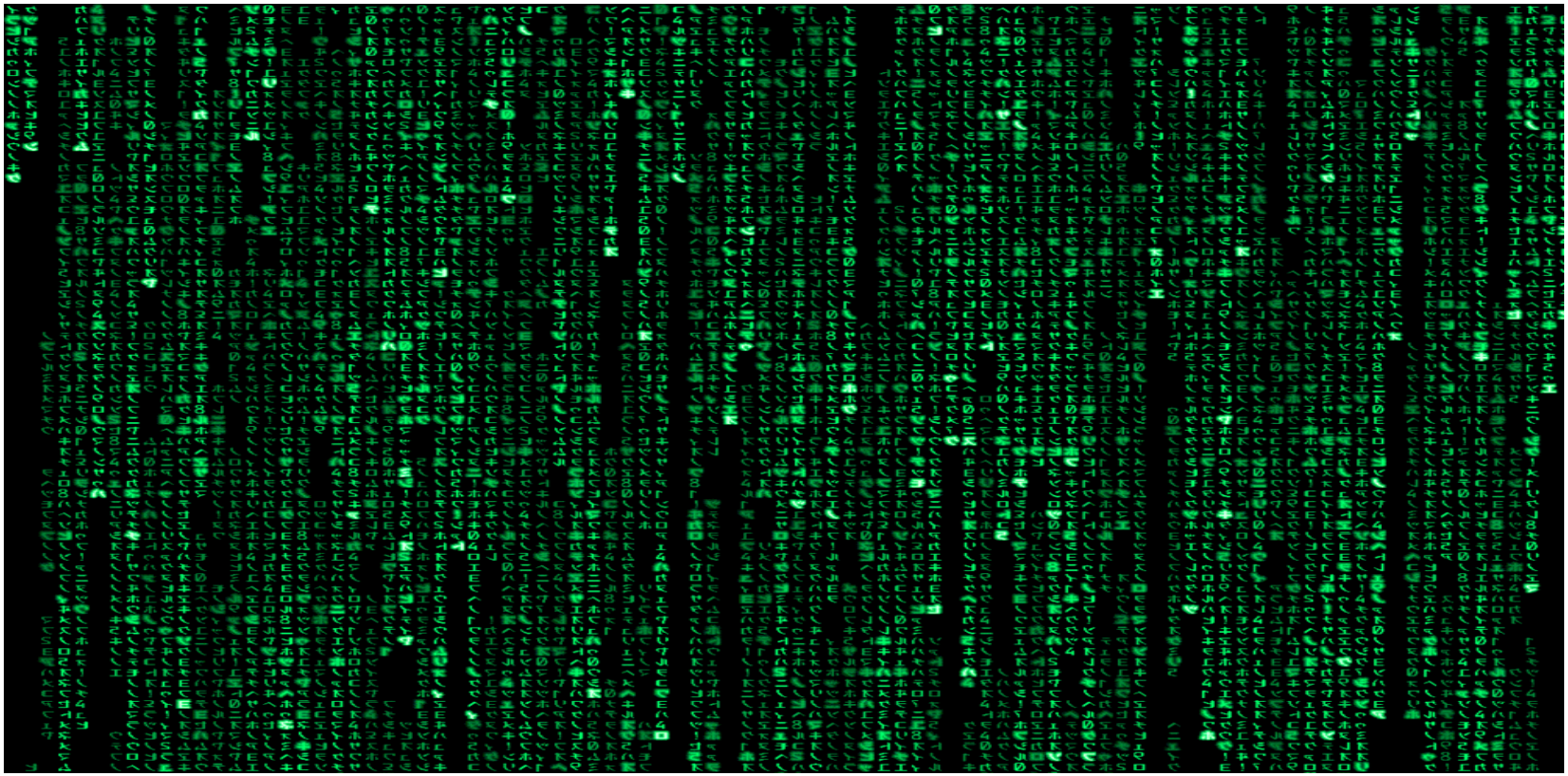
Introduction to Computing for Economics and Management

Lecture 3: Matrices



Acknowledgement

- These slides are adapted from Bert Arnrich's R lecture.

Previous lecture: variable name conventions

- Variable names can contain letters, digits, and the dot symbol
  - Name must not start with a digit
  - Name must not start with a dot followed by a digit
  - Since names that start with a dot are special, we should not introduce them in our scripts to avoid confusion
  - Some names are already used by the system

- Better use descriptive names like `person.height` instead of just `h`

Names are case-sensitive, e.g. `x` and `X` do not refer to the same variable

Previous lecture: data vectors

- The fundamental data type in R is the vector

Data vectors are created with the construct `c`

```
> person.height <- c(1.70, 1.75, 1.62)

> person.height <- c(person.height, 1.81)
```

- Vector elements must all have the same mode
- Available modes: integer, numeric, character, Boolean, complex

Previous lecture: data vectors

- Missing values are denoted with `NA`

We can assign names to the elements of a data vector to make the vector more readable

```
> person.height <- c(Can=1.70, Cem=1.75,
Hande=1.62)

> person.height
  Can   Cem Hande
 1.70  1.75  1.62
```

Previous lecture: data vector indexing

We can access a single element of a vector by providing the index of the element in square brackets

```
> person.height[1]
Can
 1.7
```

We can select a subvector by providing a Boolean index vector

```
> person.height[c(T,F,T)]
  Can Hande
  1.70  1.62
```

Previous lecture: data vector indexing

We can specify the element indices directly

```
> person.height[c(1,3)]
  Can Hande
 1.70  1.62
```

- We exclude elements with negative indices
- `> person.height[c(-1, -3)]`
- `Cem`
- `1.75`

We can change the values of the selected elements

```
person.height[1] <- 1.72
```

Previous lecture: data vector filtering

The idea behind filtering is to apply a Boolean evaluation function to each element of the vector

```
> person.height > 1.65
   Can    Cem Hande
 TRUE   TRUE FALSE
```

We use the results of the evaluation function for the filtering

```
> person.height[person.height > 1.65]
 Can  Cem
1.72 1.75
```

Previous lecture: data vector sorting

We use the function `sort` for sorting a vector

```
> sort(person.height)
Hande   Can   Cem
 1.62  1.70  1.75
```

- We can obtain a sorting in descending order
- `> sort(person.height, decreasing = TRUE)`
- `  Cem   Can Hande`
- ` 1.75  1.70  1.62`

- We can sort a vector according to the values of some other vector
- `> person.weight[order(person.height)]`
- `Hande   Can   Cem`
- `   61    65    66`

Previous lecture: vector recycling

When applying an operation to two vectors which requires them to be the same length, the shorter one will repeated until it is long enough to match the longer one

```
> c(1, 2, 3) + c(1, 2, 3, 4)
[1] 2 4 6 5

Warning message:
In c(1, 2, 3) + c(1, 2, 3, 4) :
  longer object length is not a multiple of shorter object length
```

Program today

- `ifelse ()`
- More vector functions
- Matrix creation
- Matrix operations
- Matrix indexing
- Matrix filtering
- Matrix function `apply()`
- Writing own functions
- Differences between vectors and matrices
- Higher-dimensional arrays

Conditional element selection with the ifelse() function

- We provide the `ifelse(test, yes, no)` function with a Boolean vector `test` and two vectors `yes` and `no`

- `ifelse` returns a vector which is created from selected elements from the vectors `yes` and `no`: `yes[i]` is selected if `test[i]` is true and `no[i]` is selected if `test[i]` is false

Example (which uses recycling):

```
> ifelse(person.height > 1.7, "tall", "small")
    Can     Cem   Hande
- "small"  "tall" "small"
```

Conditional element selection with the ifelse() function

- We provide the `ifelse(test, yes, no)` function with a Boolean vector `test` and two vectors `yes` and `no`

- `ifelse` returns a vector which is created from selected elements from the vectors `yes` and `no`: `yes[i]` is selected if `test[i]` is true and `no[i]` is selected if `test[i]` is false

Example (which uses recycling):

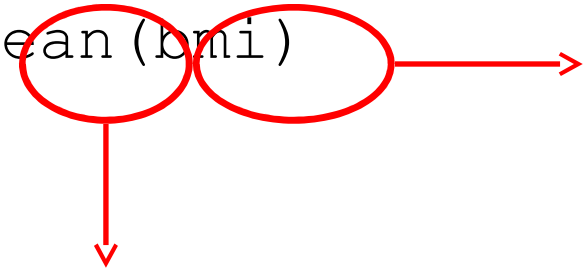
```
> ifelse(person.height > 1.7, "tall", "small")
    Can     Cem   Hande
- "small"  "tall" "small"
```

More data vector operations

- An often used functions that operates on vectors is `mean`

For example, we can compute the mean body mass index

```
> bmi <- person.weight / person.height^2

> mean(bmi)
[1] 23.31768

> mean(bmi)
```



Arguments of the function are provided in parentheses

-

R function mean

Data vector operations

Other examples of functions are `length` and `sd` which compute the length and the standard deviation of a vector

```
> bmi <- person.weight / person.height^2

> mean(bmi)
[1] 23.31768

> length(bmi)
[1] 3

> sd(bmi)
[1] 2.294295
```

Creating regular sequences

- In particular for graphics, we often need equidistant series of numbers
- For example, let's assume we need to specify the x-coordinates of a curve as 1.65,1.70,1.75,1.80,1.85, and 1.90
- In particular for long series, we are looking for way to specify the regular sequence in compact way

Creating regular sequences

The `from:to` syntax is a simple way to generate a sequence from `from` to `to` in steps of 1 or -1

```
> 1:5
[1] 1 2 3 4 5

> 11:15
[1] 11 12 13 14 15

> 3:0
[1] 3 2 1 0
```

- `> seq_1_100 <- 1:100`

Function seq

- Function `seq` allows to generate regular sequences with more options
- The help page `help(seq)` shows us the full function documentation
- Similar to many other R functions, in the parentheses we can provide arguments as a comma-separated list

The full set of arguments can be seen from the help page

- `seq(from, to, by, length.out, along.with)`

Function arguments

- `seq` function arguments
  - `from` starting value of the sequence
  - `to` end value of the sequence
  - `by` increment of the sequence
  - `length.out` desired length of the sequence
  - `along.with` take the length from the length of this argument

- Like with many other R functions, arguments have default values
  - `from = 1`
  - `to = 1`
  - `by = ((to - from)/(length.out - 1))`
  - `length.out = NULL`
  - `along.with = NULL`

Function arguments

- Usually, we only provide those arguments which values should be different from the default values

For example, let's specify only the first two arguments `from` and `to`

```
> seq(5, 10)
```

- `[1] 5 6 7 8 9 10`

- We observe that default value `by=1` was automatically used

Now, we provide the third parameter `by` in addition

```
> seq(5, 10, 2)
```

- `[1] 5 7 9`

Function arguments

- Providing function arguments in a given order becomes difficult if a function has a large number of arguments

We should better use the argument names

```
> seq(from=5, to=10, by=2)
```

- `[1] 5 7 9`

With argument names we can change order and omit arguments

```
> seq(by=2, from=5, to=10)
[1] 5 7 9

> seq(0, 1, length.out=5)
```

- `[1] 0.00 0.25 0.50 0.75 1.00`

Function arguments

In case of long argument names, it is sufficient to write the first letters of the argument name

```
> seq(0, 1, length.out=5)
[1] 0.00 0.25 0.50 0.75 1.00

> seq(0, 1, len=5)
[1] 0.00 0.25 0.50 0.75 1.00
```

Creating repeated values

- We often need to generate repeated group codes
- For example, let's assume we have 10 observations from group 1 and 15 observations from group 2

In the example we need to generate a group code which consists of 10 times 1 and 15 times 2:

```
1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
2
```

- We are looking for way to specify such repeated values in a compact way

Creating repeated values with function `rep`

- The function `rep` (stands for “replicate”), is used to generate repeated values

- `rep` function arguments and default values
  - `x` vector of factor that is repeated
  - `times = 1` number of times to repeat
  - `length.out = NA` desired length of the result
  - `each = 1` each element of `x` is repeated `each` times

Creating repeated values with function `rep`

```
> rep(1, 3)
[1] 1 1 1

> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4

> rep(1:4, each = 2)
[1] 1 1 2 2 3 3 4 4

> rep(1:4, each = 2, len = 4)
[1] 1 1 2 2
```

Creating repeated values with function `rep`

Coming back to our example: generate a group code which consists of 10 times 1 and 15 times 2:

```
> rep(1:2,c(10,15))
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Program today

- `ifelse ()`
- More vector functions
- **Matrix creation**
- Matrix operations
- Matrix indexing
- Matrix filtering
- Matrix function `apply()`
- Writing own functions
- Differences between vectors and matrices
- Higher-dimensional arrays

Matrix creation

- In R, a matrix is a vector with two additional attributes, the number of rows and number of columns

One of the ways to create a matrix is via the `matrix` function to obtain a matrix from a given data vector with `nrow` number of rows and `ncol` number of columns

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

Conditional element selection with the ifelse() function

- We provide the `ifelse(test, yes, no)` function with a Boolean vector `test` and two vectors `yes` and `no`

- `ifelse` returns a vector which is created from selected elements from the vectors `yes` and `no`: `yes[i]` is selected if `test[i]` is true and `no[i]` is selected if `test[i]` is false

Example (which uses recycling):

```
> ifelse(person.height > 1.7, "tall", "small")
    Can     Cem   Hande
- "small"  "tall" "small"
```

Conditional element selection with the ifelse() function

- We provide the `ifelse(test, yes, no)` function with a Boolean vector `test` and two vectors `yes` and `no`

- `ifelse` returns a vector which is created from selected elements from the vectors `yes` and `no`: `yes[i]` is selected if `test[i]` is true and `no[i]` is selected if `test[i]` is false

Example (which uses recycling):

```
> ifelse(person.height > 1.7, "tall", "small")
    Can     Cem   Hande
- "small"  "tall" "small"
```

Matrix column and row notation

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

Notation for columns: `[,1]` means first column, `[,2]` second column, etc.

Notation for rows: `[1,]` means first row, `[2,]` second row, etc.

Matrix column and row access

We can access single columns and rows with the respective column/row notation

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

> y[,1]
[1] 1 2

> y[2,]
[1] 2 4
```

Matrix single element access

We can access single elements of the matrix by providing the indices of row and column

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y[1,1]
[1] 1

> y[2,1]
[1] 2
```

Matrix creation order

- Storage of a matrix is in column-major order: first all of column 1 is stored, then all of column 2, etc.

In our example with the data vector `c(1,2,3,4)` the numbers 1 and 2 were stored in the first column and the numbers 3 and 4 in the second column

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

Matrix creation order

We can change the column-major order by providing the additional argument `byrow = TRUE` for filling the matrix by rows

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

> y
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Matrix creation with data vector and nrow

When we specify a data vector for matrix creation, we don't need to specify `ncol` since `nrow` is enough

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

Matrix row names and column names

We can provide names for the rows and columns of a matrix

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> rownames(y) <- c("Row1", "Row2")
> colnames(y) <- c("Col1", "Col2")

> y
     Col1 Col2
Row1    1    3
Row2    2    4
```

Matrix creation by specifying element individually

Another way to create a matrix is to first specify the dimension of the matrix and next specify elements individually

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

> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Matrix creation with `cbind` and `rbind`

We can “glue” vectors together, columnwise or rowwise, using the `cbind` and `rbind` functions

```
> cbind(c(1,2), c(3,4))
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> rbind(c(1,2), c(3,4))
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Matrix creation with `cbind` and `rbind`

- A typical use case for a matrix is that
  - rows correspond to different observations, e.g. various people
  - columns correspond to variables, e.g. height and weight

In the previous lecture we have seen how to manage height and weight observations with vectors

```
> person.height
  Can   Cem Hande
 1.70  1.75  1.62

> person.weight
  Can   Cem Hande
   65    66    61
```

Matrix creation with `cbind` and `rbind`

Now we manage height and weight observations with a matrix

```
> person.height.weight <- rbind(c(1.7,65),
c(1.75,66), c(1.62,61))

> rownames(person.height.weight) <- c("Can",
"Cem", "Hande")

> colnames(person.height.weight) <- c("Height",
"Weight")

> person.height.weight
       Height Weight
Can      1.70     65
Cem      1.75     66
Hande    1.62     61
```

Matrix modification with `cbind` and `rbind`

Add a column to an existing matrix

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

> y <- cbind(c(11, 12), y)
> y
     [,1] [,2] [,3]
[1,]   11    1    3
[2,]   12    2    4
```

Matrix modification with `cbind` and `rbind`

Add a row to an existing matrix

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

> y <- rbind(c(11,12), y)
> y
     [,1] [,2]
[1,]   11   12
[2,]    1    3
[3,]    2    4
```

Matrix recycling

- We already learned that when applying an operation to two vectors which requires them to be the same length, the shorter one will repeated until it is long enough to match the longer one

Example: vector addition

```
> c(1, 2, 3) + c(1, 2, 3, 4)
```

- `[1] 2 4 6 5`

The shorter vector was automatically “recycled” to be as

```
> c(1, 2, 3, 1) + c(1, 2, 3, 4)
```

- `[1] 2 4 6 5`

Matrix recycling

The automatic lengthening of vectors also works with matrices

```
> z <- matrix(c(1:9),nrow=3)
> z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> cbind(10, z)
     [,1] [,2] [,3] [,4]
[1,]   10    1    4    7
[2,]   10    2    5    8
[3,]   10    3    6    9
```

Matrix operations

We can perform various operations on matrices, e.g. matrix transposition, element by element product, matrix multiplication, matrix scalar multiplication and matrix addition

- Matrix transposition

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> t(y)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Matrix operations

Matrix element by element product for matrices of the same size

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y * y
     [,1] [,2]
[1,]    1    9
[2,]    4   16
```

Matrix operations

Matrix multiplication

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y %*% y
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

Matrix operations

Matrix scalar multiplication

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

 > 3 * y
     [,1] [,2]
[1,]    3    9
 [2,]    6   12
```

Matrix operations

Matrix addition

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y + y
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

Matrix indexing

We have already seen how to access single columns and rows

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y[,1]
[1] 1 2

> y[2,]
[1] 2 4
```

Matrix indexing

We have already seen how to access single elements of the matrix

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y[1,1]
[1] 1

> y[2,1]
[1] 2
```

Matrix indexing

We can access more than a single column/row/element at once

```
> z <- matrix(c(1:9),nrow=3)
> z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Select columns 2 and 3

```
> z[,c(2,3)]
     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    6    9
```

Matrix indexing

Select first and second row

```
> z[c(1,2),]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

Select third column of first and second row

```
> z[c(1,2),3]
[1] 7 8
```

Matrix indexing

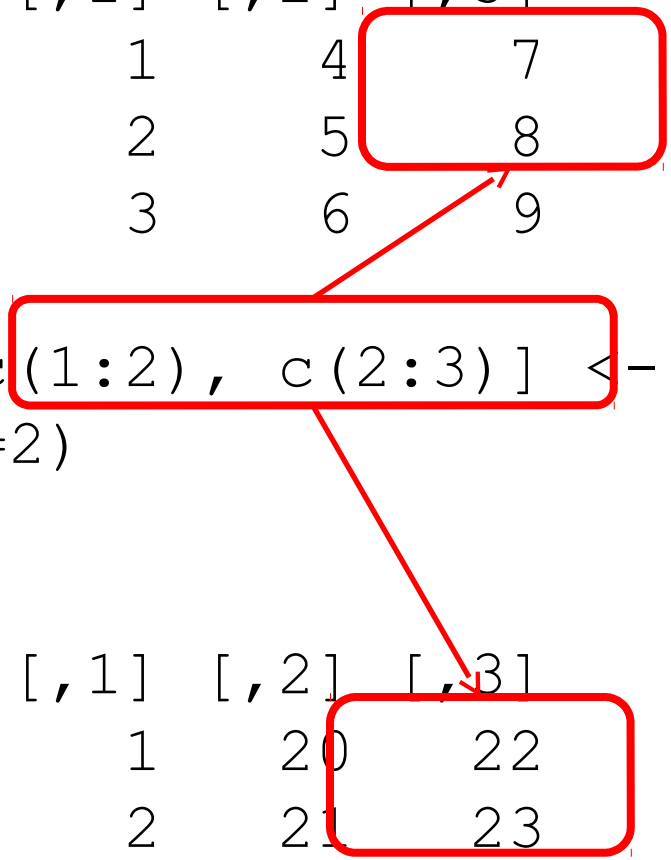
We use negative subscripts to exclude certain elements, e.g. request all rows except the second

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

> z[,-2]
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
```

Matrix indexing

We can assign new values to submatrices

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

> z[c(1:2), c(2:3)] <- matrix(c(20,21,22,23),
nrow=2)

> z
     [,1] [,2] [,3]
[1,]    1   20   22
[2,]    2   21   23
[3,]    3    6    9
```



Matrix indexing

We can delete rows or columns by reassignment, e.g. keep only first two rows and delete third row

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

> z <- z[c(1,2),]
> z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

Matrix indexing

We can delete rows or columns by reassignment, e.g. keep only first and third column and delete second column

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

> z <- z[,c(1,3)]
> z
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
```

Matrix filtering

- Similar to data vector filtering, the concept behind is to first apply a Boolean evaluation function

For each single element, the Boolean evaluation function returns `TRUE` in case of a positive evaluation and `FALSE` in case of a negative evaluation

```
> z > 3
      [,1] [,2] [,3]
[1,] FALSE TRUE TRUE
[2,] FALSE TRUE TRUE
[3,] FALSE TRUE TRUE
```

Matrix filtering

- In a second step, we use the results of the evaluation function for the filtering

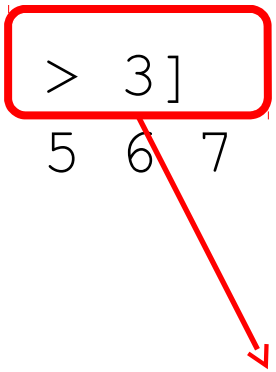
Example: obtain elements of `z` that are larger than 3

```
> greater3 <- z > 3

> greater3
      [,1] [,2] [,3]
[1,] FALSE TRUE TRUE
[2,] FALSE TRUE TRUE
[3,] FALSE TRUE TRUE

> z[greater3]
[1] 4 5 6 7 8 9
```

Matrix filtering

Similar to data vector filtering, we can perform evaluation and filtering in one line

```
> z[z > 3]
[1] 4 5 6 7 8 9
```



We provide the evaluation function directly in the square brackets for selecting those elements that fulfill the evaluation function

Matrix filtering example

- In contrast to data vector filtering, we can perform more complex filtering tasks with matrices, e.g. obtain those rows of matrix `z` having elements in the second column which are at least equal to 5

We first define the evaluation function: elements in the second column which are at least equal to 5

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

> z[,2] >= 5
 [1] FALSE  TRUE  TRUE
```

Matrix filtering example

In a second step, we apply the evaluation function when selecting the rows

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

> z[z[,2] >= 5,]
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

Matrix functions

- There exist many useful functions that operate on matrices

Some examples:

```
> rowMeans(z)
[1] 4 5 6

> colMeans(z)
[1] 2 5 8

> rowSums(z)
[1] 12 15 18

> colSums(z)
[1]  6 15 24
```

Matrix function `apply()`

- An often used generic function in R is `apply()`
- `apply()` executes a user-specified function on each of the rows or each of the columns of a matrix
- `apply(m,dimcode,f,fargs)`
  - `m` is the matrix
  - `dimcode` equal to 1 means that the function is applied to rows, `dimcode` equal to 2 means that the function is applied to columns
  - `f` is the function to be applied
  - `fargs` is an optional set of arguments to be supplied to `f`

Matrix function `apply()`

- With the generic function `apply()`, we can compute the means and sums from the previous examples

rowMeans

```
> apply(z,1,mean)
```

- `[1] 4 5 6`

colSums

```
> apply(z,2,sum)
[1]  6 15 24
```

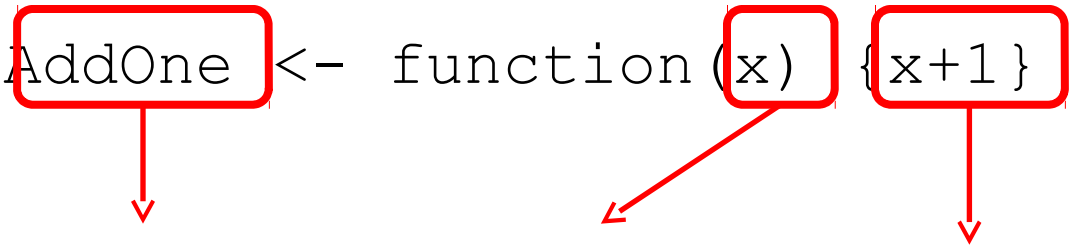
Writing our own function

- So far, we have applied some of R's inbuilt functions like `mean()`, `sum()`, `length()`, `sd()`, etc.
- Often, we need to write our own functions that fit our needs
- In general, a function is a group of instructions that takes inputs, uses them to compute other values, and returns a result
- Today, we will start with a simple example and employ it later on a matrix using `apply()`

Writing our own function

We write a simple function that adds 1 to its input and returns the result

- `> AddOne <- function(x) {x+1}`



Function name

Function inputs

Instructions that take the inputs and use them to compute other values.

The last computed value is returned by default.

Writing our own function

After defining our function, we can work with it

```
> AddOne <- function(x) {x+1}

> AddOne(1)
[1] 2

> AddOne(-5)
[1] -4

> AddOne(c(1,2,3))
[1] 2 3 4
```

Writing our own function

Let's write another more sophisticated function that adds a user-specified value to its first input

- `> AddValue <- function(x, Addend=1) {x+Addend}`

In addition to the first input `x` we specify a second input `Addend` with default value 1.

Writing our own function

After defining our new function, we can work with it

```
> AddValue <- function(x, Addend=1) {x+Addend}

> AddValue(1)
[1] 2

> AddValue(1,2)
[1] 3

> AddValue(c(1:3),2)
 [1] 3 4 5
```

Using our own function with `apply()`

First we apply `AddValue` to the rows of `z`

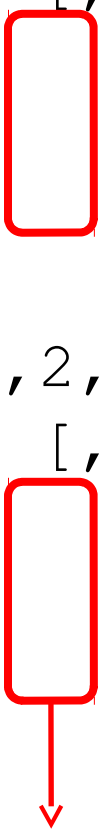
```
> z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> apply(z,1,AddValue)
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    5    6    7
[3,]    8    9   10
```

Resulting vector when adding 1 to the first row

Using our own function with `apply()`

Second we apply `AddValue` to the columns of `z`

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

> apply(z,2,AddValue)
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
[3,]    4    7   10
```



Resulting vector when adding 1 to the first column

Using our own function with `apply()`

Third we supply an optional value to `AddValue`

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

> apply(z,2,AddValue,10)
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
```

Resulting vector when adding 10 to the first column

Differences between vectors and matrices

We have learned that a matrix is a vector with two additional attributes, the number of rows and number of columns

```
> z <- matrix(c(1:8),nrow=4)
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

As `z` is still a vector, we can query its length:

```
> length(z)
[1] 8
```

Differences between vectors and matrices

As a matrix, `z` is a bit more than a vector:

```
> class(z)
[1] "matrix"

> attributes(z)
$dim
```

- `[1] 4 2`

- We observe that there exists a class called `matrix`

The matrix class has one attribute named `dim` which is a vector containing the numbers of rows and columns

- We learn more about classes in object-oriented programming

Differences between vectors and matrices

We can transform a vector into a matrix

```
> a <- c(1,2,3)

> b <- as.matrix(a)

> a
[1] 1 2 3

> b
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

Dimensions of a matrix

We can obtain the numbers of rows and columns of matrix in different ways

```
> dim(z)
[1] 4 2

> nrow(z)
[1] 4

> ncol(z)
[1] 2
```

Higher-dimensional arrays

- So far we have operated with two-dimensional matrices which represent a typical use case in data analysis:
  - rows correspond to different observations, e.g. various people
  - columns correspond to variables, e.g. height and weight

```
> person.height.weight
       Height Weight
Can      1.70     65
Cem      1.75     66
Hande    1.62     61
```

- Let's suppose we have collected data at different times, e.g. asking people every month for height and weight

- Time then becomes the third dimension, in addition to rows and columns and we call such data sets **arrays**.

Higher-dimensional arrays

- Let's consider another example: students and test scores
- Each test consists of two parts
- For each test we record two scores, one from the first part and one form the second part
- Let's create an example with two tests and three students

Higher-dimensional arrays

- For the beginning we start with matrix notation to represent first and second test

In the first test, student 1 had scores of 11 in the first part and 13 in the second part, student 2 scored 25 and 21, and so on:

```
> firsttest <- rbind(c(11,13), c(25,21),
c(33,39))

> firsttest
     [,1] [,2]
[1,]   11   13
[2,]   25   21
[3,]   33   39
```

Higher-dimensional arrays

In the second test, the same student 1 had scores of 12 in the first part and 18 in the second part, student 2 scored 22 and 26, and so on:

```
> secondtest <- rbind(c(12,18), c(22,26),
c(38,36))

> secondtest
     [,1] [,2]
[1,]   12   18
[2,]   22   26
[3,]   38   36
```

Higher-dimensional arrays

- Now let's put both tests into one data structure, which we'll name tests
- We'll arrange it to have two "layers": we'll store firsttest in the first layer and secondtest in the second
- In each layer there will be three rows for the three students' scores on the respective test
- Each layer consists of two columns for the two parts of a test

Higher-dimensional arrays

We create the two-layer data structure with the array function

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

> tests
, , 1

     [,1] [,2]
[1,]   11   13
[2,]   25   21
[3,]   33   39

, , 2

     [,1] [,2]
```

Higher-dimensional arrays

- In the dim argument we have specified
  - 3 rows for the three students
  - 2 columns for the two parts of the test
  - 2 layers for the two tests

- Each element of tests now has three subscripts which correspond to the respecitve element in the dim vector

Example: the score for student 3 in the second part of test 1 is retrieved by

```
> tests[3,2,1]
```

- `[1] 39`

Lessons learned today

- Matrix creation
  - Function matrix
  - Specify elements individually
  - Glue vectors together with `cbind` and `rbind`
- Matrix operations
  - Transposition, element by element product, matrix multiplication, matrix scalar multiplication and matrix addition
- Matrix Indexing
  - Access more than a single column/row/element at once
  - Use negative subscripts to exclude certain elements
  - Assign new values to submatrices

Lessons learned today

- Matrix Filtering
  - Boolean evaluation function
  - Perform evaluation and filtering in one line
  - Obtaining rows that fulfill a column condition
- Matrix Function apply
- Writing our own function
- Differences between vectors and matrices
- Higher-dimensional arrays

Homework

1. From the previous homework, use the body height and weight data from 10 of your friends and create a matrix

2. Assign row names and col names to your matrix

3. Add a new column to your matrix which contains the BMI

4. Increase the body weight of the first 5 persons by 10%, decrease the weight of the remaining 5 persons by 5% and recompute the BMI

5. Select those persons whose body height is larger than 1.7

Homework

1. Compute the mean BMI of those persons whose body height is less 1.75

2. Write your own function which increases the function input by x%

3. Decrease body weight by 5% when using your own function with the apply function

4. Assume you have collected body weight data a second time. Construct an array with two layers which contains the first and the second data collection