How to Easily Retain Variable Values in SAS Using the RETAIN Statement

Authored by

stats writer

November 22, 2025

The RETAIN statement is a foundational command within the SAS programming environment. Its primary function is to preserve the value of a specified variable across iterations of a DATA step. Unlike standard variables, which are automatically reset to missing at the start of each iteration, variables declared with RETAIN maintain their last assigned value. This capability is essential for operations requiring memory, such as calculating running totals or managing values read using a SET statement across multiple inputs. This guide provides comprehensive examples demonstrating how to leverage the power of the **RETAIN** statement in various analytical scenarios.

In essence, the **RETAIN** statement controls the initialization behavior of variables within the iterative processing of the DATA step. By default, SAS resets temporary variables to missing values for every observation processed. When you apply **RETAIN** to a variable, you instruct SAS to keep the value it held at the end of the previous iteration. If the variable is not explicitly initialized, SAS initializes numeric retained variables to zero and character retained variables to blanks during the first execution of the DATA step. This behavior is crucial for achieving sequential calculations.

While the utility of the **RETAIN** statement is broad, it is most frequently deployed in three key scenarios, particularly when dealing with sequential data processing or group-wise calculations:

Case 1: Using RETAIN to Calculate a Cumulative Sum

This is arguably the most common application of **RETAIN**. By retaining the previous calculation, we can continuously add the current observation's value to a running total. This method provides a highly efficient way to calculate simple cumulative metrics across an entire dataset without relying on complex SQL procedures.

```
data new_data;
set original_data;
retain cum_sum;
cum_sum + values_variable;
run;
```

Case 2: Using RETAIN to Calculate a Cumulative Sum by Group

When calculating cumulative sums that must restart (or reset) at the beginning of a new group, the **RETAIN** statement is used in conjunction with the BY statement. This setup requires the input data to be sorted by the grouping variable. The logic leverages the automatic **FIRST.variable** indicator to reset the accumulated value when a new group begins processing in the DATA step.

```
data new_data;
```

```
set original_data;
by grouping_variable
retain cum_sum_by_group;
if first.grouping_variable then cum_sum_by_group = values_variable;
else cum_sum_by_group = cum_sum_by_group + values_variable;
run;
```

Case 3: Using RETAIN to Calculate a Cumulative Count by Group

Similar to cumulative summation, **RETAIN** can track the sequential count of observations within each defined group. This is useful for assigning row numbers that restart for every subgroup or calculating the frequency order of events within a categorical variable. The logic utilizes the **FIRST.variable** flag to initialize the count to 1 whenever a new group is encountered.

```
data new_data;
set original_data;
by grouping_variable
retain count_by_group;
if first.grouping_variable then count_by_group = 1;
else count_by_group = count_by_group + 1;
run;
```

Demonstration Dataset Setup

To demonstrate these three essential applications in practice, we will use a sample dataset that tracks sales made over consecutive days by various stores. For group calculations (Cases 2 and 3), it is assumed the data is appropriately sorted by the `store` variable.

```
/*create dataset*/
data original_data;
input store $ sales;
datalines;
A 4
A 5
A 2
B 6
B 3
B 5
C 3
```

```
C 8
C 6
;
run;

/*view dataset*/
proc print data=original_data;
```

Obs	store	sales
1	A	4
2	A	5
3	A	2
4	B	6
5	B	3
6	B	5
7	C	3
8	C	8
9	C	6

Practical Example 1: Calculating the Global Running Total

The following code illustrates how to use the **RETAIN** statement alone to create a new variable, `cum_sales`, that calculates the running total of sales across the entire dataset, ignoring group boundaries. The variable `cum_sales` is declared using **RETAIN**, ensuring that its value persists and accumulates throughout the entire data processing stream.

```
/*calculate cumulative sum of sales*/
data new_data;
set original_data;
retain cum_sales;
cum_sales+sales;
run;

/*view results*/
proc print data=new_data;
```

Obs	store	sales	cum_sales
1	A	4	4
2	A	5	9
3	A	2	11
4	B	6	17
5	B	3	20
6	B	5	25
7	C	3	28
8	C	8	36
9	C	6	42

The resulting dataset now includes the `cum_sales` column, which holds the cumulative sum of values from the `sales` column. Since the **RETAIN** variable is initialized to zero, the accumulation process starts correctly from the first observation.

Reviewing the accumulation step-by-step:

Cumulative sum on row 1: The initial value (0) plus 4 equals **4**.
Cumulative sum on row 2: The retained value (4) plus 5 equals **9**.
Cumulative sum on row 3: The retained value (9) plus 2 equals **11**.

The core mechanism here is that the RETAIN statement ensures `cum_sales` is not reset. The implicit sum statement (`cum_sales+sales;`) handles the arithmetic accumulation during each pass of the DATA step.

Practical Example 2: Calculating Cumulative Sum by Store Group

To calculate cumulative metrics that reset at the boundary of a new category, we introduce the BY statement. The **RETAIN** statement maintains the running sum, but the **IF-THEN/ELSE** logic ensures the sum is reset when a new store group begins, utilizing the automatic `FIRST.store` indicator created by the BY statement.

```
/*calculate cumulative sum of sales by store*/
data new_data;
set original_data;
by store;
retain cum_sales_by_store;
```

```
if first.store then cum_sales_by_store = sales;
else cum_sales_by_store = cum_sales_by_store + sales;
run;

/*view results*/
proc print data=new_data;
```

Obs	store	sales	cum_sales_by_store
1	A	4	4
2	A	5	9
3	A	2	11
4	B	6	6
5	B	3	9
6	B	5	14
7	C	3	3
8	C	8	11
9	C	6	17

The new column, `cum_sales_by_store`, accurately reflects the cumulative sales for each store individually. When the processor encounters the first row for a new store (e.g., Store B or Store C), the condition `first.store` is true. This causes the cumulative variable to be initialized with the current `sales` value, effectively resetting the running total for that new group. For subsequent rows within that same group, the **ELSE** condition applies, allowing the retained value to accumulate the new sales figures.

Practical Example 3: Calculating Cumulative Count by Store Group

Using the same group processing logic, we calculate a cumulative count that resets for every store. This technique is often used to create sequential row indices within subgroups. Here, the initialization value is set to 1 when a new group begins, and 1 is added iteratively thereafter.

```
/*calculate cumulative count by store*/
data new_data;
set original_data;
by store
retain store_count;
```

```
if first.store then store_count = 1;
else store_count = store_count + 1;
run;

/*view results*/
proc print data=new_data;
```

Obs	store	sales	store_count
1	A	4	1
2	A	5	2
3	A	2	3
4	B	6	1
5	B	3	2
6	B	5	3
7	C	3	1
8	C	8	2
9	C	6	3

The resulting column, `store_count`, provides a sequential count for each observation within each individual store group. The RETAIN statement preserves `store_count` across iterations. When `first.store` is true, `store_count` is explicitly set to 1, marking the start of the new group's count. Otherwise, 1 is added to the retained count from the previous observation.

Conclusion and Further Exploration

The **RETAIN** statement is an indispensable tool in SAS programming, allowing programmers to maintain variable states across the repetitive processing of dataset iterations. Mastering its use, particularly alongside **FIRST.** and **LAST.** operators defined by the BY statement, unlocks significant power for statistical and business calculations that rely on sequential accumulation or group-wise comparisons.

The following tutorials explain how to perform other common tasks in SAS: