Introduction to R

25 April 2018

Quantitative Methods in Historical Linguistics
Henri Kauhanen

Contents

1 What R is 2

2 How to use this document 2

3 Working at the prompt 2

4 Variables 3

5 Vectors 4

6 Matrices 6

7 Session control 7

8 Data frames 9

9 Subsetting a data frame 9

10 Basic IO (input and output) 12

11 A very brief note on functions 13

12 Packages 14

13 Custom functions 15

14 Help!? 16

15 Homework 16

1 What R is

R is a general-purpose statistical computing environment and programming language available for free for a number of operating systems. It is widely used for purposes of exploring and describing data, running statistical tests, fitting theoretical models to data, visualizing data, and simulation. Visit https://www.r-project.org/ for more information.

The user interacts with R through what is known as a **command line** or **prompt**. When you launch R, you will see this command line, which begins with a '>' symbol. The basic operation is simple: you type a command, press Enter, and R does what it was told. The results appear printed at the prompt, in a separate window or in a separate file depending on the command issued.

2 How to use this document

Start the R program and work through this document by trying out all the commands listed here. When it comes to programming, learning by doing is pretty much the only way to learn!

3 Working at the prompt

First let's familiarize ourselves with the prompt. The simplest thing you can use R for is basic arithmetic, i.e. use it as a calculator. Try typing each of the following lines at the prompt:

```
1+1
1-9
2*4
1/10
2*(1/10)
```

You should get the obvious results.

A wealth of mathematical functions are also available "out of the box". Try the following:

```
sin(0)
cos(0)
sqrt(9)
3^2
3^(2+3)
exp(3)
log(10)
log(1)
```

A very useful thing to know in the long run is that you can use the arrow keys on your keyboard at the prompt: the up and down arrows cycle through your command

history, so that you can easily go back to a command you issued previously. The left and right arrows are used to move the cursor left and right. Use them if you need to edit a command at a specific point. Note that in order to issue a command with Enter, it is not necessary for the cursor to be at the end of the command.

4 Variables

Often we will need to store something in memory; for this we use **variables**. R variables come in many types, the simplest of which hold single numbers. Try the following:

```
x <- 10
```

The '<-' here is known as the **assignment operator**. In this case, it is used to push the number 10 in the variable whose name is x.

Once something has been stored in a variable, it can be recalled by simply typing the variable's name at the prompt:

```
x

## [1] 10
```

The variable may also be used as part of more complicated expressions:

```
(x + 2)*3

## [1] 36
```

It is also possible to store the result of a complex expression in a variable:

```
y <- (x + 2)*3
y/2

## [1] 18
```

Numbers are the most basic type of variable. Another very useful basic type is **character strings**. The following stores the string "hello world" in the variable s:

```
s <- "hello world"
s

## [1] "hello world"
```

Note the quotation marks: they are needed so that R knows that this is a string.

Not all operations are permitted if the variables involved are of the wrong type. Try:

```
y <- 36
s + y

## Error in s + y: non-numeric argument to binary operator
```

In this case R issues an error, since `s` is not a numeric variable and therefore cannot be added to `y`.

To check the type of a variable, you may do the following:

```
class(s)

## [1] "character"

class(y)

## [1] "numeric"
```

5 Vectors

A **vector** is a data structure that holds multiple things of the same type in a specified order: e.g. a sequence of numbers or a sequence of character strings. To form a vector:

```
x <- c(10, 2, 36)
y <- c(1, 0, 0, 0, 0, 0, 1)
z <- c("hello", "world")
x

## [1] 10  2 36

y

## [1] 1 0 0 0 0 0 1

z

## [1] "hello" "world"
```

Since vectors hold their elements in a specific order, it is possible to reference individual elements by specifying their position in the vector. For this, angle brackets are used:

```
x[1]

## [1] 10

z[2]

## [1] "world"

x[1] + x[2]

## [1] 12
```

To get the first two elements of x:

```
x[c(1,2)]

## [1] 10  2
```

To get everything but the first element of x:

```
x[-1]

## [1]  2 36
```

Ordinary arithmetic operations also work over vectors:

```
x <- c(1, 2, 3)
x + 2

## [1] 3 4 5

x*2

## [1] 2 4 6
```

To add together two vectors of the same length:

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

## [1] 1 3 8
```

There exist special methods for creating certain special kinds of numeric vectors. To get all the integers between 1 and 20, for example, you may type:

```
x <- 1:20
x

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

To get a sequence from 1 to 21 skipping over every other integer:

```
x <- seq(from=1, to=21, by=2)
x

##  [1]  1  3  5  7  9 11 13 15 17 19 21
```

To divide the interval between 1 and 10 into 34 equally spaced subintervals:

```
x <- seq(from=1, to=10, length.out=35)
x

##  [1]  1.000000  1.264706  1.529412  1.794118  2.058824  2.323529
##  [7]  2.588235  2.852941  3.117647  3.382353  3.647059  3.911765
## [13]  4.176471  4.441176  4.705882  4.970588  5.235294  5.500000
## [19]  5.764706  6.029412  6.294118  6.558824  6.823529  7.088235
## [25]  7.352941  7.617647  7.882353  8.147059  8.411765  8.676471
## [31]  8.941176  9.205882  9.470588  9.735294 10.000000
```

What will the following command output? (Think first, then try it out.)

```
seq(from=1, to=10, length.out=4)[1:3]
```

6 Matrices

A **matrix** is, essentially, a two-dimensional vector or a table of elements. A three-by-three matrix, for example, holds 9 elements, or **cells**, in total. Let's create such a matrix now:

```
A <- matrix(1:9, nrow=3, ncol=3, byrow=TRUE)
A

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

So we took the integers from 1 to 9 and put them in a matrix row by row. The following slightly modified command puts them in the matrix column by column:

```
B <- matrix(1:9, nrow=3, ncol=3, byrow=FALSE)
B

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

To get the element from the second row, first column:

```
A[2,1]

## [1] 4

B[2,1]

## [1] 2
```

In other words, to reference an entry in a matrix, we need to use two numbers; the first refers to the row and the second to the column. Similarly:

```
B[1,3]

## [1] 7
```

Again, arithmetic operations work over matrices:

```
B*2

##      [,1] [,2] [,3]
## [1,]    2    8   14
## [2,]    4   10   16
## [3,]    6   12   18
```

To access the entire second row of a matrix:

```
B[2,]

## [1] 2 5 8
```

Or the entire third column:

```
B[,3]

## [1] 7 8 9
```

Matrices can also hold character strings:

```
C <- matrix(c("hello", "world", "and", "goodbye"), nrow=2, byrow=TRUE)
C

##      [,1]    [,2]
## [1,] "hello" "world"
## [2,] "and"   "goodbye"

C[2,1]

## [1] "and"
```

7 Session control

R operates in a so-called **working directory**. To find out what the current working directory is, type:

```r
getwd()

## [1] "/home/kauhanen"
```

This is the default folder where files will be written, unless you specify something else. To change the working directory, use `setwd`:

```r
setwd("/home/kauhanen/Tmp")
getwd()

## [1] "/home/kauhanen/Tmp"
```

Now, you may want to save the variables we have created so far, so that you can easily access them later. First of all, to get a listing of the objects currently in R's memory:

```r
ls()

## [1] "A" "B" "C" "s" "x" "y" "z"
```

To save these, type:

```r
save.image("myvariables.RData")
```

The file `myvariables.RData` will now exist in your working directory (it is customary to use the `RData` extension for these files). You can also just type

```r
save.image()
```

in which case your data are saved in a file called `.RData`.

To access these variables later (for example, when you've restarted R), use `load`:

```r
load("myvariables.RData")
```

You can also save your command history (the one you cycle through with the up and down arrow keys):

```r
savehistory("myhistory.Rhistory")
savehistory()
```

The latter command saves the history in the file `.Rhistory` by default.

In fact, when you quit R, the program will ask you whether or not to save your current session. If you answer 'yes', both a `.RData` file and a `.Rhistory` file will be written in your current working directory. The next time you start R from the same working directory, these files will be loaded and your session restored.

To quit R, type:

```
q()
```

8 Data frames

A **data frame** is a bit like a matrix, but different columns may contain values of different types: for example, one column may contain numbers and another one strings. The easiest way to construct a data frame is to start with a number of vectors having the same length, and use the `data.frame` function to join them into a data frame:

```
x <- c(1, 0, 0)
y <- c("hello", "world", "again")
z <- c("another", "vector", "here")
df <- data.frame(x, y, z)
df

##   x     y       z
## 1 1 hello another
## 2 0 world  vector
## 3 0 again    here
```

Notice that each column of the data frame has a name, inherited directly from the names of the variables containing the original vectors. We can now reference a column of the data frame using its name after a dollar sign:

```
df$x

## [1] 1 0 0

df$y

## [1] hello world again
## Levels: again hello world
```

This shows a new kind of behaviour: `df$y` now has a number of "levels" attached to it. Technically, `df$y` is a **factor**, which in R just means a categorical variable with a specified set of values. (Those values are the levels.) While it is possible to create factors manually, the `data.frame` function converts character vectors into factors automatically. We do not normally have to worry about the distinction between character vectors and factors, though sometimes it is important.

9 Subsetting a data frame

When working with real data in R, the data are most often stored in a data frame. Suppose five speakers of English took part in an elicitation experiment where we measured how often they used the contracted form "I'm" instead of "I am" (expressed as a percentage of all occurrences of "I'm"/"I am"). Suppose these speakers were Jane, John, Molly,

Matt and Lisa, and that the first two were middle class and the last three working class speakers, and that the rates of contraction were 60.5%, 80.5%, 70.0%, 90.0% and 95.5%, respectively. We can join these data into a data frame:

```
speaker <- c("Jane", "John", "Molly", "Matt", "Lisa")
class <- c("MC", "MC", "WC", "WC", "WC")
rate <- c(60.5, 80.5, 70, 90, 95.5)
df <- data.frame(speaker, class, rate)
df

##   speaker class rate
## 1    Jane    MC 60.5
## 2    John    MC 80.5
## 3   Molly    WC 70.0
## 4    Matt    WC 90.0
## 5    Lisa    WC 95.5
```

One of the great benefits of using data frames is that they allow easy **subsetting** of the data set. Suppose we are only interested in the working class speakers. We can extract them from the data frame as follows:

```
df[df$class == "WC", ]

##   speaker class rate
## 3   Molly    WC 70.0
## 4    Matt    WC 90.0
## 5    Lisa    WC 95.5
```

This command specifies that we only want those rows of the data frame in which the `class` variable has the value `"WC"`.

Note: it is important that you have two equals signs, ==, in the above expression! This is because in R, as in most programming languages, two equals signs are used for the comparison of expressions (here, we compare the value of `df$class` to the string `"WC"` to find out if they're the same), whereas a single equals sign is used for assignments. The following just throws back an error:

```
df[df$class = "WC", ]

## Error: <text>:1:13: unexpected '='
## 1: df[df$class =
##                 ^
```

Once we have extracted the data for the working class speakers, we can of course store those data in another variable if we wish:

```
df_WC <- df[df$class == "WC", ]
```

Then accessing the contraction rates, for example, is an easy matter of:

```
df_WC$rate

## [1] 70.0 90.0 95.5
```

We could achieve the same thing directly with:

```
df[df$class == "WC", ]$rate

## [1] 70.0 90.0 95.5
```

This method of subsetting naturally works on all columns of the data frame. To get the speaker whose contraction rate equals 70%:

```
df[df$rate == 70, ]$speaker

## [1] Molly
## Levels: Jane John Lisa Matt Molly
```

Or all those speakers whose contraction rates do *not* equal 70%:

```
df[df$rate != 70, ]$speaker

## [1] Jane John Matt Lisa
## Levels: Jane John Lisa Matt Molly
```

(So != is R's way of saying "not equal to".)

The following command returns a zero-length factor, as no one in the data set has a contraction rate of 10%:

```
df[df$rate == 10, ]$speaker

## factor(0)
## Levels: Jane John Lisa Matt Molly
```

Similarly, the following command returns a data frame with zero rows:

```
df[df$rate == 10, ]

## [1] speaker class   rate
## <0 rows> (or 0-length row.names)
```

It is possible to use fairly complicated conditions to select rows from a data frame. To get the data for speakers whose contraction rates were less than 85%:

```
df[df$rate < 85, ]

##   speaker class rate
## 1    Jane    MC 60.5
## 2    John    MC 80.5
## 3   Molly    WC 70.0
```

To get the data for speakers whose contraction rates were between 65% and 85%:

```
df[df$rate > 65 & df$rate < 85, ]

##   speaker class rate
## 2    John    MC 80.5
## 3   Molly    WC 70.0
```

To get the data for working class speakers whose contraction rates were less than *or equal to* 90%:

```
df[df$class == "WC" & df$rate <= 90, ]

##   speaker class rate
## 3   Molly    WC   70
## 4    Matt    WC   90
```

And so on and so forth—you get the idea.

10 Basic IO (input and output)

The above method of inputting data quickly becomes tedious. With larger data sets, we will normally want to store our data in an external file and to simply read that file into R whenever we need it. Data is normally stored in a CSV (comma-separated values) file, which can be written using any ordinary text editor. (Another method is to collect the data in a spreadsheet program and then export them as a CSV file.)

In a CSV file, individual data fields are separated by commas (although other delimiters can also be used, the comma is traditional). It is also good practice to surround any text fields with quotation marks, since if those text fields contain commas, they will screw up the data format. Here's what our contraction data will look like in the CSV format:

```
"speaker","class","rate"
"Jane","MC",60.5
"John","MC",80.5
"Molly","WC",70.0
"Matt","WC",90.0
"Lisa","WC",95.5
```

Using a text editor (but not a word processor!), type these up and save the file as `contraction_data.csv`, placing the file in your R working directory (remember you can find out what your current working directory is with the `getwd()` command).

Reading the CSV file into R is easy as pie:

```
df2 <- read.csv("contraction_data.csv")
df2
```

```
##   speaker class rate
## 1    Jane    MC 60.5
## 2    John    MC 80.5
## 3   Molly    WC 70.0
## 4    Matt    WC 90.0
## 5    Lisa    WC 95.5
```

As you will see, df2 contains exactly the same data as df, which we previously input manually. The column names for the data frame are automatically taken from the first line of the CSV file.

Suppose we now realize that the entry for Molly in this data set is erroneous (her contraction rate should have been 71.5%, in fact). To remedy this point, we simply modify the rate entry for Molly:

```
df2[df2$speaker == "Molly", ]$rate <- 71.5
df2

##   speaker class rate
## 1    Jane    MC 60.5
## 2    John    MC 80.5
## 3   Molly    WC 71.5
## 4    Matt    WC 90.0
## 5    Lisa    WC 95.5
```

We can then export the corrected data into a CSV file:

```
write.csv(df2, file="contraction_data_corrected.csv", row.names=FALSE)
```

Open the file contraction_data_corrected.csv with a text editor to verify that this command succeeded.

Why did we write row.names=FALSE in the above write.csv command? To see why, try:

```
write.csv(df2, file="contraction_data_corrected_rownames.csv")
```

and see how this new file differs from the first one.

To get an explanation of what row.names does, you can access the documentation for the write.csv function by typing:

```
?write.csv
```

This, by the way, applies to all R functions. To exit the documentation, press the 'q' key.

11 A very brief note on functions

What exactly is a function? In R, as in programming languages more generally, a function is a little subprogram that does something for you, given some input. For example, the log function gives a logarithm, the c function constructs vectors, and the write.csv

function writes a CSV file. Functions often take optional arguments which, if not supplied explicitly, take on some default value. For example, `row.names` is an optional argument of the `write.csv` function. If we do not explicitly set it to `FALSE`, it will assume the value `TRUE`, and the row names of the data frame are written into the CSV file.

12 Packages

Sometimes the functionality you need is not available "out of the box". Luckily, however, volunteers have contributed a huge number of **packages** to R—these are bits of code that extend R's basic functionality. For example, there is no native support in R for drawing trees (and other network-like structures), but a package called `igraph` provides it.

Let's install `igraph`, as we will be needing it later on. Type:

```
install.packages("igraph")
```

You will be prompted to select a mirror location for the package download; for speed, it is best to select a mirror that's geographically close to you.

Once the installation has completed, the package exists on your system. To actually use it in an R session, you need to load it into the session:

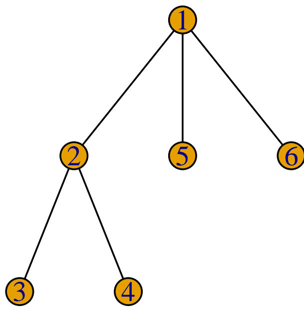
```
library(igraph)

## Loading required package:  methods
##
## Attaching package:  'igraph'
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
## The following object is masked from 'package:base':
##
##     union
```

As you load the package, R prints out some information which we don't have to worry about for now.

Here's a little preview of the sort of thing `igraph` can do. Note that we here use three functions which wouldn't otherwise be available to us: `graph` (used to construct networks, or "graphs" as the technical mathematical term goes), `layout_as_tree` (used to specify the layout of networks which are actually trees), and a custom version of the `plot` function (used to print a nice picture of the network).

```
g <- graph(edges=c(1,2,2,3,2,4,1,5,1,6), n=6, directed=FALSE)
plot(g, layout=layout_as_tree(g, root=1), edge.color="black",
     vertex.size=20)
```



We will also need the `rdist` and `phangorn` packages later on. Install these now:

```
install.packages(c("rdist", "phangorn"))
```

13 Custom functions

Sometimes you will need to do something no one has written a package for yet (if you get involved in computational research, sometimes = every day). In that case, you will need to write your own functions to accomplish what you need. Open up any text editor (but not a word processor), and type the following lines:

```
my_fun <- function(x, y) {
  x^2 + sqrt(y)
}
```

This rather silly function takes two inputs, `x` and `y`, squares `x` and takes the square root of `y`, and adds the two together. (It's unlikely you'll ever need this, but at least it illustrates the principles of writing custom functions!) Note in particular that the inputs, or **arguments** of the function, are listed in parentheses after the word `function`. The actual function definition goes in between the curly brackets and is known as the function's body.

Now save the text file as `my_fun.R` in your R working directory. Go back to R, and type:

```
source("my_fun.R")
```

This is rather like the library command for packages. R takes the file `my_fun.R` and executes all code found in that file. In this case, the result is that a variable named `my_fun` now contains our silly squaring, square-rooting function. You can now use it:

```
my_fun(x=10, y=9)

## [1] 103
```

Or even:

```
my_fun(10, 9)

## [1] 103
```

The arguments can even be vectors:

```
my_fun(x=c(1,2), y=c(4,9))

## [1] 3 7
```

In this case, the output is also a vector, whose first element is $1^2 + \sqrt{4} = 1 + 2 = 3$ and whose second element is $2^2 + \sqrt{9} = 4 + 3 = 7$.

We will return to the theme of custom functions many times during this course—the above is just a sneak preview.

14 Help!?

When programming in practice, you will often run into problems: your code won't run, you don't know how to implement some functionality, and so on. Aside from using R's built-in documentation (remember `?write.csv`), the following resources may help:

- Google "R how to" followed by your problem. This should usually be the first thing you do, and will help you to solve the problem in 99% of cases.
- Googling an R problem often takes you to a site known as Stack Overflow, where people can ask programming questions which other people then answer. Here's a direct link to R questions on Stack Overflow: https://stackoverflow.com/questions/tagged/r. You can also post your own questions, of course—but make sure first that someone else hasn't already posted that same question (very probable)!
- The official R introduction is available at https://cran.r-project.org/doc/manuals/r-release/R-intro.html. This is a very comprehensive (if sometimes not so comprehensible) document.
- As you delve deeper and deeper into the world of R, having Patrick Burns's *R Inferno* to hand will be extremely valuable: http://www.burns-stat.com/documents/books/the-r-inferno/.

15 Homework

As homework:

1. Navigate to https://www.r-project.org/ and install R on your own computer.
2. If you didn't have enough time to work through this document in class, finish at home.
3. Then modify the `my_fun` function so that it takes *three* numeric arguments, x, y and z, and outputs

$$\frac{x^{-1} + \log(y)}{\sqrt{z}}.$$

4. Instead of interacting with R directly, many people choose to use a so-called IDE (Integrated Development Environment). One of these is RStudio, available for free at `https://www.rstudio.com/`. (In addition to the R prompt, this program gives you a window displaying all your variables, a window for writing custom functions, a window for documentation, and so on—hence "integrated".) Download and install RStudio and try it out. Do you prefer to work with R directly or to use RStudio?