```

Where;

- The loop executes until the condition becomes true.
- The expression is evaluated at the bottom of the loop, so the code within the loop must be executed at least once.

Log - DOWhileandDOUntilLoop.log

```
535   data work.dowhile;
536     x=3;
537     do while(x<4);
538       x=x*2;
539       output;
540     end;
541   run;

NOTE: The data set WORK.DOWHILE has 1 obs
NOTE: DATA statement used:
      real time             0.01 seconds
      cpu time              0.01 seconds


542
543   * And the equivalent DO UNTIL;
544
545   data work.dountil;
546     x=3;
547     do until(x>4);
548       x=x*2;
549       output;
550     end;
551   run;

NOTE: The data set WORK.DOUNTIL has 1 obs
NOTE: DATA statement used:
      real time             0.01 seconds
      cpu time              0.00 seconds
```

AVOIDING LOOP LOO

With these conditional style loops, it is very easy to get SAS into an eternal loop where the condition boundary is never crossed. Simply getting the comparison wrong in the above DO UNTIL example puts SAS into an eternal loop where the code only stops executing because the number SAS is accumulating becomes too big to handle, which in SAS is a pretty big number!!

EternalDOUntilLoop.sas

```
data work.dountil;
  x=3;
  do until(x<4);
    x=x*2;
    output;
  end;
run;
```

The SAS NOTE returned in the log reads;

```
NOTE: A number has become too large at line
562 column 8. The number is >1.80E+308 or <-
1.80E+308.
```

But even this loopiness can be handled! It is possible to combine both the iterative and conditional styles of DO loop that we have introduced above. By doing this, it is possible to ensure that even conditional loops will iterate a finite number of times.

Log - BoundedDOUntilLoop.log

```
580   data work.dountil;
581     x=3;
582     do i=1 to 100 until(x<4);
583       x=x*2;
584       output;
585     end;
586   run;

NOTE: The data set WORK.DOUNTIL has 100 o
NOTE: DATA statement used:
      real time              0.01 seconds
      cpu time               0.00 seconds
```

AHH (GA) DO

As programs become more complex, longer and potentially have nested DO – END blocks within them, it becomes more difficult to debug. A common scenario is attempting to debug unbalanced DO – END statements.

There are three hints I offer to help in this situation.

Indenting is something the enhanced editor in SAS for Windows goes a little way to help with; however it is ultimately in the control of the programmer to ensure code is sufficiently indented. There is no formal right or wrong way of laying out SAS code, but common preferences is to indent by either two spaces or one tab for each nested do block.

Commenting code is, sadly, far too often rushed or simply an afterthought. Placing inline comments (not block comments) after END statements can make it easy to see which code blocks are open and closed. The following figure demonstrates both indenting and commenting, which I would strongly recommend to all programmers.

AhhGaDo.sas

```
data work.ahhgado;
  do year = 2001 to 2004;
    do month = 1 to 12;
      do day = 1 to 28;

      end; * Day DO loop;
    end;   * Month DO loop;
  end;     * Year DO loop;
run;
```

Finally a tip for curing after all has failed. I'm sure it happens to most programmers at one time or another and can be the most infuriating thing to correct for the learned eye in complex code:

Log - (Untitled)

```
8         end;    * Month DO loop;
9       end;      * Year DO loop;
10    run;

10    run;
          -
          117
ERROR 117-185: There was 1 unclosed DO block.
```

To help detect which statement is unbalanced use the enhanced editor in SAS for Windows. By holding down the ALT key and pressing [the cursor will jump between corresponding pairs of DO and END statements. If the cursor does not move from the DO or END keyword it has no corresponding statement. Similarly should the cursor jump to a statement which you were not expecting, this is probably the source if your error.

RELATED STATEMENTS

With the inclusion of some additional statements, we not only have the power to determine when to enter or continue executing the code in a DO loop, but we can also control the processing that occurs within each iteration of the loop itself.

LEAVE

The LEAVE statement causes the execution of the current DO loop to stop returning control to the next logical statement in the data step. This is somewhat similar in concept to using conditional DO loops but enables the programmer to use much more complex conditioning within the loop itself.

Log - Leave.log

```
105   data work.leave;
106     x=3;
107     do i=1 to 100;
108       x=x*2;
109       output;
110       if x>4 then LEAVE;
111     end;
112   run;

NOTE: The data set WORK.LEAVE has 1 obser
NOTE: DATA statement used:
      real time              0.00 seconds
      cpu time               0.00 seconds
```

CONTINUE

The CONTINUE statement stops the execution of the current iteration of the DO loop and returns control to the top of the loop.

Continue.sas

```
data work.continue(drop=i);
  infile datalines missover;
  DO i=1 TO 5;
    input name $ 1-14
          dept $ 17-26
          position $ 28-40;
    if position ne 'Manager' then CONTINUE;
    input title $ 1-30;
    output;
  END;
datalines;
John Smith      Sales       Manager
Southern Regional Manager
Samantha Jones  IT          Administrator
Dennis Taylor   Marketing   Designer
Sarah Allan     HR          Manager
HR Manager
Jenny Philips   Sales       Sales Person
run;
```

DONE OVER

In the examples above I dealt with looping of arrays with the DO loop.

Another approach is to use the DO OVER loop.

Done with Do Over.sas

```
data work.over;
  DO i = 1 TO 30;
    ResultA = ranuni(0)*100;
    ResultB = ranuni(0)*50;
    ResultC = ranuni(0)*25;
    output;
  END;
run;

data work.over1;
  set  work.over;
  array results result:;

  DO OVER results;
    if results<10 then put results=;
  END;
run;
```

The DO OVER method iterates sequentially for each element in the array. Although not documented in SAS since 6.12, the DO OVER is still supported in V8 and V9.

CONCLUSION

In conclusion, DO loops are an often used but also little understood. I hope that the information that has been provided in this paper enables you to make more, and hopefully better, use of DO loops in your code.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact Amadeus at:

Company:	Amadeus Software Limited
Address:	Mulberry House, 9 Church Green, Witney, Oxon OX28 4AZ
Work Phone:	+44 (0) 1993 848010
Email:	info@amadeus.co.uk
Web:	www.amadeus.co.uk