MATRICES AND ARRAYS

Introduction

- A matrix is a vector with two additional attributes: the number of rows and the number of columns.

- Since matrices are vectors, they also have modes, such as numeric and character. (On the other hand, vectors are not one column or one-row matrices.)

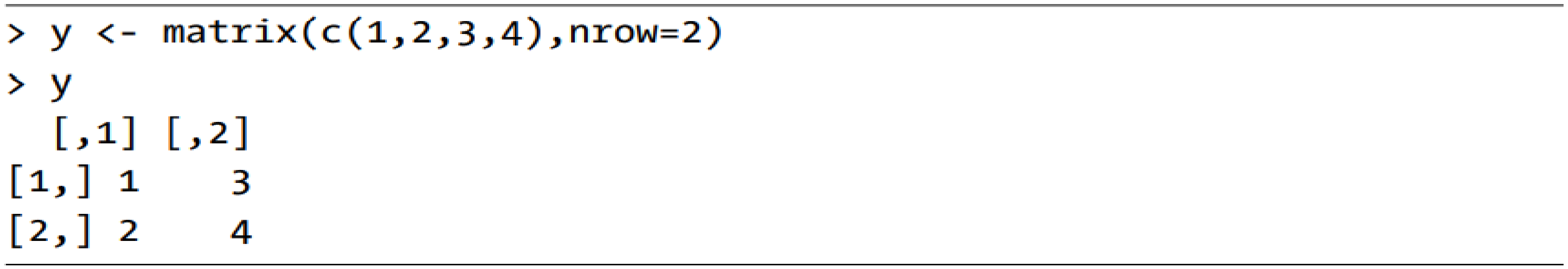
Creating Matrices

One way to create a matrix is by using the `matrix()` function:

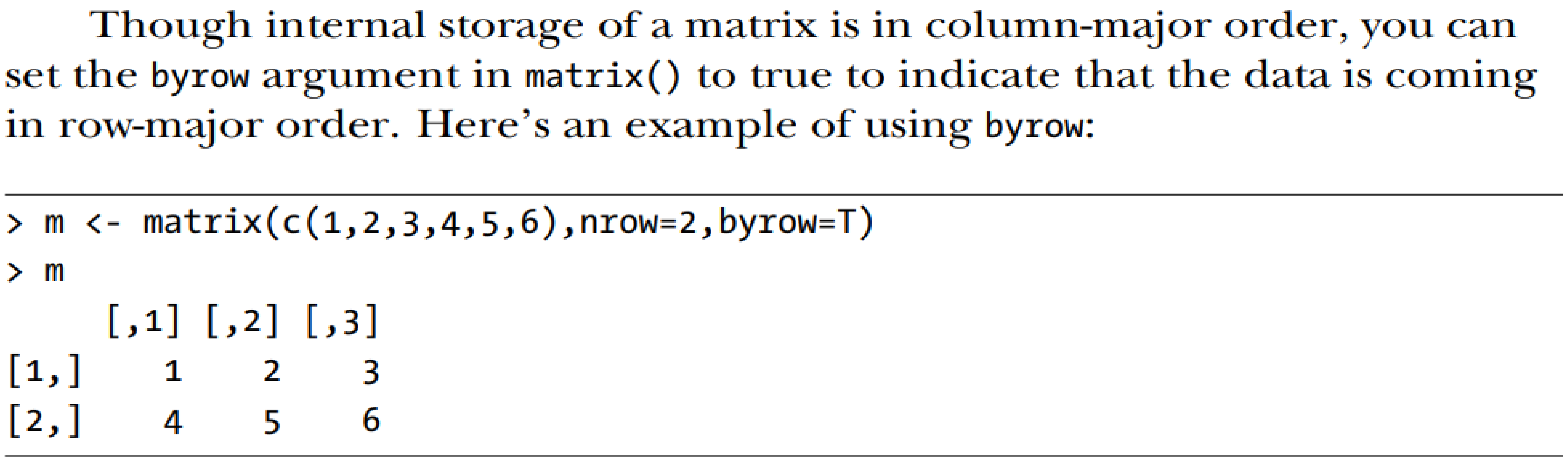
```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Here, we concatenate what we intend as the first column, the numbers 1 and 2, with what we intend as the second column, 3 and 4. So, our data is `(1,2,3,4)`. Next, we specify the number of rows and columns. The fact that R uses column-major order then determines where these four numbers are put within the matrix.

- We did not need to specify both ncol and nrow; just nrow or ncol would have been enough. Having four elements in all, in two rows, implies two columns.



```
> y <- matrix(c(1,2,3,4),nrow=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```



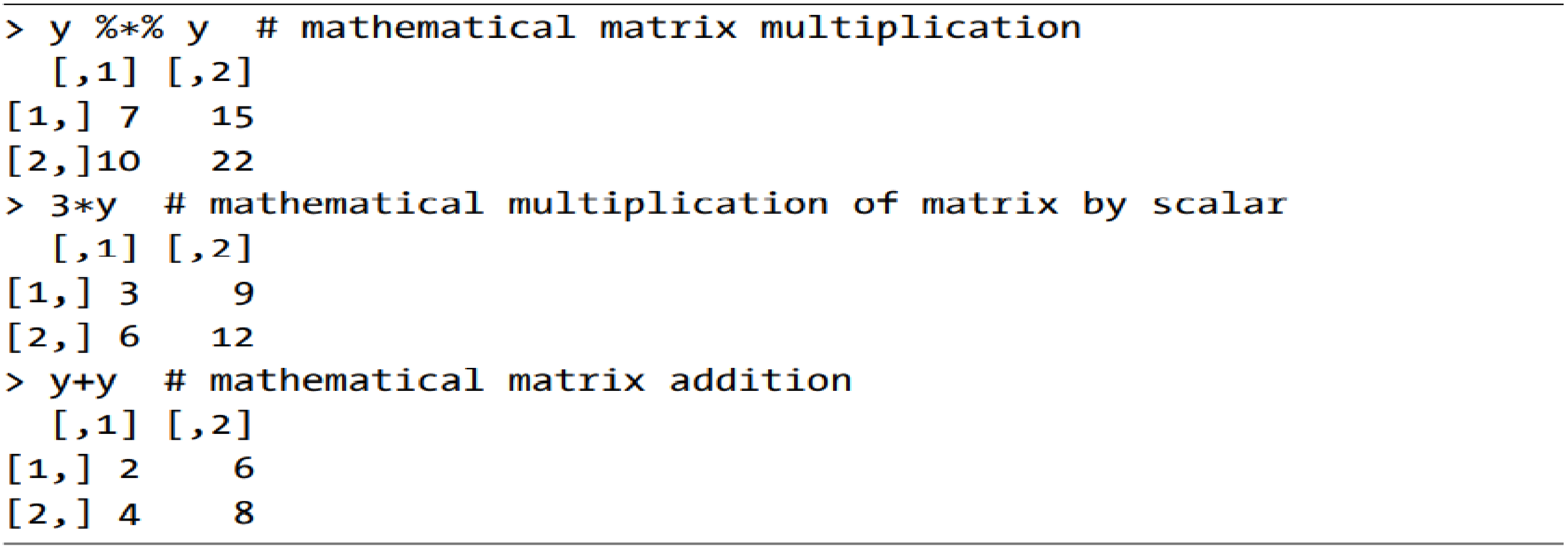
Though internal storage of a matrix is in column-major order, you can set the `byrow` argument in `matrix()` to true to indicate that the data is coming in row-major order. Here's an example of using `byrow`:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

General Matrix Operations

Performing Linear Algebra Operations on Matrices

- You can perform various linear algebra operations on matrices, such as matrix multiplication, matrix scalar multiplication, and matrix addition.
- Using y from the preceding example, here is how to perform those three operations:



```
> y %*% y  # mathematical matrix multiplication
  [,1] [,2]
[1,] 7   15
[2,]10   22
> 3*y  # mathematical multiplication of matrix by scalar
  [,1] [,2]
[1,] 3    9
[2,] 6   12
> y+y  # mathematical matrix addition
  [,1] [,2]
[1,] 2    6
[2,] 4    8
```

Matrix Indexing

```
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> z[,2:3]
  [,1] [,2]
[1,] 1    1
[2,] 1    0
[3,] 0    1
[4,] 0    0
```

Here, we requested the submatrix of z consisting of all elements with column numbers 2 and 3 and any row number. This extracts the second and third columns.

Here's an example of extracting rows instead of columns:

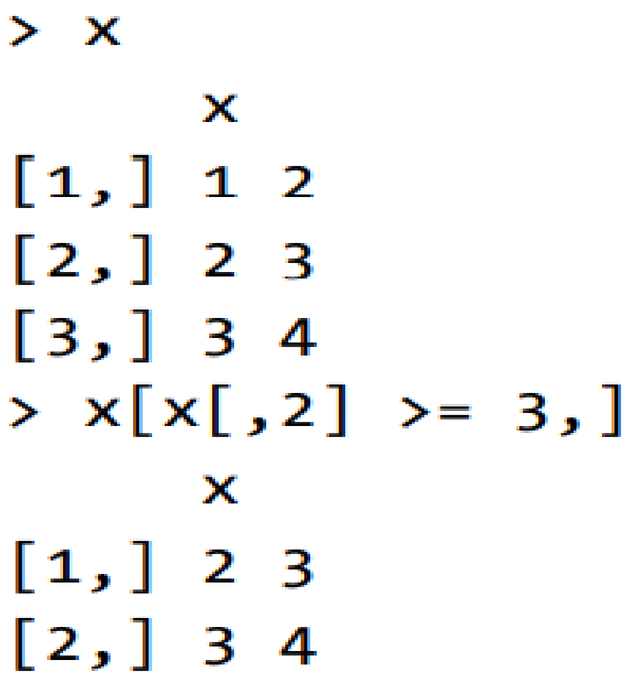
```
> y
  [,1] [,2]
[1,]11   12
[2,]21   22
[3,]31   32
> y[2:3,]
  [,1] [,2]
[1,]21   22
[2,]31   32
```

You can also assign values to submatrices:

```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)
> y
     [,1] [,2]
[1,]    1    8
[2,]    2    5
[3,]    1   12
```

Filtering on Matrices

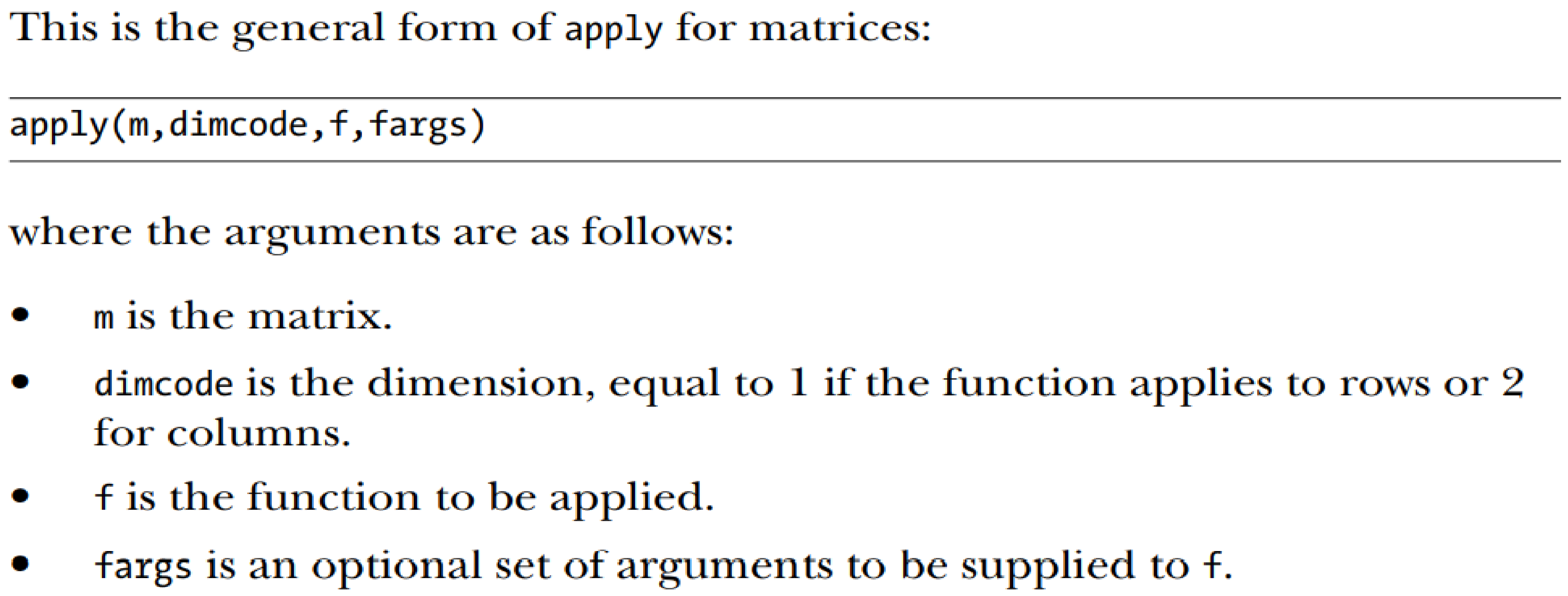
- Filtering can be done with matrices, just as with vectors.



```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

Applying Functions to Matrix Rows and Columns

- One of the most famous and most used features of R is the *apply() family of functions, such as apply(), tapply(), and lapply().
- apply() , which instructs R to call a user-specified function on each of the rows or each of the columns of a matrix.



This is the general form of `apply` for matrices:

```
apply(m,dimcode,f,fargs)
```

where the arguments are as follows:

- `m` is the matrix.
- `dimcode` is the dimension, equal to 1 if the function applies to rows or 2 for columns.
- `f` is the function to be applied.
- `fargs` is an optional set of arguments to be supplied to `f`.

For example, here we apply the R function `mean()` to each column of a matrix `z`:

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

> apply(z,2,mean)
[1] 2 5
```

In this case, we could have used the `colMeans()` function, but this provides a simple example of using `apply()`.

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

Our f() function divides a two-element vector by the vector (2,8). (Recycling would be used if x had a length longer than 2.) The call to apply() asks R to call f() on each of the rows of z. The first such row is (1,4), so in the call to f(), the actual argument corresponding to the formal argument x is (1,4). Thus, R computes the value of (1,4)/(2,8), which in R's element-wise vector arithmetic is (0.5,0.5). The computations for the other two rows are similar.

- You may have been surprised that the size of the result here is 2 by 3 rather than 3 by 2.

- That first computation, (0.5,0.5), ends up at the first column in the output of apply(), not the first row.

- But this is the behavior of apply().

- If the function to be applied returns a vector of k components, then the result of apply() will have k rows.

Adding and Deleting Matrix Rows and Columns

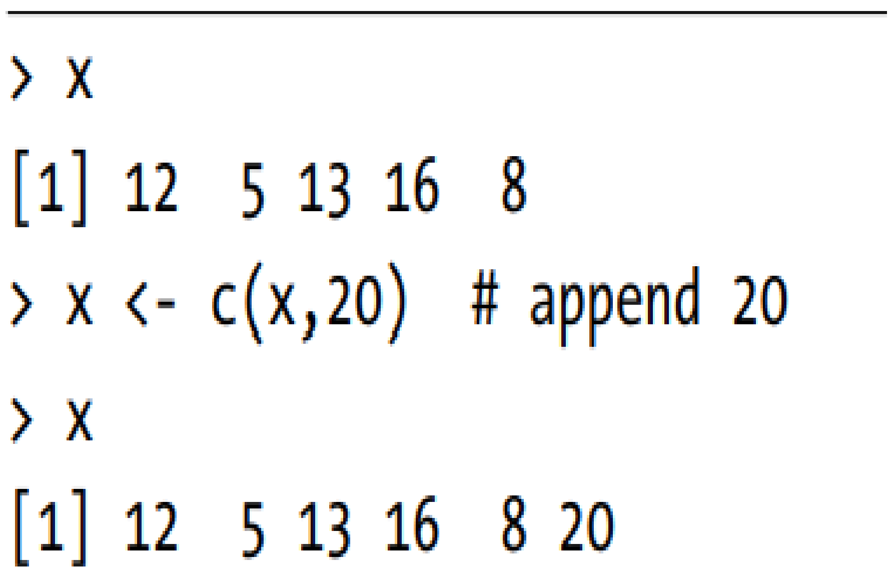
- Deletion or addition of rows and columns in a matrix of any size is mostly done by using single square brackets and it is also the easiest way.

- To delete rows and columns, we just need to use the column index or row index and if we want to delete more than one of them then we can separate them by commas by inserting them inside c as c(-1,-2).

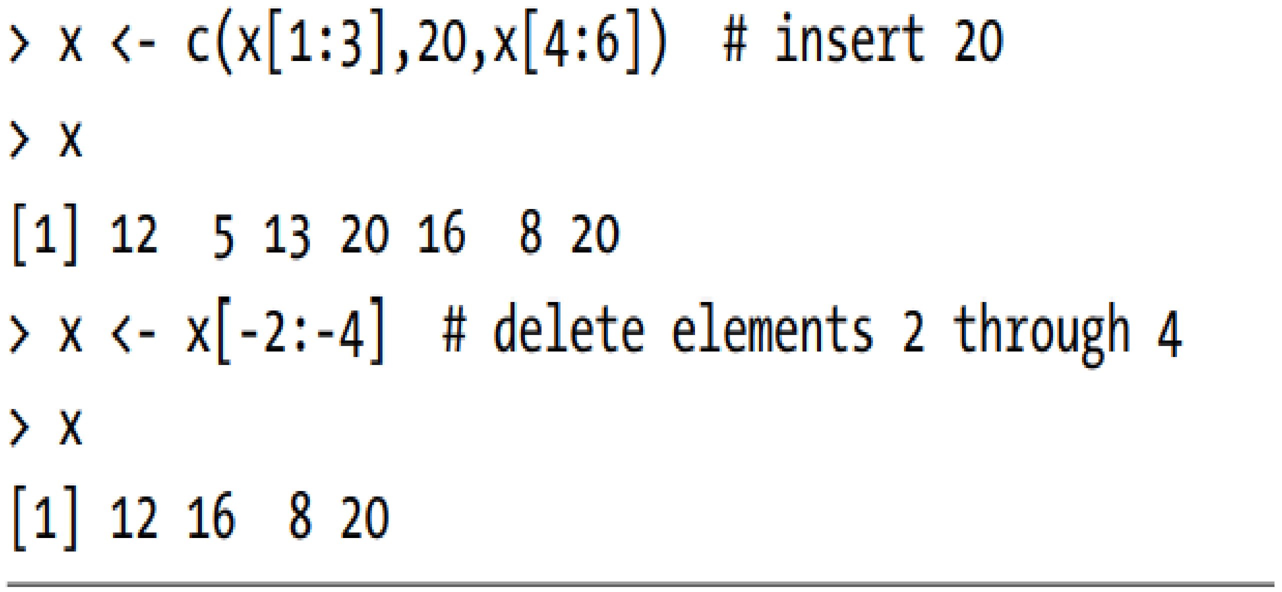
- If we want to delete more than one rows or columns in a sequence then a colon can be used.

- Technically, matrices are of fixed length and dimensions, so we cannot add or delete rows or columns.
- However, matrices can be reassigned, and thus we can achieve the same effect as if we had directly done additions or deletions.

Changing the Size of a Matrix

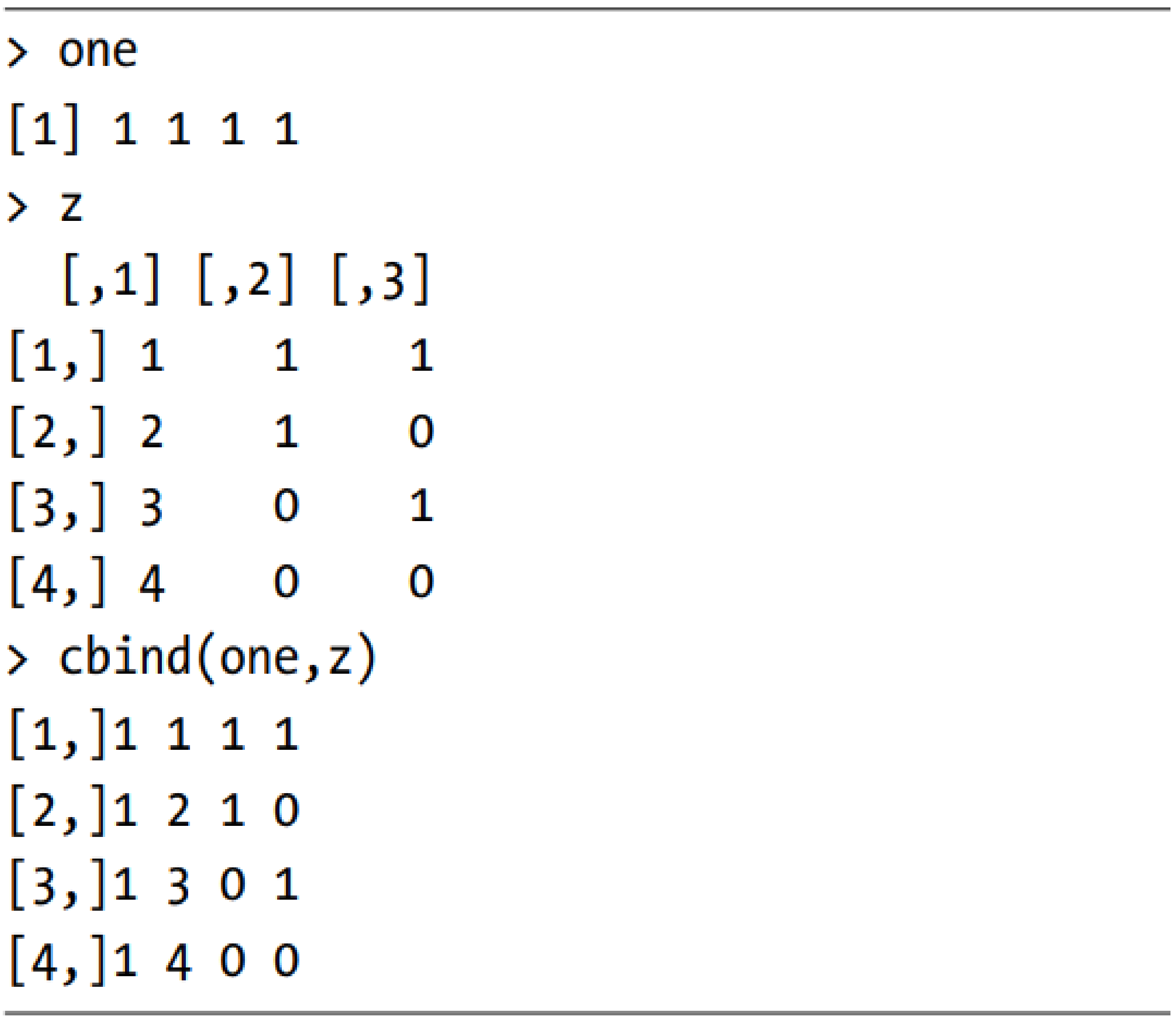


```
> x
[1] 12  5 13 16  8
> x <- c(x,20)  # append 20
> x
[1] 12  5 13 16  8 20
```



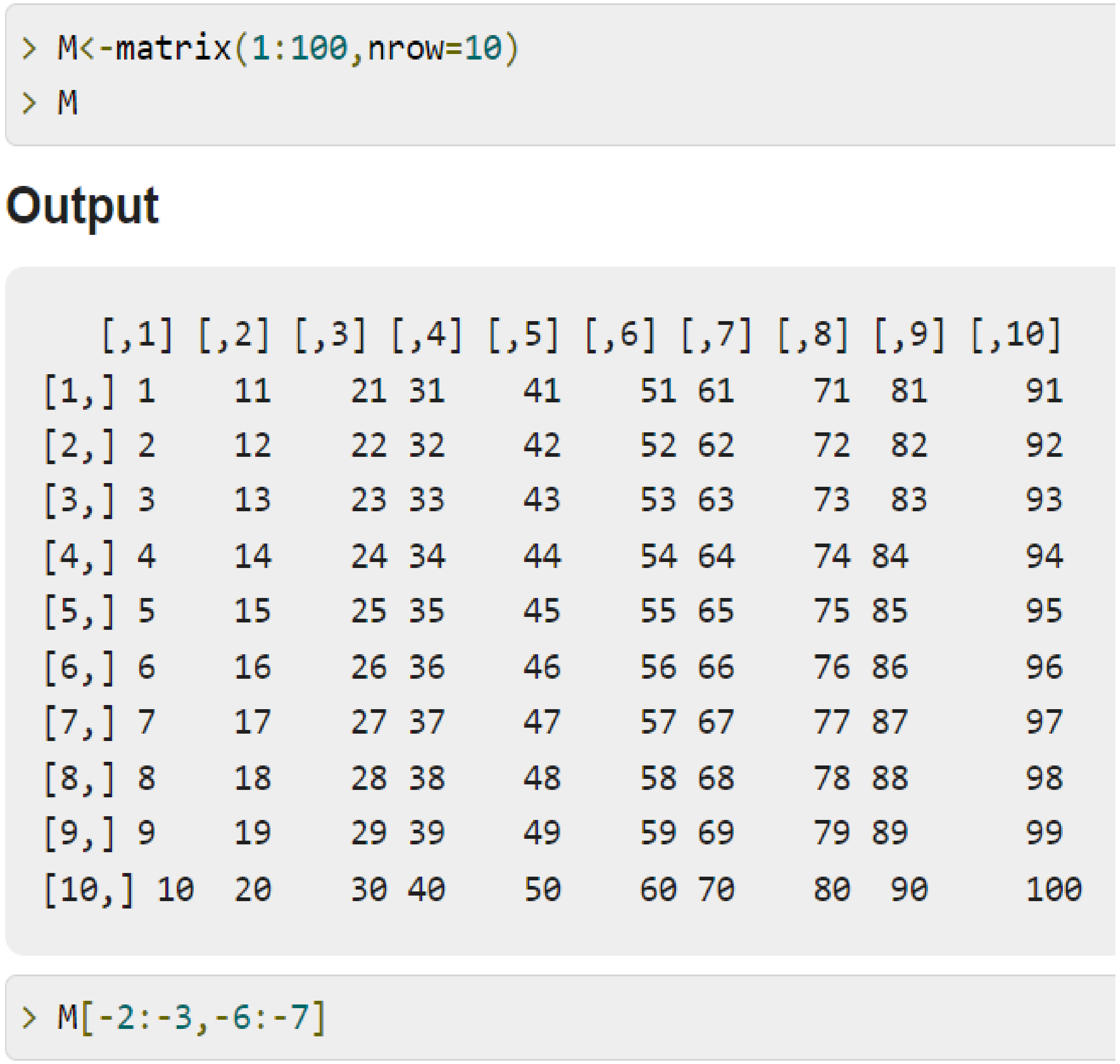
```
> x <- c(x[1:3],20,x[4:6])  # insert 20
> x
[1] 12  5 13 20 16  8 20
> x <- x[-2:-4]  # delete elements 2 through 4
> x
[1] 12 16  8 20
```

- Analogous operations can be used to change the size of a matrix. For instance, the rbind() (row bind) and cbind() (column bind) functions



```
> one
[1] 1 1 1 1
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> cbind(one,z)
[1,]1 1 1 1
[2,]1 2 1 0
[3,]1 3 0 1
[4,]1 4 0 0
```

Deleting row or column

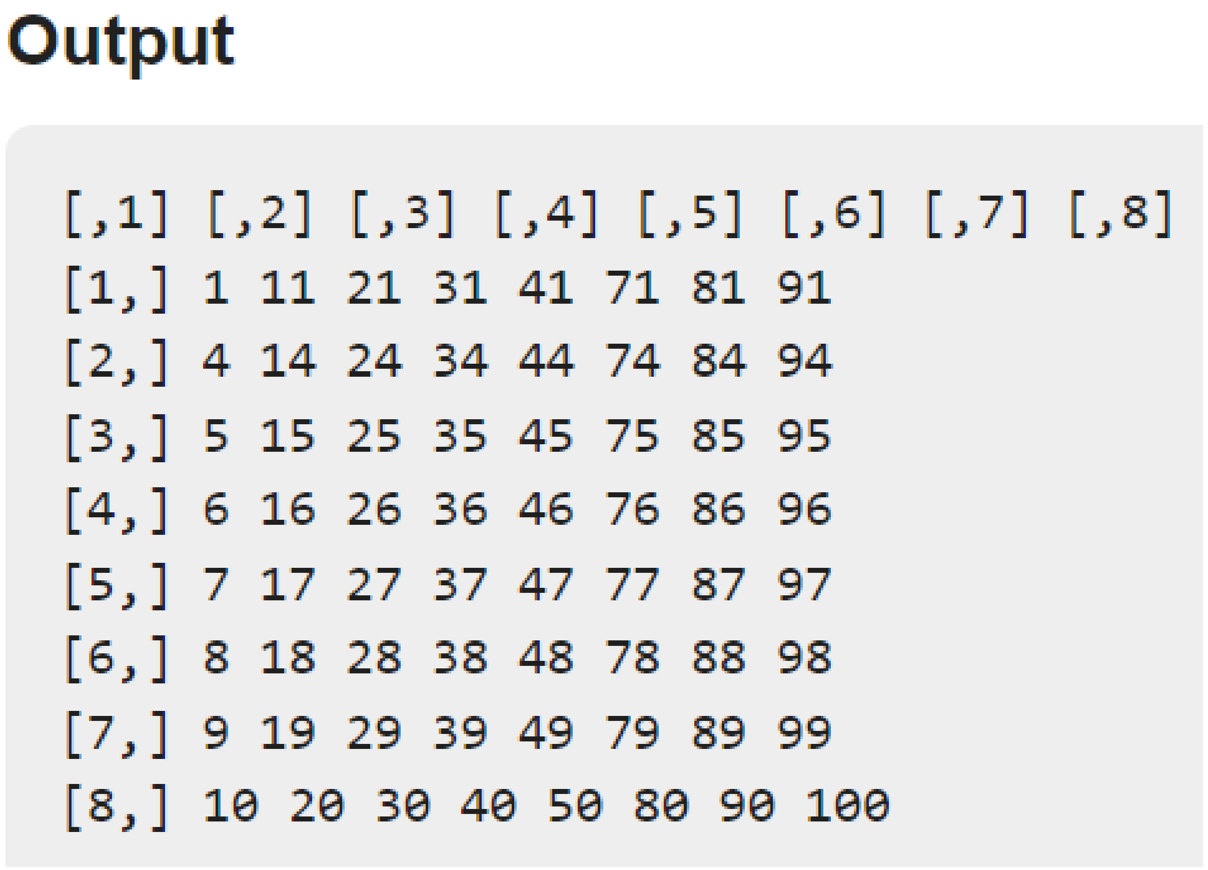


```
> M<-matrix(1:100,nrow=10)
> M
```

Output

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 1     11     21  31     41     51  61     71   81        91
[2,] 2     12     22  32     42     52  62     72   82        92
[3,] 3     13     23  33     43     53  63     73   83        93
[4,] 4     14     24  34     44     54  64     74  84         94
[5,] 5     15     25  35     45     55  65     75  85         95
[6,] 6     16     26  36     46     56  66     76  86         96
[7,] 7     17     27  37     47     57  67     77  87         97
[8,] 8     18     28  38     48     58  68     78  88         98
[9,] 9     19     29  39     49     59  69     79  89         99
[10,] 10   20     30  40     50     60  70     80   90        100
```

```
> M[-2:-3,-6:-7]
```



Output

```
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] 1 11 21 31 41 71 81 91
[2,] 4 14 24 34 44 74 84 94
[3,] 5 15 25 35 45 75 85 95
[4,] 6 16 26 36 46 76 86 96
[5,] 7 17 27 37 47 77 87 97
[6,] 8 18 28 38 48 78 88 98
[7,] 9 19 29 39 49 79 89 99
[8,] 10 20 30 40 50 80 90 100
```

Avoiding Unintended Dimension Reduction

In the world of statistics, dimension reduction is a good thing, with many statistical procedures aimed to do it well. If we are working with, say, 10 variables and can reduce that number to 3 that still capture the essence of our data, we're happy.

However, in R, something else might merit the name *dimension reduction* that we may sometimes wish to avoid. Say we have a four-row matrix and extract a row from it:

```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> r <- z[2,]
> r
[1] 2 6
```

```
> attributes(z)
$dim
[1] 4 2
> attributes(r)
NULL
> str(z)
 int [1:4, 1:2] 1 2 3 4 5 6 7 8
> str(r)
 int [1:2] 2 6
```

Here, R informs us that z has row and column numbers, while r does not. Similarly, str() tells us that z has indices ranging in 1:4 and 1:2, for rows and columns, while r's indices simply range in 1:2. No doubt about it—r is a vector, not a matrix.

Fortunately, R has a way to suppress this dimension reduction: the drop argument. Here's an example, using the matrix z from above:

```
> r <- z[2,, drop=FALSE]
> r
     [,1] [,2]
[1,]    2    6
> dim(r)
[1] 1 2
```

Now r is a 1-by-2 matrix, not a two-element vector.

For these reasons, you may find it useful to routinely include the drop=FALSE argument in all your matrix code.

If you have a vector that you wish to be treated as a matrix, you can use the `as.matrix()` function, as follows:

```
> u
[1] 1 2 3
> v <- as.matrix(u)
> attributes(u)
NULL
> attributes(v)
$dim
[1] 3 1
```

Naming Matrix Rows and Columns

- The natural way to refer to rows and columns in a matrix is via the row and column numbers. However, you can also give names to these entities. Here's an example:

```
> z
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> colnames(z)
NULL
> colnames(z) <- c("a","b")
```

```
> z
     a b
[1,] 1 3
[2,] 2 4
> colnames(z)
[1] "a" "b"
> z[,"a"]
[1] 1 2
```

Higher-Dimensional Arrays

As a simple example, consider students and test scores. Say each test consists of two parts, so we record two scores for a student for each test. Now suppose that we have two tests, and to keep the example small, assume we have only three students. Here's the data for the first test:

```
> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   50
```

Student 1 had scores of 46 and 30 on the first test, student 2 scored 21 and 25, and so on. Here are the scores for the same students on the second test:

```
> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   50
```

Now let's put both tests into one data structure, which we'll name tests. We'll arrange it to have two "layers"—one layer per test—with three rows and two columns within each layer. We'll store firsttest in the first layer and secondtest in the second.

In layer 1, there will be three rows for the three students' scores on the first test, with two columns per row for the two portions of a test. We use R's array function to create the data structure:

```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

In the argument dim=c(3,2,2), we are specifying two layers (this is the second 2), each consisting of three rows and two columns. This then becomes an attribute of the data structure:

```
> attributes(tests)
$dim
[1] 3 2 2
```

LISTS

Creating Lists

- Technically, a list is a vector.
- Ordinary vectors—those of the type are termed atomic vectors, since their components cannot be broken down into smaller components.
- In contrast, lists are referred to as recursive vectors.

General List Operations

- List Indexing
- Adding and Deleting List Elements
- Getting the Size of a List

Getting the Size of a List

Using the lapply() and sapply() Functions

The function lapply() (for *list apply*) works like the matrix apply() function, calling the specified function on each component of a list (or vector coerced to a list) and returning another list. Here's an example:

```
> lapply(list(1:3,25:29),median)
[[1]]
[1] 2

[[2]]
[1] 27
```

R applied median() to 1:3 and to 25:29, returning a list consisting of 2 and 27.

In some cases, such as the example here, the list returned by lapply() could be simplified to a vector or matrix. This is exactly what sapply() (for *simplified [l]apply*) does.

```
> sapply(list(1:3,25:29),median)
[1]  2 27
```

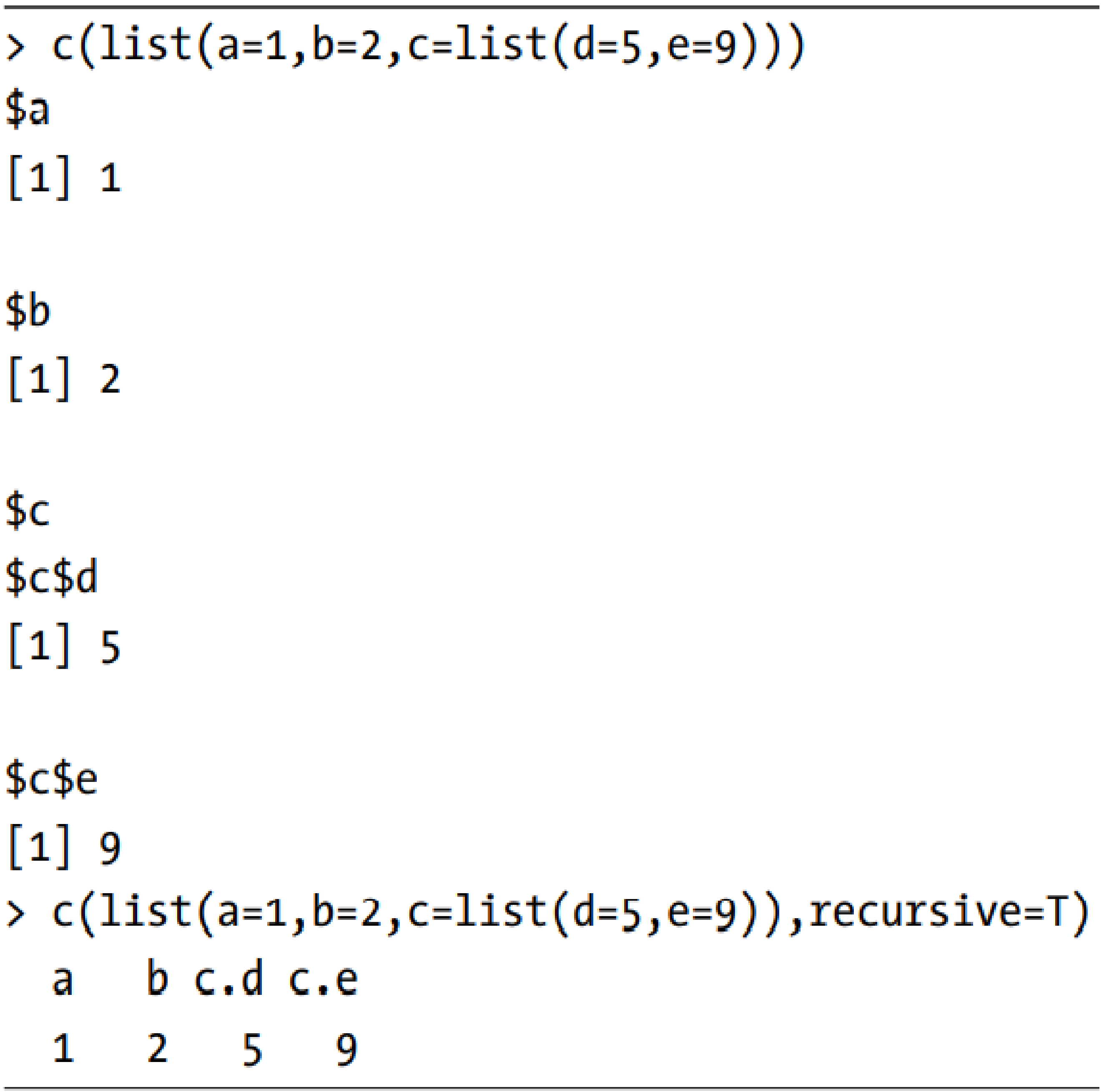
Recursive Lists

- Lists can be recursive, meaning that you can have lists within lists. Here’s an example:

```
> b <- list(u = 5, v = 12)
> c <- list(w = 13)
> a <- list(b,c)
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12


[[2]]
[[2]]$w
[1] 13


> length(a)
[1] 2
```

This code makes a into a two-component list, with each component itself also being a list.

The concatenate function `c()` has an optional argument `recursive`, which controls whether *flattening* occurs when recursive lists are combined.



```
> c(list(a=1,b=2,c=list(d=5,e=9)))
$a
[1] 1

$b
[1] 2

$c
$c$d
[1] 5

$c$e
[1] 9
> c(list(a=1,b=2,c=list(d=5,e=9)),recursive=T)
  a   b c.d c.e 
  1   2   5   9 
```

In the first case, we accepted the default value of recursive, which is FALSE, and obtained a recursive list, with the c component of the main list itself being another list. In the second call, with recursive set to TRUE, we got a single list as a result; only the names look recursive.