How to Use a DO WHILE Statement in SAS

Authored by

stats writer

November 19, 2025

1. Introduction to Iterative Processing in SAS

The ability to perform repetitive tasks, known as looping, is fundamental to efficient programming and data manipulation. In the context of SAS, iterative processing is often necessary when calculating cumulative values, simulating data, or processing records multiple times within a single observation or across the entire dataset. While simple iterations can be handled by standard **DO** loops, complex scenarios require control flow mechanisms that execute based on a dynamic, ongoing condition rather than a predetermined count. This is where the powerful **DO WHILE statement** becomes indispensable for advanced data analysts.

The **DO WHILE statement** facilitates the execution of a block of SAS statements repeatedly until a specific logical condition is no longer met. Crucially, the condition is evaluated at the beginning of each cycle. If the condition is true, the statements enclosed between the **DO WHILE** and the corresponding **END** statement are executed. This mechanism provides precise control over the duration of the iteration, ensuring the loop continues only as long as the specified criteria hold true, making it ideal for tasks where the number of required iterations is unknown beforehand, such as generating sequence data until a threshold is reached.

Understanding control flow structures like **DO WHILE** is essential for maximizing productivity within the Data Step. Unlike processing data row-by-row, which is typical of the implicit Data Step loop, the **DO WHILE** structure allows internal, nested looping within a single observation's processing cycle. This capability is frequently used for generating new observations based on existing data, performing complex simulations, or dynamically adjusting calculations based on intermediate results generated within the loop itself.

2. Understanding the DO WHILE statement Syntax and Function

The syntax of the **DO WHILE statement** is straightforward yet powerful, requiring the explicit definition of the iteration boundary using an **END statement**. The fundamental structure dictates that the logical expression defining the continuation of the loop must immediately follow the **WHILE** keyword. This expression can be any valid SAS logical comparison that evaluates to either true or false (e.g., `A < B`, `C = 'Yes'`, or complex compound conditions). As long as the expression remains true, the statements enclosed between the **DO WHILE** and the corresponding **END** statement are executed; the moment it evaluates to false, control flow exits the loop and proceeds to the next statement following the **END** keyword.

It is critical to ensure that at least one statement within the loop body modifies the variable or variables used in the **WHILE condition**. If the logical expression controlling the loop is never altered--for example, if the value of the variable being tested is static--the program will result in an infinite looping scenario, potentially consuming excessive system resources and requiring manual

termination. Experienced SAS programmers always include necessary incrementers, decrementers, or calculation updates within the **DO WHILE** block to guarantee convergence and loop termination based on the predefined criteria.

Furthermore, SAS provides flow control modifiers that can be used inside the loop to manage complex logic. The **CONTINUE statement**, for instance, allows the programmer to skip the remaining statements in the current iteration of the loop and immediately jump back to the **DO WHILE** check for the start of the next cycle. This is useful for bypassing calculations when certain interim conditions are met. Conversely, the **LEAVE statement** provides a mechanism for an immediate and unconditional exit from the current loop structure, bypassing the remaining statements in the loop body and skipping the condition check entirely. These statements offer fine-grained control over complex iterative logic within the Data Step.

3. The Role of DO WHILE within the Data Step

In SAS programming, the Data Step is the powerhouse for data creation and modification. When a **DO WHILE** loop is employed within a Data Step that does not read an existing dataset (i.e., when no `SET` statement is present, as seen in the first example below), the Data Step executes only once by default. In this specific scenario, the **DO WHILE** statement drives the entire creation of the dataset, generating multiple observations sequentially until the loop condition fails. For each successful iteration where an **OUTPUT statement** is included, a new observation is written to the output dataset.

When the **DO WHILE** loop is used in conjunction with an input dataset (i.e., using a `SET` statement), the outer Data Step loop processes one observation at a time. The **DO WHILE** loop then executes repeatedly **for that single observation**, allowing complex intra-observation calculations or the creation of multiple output records from a single input record (a process often referred to as 'unstacking' or data expansion). This structure enables highly localized iterative data processing before the standard implicit output occurs or before an explicit **OUTPUT** statement is executed.

The correct implementation of **DO WHILE** requires careful initialization of all control variables before the loop begins. Variables that control the looping process, such as counters or cumulative sums, must be set to their starting values immediately prior to the **DO WHILE statement**. Furthermore, if these variables are not explicitly updated within the loop body, the logical flow will result in an infinite loop or premature termination if the initial condition is false. The structure ensures that the loop body is only executed if the condition is met (true) upon the initial check, making it a "pre-test" loop.

You can use a **DO WHILE** statement in SAS to perform iterative execution repeatedly **while** a specified logical condition remains true.

The following examples provide practical demonstrations of utilizing this conditional looping construct.

4. Example 1: Basic Unconditional Iteration Control

This first example illustrates the fundamental application of the **DO WHILE statement** to generate a dataset based purely on iterative calculations, independent of existing input data. The objective is to create a new SAS dataset named `my_data` that contains two derived variables, **var1** and **var2**. The iterative process is designed to continue generating new values and new observations **while** the value of **var1** is strictly less than 100.

The code initializes both **var1** and **var2** to 1. These initial assignments are crucial as they set the starting point for the calculations and ensure the **WHILE condition** (`var1 < 100`) is true at the beginning, allowing the loop to execute at least once. Within the loop, the values of **var1** and **var2** are calculated based on their preceding values, creating an evolving sequence. Note the presence of the **OUTPUT statement**, which is explicitly placed inside the loop block. This command tells SAS to write the current values of all variables to a new observation in the output dataset during each iteration.

The code below demonstrates how to use the **DO WHILE** statement to control data generation. It is essential that **var1** is updated correctly to eventually exceed 100, thus fulfilling the termination criteria and preventing an infinite loop. The combination of arithmetic updates ensures that the condition is eventually violated, allowing the Data Step to complete and write the final log messages once the process concludes.

```
/*create dataset using DO WHILE statement*/
data my_data;

var1 = 1;
var2 = 1;

do while(var1 < 100);
var1 = var1 + var2;
var2 = var1 * var2;
var1 + 1;

output;

end;

run;
```

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

Obs	var1	var2
1	3	2
2	6	10
3	17	160
4	178	28320

5. Detailed Walkthrough of Example 1 Execution

When the Data Step begins execution, **var1** and **var2** are initialized to 1. The **DO WHILE statement** checks the initial condition (`1 < 100`), which is true, initiating the first cycle. Inside the loop, **var1** is updated (1 + 1 = 2), and then **var2** is updated (2 * 1 = 2). The statement `var1 + 1;` calculates a temporary value (3) but crucially does not reassign it, meaning **var1** retains the value 2 before the output step. The **OUTPUT statement** then writes the values (var1=2, var2=2) as the first observation.

The loop cycles again, checking the condition using the new value of **var1** (2 < 100). This continues for several cycles, rapidly increasing the values of both variables due to the multiplicative update of **var2** (`var2 = var1 * var2`). The **DO WHILE** statement drives this generation process, continuing to execute the enclosed statements and outputting new rows as long as the primary variable, **var1**, remains strictly below the threshold of 100.

Once the calculations within an iteration cause **var1** to jump past 100, the loop executes its final **OUTPUT** with that value. Upon returning to the **DO WHILE** check, the condition is evaluated using the final, updated value of **var1**. Because this value now exceeds 100, the condition is false. At this precise point, the **DO WHILE statement** immediately terminates the iterative process, and no further values or observations are generated within this specific Data Step execution. This demonstrates that the dataset is successfully capped precisely by the dynamic logical criterion.

6. Example 2: Combining DO Iteration with WHILE Condition

The second example introduces a more complex control structure by combining an iterative **DO loop** (using the **TO statement**) with the conditional execution control of the **WHILE clause**. This hybrid structure, written as a `DO index = start TO stop WHILE (condition)`, allows the loop to iterate based on a predefined counter (**var2** iterating from 1 to 5) but simultaneously imposes a

secondary, dynamic termination condition based on a calculated variable (**var1** < 10). This provides a powerful mechanism for conditional data generation.

In this setup, **var2** is intended to be the index variable, ranging from 1 to 5. However, the loop will only proceed to the next iteration if the **WHILE condition** is true based on the value of **var1** from the *previous cycle*. The variable **var1** is initialized to 0 outside the loop. Inside the loop, **var1** is calculated as the cube of the current index (`var2**3`). If, at the start of any iteration, the value of **var1** calculated in the prior step would cause the condition `var1 < 10` to be false, the loop terminates immediately, regardless of whether **var2** has reached its maximum value of 5.

This example demonstrates a powerful technique for conditionally limiting a fixed iterative process. The **TO statement** instructs SAS to attempt iterations 1 through 5 for **var2**. However, the **WHILE** clause acts as a gatekeeper, overriding the fixed count based on the calculated values of **var1**. This ensures that the generated dataset contains only observations where **var1** adhered to the specified numerical limit, stopping the loop as soon as the calculated value breaches the defined ceiling.

```
/*create dataset using DO WHILE statement with TO statement*/
data my_data;

var1 = 0;

do var2 = 1 to 5 while(var1 < 10);
var1 = var2**3;

output;

end;

run;

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

Obs	var1	var2
1	1	1
2	8	2
3	27	3

7. Analysis of Hybrid Loop Execution

Tracing the execution of the hybrid loop structure presented in Example 2 provides clarity on how the **WHILE** condition overrides the **TO** count. The critical point is that the **WHILE** condition is checked using the value of **var1** carried over from the previous iteration.

Iteration 1 (var2 = 1): var1 is initialized to 0. The condition (0 < 10) is true. Inside the loop, **var1** is updated: 1**3 = 1. Output is written (var1=1, var2=1).

Iteration 2 (var2 = 2): The loop increments **var2** to 2. The condition check uses the value of **var1** from the previous iteration (1 < 10), which is true. Inside the loop, **var1** is updated: 2**3 = 8. Output is written (var1=8, var2=2).

Iteration 3 (var2 = 3): The loop increments **var2** to 3. The condition check uses the value of **var1** from the previous iteration (8 < 10), which is true. Inside the loop, **var1** is updated: 3**3 = 27. Output is written (var1=27, var2=3).

Attempted Iteration 4 (var2 = 4): The loop attempts to increment **var2** to 4. Before executing the loop body, the **WHILE condition** is checked using the current value of **var1** (27 < 10). This condition is now **false**.

Because the **WHILE condition** failed before the fourth iteration could start, the loop terminates immediately after writing the third observation. The instruction to run up to `var2 = 5` is overridden by the dynamic constraint imposed by the value of **var1** exceeding 10. This demonstrates the efficiency of the hybrid loop in ensuring that generated data adheres to a defined numerical limit, providing a safeguard against undesired output.

8. Best Practices and Alternatives to DO WHILE

While the **DO WHILE statement** is exceptionally useful for conditional iteration, particularly when the exit point is determined by data calculations, programmers should consider its applicability carefully. A best practice is to always define a fail-safe mechanism, such as adding a counter that imposes a maximum number of iterations, even if the primary termination condition is based on data. This prevents runaway processes in case of unforeseen logical errors that could lead to an infinite loop, especially when dealing with complex mathematical relationships or data dependencies.

For situations where the condition must be evaluated only **after** the loop has executed at least once, SAS offers the **DO UNTIL statement**. The key difference is the timing of the check: **DO WHILE** checks the condition at the top (before execution), whereas **DO UNTIL** checks the condition at the bottom (after execution). If a block of code must execute at least once, **DO UNTIL**

is the more appropriate choice. If it is possible, or even desirable, for the loop to execute zero times if the initial condition is immediately false, **DO WHILE** should be used.

Ultimately, choosing the correct looping structure--whether **DO WHILE**, **DO UNTIL**, or a standard iterative **DO** loop--is a mark of efficient and robust SAS programming. Programmers must select the structure that most clearly and safely implements the desired control flow logic within the Data Step.

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