# Day_1_session_02

| Vector and matrices |
|---|
| - Mathematical operations<br>- Vectors and scalars<br>- Matrices |

## Module IV:

### Mathematical operations

| operator | description | example |
|---|---|---|
| + | addition | 5 + 5 = 10 |
| - | subtraction | 5 - 5 = 0 |
| * | multiplication | 2 * 8 = 16 |
| / | division | 100 / 10 = 10 |
| ^ (or **) | exponent/power | 5^2 = 25 |
| %% | modulo | 100 %% 15 = 10 |
| %/% | integer division | 100 %/% 15 = 6 |
| x %in% y | Where x is in y (value matching; character) | |
| ! x %in% y | Where x is not in y (value matching; character) | |

```
x <- 20
y <- 40
x+y
x-y
x*y
y/x
x**2 (x^2)
y%%x
z <- 70
z%/%x
```

**Relational and logical operations**

Let us assume we have an integer vector x and we want to check (i) which element is identical to 100L, or (ii) which of the elements is less than or equal to 30L.

```
x <- c(30L, 100L, 120L, 10L, -30L, 50L)
x == 100L            # Which element in 'x' is equal to 100L
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE FALSE
x <= 30L             # Which element in 'x' is less than or equal to 30L
## [1]  TRUE FALSE FALSE  TRUE  TRUE FALSE
x <= 30L | x == 100L
## [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

## Vectors and scalars

```
# Multiply a sequence by 2
x <- 1:10
x * 2
# Our vector x
x <- c(43, 100, 34, 483, 1000)
x %% 10
```

## Vectors and Vectors (Matching Length)

```
# Our data
x <- c(500, 400, 600)
y <- c(10, 5, 100)
# Call x + y (addition) and x / y (division)
x + y
## [1] 510 405 700
x / y
## [1] 50 80  6
```

### Vectors and vectors (non-matching lengths)

```
# Our data
x <- c(500, 400, 600, 800)
y <- c(100, 2)
# Call x + y (addition) and x / y (division)
x + y
## [1] 600 402 700 802
x / y
## [1]   5 200   6 400
```

In these situations, *R* **recycles** the shorter vector!

- in case of x + y the result is x + c(100, 2, 100, 2)
- in case of x / y the result is x / c(100, 2, 100, 2)

### Logical values

```
c(TRUE, FALSE) * 100L
## [1] 100   0
c(TRUE, FALSE) / 2
```

### Missing values

```
c(1, 2, NA, NA) * 100
## [1] 100 200  NA  NA
c(1, 2, 3, 4) * NA
## [1] NA NA NA NA
100 %% NA
## [1] NA
```

**Vector attributes**

All objects in R can have additional so called attributes. Attributes are used to store additional meta-information.

```
# Plain vector
(x <- c(54, 1.82))
## [1] 54.00  1.82
attributes(x)
```

**Named vectors**

```
 v # Adding optional names
(y <- c(age = 54, height = 1.82))
##    age height
##  54.00   1.82
attributes(y)
```

```
# Create a named vector
(x <- c(age = 54, height = 1.82))
##    age height
##  54.00   1.82
```

If we now call `names(x)` we will get a character vector with the names of the elements:

```
# Get/extract the names
names(x)
```

```
# Plain vector (without names)
(x <- c(1L, 2L, 3L))
## [1] 1 2 3
names(x)
## NULL
# Add names
names(x) <- c("first", "second", "third")
names(x)
## [1] "first"  "second" "third"
x
```

**Subsetting vectors**

**Subsetting by index**

```
# Create a demo vector of length 5
(x <- c(10, 20, 0, 30, 50))
## [1] 10 20  0 30 50
x[1]          # Extract first element by index
## [1] 10
x[5]          # Fifth element
## [1] 50
```

If we need multiple elements at the same time, e.g., the first *and* the fifth, we can use a vector of indices as follows:

```
x[c(1, 5)]   # Get the first and fifth
## [1] 10 50
x[c(5, 1)]   # Or the fifth and first (different order)
## [1] 50 10
```

Similarly, we can use *negative indexes* to get all values *except* some specific ones. E.g., x[-1] gives "*all but the first element of x*".

```
x[-1]         # all but first
## [1] 20  0 30 50
x[-5]         # all but fifth
## [1] 10 20  0 30
x[-c(1, 5)]   # all except first and fifth
## [1] 20  0 30
```

**Out-of-range indexes**

```
x[4:7]
## [1] 30 50 NA NA
x[100]
## [1] NA
```

**Last/first few elements**

```
x[1:3]          # First three elements (element 1 - 3)
## [1] 10 20  0
x[3:5]          # Last three elements (elements 3 - 5)
## [1]  0 30 50
head(x, n = 3)
## [1] 10 20  0
tail(x, n = 3)
## [1]  0 30 50
```

**Subsetting by name**

```
x <- c(age = 35, height = 1.72, zipcode = 6020)
```

We are interested in the value for the element "age". Instead of using x[1L] we can now use x["age"] for subsetting:

```
x["age"]
```

**Subsetting by logical vectors**

```
x <- c(30, 10, 20, 0, 30, 50)
x[c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)]
x > 25          # Shows the result of the relational comparison
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
x[x > 25]       # Use 'x > 25' (logical vector) for subsetting
## [1] 30 30 50
```

Some more examples which you can try yourself given our vector x <- c(30, 10, 20, 0, 30, 50):

- x[x == 30]: Elements in x where x is (exactly) equal to 30
- x[x == 30 | x == 50]: Elements where x is either equal to 30 OR equal to 50.
- x[x == 30 & x == 50]: Elements where x is equal to 30 AND 50 at the same time. The result should be an empty vector as this is not possible.
- x[x >= 30 & x < 40]: All elements where x is larger than or equal to 30, but less than 40.
- x[x < 30 & x > 40]: All elements where x is less than 30 AND larger than 40 at the same time. Again, impossible, the result should be an empty vector.

```
age    <- c(  25,   53,   21,   50,   18,   63)
height <- c(1.73, 1.69, 1.83, 1.57, 1.91, 1.75)
```

```
res <- height < 1.8
res
## [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
age[res]
## [1] 25 53 50 63
age[height < 1.8]
## [1] 25 53 50 63
```

Or extend the expression to get the age for those who are smaller than 1.8 and older than 50 years.

```
age[age > 50 & height < 1.8]
```

**Warning**: Avoid using == for floating point arithmetic on double vectors.

```
(1.5 - 0.5) == 1
## [1] TRUE
(1.9 - 0.9) == 1
## [1] FALSE
all.equal(1.9 - 0.9, 1)
```

**Index of TRUE elements**

```
x <- c(30, 10, 20, 0, 30, 50)
x >= 30
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
which(x >= 30)
## [1] 1 5 6
in(x)                   # The minimum
## [1] 0
x == min(x)             # Comparison
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
which(x == min(x))      # Index/position
## [1] 4
```

The same can be done to find the maximum.

```
max(x)                  # The maximum
## [1] 50
x == max(x)             # Comparison
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
which(x == max(x))      # Index/position
## [1] 6
```

```
z <- c(10, 0.5, 20, 30, 0.5)
which(z == min(z))       # Position/index of all minima
## [1] 2 5
which.min(z)             # Position/index of first occurrence of minimum
## [1] 2
```

## Single and double brackets

[] keeps the name attribute, while [[]] drops them all.

```
(x <- c(54, 1.82, 6020))
## [1]   54.00    1.82 6020.00
x[3]        # Single brackets
## [1] 6020
x[[3]]      # Double brackets
## [1] 6020
```

However, if we have a named vector the result of the two commands look different:

```
(x <- c("age" = 54, "height" = 1.82, "zipcode" = 6020))
##     age  height zipcode
##   54.00    1.82 6020.00
x[3]
## zipcode
##    6020
x[[3]]
## [1] 6020
```

**Vector functions**

| Function | Description | Function | Description |
|---|---|---|---|
| round() | Round numeric values | str() | Overview of the object structure |
| min(), max() | Minimum and maximum | any(), all() | Test vector elements |
| mean(), median() | Arithmetic mean and median | all.equal() | Test for near equality |
| sum() | Sum | sort() | Sort a vector |
| sd(), var() | Standard deviation and variance | order() | Obtain ordering of a vector |
| sqrt() | Square root | summary() | Numerical summary |

For the purpose of reproducibility we will call set.seed() to fix a random seed. Thus, we should all get the same random numbers

```
set.seed(6020)     # A specific seed; fixed using an integer
x <- rnorm(100)    # Draw 100 random numbers from the standard normal distribution
head(x, n = 10)    # Show first 10 elements
head(round(x, digits = 1), n = 10)   # Round, show first 10 elements only
```

## Summary statistics

```
c(min  = min(x),     max    = max(x),
  mean = mean(x),    median = median(x),
  var  = var(x),     sd     = sd(x),
  sum  = sum(x),     length = length(x))

summary(x) # Numerical summary
str(x)     # Structure
```

## Missing values in vector

```
# Generate the vector (again).
set.seed(6020)
x <- rnorm(100)
# Replace element 2, 4, 50, 70, and 99 with a missing value (NA)
x[c(2, 4, 50, 70, 99)] <- NA
summary(x)
sum(is.na(x))
```

```
c(min  = min(x),     max    = max(x),
  mean = mean(x),    median = median(x),
  var  = var(x),     sd     = sd(x),
  sum  = sum(x))
```

```
x2 <- na.omit(x)
```

```r
c(length(x), length(x2))
c(mean(x, na.rm = FALSE),   # Default
  mean(x, na.rm = TRUE))    # Account for missing values
```

```r
# Minimum, maximum, mean, median, variance, and standard deviation
c(min  = min(x, na.rm = TRUE),     max    = max(x, na.rm = TRUE),
  mean = mean(x, na.rm = TRUE),    median = median(x, na.rm = TRUE),
  var  = var(x, na.rm = TRUE),     sd     = sd(x, na.rm = TRUE),
  sum  = sum(x, na.rm = TRUE))
```

**Plotting vectors**

There are several different basic plot types, including:

| Function | Description |
|---|---|
| plot() | Generic X-Y plot |
| barplot() | Bar plot |
| pie() | Pie chart |
| hist() | Histogram |
| boxplot() | Box-and-Whisker plot |

*We will put special focus on plotting in our ggplot/visualization module

**Generic X-Y Plot**

```r
# Create the data vector
d <- c(10, 5, 30, 20, -5, 11)
plot(d)
```

```r
# Create the data vector
d <- c(10, 5, 30, 20, -5, 11)
plot(d,
     main = "Figure Title",
     xlab = "index/position in the vector",
     ylab = "measurement",
     type = "l", col = 4, lwd = 3)
```

**Bar plot**

```r
election_results <- c("DEM"          = 71, "REP"          = 40,
                      "GRE"          = 30, "YEL"        = 26,
                      "NEO"        = 15,  "WEST" = 1)
barplot(election_results)
```

```r
barplot(election_results,
        main = "NATIONAL ELECTION 2020",
        xlab = "ELECTORAL COLLEGE PARTY",
        ylab = "SIT GAINED",
        col  = c("turquoise3", "firebrick2", "dodgerblue3", "green4",
"violetred2", "gray70"))
```

## Pie plot

```
pyramid <- c("Sky" = 70,
             "Sunny side\nof pyramid" = 20,
             "Dark side\nof pyramid" = 10)
par(mar = rep(1, 4L)) # reduce border margin
pie(pyramid, init.angle = -35, col = c("#0099FF", "#FFE452", "#AA9526"),
    main = "Pie")
```

```
d <- rnorm(1000)
hist(d)
```

## Histograms

```
d <- c(rnorm(1000, mean = 0, sd = 1),
       rnorm(500,  mean = 7, sd = 2))
hist(d,
     freq = FALSE,    # Show density instead of frequency
     breaks = 50)     # Number of breaks
```

## Box-and-Whisker plot

```
x <- (5 + rnorm(100))^5 # Generate some values
boxplot(x)
```

## Day_1_session_02

## Module V:

### Matrices

- Arrays are **based on atomic vectors**.
- A matrix is a special array with **two dimensions**.
- Matrices can only contain data of **one type** (like vectors).
- Matrices always also have a **length** (number of elements).

- data: a data vector (default NA)
- nrow: desired number of rows (first dimension, 'top down'; default 1).
- ncol: desired number of columns (second dimension, 'left to right'; default 1).
- byrow: logical, whether or not to fill by row (default FALSE; fill by column).
- dimnames: optional list of length 2 with row names and column names (default NULL).

### Creating matrices

```
(x <- matrix(data = 999L, nrow = 2, ncol = 3))
dim(x)
```

```
nrow(x)
## [1] 2
ncol(x)
## [1] 3
length(x)           # Using length
## [1] 6
nrow(x) * ncol(x)   # Calculate 'by hand'
## [1] 6
```

### Type of data

Matrices (as vectors) can only contain data of one type.

```
x1 <- matrix(seq(0, 4.5, length.out = 9), nrow = 3)     # double
x2 <- matrix(1:9, nrow = 3)                             # integer
x3 <- matrix(LETTERS[1:9], nrow = 3)                    # character
x4 <- matrix(TRUE, nrow = 3, ncol = 3)                  # logical
```

### Order of elements

Let us have a closer look at x2, the integer matrix from above, and how the elements of the vector end up in the matrix.

```r
(x2 <- matrix(1:9, nrow = 3))
```

```r
(x <- matrix(data = 1:9, nrow = 3, byrow = TRUE))
```

## Matrix functions

| Function | Description |
|---|---|
| head() | Return first few rows. |
| tail() | Return last few rows. |
| summary(x) | Numerical summary of the matrix (column-wise). |
| rbind() | Combine objects 'by row'. |
| cbind() | Combine objects 'by column'. |
| order() | Allows to sort matrices. |
| ... | Many more functions exist |

## Mathematical operations

Matrices are often used for arithmetic (mathematics; working with numbers) to solve mathematical problems such as solving systems of linear equations, estimate regression models, and many more.

```r
(x <- matrix(1:4, ncol = 2))
x + 2          # Add 2 to each element
x * 1.5        # Multiply each element by 1.5
```

```r
x <- matrix(1:4, ncol = 2)  # (Re-)define matrix
y <- c(10, 100)             # Define vector
x * y                       # Multiply
```

The matrix has 2×2=4 elements while the vector is shorter and contains only 2 elements. Thus, *R* recycles (re-uses) the vector elements to be able to perform the calculations.

In the same way, simple 'matrix and matrix' operations work. When two matrices are of the same dimension, we can use basic element-wise arithmetic operations.

```r
(x <- matrix(c( 1,  2,  3,  4), ncol = 2, nrow = 2))
(y <- matrix(c(10, 20, 30, 40), ncol = 2, nrow = 2))
x + y
x / y
```

```
x^y
# Type of the data
```

As we know from the Vectors chapter and the first part of this chapter, all object have some mandatory attributes (such as the class), and can have additional ones.

```
x <- matrix(data = NA, ncol = 2, nrow = 10)
attributes(x)
x <- matrix(c(1L, 2L, 3L, 4L), ncol = 2)
length(x) == nrow(x) * ncol(x)    # Length gives us the number of elements
## [1] TRUE
class(x)                          # Class of the object
## [1] "matrix" "array"
typeof(x)
 (x <- matrix(data = 1:9, nrow = 3, ncol = 3))
attributes(x)                           # Only dimension is specified
rownames(x) <- c("Row 1", "Row 2", "Row 3")
colnames(x) <- c("Col A", "Col B", "Col C")
attributes(x)
```

Dimension names

- `rownames()`: names of the rows of a matrix.
- `colnames()`: names of columns of a matrix.
- `dimnames()`: returns all dimension names (as a list; works for all arrays).

```
(x <- matrix(data = 1:9, nrow = 3, ncol = 3))
attributes(x)
rownames(x); colnames(x); dimnames(x)   # All returning NULL
rownames(x) <- c("Row 1", "Row 2", "Row 3")
colnames(x) <- c("Col A", "Col B", "Col C")
```

```
x
```

```
rownames(x)
colnames(x)
dimnames(x)
```

## Changing dimension names

At any time we can use rownames() and colnames() to change (or overwrite) existing names. Let us take the matrix from above – instead of having Col A, Col B, and Col C we would like to have first, second, and third.

```
colnames(x) <- c("first", "secon", "third")
x
colnames(x)[2]              # The one to fix typo secon
colnames(x)[2] <- "second"  # Overwrite second column name
```

```
x
```

## Creating named matrices

Instead of creating a plain matrix and adding dimension names in a second step, we can (similar to what we have seen in the Vectors chapter) directly create named matrices using the matrix() function.

```
(x <- matrix(data = 1:9, nrow = 3, ncol = 3,
             dimnames = list(c("Row 1", "Row 2", "Row 3"),
                             c("Col A", "Col B", "Col C"))))
# Only row names, column names set to NULL
(x <- matrix(data = 1:9, nrow = 3, ncol = 3,
             dimnames = list(c("Row 1", "Row 2", "Row 3"),
                             NULL)))
# Only column names, row names set to NULL
(x <- matrix(data = 1:9, nrow = 3, ncol = 3,
             dimnames = list(NULL,
                             c("Col A", "Col B", "Col C"))))
```

## Combine Objects

```
# Generate vectors
x <- c( 5,  5,  5)
y <- c(11, 22, 33)
# Combine
(z <- cbind(x, y))
class(z)
(z <- rbind(x, y))
```

## Dimension names:

```
# Original vectors
participants_age <- c(  24,   25,   21,   32,   19)
participants_hgt <- c(1.73, 1.62, 1.72, 1.82, 1.71)
# Combine to matrix
rbind(participants_age, participants_hgt)
```

This works well, but the names are a bit long and unhandy. Instead, we can specify new names when calling rbind() or cbind() as follows:

```
participants <- rbind(age = participants_age, height = participants_hgt)
participants
```

The same works for column binding:

```
participants <- cbind(age = participants_age, height = participants_hgt)
participants
```

## Multiple matrices

```
x1 <- matrix(1:6,     ncol = 2)
x2 <- matrix(101:106, ncol = 2)
cbind(x1, x2)
rbind(x1, x2)
(x1 <- matrix(1:6, ncol = 2,
              dimnames = list(c("Row 1", "Row 2", "Row 4"), c("Col A", "Col B"))))
(x2 <- matrix(101:106, ncol = 2,
              dimnames = list(c("Row 1", "Row 2", "Row 4"), c("Col A", "Col B"))))
cbind(x1, x2)
rbind(x1, x2)
```

## Subsetting Matrices

- Subsetting **by index**.
- Subsetting **by name** (if set).
- Subsetting **by logical vectors**.
- For matrices, subsetting is typically done two-dimensional (but not necessarily!).

```
(x <- matrix(sprintf("x[%d, %d]", rep(1:3, 4), rep(1:4, each = 3)), nrow = 3))
# sprintf : A wrapper for the C function sprintf, that returns a character vector
containing #a formatted combination of text and variable values.
x[1, 1]
x[2, 4]
class(x[2, 3])                # 'drop = TRUE' (default): numeric vector
class(x[2, 3, drop = FALSE])  # 'drop = FALSE': matrix ...
dim(x[2, 3, drop = FALSE])    # ... dimension 1 x 1
x[3]
# Demonstration
c(x[3],  x[3, 1])
## [1] "x[3, 1]" "x[3, 1]"
c(x[11], x[2, 4])
## [1] "x[2, 4]" "x[2, 4]"
```

## Extracting multiple elements

```
mat[c(4, 2), 3]
mat[2, 1:3]
mat[c(2, 7, 12)]    # Vector subsetting
## [1] 100  10 190
mat[2, 1:3]         # Matrix subsetting
```

Extracting rows/columns

```
(x <- matrix(1:12, nrow = 3))
x[1, ]     # Returns the entire first row
## [1]  1  4  7 10
x[, 3]     # Returns the entire third column
x[1, c(3, 4), drop = FALSE]
dim(x[1, , drop = FALSE])
dim(x[, 1, drop = FALSE])
dim(x[1, 2:3, drop = FALSE])
## [1] 1 2
dim(x[1:3, 2, drop = FALSE])
```

**Subsetting by name**

In the very same way we can access elements using the corresponding row names and column names (if set).

```
# Construct matrix
countries <- c("United States", "Great Britain", "Canada", "Russia",
"Switzerland")
(medals   <- matrix(c(3, 3, 2, 1, 1, 4, 1, 1, 0, 0, 1, 5, 1, 2, 2),
                    ncol = 3, dimnames = list(countries, c("Gold", "Silver",
"Bronze"))))
medals[3, 1]   # Canada, number of gold medals
medals["Canada", "Gold"]
medals   # New, updated matrix
```

**Extracting rows/columns**

```
medals["Canada", ]
medals[, "Gold"]
medals["Canada", , drop = FALSE]
medals[, "Gold", drop = FALSE]
# Get column 'Gold' as matrix (drop = FALSE).
# Extract second element (vector subsetting).
medals[, "Gold", drop = FALSE][2]
# Get column 'Gold' as matrix (drop = FALSE).
# Get row "Russia" (single element as we only have one column; Gold).
medals[, "Gold", drop = FALSE]["Russia", ]
medals[medals[, "Gold"] > 1, ]
medals[, "Gold"]      # 1) Access the column "Gold" (results in a vector)
medals[, "Gold"] > 1  # 2) Check which elements of the vector contain a value > 1
idx <- medals[, "Gold"] > 1     # Logical vector
medals[idx, ]                   # Subsetting
medals[medals > 3]
medals > 3
which(medals > 3)
```

**Replace elements**

A nice practical application for subsetting with logical vectors is 'search and replace' Let us use the following matrix with random values (rounded to 1 digits after the comma): We would like to replace all negative values with NA.

```
set.seed(123)
```

```
(x <- round(matrix(rnorm(25), nrow = 5), 1))
x[x < 0]
x[x < 0] <- NA
x
```

We could also only replace all elements between -0.2 and +0.2 and those exactly 1.7 using some logical & and |, this time with -999:

```
set.seed(123)
(x <- round(matrix(rnorm(25), nrow = 5), 1))
x[(x > -0.2 & x < 0.2) | x == 1.7] <- -999
x
```

## Mixed subsetting

Subsetting methods can also always be mixed. This could be important if you have a matrix which has row names, but no column names, or vice versa. This could be important if you have a matrix which has row names, but no column names, or vice versa. Using the medals matrix from above, we can retrieve the element in the third row (using subsetting by index) of the column "Gold" (subsetting by name) like this:

```
medals[3, "Gold"]
# Relational expression: results in a logical vector
(idx <- medals[, "Gold"] > 2)
# Use 'idx' for subsetting.
medals[idx, "Silver"]
# The same in one command
medals[medals[, "Gold"] > 2, "Silver"]
```

## Sort & Order

We have learned that we can sort vectors using sort(), and get the order using order(). This can also be used for sorting or ordering matrices.

```
(students <- matrix(c(24, 30, 53, 24, 24, 1.67, 1.93, 1.73, 1.65, 1.71, 5, 3, 7,
2, 2),
                    nrow = 5,
                    dimnames = list(c("Peter", "Elif", "Leo", "Marcus", "Rob"),
                                    c("age", "height", "semester"))))
```

```
sort(students[, "age"])     # Increasing
```

```
students2 <- students       # Make a copy
students2[, "age"] <- sort(students2[, "age"])
```

```
students2
```

**Re-order matrix**

Instead, we make use of `order()`. As we have seen (Vectors), `order()` returns the position of the elements from smallest to largest or vice versa. This can be used to properly re-order the matrix. In step 1 we would like to get the order of the elements in column `"age"`.

```
(ridx <- order(students[, "age"]))
students[ridx, ]
# Age and height
(ridx2 <- order(students[, "age"], students[, "height"]))
students[ridx2, ]
(ridx3 <- order(rownames(students), decreasing = TRUE))
students[ridx3, ]
```

More matrix arithmetic

| **Command** | **Description** |
|---|---|
| t(x) | Transpose x. |
| diag(x), diag(3L) | Diagonal elements of x. |
| x %*% y | Matrix multiplication (inner product). |
| solve(x) | Inverse of x. |
| solve(a, b) | Solve system of linear equations. |
| crossprod(x, y) | Cross product. |
| outer(x, y), x %o% y | Outer product. |
| kronecker(x, y) | Kronecker product. |
| det(x) | Determinant. |
| qr(x) | QR decomposition |
| chol(x) | Cholesky decomposition |

**The identity matrix**

```
diag(3)
```

It is possible to use outer() function or the tensor operator %o% from the tensor package.

```
v <- c(-2, 3,  0 , 1)
w <- c(1,  4,  -7,  9)
outer(v, w)
library("tensor")
v %o% w
```

**Transposition**

```
# create a matrix with 2 rows
# using matrix() method
M <- matrix(1:6, nrow = 2)

# print the original matrix
print(M)

# transpose of matrix
# using t() function.
t <- t(M)

# print the transpose matrix
print(t)
```

**Symmetric matrices**

```
my_mat <- matrix(c(1, 2, 3, 4,                        # Create example
matrix
                   0, 1, 5, 6,
                   0, 0, 1, 7,
                   0, 0, 0, 1),
                 ncol = 4)
my_mat

my_mat_sym <- my_mat                                  # Duplicate
matrix
my_mat_sym[upper.tri(my_mat_sym)] <- t(my_mat_sym)[upper.tri(my_mat_sym)] #
Insert lower to upper matrix
my_mat_sym                                            # Print new
matrix

# Creating a diagonal matrix
x1 <- diag(3)

# Creating a matrix
x2 <- matrix(c(1, 2, 2, 3), 2)

# Calling isSymmetric() function
isSymmetric(x1)
isSymmetric(x2)
```

## skew-symmetric matrix

skew symmetric matrix, i.e., the transpose of the matrix is the negative of the matrix.

```
A <- diag( 1, 3 )
is.skew.symmetric.matrix( A )
B <- matrix( c( 0, -2, -1, -2, 0, -4, 1, 4, 0 ), nrow=3, byrow=TRUE )
is.skew.symmetric.matrix( B )
C <- matrix( c( 0, 2, 1, 2, 0, 4, 1, 4, 0 ), nrow=3, byrow=TRUE )
is.skew.symmetric.matrix( C )
all(A==-t(A))
```

## Diagonal matrices

```
diag(c(-1, 7, 5))
```

## Eigenvalues and eigenvectors

```
# R program to illustrate
# Eigenvalues and eigenvectors of matrix

# Create a 3x3 matrix
A = matrix(c(1:9), 3, 3)

cat("The 3x3 matrix:\n")
print(A)

# Calculating Eigenvalues and eigenvectors
print(eigen(A))

# R program to illustrate
# Eigenvalues and eigenvectors of matrix

# Create a 3x3 matrix
A = matrix(c(2, 3, 5, 1), 2, 2)
A

# Calculating Eigenvalues and eigenvectors
print(eigen(A))
```

Ref:

- Hadley Wickham’s Advanced R book
- Roger Peng’s R Programming for Data Science book
- DataCamp’s Intermediate R course
- Coursera’s R Programming course
- Data Carpentry (http://datacarpentry.org/), data camp, data quest, Kaggle
- Harvard Chan Bioinformatics Core (HBC) under the open access terms of the Creative Commons Attribution license (CC BY 4.0),
- The Book of R: A First Course in Programming and Statistics by Tilman M. Davies
- UC Business Analytics R Programming Guide and R_bootcamp