R Training

Lesson 1

Stefanie Molin

May 12, 2017

I. Introduction to R

Welcome to R training! R is a language and environment for statistical computing and graphics. The main goals of this training are to enable you to write your own scripts/analysis in R taking advantage of some very powerful R packages, but first things first, let's set up everyone's computers for R. We are going to be using RStudio as our IDE for R. You'll see why shortly!

Each lesson will be followed by 3 exercises; be sure to save your solutions as scripts–we will use them later.

A. Installing R and RStudio

- You can install the latest version of R from CRAN here: https://cran.r-project.org/bin/windows/base/
- RStudio can be installed from the RStudio website here: https://www.rstudio.com/products/rstudio/download/

B. Using R Studio

Let's get to know RStudio, your new best friend!

- RStudio has 4 panes: script editor, environment, console, and files/plots/help viewer (the panes can be rearranged to your liking). These make programming in RStudio much easier than using the R GUI that comes with R.
- RStudio provides helpful cheatsheets for common packages and RStudio itself. They are available from `Help -> Cheatsheets`.
- RStudio cheatsheet: https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf.
- When you aren't sure about a function or topic in R, you can get help without leaving RStudio! For help on a function you know the name of use: `?<function_name>`. To search for a topic use: `??<topic_to_search>`.
- RStudio also has R Projects which make working with others on a common repository easy. Each user can create an R project and you no longer have to worry about changing your working directories in the script, or someone else's changes breaking your version. You will all always have the same files, working directory, and directory structure.
- This can even be done with a git project in which case you will be able to switch branches, commit changes, pull and push all from RStudio.

C. Data types and structures

As with any programming language, R has different data types that you will need to be familiar with.

Data types

- **integer**: 5L (the "L" tells R to store this as an integer instead of a floating point value)
- **numeric**: 5.0

- **characters**: "a" or "hello"
- **factors**: these are categorical variables that R saves internally as integers if you don't say otherwise; this can often cause trouble unless you are modeling, so, for the most part, we are not going to use them in this training
- **logicals**: `TRUE` or `FALSE`

Structures (storing multiple elements)

We are going to ignore matrices since you won't use them much, if at all

vectors: (must be of a single type)

```
1:5 # numeric vector
```

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

```
c("a", "b", "c") # character vector
```

```
## [1] "a" "b" "c"
```

lists: (can mix types and be nested)

```
list(first = 1, second = "x", third = list(a = 1, b = 2))
```

```
## $first
## [1] 1
##
## $second
## [1] "x"
##
## $third
## $third$a
## [1] 1
##
## $third$b
## [1] 2
```

dataframes: (can be mixed types)

```
data.frame(col_1 = 1:5, col_2 = "e")
```

```
##   col_1 col_2
## 1     1     e
## 2     2     e
## 3     3     e
## 4     4     e
## 5     5     e
```

D. Variable assignment

Great so now that we know all about data types and structures, we can assign our first variable. There are 2 ways to do this in R: with "=" or with "<-". The arrow is the preferred method, so we will use that, however, there is no difference.

```
# assign the integer 5 to x
x <- 5L

# assign the numeric 5 to x
x <- 5

# assign the character string "Hello World" to y
y <- "Hello World"

# assign the logical TRUE to rIsFun
rIsFun <- TRUE

# assign the numbers 1 to 5 to z
z <- 1:5

# assign the letters a and z to vector z
z <- c("a", "z")
```

etc. You get the idea!

E. Working with dataframes

Dataframes will be the class of objects you work with most in R. You can think of them as Excel sheets. You have rows and columns and each column is of a certain data type, but this can vary from column to column. When you first get a dataframe, you should explore the data to understand what you are working with.

The following functions will help you with this:

Use `class()` or `typeof()` to see what type of object you are working with

```
class(cars)
```

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

`str()` shows the structure of the dataframe including the type of each column and some observations

```
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

`summary()` will provide you with summary of each column of the dataframe

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

`names()`/`colnames()` returns the names of each column of the dataframe

```
colnames(cars)
```

```
## [1] "speed" "dist"
```

`dim()` shows the rows and columns in your dataframe

```
dim(cars)
```

```
## [1] 50  2
```

`nrow()` and `ncol()` return the number of rows and columns respectively

```
nrow(cars)
```

```
## [1] 50
```

```
ncol(cars)
```

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

To inspect the first/last few rows of the dataframe use `head()`/`tail()`. The default is to return 6 rows.

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
tail(cars)
```

```
##    speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

The `$` operator can be used to select columns of a dataframe by name

```
cars$speed
```

You can also select specific rows and columns with `[row_index, col_index]` (note indices start at 1 and any range you specify is inclusive of both the start- and endpoints)

```
# select the 3rd row, 1st column
cars[3, 1]

# select all but the first column
cars[, -1]

# select the 3rd and 4th rows of the "speed" column
cars[3:4, "speed"]
```

F. Control Statements

A control statement is a `TRUE`/`FALSE` test that determines whether other sections of your code will be executed. They control the flow of your code by allowing for conditional execution and loops.

Logical Operators

- Greater than/less than (or equal to): `>=`, `<=`, `>`, `<`
- Equality comparision: `==`
- Not Equal: `!=`
- AND: `&`
- OR: `|`
- NOT (negation): `!`
- IN: `%in%`

Note: AND and OR above return vectors, however they can also be written as `&&` *or* `||` *in which case they will evaluate from left to right stopping when they have an answer, returning a single value.*

Conditional Statements

if/**else** – run the code *if* the condition is `TRUE`, *else* run some other code (else is optional)

```
# let's have R tell us if it is morning, afternoon, evening, or night

hour <- as.numeric(format(Sys.time(), format = "%H"))

if(hour >= 5 & hour <= 11){
  print("It is morning")
} else if(hour >= 12 & hour <= 17){
    print("It is the afternoon")
} else if(hour >= 18 & hour <= 21){
    print("It is evening")
} else {
    print("It is night")
}
```

```
## [1] "It is morning"
```

Loops

while loop – run *while* a condition is `TRUE`

```
# Let's tell the user when they are done working.

# initialize variables
isDoneWorking <- FALSE
hoursWorked <- 0

while(!isDoneWorking){
  # increment hoursWorked
  hoursWorked <- hoursWorked + 1

  # tell user how long they have worked
  print(paste("You have worked", hoursWorked, "hours."))

  # check if the user is done working
  if(hoursWorked == 8){
    isDoneWorking <- TRUE
  }
}
```

```
## [1] "You have worked 1 hours."
## [1] "You have worked 2 hours."
## [1] "You have worked 3 hours."
## [1] "You have worked 4 hours."
## [1] "You have worked 5 hours."
## [1] "You have worked 6 hours."
## [1] "You have worked 7 hours."
## [1] "You have worked 8 hours."
```

```
print("You are done working.")
```

```
## [1] "You are done working."
```

Note that `while` *loops can be extremely dangerous, since they don't automatically increment a counter; you must remember to do this in the loop, otherwise, you will have an infinite loop which will run until your computer runs out of memory and/or R crashes.*

for loop – run the loop *for each* of the elements in the input. There are 2 methods for `for` loops: you can iterate using an index or an iterable object. Which one you should use depends on the implementation.

```
# define our dataframe of names and ages
df <- data.frame(name = c("Alice", "Bob"), age = c(24, 27), stringsAsFactors = FALSE)

# let's iterate over name and print it
for(person in df$name){
  print(person)
}
```

```
## [1] "Alice"
## [1] "Bob"
```

```
# let's iterate by row (index) and print the name
for(i in 1:nrow(df)){
  print(df[i, "name"])
}
```

```
## [1] "Alice"
## [1] "Bob"
```

If you find yourself writing a `while` loop to iterate over all members of a list, you should rewrite it to a `for` loop.

Loop Control Statements

As with other languages, R has a few reserved words to modify the path through a loop.

- `break`: break out of the loop and start from the first statement outside the inner-most loop (if any)
- `next`: move on to the next iteration

G. Writing Functions

Functions are reusable pieces of code that are the building blocks of R.

functions with no arguments

```
# create a function to always print "Hello World!"

helloWorld <- function(){
  print("Hello World!")
}
```

functions with 1 argument (argument will default to "World")

```
# write a function that takes an argument "name" and prints "Hello [name]!"
# but will print "Hello World!" if only that is provided.

hello <- function(name = "World"){
  print(paste0("Hello ", name, "!"))
}
```

H. Calling Functions

We defined 2 functions above, now it's time to call them.

```
# call function with no arguments
helloWorld()
```

```
## [1] "Hello World!"
```

```
# call function with arguments in proper order
hello("Stef")
```

```
## [1] "Hello Stef!"
```

```
# call function with named arguments
# (required if you don't follow the order of arguments in the function definition)
hello(name = "Stef")
```

```
## [1] "Hello Stef!"
```

```
# notice what happens if we don't override the default value for name
hello()
```

```
## [1] "Hello World!"
```

I. Installing and Loading Packages

Base R is great, but there are some extremely useful functions that come from other packages. It is crucial that you know how to install a new package and load it so that you can access it's functions in your scripts. Let's install the packages we are going to use later in the course.

Installing packages (note that you need to load a package in your R session to use it)

```
# install RJDBC package
install.packages("RJDBC")

# install the remaining packages we will use in the trainings
install.packages(c("ggplot2", "dplyr", "data.table", "stringr", "reshape2",
                   "rmarkdown", "knitr"))
```

Loading packages – only do this when you are going to use their functions (you will have to repeat this each new R session)

```
# load dplyr
library(dplyr)
```

Note: In order to use functions that are saved in personal scripts, you will need to source them before using them: `source("path/to/script.R")`.

II. Exercises

Let's do some practice problems to challenge your understanding.

1. Calculate the average (`mean()`) and standard deviation (`sd()`) of the `speed` column in the `cars` dataframe.

```
mean(cars$speed)
```

```
## [1] 15.4
```

```
sd(cars$speed)
```

```
## [1] 5.287644
```

2. Create a new column in the dataframe `cars` called `time` defined as `dist/speed`

```
cars$time <- cars$dist/cars$speed
head(cars)
```

```
##   speed dist      time
## 1     4    2 0.5000000
## 2     4   10 2.5000000
## 3     7    4 0.5714286
## 4     7   22 3.1428571
## 5     8   16 2.0000000
## 6     9   10 1.1111111
```

3. A Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding numbers (1, 1, 2, 3, 5, 8, ...). Write a function `fibonacci()` that takes one argument `n`, the size of the sequence you want to print and outputs a Fibonacci sequence of that length. To make this easier, you can assume that the user always properly implements this function (they always provide an n of 1 or greater).

   Extra credit: Have the function handle cases for all numeric values of n and notify the user of an error i.e. if n <= 0 is given.

   Hint: Intialize a vector using `numeric(n)` to store your sequence, and have your function *return* the vector.

```
fibonacci <- function(n){

  # handle missing n input
  if(missing(n)){
    stop("Please provide the length of the sequence you want.")
  }

  # handle invalid inputs
  if(n <= 0){
    stop("Not a valid input for n. Values must be greater than or equal to 1.")
  }

  # initalize a vector of size n
  fibonacci <- numeric(n)

  # handle special cases of n = 1 and 2
  if(n >= 1){
    fibonacci[1] <- 1

    if(n >= 2){
      fibonacci[2] <- 1

      if(n >= 3){
        # loop through for a series of length n
        for(i in 3:n){
          fibonacci[i] <- fibonacci[i - 2] + fibonacci[i - 1]
        }
      }
    }
  }

  return(fibonacci)
}

# check that we can't break the function
fibonacci()
```

```
## Error in fibonacci(): Please provide the length of the sequence you want.
```

```
fibonacci(-1)
```

```
## Error in fibonacci(-1): Not a valid input for n. Values must be greater than or equal to 1.
```

```
# check fibonacci on valid inputs
fibonacci(1)
```

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

```
fibonacci(2)
```

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

```
fibonacci(5)
```

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

```
fibonacci(6)
```

```
## [1] 1 1 2 3 5 8
```