SAS® Programming 2: Data Manipulation Techniques – Syntax

Course Notes

SAS® Programming 2: Data Manipulation Techniques – Syntax Course Notes

Table of Contents

To learn more…

For information about other courses in the curriculum, contact the SAS Education Division at 1-800-333-7660, or send e-mail to training@sas.com. You can also find this information on the web at http://support.sas.com/training/ as well as in the Training Course Catalog.

For a list of SAS books (including e-books) that relate to the topics covered in this course notes, visit https://www.sas.com/sas/books.html or call 1-800-727-0025. US customers receive free shipping to US addresses.

Lesson 1 Syntax

1.1 Lesson 1 Syntax Review: Controlling DATA Step Processing

Understanding DATA Step Processing

- The DATA step is processed in two phases: compilation and execution.
- During compilation, SAS creates the program data vector (PDV) and establishes data attributes and rules for execution.
- The PDV is an area of memory established in the compilation phase. It includes all columns that will be read or created, along with their assigned attributes. The PDV is used in the execution phase to hold and manipulate one row of data at a time.
- During execution, SAS reads, manipulates, and writes data. All data manipulation is performed in the PDV.

```
PUTLOG _ALL_;
PUTLOG column=;
PUTLOG "message
```

Directing DATA Step Output

```
OUTPUT;
DATA table1 <table2 ...>;
OUTPUT table1 <table2 ...>;
```

- By default, the end of a DATA step causes an implicit output, which writes the contents of the PDV to the output table.
- The explicit OUTPUT statement can be used in the DATA step to control when and where each row is written.
- If an explicit OUTPUT statement is used in the DATA step, it disables the implicit output at the end of the DATA step.
- One DATA step can create multiple tables by listing each table name in the DATA statement.
- The OUTPUT statement followed by a table name writes the contents of the PDV to the specified table.

```
table (DROP=col1 col2…);
table (KEEP=col1 col2…);
```

- DROP= or KEEP= data set options can be added on any table in the DATA statement. If you add these options on the DATA statement, the columns are not added to the output table.
- Columns that will be dropped are flagged in the PDV and are not dropped until the row is output to the designated table. Therefore, dropped columns are still available for processing in the DATA step.

- DROP= or KEEP= data set options can be added in the SET statement to control the columns that are read into the PDV. If a column is not read into the PDV, it is not available for processing in the DATA step.

1.2 Lesson 2 Syntax Review: Summarizing Data

Creating an Accumulating Column

- At the beginning of the first iteration of the DATA step, all column values are set to missing.
- By default, all computed columns are reset to missing at the beginning of each subsequent iteration of the DATA step. This is called reinitializing the PDV. Columns read from the SET statement automatically retain their value in the PDV.
- To create an accumulating column, this default behavior must be modified.

Directing DATA Step Output

RETAIN *column* <*initial-value*>;
column+*expression*;

Processing Data in Groups

- To process data in groups, the data first must be sorted by the grouping column or columns. This can be accomplished with PROC SORT.
- The BY statement in the DATA step indicates how the data has been grouped. Each unique value of the BY column will be identified as a separate group.
- The BY statement creates two temporary variables in the PDV for each column listed as a BY column: **First.*bycol*** and **Last.*bycol***.
- **First.*bycol*** is 1 for the first row within a group and 0 otherwise. **Last.*bycol*** is 1 for the last row within a group and 0 otherwise.
- Conditional IF-THEN logic can be used based on the values of the **First./Last.** variable to execute statements in the DATA step.

BY <**DESCENDING**> *col-name(s)*;
FIRST.*bycol*
LAST*bycol*

- **First./Last.** variables can be used in combination with IF-THEN logic to execute one or more statements at the beginning or end of a group.
- The subsetting IF statement affects which rows are written from the PDV to the output table. The expression can be based on values in the PDV.
- When the subsetting IF expression is true, the remaining statements are executed for that iteration, including any explicit OUTPUT statements or the implicit output that occurs with the RUN statement.

- If the subsetting IF expression is not true, the DATA step immediately stops processing statements for that particular iteration, likely skipping the output trigger, and the row is not written to the output table.

IF *expression*;

1.3 Lesson 3 Syntax Review: Manipulating Data with Functions

Understanding SAS Functions and CALL Routines

function(*argument1*, *argument2*, ...);
CALL *routine*(*argument-1* <, ...*argument-n*>);

Using Numeric and Date Functions

RAND(*'distribution', parameter1, ...parameterk*)
LARGEST(*k, value-1 <, value-2 ...>*)
ROUND(*number <, rounding-unit>*)

RAND function

- The RAND function generates random numbers from a selected distribution.
- The first argument specifies the distribution, and the remaining arguments differ depending on the distribution.
- To generate a random, uniformly distributed integer, use 'INTEGER' as the first argument. The second and third arguments are the lower and upper limits.

LARGEST function

- The LARGEST function returns the kth largest nonmissing value.
- The first argument is the value to return, and the remaining arguments are the numbers to evaluate.
- There is also a SMALLEST function that returns the kth smallest nonmissing value.

ROUND function

- The ROUND function rounds the first argument to the nearest integer.
- The optional second argument can be provided to indicate the rounding unit.

Using Character Functions

Function	What it does
COMPBL(*string*)	Returns a character string with all multiple blanks in the *source* string converted to single blanks.
COMPRESS (*string* <, *characters*>)	Returns a character string with specified *characters* removed from the *source* string
STRIP(*string*)	Returns a character *string* with leading and trailing blanks removed

SCAN(*string, n <, 'delimiters'>***)**
PROPCASE(*string <, 'delimiters'>***)**

SCAN Function

- The SCAN function returns the nth word in a string.
- If *n* is negative, the SCAN function begins reading from the right side of the string.
- The default delimiters are as follows: blank ! $ % & ( ) * + , - . / ; < ^ |
- The optional third argument enables you to specify a delimiter list. All delimiter characters are enclosed in a single set of quotation marks.

PROPCASE Function

- The PROPCASE function converts all uppercase letters to lowercase letters. It then converts to uppercase the first character of each word.
- The default delimiters are as follows: blank / - (. tab
- The optional second argument enables you to specify a delimiter list. All delimiter characters are enclosed in a single set of quotation marks.

Finding character strings
FIND(*string***,** *substring* **<,** **'***modifiers***'>)**
Identifying character positions

Function	What it does
LENGTH(*string*)	Returns the length of a non-blank character string, excluding trailing blanks; returns 1 for a completely blank string.
ANYDIGIT(*string*)	Returns the first position at which a digit is found in the string.
ANYALPHA(*string*)	Returns the first position at which an alpha character is found in the string.
ANYPUNCT(*string*)	Returns the first position at which punctuation character is found in the string.

TRANWRD(*source*, *target*, *replacement*)
Building character strings

Function	What it does
CAT(*string1, ... stringn*)	Concatenates strings together, does not remove leading or trailing blanks
CATS(*string1, ... stringn*)	Concatenates strings together, removes leading or trailing blanks from each string
CATX('*delimiter*', *string1, ... stringn*)	Concatenates strings together, removes leading or trailing blanks from each string, and inserts the delimiter between each string

Using Special Functions to Convert Column Type

- The INPUT function converts a character value to a numeric value using a specified informat.
- The PUT function converts a numeric or character value to a character value using a specified format.
- SAS automatically tries to convert character values to numeric values using the w. informat.
- SAS automatically tries to convert numeric values to character values using the BEST12. format.
- If SAS automatically converts the data, a note is displayed in the SAS log. If you explicitly tell SAS to convert the data with a function, a note is not displayed in the SAS log.

- Some functions such as the CAT functions automatically convert data from numeric to character and also remove leading blanks on the converted data. No note is displayed in the SAS log.

```
DATA output-table;
      SET input-table (RENAME=(current-column=new-column));
      ...
      column1 = INPUT(source, informat);
      column2 = PUT(source, format);
      ...
RUN;
```

1.4 Lesson 4 Syntax Review: Creating and Using Custom Formats

Creating and Using Custom Formats

```
PROC FORMAT;
      VALUE format-name value-or-range-1 = 'formatted-value'
                        value-or-range-2 = 'formatted-value'
. . . ;
RUN;
```

- The FORMAT procedure is used to create custom formats.
- A VALUE statement specifies the criteria for creating one custom format.
- Multiple VALUE statements can be used within the PROC FORMAT step.
- The format name can be up to 32 characters in length, must begin with a $ followed by a letter or underscore for character formats, and must begin with a letter or underscore for numeric formats.
- On the left side of the equal sign, you specify individual values or a range of values that you want to convert to formatted values. Character values must be in quotation marks; numeric values are not quoted.
- On the right side of the equal sign, you specify the formatted values that you want the values on the left side to become. Formatted values need to be in quotation marks.
- The keywords LOW, HIGH, and OTHER can be used in the VALUE statement.
- You do not include a period in the format name when you create the format, but you do include the period in the name when you use the format.
- Custom formats can be used in the FORMAT statement and the PUT function.

Creating Custom Formats from Tables

```
PROC FORMAT CNTLIN=input-table FMTLIB;
      SELECT format-names;
RUN;
```

- The CNTLIN= option specifies a table from which PROC FORMAT builds formats.
- The input table must contain at a minimum three character columns:
  - **Start**, which represents the raw data values to be formatted.
  - **Label**, which represents the formatted labels.
  - **FmtName**, which contains the name of the format that you are creating. Character formats start with a dollar sign.

1.5 Lesson 5 Syntax Review: Combining Tables

Concatenating Tables

```
DATA output-table;
      SET input-table1(rename=(current-colname=new-colname))
          input-table2 ...;
RUN;
```

- Multiple tables listed in the SET statement are concatenated.
- SAS first reads all the rows from the first table listed in the SET statement and writes them to the new table. Then it reads and writes the rows from the second table, and so on.
- Columns with the same name are automatically aligned. The column properties in the new table are inherited from the first table that is listed in the SET statement.
- Columns that are not in all tables are also included in the output table.
- The RENAME= data set option can be used to rename columns in one or both tables so that they align in the new table.
- Additional DATA step statements can be used after the SET statement to manipulate the data.

Merging Tables

If data needs to be sorted prior to the merge:

```
PROC SORT DATA=input-table OUT=output-table;
      BY BY-column;
RUN;
```

```
DATA output-table;
      MERGE input-table1 input-table2 ...;
      BY BY-column(s);
RUN;
```

- Any tables listed in the MERGE statement must be sorted by the same column (or columns) listed in the BY statement.
- The MERGE statement combines rows where the BY-column values match.
- This syntax merges multiple tables in both one-to-one and one-to-many situations.

```
Identifying Matching and Nonmatching Rows
DATA output-table;
      MERGE input-table1(IN=var1) input-table2(IN=var2) ...;
      BY BY-column(s);
RUN;
```

- By default, both matches and nonmatches are written to the output table in a DATA step merge.
- The IN= data set option follows a table in the MERGE statement and names a variable that will be added to the PDV. The IN= variables are included in the PDV during execution, but they are not written to the output table. Each IN= variable relates to the table that the option follows.
- During execution, the IN= variable is assigned a value of 0 or 1. 0 means that the corresponding table did ***not*** include the BY column value for that row, and 1 means that it did include the BY-column value.
- The subsetting IF or IF-THEN logic can be used to subset rows based on matching or nonmatching rows.

1.6 Lesson 6 Syntax Review: Processing Repetitive Code

Using Iterative DO Loops

```
DATA output-table;
      . . .
      DO index-column = start TO stop <BY increment> ;
          . . . repetitive code . . .
      END;
      . . .
RUN;
```

- The iterative DO loop executes statements between the DO and END statements repetitively, based on the value of an index column.
- The *index-column* parameter names a column whose value controls execution of the DO loop. This column is included in the table that is being created unless you drop it.
- The *start* value is a number or numeric expression that specifies the initial value of the index column.
- The *stop* value is a number or numeric expression that specifies the ending value that the index column must exceed to stop execution of the DO loop.
- The *increment* value specifies a positive or negative number to control the incrementing of the index column. The BY keyword and the increment are optional. If they are omitted, the index column is increased by 1.

```
DATA output-table;
      SET input-table;
      . . .
      DO index-column = start TO stop <BY increment> ;
          . . . repetitive code . . .
            <OUTPUT;>
      END;
      . . .
      <OUTPUT;>
RUN;
```

- The DO loop iterates for each iteration of the DATA step.
- An OUTPUT statement between the DO and END statements outputs one row for each iteration of the DO loop.
- An OUTPUT statement after the DO loop outputs a row based on the final iteration of the DO loop. The index column will be an increment beyond the stop value.
- DO loops can be nested..

Using Conditional DO Loops

```
DATA output-table;
      SET input-table;
      . . .
      DO UNTIL | WHILE (expression);
            . . . repetitive code . . .
            <OUTPUT;>
      END;
RUN;
```

- A conditional DO loop executes based on a condition, whereas an iterative DO loop executes a set number of times.
- A DO UNTIL executes until a condition is true, and the condition is checked at the ***bottom*** of the DO loop. A DO UNTIL loop always executes at least one time.
- A DO WHILE executes while a condition is true, and the condition is checked at the ***top*** of the DO loop. A DO WHILE loop does not iterate even once if the condition is initially false.
- The expression needs to be in a set of parentheses for the DO UNTIL or DO WHILE

```
DATA output-table;
      SET input-table;
      . . .
      DO index-column = start TO stop <BY increment> UNTIL | WHILE (expression);
            . . . repetitive code . . .
      END;
      . . .
RUN;
```

- An iterative DO loop can be combined with a conditional DO loop. The index column is listed in the DO statement before the DO UNTIL or DO WHILE condition.
- For an iterative loop combined with a DO UNTIL condition, the condition is checked ***before*** the index column is incremented at the bottom of the loop.
- For an iterative loop combined with a DO WHILE condition, the condition is checked at the ***top*** of the loop and the index column is incremented at the ***bottom*** of the loop.

1.7 Lesson 7 Syntax Review: Restructuring Tables

Restructuring Data with the DATA Step

- The DATA step can be used to restructure tables.
- Assignment statements are used to create new columns for stacked values.
- The explicit OUTPUT statement is used to create multiple rows for each input row.
- DO loops can be nested.

Restructuring Data with the TRANSPOSE Procedure

```
PROC TRANSPOSE DATA=input-table OUT=output-table
                    PREFIX=column> <NAME=column>;
     <VAR columns(s);>
     <ID column;>
     <BY column(s);>
RUN;
```

- PROC TRANSPOSE can be used to restructure table
- The OUT= option creates or replaces an output table based on the syntax used in the step.
- By default, all numeric columns in the input table are transposed into rows in the output table.
- The VAR statement lists the column or columns to be transposed.
- The output table will include a separate column for each value of the ID column. There can be only one ID column. The ID column values must be unique in the column or BY group.
- The BY statement transposes data within groups. Each unique combination of BY values creates one row in the output table.
- The PREFIX= option provides a prefix for each value of the ID column in the output table.
- The NAME= option names the column that identifies the source column containing the transposed values.