Chapter 4
Working with Matrices

Chapter Table of Contents

Chapter 4
Working with Matrices

Overview

SAS/IML software provides many ways to create matrices. You can create matrices by doing any of the following:

- entering data yourself as a matrix literal
- using assignment statements
- using matrix-generating functions
- creating submatrices from existing matrices with subscripts
- using SAS data sets (see Chapter 6, "Working with SAS Data Sets," for more information)

Once you have defined matrices, you have access to many operators and functions for working on them in matrix expressions. These operators and functions facilitate programming and make referring to submatrices efficient and simple.

Finally, you have several means available for tailoring your displayed output.

Entering Data as Matrix Literals

The most basic way to create a matrix is to define a matrix literal, either numeric or character, by entering the matrix elements. A matrix literal can be a single element (called a *scalar*), a single row of data (called a *row vector*), a single column of data (called a *column vector*), or a rectangular array of data (called a *matrix*). The *dimension* of a matrix is given by its number of rows and columns. An $n \times m$ matrix has n rows and m columns.

Scalars

Scalars are matrices that have only one element. You define a scalar with the matrix name on the left-hand side of an assignment statement and its value on the right-hand side. You can use the following statements to create and display several examples of scalar literals. First, you must invoke the IML procedure.

```
> proc iml;

   IML Ready
```

```
> x=12;
> y=12.34;
> z=.;
> a='Hello';
> b="Hi there";
> print x y z a b;

              X          Y          Z  A      B
             12      12.34          .  Hello  Hi there
```

Notice that, when defining a character literal, you need to use either single quotes (') or double quotes ("). Using quotes preserves uppercase and lowercase distinctions and embedded blanks. It is also always correct to enclose the data values inside of braces ({ }).

Matrices with Multiple Elements

To enter a matrix having multiple elements, use braces ({ }) to enclose the data values and, if needed, commas to separate rows. Inside of the braces, all elements must be either numeric or character. You cannot have a mixture of data types within a matrix. Each row must have the same number of elements.

For example, suppose that you have one week of data on daily coffee consumption (cups per day) for your office of four people. Create a matrix **COFFEE** with each person's consumption as a row of the matrix and each day represented by a column. First, submit the RESET: PRINT statemento that results are displayed as you submit statements.

```
> reset print;
> coffee={4 2 2 3 2,
>         3 3 1 2 1,
>         2 1 0 2 1,
>         5 4 4 3 4};

        COFFEE        4 rows      5 cols    (numeric)

               4         2         2         3         2
               3         3         1         2         1
               2         1         0         2         1
               5         4         4         3         4
```

Now create a character matrix called **NAMES** with rows containing the names of the people in your office. Note that when you do not use quotes, characters are converted to uppercase.

```
> names={Jenny, Linda, Jim, Samuel};

        NAMES         4 rows      1 col     (character, size 6)

                              JENNY
                              LINDA
```

```
JIM
SAMUEL
```

Notice that the output with the RESET PRINT statement includes the dimension, type, and (when type is character) the element size of the matrix. The element size represents the length of each string, and it is determined from the length of the longest string.

Now, display the **COFFEE** matrix using **NAMES** as row labels by specifying the ROWNAME= option in the PRINT statement.

```
> print coffee [rowname=names];

      COFFEE
JENNY          4          2          2          3          2
LINDA          3          3          1          2          1
JIM            2          1          0          2          1
SAMUEL         5          4          4          3          4
```

Using Assignment Statements

Assignment statements create matrices by evaluating expressions and assigning the results to a matrix. The expressions can be composed of operators (for example, the matrix addition operator (+)), functions (for example, the INV function), and subscripts. Assignment statements have the general form

result = *expression*;

where *result* is the name of the new matrix and *expression* is an expression that is evaluated. The resulting matrix automatically acquires the appropriate dimension, type, and value. Details on writing expressions are described in "Using Matrix Expressions" later in this chapter.

Simple Assignment Statements

Simple assignment statements involve an equation having the matrix name on the left-hand side and either an expression involving other matrices or a matrix-generating function on the right-hand side.

Suppose you want to generate some statistics for the weekly coffee data. If a cup of coffee costs 30 cents, then you can create a matrix with the daily expenses, **DAYCOST**, by multiplying the per-cup cost with the matrix **COFFEE** using the element-wise multiplication operator (#). Turn off the automatic printing so that you can tailor the output with the ROWNAME= and FORMAT= options in the PRINT statement.

```
> reset noprint;
> daycost=0.30#coffee;
> print "Daily totals", daycost[rowname=names format=8.2];
```

```
Daily totals

DAYCOST
JENNY       1.20     0.60     0.60     0.90     0.60
LINDA       0.90     0.90     0.30     0.60     0.30
JIM         0.60     0.30     0.00     0.60     0.30
SAMUEL      1.50     1.20     1.20     0.90     1.20
```

You can calculate the weekly total cost for each person using the matrix multiplication operator (*). First create a 5×1 vector of 1s. This vector sums the daily costs for each person when multiplied with **COFFEE**. (You will see later that there is a more efficient way to do this using subscript reduction operators.)

```
> ones={1,1,1,1,1};
> weektot=daycost*ones;
> print "Week total", weektot[rowname=names format=8.2];

          Week total

          WEEKTOT
          JENNY       3.90
          LINDA       3.00
          JIM         1.80
          SAMUEL      6.00
```

Finally, you can calculate the average number of cups drunk per day by dividing the grand total of cups by days. To find the grand total, use the SUM function, which returns the sum of all elements of a matrix. Next, divide the grand total by 5, the number of days (which is the number of columns) using the division operator (/) and the NCOL function. These two matrices are created separately, but the entire calculation could be done in one statement.

```
> grandtot=sum(coffee);
> average=grandtot/ncol(coffee);
> print "Total number of cups", grandtot,,"Daily average",average;

                    Total number of cups

                       GRANDTOT
                             49

                     Daily average

                       AVERAGE
                           9.8
```

Matrix-generating Functions

SAS/IML software has many built-in functions that generate useful matrices. For example, the J function creates a matrix with a given dimension and element value when you supply the number of rows and columns, and an element value for the new

matrix. This function is useful to initialize a matrix to a predetermined size. Several matrix-generating functions are listed below:

BLOCK	creates a block-diagonal matrix.
DESIGNF	creates a full-rank design matrix.
I	creates an identity matrix.
J	creates a matrix of a given dimension.
SHAPE	shapes a new matrix from the argument.

The sections that follow illustrate these matrix-generating functions. Again, they are shown with automatic printing of results, activated by invoking the RESET: PRINT statement.

```
reset print;
```

The BLOCK Function

The BLOCK function has the general form

BLOCK(*matrix1,<matrix2,. . .,matrix15 >***);**

and creates a block-diagonal matrix from the argument matrices. For example, the statements

```
>  a={1 1,1 1};

          A             2 rows      2 cols    (numeric)

                           1          1
                           1          1


>  b={2 2, 2 2};

          B             2 rows      2 cols    (numeric)

                           2          2
                           2          2

>  c=block(a,b);
```

result in the matrix

```
          C             4 rows      4 cols    (numeric)

                  1          1          0          0
                  1          1          0          0
                  0          0          2          2
                  0          0          2          2
```

The J Function

The J function has the general form

J(*nrow*<,*ncol*<,*value*>>**);**

and creates a matrix having *nrow* rows, *ncol* columns, and all element values equal to *value*. The *ncol* and *value* arguments are optional, but you will usually want to specify them. In many statistical applications, it is helpful to be able to create a row (or column) vector of 1s (you did so to calculate coffee totals in the last section). You can do this with the J function. For example, the following statement creates a 1×5 row vector of 1s:

```
> one=j(1,5,1);
```

```
          ONE          1 row       5 cols    (numeric)
                 1          1          1          1          1
```

The I Function

The I function creates an identity matrix of a given size. It has the general form

I(*dimension* **);**

where *dimension* gives the number of rows. For example, the following statement creates a 3×3 identity matrix:

```
> I3=I(3);
```

```
             I3            3 rows      3 cols    (numeric)

                          1          0          0
                          0          1          0
                          0          0          1
```

The DESIGNF Function

The DESIGNF function generates a full-rank design matrix, useful in calculating ANOVA tables. It has the general form

DESIGNF(*column-vector* **);**

For example, the following statement creates a full-rank design matrix for a one-way *ANOVA*, where the treatment factor has three levels and there are $n_1 = 3$, $n_2 = 2$, and $n_3 = 2$ observations at the factor levels:

```
> d=designf({1,1,1,2,2,3,3});
```

```
             D            7 rows      2 cols    (numeric)

                             1          0
```

```
 1        0
 1        0
 0        1
 0        1
-1       -1
-1       -1
```

The SHAPE Function

The SHAPE function shapes a new matrix from an argument matrix. It has the general form

SHAPE(*matrix*<,*nrow*<,*ncol*<,*pad-value* >>>**);**

Although the *nrow*, *ncol*, and *pad-value* arguments are optional, you will usually want to specify them. The following example uses the SHAPE function to create a 3×3 matrix containing the values 99 and 33. The function cycles back and repeats values to fill in when no *pad-value* is given.

```
> aa=shape({99 33,99 33},3,3);

        AA            3 rows      3 cols    (numeric)

                   99        33        99
                   33        99        33
                   99        33        99
```

In the next example, a *pad-value* is specified for filling in the matrix:

```
> aa=shape({99 33,99 33},3,3,0);

        AA            3 rows      3 cols    (numeric)

                   99        33        99
                   33         0         0
                    0         0         0
```

The SHAPE function cycles through the argument matrix elements in row-major order and then fills in with 0s after the first cycle through the argument matrix.

Index Vectors

You can create a vector by using the index operator (:). Several examples of statements involving index vectors are shown in the following code:

```
> r=1:5;

            R             1 row       5 cols    (numeric)
                     1         2         3         4         5

> s=10:6;
```

```
              S            1 row       5 cols    (numeric)
                    10          9          8         7         6

>  t='abc1':'abc5';

              T            1 row       5 cols    (character, size 4)
                           abc1 abc2 abc3 abc4 abc5
```

If you want an increment other than 1, use the DO function. For example, if you want a vector ranging from -1 to 1 by 0.5, use the following statement:

```
>  r=do(-1,1,.5);

              R            1 row       5 cols    (numeric)
                   -1       -0.5          0       0.5         1
```

Using Matrix Expressions

Matrix expressions are a sequence of names, literals, operators, and functions that perform some calculation, evaluate some condition, or manipulate values. These expressions can appear on either side of an assignment statement.

Operators

Operators used in matrix expressions fall into three general categories:

prefix operators	are placed in front of operands. For example, $-\mathbf{A}$ uses the sign reverse prefix operator ($-$) in front of the operand $\mathbf{A}$ to reverse the sign of each element of $\mathbf{A}$.
infix operators	are placed between operands. For example, $\mathbf{A}+\mathbf{B}$ uses the addition infix operator (+) between operands $\mathbf{A}$ and $\mathbf{B}$ to add corresponding elements of the matrices.
postfix operators	are placed after an operand. For example, $\mathbf{A}`$ uses the transpose postfix operator (`) after the operand $\mathbf{A}$ to transpose $\mathbf{A}$.

Matrix operators are listed in Appendix 1, "SAS/IML Quick Reference," and described in detail in Chapter 17, "Language Reference." Table 4.1 on page 44 shows the precedence of matrix operators in Interactive Matrix Language.

Table 4.1. Operator Precedence

Priority Group	Operators					
I (highest)	^	`	subscripts	–(prefix)	##	**
II	*	#	<>	><	/	@
III	+	–				
IV	|||	//	:			
V	<	<=	>	>=	=	^=
VI	&					
VII (lowest)	|					

Compound Expressions

With SAS/IML software, you can write compound expressions involving several matrix operators and operands. For example, the following statements are valid matrix assignment statements:

```
a=x+y+z;
a=x+y*z\prime ;
a=(-x)#(y-z);
```

The rules for evaluating compound expressions are as follows:

- Evaluation follows the order of operator precedence, as shown in Table 4.1. Group I has the highest priority; that is, Group I operators are evaluated first. Group II operators are evaluated after Group I operators, and so forth. For example, the statement

  ```
  a=x+y*z;
  ```

  first multiplies matrices **Y** and **Z** since the * operator (Group II) has higher precedence than the + operator (Group III). It then adds the result of this multiplication to the matrix **X** and assigns the new matrix to **A**.

- If neighboring operators in an expression have equal precedence, the expression is evaluated from left to right, except for the highest priority operators. For example, the statement

  ```
  a=x/y/z;
  ```

  first divides each element of matrix **X** by the corresponding element of matrix **Y**. Then, using the result of this division, it divides each element of the resulting matrix by the corresponding element of matrix **Z**. The operators in Group 1 in Table 4.1 are evaluated from right to left. For example, the expression

  ```
  -x**2
  ```

  is evaluated as

  ```
  -(x**2)
  ```

When multiple prefix or postfix operators are juxtaposed, precedence is determined by their order from inside to outside.

For example, the expression

```
^-a
```

is evaluated as $\hat{}\,(-\mathbf{A})$, and the expression

```
a'[i,j]
```

is evaluated as $(\mathbf{A}`)[i,j]$.

- All expressions enclosed in parentheses are evaluated first, using the two preceding rules. Thus, the IML statement

```
a=x/(y/z);
```

is evaluated by first dividing elements of $\mathbf{Y}$ by the elements of $\mathbf{Z}$, then dividing this result into $\mathbf{X}$.

Elementwise Binary Operators

Elementwise binary operators produce a result matrix from element-by-element operations on two argument matrices. Table 4.2 on page 46 lists the elementwise binary operators.

Table 4.2. Elementwise Binary Operators

Operator	Action
$+$	addition, concatenation
$-$	subtraction
$\#$	elementwise multiplication
$\#\#$	elementwise power
$/$	division
$<>$	element maximum
$><$	element minimum
$\vert$	logical OR
$\&$	logical AND
$<$	less than
$<=$	less than or equal to
$>$	greater than
$>=$	greater than or equal to
$\hat{}=$	not equal to
$=$	equal to
MOD(m,n)	modulo (remainder)

For example, consider the following two matrices $\mathbf{A}$ and $\mathbf{B}$ given below.

$$\text{Let } \mathbf{A} = \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 4 & 5 \\ 1 & 0 \end{bmatrix}$$

The addition operator $(+)$ adds corresponding matrix elements:

$$\mathbf{A} + \mathbf{B} \text{ yields } \begin{bmatrix} 6 & 7 \\ 4 & 4 \end{bmatrix}$$

The elementwise multiplication operator $(\#)$ multiplies corresponding elements:

$$\mathbf{A}\#\mathbf{B} \text{ yields } \begin{bmatrix} 8 & 10 \\ 3 & 0 \end{bmatrix}$$

The elementwise power operator $(\#\#)$ raises elements to powers:

$$\mathbf{A}\#\#2 \text{ yields } \begin{bmatrix} 4 & 4 \\ 9 & 16 \end{bmatrix}$$

The element maximum operator $(<>)$ compares corresponding elements and chooses the larger:

$$\mathbf{A} <> \mathbf{B} \text{ yields } \begin{bmatrix} 4 & 5 \\ 3 & 4 \end{bmatrix}$$

The less than or equal to operator $(<=)$ returns a 1 if an element of $\mathbf{A}$ is less than or equal to the corresponding element of $\mathbf{B}$, and returns a 0 otherwise:

$$\mathbf{A} <= \mathbf{B} \text{ yields } \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

The modulo operator returns the remainder of each element divided by the argument:

$$\text{MOD}(\mathbf{A}, 3) \text{ yields } \begin{bmatrix} 2 & 2 \\ 0 & 1 \end{bmatrix}$$

All operators can also work in a one-to-many or many-to-one manner, as well as in an element-to-element manner; that is, they enable you to perform tasks such as adding a scalar to a matrix or dividing a matrix by a scalar. For example, the statement

```
x=x#(x>0);
```

replaces each negative element of the matrix $\mathbf{X}$ with 0. The expression (X>0) is a many-to-one operation that compares each element of $\mathbf{X}$ to 0 and creates a temporary matrix of results; an element in the result matrix is 1 when the expression is true and 0 when it is false. When the expression is true (the element is positive), the element is multiplied by 1. When the expression is false (the element is negative or 0), the element is multiplied by 0. To fully understand the intermediate calculations, you can use the RESET: PRINTALL command to have the temporary result matrices displayed.

Subscripts

Subscripts are special postfix operators placed in square brackets ([ ]) after a matrix operand. Subscript operations have the general form

$$operand[row, column]$$

where

operand	is usually a matrix name, but it can also be an expression or literal.
row	refers to an expression, either scalar or vector, for selecting one or more rows from the operand.
column	refers to an expression, either scalar or vector, for selecting one or more columns from the operand.

You can use subscripts to

- refer to a single element of a matrix
- refer to an entire row or column of a matrix
- refer to any submatrix contained within a matrix
- perform a reduction across rows or columns of a matrix

In expressions, subscripts have the same (high) precedence as the transpose postfix operator (`). Note that when both *row* and *column* subscripts are used, they are separated by a comma.

Selecting a Single Element

You can select a single element of a matrix in two ways. You can use two subscripts (*row*, *column*) to refer to its location, or you can use one subscript to look for the element down the rows. For example, referring to the coffee example used earlier, to find the element corresponding to the number of cups that Linda drank on Monday, you can use either of two statements.

First, you can refer to the element by row and column location. In this case, you want the second row and first column. You can call this matrix `c21`.

```
> print  coffee[rowname=names];

        COFFEE
        JENNY          4         2         2         3         2
        LINDA          3         3         1         2         1
        JIM            2         1         0         2         1
        SAMUEL         5         4         4         3         4

> c21=coffee[2,1];
> print c21;

                              C21
                                3
```

You can also look for the element down the rows. In this case, you refer to this element as the sixth element of **COFFEE** in row-major order.

```
> c6=coffee[6];
> print c6;
```

```
C6
 3
```

Selecting a Row or Column

To refer to an entire row or column of a matrix, write the subscript with the row or column number, omitting the other subscript but not the comma. For example, to refer to the row of **COFFEE** that corresponds to Jim, you want the submatrix consisting of the third row and all columns:

```
> jim=coffee[3,];
> print jim;
```

```
JIM
  2        1        0        2        1
```

If you want the data for Friday, you know that the fifth column corresponds to Friday, so you want the submatrix consisting of the fifth column and all rows:

```
> friday=coffee[,5];
> print friday;
```

```
FRIDAY
     2
     1
     1
     4
```

Submatrices

You refer to a submatrix by the specific rows and columns you want. Include within the brackets the rows you want, a comma, and the columns you want. For example, to create the submatrix of **COFFEE** consisting of the first and third rows and the second, third, and fifth columns, submit the following statements:

```
> submat1=coffee[{1 3},{2 3 5}];
> print submat1;
```

```
SUBMAT1
     2        2        2
     1        0        1
```

The first vector, {1 3}, selects the rows, and the second vector, {2 3 5}, selects the columns. Alternately, you can create the vectors beforehand and supply their names as arguments.

```
> rows={1 3};
> cols={2 3 5};
> submat1=coffee[rows,cols];
```

You can use index vectors generated by the index creation operator (:) in subscripts to refer to successive rows or columns. For example, to select the first three rows and last three columns of **COFFEE**, use the following statements:

```
> submat2=coffee[1:3,3:5];
> print submat2;
```

```
SUBMAT2
     2        3        2
     1        2        1
     0        2        1
```

Note that, in each example, the number in the first subscript defines the number of rows in the new matrix; the number in the second subscript defines the number of columns.

Subscripted Assignment

You can assign values into a matrix using subscripts to refer to the element or submatrix. In this type of assignment, the subscripts appear on the left-hand side of the equal sign. For example, to change the value in the first row, second column of **COFFEE** from 2 to 4, use subscripts to refer to the appropriate element in an assignment statement:

```
> coffee[1,2]=4;
> print coffee;
```

```
COFFEE
     4        4        2        3        2
     3        3        1        2        1
     2        1        0        2        1
     5        4        4        3        4
```

To change the values in the last column of **COFFEE** to 0s use the following statement:

```
> coffee[,5]={0,0,0,0};
> print coffee;
```

```
COFFEE
     4        4        2        3        0
     3        3        1        2        0
     2        1        0        2        0
     5        4        4        3        0
```

In the next example, you first locate the positions of negative elements of a matrix and then set these elements equal to 0. This can be useful in situations where negative elements may indicate errors or be impossible values. The LOC function is useful for

creating an index vector for a matrix that satisfies some condition. In the following example, the LOC function is used to find the positions of the negative elements of the matrix **T** and then to set these elements equal to 0 using subscripted assignment:

```
>  t={ 3  2 -1,
>      6 -4  3,
>      2  2  2 };
>  print t;


             T
             3         2        -1
             6        -4         3
             2         2         2

>  i=loc(t<0);
>  print i;

             I
             3         5

>  t[i]=0;
>  print t;

             T
             3         2         0
             6         0         3
             2         2         2
```

Subscripts can also contain expressions with results that are either row or column vectors. These statements can also be written

```
>  t[loc(t<0)]=0;
```

If you use a noninteger value as a subscript, only the integer portion is used. Using a subscript value less than one or greater than the dimension of the matrix results in an error.

Subscript Reduction Operators

You can use reduction operators, which return a matrix of reduced dimension, in place of values for subscripts to get reductions across all rows and columns. Table 4.3 lists the eight operators for subscript reduction in IML.

Table 4.3. Subscript Reduction Operators

Operator	Action
+	addition
#	multiplication
<>	maximum
><	minimum
<:>	index of maximum
>:<	index of minimum
:	mean (different from the MATRIX procedure)
##	sum of squares

For example, to get column sums of the matrix **X** (sum across the rows, which reduces the row dimension to 1), specify $X[+,]$. The first subscript (+) specifies that summation reduction take place across the rows. Omitting the second subscript, corresponding to columns, leaves the column dimension unchanged. The elements in each column are added, and the new matrix consists of one row containing the column sums.

You can use these operators to reduce either rows or columns or both. When both rows and columns are reduced, row reduction is done first.

For example, the expression $A[+,<>]$ results in the maximum $(<>)$ of the column sums $(+)$.

You can repeat reduction operators. To get the sum of the row maxima, use the expression $A[,<>][+,]$.

A subscript such as $A[\{23\},+]$ first selects the second and third rows of **A** and then finds the row sums of that matrix. The following examples demonstrate how to use the operators for subscript reduction.

$$\text{Let } \mathbf{A} = \begin{bmatrix} 0 & 1 & 2 \\ 5 & 4 & 3 \\ 7 & 6 & 8 \end{bmatrix}$$

The following statements are true:

$\mathbf{A}[23,+]$ yields $\begin{bmatrix} 12 \\ 21 \end{bmatrix}$ (row sums for rows 2 and 3)

$\mathbf{A}[+,<>]$ yields $\begin{bmatrix} 13 \end{bmatrix}$ (maximum of column sums)

$\mathbf{A}[<>,+]$ yields $\begin{bmatrix} 21 \end{bmatrix}$ (sum of column maxima)

$\mathbf{A}[,><][+,]$ yields $\begin{bmatrix} 9 \end{bmatrix}$ (sum of row minima)

$\mathbf{A}[,<:>]$ yields $\begin{bmatrix} 3 \\ 1 \\ 3 \end{bmatrix}$ (indices of row maxima)

$\mathbf{A}[>:<,]$ yields $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ (indices of column minima)

$\mathbf{A}[:]$ yields $\begin{bmatrix} 4 \end{bmatrix}$ (mean of all elements)

Displaying Matrices with Row and Column Headings

You can tailor the way your matrices are diplayed with the AUTONAME option, the ROWNAME= and COLNAME= options, or the MATTRIB statement.

Using the AUTONAME Option

You can use the RESET statement with the AUTONAME option to automatically diplay row and column headings. If your matrix has n rows and m columns, the row headings are ROW1 to ROWn and the column headings are COL1 to COLm. For example, the following statements produce the following results:

```
> reset autoname;
> print coffee;
```

COFFEE	COL1	COL2	COL3	COL4	COL5
ROW1	4	2	2	3	2
ROW2	3	3	1	2	1
ROW3	2	1	0	2	1
ROW4	5	4	4	3	4

Using the ROWNAME= and COLNAME= Options

You can specify your own row and column headings. The easiest way is to create vectors containing the headings and then display the matrix is with the ROWNAME= and COLNAME= options. For example, the following statements produce the following results:

```
> names={jenny linda jim samuel};
> days={mon tue wed thu fri};
> print coffee[rowname=names colname=days];
```

COFFEE	MON	TUE	WED	THU	FRI
JENNY	4	2	2	3	2
LINDA	3	3	1	2	1
JIM	2	1	0	2	1
SAMUEL	5	4	4	3	4

Using the MATTRIB Statement

The MATTRIB statement associates printing characteristics with matrices. You can use the MATTRIB statement to display **COFFEE** with row and column headings. In addition, you can format the displayed numeric output and assign a label to the matrix name. The following example shows how to tailor your displayed output:

```
> mattrib coffee rowname={jenny linda jim samuel}
>         colname={mon tue wed thu fri}
>         label='Weekly Coffee'
>         format=2.0;
> print coffee;
```

Weekly Coffee	MON	TUE	WED	THU	FRI
JENNY	4	2	2	3	2
LINDA	3	3	1	2	1
JIM	2	1	0	2	1
SAMUEL	5	4	4	3	4

More on Missing Values

Missing values in matrices are discussed in Chapter 2, "Understanding the Language." You should read that chapter and Chapter 16, "Further Notes," carefully so that you are aware of the way IML treats missing values. Following are several examples that show how IML handles missing values in a matrix.

$$\text{Let } \mathbf{X} = \begin{bmatrix} 1 & 2 & . \\ . & 5 & 6 \\ 7 & . & 9 \end{bmatrix} \text{ and } \mathbf{Y} = \begin{bmatrix} 4 & . & 2 \\ 2 & 1 & 3 \\ 6 & . & 5 \end{bmatrix}$$

The following statements are true:

$$\mathbf{X} + \mathbf{Y} \text{ yields } \begin{bmatrix} 5 & . & . \\ . & 6 & 9 \\ 13 & . & 14 \end{bmatrix} \text{ (matrix addition)}$$

$$\mathbf{X}\#\mathbf{Y} \text{ yields } \begin{bmatrix} 4 & . & . \\ . & 5 & 18 \\ 42 & . & 45 \end{bmatrix} \text{ (element multiplication)}$$

$$\mathbf{X}[+,] \text{ yields } \begin{bmatrix} 8 & 7 & 15 \end{bmatrix} \text{ (column sums)}$$

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/IML User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 846 pp.

SAS/IML User's Guide, Version 8

ISBN 1–58025–553–1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999