Advanced DATA Step Programming Techniques

WUSS Virtual Class - May 6, 2026

Originally Developed By:
Arthur L. Carpenter
California Occidental Consultants

Revised and Presented By:
Joshua M. Horstman
Nested Loop Consulting

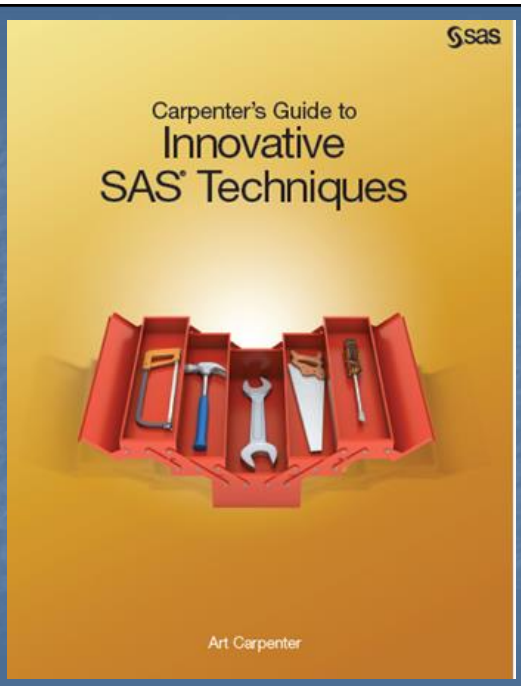
1

The material in this course is based on portions of the book:

Carpenter's Guide to Innovative SAS® Techniques
by Art Carpenter

Published by SAS Press

All material in this course was used with permission of the author.



2

Course Organization

(1.1.1)

- Course section numbering differs from book chapters.
- Numbers in parentheses in upper right of slide refer to corresponding chapter and section in the book.

3

3

Course Overview

- 1. Working with Data
- 2. Transposing Data
- 3. Working Across Observations
- 4. Using DATA Step Component Objects
- 5. Doing More with DATA Step Functions
- 6. Doing More with the SET Statement
- 7. Working with Multiple Incoming Data Sets

4

4

Chapter 1

WORKING WITH DATA

5

5

Chapter 1 Overview

- 1.1 Data Set Options
- 1.2 Evaluating Expressions
- 1.3 Shorthand Variable Naming
- 1.4 Quotes within Quotes

6

6

1.1 Data Set Options

(2.1)

Data set options modify how a data set is either read or written.

Options are placed in parentheses immediately following the name of the data set.

```
data nameonly(keep= lname fname ssn);
   set advrpt.demog;
   . . .  statements not shown . . . .
   run;
```

7

7

1.1.1 KEEP / DROP / RENAME

(2.1.3)

- KEEP, DROP, and RENAME are available as both data set options and DATA step statements.
- When multiple data sets are created or read, DATA step statements apply to all data sets, while data set options can be applied to specific data sets.
- As a general rule, when you have a choice, data set options are preferred over statements, as the data set options give you more control and the code is generally clearer.

8

8

(2.1.3)

1.1.1 KEEP / DROP / RENAME

The KEEP statement variable list is applied to the new outgoing data set.

The IF statement is executed after the entire observation is read and loaded into the Program Data Vector (PDV).

```
data labs;
   set advrpt.lab_chemistry;
   keep subject visit labdt;
   if sodium>'142';
   run;
```

Using the KEEP statement here is exactly the same as specifying the KEEP= option on the data set in the DATA statement.

9

9

(2.1.3)

1.1.1 KEEP / DROP / RENAME

The KEEP= option on the SET statement is applied before the PDV is built.

```
data labs(keep=subject visit labdt);
   set advrpt.lab_chemistry
                (keep=subject visit labdt sodium
                 where=(sodium>'142'));
   run;
```

Only those variables listed on the incoming KEEP= option will be:
- Read from the incoming data set
- Included on the PDV
- Available for use with the WHERE=.

Only those variable listed on the outgoing KEEP= option will be:
- Written to the output data set

10

10

(2.1.3)

1.1.1 KEEP / DROP / RENAME

The RENAME option allows you to change the name of a variable either as it is read or as it is written.

```
data labs(keep=subject visit labdate);
   set advrpt.lab_chemistry
                    (rename=(labdt=labdate));
   if sodium>'142';
   run;
```

Placing the RENAME= on the SET statement, causes the name to be changed before it is written to the PDV.

11

11

(2.1.3)

1.1.1 KEEP / DROP / RENAME

When the RENAME= and KEEP= are both used on the same data set, it is important to understand which is applied first.

```
data labs(keep=subject visit labdate);
   set advrpt.lab_chemistry(rename=(labdt=labdate)
                       keep=subject visit labdt sodium
                       );
   if sodium>'142';
   run;
```

Since the KEEP= is applied before the RENAME= on the SET statement, the original variable name is used on the incoming KEEP= variable list.

12

12

(2.1.4)

1.1.2 FIRSTOBS and OBS

- FIRSTOBS and OBS data set options can be used separately or together to limit which observations are read.
  - FIRSTOBS — Specifies the number of the first observation to be read
  - OBS — Specifies the number of the last observation to be read
- When used together, they operate independently.
  (i.e. OBS still counts from observation 1, regardless of FIRSTOBS)

13

13

(2.1.4)

1.1.2 FIRSTOBS and OBS

```
proc print data=sashelp.class(firstobs=4 obs=6);
  run;
```

Obs	Name	Sex	Age	Height	Weight
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0

When used with a WHERE clause:

- WHERE clause is applied first.
- Selection of observations is based on resulting subset.

14

14

(2.1.1)

1.1.3 REPLACE and REPEMPTY

The REPLACE and REPEMPTY data set options control the conditions under which a data set is replaced.

REPLACE — Determines whether replacement of a *permanent* data set is allowed. Values are NO and YES (default). (Overrides the system option of the same name.)

REPEMPTY — Determines whether or not an empty data set can overwrite an existing data set. Values are NO and YES (default).

15

15

(2.1.1)

1.1.3 REPLACE and REPEMPTY

Normally we want to be able to replace permanent data sets, unless the new version is empty.

```
data advrpt.class(replace=yes repempty=no);
   set sashelp.class;
   where age > 25;
   run;
```

This condition is never true.

In this example ADVRPT.CLASS will not be replaced. ADVRPT.CLASS can only be replaced with a non-empty version.

16

16

(2.2)

1.2 Evaluating Expressions

- SAS expressions are combinations of operators, constants, functions, and variables.
- Expressions are used in various ways in both DATA steps and PROC steps (e.g. assignment statements, comparisons, WHERE clauses, etc.)
- Expressions are evaluated according to a set of rules.
- Understanding the rules opens up new programming options!

17

17

(2.2.1)

1.2.1 Operator Hierarchy

- Operators are like "verbs" that tell SAS what to do with constants and variables.
- To avoid confusion and ambiguity, operators are applied according to a hierarchy.
- The hierarchy is formed by seven groups of operators.
- Within a group, operators of equal rank are applied from left to right (except Group 1 which is applied right to left).

18

18

(2.2.1)

1.2.1 Operator Hierarchy

Group	Operators
Parentheses	Operations within parentheses are performed first
Group 1 (performed right to left)	Exponentiation (**) Prefix operators, such as, positive (+), negative (-), and *negation* Minimum (MIN,><) and maximum (MAX, <>)
Group 2	Multiplication (*) and division (/)
Group3	Addition (+) and subtraction (-)
Group 4	Concatenation (||)
Group 5	Comparisons, such as equal (=) and less than (<)
Group 6	AND - Boolean comparison (&)
Group 7	OR - Boolean comparison (|)

19

19

(2.2.3)

1.2.2 Operators in Assignment Statements

Understanding the hierarchy allows us to expand our perception of what an expression *should* contain:

```
season = 1*(1 le month(dob) le 3)
       + 2*(4 le month(dob) le 6)
       + 3*(7 le month(dob) le 9)
       + 4*(10 le month(dob) le 12);
```

Logical comparisons produce a numeric 0 or 1 result.

```
season = 1*(0)
       + 2*(1)
       + 3*(0)
       + 4*(0);
```

For a birthday in May.

```
season = 2;
```

20

20

(2.2.1)

1.2.2 Operators in Assignment Statements

These two expressions are not exactly the same!
What happens when DOB is missing?

```
season = 1*(1 le month(dob) le 3)
       + 2*(4 le month(dob) le 6)
       + 3*(7 le month(dob) le 9)
       + 4*(10 le month(dob) le 12);
```

```
season= ceil(month(dob)/3);
```

21

21

(2.2.3)

1.2.2 Operators in Assignment Statements

Complex comparisons can be used.

```
if sex = 'M' and year(dob) >  1949 then group=1;
else if sex = 'M' and year(dob) le  1949 then group=2;
else if sex = 'F' and year(dob) >  1949 then group=3;
else if sex = 'F' and year(dob) le  1949 then group=4;
```

```
group = 1*(sex = 'M' and year(dob) >  1949)
      + 2*(sex = 'M' and year(dob) le  1949)
      + 3*(sex = 'F' and year(dob) >  1949)
      + 4*(sex = 'F' and year(dob) le  1949);
```

Use it whenever the result is numeric!

22

22

(2.2.3)

1.2.2 Operators in Assignment Statements

Use this approach to generate binary (0 or 1) flags.

```
data flags;
   set advrpt.demog (keep=lname fname dob sex);
   if year(dob) >  1949 then boomer=1;
   else boomer=0;
   boomer2 = year(dob) >  1949;
   boomer3 = ifn(year(dob) >  1949, 1, 0);
   run;
```

More about the amazing IFN function later in the course!

Assignment statements tend to be faster than IF-THEN/ELSE statements.

23

23

(2.2.3)

1.2.2 Operators in Assignment Statements

Checking for an item in a list.

Here we wish to determine if the value 7 is found in one or more of the variables x1 through x4.

```
data _null_;
input x1-x4;
array a {*} x1-x4;
flag1 = (7 in a);
flag2 = ^^whichn(7,of x:);
put flag1= flag2=;
datalines;
1 2 3 4
5 6 7 8
run;
```

Double negation builds 0,1 flag.

for the second data line x3 = 7

WHICHN returns a 3

^ 3 returns a 0

^ 0 returns a 1

24

24

(2.2.4)

1.2.3 Compound Expressions

Understand how compound expressions are evaluated.

```
if 13 le edu le 16;
```

This compound inequality is interpreted as two distinct inequalities joined with an AND.

```
if (13 le edu) and (edu le 16);
```

```
where (13 le edu) le 16 ;
```

Misplacing the parentheses changes the result.

(13 le edu) will always be either 0 or 1, which will always be less than 16. This WHERE is always true.

25

25

(2.2.4)

1.2.3 Compound Expressions

Compound macro expressions are evaluated differently!

```
%let x = 5;
%let y = 4;
%let z = 3;
%if &x lt &y lt &z %then %put &x < &y < &z;
```

Evaluates as true!
5 < 4 < 3

Evaluated as if it had been coded as:

```
%if (&x lt &y) lt &z %then %put &x < &y < &z;
```

This might have been what you meant:

```
%if (&x lt &y) & (&y lt &z) %then %put &x < &y < &z;
```

26

26

(2.2.2)

1.2.4 The Colon Comparison Modifier

The colon can be used as an operator modifier when character values are being compared.

It always compares strings of equal length.

```
data Mar;
   set advrpt.demog (keep=lname fname);
   if lname =: 'Mar';
   run;
```

All last names beginning with 'Mar'.

Note that 'Ma' would also be selected!

```
if 'Mar' =: lname;
```

Order does not matter.

```
where lname in:('Me', 'Mar', 'Adams');
```

Length can vary.

27

27

(2.2.5)

1.2.5 MIN and MAX Operators (<> , ><)

The MIN (><) and MAX (<>) operators return the minimum or maximum, respectively, of exactly two values.

Unfortunately, they have some major problems:

1. Mnemonics handled differently in IF and WHERE
2. Negative values handled after evaluation in IF

```
if -2 = (-5 min 2);      *This is TRUE;
where -2 = (-5 min 2);   *This is FALSE;
```

Recommendation: Avoid these operators and their mnemonics. Use the MIN and MAX functions instead.

28

28

(2.2.6)

1.2.6 Boolean Transformations

Transform a numeric value to Boolean (true / false) using double negation

```
x = ^^dob;
```

Converts missing to 0

Two ways to replace missing values with zero:

```
z = coalesce(dob,0);
```

COALESCE returns first non-missing value

```
y = sum(dob,0);
```

SUM ignores missing values

29

29

(2.6)

1.3 Shorthand Variable Naming

- SAS data sets often contain many variables
- Listing variable names in code can be cumbersome
- SAS provides some shortcuts
- Use wherever a list of variables is expected
  - VAR, KEEP, DROP, and ARRAY statements
  - Some DATA step functions

30

30

(2.6.1)

1.3.1 Variable List with a Single Dash

Use single dash to include variables with a common prefix and a numeric suffix.

```
array vis {10} visit1 - visit10;
```
Can create variables.

```
keep visit1 - visit10;
```
Undefined variables cause an error.

```
m = max(of visit1 - visit10);
```
OF operator required.

31

31

(2.6.1)

1.3.2 Variable List with a Double Dash

When the order of the variables on the PDV is known, you can use the double dash to specify the list.

```
keep lname -- symp;
```
list includes the end point variables

```
keep lname -character- symp;
```
only character variables

```
keep dob -numeric- ht;
```
only numeric variables

Be careful! You have to know the variable order.

32

32

(2.6.1)

1.3.3 Colon Operator

Use the colon operator to specify variables named with a common prefix.

```
array vis {10} visit:;
```
All variables whose name starts with VISIT.

```
drop _:;
```
All variables whose name starts with an underscore.

33

33

(2.6.1)

1.3.4 Special Name Lists

Three special name lists are available to address variables by their type:

CHARACTER	All character variables
NUMERIC	All numeric variables
ALL	All variables on the PDV

Since each of these lists pertain to the current list of variables, they will not create variables. In each case the resulting list of variables will be in the same order as they are on the Program Data Vector.

34

34

(2.6.3)

1.4 Quotes Within Quotes

- The quote mark is used to identify constant text to the parser.
- Sometimes that quoted string of constant text will itself contain quotes.
- This can be a problem if macro variables are involved.

35

35

(2.6.3)

1.4 Quotes Within Quotes

The following statement has a quoted string within a quoted string, and executes successfully.

```
myhtml = '<a href="myloc/myfile.html">';
```

What if we need to embed a macro variable?

```
%let temp = myloc;

myhtml = '<a href="&temp/myfile.html">';
```

Because of the single quotes, &TEMP cannot be resolved.

36

36

(2.6.3)

1.4 Quotes Within Quotes

We can delay the parser from recognizing the quotes by doubling them.

```
myhtml = "<a href=""&temp/myfile.html"">";
```

One double quote | Two double quotes | Two double quotes | One double quote

Alternatively, use macro quoting functions to temporarily mask the single quotes until the macro variable has been resolved.

```
myhtml = %unquote(%bquote(')
              <a href="&temp/myfile.html">%bquote('));
```

37

37

Chapter 2

TRANSPOSING DATA

38

38

(2.4)

Transposing Data

- Many SAS procedures are designed for normalized data
  - "Tall and narrow" (one response value per row)
  - Classification variables identify individual rows.
- Sometimes we have (or want) data in non-normal form
  - "Short and wide" (many response values in a row)
  - Separate columns for each level of classification variables.
- Multiple ways to convert between these structures
  - PROC TRANSPOSE is the workhorse
  - DATA step can provide more flexibility

39

39

Chapter 2 Overview

- 2.1 Transposing Rows to Columns
- 2.2 Transposing Columns to Rows

40

40

(2.4.2)

Transposing Data

Normalized

Obs	SUBJECT	VISIT	sodium
1	208	1	13.7
2	208	2	14.1
3	208	4	14.1
4	208	5	14.1
5	208	6	13.9
6	208	7	13.9
7	208	8	14.0
8	208	9	14.0
9	208	10	14.0
10	209	1	14.0

. . . . portions of the table are not shown . . . .

Non-normal

Obs	SUBJECT	visit1	visit2	visit3	visit4	visit5	visit6	visit7	visit8	visit9	visit10	visit11	visit12	visit13	visit14	visit15	visit16
1	208	13.7	14.1	.	14.1	14.1	13.9	13.9	14	14	14.0	.	.	.	.	.	.
2	209	14.0	14.0	.	13.9	14.2	14.5	13.8	14	.	13.8	14	14.1	14.2	14.1	14	14.1

. . . . portions of the table are not shown . . . .

41

41

(2.4.2)

2.1 Transposing Rows to Columns

To transpose rows to columns, we read all input records for a BY group and then write only a single output record.

```
data lab_nonnormal(keep=subject visit1-visit16);
   set lab_chemistry(keep=subject visit sodium);
   by subject;
   retain visit1-visit16 ;
   array visits {16} visit1-visit16;
   if first.subject then do i = 1 to 16;
      visits{i} = .;
   end;
   visits{visit} = sodium;
   if last.subject then output lab_nonnormal;
   run;
```

SUBJECT remains a CLASS variable.

Visits will become new columns

All columns are initialized to missing for each subject.

Assign the value

OUTPUT is conditional. We get only one row per subject.

42

42

(2.4.2)

2.1 Transposing Rows to Columns

The initialization step can be simplified using the CALL MISSING routine and some shorthand OF notation.

```
data lab_nonnormal(keep=subject visit1-visit16);
   set lab_chemistry(keep=subject visit sodium);
   by subject;
   retain visit1-visit16 ;
   array visits {16} visit1-visit16;
   if first.subject then do i = 1 to 16;
      visits{i} = .;
   end;
   visits{visit} = sodium;
   if last.subject then output lab_nonnormal;
   run;
```

call missing(of visits{*});

43

43

(2.4.2)

2.2 Transposing Columns to Rows

To transpose columns to rows, we write multiple output records for each input record.

```
data lab_normal(keep=subject visit sodium);
   set lab_nonnormal(keep=subject visit:);
   by subject;
   array visits {16} visit1-visit16;
   do visit = 1 to 16;
      sodium = visits{visit};
      output lab_normal;
   end;
   run;
```

VISITS already exist as columns.

Use visit number as the array index.

OUTPUT statement is inside the DO loop, resulting in one row per subject per visit.

44

44

(2.4.2)

2.2 Transposing Columns to Rows

Obs	SUBJECT	visit1	visit2	visit3	visit4	visit5	visit6	visit7	visit8	visit9
1	208	13.7	14.1	.	14.1	14.1	13.9	13.9	14	14
2	209	14.0	14.0	.	13.9	14.2	14.5	13.8	14	.

. . . . portions of the table are not sho

After transposing from rows to columns and back to rows, resulting data set contains class variable combinations that did not previously exist.

Obs	SUBJECT	visit	sodium
1	208	1	13.7
2	208	2	14.1
3	208	3	.
4	208	4	14.1
5	208	5	14.1
6	208	6	13.9
7	208	7	13.9
8	208	8	14.0
9	208	9	14.0
10	208	10	14.0
11	208	11	.
12	208	12	.
13	208	13	.
14	208	14	.
15	208	15	.
16	208	16	.
17	209	1	14.0

. . . . portions of the table are not shown . . . .

45

45

Chapter 3

WORKING ACROSS OBSERVATIONS

46

46

(3.1)

Working Across Observations

- SAS reads one observation at a time into the PDV.
- DATA step statements interact with values on the PDV.
- Generally, only values from the current observation are available for use.
- To use values from previous or subsequent records, we must do something extra.
- As usual, there are several ways to do this!

47

47

Chapter 3 Overview

- 3.1 BY Group Processing
- 3.2 Transposing to Arrays
- 3.3 Looking Back with the LAG Function
- 3.4 Looking Ahead using MERGE
- 3.5 Looking Ahead using a Double SET
- 3.6 Looking Back using a Double SET
- 3.7 Building a FIFO Stack

48

48

(3.1.1)

3.1 BY Group Processing

- Single-observation processing can be limiting when we need to work with data in groups.
- BY statement can be used to define groups.
- We still need to detect and handle group boundaries.
- BY statement automatically creates temporary variables that identify group boundaries.
  - Two temporary numeric variables per BY variable
  - Named FIRST.*varname* and LAST.*varname*
  - Values are either 1 (true) or 0 (false)

49

49

(3.1.1)

3.1 BY Group Processing

FIRST.*varname* and LAST.*varname* are created through the use of a BY statement.

```
by region clinnum;
```

Obs	REGION	CLINNUM	SSN	First Region	Last Region	First Clin	Last Clin
1	1	011234	345751123	1	0	1	0
2	1	011234	479451123	0	0	0	1
3	1	014321	075312468	0	0	1	0
4	1	014321	190473627	0	1	0	1
5	10	107211	315674321	1	0	1	0
6	10	107211	471094671	0	0	0	1
7	10	108531	366781237	0	0	1	0
8	10	108531	476587764	0	0	0	0
9	10	108531	563457897	0	0	0	0
10	10	108531	743787764	0	1	0	1
11	2	023910	066425632	1	0	1	0
12	2	023910	075345932	0	0	0	0
13	2	023910	091550932	0	0	0	1

.. . . . Portions of the output table not shown . . . .

50

50

(3.1.1)

3.1 BY Group Processing

Example: Data set contains one row per patient. We wish to count clinics and patient visits within each region.

```
data counter(keep=region clincnt patcnt);
   set regions(keep=region clinnum);
   by region clinnum;
   if first.region then do;
      clincnt=0;
      patcnt=0;
   end;

   if first.clinnum then clincnt + 1;
   patcnt+1;

   if last.region then output;
   run;
```

Input data set must already be sorted by region and clinnum.

Initialize counters to zero when we start a new region.

Increment counters as individual records are encountered.

Write totals to output data set only after all records for a given region have been read.

51

51

(3.1.1)

3.1 BY Group Processing

To simplify the code, the statement…

```
if first.clinnum then clincnt + 1;
```

could have been replaced by this statement:

```
clincnt + first.clinnum;
```

52

52

(3.1.1)

3.2 Transposing to Arrays

Arrays can be useful when we need to hold values to perform complex operations across multiple rows of data.

Example: Lab data contains one row per subject per visit. We wish to determine the average number of days between visits.

(Part 1/3 of code)

```
data labvisits(keep=subject count meanlength);
  set advrpt.lab_chemistry;
  by subject;

  array Vdate {16} _temporary_;
  retain totaldays count 0;
```

Define a temporary array to hold values of interest. We assume an upper bound of 16 visits per subject.

53

53

(3.1.2)

3.2 Transposing to Arrays

(Part 2/3 of code)

```
 if first.subject then do;
   totaldays=0;
   count = 0;
   do i = 1 to 16;
     vdate{i}=.;
   end;
 end;

 vdate{visit} = labdt;
```

Initialize counters and clear array each time we begin a new subject.

The date for each visit is stored in the corresponding location in the array.

54

54

(3.1.2)

3.2 Transposing to Arrays

(Part 3/3 of code)

```
if last.subject then do;
   do i = 1 to 15;
      between = vdate{i+1}-vdate{i};
      if between ne . then do;
         totaldays = totaldays+between;
         count = count+1;
      end;
   end;
   meanlength = totaldays/count;
   output;
end;
run;
```

When the last record for a subject has been read, the array now holds all values needed to proceed with calculations.

Write the mean for this subject to the output data set.

55

55

(3.1.2)

3.2 Transposing to Arrays

As before, the initialization step can be simplified using the CALL MISSING routine and some shorthand OF notation.

Replace this…

```
do i = 1 to 16;
   vdate{i}=.;
end;
```

With this…

```
call missing(of vdate{*});
```

The MISSING routine assigns missing values to all its arguments, regardless of type.

56

56

(3.1.3)

3.3 Looking Back with the LAG Function

- LAG stores a value in a queue and returns the previous value.
- Commonly used to read values from previous observations.
- LAG is actually a family of functions: LAG1, LAG2, LAG3, etc.
  - LAG is an alias for LAG1.
  - LAGn creates queue of length n.
- Queue is initialized to missing, so first n calls return missing.

57

57

(3.1.3)

3.3 Looking Back with the LAG Function

Example: determine the number of days since the previous visit.

```
data labvisits(keep=subject visit lagvisit
                           interval lagdate labdt);
   set labdates;
   by subject;

   lagvisit= lag(visit);
   lagdate = lag(labdt);

   if not first.subject then do;
      interval = labdt - lagdate;
      if interval ne . then output;
   end;
   format lagdate mmddyy10.;
   run;
```

WARNING: Do not put the LAG function here inside the IF block. It needs to execute unconditionally.

58

58

(3.1.4)

3.4 Looking Ahead Using MERGE

We can look ahead using a one-to-one merge (no BY statement). Example: Determine the number of days until the next laboratory date (LABDT).

```
options mergenoby=nowarn ;

data nextvisit(keep=subject visit labdt days2nextvisit);
   merge labdates(keep=subject visit labdt)
         labdates(firstobs=2
                  keep=subject labdt
                  rename=(subject=nextsubj labdt=nextdt));
   Days2NextVisit = ifn(subject=nextsubj,nextdt-labdt, ., .);
   run;
```

Turn off the no BY statement warning.

Data set merged with itself. Right-hand side of merge starts reading from second observation.

Time to the next visit is a simple subtraction.

59

59

(3.1.5)

3.5 Looking Ahead Using a Double SET

We can look ahead using a second SET statement. Example: Determine the number of days until the next laboratory date (LABDT).

First SET statement is the primary read.

```
data nextvisit(keep=subject visit labdt days2nextvisit);
   set labdates(keep=subject visit labdt)
         end=lastlab;
   if not lastlab then do;
      set labdates(firstobs=2
                   keep=subject labdt
                   rename=(subject=nextsubj labdt=nextdt));
      Days2NextVisit = ifn(subject=nextsubj,nextdt-labdt, ., .);
   end;
   run;
```

Second SET looks ahead.

Each SET statement maintains a separate pointer into the data set.

Time to the next visit is a simple subtraction.

60

60

(3.1.6)

3.6 Looking Back Using a Double SET

Example: Find all lab visits that fall between the first and second POTASSIUM reading that meets or exceeds 4.2.

Problem: We can't identify the second peak (if it exists) until we've already passed the records of interest.

Solution: Use two SET statements.

1. The first SET steps through the observations and notes the locations of the peak values.
2. The second is used to read the observations between the peaks.

61

61

(3.1.6)

3.6 Looking Back Using a Double SET

(Part 1/2 of code)

```
data BetweenPeaks(keep=subject visit labdt potassium);
  set labdates(keep=subject labdt potassium);
  by subject labdt;
  retain firstloc .
         found ' ';
  obscnt+1;
  if first.subject then do;
    found=' ';
    firstloc=.;
  end;
```

Define some bookkeeping variables that will be retained across observations.

Reset the bookkeeping variables when starting a new subject.

62

62

(3.1.6)

3.6 Looking Back Using a Double SET

(Part 2/2 of code)

```
if found=' ' and potassium ge 4.2 then do;
    if firstloc=. then firstloc=obscnt;
    else do;
      * This is the second find, write list;
      found='x';
      do point=firstloc to obscnt;
        set labdates(keep= subject visit labdt potassium)
            point=point;
        output betweenpeaks;
      end;
    end;
  end;
  run;
```

This IF block will execute when we hit the first or second peak.

First peak: simply note location.

Second peak: Read all observations from the first peak (FIRSTLOC) to the current peak (OBSCNT)

Output each value between the peaks.

63

63

(3.1.7)

3.7 Building a FIFO Stack

- Stacks are another technique for processing across a series of observations.
- Helpful when calculating certain statistics, such as running averages.
- A stack is a collection of values which rotate through according to entrance and exit rules.
- Two types of stacks:
  - FIFO (First-In-First-Out)
  - LIFO (Last-In-First-Out)

64

64

(3.1.7)

3.7 Building a FIFO Stack

Example: Calculate a three-day moving average of potassium levels for each subject.

(Part 1/2 of code)

```
data Average(keep=subject visit labdt potassium Avg3day);
   set labdates;
   by subject;

   * dimension of array is number of items to be averaged;
   retain visitcnt .;
   array stack {0:2}  _temporary_;
```

Index the array starting at 0 to facilitate easy computation of index using MOD function.

65

65

(3.1.7)

3.7 Building a FIFO Stack

(Part 2/2 of code)

```
if first.subject then do;
   call missing(of stack{*});
   visitcnt=0;
end;
visitcnt+1;
index = mod(visitcnt,3);
stack{index} = potassium;
avg3day = mean(of stack{*});
run;
```

Clear the stack when starting a new subject.

The MOD function determines the array index.

Current value always replaces the value from 3 records earlier (FIFO).

Number of items in the stack.

66

66

Chapter 4

USING DATA STEP COMPONENT OBJECTS

67

67

(3.3)

Data Step Component Objects

- Object-oriented programming (OOP)
- Unlike anything else in the DATA step
- Compiled within the DATA step
- Perform operations via built-in methods
- Advantages:
  - Performance (especially with large data)
  - Can accomplish tasks which would otherwise be difficult or impossible

68

68

(3.3)

Data Step Component Objects

HASH

- Stores a data table in memory
- Highly efficient read/write access
- Random access based on key variables

HITER (hash iterator)

- Use in conjunction with HASH object
- Steps through rows sequentially

69

69

Chapter 4 Overview

- 4.1 Instantiating a Component Object
- 4.2 Using an Object's Methods
- 4.3 Sorting using a Hash Table
- 4.4 Stepping through a Hash Table

70

70

(3.3.1)

4.1 Instantiating a Component Object

A component object is created (or instantiated) by the DECLARE (or DCL) statement in the DATA step.

Each component object is assigned a name, which becomes a variable on the PDV.

Name is established on the DECLARE statement.

```
declare hash hashname();
```

```
dcl hash hashname;
hashname = _new_ hash();
```

The object is declared on either one or two statements.

71

71

(3.3.1)

4.1 Instantiating a Component Object

When the object is created (declared), attributes can be controlled through arguments known as constructors.

Constructors appear in parentheses, followed by a colon:

dataset:	name of SAS data set to load into hash object
hashexp:	exponent that determines the number of key locations (slots)
ordered:	determines how the key variables are to be ordered in the hash table

72

72

(3.3.2)

4.2 Using Methods

Method calls must refer to their corresponding object using the dot notation:

```
declare hash hashname();
  hashname.definekey('subject', 'visit');
  hashname.definedata('subject','visit','labdt') ;
  hashname.definedone() ;
```

Methods used to define the object follow the DECLARE statement and include:

definekey	list of variables forming the primary key
definedata	list of data set variables (or use ALL:"YES")
definedone	closes the object definition

73

73

(3.3.2)

4.2 Using Methods

During DATA step execution, methods are used to read and write to the object. A few of these methods include:

add	add the specified data on the PDV to the object
delete	delete the hash or hash iterator object
find	retrieve information from the object based on the values of the key variables
output	write the object's contents to a SAS data set
replace	write data from the PDV to an object, matching key variables are replaced

There are many other methods. Refer to SAS Component Objects: Refererence.

74

74

(3.3.3)

4.3 Sorting Using a Hash Table

Because the hash object can be ordered by keys, it can be used to sort a table. The following is equivalent to a SORT with NODUPKEY.

```
data _null_;
  if 0 then set advrpt.demog(keep=clinnum subject
                             lname fname dob);
  declare hash clin (dataset:'advrpt.demog', ordered:'Y');
    clin.definekey ('clinnum','subject');
    clin.definedata ('clinnum','subject','lname','fname','dob');
    clin.definedone ();
  clin.output(dataset:'clinlist');
  stop;
  run;
```

SET adds variable attributes to the PDV but will never execute.

Load data into the hash object

Write the CLIN object to the WORK.CLINLIST data set.

STOP statement closes the implied loop created by the SET statement.

75

75

(3.3.4)

4.4 Stepping Through a Hash Table

Two approaches to process contents of hash table item by item.

- Use FIND method on HASH object with key variables to retrieve each item of interest.
- Use HITER (hash iterator) object and its methods to move forwards and backwards through the hash table.

76

76

(3.3.4)

4.4.1 Using the FIND Method

The FIND method uses values of key variables in the PDV to retrieve a record from the hash table.

Example: Find all drugs that the patient started taking within 5 days of an adverse event.

(Part 1/3 of code)

```
data drugEvents(keep=subject medstdt drug aestdt aedesc sev);
   declare hash meds(ordered:'Y') ;
      meds.definekey ('subject', 'counter');
      meds.definedata('subject', 'medstdt','drug') ;
      meds.definedone () ;
```

Define the MEDS hash object to hold the medication data.

77

77

(3.3.4)

4.4.1 Using the FIND Method

Here, we load the hash object using a SET statement and the ADD method instead of the DATASET: constructor.

(Part 2/3 of code)

Input data set must be sorted since we used a BY statement.

```
* Load the medication data into the hash object;
do until(allmed);
   set advrpt.conmed(keep=subject medstdt drug) end=allmed;
   by subject;
   if first.subject then counter=0;
   counter+1;
   rc=meds.add();
end;
```

Read each record with the SET statement.

Create a new counter variable to number records within each subject.

Add each record to the hash object.

78

78

(3.3.4)

4.4.1 Using the FIND Method

(Part 3/3 of code)

```
do until(allae);
     set advrpt.ae(keep=subject aedesc aestdt sev) end=allae;
     counter=1;
     rc=meds.find();
     do while(rc=0);
        * Was this drug started within 5 days of the AE?;
        if (0 le aestdt - medstdt lt 5) then output drugevents;
        counter+1;
        rc=meds.find()
     end;
  end;
  stop;
  run;
```

Read each adverse event

Read the first medication record for this SUBJECT from the MEDS object.

Read the next medication record for this SUBJECT from the MEDS object.

79

79

(3.3.4)

4.4.2 Using the Hash Iterator Object

The hash iterator object, HITER, provides methods used to step through a hash object.

Each HITER object is tied to a specific HASH object.

(Part 1/3 of code)

```
data drugEvents(keep=subject medstdt drug aestdt aedesc sev);
   declare hash meds(ordered:'Y') ;
      declare hiter medsiter('meds');
      meds.definekey ('subj', 'counter');
      meds.definedata('subj', 'medstdt','drug','counter') ;
      meds.definedone () ;
```

The hash iterator object MEDSITER is tied to the MEDS object.

80

80

(3.3.4)

4.4.2 Using the Hash Iterator Object

As before, the medication data is loaded into the MEDS hash object using the ADD method.

(Part 2/3 of code)

```
do until(allmed);
      set advrpt.conmed(keep=subject medstdt drug) end=allmed;
      by subject;
      if first.subject then do;
         counter=0;
         subj=subject;
      end;
      counter+1;
      rc=meds.add();
   end;
```

81

81

(3.3.4)

4.4.2 Using the Hash Iterator Object

For each adverse event record, step through the MEDS object to find entries matching the key variable values.
(Part 3/3 of code)

```
do until(allae);
      set advrpt.ae(keep=subject aedesc aestdt sev) end=allae;
      rc = medsiter.first();
      do until(rc);
         * Was this drug started within 5 days of the AE?;
         if subj=subject & 0<=aestdt-medstdt<5 then output drugevents;
         if subj gt subject then leave;
         rc=medsiter.next();
      end;
   end;
   stop;
   run;
```

Since FIRST does not take the key variable values into consideration, except for the first patient, we must cycle through earlier patients until we get to the patient of interest.

82

Chapter 5

DOING MORE WITH DATA STEP FUNCTIONS

83

83

(3.6)

Doing More with DATA Step Functions

- As of SAS 9.4, nearly 500 built-in functions are available in the DATA step.
- Using functions:
  - Saves time
  - Shortens and simplifies code
  - Reduces errors
- Functions are an essential part of the SAS programmer's toolbox. Become familiar with them and review regularly!

84

84

Chapter 5 Overview

- 5.1 String Concatenation Functions
- 5.2 COMPRESS Function
- 5.3 ANY and NOT Families
- 5.4 INDEX and FIND Families
- 5.5 TRANWRD Function
- 5.6 SCAN Function
- 5.7 COUNTW Function
- 5.8 IFN and IFC Functions
- 5.9 WHICHN and WHICHC Functions
- 5.10 Maximum and Minimum Value Functions

85

85

(3.6.3)

5.1 String Concatenation Functions

The CAT family of functions allow us to perform concatenation operations without the concatenation operator (||):

CAT same as ||, it leaves leading and trailing blanks

CATS removes leading and trailing blanks

CATT removes only trailing blanks

CATX removes leading and trailing blanks and adds the separator of your choice between strings (Duplicate consecutive delimiters caused by missing strings are automatically suppressed!)

```
_c5_  = cats(put(_c3_,6.2),' (',put(_c4_,7.3),')');
```

```
mylist  = catx(', ',item1,item2,item3);
```

86

86

(3.6.7)

5.2 COMPRESS Function

The COMPRESS function can remove much more than just blanks from a string.

```
string1 = 'ABCDEABCDE';
string2 = compress(string1,'CAE','k');
```

The 'K' modifier causes COMPRESS to keep the specified characters and remove everything else.

```
string1=ABCDEABCDE
string2=ACEACE
```

87

87

(3.6.7)

5.2 COMPRESS Function

Count the number of lines of code in a SAS program.

```
filename code "c:\sascode\ABC.sas";
data _null_;
  infile code truncover;
  input ;
  justsemi = compress(_infile_,';','k');
  cnt+index(justsemi,';')*length(justsemi);
  call symputx('cnt',cnt);
  run;
  %put line count is: &cnt;
```

Remove everything except semi-colons.

Count semi-colons using INDEX.

88

88

(3.6.1)

5.3 The ANY and NOT Families

The ANY family of functions return the position of the first occurrence of certain target characters within a string (or zero if not found).

The NOT functions are analogous but return the position of the first occurrence of any character which is NOT one of the target characters.

ANYALNUM	first alpha or numeric value
ANYALPHA	first alpha character
ANYDIGIT	first digit (number)
ANYPUNCT	first punctuation character
ANYSPACE	first whitespace character

And several others…

89

89

(3.6.1)

5.3 The ANY and NOT Families

Example: Before converting SODIUM, POTASSIUM, and CHLORIDE from character to numeric, we wish to verify there are no non-numeric values.

```
array val {3} $6 sodium potassium chloride;
array nval {3} sodium_n potassium_n chloride_n;
do i = 1 to 3;
  if anyalpha(left(val{i})) then do;
    variable = vname(val{i});
    value=val{i};
    note = 'Value is non-numeric';
    output valcheck;
  end;
```

detect non-numeric values

90

90

(3.6.1)

5.3 The ANY and NOT Families

NOTDIGIT is not quite equivalent to ANYALPHA.

```
if anyalpha(left(val{i})) then do;
```

TRUE when any uppercase or lowercase letter is encountered.

```
if notdigit(left(val{i})) then do;
```

TRUE when anything other than a digit 0-9 is found, including special characters and whitespace.

When checking for valid numeric data, we may wish to allow for certain characters other than just the digits.

```
if notdigit(trim(left(compress(val{i},'+-.')))) then do;
```

TRUE when anything other than a digit 0-9 or the plus, minus, or decimal characters is found.

91

91

(3.6.6)

5.4 INDEX and FIND Families

- The INDEX family (INDEX, INDEXC, and INDEXW) has been around awhile.
- The newer FIND family (FIND, FINDC, and FINDW) provide the same basic functionality with a great deal of additional flexibility.

92

92

(3.6.6)

5.4 INDEX and FIND Families

Example: Unless there is only one word, remove the last word in the list.

```
data lists;
input list $;
datalines;
A
A,B
A,B,C
A,B,C,
run;
```

Use the comma as the word delimiter.

B modifier = search right to left

Start position is the last character in the string.

```
data shorter;
   set lists;
   commaloc=findc(list, ',','b',-length(list)) ;
   if commaloc=0 then newlist=list;
   else newlist=substr(list,1,commaloc-1);
run;
```

Remove the last word only if a delimiter was found.

Obs	list	commaloc	newlist
1	A	0	A
2	A,B	2	A
3	A,B,C	4	A,B
4	A,B,C,	6	A,B,C

93

93

(3.6.6)

5.5 TRANWRD Function

The TRANWRD function replaces words within a text string.

There is a hidden potential problem related to how it handles lengths.

```
data _null_;
  length new1 $34;
  statement = "I enjoy going to SUGI conferences.";
  new1 = tranwrd(statement,"SUGI", "SGF");
  new2 = tranwrd(statement,"SUGI", "SGF");
  length_new1 = lengthc(new1);
  length_new2 = lengthc(new2);
  put length_new1 = ;
  put length_new2 = ;
  run;
```

NEW1 has a length of $34
NEW2 has a length of $200

Be sure to specify a length for the new variable receiving the translated text.

94

94

(3.6.6)

5.6 SCAN Function

The SCAN function retrieves the nth word from a string.

```
data locations;
autoloc = " 'c:\my documents' 'c:\temp' sasautos";
do i = 1 to 3;
   woq = scan(autoloc,i,' ');
   wq  = scan(autoloc,i,' ','q');
   wqr = scan(autoloc,i,' ','qr');
   output ;
end;
run;
```

'Q' ignores delimiters inside quoted substrings
'R' removes leading and trailing blanks
'QR' quotes are also removed.

i	woq	wq	wqr
1	'c:\my	'c:\my documents'	c:\my documents
2	documents'	'c:\temp'	c:\temp
3	'c:\temp'	sasautos	sasautos

95

95

(3.6.6)

5.7 Character Counting Functions

These functions count the number of items in a string. Each provides tremendous flexibility through a variety of options and modifiers.

COUNT	Counts appearances of a specific string of characters
COUNTC	Counts the characters that either do or do not appear in a string of characters.
COUNTW	Counts the number of words in a string

96

96

(3.6.6)

5.7 Character Counting Functions

Example: Count the words in a macro variable.

```
proc sql noprint;
select lname
   into :namelist separated by '/'
      from advrpt.demog(where=(lname=:'S'));
quit;
%put &namelist;
%put the number of names is %sysfunc(countw(&namelist,/));
```

97

97

(3.6.6)

5.8 IFN and IFC Functions

IFN and IFC provide a mechanism for conditional logic by returning a value based on evaluating a conditional.

IFN is used to return a numeric result, while IFC returns a character string. For both functions the arguments are:

1st logical expression to be evaluated
2nd result returned when the expression is true
3rd result returned when the expression is false
4th result returned when the expression is missing (optional)

98

98

(3.6.6)

5.8 IFN and IFC Functions

Divide patients into GENERATION according to birth year.

```
data generation;
   set advrpt.demog(keep=lname fname dob);
   length generation $10;
   if year(dob) = . then generation='Unknown';
   else if year(dob) lt 1945 then generation= 'Greatest';
   else if year(dob) ge 1945 then generation = 'Boomer';
   run;
```

```
data generation2;
   set advrpt.demog(keep=lname fname dob);
   length generation $10;
   generation =
         ifc(year(dob) ge 1945,'Boomer','Greatest','Unknown');
   run;
```

Not quite the same!!
Nothing will map to 'Unknown'

99

99

(3.6.6)

5.8 IFN and IFC Functions

Divide patients into GENERATION according to birth year.

```
ifc(year(dob) ge 1945,'Boomer','Greatest','Unknown')
```

The expression can only resolve to 0 or 1

```
ifc(dob*(year(dob) ge 1945),'Boomer','Greatest','Unknown')
```

The expression can now return a missing value.

100

100

(3.6.6)

5.9 WHICHN and WHICHC Functions

- WHICHC / WHICHN – Searches a list for values for a specific value and returns the position of the match
- WHICHC takes character values, WHICHN takes numeric values
- Syntax: **WHICHC**(*argument*, *value1* <, *value2*, ...>)
  WHICHN(*argument*, *value1* <, *value2*, ...>)
- Both functions return a numeric value.
  - If ARGUMENT matches VALUE1 → 1 is returned.
  - If ARGUMENT matches VALUE2 → 2 is returned.
  - And so on…
  - No match → 0 is returned.

101

101

(3.6.6)

5.9 WHICHN and WHICHC Functions

Create a custom sorting variable with WHICHC

```
proc summary data=sashelp.cars;
   var mpg_city;
   output out=carsumm
      (drop=_TYPE_ _FREQ_);
run;

data carsumm2;
   set carsumm;
   rowsort = whichc(_STAT_,'N','MEAN','STD','MIN','MAX');
run;
```

Obs	_STAT_	MPG_City	rowsort
1	N	428.000	1
2	MIN	10.000	4
3	MAX	60.000	5
4	MEAN	20.061	2
5	STD	5.238	3

102

102

(3.6.4)

5.10 Maximum and Minimum Values

MAX	Returns the largest non-missing value from a list
MIN	Returns the smallest non-missing value from a list
LARGEST	Returns the nth largest non-missing value from a list
SMALLEST	Returns the nth non-missing smallest value from a list
ORDINAL	Returns the nth smallest value from a list of values (missing and non-missing)

- The LARGEST and SMALLEST functions can return something other than the single extreme.
- The ORDINAL function allows the consideration of missing values.

103

103

Chapter 6

DOING MORE WITH THE SET STATEMENT

104

104

Chapter 6 Overview

- 6.1 SET Statement Options
- 6.2 DATA Step with Two SET Statements
- 6.3 Using the DOW Loop

105

105

(3.8)

6.1 SET Statement Options

Although most DATA steps use the SET statement, few programmers take advantage of its full potential. The SET statement has options that can be used to control how the data are to be read.

END=	used to detect the last observation from the incoming data set(s)
KEY=	specifies a index to be used when reading
INDSNAME=	used to identify the current data source
NOBS=	stores number of observations
OPEN=	determines when to open a data set
POINT=	designates the next observation to read
UNIQUE	used with KEY= to read from the top of the index

106

106

(3.8.1)

6.1.1 Using POINT= and NOBS=

- SET statement by default reads observations sequentially.
- The POINT= option perform a non-sequential read.
- POINT= option identifies a temporary variable that indicates the number of the next observation to read.
- NOBS= option identifies a temporary variable, which will hold the number of observations on the incoming data set.

107

107

(3.8.1)

6.1.1 Using POINT= and NOBS=

Example: Randomly read a subset of observations from &DSN.

```
%macro rand_wo(dsn=,pcnt=0);
  data rand_wo(drop=cnt totl);
    totl = ceil(&pcnt*obscnt);
    array obsno {10000} _temporary_;
     do until(cnt = totl);
       point = ceil(ranuni(0)*obscnt);
       if obsno{point} ne 1 then do;
          set &dsn point=point nobs=obscnt;
          output;
          obsno{point}=1;
          cnt+1;
       end;
    end;
    stop;
    run;
%mend rand_wo;
%rand_wo(dsn=advrpt.demog,pcnt=.3)
```

Select total number of observations based on PCNT parameter and value of OBSCNT.

Randomly select the next observation.

OBSCNT receives its value during DATA step compilation.

The value of POINT is the next observation to read.

Flag this observation, it has now been read.

108

108

(3.8.2)

6.1.2 Using INDSNAME=

The INDSNAME= option stores the name of the data set from which the current observation was read.

```
data grouped1;
   set boomer(in=inboom)
       others(in=inoth);
   if inboom then group='BOOMER';
   else if inoth then group='OTHERS';
   run;
```

The IN= data set options create binary flags (0 or 1).

```
data grouped2;
   set boomer
       others indsname=dsn;
   length group $6;
   group=scan(dsn,2,'.');
   run;
```

DSN contains either work.boomer or work.others.

109

109

(3.7.2)

6.2 DATA Step with Two SET Statements

- Using multiple SET statements provides great power and flexibility (as we saw in sections 3.5 - 3.6).
- Caution must be exercised when taking control of the read process.

```
data new;
   set a;
   set b;
   run;
```

- This is essentially a one-to-one merge with restrictions.
- Once SAS reads the last observation from either data set the full DATA step is not fully executed again.
- Number of observations in the new data set is determined by the number of observations in the smallest original data set.
- For variables in common, values read from later data sets replace those read from earlier ones.

110

110

(3.9.1)

6.3 Using the DOW Loop

The DOW loop, which is also known as the DO-W loop, was named for Ian Whitlock who popularized the technique and first demonstrated its efficiencies.

```
data implied;
   set big;
   output implied;
   run;
```

The DATA step has an implied loop. The loop is executed once for each incoming observation.

```
data dowloop;
   do until(eof);
      set big end=eof;
      output dowloop;
   end;
   stop;
   run;
```

The implicit loop is replaced with an explicit one (here a DO UNTIL).

Terminate the step with a STOP.

111

111

(3.9.1)

6.3 Using the DOW Loop

Example: Merge a single observation summary data set onto the analysis data and calculate percent change.

Conventional approach:

```
proc summary data=advrpt.demog;
   var wt;
   output out=means mean=/autoname;
   run;

data Diff1;
   if _n_=1 then set means(keep=wt_mean);
   set advrpt.demog(keep=lname fname wt);
   diff = (wt-wt_mean)/wt_mean;
   run;
```

The IF statement conditionally executes the first SET statement.

However, the IF statement must be checked for every observation in the analysis data set!

112

112

(3.9.1)

6.3 Using the DOW Loop

Example: Merge a single observation summary data set onto the analysis data and calculate percent change.

Using the DOW Loop:

```
data Diff2;
  set means(keep=wt_mean);
  do until(eof);
    set advrpt.demog(keep=lname fname wt)
        end=eof;
    diff = (wt-wt_mean)/wt_mean;
    output diff2;
  end;
  stop;
  run;
```

The IF on the SET statement is no longer needed. This DATA steps executes only once!

The DO loop executes for each observation read from the MEANS data set.

The second SET statement is inside the DO UNTIL loop and will continue executing until it has read every observation.

Terminate the DATA step.

113

113

Chapter 7

WORKING WITH MULTIPLE INCOMING DATA SETS

114

114

Multiple Incoming Data Sets

- We often need to combine data from multiple data sets.
- There are many methods available:
  - MERGE statement (most common)
  - Double SET statements
  - Using formats
  - Using indexes
  - Using hash objects
  - The SQL procedure
  - And many more…

115

115

Chapter 7 Overview

- 7.1 Merging Data Sets with the MERGE Statement
- 7.2 Merging with Two SET Statements
- 7.3 Controlling the Merge with an Index
- 7.4 Key Indexing – a Simple Hash
- 7.5 Merging with a Hash Object

116

116

(3.7)

7.1 Merging Data Sets with MERGE

- MERGE statement is very common way to merge data sets.
- Although seemingly simple, many pitfalls exist.
- We will examine three specific issues:
  - Inconsistent BY variable attributes
  - Unequal BY variable lengths
  - Variables in common (other than the BY variables)

117

117

(3.7.1)

7.1.1 Inconsistent BY Variable Attributes

When BY variables have inconsistent attributes across the data sets being merged, unfortunate things can happen.

```
data labnames;
   merge advrpt.demog(keep=subject lname fname)
         advrpt.lab_chemistry(keep=subject visit labdt
                              in=inlab);
   by subject;
   if inlab;
   run;
```

SUBJECT is numeric in one data set and character in the other.

ERROR: Variable subject has been defined as both character and numeric.

Typically when we misuse a variable's type, such as when we use a character variable in an arithmetic statement, SAS will attempt to convert the variable's type. When the variable is in the BY statement, a conversion is not possible and the step fails.

118

118

(3.7.1)

7.1.1 Inconsistent BY Variable Attributes

What if we convert the numeric SUBJECT to character before merging?

```
data demog_c;
   set advrpt.demog(keep=subject lname fname
                    rename=(subject=ptid));
   subject = put(ptid,4.);
   run;
```

The original values of SUBJECT contain 3 digits.

If SUBJECT contains a leading blank and the MERGE will fail, not because of an error but by not finding any matching subjects.

```
subject = left(put(ptid,4.));
```

119

119

(3.7.1)

7.1.2 Unequal BY Variable Lengths

Unequal BY variable length can be a problem.

VIEWTABLE: Work.Pets

	lname	FNAME	pet
1	Adams	Joan	Dog
2	Adams	Mary	Cat
3	Alexa	Mark	Cat
4	Antle	Peter	Dog

VIEWTABLE: Work.Demogsymp

	subject	lname	fname	symp
1	200	Adams	Mary	02
2	201	Adamson	Joan	10
3	202	Alexander	Mark	
4	203	Antler	Peter	10

```
data petsymptoms;
   merge pets(keep=lname fname pet)
         demogsymp(keep=subject lname fname symp);
   by lname fname;
   run;
```

Incoming data sets are sorted.

ERROR: BY variables are not properly sorted on data set WORK.DEMOGSYMP.

120

120

(3.7.1)

7.1.2 Unequal BY Variable Lengths

Truncation occurs because the length of LNAME in the data set PETS ($5) determines the length for LNAME on the PDV. The result is truncation for LNAME values from DEMOGSYMP.

VIEWTABLE: Work.Pets

	lname	FNAME	pet
1	Adams	Joan	Dog
2	Adams	Mary	Cat
3	Alexa	Mark	Cat
4	Antle	Peter	Dog

VIEWTABLE: Work.Demogsymp

	subject	lname	fname	symp
1	200	Adams	Mary	02
2	201	Adamson	Joan	10
3	202	Alexander	Mark	
4	203	Antler	Peter	10

```
data petsymptoms;
   merge pets(keep=lname fname pet)
         demogsymp(keep=subject lname fname symp);
   by lname fname;
   run;
```

Luckily, Mary Adams is sorted before Joan Adamson, which causes the error. No error would be detected for Tanya Adamson!!

121

121

(3.7.1)

7.1.2 Unequal BY Variable Lengths

Two ways to resolve:

- Reorder data sets on the MERGE statement
  (not always possible with multiple BY variables of unequal length)
- Explicitly declare the length with a LENGTH statement
  (Must come prior to MERGE so compiler encounters it first when constructing PDV)

```
data petsymptoms;
   length lname $10;
   merge pets(keep=lname fname pet)
         demogsymp(keep=subject lname fname symp);
   by lname fname;
   run;
```

122

122

(3.7.2)

7.1.3 Variables in Common

After a merge or join, variables common to more than one data set will only appear once in the new data set.

```
data aelab;
   merge labchem(keep=subject date
                 where=(date<'01jul2003'd))
         ae(keep=subject date);
   by subject;
   run;
```

Select only dates before 7/2003

DATE is in both data sets. These dates are all from AE none are from LABCHEM.

Obs	SUBJECT	date
1	200	07/28/2006
2	201	07/06/2006
3	202	.
4	203	09/13/2006
5	204	09/27/2006

. . . . portions of the are table not shown . . . .

123

123

(6.4)

7.2 Merging with Two SET Statements

When merging with two SET statements, the programmer must control the read process.

A number of extremely powerful and efficient performance enhancements are available with these techniques.

Exercise caution until you understand the process.

124

124

(6.4)

7.2 Merging with Two SET Statements

As we've seen previously, the MERGE statement is the traditional way of performing a DATA step merge.

```
data mrgnames;
  merge demog(keep=subject clinnum edu)
        clinicnames(keep=clinnum clinname);
  by clinnum;
  run;
```

Both data sets must either be sorted or indexed on CLINNUM.

125

125

(6.4)

7.2 Merging with Two SET Statements

This technique still requires sorted data sets. It's faster than a MERGE, but also more complex.

```
data withnames(keep=subject clinnum clinname);
  set demog(rename=(clinnum=code));
  * The following expression is true only when
  * the current CODE is a duplicate.;
  if code=clinnum then output;
  do while(code>clinnum);
    * lookup the clinic name using the code (clinnum)
    * from the primary data set;
    set clinicnames(keep=clinnum clinname);
    if code=clinnum then output;
  end;
  run;
```

Read a record from the primary data set, renaming our merge variable.

Loop through records from the secondary data set.

Only write out a record when CODE (clinic number from DEMOG) matches CLINNUM (clinic number from CLINICNAMES).

126

126

(6.6.2)

7.3 Controlling the Merge with an Index

Indexes:

- Logically sort your data without physically sorting it.
- Can help avoid frequent sorting and re-sorting of the data.
- Can be created using the DATASETS procedure.

```
proc datasets library=advrpt nolist;
  modify clinicnames;
    index create clinnum / unique;
  modify demog;
    index create clinnum;
  quit;
```

127

127

(6.6.2)

7.3 Controlling the Merge with an Index

You can merge when an index exists on only one data set.

The KEY= option on the SET statement identifies an index to be used.

The data set DEMOG does not need an index and does not need to be sorted!!!

```
data keynames;
  set advrpt.demog
        (keep=subject clinnum lname fname);
  set advrpt.clinicnames key=clinnum/unique;
  if _iorc_ ne 0 then clinname=' ';
run;
```

The value of CLINNUM read from DEMOG is used to perform an indexed read from CLINICNAMES.

When no match is found, set CLINNAME to missing so the previous value isn't retained.

```
data keynames;
  merge demog
        clinicnames;
  by clinnum;
  run;
```

128

128

(6.6.2)

7.3 Controlling the Merge with an Index

Values of _IORC_ may change in future releases.

```
data rckeylookup;
  set advrpt.demog(keep=subject clinnum lname fname);
  set advrpt.clinicnames key=clinnum/unique;
  select (_iorc_);
    when (%sysrc(_sok)) do; *lookup was successful;
      output;
    end;
    when (%sysrc(_dsenom)) do;  *No matching clinic number found;
      clinname='Unknown';
      output;
    end;
    otherwise do;
      put  'Problem with lookup ' clinnum=;
      stop;
    end;
  end;
run;
```

SAS supplies an autocall macro %SYSRC that can be used to decode the values in _IORC_.

```
data rckeylookup;
  merge demog
        clinicnames;
  by clinnum;
  run;
```

129

129

(6.7.2)

7.4 Key Indexing – A Simple Hash

Use an array to hold values. Neither data set needs to be sorted.

```
data clinnames(keep=subject lname fname clinnum clinname);
  array chkname {999999} $35 _temporary_;
  do until(allnames);
    set advrpt.clinicnames end=allnames;
    chkname{input(clinnum,6.)}=clinname;
  end;
  do until(alldemog);
    set advrpt.demog(keep=subject lname fname clinnum)
          end=alldemog;
    clinname = chkname{input(clinnum,6.)};
    output clinnames;
  end;
  stop;
  run;
```

Establish an array sufficiently large to hold all clinic numbers.

Read and store the names, indexed by clinic number.

Read a clinic number from DEMOG and retrieve the corresponding clinic name from the array.

```
data clinnames;
  merge demog
        clinicnames;
  by clinnum;
  run;
```

130

130

(6.7.2)

7.5 Merging Using a Hash Object

Hash objects also avoid the need to sort.

```
data hashnames(keep=subject clinnum clinname lname fname);
   if 0 then set advrpt.clinicnames;
   declare hash lookup(dataset: 'advrpt.clinicnames',
                       hashexp: 8);
  lookup.defineKey('clinnum');
  lookup.defineData('clinname');
  lookup.defineDone();
   * Read the primary data;
  do until(done);
     set advrpt.demog(keep=subject clinnum lname fname)
         end=done;
     if lookup.find() = 0 then output hashnames;
  end;
  stop;
  run;
```

Load the clinic names in the LOOKUP hash object, indexed by clinic number.

For each clinic number retrieve the clinic name.

```
data hashnames;
  merge demog
        clinicnames;
  by clinnum;
  run;
```

131

131

WRAP-UP

132

132

Recommended Resources

- Carpenter's Guide to Innovative SAS® Techniques
- Art Carpenter (SAS Press, 2012)
- Available from SAS Bookstore and other booksellers in hardcopy and electronic formats.

133

133

Recommended Resources

- Over 30,000 SAS User Group papers available free online at http://www.lexjansen.com.
- 40+ years of papers from SAS Global Forum (formerly SUGI), PharmaSUG, and regional conferences (SCSUG, SESUG, MWSUG, WUSS, etc.)
- Fully searchable database
- Includes many papers authored by SAS staff, SAS Press authors, and well-known industry experts

134

134

Recommended Resources

- Selected DATA Step Papers by Art Carpenter
  - Innovative Techniques: Doing More with Loops and Arrays
  - Table Lookup Techniques: From the Basics to the Innovative
  - DATA Step Merging Techniques: From Basic to Innovative
  - Using Arrays to Quickly Perform Fuzzy Merge Look-ups: Case Studies in Efficiency
  - Are You Missing Out? Working with Missing Values to Make the Most of What is Not There
  - Not All Equals are Created Equal: Nonstandard Statement Structures in the DATA Step
  - Name that Function: Punny Function Names with Multiple MEANings and Why You Do Not Want to be MISSING Out (Coauthored with Ben Cochran)

135

135

Recommended Resources

- Selected DATA Step Papers by Josh Horstman
  - Merge with Caution: How to Avoid Common Problems when Combining SAS® Datasets
  - Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS® Code
  - Fifteen Functions to Supercharge Your SAS® Code
  - Dating for SAS Programmers
  - Let the CAT Out of the Bag: String Concatenation in SAS® 9
  - Five Ways to Flip-Flop Your Data
  - Inside the DATA Step: Pearls of Wisdom for the Novice SAS Programmer (co-authored with Britney Gilbert)

136

136

137