临床研究SAS高级编程

—— Do-to & Array

Contents

Generating Data with DO Loops

Contents

Introduction (1)

DO loops can execute any number of times in a single iteration of the DATA step. Using DO loops enables you to write concise DATA steps that are easier to change and debug.

For example, the DO loop in this program eliminates the need for 12 separate programming statements to calculate annual earnings:

```
data finance.earnings;
   set finance.master;
   Earned=0;
   do count=1 to 12;
      earned+(amount+earned)*(rate/12);
   end;
run;
```

Introduction (2)

In this section, you learn to

- construct a DO loop to perform repetitive calculations
- control the execution of a DO loop
- generate multiple observations in one iteration of the DATA step
- construct nested DO loops.

Constructing DO Loops (1)

Introduction

- DO loops process a group of statements repeatedly rather than once. This can greatly reduce the number of statements required for a repetitive calculation. For example, these 12 Sum statements compute a company's annual earnings from investments. Notice that all 12 statements are identical.

Constructing DO Loops (2)

Example:

```
data finance.earnings;
   set finance.master;
   Earned=0;
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
   earned+(amount+earned)*(rate/12);
Run;
```

Each Sum statement accumulates the calculated interest earned for an investment for one month. The variable Earned is created in the DATA step to store the earned interest. The investment is compounded monthly, meaning that the value of the earned interest is cumulative. A DO loop enables you to achieve the same results with fewer statements. In this case, the Sum statement executes 12 times within the DO loop during each iteration of the DATA step.

```
data finance.earnings;
   set finance.master;
   Earned=0;
   do count=1 to 12;
      earned+(amount+earned)*(rate/12);
   end;
run;
```

Constructing DO Loops (3)

General form of do loops

- To construct a DO loop, you use the DO and END statements along with other SAS statements.
- General form, simple iterative DO loop:
  - DO *index-variable=start* TO *stop* BY increment;

    SAS statements

    END;

    where the *start*, *stop*, and *increment* values
    - **are set upon entry into the DO loop**
    - **cannot be changed during the processing of the DO loop**
    - **can be numbers, variables, or SAS expressions.**
  - The END statement terminates the loop.

    Note The value of the index variable can be changed within the loop.

Constructing DO Loops (4)

- When creating a DO loop with the iterative DO statement, you must specify an **index variable**. The index variable stores the value of the current iteration of the DO loop. You can use any valid SAS name.

```
DO index-variable = start TO stop BY increment;
   SAS statements
END;
```

- Next, specify the conditions that execute the DO loop. A simple **specification** contains a **start value**, a **stop value**, and an **increment value** for the DO loop.

```
DO index-variable = start TO stop BY increment;
   SAS statements
END;
```

- The start value specifies the initial value of the index variable.

```
DO index-variable = start TO stop BY increment;
      SAS statements
END;
```

Constructing DO Loops (5)

- The TO clause specifies the stop value. The stop value is the last index value that executes the DO loop.

```
DO index-variable=start TO stop BY increment;
   SAS statements
END;
```

- The optional BY clause specifies an increment value for the index variable. Typically, you want the DO loop to increment by 1 for each iteration. If you do not specify a BY clause, the default increment value is *1*.

```
DO index-variable=start TO stop BY increment;
   SAS statements
END;
```

- For example, the specification below increments the index variable by 1, resulting in quiz values of *1*, *2*, *3*, *4*, and *5*:

```
do quiz=1 to 5;
```

- By contrast, the following specification increments the index variable by 2, resulting in rows values of *2*, *4*, *6*, *8*, *10*, and *12*:

```
do rows=2 to 12 by 2;
```

DO Loop Execution (1)

- Using the form of the DO loop that was just presented, let's see how the DO loop executes in the DATA step. This example calculates how much interest was earned each month for a one-year investment.
  - Example:

```
data finance.earnings;
   Amount=1000;
   Rate=.075/12;
   do month=1 to 12;
      Earned+(amount+earned)*(rate);
   end;
run;
```

DO Loop Execution (2)

- This DATA step does not read data from an external source. When submitted, it compiles and then executes only once to generate data. During compilation, the program data vector is created for the **Finance.Earnings** data set.

Program Data Vector

_N_	Amount	Rate	month	Earned
•	•	•	•	•

- When the DATA step executes, the values of Amount and Rate are assigned.

Program Data Vector

_N_	Amount	Rate	month	Earned
1	1000	0.00625	•	•

DO Loop Execution (3)

- Next, the DO loop executes. During each execution of the DO loop, the value of Earned is calculated and is added to its previous value; then the value of month is incremented. On the twelfth execution of the DO loop, the program data vector looks like this:

Program Data Vector

_N_	Amount	Rate	month	Earned
1	1000	0.00625	12	77.6326

DO Loop Execution (4)

- After the twelfth execution of the DO loop, the value of month is incremented to 13. Because 13 exceeds the stop value of the iterative DO statement, the DO loop stops executing, and processing continues to the next DATA step statement. The end of the DATA step is reached, the values are written to the **Finance. Earnings** data set, and in this example, the DATA step ends. Only one observation is written to the data set.

SAS Data Set Finance.Earnings

Amount	Rate	month	Earned
1000	0.00625	13	77.6326

Notice that the index variable month is also stored in the data set. In most cases, the index variable is needed only for processing the DO loop and can be dropped from the data set.

Counting do loop iterations (1)

Counting iterations of DO loops

- In some cases, it is useful to create an index variable to count and store the number of iterations in the DO loop. Then you can drop the index variable from the data set.
  - Example:

```
data work.earn (drop=counter);
   Value=2000;
   do counter=1 to 20;
      Interest=value*.075;
      value+interest;
      Year+1;
   end;
run;
```

SAS Data Set Work.Earn

Value	Interest	Year
8495.70	592.723	20

The Sum statement Year+1 accumulates the number of iterations of the DO loop and stores thetotal in the new variable Year. The final value of Year is then stored in the data set, whereas the index variable counter is dropped. The data set has one observation.

Counting do loop iterations (2)

Explicit OUTPUT statements

- To create an observation for each iteration of the DO loop, place an OUTPUT statement inside the loop. By default, every DATA step contains an implicit OUTPUT statement at the end of the step. But placing an explicit OUTPUT statement in a DATA step overrides automatic output, causing SAS to add an observation to the data set only when the explicit OUTPUT statement is executed.
- The previous example created one observation because it used automatic output at the end of the DATA step. In the following example, the OUTPUT statement overrides automatic output, so the DATA step writes 20 observations.

Counting do loop iterations (3)

- Explicit OUTPUT statements
  - Example:

```
data work.earn;
   Value=2000;
   do Year=1 to 20;
      Interest=value*.075;
      value+interest;
      output;
   end;
run;
```

SAS Data Set Work.Earn (Partial Listing)

Value	Year	Interest
2150.00	1	150.000
2311.25	2	161.250
2484.59	3	173.344
2670.94	4	186.345
2871.26	5	200.320
3086.60	6	215.344
3318.10	7	231.495
3566.96	8	248.857
...	...	...
8495.70	20	592.723

Decrementing DO loops

Decrementing DO loops

- You can decrement a DO loop's index variable by specifying a negative value for the BY clause. For example, the specification in this iterative DO statement decreases the index variable by 1, resulting in values of *5*, *4*, *3*, *2*, and *1*.

```
DO index-variable=5 to 1 by -1;
   SAS statements
END;
```

- When you use a negative BY clause value, the start value must always be greater than the stop value in order to decrease the index variable during each iteration.

```
DO index-variable=5 to 1 by -1;
   SAS statements
END;
```

Specifying a series of items (1)

Specifying a series of items

- You can also specify how many times a DO loop executes by listing items in a series.
  - General form, DO loop with a variable list:

DO *index-variable=value1, value2, value3...;*

SAS statements

END;

where *values* can be character or numeric.

Specifying a series of items (2)

- When the DO loop executes, it executes once for each item in the series. The index variable equals the value of the current item. You must use commas to separate items in the series.
- To list items in a series, you must specify either all numeric values

```
DO index-variable=2,5,9,13,27;
   SAS statements
END;
```

- all character values, with each value enclosed in quotation marks

```
DO index-variable='MON','TUE','WED','THR','FRI';
   SAS statements
END;
```

- all variable names—the index variable takes on the values of the specified variables.

```
DO index-variable=win,place,show;
   SAS statements
END;
```

- Variable names must represent either all numeric or all character values. Do **not** enclose variable names in quotation marks.

Nesting DO Loops (1)

- Iterative DO statements can be executed within a DO loop. Putting a DO loop within a DO loop is called **nesting**.

```
do i=1 to 20;
   SAS statements
   do j=1 to 10;
      SAS statements
   end;
   SAS statements
end;
```

- The following DATA step computes the value of a one-year investment that earns 7.5% annual interest, compounded monthly.

```
data work.earn;
   Capital=2000;
   do month=1 to 12;
      Interest=capital*(.075/12);
      capital+interest;
   end;
run;
```

Nesting DO Loops (2)

- Let's assume the same amount of capital is to be added to the investment each year for 20 years. The new program must perform the calculation for each month during each of the 20 years. To do this, you can include the monthly calculations within another DO loop that executes 20 times.

```
data work.earn;
     do year=1 to 20;
        Capital+2000;
        do month=1 to 12;
           Interest=capital*(.075/12);
           capital+interest;
        end;
     end;
run;
```

Nesting DO Loops (3)

- During each iteration of the outside DO loop, an additional 2,000 is added to the capital, and the nested DO loop executes 12 times.

```
data work.earn;
   do year=1 to 20;
      Capital+2000;
      do month=1 to 12;
         Interest=capital*(.075/12);
         capital+interest;
      end;
   end;
run;
```

Note: It is easier to manage nested DO loops if you indent the statements in each DO loop as shown above.

Iteratively Processing Data That Is Read from a Data Set (1)

- So far you have seen examples of DATA steps that use DO loops to generate one or more observations from one iteration of the DATA step. Now let's look at a DATA step that reads a data set to compute the value of a new variable.

- The SAS data set **Finance.CDRates**, shown below, contains interest rates for certificates of deposit (CDs) that are available from several institutions.

Iteratively Processing Data That Is Read from a Data Set (2)

SAS Data Set Finance.CDRates

Institution	Rate	Years
MBNA America	0.0817	5
Metropolitan Bank	0.0814	3
Standard Pacific	0.0806	4

- Suppose you want to compare how much each CD will earn at maturity with an investment of $5,000. The DATA step below creates a new data set, **Work.Compare**, that contains the added variable, Investment.

```
data work.compare(drop=i);
     set finance.cdrates;
     Investment=5000;
     do i=1 to years;
        investment+rate*investment;
     end;
run;
```

Iteratively Processing Data That Is Read from a Data Set (3)

- The index variable is used only to execute the DO loop, so it is dropped from the new data set. Notice that the data set variable Years is used as the stop value in the iterative DO statement. As a result, the DO loop executes the number of times that are specified by the current value of Years. During the first iteration of the DATA step, for example, the DO loop executes five times.
- During each iteration of the DATA step,
  - an observation is read from **Finance.CDRates**
  - the value *5000* is assigned to the variable Investment
  - the DO loop executes, based on the current value of Years
  - the value of Investment is computed (each time that the DO loop executes), using the current value of Rate.

Iteratively Processing Data That Is Read from a Data Set (4)

- At the bottom of the DATA step, the first observation is written to the **Work.Compare** data set. Control returns to the top of the DATA step, and the next observation is read from **Finance.CDRates**. These steps are repeated for each observation in **Finance.CDRates**. The resulting data set contains the computed values of Investment for all observations that have been read from **Finance.CDRates**.

SAS Data Set Work.Compare

Institution	Rate	Years	Investment
MBNA America	0.0817	5	7404.64
Metropolitan Bank	0.0814	3	6323.09
Standard Pacific	0.0806	4	6817.57

Conditionally Executing DO Loops (1)

- The iterative DO statement requires that you specify the number of iterations for the DO loop. However, there are times when you want to execute a DO loop until a condition is reached or while a condition exists, but you don't know how many iterations are needed.
- Suppose you want to calculate the number of years that are required for an investment to reach $50,000. In the DATA step below, using an iterative DO statement is inappropriate because you are trying to determine the number of iterations required for Capital to reach $50,000.

```
data work.invest;
   do year=1 to ? ;
      Capital+2000;
      capital+capital*.10;
   end;
run;
```

Conditionally Executing DO Loops (2)

Using the DO UNTIL statement

- The DO UNTIL statement executes a DO loop **until** the expression is true.
  - General form, DO UNTIL statement:

DO UNTIL(*expression*);
  more SAS statements
END;

where *expression* is a valid SAS expression enclosed in parentheses.

Conditionally Executing DO Loops (3)

- The expression is not evaluated until the bottom of the loop, so a DO UNTIL loop always executes at least once. When the expression is evaluated as true, the DO loop is not executed again.
- Assume you want to know how many years it will take to earn $50,000 if you deposit $2,000 each year into an account that earns 10% interest. The DATA step that follows uses a DO UNTIL statement to perform the calculation until the value is reached. Each iteration of the DO loop represents one year of earning.

```
data work.invest;
    do until(Capital>=50000);
        capital+2000;
        capital+capital*.10;
        Year+1;
    end;
run;
```

Conditionally Executing DO Loops (4)

- During each iteration of the DO loop,
  - *2000* is added to the value of Capital to reflect the annual deposit of $2,000
  - the value of Capital with 10% interest is calculated
  - the value of Year is incremented by 1.

Because there is no index variable in the DO UNTIL statement, the variable Year is created in a Sum statement to count the number of iterations of the DO loop. This program produces a data set that contains the single observation shown below. To accumulate more than $50,000 in capital requires 13 years (and 13 iterations of the DO loop).

SAS Data Set Work.Invest

Capital	Year
53949.97	13

Conditionally Executing DO Loops (5)

Using the DO WHILE statement

- Like the DO UNTIL statement, the DO WHILE statement executes DO loops conditionally. You can use the DO WHILE statement to execute a DO loop **while** the expression is true.
  - General form, DO WHILE statement:

DO WHILE (*expression*);

more SAS statements

END;

where *expression* is a valid SAS expression enclosed in parentheses.

Conditionally Executing DO Loops (6)

An important difference between the DO UNTIL and DO WHILE statements is that the DO WHILE expression is evaluated at the top of the DO loop. If the expression is false the first time it is evaluated, then the DO loop never executes. For example, in the following program, if the value of Capital is less than 50,000, the DO loop does not execute.

```
data work.invest;
   do while(Capital>=50000);
       capital+2000;
       capital+capital*.10;
       Year+1;
   end;
run;
```

Using Conditional Clauses with the Iterative DO Statement (1)

You have seen how the DO WHILE and DO UNTIL statements enable you to execute statements conditionally and how the iterative DO statement enables you to execute statements a set number of times, unconditionally.

- DO WHILE(*expression*);
- DO UNTIL(*expression*);
- DO *index-variable=start* TO *stop* BY *increment;*

Using Conditional Clauses with the Iterative DO Statement (2)

- Now let's look at a form of the iterative DO statement that combines features of both conditional and unconditional execution of DO loops.
- In this DATA step, the DO UNTIL statement determines how many years it takes (13) for an investment to reach $50,000.

```
data work.invest;
   do until(Capital>=50000);
      Year+1;
      capital+2000;
      capital+capital*.10;
   end;
run;
```

SAS Data Set Work.Invest

Capital	Year
53949.97	13

Using Conditional Clauses with the Iterative DO Statement (3)

Suppose you also want to limit the number of years that you invest your capital to 10 years. You can add the UNTIL or WHILE expression to an iterative DO statement to further control the number of iterations. This iterative DO statement enables you to execute the DO loop until Capital is greater than or equal to *50000* or until the DO loop executes 10 times, whichever occurs first.

```
data work.invest(drop=i);
   do i=1 to 10 until(Capital>=50000);
      Year+1;
      capital+2000;
      capital+capital*.10;
   end;
run;
```

SAS Data Set Work.Invest

Capital	Year
35062.33	10

Using Conditional Clauses with the Iterative DO Statement (4)

- In this case, the DO loop stops executing after 10 iterations, and the value of Capital never reaches *50000*. If you increase the amount added to Capital each year to *4000*, the DO loop stops executing after the eighth iteration when the value of Capital exceeds *50000*.

```
data work.invest(drop=i);
   do i=1 to 10 until(Capital>=50000);
      Year+1;
      capital+4000;
      capital+capital*.10;
   end;
run;
```

SAS Data Set Work.Invest

Capital	Year
50317.91	8

Using Conditional Clauses with the Iterative DO Statement (5)

- The UNTIL and WHILE specifications in an iterative DO statement function similarly to the DO UNTIL and DO WHILE statements. Both statements require a valid SAS expression enclosed in parentheses.
  - UNTIL(*expression*);
  - **DO *index-variable=start* TO *stop* BY *increment***
    WHILE(*expression*);
- The UNTIL expression is evaluated at the **bottom** of the DO loop, so the DO loop always executes at least once. The WHILE expression is evaluated **before** the execution of the DO loop. So, if the condition is not true, the DO loop never executes.

Creating Samples (1)

- Because it performs iterative processing, a DO loop provides an easy way to draw sample observations from a data set. For example, suppose you would like to sample every tenth observation of the 5,000 observations in **Factory.Widgets**. Start with a simple DATA step:

```
data work.subset;
   set factory.widgets;
run;
```

- You can create the sample data set by enclosing the SET statement in a DO loop. Use the start, stop, and increment values to select every tenth observation of the 5,000. Add the POINT= option to the SET statement, setting the POINT= option equal to the index variable that is used in the DO loop.

Creating Samples (2)

- Example:

```
data work.subset;
   do sample=10 to 5000 by 10;
      set factory.widgets point=sample;
   end;
run;
```

- Remember that, in order to prevent continuous DATA step looping, you need to add a STOP statement when using the POINT= option. Then, because the STOP statement prevents the output of observations at the end of the DATA step, you also need to add an OUTPUT statement. Place the statement inside the DO loop in order to output each observation that is selected. (If the OUTPUT statement were placed after the DO loop, only the last observation would be written.)

Creating Samples (3)

```
data work.subset;
   do sample=10 to 5000 by 10;
      set factory.widgets point=sample;
      output;
   end;
   stop;
run;
```

- When the program runs, the DATA step reads the observations that are identified by the POINT=option in **Factory.Widgets**. The values of the POINT= option are provided by the DO loop, which starts at 10 and goes to 5,000 in increments of 10. The data set **Work.Subset** contains 500 observations.

Processing Variables with Arrays

Contents

Introduction (1)

An **array** is a temporary grouping of variables under a single name. This can reduce the number of statements that are needed to process variables and can simplify the maintenance of DATA step programs.

In DATA step programming, you often need to perform the same action on more than one variable. Although you can process variables individually, it is easier to handle them as a group. You can do this by using array processing.

Introduction (2)

For example, the program below eliminates the need for 365 separate programming statements to convert the daily temperature from Fahrenheit to Celsius for the year:

```
data work.report (drop=i);
   set master.temps;
   array daytemp{365} day1-day365;
   do i=1 to 365;
      daytemp{i}=5*(daytemp{i}-32)/9;
   end;
run;
```

Introduction (3)

- One reason for using an array
  --- reduce the number of statements:

```
data work.report;
   set master.temps;
   mon=5*(mon-32)/9;
   tue=5*(tue-32)/9;
   wed=5*(wed-32)/9;
   thr=5*(thr-32)/9;
   fri=5*(fri-32)/9;
   sat=5*(sat-32)/9;
   sun=5*(sun-32)/9;
run;
```

With array:

```
data work.report(drop=i);
   set master.temps;
   array wkday{7} mon tue wed thr fri sat sun;
   do i=1 to 7;
      wkday{i}=5*(wkday{i}-32)/9;
   end;
run;
```

The same calculation is performed on each variable.

Use fewer statements
Easier to be modified or corrected

Defining an Array (1)

General form of an array

ARRAY *array-name{dimension} <elements>;*

```
array daytemp{365} day1-day365;
```

- Arrays exist only for the duration of the DATA step. They do not become part of the output data set.
- Do not give an array the same name as a variable in the same DATA step.
- Array elements must be either all numeric or all character.
- If no elements are listed, new variables will be created with default names.
- You cannot use array names in LABEL, FORMAT, DROP, KEEP, or LENGTH statements.

Defining an Array (2)

Dimension

- The **dimension** describes the number and arrangement of **elements** in the array. There are several ways to specify the dimension:

  - the number of array elements

    ```
    array sales{4} qtr1 qtr2 qtr3 qtr4;
    ```

  - a range of values

    ```
    array sales{96:99} totals96 totals97 totals98 totals99;
    ```

  - using （*）, the dimension is determined by counting the number of elements.

    ```
    array sales{*} qtr1 qtr2 qtr3 qtr4;
    ```

Defining an Array (3)

Elements

- Specifying array **elements**
  - list each variable name

```
array sales{4} qtr1 qtr2 qtr3 qtr4;
```

  - specify array elements as a variable list.

```
array sales{4} qtr1-qtr4;
```

    - Let's look more closely at array elements that are specified as variable lists. It has several forms.

Variable List as array Elements

- a numbered range of variables: `Var1-Varn`

```
array sales{4} qtr1-qtr4;
```

  - must have the same name except for the last character
  - the last character must be numeric
  - must be numbered consecutively.

- all numeric variables: `_NUMERIC_`

```
array sales{*} _numeric_;
```

- all character variables: `_CHARACTER_`

```
array sales{*} _character_;
```

- all variables: `_ALL_`

```
array sales{*} _all_;
```

Array Reference (1)

Overview

➤ When you define an array in a DATA step, an index value is assigned to each element. During execution, you can use an **array reference** to perform actions on specific array elements. When used in a DO loop, for example, the index variable of the iterative DO statement can reference each element of the array.

Array Reference (2)

- General form of ARRAY reference:

  array-name{index value}

where index value

- is enclosed in parentheses, braces, or brackets
- specifies an integer, a numeric variable, or a SAS numeric expression
- is within the lower and upper bounds of the dimension of the array.

Array Reference (3)

- Examples
  - reference the elements of an array by an **index value**

```
data work.report(drop=i);
  set master.temps;
  array wkday{7} mon tue wed thr fri sat sun;
  do i=1 to 7;
    if wkday{i}>95 then output;
  end;
run;
```

Typically, arrays are used with **DO loops**.

  - The index values are assigned in the order of the array elements.

```
                     1   2   3   4
array quarter{4} jan apr jul oct;
do i=1 to 4;
  YearGoal=quarter{i}*1.2;
end;
```

Array Reference (4)

Compilation and execution (1)

- An example – Kilograms to be converted to pounds:

SAS Data Set Hrd.Fitclass

Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6
Alicia	69.6	68.9	68.8	67.4	66.0	66.2
Betsy	52.6	52.6	51.7	50.4	49.8	49.1
Brenda	68.6	67.6	67.0	66.4	65.8	65.2
Carl	67.6	66.6	66.0	65.4	64.8	64.2
Carmela	63.6	62.5	61.9	61.4	60.8	58.2
David	70.6	69.8	69.2	68.4	67.8	67.0

```
data hrd.convert;
  set hrd.fitclass;
  array wt{6} weight1-weight6;
  do i=1 to 6;
    wt{i}=wt{i}*2.2046;
  end;
run;
```

Array Reference (5)

Compilation and execution (2)

- An example – Kilograms to be converted to pounds:

Program Data Vector

_N_	Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6	i

- The program data vector is created for the **Hrd.Convert** data set.
- The index values of the array elements are assigned.
- Note that the array name and the array references are not included in the program data vector.

Program Data Vector

		wt{1}	wt{2}	wt{3}	wt{4}	wt{5}	wt{6}	
_N_	Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6	i
1	Alicia	69.6	68.9	68.8	67.4	66.0	66.2	

- The first observation is read into the program data vector.
- Because the ARRAY statement is a compile-time only statement, it is ignored during execution. The DO loop is executed next.

Array Reference (6)

Compilation and execution (3)

- An example – Kilograms to be converted to pounds:
  - Because `wt{1}` refers to the first array element, `Weight1`, the value of `Weight1` is converted from kilograms to pounds.

Program Data Vector

		wt{1}	wt{2}	wt{3}	wt{4}	wt{5}	wt{6}	
_N_	Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6	i
1	Alicia	153.4	68.9	68.8	67.4	66.0	66.2	1

  - Continues its DO loop iterations,
  - The index variable `i` is changed from 1 to 6, causing `Weight2` through `Weight6` to receive new values in the program data vector, as shown below.

Program Data Vector

		wt(1)	wt(2)	wt(3)	wt(4)	wt(5)	wt(6)	
_N_	Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6	i
1	Alicia	153.4	151.9	151.7	148.6	145.5	145.9	6

The DIM Function

You can also use the DIM function to return the number of elements in the array.

When you use the DIM function, you do not have to re-specify the stop value of a DO statement if you change the dimension of the array.

- General form of DIM function:

  DIM(*array-name*)

```
array wt{*} weight1-weight6;
do i=1 to dim(wt);
   wt{i}=wt{i}*2.2046;
end;
```

- `dim(wt)` returns a value of *6*.
- *array-name* specifies the array.

Quiz

Which DO statement would not process all the elements in the factors array shown below?

```
array factors{*} age height weight bloodpr;
```

a. do i=1 to dim(factors);
b. do i=1 to dim(*);
c. do i=1,2,3,4;
d. do i=1 to 4;

- Correct answer: b
  - To process all the elements in an array, you can either specify the array dimension or use the DIM function with the array name as the argument.

Creating Variables with Array (1)

Overview

- You can also **create** variables in an ARRAY statement by omitting the array elements from the statement.

- Because you are not referencing existing variables, SAS automatically creates the variables for you and assigns default names to them.

Creating Variables with Array (2)

General form (1)

- General form of ARRAY statement to create new variables:

  ARRAY *array-name***{***dimension***};**

  - where *array-name* specifies the name of the array.
  - *dimension* describes the number and arrangement of array elements. The default dimension is one.

Creating Variables with Array (3)

General form (2)

In creating variables in array statement, you can

- Creates default variable names
  - concatenating the array name and the numbers 1, 2, 3, and so on, up to the array dimension

```
array WgtDiff{5};
Variables created: WgtDiff1 WgtDiff2 WgtDiff3 WgtDiff4 WgtDiff5
```

- Specify individual variable names
  - list each name as an element of the array.

```
array WgtDiff{5} Oct12 Oct19 Oct26 Nov02 Nov09;
Variables created: Oct12 Oct19 Oct26 Nov02 Nov09
```

Creating Variables with Array (4)

- General form (3)

  In creating variables in array statement, you can

  - Creating arrays of character variables
    - To create an array of **character** variables, add a dollar sign（$）after the array dimension.

      ```
      array firstname{5} $;
      ```

    - By default, all character variables that are created in an ARRAY statement are assigned a length of 8. You can assign your own length by specifying the **length** after the dollar sign..

      ```
      array firstname{5} $ 24;
      ```

Creating Variables with Array (5)

Example

- Create variables that contain this weekly difference
  - group the variables **Weight1** through **Weight6** into an array
  - create the new variables to store the differences.
    - use an additional ARRAY statement without elements to create the new variables.
  - use a DO loop to calculate the differences between each of the recorded weights.

SAS Data Set Hrd.Convert

Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6
Alicia	153.4	151.9	151.7	148.6	145.5	145.9
Betsy	116.0	116.0	114.0	111.1	109.8	108.2
Brenda	151.2	149.0	147.7	146.4	145.1	143.7
Carl	149.0	146.8	145.5	144.2	142.9	141.5
Carmela	140.2	137.8	136.5	135.4	134.0	128.3

```
data hrd.diff;
   set hrd.convert;
   array wt{6} weight1-weight6;
   array WgtDiff{5};
   do i=1 to 5;
      wgtdiff{i}=wt{i+1}-wt{i};
   end;
run;
```

SAS Data Set Hrd.Diff

Name	WgtDiff1	WgtDiff2	WgtDiff3	WgtDiff4	WgtDiff5
Alicia	-1.54322	-0.22046	-3.08644	-3.08644	0.44092
Betsy	0.00000	-1.98414	-2.86598	-1.32276	-1.54322
Brenda	-2.20460	-1.32276	-1.32276	-1.32276	-1.32276

(A portion of the resulting data set)

Creating Variables with Array (6)

Compilation

► During the **compilation** of the DATA step, the variables that this ARRAY statement creates are added to the vector.

Program Data Vector

_N_	Name	Weight1	Weight2	Weight3	Weight4	Weight5	Weight6

► Be careful not to confuse the array references **`WgtDiff{1}`** through **`WgtDiff{5}`** (note the braces) with the variable names **`WgtDiff1`** through **`WgtDiff5`**. Below shows the relationship.

WgtDiff{1}	WgtDiff{2}	WgtDiff{3}	WgtDiff{4}	WgtDiff{5}
WgtDiff1	WgtDiff2	WgtDiff3	WgtDiff4	WgtDiff5

Assigning Initial Values to Array (1)

General form

- Assign **initial values** to elements of an array when you define the array.

```
array goal{4} g1 g2 g3 g4 (initial values);
array goal{4} g1 g2 g3 g4 (9000 9300 9600 9900);
```

  - place the values after the array elements
  - specify one initial value for each corresponding array element
  - separate each value with a comma or blank
  - enclose the initial values in parentheses.

Assigning Initial Values to Array (2)

Examples (1)

- Enclose each character value in quotation marks.

```
array col{3} $ color1-color3 ('red','green','blue');
```

- Assign initial values without specifying array element.

```
array Var{4} (1 2 3 4);
```

  - It creates the variables **Var1, Var2, Var3**, and **Var4**, and assigns them initial values of *1*, *2*, *3*, and *4*:

Assigning Initial Values to Array (3)

Examples (2)

► To compare the actual sales figures to the goals. The actual sales are stored in Finance.Qsales while the goals are not recorded in.

```
data finance.report(drop=i);
  set finance.qsales;
  array sale{4} sales1-sales4;
  array Goal{4} (9000 9300 9600 9900);
  array Achieved{4};
  do i=1 to 4;
    achieved{i}=100*sale{i}/goal{i};
  end;
run;
```

SAS Data Set Finance.Report

SalesRep	Sales1	Sales2	Sales3	Sales4	Goal1	Goal2	Goal3	Goal4	Achieved1	Achieved2	Achieved3	Achieved4
Britt	8400	8800	9300	9800	9000	9300	9600	9900	93.333	94.624	96.875	98.990
Fruchten	9500	9300	9800	8900	9000	9300	9600	9900	105.556	100.000	102.083	89.899
Goodyear	9150	9200	9650	11000	9000	9300	9600	9900	101.667	98.925	100.521	111.111

Variables to which initial values are assigned in an ARRAY statement are automatically retained.

Creating Temporary Array Elements

How to eliminate **`Goal1`** through **`Goal4`** as they are not needed in the previous example? Here Temporary Array Elements comes in.

```
data finance.report;
  set finance.qsales;
  array sale{4} sales1-sales4;
  array goal{4} _temporary_ (9000 9300 9600 9900);
  array Achieved{4};
  do i=1 to 4;
    achieved{i}=100*sale{i}/goal{i};
  end;
run;
```

SAS Data Set Finance.Report

SalesRep	Sales1	Sales2	Sales3	Sales4	Achieved1	Achieved2	Achieved3	Achieved4
Britt	8400	8800	9300	9800	93.333	94.624	96.875	98.990
Fruchten	9500	9300	9800	8900	105.556	100.000	102.083	89.899
Goodyear	9150	9200	9650	11000	101.667	98.925	100.521	111.111

Multidimensional Arrays (1)

- To define a multidimensional array, you specify the number of elements in each dimension, separated by a comma.

```
array new{3,4} x1-x12;
```

	columns			
rows	x1	x2	x3	x4
	x5	x6	x7	x8
	x9	x10	x11	x12

  - The first dimension in the ARRAY statement specifies the number of rows.
  - The second dimension specifies the number of columns.

- Reference any element of the array by specifying the two dimensions.

```
array new{3,4} x1-x12;
new{2,3}=0;
```

x1	x2	x3	x4
x5	x6	x7	x8
x9	x10	x11	x12

Multidimensional Arrays (2)

When you define a two-dimensional array, the array elements are grouped in the order in which they are listed in the ARRAY statement.

```
array new{3,4} x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12;
```

- The array elements x1 through x4 can be thought of as the first row of the table.

x1	x2	x3	x4
x5	x6	x7	x8
x9	x10	x11	x12

- The elements x5 through x8 become the second row of the table, and so on.

x1	x2	x3	x4
x5	x6	x7	x8
x9	x10	x11	x12

Referencing Elements of a Two-Dimensional Array (1)

- Multidimensional arrays are typically used with nested DO loops.

- If a DO loop processes a two dimensional array, you can reference any element within the array by specifying the two dimensions.

Referencing Elements of a Two-Dimensional Array (2)

An example (1):

- A company's sales figures are stored by month (Finance.Monthly). Your task is to generate a new data set of quarterly sales rather than monthly sales.

Raw data set:
Sales by month

Description of Finance.Monthly

Variable	Type	Length
Year	num	8
Month1	num	8
Month2	num	8
Month3	num	8
Month4	num	8
Month5	num	8
Month6	num	8
Month7	num	8
Month8	num	8
Month9	num	8
Month10	num	8
Month11	num	8
Month12	num	8

Task:
Sales by quarter

SAS Data Set Finance.Quarters (Partial Listing)

Year	Qtr1	Qtr2	Qtr3	Qtr4
1989	69100	64400	69200	71800
1990	73100	72000	83200	82800
1991	73400	81800	85200	87800

Referencing Elements of a Two-Dimensional Array (3)

An example (2):

1. Defining the array **m{4,3}** puts the variables **Month1** through **Month12** into four groups of three months (yearly quarters).

Table Representation of m Array

Month1	Month2	Month3
Month4	Month5	Month6
Month7	Month8	Month9
Month10	Month11	Month12

```
Data inance.quarters(drop=i j);
  set finance.monthly;
  array m{4,3} month1-month12;
  array Qtr{4};
  do i=1 to 4;
    qtr{i}=0;
    do j=1 to 3;
      qtr{i}+m{i,j};
    end;
  end;
run;
```

Referencing Elements of a Two-Dimensional Array (4)

An example (3):

2. Defining the array **Qtr{4}** creates the numeric variables **Qtr1, Qtr2, Qtr3, Qtr4**, which will be used to sum the sales for each quarter.

```
Data inance.quarters(drop=i j);
  set finance.monthly;
  array m{4,3} month1-month12;
  array Qtr{4};
  do i=1 to 4;
    qtr{i}=0;
    do j=1 to 3;
      qtr{i}+m{i,j};
    end;
  end;
run;
```

Referencing Elements of a Two-Dimensional Array (5)

An example (4):

3. A nested DO loop is used to reference the values of the variables **Month1 through Month12** and to calculate the values of **Qtr1 through Qtr4**.

```
Data inance.quarters(drop=i j);
   set finance.monthly;
   array m{4,3} month1-month12;
   array Qtr{4};
   do i=1 to 4;
      qtr{i}=0;
      do j=1 to 3;
         qtr{i}+m{i,j};
      end;
   end;
run;
```

To see how the nested DO loop processes these arrays, let's examine the execution of this DATA step.

Referencing Elements of a Two-Dimensional Array (6)

Steps of Execution (1):

- When this DATA step is compiled, the vector is created.
- The PDV contains the variables **`Year, Month1`** through **`Month12`**, and the new variables **`Qtr1`** through **`Qtr4`**.

Program Data Vector

_N_	Year	Month1	Month2	Month3	Qtr1	Qtr2	Qtr3	Qtr4	i	j
•	•	•	•	•	•	•	•	•	•	•

- In the first execution of the DATA step, the 1st observation of **Finance.Monthly** are read into the program data vector.

Program Data Vector

_N_	Year	Month1	Month2	Month3	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	1989	23000	21500	24600	•	•	•	•	1	•

Referencing Elements of a Two-Dimensional Array (7)

Steps of Execution (2):

- During the first iteration of the nested DO loop, the value of **`Month1`**, which is referenced by **`m{i,j}`**, is added to **`Qtr1`**.

Program Data Vector

_N_	Year	Month1	Month2	Month3	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	1989	23000	21500	24600	23000	•	•	•	1	1

- During the second iteration of the nested DO loop, the value of **`Month2`**, which is referenced by **`m{i,j}`**, is added to **`Qtr1`**.

Program Data Vector

_N_	Year	Month1	Month2	Month3	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	1989	23000	21500	24600	44500	•	•	•	1	2

Referencing Elements of a Two-Dimensional Array (8)

Steps of Execution (3):

- The nested DO loop continues to execute until the index variable **`j`** **exceeds** the stop value, *3*.

- When the nested DO loop completes execution, the total sales for the first quarter, **`Qtr1`**, have been computed.

Program Data Vector

_N_	Year	Month1	Month2	Month3	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	1989	23000	21500	24600	69100	•	•	•	1	4

Referencing Elements of a Two-Dimensional Array (9)

Steps of Execution (4):

- The outer DO loop increments **`i`** to *2*, and the process continues for the array element **`Qtr2`** and the **`m`** array elements **`Month4`** through **`Month6`**.

Program Data Vector

_N_	Month2	Month3	Month4	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	21500	24600	23300	69100	23300	.	.	2	1

- After the outer DO loop completes execution, the end of the DATA step is reached. The variable values for the first observation are written to the data set Finance.Quarters.

Program Data Vector

_N_	Month2	Month3	Month4	Qtr1	Qtr2	Qtr3	Qtr4	i	j
1	21500	24600	23300	69100	23300	69200	71800	5	4

Referencing Elements of a Two-Dimensional Array (10)

Steps of Execution (5):

- All observations in the data set **Finance.Monthly** are processed in the same manner.
- Below is a portion of the resulting data set, which contains the sales figures grouped by quarters.

SAS Data Set Finance.Quarters (Partial Listing)

Year	Qtr1	Qtr2	Qtr3	Qtr4
1989	69100	64400	69200	71800
1990	73100	72000	83200	82800
1991	73400	81800	85200	87800

Quiz

Based on the ARRAY statement below, select the array reference for the array element q50.

```
array ques{3,25} q1-q75;
```

a. ques{q50}

b. ques{1,50}

c. ques{2,25}

d. ques{3,0}

- Correct answer: c
  - This two-dimensional array would consist of three rows of 25 elements. The first row would contain q1 through q25, the second row would start with q26 and end with q50, and the third row would start with q51 and end with q75.

Rotating Data Sets (1)

- We've seen a number of uses for arrays, including creating variables, performing repetitive calculations, and performing table lookups. We can also use arrays for rotating (transposing) a SAS data set.

- When we rotate a SAS data set, we change variables to observations or observations to variables.

Rotating Data Sets (2)

Example:

Rotate the Finance.Funddrive data set to create four output observations from each input observation.

SAS Data Set Finance.Funddrive

LastName	Qtr1	Qtr2	Qtr3	Qtr4
ADAMS	18	18	20	20
ALEXANDE	15	18	15	10
APPLE	25	25	25	25
ARTHUR	10	25	20	30
AVERY	15	15	15	15
BAREFOOT	20	20	20	20
BAUCOM	25	20	20	30
BLAIR	10	10	5	10
BLALOCK	5	10	10	15
BOSTIC	20	25	30	25
BRADLEY	12	16	14	18
BRADY	20	20	20	20
BROWN	18	18	18	18
BRYANT	16	18	20	18
BURNETTE	10	10	10	10
CHEUNG	30	30	30	30
LEHMAN	20	20	20	20
VALADEZ	14	18	40	25

Rotating Data Sets (3)

Example:

The following program rotates the data set and lists the first 16 observations in the new data set.

```
data work.rotate(drop=qtr1-qtr4);
  set finance.funddrive;
  array contrib{4} qtr1-qtr4;
  do Qtr=1 to 4;
    Amount=contrib{qtr};
    output;
  end;
run;
proc print data=rotate(obs=16)
noobs;
run;
```

LastName	Qtr	Amount
ADAMS	1	18
ADAMS	2	18
ADAMS	3	20
ADAMS	4	20
ALEXANDER	1	15
ALEXANDER	2	18
ALEXANDER	3	15
ALEXANDER	4	10
APPLE	1	25
APPLE	2	25
APPLE	3	25
APPLE	4	25
ARTHUR	1	10
ARTHUR	2	25
ARTHUR	3	20
ARTHUR	4	30

Points to Remember (1)

- A SAS array exists only for the duration of the DATA step.

- Do not give an array the same name as a variable in the same DATA step. Also, avoid using the name of a SAS function as an array name—the array will be correct, but you won't be able to use the function in the same DATA step, and a warning will be written to the SAS log.

Points to Remember (2)

- You can indicate the dimension of a one-dimensional array with an asterisk (*) as long as you specify the elements of the array.

- When referencing array elements, be careful not to confuse variable names with the array references. WgtDiff1 through WgtDiff5 is not the same as WgtDiff{1} through WgtDiff{5}.

Applications - Examples

This section introduces some array applications in

- Data manipulations from data search - Example 1
- Count consecutive days - Example 2
- LOCF - Example 3
- Find and replace - Example 4
- Shift - Example 5

Leading to a more complicated efficient process.

Example 1 - Search Specified Value

- The example is to find on which day the maximum efficacy is reached.
- The algorithm is to compare the target value against an array and perform an action if the target value is found in the array.

SUBJECT	DAY1	DAY2	DAY3	DAY4	TMAX
101	0.0	0.0	0.0	0.0	.
102	.	0.5	0.5	0.0	2
106	0.5	0.0	0.0	0.0	1
107	0.5	2.0	0.5	2.0	2
111	1.0	3.0	2.0	2.5	2
112	2.0	3.0	2.5	3.5	4

```
data pd2;
  set pd1;
  array days[4] day1-day4;
  maxscore=max (of days [*]);
  do i=1 to dim(days);
      if maxscore >0 and
              days[i]=maxscore then do;
          tmax=i;
          return;
      end;
  end;
  drop i maxscore;
run;
```

Example 2 - Count Consecutive Days (1)

SUBJID	DATE	DATECNT
1	25MAR2004	1
1	26MAR2004	2
1	27MAR2004	3
1	28MAR2004	4
1	29MAR2004	5
2	26MAR2004	1
2	27MAR2004	2
2	29MAR2004	3
3	26MAR2004	1
3	27MAR2004	2
3	28MAR2004	3
3	02APR2004	4
4	02APR2004	1

- We check to see whether a subject has experienced night awakening for more than 3 consecutive days.
- From the DIARY data set and program below, we can easily list the subjects and their consecutive days along with start date and stop date by using array.

Target dataset

SUBJID	count	f_date	l_date
1	5	25MAR2004	29MAR2004
3	3	26MAR2004	28MAR2004

Example 2 - Count Consecutive Days (2)

Step1. Transpose

Temp1

SUBJID	***_NAME_***	***_dat1***	***_dat2***	***_dat3***	***_dat4***	***_dat5***
1	date	25MAR2004	26MAR2004	27MAR2004	28MAR2004	29MAR2004
2	date	26MAR2004	27MAR2004	29MAR2004	.	.
3	date	26MAR2004	27MAR2004	28MAR2004	02APR2004	.
4	date	02APR2004	.	.	.	.

```
proc transpose data=diary prefix=_dat out=temp1;
    by subjid;
    var date;
run;
```

Example 2 - Count Consecutive Days (3)

Step2. Count Consecutive Days using Array

```
data temp2
   (keep=subjid flag count I rename=(i=datecnt));
   set temp1 ;
   array dates {*} _dat: dummy ;
   retain flag count 1;
   do i=1 to dim(dates)-1;
     if dates[i]^=. then do;
       if dates[i] = dates[i+1]-1 then do;
         output;  count=count + 1;
       end;
       else do;
         output;  flag =flag + 1;  count=1;
       end;
     end;
   end;
run;
```

Temp2

SUBJID	flag	count	datecnt
1	1	1	1
1	1	2	2
1	1	3	3
1	1	4	4
1	1	5	5
2	2	1	1
2	2	2	2
2	3	1	3
3	4	1	1
3	4	2	2
3	4	3	3
3	5	1	4
4	6	1	1

Example 2 - Count Consecutive Days (4)

Step3. Manipulation

```
data temp3;
   merge temp2 diary;
   by subjid datecnt;
run;
```

Temp3

SUBJID	flag	count	datecnt	date
1	1	1	1	25MAR2004
1	1	2	2	26MAR2004
1	1	3	3	27MAR2004
1	1	4	4	28MAR2004
1	1	5	5	29MAR2004
2	2	1	1	26MAR2004
2	2	2	2	27MAR2004
2	3	1	3	29MAR2004
3	4	1	1	26MAR2004
3	4	2	2	27MAR2004
3	4	3	3	28MAR2004
3	5	1	4	02APR2004
4	6	1	1	02APR2004

Example 2 - Count Consecutive Days (5)

Step3. Manipulation

```
data continue (where=(count >=3 ));
  /*3 consecutive days defined*/
  set temp3;
    by subjid flag;
    retain f_date ;
    if first.flag then f_date=date ;
    if last.flag then do;
        l_date=date ;
        output;
    end;
    keep subjid f_date l_date count;
    format f_date l_date date9. ;
run;
```

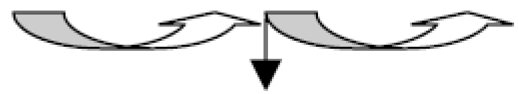
Target dataset

SUBJID	*count*	*f_date*	*l_date*
1	5	25MAR2004	29MAR2004
3	3	26MAR2004	28MAR2004

Example 3 - Data LOCF (1)

"LOCF" stands for "Last Observation Carried Forward", it means last non-missing value carried forward.

TIME1	TIME2	TIME3	TIME4	TIME5
A	B	.	.	E



TIME1	TIME2	TIME3	TIME4	TIME5
A	B	B	B	E

In LOCF analyses, when a patient drops out of a trial, the results of the last evaluation are carried forward as if the he had continued to the completion of the trial without further change.

Since patients who discontinue medication are regarded as treatment failures, LOCF analyses are widely considered to provide a more conservative test of drug effects.

Example 3 - Data LOCF (2)

The following data set called SCORE will be used as the example.

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
1	0.5	0.5	0.0	0.0	0.5	0.5
2	0.0	0.5	.	.	1.5	0.0
3	0.0	0.0	1.0	.	0.0	0.0
4	0.0	0.0	0.0	.	0.0	0.0
5	0.0	0.5	1.5	.	0.5	0.5
6	0.0	1.0	1.5	0.5	.	1.0

```
data locf ;
  set score ;
  array time [*] time: ;
  do i=1 to dim(time);
    if time[i]=. then time[i]=time[i-1];
  end;
  drop i makeup;
run;
```

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5
1	0.5	0.5	0.0	0.0	0.5
2	0.0	0.5	0.5	0.5	1.5
3	0.0	0.0	1.0	1.0	0.0
4	0.0	0.0	0.0	0.0	0.0
5	0.0	0.5	1.5	1.5	0.5
6	0.0	1.0	1.5	0.5	0.5

Example 4 - Find and Replace (1)

The array-implemented find& replace is exceptionally powerful and fast. The algorithm replaces elements referred to by iterator i in the array with new value when the condition holds, such as to find and replace the missing data. In some cases, the experiment measurements are not conducted continuously. They are discrete instead. To test the irritation of skins to patch or ointment as the example, the skin at different positions are supposed to be tested in the order of left arm, right arm, back, ... If one or more points somehow are skipped, the makeup tests would be done to get those data missed.

TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
A	B	.	D	E	C

Example 4 - Find and Replace (2)

```
data replace;
    set score;
    array apps [5] time1- time5;
    do i=1 to dim(apps);
        if apps[i] =. then apps[i]=makeup ;
    end;
    drop i ;
run;
```

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
1	0.5	0.5	0.0	0.0	0.5	0.5
2	0.0	0.5	.	.	1.5	0.0
3	0.0	0.0	1.0	.	0.0	0.0
4	0.0	0.0	0.0	.	0.0	0.0
5	0.0	0.5	1.5	.	0.5	0.5
6	0.0	1.0	1.5	0.5	.	1.0

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
1	0.5	0.5	0.0	0.0	0.5	0.5
2	0.0	0.5	0.0	0.0	1.5	0.0
3	0.0	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.5	1.5	0.5	0.5	0.5
6	0.0	1.0	1.5	0.5	1.0	1.0

Example 5 - Data Shift (1)

One subject should undergo a certain times of tests in some situations, and the time order should be kept, then a data shift process can be applied with the help of array.

TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
A	B	.	D	E	C

↓

TIME1	TIME2	TIME3	TIME4	TIME5
A	B	D	E	C

Example 5 - Data Shift (2)

```
data shift;
  set score;
  array apps[*] time: makeup;
  do i = 1 to dim(apps)-1;
    if apps[i] = . then do;
       do j = i to dim(apps)-1;
           apps[j] = apps[j+1];
       end;
       mu='-'||compress(i);
       if apps[i] = . then i=i-1;
    end;
  end;
  drop i j ;
run;
```

A do loop would be useful if there are more than one missing data in the rows.

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP
1	0.5	0.5	0.0	0.0	0.5	0.5
2	0.0	0.5	.	.	1.5	0.0
3	0.0	0.0	1.0	.	0.0	0.0
4	0.0	0.0	0.0	.	0.0	0.0
5	0.0	0.5	1.5	.	0.5	0.5
6	0.0	1.0	1.5	0.5	.	1.0

SUBJID	TIME1	TIME2	TIME3	TIME4	TIME5	MAKEUP	mu
1	0.5	0.5	0.0	0.0	0.5	0.5	
2	0.0	0.5	1.5	0.0	0.0	0.0	-3
3	0.0	0.0	1.0	0.0	0.0	0.0	-4
4	0.0	0.0	0.0	0.0	0.0	0.0	-4
5	0.0	0.5	1.5	0.5	0.5	0.5	-4
6	0.0	1.0	1.5	0.5	1.0	1.0	-5

Example 6 - Data Merge (1)

It is often required to merge dose data with other safety data, such as adverse events, vital signs, ECG, and lab results, and locate the dose-related safety profiles. For example, a patient is given several doses at certain time points. After each dose, some adverse events may occur to the patient. We need to know which adverse event is associated with which dose. Suppose there is a dose dataset and an adverse event dataset.

AE

SUBJID	***AE***	***AEDTTM***
1	NERVOUSNESS	30JAN1999:06:00:00
1	TACHYCARDIA	30JAN1999:12:15:00
1	NAUSEA	06FEB1999:16:20:00
1	DIZZINESS	20FEB1999:09:20:00
2	HEADACHE	06FEB1999:14:10:00
2	NAUSEA	06FEB1999:17:40:00

Dose

SUBJID	***DOSEN1***	***DOSEN2***	***DOSEN3***	***DOSEN4***
1	30JAN1999:08:00:00	06FEB1999:08:00:00	20FEB1999:08:00:00	27FEB1999:08:00:00
2	30JAN1999:08:01:00	06FEB1999:08:01:00	20FEB1999:08:01:00	06MAR1999:08:01:00
3	30JAN1999:08:02:00	06FEB1999:08:02:00	13FEB1999:08:02:00	27FEB1999:08:02:00

Example 6 - Data Merge (2)

SUBJID	AEDTTM	AE	dosedttm	dosenum	hrpostds
1	30JAN99:12:15:00	TACHYCARDIA	30JAN1999:08:00:00	1	4.3
1	06FEB99:16:20:00	NAUSEA	06FEB1999:08:00:00	2	8.3
1	20FEB99:09:20:00	DIZZINESS	20FEB1999:08:00:00	3	1.3
2	06FEB99:14:10:00	HEADACHE	06FEB1999:08:01:00	2	6.2
2	06FEB99:17:40:00	NAUSEA	06FEB1999:08:01:00	2	9.7

```
data dose_ae;
    merge dose ae;
    by subjid;
    array dosen {*} dosen:;
    do i=1 to dim(dosen);
        if dosen[i]^=. and aedttm > dosen[i] then do;
            dosedttm = dosen[i];            dosenum = i;
        end;
    end;
    if dosenum^=.;
    hrpostds = round(((aedttm-dosedttm)/3600), 0.1);
    format dosedttm datetime20.;
    drop i dosen1 - dosen4;
run;
```

The final result is shown above, variable DOSENUM is the order number of doses, and HRPOSTDS is time in hours after dosing.