Paper TT15-SAS

The REPORT Procedure: A Primer for the Compute Block

Jane Eslinger, SAS Institute Inc.

ABSTRACT

It is well-known in the world of SAS® programming that the REPORT procedure is one of the best procedures for creating dynamic reports. However, you might not realize that the compute block is where all of the action takes place! Its flexibility enables you to customize your output.

This paper is a primer for using a compute block. With a compute block, you can easily change values in your output with the proper assignment statement and add text with the LINE statement. With the CALL DEFINE statement, you can adjust style attributes such as color and formatting. Through examples, you learn how to apply these techniques for use with any style of output. Understanding how to use the compute-block functionality empowers you to move from creating a simple report to creating one that is more complex and informative, yet still easy to use.

INTRODUCTION

PROC REPORT can create simple reports that list the data in your data set or complex reports that contain summary lines, text lines, coloring, and special formatting. To create a report, you have to decide what type of report you need and what you want it to look like. But how do you use PROC REPORT to move from the concept stage to a deliverable report? The answer: use a compute block within PROC REPORT.

A *compute block* starts with a COMPUTE statement and ends with an ENDCOMP statement. Between these two statements, you can use other SAS statements (assignment, CALL DEFINE, and LINE statements) that customize your output. Understanding the purpose of a compute block and how it works is a very important skill for obtaining the output that you want.

This paper, a primer that teaches you how to use the compute block, presents the following topics:

- a description of the COMPUTE statement and an explanation of how to reference variables within the compute block
- compute-block concepts, followed by examples that show you how to change values and add text
- a discussion of how to use a CALL DEFINE statement to change formats and attributes

LESSON 1: THE COMPUTE STATEMENT

PROC REPORT processes a data set by reading the variables in the order in which they appear from left to right in the COLUMN statement. The procedure builds the report one column and one row at a time, and COMPUTE statements are executed as the report is built.

The content of a COMPUTE statement is vital to the output that you want to generate. A COMPUTE statement can contain a number of arguments (for example, *report-item*, *location*, or *target).* The argument *report-item* represents either a data set variable, a computed variable, or a statistic. The argument *location* specifies at what point in the process the compute block executes in relation to the value for *target,* while the *target* argument controls when execution takes place. For details about all of the arguments that are available in the COMPUTE statement, see the *Base SAS 9.x Procedures Guide* for your release of SAS (**`support.sas.com/documentation/onlinedoc/base/index.html`**).

In general, compute blocks are executed in the following order:

1. COMPUTE *report-item*;
2. COMPUTE BEFORE;

(list continued)

3. COMPUTE BEFORE *target*;
4. COMPUTE BEFORE _PAGE_;
5. COMPUTE AFTER;
6. COMPUTE AFTER *target*;
7. COMPUTE AFTER _PAGE_;

The COMPUTE statement's behavior is dependent on the arguments that you include. The following **Syntax/Behavior** sections explain the actions of the compute block based on specific arguments in the COMPUTE statement.

Syntax:

COMPUTE *report-item*;

Behavior:

When you include the *report-item* argument, the compute block executes on every observation when that particular column is processed. In general, this statement is used for a specific *report-item* column so that you can calculate a value, change a value, change a format, apply style attributes, or create a temporary variable.

Syntax:

COMPUTE BEFORE;

Behavior:

With this syntax, the compute block is executed before the first detail row. The block is executed only once. Overall summary values for the analysis variables are available in the compute block. In addition, values of temporary variables that are created in this block are available to other compute blocks. Values for group or order variables **are not** available in this block. Text that is generated by LINE statements appears below the headers and above the first detail row in the report. CALL DEFINE statements set attributes on rows that are created by an RBREAK BEFORE statement.

Syntax:

COMPUTE BEFORE *target*;

Behavior:

When you use this syntax, the compute block is executed when the value of the target variable changes.

In this syntax:

- *target* specifies either a group variable or an order variable.
- BEFORE is a value for *location*. This value specifies that the compute block should be executed at the top (or beginning) of the section for a specific value of *target*.
- The value of the target variable for this specific section of the report is available to the compute block.
- The values of temporary variables created in this block are available to other compute blocks.
- Summary values for this specific *target* value are available to the compute block.
- CALL DEFINE statements set attributes on rows that are created by a BREAK BEFORE *target* statement.

Syntax:

COMPUTE BEFORE _PAGE_;

Behavior:

The compute block is executed once for each page. The page break can be generated either by the destination to which you send the output or by using the following BREAK statement syntax:

BREAK *location break-var* /PAGE;

Typically, this block outputs text by using a LINE statement. The text appears at the top of the page above the headers. However, it is still part of the table. Any variable that is listed in the LINE statement takes its value from the first detail row in the report.

Syntax:

COMPUTE AFTER;

Behavior:

The compute block is executed after the last detail row, and the block is executed only once (for each report). Overall summary values for the analysis variables are also available in the compute block. Values for group or order variables **are not** available in this block. Text that is generated by LINE statements appears after the last detail row in the report. This block is executed after each BY value if the PROC REPORT code contains a BY statement. CALL DEFINE statements set attributes on rows created by an RBREAK AFTER statement.

Syntax:

COMPUTE AFTER *target*;

Behavior:

With this syntax, the compute block is executed when the value of the target (break) variable changes. In this syntax:

- *target* specifies a variable that is defined as either a group variable or an order variable.
- AFTER is a value for *location*. This value specifies that the compute block should be executed at the bottom (or end) of the section for a specific value of *target*.
- The value of the target variable for this specific section of the report is available to the compute block.
- Summary values for this specific *target* value are available to the compute block.
- CALL DEFINE statements set attributes on rows created by a BREAK AFTER *target* statement.

Syntax:

COMPUTE AFTER _PAGE_;

Behavior:

The compute block is executed once for each page. The page break can be generated either by the destination to which you send the output or by using the following BREAK statement syntax:

BREAK *location break-var* /PAGE;

Any variable that is listed in the LINE statement takes its value from the last detail row in the report. The text is placed at the end of the table, but it is still part of the table.

Note: The exact location of the text might change for each page, based on the amount of information that is output to each page.

LESSON 2: REFERENCING VARIABLES INSIDE OF COMPUTE BLOCK

Now that you know when a compute block executes (based on the syntax you use), it is time to move to the next lesson and learn how to reference variables inside of a compute block. You must use the correct variable reference inside of all compute blocks and all statements (including assignment statements and CALL DEFINE statements).

Inside of a compute block, you must reference variables from the COLUMN statement in one of following four ways:

- **by name:** Variables that are defined as type GROUP, ORDER, DISPLAY, or COMPUTED are referred to by name. In the following example, the compute block displays the values in the report (or data set) variable, COMPANY, in all uppercase.

```
compute company;
   company=upcase(company);
endcomp;
```

- **by compound name:** Analysis variables are referred to by a compound name with the syntax *variable-name.statistic*. For example, the following compute block refers to an analysis variable named SALE.SUM:

```
compute sale;
   sale.sum=sale.sum * 1.1;
endcomp;
```

  By default, all numeric variables are analysis variables, and the default statistic is SUM. Even if you do not include a statistical keyword in a DEFINE statement, you must use the compound name in the compute block. If you do not use the correct reference to an analysis variable, you might see the following note in the log:

```
NOTE: Variable myvar is uninitialized.
```

  In this note, MYVAR is the analysis variable that is listed in the compute-block logic.

- **by alias:** When you create an alias in the COLUMN statement, you must use that alias in the compute block, as shown in the following example:

```
column weight=wt;
compute wt;
   wt=wt*1.1;
endcomp;
```

  If you use *variable-name.statistic* for an aliased column, the following error is generated:

```
ERROR: The variable type of MYVAR.SUM is invalid in this context.
```

- **by column number:** A variable that appears in the report under an ACROSS variable is referred to by the column number in the form _C*n*_, where *n* is number of the column (from left to right) in the report.

```
compute weight;
   _c2_=_c2_*1.1;
endcomp;
```

  Note: Columns that are not printed still count with regard to the numbering of the columns.

LESSON 3: CREATING A COMPUTE BLOCK

In this lesson, you want to calculate a new column based on the value of two other variables in the COLUMN statement. Pay close attention to the order of the variables in the example and what happens if you do not place the correct variable in the COMPUTE statement.

CREATING A NEW VARIABLE

In the following example, a new variable, PCT, is created. This variable is defined as a variable of type COMPUTED. A *computed variable* is a variable that does not exist in the data set.

```
proc report data=sashelp.cars(obs=50);
   column make invoice msrp pct;
   define make / group;
   define pct / computed format=percent8.2 'PCT';
   compute pct;
      pct = invoice.sum / msrp.sum;
   endcomp;
run;
```

In This Example:

- A COMPUTE statement with PCT as the report item indicates how the variable will be created. In this example, PCT is numeric; therefore, no additional options are necessary. However, if you want to create a character variable, you use the CHAR option in the COMPUTE statement.
- The compute block contains an assignment statement for creating the PCT variable:

  ```
  pct = invoice.sum /msrp.sum;
  ```

  In this case, the value for PCT is the ratio of INVOICE over MSRP.

Output:

Make	Invoice	MSRP	PCT
Acura	$270,136	$300,570	89.87%
Audi	$747,272	$822,850	90.82%
BMW	$792,413	$865,705	91.53%
Buick	$103,075	$113,090	91.14%

In this report, everything appears exactly as you expect it to appear. The PCT variable is the last variable in the COLUMN statement, and PCT is the last column in the report. All values are populated, and no errors, warnings, or messages are written to the log.

Now, suppose you want to make a change to this report. In the new output, you want the PCT column to appear between the MAKE and INVOICE columns. To change the column position in the report, you only need to change the COLUMN statement, as shown below:

```
proc report data=sashelp.cars(obs=50);
   column make pct invoice msrp;
   define make / group;
```

(code continued)

```
    define pct / computed format=percent8.2 'PCT';
    compute pct;
       pct=invoice.sum / msrp.sum;
    endcomp;
run;
```

However, when you submit this modified code, the log contains the following message:

```
NOTE: Missing values were generated as a result of performing an operation
on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      8 at 1:20
```

Output:

Make	PCT	Invoice	MSRP
Acura	.	$270,136	$300,570
Audi	.	$747,272	$822,850
BMW	.	$792,413	$865,705
Buick	.	$103,075	$113,090

As you see from the log and the output, PCT is not calculated as you expect. So what went wrong? Why does the PCT column have missing values in each observation?

In a compute block, if you reference a variable that is to the right of the compute-block variable in the COLUMN statement, then the referenced variable is missing or blank. In this example, both INVOICE and MSRP are listed **after** PCT in the COLUMN statement. Therefore, the values for INVOICE and MSRP are not available for use inside of the compute block to compute PCT.

GETTING IT RIGHT: A SHORTCUT

When you use multiple variables from the COLUMN statement inside of a compute block, it is important to pay attention to order of the variables in the COLUMN statement as well as which variable is specified as the report item in the COMPUTE statement. One method for avoiding the problem that is described in the previous section is to make sure that you always use the last variable in the COLUMN statement as the report item in the COMPUTE statement. The last variable in the COLUMN statement is the last column that is built for each row, which means that all of the values in that row are available to you. You can use one compute block for most of the statements that you need for the entire report, including assignment statements and CALL DEFINE statements (which are covered in a later lesson).

Note: This technique is programmer preference. You must decide what makes sense to you and what is easiest for you to read and modify.

As a beginner, one way to ensure that you always put the correct report item in the COMPUTE statement is to add a dummy, or placeholder, variable at the end of the COLUMN statement. For example, each time that you write PROC REPORT code, include the dummy variable _LASTVAR in the COLUMN statement, define it as NOPRINT, and use that variable in the COMPUTE statement.

```
proc report data=sashelp.cars(obs=50);
   column make pct invoice msrp _lastvar;
   define make / group;
```

(code continued)

```
    define pct / computed format=percent8.2 'PCT';
    define _lastvar / noprint;
    compute _lastvar;
       pct=invoice.sum / msrp.sum;
    endcomp;
run;
```

Output:

Make	PCT	Invoice	MSRP
Acura	89.87%	$270,136	$300,570
Audi	90.82%	$747,272	$822,850
BMW	91.53%	$792,413	$865,705
Buick	91.14%	$103,075	$113,090

LESSON 4: CHANGING REPORT VALUES

This lesson explains how you can use the skills you learned in the previous lessons to change values in a report.

CHANGING A VALUE IN A DETAIL ROW

In the simplest circumstance, you might need to change a specific value for one column. For example, suppose you have a character variable, CHARVAR, which has a value of **A**. You want to change the value to **A*** in order to point readers to a footnote. In this circumstance, you can use the CHARVAR variable in the COMPUTE statement. Inside of the compute block you need an IF-THEN statement to restrict the change to the correct records. A character variable is defined as DISPLAY by default so the variable reference is simply the variable name.

```
compute charvar;
  if charvar='A' then charvar='A*';
endcomp;
```

In a more complex circumstance, you might want to change the exact same value, but the character variable is under an ACROSS variable. So the final report has more than one column for CHARVAR. You can use the same COMPUTE statement as before. However, the content of the compute block looks quite different. Remember that variables under an ACROSS variable must be referenced by column number, _CN_. Therefore, you must have one IF-THEN statement for each column of CHARVAR.

```
compute charvar;
   if _c3_='A' then _c3_='A*';
   if _c6_='A' then _c6_='A*';
   if _c9_='A' then _c9_='A*';
   if _c12_='A' then _c12_='A*';
endcomp;
```

CHANGING A SUMMARY VALUE

The previous section changed values in detail rows. The next progression, then, is to change values in summary rows, which you create with BREAK statements. You can change summary values in multiple

compute blocks. The summary variables are available in both a compute block for that column and a COMPUTE AFTER block for the group or order variable that is being summarized. The syntax for both compute blocks is illustrated in the syntax that is shown in the next section. You can choose the syntax that makes the most sense to you.

Example: Changing Analyses Variables from a Summed Value to a Mean Value

In this next example, the values of two analysis variables, INVOICE and MSRP, need to be changed from the summed value to the mean in the summary rows. The *N statistic*, which is used in the COLUMN statement, displays the number of observations within each ORIGIN/MAKE combination. The N statistic is used as the denominator to calculate the mean value for INVOICE and MSRP within each ORIGIN value.

```
proc report data=sashelp.cars (obs=150);
   column origin make n invoice msrp;
   define origin / group;
   define make / group;
   break after origin / summarize;
   compute invoice;
      if upcase(_break_)='ORIGIN' then invoice.sum = invoice.sum/n;
   endcomp;
   compute after origin;
      msrp.sum=msrp.sum /n;
   endcomp;
run;
```

First, consider the COMPUTE INVOICE block.

- This block of code is executed each time the procedure reaches the variable INVOICE as the procedure moves left to right through the COLUMN statement.
- The IF condition restricts the number of times that the assignment statement is executed.
- The automatic _BREAK_ variable is populated only for summary rows. Because there are only three values for ORIGIN, _BREAK_ contains the value ORIGIN in only three rows. Therefore, the IF condition is true just three times. When the condition is true, the assignment statement is executed and the mean is calculated based on the current value of INVOICE.SUM and the N statistic.
- On the summary row, INVOICE.SUM contains the overall sum for that specific ORIGIN value and N contains the count for that ORIGIN value. The assignment statement changes the overall sum to the mean.

Next, look at the COMPUTE AFTER ORIGIN block.

- This block executes only when the value of ORIGIN changes. The summary row is output at the same time. Therefore, you can use this compute block to manipulate values in the summary row.
- This compute block only has access to the values ion the summary row, so an IF-THEN statement is not necessary.
- A simple assignment statement changes the sum of MSRP to the mean of MSRP.

Output:

Origin	Make	n	Invoice	MSRP
Asia	Acura	7	$270,136	$300,570
	Honda	1	$18,451	$20,140
Asia		*8*	*$36,073*	*$40,089*
Europe	Audi	19	$747,272	$822,850
	BMW	20	$792,413	$865,705
Europe		*39*	*$39,479*	*$43,296*

Example: Changing the Text in a Summary Row

In the previous example, the values for INVOICE and MSRP changed in the summary rows. However, it is not obvious that these values are means, and they look incorrect compared to the larger values above them. It would be helpful to include text in that row to indicate the analysis value is a mean value. In the previous output, the MAKE column has no value for those rows, so adding the text **`Mean:`** in that column can make the meaning of the analysis-variable value clear. Again, be aware that you can change the value either in a compute block with MAKE as the report item or in a COMPUTE AFTER block.

The following code examples show two methods for adding the **`Mean:`** text.

Example 1

```
compute after origin;
   invoice.sum=invoice.sum / n;
   msrp.sum=msrp.sum / n;
   make='Mean:';
endcomp;
```

Example 2

```
compute after origin;
   invoice.sum=invoice.sum / n;
   msrp.sum=msrp.sum / n;
endcomp;
compute make;
   if upcase(_break_)='ORIGIN' then make='Mean:';
endcomp;
```

Output:

Origin	Make	n	Invoice	MSRP
Asia	Acura	7	$270,136	$300,570
	Honda	1	$18,451	$20,140
Asia	*Mean:*	*8*	*$36,073*	*$40,089*
Europe	Audi	19	$747,272	$822,850
	BMW	20	$792,413	$865,705
Europe	*Mean:*	*39*	*$39,479*	*$43,296*

When you include text in a variable, one caveat to keep in mind is the type for the variable in which you are adding the text. For example, if you want to insert text into a character variable, as is the case in the assignment statement for the previous example, the insertion is straightforward. Keep in mind that the length of the inserted text must be less than or equal to the length of the report-item variable or truncation will occur. If the text needs to be longer, you must change the length of the variable in a DATA step that must appear before the PROC REPORT step.

A numeric variable cannot hold a text string. You can work around this behavior by making a character copy of the numeric variable in either a DATA step or in PROC REPORT. The following code illustrates how to make a character copy of the numeric variable CYLINDERS inside PROC REPORT code.

```
proc report data=sashelp.cars;
   column origin cylinders cychar n invoice msrp;
   define origin / group;
   define cylinders / group noprint;
   define cychar / computed 'CYCHAR'
   break after origin / summarize;
   compute cychar / char length=25;
      cychar=strip(put(cylinders,8.));
   endcomp;
   compute after origin;
      invoice.sum = invoice.sum/ n;
      msrp.sum = msrp.sum / n;
      cychar='Mean: ';
   endcomp;
run;
```

Output:

In the output, the character version of the variable, CYCHAR, is displayed. The numeric variable, CYLINDERS, is not displayed.

Origin	CYCHAR	n	Invoice	MSRP
Asia	3	1	$17,911	$19,110
	4	74	$1225495	$1321657
	6	69	$1781552	$1954827
	8	12	$497,213	$560,635
Asia	*Mean:*	*156*	*$22,578*	*$24,719*

LESSON 5: USING LINE STATEMENTS

A *LINE statement* is valid only within a compute block that is associated with a location, and this statement is used to output text. The text can include hardcoded text strings, variable values, or statistics. A LINE statement is executed after all other statements that appear inside of a compute block. In addition, this statement is always executed; you cannot use IF-THEN logic to conditionally execute the LINE statement.

A LINE statement's value is output to one cell that spans the width of the report. In non-listing ODS destinations, the LINE statement value appears as a large, merged cell. PROC REPORT considers multiple LINE statements within one compute block as one long text string.

ADDING A BLANK ROW BETWEEN GROUPS

The simplest application of the LINE statement is to output a blank row between target values. You can also use the SKIP option in a BREAK statement to write a blank line for the last row in a grouping. However, this this option is valid only in traditional SAS monospace output (ODS LISTING destination). For other types of output (other ODS destinations), a compute block that uses the LINE statement mimics this behavior, as illustrated by the following example:

```
proc report data=sashelp.class;
   column age name height;
   define age / order;
   compute after age;
      line ' ';
   endcomp;
run;
```

Output:

Age	Name	Height
11	Joyce	51.3
	Thomas	57.5
12	James	57.3
	Jane	59.8
	John	59
	Louise	56.3
	Robert	64.8

INSERTING A SECTION HEADING

You can also use a LINE statement to save space or to create section (group) headings. Often, a report can be too wide to fit on a page. In such a circumstance, it is helpful to prevent a column from being output and to use a LINE statement to output the value of a target variable as a row value.

The following example illustrates how to create section (or, group) headings.

```
proc report data=sashelp.cars(obs=100);
   column make type msrp invoice mpg_city mpg_highway horsepower wheelbase;
   define make / group noprint;
   define type / group;
```

(code continued)

```
   compute before make;
      line make $13.;
   endcomp;
run;
```

In This Example:

- The MAKE variable is defined with the GROUP and NOPRINT options. The NOPRINT option prevents the column from being printed, therefore giving more room for other columns.
- The values for MAKE (**Acura** and **Audi**) are made available in the COMPUTE BEFORE MAKE block that is included in the program. Those values are output when a LINE statement is executed.
- Each item in a LINE statement must have a SAS format. The LINE statement in this example uses a $13. format for MAKE. If you do not specify a format for MAKE, an error is generated.

Output:

Type	MSRP	Invoice	MPG (City)	MPG (Highway)	Horsepower	Wheelbase (IN)
Acura						
SUV	$36,945	$33,337	17	23	265	106
Sedan	$173,860	$156,821	102	136	1120	544
Sports	$89,765	$79,978	17	24	290	100
Audi						
Sedan	$534,400	$486,207	242	337	3100	1392
Sports	$198,520	$179,559	76	107	1105	396
Wagon	$89,930	$81,506	33	46	560	213

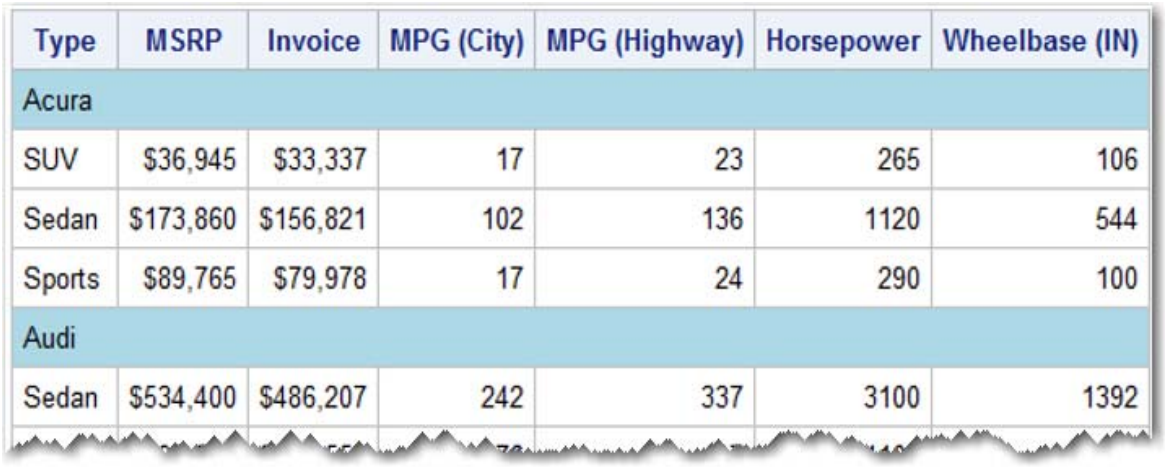
CONTROLLING THE ATTRIBUTES IN LINE STATEMENTS

By default, the text that is created by a LINE statement is centered. However, you can change the justification and other style attributes with the STYLE= option in the COMPUTE statement.

```
proc report data=sashelp.cars(obs=100);
   column make model type msrp invoice mpg_city mpg_highway horsepower;
   define make /group noprint;
   compute before make /style={just=l background=lightblue};
      line make $13.;
   endcomp;
run;
```

Output:

In this example, the section heading that contains the values for MAKE is now left justified and the background is changed to light blue.



Type	MSRP	Invoice	MPG (City)	MPG (Highway)	Horsepower	Wheelbase (IN)
Acura						
SUV	$36,945	$33,337	17	23	265	106
Sedan	$173,860	$156,821	102	136	1120	544
Sports	$89,765	$79,978	17	24	290	100
Audi						
Sedan	$534,400	$486,207	242	337	3100	1392

Note: Style attributes for multiple LINE statements from one compute block cannot be controlled separately.

But suppose that you want to add another style element, for example, a yellow line to separate the sections. How can you accomplish this task? To do that, you need another compute block with a target value that changes at the same time that the value for MAKE changes.

Your first thought might be to create an alias for MAKE in the COLUMN statement. However, PROC REPORT cannot have multiple summary rows for the same variable or location. An aliased variable is considered to be a copy. Therefore, that variable creates a break row at the same location. An error occurs if you create an alias and you use a COMPUTE BEFORE statement for both the original variable and the alias.

Instead of using an alias, you must create a copy of the MAKE variable within a DATA step, as shown below:

```
data cars;
   set sashelp.cars;
   make2=make;
run;
```

You can now use the copy, MAKE2, as necessary. In the following example, you want to add a yellow divider row at the top of each section.

Note: The order of the LINE statements in the output is based on the order of the MAKE and MAKE2 variables in the COLUMN statement.

```
proc report data=cars(obs=100);
   column make make2 type msrp invoice mpg_city mpg_highway horsepower
   wheelbase;
   define make / group noprint;
   define make2 / group noprint;
   define type / group;
```

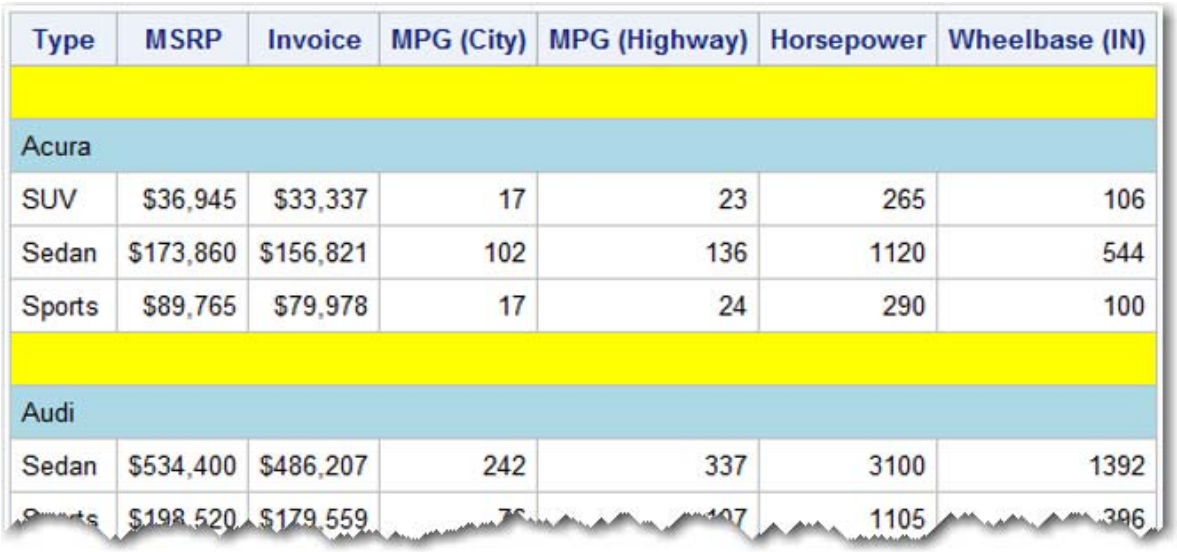
(code continued)

```
    compute before make2 / style={just=l background=lightblue};
       line make $13.;
    endcomp;
    compute before make / style={background=yellow};
       line ' ';
    endcomp;
run;
```

In This Example:

- The yellow divider rows are output first. Those rows are created with the MAKE variable, so MAKE should be first variable in the COLUMN statement.
- Then, MAKE2 is used to create the blue rows that contains the text value for MAKE.

Output:



Type	MSRP	Invoice	MPG (City)	MPG (Highway)	Horsepower	Wheelbase (IN)
Acura						
SUV	$36,945	$33,337	17	23	265	106
Sedan	$173,860	$156,821	102	136	1120	544
Sports	$89,765	$79,978	17	24	290	100
Audi						
Sedan	$534,400	$486,207	242	337	3100	1392
[illegible]	$198,520	$179,559	[illegible]	[illegible]	1105	396

USING A BREAK STATEMENT IN PLACE OF A LINE STATEMENT

LINE statements are very useful for writing out text or summary values that have a general justification. In the examples in the previous sections, the text was center justified or left justified across the entire table. However, LINE statements are not as useful when you need to align the text or summary values under specific columns. In LISTING output, you can create alignment using pointer controls such as the at-sign (@) and the plus sign (+). In other ODS destinations, pointer controls do not work in a predictable manner because of font face, font size, margins, and other style attributes.

Recall that LINE statements create one large, merged cell. Alignment is best achieved by creating a row with columns that match the detail rows above it. In terms of cell creation, a row that is created with a BREAK or RBREAK statement looks just like a detail row. When alignment is a requirement, it is best to use a BREAK or RBREAK statement rather than a LINE statement.

It is important to consider where you want the new row to appear in the report. If you require multiple rows, one for each section, you should use a BREAK statement. If you require only one row at the top or bottom of the report, an RBREAK statement will work. Depending on the situation, you might need to create a new variable in a DATA step.

The following example uses an RBREAK statement to create an overall sum for the PRICE variable.

```
proc report data=sashelp.pricedata;
   column productline price;
   define productline / group;
   define price / format=dollar10.2;
   rbreak after / summarize;
run;
```

Output:

Name of product line	Unit Price
Line1	$11,914.68
Line2	$11,537.37
Line3	$20,911.93
Line4	$26,236.98
Line5	$9,703.73
	$80,304.68

In your report, suppose that you also want to display the overall mean, and you want it to align with the overall sum. To achieve this output, you need a BREAK statement in addition to the RBREAK statement. The RBREAK row is output last. Therefore, in the new output, the RBREAK row will display the overall mean. A BREAK statement is required in order to output the overall sum.

As illustrated by the following example, you can present the overall sum below the detail rows by creating a variable that has the same value on every row of the data set.

```
data pricedata;
   set sashelp.pricedata;
   dummy=1;
run;

proc report data=pricedata;
   column dummy productline price price=price2;
   define dummy / group noprint;
   define productline / group;
   define price / format=dollar10.2;
   define price2 / mean noprint;
   rbreak after / summarize;
   break after dummy / summarize;
   compute after ;
      price.sum=price2;
      productline='Avg:';
   endcomp;
run;
```

In This Example:

The PROC REPORT code in this example includes the following modifications:

1. The new variable, DUMMY, is added to the beginning of the COLUMN statement.
2. An alias (PRICE2) for the PRICE variable is placed at the end of the COLUMN statement.
3. A DEFINE statement for the variable DUMMY specifies the GROUP and NOPRINT options.
4. A DEFINE statement for PRICE2 specifies the MEAN statistic and the NOPRINT option.
5. A BREAK statement that includes the SUMMARIZE option is added for the DUMMY variable.
6. A COMPUTE AFTER block reassigns the default SUM statistic to the MEAN statistic from the aliased variable.
7. An assignment statement assigns the text value **Avg:** to the PRODUCTLINE variable.

Output:

Name of product line	Unit Price
Line1	$11,914.68
Line2	$11,537.37
Line3	$20,911.93
Line4	$26,236.98
Line5	$9,703.73
	$80,304.68
Avg:	*$78.73*

LESSON 6: CALL DEFINE STATEMENTS

In the previous lessons you learned when the COMPUTE executes, how to reference variables, change values, and add text. In this lesson you will learn the CALL DEFINE statement. The CALL DEFINE statement is used to change attributes; it is where you can change how a cell or value looks. As with other statements the CALL DEFINE statement can be used in multiple compute blocks.

REFERENCING A COLUMN IN A CALL DEFINE STATEMENT

Similar to what you learned about the compute block, one of the first lessons you need to learn about the CALL DEFINE statement is how to reference the part of the report that you want to change. The syntax for the CALL DEFINE statement is as follows:

CALL DEFINE (*column-ID* | _ROW_ , '*attribute-name*', *value*);

The first argument in the syntax, *column-id*, specifies the part of the report to which you want to apply the attribute. The *column-ID* argument can be either of the following items:

- a character literal (in quotation marks) that is the column name
- a numeric literal that is the column number
- a numeric expression that resolves to the column number
- the _COL_ automatic variable
- the _ROW_ automatic variable

In general, referencing variables in a CALL DEFINE statement follow the same rules as referencing variables in a compute block. An analysis variable uses a compound name (for example, WEIGHT.SUM). Columns under an ACROSS variable must be referred to by a column number (in the following code, _C6_ is the column number). Group, order, or display variables must be referred to by name. All of the column references must be enclosed in quotation marks, as shown in the following example:

```
call define('name','style','style=[foreground=green]');
call define('_c6_','style','style=[foreground=blue]');
call define('weight.sum','style','style=[foreground=yellow]');
```

The _COL_ automatic variable refers to the report item in the COMPUTE statement. You can use _COL_ when the column that you want to change is the same column as is in the COMPUTE statement. If the column that you want to change is not the one in the COMPUTE statement, you need to use one of the other reference techniques (a column name, column number, or a compound name, as mentioned above). Do not put quotation marks around the _COL_ automatic variable.

```
compute name;
  call define(_col_,'style','style=[background=lightblue]');
endcomp;
```

The _ROW_ automatic variable, shown in the example that follows, applies to every column in a specific row. You can use _ROW_ in any compute block. Be aware that you cannot use _ROW_ when FORMAT is the value for the *attribute-name* argument. If you use _ROW_ with the FORMAT value, an error is generated in the SAS log. Do not put quotation marks around the _ROW_ automatic variable.

```
compute name;
   if name='John' then call
   define(_row_,'style','style=[background=lightblue]');
endcomp;
```

CHANGING A FORMAT

The CALL DEFINE statement has a number of possible values for the *attribute-name* argument. No matter which attribute name you use, the name must be enclosed in quotation marks. This paper focuses on just two attributes: FORMAT and STYLE.

CHANGING A FORMAT ATTRIBUTE

The CALL DEFINE statement with the FORMAT attribute is most often used to change the format on specific rows of output. Consider the following example:

```
proc report data=pricedata;
   column dummy productline price price=price2;
   define dummy / group noprint;
   define productline / group;
   define price / format=dollar10.2;
   define price2 / mean noprint;
   rbreak after / summarize;
   break after dummy / summarize;
   compute after;
      price.sum=price2;
      productline='Avg:';
      call define('price.sum','format','dollar10.');
   endcomp;
```

(code continued)

```
   compute after dummy;
      call define('price.sum','format','dollar10.');
   endcomp;
run;
```

This code generates a report that displays detail rows as well as the overall SUM and MEAN values. All of the price values are displayed with the DOLLAR10.2 format. You want to change the report so that the overall statistics are displayed without decimal places. To do that, the example code uses a CALL DEFINE statement that changes the format to DOLLAR10. for both rows.

In This Example:

- The same CALL DEFINE statement is used in two different compute blocks. The COMPUTE AFTER block controls the row that contains the overall mean. The COMPUTE AFTER DUMMY block controls the row that contains the overall sum.
- All three arguments in the CALL DEFINE statement are enclosed in quotation marks.
- For the third argument, only the format name (DOLLAR10.) is needed inside of the quotation marks. Using just the name differs from how you how define the style attribute, which is demonstrated in the next section.
- For the format, you can specify a SAS format or a user-defined format.

Output:

Name of product line	Unit Price
Line1	$11,914.68
Line2	$11,537.37
Line3	$20,911.93
Line4	$26,236.98
Line5	$9,703.73
	$80,305
Avg:	*$79*

CHANGING A STYLE ATTRIBUTE WITH THE STYLE ARGUMENT

You can also change your report style by using STYLE as the attribute name in the CALL DEFINE statement. The attribute is used most often to override style attributes such as foreground color, background color, and font weight in specific rows of the output. For a full list of attributes, see the section "Using ODS Style with Base Report Writing Procedures" in "Chapter 51: REPORT Procedure" of the *Base SAS® 9.4 Procedures Guide, Third Edition*.
(**support.sas.com/documentation/cdl/en/proc/67327/PDF/default/proc.pdf)**

When you change a style attribute, the third argument to the CALL DEFINE statement has a specific syntax:

'**STYLE=**{*style-attribute-name=style-attribute-value*<. . .*style-attribute-name-n=style-attribute-value-n*>}'

The STYLE= option must be enclosed in quotation marks. For this argument, you can use either bracket style: [ ] or { }.

In the following example, the SASHELP.HEART data set contains a variable called WEIGHT_STATUS. The report that you want to generate needs to draw the readers' attention to the **Overweight** status. You can accomplish this task by changing the style attributes with a CALL DEFINE statement.

```
proc report data=sashelp.heart;
   column weight_status ageatstart;
   define weight_status /group;
   define ageatstart/mean format=8.2;
   compute weight_status;
       if weight_status='Overweight' then
          call define(_row_,'style','style={background=red foreground=white
                      font_weight=bold}');
   endcomp;
run;
```

In This Example:

This example uses the _ROW_ automatic variable (for *column-ID*) because both columns need the style change.

Output:

Weight Status	Age at Start
Normal	41.83
Overweight	45.15
Underweight	41.27

You can build on the previous example by adding a blood-pressure status column, as shown in the following example. In this case, you want the blood-pressure status column to maintain the default style attributes for background, foreground, and font weight.

```
proc report data=sashelp.heart;
   column bp_status weight_status ageatstart;
   define bp_status /group;
   define weight_status /group;
   define ageatstart/mean format=8.2;
   compute ageatstart;
      if weight_status='Overweight' then do;
         call define('weight_status','style','style=[background=red
         foreground=white font_weight=bold]');
        call define('ageatstart.mean','style','style=[background=red
        foreground=white font_weight=bold]');
      end;
   endcomp;
run;
```

In This Example:

- Because you want the blood-pressure status column to maintain the default style attributes for (background, foreground, and font weight, you cannot use _ROW_ as the column-ID variable. You need to use a CALL DEFINE statement for each of the columns that need to change.
- The report-item column is changed to AGEATSTART, which follows after WEIGHT_STATUS in the COLUMN statement so that the compute block can change both columns.

Output:

Blood Pressure Status	Weight Status	Age at Start
High	Normal	45.61
	Overweight	47.15
	Underweight	44.94
Normal	Normal	41.20
	Overweight	43.60
	Underweight	41.13
Optimal	Normal	39.04
	Overweight	40.82

CHANGING A STYLE ATTRIBUTE BY USING FORMATS

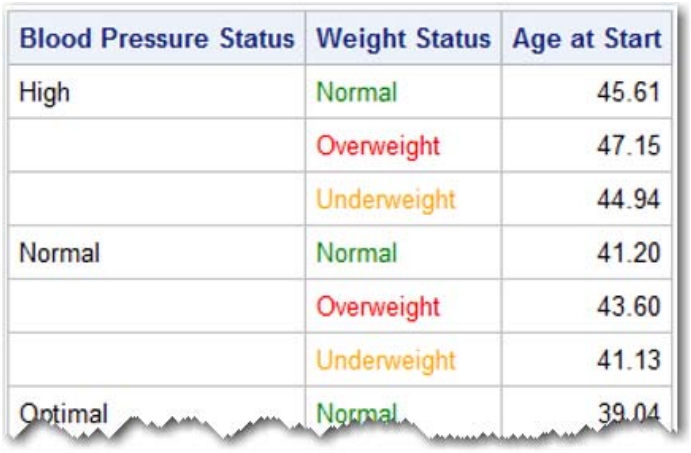
Attribute values are not limited to hardcoded values such as **`red`** or **`white`**. You can also use a format to apply multiple attribute values (for example, a color value) to a single column. Whereas the code in the previous example changed the foreground color with a hardcoded color value, the following example demonstrates how to change the color of the category values (**`Normal`**, **`Overweight`**, and **`Underweight`**) for WEIGHT_STATUS based on a user-defined format. First, you need to create a user-defined format (in this case, $STAT.) to match the category values to the color values that you want.

```
proc format;
   value $stat 'Normal'='green'
               'Underweight'='orange'
               'Overweight'='red';
run;
```

Then, change the CALL DEFINE statement to use the $STAT. format in place of hardcoded values.

```
proc report data=sashelp.heart;
   column bp_status weight_status ageatstart;
   define bp_status /group ;
   define weight_status /group;
   define ageatstart/mean format=8.2;
   compute weight_status;
      call define(_col_,'style','style={foreground=$stat.}');
   endcomp;
run;
```

Output:



Blood Pressure Status	Weight Status	Age at Start
High	Normal	45.61
	Overweight	47.15
	Underweight	44.94
Normal	Normal	41.20
	Overweight	43.60
	Underweight	41.13
Optimal	Normal	39.04

BUILDING A STYLE ATTRIBUTE

The final example in this primer is more advanced than the other examples, but this example encompasses a skill that is good to possess: building (or, creating) a complete, new style attribute. So far, you have learned to change a style attribute with a hardcoded value and with a format. But in some reports, you might want to change style attributes based on the value of another variable. You can do that using the CALL DEFINE statement, but you first have to build the attribute values.

Again, you want to change style attributes for the WEIGHT_STATUS variable in SASHELP.HEART. For this example, you use a DATA step to create a data-set variable (COLOR) that contains the color values that you want for **`Normal`**, **`Underweight`**, and **`Overweight`**.

```
data heart;
   set sashelp.heart;
   length color $20;
   if weight_status='Normal' then color='green';
   else if weight_status='Underweight' then color='orange';
   else if weight_status='Overweight' then color='red';
run;
```

You then add the new variable, COLOR to the appropriate place in the COLUMN statement and you reference the variable in the CALL DEFINE statement.

```
proc report data=heart(obs=20);
   column weight_status ageatstart color;
   define color /noprint;
   compute color;
     call define('weight_status','style','style={foreground='||color||'}');
   endcomp;
run;
```

In This Example:

- COLOR is placed at the end of the COLUMN statement, and it is defined with the NOPRINT option because its value does not need to be shown in the output.
- COLOR is used as the report-item column because it is the last variable in the COLUMN statement and its values are needed in the CALL DEFINE statement.
- WEIGHT_STATUS is placed as the column ID because that is the column to which you want to apply the color value.
- The third argument is built using the concatenation (||) operator. Notice the variable name COLOR does not appear directly inside of quotation marks. This is because COLOR is a variable name, and its value needs to be concatenated into the string. If COLOR is inside of quotation marks, no format will be applied. As a result, FOREGROUND=COLOR has no meaning to the CALL DEFINE statement.

Output:

Weight Status	Age at Start
Overweight	29
Overweight	41
Overweight	57
Overweight	39
Overweight	42
Overweight	58
Overweight	36
Normal	53
Overweight	35
Normal	52

Origin	cychar	n	Invoice	MSRP
Asia	3	1	$17,911	$19,110
	4	74	$1225495	$1321657
	6	69	$1781552	$1954827
	8	12	$497,213	$560,635
Asia	*Mean:*	*156*	*$22,578*	*$24,719*

CONCLUSION

In this primer, you learned how a compute block executes and what values are available in each compute block. With this knowledge, you can now change values on any row of a report. The primer also demonstrated how to manipulate LINE statements and formats. You should now understand that the compute block is where all of the action takes place inside of PROC REPORT. With this solid base of knowledge, you can create any kind of report you need.

RECOMMENDED READING

Carpenter, Art. 2007. "Chapter 7: Extending Compute Blocks" in *Carpenter's Complete Guide to the SAS REPORT Procedure.* 153-154. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2014. "Chapter 51: REPORT Procedure" in *Base SAS® 9.4 Procedures Guide: Third Edition*. Cary, NC: SAS Institute Inc. Available at **support.sas.com/documentation/cdl/en/proc/67327/PDF/default/proc.pdf**.

ACKNOWLEDGMENTS

The author is immensely grateful to Bari Lawhorn and Kathryn McLawhorn, both of whom contributed to this paper, and to Susan Berry, who edited the paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jane Eslinger
SAS Institute Inc.
Cary, NC
E-mail: **support@sas.com**
Web: **support.sas.com**