Extremely short introduction to R

Jean-Yves Sgro

Feb 20, 2018

Contents

1 Suggested ahead activities

The following is a very succinct introduction to aspects of the software `R` that we'll explore in lab exercises. If you want to learn more ahead of time before class you could do the following fun activity:

What	Description	link
Interactive **Try R** short course. 	There are 7 chapters rewarding you with a "badge" for each complete chapter. No software installation required, only a web browser.	`tryr.codeschool.com` 

2 Introduction to R

2.1 Learning Objectives

- Run `R`
- understand `R` *objects*
- understand objects data structure
- generate data
- learn basic plotting methods

Acknowledgments: loosely based on "R Tutorial" by Chi Yau[1]

3 Starting R

R can be accessed by double-clicking on the R icon, or within a Terminal by simply typing the letter `R` at the prompt.

TASK

Do one of the following:.

- Find the `R` icon in the `/Applications` directory
- Find `Terminal` in `/Applications/Utilities` (or use the top-right icon that looks like a magnifying glass - *Sportlight Search* and type the word `Terminal`.

If you are using the terminal type `R` at the `$` prompt:

```
R
```

A "splash screen" will be typed on the terminal. The welcome screen will list the current version being run and will await further commands after the `R` prompt ">"

```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)
```

[1] http://www.r-tutor.com/r-introduction

```
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.

>
```

At the bottom the `R` prompt `>` invites the user to type commands. At this point `R` can be used as a "calculator" by performing simple arithmetic functions.

TASK

Type within your R console:.

For example:

```
1 + 3
```

```
[1] 4
```

```
sqrt(2)
```

```
[1] 1.414214
```

```
pi
```

```
[1] 3.141593
```

Note: `[1]` at the beginning is the line number. In more complex situations each line would be numbered.

4 R objects and variable assignment

"In every computer language variables provide a means of accessing the data stored in memory. R does not provide direct access to the computer's memory but rather provides a number of specialized data structures we will refer to as objects. These objects are referred to through symbols or variables."[2]

Assigning alues to variables can be accomplished with the assignment operators "`=`" or "`<-`".

For data containing characters the assignment should be within quotes. We'll see later the different "types" of data available.

Note that "nothing happens" when the assignment occurs:

TASK

Try it as we go.

[2] https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

```
x = 1
w ="word"
```

However, simply typing the variable name at the prompt will print out its value:

```
x
```

```
[1] 1
```

```
w
```

```
[1] "word"
```

The name of an object must start with a letter (`A-Z` or `a-z`) but can include letters, digits (`0-9`), dots (`.`), and underscores (`_`). `R` **case sensitive** and discriminates between uppercase and lowercase letters in the names of the objects, so that `a` and `A` can name two distinct objects (even under Windows).

4.1 Data types

As we just saw, characters have to be placed within quotes. The following data types occur often with routine `R` calculations:

- Numeric
- Integer
- Complex
- Logical
- Character

5 Built-in functions

`R` functions are invoked by its name, then followed by parenthesis. Parenthesis contain mandatory or optional arguments to pass to the function. Parenthesis are always written even if they remain empty.

5.1 list: ls()

For example we can now *list* the R objects that we created above with the function **`ls()`**:

```
ls()
```

```
[1]  "w"      "x"
```

5.2 class()

We can verify the type, or class of these variables with the function **`class()`**

```
class(x)
```

```
[1] "numeric"
```

```
class(w)
```

```
[1] "character"
```

5.3 combine: c()

The combine function is essential in R.

For example the follwing three numeric values are combined into a vector.

```
c(1, 2, 3)
```

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

Since we did not assign to a variable the output is immediately printed.

Here is the same vector assigned to variable v

```
v <- c(1, 2, 3)
```

This time no out put is produced but the data is stored in memory and can be called again.

5.4 length()

It may be useful to know the length of an object:

```
length(v)
```

```
[1] 3
```

6 Getting help

R provides extensive documentation. Depending on the installation method or how you access R the results appear either in plain text within the R console, an HTML page etc.

For example, entering `?c` or `help(c)` at the prompt gives documentation of the function `c()` in R.

TASK | EXERCISE

Try a help command on at least one of the function we have seen.

> *Note*: in help, `...` often means that arguments can be passed along by other funtions

7 Vectorisation

R calculations are "vectorized" in the sense that any calculation can be applied to all elements of *e.g.* a vector. For example:

```
# multiply elements of vector v by 10:
v * 10
```

```
[1] 10 20 30
```

```
# divide elements of vector v by 2:
v / 2
```

```
[1] 0.5 1.0 1.5
```

Note that the character `#` can be used as a way to comment the text. This is useful when saving all the commands into a text script, so that the "future you" or anyone else trying your code should understand the intention of the calculation(s.)

8 More complex data

There exists other types of more complex data that `R` can handle, most of them can be tabular or multidimensional:

- Vector
- Matrix
- List
- Data Frame

8.1 Vector

We already created a one-dimensional vector `v` above containing numeric values. But vectors can also contain characters or logical data. However, all data in one vector has to be of the same nature.

```
# create a vector of character
vc <- c("a", "b","c")
```

8.2 Matrix

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. All elements have to be of the same nature, *e.g.* numeric or character.

The function **`matrix()`** can be used to create a new matrix object.

```
matrix(c(1,2,3,4,5,6), nrow=2)
```

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

However, some more information needs to be given, for example how many rows should the matrix have, this is done by the **`nrow=`** option. Obviously the number of elements given should be in the number of expected row by columns. The default values are **`nrow = 1`**, **`ncol = 1`** and the default filling method is by column since the default is **`byrow = FALSE`**.

TASK | EXERCISE

Try to change some of the defaults. For example change **`byrow = FALSE`** to **`byrow = TRUE`**.

Your results:

8.2.1 Combining vectors to create a matrix

Another way to create a matrix is combining vectors of the same length with the functions `cbind()` or `rbind()` to combine by colum or row.

TASK | EXERCISE

Try these commands on the vectors v and vc - for example:

```
# with v
cvv <- cbind(v,v)

rvv <- rbind(v,v)

cvvvc <- cbind(v,v,v)

# with character vector vc
vc2 <- cbind(vc,vc)

# with both v and vc
vc3 <- cbind(v,vc)
```

Your results:

What happened when using both v and vc (hint: class(), quotes)

---

9 Dataframes

Dataframes are a type of table that allows each column to be of a different variable type. For example one column can be characters and another column can be numbers.

We can construct a dataframe starting with vectors with the function **`data.frame()`**

```
# num: a vector if numbers
num <- c(2, 3, 5)

# let: a vector or letters
let <- c("aa", "bb", "cc")

# tf: a vector or logicals true or false
tf <- c(TRUE, FALSE, TRUE)
```

```
 # df is a data frame
df = data.frame(num, let, tf)
```

We can inquire about `df`: the class of the object, its dimensions, the name of the headers for the columns,

```
class(df)
```

```
[1] "data.frame"
```

```
dim(df)
```

```
[1] 3 3
```

```
names(df)
```

```
[1] "num" "let" "tf"
```

9.1 Dataframe manipulation

As just as simple demonstration we'll change the name of the rows.

For now the dataframe looks like this:

```
df
```

```
  num let    tf
1   2  aa  TRUE
2   3  bb FALSE
3   5  cc  TRUE
```

and if we ask the name of each row we get the current list:

```
rownames(df)
```

```
[1] "1" "2" "3"
```

In `R` things can change by reassigning new values, so we can indeed change the row names with the function**`rownames()` and giving new values. For example:

```
row.names(df) <- c("row1", "row2", "row3")

# print df
df
```

```
     num let    tf
row1   2  aa  TRUE
row2   3  bb FALSE
row3   5  cc  TRUE
```

In the same way we could change the column names:

```
colnames(df) <- c("numbers", "letters", "logical")
```

Note: functions `row.names` and `rownames` exist for rows, but only `colnames` exist for columns.

In this final version the data itself is not altered but we changed both the column and row names:

```
df
```

```
     numbers letters logical
row1       2      aa    TRUE
row2       3      bb   FALSE
row3       5      cc    TRUE
```

10 Generating data

There are many ways to generate data from within `R` as series of numbers, in sequence or as random numbers. This section is purposefully kept simple.

10.1 Regular sequences

The generation of numbers in sequence can be useful to create lists.

The following command will generate an object with 10 elements; a regular sequence of integers ranging from 1 to 10, saved wihtin variable `x` thanks to the operator `:`

```
x <- 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Various options can be used to alter the results, for example requesting 11 values, starting with **`3`** and ending at **`5`**.

```
seq(length=11, from=3, to=5)
```

```
 [1] 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
```

10.2 Repeat and sequence functions:

It may be useful to print a number multiple time. This can be done with the `rep()` function. For example:

```
rep(1,15)
```

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

The function `sequence()` creates a series of sequences of integers each ending by the numbers given as arguments (** separators added for clarity).

```
 sequence(2:5)
```

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

To understand this output it is useful to also remember that **`2:5`** means **2, 3, 4, 5** and that the function will apply to each of these digits in turn.

10.3 Levels: gl() and expand.grid()

These two functions are very useful for creating tables containing experimental data.

The function `gl()` generates "levels"series of "factors" or "categories" as values or labels. The following example will generate 4 each of 2 levels:

```
gl(2, 4, labels = c("Control", "Treat"))
```

```
[1] Control Control Control Control Treat   Treat   Treat   Treat
Levels: Control Treat
```

The function`expand.grid()` creates a `data frame` with *all possible combinations* of vectors or factors given as arguments.

This example

```
expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
```

```
  h   w    sex
1 60 100   Male
2 80 100   Male
3 60 300   Male
4 80 300   Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

Note: The arguments are rotated as a function of their position in the command

Try the following:

```
expand.grid(sex=c("Male", "Female"), h=c(60,80), w=c(100, 300))
```

How many lines is the table (not counting the header? (hint: row numbers)

The use of `seq()` can also be useful in this context. Try the following examples:

```
expand.grid(height = seq(3, 3, 5), weight = seq(100, 250, 50), sex = c("Male","Female"))
```

How many lines is the table (not counting the header? (hint: row numbers)

Add one more variable `treatment = c("control", "drug"))` and see how much the table explands:

```
expand.grid(height = seq(3, 3, 5), weight = seq(100, 250, 50), sex = c("Male","Female"))
```

How many lines is the table (not counting the header? (hint: row numbers)

Note: the function `dim()` can be applied directly as well, for example:

```
dim(expand.grid(sex=c("Male", "Female"), h=c(60,80), w=c(100, 300)))
```

10.4 Random numbers

Most of the statistical functions are available within `R` such as Gaussian (Normal), Poisson, Student t-test etc.

To generate random numbers, the function based on the Normal distribution we use the function `rnorm()` (`r` for random and `norm` for Normal.) The number of desired random numbers is given as argument.

Since these are random, *the answers are never the same!*

TASK

Perform the following command requesting a single random number a few times (*e.g.* 5 times) in a row:

```
rnorm(1)
```

Do you get the same result every time?

`[ ] Yes` `[ ] No`

To provide means of reproducible the function `set.seed()` can be used to obtain the same result every time. The `seed` is a number chosen by the author. Here is an example selecting three numbers.

```
set.seed(33); rnorm(3)
```

```
[1] -0.13592452 -0.04079697  1.01053901
```

```
set.seed(33); rnorm(3)
```

```
[1] -0.13592452 -0.04079697  1.01053901
```

```
set.seed(33); rnorm(3)
```

```
[1] -0.13592452 -0.04079697  1.01053901
```

However, changing the seed value will change the results:

```
set.seed(22); rnorm(3)
```

```
[1] -0.5121391  2.4851837  1.0078262
```

> *Important note*[3] "[these] Pseudo Random Number Generators because they are in fact *fully algorithmic*: given the same seed, you get the same sequence. And that is a *feature* and not a bug."

One `R` method for choosing letters at random is with the function `sample()`. The term `LETTERS` represents the alphabet and is built-in.

```
sample(LETTERS, 5)
```

```
[1] "P" "S" "K" "I" "V"
```

```
sample(LETTERS, 5)
```

```
[1] "P" "L" "U" "K" "B"
```

In the same way as before seeting a seed will reproduce the same result every time.

```
set.seed(42); sample(LETTERS, 5)
```

```
[1] "X" "Z" "G" "T" "O"
```

```
set.seed(42); sample(LETTERS, 5)
```

```
[1] "X" "Z" "G" "T" "O"
```

11 Simple graphics with plot()

We will create a very simple graphic output from generated random numbers:

Create a data vector of 100 random numbers (note: if you choose the same seed number your final plot will be identical.)

```
set.seed(9)
data <- rnorm(100)
```

The plot() function will create a simple scatter plot with circles as the default symbol.

```
plot(data)
```

[3] https://stackoverflow.com/questions/13605271/reasons-for-using-the-set-seed-function

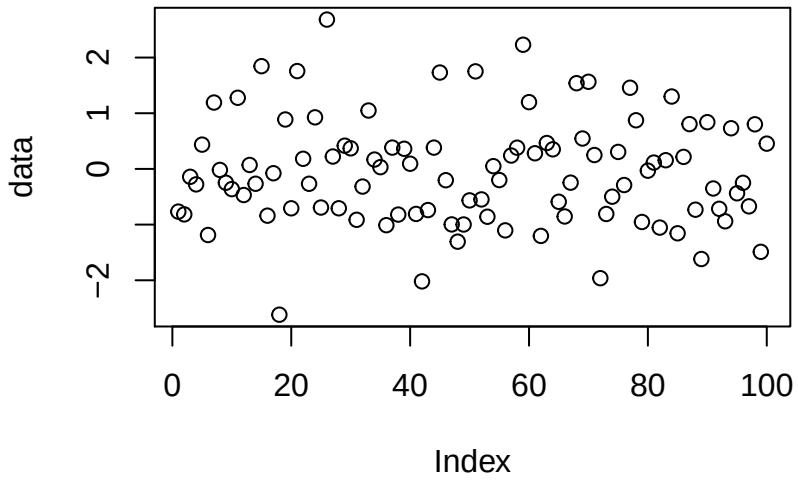

Figure 1: *default two-dimensional plot.*

It is possible to include more than one plot on the same figure/page with the parameter function modifying the number of rows and columns planned for plotting: `par(mfrow=c(1,1))` by default.

As a brief example we'll replot these data points as points, lines, both, and overlay. The labels for the axes are rendered blank to make the final layout less cluttered.

```
par(mfrow = c(2,2))

plot(data, type = "p", main = "points", ylab = "", xlab = "")
plot(data, type = "l", main = "lines", ylab = "", xlab = "")
plot(data, type = "b", main = "both", ylab = "", xlab = "")
plot(data, type = "o", main = "both overplot", ylab = "", xlab = "")
```

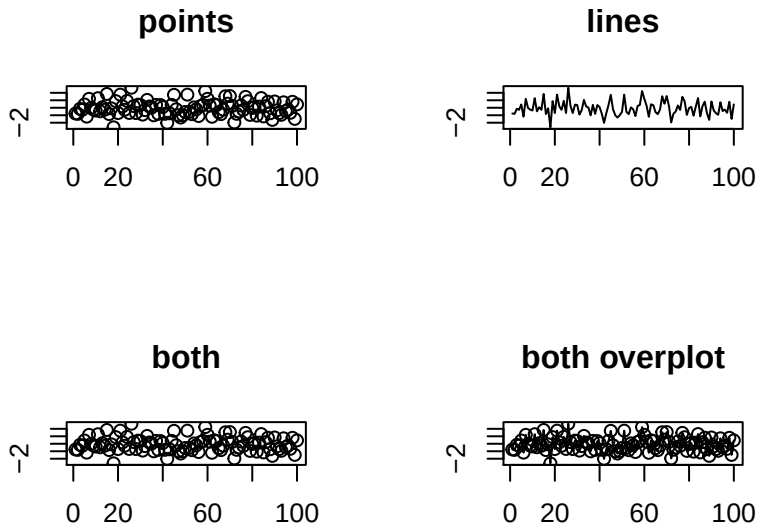

Figure 2: *split screen plots.*

It is useful to reset the number of images or plots to 1:

```
par(mfrow = c(1,1))
```