How-To-R: A tutorial series on coding, and data analysis for Biologists

Part 2: Manipulating Data

OKR

2025-08-12

Loosely adapted from "Introductory R: A Beginner's Guide to Data Visualisation, Statistical Analysis and Programming in R" by Robert Knell

(2.0) Working with data

(2.1) Keeping track with R scripts

In Part 1, we directly input commands into the console and submit them with **ENTER**. We can retrieve commands via console by pressing the "up" direction key while within the console, useful to make quick adjustments to recent input.
With larger and longer projects, we can use **Scripts** to keep track of, and draft commands; as well as leave comments on our code. Well commented scripts are also a good way to share your analysis methods among peers.

You can start a new **R Script** a few ways. Look at the top left of RStudio:
1. File -> New File -> R Script **OR**
2. The white paper and plus icon just below file **OR**
3. CTRL + Shift + N

The **script** now takes the top left portion of the interface, pushing **console** to bottom left. We will be looking at both left side windows very often, the script for our commands, and the console for our output.

Let's test the script with a simple calculation. To submit commands from scripts, we click a *line of code* and press **CTRL + ENTER** instead. We can also **selectively highlight** (click(hold) + drag) specific parts of code.

```
2+2
```

```
## [1] 4
```

You should see the output in the console. Your command also remains on the script! We should leave comments on the script to remind ourselves of purposes of code, or adjustments made for review. But we cannot simply write (naked) sentences like these. Add a "#" in front of comments. Any text after "#" is ignored in terms of running code.

```
#2+2
```

The console outputs the commented line as is, and does not compute it.
Do use comments while learning R to explain, in your own words, what the code does!
Perhaps you can start by adding 3 lines of comments at the beginning of your script noting today's tutorial, your name, and today's date?

(2.2) Manual data entry

We'll now try handling some real-looking data.
Let's say you ran an experiment investigating how phage incubation duration affects the amount of virions released when a cell bursts. We have a set of observations from a single series taken at 13 time points.
Youve infected cells with the phage, experimentally delayed their lysis until a predetermined incubation period (in minutes), and then induced lysis and measured the average particles produced. Below is what the data might look like.

Table 1: Phage burst size vs accumulation duration

Record	Time to lysis (min)	Mean phage virions
1	40	18
2	50	24
3	60	392
4	80	192
5	90	440
6	100	534
7	120	432
8	150	1048
9	180	1240
10	210	1200
11	240	3944
12	300	2544
13	360	1296

Firstly, we need to get this data into R. We could run the command directly in console, but let's use our script to keep track.

```
lysis.time <- c(40, 50, 60, 80, 90, 100, 120, 150, 180, 210, 240, 300, 360)
```

We created a set of numbers (a **vector**) and assigned it to a *lysis.time* object. the "c" before parentheses "()" tells R to concatenate everything in between the brackets as a single object. Call the object to check that it worked.

```
lysis.time
```

```
##  [1]  40  50  60  80  90 100 120 150 180 210 240 300 360
```

Now do the same for the virion counts. Do check objects as you make them.

```
burst.size <- c(18, 24, 392, 192, 440, 534, 432, 1048, 1240, 1200, 3944, 2544, 1296)
```

Now we have our data entered, it is good idea to visualise the data before processing it. Ask R to **plot** the data. This command should get you a scatterplot in the 'Plots' tab in the bottom right window.

```
plot(lysis.time, burst.size)
```

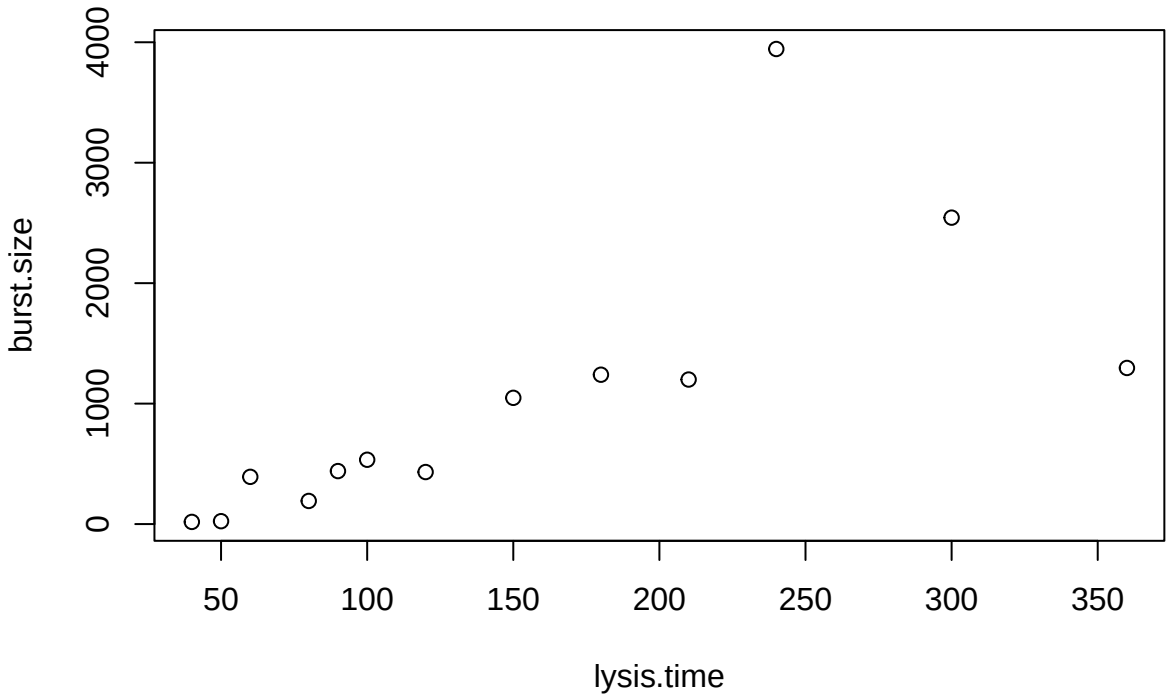

Figure 1: Figure 1: example of scatterplot

A basic plot. Not quite publication quality, but still useful! we can see:
1. the relationship of our variables. looks like burst size is positively correlated with lysis time
2. there are no terribly obvious outliers that might affect analysis, otherwise we would want to remove them before proceeding

If we are happy with our data, we can start proper analysis.
We could **visually** see a positive trend in our data, but we should provide **statistical proof** of the correlation. Specifically we want the correlation coefficient r for the two variables, and to know if r is **statistically significant** from zero (we will cover what this means in a later tutorial). Try this command:

```
cor.test(lysis.time, burst.size)
```

```
##
##  Pearson's product-moment correlation
##
## data:  lysis.time and burst.size
## t = 3.4143, df = 11, p-value = 0.005781
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.2749927 0.9090190
```

```
## sample estimates:
##       cor
## 0.7172956
```

What does this output tell you?

1. The name and kind of test done
2. Names of variables you provided
3. Test statistic (t = 3.414) and p-value (p = 0.005)
4. Correlation coefficient estimate ($r = 0.7172$)

Many tests in R generate output like this. At a minimum you will want the estimate and p-value, but the output should have all the info you'll need to report.

(2.3) Functions

A **Function** is a piece of ready-made code that carries out *usually* a specific set of tasks. We already used a few functions just now:
c() is a function that combines values into a vector
plot() is a function that draws points in a diagram
cor.test() is a function that tests for association between paired samples

From the start, R comes with a set of functions pre-installed. We can briefly view a list of these **base R functions** here:

```
library(help = "base") # opens a new tab with the list of functions available
```

each function name ends with brackets, and usually you put the name of the object you want processed within them. Try these:

```
log(50) #the natural log of 27
exp(7) #e raised to the power of 7
sqrt(25) #square-root of 25
abs(-4) #Absolute (i.e. unsigned) value of -4
sin(45) #sine of 45
```

Functions *usually* let you control/modify specific aspects of the operation by providing **arguments**. for example, the *round()* function will work with just a value, it will return an integer.

```
round(25.1234567)
```

```
## [1] 25
```

but we can also round to specific decimal places by using the **argument** "digits"...

```
round(25.1234567, digits = 2)
```

```
## [1] 25.12
```

another example

```
logb(4)
```

```
## [1] 1.386294
```

```
logb(4, base = 2)
```

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

If we dont provide a value for arguments like "base", functions have a **default** value to use. *Published* functions come with a **help page**, it is always worth it to read them, at least a little bit. try running the command *?round()*, it details several functions of similar purpose, and the arguments they use in common.

(2.4) Exploring Vectors

Previously we made vectors by manually entering many values in a comma-separated sequence and combined them with c() into an object. Let's make vectors using functions this time.

```
vec1 <- seq(from = 1, to = 100, by = 2)
```

Have a look at what *vec1* is, can you figure it out what **seq()** does?
How about this function?

```
vec2 <- rnorm(n = 50, mean = 20, sd = 1)
```

Maybe visualising the data will help. try using *plot()* and *hist()* on vec1 and vec2. You can use the arrow icons in that window to return to past plots.

To better understand our vectors, we can use functions to **describe** them.

1. length() # how many elements/things in the vector
2. str() # simplified structure
3. summary() # general statistics

randomly generating values

Try running vec2 again (or compare your vec2 with anothers), are there differences? Many functions in R that **generate random values** will re-roll those values each time the function is called, giving different output.

There are many uses for this in statistics and probability, but in some cases (like when running a tutorial) we want consistency and predictability when running analyses. We have the option to **set a seed value** which will cause R to always give the same output when a randomiser function is used.

```
set.seed(18111992)
```

We can put any number here, it just means if we set this same seed, we will always get the same values. to remove seed we call **set.seed(NULL)**
now try making vec2 again, and again. it should not be changing anymore.

manipulating vectors

We can apply operations to whole vectors as well. Try to guess what will happen to the vec1 below before you run the command.

```
vec1 * 2
vec1 + vec2
vec1/vec2
```

When given a simple operation, it's applied to each element of the vector.
when given a vector, the operation is carried on *each element in turn* from all vectors.
So the 1st element in vec1 operates with the 1st element of vec2, and so on... What if the vectors are NOT the same length?

```
vec3 <- seq(from = 1, to = 100, by = 20)
vec1 + vec3
```

```
##  [1]   2  24  46  68  90  12  34  56  78 100  22  44  66  88 110  32  54  76  98
## [20] 120  42  64  86 108 130  52  74  96 118 140  62  84 106 128 150  72  94 116
## [39] 138 160  82 104 126 148 170  92 114 136 158 180
```

We can still add the *5-lengthed vec3* to the *50-lengthed vec1*. The shorter vector simply *cycles* back to its first element and continues.

(2.5) Exploring Matrices

A **Matrix** is a way to display 2-dimensional data (1 set of rows, 1 set of columns), and the contents, the **cells** of the matrix is only numbers.
The distinction between a matrix and a table isnt too important visually (as human-readable data), but many functions only take one format or the other. So it is useful to understand the structure when you need to transform data into a matrix.
For now, let's make a simple one.

```
mat1 <- matrix(data= seq(1,12), nrow = 3, ncol = 4,
dimnames = list(c("R1", "R2", "R3"), c("C1","C2", "C3", "C4")))
```

Don't be intimidated by the code above, we'll examine bit by bit, also remember you can check function help pages.
The function is called **matrix()**, the data we want to turn into a matrix is a **sequence from 1 to 12**, we want the matrix to have **3 rows** and **4 columns**, we provide the **names for our 2 dimensional matrix** as a **list** of **rows R1-R3** and **columns C1-C4**, we then save this as an object **mat1**

Notice that we provided a vector, and it filled the matrix column-by-column. Try adding the argument *byrow = TRUE* as a new object *matrows*, what happens?

similar to vectors, you can have R do math operations on matrices.

```
mat2 <- mat1/3
```

```
sqrt(mat1)
```

```
##          C1       C2       C3       C4
## R1 1.000000 2.000000 2.645751 3.162278
## R2 1.414214 2.236068 2.828427 3.316625
## R3 1.732051 2.449490 3.000000 3.464102
```

```
mat1 + mat2
```

```
##          C1       C2        C3       C4
## R1 1.333333 5.333333  9.333333 13.33333
## R2 2.666667 6.666667 10.666667 14.66667
## R3 4.000000 8.000000 12.000000 16.00000
```

Exploring Tables and Data Types

We should understand how R categorises vectors and types of data before we combine them in tables. So far we have only handled numbers: numerical, or quantitative data.There are many data types, functions can behave differently (or not work at all) when detects certain data types. For now, let's look at **Numeric/Integer**, **Character**,**Factor**, and **Logical** data types.

Numeric deals with numbers, integers, and decimals. this is what we have been working with so far. R sometimes called numeric data as **Double (dbl)**

```
vecX1 <- seq(1,12, by=1)
vecX2 <- 1:12
mode(vecX1)
```

```
## [1] "numeric"
```

```
mode(vecX2)
```

```
## [1] "numeric"
```

```
class(vecX1)
class(vecX2)
```

```
class(sqrt(vecX2))
```

Character stores non-numeric text data. You have to input them with quotation marks, otherwise R will interpret the words as object names and return an error when it can't find them. For example, you might record sample and patient names, addresses, etc.

```
txt1 <- c("Hello", "This", "Is", "A", "Character", "Vector")
txt1
```

```
## [1] "Hello"     "This"      "Is"        "A"         "Character" "Vector"
```

```
class(txt1)
```

```
## [1] "character"
```

Factors are a special kind of character data that suggests some level of organisation. For example, you might record whether your subjects are male or female, adult or juvenile, symptomatic or otherwise, etc.

```r
agegroup <- c("Adult", "Child", "Elderly", "Adult", "Elderly", "Adult")
agegroup
```

```
## [1] "Adult"   "Child"   "Elderly" "Adult"   "Elderly" "Adult"
```

```r
class(agegroup)
```

```
## [1] "character"
```

```r
ages <- factor(agegroup)
ages
```

```
## [1] Adult   Child   Elderly Adult   Elderly Adult
## Levels: Adult Child Elderly
```

```r
is.factor(agegroup)
```

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

```r
is.factor(ages)
```

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

Logical is another special kind of character type that specifically deals with data that is either 'true' or false', and so R accepts this as... "TRUE" or "FALSE" by default.

let's try putting together these data types in a single table. We first create the vectors, then attach them using *dataframe()*. For example, a medical dataset might look like:

```r
PatientNo <- c(1,2,3,4,5,6)
FirstName <- c("Aisyah", "Sarah", "Peter", "Xi Yin", "Kamarul", "John")
Sex <- as.factor(c("F", "F", "M", "F", "M", "M"))
Height_cm <- c(156, 169, 180, 175, 150, 200)
Weight_kg <- c(40, 50, 50, 80, 70, 80)
Exercise <- as.factor(c("rarely", "often", "sometimes", "often", "rarely", "sometimes"))
Married <- as.logical(c("TRUE", "FALSE", "FALSE", "TRUE", "FALSE", "FALSE"))

PatData <- data.frame(PatientNo, FirstName, Sex, Height_cm, Weight_kg, Exercise, Married)
```

Check the structure of the dataframe with *str()*.

(2.6) Saving Objects

We will go through how to manipulate tables in the next tutorial, for now let's **save/export** the table. By default, R will look at the **working directory** when you ask it to import and export objects. use *getwd()* to check the working directory, and optionally, you can use *setwd()* to change it, by providing a path.

The most common file type to use with R is **.csv**, there are functions for use with other formats which we can cover later. let's save our PatData file onto the computer.

the x is the object we want to save, *file* is the destination on the computer to save in (default is the working directory), you should attach the file format, in this case "*.csv*" to the file name, so your computer knows how to interpret it when you try to access.

```r
write.csv(x = PatData, file = "TutorialPt2Data.csv")
```

Go ahead and check that the csv file has been saved and looks correct, you can open it with a text editor (e.g. notepad), or spreadsheets (e.g. excel)

(2.7) Saving Scripts and workspace

This script is an important product of your work. You should save it (CTRL + Save), and give it an informative name. If you move this script around to another computer and open it in R, you can replicate this exact work. This kind of reproducible research will help greatly in sharing analysis methods with your peers.

Additionally, you can also save the workspace. this includes all the objects and vectors youve made (e.g. the top left window), such that next time you open R, you can have them at the start. This is a less common practice (and will not be used in these tutorials), since if you have the script you can choose to make everything again, but workspaces can be saved using the 'save icon' (floppy disk) in the top right window.

No exercises this tutorial, you can move onto the next one!

END of tutorial 2