PART I: SAS PROGRAMMING BOOK I & II SUMMAIRES

BOOK I

Summary of Lesson 1: Getting Started with SAS Programming InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries.
InstanceEndEditable
Topic Summaries
InstanceBeginEditable name="introText" *To go to the movie where you learned a task or concept, select a link.* InstanceEndEditable

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry
InstanceBeginEditable name="topicTitle" Exploring SAS InstanceEndEditable
InstanceBeginEditable name="topicText" SAS is a suite of business solutions and technologies to help organizations solve business problems. Base SAS is the centerpiece of all SAS software.

It can be useful to look at SAS capabilities in a simple framework:
- **Access data**: Using SAS, you can read any kind of data.
- **Manage data**: SAS gives you excellent data management capabilities
- **Analyze data**: For statistical analysis, SAS is the gold standard.
- **Present data**: You can use SAS to present your data meaningfully.

InstanceEndEditable
InstanceEndRepeatEntry InstanceBeginRepeatEntry
InstanceBeginEditable name="topicTitle" Understanding the SAS Programming Process InstanceEndEditable
InstanceBeginEditable name="topicText" Here is the overall process of programming in SAS.
1 **Define the business need**.
2 **Write a SAS program** based on the desired output, the necessary input, and the required processing.
3 **Run the program**.
4 **Review your results**.
5 If you find inaccuracies or errors, you **debug or modify** the program.

Depending on your results, you might need to repeat some of the steps.

The power of SAS is that you can use it to read any type of data, including the following three major file types:
- **Raw data files** contain data that has not been processed by any other computer program. They are text files that contain one record per line, and the record typically contains multiple fields. Raw data files aren't reports; they are unformatted text.
- **SAS data sets** are specific to SAS. A SAS data set is data in a form that SAS can understand. Like raw data files, SAS data sets contain data. But in SAS data sets, the data is created only by SAS and can be read only by SAS.

SAS program files contain SAS programming code. These instructions tell SAS how to process your data and what output to create. You can save and reuse SAS program files.

Summary of Lesson 2: Working with SAS Programs InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Exploring SAS Programs InstanceEndEditable

InstanceBeginEditable name="topicText"

A SAS program consists of DATA steps and PROC steps. A SAS programming step is comprised of a sequence of statements. Every step has a beginning and ending step boundary. SAS compiles and executes each step independently, based on the step boundaries.

A SAS program can also contain global statements, which are outside DATA and PROC steps, and typically affect the SAS session. A TITLE statement is a global statement. After it is defined, a title is displayed on every report, unless the title is cleared or canceled.

SAS statements usually begin with an identifying keyword, and always end with a semicolon. SAS statements are free format and can begin and end in any column. A single statement can span multiple lines, and there can be more than one statement per line. Unquoted values can be lowercase, uppercase, or mixed case. This flexibility can result in programs that are difficult to read.

Conventional formatting, also called *structured formatting*, uses consistent spacing to make a SAS program easy to read. To follow best practices, begin each statement on a new line, indent statements within each step, and indent subsequent lines in a multi-line statement.

Comments are used to document a program and to mark SAS code as non-executing text. There are two types of comments: *block comments* and *comment statements*.

/* *comment* ***/**
***** *comment statement***;**

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Diagnosing and Correcting Syntax Errors InstanceEndEditable

InstanceBeginEditable name="topicText"

Syntax errors occur when program statements do not conform to the rules of the SAS language. Common syntax errors include misspelled keywords, missing semicolons, and invalid options. SAS finds syntax errors during the compilation phase, before it executes the program. When SAS encounters a syntax error, it writes the following to the log: the word ERROR or WARNING, the location of the error, and an explanation of the error. You should always check the log, even if the program produces output.

Mismatched or unbalanced quotation marks are considered a syntax error. In some programming

environments, this results in a simple error message. In other environments, it is more difficult to identify this type of error.
InstanceEndEditable
InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable
InstanceBeginEditable name="samplePrograms"
Sample Programs
Submitting a SAS Program

```
data work.newsalesemps;
  set orion.sales;
  where Country='AU';
run;

title 'New Sales Employees';

proc print data=work.newsalesemps;
run;

proc means data=work.newsalesemps;
  class Job_Title;
  var Salary;
run;

title;
```

Adding Comments to Your SAS Programs

```
*This program uses the data set orion.sales to create work.newsalesemps.;
data work.newsalesemps;
  set orion.sales;
  where Country='US';
run;

 /*
proc print data=work.newsalesemps;
```

```
run;*/
proc means data=work.newsalesemps;
  class Gender;
  var Salary/*numeric variable*/;
run;
```

Viewing and Correcting Syntax Errors

```
daat work.newsalesemps;
  length First_Name $ 12
       Last_Name $ 18 Job_Title $ 25;
  infile "&path/newemps.csv" dlm=',';
  input First_Name $ Last_Name $
      Job_Title $ Salary;
run;

proc print data=work.newsalesemps
run;

proc means data=work.newsalesemps average max;
  class Job_Title;
  var Salary;
run;
```

Summary of Lesson 3: Accessing Data InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Accessing SAS Libraries InstanceEndEditable

InstanceBeginEditable name="topicText"

SAS data sets are stored in SAS libraries. A SAS library is a collection of one or more SAS files that are recognized by SAS. SAS automatically provides one temporary and at least one permanent SAS library in every SAS session.

Work is a temporary library that is used to store and access SAS data sets for the duration of the session. **Sasuser** and **sashelp** are permanent libraries that are available in every SAS session.

You refer to a SAS library by a library reference name, or libref. A libref is a shortcut to the physical location of the SAS files.

All SAS data sets have a two-level name that consists of the libref and the data set name, separated by a period. Data sets in the **work** library can be referenced with a one-level name, consisting of only the data set name, because **work** is the default library. Data sets in permanent libraries must be referenced with a two-level name.

You can create and access your own SAS libraries. User-defined libraries are permanent but are not automatically available in a SAS session. You must assign a libref to a user-created library to make it available. You use a LIBNAME statement to associate the libref with the physical location of the library, that is, the physical location of your data. You can submit the LIBNAME statement alone at the start of a SAS session, or you can store it in a SAS program so that the SAS library is defined each time the program runs. If your program needs to reference data sets in multiple locations, you can use multiple LIBNAME statements.

LIBNAME *libref 'SAS-library'* <*options*>**;**

Use PROC CONTENTS with *libref*._ALL_ to display the contents of a SAS library. The report will list all the SAS files contained in the library, as well as the descriptor portion of each data

set in the library. Use the NODS option in the PROC CONTENTS statement to suppress the descriptor information for each data set.

```
PROC CONTENTS DATA=libref._ALL_
NODS;  RUN;
```

After associating a libref with a permanent library, you can write a PROC PRINT step to display a SAS data set within the library.

```
PROC PRINT DATA=libref.SAS-data-set;
RUN;
```

In an interactive SAS session, a libref remains in effect until you cancel it, change it, or end your SAS session. To cancel a libref, you submit a LIBNAME statement with the CLEAR option. This clears or disassociates a libref that was previously assigned. To specify a different physical location, you submit a LIBNAME statement with the same libref name but with a different filepath.

```
LIBNAME libref
CLEAR;
```

When a SAS session ends, everything in the **work** library is deleted. The librefs are also deleted. Remember that the contents of permanent libraries still exist in in the operating environment, but each time you start a new SAS session, you must resubmit the LIBNAME statement to redefine a libref for each user-created library that you want to access.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Examining SAS Data Sets InstanceEndEditable

InstanceBeginEditable name="topicText"

SAS data sets are specially structured data files that SAS creates and that only SAS can read. A SAS data set is displayed as a table composed of variables and observations. A SAS data set contains a descriptor portion and a data portion.

The descriptor portion contains general information about the data set (such as the data set name and the number of observations) and information about the variable attributes (such as name, type, and length). There are two types of variables: character and numeric. A character variable can store any value and can be up to 32,767 characters long. Numeric variables store numeric values in floating point or binary representation in 8 bytes of storage by default. Other attributes

include formats, informats, and labels. You can use PROC CONTENTS to browse the descriptor portion of a data set.

PROC CONTENTS DATA=*libref*.*SAS-data-set*; **RUN;**

The data portion contains the data values. Data values are either character or numeric. A valid value must exist for every variable in every observation in a SAS data set. A missing value is a valid value in SAS. A missing character value is displayed as a blank, and a missing numeric value is displayed as a period. You can specify an alternate character to print for missing numeric values using the MISSING= SAS system option. You can use PROC PRINT to display the data portion of a SAS data set.

SAS variable and data set names must be 1 to 32 characters in length and start with a letter or underscore, followed by letters, underscores, and numbers. Variable names are not case sensitive.

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Accessing a SAS Library

```
/*Replace filepath with the physical location of your practice files.*/
%let path=filepath;
libname orion "&path";
```

Browsing a Library

```
proc contents data=orion._all_;
run;

proc contents data=orion._all_ nods;
run;
```

Viewing a Data Set with PROC PRINT

```
proc print data=orion.country;
```

run;

Viewing the Descriptor Portion of a Data Set

proc contents data=orion.sales;

run;

Viewing the Data Portion of a SAS Data Set

proc print data=orion.sales;

run;

Summary of Lesson 4: Producing Detail Reports InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Subsetting Report Data InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use the VAR statement in a PROC PRINT step to subset the variables in a report. You specify the variables to include and list them in the order in which they are to be displayed.

You can use the SUM statement in a PROC PRINT step to calculate and display report totals for the requested numeric variables.

```
PROC PRINT DATA=SAS-data-set;
    VAR variable(s);        SUM
variable(s);
RUN;
```

The WHERE statement in a PROC PRINT step subsets the observations in a report. When you use a WHERE statement, the output contains only the observations that meet the conditions specified in the WHERE expression. This expression is a sequence of operands and operators that form a set of instructions that define the condition. The operands can be constants or variables. Remember that variable operands must be defined in the input data set. Operators include comparison, arithmetic, logical, and special WHERE operators.

```
WHERE where-expression;
```

You can use the ID statement in a PROC PRINT step to specify a variable to print at the beginning of the row instead of an observation number. The variable that you specify replaces the Obs column.

```
ID
variable(s);
```

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Sorting and Grouping Report Data InstanceEndEditable

InstanceBeginEditable name="topicText"

The SORT procedure sorts the observations in a data set. You can sort on one variable or multiple variables, sort on character or numeric variables, and sort in ascending or descending order. By default, SAS replaces the original SAS data set unless you use the OUT= option to specify an output data set. PROC SORT does not generate printed output.

Every PROC SORT step must include a BY statement to specify one or more BY variables. These are variables in the input data set whose values are used to sort the data. By default, SAS sorts in ascending order, but you can use the keyword DESCENDING to specify that the values of a variable are to be sorted in descending order. When your SORT step has multiple BY variables, some variables can be in ascending and others in descending order.

You can also use a BY statement in PROC PRINT to display observations grouped by a particular variable or variables. The groups are referred to as BY groups. Remember that the input data set must be sorted on the variables specified in the BY statement.

PROC SORT DATA=*input-SAS-data-set*
<OUT=*ouput-SAS-data-set*>**;**
BY <DESCENDING> *by-variable(s)***;**
RUN;

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Enhancing Reports InstanceEndEditable

InstanceBeginEditable name="topicText"

You can enhance a report by adding titles, footnotes, and column labels. Use the global TITLE statement to define up to 10 lines of titles to be displayed at the top of the output from each procedure. Use the global FOOTNOTE statement to define up to 10 lines of footnotes to be displayed at the bottom of the output from each procedure.

TITLE*n* '*text*'**;**
FOOTNOTE*n* '*text*'**;**

Titles and footnotes remain in effect until you change or cancel them, or until you end your SAS

session. Use a null TITLE statement to cancel all titles, and a null FOOTNOTE statement to cancel all footnotes.

Use the LABEL statement in a PROC PRINT step to define temporary labels to display in the report instead of variable names. Labels can be up to 256 characters in length. Most procedures use labels automatically, but PROC PRINT does not. Use the LABEL option in the PROC PRINT statement to tell SAS to display the labels. Alternatively, the SPLIT= option tells PROC PRINT to use the labels and also specifies a split character to control line breaks in column headings.

PROC PRINT DATA=*SAS-data-set* **LABEL;**
LABEL *variable*='*label*'
variable='*label*' ...
; RUN;

SPLIT='*split-character*'**;**

InstanceEndEditable
InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable
InstanceBeginEditable name="samplePrograms"

Sample Programs

Subsetting Your Report

```
proc print data=orion.sales;
  var Last_Name First_Name Salary;
  sum Salary;
run;
```

Selecting Observations

```
proc print data=orion.sales noobs;
  var Last_Name First_Name Salary Country;
  where Country='AU' and Salary<25500;
run;
```

Using the CONTAINS Operator

```
proc print data=orion.sales noobs;
  var Last_Name First_Name Country Job_Title;
  where Country='AU' and Job_Title contains 'Rep';
run;
```

Subsetting Observations and Replacing the Obs Column

```
proc print data=orion.customer_dim;
  where Customer_Age=21;
  id Customer_ID;
  var Customer_Name
      Customer_Gender Customer_Country
      Customer_Group Customer_Age_Group
      Customer_Type;
run;
```

Sorting a Data Set

```
proc sort data=orion.sales
        out=work.sales_sort;
  by Salary;
run;

proc print data=work.sales_sort;
run;
```

Sorting a Data Set by Multiple Variables

```
proc sort data=orion.sales
        out=work.sales2;
  by Country descending Salary;
run;

proc print data=work.sales2;
run;
```

Grouping Observations in Reports

```
proc sort data=orion.sales
        out=work.sales2;
  by Country descending Salary;
run;

proc print data=work.sales2;
  by Country;
run;
```

Displaying Titles and Footnotes in a Report

```
title1 'Orion Star Sales Staff';
title2 'Salary Report';
footnote1 'Confidential';

proc print data=orion.sales;
  var Employee_ID Last_Name Salary;
run;

proc print data=orion.sales;
  var Employee_ID First_Name Last_Name Job_Title Hire_Date;
run;
```

Changing and Canceling Titles and Footnotes

```
title1 'Orion Star Sales Staff';
title2 'Salary Report';
footnote1 'Confidential';

proc print data=orion.sales;
  var Employee_ID Last_Name Salary;
run;

title1 'Employee Information';
proc print data=orion.sales;
```

```
  var Employee_ID First_Name Last_Name Job_Title Hire_Date;
run;
```

Displaying Labels in a Report

```
title1 'Orion Star Sales Staff';
title2 'Salary Report';
footnote1 'Confidential';

proc print data=orion.sales label;
  var Employee_ID Last_Name Salary;
  label Employee_ID = 'Sales ID'
        Last_Name = 'Last Name'
        Salary = 'Annual Salary';
run;
title;
footnote;
```

Summary of Lesson 5: Formatting Data Values InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using SAS Formats InstanceEndEditable

InstanceBeginEditable name="topicText"

A format is an instruction that tells SAS how to display data values in output reports. You can add a FORMAT statement to a PROC PRINT step to specify temporary SAS formats that control how values appear in the report. There are many existing SAS formats that you can use. Character formats begin with a dollar sign, but numeric formats do not.

FORMAT *variable(s) format*;

SAS stores date values as the number of days between January 1, 1960, and a specific date. To make the dates in your report recognizable and meaningful, you must apply a SAS date format to the SAS date values.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Creating and Applying User-Defined Formats InstanceEndEditable

InstanceBeginEditable name="topicText"

You can create your own user-defined formats. When you create a user-defined format, you don't associate it with a particular variable or data set. Instead, you create it based on values that you want to display differently. The formats will be available for the remainder of your SAS session. You can apply user-defined formats to a specific variable in a PROC PRINT step.

You use the FORMAT procedure to create a format. You assign a format name that can have up to 32 characters. The name of a character format must begin with a dollar sign, followed by a letter or underscore, followed by letters, numbers, and underscores. Names for numeric formats must begin with a letter or underscore, followed by letters, numbers, and underscores. A format

name cannot end in a number and cannot be the name of a SAS format.

You use a VALUE statement in a PROC FORMAT step to specify the way that you want the data values to appear in your output. You define value-range sets to specify the values to be formatted and the formatted values to display instead of the stored value or values. The value portion of a value-range set can include an individual value, a range of values, a list of values, or a keyword. The keyword OTHER is used to define a value to display if the stored data value does not match any of the defined value-ranges.

```
PROC FORMAT;
    VALUE format-name value-or-range1='formatted-value1'
                      value-or-range2='formatted-value2'
                      ...;
RUN;
```

When you define a numeric format, it is often convenient to use numeric ranges in the value-range sets. Ranges are inclusive by default. To exclude the endpoints, use a less-than symbol after the low end of the range or before the high end.

The LOW and HIGH keywords are used to define a continuous range when the lowest and highest values are not known. Remember that for character values, the LOW keyword treats missing values as the lowest possible values. However, for numeric values, LOW does not include missing values.

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Applying Temporary Formats

```
proc print data=orion.sales label noobs;
  where Country='AU' and
      Job_Title contains 'Rep';
  label Job_Title='Sales Title'
      Hire_Date='Date Hired';
```

```
  var Last_Name First_Name Country Job_Title
      Salary Hire_Date;
run;

proc print data=orion.sales label noobs;
  where Country='AU' and
        Job_Title contains 'Rep';
  label Job_Title='Sales Title'
        Hire_Date='Date Hired';
  format Hire_Date mmddyy10. Salary dollar8.;
  var Last_Name First_Name Country Job_Title
      Salary Hire_Date;
run;
```

Specifying a User-Defined Format for a Character Variable

```
proc format;
  value $ctryfmt 'AU'='Australia'
                 'US'='United States'
                 other='Miscoded';
run;

proc print data=orion.sales label;
  var Employee_ID Job_Title Salary
      Country Birth_Date Hire_Date;
  label Employee_ID='Sales ID'
        Job_Title='Job Title'
        Salary='Annual Salary'
        Birth_Date='Date of Birth'
        Hire_Date='Date of Hire';
  format Salary dollar10.
         Birth_Date Hire_Date monyy7.
         Country $ctryfmt.;
```

```
run;
```

Specifying a User-Defined Format for a Numeric Variable

```
proc format;
   value tiers low-<50000='Tier1'
               50000-100000='Tier2'
               100000<-high='Tier3';
run;

proc print data=orion.sales;
   var Employee_ID Job_Title Salary
       Country Birth_Date Hire_Date;
   format Birth_Date Hire_Date monyy7.
          Salary tiers.;
run;
```

Summary of Lesson 6: Reading SAS Data Sets InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Reading a SAS Data Set InstanceEndEditable

InstanceBeginEditable name="topicText"

You use a DATA step to create a new SAS data set from an existing SAS data set. The DATA step begins with a DATA statement, which provides the name of the SAS data set to create. Include a SET statement to name the existing SAS data set to be read in as input.

You use the WHERE statement to subset the input data set by selecting only the observations that meet a particular condition. To subset based on a SAS date value, you can use a SAS date constant in the WHERE expression. SAS automatically converts a date constant to a SAS date value.

```
DATA output-SAS-data-set;
    SET input-SAS-data-set;
    WHERE where-expression;
RUN;
```

You use an assignment statement to create a new variable. The assignment statement evaluates an expression and assigns the resulting value to a new or existing variable. The expression is a sequence of operands and operators. If the expression includes arithmetic operators, SAS performs the numeric operations based on priority, as in math equations. You can use parentheses to clarify or alter the order of operations.

```
variable=expression;
```

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Customizing a SAS Data Set

InstanceEndEditable

InstanceBeginEditable name="topicText"

By default, the SET statement reads all of the observations and variables from the input data set and writes them to the output data set. You can customize the new data set by selecting only the observations and variables that you want to include. You can use a WHERE statement to select the observations, as long as the variables included in the condition come from the input data set. You can use a DROP statement to list the variables to exclude from the new data set, or use a KEEP statement to list the variables to include. If you use a KEEP statement, you must include every variable to be written, including any new variables.

DROP *variable-list***;**

KEEP *variable-list***;**

SAS processes the DATA step in two phases: the compilation phase and the execution phase. You can subset the original data set with a WHERE statement for variables that are defined in the input data set, and a subsetting IF statement for new variables that are created in the DATA step. Remember that, although IF expressions are similar to WHERE expressions, you cannot use special WHERE operators in IF expressions.

IF *expression***;**

To subset observations in a PROC step, you must use a WHERE statement. You cannot use a subsetting IF statement in a PROC step. To subset observations in a DATA step, you can always use a subsetting IF statement. However, a WHERE statement can make your DATA step more efficient because it subsets on input.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Adding Permanent Attributes

InstanceEndEditable

InstanceBeginEditable name="topicText"

When you use the LABEL statement in a DATA step, SAS permanently associates the labels to the variables by storing the labels in the descriptor portion of the data set. Using a FORMAT statement in a DATA step permanently associates formats with variables. The format information is also stored in the descriptor portion of the data set. You can use PROC CONTENTS to view the label and format information. PROC PRINT does not display permanent labels unless you use the LABEL or SPLIT= option.

LABEL *variable='label'*
variable='label'
... **;**

FORMAT *variable(s) format ...***;**

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Subsetting Observations in the DATA Step

```
proc print data=orion.sales;
run;

data work.subset1;
  set orion.sales;
  where Country='AU' and
      Job_Title contains 'Rep';
run;

proc print data=work.subset1;
run;
```

Subsetting Observations and Creating a New Variable

```
data work.subset1;
  set orion.sales;
  where Country='AU' and
      Job_Title contains 'Rep' and
      Hire_Date<'01jan2000'd;
  Bonus=Salary*.10;
run;
```

```
proc print data=work.subset1 noobs;
  var First_name Last_Name Salary
     Job_Title Bonus Hire_Date;
  format Hire_Date date9.;
run;
```

Subsetting Variables in a DATA Step: DROP and KEEP

```
data work.subset1;
  set orion.sales;
  where Country='AU' and
       Job_Title contains 'Rep';
  Bonus=Salary*.10;
  drop Employee_ID Gender Country Birth_Date;
run;

proc print data=work.subset1;
run;

data work.subset1;
  set orion.sales;
  where Country='AU' and
       Job_Title contains 'Rep';
  Bonus=Salary*.10;
  keep First_Name Last_Name Salary Job_Title Hire_Date Bonus;
run;

proc print data=work.subset1;
run;
```

Selecting Observations by Using the Subsetting IF Statement

```
data work.auemps;
  set orion.sales;
```

```
  where Country='AU';
  Bonus=Salary*.10;
  if Bonus>=3000;
run;

proc print data=work.auemps;
run;
```

Adding Permanent Labels to a SAS Data Set

```
data work.subset1;
  set orion.sales;
  where Country='AU' and
        Job_Title contains 'Rep';
  Bonus=Salary*.10;
  label Job_Title='Sales Title'
        Hire_Date='Date Hired';
  drop Employee_ID Gender Country Birth_Date;
run;

proc contents data=work.subset1;
run;

proc print data=work.subset1 label;
run;
```

Adding Permanent Formats to a SAS Data Set

```
data work.subset1;
  set orion.sales;
  where Country='AU' and
        Job_Title contains 'Rep';
  Bonus=Salary*.10;
  label Job_Title='Sales Title'
        Hire_Date='Date Hired';
```

```
   format Salary Bonus dollar12.
        Hire_Date ddmmyy10.;
   drop Employee_ID Gender Country Birth_Date;
run;


proc contents data=work.subset1;
run;

proc print data=work.subset1 label;
run;
```

Summary of Lesson 7: Reading Spreadsheet and Database Data InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Reading Spreadsheet Data InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use SAS/ACCESS Interface to PC Files to read the worksheets within a Microsoft Excel workbook. After you submit a SAS/ACCESS LIBNAME statement, SAS treats the Excel workbook as if it were a SAS library and treats the worksheets as if they were SAS data sets within that library. You submit a LIBNAME statement to specify a libref, an engine name, and the location and name of the workbook. The engine tells SAS the type of input file and which engine to use to read the input data.

LIBNAME *libref* <*engine*> "*workbook-name*" <*options*>**;**
LIBNAME *libref* <*engine*> <**PATH=**> "*workbook-name*" <*options*>**;**

When you browse the library, you might see worksheets and named ranges. Worksheet names end with a dollar sign, and named ranges do not. Because the dollar sign is a special character, you must use a SAS name literal when you refer to a worksheet in a program.

libref.'*worksheetname*$'**n**

When you assign a libref to an Excel workbook in SAS, the workbook cannot be opened in Excel. To disassociate a libref, you submit a LIBNAME statement specifying the libref and the CLEAR option. SAS disconnects from the data source and closes any resources that are associated with the connection.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Reading Database Data InstanceEndEditable

InstanceBeginEditable name="topicText"

You can also read database tables as if they were SAS data sets by using the LIBNAME

statement supported by SAS/ACCESS Interface to Oracle. This SAS/ACCESS LIBNAME statement includes a libref, an engine name, and additional connection options that are site- and installation-specific. After you submit the LIBNAME statement, SAS treats the Oracle database as if it were a SAS library, and any table in the database can be referenced using a SAS two-level name, as if it were a SAS data set.

LIBNAME *libref engine <SAS/ACCESS Oracle options>***;**

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Accessing Excel Worksheets in SAS

```
libname orionx pcfiles path="&path/sales.xls";

proc contents data=orionx._all_;
run;
```

Printing an Excel Worksheet

```
proc print data=orionx.'Australia$'n;
run;

proc print data=orionx.'Australia$'n noobs;
  where Job_Title ? 'IV';
  var Employee_ID Last_Name Job_Title Salary;
run;
```

Creating a SAS Data Set from an Excel Worksheet

```
libname orionx pcfiles path="&path/sales.xls";

data work.subset;
  set orionx.'Australia$'n;
  where Job_Title contains 'Rep';
```

```
   Bonus=Salary*.10;
   label Job_Title='Sales Title'
         Hire_Date='Date Hired';
   format Salary comma10. Hire_Date mmddyy10.
          Bonus comma8.2;
run;

proc contents data=work.subset;
run;

proc print data=work.subset label;
run;

libname orionx clear;
```

Summary of Lesson 8: Reading Raw Data Files InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Introduction to Reading Raw Data Files InstanceEndEditable

InstanceBeginEditable name="topicText"

A raw data file is an external text file that contains one record per line, and a record typically contains multiple fields. The fields can be delimited or arranged in fixed columns. Typically, there are no column headings. The file is usually described in an external document called a *record layout*.

In order for SAS to read a raw data file, you must specify the location of each data value in the record, along with the names and types of the SAS variables in which to store the values. Three styles of input are available: list input, column input, and formatted input. List input reads delimited files, and column and formatted input read fixed column files. List input and formatted input can read both standard and nonstandard data, and column input can read only standard data. In this course, we are reading delimited raw data files, so list input is used.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Reading Standard Delimited Data InstanceEndEditable

InstanceBeginEditable name="topicText"

You use a DATA step with INFILE and INPUT statements to read data from a raw data file. The INFILE statement identifies the name and location of the input file. You use the DLM= option if the file has a delimiter other than a blank space. The INPUT statement tells SAS how to read the values, and specifies the name and type for each variable to be created. In the INPUT statement, you list the variables in the order that the corresponding values appear in the raw data file, from

left to right. You specify character variables by adding a dollar sign after the variable name. With list input, the default length for all variables is 8 bytes, regardless of type.

DATA *output-SAS-data-set-name*; **INFILE** '*raw-data-file-name*' **DLM=**'*delimiter*'; **INPUT** *variable1 <$> variable2 <$> ... variableN <$>*; **RUN;**

SAS processes the DATA step in two phases: compilation and execution. During compilation, SAS creates an input buffer to hold a record from the raw data file. The input buffer is an area of memory that SAS creates only when reading raw data, not when reading a SAS data set. SAS also creates the PDV, an area of memory where an observation is built. In addition to the variables named in the INPUT statement, SAS creates the iteration counter, **_N_**, and the error indicator, **_ERROR_**, in the PDV. These temporary variables are not written to the output data set. At the end of the compilation, SAS creates the descriptor portion of the output data set.

At the start of the execution phase, SAS initializes the PDV and then reads the first record from the raw data file into the input buffer. It scans the input buffer from non-delimiter to delimiter and assigns each value to the corresponding variable in the PDV. SAS ignores delimiters. At the bottom of the DATA step, SAS writes the values from the PDV to the new SAS data set and then returns to the top of the DATA step.

Truncation often occurs with list input, because character variables are created with a length of 8 bytes, by default. You can use a LENGTH statement before the INPUT statement in a DATA step to explicitly define the length of character variables. Numeric variables can be included in the LENGTH statement to preserve the order of variables, but you need to specify a length of 8 for each numeric variable.

LENGTH *variable(s) <$> length*;

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Reading Nonstandard Delimited Data InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use modified list input to read standard and nonstandard data from a delimited raw data file. Modified list input uses an informat and a colon format modifier for each field to be read. An informat tells SAS how to read data values, including the number of characters. When SAS

reads character data, a standard character informat, such as $12., is often used instead of a LENGTH statement. With list input, the data fields vary in length. The colon format modifier tells SAS to ignore the specified length when it reads data values, and instead to read only until it reaches a delimiter. Omitting the colon format modifier is likely to result in data errors.

INPUT *variable <$> variable <:informat>***;**

An informat is required to read nonstandard numeric data, such as calendar dates, and numbers with dollar signs and commas. Many SAS informats are available for nonstandard numeric values. Every informat has a width, whether stated explicitly or set by default.

When reading a raw data file, you can use a DROP or KEEP statement to write a subset of variables to the new data set. You must use a subsetting IF statement to select observations, because the variables are not coming from an input SAS data set. You can use LABEL and FORMAT statements to permanently store label and format information in the new data set.

A DATA step can also read instream data, which is data that is within a SAS program. To specify instream data, you use a DATALINES statement in a DATA step, followed by the lines of data, followed by a null statement.

DATALINE
S;
<data line
1>
<data line
2>
...
;

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Validating Data InstanceEndEditable

InstanceBeginEditable name="topicText"

When data values in the input file aren't appropriate for the INPUT statement in a program, a data error occurs during program execution. SAS records the error in the log by writing a note about the error, along with a ruler and the contents of the input buffer and the PDV. The variable **_ERROR_** is set to *1*, a missing value is assigned to the corresponding variable, and execution

continues.

You can use the DSD option in the INFILE statement if data values are missing in the middle of a record. When you use the DSD option, SAS assumes that the file is comma delimited, treats consecutive delimiters as missing data, and allows embedded delimiters in a field that is enclosed in quotation marks. If you have missing data values at the end of a record, you can use the MISSOVER option in the INFILE statement. SAS sets the variable values to missing.

INFILE '*raw-data-file-name*' <**DLM**=> <**DSD**> <**MISSOVER**>**;**

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Creating a SAS Data Set from a Delimited Raw Data File

```
data work.sales1;
  infile "&path/sales.csv" dlm=',';
  input Employee_ID First_Name $
        Last_Name $ Gender $ Salary
        Job_Title $ Country $;
run;

proc print data=work.sales1;
run;
```

Specifying the Lengths of Variables Explicitly

```
data work.sales2;
  length First_Name $ 12 Last_Name $ 18
         Gender $ 1 Job_Title $ 25
         Country $ 2;
  infile "&path/sales.csv" dlm=',';
  input Employee_ID First_Name $ Last_Name $
```

```
      Gender $ Salary Job_Title $ Country $;
run;

proc contents data=work.sales2;
run;

proc print data=work.sales2;
run;



data work.sales2;
  length Employee_ID  8 First_Name $ 12
       Last_Name $ 18 Gender $ 1
       Salary  8 Job_Title $ 25
       Country $ 2;
  infile "&path/sales.csv" dlm=',';
  input Employee_ID First_Name $ Last_Name $
      Gender $ Salary Job_Title $ Country $;
run;

proc contents data=work.sales2 varnum;
run;

proc print data=work.sales2;
run;
```

Specifying Informats in the INPUT Statement

```
data work.sales2;
  infile "&path/sales.csv" dlm=',';
  input Employee_ID First_Name :$12. Last_Name :$18.
      Gender :$1. Salary Job_Title :$25. Country :$2.
      Birth_Date :date. Hire_Date :mmddyy.;
```

```
run;

proc print data=work.sales2;
run;
```

Subsetting and Adding Permanent Attributes

```
data work.subset;
  infile "&path/sales.csv" dlm=',';
  input Employee_ID First_Name :$12.
        Last_Name :$18. Gender :$1. Salary
        Job_Title :$25. Country :$2.
        Birth_Date :date. Hire_Date :mmddyy.;
  if Country='AU';
  keep First_Name Last_Name Salary
       Job_Title Hire_Date;
  label Job_Title='Sales Title'
        Hire_Date='Date Hired';
  format Salary dollar12. Hire_Date monyy7.;
run;

proc print data=work.subset label;
run;
```

Reading Instream Data

```
data work.newemps;
  input First_Name $ Last_Name $
        Job_Title $ Salary :dollar8.;
  datalines;
Steven Worton Auditor $40,450
Merle Hieds Trainee $24,025
Marta Bamberger Manager $32,000
;
```

```
proc print data=work.newemps;
run;

data work.newemps2;
  infile datalines dlm=',';
  input First_Name $ Last_Name $
      Job_Title $ Salary :dollar8.;
  datalines;
Steven,Worton,Auditor,$40450
Merle,Hieds,Trainee,$24025
Marta,Bamberger,Manager,$32000
;

proc print data=work.newemps2;
run;
```

Reading a Raw Data File That Contains Data Errors

```
data work.sales4;
  infile "&path/sales3inv.csv" dlm=',';
  input Employee_ID First $ Last $
      Job_Title $ Salary Country $;
run;

proc print data=work.sales4;
run;
```

Reading a Raw Data File That Contains Missing Data

```
data work.contacts;
  length Name $ 20 Phone Mobile $ 14;
  infile "&path/phone2.csv" dsd;
  input Name $ Phone $ Mobile $;
run;
```

```
proc print data=work.contacts noobs;
run;
```

Reading a Raw Data File Using the MISSOVER Option

```
data work.contacts2;
  infile "&path/phone.csv" dlm=',' missover;
  input Name $ Phone $ Mobile $;
run;

proc print data=contacts2 noobs;
run;


data work.contacts2;
  length Name $ 20 Phone Mobile $ 14;
  infile "&path/phone.csv" dlm=',' missover;
  input Name $ Phone $ Mobile $;
run;

proc print data=contacts2 noobs;
run;
```

Summary of Lesson 9: Manipulating Data InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using SAS Functions InstanceEndEditable

InstanceBeginEditable name="topicText"

You use an assignment statement in a DATA step to evaluate an expression and assign the result to a new or existing variable. The expression on the right side of an assignment statement can include calls to SAS functions. A SAS function is a routine that accepts arguments and returns a value.

variable=expression;

The SUM function, a descriptive statistics function, returns the sum of its arguments and ignores missing values. The arguments can be numeric constants, numeric variables, or arithmetic expressions, but the arguments must be numeric values and must be enclosed in parentheses and separated with commas. The parentheses are required, even if no arguments are passed to the function.

SUM(*argument1,argument2, ...***)**

In addition to descriptive statistics functions, many SAS date functions are available. Some of these functions create SAS dates, and others extract information from SAS dates. The MONTH function extracts and returns the numeric month from a SAS date.

MONTH(*SAS-date***)**

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Conditional Processing InstanceEndEditable

InstanceBeginEditable name="topicText"

The IF-THEN statement is a conditional statement. It executes a SAS statement for observations that meet specific conditions. The statement includes an expression and a SAS program

statement. The expression defines a condition that must be true for the statement to be executed. The expression is evaluated during each iteration of the DATA step. If the condition is true, the statement following the THEN statement is executed; otherwise, SAS skips the statement.

```
IF expression THEN statement;
```

A program often includes a sequence of IF statements with mutually exclusive conditions. When SAS encounters a true condition in this series, evaluating the other conditions isn't necessary. You can use the ELSE statement to specify an alternative action to be performed when the condition in an IF-THEN statement is false. This increases the efficiency of the program.

You can use the logical operators AND and OR to combine conditions in an IF expression. You use the AND operator when both conditions must be true, and you use the OR operator when only one of the conditions must be true. An optional final ELSE statement can be used at the end of a series of IF-THEN/ELSE statements. The statement following the final ELSE executes if none of the IF expressions is true.

Use a DO group with an IF-THEN or an ELSE statement when multiple statements must be executed based on one condition. The DO group consists of a DO statement, the SAS statements to be executed, and an END statement. Each DO statement must have a corresponding END statement.

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

Truncation can occur when new variables are assigned values within conditional program statements. During compilation, SAS creates a variable in the PDV the first time it encounters the variable in the program. If this is in conditional code, be sure that it is created with a length long enough to store all possible values. It is a best practice to use a LENGTH statement to explicitly define the length.

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Creating Variables by Using Functions

```
data work.comp;
  set orion.sales;
  Bonus=500;
  Compensation=sum(Salary,Bonus);
  BonusMonth=month(Hire_Date);
run;

proc print data=work.comp;
  var Employee_ID First_Name Last_Name
      Salary Bonus Compensation BonusMonth;
run;
```

Assigning Values Conditionally

```
data work.comp;
  set orion.sales;
  if Job_Title='Sales Rep. IV' then
    Bonus=1000;
  if Job_Title='Sales Manager' then
    Bonus=1500;
  if Job_Title='Senior Sales Manager' then
    Bonus=2000;
  if Job_Title='Chief Sales Officer' then
    Bonus=2500;
run;

proc print data=work.comp;
```

```
  var Last_Name Job_Title Bonus;
run;
```

Using Compound Conditions

```
data work.comp;
  set orion.sales;
  if Job_Title='Sales Rep. III' or
    Job_Title='Sales Rep. IV' then
    Bonus=1000;
  else if Job_Title='Sales Manager' then
    Bonus=1500;
  else if Job_Title='Senior Sales Manager' then
    Bonus=2000;
  else if Job_Title='Chief Sales Officer' then
    Bonus=2500;
  else Bonus=500;
run;

proc print data=work.comp;
  var Last_Name Job_Title Bonus;
run;
```

Using IF-THEN/ELSE Statements

```
data work.bonus;
  set orion.sales;
  if Country='US' then Bonus=500;
  else Bonus=300;
run;

proc print data=work.bonus;
  var First_Name Last_Name Country Bonus;
run;
```

Creating Two Variables Conditionally

```
data work.bonus;
  set orion.sales;
  if Country='US' then
    do;
      Bonus=500;
      Freq='Once a Year';
    end;
  else if Country='AU' then
    do;
      Bonus=300;
      Freq='Twice a Year';
    end;
run;
proc print data=work.bonus;
  var First_Name Last_Name Country Bonus Freq;
run;
```

Adjusting the Program

```
data work.bonus;
  set orion.sales;
  length Freq $ 12;
  if Country='US' then
    do;
      Bonus=500;
      Freq='Once a Year';
    end;
  else if Country='AU' then
    do;
      Bonus=300;
      Freq='Twice a Year';
    end;
run;
```

```
proc print data=work.bonus;
   var First_Name Last_Name Country Bonus Freq;
run;

data work.bonus;
   set orion.sales;
   length Freq $ 12;
   if Country='US' then
      do;
         Bonus=500;
         Freq='Once a Year';
      end;
   else do;
         Bonus=300;
         Freq='Twice a Year';
      end;
run;

proc print data=work.bonus;
   var First_Name Last_Name Country
       Bonus Freq;
run;
```

Summary of Lesson 10: Combining SAS Data Sets InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Concatenating Data Sets InstanceEndEditable

InstanceBeginEditable name="topicText"

You can concatenate two or more data sets by combining them vertically to create a new data set. It is important to know the structure and contents of the input data sets.

You use a DATA step to concatenate multiple data sets into a single, new data set. In the SET statement, you can specify any number of input data sets to concatenate. During compilation, SAS uses the descriptor portion of the first data set to create variables in the PDV, and then continues with each subsequent data set, creating additional variables in the PDV as needed. During execution, SAS processes the data sets in the order in which they are listed in the SET statement.

```
DATA SAS-data-set;
    SET SAS-data-set1 SAS-data-set2 ...;
RUN;
```

If the data sets have differently named variables, every variable is created in the new data set, and some observations have missing values for the differently named variables. You can use the RENAME= data set option to change variable names in one or more data sets. After they are renamed, they are treated as the same variable during compilation and execution, and in the new data set.

```
SAS-data-set (RENAME=(old-name-1=new-name-1;
                      old-name-2=new-name-2
                      ...
                      old-name-n=new-name-n))
```

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Merging SAS Data Sets One-to-One InstanceEndEditable

InstanceBeginEditable name="topicText"

Merging combines observations from two or more SAS data sets into a single observation in a new data set. A simple merge combines observations based on their positions in the original data sets. A match-merge combines them based on the values of one or more common variables. The result of a match-merge is dependant on the relationship between observations in the input data sets.

You use a DATA step with a MERGE statement to merge multiple data sets into a single data set. The BY statement indicates a match-merge and specifies the common variable or variables to match. The common variables are referred to as BY variables. The BY variables must exist in every data set, and each data set must be sorted by the value of the BY variables.

DATA *SAS-data-set*;
    MERGE *SAS-data-set1 SAS-data-set2 ...*;
    BY <DESCENDING> *BY-variable(s)*;
    <additional SAS statements>
RUN;

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Merging SAS Data Sets One-to-Many InstanceEndEditable

InstanceBeginEditable name="topicText" In a one-to-many merge, a single observation in one data set matches more than one observation in another data set. The DATA step is the same, regardless of the relationship between the data sets being merged. SAS processes each BY group before reinitializing the PDV. InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Merging SAS Data Sets That Have Non-Matches InstanceEndEditable

InstanceBeginEditable name="topicText"

When you merge data sets, observations in one data set might not have a matching observation in

another data set. These are called non-matches. By default, both matches and non-matches are included in a merged data set. The observations that are matches contain data from every input data set. The non-matching observations do not contain data from every input data set.
You can use the IN= data set option in a MERGE statement to create a temporary variable that indicates whether a data set contributed information to the observation in the PDV. The IN= variables have two possible values: *0* and *1*. You can test the value of this variable using subsetting IF statements to output only the matches or only the non-matches to the merged data set.

MERGE *SAS-data-set1* <**(IN=*variable)***>...

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Concatenating Data Sets with Different Variables

```
********** Create Data **********;
data empscn;
  input First $ Gender $ Country $;
  datalines;
Chang   M   China
Li      M   China
Ming    F   China
;
run;

data empsjp;
  input First $ Gender $ Region $;
  datalines;
Cho     F   Japan
Tomi    M   Japan
```

```
;
run;

********** Unlike-Structured Data Sets **********;
data empsall2;
  set empscn empsjp;
run;

proc print data=empsall2;
run;
```

Merging Data Sets One-to-One

```
********** Create Data **********;
data empsau;
  input First $ Gender $ EmpID;
  datalines;
Togar   M   121150
Kylie   F   121151
Birin   M   121152
;
run;

data phoneh;
  input EmpID Phone $15.;
  datalines;
121150 +61(2)5555-1793
121151 +61(2)5555-1849
121152 +61(2)5555-1665
;
run;

********** Match-Merge One-to-One**********;
```

```
data empsauh;
  merge empsau phoneh;
  by EmpID;
run;

proc print data=empsauh;
run;
```

Match-Merging Data Sets with Non-Matches

```
********** Create Data **********;
data empsau;
  input First $ Gender $ EmpID;
  datalines;
Togar   M   121150
Kylie   F   121151
Birin   M   121152
;
run;

data phonec;
  input EmpID Phone $15.;
  datalines;
121150 +61(2)5555-1795
121152 +61(2)5555-1667
121153 +61(2)5555-1348
;
run;

********** Match-Merge with Non-Matches**********;
data empsauc;
  merge empsau phonec;
  by EmpID;
```

```
run;

proc print data=empsauc;
run;
```

Selecting Non-Matches

```
********** Create Data **********;
data empsau;
  input First $ Gender $ EmpID;
  datalines;
Togar   M   121150
Kylie   F   121151
Birin   M   121152
;
run;

data phonec;
  input EmpID Phone $15.;
  datalines;
121150 +61(2)5555-1795
121152 +61(2)5555-1667
121153 +61(2)5555-1348
;
run;

********** Non-Matches from empsau Only **********;
data empsauc2;
  merge empsau(in=Emps)
        phonec(in=Cell);
  by EmpID;
  if Emps=1 and Cell=0;
run;
```

```
proc print data=empsauc2;
run;
```

Summary of Lesson 11: Creating Summary Reports InstanceEndEditable

InstanceBeginEditable name="linkHeader" This summary contains topic summaries, syntax, and sample programs. InstanceEndEditable

Topic Summaries

To go to the movie where you learned a task or concept, select a link.

InstanceBeginRepeat name="topicSummary" InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using PROC FREQ to Create Summary Reports InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use PROC FREQ to produce frequency tables that report the distribution of any or all variable values in a SAS data set. You use the TABLES statement to specify the frequency tables to produce. You can include a WHERE statement in the PROC FREQ step to subset the observations. SAS has TABLES statement options that you can use to suppress the default statistics.

```
PROC FREQ DATA=SAS-data-set <option(s)>;
    TABLES variable(s) </option(s)>;
    <additional SAS statements>
RUN;
```

Frequency distributions work best with variables whose values are categorical and best summarized by counts instead of averages. Variables that have continuous numeric values, such as dollar amounts and dates, or many discrete values, can result in a lengthy and meaningless frequency table. To create a useful frequency report for these variables, you can apply a SAS or user-defined format to group the values into categories.

You can list multiple variables in a TABLES statement, separated by spaces. This creates a one-way frequency table for each variable. You can request a separate analysis for each group by including a BY statement. You can request a two-way frequency table by separating the variables with an asterisk instead of a space. The resulting crosstabulation table displays statistics for each distinct combination of values of the selected variables. You can use TABLES statement options to suppress statistics, change the table format, and format the displayed values.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using PROC FREQ for Data Validation InstanceEndEditable

InstanceBeginEditable name="topicText"

The FREQ procedure can also be used to validate a data set. A one-way frequency table, which displays all discrete values for a variable and reports on missing values, easily identifies the existence of invalid or missing values. You can use the ORDER=FREQ and NLEVELS options to identify duplicate values. After you've identified invalid values, you can use PROC PRINT to display the corresponding observations.

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using the MEANS and UNIVARIATE Procedures InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use PROC MEANS to produce summary reports with descriptive statistics. By default, it reports the number of nonmissing values, the mean, the standard deviation, the minimum, and the maximum value of every numeric variable in a data set. You can use the VAR statement to specify the numeric variables to analyze, and add a CLASS statement to request statistics for groups of observations. The variables listed in the CLASS statement are called classification variables, or class variables, and each combination of class variable values is called a class level.

```
PROC MEANS DATA=SAS-data-set <statistic(s)>;
    VAR analysis-variable(s);
    CLASS classification-variable(s);
RUN;
```

When you use the CLASS statement, the output includes N Obs, which reports the number of observations with each unique combination of class variables. You can request specific statistics by listing them as options in the PROC MEANS statement. Other options are available to control the output.

You can also use PROC MEANS to validate a data set. The MIN, MAX, and NMISS statistics can be used to validate numeric data when you know the range of valid values. PROC

UNIVARIATE can be more useful because it displays the extreme observations, or outliers. By default, it displays the five highest and five lowest values of the analysis variable, and the number of the observation with each extreme value. You can use the NEXTROBS= option to display a different number of extreme observations.

```
PROC UNIVARIATE DATA=SAS-data-set;
    VAR variable(s);
RUN;
```

InstanceEndEditable

InstanceEndRepeatEntry InstanceBeginRepeatEntry

InstanceBeginEditable name="topicTitle" Using the Output Delivery System

InstanceEndEditable

InstanceBeginEditable name="topicText"

You can use the SAS Output Delivery System to create different output formats by directing output to various ODS destinations. For each type of formatted output that you want to create, you use an ODS statement to open that destination, submit one or more procedures that generate output, and then close the destination. A file is created for each open destination.

```
ODS destination FILE="filename" <options>;
    <SAS code to generate the report>
ODS destination CLOSE;
```

InstanceEndEditable

InstanceEndRepeatEntry InstanceEndRepeat

InstanceBeginEditable name="syntaxSummary" InstanceEndEditable

InstanceBeginEditable name="samplePrograms"

Sample Programs

Creating a One-Way Frequency Report

```
proc freq data=orion.sales;
  tables Gender;
  where Country='AU';
run;
```

Using Formats in PROC FREQ

```
proc format;
  value Tiers low-25000='Tier1'
              25000<-50000='Tier2'
              50000<-100000='Tier3'
              100000<-high='Tier4';
run;

proc freq data=orion.sales;
  tables Salary;
  format Salary Tiers.;
run;
```

Listing Multiple Variables on a TABLES Statement

```
proc freq data=orion.sales;
  tables Gender Country;
run;

proc sort data=orion.sales out=sorted;
  by Country;
run;

proc freq data=sorted;
  tables Gender;
  by Country;
run;
```

Creating a Crosstabulation Table

```
proc freq data=orion.sales;
  tables Gender*Country;
run;
```

Examining Your Data

```
proc print data=orion.nonsales2 (obs=20);
```

```
run;
```

Using PROC FREQ Options to Validate Your Data

```
proc freq data=orion.nonsales2 order=freq;
  tables Employee_ID/nocum nopercent;
run;
proc freq data=orion.nonsales2 nlevels;
  tables Gender Country Employee_ID/nocum nopercent;
run;
proc freq data=orion.nonsales2 nlevels;
  tables Gender Country Employee_ID/nocum nopercent noprint;
run;
```

Using PROC PRINT to Validate Your Data

```
proc print data=orion.nonsales2;
  where Gender not in ('F','M') or
        Country not in ('AU','US') or
        Job_Title is null or
        Salary not between 24000 and 500000 or
        Employee_ID is missing or
        Employee_ID=120108;
run;
```

Creating a Summary Report with PROC MEANS

```
proc means data=orion.sales;
  var Salary;
run;
```

Creating a PROC MEANS Report with Grouped Data

```
proc means data=orion.sales;
  var Salary;
  class Gender Country;
run;
```

Requesting Specific Statistics in PROC MEANS

```
proc means data=orion.sales n mean;
```

```
  var Salary;
run;

proc means data=orion.sales min max sum;
  var Salary;
  class Gender Country;
run;
```

Validating Data Using PROC MEANS

```
proc means data=orion.nonsales2 n nmiss min max;
  var Salary;
run;
```

Validating Data Using PROC UNIVARIATE

```
proc univariate data=orion.nonsales2;
  var Salary;
run;

proc univariate data=orion.nonsales2 nextrobs=3;
  var Salary;
run;

proc univariate data=orion.nonsales2 nextrobs=3;
  var Salary;
  id Employee_ID;
run;
```

Using the SAS Output Delivery System

```
/*Use a filepath to a location where you have Write access.*/
ods pdf file="c:/output/salaries.pdf";

proc means data=orion.sales min max sum;
  var Salary;
  class Gender Country;
```

```
run;

ods pdf close;

ods csv file="c:/output/salarysummary.csv";
proc means data=orion.sales min max sum;
  var Salary;
  class Gender Country;
run;
ods csv close;
InstanceEndEditable
```

PART I: SAS PROGRAMMING BOOK I & II SUMMAIRES

BOOK II

Summary

Main Points

Outputting Multiple Observations

```
OUTPUT <SAS-data-set(s)>;
```

- You can control when SAS writes an observation to a SAS data set by using an explicit OUTPUT statement in your code. After you use an explicit OUTPUT statement, there is no implicit OUTPUT statement at the bottom of the DATA step.

Writing to Multiple SAS Data Sets

```
DATA SAS-data-set-name SAS-data-set-name-n;
```

- To create more than one data set, you specify the names of the SAS data sets you want to create in the DATA statement. Separate SAS data set names with a space.
- You can use OUTPUT statements with conditional logic to create multiple data sets that contain observations based on the value of a variable in the input data set.

```
SELECT <(select-expression)>;
    WHEN-1 (when-expression -1 <…, when-expression-n>) statement;
    WHEN-n (when-expression -1 <…, when-expression-n>) statement;
    <OTHERWISE statement;>
END;
```

- You can use a SELECT group for conditional processing in a DATA step. The SELECT group begins with the keyword SELECT. The optional SELECT expression specifies any SAS expression that evaluates to a single value. Often a variable name is used as the SELECT expression.
- The WHEN statement begins with the keyword WHEN followed by at least one WHEN expression. The WHEN expression can be any SAS expression, including a compound expression. The WHEN expression is followed by a statement, which can be any executable SAS statement.
- The optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met.
- The keyword END signals the end of the select group.

- When a SELECT expression is specified, SAS evaluates the SELECT expression and compares that value to each WHEN expression and returns either a true or false. If no SELECT expression is specified, SAS evaluates each WHEN expression in order until it finds a true expression. If no WHEN expression is true, SAS executes the OTHERWISE statement if one is present.
- You can use functions in a SELECT expression.
- You can use DO-END groups in a SELECT group. You can execute multiple statements when a WHEN expression is true by using DO-END groups.

Controlling Variable Input and Output

SAS-data-set-name **(DROP=***variable(s)***)**
SAS-data-set-name **(KEEP=***variable(s)***)**

- The DROP= and KEEP= data set options can be used to specify variables to drop or keep in the output data.
- When the DROP= and KEEP= data set options are used in the SET statement, the variables are not processed and are not available in the program data vector.
- When the DROP= and KEEP= data set options are used in the DATA statement, they affect the variables in the output data set they are associated with.
- DROP and KEEP statements affect all output data sets listed in the DATA statement.
- You can use a combination of data set options and statements. If you use them together, statements are applied before data set options. If you attempt to drop and keep the same variable, you will get a warning.

Controlling Observation Input and Output

SAS-data-set-name **(OBS=***n***)**
SAS-data-set-name **(FIRSTOBS=***n***)**

- You can use the OBS= and FIRSTOBS= data set options to limit the number of observations that SAS processes.
- The OBS= data set option specifies the number of the last observation to process. It does **not** specify how many observations should be processed.
- The FIRSTOBS= data set option specifies a starting point for processing an input data set. By default, FIRSTOBS=1.
- You can use FIRSTOBS= and OBS= together to define a range of observations for SAS to process.

- FIRSTOBS= and OBS= can be used in a procedure step to limit the number of observations that are processed. If a WHERE statement is used to subset the observations, it is applied before the data set options.

Sample Code

Outputting Multiple Observations

```
data forecast;
   set orion.growth;
   Year=1;
   Total_Employees=Total_Employees*(1+Increase);
   output;
   Year=2;
   Total_Employees=Total_Employees*(1+Increase);
   output;
run;
```

Writing to Multiple SAS Data Sets (Using a SELECT Group)

```
data usa australia other;
   set orion.employee_addresses;
   select (Country);
   when ('US') output usa;
   when ('AU') output australia;
   otherwise output other;
   end;
run;
```

Writing to Multiple Data Sets (Using a SELECT Group with DO-END Group in the WHEN statement)

```
data usa australia other;
   set orion.employee_addresses;
   select (upcase(Country));
      when ('US') do;
         Benefits=1;
         output usa;
      end;
      when ('AU') do;
         Benefits=2;
         output australia;
      end;
      otherwise;
         Benefits=0;
         output other;
      end;
   end;
run;
```

Controlling Variable Input and Output

```
data usa australia(drop=State) other;
   drop Country;
   set orion.employee_addresses
      (drop=Employee_ID);
   if Country='US' then output usa;
   else if Country='AU' then output australia;
   else output other;
run;
```

Controlling Observation Input and Output

```
data australia;
   set orion.employee_addresses
      (firstobs=50 obs=100);
   if Country='AU' then output;
run;
```

```
proc print data=orion.employee_addresses
           (obs=10);
   where Country='AU';
   var Employee_Name City State Country;
run;
```

Summary

Main Points

Creating an Accumulating Variable Using the RETAIN Statement

```
RETAIN variable-name <initial-value> …;
```

- An accumulating variable accumulates the value of another variable and keeps its value from one observation to the next. You can use the RETAIN statement to create an accumulating variable.
- The RETAIN statement is a compile-time-only statement that prevents SAS from reinitializing the variable at the top of the DATA step. Because the variable is not reinitialized, it retains its value across multiple iterations of the DATA step.
- The RETAIN statement starts with the keyword RETAIN followed by the name of the variable whose values you want to retain. You can optionally specify an initial value for the variable. If you don't specify an initial value, the RETAIN statement initializes the variable to missing before the first execution of the DATA step.

Creating an Accumulating Variable Using the Sum Statement

```
variable+expression;
```

- As an alternative to using the RETAIN statement with an assignment statement, you can use the sum statement. By default, the sum statement initializes the variable to *0* and retains the variable. It also ignores missing input values from the expression.

Using BY-Group Processing: Summarizing Data by Groups

```
DATA output-SAS-data-set;
    SET input-SAS-data-set;
    BY BY-variable …;
      <additional SAS statements>
RUN;
```

- When you need to accumulate totals for a group of data, (for example, if you need to see total salaries allocated to special projects by department), the input data set needs to be sorted on the BY-variable.
- You can then use a BY statement in the DATA step to process the data in groups.

FIRST. *BY-variable*
LAST. *BY-variable*

- The BY statement creates two temporary variables (**FIRST.***variable* and **LAST.***variable*) for each BY variable listed. These variables identify the first and last observation in each BY group.
- You can use the **FIRST.** and **LAST.** variables in a DATA step to summarize the grouped data.
  - First, set the accumulating variable equal to *0* at the start of each BY group.
  - Second, increment the accumulating variable with a sum statement.
  - Third, output only the last observation of each BY group.

Using BY-Group Processing: Summarizing Data by Multiple Groups

- When you need to accumulate totals for multiple groups, you specify two or more BY variables. The first variable listed becomes the primary sort variable, and the second variable listed is the secondary sort variable.
- The BY statement creates two temporary variables for each BY variable listed.
- If the last observation for a value of the primary sort variable is encountered, it sets LAST. to 1 for all subsequent BY variables.

Sample Code

Creating an Accumulating Variable Using the RETAIN Statement

```
data mnthtot2;
   set orion.aprsales;
   retain Mth2Dte 0;
   Mth2Dte=Mth2Dte+SaleAmt;
run;
```

Creating an Accumulating Variable Using the Sum Statement

```
data mnthtot2;
   set orion.aprsales;
   Mth2Dte+SaleAmt;
run;
```

Using BY-Group Processing: Summarizing Data by Groups

```
data deptsals (keep=Dept DeptSal);
   set SalSort;
   by Dept;
   if First.Dept then DeptSal=0;
   DeptSal+Salary;
   if Last.Dept;
run;
```

Using BY-Group Processing: Summarizing Data by Multiple Groups

```
data deptsals (keep=Proj Dept DeptSal NumEmps);
   set projsort;
   by Proj Dept;
   if First.Dept then
      do;
         DeptSal=0;
         NumEmps=0;
      end;
   DeptSal+Salary;
   NumEmps+1;
   if Last.Dept;
run;
```

Summary

Main Points

Using Column Input

```
INPUT variable <$> startcol-endcol . . . ;
```

- You can use column input to read input data that is arranged in columns or fixed fields.
- To use column input your data must be standard data in fixed columns.
- Standard data is data that SAS can read without special instructions. Nonstandard data is data that SAS needs special instructions to read.

Using Formatted Input

```
INPUT column-pointer-control variable informat . . . ;
```

```
INPUT @n variable informat . . . ;
```

```
INPUT +n variable informat . . . ;
```

- You can use formatted input to read both standard and nonstandard data that is arranged in fixed fields.
- An informat is the special instruction that specifies how SAS reads raw data.
- The @n is an absolute pointer control that specifies the beginning location for a field. The +n is a relative pointer control that specifies how many columns to move the pointer before reading the next field.

Creating a Single Observation from Multiple Records

```
DATA SAS-data-set;
    INFILE 'raw-data-file-name';
    INPUT specifications;
    INPUT specifications;
   <additional SAS statements>
```

```
DATA SAS-data-set;
    INFILE 'raw-data-file-name';
    INPUT specifications /
          #n specifications;
    <additional SAS statements>
```

- You can use multiple INPUT statements to read a group of records in a raw data file as a single observation in a new data set.
- As an alternative to writing multiple INPUT statements, you can write one INPUT statement that contains line pointer controls to specify the record(s) from which values are to be read. There are two line pointer controls, the forward slash (a relative line pointer control) and the #*n* (an absolute line pointer control).

Controlling When a Record Loads

```
INPUT specifications . . . @ ;
```

- By default, each INPUT statement in a DATA step reads a new data record into the input buffer.
- You can use a line-hold specifier, the single trailing @, to prevent the second INPUT statement in a DATA step from moving to the second line in a raw data file. When you use the trailing @, the pointer position doesn't change and a new record isn't loaded in the input buffer when a subsequent input statement executes.
- The single trailing @ holds a raw data record in the input buffer until an INPUT statement without a trailing @ executes or the next iteration of the DATA step begins.

Reading Raw Data with Missing Values

```
INFILE 'raw-data-file' MISSOVER;
```

```
INFILE 'raw-data-file' DSD;
```

- By default, the DATA step looks in the next record if the end of the current record is encountered before all of the variables are assigned values. This default action is known as the FLOWOVER option.
- You can override the FLOWOVER option by using the MISSOVER option, which causes the DATA step to assign missing values to any variables that do not have values in the PDV when the end of a record is reached.

- The MISSOVER option works only for missing values that occur at the end of the record.
- You can use the DSD option in the INFILE statement to change how SAS treats delimiters when list input is used. The DSD option sets the default delimiter to a comma, treats consecutive delimiters as missing values, and enables SAS to read values with embedded delimiters if the value is surrounded by quotation marks.

Creating Multiple Observations from a Single Record

INPUT *specifications* @@**;**

INFILE '*raw-data-file*' **DSD;**

- You can use the the double trailing @@, which is a line pointer control, to hold a record across iterations of the DATA step. The double trailing @@ should only be used with list input and should not be used with the MISSOVER option.
- A record that is being held by the double trailing @@ is not released until the input pointer moves past the end of the record or an INPUT statement that has no line-hold specifier executes.

Sample Code

Windows: Replace *my-file-path* with the location where you stored the practice files.
UNIX and **z/OS**: Specify the fully qualified path in your operating environment.

Using Formatted Input

```
data work.discounts;
   infile 'my-file-path\offers.dat';
   input @1 Cust_type 4.
         @5 Offer_dt mmddyy8.
         @14 Item_gp $8.
         @22 Discount percent3.;
run;

proc print data=work.discounts;
run;
```

Creating a Single Observation from Multiple Records

```
data contacts;
   infile 'my-file-path\address.dat';
   input FullName $30.;
   input;
   input Address2 $25.;
   input Phone $8.;
run;

proc print data=work.contacts;
run;
```

Using a Single Trailing @

```
data salesQ1;
   infile 'my-file-path\sales.dat';
   input SaleID $4. @6 Location $3. @;
   if Location='USA' then
      input @10 SaleDate mmddyy10.
            @20 Amount 7.;
   else if Location='EUR' then
      input @10 SaleDate date9.
            @20 Amount commax7.;
run;

proc print data=salesQ1;
run;
```

Using the MISSOVER Option

```
data work.contacts;
   length Name $ 20. Phone Mobile $ 14.;
   infile 'my-file-path\phone.csv' dlm=',' missover;
   input Name $ Phone $ Mobile $;
run;

proc print data=work.contacts;
run;
```

Using the DSD Option

```
data contacts2;
   length Name $ 20. Phone Mobile $ 14.;
   infile 'my-file-path\phone2.csv' dsd;
   input Name $ Phone $ Mobile $;
run;

proc print data=contacts2;
run;
```

Using the Double Trailing @@

```
data donate07;
   length ID $ 4;
   infile 'my-file-path\charity.dat';
   input ID $ Amount @@;
run;

proc print data=work.donate07;
run;
```

Summary

Main Points

Using SAS Functions

function-name*(argument-1*<,*argument-n*>)**

- A SAS function is a routine that performs a calculation on, or a transformation of, the arguments listed in parentheses and returns a value.
- A target variable is a variable to which the result of the function is assigned. If the target variable is a new variable, the type and length are determined by the expression on the right side of the equals sign. If the expression uses a function whose result is numeric, then the target variable is numeric with a length of 8 bytes. If the expression uses a function whose result is character, then the target variable is character and the length is determined by the function.

Extracting and Transforming Character Values

Function	Purpose
Var=**SUBSTR(***string*,*start*<,*length*>**);**	On the right side of an assignment statement, the SUBSTR function extracts a substring of characters from a character string, starting at a specified position in the string.
LENGTH(*argument***)**	The LENGTH function returns the length of a character string, excluding trailing blanks.
RIGHT(*argument***)**	The RIGHT function right-aligns a value. If there are trailing blanks, they are moved to the beginning of the value.
LEFT(*argument***)**	The LEFT function left-aligns a character value. If there are leading blanks, they are moved to the end of the value.
CHAR(*string*,*position***)**	The CHAR function returns a single character from a specified position in a character string.
PROPCASE(*argument*<,*delimiter(s)*>**)**	The PROPCASE function converts all letters in a value to proper case.

Function	Purpose
UPCASE(*argument***)**	The UPCASE function converts all letters in a value to uppercase.
LOWCASE(*argument***)**	The LOWCASE function converts all letters in a value to lowercase.

Separating and Concatenating Character Values

Function	Purpose
SCAN (*string*,*n*<,'*delimiter(s)*'>**)**	The SCAN function enables you to separate a character value into words and to return the *n*th word.
CATX (*separator*,*string1*,…,*string-n***)**	The CATX function removes leading and trailing blanks, inserts separators, and returns a concatenated character string.
NewVar=*string1* !! *string2*;	The concatenation operator joins character strings.
TRIM(*argument***)**	The TRIM function removes trailing blanks from a character string.
STRIP(*argument***)**	The STRIP function removes leading and trailing blanks from a character string.
CAT(*string1*,…,*string-n***)** **CATT(***string1*,…,*string-n***)** **CATS(***string1*,…,*string-n***)**	These functions return concatenated character strings. The CAT function does not remove any leading or trailing blanks. The CATT function trims trailing blanks. The CATS function strips leading and trailing blanks.

- You use the SCAN function when you know the relative order of words but their starting positions vary. You use the SUBSTR function when you know the exact position of the string that you want to extract from a character value.

Finding and Modifying Character Values

Function	Purpose
FIND (*string*,*substring*<,*modifiers*,*start*>)	The FIND function searches for a specific substring of characters within a character string that you specify. The function searches for the first occurrence of the substring and returns the starting position of that substring. If the substring is not found in the string, FIND returns a value of 0.
SUBSTR(*string*,*start*<,*length*>)=*value*;	On the left side of the assignment statement, the SUBSTR function replaces characters at a specified position within the value.
TRANWRD (*source*,*target*,*replacement*)	The TRANWRD function replaces or removes all occurrences of a given word (or a pattern of characters) within a character string.
COMPRESS (*source*<,*chars*>)	The COMPRESS function removes the characters listed in the *chars* argument from the source. If no characters are specified, the COMPRESS function removes all blanks from the source.

Sample Code

Extracting and Transforming Character Values

```
data charities(drop=Len);
   length ID $ 5;
   set orion.biz_list;
   Len=length(Acct_Code);
   if substr(Acct_Code,Len,1)='2';
   ID=substr(Acct_Code,1,Len-1);
run;

data charities(drop=Code_Rt);
   length ID $ 5;
   set orion.biz_list;
   Code_Rt=right(Acct_Code);
   if char(Code_Rt,6)='2';
   ID=left(substr(Code_Rt,1,5));
run;
```

Separating and Concatenating Character Values

```
data labels;
   set orion.contacts;
   length FMName LName $ 15;
   FMName = scan(Name,2,',');
   LName = scan(Name,1,',');
   FullName=catx(' ',title,fmname,lname);
run;
```

Finding and Modifying Character Values

```
data correct;
   set orion.clean_up;
   if find(Product,'Mittens','I')>0 then do;
      substr(Product_ID,9,1) = '5';
      Product=tranwrd(Product,'Luci ','Lucky ');
   end;
   Product=propcase(Product);
   Product_ID=compress(Product_ID);
run;
```

Summary

Main Points

Using Descriptive Statistics Functions

function-name(*argument-1, argument-2,…,argument-n*)

Function	Returns
SUM	the sum of the nonmissing arguments
MEAN	the arithmetic mean (average) of the arguments
MIN	the smallest value from the arguments
MAX	the largest value from the arguments
N	the number of nonmissing arguments
NMISS	the number of missing numeric arguments
CMISS	the number of missing numeric or character arguments

- A SAS function is a routine that performs a calculation on, or a transformation of, the arguments listed in parentheses and returns a value.
- You can list all the variables in the function, or you can use a variable list by preceding the first variable name in the list with the keyword OF. There are several types of variable lists including numbered ranges, name ranges, name prefixes, and special SAS names.

Variable List	Description	Example
Numbered Range	all variables x1 to x*n*, inclusive	`Total = sum(of Qtr1-Qtr4);`

Variable List	Description	Example
Name Range	all variables ordered as they are in the program data vector, from **x** to **a** inclusive	`Total = sum(of Qtr1--Fourth);`
Name Prefix	all variables that begin with the same string	`Total = sum(of Tot:);`

Special SAS Name List:	all of the variables, all of the character variables, or all of the numeric variables that are already defined in the current DATA step	`Total = sum(of _All_);`
		`Total = sum(of _Character_);`
		`Total = sum(of _Numeric_);`

Truncating Numeric Values

ROUND(*argument<,round-off-unit>***)**

CEIL(*argument***)**

FLOOR(*argument***)**

INT(*argument***)**

- There are four truncation functions that you can use to truncate numeric values. They are the ROUND, CEIL, FLOOR, and INT functions.
- The ROUND function returns a value rounded to the nearest multiple of the round-off unit. If you don't specify a round-off unit, the argument is rounded to the nearest integer.
- The CEIL function returns the smallest integer greater than or equal to the argument.
- The FLOOR function returns the greatest integer less than or equal to the argument.

- The INT function returns the integer portion of the argument.

Converting Values Between Data Types

INPUT(*source, informat***)**

PUT(*source, format***)**

- You can allow SAS to automatically convert data to a different data type for you, but it can be more efficient to use SAS functions to explicitly convert data to a different data type.
- By default, if you reference a character variable in a numeric context, SAS tries to convert the variable values to numeric. Automatic conversion uses the *w*. informat, and it produces a numeric missing value from any character value that does not conform to standard numeric notation.
- You can use the INPUT function to explicitly convert character values to numeric values. The INPUT function returns the value that is produced when the source is read with a specified informat.
- Numeric data values are automatically converted to character values whenever they are used in a character context. For example, SAS automatically converts a numeric value to a character value when you use the concatenation operator.
- When SAS automatically converts a numeric value to a character value, SAS writes the numeric value with the BEST12. format and right aligns the value. The resulting value might contain leading blanks.
- You can use the PUT function to explicitly control the numeric-to-character conversion using a format.

Sample Code

Using Descriptive Statistics Functions and Truncating Numeric Values

```
data donation_stats;
   set orion.employee_donations;
   keep Employee_ID Total AvgQT NumQT;
   Total = sum(of Qtr1-Qtr4);
   AvgQT = round(Mean(of Qtr1-Qtr4),1);
   NumQt = n(of Qtr1-Qtr4);
run;

proc print data=donation_stats;
run;
```

Converting Values Between Data Types

```
data hrdata;
   keep EmpID GrossPay Bonus Phone HireDate;
   set orion.convert;
   EmpID = ID+11000;
   Bonus = input(GrossPay,comma6.)*.10;
   Phone = '(' !! put(Code,3.) !! ') ' !! Mobile;
   HireDate = input(Hired,mmddyy10.);
run;

proc print data=hrdata;
   format HireDate mmddyy10.;
run;
```

Summary

Main Points

Identifying Logic Errors

- Syntax errors occur when programming statements don't conform to the rules of the SAS language. When a syntax error occurs, SAS writes an error message to the log.
- Logic errors occur when the programming statements follow the rules but the results aren't correct. Since the statements conform to the rules, SAS doesn't write an error message to the log.
- The lack of messages can make logic errors more difficult to detect and correct than syntax errors.

Using PUTLOG Statements

```
PUTLOG <specifications>;
PUTLOG 'text';
PUTLOG variable-name=;
PUTLOG variable-name=format-namew.;
PUTLOG _ALL_;
```

- You can use PUTLOG statements to display messages, variable names, and variable values in the log. This technique is helpful when you suspect that the value of a variable might be causing a logic error.
- By default, the PUTLOG statement writes character values with the standard character format $*w*. To use a different format, specify the name of the variable followed by the format name and width.
- When you're debugging a program, it's often helpful to see what SAS has stored in the program data vector. To write the current contents of the PDV to the log, use the _ALL_ option in the PUTLOG statement. When you use the _ALL_ option, the values of the automatic variables _ERROR_ and _N_ are included in the log.

SET *SAS-data-set* **END=** *variable* <*options*>**;**

INFILE *'raw-data-file'* **END=** *variable* <*options*>**;**

- You can use the END= option in the SET statement to create and name a temporary variable that acts as an end-of-file indicator. You can also use the END= option in an INFILE statement to indicate the end of a raw data file.
- The value of the END= variable is initialized to *0* and is set to *1* when the SET statement reads the last observation from an input data set or when the INPUT statement reads the last observation from a raw data file. You can check the value of the END= variable in an IF statement to conditionally execute PUTLOG statements.

Using the DATA Step Debugger

DATA *data-set-name***/DEBUG;**

- You can also debug logic errors by using the DATA step debugger. This tool consists of windows and a group of commands. By issuing commands, you can execute DATA step statements one by one and pause to display the resulting variable values in a window. By observing the results that are displayed, you can determine where the logic error occurs.
- The DATA Step Debugger can only be used in an interactive session. To invoke the debugger, you use the DEBUG option in the DATA statement. Because the debugger is interactive, you can repeat the process of issuing commands and observing results as many times as you need during a single debugging session.
- Once the DATA step comes to an end, it can't be restarted. You can examine the final values of the variables. But, to restart the debugging process, you have to quit and then restart the DATA Step Debugger.

Sample Code

Using PUTLOG Statements

```
data us_mailing;
   set orion.mailing_list (obs=10);
   drop Address3;
   length City $ 25 State $ 2 Zip $ 5;
   putlog _n_=;
   putlog "Looking for country";
   if find(Address3,'US') ;
   putlog "Found US";
   Name=catx(' ',scan(Name,2,','),scan(Name,1,','));
   City=scan(Address3,1,',');
   State=scan(address3,2,',');
   Zip=scan(Address3,3,',');
   putlog State=$quote4. Zip=$quote7.;
run;
```

Using the DATA Step Debugger

```
data us_mailing /debug;
   set orion.mailing_list;
   drop Address3;
   length City $ 25 State $ 2 Zip $ 5;
   if find(Address3,'US');
   Name=catx(' ',scan(Name,2,','),scan(Name,1,','));
   City=scan(Address3,1,',');
   State=scan(Address3,2,',');
   Zip=scan(Address3,3,',');
run;
```

Summary

Main Points

Constructing a Simple DO Loop

```
DO index-variable=start TO stop <BY increment>;
    iterated SAS statements…
END;
```

```
DO index-variable=item-1 <,… item-n>;
    iterated SAS statements…
END;
```

- An iterative DO loop executes the statements between the DO statement and the END statement repetitively.
- If you do not specify an increment for a DO loop, the increment defaults to 1.
- If your start value is greater than your stop value, you must specify an increment that is negative.
- You can use an item list rather than a start value and stop value to control your DO loop. The items must either be all numeric or all character constants, or they can be variables.
- You can use the OUTPUT statement within a DO loop to explicitly write the values out to the data set on each iteration of the DO loop.

Conditionally Executing DO Loops

```
DO UNTIL (expression);
    iterated SAS statements…
END;
```

```
DO WHILE (expression);
    iterated SAS statements…
END;
```

```
DO index-variable=start TO stop <BY increment>
    UNTIL | WHILE (expression);
    iterated SAS statements…
END;
```

- You can use a DO UNTIL statement instead of a simple DO statement. In a DO UNTIL statement, you specify a condition and SAS executes the loop until that condition is true.
- In a DO UNTIL loop, SAS evaluates the expression at the bottom of the loop after each iteration.
- You can use a DO WHILE statement instead of a simple DO statement. In a DO WHILE statement, you specify a condition and SAS executes the loop while the condition is true.
- In a DO WHILE loop, SAS evaluates the condition at the top of the loop and executes the statements within the loop if the condition is true.
- It is possible to create a DO WHILE loop that never executes; it is also possible to create a DO WHILE or a DO UNTIL loop that executes infinitely. You should write your conditions and iterated statements carefully.
- You can combine DO UNTIL and DO WHILE statements with the iterative DO statement, which is one way to avoid creating an infinite loop. In this case, the loop executes either until the value of the index variable exceeds the specified range or until the expression is true for an UNTIL clause or false for a WHILE clause.

Nesting DO Loops

```
DO index-variable=start TO stop <BY increment>;
    iterated SAS statements…
    DO index-variable=start TO stop <BY increment>;
        iterated SAS statements…
    END;
    iterated SAS statements…
END;
```

- You can nest DO loops in a DATA step. You must use different index variables for each loop, and you must be certain that each DO statement has a corresponding END statement.

Sample Code

Using an Iterative DO Loop

```
data compound;
   Amount=50000;
   Rate=.045;
   do i=1 to 20;
      Yearly+(Yearly+Amount)*Rate;
   end;
   do i=1 to 80;
      Quarterly+((Quarterly+Amount)*Rate/4);
   end;
run;

proc print data=work.compound;
run;
```

Using a DO Loop to Reduce Redundant Code

```
data forecast;
   set orion.growth;
   do Year=1 to 6;
      Total_Employees=
         Total_Employees*(1+Increase);
      output;
   end;
run;

proc print data=forecast noobs;
run;
```

Conditionally Executing DO Loops

```
data invest;
   do until (Capital>1000000);
      Year+1;
      Capital +5000;
      Capital+(Capital*.045);
   end;
run;

proc print data=invest noobs;
run;
```

Using an Iterative DO Loop with a Conditional Clause

```
data invest;
   do year=1 to 30 until (Capital>250000);
      Capital +5000;
      Capital+(Capital*.045);
   end;
run;

data invest2;
   do year=1 to 30 while (Capital<=250000);
      Capital +5000;
      Capital+(Capital*.045);
   end;
run;

proc print data=invest;
   format Capital dollar14.2;
run;

proc print data=invest2;
   format Capital dollar14.2;
run;
```

Nesting DO Loops

```
data invest (drop=Quarter);
   do Year=1 to 5;
      Capital+5000;
      do Quarter=1 to 4;
         Capital+(Capital*(.045/4));
      end;
      output;
   end;
run;

proc print data=invest;
   format Capital dollar14.2;
run;
```

Summary

Main Points

Understanding SAS Arrays

- A SAS array is a temporary grouping of elements that exists only for the duration of the DATA step. Unlike arrays in other programming languages, SAS arrays are not data structures.
- An array is identified by a single unique name. The array name must be different from any variable name in the data set you are referencing.
- All variables that are grouped together in an array must be the same type: either all character or all numeric.
- Arrays can be one-dimensional or multi-dimensional.

Creating SAS Arrays

ARRAY *array-name* **{***dimension***}** *<array-elements>***;**

- *array-name* specifies the name of the array.
- *dimension* describes the number and arrangement of elements in the array. The default dimension is one. The array dimension must be enclosed in braces, parentheses, or brackets. It is best to use braces or brackets so that there is no confusion with functions.
- You can use an asterisk to indicate the dimension of a one-dimensional array. When you use an asterisk, SAS determines the dimension of the array by counting the variables in the list of array elements. You can list each variable name separated by a space, or you can use a numbered range list or name range list specification.
- To create an array of character variables, type a dollar sign ($) after the dimension in the ARRAY statement. By default, all character variables that are created in an ARRAY statement are assigned a length of 8. You can assign a different length by specifying the length after the dollar sign.
- *array-elements* lists the variables to include in the array. Array elements can be listed in any order, but they must be either all numeric or all character. You can list each variable name separated by a space, or you can use a variable list. You can also specify the keywords _NUMERIC_ or _CHARACTER_ .

Processing SAS Arrays

*array-name***{***subscript***}**

- The syntax for an array reference is the name of the array, followed by a subscript enclosed in braces, brackets, or parentheses. The subscript can be an integer, a variable, or a SAS expression.
- Typically, arrays are used with DO loops to process multiple variables and to perform repetitive calculations. When you use a DO loop, the index variable is used as the array subscript and the DO loop references each element of the array.

DIM(*array-name***)**

- Another way to specify the stop value of a DO loop is to use the DIM function. The DIM function returns the number of elements in the array.

Using SAS Arrays to Create Variables and Perform Calculations

ARRAY *array-name* **{***dimension***}** <**$**> <*length*> <*array-elements*>**;**

- When you do not reference existing variables in the ARRAY statement, SAS automatically creates the variables and assigns default names to them. The default variable names are created by concatenating the array name and the numbers 1, 2, 3, and so on, up to the array dimension. When you create variables in an ARRAY statement, the default variable names will match the case that you used for the array name. Alternatively, you can specify the new variable names by listing them in the ARRAY statement.
- Variables that you create in an ARRAY statement all have the same variable type. If you want to create an array of character variables, you must add a dollar sign after the array dimension. By default, all character variables that are created in an ARRAY statement are assigned a length of 8. You can assign a different length for all variables by specifying the length for each variable in a LENGTH statement prior to the ARRAY statement.
- You can pass an array to a function using the keyword OF. This is similar to passing a variable list to a function.

Assigning Initial Values to an Array

ARRAY *array-name* **{***dimension***}** <_**TEMPORARY**_>
<*array-elements*> <(*initial-value-list*)>**;**

- To assign initial values in an ARRAY statement, you place the values in an initial value list. In this list, you specify one initial value for each corresponding array element. Elements and values are matched by position, so the values must be listed in the order of the array elements. You separate each value with a comma or blank, and you enclose the values in parentheses. If you are assigning character values, each value must be enclosed in quotation marks.
- When you specify an initial value list, all elements behave as if they were named in a RETAIN statement. This creates a lookup table, that is, a list of values to refer to during DATA step processing.
- You can use the keyword _TEMPORARY_ in an ARRAY statement to indicate that the elements are not needed in the output data set. Temporary arrays are useful when you only need the array to perform a calculation. You can improve performance time by using temporary array elements.

Sample Code

Processing SAS Arrays

```
data charity;
   set orion.employee_donations;
   keep employee_id qtr1-qtr4;
   array contrib{*} qtr1-qtr4;
   do i=1 to dim(contrib);
      contrib{i}=contrib{i}*1.25;
   end;
run;
```

Using SAS Arrays to Create Variables and Perform Calculations

```
data percent(drop=i);
   set orion.employee_donations;
   array contrib{4} qtr1-qtr4;
   array Pct{4};
   Total=sum(of contrib{*});
   do i=1 to 4;
      pct{i}=contrib{i}/Total;
   end;
run;
```

Assigning Initial Values to an Array

```
data compare(drop=i);
   set orion.employee_donations;
   array Contrib{4} Qtr1-Qtr4;
   array Diff{4};
   array Goal{4} _temporary_ (10,20,20,15);
   do i=1 to 4;
      Diff{i}=sum(Contrib{i},-Goal{i});
   end;
run;
```

Summary

Main Points

An Overview of Data Set Structure

- Some data sets store all the information about one entity in a single observation. This type of data set is referred to as a **wide data set**.
- Other data sets have multiple observations per entity. Each observation typically contains a small amount of data and missing values might or might not be stored. This type of data set is known as a **narrow data set**.
- When you write a program, you need to consider the data available, the output you want, and the processing required. To create your output, you might need to restructure the data. Depending on the type of analysis or report you want to run, you might need to convert a narrow data set to a wide data set, or a wide data set to a narrow data set.

Rotating with the DATA Step

- Restructuring a data set is sometimes referred to as **rotating a data set**.
- You can use a DATA step to restructure a data set. Using a DATA step enables you to create multiple data sets, direct output to new data sets based on which data set contributed to the observation, use **FIRST.** and **LAST.** processing, and use complex data manipulation techniques.

Transposing a Data Set

```
PROC TRANSPOSE DATA=input-data-set
      <OUT=output-data-set>
      <NAME=variable-name>
      <PREFIX=variable-name>;
   BY variable(s) <NOTSORTED>;
   VAR variable(s);
   ID variable;
RUN;
```

- As an alternative to writing a lengthy DATA step, you can often use the TRANSPOSE procedure to restructure the values in a SAS data set. The TRANSPOSE procedure transposes selected variables into observations.
- You can use a number of options and statements with PROC TRANSPOSE. The NAME= option names the column in the output data set containing the rotated variable names. The ID statement names the variable whose values become the names of the new variables. You can use the OUT= option to specify a new name for the output data set.

- PROC TRANSPOSE transposes numeric variables by default. Character variables are transposed only if they're explicitly listed in a VAR statement.
- PROC TRANSPOSE doesn't print the output data set. So, you need to use PROC PRINT, PROC REPORT, or some other SAS reporting tool if you want to print the output data.
- You can use a BY statement with PROC TRANSPOSE. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable is not transposed.
- When you use a BY statement with PROC TRANPOSE, unless you specify the NOTSORTED option, the original data set must be sorted or indexed by all BY variables. This option specifies that the observations are not necessarily sorted in alphabetic or numeric order. For example, the data might be grouped in some other way, such as chronological order.
- In the output data set, **_NAME_** is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. The remaining transposed variables are named **COL1** through **COL*n***.
- The default label for the **_NAME_** variable is NAME OF FORMER VARIABLE. You can use the LABEL option to see this label in your PROC PRINT output. You can use the NAME= option to give the **_NAME_** variable a more descriptive name.

Modifying and Enhancing a Transformation

- You can use an ID statement to specify the variable whose values will become the names of the new columns.
- When you use a numeric variable as an ID variable, PROC TRANSPOSE changes the formatted ID value into a valid SAS name.
- You can use the PREFIX= option to specify a prefix for each new variable name.

Sample Code

Rotating with the DATA Step

```
data rotate (keep=Employee_ID Period Amount);
   set orion.employee_donations
            (drop=Recipients Paid_By);
   length Period $4;
   array contrib{4} Qtr1-Qtr4;
   do i=1 to 4;
       if contrib{i} ne . then do;
        Period=cats("Qtr",i);
        Amount=contrib{i};
        output;
     end;
   end;
run;
```

Using the TRANSPOSE Procedure

```
proc transpose
     data=targets
     out=sales_targets
     name=Month;
   by year;
run;
proc print
   data=sales_targets noobs
   label;
run;
```

Summary

Main Points

Match-Merging SAS Data Sets

```
DATA SAS-data-set;
      MERGE SAS-data-set1 SAS-data-set2 . . . ;
      BY <DESCENDING> BY-variable(s);
      <additional SAS statements>
RUN;
```

```
SAS-data-set (IN=variable)
```

- You can use the MERGE statement in a DATA step to combine SAS data sets with related data into a single observation in a new data set based on the values of one or more common variables. This process is called match-merging.
- Before the merge, each input data set must first be sorted in order of the values of the BY variable(s).
- A match-merge produces matches (observations containing data from both input data sets) and non-matches (observations containing data from only one input data set) by default.
- To identify the data sets that contribute observations, you can use the IN= data set option. You specify an IN= data set option after an input data set in the MERGE statement. SAS creates a temporary variable that indicates whether the data set contributed data to each output observation.
- You can use the subsetting IF statement to output only those observations that contain data from all the input data sets or from just one data set.

Using Data Manipulation Techniques with a Match-Merge

```
OUTPUT <SAS-data-set(s)>;
```

- You can use an OUTPUT statement with the MERGE statement to better control your output. For example, you can direct the matches (observations containing data from both input data sets) to one data set, and the non-matches (observations containing data from only one input data set) to another data set.
- To control which variables appear in your output, you can use the KEEP= data set option, the DROP= data set option, the KEEP statement, or the DROP statement.

- You can use **FIRST.** and **LAST.** processing along with a sum statement to summarize merged data.

Match-Merging Data Sets That Lack a Common Variable

- If data sets don't share a common variable, you can merge them using a series of merges in separate DATA steps. The data sets must be sorted by the appropriate BY variable.

Match-Merging a SAS Data Set and an Excel Workbook

LIBNAME *libref* '*physical-file-name*'**;**

- You can merge a SAS data set with an Excel workbook. You can use the SAS/ACCESS LIBNAME statement to assign a libref to an Excel workbook. This way, SAS treats each worksheet within the workbook as though it is a SAS data set.
- When you assign a libref to an Excel workbook, SAS appends a dollar sign to the end of each Excel worksheet name. But a valid SAS data set name can't contain a dollar sign. You use a SAS name literal to refer to an Excel worksheet in SAS code. You enclose the name of the worksheet, including the dollar sign, in quotation marks followed by the letter **n**.
- Remember to clear the libref to unlock the Excel workbook file.

Match-Merging Data Sets with Same-Named Variables

SAS-data-set **(RENAME = (***old-name-1 = new-name-1*
*<…old-name-n = new-name-n>***))**

- When you match-merge data sets that contain same-named variables (other than the BY-variables), the DATA step overwrites values of the same-named variable in the first data set in which it appears with values of the same-named variable in subsequent data sets.
- You can use the RENAME= data set option in the MERGE statement to rename variables in a data set. The RENAME= option doesn't rename the variable in the input data set. Instead, it tells SAS which slot in the PDV to use when building the observation.

Sample Code

Windows: Replace *my-file-path* with the location where you stored the practice files.
UNIX and **z/OS**: Specify the fully qualified path in your operating environment.

Match-Merging SAS Data Sets

```
data CustOrd;
   merge orion.customer(in=cust)
         work.order_fact(in=order);
   by Customer_ID;
   if cust and order;
run;
```

Using Data Manipulation Techniques with a Match-Merge

```
data orders(keep=Customer_Name Quantity
       Total_Retail_Price)
   summary(keep=Customer_Name NumberOrders)
   noorders(keep=Customer_Name Birth_Date);
   merge orion.customer
         work.order_fact(in=order);
   by Customer_ID;
   if order=1 then do;
      output orders;
      if first.Customer_ID then NumberOrders=0;
      NumberOrders+1;
      if last.Customer_ID then output summary;
   end;
   else output noorders;
run;
```

Match-Merging a SAS Data Set and an Excel Workbook

```
libname survey 'my-file-path\survey.xls';

data CustOrdProdSurv;
   merge CustOrdProd(in=c)
         survey.'Supplier$'n(in=s);
   by Supplier;
   if c=1 and s=1;
run;

libname survey clear;
```

Match-Merging SAS Data Sets with Same-Named Variables

```
data CustOrdProdSurv;
   merge CustOrdProd
         (in=c rename=(Quantity=Qty))
         survey.'Supplier$'n(in=s);
   by Supplier;
   if c=1 and s=1;
   if Qty>=Quantity and
      Total_Retail_Price>=Price;
run;
```

Summary

Main Points

Understanding PROC SQL

- SQL (Structured Query Language) is a standardized language that many software products use to retrieve, join, and update data in tables. Using PROC SQL, you can write ANSI standard SQL queries that retrieve and manipulate data.
- PROC SQL can query data that is stored in one or more SAS data sets or any other types of data files that you can access by using SAS/ACCESS engines.
- In SQL terminology, a SAS data set is a table, a variable is a column, and an observation is a row.
- You can perform many of the same tasks by using PROC SQL and the DATA step. Each technique has advantages.

```
PROC SQL;
      SELECT column-1<, column-2>…
            FROM table-1…
            <WHERE expression>
            <additional clauses>;
QUIT;
```

- In a PROC SQL step, the PROC SQL statement starts the SQL procedure. The SQL procedure stops running when SAS encounters one of the following step boundaries: the QUIT statement, or the beginning of another PROC step or a DATA step.
- A PROC SQL step can contain one or more statements. The SELECT statement (a query) retrieves data from one or more tables and creates a report by default. A PROC SQL step can contain one or more SELECT statements. The CREATE TABLE statement is one of many other statements that can appear in a PROC SQL step.

Querying a Table

```
SELECT column-1<, column-2>…
      FROM table-1…
      <WHERE expression>
      <additional clauses>;
```

- The SELECT statement is composed of clauses and ends with a semicolon. The SELECT clause and the FROM clause are the only required clauses. Optional clauses include the

WHERE clause, the GROUP BY clause, the HAVING clause, and the ORDER BY clause. In the SELECT statement, the clauses must appear in the order in which they are listed here.

SELECT *column-1<, column-2>…*

SELECT *

- The SELECT clause specifies the columns to include in the query result. After the keyword, you can specify the names of one or more columns in the table(s) you're querying, separated by commas. To select all columns in the input table(s), you specify an asterisk instead of column names.

FROM *table-1…*

- The FROM clause specifies the table or tables that contain the columns. After the keyword, you specify the name(s) of the table(s).

<**WHERE** *expression*>

- The WHERE clause subsets rows by identifying a condition that must be satisfied for each row to be included in the output. You can use any valid SAS expression to specify the condition.

CREATE TABLE *table-name* **AS**
SELECT *column-1<, column-2>…*
FROM *table-1…*
<**WHERE** *expression*>
<*additional clauses*>**;**

- To create an output table (a SAS data set) from the results of a query instead of a report, you can use the CREATE TABLE statement. After the keywords CREATE TABLE, you specify the name of the output table, followed by the keyword AS. Then, you specify the clauses that are used in a query: SELECT, FROM, and any optional clauses. As in the SELECT statement, the query clauses must appear in the order shown here.

Joining Tables

- Joining tables enables you to select data from multiple tables as if the data were contained in one table. Joins do not alter the original tables. To join tables, you can use a SELECT statement.

```
SELECT column-1<, column-2>…
    FROM table-1, table-2…
    <WHERE join-condition(s)>
    <additional clauses>;
```

- The SELECT clause specifies the columns that appear in the report.
- The FROM clause specifies the tables to be joined. You use commas to separate table names.
- The WHERE clause specifies one or more *join-conditions* that PROC SQL uses to combine and select rows for the result set. The *join-conditions* are expressed as an *sql-expression*, which can be any valid SAS expression. In addition to *join-conditions*, the WHERE clause can also specify an *expression* that subsets the rows.
- Additional clauses might also appear in a SELECT statement that joins tables.
- In a basic PROC SQL join, the SELECT statement does not have a WHERE clause. PROC SQL combines each row from the first table with every row from the second table to create a Cartesian product. A basic join is resource intensive and is rarely used.
- To join tables by matching rows based on the values of a common column, you can include a WHERE clause in the SELECT statement. An inner join is a specific type of join that returns only a subset of the rows from the first table that matches the rows from the second table.
- If the SELECT clause, the FROM clause, or the WHERE clause references a column that has the same name in multiple tables, you must specify the table name and a period before the column name. Prefixing the table name is called qualifying a column.

```
FROM table-1 <AS> alias-1,
     table-2 <AS> alias-2 …
```

- A qualified table name can specify an alias instead of a full table name. You can specify aliases for tables in the FROM clause. After the table name, you can optionally specify the keyword AS. Then, you specify an alias, which can be any valid SAS name.

Sample Code

Querying a Table to Create a Report

```
proc sql;
   select Employee_ID, Job_Title, Salary
      from orion.sales_mgmt
      where Gender='M';
quit;
```

Querying a Table to Create an Output Data Set

```
proc sql;
   create table direct_reports as
      select Employee_ID, Job_Title, Salary
         from orion.sales_mgmt;
quit;
```

Joining Tables by Using Full Table Names to Qualify Columns

```
proc sql;
   select sales_mgmt.Employee_ID, Employee_Name,
          Job_Title, Salary
      from orion.sales_mgmt,
           orion.employee_addresses
      where sales_mgmt.Employee_ID =
            employee_addresses.Employee_ID;
quit;
```

Joining Tables by Using Table Aliases to Qualify Columns

```
proc sql;
   select s.Employee_ID, Employee_Name, Job_Title, Salary
      from orion.sales_mgmt as s,
           orion.employee_addresses as a
      where s.Employee_ID = a.Employee_ID;
quit;
```

Summary

Main Points

What Is the Macro Facility?

- The SAS macro facility is a tool for extending and customizing SAS and for reducing the amount of text that you must enter to complete tasks. The macro facility consists of the macro processor and the SAS macro language.
- You can package small amounts of text into units called macro variables, or you can package larger amounts of text into units called macro programs.
- You can use macro variables to substitute text into a SAS program, which makes the program easily adaptable.

Basic Concepts of Macro Variables

```
&macro-variable-name
```

- Macro variables can supply a variety of information, including operating system information, SAS session information, or text strings.
- There are two types of macro variables: automatic macro variables and user-defined macro variables.
- To use a macro variable in a program, you reference it in your code by preceding the macro variable name with an ampersand. When you submit the program, the macro processor resolves the reference before the program compiles and executes.
- You can reference a macro variable anywhere within a SAS program except within datalines. To reference a macro variable within quotation marks, you must use double quotation marks.

Using Automatic Macro Variables

- Automatic macro variables contain system information such as the date and time that the current SAS session began. SAS creates these automatic macro variables when the SAS session begins, and they are always available.
- Common automatic macro variables include SYSDATE, SYSDATE9, SYSDAY, SYSLAST, SYSSCP, SYSTIME, and SYSVER.

Creating and Using Your Own Macro Variables

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

- You use the %LET statement to create a macro variable and assign a value to it. The name that you assign to a macro variable must follow SAS naming rules. The value can be any text string. You don't need to enclose the value in quotation marks.
- SAS stores all macro variable values as text strings, even if they contain numbers. SAS doesn't evaluate mathematical expressions in macro variable values. SAS stores everything in a %LET statement between the equal sign and the semicolon, except leading and trailing blanks, as the value of the macro variable.

Displaying Macro Variables in the SAS Log

```
OPTIONS SYMBOLGEN;
```

%PUT *text*;

- Because the macro processor resolves macro variable references after a program is submitted but before it is compiled and executed, you cannot see the value that gets substituted into the program.
- You can use the SYMBOLGEN system option to control whether or not SAS writes messages about the resolution of macro variable references to the SAS log. The default value of this option is NOSYMBOLGEN.
- You can use the %PUT statement to write messages to the SAS log. If you include a macro variable reference in a %PUT statement, SAS resolves the reference before writing the message in the SAS log, so the macro variable value appears in the log. You can also use keywords such as _USER_, _AUTOMATIC_, or _ALL_ to list macro variable values in the SAS log.

Sample Code

Using Automatic Macro Variables

```
proc print data=orion.customer_type;
title "Listing of Customer_Type Data Set";
footnote1 "Created &SYSTIME &SYSDAY, &SYSDATE9";
footnote2 "on the &SYSSCP System Using SAS &SYSVER";
run;
```

Creating and Using User-Defined Macro Variables

```
options symbolgen;
%let year=2007;

proc print data=orion.order_fact;
   where year(order_date)=&year;
   title "Orders for &year";
run;

proc means data=orion.order_fact mean;
   where year(order_date)=&year;
   class order_type;
   var total_retail_price;
   title "Average Retail Price for &year";
   title2 "by Order_Type";
run;
```

Displaying Macro Variable Values in the SAS Log

```
%put _user_;
```

Part II: SAS PROGRAMMING BOOK I & II FAQs

I. General questions

1. How to download SAS?

Ans. Do you know you could download SAS software for free in NCSU. The website is http://software.ncsu.edu/vendor/sas/package/sas, when download, you need to login in with your unity ID. Unfortunately, SAS cannot be used in the mac environment. If you are a mac user, you can use SAS through VCL, https://vcl.ncsu.edu/.

2. What is the SAS programming standard?

Ans. 1. All programs should be well organized and easy to follow;

2. Programs should have a complete header comment, including students name, date, assignment name, goal of program, and data files used;
3. Comments should be used throughout the program to identify and explain the rationale for each important section of the code;
4. Students should check data to ensure it is properly read-in, merged, sorted, etc;
5. There should be no errors or warnings in the log file when the code is run;
6. Student should follow all specifications in the assignment;
7. Student should complete all the tasks in the assignment;
8. Program and output should be correct;
9. Homework should submit with a .sas code file, a log file and an output file.

3. What is Group discussion direction?

Ans. The group discussion direction contains grading policy and discussion rules.

Grading policy: Participators who post their discussion and reply other`s discussion will receive full credit. Absent a group discussion will receive 0 credit.

Discussion rules: Each group should have **only one** group discussion post. The first person who creates the group discussion board should include an **appropriate title** such as "Group discussion Week #, Group Letter", or follow specific directions provided. Other members should only post their discussions on that forum; only one discussion forum/thread per group each week.

4. SAS programming template.

Ans. The following is an example of a SAS program that follows 1. SAS programming standards:

```
/***Your Name here
    Date here
    ST 555
    Homework number
    Data file(s)
    Goal: What is the goal for this program, for example you could write:
        To apply concepts from the chapter # to a set of problems
        or To use proc print/proc means/where statement ***/
```

(Header contains name, date, course and homework number, data file and purpose of this programming)

```
/*Problem 1: To create a dataset*/ (State the problem number)

proc contents data=dataset; *viewing variable names and attributes;
(State what are you trying to do in this step)
    title; *clearing titles;
run;
data dataset; *question 1a: Creating and specifying the parameters of dataset;
    /* program.... */
run;
proc print data=dataset; *Printing the results of the dataset;
    title 'I am the dataset hahahaha'; *Giving the printout an appropriate title;
run;

/* questions 1a, how many observations in your log?  100 observations  */
```

(Use comments to fill out the blank for questions.)

II. FAQs from SAS Programming Book I

1. Why "_ALL_NODS" does not work?

Ans. Notice a space is needed after the second underscore and before NODS, the correct answer is: _ALL_ NODS.

2. When I list data files used in the header comment, do I need to list the datasets typed in the SAS program or only the data files imported into the program?

Ans. A header for SAS is for the programmer and any of the programmer's colleagues who will also utilize the SAS program. Thus you want to provide enough information/directions for the program to make it easily for anyone to follow. The best header would include both datasets with an indication of which ones are written in using datalines and which ones are imported into SAS.

3. Is Orion is a temporal library like work?

Ans. No, Orion is a permanent dataset name.

4. Is a comment considered a SAS statement?

Ans. There are two types of statements,

1. Statements that are used in DATA step programming;
2. Statements that are global in scope and can be used anywhere in a SAS program.

so yes a comment is a SAS statement.

http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000289375.htm

5. What is a step and what is a step boundary?

Ans. **Step**: there is no one link that succinctly defined a **Step** but here is text verbatim from SAS: "A SAS program is one or more steps that you submit to SAS for processing. There are two kinds of steps: DATA steps and PROC steps."

Step boundaries:

The definition of step boundaries is as follows:

When programming in SAS, step boundaries determine when programming statements, DATA Steps, and PROC steps take effect.

The following are considered step boundaries in SAS:

DATA statement

ENDSAS statement

PROC statement

QUIT statement

RUN statement

The single semicolon (;) statement following a DATALINES statement

The four semicolon (;;;;) statement following a DATALINES4 statement

The very last line of a program in batch or non-interactive processing

http://support.sas.com/kb/40/887.html

6. If not given specifics, how do we validate a variable?

Ans. To validate a variable, you should write a step to exam the summary statistics for the variable; an example could be to check the extreme values.

7. Why doesn't Proc FREQ print out missing values? How can I print them out?

Ans. Here is the answer based on SAS website:

http://support.sas.com/documentation/cdl/en/procstat/63104/HTML/default/viewer.htm - procstat_freq_sect016.htm

If an observation has a missing value for a variable in a TABLES request, by default PROC FREQ does not include that observation in the frequency or crosstabulation table. Also by default, PROC FREQ does not include observations with missing values in the computation of percentages and statistics. The procedure displays the number of missing observations below each table. In order to print out the missing values, you can use MISSPRINT or MISSING option under Proc Freq statement.

8. When should I include a period in the format statement?

Ans. We don`t need to include a period when defining a format statement, but we need to include a period when applying the format statement. For a character format, it is also necessary to add a '$' sign before the format name.

9. Why the following program`s output is Sales R? (Question 8 from Lecture P1ch5)

```
proc format;

format $Title

'Sales Manager',

'Senior Sales Mgr'='Manager'

'Sales Rep. I' ,

'Sales Rep. II'='Rep' ;

run;
```

What would be the output for 'Sales Rep II'?

The correct answer is: Sales R

Ans. Notice that in the SAS program, no format is given for Sales Rep II The program gives a format for 4 specific job titles: a format of 'Manager' is given for 'Sales Manager' and 'Senior Sales Mgr' and a format of 'Rep' is given for 'Sales Rep. I' and Sales Rep. II'; notice these include periods but "Sales Rep II" does not have a period after Rep. Thus "Sales Rep II" and anything else beyond the 4 specific job titles in the proc format is displayed with the default length.

Now how is the default length defined. Usually the default length is 8. However Sales R is a length of 7. Why? The reason is if you don't specifically set the length, SAS sets the length based off the first value listed in the dataset for that variable. In this case the first value for Job Title in the orion.sales dataset is "Manager" which has a length of 7. Thus it's setting a length of 7 for all values not included in the proc format.

If we ask for a print out, 'Sales Rep. II' will be displayed as 'Rep', but for 'Sales Rep II, the value displayed will be 'Sales R'.

10. Why the answer is "No output destinations active"? (Question 10 from Lecture P1ch11 Part2)

Suppose you submit the program shown below, which happens if you then submit a PROC PRINT step?

```
ods _all_ close;
ods csvall file = 'c:\ctry.csv";

proc freq data = orion.sales;
    tables Country;
run;

ods csvall close;
```

The correct answer is: The PROC PRINT output is not displayed and a warning is written to the log indicating that there are no active destinations.

Ans. (*answer from a student*) The first ods _all_ close statement closed all active destinations at the beginning of execution. The next ods csvall statement opened only the csvall destination. After you close that csvall destination near the end of the code, you go back to no output destinations being active. From that point in the code, no proc output will occur.

11. When is it better to use a concatenate step and a merge step? Is there a point where one is more optimal than the other?

Ans. (*answer from a student*) Concatenating appends files together. So if we had two files that we really just needed to become one, we would use concatenate. For instance, Sales by franchise for east coast, and Sales by franchise for west coast, become national sales. We are not really doing anything to the observations, just stacking them together. And they do not have to have anything in common, though it would help in most tasks I can think of.

Merge actually will look for a common identifier that you choose and will merge observations in one observation with another. This is not just simply appending files. Merge will find the like observations and utilize all the variables to make a new merged observation. Say if we tried to merge the above East/West coast files by Franchise Number, the output would have zero observations. Since all the west coast observations would have different franchise numbers than the east coast franchises, there would be no matches- but if we concatenated, all franchises east & west would show.