Use the RETAIN Statement in SAS (With Examples)

Authored by

Mohammed looti

April 9, 2026

Introduction to the RETAIN Statement in SAS

In the realm of data manipulation, particularly when dealing with sequential processing or calculations that depend on previous observations, the behavior of variables within a **DATA step** in **SAS** is crucial. By default, **SAS DATA step** variables are reinitialized to a **missing value** at the beginning of each iteration for most operations. This default behavior, while suitable for many tasks, can pose a challenge when you need a variable to retain its value from one observation to the next.

This is precisely where the **RETAIN statement** becomes an indispensable tool. The **RETAIN statement** explicitly instructs **SAS** to prevent a specified variable from being reset to **missing** at the start of a new observation's processing within a **DATA step**. Instead, it holds onto its value from the previous iteration, making it ideal for calculations requiring continuity across rows.

Mastering the **RETAIN statement** significantly enhances your ability to perform complex data transformations and aggregations in **SAS**. This article will delve into its core functionality and demonstrate its practical applications through detailed examples.

Understanding the Core Functionality of RETAIN

At its heart, the **RETAIN statement** modifies the default variable initialization behavior within the **DATA step**. Normally, when **SAS** processes a new observation, all variables created in the **DATA step** (that are not read from the input dataset) are set to **missing**. This ensures that calculations for each row are independent, preventing unintended carry-overs from previous rows.

However, for tasks like computing running totals, sequential counts, or carrying forward values, this reinitialization is undesirable. The **RETAIN statement** overrides this reset, allowing a variable to retain its value from the preceding observation. When a variable is first declared with **RETAIN**, it is initialized to **zero** for numeric variables or **blanks** for character variables. Subsequent iterations then use the value from the prior row.

This capability is fundamental for iterative calculations, enabling you to build complex logic that processes data sequentially, referencing values computed in earlier stages of the **DATA step**.

Key Applications of the RETAIN Statement

The versatility of the **RETAIN statement** makes it suitable for a broad array of data manipulation tasks. While its applications are extensive, three scenarios stand out as particularly common and illustrative of its power:

Case 1: Use RETAIN to Calculate a Cumulative Sum

This is perhaps the most straightforward and frequently used application. By retaining the sum from the previous observation, you can easily compute a running total across your dataset.

Case 2: Use RETAIN to Calculate a Cumulative Sum by Group
When you need to compute running totals that reset for each category or group within your data, the **RETAIN statement**, combined with the **BY statement** and **first.variable** processing, becomes invaluable.

Case 3: Use RETAIN to Calculate a Cumulative Count by Group
Similar to cumulative sums by group, you can also use **RETAIN** to count observations within each group, resetting the count at the start of a new group.

These three cases cover a wide range of analytical needs and provide a solid foundation for understanding the practical implementation of the **RETAIN statement**. Let's explore the syntax for each.

Syntax for Common RETAIN Use Cases

Before diving into real-world examples, it's beneficial to review the general syntax for each of the primary use cases. These code snippets illustrate the fundamental structure you'll employ when applying the **RETAIN statement**.

Case 1: Use RETAIN to Calculate a Cumulative Sum

To calculate a simple **cumulative sum** across all observations in a dataset, you declare a new variable with **RETAIN** and then use a sum statement (`+`) to add the current row's value to the retained sum.

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

In this syntax, `cum_sum` is the new variable that will store the running total, and `values_variable` is the variable from your input dataset that you want to sum. The `retain cum_sum;` statement ensures that `cum_sum` holds its value from the previous iteration, allowing the sum statement `cum_sum + values_variable;` to correctly accumulate values.

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

For group-wise **cumulative sums**, you introduce the **BY statement** to identify groups and conditional logic using **first.grouping_variable** to reset the sum at the start of each new group.

```
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;
```

Here, `grouping_variable` is the categorical variable defining your groups (e.g., 'store', 'region'). The **BY statement** creates temporary variables, `first.grouping_variable` and `last.grouping_variable`, which are `1` (true) for the first/last observation in a group and `0` (false) otherwise. The `if first.grouping_variable then...` condition ensures the cumulative sum resets when a new group begins, starting with the `values_variable` for that group's first observation.

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

Calculating a cumulative count within groups follows a similar logic to the cumulative sum by group. You use the **BY statement** and **first.grouping_variable** to manage the reset, but instead of adding a value, you increment a counter.

```
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;
```

In this structure, `count_by_group` is the variable that will store the running count. Upon the first observation of a new group (`first.grouping_variable`), `count_by_group` is initialized to `1`. For subsequent observations within the same group, it is incremented by `1`, effectively providing a sequential count for each item within its respective group.

Setting Up Our Example Dataset

To illustrate these concepts, we will use a straightforward dataset that simulates daily sales figures for different stores. This dataset will serve as the input for all subsequent examples, allowing us to see how the **RETAIN statement** transforms the data.

The `original_data` dataset contains two variables: `store` (a character variable indicating the store identifier) and `sales` (a numeric variable representing daily sales). The data is intentionally structured to demonstrate sequential processing and group-wise calculations effectively.

```
/*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;
```

After executing the code above, the `original_data` dataset will be displayed, showing the initial structure of our sales data. This view is crucial for understanding the input before we apply transformations using the **RETAIN statement**.

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

Example 1: Calculating a Global Cumulative Sum

Let's begin by demonstrating how to compute a simple **cumulative sum** of sales across the entire dataset. This operation provides a running total that accumulates sales values from the first observation to the current one, without resetting.

The following **SAS DATA step** utilizes the **RETAIN statement** to achieve this. We create a new variable, `cum_sales`, which will store our running total.

```
/*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;
```

In this code, the `retain cum_sales;` statement is critical. It ensures that `cum_sales` is not reset to **missing** at the start of each observation. Initially, `cum_sales` is automatically set to **zero**. Then, `cum_sales+sales;` is a **sum statement**, which is shorthand for `cum_sales = cum_sales + sales;` and automatically handles **missing values** by treating them as zero in the summation.

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 `new_data` dataset now includes the `cum_sales` column, which accurately reflects the **cumulative sum** of sales. For instance:

Cumulative sum on row 1: **4** (initial `cum_sales` 0 + `sales` 4)
Cumulative sum on row 2: 4 + 5 = **9** (previous `cum_sales` 4 + `sales` 5)
Cumulative sum on row 3: 9 + 2 = **11** (previous `cum_sales` 9 + `sales` 2)

This progressive addition continues for every observation, providing a complete running total for all sales.

Example 2: Calculating a Cumulative Sum by Group

Often, you'll need to calculate running totals that reset for each distinct category within your data. In our example, this means computing the **cumulative sum** of sales for each individual store, resetting the sum when a new store's data begins. This requires combining the **RETAIN statement** with the **BY statement** and **first.variable** logic.

First, it is crucial that your dataset is sorted by the grouping variable (`store` in this case). The **BY statement** in the **DATA step** requires the input data to be sorted by the specified **BY variable**. Our `original_data` is already sorted by `store`.

```
/*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;
```

Here, `by store;` tells **SAS** to process the data in groups based on the `store` variable. The `first.store` automatic variable is `1` (true) only for the first observation of each new store group. When `first.store` is true, `cum_sales_by_store` is initialized with the current `sales` value, effectively resetting the cumulative sum for that new store. For all subsequent observations within the same store group (`else`), the current `sales` value is added to `cum_sales_by_store`, continuing the running total for that specific store.

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 output clearly shows the `cum_sales_by_store` column. Notice how the cumulative sum resets to the initial sales value whenever a new store (A, B, or C) appears. This demonstrates the powerful combination of **RETAIN** and **BY-group processing** for performing group-specific calculations.

Example 3: Calculating a Cumulative Count by Group

Beyond summing values, the **RETAIN statement** is equally effective for generating cumulative counts within groups. This allows you to number observations sequentially within each category, resetting the count for every new group.

Similar to the cumulative sum by group, this operation relies on the dataset being sorted by the grouping variable (`store`). The **BY statement** and `first.store` logic are again central to managing the group boundaries.

```
/*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;
```

In this **DATA step**, `store_count` is the new variable designed to hold the cumulative count for each store. The `retain store_count;` statement ensures that `store_count` maintains its value across observations within a group. When **first.store** is true, `store_count` is set to `1`, marking the first observation of a new store. For subsequent observations within the same store, `store_count` is incremented by `1`, providing a continuous count.

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 `store_count` column in the output demonstrates the cumulative count for each store. You can observe how the count restarts from `1` for Store B and then again for Store C, providing an ordered sequence within each group. This technique is invaluable for generating row numbers within groups, identifying the Nth observation, or performing other sequence-dependent analyses.

Conclusion: Mastering Data Transformation with RETAIN

The **RETAIN statement** is a fundamental and powerful feature in **SAS** for controlling how variables behave across observations within a **DATA step**. By preventing the reinitialization of variables to **missing**, it enables the creation of iterative calculations, such as **cumulative sums** and counts, both globally and within specific groups.

Its synergistic use with the **BY statement** and **first.variable** processing unlocks advanced data manipulation capabilities, allowing you to elegantly handle complex analytical requirements. Understanding and effectively applying the **RETAIN statement** is a hallmark of efficient **SAS programming**, empowering you to derive deeper insights from your datasets.

Additional Resources

For further exploration of **SAS** programming techniques and common data tasks, consider these related tutorials:

Understanding the **Sum Statement** in SAS

Guide to **PROC PRINT** for Data Inspection

Advanced **BY-Group Processing** Techniques