R: Introduction to Basic Features

Tutors: Helen Lockstone and Ben Wright Bioinformatics Core Wellcome Centre for Human Genetics University of Oxford

Practical Tutorial

Earlier we introduced the R software environment, some key features of the R programming language and how to start using it. We will now do some practical exercises working with example data to perform typical tasks. First, some important acknowledgements:

The tutorial that follows is in part adapted from the Software Carpentry Foundation Programming with R, specifically the Analysing Patient Data tutorial. The Software Carpentry material is available for re-use under a Creative Commons License and I am grateful to the original authors.

This document and a variety of extensions to the tutorial material were written and developed by Helen Lockstone, with contributions from Ben Wright. All the material for today's course is available here

You can open a copy of this tutorial guide on the machine you are using to copy and paste any particularly long commands. Generally, manually type in the shorter commands to get used to the R environment and structure of commands – most are very short. Commands are shown within light-grey boxes; remember that lines starting with a # are comments and do not need to be run.

Setup

We set up our R session for this practical before the break. You can check the current working directory with the following command:

```
getwd()
```

The output (directory path) should match the location you are working in today, and this folder should also contain the files named "inflammation_data.csv" and "sample.csv".

If this is the case, you should have everything you need to run this tutorial. If not, please let us know and we will get you started.

If you see an error message at any point, first check the command matches that in the tutorial exactly and that you haven't accidentally missed an earlier command out. Pay particular attention to lower/upper case letters, underscores, dashes or dots in function or object names, and that brackets and quotes are correctly paired. If you can't spot the problem or have a question at any point, please don't hesitate to ask.

Helpful Tips

- Enter your commands in the top-left panel of RStudio (a text editor) as this means they can be saved to keep a record of what you have done. To run a command written in this panel, make sure the cursor is located somewhere in the line of code and click the Run icon with the green arrow. The command is automatically copied into the lower console panel and executed by R.
- To save your work, click on the disk icon in the same toolbar as the 'run' button. Giving a filename with a .R extension, such as 'R_course_code.R' saves it as an R script file - this can be opened like any text file but the .R extension is useful to identify your files that contain scripts. This is the usually the easiest way to work; if you need to close your session and return to it another time, it is easy to run

the code again. If it were appropriate, all of the code stored in a script can be executed in R from start to finish with the command:

```
source("script_name.R")
```

- The # symbol is the comment character in R – lines in a script starting with a # (or more commonly ##) can include comments about what the code is doing. It is strongly recommended to comment your code as much as possible because it will help others understand what it is doing, including yourself if you revisit it sometime after originally working on it. A # can also be used after a command to note any result or information relating to that command – everything after the # will be ignored by R but serve as useful information to the programmer. You can also use long lines of # symbols to break your code into sections.
- In the console panel, you can use the up/down arrow keys to scroll through previous commands to re-run or edit them easily if needed. RStudio also has features to help auto-complete names of functions and objects, and pairing brackets and quotes.

Reading in data from a file

The first thing we need to do is load or read in the data from our files so it is accessible in the current R session. There are a few possible ways to do this but we will use the function **read.csv** because our files are saved in 'comma-separated values' or csv format. To find out details of how to use this function you can search RStudio's Help menu (bottom right panel) or type:

```
help(read.csv)
```

The help page shows us the arguments for this function and their default values where applicable. For example, we see sep=",", which means the fields in a row will be separated on commas, and header=TRUE, which means it is expecting the first row of the file to contain names for each column. The first file we will work with is the one named 'inflammation_data.csv'. If we inspect this file in Excel or a text editor (by opening it directly from its location on your computer) we see there are no column names, just a large set of numerical values. Therefore, we need to explicitly include the header argument in our command, changing it to FALSE to over-ride the default behaviour of the function. This is an example of how arguments modify a function's precise behaviour, rather than requiring two separate functions to exist for files with/without header rows.

```
inf.data <- read.csv("inflammation_data.csv", header=FALSE)
```

Note: if you see an error message similar to 'No such file or directory' when trying to read in a file, it is likely that either (i) the file is not located in the current working directory; (ii) there is one (or more) typos in the filename.

It is worth breaking this command apart to refresh on some of the terminology used this morning, as it can be hard at first to differentiate object names (decided by us) from R functions (pre-defined in the language) when looking at R code or commands.

We have given a name for a new object, **inf.data**, in which to store the contents of the file 'inflammation_data.csv'. Our object name is descriptive without being too long, shortening 'inflammation' to 'inf' for our convenience. We are using the in-built R function **read.csv**, and provide two arguments:

- the name of the file to read in
- header=FALSE, indicating to R that our file does not have a header row

There are many other arguments to the **read.csv** function to further refine its behaviour but these are either optional or the default settings are fine for most situations. By running this command, R creates the object **inf.data** and information about it appears in the top-right panel of RStudio. It is detailed as 60 obs. (observations) of 40 variables and if you hover the mouse pointer over the name, **inf.data**, it indicates that the object is a data frame. Finally by clicking the spreadsheet icon to the righthand side, the contents of the object are loaded in the top-left panel in a new window, titled by the name of object. This shows we have successfully loaded the data. The columns have been automatically named by R as V1 through to V40, as column names have to start with a letter. The rows are simply numbered - the only restriction to row names is that they must be unique.

What do you think will happen if you run the command above without including the header argument? Try it by saving the contents into a new object called **test** and compare this to **inf.data**:

```
test <- read.csv("inflammation_data.csv")
```

Another object named **test** now appears. We can use the **head** command to inspect the first 6 rows of each object. For display purposes, we'll also only include the first 6 columns:

```
head(test [, 1:6])
head(inf.data[ , 1:6])
```

Discuss with a neighbour what you observe.

This is a good example of how easily something unwanted can happen in R and the importance of checking your objects contain what you intend them to. Any mistakes can simply be corrected by re-running the command e.g. with the appropriate header argument, and overwriting any previous version of the object.

Two-dimensional data structures

Our original file contained rows and columns of data, and R has suitable 2-dimensional data structures to store such data: matrices and dataframes. These can both be thought of as tables of data, analogous to an Excel spreadsheet. Matrices require all columns to be of the same type, while data frames can have columns of different data types. Given that experimental data is often a mixture of numeric values (e.g. measurements) and associated descriptive information, data frames are a very commonly used data structure in R.

NB while it is possible to hold mixed data types in a matrix object as well, R will use its internal hierarchy of data types to choose one that is applicable to all columns – often this means numerical columns get converted to character strings, and certain functions may not perform as expected if this is not noticed, or produce an error message.

In this case, a data frame object has been created:

```
class(inf.data)
```

We can check how each column of data has been treated by R e.g. for the first column:

```
class(inf.data[,1])
```

*In fact, as all the columns contain data of the same type, R could equally well store this data as a matrix object. A data frame has been created because the functions **read.csv** and **read.table** are specifically designed to deal with mixed column classes and produce data frames by default. Another function **scan** can be used to read in matrices, especially large ones.*

It is worth noting that some functions operate on matrix objects, and so converting between classes is sometimes needed.

```
inf.data <- as.matrix(inf.data)
class(inf.data)
```

We will continue with the matrix form of this object for now, and load a mixed dataset later. Most operations on either kind of 2-dimensional object are the same. For example, we can find the dimensions of a matrix or a data frame with the **dim** function.

```
dim(inf.data)
```

The output of **dim** is printed to the screen and shows the number of rows the object contains, followed by the number of columns (the convention is always rows, then columns but this can be hard to remember at first as there is no indication).

If unsure, the functions **nrow** and **ncol** will return the number of rows or columns respectively; these take as their argument the name of the object:

```
nrow(inf.data)
ncol(inf.data)
```

This particular file suffers from the lack of any labels to annotate what data is recorded in the rows and columns. The Software Carpentry tutorial provides the following information: *"We are studying inflammation in patients who have been given a new treatment for arthritis. Each row holds the observations for just one patient. Each column holds the inflammation measured in a day, so we have a set of values in successive days."*

Our object has 60 rows and 40 columns, so we infer from the information above that there are 60 patients, and 40 days.

Again we see a way for mistakes to easily creep into data analysis – here we have to rely on information given to us second-hand to know what is what. What if that information were wrong? Are there any checks we can make ourselves to be sure patients are in rows? We are not told how many patients were included so simply checking the number of rows won't help. And what if there were 50 patients and measurements taken over 50 days?

Even with careful scrutiny it would be hard to know how the data are presented (patients in rows or columns) from the data alone. We could perhaps make some plots to help us, or we might spot the zero values in the first column. Scrolling down the object display in the top-left panel or displaying the first column in the console confirms they are all zeroes, and the values in each row tend to rise across the first few columns. We may be reassured by this that the patients are indeed in the rows, since we might expect inflammation to rise over time, and an individual recording 0 on every single day might be unlikely (though not impossible).

It would be prudent to add some row and column names to reduce the chance of making a mistake later when dealing with this data:

```
rownames(inf.data) <- paste("Patient", 1:60, sep="_")
colnames(inf.data) <-  paste("Day", 1:40, sep="_")
```

This introduces the very useful and versatile function **paste**. Note that adding row and column names does not change the size of the data object, but we can see them displayed by reloading the object. They are similar to the alphabetical columns and numbered rows in an Excel spreadsheet.

Accessing Data

Earlier we looked at accessing elements of a one-dimensional vector object. For matrices and dataframes, a similar approach with square brackets is used:

object_name[rows, cols]

By specifying the rows and columns of interest, an object can be subset in a variety of ways to inspect or extract different parts of it.

```
inf.data[1,1] # this pulls out the data value in the first row of the first column

inf.data[30, 20] # any single entry can be extracted by specifying the row and column
```

How might you select the data in the first 5 rows for the first 5 columns? Add your command for this below.

```
## If we need to select non-contiguous portions of the object, we'll need the help of the c() function:
inf.data[c(1, 3, 5), c(10, 20)]

## If you want to display all columns for selected row(s), leave blank space after the comma:

inf.data[5, ] # All columns for row 5
```

```r
## Or blank space before the comma to select all rows for given column(s):
inf.data[ , 1:5] # all rows, columns 1 through 5
```

We added column names to our object earlier, and columns can also be accessed by name (in quotes):

```r
inf.data["Patient_1", ]
```

Suppose you want to determine the maximum inflammation for patient 5 across days three to seven. To do this you would extract the relevant subset from the data frame and calculate the maximum value. Which of the following lines of R code gives the correct answer?

1. max(inf.data[5,])
2. max(inf.data[3:7, 5])
3. max(inf.data[5, 3:7])
4. max(inf.data[5, 3, 7])

Analysing Data

We can perform many simple analyses of the data by applying functions such as max, min, mean, or summary to our data object. We might want to determine the maximum value per patient or the average value per day. The following examples illustrate how this can be done extremely efficiently in R, starting with an approach that is the opposite (and definitely not recommended!).

Suppose we want to find the maximum inflammation score for each patient across the 40 days of measurements. Let's start by calculating it for patient 1.

Extracting the data for patient 1 (i.e. the first row) is the first obvious step, and perhaps we decide it makes sense to store the values for this patient in a new object:

```r
inf.patient1 <- inf.data[1, ]
```

We can then calculate the maximum value for Patient 1:

```r
max(inf.patient1)
```

Although this seems reasonable enough, there are several issues:

- We've created an additional object to store data that is simply a duplicate of what is already contained in our original object
- It doesn't scale well to do this for all 60 patients
- The result is output to the console and therefore hard to do anything further with

If we did continue with this approach, there would be 60 new objects (all with very similar names), a high probability of having made a typing mistake somewhere (perhaps overwriting one patient's data with another), and a large set of results that we'd have to manually write down or transfer to an Excel spreadsheet - all of which is very messy and prone to error.

We can easily dispense with the intermediate step of creating a new object:

```r
# max inflammation for patient 1
max(inf.data[1, ])

# or equivalently
max(inf.data["Patient_1", ])
```

These commands are the same as extracting the data for patient 1 as we did earlier, but instead of printing to the screen or storing in a new object, the command is used directly as an argument to the function **max** by enclosing in the ().

Commands can be nested in this way to achieve multiple steps in a single line of code; too many commands in one line though can make it harder to work out what the code is doing, as well as increase the chance of the code not doing as intended – the location of brackets becomes vital.

We'd really like a way to this for all 60 patients without duplicating the code 60 times. Loops are one option (not discussed here) but the **apply** function is the most efficient approach:

apply allows us to repeat a function on all of the rows (MARGIN = 1) or columns (MARGIN = 2) of a data frame simultaneously:

```
max_inf_patient <- apply(inf.data, MARGIN=1, max)
```

Similarly, we could compute the average inflammation per day with a single line of code:

```
avg_inf_day <- apply(inf.data, MARGIN=2, mean)
```

Comparing these two commands will help understand the **apply** function, which is not intuitive but highly efficient as we have seen. The arguments to apply are:

- The data object
- MARGIN, indicating whether to apply over rows (1) or columns (2)
- The name of (another) function to be applied

We wanted to find the maximum inflammation score for each patient, so we looked across the rows and used the **max** function. To modify the command to find the average inflammation per day, we switched the MARGIN argument to 2 for columns, and gave the final argument as **mean**.

While the MARGIN argument is explicitly assigned above, R is equally happy to infer from the shortened command apply(inf.data, 1, max) that the 1 should be assigned to the second defined argument of apply. You can also write your own bespoke functions as required and use apply to run them over an object.

We have also solved the final issue with our initial approach by storing the results in suitably-named objects for further work.

```
length(max_inf_patient)
head(max_inf_patient)
```

Another useful function is **summary**. This returns the minimum value, first quartile, median, mean, third quartile and the maximum value, all very useful information to make an initial inspection of your data.

```
summary(inf.data[, 1:4]) # for each of the first 4 days
```

Plotting Data

Visualising data is a vital part of statistical analysis, and R's plotting capabilities are a key reason for its popularity. There is a related course R: Visualisation that you can take if interested to learn more. Here, we introduce ways to make a few simple plots. Let's take a look at the average inflammation over time. Recall that we already calculated these values above and saved them in an object named **avg_inf_day**. Plotting the values is done with the function **plot**:

```
plot(avg_inf_day)

## Default labels and settings are used but we can refine our plot with some additional arguments:

plot(avg_inf_day, main="Inflammation Scores Over Time", xlab="Day", ylab="Average_inflammation_score")

plot(avg_inf_day, main="Inflammation Scores Over Time", xlab="Day", ylab="Average_inflammation_score",

## Similarly, we could plot the data per patient:
plot(max_inf_patient)
```

```
## Here, we might decide to use a boxplot instead:
boxplot(max_inf_patient, main="Maximum Inflammation Scores", ylab="Max_inf_score")
legend("topright", legend="n=60 patients", cex=0.8) # adding a legend
```

When we are happy with our plots, they can be saved to a file.

```
 pdf("Inflammation_plots.pdf", onefile=T)
plot(avg_inf_day, main="Inflammation Scores Over Time", xlab="Day", ylab="Average_inflammation_score", 
boxplot(max_inf_patient, main="Maximum Inflammation Scores", ylab="Max_inf_score")
legend("topright", legend="n=60 patients", cex=0.8)
dev.off()
```

This will be saved to the current working directory by default so if we check the folder, a new file named 'Inflammation_plots.pdf' should have been created. The onefile=T argument instructs R to append additional plots to the same file and the **dev.off()** command at the end closes the file connection. You can also export plots directly to a pdf file from the RStudio plot panel.

Data Handling

We'll next read in data from another file to illustrate a few more features of data frames and how to work with them in R. In this case the file does contain a header row and the default arguments for read.csv are appropriate for this file so we only need provide the filename:

```
data2 <- read.csv("sample.csv")
head(data2)
```

This displays the first 6 rows, and we can see immediately that we have a range of different types of data in each column. Let's see how R has treated it (you can paste the following 4 lines as one block).

```
for(i in 1:ncol(data2))
{
    print(class(data2[,i]))
}
```

Here, we've used a **for** loop to iterate over each column in the object **data2**, and print to screen the class of each column. The output tells us that columns 1:3 are treated as factors, column 5 as numeric and the remaining columns as integer values. We haven't yet mentioned factors and will only briefly discuss them here but they are very important for statistical analysis in R. They are one-dimensional, like vectors, and are particularly useful for categorical data.

```
length(data2$Group)
data2$Group
```

Each of the 100 entries in the Group column are printed to the screen, and at the end is the additional information: *Levels: Control Treatment1 Treatment2*

These are the unique set of entries in this column, known as the levels of the factor. You may have come across factors before in the context of experimental design or ANOVA - in this case the experiment might test the effect of 2 treatments (Treatment1 and Treatment2) on blood pressure, perhaps to see if it reduces compared to a control group. Other information about the patients, such as their age and gender may be useful to include in the analysis, especially if they are not matched across the treatment groups.

R will treat any columns containing character strings (text) as a factor by default with **read.csv** or **read.table**. We don't always want to do this though, and indeed it is usually preferable to switch this behaviour off, and specifically convert data we do want to treat as factors later. This is because factors store data differently and so can sometimes behave differently to vectors. For example, here the first column of IDs would preferably be a character vector, as could Gender unless we needed to include it as an additional explanatory factor in our analysis model.

The way to switch off this default behaviour is with the argument 'stringsAsFactors' – if you check the help page for **read.csv** again, you'll see it listed among the arguments, and it is TRUE by default (although it's not readily apparent that this is the case).

```
data2 <- read.csv("sample.csv", stringsAsFactors=FALSE)
```

Repeating our loop to check the class of each column, we now see that the first 3 columns are character vectors rather than factors:

```
for(i in 1:ncol(data2))
{
    print(class(data2[,i]))
}
```

We can specifically convert the Group column to a factor:

```
data2$Group <- as.factor(data2$Group)
class(data2$Group)
```

A very useful summary function is **table**:

```
table(data2$Group)
table(data2$Gender)
```

This alerts us to the fact that data in the Gender column has not been entered consistently, which we might have already spotted from viewing the object in RStudio.

To fix this, we can make sure F and M are used throughout; this involves determining which rows contain a lowercase f for example, and substituting with F. Similarly for the lowercase case m:

```
data2$Gender <- gsub("f", "F", data2$Gender)
data2$Gender <- gsub("m", "M", data2$Gender)

## checking we have modified the data as intended
table(data2$Gender)
```

This last section is just a brief foray into data handling and manipulation in R, which enables all manner of formatting, editing, updating or data cleaning tasks to be performed. These are frequently required before embarking on some analysis, and often take longer too! It is the topic of the next course in the series, R: Data Handling - please sign up if interested.

Well, we have reached the end of today's course – we hope it was useful for you and good luck on your R journey! Please see below for further reading and information.

Further Resources and Useful Information

We have helped collate a comprehensive set of R resources with IT Services (with particular thanks to Dave Baker for creating the website, and contributions from Samantha Curle and Andre Python), which lists courses (online and workshops) as well as recommended textbooks, websites etc.

The Software Carpentry Foundation website contains many tutorials for learning a variety of programming languages, including R. There is also a series of domain-specific Data Carpentry courses, which focus on computational skills needed to handle and analyse data - tutorials are currently available for Ecology, Genomics, Geospatial Data and Social Sciences, with others in development.

- Software Carpentry lessons
- Data Carpentry lessons

In particular, the tutorials at the following links give further details and examples on some of the ideas already introduced or extend to other topics once you feel comfortable interacting with the R environment.

- Data structures
- Read and write csv files
- Factors
- Addressing data
- Best practices
- R genomics
- Geospatial data

Getting Help

As you read the R help pages, you may not find them terribly helpful. The R help function is most useful for refreshing your memory about specific functions you have used before. Unfortunately, it is not very useful for learning the language itself.

There are several online forums used by R programmers, novice and expert, to get help and advice from their peers. Searching for your R problem will often give results from one of these forums.

One page 'quick reference' documents: - R cheat sheet - R reference card Full R manual: - R manual

Installing Packages

When freshly installed, R has only its basic functions available. This is still a considerable number of functions and is adequate for a great many tasks. R's functionality is extended by the use of packages, each of which is a self-contained bundle of additional functions. These are typically written by people other than the main R developers, but a centralised repository of these packages (CRAN) is maintained and accessible from within R. For example, to install the package called 'limma', you would use:

```
install.packages("limma")
```

which downloads the files needed. You may be asked to choose where to download the files from, or asked to confirm that they will be installed to a user directory if you do not have admin permissions for your computer. Some R packages rely on other R packages. R handles all of those dependencies in the background and will download and install every needed package as part of installing the requested one. Installing packages does not make them available straight away. You need to tell R to make a package available in your current session using the command:

```
library(limma)
```

You will need to do this each time you restart R. This step is required so you don't waste memory by loading in packages you don't need every session. Note that the help() function only knows about functions that have been made available using the library() command.

Bioconductor

Bioconductor is a separate third-party repository of R code, specifically geared towards bioinformatics. Many of the packages in Bioconductor are also available via CRAN using the usual package installation method. However, Bioconductor has its own preferred installation mechanism which gets around some of R's more annoying limitations with version incompatibilities. In particular, the packages in Bioconductor are updated more swiftly following the release of a new version of R. To get started with Bioconductor, use the following command:

```
source("https://bioconductor.org/biocLite.R")
```

This loads an R script hosted on the web that defines a new function. You then use this function in a similar way to how you would use the install.packages() function:

```
biocLite("limma")
```

Loading Bioconductor packages this way is typically a longer process but more reliable.