Print

Common Date Functions for Creating Columns

SAS date functions are incredibly helpful for creating and manipulating SAS dates.

These functions **extract** information from the SAS date column or value provided in the argument:

Date Function	What it Does
MONTH (*SAS-date*)	Returns a number from 1 through 12 that represents the month
YEAR (*SAS-date*)	Returns the four-digit year
DAY (*SAS-date*)	Returns a number from 1 through 31 that represents the day of the month
WEEKDAY (*SAS-date*)	Returns a number from 1 through 7 that represents the day of the week (Sunday=1)
QTR (*SAS-date*)	Returns a number from 1 through 4 that represents the quarter

These functions enable you to **create** SAS date values from the arguments.

Date Function	What it Does
TODAY ()	Returns the current date as a numeric SAS date value (no argument is required because the function reads the system clock)
MDY (*month*, *day*, *year*)	Returns a SAS date value from numeric month, day, and year values
YRDIF (*startdate*, *enddate*, 'AGE')	Calculates a precise age between two dates

Close

Lesson 1 Syntax Review: Essentials

Running a SAS Program

- Programs can be submitted by clicking the **Run** icon or pressing the F3 key.
- To run a subset of a program, highlight the desired statements first. If you are using SAS Studio, click the **Run** icon or press F3. If you are using SAS Enterprise Guide, click the arrow next to **Run** and click **Run Selection** or press F3.
- A program can create a log, results, and output data.
- Submitting a program that has run previously in Enterprise Guide or SAS Studio replaces the log, output data, and results.
- Submitting a program that has run previously in the SAS windowing environment appends the log and results.

Understanding SAS Syntax

- SAS programs consist of DATA and PROC steps, and each step consists of statements.

```
DATA ... ;
    other statements
RUN;
```

```
PROC ... ;
    other statements
RUN;
```

- Global statements are outside steps.

```
TITLE ... ;
```

```
OPTIONS ... ;
```

```
LIBNAME ... ;
```

- All statements end with a semicolon.
- Spacing doesn't matter in a SAS program.
- Unquoted values can be lowercase, upper case, or mixed case.
- Consistent program spacing is a good practice to make programs legible.
- Use the following automatic spacing features:

  SAS Studio: Click the **Format Code** icon.
  Enterprise Guide: Select **Edit** > **Format Code** or press Ctrl+I.

- Comments can be added to prevent text in the program from executing.
- Some common syntax errors are unmatched quotes, missing semicolons, misspelled keywords, and invalid options.
- Syntax errors might result in a warning or error in the log.
- Refer to the log to help diagnose and resolve syntax errors.

Last modified: Tuesday, August 28, 2018, 12:09 PM

Lesson 2 Syntax Review: Accessing Data

Understanding SAS Data

- SAS data sets have a data portion and a descriptor portion.
- SAS columns must have a name, type, and length.
- Column names can be 1-32 characters, must start with a letter or underscore and continue with letters, numbers or underscores, and can be in any case.
- Columns are either character or numeric.
- Character columns can have a length between 1 and 32,767 bytes (1 byte = 1 character).
- Numeric columns are stored with a length of 8 bytes.
- Character columns can consist letters, numbers, special characters, or blanks.
- Numeric columns can consist of the digits 0-9, minus sign, decimal point, and E for scientific notation.
- SAS date values are a type of numeric value and represent the number of days between January 1, 1960, and a specified date.

```
PROC CONTENTS DATA=table-name;
RUN;
```

Accessing Data through Libraries

- A *libref* is the name of the library that can be used in a SAS program to read data files.
- The *engine* provides instructions for reading SAS files and other types of files.
- The *path* provides the directory where the collection of tables is located.
- The libref remains active until you clear it, delete it, or shut down SAS.

```
LIBNAME libref engine "path";
LIBNAME libref CLEAR;
```

Automatic Libraries

- Tables stored in the **Work** library are deleted at the end of each SAS session.
- **Work** is the default library, so if a table name is provided in the program without a libref, the table will be read from or written to the Work library.
- The **Sashelp** library contains a collection of sample tables and other files that include information about your SAS session.

Using a Library to Read Excel Files

- The XLSX engine enables us to read data directly from Excel workbooks. The XLSX engine requires the SAS/ACCESS to PC Files license.
- The VALIDVARNAME=V7 system option forces table and column names read from Excel to adhere to recommended SAS naming conventions. Spaces and special symbols are replaced with underscores, and names greater than 32 characters are truncated.
- Date values are automatically converted to numeric SAS date values and formatted for easy interpretation.
- Worksheets or named ranges from the Excel workbook can be referenced in a SAS program as *libref.spreadsheet-name*.
- When you define a connection to a data source such as Excel or other databases, it's a good practice to delete the libref at the end of your program with the CLEAR option.

```
OPTIONS VALIDVARNAME=V7;
LIBNAME libref XLSX "path/file.xlsx";
```

Importing Data

- The DBMS option identifies the file type. The CSV value is included with Base SAS.

- The OUT= option provides the library and name of the SAS output table.
- The REPLACE option is necessary to overwrite the SAS output table if it exists.
- SAS assumes that column names are in the first line of the text file and data begins on the second line.
- Date values are automatically converted to numeric SAS date values and formatted for easy interpretation.
- The GUESSINGROWS= option can be used to increase the number of rows SAS scans to determine each column's type and length from the default 20 rows up to 32,767.

Importing a Comma-Delimited (CSV) File

```
PROC IMPORT DATAFILE="file.csv" DBMS=CSV
                    OUT=output-table <REPLACE>;
        <GUESSINGROWS=n>;
RUN;
```

Importing an Excel (XLSX) File

```
PROC IMPORT DATAFILE="file.xlsx" DBMS=XLSX
                    OUT=output-table <REPLACE>;
        <SHEET=sheet-name;>
RUN;
```

Last modified: Tuesday, August 28, 2018, 12:13 PM

Lesson 3 Syntax Review: Exploring and Validating Data

Exploring Data

- PROC PRINT lists all columns and rows in the input table by default. The OBS= data set option limits the number of rows listed. The VAR statement limits and orders the columns listed.

```
PROC PRINT DATA=input-table(OBS=n);
    VAR col-name(s);
RUN;
```

- PROC MEANS generates simple summary statistics for each numeric column in the input data by default. The VAR statement limits the variables to analyze.

```
PROC MEANS DATA=input-table;
    VAR col-name(s);
RUN;
```

- PROC UNIVARIATE also generates summary statistics for each numeric column in the data by default, but includes more detailed statistics related to distribution and extreme values. The VAR statement limits the variables to analyze.

```
PROC UNIVARIATE DATA=input-table;
    VAR col-name(s);
RUN;
```

- PROC FREQ creates a frequency table for each variable in the input table by default. You can limit the variables analyzed by using the TABLES statement.

```
PROC FREQ DATA=input-table;
    TABLES col-name(s) </ options>;
RUN;
```

Filtering Rows

- The WHERE statement is used to filter rows. If the expression is true, rows are read. If the expression is false, they are not.
- Character values are case sensitive and must be in quotation marks.
- Numeric values are not in quotation marks and must only include digits, decimal points, and negative signs.
- Compound conditions can be created with AND or OR.
- The logic of an operator can be reversed with the NOT keyword.
- When an expression includes a fixed date value, use the SAS date constant syntax: “ddmmmyyyy”d, where *dd* represents a 1- or 2-digit day, *mmm* represents a 3-letter month in any case, and *yyyy* represents a 2- or 4-digit year.

```
PROC procedure-name ... ;
    WHERE expression;
RUN;
```

WHERE Operators

= or EQ
^= or ~= or NE
> or GT
< or LT
>= or GE
<= or LT

SAS Date Constant

"ddMONyyyy"d

IN Operator

WHERE *col-name* **IN(***value-1<...,value-n>***);**
WHERE *col-name* **NOT IN (***value-1<...,value-n>***);**

Special WHERE Operators

WHERE *col-name* **IS MISSING;**
WHERE *col-name* **IS NOT MISSING;**
WHERE *col-name* **IS NULL;**
WHERE *col-name* **BETWEEN** *value-1* **AND** *value-2***;**
WHERE *col-name* **LIKE "***value***";**
WHERE *col-name* **=* "***value***";**

Filtering Rows with Macro Variables

%LET *macro-variable=value***;**

Example WHERE Statements with Macro Variables:

WHERE *numvar=&macrovar***;**
WHERE *charvar="&macrovar"***;**
WHERE *datevar="&macrovar"*d

- A macro variable stores a text string that can be substituted into a SAS program.
- The %LET statement defines the macro variable name and assigns a value.
- Macro variable names must follow SAS naming rules.
- Macro variables can be referenced in a program by preceding the macro variable name with an &.
- If a macro variable reference is used inside quotation marks, double quotation marks must be used.

Formatting Columns

- Formats are used to change the way values are displayed in data and reports.
- Formats do not change the underlying data values.
- Formats can be applied in a procedure using the FORMAT statement.
- Visit SAS Language Elements documentation to access a list of available SAS formats.

PROC PRINT DATA=*input-table***;**
    FORMAT *col-name(s) format***;**
RUN;

<$>*format-name*<*w***>.<***d***>**

Sorting Data and Removing Duplicates

- PROC SORT sorts the rows in a table on one or more character or numeric columns.
- The OUT= option specifies an output table. Without this option, PROC SORT changes the order of rows in the input table.
- The BY statement specifies one or more columns in the input table whose values are used to sort the rows. By default, SAS sorts in ascending order.

```
PROC SORT DATA=input-table <OUT=output-table>;
    BY <DESCENDING> col-name(s);
RUN;
```

- The DUPOUT= option creates an output table containing duplicates removed.
- The NODUPKEY option keeps only the first row for each unique value of the column(s) listed in the BY statement.
- Using _ALL_ in the BY statement sorts by all columns and ensures that entirely duplicate rows are adjacent in the sorted table and are removed.

```
PROC SORT DATA=input-table <OUT=output-table>
            NODUPKEY <DUPOUT=output-table>;
    BY <DESCENDING> col-name(s);
RUN;
```

```
PROC SORT DATA=input-table <OUT=output-table>
            NODUPKEY <DUPOUT=output-table>;
    BY _ALL_;
RUN;
```

Lesson 4 Syntax Review: Preparing Data

Reading and Filtering Data

- Creating a copy of data:

```
DATA output-table;
    SET input-table;
RUN;
```

- Filtering rows in the DATA step:

```
DATA output-table;
    SET input-table;
    WHERE expression;
RUN;
```

- Specifying columns to include in the output data set:

```
DROP col-name <col-name(s)>;
```

```
KEEP col-name <col-name(s)>;
```

- Formatting columns in the DATA step:

```
DATA output-table;
    SET input-table;
    FORMAT col-name format;
RUN;
```

Computing New Columns

- Using expressions to create new columns:

```
DATA output-table;
    SET input-table;
    new-column = expression;
RUN;
```

- The name of the column to be created or updated is listed on the left side of the equals sign.
- Provide an expression on the right side of the equal sign.
- SAS automatically defines the required attributes if the column is new – name, type, and length.
- A new numeric column has a length of 8.
- The length of a new character column is determined based on the length of the assigned string.
- Character strings must be quoted and are case sensitive.
- Creating character columns:

```
LENGTH char-column $ length;
```

- Using functions in expressions:

```
function(argument1, argument 2, ...);
```

```
DATA output-table;
    SET input-table;
    new-column=function(arguments);
RUN;
```

- Functions for calculating summary statistics (ignore missing values):

SUM(*num1, num2, ...***)**	calculates the sum
MEAN(*num1, num2, ...***)**	calculates the mean
MEDIAN(*num1, num2, ...***)**	calculates the median
RANGE(*num1, num2, ...***)**	calculates the range
MIN(*num1, num2, ...***)**	calculates the minimum
MAX(*num1, num2, ...***)**	calculates the maximum
N(*num1, num2, ...***)**	calculates the nonmissing
NMISS(*num1, num2, ...***)**	calculates the missing

- Character functions:

UPCASE(*char1***)** **LOWCASE(***char1***)**	changes letters in a character string to uppercase or lowercase
PROPCASE(*char1***)**	changes the first letter of each word to uppercase and other letters to lowercase
CATS(*char1, char2, ...***)**	concatenates character strings and removes leading and trailing blanks from each argument
SUBSTR(*char, position, <length>***)**	returns a substring from a character string

- Date functions that extract information from SAS date values:

MONTH(*sas-date-value***)**	returns a number from 1 through 12 that represents the month
YEAR(*sas-date-value***)**	returns the four-digit year
DAY(*sas-date-value***)**	returns a number from 1 through 31 that represents the day of the month
WEEKDAY(*sas-date-value***)**	returns a number from 1 through 7 that represents the day of the week (Sunday=1)
QTR(*sas-date-value***)**	returns a number from 1 through 4 that represents the quarter

- Date functions that create SAS date values:

TODAY()	returns the current date as a numeric SAS date value
MDY(*month*, *day*, *year***)**	returns SAS date value from month, day, and year values
YRDIF(*startdate*, *enddate*, 'AGE'**)**	calculates a precise age between two dates. There are various values for the third argument. However, "AGE" should be used for accuracy.

Conditional Processing

- Conditional processing with IF-THEN logic:

```
IF expression THEN statement;
```

- Conditional processing with IF-THEN-ELSE:

```
IF expression THEN statement;
<ELSE IF expression THEN statement;>
<ELSE IF expression THEN statement;>
ELSE statement;
```

- Processing multiple statements with IF-THEN-DO:

```
IF expression THEN DO;
  <executable statements>
END;
<ELSE IF expression THEN DO;
  <executable statements>
END;
ELSE DO;
  <executable statements>
END;
```

- After the IF-THEN-DO statement, list any number of executable statements.
- Close each DO block with an END statement.

Last modified: Thursday, August 30, 2018, 1:53 PM

Lesson 5 Syntax Review: Analyzing and Reporting on Data

Enhancing Reports with Titles, Footnotes, and Labels

- TITLE is a global statement that establishes a permanent title for all reports created in your SAS session.
- You can have up to 10 titles, in which case you would just use a number 1-10 after the keyword TITLE to indicate the line number. TITLE and TITLE1 are equivalent.
- Titles can be replaced with an additional TITLE statement with the same number. TITLE; clears all titles.
- You can also add footnotes to any report with the FOOTNOTE statement. The same rules for titles apply for footnotes.
- Labels can be used to provide more descriptive column headers. A label can include any text up to 256 characters.
- All procedures automatically display labels with the exception of PROC PRINT. You must add the LABEL option in the PROC PRINT statement.

```
TITLE<n> "title-text";
```

```
FOOTNOTE<n> "footnote-text";
```

```
LABEL col-name="label-text";
```

- To create a grouped report, first use PROC SORT to arrange the data by the grouping variable, and then use the BY statement in the reporting procedure.

```
PROC procedure-name;
    BY col-name;
RUN;
```

Creating Frequency Reports

- PROC FREQ creates a frequency table for each variable in the input table by default. You can limit the variables analyzed by using the TABLES statement.

```
PROC FREQ DATA=input-table;
    TABLES col-name(s) < / options>;
RUN;
```

- PROC FREQ statement options:

  ORDER=FREQ|FORMATTED|DATA
  NLEVELS

- TABLES statement options:

  NOCUM
  NOPERCENT
  PLOT=FREQPLOT (must turn on ODS graphics)
  OUT=*output-table*

- One or more TABLES statements can be used to define frequency tables and options.
- ODS graphics enable graph options to be used in the TABLES statement.
- WHERE, FORMAT, and LABEL statements can be used in PROC FREQ to customize the report.
- When you place an asterisk between two columns in the TABLES statement, PROC FREQ produces a two-way frequency or crosstabulation report.

```
PROC FREQ DATA=input-table;
    TABLES col-name*col-name < / options>;
RUN;
```

Creating Summary Statistics Reports

- Options in the PROC MEANS statement control the statistics included in the report.
- The CLASS statement specifies variables to group the data before calculating statistics.
- The WAYS statement specifies the number of ways to make unique combinations of class variables.

```
PROC MEANS DATA=input-table <stat-list> </ options>;
    VAR col-name(s);
    CLASS col-name(s);
    WAYS n;
RUN;
```

- The OUTPUT statement creates an output SAS table with summary statistics. Options in the OUTPUT statement determine the contents of the table.

```
PROC MEANS DATA=input-table <stat-list> </ options>;
    VAR col-name(s);
    CLASS col-name(s);
    WAYS n;
    OUTPUT OUT=output-table <statistic(col-name)=col-name> </ option(s)>;
RUN;
```

Last modified: Tuesday, August 28, 2018, 12:22 PM

Lesson 6 Syntax Review: Exporting Results

Exporting Data

- PROC EXPORT can export a SAS table to a variety of file formats outside SAS.

```
PROC EXPORT DATA=input-table OUTFILE="output-file"
            <DBMS=identifier> <REPLACE>;
RUN;
```

- The XLSX engine requires a license for SAS/ACCESS to PC Files.
- The XLSX engine can read and write data in Excel files.
- To write data to a new or existing Excel workbook, use the LIBNAME statement to assign a libref pointing to the Excel file. Use the libref when naming output tables. The table name is the worksheet label in the Excel file.

```
OPTIONS VALIDVARNAME=V7;
```

```
LIBNAME libref XLSX "path/file.xlsx";
```

- Use libref for output table(s).

```
LIBNAME libref CLEAR;
```

Exporting Reports

- The SAS Output Delivery System (ODS) is programmable and flexible, making it simple to automate the entire process of exporting reports to other formats.

```
ODS <destination> <destination-specifications>;

/* SAS code that produces output */

ODS <destination> CLOSE;
```

- Selected destinations:
  - CSVALL
  - PowerPoint
  - RTF
  - PDF
- Exporting results to Excel:

```
ODS EXCEL FILE="filename.xlsx" STYLE=style
          OPTIONS(SHEET_NAME='label');

/* SAS code that produces output */

ODS EXCEL OPTIONS(SHEET_NAME='label');

/* SAS code that produces output */

ODS EXCEL CLOSE;
```

- The ODS EXCEL destination creates an .XLSX file.
- By default, each procedure output is written to a separate worksheet with a default worksheet name. The default style is also applied.
- Use the STYLE= option on the ODS EXCEL statement to apply a different style.
- Use the OPTIONS(SHEET_NAME=’label’) on the ODS EXCEL statement to provide a custom label for each worksheet.

- Exporting results to PDF:

```
ODS PDF FILE="filename.pdf" STYLE=style
          STARTPAGE=NO PDFTOC=1;


ODS PROCLABEL"label";
/* SAS code that produces output */

ODS PDF CLOSE;
```

- The ODS PDF destination creates a .PDF file.
- The PDFTOC=n option controls the level of the expansion of the table of contents in PDF documents.
- The ODS PROCLABEL statement enables you to change a procedure label.

Last modified: Thursday, August 30, 2018, 1:54 PM

Lesson 7 Syntax Review: Using SQL in SAS

Reading and Filtering Data with SQL

- PROC SQL creates a report by default.
- The SELECT statement describes the query. List columns to include in the results after SELECT, separated by commas.
- The FROM clause lists the input table(s).
- The WHERE clause filters the input rows.
- The ORDER BY clause arranges rows based on the columns listed. The default order is ascending. Use DESC after a column name to reverse the sort sequence.
- PROC SQL ends with a QUIT statement.

```
PROC SQL;
SELECT col-name, col-name
      FROM input-table;
      WHERE expression;
      ORDER BY col-name <DESC>;
QUIT;
```

- Creating output data:

```
PROC SQL;
CREATE TABLE table-name AS
SELECT ...
```

- Deleting data:

```
PROC SQL;
DROP TABLE table-name;
QUIT;
```

Joining Tables Using SQL in SAS

- An SQL inner join combines matching rows between two tables.
- The two tables to be joined are listed in the FROM clause separated by INNER JOIN.
- The ON expression indicated how rows should be matched. The column names must be qualified as *table-name.col-name*.

```
FROM from table1 INNER JOIN table2
ON table1.column = table2.column
```

- Assign an alias (or nickname) to a table in the FROM clause by adding the keyword AS and the alias of your choice. Then you can use the alias in place of the full table name to qualify columns in the other clauses of a query.

```
FROM from table1 AS alias1 INNER JOIN table2 AS alias 2
```

Last modified: Thursday, August 30, 2018, 3:30 PM

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.

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*;

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;
```

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.

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.

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.

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.