University of Glasgow

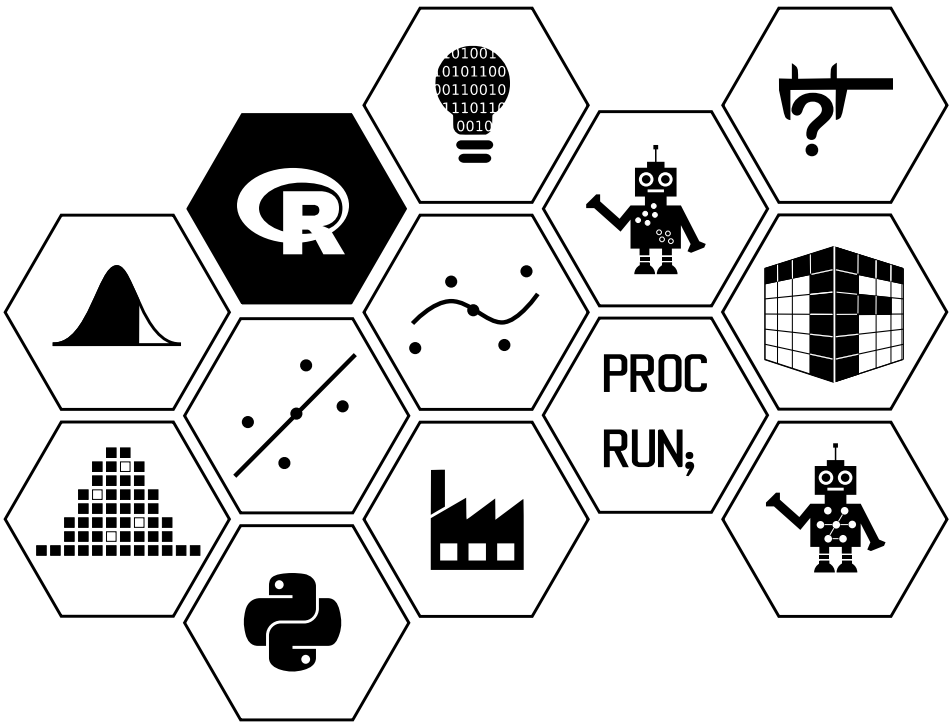
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 2:

R Basics



DATA ANALYTICS GLASGOW

Logical variables and comparisons

Logical variables and comparison operators

https://youtu.be/Xrm1cp-WSLM

Duration: 6m48s

Logical variables

A logical variable can only hold the two values `TRUE` and `FALSE`. Logical variables are sometime called Boolean variables, after George Boole (1815-1864), an English mathematician, logician and philosopher. R has the following three binary operators `!` (negation), `&` (logical AND) and `|` (logical OR). The table below shows some examples of logical statement combinations and outputs.

`expr1`	`expr2`	`!expr1` (NOT)	`expr1 & expr2` (AND)	`expr1 | expr2` (OR)
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE

Example 1.

```
a <- TRUE
b <- FALSE
c <- a & !b
c

## [1] TRUE
```

In this example `c` is `TRUE`, because `!b` is `TRUE` and `TRUE & TRUE` is `TRUE`.

In R, `&` has higher precedence than `|`. So, in the absence of parentheses, `&` is evaluated before `|`. For example, `TRUE | FALSE & FALSE` is treated by R as `TRUE | (FALSE & FALSE)`, which is `TRUE`. We have to use parentheses to calculate `(TRUE | FALSE) & FALSE`, which is `FALSE`.

Task 1.

Consider three logical variables.

```
a <- TRUE
b <- FALSE
c <- TRUE
```

Without using R determine the values of ...

```
!a & b
a | !b
```

FALSE
TRUE
FALSE
TRUE

```
!(a | !b)
(a & b) | c
```

Supplementary material:
TRUE and FALSE vs. T and F

R also allows using the shorthands `T` instead of `TRUE` and `F` instead of `FALSE`. It is however *not* recommended to use these shorthands. Whereas `TRUE` and `FALSE` are reserved keywords, which cannot be overwritten, `T` and `F` are just global variables set to `TRUE` and `FALSE`, respectively. This means they can be masked by local variables. Nothing (in R) prevents you from setting

```
T <- FALSE
F <- TRUE
```

and `T` and `F` have become the exact opposite of what they are meant to be! Of course, few users would do this deliberately, but it is not inconceivable that you happen to define a variable `T` or `F`, which can, depending on its values, have exactly the same effect.

Supplementary material:
&& and || vs. & and |

R also has "lazy" operators `&&` and `||`. In contrast to `&` and `|` they will only evaluate the arguments until the result has become clear. On the other hand, `&` and `|` will always evaluate all arguments. `expr1&&expr2` will evaluate `expr2` only if `expr1` is `TRUE` (otherwise the result is guaranteed to be `FALSE` no matter what `expr2` is). Similarly, `expr1||expr2` will evaluate `expr2` only if `expr1` is `FALSE` (otherwise the result is guaranteed to be `TRUE` no matter what `expr2` is). `&&` and `||` can be helpful in conditional `if` statements. You should *not* use `&&` and `||` for vectors of length greater than 1.

In contrast to many other programming languages `&` and `|` are not bitwise operators when applied to numbers, but simply coerce the number to a logical variable (ie. everything other than 0 will be treated as `TRUE`).

Comparison operators

The comparison operators in R are == (testing for exact equality), `!=` ("not exactly equal") <, <=, >, and >=.

The comparison operators return a logical value (i.e. `TRUE` or `FALSE`), so you can use the operators `!`, `&` and `|` to combine them to more complex expressions.

Example 2.

Consider a variable `x` set to the number 2.

```
x <- 2
```

We can then test whether it is negative or less than or equal to 3.

```
x < 0
```

```
## [1] FALSE
```

```
x <= 3
```

```
## [1] TRUE
```

If we want to test whether `x` is in the unit interval we can use

```
x>0 & x<1
```

```
## [1] FALSE
```

or, equivalently,

```
!(x<=0 | x>=1)
```

```
## [1] FALSE
```

Due to rounding and representation errors, you do not want to use == to compare non-integers. For example, despite $0.3 - 2 \times 0.1 = 0.1$, R yields

```
0.3 - 2 * 0.1 == 0.1
```

```
## [1] FALSE
```

because the expression on the left-hand side is not exactly 0.1. We see this by subtracting 0.1 from the left-hand side (which should then be exactly zero, but isn't)

```
0.3-2*0.1 - 0.1
```

```
## [1] -2.775558e-17
```

For non-integers we really only want to test whether they are "nearly equal", we can do so by comparing the absolute difference to a small number (say 10^{-8}).

```
abs(0.3-2*0.1 - 0.1) < 1e-8
```

Use absolute call

```
## [1] TRUE
```

or use the built-in function `all.equal`.

```
isTRUE(all.equal(0.3-2*0.1, 0.1))
```

```
## [1] TRUE
```

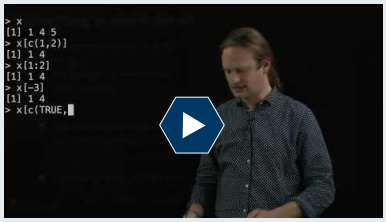
Task 2.

Create a variable `x` storing the fraction $\frac{355}{113}$. Use R to test

- whether this fraction is less than π (`pi` in R),
- whether this fraction is between 3 and 4, and
- whether this fraction is within $\pm 10^{-6}$ of π.

FALSE
TRUE
TRUE

Vectors in R



Vectors

https://youtu.be/m8ER5bkuRJI

Duration: 6m20s

So far we have only looked at scalar variables, i.e. variables containing only one value. In most programming languages such scalar variables are the basic data types. R however does not have scalar variables, all variables are vectors. A variable storing a number is for example just a numeric vector of length 1.

Defining vectors

The simplest way to create vectors in R is to use the function `c`:

```
a <- c(1, 4, 2)
a
```

```
## [1] 1 4 2
```

We can use the function `c` as well to concatenate ("stick together") two vectors:

```
a <- c(1, 4, 2)
b <- c(5, 9, 13)
c <- c(a, b)
c
```

```
## [1]  1  4  2  5  9 13
```

Task 3.

Create a vector `x` that contains the values $\mathbf{x} = (1, 3, 2, 5)$ and a vector `y` which contains the values $\mathbf{y} = (1, 0)$. Finally, concatenate `x` and `y` and store the result as `z`. Print `z`.

```
x=c(1,3,2,5)
y=c(1,0)
z=c(x,y)
z
```

Naming entries

If the entries of the vectors correspond to quantities with natural names it is a good idea to associate names with the entries of the vector. This can be done using

```
names(a) <- c("first", "second", "third")
a
```

```
##  first second  third
##      1      4      2
```

Alternatively we can set the names when creating the vector using `c`

```
a <- c(first=1, second=4, third=2)
```

Accessing elements

You can use square brackets to access a single element of a vector. `x[i]` returns the `i`-th element of the vector `x`. Using the vector `a` from above,

```
a[3]
```

```
## third
##     2
```

You can use the same notation to change elements of a vector:

```
a[3] <- 10
a
```

```
##  first second  third
##      1      4     10
```

If the element you want to change is beyond the last element of the vector, the vector will be extended, such that the `i`-th element is the last element.

```
a[7] <- 99
a
```

```
##  first second  third
##      1      4     10     NA     NA     NA     99
```

If the entries of the vector are named, you can also use the names for accessing elements.

```
a["third"]
```

```
## third
##    10
```

Subsetting vectors

You can use square brackets not only to access a single element of a vector, but also to subset a vector. There are three ways of specifying subsets of vectors:

You can use a vector specifying the indices to be returned:

```
a <- c(1, 4, 9, 16)
a[c(1,2,3)]
```

```
## [1] 1 4 9
```

You can use a vector specifying the indices to be removed (as negative numbers):

```
a[-4]
```

```
## [1] 1 4 9
```

You can use a logical vector specifying the elements to be returned:

```
a[c(TRUE, TRUE, TRUE, FALSE)]
```

```
## [1] 1 4 9
```

We can exploit the latter when we want to subset a vector based on its values. Suppose you want to keep all elements in `a` that are divisible by 2:

```
a[a%%2==0]
```

```
## [1]  4 16
```

Why does this work? `a%%2==0` returns a logical vector of length 4, indicating which elements of `a` are divisible by 2. These elements are then selected from `a`.

Task 4.

Consider the vector `x` defined as

```
x <- c(1, 5, 9, 3, 8)
```

Use all three of the above methods to extract the first, third and fifth entry.

Vectorised calculations

Vectorisation

https://youtu.be/dlLiux93ueA

Duration: 4m51s

Vectors can be used in arithmetic expressions using the arithmetic operators and the mathematical and statistical functions we have seen when we used R as a calculator in Week 1. In this case the computations are carried out element-wise.

For example,

```
a <- c(1, 2, 3, 4)
b <- c(2, 0, 1, 3)
c <- 2 * a + b
c
```

```
## [1]  4  4  7 11
```

The third entry of the result, 7, is obtained by taking twice the third entry of the vector `a` and adding it to third entry of the vector `b`, i.e. $c_3 = 2 \times a_3 + b_3 = 2 \times 3 + 1 = 7$.

Recycling rules

If vectors of different length are used in an arithmetic expression, the shorter vector(s) are repeated ("recycled") until they match the length of the longest vector.

```
a <- c(1, 2, 3, 4)
b <- c(2, 0)
a * b
```

```
## [1] 2 0 6 0
```

R has thus "recycled" the vector `b` once.

If the length of the longest vectors is not a multiple of the length of the shorter vector(s), R will produce a warning. For example,

```
a <- c(1, 2, 3, 4)
b <- c(2, 0, 1)
a * b
```

```
## Warning in a * b: longer object length is not a multiple of shorter object length
```

```
## [1] 2 0 3 8
```

The vector `b` is shorter than the vector `a`. However, if `b` is recycled once, it would have 6 elements, making it longer than `a`. Thus R produces a warning. Such a warning is almost always a sign that you have made a mistake.

Task 5.

Consider the following two vectors `x` and `y` and their sum

```
x <- c(1, 2, 9, 4, 5, 6)
```

```
y <- c(0, 2)
x+y

## [1] 1 4 9 6 5 8
```

Explain how R has calculated the result.

Sequences and patterned vectors

Sequences R has built-in functions to create simple sequences and patterned vectors:
The operator `:` can be used for creating basic sequences.

```
2:5

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

If the first argument is larger then the second argument, then the sequence will be decreasing.

```
1:0

## [1] 1 0
```

`seq(from, to, by)` creates a sequence from `from` to `to` using `by` as increment.

```
seq(1, 2, by=0.2)

## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

`seq(from, to, length.out=n)` creates a sequence of total length `n` with initial value `from` and ending value `to`.

```
seq(3, 5, length.out=5)

## [1] 3.0 3.5 4.0 4.5 5.0
```

Repeats `rep(x, times=n)` repeats the vector `x` `n` times.

```
rep(1:3, times=3)

## [1] 1 2 3 1 2 3 1 2 3
```

`rep(x, each=n)` repeats each element of the vector `x` `n` times.

```
rep(1:3, each=3)

## [1] 1 1 1 2 2 2 3 3 3
```

Task 6.

Create each of the following vectors using `:`. `seq` and `rep`.

```
2 3 4 5 6
2 4 6
1.00 1.25 1.50 1.75 2.00
3 3 4 4 5 5
2 3 4 2 3 4
```

Useful functions for vectors `numeric(n)` creates a numeric vector of length `n` (containing 0's).

`length(x)` returns the length (number of elements) of the vector `x`.

`unique(x)` returns the unique elements of `x`.

`rev(x)` reverses the vector `x`, i.e. returns $(x_n \ldots, x_1)$

Sorting The function `sort(x)` sorts the vector `x`. The function `order(x)` returns the permutation required to sort the vector `x` Consider the vector

```
x <- c(11, 7, 3, 9, 4)
```

We can sort `x` using

```
sort(x)
```

```
## [1]  3  4  7  9 11
```

So what does the function order do?

```
p <- order(x)
p
```

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

The first entry of the result `p` is 3. The third entry of `x` is the smallest: if we want to sort `x` we have to put the third entry first, or, in other words, the first entry of the sorted vector would be the third entry of `x`. The second entry of `p` is 5, because the second smallest entry of `x` is its fifth entry.

We can obtain the sorted vector by applying the permutation obtained from `order` to `x`

```
x[p]
```

```
## [1]  3  4  7  9 11
```

This trick can be used to sort an entire dataset by one column.

If we want to sort the vector is descending order, we have to append the argument `decreasing=TRUE`.

```
sort(x, decreasing=TRUE)
```

```
## [1] 11  9  7  4  3
```

Scalar summary functions

R has a large selection of functions that compute a scalar function of a vector. Their names are self-explanatory: `min`, `max`, `sum`, `prod`, `mean`, `median`, `sd`, `var`, ...

Task 7.

Use R to calculate the sums $\sum_{i=1}^{10} i$ and $\sum_{i=1}^{100} i^2$.

Example 3 (A simple Monte Carlo experiment).

Suppose we want to know the probability

$$\mathbb{P}(X > 1)$$

if X has a standard normal distribution ($X \sim \mathrm{N}(0, 1)$). (You will learn about the normal distribution in your probability courses. For now, treat this as a range of data we believe X falls within.)

We could try to answer this problem using a simulation ("Monte Carlo") as follows. (You will learn more about Monte Carlo methods in the Uncertainty Assessment and Bayesian Computation course).

- We first generate a large number of realisations from the standard normal distribution. This can be done using the built-in function `rnorm`.
- We then count the proportion of values which are greater than 1. This is an estimate of the probability we are looking for.

```
n <- 1e7        # Set sample size (the bigger the better, within reason)
x <- rnorm(n)   # Draw a sample of size n from the standard normal distribution
sum(x>1)/n      # Our estimate of the probability
```

```
## [1] 0.1588362
```

How does this work? `x>1` returns a logical vector of the same length as `x`. Its i-th entry is `TRUE` if $x_i > 1$, otherwise it is `FALSE`. The number of `TRUE`s in `x>1` is thus the number of times x_i is greater than 1. Summing over a logical vector counts the number of `TRUE` as R views `TRUE` as 1 and `FALSE` as 0.

We could have used the cumulative distribution function of the standard normal distribution, which gives an exact result for this question.

```
pnorm(1, lower.tail=FALSE)
```

```
## [1] 0.1586553
```

You will soon learn more cumulative distribution functions in the Probability and Stochastic Models course or in the Probability and Sampling Fundamentals course.

Other data types in R

So far we have considered vectors which were either numerical or logical. In this section we will look at other data types in R.

Character strings

Character strings in R are defined using double or single quotes.

```
string <- "A string in single quotes"
another.string <- 'This time defined using single quotes'
```

Supplementary material:
Line breaks and escaped characters

A new line can be inserted into a string using `\n`.

```
string.with.newline <- "Text\nText on a new line"
```

Quotes and backslash must be “escaped” when used inside strings.

```
string.with.escapes <- "This \" is a quote and this \\\\ is a single backslash"
```

Strings can be printed using `print`. As an example (or just the variable name)

```
string
```

```
## [1] "A string in single quotes"
```

Use the function `cat` to print the string as it is

```
cat(string)
```

```
## A string in single quotes
```

R has a variety of built-in functions for manipulating strings. It is however easier to use the functions from the package stringr (which is a part of the Tidyverse suite of packages, which we will look at in Week 4).

For example, two strings can be concatenated using the function `str_c` (the equivalent base R function is `paste`).

```
library(stringr)
str_c("Two strings", "put together", sep=" - ")
```

```
## [1] "Two strings - put together"
```

Supplementary material:
An operator for string concatenation

In contrast to many other programming languages R has no built-in operator for concatenating strings. However, R lets you define your own operators: you can define your own as follows.

```
"%.%" <- function(lhs, rhs) {
  str_c(lhs, rhs)
}
left <- "Two strings"
mid <- " - "
right <- "put together"
left %.% mid %.% right
```

```
## [1] "Two strings - put together"
```

If you are wondering about the use of percent signs in the operator, R requires this to make sure it is being recognised as an operator.

Note that you cannot use character vectors for any sort of arithmetic. For example

```
"120" + "5"
```

```
Error in "120" + "5" : non-numeric argument to binary operator
```

You are unlikely to do this deliberately, but you might come across it if R treats a supposedly numerical variable as a character string because of at least one non-numerical entry.

Comparisons between character strings use lexicographic ordering, i.e. they are compared based on where they would be in a dictionary. For example

```
"apple">"pear"
```

```
## [1] FALSE
```

However character strings containing numbers are compared in an unexpected way

```
"120" > "5"
```

```
## [1] FALSE
```

In lexicographic ordering "5" comes after "120" (as only the first digit "1" matters).

Factors

Factors are a variant on that theme and designed to be used with categorical variables. The main difference between a factor and a character vector is that factors are only allowed to take a pre-defined set of values ("levels"). You will learn more about factors in Learning From Data/Data Science Fundamentals.

Any vector can be converted to a factor using the function `factor`. It has the additional (optional) arguments `levels` and `labels` which can be used to set the labels printed when a vector is displayed.

```
x <- c(1, 4, 2, 4, 1, 3, 1, 2, 4)
X <- factor(x, levels=1:5, labels=c("one", "two", "three", "four", "five"))
X
```

```
## [1] one   four  two   four  one   three one   two   four
## Levels: one two three four five
```

In our example, the factor `X` is only allowed to take the values `"one"`, `"two"`, `"three"`, `"four"` and `"five"` (the level `"five"` is currently not being used). Thus we can set for example

```
X[1] <- "five"
X
```

```
## [1] five  four  two   four  one   three one   two   four
## Levels: one two three four five
```

We cannot set the first entry to `"six"`. `"six"` is not in the set of allowed labels.

```
X[1] <- "six"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "six"): invalid factor level, NA generated
```

```
X
```

```
## [1] <NA>  four  two   four  one   three one   two   four
## Levels: one two three four five
```

In order to be able to set the first entry to `"six"` we need to first expand the set of levels.

```
levels(X) <- c(levels(X), "six")
X[1] <- "six"
X
```

```
## [1] six   four  two   four  one   three one   two   four
## Levels: one two three four five six
```

To turn `X` back into its original numerical format we can use the function `unclass`.

```
unclass(X)
```

```
## [1] 6 4 2 4 1 3 1 2 4
## attr(,"levels")
## [1] "one"   "two"   "three" "four"  "five"  "six"
```

Converting between data types

You can convert between different data types by using `as.<target-datatype>`. For example you can convert

```
x <- pi                  # x is numeric
x <- as.character(x)     # now x is a character string
x <- as.numeric(x)       # x is numeric again (but we lost some digits)
```

Often R will convert between different types automatically. The most common data types are

Data type	Conversion function	Description
numeric	`as.numeric`	Floating point numbers
integer	`as.integer`	Integer numbers
logical	`as.logical`	`TRUE` or `FALSE`
character	`as.character`	Character string
factor	`as.factor`	Factor

Missing values

R uses the special value `NA` to indicate that a value is missing. It is different from `NaN`, which is "not a number", i.e. a value for which the calculations have gone wrong.

You cat set a value to `NA` by simply assigning it the value `NA`.

```
x <- 1:4
x[4] <- NA
x
```

```
## [1]  1  2  3 NA
```

Calculations (arithmetic, summary functions, etc.) involving `NA`s have the result set to `NA` as well: the result depends on the value that is not available.

```
mean(x)
```

```
## [1] NA
```

If you want R to omit the missing values you can either use

```
mean(na.omit(x))
```

```
## [1] 2
```

or

```
mean(x, na.rm=TRUE)
```

```
## [1] 2
```

The former is more generic as not every function support the additional argument of `na.rm=TRUE`.

Use the function `is.na` to test whether a value is missing.

```
is.na(x)
```

```
## [1] FALSE FALSE FALSE  TRUE
```

You cannot use == to test whether a value is missing

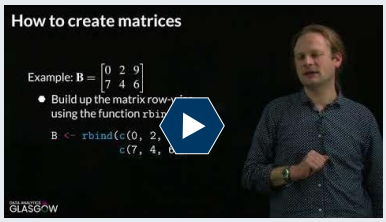
```
x==NA
```

```
## [1] NA NA NA NA
```

The comparison just results in `NA`, use `is.na` instead.

START HERE

Matrices



Matrices

https://youtu.be/slxsFdCNfkk

Duration: 6m45s

Matrices in R

Matrices are the two-dimensional generalisation of vectors. The main difference between a vector and a matrix is that a vector has a single index, whereas a matrix has two indexes: row and column.

Internally, R stores matrices in column-major mode, i.e. the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

is stored as

```
1 2 3 4 5 6 7 8 9
```

i.e. R internally stacks the columns on top of each other, which is known as "column-major mode". If you had stored the matrix $\mathbf{A}$ is a two-dimensional array in C or Java, it would be stored in what is called "row-major mode", i.e. the rows would be stacked on top of each other.

Creating matrices

There are essentially three ways of creating a matrix in R.

Using the internal representation The first one consists of using the internal representation of matrices as vectors. If we want to create a matrix

$$\mathbf{B} = \begin{bmatrix} 0 & 2 & 9 \\ 7 & 4 & 6 \end{bmatrix}$$

we can use the command `matrix`.

```
B <- matrix(c(0, 7, 2, 4, 9, 6), nrow=2)
```

Alternatively, you could specify the number of columns using `ncol=3`.

The function `matrix` can also be used to create "empty" matrices. `matrix(a, nrow, ncol)` creates a `nrow`×`ncol` matrix in which every entry is set to `a`.

Row-wise build-up Another option is to build up the matrix row-wise using the function `rbind`.

```
B <- rbind(c(0, 2, 9),
           c(7, 4, 6))
```

We can also use `rbind` to add a row to an existing matrix.

```
rbind(B, c(1, 2, 9))
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
## [3,]    1    2    9
```

If a vector given to `rbind` is shorter than the other rows, it is "recycled" using the same rules as used for vector arithmetic. For example, to add a row of 0's to the matrix $\mathbf{B}$ we can use

```
rbind(B, 0)
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
## [3,]    0    0    0
```

Column-wise build-up The third option consists of using `rbind`'s sibling `cbind`. `cbind` adds a column to a matrix and can be used to build up a matrix column-wise. Thus we can create the matrix $\mathbf{B}$ using

```
B <- cbind(c(0, 7), c(2, 4), c(9, 6))
```

Task 8.

Use all three methods fom above to create the matrix

$$\mathbf{M} = \begin{bmatrix} 9 & 2 & 4 \\ 3 & -2 & 7 \\ 4 & 8 & -1 \end{bmatrix}$$

Dimensions of a matrix To find out the dimensions of a matrix you can use the three functions `nrow`, `ncol` and `dim`.

```
nrow(B)
```

```
## [1] 2
```

```
ncol(B)
```

```
## [1] 3
```

```
dim(B)
```

```
## [1] 2 3
```

The function `length` returns the number of entries of a matrix ($2 \times 3 = 6$ in our case)

```
length(B)
```

```
## [1] 6
```

Diagonal matrices

Diagonal matrices have a special role in Linear Algebra and thus in Statistics. For this reason R has a function dedicated to diagonal matrices: `diag`.

```
E <- diag(c(1 ,4 , 2))
E
```

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

`diag` can also be used to access the diagonal of an existing matrix. The matrix does not even need to be diagonal for this. You can change the second element of the diagonal of the matrix `E` to `5` using

```
diag(E)[2] <- 5
E
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    5    0
## [3,]    0    0    2
```

Task 9.

Create the diagonal matrix

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & -9 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

in R.

Supplementary material:
Sparse matrices

Sometimes we have to work with matrices which contain mostly zeros. Diagonal matrices are a prime example. Standard R matrices explicitly store all the entries of a matrix ("dense storage"). This is not very efficient. A zero entry costs as much storage space as a non-zero entry. Calculations with matrices involving many zero-entries are as slow as the ones for matrices with no zeros.

Consider a diagonal matrix. Storing it densley requires storing $n \times n = n^2$ entries even though it would be enough to only store the n values on the diagonal. Explicitly storing the off-diagonal zeros also makes it impossible to exploit the special structure of diagonal matrices, which would allow for a significant speed-up of calculations.

Sparse matrices only store the non-zero entries. For a matrix with many zeros ($\gg 90\%$) this can be much more efficient, both in terms of storage space and computational speed. There are several R packages, such as Matrix or spam which use sparse matrices.

We will look at a small example using the `Matrix` package. We start by using dense matrices to represent and invert a 5000×5000 matrix.

```
A <- diag(1:5e3)          # Create a dense matrix A
object.size(A)            # Estimate storage space required
system.time(solve(A))     # Get timing for matrix inversion
                          # Third number is the time required in seconds
```

```
##    user  system elapsed
##  10.513   0.143  10.684
```

We now use a sparse matrix structure using the `Diagonal()` command.

```
library(Matrix)           # Make commands from package Matrix available
A <- Diagonal(x=1:5e3)    # Create a sparse matrix A
object.size(A)            # Estimate storage space required
system.time(solve(A))     # Get timing for matrix inversion
                          # Third number is the time required in seconds
```

```
##    user  system elapsed
##       0       0       0
```

Naming rows and columns

When working with data matrices it is a good idea to name at least the columns of a matrix. It is much better to refer to variables in a matrix by their name rather than the column index in which they are stored.

Rows and columns can be named using the functions `rownames` and `colnames`.

```
colnames(B) <- c("First column", "Second column", "Third column")
rownames(B) <- c("First row", "Second row")
B
```

```
##            First column Second column Third column
## First row             0             2            9
## Second row            7             4            6
```

Accessing elements and submatrices

Single entries of matrices can be accessed using square brackets. To extract B_{23}, i.e. the value in the second row and third column of $\mathbf{B}$ we can use

```
B[2,3]
```

```
## [1] 6
```

Similarly, we can set B_{23} to `-1` using

```
B[2,3] <- -1
```

Though this is not recommended, we could have also used the internal vector representation and extract the sixth element of the internal representation

```
B[6]
```

```
## [1] -1
```

You can access arbitrary submatrices by specifying the rows and columns you wish to access. You can do so by using any combination of the three methods used for vectors and mentioned in the third handout. To extract the first row and first and second column of $\mathbf{B}$ you can use any of the following lines (. . . and there are many other ways of doing so):

```
B[1, 1:2]
```

```
##  First column Second column
##             0             2
```

```
B[-2, 1:2]
```

```
##  First column Second column
##             0             2
```

```
B[c(TRUE, FALSE), -3]
```

```
##  First column Second column
##             0             2
```

Each line returns the vector $[0, 2]$

Because the output object has only one row it is returned as a vector (instead of a matrix). In complex programmes in which the result is then expected to be matrix this can sometimes cause problems. In this case you can use the additional argument `drop=FALSE`:

```
B[1, 1:2, drop=FALSE]
```

```
##           First column Second column
## First row            0             2
```

If we only want to subset rows and columns the selector for the other dimension can be left empty. To access the first row of the matrix $\mathbf{B}$ use

```
B[1,]
```

```
##  First column Second column  Third column
##             0             2             9
```

To access the third column of $\mathbf{B}$ use

```
B[, 3]
```

```
##  First row Second row
##          9         -1
```

To replace the third column of $\mathbf{B}$ with the numbers 1 and 8 you can use

```
B[, 3] <- c(1, 8)
```

Furthermore, logical expressions can be used to subset matrices in the same way as they are used to subset vectors. Suppose you want to set all entries larger than 5 to 6.

```
B[B > 5] <- 6
B
```

```
##            First column Second column Third column
## First row             0             2            1
## Second row            6             4            6
```

Task 10.

Use the matrix $\mathbf{M}$ from task 8

- extract the first row,
- set the top-right entry to 0,
- add 1 to the last column.

Matrix multiplication and basic linear algebra

Matrix multiplication The basic arithmetic operators can be applied to matrices in the same way as they can be applied to vectors. They are interpreted element-wise. Most importantly, `*` performs element-wise multiplication. In order to perform matrix multiplication you need to use the operator `%*%`.

To compute the matrix product

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 9 \\ 7 & 4 & -1 \end{bmatrix}$$

in R, you can use:

```
A <- matrix(c(1, 0, -1, -1, 1, 0), ncol=2)
B <- matrix(c(0, 7, 2, 4, 9, -1), ncol=3)
A %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   -7   -2   10
## [2,]    7    4   -1
## [3,]    0   -2   -9
```

Note that matrix multiplication is generally not commutative (unless $\mathbf{A}$ and $\mathbf{B}$ are symmetric), thus `A%*%B` is not the same as `B%*%A`.

The transpose $\mathbf{A}^\top$ of a matrix $\mathbf{A}$ can be computed using the function `t(A)`. The cross product $\mathbf{A}^\top\mathbf{A}$ can be computed using the function `crossprod(A)`.

Supplementary material:
Matrix inverse and linear systems of equations

The function `solve` computes the matrix inverse. For example,

```
C <- B%*%t(B)             # Create a square matrix C=BB'
C.inv <- solve(C)         # Compute the inverse of C
C.inv%*%C                 # Check whether C.inv is indeed the inverse
```

```
##      [,1]         [,2]
## [1,]    1 7.589415e-19
## [2,]    0 1.000000e+00
```

```
C%*%C.inv
```

```
##              [,1] [,2]
## [1,] 1.000000e+00    0
## [2,] 7.589415e-19    1
```

The function `solve` can be used not only for inverting a matrix, but also for solving a (non degenerate) system of linear equations. `solve(A, b)` solves the system of equations $\mathbf{Az} = \mathbf{b}$ for $\mathbf{z}$.

To solve the system of equations

$$\begin{aligned} 5z_2 + z_3 &= 7 \\ 7z_2 - z_3 &= 5 \\ 11z_1 + z_2 + z_3 &= 14 \end{aligned}$$

you can use

```
A <- rbind(c( 0, 5,  1),
           c( 0, 7, -1),
           c(11, 1,  1))
b <- c(7, 5, 14)
z <- solve(A, b)
z
```

```
## [1] 1 1 2
```

We can check the answer by computing $\mathbf{Az}$, which should then be $\mathbf{b}$.

```
A%*%z
```

```
##      [,1]
## [1,]    7
## [2,]    5
## [3,]   14
```

You could as well first compute the inverse of $\mathbf{A}$ and then compute $\mathbf{A}^{-1}\mathbf{b}$, i.e. used

```
z <- solve(A)%*%b
```

but this is slightly less efficient than the above code.

Linear systems of equations play a key role in Data Analytics and Statistics. For example, you will learn in Predictive Modelling that we can find the least-squares estimate of the coefficients in a linear regression model by computing

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y},$$

which we can rewrite as solving the system of linear equations

$$\underbrace{(\mathbf{X}^\top\mathbf{X})}_{=\mathbf{A}}\underbrace{\boldsymbol{\beta}}_{=\mathbf{z}} = \underbrace{\mathbf{X}^\top\mathbf{y}}_{=\mathbf{b}}$$

Consider a simple regression problem using the cars dataset (which is included within R, simply type `cars` to view this). To compute the regression coefficients for the cars dataset, we can use the following code

```
y <- cars$dist              # Set y (response) to distance
X <- cbind(1, cars$speed)   # Set X (covariate) to a column of ones and speed
A <- t(X)%*%X               # Assemble matrix A
b <- t(X)%*%y               # Assemble vector b
beta <- solve(A, b)         # Solve for z (which is beta)
beta
```

```
##            [,1]
## [1,] -17.579095
## [2,]   3.932409
```

This is exactly the same answer as have obtained from the built-in function for linear regression.

```
lm(dist~speed, data=cars)$coef
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

It turns out that the above way of calculating $\boldsymbol{\beta}$ is numerically less stable than necessary. Later on in the programme you will look at how this calculation be done in a numerically more stable way.

Supplementary material:
Matrix multiplication is associative, but parentheses might matter

Matrix multiplication is associative, i.e. for any two matrices $\mathbf{A}$ and $\mathbf{B}$ and a vector $\mathbf{x}$ we have that $(\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{x} = \mathbf{A}\cdot(\mathbf{B}\cdot\mathbf{x})$.

Let's create matrices `A` and `B` and a vector `x`.

```
A <- matrix(rnorm(9e6), nrow=3e3)    # Create matrix A with random entries (3000x3000)
B <- matrix(rnorm(9e6), nrow=3e3)    # Create matrix B with random entries (3000x3000)
x <- rnorm(3e3)                      # Create vector x with random entries (3000x1)
```

Because of the associativity of matrix multiplication both

```
(A%*%B)%*%x
```

which is equivalent to `A%*%B%*%x`, and

```
A%*%(B%*%x)
```

give, except for rounding errors, the same answer. However the first calculation is a lot slower.

```
system.time((A%*%B)%*%x)
```

```
##    user  system elapsed
##   8.821   0.039   8.972
```

```
system.time(A%*%(B%*%x))
```

```
##    user  system elapsed
##   0.011   0.000   0.012
```

Why?

Multiplying to matrices is much slower than computing the product of a matrix (of the same size) and a vector.

The first line of code multiplies the two matrices $\mathbf{A}$ and $\mathbf{B}$ first (which takes very long), and then multiplies the result by the vector $\mathbf{x}$.

The second line of code never multiplies two matrices. The result of $\mathbf{B} \cdot \mathbf{x}$ is another vector of length 1000. $\mathbf{A}$ is then multiplied by this vector.

A more detailed answer

This question is about what is called "FLOP counting". FLOP stands for "floating point operation", i.e. an addition or multiplication.

To multiply two matrices $\mathbf{C}$ and $\mathbf{D}$ of dimensions $m \times k$ and $k \times n$, we need to compute the $m \times n$ entries of the resulting matrix $\mathbf{E} = \mathbf{C} \cdot \mathbf{D}$. Each entry E_{ij} of $\mathbf{E}$ is a sum $E_{ij} = \sum_{\iota=1}^{k} C_{i\iota} C_{\iota j}$. To compute this sum we need to carry out k multiplications and k additions, thus we require $2k$ FLOPs. There are $m \cdot n$ entries in the resulting matrix $\mathbf{E}$, thus the overall computational cost of computing $\mathbf{E} = \mathbf{C} \cdot \mathbf{D}$ is $2kmn$ FLOPs.

Multiplying a $m \times k$ matrix $\mathbf{C}$ with a vector $\mathbf{y}$ of length k is the same as multiplying a $m \times k$ matrix by a $k \times 1$ matrix, thus we need only $2mk$ FLOPS.

In the first line of code we first compute $\mathbf{A} \cdot \mathbf{B}$, and then multiply the result by $\mathbf{x}$. $\mathbf{A}$ and $\mathbf{B}$ are 1000×1000 matrices, so computing $\mathbf{A} \cdot \mathbf{B}$ requires $2 \cdot 10^9$ FLOPs. Multiplying the result by $\mathbf{x}$ adds another $2 \cdot 10^6$ FLOPs, so the first line takes $2.002 \cdot 10^9$ FLOPs.

In the second line of code we first compute $\mathbf{B} \cdot \mathbf{x}$, which requires $2 \cdot 10^6$ FLOPs. Then $\mathbf{A}$ is multiplied by this vector, which again requires $2 \cdot 10^6$ FLOPs. Thus computing the second line requires only $4 \cdot 10^6$ FLOPs.

Lists in R

Lists

https://youtu.be/-MumGJIIrTI

Duration: 6m46s

Lists allow us to combine several variables into one object. In contrast to regular vectors and matrices, the entries of a list do not all have to be of the same data type.

Creating lists You can create a list using the `list` command.

```
example1 <- list(1:3, c(TRUE,FALSE), 7)
example1
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] 7
```

Lists can be nested within each other.

```
example2 <- list(list(pi, 1:2), list("some text", 1:3))
example2
```

```
## [[1]]
## [[1]][[1]]
## [1] 3.141593
##
## [[1]][[2]]
## [1] 1 2
##
##
## [[2]]
## [[2]][[1]]
## [1] "some text"
##
## [[2]][[2]]
## [1] 1 2 3
```

Just like vectors, lists can be concatenated using the function `c`. The fist three entries of the resulting list will come from `example1`. the remaining fourth and fifth entry from `example2`.

```
c(example1, example2)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] 7
##
## [[4]]
## [[4]][[1]]
## [1] 3.141593
##
## [[4]][[2]]
## [1] 1 2
##
##
## [[5]]
## [[5]][[1]]
## [1] "some text"
##
## [[5]][[2]]
## [1] 1 2 3
```

It is good programming style to name the elements of a list (if possible). This can be done using `names`:

```
names(example1) <- c("a", "b", "c")
example1
```

```
## $a
## [1] 1 2 3
##
## $b
## [1]  TRUE FALSE
##
## $c
## [1] 7
```

Alternatively, you can specify the names directly in the `list` command:

```
example1 <- list(a=1:3, b=c(TRUE,FALSE), c=7)
```

Accessing elements Elements of a list can be accessed using either double square brackets or, if the list has names, `$`. The following three lines of R code all return the first entry of the list `example1` (using either its position or its name).

```
example1[[1]]
```

```
## [1] 1 2 3
```

```
example1[["a"]]
```

```
## [1] 1 2 3
```

```
example1$a
```

```
## [1] 1 2 3
```

If you want to subset a list (in the sense of extracting more than one entry) use single square brackets:

```
example1[1:2]
```

```
## $a
## [1] 1 2 3
##
## $b
## [1]  TRUE FALSE
```

If we extract an individual entry using single square brackets we obtain a list containing that entry rather than the entry itself.

```
example1[1]
```

```
## $a
## [1] 1 2 3
```

Lists are everywhere ... Most R functions return lists. Take the `lm` function as an example. The function `lm` fits a linear regression model, which is essentially a straight line fit to the data. First of all we fit a linear model using the cars dataset within R:

```
fit <- lm(dist~speed, data=cars)          # Fit a linear model
fit                                       # Print the model
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

The output object `fit` is in fact a list.

```
is.list(fit)
```

```
## [1] TRUE
```

Let's look at its entries

```
names(fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values" "assign"
##  [8] "df.residual"   "xlevels"       "call"          "terms"         "model"
```

So, if we for example want to retrieve the regression coefficients we can use

```
fit$coefficients
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

Actually, we do not have to fully spell out the name of the entry after the `$`, we only have to use the first few characters until the name is uniquely determined. So we could have also used

```
fit$coef
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

We could have not used

```
fit$c
```

```
## NULL
```

as there is more than one element with a name starting with a "c" (`"call"` and `"coefficients"`).

Supplementary material:
R classes and lists

R classes are typically also based on lists. A list becomes a class when it has an attribute called `"class"`.

```
attr(fit, "class")
```

```
## [1] "lm"
```

So in the above example, the class of the linear model fit is `"lm"`. This tells R which functions to use to print, plot or summarise this object. Later on in the course we will look at how this (compared to other programming languages) rather simplistic concept of object-oriented programming works.

Answers to tasks

Answer to Task 1. If we use R to work out the answers we obtain

```
!a & b
```

```
## [1] FALSE
```

```
a | !b
```

```
## [1] TRUE
```

```
!(a | !b)
```

```
## [1] FALSE
```

```
(a & b) | c
```

```
## [1] TRUE
```

Answer to Task 2. We can use the following R code.

```
x <- 355 / 113
x < pi
```

```
## [1] FALSE
```

```
x>3 & x<4
```

```
## [1] TRUE
```

```
abs(x-pi) < 1e-6
```

```
## [1] TRUE
```

Answer to Task 3. We can use the following R code.

```
x <- c(1, 3, 2, 5)
y <- c(1, 0)
z <- c(x, y)
z
```

```
## [1] 1 3 2 5 1 0
```

Answer to Task 4. We can use the following R code (there are many other equally good answers).

```
x[c(1,3,5)]
```

```
## [1] 1 9 8
```

```
x[-c(2,4)]
```

```
## [1] 1 9 8
```

```
x[c(TRUE,FALSE,TRUE,FALSE,TRUE)]
```

```
## [1] 1 9 8
```

Answer to Task 5. The recycling rules mean that R treats `x+y` as `x+yyy` with

```
yyy <- c(0, 2, 0, 2, 0, 2)
x+yyy
```

```
## [1] 1 4 9 6 5 8
```

Answer to Task 6. We can use the following R code (there are many other equally good answers).

```
2:6
```

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

```
seq(2, 4, by=2)
```

```
## [1] 2 4
```

```
seq(1, 2, length.out=5)
```

```
## [1] 1.00 1.25 1.50 1.75 2.00
```

```
rep(3:5, each=2)
```

```
## [1] 3 3 4 4 5 5
```

```
rep(2:4, 2)
```

```
## [1] 2 3 4 2 3 4
```

Answer to Task 7. We can calculate $\sum_{i=1}^{10} i$ using

```
sum(1:10)
```

```
## [1] 55
```

We can calculate $\sum_{i=1}^{100} i^2$ using

```
sum((1:100)^2)    # correct
```

```
## [1] 338350
```

Note that we need to put the power of 2 inside parenthesis. If we had used

```
sum(1:100)^2      # NOT correct
```

```
## [1] 25502500
```

we would have calculated $\left(\sum_{i=1}^{100} i\right)^2$.

We also have to put `1:100` inside parentheses. If we had used

```
sum(1:100^2)      # NOT correct
```

```
## [1] 50005000
```

we would have calculated $\sum_{i=1}^{100^2} i$.

Answer to Task 8. Using the internal representation …

```
M <- matrix(c(9,3,4,2,-2,8,4,7,-1), ncol=3)
```

Using `rbind` …

```
M <- rbind(c( 9, 2, 4),
           c( 3,-2, 7),
       c( 4, 8,-1))
```

Using `cbind` …

```
M <- cbind(c( 9, 3, 4),
           c( 2,-2, 8),
       c( 4, 7,-1))
```

Answer to Task 9. You can use the following R code.

```
diag(c(4,7,-9,4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4    0    0    0
## [2,]    0    7    0    0
## [3,]    0    0   -9    0
## [4,]    0    0    0    4
```

Answer to Task 10. You can use the following R code.

```
M[1,]                # Extract first row
```

```
## [1] 9 2 4
```

```
M[1,3] <- 0          # Set top-right entry to 0.
M[,3] <- M[,3] + 1   # Add 1 to the last column
```