Using R for Data Analysis and Graphics (Part I)

Fynn Krebser–fkrebser@student.ethz.ch

October 21, 2025

1 Introduction to R

R is a programming language for statistical computing. It is useful as it is reproducible, covers most of statistical methods and has a strong graphial library. A big competitor is Python, however at the current time it is less specialized for statistical methods.

Inside the R console, a > symbol indicates the beginning of a new statement, whilst a + symbol indicates that the current statement is not yet complete.

In R, the assignment operator is `<-`. In principle, the equal sign `=` can also be used, but it has some strange surprises so it is best to avoid it.

Lines starting with a `#` symbol are comments and thus ignored by R.

1.1 Naming rules

Variable names can consist of letters, digits, periods and underscores. However, they must start with a letter or a period. When starting with a period, the second character cannot be a digit. Variable names are case sensitive. Reserved words (like `if`, `TRUE`, `NULL`, etc.) cannot be used as variable names.

1.2 Functions

Functions are called using the syntax `function(arg1, arg2, ...)`. The arguments can be seen by using the `args()` function. Sometimes this isn't that useful, so you can also use `?function` to get documentation on the function.[1]

When calling functions, it is common to just use the first argument by position and then use named arguments for the rest.

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

1.3 Leaving R

When closing R, it will ask you to save the workspace image. It is strongly recommended to answer `no` to this question.

[1]Obviously, this will not be available in the exam.

2 Basics

A vector can be created using the `c()` function (combine).

```
> x <- c(1, 2, 3, 4, 5, 6)
> y <- c(1.3, -2.5, 7.9)
```

Possible shortcuts are:

```
> x <- 1:6 # creates a vector with values from 1 to 6
> y <- seq(1, 3, by=0.5) # creates a vector from 1 to 3
↪  with step size 0.5
> z <- rep(0, times=5) # creates a vector with five 0s
> w <- rep(c(1, 2, 3), times=3) # creates a vector
↪  1,2,3,1,2,3,1,2,3
> v <- rep(c(1,2), lenght=5) # creates a vector
↪  1,2,1,2,1
```

For vectors the following operations are defined:

```
> length(v)
> sum(v)
> mean(v)
> var(v)
> range(v) # min and max
> min(v)
> max(v)
```

A nice thing about R is that one can apply operations to whole vectors easily. For example `2*v` will multiply each element of `v` by 2.

This can also be combined for example for vectors `x` and `y`, `2*x - y` will do what one would expect.

If the vectors are of different lengths, R will recycle the shorter one. For example if v = (4, 2, 7, 8 , 2) and b = (3.1, 5.0, -0.7) then v - b will be (4-3.1, 2-5.0, 7-(-0.7), 8-3.1, 2-5.0).

Elements of a vector can be accessed using square brackets:

`x[2] # second element of x`

Or a range of elements:

`x[c(1,1,4,5,1)]`

2.1 Character Vectors

Character vectors can be created in the same way as numeric vectors.

`names <- c("Alice", "Bob", "Charlie")`

Possible operations are:

```
> substring(names, 1, 2) # first two letters of each
↪  name
> nchar(names) # number of characters in each name
> paste("Hello", names, sep = " @ ") # concatenate
↪  "Hello" to each name with " @ " in between
```

2.2 Logical Vectors

Logical vectors can be created using the keywords `TRUE` and `FALSE`.

`w <- c(TURE, FALSE, FALSE)`

This can be combined with relational operators:

```
> x <- c(1, 2, 3, 4, 5)
> y <- c(6, 8, 3, 5, 7)
> x < y
[1] TRUE TRUE FALSE TRUE TRUE
> x == y
[1] FALSE FALSE TRUE FALSE FALSE
```

Logical vectorrs can be used to index other vectors:

```
> x[x < 3] # elements of x that are less than 3
[1] 1 2
```

2.3 Data Frames

Data frames can be read using the `read.table()` function. The most useful arguments are:

- `header=TRUE` if the first row contains the column names
- `sep=","` if the file is comma separated (CSV file)
- `na.strings=c("","NA")` is used to specify which strings should be treated as NA
- dec="." to specify the decimal point character

Similarly, data frames can be written using the function `write.table()`.

R also has the possibility to store any object to a binary Rdata file.

```
> x <- 1:20
> y <- sport_results[, "shotpun"]
> save(x, y, file="mydata.Rdata")
> load("mydata.Rdata")
```

2.4 Selecting elements

Elements are accessed using square brackets, but with two indices: `[i, j]` where `i` is the row index and `j` the column index. Notice, indices start at 1 in R.

In R, its also possible to use a negative index to exclude elements. Furthermore, logical vectors can be used to select elements.

```
> sport_results[sport_results[,"shotpun"] > 20, ] # all
↪  rows where shotput is greater than 20
```

2.5 matrices

In a matrix, all elements must be of the same type (numeric, character or logical). This is different from data frames.

The function `t()` can be used to transpose a matrix.

The element wise multiplication of two matrices can be done using `A * B`. The matrix multiplication can be done using `A \%*\% B`.

3 Simple Statistics

When working with vectors, `length(v)` returns the number of elements in the vector. For data.frames and matrices, dimensions can be obtained using `dim(df)`. `nrow(df)` and `ncol(df)` return the number of rows and columns, respectively. Applying theese functions to a vector returns `NULL`. However, `NROW(v)` and `NCOL(v)` return the length of the vector and 1, respectively.

Another very useful function is `str(df)` which gives a concise summary of the structure of an R object. The function can even be applied to another function, but this shows just the mandatory arguments.

To go beyond the structure of a data frame, `summary(df)` gives a summary of each column. For numeric columns, this includes the min, quartiles, mean and max.

Other useful functions for numeric vectors include:

- `mean(v)`: arithmetic mean
- `median(v)`: median
- `var(v)`: variance
- `sd(v)`: standard deviation
- `quantile(v, probs=c(0.25, 0.75))`: specified quantiles (here 1st and 3rd quartiles)
- `cor(v1, v2)`: correlation between two vectors

The `cor()` function only checks for linear relationships.

3.1 Factors

A Factor is a categorical variable. It can take values (called levels) from a fixed, finite set.

A common problem is using a integer vector to represent a categorical data. This is dangerous as R will treat it as numeric data.

To produce a factor variable, create a vector and then do something like

`results[, "team"] <- as.factor(team_name)`

When having a factor in a df, `levels(df[, factor_col])` returns the levels of the factor.

The `cut()` function can be used to convert a numeric vector into a factor by cutting it into intervals. It has an argument `breaks` which can be used to specify the intervals.

3.2 Comparison of Groups

A comparison can be done using for example a boxplot:

```
> boxplot(y1, y2, y3, names=levels(results[, "team"]))
> plot(shotpun ~ team, data=results)
```

3.3 Hypothesis Test

The wilcoxon rank sum test can be used to test if two groups are different without an assumption of normality. It is done using the function

`wilcox.test(y1, y2, paired = FALSE)`.

4 Missing Values

In R, missing values are represented by `NA`. This is useful as we do not need to give a special value to represent missing data.

To see if a value is missing, use `is.na(x)` which returns a logical vector. Importantly, `x == NA` does not work as expected as `NA` is not a value.

Most simple statistical functions have an argument `na.rm` which can be set to `TRUE` to remove missing values before the calculation.

To remove rows with missing values in a data frame, use the `na.omit(df)`. Something similar can be done using the statement `df[complete.cases(df), ]`.

If one gets a dataset that encodes missing values with a special value (e.g., -99), one can convert these to `NA` using the argument `na.strings = c(".", "-99")` in the function `read.table()`.

The data can also be cleaned after reading it in using

```
> df[df == -99] <- NA
```

5 Functions

A function is made using the syntax

```
> my_function <- function(arg1, arg2, ...) {
+     # function body
+     return(value)
+ }
>
> my_function(val1, val2, ...)
```

It is recommended to always document functions using comments. A common way is to write a one-liner followed by the arguments and then detailed description

We can also use the `apply()` function to apply a function to the rows or columns of a matrix or data frame. The syntax is `apply(X, MARGIN, FUN, ...)` where `X` is the matrix or data frame, `MARGIN` is 1 for rows and 2 for columns, `FUN` is the function to apply.

Calling a function without any arguments returns simply the function body.

When putting functions in a seperate file, they can be loaded using `source("file.R")`.

To make a named vector, use the following syntax

`v <- c(name1=val1, name2=val2, ...)`.

6 Scatter and Boxplots

We have seen some diffrent ways to make plots. For example using hist, plot, pairs and boxplot.

There exist diffrent kinds of plotting functions.

High-level plotting functions create a new plot.

Low-level plotting functions add to an existing plot. For example `lines()`, `points()`, and `text()`.

Interactive plotting functions allow the user to interact with the plot. (Not important for exam???)

Device Control functions control the plotting device. this can be used to export plots to diffrent formats.

Control functions control diffrent aspects of the plot. For example `par()` can be used to set graphical parameters.

6.1 Scatterplots

Can be made with the syntax

```
> plot(x, y, xlab="X-axis label", ylab="Y-axis label",
↪  main="Title")
```

where `x` and `y` are numeric vectors of the same length.

The argument `asp` can be used to set the aspect ratio of the plot. Typically `asp=1` is used to get equal scaling on both axes.

To print data against its index, use

```
> plot(x[,"var"], ylab="var")
```

If the data passed to `plot()` is a matrix or data frame with two columns, the first column is used for the x-coordinates and the second for the y-coordinates automatically.

There also exists the formula interface. It is used by the RESPONSE VALUE and EXPLANATORY VALUE syntax. For example

```
> plot(y ~ x, data=df)
```

where `y` and `x` are column names in the data frame `df`.

The formula notation can also be used to do stuff such as `y ~ x1 + x2` In this case, a scatterplot matrix is produced.

Using the parameters `xlim` and `ylim`, the limits of the axes can be set. For example

```
> plot(x, y, xlim=c(0, 10), ylim=c(-5, 5))
```

When data covers multiple diffrent orders of magnitude, a log scale can be used. This is done using the argument `log` which can be set to `"x"`, `"y"`, or `"xy"`. The same can be done using `plot(log(y) ~ log(x), data=df)`.

The first variant is prefered as the axes are still labeled with the original values and not the log values.

Other characteristics include pch (plotting character), col (color), and cex (character expansion / size of characters).

If `pch` is a letter, it is plotted as that letter. If it is a number, it is plotted as a symbol. The numbers 1 to 25 are diffrent symbols.

The type of a plot can be set using the argument `type` which has arguments "p" (points), "l" (lines), "b" (both points and lines), "n" (empty plot)

In general, it is not nessecairy to have constants in the code. For example

```
> plot(x, y, col=z, cex=sqrt(w))
```

where `z` and `w` are vectors of the same length as `x` and `y`.

6.2 Boxplots

The syntax for boxplots is

```
> boxplot(y ~ x, data=df, notch=FALSE,
↪  horizontal=FALSE,)
```

where `y` is a numeric vector and `x` is a factor.

The notch is a useful feature to compare groups. If the notches of two boxes do not overlap, this is strong evidence that the two medians differ. This is a very quick visual test, but it is not very precise.

7 Adding Elements to and controlling appearance of Graphics

7.1 Adding Elements

To add another set of data points to an existing plot, use the `points()` function. The arguments are the same as for `plot()` and it is called after the high-level plotting function.

However it is important to ensure that the new data fits within the existing plot region, as this cannot be changed after the plot is created. This can be done using something like `ylim=range(c(y1, y2))` in the plot function.

Similar function exists with `lines()` to add lines to an existing plot. Arguments here are `lty` (line type) and `lwd` (line width).

To add straight lines going through the plot, `abline()` can be used. If the argument `h` is used, horizontal lines are drawn at the specified y-values. Similarly, `v` draws vertical lines at the specified x-values.

If one wants to add a linear line, the two arguments `a` (intercept) and `b` (slope) can be used.

Text can be added to a plot using the `text()` function. Data can for example be annotated by using

```
> text(y ~ x, data=df, labels=labelcol, pos=4, cex=0.7)
```

A legend can be added using the `legend()` function. The first argument is something like `"topright"` to specify the position. The argument `legend` is a character vector with the labels. Other arguments include `pch`, `col`, and `title`.

The argument `bty` can be used to specify the type of box around the legend. Setting it to `"n"` removes the box.

7.2 The par() Function

The parameter function `par()` can be used to extract and set settings for all the plots that follow until `par()` is called again.

For example, `par("pch")` returns the current plotting character. To set a new plotting character, use `par(pch=2)`. The first can also be called with a vector of parameters to get multiple values.

To place several plots in one big plot, use the argument `mfrow` or `mfcol`. They both take a vector of two values specifying the number of rows and columns. The difference is that `mfrow` fills the plots row-wise, while `mfcol` fills them column-wise.

As a good coding practice, it is recommended to save the old parameters before changing them and then restore them afterwards. For example

```
> oldpar <- par(mfrow=c(2,2))
> # plotting code
> par(oldpar)
```

To add colors to text, `col.main` or similar arguments can be used in the high-level plotting functions or par. Similarly, `cex.main` can be used to change the size of the title text.

7.3 Colors

The function `palette()` can be used to see the current color palette. To set a new palette, use the function

`palette(c("red", "green", "blue"))`. If needed, hex codes can also be passed to the function.

Preferably functions such as rainbow() and heat.colors() should be used to generate color palettes. The argument here is the number of colors to generate.

```
> palette(rainbow(5))
```