R basics

Florian D. Schneider

June 17, 2015

Introduction

what is R?

- a script-based software (vs. graphical user interface software)
- a functional programming language, i.e. it is build around functions
- a command line tool
- a basic software, can be extended by thousands of packages
- it can handle, analyse and plot data
- widespread in Ecology and Evolution
- open source, free as in free speech

Install R

R can be downloaded on CRAN for any platform and operating system. There are integrated development environments (IDEs) that help you to write your entire code projects, like RStudio or emacs. However, for this introduction, the basic R software is sufficient. Additionally, an advanced text editor that highlights code structure is recommended (e.g. Atom, Notepad++).

The R Console

Once R is installed, just launch it! You will see a terminal window, which is your interface to the software:

```
R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

```
>
```

This interface replies to your commands. For instance, basic math can be executed directly by typing a prompt:

```
> 2+4
[1] 6
> 23*1.5
> sqrt((1+34*2.23)/32.3^2)
[1] 0.271353
```

Objects

Everything in R is an object.

As a programming language, R can handle variables. They are assigned using the `<-` arrow:

```
> x <- 12
> x
[1] 12
```

Of course they can be used for any kind of calculation.

```
> x * 2
[1] 24
```

There are many types of variables that come with different features. Just a few examples:

```
x <- 432/152
n <- "A"
v <- c(21,32,3,42,53,43,64,23,34)
s <- "Carcinus maenas"
f <- as.factor(c("A", "A", "B", "C", "A", "B", "B", "C", "C"))
m <- matrix(sample(1:25,25), nrow = 5)
d <- data.frame(x = f, y = c(1,1,1.4,1.2,1.3,1.2,1.4,1.2,1.1))
l <- list(x,n,v,s,f,d,m)
b <- TRUE
```

In general, variables are 'objects'. Thus, R can store single numerical or character values, it can store vectors of values (using `c()`), and more complex data structures like homogeneous data matrices (`matrix()`)[1] or heterogeneous data tables (`data.frame()`). The most complex object is a list (`list()`), which can contain multiple objects of different class.

[1] There is a multidimensional version of matrix, called an array (`array()`).

object class	dimensions	function for creation	type of data
single value	no dimensions	e.g. `2.32`, `"a"`, `FALSE`	any type (numerical, integer, factor, binary, character)
vector	one dimensional	`c()`	a vector of single values of the same type
dataframe	two-dimensional	`data.frame()`	contains multiple vectors of same length
matrix	two-dimensional	`matrix()`	contains values of same type in rows and columns
array	n-dimensional	`array()`	contains values of same type in n-dimensions
list	undimensional	`list()`	contains entries of different types of objects

Functions

> everything you do in R is done by a function!

Functions are the core concept of any programming language. You have a given set of functions that take a particular *input* variable and return a defined *output* variable. The base package already provides many fundamental ones for doing maths and plotting. The standard format of a function is

```
function(x, ... )
```

where `x` is the first argument of the function, it's *input* object[2], and `...` can be a variable number of function arguments that specify the function's behaviour. Examples:

```
mean(v)       # returns the mean of the values
```

```
## [1] 35
```

```
runif(12)     # returns random numbers from uniform distribution between 0 and 1
```

```
##  [1] 0.5412287 0.5757853 0.5956550 0.7317122 0.9278485 0.5192816 0.2378902
##  [8] 0.3431128 0.4355487 0.7190987 0.6808003 0.5766829
```

```
sample(1:100, 12)   # samples 12 values from the vector given in the first argument
```

```
##  [1] 90 92 48 75 68 44 65 49 52  4 54 87
```

[2] some functions require a second input argument, or maybe even more

```r
rep(c("A","B","C"), each = 10)  # repeats each of the values in the vector ten times
```

```
##  [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "B" "B" "B" "B" "B" "B" "B"
## [18] "B" "B" "B" "C" "C" "C" "C" "C" "C" "C" "C" "C" "C" "C"
```

```r
cor(v, d$y, method = "pearson")   # returns pearson correlation of the two vectors
```

```
## [1] 0.2151171
```

```r
t.test(v, d$y)  # runs a t.test to compare the two vectors
```

```
##
##  Welch Two Sample t-test
##
## data:  v and d$y
## t = 5.5691, df = 8.0011, p-value = 0.0005288
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  19.80461 47.79539
## sample estimates:
## mean of x mean of y
##      35.0       1.2
```

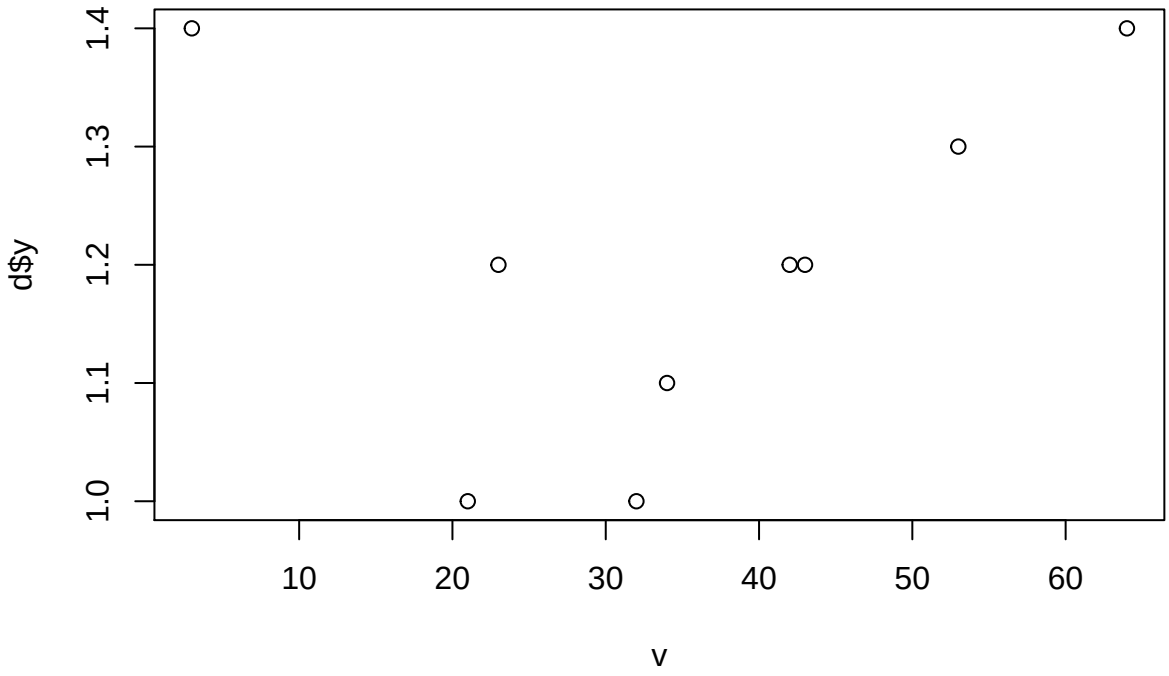
```r
# some functions have a very different syntax
1:5           # returns a sequence of integer numbers
```

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

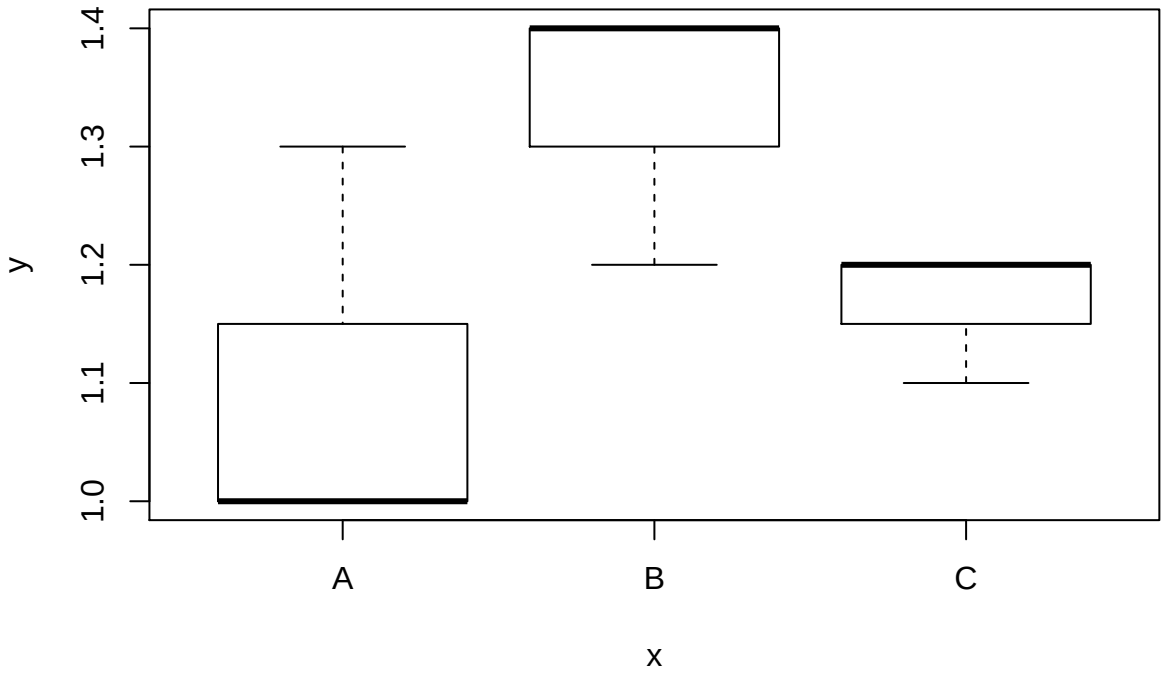
```r
2 + 2 == 5    # logical operators `==` and `!=` are functions
```

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

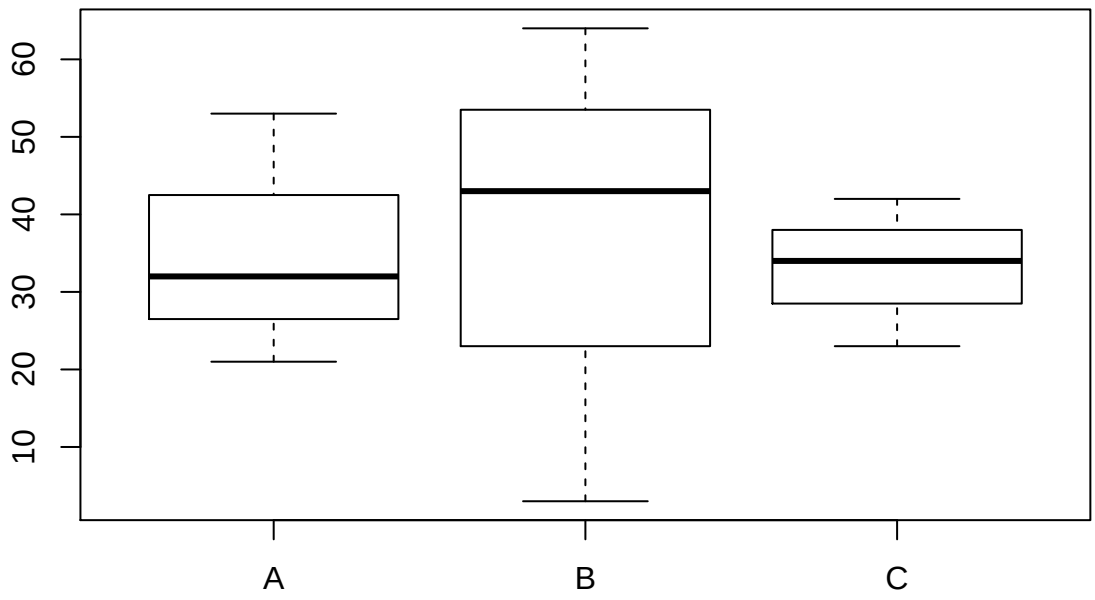
```r
# plot functions don't return an output to the console, but they open a plot window.
plot(v, d$y)
```




```
plot(d)
```




```
plot(f, v)
```



Wrap up:

Objects and Functions are the grammar of R. Imagine objects as nouns and functions as verbs, e.g.

- `average bodysizes`
- `plot productivity` against `temperature`
- `fit relationship` between `diversity` and `temperature`

By the way: if the function you require does not exist, just write it:

```
add_one_to <- function(x) x+1
a <- c(0.23,0.53,0.42,0.75,0.26)

add_one_to(a)
```

```
## [1] 1.23 1.53 1.42 1.75 1.26
```

EXERCISE 1

(footnotes contain hints)

1. objects

- [ ] create a vector `f` containing 5 factorial values
- [ ] create a vector `n` containing 640 numerical values
- [ ] create a matrix `m` of 20 rows and 32 columns filled with vector `n`
- [ ] create a dataframe with 3 columns, a unique ID, one factorial and one continuous column. Each column should have a name.

2. functions

- [ ] get the median of a numerical vector[3]
- [ ] round `34/343` to 2 digits[4]
- [ ] get highest value in your matrix from above[5]

Scripting

This already makes a quite powerful calculator. But you need to remember which variables were stored before and what functions you have available for calculations. That is why we write scripts.

```
rm(list = ls())  # clear all objects from workspace

d0 <- read.csv("rbasics/d0.csv")  # read in raw data

str(d0)  # view data structure

model <- lm(response ~ explanatory, d0) # fit linear model
summary(model)                # summarize model

par(mar = c(4,4,2,2), las = 1, bty = "n") # plotting parameters
plot(response ~ explanatory, d0, pch = 20)     # plot relationship
abline(model, col = "red")              # add linear model to plot
```

A script

- can read in and create data, transform and plot them
- can contain comments, labelled `#`
- can be written in any text editor, but some provide syntax highlighting

In principle, a script can be executed by saving it to a filename and calling it from the command line (i.e. outside R) using

```
Rscript filename.R
```

But most commonly scripts are run in parts, by copy-and-pasting line by line into the Console. The advanced Integrated Development Environments (IDEs) like RStudio, allow you to 'Run' the selected lines of code or to 'Source' the entire file on display by clicking on buttons.

Probably in most cases the structure of a statistical script is like this:

[3] Hint: try function `median()` or `summary()`
[4] Hint: try function `round()`
[5] Hint: try function `max()`

1. read in data
2. select/rearrange data
3. perform statistical test
4. plot a figure

We are going to walk through those tasks one by one in the following sections.

Save and read data

raw-data files

Copy the files `d0.csv`, `d1.csv` and `d2.csv` to a directory on your computer. The first thing that is important is a proper file structure. Just have a look at the files in Excell or Libre Calc!

X	exp1	exp2	cov1	resp
1	ae	A	5.817095	10.38
2	ae	A	1.908497	8.31
3	ae	A	3.345305	9.82
4	ae	A	4.651219	10.98
5	ae	A	2.547653	7.50
6	ae	B	3.517860	6.22

You will see that they have precisely one header row (not two rows, no empty rows!), that contains column names. The column names should

- be unique within the table
- be unambiguous and easy to remember
- use lower case *or* upper case letters
- not contain spaces or points or special characters except `_`

I recommend that you use english column names, that makes it easier to publish or share your data. For the same reason, make sure your cells are separated by commas and your decimal separator is a point , e.g. `0.4324`, which is conventional in english language and not a comma `0,4324`! Note that Microsoft Excel by default uses semi-colon `;` separators between columns and decimal commas depending if your system language is German.

A word about the data structure:

Usually you will structure your data in Excell in a way that is readable in it's raw form, e.g. by putting the outcome of different treatments into different columns. And of course, R can read in data tables of any structure. Any data structure can be transferred with a couple of lines of code into any form you like. But you can skip this if you decide on a raw data structure that is easy to handle in R. To achieve this, there are a couple of important rules of thumb:

- **one line = one data point**
- **save measured raw data**, no transformed or derived data
- **save in .csv or .txt format**, not .xxls

Don't

- merge cells

- put replicates in the same row
- put different treatments in the same row
- keep empty lines (missing data values are okay!)

This helps to keep your data files simple and unambiguous and forces you to use a clear structure.

read data

R can read data tables of many file formats but the most basic file format are pure text files (.csv, .txt), where columns are separated by line breaks and rows are separated by a 'Separator', which is usually a comma (hence .csv = **c**omma **s**eparated **v**alue).

The command to read a csv file is

```
d1 <- read.csv("d1.csv")
```

This probably does not work right away. Since you refer to a file outside of R, you need to tell R which place you are referring to by specifying the path to that file. There are two ways of doing this.

1. giving an absolute path for the file

```
d1 <- read.csv("C:/Windows/Users/florian.schneider/Documents/projects/stats/rbasics/d1.csv")
```

This can be very long, and will not work, if you switch to a different computer.

2. setting the working directory for R

```
setwd("C:/Users/flschneider/Documents/projects/R/r_basics")
d0 <- read.csv("d0.csv")
d1 <- read.csv("d1.csv")
d2 <- read.csv("d2.csv")
```

Now you can access the content of the file in R as an object, more specifically as a 'dataframe'. You can have a look at how the data are structured using the function `str()` or `head()`

```
str(d1)  # returns the type of data for each column of the table
```

```
## 'data.frame':    60 obs. of  5 variables:
##  $ X   : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ exp1: Factor w/ 3 levels "ae","ge","rw": 1 1 1 1 1 1 1 1 1 1 ...
##  $ exp2: Factor w/ 4 levels "A","B","C","D": 1 1 1 1 1 2 2 2 2 2 ...
##  $ cov1: num  5.82 1.91 3.35 4.65 2.55 ...
##  $ resp: num  10.38 8.31 9.82 10.98 7.5 ...
```

```
head(d1) # returns the first 6 rows of a dataframe by default
```

```
##   X exp1 exp2     cov1  resp
## 1 1   ae    A 5.817095 10.38
## 2 2   ae    A 1.908497  8.31
## 3 3   ae    A 3.345305  9.82
## 4 4   ae    A 4.651219 10.98
## 5 5   ae    A 2.547653  7.50
## 6 6   ae    B 3.517860  6.22
```

Select data

We want to work with the data from our object `d1`. There are several ways to access them.

- **attach** the dataframe which creates a vector for each column. This is very simple but usually considered bad practice, because it makes more objects available than you need and you don't have control over the object names created.

```
attach(d1)
head(exp1)
```

```
## [1] ae ae ae ae ae ae
## Levels: ae ge rw
```

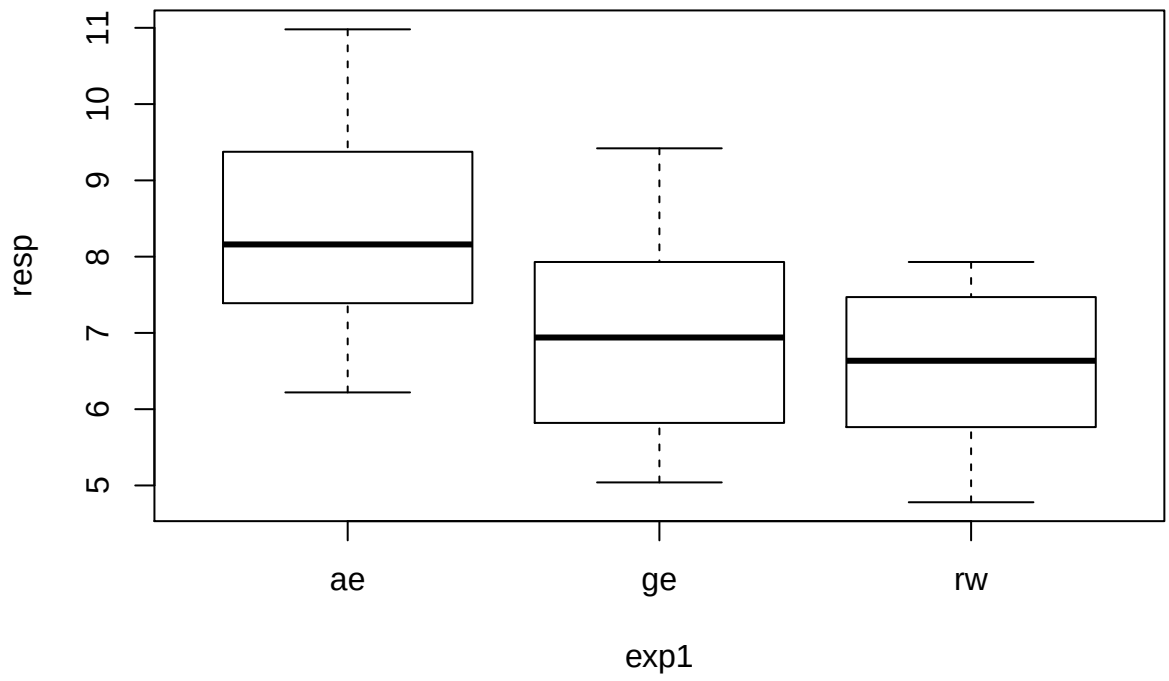
- **extract** the columns of interest by assigning them into a new object.

```
response <- d1$resp
explanatory1 <- d1[,"exp1"]  # an alternative way of extracting data
head(resp)
```

```
## [1] 10.38  8.31  9.82 10.98  7.50  6.22
```

- **refer** to the columns within the dataframe when calling functions.

```
plot(resp ~ exp1, data = d1)
```



Subsetting

Sometimes you are not interested in the entire dataset, but only a part of it, i.e. a subset of it's rows. There are again a couple of ways of selecting rows from a dataset,

1. use the `subset()` function

```
s1 <- subset(d1, exp1 == "ae")
s2 <- subset(d1, exp2 %in% c("A","B","D"))
```

2. use the squared brackets `[]`

```
s1 <- d1[d1$exp1 == "ae",]
s2 <- d1[d1$exp2 %in% c("A","B","D"), ]
s3 <- d2[d2$exp1 > median(d2$exp1), ]
```

You might have noticed that the squared brackets can be used for both, subsetting of rows *and* selection of columns. This is a very powerful tool, since you can use it both at the same time. This allows us to create very compact data extracts.

```
s1 <- d1[d1$exp2 %in% c("A","B","D"), c("exp1","resp")]
head(s1)
```

```
##   exp1  resp
## 1   ae 10.38
## 2   ae  8.31
## 3   ae  9.82
## 4   ae 10.98
## 5   ae  7.50
## 6   ae  6.22
```

```
s2 <- d1[d1$exp2 %in% c("C"), c("exp1","resp")]
head(s2)
```

```
##    exp1 resp
## 11   ae 7.49
## 12   ae 7.38
## 13   ae 8.26
## 14   ae 7.41
## 15   ae 9.60
## 31   ge 7.22
```

EXERCISE 2

- [ ] extract all data points from `d1` that have value "A" in `exp2` into an object `s`.
- [ ] extract all datapoints from `d2` that have response value larger than 100, and that have value "a" for `cof1`.
- [ ] paste another column to `d2` that contains random numbers[6]

---

[6] Hint: use function `cbind()`

Statistical modelling

The core feature and strength of R is statistical analysis of data. Basically every statistical model that has been developed is available in one of the many packages available for R. The choice of model depends on the structure of your data and the question you are asking. You should have a good knowledge of when and how to apply statistical methods before you apply them to your data. We are going to walk through the most basic analyses of ANOVA and ANCOVA as examples.

ANOVA

An Analysis of Variance tests if the means of the response variable for two or more subsets of the data (specified by factorial explanatory variables). The test compares the means by comparing the difference in variance between the data. Our dataset `d1` is well suited for this analysis.

The function to call an ANOVA is `aov()`

```
model <- aov(resp ~ exp1 , data = d1)
summary(model)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## exp1         2  33.54  16.769   10.57 0.000125 ***
## Residuals   57  90.45   1.587
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output of the model is returned by the function `summary()` and it tells us that the effect of `exp1` on our response is highly significant (with probability $p < 0.001$), which means that the different values of `exp1` do cause differences in the mean of `resp`. That is quite abstract, because we still do not know what means are to be expected for the different values of `exp1`.

A post-hoc test can clarify

```
TukeyHSD(model)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = resp ~ exp1, data = d1)
##
## $exp1
##          diff       lwr        upr     p adj
## ge-ae -1.3515 -2.310084 -0.3929159 0.0035663
## rw-ae -1.7460 -2.704584 -0.7874159 0.0001488
## rw-ge -0.3945 -1.353084  0.5640841 0.5859219
```

It tells us the pairwise difference between the three different levels of `exp1`. Some of them are significantly different, but not all.

linear models & ANCOVA

If the explanatory is not factorial, but continuous, we would do an Analysis of Co-Variance (ANCOVA), which is assuming that the explanatory variable x has a linear relationship with the response variable y.

The function to do this is, `lm()`, which fits a linear model using the least-squares method to descripe the relationship as a linear equation:

$$y = a + bx$$

```
model <- lm(resp ~ exp1 , data = d2)
summary(model)
```

```
##
## Call:
## lm(formula = resp ~ exp1, data = d2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -58.042 -12.897   1.609  13.104  55.288
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   67.914      2.047  33.170   <2e-16 ***
## exp1           2.155      1.004   2.146   0.0339 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22.11 on 118 degrees of freedom
## Multiple R-squared:  0.03757,    Adjusted R-squared:  0.02942
## F-statistic: 4.607 on 1 and 118 DF,  p-value: 0.03389
```

The summary looks different from the ANOVA. It reports the *intercept* a and *slope* b along `exp1` and tells us if those parameters are significant. The equation of the linear relationshipt would be

$$y = 67.9 + 2.155x$$

To rearrange the result into an ANCOVA we can call

```
anova(model)
```

```
## Analysis of Variance Table
##
## Response: resp
##            Df Sum Sq Mean Sq F value  Pr(>F)
## exp1        1   2252 2252.35  4.6069 0.03389 *
## Residuals 118  57691  488.91
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

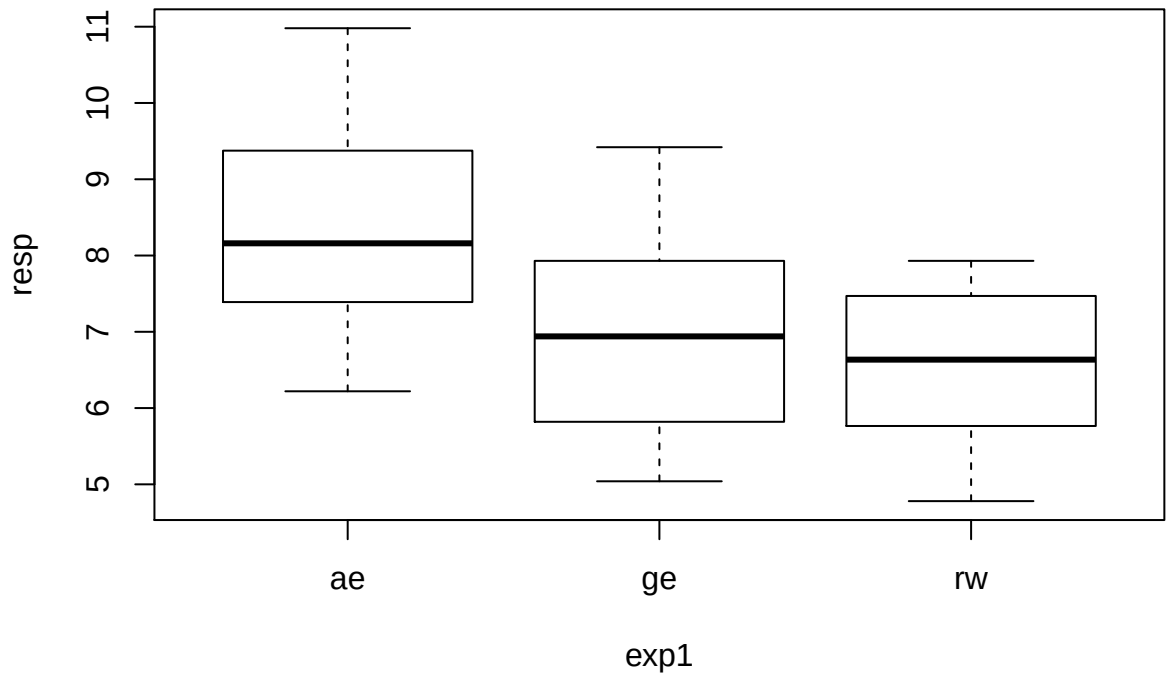
To conclude, the continuous variable `exp1` has a significant positive effect on the response variable `resp`.

Such statistical results usually are visualized by a plot. R has convenient predefined plotting functions.

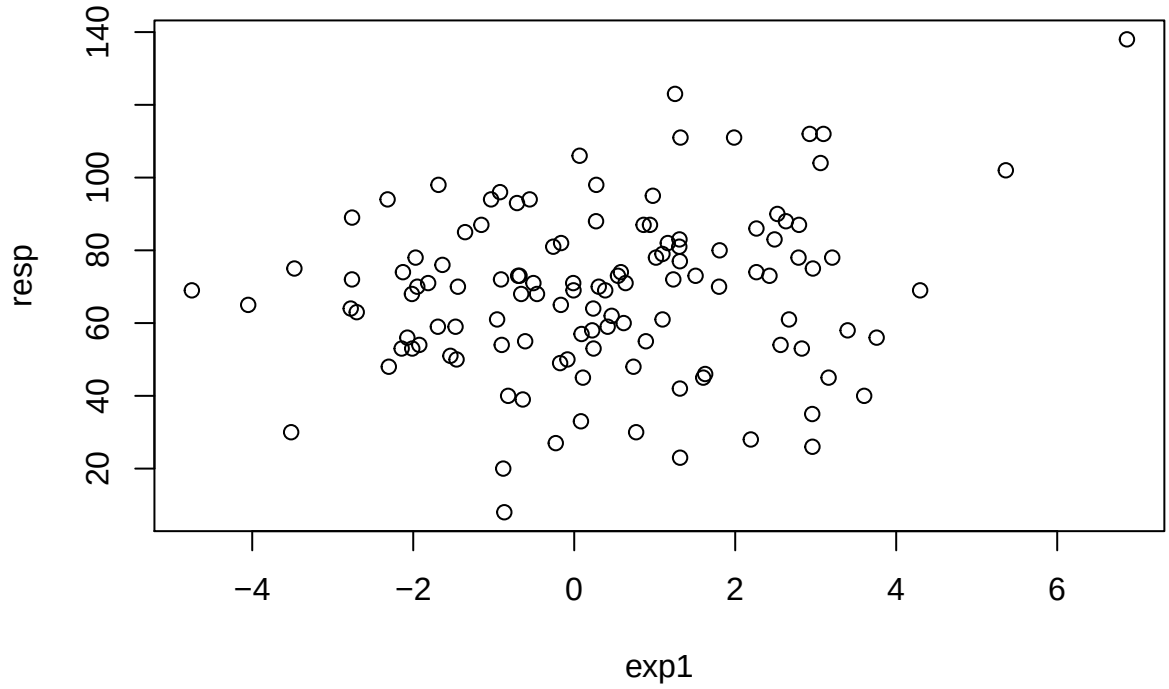
Plots

In R the `plot()` function selects a default plot, depending on the type of data.

```
plot(resp ~ exp1, d1) # with a factorial variable 'resp'
```

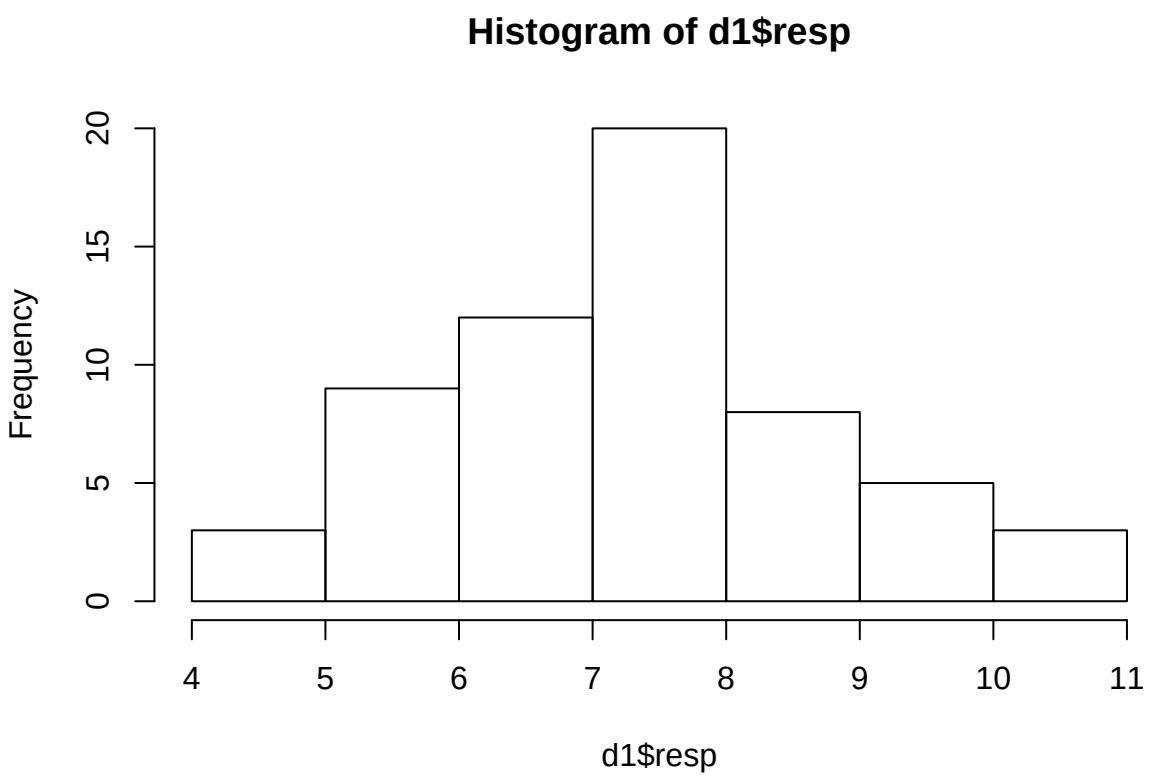



```
plot(resp ~ exp1, d2) # with a continuous variable 'resp'
```

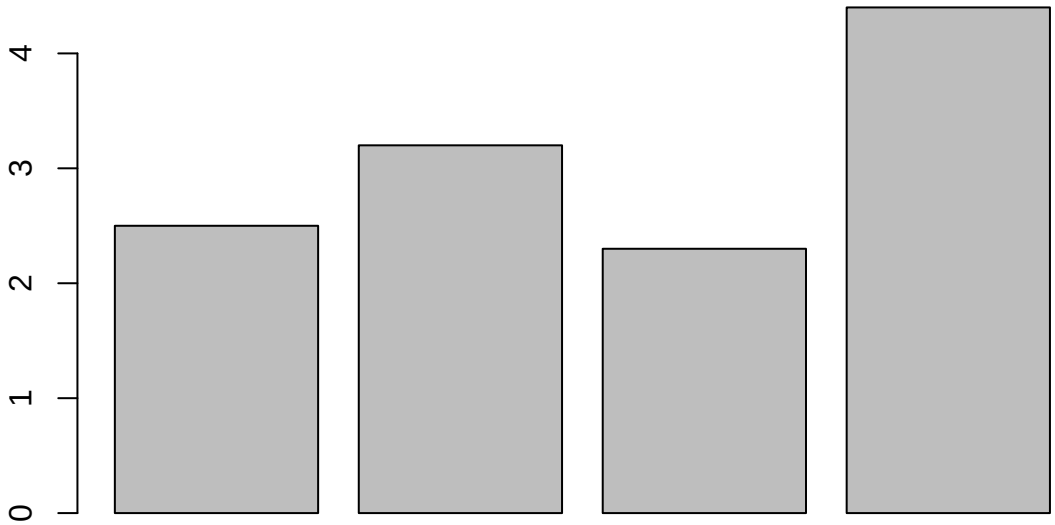


Thus, for factorial explanatories, R choses the *boxplot*, while for continuous variables it choses the *scatterplot*. Other basic plotting functions are available:

```
hist(d1$resp)
```




```
barplot(c(2.5,3.2,2.3,4.4) )
```



Types of plots:

type	input	function
scatterplot	a continuous explanatory & a continuous response	`plot(y~x)`
boxplot	a factorial explanatory & a continuous response	`plot(y~x)`, `boxplot(y~x)`
histogram	a continuous vector	`hist(x)`
barplot	a continuous vector or a matrix	`barplot(x)`

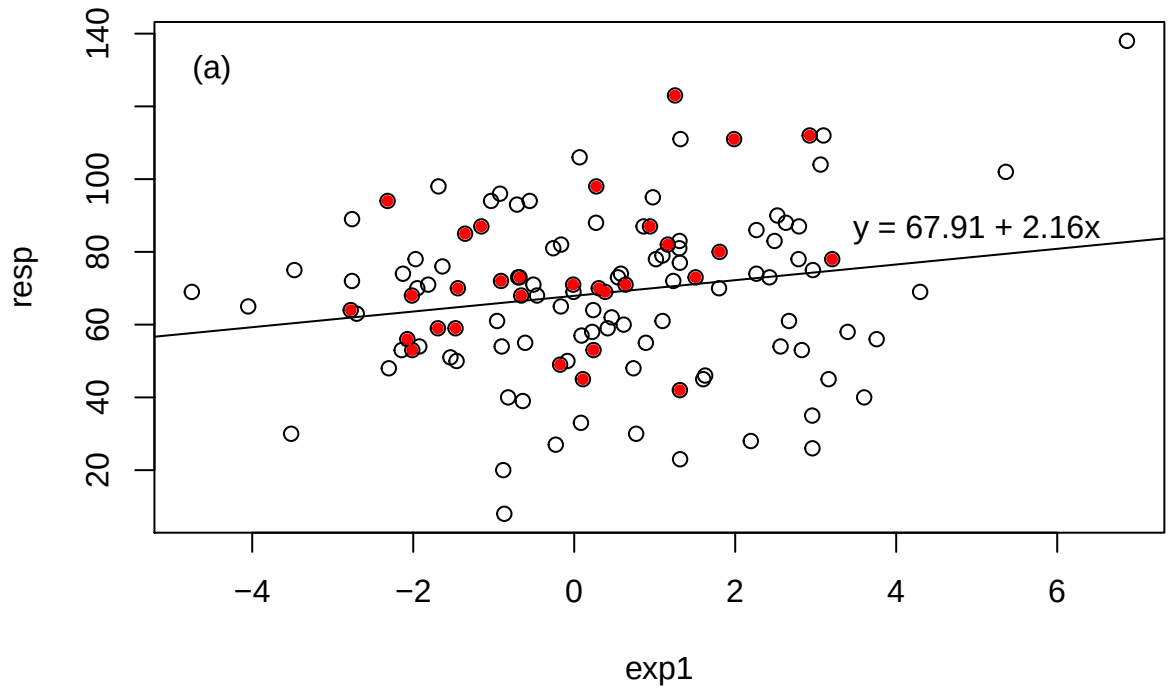
high-level vs. low-level plotting commands

All those functions are so called 'high-level' plotting commands, *i.e.* they create a new plot into the current plotting device[7]. Those can be completed by adding 'low-level' plotting commands, e.g.

- `abline()` to add straight lines
- `lines()` to add curves
- `points()` to add or highlight certain data points
- `arrows()` to add arrows
- `text()` to add text

Example:

```
plot(resp ~ exp1, d2)  # creates the plot
model <- lm(resp ~ exp1, d2)
abline(model) # adds the linear model fit
text(-4.5, 130, "(a)") # adds a label for the panel
text(5,86,
     paste0("y = ", round(model$coefficients[1], 2),
         " + ", round(model$coefficients[2], 2), "x" )
     )  # adds the model equation
points(resp ~ exp1,
       data = subset(d2, cof1 == "a"),
       col = "red", pch = 20) # highlights a part of the data
```

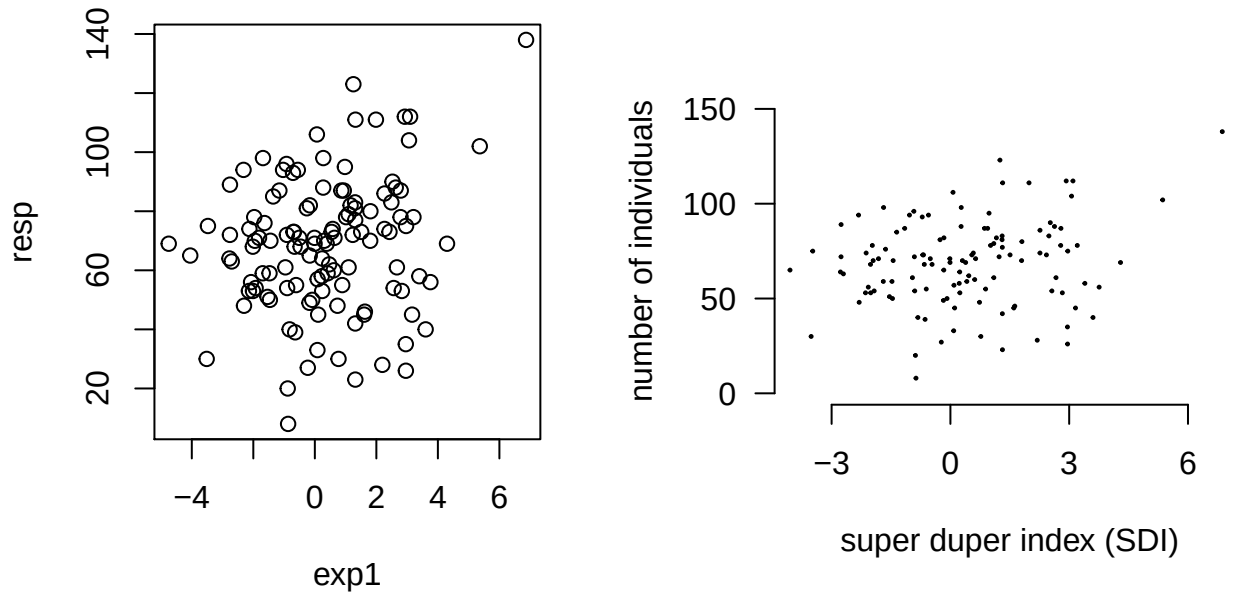


[7] A plotting 'device' is the window or screen or file where the plot is created.

plotting options

To modify the default plotting style or add and remove elements from the plot, you can use plotting options. They can be specified within the high-level plotting command (as an argument) or before using the function `par()`.

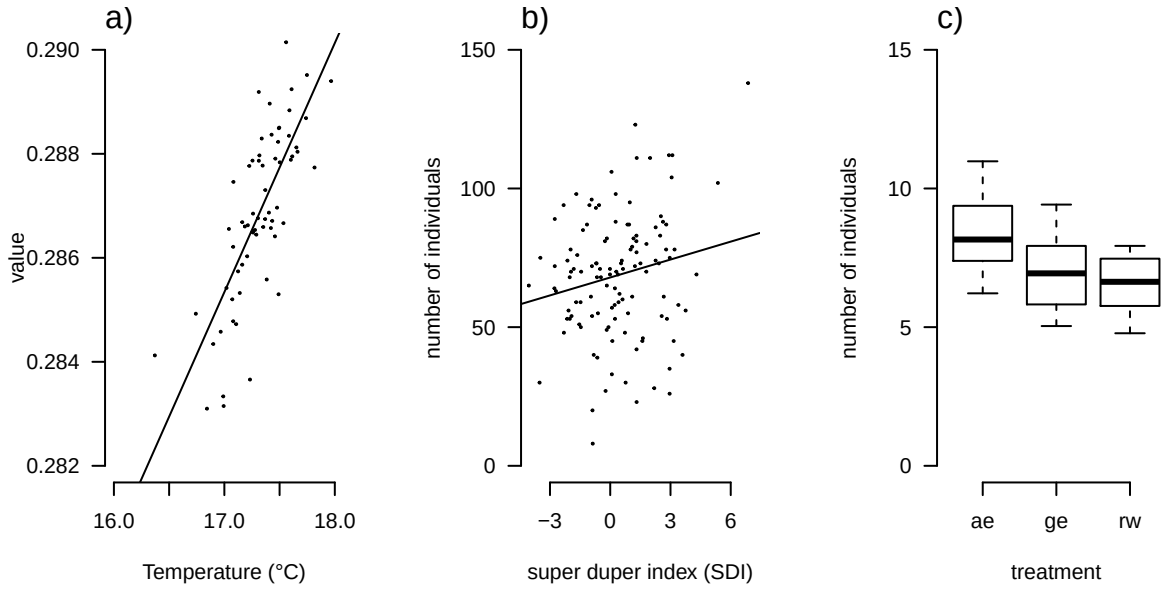
```
par(mfrow = c(1,2)) # creates a plot with two panels
plot(resp ~ exp1, d2)
par(bty = "n",         # disable box around plot
    mar = c(6,4,6,0)  # set plot margins
    )
plot(resp ~ exp1, d2,
     las = 1,                  # makes axis labels horizontal
     ylim = c(0, 150), xlim = c(-4,7),  # sets axis limits
     ylab = "number of individuals",
       xlab = "super duper index (SDI)",  # set axis labels
     yaxp = c(0,150,3), xaxp = c(-3,6,3), # sets axis marks
     pch = 20, cex = 0.3     # sets point style and size
     )
```



Thus, any plot can be generated by a combination of **options**, **high-level** and **low-level** plotting commands. Typically, a small script says how the plot would look like this:

```
par(mfrow = c(1,3), las = 1, mar = c(4,6,1,1), las = 1, bty = "n")

plot(response ~ explanatory, d0,
      xlim = c(16,18), ylim = c(0.282, 0.29), # sets axis limits
     ylab = "value",
       xlab = "Temperature (°C)",  # set axis labels
```

```
     pch = 20, cex = 0.3      # sets point style and size
     )

mtext("a)",  adj = 0)
abline(lm(response ~ explanatory, d0))

plot(resp ~ exp1, d2,
     ylim = c(0, 150), xlim = c(-4,7),  # sets axis limits
     ylab = "number of individuals",
       xlab = "super duper index (SDI)",  # set axis labels
     yaxp = c(0,150,3), xaxp = c(-3,6,3), # sets axis marks
     pch = 20, cex = 0.3      # sets point style and size
     )
mtext("b)", adj = 0)
abline(lm(resp ~ exp1, d2))

plot(resp ~ exp1, d1,
     ylim = c(0, 15),
     xlab = "treatment", ylab = "number of individuals"
     )
mtext("c)",  adj = 0)
```



save the plot to a file

Plots can easily be saved to a file. This is achieved by opening a new plotting device, before calling the plot and closing it afterwards

```
png("d0.png", width = 900, height = 600, res = 144)
plot(response ~ explanatory, d0)
dev.off()
```

```
## pdf
##   2
```

Many other raster graphics file formats are available (`bmp()`, `jpg()`, `tiff()`) but keep in mind that figures created by R usually are vector graphics composed from lines, points and polygons. The ideal file format for those are postscript based formats (.pdf or .eps) because they save the output without loss and can be scaled to any size without creating artifacts. The functions to save vector graphics are `postsript()` and `pdf()`:

```
pdf("d0.pdf", width = 9, height = 6)
plot(response ~ explanatory, d0)
dev.off()
```

```
## pdf
##   2
```

EXERCISE 3

- [ ] make two different boxplots of the subset of the datapoints from `d1` with value "A" or "B" in `exp2`, respectively.
- [ ] highlight points in `d2` with different symbols for each value in `cof1`.
- [ ] add a model line and equation to a plot of data `plot(explanatory~ response, d0)`

Packages

The amazing flexibility of R comes from it's more than 6000 packages that have been developed and tested by a huge community of researchers and code developers. The official repository for packages is called CRAN, which is mirrored on many servers around the world. Here you find descriptions and the official documentation for each package.

A package can be installed from within R by using the function (example 'ggplot2')

```
install.packages("ggplot2")
```

Before you can use it in your current R session, you have to attach it by calling

```
library("ggplot2")
```

This makes the packages functions and objects available.

Some interesting and widespread packages are

- 'ggplot2': An implementation of the Grammar of Graphics
- 'plyr': a package to organise, split, rearrange data
- 'nlme': non-linear mixed effect models
- 'Rcpp': Seamless R and C++ Integration
- 'knitr': A general-purpose package for dynamic report generation in R

how to go on?

If you want to learn more about R, the best way is to continue using it.

Swirl: learn R in R

Turns your R console into an interactive teacher. You can chose from several classes of different level. Swirl walks you through many examples.

```
install.packages("swirl")
library("swirl")
swirl()
```

Help function

Just type `?function` or `help(function)` if you need help for the function called `function()`. This prompts the usually very detailled R documentation that comes along with all packages.

If you are not sure about the exact name of a function you can search for keywords using `??keyword` which will suggest a list of documentation files that might match your problem.

Books, websites

There is a growing number of books that explain R programming for beginners or that use R code to illustrate statistics. Some are particularly targeting an ecological audience.

- Official Basics Manual
- Getting Started with R: An introduction for biologists by Andrew Beckerman and Owen Petchey
- The R Book by Michael J. Crawley
- Mixed Effects Models and Extensions in Ecology with R by Alain Zuur et al.
- Ask for help at StackOverflow.

Glossary

term	definition
console	The terminal interface of R where you type, or paste the code you want to have executed
script	A text file, usually with file ending .R, that contains R-code.
working directory	The path that R assumes it is working in. This is important if you load or save files. can be set by `setwd("path/to/workingdirectory/")`

term	definition
object	An R object stores any kind of data, like a single value, a vector, a dataframe, a matrix
function	A function handles one or several objects and returns an output. Arguments specify how the function processes the objects.
plot	A plot is a vector graphic output on a graphic device. The default device is a plot window on the screen.
package	A package is an extension to the basic R functions, that provides additional statistics, plots or data management features.