Matrices

Chapter 5

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Create matrices with `matrix()`, `rbind()`, and `cbind()`.
- Understand how R stores and performs operations on matrices.
- Add row and column names to two-dimensional objects.
- Extract and assign values to two-dimensional objects.
- Differentiate between `*` and `%*%` for matrices.
- Compute the inverse of a square matrix.

1 Basic Definitions and Functions

Matrices play an important role in statistics, especially in multivariate analyses, such as multiple regression or estimating the covariance matrix for a collection of random variables (i.e., a random vector). We will not focus on linear algebra or matrix theory in this course, but we will introduce the `matrix` object in R and some important concepts related to them. Matrices provide intuition and a natural segue into working with data frames, possibly the most commonly used object in R to store rectangular data. Most of the syntax and functions that apply to matrices will also apply to data frames.

1.1 The `matrix()` Function

A **matrix** is a two-dimensional (rectangular) array of values. In R, *every value in a matrix must be of the same type* (i.e., all numeric, character, logical). The `matrix()` function takes in a vector of values (called `data`) and arranges them into a matrix according to the desired number of rows (`nrow`) and number of columns (`ncol`).


```
A <- matrix(1:6, nrow = 2, ncol = 3)
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Note: By default, the `matrix()` function reads the input vector and fills in the matrix by column. For example, in matrix `A`, the first two elements fill in the first column, the next two elements in the second column, and the last two elements fill in the last column. To fill the matrix in by row, set the optional `byrow` argument to `TRUE` (the default value is `byrow=FALSE`, which means R will fill the matrix by column).

```
B <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
B
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Note: The default values are `nrow=1` and `ncol=1`. Leaving both blank will produce a matrix with a single column. If only the `nrow` or `ncol` argument is defined, the argument left blank will be automatically computed based on the length of the `data` vector.

Caution: If the length of the `data` vector is too short to fill the entire matrix, the values of `data` will be recycled. The way R behaves when recycling in this context is similar to the vector recycling from Chapter 2. If the vector is recycled a whole number of times, R will fill the matrix with no warning that it has recycled

the `data` values. If the vector is not recycled completely, the matrix will still be filled, but R will also throw a warning.

```
matrix(1:4, nrow = 2, ncol = 6) # Complete recycling: no warning thrown
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    1    3    1    3
[2,]    2    4    2    4    2    4
```

```
matrix(1:4, nrow = 2, ncol = 5) # Incomplete recycling: warning thrown
```

```
Warning in matrix(1:4, nrow = 2, ncol = 5): data length [4] is not a sub-
multiple or multiple of the number of columns [5]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    1    3    1
[2,]    2    4    2    4    2
```


1.2 The Dimension of a Matrix

The **dimension** of a matrix (or any two-dimensional object) is the number of rows and the number of columns, often written as nrow × ncol, read aloud as "nrow by ncol". For example, matrix A is a 2×3 (2 by 3) matrix. A matrix is called **square** if the number of rows equals the number of columns.

The `dim()` function returns a numeric vector specifying the dimensions of the input object.

```
dim(A) # dimension of A
```

```
[1] 2 3
```

```
dim(B) # dimension of B
```



```
[1] 3 3
```

Caution: The *order matters* in the dimension of a matrix. A 2×3 matrix has a different dimension from a 3×2 matrix. The number of rows is always listed first, and the number of columns is always listed second.

The functions **`nrow()`** and **`ncol()`** input a two-dimensional object and output the number of rows or columns, respectively. These will produce the individual entries from the `dim()` function.

```
nrow(A) # The number of rows of A
```

```
[1] 2
```

```
ncol(B) # The number of columns of B
```

```
[1] 3
```

Question: How would you extract the number of rows from the output of `dim(A)`?

1.3 The `cbind()` and `rbind()` Functions

Another way to create a matrix is to use the **`cbind()`** and **`rbind()`** functions, which combine ("bind") columns and rows (respectively) into a matrix. Input each column or row values as a separate argument, as shown below:

```
# An alternative way to create the matrix A
cbind(c(1, 2), c(3, 4), c(5, 6))
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```r
# An alternative way to create the matrix B
rbind(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

If the number of columns or rows match in a way that makes sense, i.e., they are **conformable**, then `rbind()` and `cbind()` can also input two-dimensional objects and bind them together. Trying to combine matrices that are not conformable will produce an error.

```r
# Bind the rows of A and B together (i.e., stack A on top of B)
rbind(A, B)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    1    2    3
[4,]    4    5    6
[5,]    7    8    9
```

```r
# This will give an error, because the number of rows are not conformable
cbind(A, B)
```

```
Error in cbind(A, B): number of rows of matrices must match (see arg 2)
```

Side Note: The `rbind()` and `cbind()` functions also use recycling when necessary. This is particularly useful when inserting a column or row of repeated values. For an application to statistics: In linear regression (especially multiple regression), the observed values of the predictor variable(s) are often organized into a matrix, called the design matrix. The design matrix typically has a column of 1's to account for the intercept term in the linear model. An example of appending a column of 1's is given below.

```r
cbind(1, B) # Append a column of 1's to matrix B
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    3
[2,]    1    4    5    6
[3,]    1    7    8    9
```

1.4 Matrices Are Two-Dimensional Vectors

The reason why every value in a matrix must be of the same type is because *all matrix objects are actually internally stored as vectors* in R. This can be verified by using the `mode()` function on a matrix.

```r
mode(A) # A is stored as a numeric vector
```

```
[1] "numeric"
```

Matrices are just vectors with an additional **attribute** called dimension (`dim`). Vectors have no dimension attribute (i.e., vectors are not just one-dimensional matrices).

The `attributes()` function allows us to see the attributes of an R object.

```r
attributes(A)
```

```
$dim
[1] 2 3
```

```r
attributes(1:6)
```

```
NULL
```

We could strip the `A` matrix of its `dim` attribute by assigning `NULL` to its `attributes(A)` object. The matrix object will revert back to a vector.

```
attributes(A) <- NULL # Remove all of A's attributes
A # A is now a vector
```

```
[1] 1 2 3 4 5 6
```

We can also give a vector the `dim` attribute in a similar way to convert a vector into a matrix.

```
# Assign the dim attribute to A (with 2 rows and 3 columns)
attributes(A) <- list(dim = c(2, 3))
A # A is a matrix again
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```


2 Naming Rows and Columns of Two-Dimensional Objects

Suppose we have data on a few employees at the Pawnee Parks and Recreation Department, shown in a data table below.

Name	Height (inches)	Weight (pounds)	Income ($/month)
Leslie	62	115	4000
Ron	71	201	(Redacted)
April	66	119	2000

We can input the numeric data into a matrix in R.

```
parks_mat <- cbind(c(62, 71, 66), c(115, 201, 119), c(4000, NA, 2000))
parks_mat # Make sure the data was entered correctly
```



```
     [,1] [,2] [,3]
[1,]   62  115 4000
[2,]   71  201   NA
[3,]   66  119 2000
```

Question: Can you find at least two other ways to produce the same matrix of values in R?

While the numeric values from the dataset have been entered, the names of the employees and the names of the variables have been lost. To add names to the rows and columns of a two-dimensional object, we can use the **`rownames()`** and **`colnames()`** functions.

Applying the `rownames()` and `colnames()` functions to a two-dimensional object will retrieve (get), respectively, the current row and column names. If there are no names, the functions will output `NULL`.

```
rownames(parks_mat)
```

```
NULL
```

```
colnames(parks_mat)
```

```
NULL
```

To set the names, we can create a character vector of names and use the assignment `<-` operator on the row and column names.

```
rownames(parks_mat) <- c("Leslie", "Ron", "April")
colnames(parks_mat) <- c("Height", "Weight", "Income")
parks_mat
```

```
       Height Weight Income
Leslie     62    115   4000
Ron        71    201     NA
April      66    119   2000
```

Technically, these functions access the row and column name attributes of the input object. Setting names will add the `dimnames` attribute to the object.

```
attributes(parks_mat)
```

```
$dim
[1] 3 3

$dimnames
$dimnames[[1]]
[1] "Leslie" "Ron"    "April"

$dimnames[[2]]
[1] "Height" "Weight" "Income"
```

The `attributes(parks_mat)` object is an example of a list object in R. We will go over the syntax and functions for lists in a later chapter. For now, notice that the `dimnames` attribute has two components. The first component is the vector of row names, and the second component is the vector of column names.

Side Note: The `dimnames()` function can get and set both the row and column name attributes at once. The assignment input using `dimnames()` needs to be a list with two vector components.

```
dimnames(parks_mat) <- list(NULL, NULL) # Remove the row and column names
parks_mat
```

```
     [,1] [,2] [,3]
[1,]   62  115 4000
[2,]   71  201   NA
[3,]   66  119 2000
```

```
# Add the same names as before
dimnames(parks_mat) <- list(c("Leslie", "Ron", "April"), c("Height", "Weight", "Income"))
dimnames(parks_mat)
```

```
[[1]]
[1] "Leslie" "Ron"    "April"

[[2]]
[1] "Height" "Weight" "Income"
```

Side Note: There are a few other ways to add names to rows and columns. The `matrix()` function has an optional `dimnames` argument that allows us to add names directly when creating a matrix object. The syntax is the same as the `dimnames()` function.

```
matrix(1:9, nrow = 3, ncol = 3, dimnames = list(c("a", "b", "c"), c("A", "B", "C")))
```

```
  A B C
a 1 4 7
b 2 5 8
c 3 6 9
```

The `rbind()` and `cbind()` functions allow us to name, respectively, each row or column by just typing the name of the row or column in quotation marks, as shown below. Setting names for the unnamed dimension would still need to be done separately.

```
rbind("a" = 1:3, "b" = 4:6, "c" = 7:9)
```

```
  [,1] [,2] [,3]
a    1    2    3
b    4    5    6
c    7    8    9
```

```
cbind("A" = 1:3, "B" = 4:6, "C" = 7:9)
```

```
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```


3 Extracting Data From Two-Dimensional Objects

Recall that square brackets are used to extract specific parts of data from objects in R. Vectors are one-dimensional objects, so a single number inside of square brackets extracts that element from a vector.

For two-dimensional objects, there are typically two pieces of information corresponding to the desired row index and the column index. The index inside the square brackets needs to be an ordered pair, separated by a comma. For example, an index of `[i,j]` means to extract the entry in the `i`th row and `j`th column, also called the (i, j)th entry.

3.1 Numeric Indices

Leaving one value blank means to extract all the values in that dimension. Positive, negative, and fractional indices work the same way as they do for vectors. The row and column indices are independent of each other, so a positive index for one can be mixed with a negative index for the other.


```
B # Notice the row and column indices in the output
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```
B[2, 1] # Extract the (2,1) element
```

```
[1] 4
```

```
B[2, ] # Extract the second row
```

```
[1] 4 5 6
```

```
B[, 3] # Extract the third column
```

```
[1] 3 6 9
```

```
B[, -2] # Remove the second column
```

```
     [,1] [,2]
[1,]    1    3
[2,]    4    6
[3,]    7    9
```

```
B[-1, c(2, 3)] # Remove the first row and extract the second and third columns
```

```
     [,1] [,2]
[1,]    5    6
[2,]    8    9
```

Note: Notice that when the resulting output is one-dimensional (i.e., a single row or a single column), the output object is a vector, not a one-dimensional matrix.

Caution: Using a single index `[]` instead of an index pair `[,]` will not throw a warning or an error for matrix objects. Since a matrix is stored as one long (column) vector, R will interpret the single index as a vector index and extract the values from the stored vector version of the matrix. The matrix values are stored in **column-major** storage order, meaning the values are stored in order from top to bottom down the first column, then top to bottom down the second column, etc. Using a single index for matrices is not recommended, since it can be hard to keep track of which entry in a two-dimensional array corresponds to a single index.

```
B[8] # Do not extract elements of matrices this way
```

```
[1] 6
```


3.2 Logical Indices

Logical vectors can also be used for subsetting two-dimensional objects. They typically work the same way as they do for vectors. This can be a useful way to extract rows or columns that satisfy certain conditions.

```
# Which heights are above 65 inches?
tall_index <- parks_mat[, 1] > 65
# Extract only the rows/observations for people who are taller than 65 inches
parks_mat[tall_index, ]
```

```
      Height Weight Income
Ron       71    201     NA
April     66    119   2000
```

Question: How can we use the `tall_index` vector to extract only the rows/observations for people in the data who are at most 65 inches tall?

3.3 Named (Character) Indices

If the rows or columns of a two-dimensional object are named, we can use the name as an index. Row names can only be used as a row index, and column names can only be used as a column index.

```
parks_mat["Leslie", ] # Extract the row of data for Leslie
```

```
Height Weight Income
    62    115   4000
```

```
parks_mat[, "Income"] # Extract the column of data for Income
```

```
Leslie    Ron  April
  4000     NA   2000
```

```
parks_mat["Ron", "Height"] # Extract the height of Ron
```

```
[1] 71
```

```r
parks_mat[c("Leslie", "April"), "Weight"] # Extract the weight of Leslie and April
```

```
Leslie  April
   115    119
```

Note: Notice that we do not need to know the numeric index for the observations or variables. Using names as indices can be useful if the object has many rows or columns and the numeric index is not as easy to use.

4 Matrix Operations

4.1 Entrywise Arithmetic Operations

Because matrices are stored as vectors, the usual arithmetic operations (`+`, `-`, `*`, `/`, `^`, `%%`, `%/%`) that apply to numeric vectors will also work for numeric matrices. Just like for vectors, arithmetic operations will be applied entrywise. However, for arithmetic operations between matrices to make sense, the matrices need to be conformable. We can only apply the usual arithmetic operators to matrices with the same dimensions (same number of rows and columns). The corresponding entries of the two matrices will be operated together.


```r
A + 10 # Add 10 to every entry in A
```

```
     [,1] [,2] [,3]
[1,]   11   13   15
[2,]   12   14   16
```

```r
A^2 # Square each entry in A
```

```
     [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
```

```r
C <- matrix(1:3, nrow = 2, ncol = 3) # Construct a matrix C
C
```

```
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

```r
A + C # Add A and C
```

```
     [,1] [,2] [,3]
[1,]    2    6    7
[2,]    4    5    9
```

```r
A^C # Exponentiate A by C
```

```
     [,1] [,2] [,3]
[1,]    1   27   25
[2,]    4    4  216
```

```r
A * C # Multiply A and C (entrywise)
```

```
     [,1] [,2] [,3]
[1,]    1    9   10
[2,]    4    4   18
```

If we try to apply the operators to matrices that are not conformable, R will throw an error.

```r
A * B # A and B are not conformable
```

```
Error in A * B: non-conformable arrays
```

4.2 Matrix Multiplication

Caution: The entrywise multiplication performed using the regular multiplication `*` operator is *not* the way matrices are multiplied together in linear algebra.

For **matrix multiplication**, two matrices are conformable if the number of columns in the left matrix is the same as the number of rows in the right matrix. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then A and B are conformable, and AB will be an $m \times p$ matrix.

For example, let A be a 2×3 matrix and B be a 3×2 matrix, denoted by

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}.$$

The matrix product AB is the 2×2 matrix given by

$$AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{bmatrix}.$$

Written generally, if $A = [a_{ik}]_{m \times n}$ and $B = [b_{kj}]_{n \times p}$, then $AB = \left[\sum_{k=1}^{n} a_{ik} b_{kj} \right]_{m \times p}$. The (i, j)th entry of the product AB is the **dot product** of the ith row of A with the jth column of B.

Matrix multiplication is performed using the `%*%` operator.

```
A %*% C # Non-conformable error
```

```
Error in A %*% C: non-conformable arguments
```

```
A %*% B # Matrix multiplication of A and B
```

```
     [,1] [,2] [,3]
[1,]   48   57   66
[2,]   60   72   84
```

Notice that matrix `A` and `C` are conformable for entrywise multiplication but are not conformable for matrix multiplication, while `A` and `B` are conformable for matrix multiplication but not for entrywise.

4.3 The Identity Matrix

The **identity matrix** of size n, denoted by I_n (or simply I if the dimension is implicit) is an $n \times n$ square matrix with 1's in the main diagonal entries and 0 everywhere else. Formally, the (i, j)th entry of I_n is 1 if $i = j$ and 0 if $i \neq j$, written mathematically as

$$[I_n]_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$$

For example:

$$I_1 = \begin{bmatrix} 1 \end{bmatrix}, \quad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \ldots$$

The identity matrix essentially plays the role of the number 1 for matrix multiplication. That is, for any $n \times n$ matrix A, a property of matrix multiplication is that $AI_n = I_n A = A$.

The `diag()` function has two main functionalities. By inputting a number, the `diag()` function will generate an identity matrix of that size.

```r
diag(4) # Create a 4x4 identity matrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

By inputting a vector, the `diag()` function will generate a **diagonal matrix** (the only nonzero entries are along the diagonal) with the vector values along the diagonal.

```r
diag(c(1, 2, 3)) # Create a diagonal matrix with 1, 2, 3 along the diagonal
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```


The `nrow` and `ncol` arguments can also be used to specify the dimensions for a rectangular (non-square) matrix.


```r
diag(c(1, 2, 3), nrow = 3, ncol = 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
```


4.4 The Inverse of a Matrix

The **inverse** of an $n \times n$ square matrix A is the unique $n \times n$ matrix, denoted by A^{-1}, such that

$$AA^{-1} = A^{-1}A = I_n$$

The inverse of a matrix is analogous to the reciprocal (also called the *multiplicative inverse*) for numbers. For example, the reciprocal of 2 is $2^{-1} = \frac{1}{2}$, since $2 \times \frac{1}{2} = \frac{1}{2} \times 2 = 1$.

The function **`solve()`** computes the inverse of the inputted matrix. For example:

```r
M <- matrix(c(1, 4, 2, 1), nrow = 2, ncol = 2) # Create a matrix M
M
```

```
     [,1] [,2]
[1,]    1    2
[2,]    4    1
```

```r
M_inv <- solve(M) # Compute the inverse of M
M_inv
```

```
           [,1]       [,2]
[1,] -0.1428571  0.2857143
[2,]  0.5714286 -0.1428571
```

```r
# Verify that M_inv is the inverse of M
M %*% M_inv
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
M_inv %*% M
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Caution: The inverse of a (square) matrix does not always exist. A matrix that has an inverse is called **invertible** or **nonsingular**. There are conditions in linear algebra for when an inverse exists. A matrix that does not have an inverse is called **singular**. If we try to invert a singular matrix, R will throw an error.

```
solve(B)
```

```
Error in solve.default(B): system is computationally singular: reciprocal condition
number = 2.59052e-18
```


5 Operations on Matrix Columns and Rows

5.1 The `apply()` Function

Suppose we want to compute the mean of each variable in the `parks_mat` matrix.

```
parks_mat
```



```
       Height Weight Income
Leslie     62    115   4000
Ron        71    201     NA
April      66    119   2000
```


We could compute these individually, but it would require repetitive code (or a for loop).

```
mean(parks_mat[, "Height"], na.rm = TRUE) # Or mean(parks_mat[, 1], na.rm = TRUE)
```

```
[1] 66.33333
```



```
mean(parks_mat[, "Weight"], na.rm = TRUE) # Or mean(parks_mat[, 2], na.rm = TRUE)
```

```
[1] 145
```

```
mean(parks_mat[, "Income"], na.rm = TRUE) # Or mean(parks_mat[, 3], na.rm = TRUE)
```

```
[1] 3000
```

For large matrices (or other data objects you will see later), using repetitive code is inefficient and cumbersome.

The `apply()` function is used to apply a function to the rows or columns (the **margins**) of matrices, arrays (higher dimension matrices), and data frames (which you will see soon).

Similar to `vapply()`, the syntax of `apply()` is `apply(X, MARGIN, FUN, ...)`, where the arguments are:

- `X`: A matrix or data frame
- `MARGIN`: A vector giving the subscript(s) over which the function will be applied over. A `1` indicates rows, `2` indicates columns, and `c(1, 2)` indicates rows and columns.
- `FUN`: The function to be applied.
- `...`: Any optional arguments to be passed to the `FUN` function (for example, `na.rm=TRUE`).

Using `apply()`, we can apply the `mean()` function to each column in `parks_mat` simultaneously with a single command.

```
# Compute the mean of every column of the parks_mat matrix
apply(parks_mat, 2, mean, na.rm = TRUE)
```

```
    Height     Weight     Income
  66.33333  145.00000 3000.00000
```

To compute the mean of each row, we can change the margin argument `MARGIN` from `2` (columns) to `1` (rows).

```
# Compute the mean of every row of the parks_mat matrix
apply(parks_mat, 1, mean, na.rm = TRUE)
```

```
   Leslie       Ron     April
1392.3333  136.0000  728.3333
```

Side Note: Technically, the mean of every row or column in a matrix (or data frame) can also be computed using the `rowMeans()` and `colMeans()` functions, but `apply()` works more generally, since `apply()` allows us to apply any function, not just `mean()`.

Note: If the applied function in the `FUN` argument of `apply()` returns a single value, the output of the `apply()` function will be a vector. If the applied function returns a vector (with more than one element), then the output of `apply()` will be a matrix.

```
# Compute the range (min and max) of every column of the parks_mat matrix
apply(parks_mat, 2, range, na.rm = TRUE)
```

```
     Height Weight Income
[1,]     62    115   2000
[2,]     71    201   4000
```

Note: The `FUN` argument does not have to be a built-in function. We can also write our own function and apply it to each row or column.

As an example, suppose we want to compute the squared deviations from the mean for each variable in the `parks_mat` matrix.

```
squared_devs <- function(x, na.rm = FALSE) {
  # This function inputs a vector and computes the squared deviations away from the mean.
  (x - mean(x, na.rm = na.rm))^2
}
# Apply the squared_devs() function to every column of the parks_mat matrix
apply(parks_mat, 2, squared_devs, na.rm = TRUE)
```

```
           Height Weight Income
Leslie 18.7777778    900  1e+06
Ron    21.7777778   3136     NA
April   0.1111111    676  1e+06
```

The function can also be written directly into the `FUN` argument without having to save it as a separate object.

```
# Creates the same object as apply(parks_mat, 2, squared_devs, na.rm = TRUE)
apply(parks_mat, 2, function(x) {
  (x - mean(x, na.rm = TRUE))^2
})
```

```
           Height Weight Income
Leslie 18.7777778    900  1e+06
Ron    21.7777778   3136     NA
April   0.1111111    676  1e+06
```