Chapter 5

Working with Matrices

Contents

Overview of Working with Matrices

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

- entering data as a matrix literal
- using assignment statements
- using functions that generate matrices
- creating submatrices from existing matrices with subscripts

- using SAS data sets (see Chapter 7, "Working with SAS Data Sets," for more information)

Chapter 3, "Understanding the SAS/IML Language," describes some of these techniques.

After you define matrices, you have access to many operators and functions for forming matrix expressions. These operators and functions facilitate programming and enable you to refer to submatrices. This chapter describes how to work with matrices in the SAS/IML language.

Entering Data as Matrix Literals

The simplest way to create a matrix is to define a matrix literal by entering the matrix elements. A matrix literal can contain numeric or character data. 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 $n \times p$ matrix has n rows and p columns.

Scalars

Scalars are matrices that have only one element. You can define a scalar by typing the matrix name on the left side of an assignment statement and its value on the right side. The following statements create and display several examples of scalar literals:

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

The output is displayed in Figure 5.1. Notice that you need to use either single quotes (') or double quotes (") when defining a character literal. Using quotes preserves the case and embedded blanks of the literal. It is also always correct to enclose data values within braces ({ }).

Figure 5.1 Examples of Scalar Quantities

x	y	z	a	b
12	12.34	.	Hello	Hi there

Matrices with Multiple Elements

To enter a matrix having multiple elements, use braces ({ }) to enclose the data values. If the matrix has multiple rows, use commas to separate them. Inside 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 you have one week of data on daily coffee consumption (cups per day) for four people in your office. Create a 4×5 matrix called **coffee** with each person's consumption represented by a row of the matrix and each day represented by a column. The following statements use the RESET PRINT command so that the result of each assignment statement is displayed automatically:

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

Figure 5.2 A 4×5 Matrix

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

Next, you can create a character matrix called **names** with rows that contains the names of the coffee drinkers in your office. Notice in Figure 5.3 that if you do not use quotes, characters are converted to uppercase.

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

Figure 5.3 A Column Vector of Names

names	4 rows	1 col	(character, size 6)
		JENNY	
		LINDA	
		JIM	
		SAMUEL	

Notice that RESET PRINT statement produces output that includes the name of the matrix, its dimensions, its type, and (when the type is character) the element size of the matrix. The element size represents the length of each string, and it is determined by the length of the longest string.

Next display the **coffee** matrix using the elements of **names** as row names by specifying the ROWNAME= option in the PRINT statement:

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

Figure 5.4 Rows of a Matrix Labeled by a Vector

	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 about writing expressions are described in the section "Using Matrix Expressions" on page 50.

Simple Assignment Statements

Simple assignment statements involve an equation that has a matrix name on the left side and either an expression or a function that generates a matrix on the right side.

Suppose that 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**. You can turn off the automatic printing so that you can customize the output with the ROWNAME=, FORMAT=, and LABEL= options in the PRINT statement, as shown in the following statements:

```
reset noprint;
dayCost = 0.30 # coffee;   /* elementwise multiplication */
print dayCost[rowname=names format=8.2 label="Daily totals"];
```

Figure 5.5 Daily Cost for Each Employee

	Daily totals				
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 by using the matrix multiplication operator (*). First create a 5×1 vector of ones. This vector sums the daily costs for each person when multiplied with the **coffee** matrix. (A more efficient way to do this is by using subscript reduction operators, which are discussed in "Using Matrix Expressions" on page 50.) The following statements perform the multiplication:

```
ones = {1,1,1,1,1};
weektot = dayCost * ones;  /* matrix-vector multiplication */
print weektot[rowname=names format=8.2 label="Weekly totals"];
```

Figure 5.6 Weekly Total for Each Employee

	Weekly totals
JENNY	3.90
LINDA	3.00
JIM	1.80
SAMUEL	6.00

You might want to calculate the average number of cups consumed per day in the office. You can use the SUM function, which returns the sum of all elements of a matrix, to find the total number of cups consumed in the office. Then divide the total by 5, the number of days. The number of days is also the number of columns in the **coffee** matrix, which you can determine by using the NCOL function. The following statements perform this calculation:

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

Figure 5.7 Total and Average Number of Cups for the Office

Total number of cups	Daily average
49	9.8

Functions That Generate Matrices

SAS/IML software has many useful built-in functions that generate matrices. For example, the J function creates a matrix with a given dimension and specified element value. You can use this function to initialize a matrix to a predetermined size. Here are several functions that generate matrices:

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
REPEAT	creates a new matrix by repeating elements of the argument matrix
SHAPE	shapes a new matrix from the argument

The sections that follow illustrate the functions that generate matrices. The output of each example is generated automatically by using the RESET PRINT statement:

```
reset print;
```

The BLOCK Function

The BLOCK function has the following general form:

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

The BLOCK function creates a block-diagonal matrix from the argument matrices. For example, the following statements form a block-diagonal matrix:

```
a = {1 1, 1 1};
b = {2 2, 2 2};
c = block(a,b);
```

Figure 5.8 A Block-Diagonal 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 following general form:

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

It creates a matrix that has *nrow* rows, *ncol* columns, and all elements equal to *value*. The *ncol* and *value* arguments are optional; if they are not specified, default values are used. In many statistical applications, it is helpful to be able to create a row (or column) vector of ones. (You did so to calculate coffee totals in the previous section.) You can do this with the J function. For example, the following statement creates a 5×1 column vector of ones:

```
ones = j(5,1,1);
```

Figure 5.9 A Vector of Ones

```
                ones      5 rows      1 col     (numeric)

                                        1
                                        1
                                        1
                                        1
                                        1
```

The I Function

The I function creates an identity matrix of a given size. It has the following 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);
```

Figure 5.10 An Identity Matrix

```
                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, which is useful in calculating ANOVA tables. It has the following 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});
```

Figure 5.11 A Design Matrix

```
                 d      7 rows      2 cols    (numeric)
```

Figure 5.11 *continued*

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

The REPEAT Function

The REPEAT function creates a new matrix by repeating elements of the argument matrix. It has the following syntax:

REPEAT *(matrix, nrow, ncol)* **;**

The function repeats *matrix* a total of *nrow*× *ncol* times. The argument is repeated *nrow* times in the vertical direction and *ncol* times in the horizontal direction. For example, the following statement creates a 4 × 6 matrix:

```
x = {1 2, 3 4};
r = repeat(x, 2, 3);
```

Figure 5.12 A Matrix of Repeated Values

```
                  r      4 rows      6 cols    (numeric)

              1          2          1          2          1          2
              3          4          3          4          3          4
              1          2          1          2          1          2
              3          4          3          4          3          4
```

The SHAPE Function

The SHAPE function creates a new matrix by reshaping an argument matrix. It has the following general form:

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

The *ncol* and *pad-value* arguments are optional; if they are not specified, default values are used. The following statement uses the SHAPE function to create a 3 × 3 matrix that contains the values 99 and 33. The function cycles back and repeats values to fill in the matrix when no *pad-value* is given.

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

Figure 5.13 A Matrix of Repeated Values

```
                 aa      3 rows      3 cols    (numeric)
```

Figure 5.13 *continued*

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

Alternatively, you can specify a value for *pad-value* that is used for filling in the matrix:

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

Figure 5.14 A Matrix Padded with Zeroes

```
                    bb      3 rows      3 cols    (numeric)

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

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

Index Vectors

You can create a row vector by using the index operator (:). The following statements show that you can use the index operator to count up, count down, or to create a vector of character values with numerical suffixes:

```
r = 1:5;
s = 10:6;
t = 'abc1':'abc5';
```

Figure 5.15 Row Vectors Created with the Index Operator

```
                    r      1 row       5 cols    (numeric)

                      1          2          3          4          5

                    s      1 row       5 cols    (numeric)

                     10          9          8          7          6

                t      1 row       5 cols    (character, size 4)

                           abc1 abc2 abc3 abc4 abc5
```

To create a vector based on an increment other than 1, use the DO function. For example, if you want a vector that ranges from −1 to 1 by 0.5, use the following statement:

```
u = do(-1,1,.5);
```

Figure 5.16 Row Vector Created with the DO Function

```
                    u      1 row       5 cols     (numeric)

                   -1       -0.5          0        0.5          1
```

Using Matrix Expressions

A matrix expression is a sequence of names, literals, operators, and functions that perform some calculation, evaluate some condition, or manipulate values. Matrix 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, **-A** uses the sign reversal prefix operator (−) in front of the matrix **A** to reverse the sign of each element of **A**.

Binary operators — are placed between operands. For example, **A + B** uses the addition binary operator (+) between matrices **A** and **B** to add corresponding elements of the matrices.

Postfix operators — are placed after an operand. For example, **A`** uses the transpose postfix operator (`) after the matrix **A** to transpose the matrix.

Matrix operators are described in detail in Chapter 23, "Language Reference."

Table 5.1 shows the precedence of matrix operators in the SAS/IML language.

Table 5.1 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 that involve several matrix operators and operands. For example, the following statements are valid matrix assignment statements:

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

The rules for evaluating compound expressions are as follows:

- Evaluation follows the order of operator precedence, as described in Table 5.1. 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. Consider the following statement:

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

  This statement 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 Group I operators. Consider the following statement:

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

  This statement 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 I, described in Table 5.1, are evaluated from right to left. For example, the following expression is evaluated as $-(\mathbf{X}^2)$:

  ```
  -x**2
  ```

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

  For example, the following expression is evaluated as $(\mathbf{A}`)[i, j]$:

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

- All expressions enclosed in parentheses are evaluated first, using the two preceding rules. Consider the following statement:

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

  This statement is evaluated by first dividing elements of **y** by the elements of **z**, then dividing this result into **x**.

Elementwise Binary Operators

Elementwise binary operators produce a result matrix from element-by-element operations on two argument matrices.

Table 5.2 lists the elementwise binary operators.

Table 5.2 Elementwise Binary Operators

Operator	Description
+	Addition; string concatenation
−	Subtraction
#	Elementwise multiplication
##	Elementwise power
/	Division
<>	Element maximum
><	Element minimum
|	Logical OR
&	Logical AND
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
^=	Not equal to
=	Equal to

For example, consider the following two matrices:

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

The addition operator (+) adds corresponding matrix elements, as follows:

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

The elementwise multiplication operator (#) multiplies corresponding elements, as follows:

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

The elementwise power operator (##) raises elements to powers, as follows:

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

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

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

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

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

All operators can work on scalars, vectors, or matrices, provided that the operation makes sense. For example, you can add a scalar to a matrix or divide a matrix by a scalar. For example, the following statement replaces each negative element of the matrix **x** with 0:

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

The expression **x>0** is an operation that compares each element of **x** to (scalar) zero and creates a temporary matrix of results; an element of the temporary matrix is 1 when the corresponding element of **x** is positive, and 0 otherwise. The original matrix **x** is then multiplied elementwise by the temporary matrix, resulting in the matrix **y**. To fully understand the intermediate calculations, you can use the RESET statement with the PRINTALL option 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 a scalar or vector expression that selects one or more rows from the operand.
column	refers to a scalar or vector expression that selects one or more columns from the operand.

You can use subscripts to do any of the following:

- 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. A reduction is a statistical operation (often a sum or mean) applied to the rows or to the columns of a matrix.

In expressions, subscripts have the same (high) precedence as the transpose postfix operator (`). When both *row* and *column* subscripts are used, they are separated by a comma. If a matrix has row or column names associated with it from a MATTRIB or READ statement, then the corresponding row or column subscript can also be a character matrix whose elements match the names of the rows or columns to be selected.

Selecting a Single Element

You can select a single element of a matrix in several ways. You can use two subscripts (*row*, *column*) to refer to its location, or you can use one subscript to index the elements in row-major order.

For example, for the coffee example used previously in this chapter, there are several ways to find the element that corresponds to the number of cups that Samuel drank on Monday.

First, you can refer to the element by row and column location. In this case, you want the fourth row and first column. The following statements extract the datum and place it in the matrix **c41**:

```
coffee={4 2 2 3 2, 3 3 1 2 1, 2 1 0 2 1, 5 4 4 3 4};
names={Jenny, Linda, Jim, Samuel};
print coffee[rowname=names];
c41 = coffee[4,1];
print c41;
```

Figure 5.17 Datum Extracted from a Matrix

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

c41

5

You can also use row and column names, which can be assigned with an MATTRIB statement as follows:

```
mattrib coffee rowname=names
   colname={'MON' 'TUE' 'WED' 'THU' 'FRI'};
cSamMon = coffee['SAMUEL','MON'];
print cSamMon;
```

Figure 5.18 Datum Extracted from a Matrix with Assigned Attributes

cSamMon

5

You can also look for the element by enumerating the elements of the matrix in row-major order. In this case, you refer to this element as the sixteenth element of **coffee**:

```
c16 = coffee[16];
print c16;
```

Figure 5.19 Datum Extracted from a Matrix by Specifying the Element Number

```
c16

   5
```

Selecting a Row or Column

To refer to an entire row of a matrix, specify the subscript for the row but omit the subscript for the column. For example, to refer to the row of the **coffee** matrix that corresponds to Jim, you can specify the submatrix that consists of the third row and all columns. The following statements extract and print this submatrix:

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

Alternately, you can use the row names assigned by the MATTRIB statement. Both results are shown in Figure 5.20.

```
jim2 = coffee['JIM',];
print jim2;
```

Figure 5.20 Row Extracted from a Matrix

```
jim

 2   1   0   2   1

jim2

 2   1   0   2   1
```

If you want to extract the data for Friday, you can specify the subscript for the fifth column. You omit the row subscript to indicate that the operation applies to all rows. The following statements extract and print this submatrix:

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

Figure 5.21 Column Extracted from a Matrix

```
friday

   2
   1
   1
   4
```

Alternatively, you could also index by the column name as follows:

```
friday = coffee[,'FRI'];
```

Submatrices

You refer to a submatrix by specifying the rows and columns that determine the submatrix. For example, to create the submatrix of **coffee** that consists of the first and third rows and the second, third, and fifth columns, use the following statements:

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

Figure 5.22 Submatrix Extracted from a Matrix

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 of indices and use them to extract the submatrix, as shown in following statements:

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

You can also use the row and column names:

```
rows = {'JENNY' 'JIM'};
cols = {'TUE' 'WED' 'FRI'};
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, the following statements extract the first three rows and last three columns of **coffee**:

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

Figure 5.23 Submatrix of Contiguous Rows and Columns

submat2		
2	3	2
1	2	1
0	2	1

Selecting Multiple Elements

All SAS/IML matrices are stored in row-major order. This means that you can index multiple elements of a matrix by listing the position of the elements in an $n \times p$ matrix. The elements in the first row have positions

1 through p, the elements in the second row have positions $p + 1$ through $2p$, and the elements in the last row have positions $(n - 1)p + 1$ through np.

For example, in the coffee data discussed previously, you might be interested in finding occurrences for which some person (on some day) drank more than two cups of coffee. The LOC function is useful for creating an index vector for a matrix that satisfies some condition. The following statement uses the LOC function to find the data that satisfy the desired criterion:

```
h = loc(coffee > 2);
print h;
```

Figure 5.24 Indices That Correspond to a Criterion

```
                                h
                 COL1       COL2       COL3       COL4       COL5

ROW1                1          4          6          7         16

                                h
                 COL6       COL7       COL8       COL9

ROW1               17         18         19         20
```

The row vector **h** contains indices of the **coffee** matrix that satisfy the criterion. If you want to find the number of cups of coffee consumed on these occasions, you need to subscript the **coffee** matrix with the indices, as shown in the following statements:

```
cups = coffee[h];
print cups;
```

Figure 5.25 Values That Correspond to a Criterion

```
                cups

                   4
                   3
                   3
                   3
                   5
                   4
                   4
                   3
                   4
```

Notice that SAS/IML software returns a column vector when a matrix is subscripted by a single array of indices. This might surprise you, but clearly the **cups** matrix cannot be the same shape as the **coffee** matrix since it contains a different number of elements. Therefore, the only reasonable alternative is to return either a row vector or a column vector. Either would be a valid choice; SAS/IML software returns a column vector.

Even if the original matrix is a row vector, the subscripted matrix will be a column vector, as the following example shows:

```
v = {-1 2 5 -2 7};  /* v is a row vector */
v2 = v[{1 3 5}];    /* v2 is a column vector */
print v2;
```

Figure 5.26 Column Vector of Extracted Values

v2
-1
5
7

If you want to index into a row vector and you want the resulting variable also to be a row vector, then use the following technique:

```
v3 = v[ ,{1 3 5}];  /* Select columns. Note the comma. */
print v3;
```

Figure 5.27 Row Vector of Extracted Values

v3		
-1	5	7

Subscripted Assignment

You can assign values into a matrix by using subscripts to refer to the element or submatrix. In this type of assignment, the subscripts appear on the left side of the equal sign. For example, to assign the value 4 in the first row, second column of **coffee**, use subscripts to refer to the appropriate element in an assignment statement, as shown in the following statements and in Figure 5.27:

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

To change the values in the last column of **coffee** to zeros, use the following statements:

```
coffee[,5] = {0,0,0,0};   /* alternatively: coffee[,5] = 0; */
print coffee;
```

Figure 5.28 Matrices after Assigning Values to Elements

coffee				
4	4	2	3	2
3	3	1	2	1
2	1	0	2	1
5	4	4	3	4

Figure 5.28 *continued*

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 locate the negative elements of a matrix and set these elements to zero. (This can be useful in situations where negative elements might indicate errors.) The LOC function is useful for creating an index vector for a matrix that satisfies some criterion. The following statements use the LOC function to find and replace the negative elements of the matrix **T**:

```
t = {3  2 -1,
     6 -4  3,
     2  2  2 };
i = loc(t<0);
print i;
t[i] = 0;
print t;
```

Figure 5.29 Results of Finding and Replacing Negative Values

i

3
5

t

3	2	0
6	0	3
2	2	2

Subscripts can also contain expressions. For example, the previous example could have been written as follows:

```
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

A reduction operator is a statistical operation (for example, a sum or a mean) that returns a matrix of a smaller dimension. Reduction operators are often encountered in frequency tables: the marginal frequencies represent the sum of the frequencies across rows or down columns.

In SAS/IML software, you can use reduction operators in place of values for subscripts to get reductions across all rows or columns. Table 5.3 lists operators for subscript reduction.

Table 5.3 Subscript Reduction Operators

Operator	Description
+	Addition
#	Multiplication
<>	Maximum
><	Minimum
<:>	Index of maximum
>:<	Index of minimum
:	Mean
##	Sum of squares

For example, to get row sums of a matrix **X**, you can sum across the columns with the syntax **X[,+]**. Omitting the first subscript specifies that the operator apply to all rows. The second subscript (+) specifies that summation reduction take place across the columns. The elements in each row are added, and the new matrix consists of one column that contains the row sums.

To give a specific example, consider the coffee data from earlier in the chapter. The following statements use the summation reduction operator to compute the sums for each row:

```
coffee={4 2 2 3 2, 3 3 1 2 1, 2 1 0 2 1, 5 4 4 3 4};
names={Jenny, Linda, Jim, Samuel};
mattrib coffee rowname=names colname={'MON' 'TUE' 'WED' 'THU' 'FRI'};
Total = coffee[,+];
print coffee Total;
```

Figure 5.30 Summation across Columns to Find the Row Sums

coffee	MON	TUE	WED	THU	FRI	Total
JENNY	4	2	2	3	2	13
LINDA	3	3	1	2	1	10
JIM	2	1	0	2	1	6
SAMUEL	5	4	4	3	4	20

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

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

You can repeat reduction operators. To get the sum of the row maxima, use the expression $\mathbf{A}[, <>][+,]$, or, equivalently, $\mathbf{A}[, <>][+]$.

A subscript such as $\mathbf{A}[\{2\ 3\}, +]$ first selects the second and third rows of **A** and then finds the row sums of that submatrix.

The following examples demonstrate how to use the operators for subscript reduction. Consider the following matrix:

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

The following statements are true:

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

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

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

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

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

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

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

Displaying Matrices with Row and Column Headings

You can customize the way matrices are displayed with the AUTONAME option, with the ROWNAME= and COLNAME= options, or with the MATTRIB statement.

The AUTONAME Option in the RESET Statement

You can use the RESET statement with the AUTONAME option to automatically display row and column headings. If your matrix has *n* rows and *p* columns, the row headings are ROW1 to ROW*n* and the column headings are COL1 to COL*p*. For example, the following statements produce the subsequent matrix:

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

Figure 5.31 Result of the AUTONAME Option

	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

The ROWNAME= and COLNAME= Options in the PRINT Statement

You can specify your own row and column headings. The easiest way is to create vectors that contain the headings and then display the matrix by using the ROWNAME= and COLNAME= options in the PRINT statement. For example, the following statements display row names and column names for a matrix:

```
names={Jenny, Linda, Jim, Samuel};
days={Mon Tue Wed Thu Fri};
mattrib coffee rowname=names colname=days;
print coffee;
```

Figure 5.32 Result of the ROWNAME= and COLNAME= Options

	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

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 customize your displayed output:

```
mattrib coffee rowname=names
        colname=days
        label='Weekly Coffee'
        format=2.0;
print coffee;
```

Figure 5.33 Result of the MATTRIB Statement

	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 about Missing Values

Missing values in matrices are discussed in Chapter 3, "Understanding the SAS/IML Language." You should carefully read that chapter and Chapter 22, "Further Notes," so that you are aware of the way SAS/IML software handles missing values. The following examples show how missing values are handled for elementwise operations and for subscript reduction operators.

Consider the following two matrices **X** and **Y**:

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

The following operations handle missing values in matrices:

Matrix addition: $\mathbf{X} + \mathbf{Y}$ is $\begin{bmatrix} 5 & . & . \\ . & 6 & 9 \\ 13 & . & 14 \end{bmatrix}$

Elementwise multiplication: $\mathbf{X}\#\mathbf{Y}$ is $\begin{bmatrix} 4 & . & . \\ . & 5 & 18 \\ 42 & . & 45 \end{bmatrix}$

Subscript reduction: $\mathbf{X}[+,]$ is $\begin{bmatrix} 8 & 7 & 15 \end{bmatrix}$