

CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

<http://www.cs.utoronto.ca/~radford/csc120/>

Week 2

Typing Stuff into R Can be Good ...

You can learn a lot by seeing what happens when you type an R command.

Interactive use of R is also a good way to start exploring a new data set.

For instance, you can

- make sure it's actually the data you were told it was
- play around with how best to plot the data
- look at plots to see if there are any obviously erroneous data points
- see if relationships between variables seem to be roughly linear

... Or Bad

But when you're seriously analysing data, you **don't** want to just type stuff, since

- it's tedious to type things again and again
- it's easy to make a mistake when typing something and not notice
- you won't be able to remember exactly what you typed
- other people won't be able to replicate your analysis
- if you decide to change the analysis slightly, you have to do it all again
- if you get another similar data set, you have to do it all again

Instead, you want to write a *program* to analyse your data, saving your program in a text file. Once you've written it

- you can look it over carefully to make sure it's correct
- make changes to it without starting over again
- run it on as many data sets as you have
- share it with someone else

The ability to write readable and reliable programs is one big advantage of using R rather than less flexible analysis tools, such as spreadsheets.

Creating and Using R Scripts

One kind of R program is a text file containing R commands that you can ask R to perform — much as if you had typed them at the R prompt. This kind of program is called an R *script*.

You can create an R script with whatever your favourite text editor is (but **not** with a word processor, unless you save the document in `.txt` format).

RStudio has a built-in text editor, which may be the most convenient one to use.

Once you've created a script, you can get R to read it — and do the commands it contains — with the `source` function, giving it the name of the script file:

```
> source("myscript.r")
```

RStudio has a button you can click to do this for a script created with its editor.

If you type the command yourself, you may have to use the full path to the file (such as `/Users/mary/myscript.r`).

If the script doesn't work as desired, you can change it in the editor (and save the new version), and then get R to do it again, until you have “debugged” it.

Example Script: Read Data, Compute its Mean and SD, Plot it

```
# Read a file of numbers from the course web page.

data <- scan ("http://www.cs.utoronto.ca/~radford/csc120/data2")

# Compute the sample mean and sample standard deviation of the data.

m <- mean(data)
s <- sd(data)

# Plot the data points, along with a horizontal line at the mean, two
# dashed lines at the mean plus and minus the standard deviation, and
# two dotted lines at mean plus and minus twice the standard deviation.

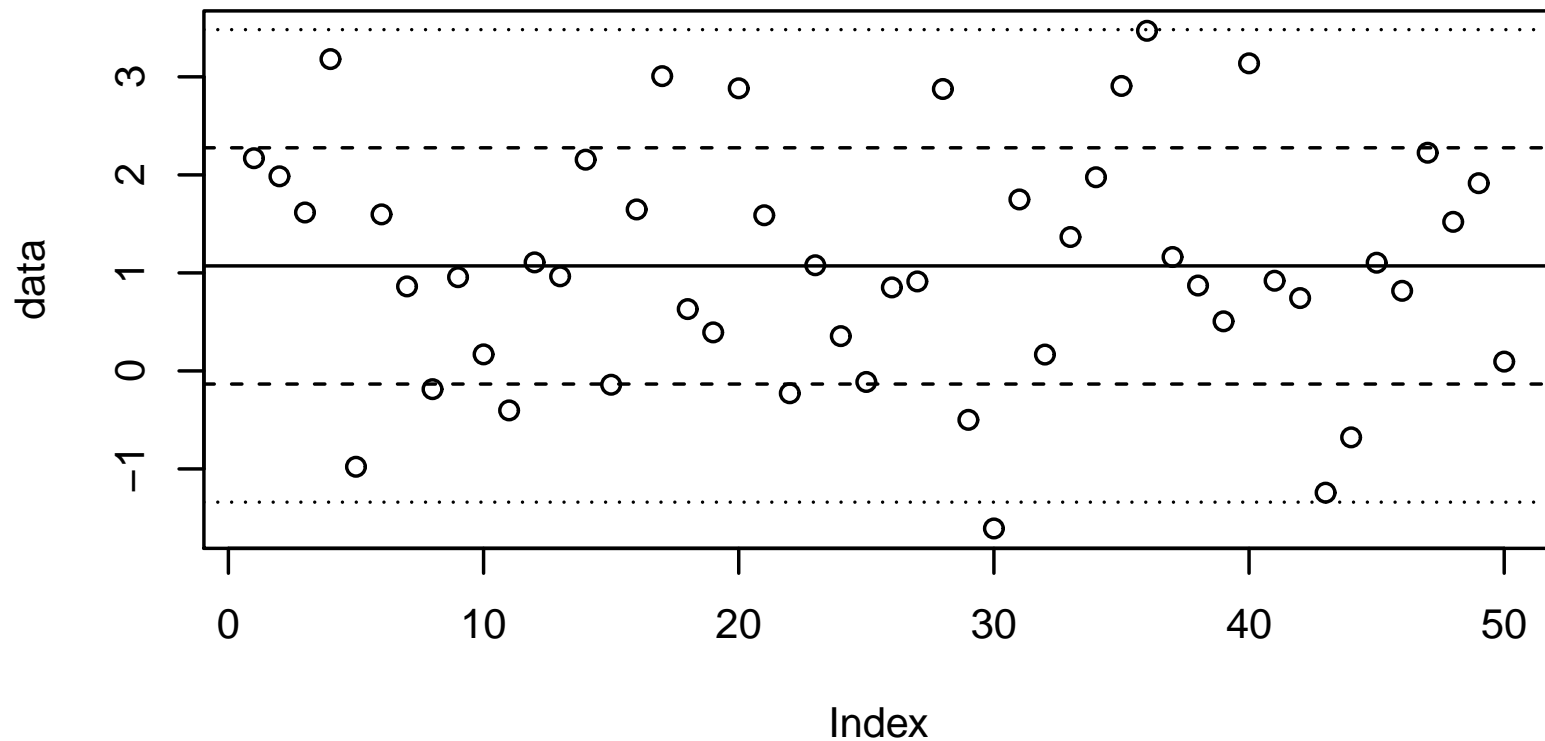
plot(data)

abline (h=m)
abline (h=c(m-s,m+s), lty="dashed")
abline (h=c(m-2*s,m+2*s), lty="dotted")
```

How to Run This Script, and the Plot It Produces

This script is stored in a file on the course web page. You can run scripts obtained from the internet using the URL rather than a file name:

```
> source("http://www.cs.utoronto.ca/~radford/csc120/demo-script2a.r")
```



Note: For security reasons, don't run a script from a URL at a website you don't trust! Instead, download the script and verify it's OK before running it.

Looking at the Variables Set in the Script

The variables used by the R script will still exist after it has run, and can be examined:

```
> source("http://www.cs.utoronto.ca/~radford/csc120/demo-script2a.r")
Read 50 items
> data
 [1]  2.170  1.985  1.616  3.181 -0.978  1.597  0.862 -0.186  0.956
[10]  0.169 -0.403  1.107  0.965  2.155 -0.141  1.647  3.007  0.631
[19]  0.393  2.883  1.588 -0.228  1.078  0.355 -0.113  0.852  0.913
[28]  2.876 -0.499 -1.607  1.749  0.167  1.366  1.976  2.907  3.470
[37]  1.162  0.871  0.506  3.138  0.920  0.743 -1.242 -0.678  1.104
[46]  0.817  2.226  1.521  1.915  0.095
> m
 [1] 1.07128
> s
 [1] 1.205056
```

Ways to Run an R Script

As was mentioned, you can run an R script in the file `myscript.r` with the command `source("myscript.r")`.

But this isn't quite the same as typing the contents of `myscript.r`. The commands in `myscript.r` aren't displayed, and you don't see the value of each expression. (Though the `print` function can be used to explicitly display values.)

If you want to see everything, much as if you had typed the commands, use

```
> source("myscript.r",echo=TRUE)
```

In RStudio, there is a button for sourcing the script being edited, with an option for whether `echo=TRUE`.

You can run a script non-interactively (plots going to the file `Rplots.pdf`) with the Unix/Linux command

```
Rscript myscript.r
```

Later, we'll see how to run scripts and get pretty output using the `spin` function in the `knitr` package.

Uses and Limitations of Scripts

R scripts are a good way to do *one* thing — such as produce output and plots from analysing one data set in one way. The script helps document exactly what you did for later reference.

But a script isn't a good way of doing many things, for instance:

- analysing several different data sets, or
- varying the way that the data is analysed

For example, the source of the dataset is fixed in the R script shown earlier.

It's also not very convenient to take the output of an R script and do more with it.

It is possible to change what a script does by setting variables before you run the script. And the script can set variables to values that can be looked at later.

But there is a better way to write programs that can do many things, and be used as part of a larger program — using *functions*.

Programming by Defining Functions

An R *function* specifies how to compute an output — the *value* of the function — from one or more inputs — the *arguments* (or *parameters*) of the function.

Within a function, the arguments are referred to by their names. Each time the function is used (“called”), values for the arguments are specified, and the argument names will refer to those values during that use of the function.

The next time the function is called, the arguments may have different values.

A function will compute a value from its arguments. When the arguments are different, in a different call of the function, the value may also be different.

The value computed by a function call can be assigned to a variable, or used in arithmetic, just like for R’s built-in functions like `log` and `sin`.

When defining a function, you can make use of other functions you have defined. In this way, large, complex programs can be built from simpler parts, which helps make them easier to understand.

Defining and Using a Simple Function

Let's define a function called `sin_deg` that computes the sine of an angle specified in degrees, rather than in radians (as for `sin`):

```
> sin_deg <- function (angle) sin(angle*pi/180)
```

This sets the variable `sin_deg` to the function specified by the expression `function (angle) sin(angle*180/pi)`, in the same way we can set a variable to a number or a string. This function has one argument, which is referred to by the name `angle`. The value of the function is computed as `sin(angle*180/pi)`. (The variable `pi` is pre-defined by R as $\pi = 3.14159\dots$)

We can then use this function just like we can use R's built-in functions:

```
> sin_deg(30)
[1] 0.5
> sin_deg(45)
[1] 0.7071068
> 100 + sin_deg(90)
[1] 101
```

Within `sin_deg`, the argument named `angle` will have values 30, 45, and 90 for the three uses of `sin_deg` above.

A Function With Two Vector Arguments

Functions can have more than one argument, and the arguments can be vectors rather than single numbers.

Here's a function that computes the distance between two points in a plane, with each point specified by a vector of two coordinates:

```
> distance <- function (a,b) sqrt ((a[1]-b[1])^2 + (a[2]-b[2])^2)
```

Within this function, the two arguments (the points we want to compute the distance between) are referred to by the names `a` and `b`.

Here are some uses of this function:

```
> distance (c(1,2), c(4,-2))  
[1] 5  
> x <- c(1.3,2.4)  
> y <- x + c(1,1)  
> distance(x,y)  
[1] 1.414214
```

Defining a Function With Several Steps

In the examples we've just looked at, the functions were simple enough that their value could be computed with a single expression that wasn't too complex.

For complicated functions, it can be convenient to break the computation into several steps, enclosed in curly brackets (`{` and `}`). The early steps assign values to variables, which are used in the last step to compute the final value of the function.

Here is a version of `distance` function re-written in this way:

```
distance <- function (a,b) {  
  diff1 <- a[1] - b[1]  
  diff2 <- a[2] - b[2]  
  sqrt (diff1^2 + diff2^2)  
}
```

In this example, the steps inside the function definition are *indented* by four spaces, so that it's easier to see that they are part of the `distance` function. This is a good practice, which you should follow.

Computing the Perimeter of a Diamond

Here's a function that computes the total length of the four sides of a diamond that has widths `width1` and `width2` for its two axes:

```
diamond_perimeter <- function (width1, width2) {  
  
  vertex1 <- c(width1/2,0)  
  vertex2 <- c(0,width2/2)  
  vertex3 <- c(-width1/2,0)  
  vertex4 <- c(0,-width2/2)  
  
  ( distance(vertex1,vertex2) +  
    distance(vertex2,vertex3) +  
    distance(vertex3,vertex4) +  
    distance(vertex4,vertex1) )  
  
}
```

(As you may realize, the four sides are actually all the same length, so this could be simplified — but we'll pretend we don't realize that for this example.)

Using Functions Defined in a File from an R Script

We usually don't type functions into R — they're inconveniently long, and we may wish to change them without having to re-type everything.

Instead, we store the definitions in a file, just as for an R script.

When we want to use these functions in an R script, we use `source` at the start of the script to read these functions definitions into R.

For example, we could put the definitions of `distance` and `diamond_perimeter` into a file called `distfuns.r`, and then use these functions in a script as follows:

```
source("distfuns.r")
big_diamond_perim <- diamond_perimeter (12.1, 4.7)
small_diamond_perim <- diamond_perimeter (0.4, 0.9)
print(big_diamond_perim)
print(small_diamond_perim)
```

By putting the definitions of these function in a file separate from the script that uses them, we can easily use the same functions in other scripts as well.

Functions that Do Things

The purpose of the functions in the previous examples is to compute some output value from the inputs given as arguments. We can also define functions whose purpose is to *do* something, instead of (or in addition to) computing something.

Here's an example:

```
> parrot <- function () {  
+   what <- readLines (n=1)  
+   cat (what, what, what, "\n")  
+ }
```

This function has no arguments, and produces no value as output. It just reads a line of text typed by the user, and then prints it three times (followed by an end-of-line marker, which is written as "\n").

For example:

```
> parrot()  
Hello!  
Hello! Hello! Hello!
```


Example: Plotting Data with Mean and SD

```
# PLOT DATA VECTOR SHOWING MEAN AND STANDARD DEVIATION. Plots a vector  
# of data points, along with horizontal lines showing the mean (solid  
# line), the mean +/- sd (dashed lines), and the mean +/- 2*sd (dotted  
# lines). The single argument must be a numeric vector. No return value.
```

```
plot_showing_mean_sd <- function (data) {  
  
  m <- mean(data)  
  s <- sd(data)  
  
  plot(data)  
  
  abline (h=m)  
  abline (h=c(m-s,m+s), lty="dashed")  
  abline (h=c(m-2*s,m+2*s), lty="dotted")  
}
```

Using this Function in a Script

We can put this function definition in a script called `demo-funs2b.r`, and then use it (twice) in another script, `demo-script2b.r`, which starts by reading in the script that defines the function;

```
source("http://www.cs.utoronto.ca/~radford/csc120/demo-funs2b.r")

par(mfrow=c(1,2)) # Put two plots side-by-side

data1 <- scan ("http://www.cs.utoronto.ca/~radford/csc120/data1")
plot_showing_mean_sd(data1)

data2 <- scan ("http://www.cs.utoronto.ca/~radford/csc120/data2")
plot_showing_mean_sd(data2)
```

We can run this script by

```
source("http://www.cs.utoronto.ca/~radford/csc120/demo-script2b.r")
```

Try it and see what you get!

Better yet, download both scripts, change them a bit to do something else, and try that.