**SAS Programming 1: Essentials**

**Try Using SAS Studio**

In this short exercise, you try the programming tasks you'll perform in this course - you browse data, submit code, check the log, and view results.

1. Open SAS Studio.
2. Open the **Cars** data set in the **Sashelp** library and browse the data. How many rows are in the data set?
   * Click **Libraries** in the navigation panel and open the **Sashelp** library, then open the**Cars**data.
   * Scroll through the data. There are 428 rows in the **Cars** data set.
   * Close the data viewer.
3. The example program below creates a report that lists the number and percentage of each type of car in the data set **Sashelp.Cars**. Copy and paste the program into the Editor window and submit it.

title "Car Inventory by Type";

proc freq data=sashelp.cars;

tables Type / nocum;

run;

title;

* + Copy the code (highlight the code and press **Crtl-C**) and paste it (press **Ctrl-V**) into the **Code** tab.
  + Click the **Run** button to submit the code to SAS.

1. View the log to verify that the code ran successfully. What note do you see?
   * Click the **Log** tab.
   * Check the Log Summary to be sure that there are no warning or error messages in the log. In the list of notes, you should see the following note:  
     There were 428 observations read from the data set SASHELP.CARS.
2. View the results. The **Frequency** column represents the number of each type of car. Which type had the highest number of cars in the data?
   * Click the **Results**tab.
   * In this data set, most cars have the type Sedan -- 262 sedans accounted for about 61% of the cars in the data.

# **Setup Instructions for SAS Studio**

To complete the practices in this course, you must follow these steps to set up the practice files for SAS Studio.   
  
**IMPORTANT:** You must perform these two tasks in the **same** SAS Studio session. Do not close your browser until you have completed both tasks or you will need to start again.

## Task 1: Create a folder for your practice files and define the orion library.

1. Start SAS Studio.
2. At the top of the **Files and** **Folders** pane, click **New**  and select **Folder**. Note the path to the location of your folder.
3. In the **Name** box, type **ecprg193**. Click **Save**.
4. The following code creates a macro variable to store the location of your practice files and defines the **orion** library. Copy and paste the following two lines of code into the **Code** tab in SAS Studio. Don't run the code yet - you must edit the code before you run it.

%let path=FILEPATH;

libname orion "&path";

1. In the **Files and Folders** pane, navigate to the location where you created the new folder. Right-click **ecprg193** and select**Properties**. Highlight the filepath shown in **Location** and copy it.
2. In the **Code** tab in SAS Studio, highlight FILEPATH and paste the filepath that you copied in the previous step.
3. Click **Run** to submit this program to SAS.
4. Check the log to make sure that the libref **orion** was successfully assigned.
   1. You should see this message in the log:

NOTE: Libref ORION was successfully assigned   
You have defined the **orion** library and SAS knows where you are storing your practice files.

* 1. If the program does not run successfully, make sure you created the folder that you are referencing, and review your program for errors. Correct the errors and resubmit the program.

1. Click the **Code** tab and click**Add to My Snippets**. In the **Name** box, type **setup** and click **Save**. In the confirmation window, click **Close**.
2. Close the **setup** program tab in the workspace.

**IMPORTANT:** Each time you start SAS Studio, open the **Snippets** panel and expand the **My Snippets** folder. Double-click the**setup** snippet to add it to the **Code** tab. Run the code to define the **orion** library so that you have access to your practice files.

## Task 2: Create the practice files for the course — you only have to do this once.

1. At the top of the **Files and Folders** pane, click **New**  and select **SAS** **Program**.
2. Click [here](javascript:%20openOther3('../createdata.txt');) to open a popup window with the SAS code that creates your practice data. Select all of the code (**Ctrl+A**or**Command+A**) and then copy it (**Ctrl+C** or **Command+C**).
3. Paste the code (**Ctrl+V**or**Command+V**) into the **Code** tab in SAS Studio.
4. Click **Run** to submit the program to SAS. This program creates the practice files in the **ecprg193** folder.
5. Check the log to verify that there are no errors or warnings. There will be many notes because SAS writes a note for each data set that the program creates.  
     
   **NOTE:** If your program does not run successfully, make sure you completed all the steps in**Task 1** correctly.
6. On the **Code** tab, click **Clear All code** . You have set up your practice data and you are ready to work in SAS Studio.   
     
   **NOTE**: Unless you delete your practice files folder, you do not need to perform this task again.

**You have set up your practice data.** Each time you start SAS Studio to practice in this course, open the **Snippets** panel and expand the **My Snippets** folder. Double-click **setup** to add the code to the **Code** tab. After you run the program, you can access your practice data for the course. Each practice page reminds you to define the **orion** library and has a link to this setup page.

**Lesson 1: Getting Started with SAS Programming**

#### Introduction

In this lesson, you’ll get an overview of SAS software and how you can use SAS to access, manage, analyze, and present your data. You’ll explore the SAS programming process and learn the iterative manner of working with SAS programs. In short, you'll learn some important SAS concepts that lay the foundation for you as a SAS programmer.

#### Objectives

* describe SAS capabilities
* explain the SAS programming process
* identify the types of files used in SAS

## Exploring SAS

#### What Is SAS?

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 provides a flexible and extensible programming language designed for data access, transformation, and reporting.  
  
To extend the capabilities of Base SAS, you can add other SAS components. For example, you can use a component to access third-party data. Other components give you tools for report writing, high-resolution graphics, statistical analysis, visualization and discovery, and business solutions.

#### The SAS Framework

Let’s look at all of these SAS capabilities in a simple framework. No matter what type of business or industry you work in, you need to access your data. You might have data stored in SAS, in a raw data file, in Oracle, in Excel, or in other types of files. Using SAS, you can read any kind of data.   
  
Once you access your data, you can manage it. For example, you might need to subset data, create variables, validate and clean data, or combine data to ready it for analysis. SAS gives you excellent data management capabilities. You’ll probably want to analyze your data as well. You can perform some simple analyses, such as finding frequency counts or calculating averages. Or you can run more complex analyses, such as regression or forecasting. For statistical analysis, SAS is the gold standard.   
  
Finally, you'll want to present your data meaningfully. You can create list reports, summary reports, or graphic reports. And you can print these reports, write them to new data files, or publish them on the web. You have lots of options for presenting your data.

## Understanding the SAS Programming Process

#### Exploring the SAS Programming Process

Now that you know a little bit about the power of SAS, let’s take a look at the overall programming process in SAS.   
  
The first step in the programming process is to define the business need. You do this by communicating with the business team or by reviewing a written specification. After you define the business need, you write a SAS program based on the desired output, the necessary input, and the required processing. After you finish coding, you run the program and review your results, which can be reports or notes and messages from SAS regarding your code. As you review the results, you might find inaccuracies or errors, in which case you might need to debug or modify the program. Depending on your results, you might need to repeat some of the steps.

#### Types of Files Used in SAS

As you know, the power of SAS is that you can use it to read any type of data. Let's take a few moments and learn about the three major file types you'll use in this course: raw data files, SAS data sets, and SAS program files. Keeping the overall programming process in mind, let’s see how you use each type of file.  
  
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.   
  
The second major type of file you’ll use is a SAS data set. This important type of file is specific to SAS. A SAS data set is your 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.   
  
Now let’s explore the third major type of file you’ll use in SAS: the SAS program file. SAS program files contain SAS programming code. These instructions tell SAS how to process your data and what output to create.

## Topic Summaries

### Exploring SAS

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.

### Understanding the SAS Programming Process

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.

# **Lesson 2: Working with SAS Programs**

## Lesson Overview

#### Introduction

In this lesson, you'll learn how to work with SAS code. First, you'll learn the main components of SAS programs. Then you'll learn the syntax rules and formatting guidelines for writing SAS programs. As you work with SAS programs, you'll add descriptive comments, and identify and correct common syntax errors.

#### Objectives

* list the components of a SAS program
* identify the characteristics of SAS statements
* define SAS syntax rules
* document a program using comments
* identify common syntax errors
* diagnose and correct syntax errors in a SAS program

## Exploring SAS Programs

#### Understanding SAS Programs

Let's investigate the main components of SAS programs. Generally speaking, a SAS program is a sequence of steps that you submit to SAS for execution. Each step in the program performs a specific task. Only two kinds of steps make up SAS programs: DATA steps and PROC steps. A SAS program can contain a DATA step, or a PROC step, or any combination of DATA steps and PROC steps. The number and kind of steps depend on what tasks you need to perform.   
  
A DATA step typically reads data from an input source, processes it, and creates a SAS data set, which is data in a form that SAS understands. So, one of the primary purposes of a DATA step is to create a SAS data set. In addition, you can use a DATA step to create new variables that were not in your original data. In SAS terminology, variables are the columns in your data.   
  
For example, suppose your raw data file contains the fields **Cost Price Per Unit** and **Quantity Sold**. In a DATA step, you can multiply these variables and assign the value to a new variable named **Total\_Retail\_Price**.   
  
A PROC or procedure step typically processes a SAS data set. Various PROC steps generate reports and graphs, manage data, and sort data.   
  
One way to use these two steps together is to use a DATA step to create a SAS data set, and then use a PROC step to create a report. Remember, though, that this is just one possible combination of steps in a SAS program. Your SAS programs might perform other tasks. Now let's learn more about what makes up a SAS step.

#### SAS Programming Steps

A SAS program is comprised of a sequence of steps, and a step is comprised of a sequence of statements. Every step has a beginning and ending boundary. These are calledstep boundaries. SAS compiles and executes each step independently based on the step boundaries.   
  
A DATA step begins with a DATA statement, and a PROC step begins with a PROC statement. SAS detects the end of a step when it encounters one of the following: a RUN statement for most steps, a QUIT statement for some procedures, or the beginning of another step. Occasionally, a user might omit a RUN or QUIT statement, and the step will end implicitly when the next step begins. It is a best practice to include a RUN or QUIT statement to explicitly end each step in a SAS program.   
  
Take a look at this 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;

Can you tell how many steps it contains? This program contains three steps: one DATA step and two PROC steps. In the first line of code, the DATA step creates a temporary SAS data set named **work.newsalesemps** by reading the **orion.sales** data set. In the eighth line, the PROC PRINT step creates a list report of the **work.newsalesemps** data set. In line 11, the PROC MEANS step creates a summary report of **work.newsalesemps** with statistics for the variable **Salary** for each value of **Job\_Title**.   
  
In addition to DATA and PROC steps, this SAS program also contains global statements. These statements can lie outside DATA and PROC steps, and they can affect more than one step. For example, the first TITLE statement located before the PROC statements, specifies a title that appears on both reports. The second TITLE statement located at the end of the SAS program, turns all titles off for all subsequent output. You'll learn several global statements in this course.

#### Submitting a SAS Program

In this demonstration, you submit a program and examine the log and results.

1. Copy and paste the following program into the editor.

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;

1. Submit the code and check the log. It's a good programming practice to first check the log, even if the program appears to produce results. You want to ensure that the code ran successfully before you look at any reports SAS created. Notice that SAS processed the code without warnings or errors.
2. View the results. The first report is the PROC PRINT report. Recall that this type of report simply lists your data. You can see columns for the various variables and all of their values. Notice that the title you specified appears at the top of the report. The next report is the PROC MEANS report. The MEANS procedure provides data summarization tools to compute descriptive statistics on your data, and displays output by default. Here, SAS calculated statistics for the analysis variable **Salary**.

#### Business Scenario

Orion Star management encourages their programmers to write well-formatted, clearly documented SAS programs. So, you need to know the syntax rules and recommended structure for SAS programming statements, as well as how to use comments in your SAS programs.

#### Characteristics of SAS Programs

SAS statements usually begin with an identifying keyword, and they always end with a semicolon. Keywords identify the type of statement, and semicolons end the statement. For example, in the following SAS program, the second statement is a SET statement, and the fourth statement is a RUN statement.

data work.newsalesemps;

set orion.sales;

where Country='AU';

run;

proc print data=work.newsalesemps;

run;

proc means data=work.newsalesemps;

class Job\_Title;

var Salary;

run;

Take a look at this program. Can you tell how many statements make up this DATA step?

data 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;

This step contains five statements: a DATA statement, a LENGTH statement, an INFILE statement, an INPUT statement, and a RUN statement. Each statement has an identifying keyword and ends in a semicolon.

#### SAS Program Structure

In the following program, the statements are pretty easy to read.

data work.newsalesemps;

set orion.sales;

where Country='AU';

run;

proc print data=work.newsalesemps;

run;

proc means data=work.newsalesemps;

class Job\_Title;

var Salary;

run;

The DATA, PROC, and RUN statements begin in column one, and the other statements are indented. Each statement begins on a new line, and a blank line separates each step. Using conventional formatting (that is, structured, consistent spacing) makes a SAS program easy to read.   
  
However, SAS statements are free format. In other words, they can begin and end anywhere. In SAS, you can have as much or as little white space as you want. You can begin or end a statement in any column and span multiple lines. You can also place multiple statements on one line, and unquoted values can be lowercase, uppercase, or mixed case.  
  
The following program takes advantage of the free-format style that SAS permits, but at a cost of being difficult to read:

data work.newsalesemps;set orion.

sales;where Country='AU';run;

proc print data=work.newsalesempls;

run;proc means

data=work.newsalesemps;class

Job\_Title;var Salary;run;

Remember the old saying: "Just because you can do something, doesn't mean that you should." Again, in this program, the SAS syntax rules have been followed, but this unconventional formatting might be especially difficult for other programmers to read.   
  
Using conventional formatting can take the guesswork out of your programs. It's recommended that you use a conventional programming style. Click the **Information** button in the course to learn about automatic formatting in SAS Enterprise Guide and other SAS environments.

#### Using SAS Comments

In addition to using conventional formatting, another way to make your program easier for others to follow is to add comments to the program. A comment is text in your program that SAS ignores during processing but writes to the SAS log. You can use comments anywhere in a SAS program to document the purpose of the program, explain segments of the program, or mark SAS code as non-executing text. Using comments to mark SAS code as non-executing text is also called commenting out code.   
  
Comments can also help you test your SAS programs in stages. By commenting out your error-free code, you can use comments to submit only the steps that you're testing. When your entire program is error-free, you can remove the comment symbols without damaging the SAS program.

#### Types of Comments

Let's take a closer look at comments. In SAS, you can create comments in two ways. Using the first method, called a block comment, you begin with a forward slash and asterisk, your comment text, and then end with an asterisk and a forward slash.

/\* create a temporary data

set, newsalesemps, from

the data set orion.sales \*/

data work.newsalesemps;

set orion.sales;

where Country='AU';

run;

proc print data=work.newsalesemps;

run;

proc means data=work.newsalesemps;

class Job\_Title;

var Salary;

run;

These comments can be any length, and can contain semicolons. They cannot be nested. You should avoid placing block comment symbols in the first or second columns. In some operating environments, SAS might interpret block comment symbols in columns 1 and 2 as a request to end the SAS job or session.   
  
The second method is called a comment statement. It begins with an asterisk, followed by the comment text, and ends with a semicolon.

\*create a temporary data set,

newsalesemps, from the data set

orion.sales;

data work.newsalesemps;

set orion.sales;

\*where Country='AU';

run;

proc print data=work.newsalesemps;

run;

proc means data=work.newsalesemps;

class Job\_Title;

var Salary;

run;

Comment statements can begin in columns 1 and 2. To comment out a statement in one of these steps, you simply add an asterisk to the beginning of the statement, as shown above in the WHERE statement. Comments in this form are complete statements, and they can't contain internal semicolons.

#### Adding Comments to Your SAS Programs

In this demonstration, you add comments to a program to make sure that another programmer understands it.

1. Copy and paste the following code into the editor.

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;

run;

1. At the beginning of the program, add a comment statement stating that you're using**orion.sales**to create **work.newsalesemps**.

\*This program uses the data set orion.sales to create work.newsalesemps.;

1. In the PROC MEANS step, add a block comment stating that the variable **Salary** is numeric. Place the comment immediatately following the variable name. Remember that SAS ignores any text between the comment symbols.

proc means data=work.newsalesemps;

class Gender;

var Salary/\*numeric variable\*/;

run;

1. Next, comment out the PROC PRINT step so that it doesn't run when you submit the code.

/\*

proc print data=work.newsalesemps;

run;\*/

1. Submit this code and examine the log. Notice that SAS didn't process the portions of code that were commented out. You can see that the data set was created, and that the PROC MEANS step created output, but the PROC PRINT step that was commented out produced no other messages or output.

## Diagnosing and Correcting Syntax Errors

#### Business Scenario

As an Orion Star programmer, you work with a lot of code…some that's yours and some that's not. You need to be able to diagnose and correct syntax errors in any of these SAS programs.

#### What Is a Syntax Error?

Syntax errors occur when program statements do not conform to the rules of the SAS language. Some common syntax errors are misspelled keywords, missing semicolons, and invalid options. The editor uses the color red to indicate a potential error in your SAS code. Notice that in the first line of the program below SAS displays the misspelled word DAAT in red. 

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;

This misspelling affects other statements following it. Although the following statements in the DATA step are syntactically correct, they are only permitted in a DATA step. The editor doesn't recognize this as a DATA step though, due to the misspelled keyword, so SAS also displays the other statements in the DATA step in red.   
  
SAS finds syntax errors during the compilation phase, before it executes the program. So, when you submit a SAS program, SAS scans each statement for syntax errors. If no errors are found, SAS executes the step when it reaches the step boundary. Then SAS goes to the next step and repeats the process.   
  
When SAS encounters a syntax error, it writes the following to the SAS log: the word ERROR or WARNING, the location of the error, and an explanation of the error. SAS continues the syntax scan until it reaches the step boundary, but the step doesn't execute if errors are found. Then SAS continues scanning the rest of the program, and reports any additional errors as needed. When you check the log, as all good SAS programmers do, and find a warning or error message, you need to correct your code.

#### Viewing and Correcting Syntax Errors

In this demonstration, you diagnose and correct syntax errors in your program.

1. Copy and paste the following program into the editor. As you know, the DATA step keyword is misspelled. Also, the semicolon is missing from the PROC PRINT statement, and the PROC MEANS step includes an option that is not valid. As you can see, SAS color-codes the program to indicate the 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 min;

var Salary;

run;

1. Submit the program and check the log. You should always check the log to make sure that the program ran successfully, even if output is generated.

Notice that there is a WARNING message and the word DAAT is underlined. In this case, SAS resolved the issue by assuming that DAAT was simply DATA misspelled. A warning means that SAS was able to perform the action. In this case, SAS processed the DATA step. But this is a rare situation, as SAS might not always be able to interpret your misspelled words.   
  
Next, notice that the RUN statement is underlined. In this case, the previous line is missing the semicolon. The message 'Syntax error, expecting one of the following...' indicates that something was missing. Consider how SAS processed this step. SAS started with the PROC PRINT statement and kept going until it reached the semicolon at the end of the RUN statement. So, SAS thought that the PROC PRINT and the RUN statements were all one statement. SAS interpreted RUN as an option for PROC PRINT and printed an error message about an invalid option. Notice that SAS did list the semicolon as one of the expected options.   
  
You might be thinking, “Why did SAS report an error in the RUN statement? There's nothing wrong with the RUN statement.” When you encounter this type of error, always check the statement before the underlined statement. In many cases you will find that the statement before the error is missing a semicolon.  
  
Now look at the next error message. SAS did not recognize the word AVERAGE as a valid option in the PROC MEANS statement, so the PROC MEANS step didn't execute. Notice that SAS lists the valid options. The word MEAN is listed as a valid option and should be used to calculate an average.

1. In the editor, correct the program. First, correct the spelling of DATA, and then add a semicolon to the end of the PROC PRINT statement. Lastly, change the word AVERAGE to MEAN in the PROC MEANS statement.

data 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 mean max;

class Job\_Title;

var Salary;

run;

1. Submit the revised code and check the log. The log shows that the code ran successfully. No errors or warnings appear. Also, SAS produced the reports you requested. As demonstrated, you can easily view and correct syntax errors in SAS.

#### Business Scenario

Another common mistake that programmers make is leaving off a matching quotation mark. For example, suppose you write a program that creates a data set and generates two reports. You submit the program, but it doesn't produce results. The program might have unbalanced quotation marks.

#### Unbalanced Quotation Marks

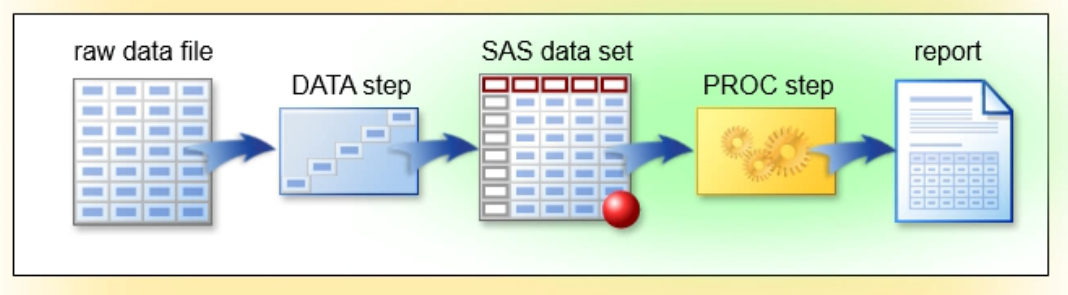
In SAS, a quotation counter keeps count of the quotation marks in your code. SAS expects an even number, or matching number, of quotation marks. If SAS detects an uneven number of quotation marks, the code won't execute properly. Also, although SAS allows either single or double quotation marks, you can't mix the types. If you begin with a single quotation mark, you must end with a single quotation mark; otherwise, SAS considers the quotation marks unbalanced. When your program contains unbalanced quotation marks, whether from an uneven number or mismatched quotation marks, SAS misreads both the statement containing the error and any following statements.

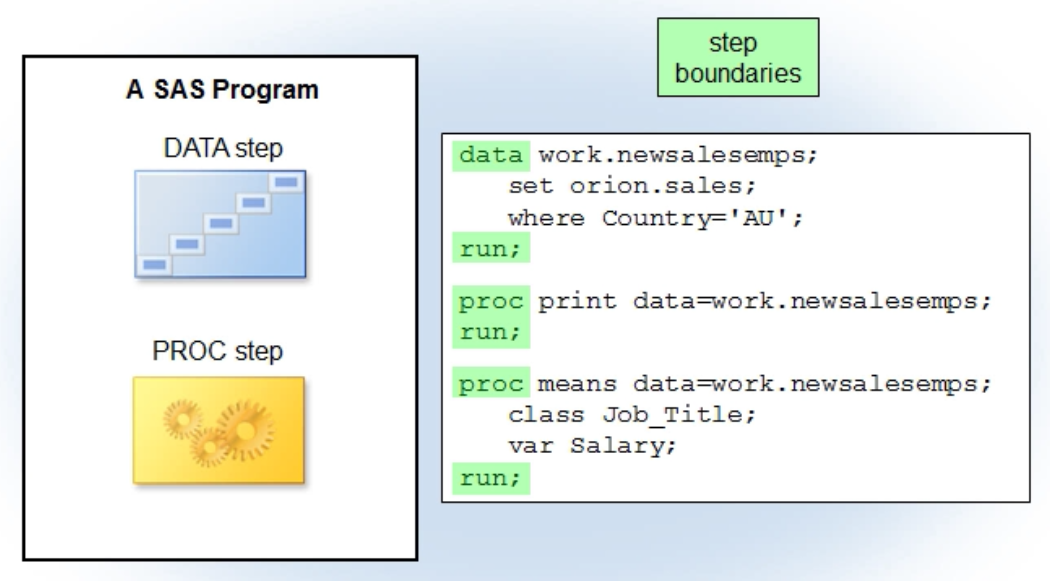
You should notice that there's a problem because much of the program will be colored purple in the editor. Purple represents a quoted string. In this example, the string begins with a single quotation mark followed by a comma, a semicolon, and then all the remaining statements in the program. Because the string does not contain a matching or ending quotation mark, SAS reads all of this text as a quoted string.

When you submit a program with unbalanced quotation marks in the SAS windowing environment, the program doesn't stop running, and the log includes only the code you submitted. You won't see any error or warning messages, nor will you see any indication that any of the steps executed. You'll also see a message in the banner of the editor stating that the step is still running. You have to stop an executing program by cancelling the submitted statements. You can then correct your program by adding the missing quotation mark.

When you submit a program with unbalanced quotation marks in client applications such as SAS Enterprise Guide and SAS Studio, SAS writes messages to the log to alert you of the error. A warning in the SAS log stating that a quoted string has become too long, or that a statement containing quotation marks is ambiguous, sometimes indicates unbalanced quotation marks. In fact, any log message about a quoted string should alert you to the possibility of unbalanced quotation marks. In client applications, SAS submits additional code, or wrapper code, including a single and double quotation mark. SAS is attempting to repair any potential unbalanced quotes in a submitted program. The wrapper code balances quotation marks and the code stops running, but your results will still contain errors and you must correct the program. To do this, you either add the missing quotation mark, or match the quotation mark, and then resubmit the program.

For more information on correcting unbalanced quotation marks in the SAS windowing environment and client applications, click the **Information** button.

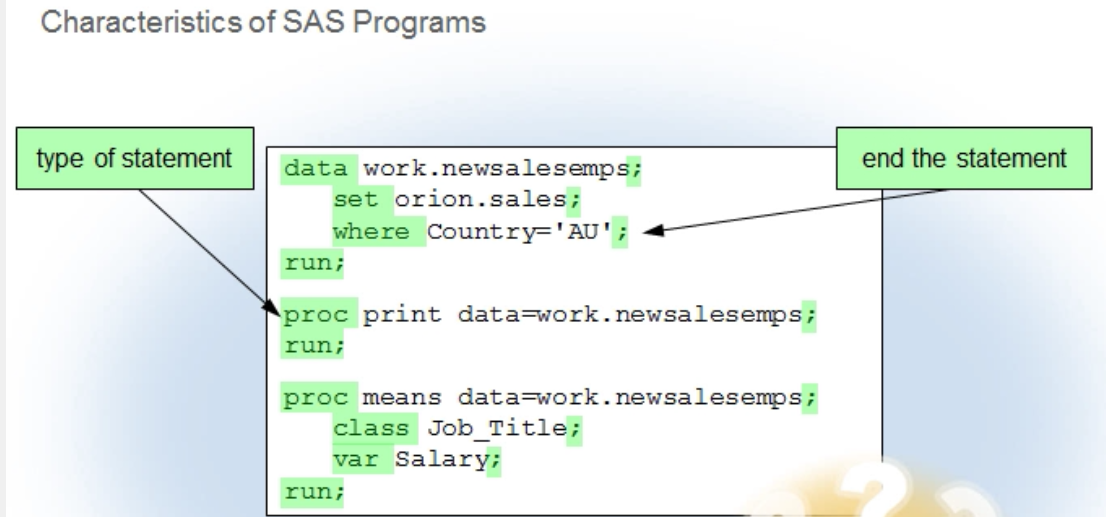


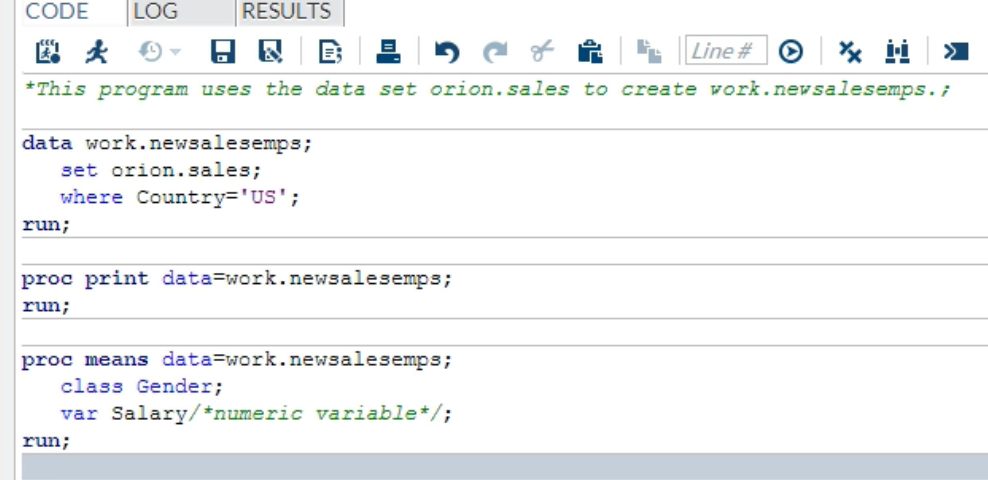


Which of the following can represent a step boundary? 

Top of Form

|  |  |
| --- | --- |
|  | *a.*  a RUN statement |
|  | *b.*  a QUIT statement |
|  | *c.*  a DATA statement |
|  | *d.*  a PROC statement |
|  | *e.*  all of the above |





The DATA step executes and creates an output data set. The PROC PRINT step executes and produces a report. The PROC MEANS step is commented out, and therefore does not execute.

## Topic Summaries

### Exploring SAS Programs

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**;** |

### Diagnosing and Correcting Syntax Errors

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.

## 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;

# **Lesson 3: Accessing Data**

## Lesson Overview

#### Introduction

In this lesson, you learn to access and view the Orion Star data sets that you'll be working with. First, you'll learn to write a simple program to define a SAS library. Then, you'll view the contents of the library and examine individual data sets. You’ll explore the two portions of a SAS data set, and investigate variable attributes. Finally, you’ll learn the SAS naming conventions for variables and data sets.

#### Objectives

* explain the concept of a SAS library
* state the difference between a temporary library and a permanent library
* assign a library reference name to a SAS library by using the LIBNAME statement
* explore the contents of SAS libraries using the CONTENTS procedure
* access a data set in a user-created permanent library
* define the components of a SAS data set
* browse the descriptor portion of a SAS data set using the CONTENTS procedure
* browse the data portion of a SAS data set using the PRINT procedure
* identify the two main types of values (character and numeric) and missing values
* explain the SAS naming conventions for variables and data sets

## Accessing SAS Libraries

#### Business Scenario

You know the three major types of files used in SAS: raw data files, SAS data sets, and SAS program files. But you haven’t yet learned how SAS organizes and stores these files, or more importantly, how you can access them. For example, as a new SAS programmer at Orion Star, you need to access existing SAS data sets and use them to perform your duties. You need to know about SAS libraries.

#### Exploring SAS Libraries

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 and that are referenced and stored as a unit. A library is the highest level of organization for information within SAS. You can think of a SAS library as a drawer in a filing cabinet and a SAS data set as one of the files in the drawer. Each file is referred to as a member of the library.   
  
At the beginning of each SAS session, SAS automatically provides one temporary and at least one permanent SAS library that you can access. These libraries, or drawers, are open and ready for use. SAS provides a temporary library called **work** where you can store and access SAS data sets for the duration of the SAS session. At the end of every SAS session, SAS deletes the work library and its contents.   
  
SAS also provides permanent libraries. SAS data sets in permanent libraries are saved after your SAS session terminates. For example, **sashelp** is a permanent library that SAS makes available. It contains sample SAS data sets that you can access anytime you start a SAS session. Traditionally, SAS defines a permanent library named **sasuser** that you can use for storing and accessing SAS data sets during your SAS session. At your site, there might be different permanent libraries that are defined for you to use. These data sets will be available in later sessions as well.

#### Accessing SAS Libraries

Regardless of the operating environment you use, you refer to a SAS library by a logical name called a library reference name, or libref. A libref references a particular physical location that the operating environment recognizes, so you can think of a libref as a shortcut to that physical location. So, **work**, **sasuser**, and **sashelp** are librefs that refer to physical locations. The implementation of a SAS library corresponds to the way that your operating environment stores files. In the Windows operating environment, for example, a libref typically refers to a group of SAS files in the same folder or directory.

#### Using Two-Level Data Set Names

All SAS data sets have a two-level name that consists of the libref and the data set name, separated by a period. When a data set is in the temporary **work** library, you can optionally use a one-level name. A one-level name consists of just the data set name, such as **newsalesemps**. When you specify a one-level name, SAS assumes that the data set is stored in the **work** library. So, **work** is the default libref.   
  
When the data set is in a permanent library, you must use a two-level name. Let’s take a look at the following program to further understand how SAS data sets are named and how you refer to them in code.

data **work.newsalesemps**;

set orion.sales;

run;

title 'New Sales Employees';

proc print data=work.newsalesemps;

run;

proc means data=work.newsalesemps;

class Job\_Title;

var Salary;

run;

title;

In this program, the DATA step creates a temporary data set named **newsalesemps** by using the two-level name **work.newsalesemps**. Both of the PROC steps reference this data set. When the current SAS session ends, SAS deletes the **newsalesemps** data set, along with any other data sets that are stored in the **work** library.

#### Business Scenario

Suppose you know the physical location of the SAS data sets that contain Orion Star data. You want to define a SAS libref named **orion** to access and view those data sets. After you define the libref, you can explore the data sets and reference the data sets in your SAS programs.

#### Creating SAS Libraries

You can create and access your own SAS libraries. A user-created library has the following characteristics. It’s permanent, meaning that the data sets are stored there until you delete them. It’s implemented within the operating environment’s file system. And it’s not automatically available in a SAS session. You must assign a libref to a user-created library to make it available in a SAS session.   
  
Let’s look at how you define a library. First, you identify the location of the library to SAS. For example, suppose you have Orion Star data stored in a Microsoft Windows folder, and you want to use the folder as your SAS library. Your operating system knows about the folder, but SAS doesn’t. To use this folder as your SAS library, you must tell SAS where it is. In other words, you need to make a connection between the folder containing your data and SAS.

#### Using the LIBNAME Statement

In SAS code, you use the LIBNAME statement to associate the libref with the physical location of the library, that is, the physical location of your data. The LIBNAME statement makes the SAS library—your data—available for the duration of your current SAS session.

Let's explore the syntax.

|  |
| --- |
| **LIBNAME**libref  'SAS-data-set' <options>**;** |

You begin with the keyword LIBNAME. Next, you specify the name of the libref. A valid libref must follow several rules. It must have a length of one to eight characters, and must begin with a letter or underscore. The remaining characters must be letters, numbers, or underscores. You then specify the physical location of the SAS library—your data. You must reference an existing folder; the LIBNAME statement does not create a new folder. You enclose the physical location in single or double quotation marks.

The following LIBNAME statement creates the libref named **orion**.

libname **orion** 'filepath';

In place of filepath, you specify the actual physical location of the library. Here's an example in the Windows operating environment.

libname orion '**c:\oriondata**';

However, depending upon your working environment, the path might look more like this.

libname orion '**/oriondata**';

The LIBNAME statement is a global statement. It's not part of a DATA or PROC step, and it doesn't need a RUN statement. You can submit the LIBNAME statement alone, or you can store it with any 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, as many as you want.

#### Accessing a SAS Library

In this demonstration, you submit a LIBNAME statement to assign the **orion** libref.

1. Copy and paste the following LIBNAME statement into the editor:
2. %let path=/dept/dvt/WebDMS/Education/Prg1PracticeFiles;

libname orion "&path";

You might recognize the code from the data setup program you used for this course. The first line of code references a macro variable that stores the location of the practice files. Your location will be different from the one shown here.   
  
The next line is the LIBNAME statement, which is the focus of this demo. You specify the **orion** libref and then the physical location of the library. Again, because of the macro variable, the location is simply &path. Remember that the location must be enclosed in quotation marks. Note that any time you reference a macro variable within quotation marks, such as in a LIBNAME statement, you must use double quotation marks. So we'll use double quotation marks here.

1. Submit the code and check the log. Verify that the **orion** libref was assigned successfully.

#### Business Scenario

The **orion** library is now active, so you have access to all of the files that it contains. You can use DATA steps and PROC steps to work with the files. Suppose you don't know what files are available in the library. To view the contents of the library, you can write a SAS program that creates a report with general information about the library. The report will also list the members of the library. Let's find out how to generate this report.

#### Browsing a Library Programmatically

To create a report that displays the contents of a SAS library, you can write a PROC CONTENTS step.

|  |
| --- |
| **PROC CONTENTS DATA=** libref.**\_ALL\_; RUN;** |

The syntax of a basic PROC CONTENTS step has two statements: a PROC CONTENTS statement and a RUN statement. The PROC CONTENTS statement begins with the keywords PROC CONTENTS, followed by the DATA= option.   
  
When you use PROC CONTENTS to list the contents of a SAS library, you indicate the library after the DATA= option by specifying the libref, a period, and the keyword \_ALL\_. When you use \_ALL\_ in the DATA= option, PROC CONTENTS displays a list of all the SAS files that are contained in the SAS library. Finally, remember that the RUN statement tells SAS to execute the preceding SAS statements.

#### Browsing a Library

In this demonstration, you submit a PROC CONTENTS step to browse a SAS library programmatically.

1. Copy and paste the following PROC CONTENTS step into the editor.

proc contents data=orion.\_all\_;

run;

1. Submit the code and view the results. You might see additional details in your PROC CONTENTS results, depending upon your SAS environment. The first table,**Directory**, lists general information about the library. The second table lists all members of the library in alphabetical order, and provides basic information about each member. As the **Member Type** column indicates, the **orion** library contains two types of SAS files: data sets and indexes. Only the data sets are numbered in the first column. The names of the data sets indicate the type of Orion Star information that you'll work with.
2. Scroll down in the report. By default, a PROC CONTENTS report also includes information about each individual data set in the library, called the descriptor portion. If a library has many data sets, a report that includes all the descriptors can be very long. To suppress the descriptor portions in the report, you specify the NODS option.
3. In the editor, add the NODS option after the \_ALL\_ keyword. You must use a space to separate \_ALL\_ from the NODS option.

proc contents data=orion.\_all\_ nods;

run;

1. Submit the code and view the results. The results now contain only the two tables with the information you're interested in working with.

#### Accessing a Permanent Data Set

Now that you’ve assigned the **orion** libref, the connection has been made between the folder containing your data and SAS. You can access the SAS files in the **orion** library. One way to do this is to use a PROC PRINT step. The syntax is very similar to the PROC CONTENTS syntax.

|  |
| --- |
| **PROC PRINT DATA=** SAS-data set**; RUN;** |

After DATA=, you specify the libref name **orion** as the first part of the two-level data set name. Then you specify the name of the data set that you want to display. For example, suppose you know that one of the data sets is named **country**. You type **data=orion.country** in your PROC PRINT step to view the data set.

#### Viewing a Data Set with PROC PRINT

In this demonstration, you view a data set with PROC PRINT.

1. Copy and paste the following PROC PRINT step into the editor.

proc print data=orion.country;

run;

1. Submit the code and check the log. Verify that SAS read 7 observations from the data set and that there are no warnings or errors.
2. View the results. You don't need to be concerned with the details of the data set. Just remember that you must have already assigned the **orion** libref in order to view and/or work with data in the **orion** library.

#### Changing or Cancelling a Libref

Now let's look at how long a libref stays in effect. In an interactive SAS session, a libref that you assign remains in effect until you cancel the libref, change the libref, or end your SAS session. You can use the CLEAR option in the LIBNAME statement to cancel, or disassociate, a libref that you previously assigned. Disassociating the libref disconnects the library from SAS.

For example, in the following program, the LIBNAME statement associates the libref **perm** with the data in the folder **myfiles**.

libname **perm** 'filepath/myfiles';

proc print data=perm.orders;

var Order\_ID Order\_Type Order\_Date;

run;

libname perm clear;

At the end of the program, another LIBNAME statement disassociates the **perm** libref. Suppose you need to specify a different physical location for the files. To change the location, you can submit a LIBNAME statement with the same libref name but with a different filepath.

When you end your SAS session, the contents of a permanent library still exist in their physical location in your operating environment, but SAS deletes everything in the **work**library. Each time you start a new SAS session, you must resubmit the LIBNAME statement for the SAS libraries that you need to use.

## Examining SAS Data Sets

#### Business Scenario

As an Orion Star programmer, you’ll work with permanent data sets in the **orion** library, as well as with temporary data sets in the **work** library. All SAS data sets, both permanent and temporary, are data files that SAS creates and that only SAS can read. Before you begin your work at Orion Star, you first need to understand the components and structure of SAS data sets.

#### Examining SAS Data Sets

What is a SAS data set? A SAS data set is a specially structured data file that is displayed as a table with variables and observations. You might be more familiar with the terms table, columns, and rows, as these terms are commonly used across different types of databases. In SAS, a table is usually called a data set, a column is called a variable, and a row is called an observation.   
  
Let’s look at the following example. In the data set **work.newsalesemps**, the third column is the variable **Job\_Title**, and observation 3 contains the data for Kevin Lyon. A variable is a container that stores values. The value of the variable **Job\_Title**in observation 3 is Sales Rep. I.

**work.newsalesemps**

|  |  |  |  |
| --- | --- | --- | --- |
| **First\_Name** | **Last\_Name** | **Job\_Title** | **Salary** |
| Satyakam | Denny | Sales Rep. II | 26780 |
| Monica | Kletschkus | Sales Rep. IV | 30890 |
| Kevin | Lyon | Sales Rep. I | 26955 |
| Petrea | Soltau | Sales Rep. II | 27440 |

A SAS data set contains a descriptor portion and a data portion. The descriptor portion contains information about the attributes of the data set, or metadata. The metadata includes general properties such as the data set name, the number of observations, and the date and time that the data set was created, as well as variable properties such as name, type, and length. You can browse the descriptor portion of your SAS data sets using PROC CONTENTS.   
  
The data portion of a SAS data set contains the data values, stored in variables. Remember that the variable names are part of the descriptor portion, not the data portion. Data values are either character or numeric. For example, **First\_Name**, **Last\_Name** and **Job\_Title** have character values, and **Salary** has numeric values. You can use PROC PRINT to display the data portion of your SAS data sets. You’ll learn more about the data portion of SAS data sets in a bit.

#### Viewing the Descriptor Portion of a Data Set

In this demonstration, you run a PROC CONTENTS step to display the descriptor portion of the **sales** data set.

1. Copy and paste the following code into the editor. The **sales** data set is in the permanent orion library, so you must use a two-level data set name, **orion.sales**.

proc contents data=orion.sales;

run;

1. Submit the code and view the results. The PROC CONTENTS output displays the descriptor portion of the data set in three tables. The first table shows general information about the data set, such as the data set name, and the date and time the data set was created.   
     
   Take a look at the other information in this table. Can you determine how many observations are in this data set? There are 165 observations in **orion.sales**.     
     
   The second table displays operating environment information, the physical location of the file, and other data set information. The third table is an alphabetic list of variables in the data set and their attributes.

#### Viewing the Data Portion of a Data Set

In this demonstration, you run a PROC PRINT step to display the data portion of the data set **orion.sales**

1. Copy and paste the following code into the editor. Remember that the data portion contains the data values.

proc print data=orion.sales;

run;

1. Submit the code and check the log. A note in the log confirms that SAS read 165 observations from the data set.
2. View the report. By default, PROC PRINT displays all variables and observations, using the variable names as column headings. The Obs column is displayed to identify each observation, similar to a row number.   
     
   In the PROC CONTENTS output, SAS displayed a table with variable attributes, including the variable type: character or numeric. You can determine a variable’s type by looking at your PROC PRINT output; SAS automatically displays character variables left-aligned, and numeric variables right-aligned.

#### Understanding Missing Values

Now consider this: What happens if you have missing values in your data set? In a SAS data set, a value must exist for every variable and observation. If a data value is unknown for a particular observation, a missing value is recorded in the SAS data set. Missing values are valid values in a SAS data set. A variable’s type determines how SAS displays missing values for a variable.   
  
For character variables such as **Job\_Title**, a blank represents a missing value. For numeric variables such as **Salary**, SAS uses a period, by default, to represent a missing value. 

**work.newsalesemps**

|  |  |  |  |
| --- | --- | --- | --- |
| **First\_Name** | **Last\_Name** | **Job\_Title** | **Salary** |
| Satyakam | Denny | Sales Rep. II | 26780 |
| Monica | Kletschkus | Sales Rep. IV | . |
| Kevin | Lyon |  | 26955 |
| Petrea | Soltau | Sales Rep. II | 27440 |

You can alter this default with the MISSING= SAS system option, which specifies a character to print for missing numeric variable values.

|  |
| --- |
| **MISSING=**'character' |

#### Exploring SAS Variable Attributes

When you write SAS programs, it's important to understand the attributes of the variables that you use. Using the following PROC CONTENTS output of the data set **orion.sales**, let's take a closer look at the two variable attributes type and length.

| **Alphabetic List of Variables and Attributes** | | | | |
| --- | --- | --- | --- | --- |
| **#** | **Variable** | **Type** | **Len** | **Format** |
| **8** | Birth\_Date | Num | 8 |  |
| **7** | Country | Char | 2 |  |
| **1** | Employee\_ID | Num | 8 | 12. |
| **2** | First\_Name | Char | 12 |  |
| **4** | Gender | Char | 1 |  |
| **9** | Hire\_Date | Num | 8 |  |
| **6** | Job\_Title | Char | 25 |  |
| **3** | Last\_Name | Char | 18 |  |
| **5** | Salary | Num | 8 |  |

As you know, a variable's type is either character or numeric. Character variables can store any values, such as letters, numbers, special characters, and blanks.

|  |
| --- |
| **Character Values** |
| Monica |
| 120101 |
| 3Top Sports |
| Auditing & Wages |

Now let's look at some examples of valid numeric values. Numeric variables can store only numeric values, which can include the digits 0 through 9, a minus sign, a single decimal point, and E for scientific notation.

|  |
| --- |
| **Numeric Values** |
| 26780 |
| -30 |
| -29.92 |
| 3.1E6 |

A variable's length indicates the number of bytes used to store it. The length is related to the variable's type. Character values are stored with a length of 1 to 32,767 bytes. One byte equals one character. In **orion.sales**, the **First\_Name** variable has a length of 12 characters and uses 12 bytes of storage.   
  
Numeric variables have 8 bytes of storage by default, no matter how many digits they contain. When stored in floating point or binary representation, 8 bytes of storage provide space for 16 or 17 significant digits. SAS variables can have additional attributes, such as formats, as well as some that are not shown in this PROC CONTENTS output: informat and label. You will learn more about the additional attributes in a later lesson.

#### Naming SAS Variables and Data Sets

You must follow the SAS naming conventions when naming variables and data sets. SAS variable names can be 1 to 32 characters long. The name must start with a letter or underscore and can continue with any combination of numbers, letters, or underscores. SAS variable names can be uppercase, lowercase, or mixed case.

|  |
| --- |
| **Valid SAS Names** |
| Job\_Title |
| \_quantity2012\_ |
| Customer\_FirstName |

SAS creates each variable name in the same case that you first specify it, and that is the way it appears in reports. After a variable has been created, you can refer to it in any case in your code without affecting the way that it is stored. You apply the same naming conventions to SAS data set names.

proc print data=orion.sales;

var salary;

run;

**Partial output**

|  |  |
| --- | --- |
| **Obs** | **Salary** |
| **1** | 108255 |
| **2** | 87975 |
| **3** | 26600 |
| **4** | 27475 |
| **5** | 26190 |
| **6** | 26480 |

Click the **Information** button in the course interface to learn about using special characters in variable names.

## \

## Which PROC CONTENTS step prints only general information about a SAS library and a listing of the members of the library?

proc contents data=orion.\_all\_ nods;

run;

# Practice: Browsing a Library

## Task

In this practice, you use PROC CONTENTS to view the contents of the **orion** library.

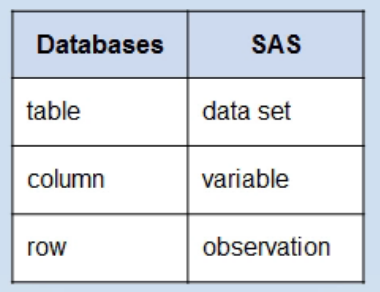
**Reminder**: Make sure you've defined the **orion** library.

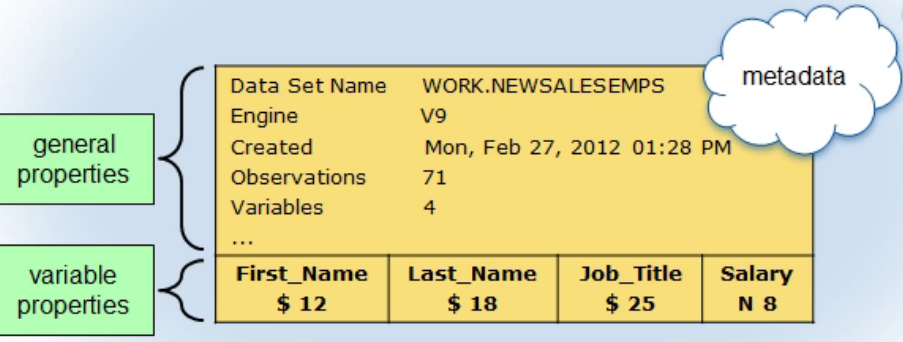
1. Write and submit a PROC CONTENTS step to display all of the SAS files in the **orion** library.

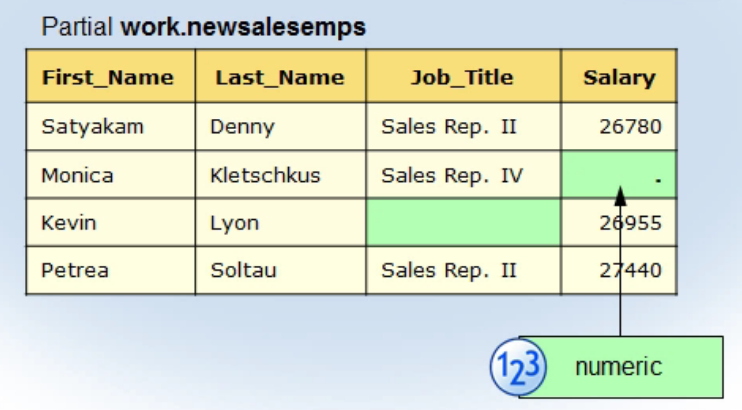
proc contents data=orion.\_all\_;

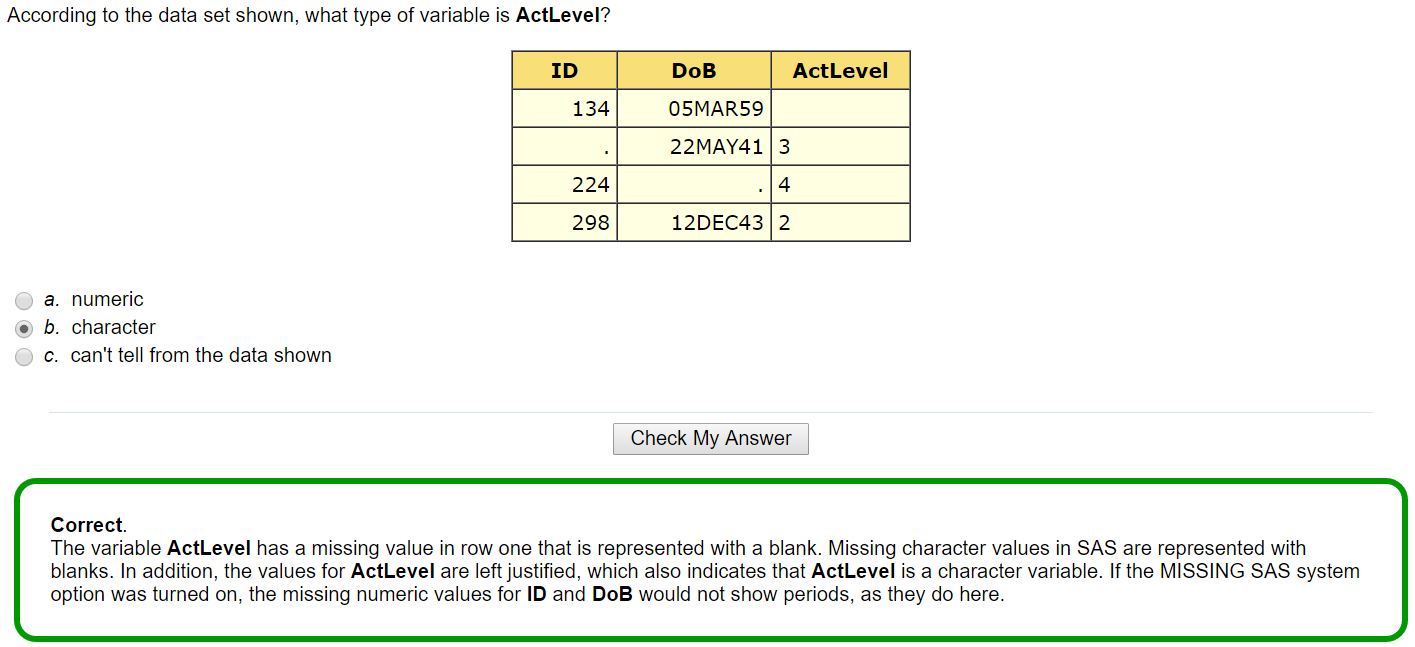
run;

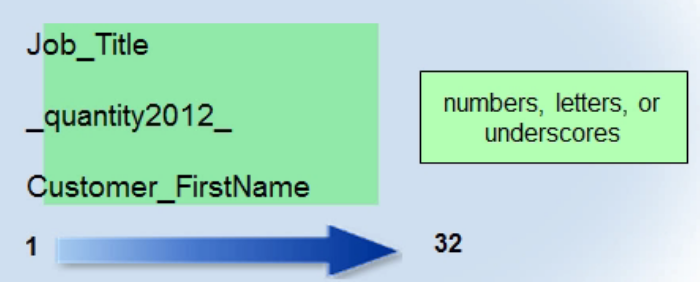
1. How many data sets are in the **orion** library?   
     
   The library members table indicates that the **orion** library contains 37 data sets.
2. Examine the PROC CONTENTS results for the data set **orion.charities**. How many observations and variables are in this data set?  
     
   The data set **orion.charities** contains 12 observations and 3 variables.











Add a PROC PRINT step to display all observations, all variables, and the Obs column for the data set **work.donations**. Submit the code to create this report.  
  
proc print data=work.donations;

run;

The data set contains 124 observations and 6 variables.

## Topic Summaries

### Accessing SAS Libraries

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](javascript:%20return%20false;). A libref is a shortcut to the physical location of the SAS files.

All SAS data sets have a [two-level name](javascript:%20return%20false;) 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](javascript:%20return%20false;). 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;), 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.

### Examining SAS Data Sets

[SAS data sets](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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.

## 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;

# **Lesson 4: Producing Detail Reports**

## Lesson Overview

#### Introduction

Sometimes, you might want to display all of the information from a SAS data set in a report. But other times, you might want to display very specific information from the data set. In this lesson, you learn to enhance the way that information is arranged and formatted in your reports. By adding statements to your PROC PRINT steps, you can select variables to print, subset observations, sort and group observations, specify titles and footnotes, and assign labels to variables.

#### Objectives

* select variables to print by using the VAR statement
* calculate column totals by using the SUM statement
* subset observations by using the WHERE statement
* sort observations by using the SORT procedure
* group observations in reports by using the BY statement
* specify titles and footnotes for your reports
* assign temporary labels to variables by using the LABEL statement

## Subsetting Report Data

#### Business Scenario

Orion Star management wants you to create a report that displays only the names and salaries of sales employees. They also want to see a salary total for these employees. You'll need to subset the data in your report to show the required variables, and create a sum of all the salaries.

#### Selecting Variables

By default, a PROC PRINT step displays all observations and variables in a data set, and the variables appear in the order in which they occur in the data set. You can use the VAR statement to modify the default behavior and display only the variables you want.

|  |
| --- |
| **VAR**variable(s)**;** |

In a VAR statement, you list the variables to include in the report. In this scenario, you want to print the last name and first name of each sales employee, as well as their salaries, so you list the variables in that order, as shown below.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

run;

#### Generating Column Totals

Now you need to generate a salary total for all sales employees and display it in your report. The SUM statement calculates and displays report totals for the requested numeric variable, which in this case is **Salary**. The general form of the SUM statement is shown below.

|  |
| --- |
| **SUM**variable(s)**;** |

This is the code that produces the column totals that will appear at the end of the report.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

sum Salary;

run;

#### Subsetting Your Report

In this demonstration, you subset your report and summarize the values of a variable.

1. Copy and paste the following code into the editor.

proc print data=orion.sales;

run;

1. Submit the code and view the results. Notice that there are nine variables.
2. In the editor, modify the code. You want to display **Last\_Name**, **First\_Name**, and **Salary**, and you want to summarize the values of **Salary**. Add a VAR statement and a SUM statement to the PROC PRINT step.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

sum Salary;

run;

1. Submit the code. Check the log to make sure the code ran successfully.
2. View the report. The report shows only the three variables that you requested, and the salary total is in the last row of the report.

#### Business Scenario

The management team now wants a report that displays the names and salaries of the sales employees earning a salary that is less than $25,500. To create this report, you'll need to subset the observations.

#### Subsetting Observations Using the WHERE Statement

You can use the WHERE statement in a PROC PRINT step to subset observations in a report.

|  |
| --- |
| **WHERE**where-expression**;** |

When you use a WHERE statement, your output contains only the observations that meet the conditions specified in the where-expression. The WHERE expression defines a condition for selecting observations, and can be any valid SAS expression.   
  
This WHERE statement defines the condition for your task: it selects observations where the value of **Salary** is less than $25,500. Notice that we still have the VAR statement, because those are still the variables we want to see. We've removed the SUM statement here though, because we aren't interested in seeing a salary total anymore.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

where Salary<25500;

run;

Now let's explore the WHERE expressiona bit. An expression is a sequence of operands and operators that form a set of instructions. The operands can be constants or variables. A constant operand is a fixed value, such as 25500. Numeric constants do not use quotation marks or special characters.   
  
This example shows a WHERE statement with a character constant.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

where Gender='M';

run;

Character constants must be enclosed in quotation marks and are case sensitive. If the value of the variable **Gender** is equal to the uppercase M character constant, SAS will select that observation.   
  
A variable operand must be a variable from the input data set. **Orion.sales** is the input data set in this program, so the variable **Salary** must exist in that data set. SAS selects observations where the value of **Salary** is less than 25500.   
  
You can use comparison, arithmetic, or logical operators in a WHERE expression. There are also several special WHERE operators that can only be used in WHERE statements. Let's look at each of these types of operators in more detail.

#### Comparison and Arthmetic Operators

The comparison operators, which are below, compare a variable with a constant or another variable. You can use either the symbol or the mnemonic in your code.

|  |  |  |
| --- | --- | --- |
| **Symbol(s)** | **Mnemonic** | **Definition** |
| = | EQ | equal to |
| ^=  ¬=  ~= | NE | not equal to |
| > | GT | greater than |
| < | LT | less than |
| >= | GE | greater than or equal to |
| <= | LE | less than or equal to |
|  | IN | equal to one of a list |

Look over the following examples of comparison operators used in WHERE statements.

where Gender='M';

where Gender eq 'M';

where Salary ne .;

where Salary>50000;

where Salary lt 50000;

where Salary<=60000;

where Country in ('Au','US');

The IN operator selects observations if they are equal to one of a list, and the value list in the IN operator must be enclosed in parentheses and separated by either commas or blanks. In the last example, SAS selects observations where the value of **Country** is equal to AU or US. Remember that character values must be enclosed in quotation marks.   
  
The arithmetic operators indicate that an arithmetic calculation is performed. 

|  |  |
| --- | --- |
| **Symbol** | **Definition** |
| \*\* | exponentiation |
| \* | multiplication |
| / | division |
| + | addition |
| - | subtraction |

Here's an example of arithmetic operators used in a WHERE statement:

where Salary+Bonus<=10000;

#### Logical Operators

You can use logical operators in a WHERE statement to combine or modify expressions.

|  |
| --- |
| **WHERE**where-expression-1**AND | OR**where-expression-n**;** |

For example, suppose you want to further modify your report to show not only the employees with salaries that are less than $25,500, but also only the employees who are from Australia. You know that SAS won't execute two WHERE statements. So to create the report you need, you can use logical operators. You can use either the symbol or the mnemonic in your code. 

|  |  |  |
| --- | --- | --- |
| **Symbol(s)** | **Mnemonic** | **Definition** |
| & | AND | logical and |
| | | OR | logical or |
| ^ ¬ ~ | NOT | logical not |

Here, you can see the logical operators that are available and several examples of WHERE statements that use these operators. The logical operator AND finds observations that satisfy both conditions. The logical operator OR finds observations that satisfy one or both conditions. And the logical operator NOT modifies a condition by finding the complement to the specified criteria.

where Country ne 'AU' and Salary>=50000;

where Gender eq 'M' or Salary ge 50000;

where Country='AU' | Country='US';

where Country in ('AU' 'US');

where Country not in ('AU', 'US');

#### Selecting Observations

In this demonstration, you select specific observations to display in a report.

1. Copy and paste the following code into the editor.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

run;

1. Add a WHERE statement to select only the employees who are from the country Australia and who have a **Salary** value that is less than $25,500. Notice that you use the logical operator AND to combine the two expressions.

proc print data=orion.sales;

var Last\_Name First\_Name Salary;

where Country='AU' and Salary<25500;

run;;

1. Add the variable **Country** to the VAR statement so that it's included in the report.

proc print data=orion.sales;

var Last\_Name First\_Name Salary **Country**;

where Country='AU' and Salary<25500;

run;

1. Submit the code and check the log. The log shows that SAS processed two observations from **orion.sales** due to the WHERE statement. You might recall that the previous report contained 165 observations.
2. View the report. Only the employees whose **Salary** and **Country** values met the WHERE expression conditions are displayed. But take a look at the Obs column. SAS displays the original observation numbers.
3. To suppress the Obs column, modify the PROC PRINT statement by adding the NOOBS option.

proc print data=orion.sales noobs;

var Last\_Name First\_Name Salary Country;

where Country='AU' and Salary<25500;

run;

1. Submit the code and verify that the report does not have the Obs column.

#### Business Scenario

You need to create a report that lists only the Australian sales representatives with Rep anywhere in their job titles, so you need to subset by **Country** and **Job\_Title**. Think about it. You can add another condition using the AND operator, but how do you specify this condition? Let's find out.

#### The CONTAINS Operator

You can use the CONTAINS special WHERE operator with the AND operator. The CONTAINS operator selects observations that include the specified substring.

|  |  |  |
| --- | --- | --- |
| **Symbol(s)** | **Mnemonic** | **Definition** |
| ? | CONTAINS | includes a substring |

The position of the substring within the variable's value does not matter. For example, the WHERE statements shown here will select observations in which the value for**Job\_Title** is Sales Rep I, Sales Rep II, and so on.

where Country='AU' and

Job\_Title **contains** 'Rep';

where Country='AU' and

Job\_Title **?** 'Rep';

The CONTAINS operator is case sensitive, so you must specify the substring in the exact case that you want it to match.

#### Using the CONTAINS Operator

In this demonstration, you use the CONTAINS operator to select observations for your report.

1. Copy and paste the following program into the editor. The WHERE statement uses the special operator CONTAINS to specify the conditions for the report.

proc print data=orion.sales noobs;

var Last\_Name First\_Name Country

Job\_Title;

where Country='AU' and

Job\_Title contains 'Rep';

run;

1. Submit the code and view the log. The log shows that SAS read 61 observations from **orion.sales**.
2. View the report. You can see that all of the values for **Country** are AU and all of the **Job\_Title** values contain the Rep character string.

#### Special WHERE Operators

In addition to the general categories of operators that you've seen, there are several other special operators that can only be used in a WHERE statement. Let's look more closely at these special WHERE operators. 

|  |  |
| --- | --- |
| **Mnemonic** | **Definition** |
| BETWEEN-AND | an inclusive range |
| WHERE SAME AND | augment a where expression |
| IS NULL | a missing value |
| IS MISSING | a missing value |
| LIKE | matches a pattern |

The BETWEEN-AND operator selects observations in which the value of a variable falls within an inclusive range of values. For example, the first WHERE statement in the program below selects observations in which the value of **Salary** is between 50000 and 100000. You could write an equivalent statement without using the BETWEEN-AND operator, as shown in the second WHERE statement. You could use the BETWEEN-AND operator along with the NOT operator to select values outside of the specified range, as shown in the last WHERE statement.

where Salary between 50000 and 100000;

where 50000<=Salary<=100000;

where Salary not between 50000 and 100000;

Use the WHERE SAME AND operator to add more conditions to an existing WHERE expression later in the program without retyping the original conditions. The WHERE SAME AND condition augments the original condition. This means that SAS will read observations where **Country**=AU and **Gender**=F and **Salary**<25500.

proc print data=orion.sales;

var First\_Name Last\_Name Gender Salary Country;

where Country='AU' and Salary<25500;

where same and Gender='F';

run;

The IS NULL and IS MISSING operators select observations in which the value of a variable is missing. These operators can be used for both character and numeric variables. For example, the WHERE statements shown below both select observations in which the value of **Employee\_ID** is missing.

where Employee\_ID **is null**;

where Employee\_ID **is missing**;

Here's a question: can you think of another operator you could use to select observations with a missing value? You could use the equals operator, as long as you knew whether the variable was numeric or character.

where Salary=.;

where Last\_Name=' ';

The NOT logical operator can be added to select observations with nonmissing values.

where Employee\_ID is not null;

where Employee\_ID is not missing;

The LIKE operator selects observations by comparing character values to specified patterns. There are two special characters available for specifying a pattern: the percent sign specifies that any number of characters, including zero characters, can occupy that position. The underscore specifies that exactly one character must occupy that position. You can specify consecutive underscores. You can also specify a percent sign and an underscore in the same pattern. 

|  |  |
| --- | --- |
| **Symbol** | **Replaces** |
| % | any number of characters |
| - | one character |

For example, in the following program the first WHERE statement selects observations in which the value of **Name** ends in an uppercase N, which is preceded by any number of characters. The second WHERE statement selects observations in which the value of **Name** begins with an uppercase T, followed by a single character, followed by a lowercasem, followed by any number of characters. This statement can match both the name Tom and the name Tommy.

where Name like '%N';

where Name like 'T\_m%';

#### Business Scenario

The sales manager wants a report that includes only customers who are 21 years old. He also wants the **Customer\_ID** variable to print at the beginning of each row.

#### Using the ID Statement

You know that you can use a WHERE statement to subset the data correctly. To specify the variable to print at the beginning of the row instead of an observation number, you can use the ID statement.

|  |
| --- |
| **ID**variable(s)**;** |

The variable you specify replaces the Obs column. We'll specify **Customer\_ID**.

proc print data=orion.customer\_dim;

where Customer\_Age=21;

id Customer\_ID;

run;

#### Subsetting Observations and Replacing the Obs Column

In this demonstration, you subset observations by replacing the Obs column.

1. Copy and paste the following program into the editor.

proc print data=orion.customer\_dim;

run;

1. Submit the code and view the report. In the report, you can see that the data set **orion.customer\_dim**contains quite a few variables, and some of these values are quite lengthy.
2. In the editor, add a WHERE statement to select the observations for customers who are 21 years old. Also add a VAR statement to display only the variables**Customer\_ID**, **Customer\_Name**, **Customer\_Gender**, **Customer\_Country**, **Customer\_Group**, **Customer\_Age\_Group**, and **Customer\_Type**.

proc print data=orion.customer\_dim;

where Customer\_Age=21;

var Customer\_ID Customer\_Name

Customer\_Gender Customer\_Country

Customer\_Group Customer\_Age\_Group

Customer\_Type;

run;

1. Submit the code and check the log. Verify that SAS read 6 observations.
2. View the report and verify that the **Customer\_Age\_Group** has the value15-30 years in all observations.   
     
   Consider this: If the observations were to wrap to another line, the observation numbers would make it easy to find the continuation. But a better technique is to choose a variable that uniquely identifies observations.
3. In the editor, add the ID statement to replace the Obs column and identify observations based on the **Customer\_ID** variable. Remove **Customer\_ID** from the VAR statement because you don't want the variable to appear twice.

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;

1. Submit the code and check the log. The log still shows that SAS read 6 observations, and in the report, the **Customer\_ID** variable identifies the observations rather than the observation numbers.

## Sorting and Grouping Report Data

#### Business Scenario

The payroll manager wants a report that displays the observations from **orion.sales** in ascending order by the variable **Salary**. As you've probably noticed, PROC PRINT displays observations in the order in which they appear in your data set. But you can write a SAS program to sort the observations in the data set and then use PROC PRINT to display the sorted data set.

#### Using the SORT Procedure

To sort the observations in a data set, you use PROC SORT. Using PROC SORT, you can sort on one variable or multiple variables, sort on character or numeric variables, and sort in ascending or descending order. Here's how PROC SORT works. First, SAS rearranges the observations in the input data set. Then, SAS creates a data set that contains the rearranged observations either by replacing the original data set or by creating a new data set. 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.

#### PROC SORT

A basic PROC SORT step has three statements: a PROC SORT statement, a BY statement, and a RUN statement.

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

proc sort data=orion.sales

out=work.sales\_sort;

by Salary;

run;

As with a PROC PRINT step, you use the DATA= option to specify the input data set. We need to sort **orion.sales**, so it's our input data set. Next, we use the OUT= option in the PROC SORT step because we don't want to permanently sort**orion.sales**. We want to sort **orion.sales** and create a new data set, **work.sales\_sort**. Note that in the syntax box, the general form of the OUT= option is enclosed in angle brackets (<>). These brackets indicate this element is optional.

Every PROC SORT step must include a BY statement. The BY statement specifies one or more variables in the input data set whose values are used to sort the data. These are called BY variables. The BY statement also indicates whether you want to sort in ascending or descending order. By default, SAS sorts in ascending order and you don't have to specify anything additional in the BY statement. In the example above, PROC SORT will sort the observations by the values of **Salary** in ascending order.

#### Sorting a Data Set

In this demonstration, you sort a SAS data set.

1. Copy and paste the following program into the editor. SAS will display the **Salary** values from lowest to highest, that is, in ascending order.

proc sort data=orion.sales

out=work.sales\_sort;

by Salary;

run;

1. Submit the code and check the log. You can see that SAS read 165 observations from**orion.sales** and created **work.sales\_sort** successfully.   
     
   You haven't created the report that the payroll manager requested yet. You've only sorted the data. To create the report, you need to write a PROC PRINT step.
2. Copy and paste the following code into the editor.

proc print data=work.sales\_sort;

run;

1. Submit the code and view the report. The report shows that the data set is sorted by the value of the variable **Salary**. You can see that the values are in ascending order, with the highest salary in the last observation.

#### Business Scenario

Now that you're familiar with sorting, the payroll manager at Orion Star has asked you to create another report that displays sales employees grouped by **Country** and displays them in descending **Salary** order within **Country**. So within each country, you want the observations arranged by **Salary**, from highest to lowest.

#### Specifying Multiple BY Variables

For this scenario, you'll first sort the data set **orion. sales** to group the observations. In a PROC SORT step, remember that you can list multiple variables in the BY statement, separated by spaces. In the following step, SAS first arranges the data set by the values of the first BY variable, **Country**, in ascending order. SAS then arranges any observations that have the same value of the first BY variable by the values of the second BY variable, **Salary**.

proc sort data=orion.sales;

out=work.sales2;

by Country Salary;

run;

To sort on a variable in descending order, you must specify the DESCENDING keyword immediately before each variable that you want in descending order. SAS will sort the observations from the largest value to the smallest value.

proc sort data=orion.sales;

out=work.sales2;

by Country **descending** Salary;

run;

#### Sorting a Data Set by Multiple Variables

In this demonstration, you sort a data set by multiple variables.

1. Copy and paste the following code into the editor. Notice that you're creating the output data set **work.sales2**.

proc sort data=orion.sales

out=work.sales2;

by Country descending Salary;

run;

proc print data=work.sales2;

run;

1. Submit the code and check the log. The log shows that the code ran successfully. SAS read 165 observation from the data set **orion.sales**.
2. View the report. The first country listed is Australia, or AU. Can you tell whether the **Salary** variable has been sorted in descending order? Yes. Scroll down the report to see that the **Salary** values decrease. And starting with observation 64, you see the employees from the US, and the highest **Salary** value is listed first.  
     
   The payroll manager wanted the report grouped by **Country**. This report displays the AU employees first because you sorted **Country** in ascending order, but it doesn't actually group the observations by **Country**.

#### Specifying Report Groupings

You use a BY statement in PROC PRINT to display the sorted observations grouped by **Country**.

proc print data=work.sales2 noobs;

by Country;

run;

This BY statement specifies the variable to use to form BY groups. The variables in the BY statement are called BY variables. Think about this: when you specify variables in the BY statement, what do you need to verify in the input data set? The input data set must be sorted on the variables specified in the BY statement. The variables must also be sorted in the order specified, either ascending or descending.

#### Grouping Observations in Reports

In this demonstration, you create a report that displays observations grouped by the variable **Country**.

1. Copy and paste the following code into the editor.

proc sort data=orion.sales

out=work.sales2;

by Country descending Salary;

run;

proc print data=work.sales2;

run;

1. In the PROC PRINT step, add a BY statement that groups the data by **Country**.

proc print data=work.sales2;

by Country;

run;

1. Submit the code and examine the report. You can see that SAS grouped the report by **Country**. The first table in the report is for employees who are from Australia, and the second table is for employees who are from the US. Notice that the **Salary** values are still listed in descending order.

## Enhancing Reports

#### Business Scenario

Suppose you need to share one of your reports at an upcoming staff meeting. You know that the information in the report is accurate, but your report could use some improvements in the way it looks. You can enhance the report by adding titles and footnotes.

#### Assigning Titles and Footnotes

You'd like to add a title that's specific to your report, such asOrion Star Sales Staff Salary Report. At the bottom of the report, you want to display the footnote Confidential. Titles appear at the top and footnotes appear at the bottom of the output from each procedure, no matter how long the output is. If you don't specify a title, the default title is The SAS System.   
  
To add a title to your report, you use the TITLE statement, and to add a footnote, you use the FOOTNOTE statement. Aside from the keyword at the beginning, these two statements have the same syntax.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **TITLE**n 'text'**;** | | |  | | --- | | **FOOTNOTE**n 'text'**;** | |

Following the keyword, you specify a value for n: a number from 1 to 10 that indicates the line on which the title or footnote appears. In the title or footnote area, line 1 is the first line and line 10 is the last. If you don't specify a number, SAS assumes that you're referring to line 1.   
  
Then, within quotation marks, you specify the text that you want to appear in the title or footnote. You can use either a set of single quotation marks or a set of double quotation marks to enclose the text string.   
  
In the following example, the TITLE statement specifies a title for line 1, Orion Star Sales Staff. You can specify only one title or footnote in a single statement.

title1 'Orion Star Sales Staff';

footnote1 'Confidential';

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

run;

You'd like the title Salary Report to appear on the second title line, so you need to add another TITLE statement, TITLE2. The FOOTNOTE statement specifies Confidential.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

footnote1 'Confidential';

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

run;

TITLE and FOOTNOTE statements are global statements, so they can stand alone. Also, any titles or footnotes that you assign remain in effect until you change them, cancel them, or end your SAS session.

#### Displaying Titles and Footnotes in a Report

In this demonstration, you display titles and footnotes in a report.

1. Copy and paste the following code into the editor to add titles and a footnote to your report.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

footnote1 'Confidential';

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

run;

1. Submit the code and examine the report. You can see that the titles Orion Star Sales Staff and Salary Report have been added to the top, and the footnote Confidential to the bottom of the report.   
     
   Suppose you need to run another PROC PRINT step.
2. In the editor, copy and paste the following code into the editor, and then submit it.

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;

1. Examine the new report. This report shows the same titles and the same footnote as the previous report, but you don't want these titles. You want different titles to appear that are meaningful for this information.   
     
   Remember that titles and footnotes are global statements, and when you assign them, they remain in effect until you change them, cancel them, or end your SAS session.

#### Changing Titles and Footnotes

To change a previously defined title, you add another TITLE statement that has the same number as the one you want to replace, but with different text. In the following example, we start by defining three lines of titles.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

title3 'Human Resources';

When we add another TITLE1 statement, the new text replaces the current text for TITLE1.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

title3 'Human Resources';

title1 'Salary Report';

Redefining a title also cancels all higher-numbered titles, so this TITLE1 statement changes title 1 and cancels titles 2 and 3. You change footnotes the same way.   
  
To cancel all previously defined titles and footnotes, you can specify null TITLE and FOOTNOTE statements, which have no numbers and no text.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

title3 'Human Resources';

title1 'Salary Report';

title;

footnote;

It's a good practice to cancel all titles and footnotes at the end of your program so that no unexpected titles and footnotes appear on output that you generate later in your SAS session.

#### Changing and Canceling Titles and Footnotes

In this demonstration, you change the title for the second report and then cancel all titles and footnotes.

1. Copy and paste the following code into the editor to add a new title to the second report. This code adds a TITLE statement before the second PROC PRINT step. You want this titleto replace the previous TITLE1 statement, so you number it TITLE1.

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;

1. Submit the code and view the report. As you can see, the first report has the titles Orion Star Sales Staff and Salary Report. The second report now has its own title.
2. As part of good programming practice, add the null TITLE and FOOTNOTE statements to the bottom of the program to cancel all titles and footnotes for any output that you might generate later.

title;

footnote;

1. Submit these statements, and now your session is ready for the next task.

#### Business Scenario

So far, you haven't made any changes to the appearance of variable names in the body of your reports. In the reports that you've created, the variable names appear exactly as they are stored in the input data set. For your upcoming meeting, suppose you want your reports to display more descriptive text instead of the variable names. For example, you want to change the appearance of the variables **Employee\_ID**, **Last\_Name**, and **Salary** in your report.

#### Assigning Temporary Labels by Using the LABEL Statement

To display temporary labels in your report instead of variable names, you can use the LABEL statement in your PROC PRINT step. You use the keyword LABEL, followed by the variable name, an equal sign, and a descriptive label in quotation marks.

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

In this example, you want to display the variable **Employee\_ID** as Sales ID, the variable **Last\_Name** as Last Name with no underscore, and the variable **Salary** as Annual Salary.

title1 'Orion Star Sales Staff';

title2 'Salary Report';

footnote1 'Confidential';

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

label Employee\_ID='Sales ID'

Last\_Name='Last Name'

Salary='Annual Salary';

run;

title;

footnote;

A label can be up to 256 characters long. You can specify labels for multiple variables in one LABEL statement, or you can use a separate LABEL statement for each variable. 

title1 'Orion Star Sales Staff';

title2 'Salary Report';

footnote1 'Confidential';

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

label Employee\_ID='Sales ID';

label Last\_Name='Last Name';

label Salary='Annual Salary';

run;

title;

footnote;

Most SAS procedures display labels automatically, but PROC PRINT does not. You have to add the LABEL option to your PROC PRINT statement to tell SAS to display the labels. 

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';

label Last\_Name='Last Name';

label Salary='Annual Salary';

run;

title;

footnote;

#### Displaying Labels in a Report

In this demonstration, you create temporary labels and display the labels rather than the variable names in a report.

1. Copy and paste the following step into the editor.

proc print data=orion.sales;

var Employee\_ID Last\_Name Salary;

run;

1. Submit the step and view the variable names as they currently appear in the data set **orion.sales**.
2. In the editor, copy and paste the following code to replace the existing code. You can see the labels for the three variables. Remember that in order to print the labels, you must include the LABEL option in the PROC PRINT step.

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;

1. Submit this code and then check the log. Notice that SAS doesn't print any messages regarding whether labels were applied. But the code ran without errors.
2. View the report. As you can see, the temporary labels now appear instead of the variable names.

#### Using the SPLIT= Option

The SPLIT= option in PROC PRINT specifies a split character to control line breaks in column headings. This option is most useful if you are using text or listing output rather than HTML output. Here's the syntax.

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

After SPLIT=, you specify a split-character, which will indicate where to wrap the label in the report. In the following example, the split character is an asterisk.

proc print data=orion.sales split='\*';

var Employee\_ID Last\_Name Salary;

label Employee\_ID='Sales ID'

Last\_Name='Last Name'

Salary='Annual Salary';

Next, in the LABEL statement, you add the split character to the label text at the place where you want the label to wrap to the next line. Here, an asterisk appears between the two words of the label Annual Salary. In the output, these two words will appear on different lines.

proc print data=orion.sales split='\*';

var Employee\_ID Last\_Name Salary;

label Employee\_ID='Sales ID'

Last\_Name='Last Name'

Salary='Annual\*Salary';

Do you think the LABEL option is necessary in the PROC PRINT statement now? Actually, SAS knows to print the labels because you're using the SPLIT= option. The PRINT procedure uses labels only when the LABEL or SPLIT= option is specified.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

# Practice: Subsetting a Report

## Ask

In this practice, you add statements and options to a PROC PRINT step to modify the default report of [**orion.order\_fact**](javascript:%20openOther('eclibjr/order_fact93.htm');).

**Reminder**: Make sure you've defined the **orion** library.

1. Copy and paste this starter code into the editor. Submit the program and view the output. Observe that there are 617 observations. Observations might be displayed over two lines, depending on output settings.

proc print data=orion.order\_fact;

run;

1. Add a SUM statement to display the sum of **Total\_Retail\_Price**. Confirm that the sum value of **Total\_Retail\_Price**is $100,077.46.

proc print data=orion.order\_fact;

**sum Total\_Retail\_Price;**

run;

1. Add a WHERE statement to select only the observations with **Total\_Retail\_Price** equal to more than 500. Submit the program. Verify that 35 observations were displayed.
   * What do you notice about the Obs column?
   * Did the sum value of **Total\_Retail\_Price** change to reflect only the subset?

proc print data=orion.order\_fact;

**where Total\_Retail\_Price>500;**

sum Total\_Retail\_Price;

run;

* + What do you notice about the Obs column?

The numbers are not sequential. The original observation numbers are displayed.

* + Did the sum value of **Total\_Retail\_Price** change to reflect only the subset?

Yes

1. Add an option to suppress the Obs column. Submit the program. Verify that there are 35 observations in the results. How can you verify the number of observations in the results?

proc print data=orion.order\_fact **noobs**;

where Total\_Retail\_Price>500;

sum Total\_Retail\_Price;

run;

You can verify the number of observations by checking the log.

1. Add an ID statement to use **Customer\_ID** as the identifying variable. Submit the program. The results contain 35 observations. How did the output change?

proc print data=orion.order\_fact noobs;

where Total\_Retail\_Price>500;

**id Customer\_ID;**

sum Total\_Retail\_Price;

run;

**Customer\_ID** is the leftmost column and is displayed on each line for an observation.

1. Add a VAR statement to display **Customer\_ID**, **Order\_ID**, **Order\_Type**, **Quantity**, and **Total\_Retail\_Price**. Submit the program and view the results. What do you notice about **Customer\_ID**?

proc print data=orion.order\_fact noobs;

where Total\_Retail\_Price>500;

id Customer\_ID;

**var Customer\_ID Order\_ID Order\_Type**

**Quantity Total\_Retail\_Price;**

sum Total\_Retail\_Price;

run;

There are two **Customer\_ID** columns. The first column is the **ID** field, and a second one is included because**Customer\_ID** is listed in the VAR statement.

1. Remove the duplicate column by removing **Customer\_ID**from the VAR statement. Submit the program and compare your results to this [report](javascript:openOther('m415/m415_1_r_l1results.htm');).

proc print data=orion.order\_fact noobs;

where Total\_Retail\_Price>500;

id Customer\_ID;

**var Order\_ID Order\_Type Quantity**

**Total\_Retail\_Price;**

sum Total\_Retail\_Price;

run;

Confirm that the sum value of **Total\_Retail\_Price** is $32,696.60.

## 

## Group statement:

## 

## Task

In this practice, you sort the observations in [**orion.employee\_payroll**](javascript:%20openOther('eclibjr/employee_payroll93.htm');) by **Salary** in ascending order.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a PROC SORT step to sort **orion.employee\_payroll** by **Salary**, and place the sorted observations into a temporary data set named **work.sort\_salary**.

proc sort data=orion.employee\_payroll

out=work.sort\_salary;

by Salary;

run;

1. Add a PROC PRINT step to display the new data set. Submit your program to create this [report](javascript:openOther('m415/m415_2_j_l1results.htm');).

proc print data=work.sort\_salary;

run;

## Task

In this practice, you sort the observations in [**orion.employee\_payroll**](javascript:%20openOther('eclibjr/employee_payroll93.htm');) by two variables to create **work.sal\_sort**. You create a report that displays a subset of the sorted observations, grouped by a variable.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a PROC SORT step to sort **orion.employee\_payroll** by **Employee\_Gender**, and within gender by **Salary** in descending order. Place the sorted observations into a temporary data set named **work.sort\_sal**.
2. proc sort data=orion.employee\_payroll
3. out=work.sort\_sal;
4. by Employee\_Gender descending Salary;

run;

1. Add a PROC PRINT step to print a subset of the **work.sort\_sal** data set.
   * Group the report by **Employee\_Gender**, and include a total and subtotals for **Salary**.
   * Select only the observations for active employees (those without a value for **Employee\_Term\_Date**) who earn more than $65,000.
   * Display only **Employee\_ID**, **Salary**, and **Marital\_Status**.
   * Suppress the Obs column.
   * Submit the program to create this [report](javascript:openOther('m415/m415_2_j_l2results.htm');).

proc print data=work.sort\_sal noobs;

by Employee\_Gender;

sum Salary;

where Employee\_Term\_Date is missing

and Salary>65000;

var Employee\_ID Salary Marital\_Status;

run;

**Note**: Because **Employee\_Term\_Date** is a numeric variable, and missing numeric variable values are represented in SAS by a period, your WHERE statement could look like this:

where Employee\_Term\_Date=**.**

and Salary>65000;

1. What is the summarized **Salary** value for females?  
     
   The summarized **Salary** value for females is 605190.

## Task

In this practice, you sort the data set [**orion.orders**](javascript:%20openOther('eclibjr/orders93.htm');). You use SAS Help or the SAS product documentation online to research PROC SORT options.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a step to sort **orion.orders** by **Customer\_ID**. Place the sorted observations in a temporary data set.

proc sort data=orion.orders

out=work.custorders;

by Customer\_ID;

run;

1. Add a step to display the sorted data set. Submit the program. The resulting report should contain 490 observations.**Customer\_ID** is listed multiple times for customers who placed more than one order.

proc print data=work.custorders;

run;

1. In the SAS documentation, investigate an option that causes PROC SORT to retain only the first observation in each BY group. Add the appropriate option to the PROC SORT step to retain only the first observation in each BY group. Submit the program to create this [report](javascript:openOther('m415/m415_2_j_l3_step3results.htm');). The results contain 75 observations with no duplicate values for**Customer\_ID**.  
     
   Suggested search terms: PROC SORT or PROC SORT options. The NODUPKEY option deletes observations with duplicate BY values.

proc sort data=orion.orders

out=work.custorders **nodupkey**;

by Customer\_ID;

run;

proc print data=work.custorders;

run;

1. In the SAS documentation, explore the DUPOUT= option to write duplicate observations to a separate output data set. Add the DUPOUT= option, and then submit the program to create this [report](javascript:openOther('m415/m415_2_j_l3results.htm');) of the duplicates.  
     
   The DUPOUT= option specifies the output data set to which duplicate observations are written. The DUPOUT= option can be used only with the NODUPKEY option.

proc sort data=orion.orders

out=work.custorders nodupkey

**dupout=work.duplicates;**

by Customer\_ID;

run;

proc print data=work.duplicates;

run;

## 

## 

## 

**footnote1 'Orion Star';**

**footnote2 'Sales Employees';**

**footnote3 'Confidential';**

**proc print data=orion.sales;**

**run;**

**footnote2 'Non Sales Employees';**

**proc print data=orion.nonsales;**

**run;**

When you run the second PROC PRINT step, the FOOTNOTE2 statement replaces the previous footnote with the same number: Non Sales Employees replaces Sales Employees. It also cancels all footnotes with higher numbers, so FOOTNOTE3, Confidential, does not appear in the results. The resulting footnotes are Orion Star and Non Sales Employees.

## 

## 

## 

## Task

In this practice, you write a program to display a subset of the data set [**orion.employee\_addresses**](javascript:%20openOther('eclibjr/employee_addresses93.htm');).

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a program to display a subset of **orion.employee\_addresses** as shown in this [report](javascript:openOther('m415/m415_3_j_l3results.htm');). The program should sort the observations by **State**, **City**, and **Employee\_Name**, and then display the sorted observations grouped by**State**. The resulting report should contain 311 observations.

proc sort data=orion.employee\_addresses

out=work.address;

where Country='US';

by State City Employee\_Name;

run;

title "US Employees by State";

proc print data=work.address noobs split=' ';

var Employee\_ID Employee\_Name

City Postal\_Code;

label Employee\_ID='Employee ID'

Employee\_Name='Name'

Postal\_Code='Zip Code';

by State;

run;

title;

## Task

In this practice, you specify new labels to appear in a report, and then modify their appearance.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Copy and paste this starter code into the editor.

title 'Entry-level Sales Representatives';

footnote 'Job\_Title: Sales Rep. I';

proc print data=orion.sales noobs;

where Country='US' and

Job\_Title='Sales Rep. I';

var Employee\_ID First\_Name Last\_Name

Gender Salary;

run;

title;

footnote;

1. Modify the program to define and use the following labels:

|  |  |
| --- | --- |
| **Variable** | **Label** |
| Employee\_ID | Employee ID |
| First\_Name | First Name |
| Last\_Name | Last Name |
| Salary | Annual Salary |

title 'Entry-Level Sales Representatives';

footnote 'Job\_Title: Sales Rep. I';

proc print data=orion.sales noobs **label**;

where Country='US' and

Job\_Title='Sales Rep. I';

var Employee\_ID First\_Name Last\_Name

Gender Salary;

**label Employee\_ID="Employee ID"**

**First\_Name="First Name"**

**Last\_Name="Last Name"**

**Salary="Annual Salary";**

run;

title;

footnote;

1. Submit the program to create this [report](javascript:openOther('m415/m415_3_j_l2results.htm');).
2. Modify the program to use a blank space as the SPLIT= character to generate two-line column headings. Submit the modified program and verify that two-line column labels are displayed as in this [report](javascript:openOther('m415/m415_3_j_l2_step4results.htm');).

title 'Entry-Level Sales Representatives';

footnote 'Job\_Title: Sales Rep. I';

proc print data=orion.sales noobs **split=' '**;

where Country='US' and

Job\_Title='Sales Rep. I';

1. var Employee\_ID First\_Name Last\_Name

Gender Salary;

label Employee\_ID="Employee ID"

First\_Name="First Name"

Last\_Name="Last Name"

Salary="Annual Salary";

run;

title;

footnote;

## Topic Summaries

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

### Subsetting Report Data

You can use the [VAR statement](javascript:%20return%20false;)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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;), [logical](javascript:%20return%20false;), and [special WHERE operators](javascript:%20return%20false;).

|  |
| --- |
| **WHERE**where-expression**;** |

You can use the [ID statement](javascript:%20return%20false;) 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)**;** |

### Sorting and Grouping Report Data

The [SORT procedure](javascript:%20return%20false;) 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](javascript:%20return%20false;), some variables can be in ascending and others in descending order.

You can also use a [BY statement](javascript:%20return%20false;) 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;** |

### Enhancing Reports

You can enhance a report by adding titles, footnotes, and column labels. Use the global [TITLE statement](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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=](javascript:%20return%20false;) 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'**;** |

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 Report**s

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;

# **Lesson 5: Formatting Data Values**

## Lesson Overview

#### Introduction

In your SAS reports, formats control the way data values are displayed. You might want to make some data values more understandable or descriptive. In this lesson, you learn to enhance the way that variable values are displayed and formatted in your reports by associating existing SAS formats with variables. You also learn how to create and apply your own custom formats.

#### Objectives

* describe SAS formats
* apply SAS formats with the FORMAT statement
* create user-defined formats using the FORMAT procedure
* apply user-defined formats using the FORMAT statement
* use formats to recode data values
* use formats to collapse or aggregate data

## Using SAS Formats

#### Business Scenario

Suppose you want to enhance the appearance of variable values in your reports. For example, by default, your reports display values as they are stored in the input data set. But those values are not always formatted in a way that's easy to understand. This PROC PRINT report shows hire dates for employees, but as unformatted numeric values, who knows what these mean!

| **Last\_Name** | **First\_Name** | **Country** | **Job\_Title** | **Salary** | **Hire\_Date** |
| --- | --- | --- | --- | --- | --- |
| Zhou | Tom | AU | Sales Manager | 108255 | 12205 |
| Dawes | Wilson | AU | Sales Manager | 87975 | 6575 |
| Elvish | Irenie | AU | Sales Rep. II | 26600 | 6575 |
| Ngan | Christina | AU | Sales Rep. II | 27475 | 8217 |
| Hotstone | Kimiko | AU | Sales Rep. I | 26190 | 10866 |
| Daymond | Lucian | AU | Sales Rep. I | 26480 | 8460 |
| Hofmeister | Fong | AU | Sales Rep. IV | 32040 | 8460 |

You need to create a report with more easily understood variable values. Displaying these SAS dates as calendar dates would improve the report. You could also add dollar signs and commas to the variable **Salary** to improve its appearance.

#### Using the FORMAT Statement

To control how values appear in your reports, you can specify temporary SAS formats by adding the FORMAT statement to your PROC PRINT step.

|  |
| --- |
| **FORMAT**variable(s) format***;*** |

proc print data=orion.sales noobs label;

where Country='AU' and

Job\_Title contains 'Rep';

label Job\_Title='Sales Title'

Hire\_Date='Date Hired';

format Salary dollar8.;

format Hire\_Date mmddyy10.;

var Last\_Name First\_Name Country Job\_Title

Salary Hire\_Date;

run;

You use the keyword FORMAT, followed by the variable and the SAS format that you want to apply to the variable. You can use a separate FORMAT statement for each variable, or you can format several variables using either the same format or different formats in a single FORMAT statement.

format Salary dollar8. Hire\_Date mmddyy10.;

#### SAS Formats

What exactly is a format? A format is an instruction that tells SAS how to display data values. For example, you can display a numeric value with commas and a dollar sign.   
  
There are many existing SAS formats that you can use, and they all use the same form, as shown here.

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

The dollar sign indicates a character format and precedes the name of the SAS format. Then you specify the total format width, including decimal places and special characters. The period is required syntax. Finally, you can specify the number of decimal places in numeric formats.   
  
Let's look at a few general examples. This list shows the general form for several common SAS formats. Take a moment to read through these definitions.

|  |  |
| --- | --- |
| **Format** | **Definition** |
| w.d | is a standard character format. It is used to write character data in a field wpositions wide. |
| w.d | is a standard numeric format. It is used to write numeric data in a field wpositions wide with d decimal places. The value of w includes the decimal point and decimal places. |
| COMMAw.d | writes numeric values with a comma separating every three digits and a period separating the decimal fraction. |
| DOLLARw.d | writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction. |
| COMMAXw.d | is a non-US numeric format. It writes numeric values with a period separating every three digits and a comma separating the decimal fraction. |
| EUROXw.d | is like COMMAXw.d, but it adds a leading euro symbol (€). |

#### Examples of SAS Formats

This table shows several specific SAS formats and their effect on stored values.

|  |  |  |
| --- | --- | --- |
| **Format** | **Stored Value** | **Displayed Value** |
| $4. | Programming | Prog |
| 12. | 27134.5864 | 27135 |
| 12.2 | 27134.5864 | 27134.59 |
| COMMA12.2 | 27134.5864 | 27,134.59 |
| DOLLAR12.2 | 27134.5864 | $27,134.59 |
| COMMAX12.2 | 27134.5864 | 27.134,59 |
| EUROX12.2 | 27134.5864 | €27.134,59 |

Character values are truncated if they do not fit in the specified width. In this first example, you can see that the stored value, Programming, is displayed as Prog because the assigned format only has a width of 4. If you do not specify a width that is large enough to accommodate a numeric value, the displayed value is automatically adjusted to fit into the width.   
  
Let's look at some of the numeric formats. The 12. format, which is the same as 12.0, doesn't specify the number of decimal places, so none are displayed. SAS rounds the displayed value to the nearest integer. With the 12.2 format, the value is displayed in a field 12 positions wide with 2 decimal places. The decimal value is rounded to the nearest hundredth. The COMMA12.2 format inserts a comma between the three digits. The DOLLAR12.2 format inserts a dollar sign in the displayed value. 12 is the total width of the displayed value, including the dollar sign, commas, decimal point, and decimal places. The COMMAX12.2 format inserts a period between the three digits, and a comma separates the decimal fraction. Lastly, in the EUROX12.2 format, a euro symbol is inserted in the displayed value.   
  
This table shows additional examples.

|  |  |  |
| --- | --- | --- |
| **Format** | **Stored Value** | **Displayed Value** |
| DOLLAR12.2 | 27134.5864 | $27,134.59 |
| DOLLAR9.2 | 27134.5864 | $27134.59 |
| DOLLAR8.2 | 27134.5864 | 27134.59 |
| DOLLAR5.2 | 27134.5864 | 27135 |
| DOLLAR4.2 | 27134.5864 | 27E3 |

In the first row of this table, 12 is wide enough to display the value, including the dollar sign, comma, decimal point, and decimal places. You can see that the displayed value is 10 positions wide. What if you specify a width of 9, as in the second row? The comma is not displayed. With a width of 8 in the third row, the dollar sign is also dropped. The fourth row shows that a width of 5 results in the omission of decimal places in the displayed value. And the last row shows that when a width of 4 is specified, the value is rounded to 27,000 and displayed in E-notation. But remember, the format only affects the displayed value. The stored value is not affected by a format.

#### Working with SAS Date Values and SAS Date Formats

SAS date values are a special category of numeric values. SAS stores date values as the number of days between January 1, 1960, and a specific date.

For example, SAS stores January 1, 1960, as 0, and January 2, 1960, as 1, and so on. Notice that dates earlier than January 1, 1960, have negative SAS date values. When your report displays a SAS date like 12205, it's difficult to know what date this really is! To make the dates in your report recognizable and meaningful, you must apply a SAS date format to the SAS date values.

This table lists several common SAS date formats and shows how each one affects a stored SAS date value.

|  |  |  |
| --- | --- | --- |
| **Format** | **Stored Value** | **Displayed Value** |
| MMDDYY6. | 0 | 010160 |
| MMDDYY8. | 0 | 01/01/60 |
| MMDDYY10. | 0 | 01/01/1960 |
| DDMMYY6. | 365 | 311260 |
| DDMMYY8. | 365 | 31/12/60 |
| DDMMYY10. | 365 | 31/12/1960 |

Let's look at the first three formats, the MMDDYY formats. These formats display values as a numeric month, day, and year. The number at the end of the format is the width of the displayed field. It determines whether a forward slash separator will be used, and if the year displays as two digits or four digits. A width of 6 does not provide enough room for a separator and only allows for a two-digit year. A width of 8 allows for a separator and a two-digit year. With a width of 10, the value displays with a separator and a four-digit year. So the format MMDDYY10. displays a date value with a width of 10. It includes slashes to separate the month, day, and year values, and displays the year as a four-digit value.

The DDMMYY formats are similar to the MMDDYY formats except that they display values as a numeric day, month, and year. Click the **Information** button in the course to see more SAS date format examples, as well as the tables of formats you saw earlier.

#### Applying Temporary Formats

In this demonstration, you apply SAS formats to variables to make them easier to read.

1. Copy and paste the following program into the editor to view the variables in the **orion.sales** data set before you apply 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;

1. Submit the program and view the results. As you can clearly see, the**Hire\_Date** values are difficult to decipher. Remember that these date values are shown as numeric values. Now let's format the variables for your new report.
2. In the editor, add a FORMAT statement that assigns the MMDDYY10. format to the variable **Hire\_Date** and assigns the DOLLAR8. format to the variable **Salary**.

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;

1. Submit this code and look at the results. Notice that **Salary** has commas inserted to separate each group of three digits and is displayed in a column that is 10 positions wide. **Hire\_Date** is displayed as a two-digit month, a slash, a two-digit day, a slash, and a four-digit year. This column is also 10 positions wide. This report is much easier to understand now.

## Creating and Applying User-Defined Formats

#### Business Scenario

The Sales Department requested a special report in which the full country names will appear instead of the country code. SAS provides many formats but cannot provide every format possibly needed. Fortunately, you can create and apply your own formats to your reports.

#### Creating and Applying User-Defined Formats

To create and apply your own formats, you must use two PROC steps. First, you use PROC FORMAT to create the user-defined format. Then you use a FORMAT statement in the PROC PRINT step to apply the format to a specific variable. 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.

#### PROC FORMAT

The general form of the PROC FORMAT statement is shown below:

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

In a basic PROC FORMAT step, the PROC FORMAT statement consists only of the keywords PROC FORMAT. The VALUE statement defines the format. First, you specify a format name. Then you specify a value or range of values, and lastly you specify the formatted value, or how you want the value to be displayed.   
  
These examples illustrate the rules for constructing a format name.

|  |  |
| --- | --- |
| **Type** | **Format-name** |
| character | $CTRYFMT |
| character | $\_ST3FMT\_ |
| numeric | ORIONSTAR\_SALRANGE2\_FMT\_ |
| numeric | \_SALRANGE |

A format name can have a maximum of 32 characters. The name of a format that applies to character values must begin with a dollar sign ($), followed by a letter or underscore. The name of a format that applies to numeric values must begin with a letter or underscore. A format name cannot end in a number. All remaining characters can be letters, underscores, or numbers. A user-defined format name cannot be the name of a SAS format. Also, notice that a format name does not end with a period in the VALUE statement. Later, when you refer to the format in a FORMAT statement, you'll specify the period.

#### Using the VALUE Statement

Now let's see how you specify the way you want the data values to appear in your output. You have already seen the general form of the PROC FORMAT step. You use the VALUE statement in the PROC FORMAT step to specify one or more expressions that we'll call value-range sets. Each value-rangeset has three parts: the value or range, which specifies one or more values to be formatted, an equal sign, and then the formatted value that you want SAS to display instead of the stored value or values.

You can specify the value-or-range in several ways: as an individual value, as a range of values, or as a list of values. In each set of examples shown below, the top example has character values and the bottom example has numeric values.

|  |  |  |
| --- | --- | --- |
| **value-or-range** | **=** | **formatted-value** |
| 'AU' or 1 | = | 'Australia' |
| 'B'-'D' or 0-50000 | = | 'Tier 1' |
| 'U', 'V' or 1,2,3 | = | 'Below 49.9' |

When you specify the value-or-range, you must enclose character values in quotation marks. The character values that you specify must match the case of the variable's values. You do not enclose numeric values in quotation marks.   
  
In a range, a hyphen (-) separates the values that define the endpoints of the range. When you specify a range of character values, be careful not to enclose the entire range in quotation marks. If you do this, SAS assumes that all of the characters, including the hyphen, are part of a single character value.   
  
In a list, commas separate the individual values. The formatted value is always a character string, no matter whether the format applies to character values or numeric values. A character string can consist of any type of character. Usually, each formatted value is enclosed in quotation marks, as shown above. However, SAS does not require the quotation marks for a formatted value. Formatted values can be up to 32,767 characters in length.   
  
When you specify a value-or-range, you can also use the keyword OTHER to specify values that do not match any other value-or-range. If you do not include the keyword OTHER, then SAS applies the format only to values that match the value-range sets that you specify. If SAS encounters a value that you did not anticipate, SAS cannot apply the format but instead displays that value as it's stored in the data set.

|  |  |  |
| --- | --- | --- |
| **value-or-range** | **=** | **formatted-value** |
| OTHER | = | 'Australia' |

#### Using PROC FORMAT

Here's the PROC FORMAT step for our scenario:

*proc format;*

*value $ctryfmt 'AU'='Australia'*

*'US'='United States'*

*other='Miscoded';*

*run;*

Remember that the report needs to display full country names rather than the country codes. The country values are character, so you need a character format. Its name must begin with a dollar sign. Here we chose the name $CTRYFMT. We can call this the $ country format. Notice that there is no period at the end of the format name.   
  
Now look at our value ranges. When you apply this format to a variable later, the output will display full country names instead of country codes. Notice that the last value-range set specifies the keyword OTHER to include all values that do not match any other value or range. In this example, the output will label any values other than the two country codes as miscoded.   
  
You can only define one format in the VALUE statement. However, you can define multiple formats in a single PROC FORMAT step by adding multiple VALUE statements.

*proc format;*

*value $ctryfmt 'AU'='Australia'*

*'US'='United States'*

*other='Miscoded';*

*value $sports*

*'FB'='Football'*

*'BK'='Basketball'*

*'BS'='Baseball';*

*run;*

#### Using PROC PRINT to Apply a User-Defined Format

Now that you know how to create your own formats, let's look at the second PROC step that you can use to apply the format. You use the FORMAT statement in a PROC PRINT step to apply your formats to variables. The following FORMAT statement applies the user-defined format $CTRYFMT to the variable **Country**.

*proc print data=orion.sales label;*

*format Salary dollar10.*

*Birth\_Date Hire\_Date monyy7.*

*Country $ctryfmt.;*

*run;*

When you refer to a user-defined format in the FORMAT statement, notice that you must specify a period after the format name, the same way that you do for a SAS format name. However, remember that you do not have to include a period after a user-defined format name when you create it.   
  
We could have added another FORMAT statement, but both SAS and user-defined formats can be applied in a single FORMAT statement.

#### Specifying a User-Defined Format for a Character Variable

In this demonstration, you create a user-defined format and assign the format to a character variable.

1. Copy and paste the following program into the editor. The PROC FORMAT step creates the format $CTRYFMT. The PROC PRINT step assigns the format to the variable**Country**.

*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;*

1. Submit the program and check the log. You can see that SAS created the format because there's a note that states that the $CTRYFMT has been output.
2. View the report. The **Country** values are displayed as Australia and as United States. None of the **Country** values are displayed as Miscoded.

#### Business Scenario

Now let's look at a numeric example. The Orion Star HR manager has asked for a report showing employee salaries organized into three user-defined groups, or tiers. You need to display the tiers in the report instead of the dollar amount.

#### Specifying Ranges of Values

Let's create a format that applies to numeric values. Suppose we know that the **Salary** values in **orion.sales** are between 20,000 and 250,000. Let's say that Tier1 includes salaries from 20,000 to 49,999, Tier2 includes salaries from 50,000 to 99,999, and Tier3 includes salaries from 100,000to 250,000. This PROC FORMAT step defines the TIERS format.

*proc format;*

*value tiers 20000-49999='Tier1'*

*50000-99999='Tier2'*

*100000-250000='Tier3';*

*run;*

In this VALUE statement, each value-range set specifies an inclusive range of values. An inclusive range includes the first value and the last value. Think about this. After you create the TIERS format, how will the **Salary** value 99,999.87 appear in your report? Oops! The value falls outside of the ranges that are specified here: between Tier2 and Tier3. The ranges defined here assume that the values of **Salary** are stored as whole numbers.   
  
To create a set of ranges that have no gaps between them, you can add the less-than (<) symbol to exclude one or both numbers in individual ranges. First, you make sure that the last number in the range is the same as the number at the beginning of the next range. The number at the end of the first range becomes 50,000. The number at the end of the middle range becomes 100,000.   
  
Let's look at all possible ways that you can use the less-than symbol in a range. We'll use the middle range in the TIERS format as an example. Note that the values in bold will be excluded in the range.

|  |  |  |
| --- | --- | --- |
| **First Value** | **Symbol(s)** | **Last Value** |
| 50000 | - | 100000 |
| **50000** | <- | 100000 |
| 50000 | -< | **100000** |
| **50000** | <-< | **100000** |

To exclude the first value in a range, you put the less-than symbol after the first value. To exclude the last value in a range, you put the less-than symbol before the last value. And, to exclude both the first and last values, you put a less-than symbol in both places.   
  
Now, let's add the less-than symbol to the ranges in this PROC FORMAT statement so that the ranges have no gaps between them.

*proc format;*

*value tiers 20000-<50000='Tier1'*

*50000-<100000='Tier2'*

*100000-250000='Tier3';*

*run;*

In the VALUE statement, the ranges are now defined by using the less-than symbol. Here's a question. If SAS applies the TIERS format to the value 100,000, how do you think it will be displayed? 100,000 is the end value of the second range. A less-than symbol appears in front of this number, so it is excluded in the second range. This value will appear as Tier3 in the report.

#### Defining a Continuous Range

So far, we've assumed that the values of **Salary** are between 20,000and 250,000. However, what if we don't know the highest and lowest values? To specify the lowest possible value of a variable, you can use the keyword LOW. And to specify the highest possible value, you can use the keyword HIGH.

*proc format;*

*value tiers low-<50000='Tier1'*

*50000-<100000='Tier2'*

*100000-high='Tier3';*

*run;*

The LOW keyword can be used to define ranges that apply to character values as well as to numeric values. It's important to know 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.   
  
Consider this. If you apply the TIERS format to a variable, what does SAS display in the report for a missing value? The TIERS format applies to numeric values, so a missing value will appear as a period in the report.

#### Specifying a User-Defined Format for a Numeric Variable

In this demonstration, you create a user-defined format and assign the format to a numeric variable.

1. Copy and paste the following program into the editor. The VALUE statement uses the LOW keyword up to and excluding 50000 for Tier1. For Tier2, you're including 50000up to and including 100000. And for Tier3, you're excluding 100000 and including every value above it.

*proc format;*

*value tiers low-<50000='Tier 1'*

*50000-100000='Tier 2'*

*100000<-high='Tier 3';*

*run;*

1. To apply the TIERS format to the variable **Salary** in the report, copy and paste the following PROC PRINT step into the editor. Recall that when you refer to a user-defined format in the FORMAT statement, you must specify a period after the format name, the same way that you do for a SAS format name.

*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;*

1. Submit these steps and then check the log to ensure SAS creates the format. A note states that the format TIERS has been output.
2. View the report. The **Salary** values have been replaced with the appropriate tier values.

## Task

In this practice, you create two formats and specify values and value ranges. Then you apply your user-defined formats to variables in the data set [**orion.nonsales**](javascript:%20openOther('eclibjr/nonsales93.htm');) and print the results.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a step to create a character format named $GENDER that displays gender codes as follows:

|  |
| --- |
|  |
| **Gender** | **Formatted Value** |
| F | Female |
| M | Male |
| any other value | Invalid code |

proc format;

value $gender

'F'='Female'

'M'='Male'

other='Invalid code';

run;

1. Write a step to create a numeric format named SALRANGE that displays salary ranges as follows:

|  |  |
| --- | --- |
| **Salary Ranges** | **Formatted Values** |
| At least 20,000 but less than 100,000 | Below $100,000 |
| At least 100,000 and up to 500,000 | $100,000 or more |
| missing | Missing salary |
| any other value | Invalid salary |

proc format;

value salrange 20000-<100000=

'Below $100,000'

100000-500000='$100,000 or more'

.='Missing salary'

other='Invalid salary';

run;

1. Add a PROC PRINT step to create this [report](javascript:openOther('m416/m416_2_j_l2results.htm');) and apply your two user-defined formats to the variables **Gender** and**Salary**, respectively, in the data set **orion.nonsales**. Submit the program and confirm that observation 1 has a**Salary** value of $100,000 or more and a **Gender** value of Male.

proc format;

value $gender

'F'='Female'

'M'='Male'

other='Invalid code';

value salrange .='Missing salary'

20000-<100000='Below $100,000'

100000-500000='$100,000 or more'

other='Invalid salary';

run;

**title1 'Salary and Gender Values';**

**title2 'for Non-Sales Employees';**

**proc print data=orion.nonsales;**

**var Employee\_ID Job\_Title Salary Gender;**

**format Salary salrange. Gender $gender.;**

**run;**

**title;**

## Topic Summaries

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

### Using SAS Formats

A [format](javascript:%20return%20false;) 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](javascript:%20return%20false;) to the SAS date values.

### Creating and Applying User-Defined Formats

You can create your own [user-defined formats](javascript:%20return%20false;). 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;), use a less-than symbol after the low end of the range or before the high end.

The [LOW and HIGH](javascript:%20return%20false;) 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.

## A user-defined format that applies to numeric values cannot start with $.

## 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;

# Lesson 6: Reading SAS Data Sets

## Lesson Overview

#### Introduction

One common SAS programming task is to create a SAS data set. For example, information on Orion Star sales employees might reside in several different input sources, such as a SAS data set, a raw data file, or even a Microsoft Excel worksheet. You can create a SAS data set from any of these types of data. In this lesson, you use a DATA step to create a SAS data set from an existing SAS data set. You also learn to create new variables, select variables and observations, and add permanent attributes to variables.

#### Objectives

* use a DATA step to create a SAS data set from an existing SAS data set
* subset observations by using the WHERE statement
* create a new variable by using the assignment statement
* subset variables by using the DROP and KEEP statements
* describe the compilation and execution phases of the DATA step
* store labels and formats in the descriptor portion of a SAS data set

## Reading a SAS Data Set

#### Business Scenario

**Orion.sales** is a SAS data set that contains information about Orion Star sales employees from Australia and from the United States. Suppose you want to use the data in**orion.sales** to create a new SAS data set that contains a subset of this data. For example, you want the new data set, **work.subset1**, to contain only the Australian sales representatives with the substring Rep in their job title.

#### Using the DATA Step

To create a new SAS data set from an existing SAS data set, you use a DATA step. The following DATA step contains a DATA statement, a SET statement, and a RUN statement.

|  |
| --- |
| **PROC**output-SAS-data-set**;**         **SET**input-SAS-data-set**;** **RUN;** |

You begin the DATA step with the DATA statement, which provides the name of the SAS data set that you're creating. The data set can be temporary or permanent. In the following example, you're creating the temporary SAS data set **subset1** in the **work** library.

*data work.subset1;*

*set orion.sales;*

*run;*

The SET statement names **orion.sales** as the existing SAS data set that you want to read in as input data. Can you tell whether this is a permanent or temporary data set? Yes, you can. The data is in the permanent library, **orion**.   
  
By default, a SET statement reads all observations and all variables from the input data set sequentially. You use the WHERE  statement to subset the input data set by selecting only the observations that meet a particular condition. The following WHERE statement selects only those observations where the variable **Country** has a value of AU and where the value of **Job\_Title**contains the substring Rep.

*data work.subset1;*

*set orion.sales;*

*where Country='AU' and*

*Job\_Title contains 'Rep';*

*run;*

If either of these expressions is false, SAS will not process the observation, and therefore, the observation won't be included in the output data set.   
  
Here's a question. Can you think of a way to write this WHERE statement using different operators or symbols? You could write it like this.

*data work.subset1;*

*where Country eq 'AU' and*

*Job\_Title like '%Rep%';*

*run;*

These statements will return the same results.

#### Subsetting Observations in the DATA Step

In this demonstration, you subset observations in the DATA step.

1. Copy and paste the following PROC PRINT step into the editor to examine the data set **orion.sales**.

*proc print data=orion.sales;*

*run;*

1. Submit the step and then check the log. As you can see, SAS read 165 observations from **orion.sales**.
2. View the report. You can see that there are nine variables. Notice that the variable **Job\_Title** includes mostly titles that contain the substring Rep already, but a few of them don't. The **Country** variable includes both AU and US values.
3. Copy and paste the following program into the editor. The DATA step uses a WHERE statement to subset the observations and create the new data set, **work.subset1**. The PROC PRINT step prints the new data set.

*data work.subset1;*

*set orion.sales;*

*where Country='AU' and*

*Job\_Title contains 'Rep';*

*run;*

*proc print data=work.subset1;*

*run;*

1. Submit the code and check the log. The log shows that SAS read 61 observations from **orion.sales**. In the report, notice that all of the observations have AU as the value for **Country** and have Rep somewhere in their value for **Job\_Title**.

#### Business Scenario

Suppose that the management staff wants to give a 10% bonus to each Australian sales representative hired before January 1, 2000. You'll use the SAS data set **orion.sales** to create a new data set named **work.subset1**. In addition to subsetting for AU and the substring Rep, you'll subset the data based on the employee hire date. You'll also calculate a 10% bonus for these employees based on their salary.

#### Using a SAS Date Constant

In this scenario, you need to subset the data based on the variable**Hire\_Date**, which contains a SAS date value. How do you think you can compare a SAS date value to a calendar date? You can use a SAS date constant. A SAS date constant is a date written in the form of a two-digit day, followed by a three-letter month abbreviation, and then a two or four-digit year, enclosed in quotes and followed by the letter D.

|  |
| --- |
| **'**ddmmm<yy>yy' D |

SAS will automatically convert a date constant to a SAS date value. The following table shows some examples of SAS date constants.

|  |
| --- |
| **Examples** |
| '01JAN2000'D |
| '31Dec11'd |
| '1jan04'd |
| '06Nov2000'D |

You can use a date constant in any SAS expression, including a WHERE expression.

In the following example, we want the employees whose **Hire\_Date**value is before January 1, 2000, so we use the less-than symbol to indicate this.

*data work.subset1;*

*set orion.sales;*

*where Country='AU' and*

*Job\_Title contains 'Rep' and*

*Hire\_Date<'01jan2000'd;*

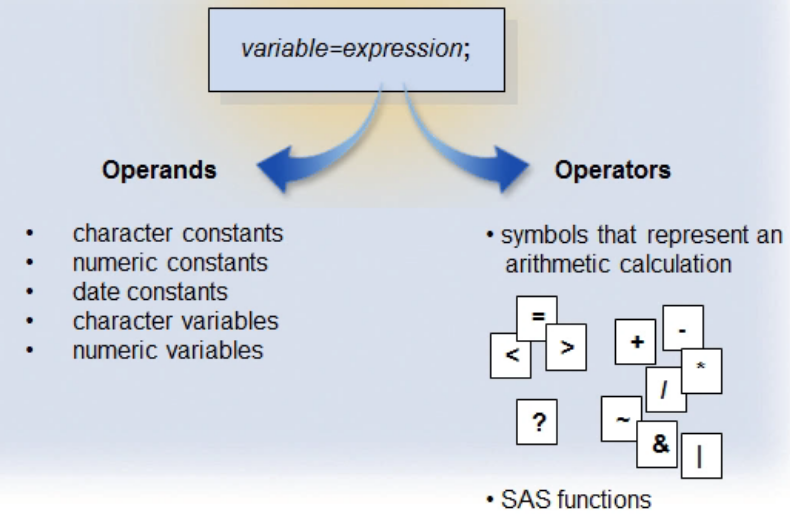
*run;*

You enclose a SAS date constant in quotation marks. Notice that this program already contains the WHERE expression for the Australian employees whose **Job\_Title** value includes the substring Rep.

#### Using the Assignment Statement

Our next task in the scenario is to calculate a 10% bonus for these employees. We're going to create the new variable **Bonus** to store the bonus value. To modify existing values or to create new variables, you can use an assignment statement in a DATA step. The assignment statement evaluates an expression and assigns the resulting value to a new or existing variable.

|  |
| --- |
| variable**=**expression**;** |

Notice that the assignment statement is one of the few SAS statements that doesn't begin with a keyword. Variable names an existing or new variable, and expression is a set of instructions that produces a value.   
  
As in the WHERE statement, an expression is a sequence of operands and operators. Operands are character constants, numeric constants, date constants, character variables, or numeric variables. Operators are either symbols that represent an arithmetic calculation or they are SAS functions.   
  
Here are some examples of assignment statements.

|  |  |
| --- | --- |
| **Example** | **Type** |
| Salary=26960; | numeric constant |
| Gender='F'; | character constant |
| Hire\_Date='21JAN1995'd; | date constant |
| Bonus=Salary\*.10; | arithmetic expression |
| BonusMonth=month(Hire\_Date); | SAS functions |

In the first row of the table, we're assigning the numeric constant 26960 to **Salary**. The next row shows how to assign a character constant. Remember that it needs to be in quotes, either single or double. Next you can see how to use a date constant. In the fourth row, we're using an arithmetic expression. We're multiplying **Salary** by .10 to calculate the 10% bonus and assigning the resulting value to the new variable **Bonus**. In the last row, you can see how to use a SAS function. This function extracts the month from the**Hire\_Date** value.

You should be mindful when using arithmetic operators in an assignment statement. When you use more than one arithmetic operator in an expression, SAS performs operations based on priority, as is the case normally in math equations. You can use parentheses to clarify or alter the order of operations. Also, if any operand in the expression has a missing value in the observation, the result is a missing value.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Definition** | **Priority** |
| \*\* | exponentiation | I |
| \* | multiplication | II |
| / | division | II |
| + | addition | III |
| - | subtraction | III |

#### Subsetting Observations and Creating a New Variable

In this demonstration, you subset observations and create a new variable in the DATA step.

1. Copy and paste the following program into the editor. The DATA step subsets the data set **orion.sales** by the Australian sales representatives based on their hire date. Using an assignment statement, the program creates the new variable **Bonus**. The PROC PRINT step displays the report and includes a FORMAT statement to format**Hire\_Date** values with the DATE9. format.

*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;*

1. Submit the program and check the log. The log shows that SAS read 29 observations from **orion.sales**, and 29 observations and 10 variables were output to**work.subset1**. Originally, **orion.sales** contained nine variables, so it looks like SAS created the new variable.
2. View the report. The new variable, **Bonus**, is displayed. In the first observation, notice that the calculation was performed correctly: 26600 multiplied by .10equals 2660. Also, notice that the variable **Hire\_Date** does not include any dates after January 1, 2000.

## Task

In this practice, you write a basic DATA step to create the temporary data set **work.youngadult**. You then modify the DATA step to meet several criteria.

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Copy and paste this starter code into the editor. Submit the program.

*proc print data=orion.customer\_dim;*

*run;*

1. What is the name of the variable that contains gender values?  
     
   **Customer\_Gender**
2. What are the two observed **Gender** values?  
     
   M or F
3. Replace the PROC PRINT step with a DATA step that creates a new data set named **work.youngadult**. Use the data set **orion.customer\_dim** as input. Include a WHERE statement to select only female customers. Submit the program and check the log to confirm that **work.youngadult** was created with 30 observations and 11 variables.

data work.youngadult;

set orion.customer\_dim;

where Customer\_Gender='F';

run;

An alternate WHERE statement is the following:

where Customer\_Gender eq 'F';

1. Modify the DATA step to select female customers whose age is between 18 and 36. Submit the program and check the log to confirm that **work.youngadult**was created with 15 observations and 11 variables.

data work.youngadult;

set orion.customer\_dim;

where Customer\_Gender='F' and

**Customer\_Age between 18 and 36**;

run;

1. Modify the DATA step to select 18- to 36-year-old female customers who have the word Gold in their**Customer\_Group**values. Submit the program and check the log to confirm that **work.youngadult** was created with 5 observations and 11 variables.

data work.youngadult;

set orion.customer\_dim;

where Customer\_Gender='F' and

**Customer\_Age between 18 and 36 and**

**Customer\_Group contains 'Gold'**;

run;

1. Add an assignment statement to the DATA step to create a new variable, **Discount**, and assign it a value of .25. Submit the program and check the log to confirm that **work.youngadult** was created with 5 observations and 12 variables.

data work.youngadult;

set orion.customer\_dim;

where Customer\_Gender='F' and

Customer\_Age between 18 and 36 and

Customer\_Group contains 'Gold';

**Discount=.25;**

run;

1. Print the new data set. Use an ID statement in the PROC PRINT step to display **Customer\_ID**instead of the Obs column. The [results](javascript:openOther('m417/m417_1_h_l1results.htm');) should contain 5 observations.

proc print data=work.youngadult;

var Customer\_Name Customer\_Age

Customer\_Gender Customer\_Group Discount;

**id Customer\_ID;**

run;

## Customizing a SAS Data Set

#### Business Scenario

You've seen how to modify a DATA step so that the output data set contains a subset of the observations in the input data set. Suppose that now you want to modify your DATA step further. You want all of the Australian sales reps to receive a bonus, regardless of hire date, and you only want to include some of the variables from **orion.sales** in your new data set. That is, you want your output data set to contain a subset of the variables in the input data set, as well as a subset of the observations in the input data set.

#### Using the DROP and KEEP Statements

As you've seen, the SET statement reads all of the variables from the input data set and writes them to the output data set. You can exclude variables from your output data set by using a DROP statement or a KEEP statement in a DATA step.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **DROP**variable-list**;** | | |  | | --- | | **KEEP**variable-list**;** | |

You use the DROP statement to specify the variables to exclude from the output data set. The DROP statement begins with the keyword DROP, followed by a space-separated list of the variables that you want to drop from the output data set.  
 *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;*

You use the KEEP statement to specify a list of variables to include in the output data set.

*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;*

How can you decide which statement to use? You might want to use the KEEP statement instead of the DROP statement if the number of variables to keep is significantly smaller than the number to drop. Also, if you use a KEEP statement, you must include every variable to be written, including any new variables. One more note: the DROP and KEEP statements have no effect on the input data set.

#### Subsetting Variables in a DATA Step: DROP and KEEP

In this demonstration, you subset variables in a DATA step using the DROP and KEEP statements.

1. Copy and paste the following program into the editor. This DATA step contains the DROP statement to exclude the variables **Employee\_ID**, **Gender**, **Country**, and**Birth\_Date** from the output data set **work.subset1**.

*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;*

1. Submit the code and then check the log. The log shows that SAS read 61 observations from **orion.sales**. The new data set, **work.subset1** contains six variables. You added the variable **Bonus** and dropped four others.
2. View the report. You can see that the report displays the six variables, including the new variable **Bonus**.
3. Copy and paste the following program into the editor. This DATA step contains the KEEP statement and lists the variables to include in the output data set.

*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;*

1. Submit the program and check the log. Again, SAS read 61 observations, and **work.subset1** contains six variables. The report is exactly the same. You can use the DROP or KEEP statements to produce the same output data set.

#### How SAS Processes the DATA Step

Let's investigate how SAS processes the DATA step. SAS processes the DATA step in two phases: the compilation phase and the execution phase. During the compilation phase, SAS scans each DATA step statement for syntax errors. It converts the program to machine code if no syntax errors are found. SAS also creates the program data vector to hold the current observation.  
  
When the compilation phase is complete, SAS creates the descriptor portion of the new data set. Remember, the descriptor portion contains information such as the data set name and the names of all the data set's variables.

#### Compilation Phase

Let's walk through the compilation phase of our previous program.

*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;*

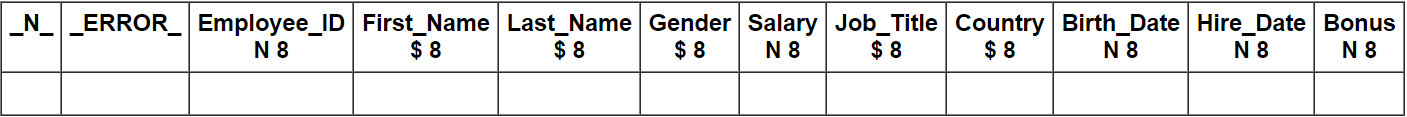
During the compilation phase, SAS creates the program data vector, or PDV. The PDV is an area of memory where SAS builds one observation. The PDV contains two automatic variables that can be used for processing, but that are not written to the data set as part of an observation. **\_N\_** is the iteration number of the DATA step, and **\_ERROR\_** signals the occurrence of an error that is caused by the data during execution. The default value of **\_ERROR\_** is 0, which means there is no error. When one or more errors occur, the value is set to 1.

**PDV**

|  |  |  |  |
| --- | --- | --- | --- |
| **\_N\_** | **\_ERROR\_** |  |  |
|  |  |  |  |

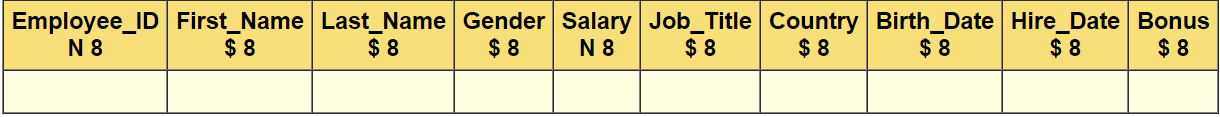
SAS scans each statement in the DATA step, looking for syntax errors such as missing or misspelled keywords, invalid variable names, missing or invalid punctuation, or invalid options.   
  
In our example code above, SAS scans the DATA step. When SAS compiles the SET statement, a slot is added to the PDV for each variable in the input data set: **Employee\_ID**,**First\_Name**, **Last\_Name**, **Gender**, **Salary**, **Job\_Title**, **Country**, **Birth\_Date**, and **Hire\_Date**. The descriptor portion of the input SAS data set, **orion.sales**, supplies the variable names, as well as attributes such as type and length. Then SAS adds the new variable **Bonus** to the PDV based on the assignment statement.

**PDV**



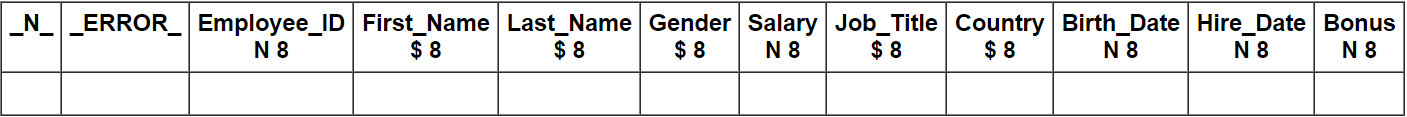
SAS determines that **Bonus** is a numeric variable because the expression on the right is a numeric constant. SAS then flags the variables to be dropped from the output. In this case, the variables **Employee\_ID**, **Gender**, **Country**, and **Birth\_Date** are marked to be dropped.   
  
At the bottom of the DATA step, the compilation phase is complete, and the descriptor portion of the new SAS data set **work.subset1** is created.

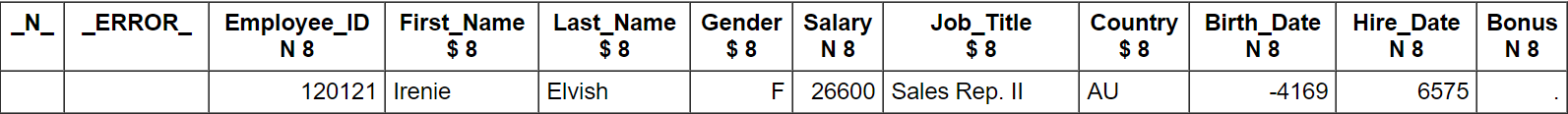
**Descriptor portion of work.subset1**



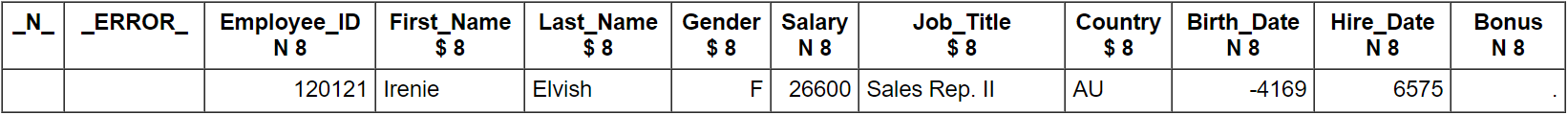
Execution Phase

If the DATA step compiles successfully, then the execution phase begins. During the execution phase, the DATA step reads and processes the observations from the input data set, and creates observations in the data portion of the output data set. By default, the DATA step executes once for each observation in the input data set.   
At the start of the execution phase, SAS initializes the PDV to missing.   
  
**PDV**

  
Remember that missing character values are displayed as blanks, and missing numeric values are displayed as a period. In our example program, when the SET statement executes, SAS reads the first observation from **orion.sales** into the PDV, providing a value for each variable.  
  
**PDV**

  
The value of **Bonus** is missing because **Bonus** doesn't come from the input data set. It's a new variable being created in this DATA step. When SAS executes the assignment statement, it assigns a value to **Bonus**. At the bottom of the DATA step, SAS uses the values in the PDV to write the first observation to the new SAS data set. SAS doesn't write the variables in the DROP statement to **work.subset1**.   
  
**work.subset1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **First\_Name** | **Last\_Name** | **Salary** | **Job\_Title** | **Hire\_Date** | **Bonus** |
| Irenie | Elvish | 26600 | Sales Rep. II | 6575 | 2660.00 |

Then control returns to the top of the DATA step for the next iteration. This is referred to as implicit output and implicit return. SAS retains the values of variables that were read from the input data set in the PDV. These values will be overwritten when the next observation is read into the PDV. SAS reinitializes the value of new variable, **Bonus**, to missing.   
  
As the SET statement executes on the second iteration of the DATA step, SAS reads the second observation into the PDV. It overwrites previous values in the PDV. SAS calculates **Bonus** for this observation, and then uses the values in the PDV to write the second observation to the new data set.

**work.subset1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **First\_Name** | **Last\_Name** | **Salary** | **Job\_Title** | **Hire\_Date** | **Bonus** |
| Irenie | Elvish | 26600 | Sales Rep. II | 6575 | 2660.00 |
| Christina | Ngan | 27475 | Sales Rep. II | 8217 | 2747.50 |

Then control returns to the top of the DATA step for the next iteration. This process continues until all of the observations are read.

#### Business Scenario

Suppose that you want to create a new SAS data set that contains only the Australian employees whose **Bonus** is at least $3000. In this situation, you want to subset observations based on the variable **Country** and the variable **Bonus**. **Country** is a part of the **orion.sales** input data set. Recall that the variable **Bonus** does not exist in our input data set. We created it with an assignment statement in the DATA step.   
  
You could consider using a WHERE statement, but the WHERE statement selects observations when they are read from the input data set to the PDV. Let's find out what happens if you use a variable that does not exist in the input data set in a WHERE statement.

#### The Subsetting IF Statement

To subset observations based on the value of a variable you create, you can use the subsetting IF statement. The syntax for the subsetting IF statement is the keyword IF and anexpression that you want to evaluate.

|  |
| --- |
| **IF**expression**;** |

Remember that an expression is a sequence of operands and operators that form a set of instructions. You can specify multiple expressions in a subsetting IF statement.

*if Salary>5000;*

*if Hire\_Date='15APR2008'd;*

*if Country not in ('GB', 'FR', 'NL');*

*if Country='US' and Salary>75000;*

Although IF expressions are similar to WHERE expressions, you cannot use special WHERE operators in IF expressions.

The subsetting IF statement causes the DATA step to continue processing only those observations that meet the condition of the expression that you specify. That is, if the expression is true for the observation, SAS continues to execute statements in the DATA step and writes the current observation to the output data set. The resulting SAS data set contains a subset of the original SAS data set. If the expression is false, no further statements are processed for that observation, the current observation is not written to the data set, and the remaining program statements in the DATA step are not executed. SAS immediately returns to the beginning of the DATA step for the next iteration.

#### Selecting Observations by Using the Subsetting IF Statement

In this demonstration, you select observations from a data set using the subsetting IF statement in your DATA step.

1. Copy and paste the following program into the editor. You use **orion.sales** to create the temporary data set **auemps**. The WHERE statement selects only the observations where the value of **Country** is equal to AU. The assignment statement creates the variable **Bonus** by calculating 10% of the employees' salaries. The subsetting IF statement specifies that you only want the observations where the value of **Bonus** is equal to or greater than 3000. The PROC PRINT step creates the report.

data work.auemps;

set orion.sales;

where Country='AU';

Bonus=Salary\*.10;

if Bonus>=3000;

run;

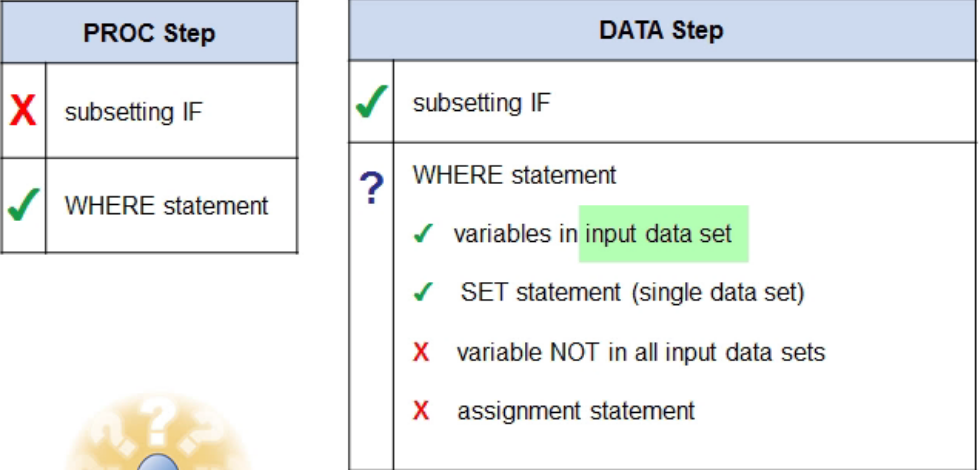
proc print data=work.auemps;

run;

1. Submit the code and check the log. You can see that of the 165 observations in **orion.sales**, 63 were read into the PDV for processing, and only 12 were written to**work.auemps**.
2. View the report. Notice that only AU values for **Country** and only **Bonus** values equal to or greater than 3000 are displayed.

#### Choosing a Statement for Subsetting Observations

It can be confusing to determine which statement to use—the WHERE statement or the subsetting IF statement. Let's discuss how you decide. If you're subsetting observations in a PROC step, you must use a WHERE statement. You cannot use a subsetting IF statement. That's easy.   
  
If you're subsetting observations in a DATA step, you can always use a subsetting IF statement. That's easy, too. The tricky part is knowing when you can use a WHERE statement in the DATA step. You only have to remember one rule: when you use a WHERE statement in the DATA step, the WHERE expression must reference only variables from the input data set.   
  
If you're trying to subset based on a variable that SAS is reading from a single data set using the SET statement, you can use a WHERE statement. If the variable is not in all data sets, you can't use a WHERE statement.   
  
Why can't you use a WHERE statement based on a variable that's created with an assignment statement? A variable that's created using an assignment statement doesn't exist in the input data set. Remember, the WHERE statement subsets data as SAS reads the data into the PDV.



## Adding Permanent Attributes

#### Business Scenario

Now that you know how to subset observations and variables to create a customized SAS data set, suppose that you want to create **work.subset1**so that it includes permanent labels and formats. In other words, you want to permanently associates labels and formats to the variables and store them in the descriptor portion of the data set.

#### Assigning Permanent Labels in a DATA Step

You use a LABEL statement in a PROC step to assign descriptive labels to variables in your reports.

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

These are temporary labels. When you use the LABEL statement in a DATA step, SAS permanently associates the labels to the variables. In the following DATA step, we're assigning the label **Sales Title**to **Job\_Title**, and the label **Date Hired** to **Hire\_Date**.

*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;*

SAS will add these labels to the descriptor portion of the data set. Remember that the descriptor portion of a SAS data set stores variable attributes including the name, type, and length of the variable.

#### Adding Permanent Labels to a SAS Data Set

In this demonstration, you add permanent labels to the descriptor portion of a SAS data set and then print the labels in a report.

1. Copy and paste the following program into the editor. The DATA step includes the LABEL statement, which specifies labels for the variables **Job\_Title** and **Hire\_Date**. The PROC CONTENTS creates the descriptor portion of **work.subset1**.

*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;*

1. Submit the program and check the log. The log shows that the program ran successfully.
2. Examine the results. The PROC CONTENTS report shows that the labels are now associated with the variables. Consider this: If you write a PROC PRINT step to display**work.subset1**, will these new labels appear in the report? No. The report will only include these descriptive labels if you add the LABEL option to the PROC PRINT step. So even though the labels are permanently associated with the variables, you have the choice of how to display the variables in your output.
3. Copy and paste the following PROC PRINT step, which includes the LABEL option, into the editor and submit it.

*proc print data=work.subset1 label;*

*run;*

1. View the results. The report shows that **Sales Title** and **Date Hired** have replaced their variable names as headings.

#### Using the FORMAT Statement in a DATA Step

As with the LABEL statement, you can use the FORMAT statement in a DATA step to permanently associate formats with variables.

|  |
| --- |
| **FORMAT**variable(s) format**;** |

The format information is also stored in the descriptor portion of the data set.   
  
In the data set **work.subset1**, you can apply SAS formats to the variables **Salary**, **Hire\_Date**, and **Bonus** to permanently format the values so that they are easier to understand.  
  
**Partial work.subset1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **First\_Name** | **Last\_Name** | **Salary** | **Job\_Title** | **Hire\_Date** | **Bonus** |
| Irenie | Elvish | 26600 | Sales Rep. II | 6575 | 2660.00 |
| Christina | Ngan | 27475 | Sales Rep. II | 8217 | 2747.50 |

#### Adding Permanent Formats to a SAS Data Set

In this demonstration, you add permanent formats to the descriptor portion of a SAS data set.

1. Copy and paste the following program into the editor. The FORMAT statement applies the format DOLLAR12. to both the **Salary** and **Bonus** variables, and the format DDMMYY10. to the **Hire\_Date** variable. Notice that you format the variable name and not the label name. The PROC CONTENTS creates the descriptor portion of**work.subset1**, and the PROC PRINT step prints the 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;*

1. Submit this program and then check the log. The log shows that SAS ran successfully.
2. Examine the results. PROC CONTENTS shows that the formats were associated with our variables. Notice that the labels are also still associated. In the report, the **Salary**and **Bonus** variable values now have dollar signs and commas, and the **Date Hired**values are much easier to read and understand.

## Task

In this practice, you write a DATA step to subset observations based on three conditions. You create the temporary data set **work.delays**from the data set [**orion.orders**](javascript:%20openOther('eclibjr/orders93.htm');).

**Reminder**: Make sure you've [defined the **orion** library](javascript:%20openOther('setup_reminder.htm');).

1. Write a DATA step to create a new data set named **work.delays**. Use the data set **orion.orders** as input. Submit the code to confirm that **work.delays** contains 490 observations and 6 variables.

data work.delays;

set orion.orders;

run;

1. Modify the DATA step to create a new variable, **Order\_Month**, and set it to the month of the**Order\_Date**. Hint: Use an assignment statement to extract the month from the **Order\_Date** value.

data work.delays;

set orion.orders;

**Order\_Month=month(Order\_Date);**

run;

1. Use a WHERE statement and a subsetting IF statement to select only the observations that meet all of the following conditions:
   * **Delivery\_Date** values that are more than four days beyond**Order\_Date**
   * **Employee\_ID** values that are equal to 99999999
   * **Order\_Month** values occurring in August

data work.delays;

set orion.orders;

**where Order\_Date+4<Delivery\_Date and**

**Employee\_ID=99999999;**

Order\_Month=month(Order\_Date);

**if Order\_Month=8;**

run;

1. The new data set should include only **Employee\_ID**, **Customer\_ID**, **Order\_Date**, **Delivery\_Date**, and**Order\_Month**.

data work.delays;

set orion.orders;

where Order\_Date+4<Delivery\_Date and

Employee\_ID=99999999;

Order\_Month=month(Order\_Date);

if Order\_Month=8;

1. **keep Employee\_ID Customer\_ID Order\_Date**

**Delivery\_Date Order\_Month;**

run;

1. Add permanent labels for **Order\_Date**, **Delivery\_Date**, and **Order\_Month** as shown in this [report](javascript:openOther('m417/m417_3_f_l2_step5results.htm');).

data work.delays;

set orion.orders;

where Order\_Date+4<Delivery\_Date

and Employee\_ID=99999999;

Order\_Month=month(Order\_Date);

if Order\_Month=8;

**label Order\_Date='Date Ordered'**

**Delivery\_Date='Date Delivered'**

**Order\_Month='Month Ordered';**

keep Employee\_ID Customer\_ID Order\_Date

Delivery\_Date Order\_Month;

run;

1. Add permanent formats to display **Order\_Date** and **Delivery\_Date**as MM/DD/YYYY. Add a PROC CONTENTS step and submit it to verify that the labels and formats were stored permanently.

data work.delays;

set orion.orders;

where Order\_Date+4<Delivery\_Date

and Employee\_ID=99999999;

Order\_Month=month(Order\_Date);

if Order\_Month=8;

label Order\_Date='Date Ordered'

Delivery\_Date='Date Delivered'

Order\_Month='Month Ordered';

**format Order\_Date Delivery\_Date mmddyy10.;**

keep Employee\_ID Customer\_ID Order\_Date

Delivery\_Date Order\_Month;

run;

**proc contents data=work.delays;**

**run;**

1. Write a PROC PRINT step to create this [report](javascript:openOther('m417/m417_3_f_l2_step7results.htm');). The results should contain nine observations.

proc print data=work.delays;

run;

## Topic Summaries

### Reading a SAS Data Set

You use a [DATA step](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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**;** |

### Customizing a SAS Data Set

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](javascript:%20return%20false;) 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](javascript:%20return%20false;)and the [execution phase](javascript:%20return%20false;).

You can [subset](javascript:%20return%20false;) the original data set with a WHERE statement for variables that are defined in the input data set, and a [subsetting IF statement](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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.

### Adding Permanent Attributes

When you use the [LABEL statement](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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](javascript:%20return%20false;) 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 ...**;** |

**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;