R: Introduction to Basic Features

Author: Helen Lockstone

Introduction

This workshop will introduce you to the R environment, with the aim of becoming familiar with how commands are written and structured and how to perform basic tasks associated with data analysis using R.

All material can be found at the course link:
https://www.well.ox.ac.uk/bioinformatics/training/R_Basic_Features

For today's session you need a basic installation of the R software and the RStudio interface. You can use the desktop machines in front of you, or your own laptop if you prefer.

The R environment

Open a new R session by launching RStudio. You should see the 4 panel display briefly introduced earlier. You can run any commands shown in this document by copying the R code into the topleft panel of RStudio and clicking the icon with a green forward arrow in the toolbar. The line of code where the cursor is will be automatically copied into the bottom left panel and executed by R in realtime. You should see the R prompt **>** again when it has finished. Depending on the command, some output may also be printed to the screen or it may simply return to the prompt (ready to receive another command). Note that something may have happened 'behind the scenes' - such as creation of a new object. It is important to know what each command has done and it was as intended.

If you see an error message, first double check that the command is entered exactly as shown. If you can't find the problem, please ask for help. The advantage of copying the commands into the text editor panel of RStudio means that you can save the file afterwards to have a record of what you have done and can easily re-run again in future.

If a command is not valid in the way it is constructed (its syntax), R will print an error message to the screen. These can sometimes be hard to interpret but particularly common culprits are simple typing mistakes or quotes and brackets, whether they are in the wrong place, missing, or not in pairs (e.g. missing a closing bracket).

R is a very interactive environment; commands can be entered into the console one by one, and each is interpreted and executed by R in real time. R is a high-level programming language, meaning that it is fairly human-readable. We will start by entering a few simple commands and discuss what is happening.

You can type the R commands directly, which is useful at first to get used to the syntax (structure) of commands. However, you can copy/paste if you prefer.

To do anything useful in R, we need to use **objects** to hold data or information and perform various operations on them. The terms *object*, *variable* and *data structure* can all refer generally to objects created in R.

Although *variable* is a widely used programming term and would be the preferred term in certain situations, I will use object as a general term throughout to refer to any of R's data structures. These include vectors, factors, matrices, dataframes and lists. We'll focus on vectors initially and meet dataframes later on.

R command syntax

The following few commands give some sense of how R stores information in objects and how simple manipulations of data can be performed. The # symbol indicates a comment or output that will be produced by running the command - you only need to type the lines without # symbols at the start.

The first command assigns the value 1 to a new object we create and name **x**.

```
x <- 1
```

The assignment operator is <- and running this command creates a new object named **x** in R's memory. To inspect the contents of the new object, we simply type its name **x**:

```
## Inspect the contents of the new object
x

## [1] 1
```

Note that the contents of an object will be overwritten - without any warning message -if later assigned a different value:

```
x <- 4
# x now contains the value 4
x

## [1] 4
```

We can also perform operations directly on the object (note the object itself does not change):

```
x * 2

## [1] 8

# x still contains the value 4
x

## [1] 4
```

```
## unless we were to re-assign the output back to the same object:
x <- x * 2
# x now contains the value 8
x

## [1] 8
```

R is case sensitive so x and X are different:

```
X
```

Trying to run this command will produce an error message: Error: object "X" not found

Object names are chosen by the programmer – informative names are helpful for several reasons. You can use capitalization, _ or . to separate parts of an object name but they cannot contain spaces, nor start with a number. To avoid confusion or potential issues, it is also best not to give them the same name as an R function, which have their own defined names. An object named **raw_data**, **raw.data**, **rawData** (or even **d.raw** for minimal typing!) is fine, but trying to assign a value to a variable named **raw data** will give an error because R cannot parse it correctly.

Elsewhere R ignores whitespace so the following commands are equivalent:

```
x<-4+3
x <- 4 + 3
```

Now we will assign a set of numbers to our variable **x**, which R will store as a *vector*

```
x <- c(1,2,3,4,5)
x

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

If we forget the closing bracket before pressing enter, a + sign indicates the command is incomplete:

```
x <- c(1,2,3,4,5
+
)
>
```

If we can't simply continue our command, use Esc or control-C to return to the prompt and start again. There is also a useful command recall option – you can use the up/down arrows to scroll through previously entered commands, which can be edited or re-run to save typing again

Rstudio and some text editor programs highlight different parts of the syntax in different colours and automatically close brackets and quotation marks to help eliminate typing mistakes.

R Objects

Objects can be created in many different ways and hold different kinds of information. Unlike other programming languages, there is no need to initialise a variable or object in R (define it before first use) – it can simply be created and used directly. R also automatically decides which of its data structures and types are most appropriate for the data given, rather than being explicitly specified by the programmer.

We'll work through some examples and look at ways to access or manipulate the data contained within an object. Be aware that the type (class) of an object and data it contains (numeric, character etc) can affect how it is treated by R.

There are many shortcuts in R to avoid tedious or error-prone steps. When we created our small example vector containing the numbers 1 to 5, we issued this command:

```
x <- c(1,2,3,4,5) # the c function tells R to concatenate these 5 numbers

## equivalently we could use this command
x <- 1:5 # handy if we needed a much longer vector such as 1 to 100, or 1 to
100000
```

We can also put together non-consecutive strings of numbers or a mixture

```
x2 <- c(1,3,5,7,9)
x3 <- c(1:5, 7, 9, 10:15)
```

If we need to create a sequence of numbers, the function *seq* is very useful.

```
seq1 <- seq(from=1, to=99, by=2)
seq2 <- seq(from=0, to=1, by=0.01)
```

Functions are defined blocks of code that perform a particular task. R has many hundreds of in-built functions for common tasks, and they can be used by 'calling' the function by its name. Above, we used the function *seq*; the function name is followed by any *arguments* in parantheses - the arguments determine exactly what the function does and enable it to be useful for many situations. By changing the arguments, we changed the sequence of numbers that was produced. Some arguments are required and others are optional or may have default settings defined in the function. Documentation about the usage of any R function is given in the corresponding help page:

```
help(seq)
```

Vectors

Vectors are one-dimensional objects; in the case of the object **x** we created earlier, it has length 5.

There is an in-built R function called *length* that we can use to check how long any given vector object is:

```
length(x)

## [1] 5

## If we change what is assigned to x, the length of the vector is automatica
lly adjusted:
x <- 1:10
length(x)

## [1] 10
```

Vectors are R's primary object type and many computations in R are highly efficient because they operate on the whole vector at once, rather than element by element.

Vectors can contain numeric or character data (or both). We can create a new vector, y, containing the letters 'abc':

```
y <- c("a", "b", "c")
```

In Rstudio's top-right panel, we see details of all the objects that have been created in the current session and are available to use. Note the differences between x and y. We can also see how R has automatically treated them differently by checking the class of the objects directly:

```
class(x)

## [1] "integer"

class(y)

## [1] "character"
```

Now try running the following command:

```
y <- c(a, b, c)
```

What do you think R has tried to do and why does it result in an error message?

Accessing elements of a vector

Square brackets are used to access specific elements or subsets of a vector, factor, matrix or dataframe. Let's create a new vector as an example:

```
x <- c(1:5, 10:14)

## now extract 3rd element
x[3]

## or extract alternate elements
x[c(1,3,5,7,9)]

## or extract subset of elements
x[3:6]
```

R will decide the most appropriate way to store the data it is provided with, and there are ways to convert between different object structures and classes if needed. To give more examples of how data is interpreted by R, run the following and note the results (discuss with your neighbour).

```
x2 <- c(1:5, 6.5)
class(x2)

## [1] "numeric"

x3 <- c(1:5, 6.5, "a", "b", "c")
class(x3)

## [1] "character"
```

This gives some idea of R's internal rules. Because the way data is being handled by R is important for both performing computations correctly and the source of many error messages, it is useful to be familiar with the common data types. Some functions, such as computing a mean for example, require numeric data objects to operate on. What happens when you try to find the mean of x2 and x3?

```
mean(x2)
mean(x3)
```

General Comments

The topright panel in RStudio showing existing objects and information on their contents is invaluable. Not only does it save writing separate commands to check these details, it can help you check:

- that your object has been created correctly and contains what you wanted it to

- how R will treat the object internally (when functions are applied to it)
- possible reasons for an error message
- spot any changes that happen to your object (intentionally or otherwise)
- that you do not have too many and/or poorly named objects that could lead to mistakes

Sometimes you are testing things out and creating lots of objects – that's fine but it's always good to start a new session when running or checking your final code to be sure previous objects do not affect it in any way. Sessions can also be cleaned up by deleting objects with the *rm* command.

There are hundreds, probably thousands, of in-built functions in R. Some you will use very often and others rarely or never. There are always several ways to do the same thing in R, using closely-related functions.

Examples of the functions we have used so far include *length*, *mean*, and *class*. In the case of *length*, the argument supplied is the name of the object we wish to find the length of. We can check the length of another vector object simply by changing the argument.

The *length* function is only applicable to vectors (or factors) and does not work on other data types such as matrices or dataframes, which are 2-dimensional. Often, data is stored in a table format (e.g. in Excel), and commonly handled as a data frame in R. We will introduce these ideas using practical exercises after the break.

An important concept before we get started is the working directory. If we want to read in data from existing files or create new ones to save any plots or analysis results, R needs to know where to find/save them.

If you are working on a personal laptop, create a folder for today's course (you can name it 'R_workshop' or whatever you wish) somewhere in your file directory.

We now need to set this as the working directory:

From the Session menu, select 'Set working directory', and then 'Choose directory…'

Navigate to the relevant directory and select OK to set it as the working directory. Note that this will automatically execute a *setwd* command in the console.

You will also need to download two data files by entering the following commands:

```
download.file("https://www.well.ox.ac.uk/bioinformatics/training/R_materials/
inflammation_data.csv", "./inflammation_data.csv")

download.file("https://www.well.ox.ac.uk/bioinformatics/training/R_materials/
sample.csv", "./sample.csv")
```