Introduction to R Programming - University of York

Dr. Lewis Ramsden

Autumn Term (2022/23)

Contents

Preface

These lecture notes have been created as supplementary material for this course and are mostly built up from the R scripts seen in the workshops, as well as a few additional comments. At the end of each chapter, you will also find the exercise problems discussed in the workshop sessions; the solutions will be added as we progress through the course.

These notes are a work in progress and as such, will be updated throughout the duration of the course, so please make sure to revisit them on a regular basis. If there is anything missing from these notes that you believe would be beneficial or if you notice any mistakes, please let me know so I can improve them as best I can. Remember, they are here for your benefit so it would be great to have your input too.

About This Course

This course is by no means exhaustive and is designed to introduce you to the basics of programming in R, improving your confidence with coding and signposting you to additional resources for you to further enhance your skills.

The course is non-credit bearing and thus, there are no formal assessments for you to submit. However, at the end of each week/chapter there are a number of exercises for you to complete, which I strongly recommend you attempt. The best way to learn and improve you coding skills is by doing it yourself and learning how to overcome the obstacles/errors that you will inevitably encounter. Remember, do not be afraid to search the web for hints and ideas when programming, it is usually the most effective way to solve your problems, I have to do it on a daily basis!

Schedule

As this is a non-credit bearing course, the syllabus and schedule are flexible and can be delivered as we see fit. However, a rough schedule over the 6-week course is as follows:

1. RStudio and R basics (Revision)
2. Conditional statements and IF statements
3. FOR/WHILE loops
4. Functions
5. Creating, importing and analysing data
6. Creating documents in RMarkdown

DataCamp

In order to assist you in your journey to learning all about R and RStudio, this course is supplemented via an online interactive tutorial website known as DataCamp.

In DataCamp, you will have access to hundreds of interactive courses for R (and other languages such as Pyton and SQL), each tailored to a different aspect of the fundamentals of R programming or an area of application. In general, only a few of these courses are free to use, with the remaining requiring a paid subscription for access. However, for those of you sitting the short course on 'R programming' you will have free access to the full library of courses for 6 months from the start of the course. Registration for this free access will be discussed in the lecture itself and is only available to those invited by the lecturer via an email link. To learn more about DataCamp and what it has to offer visit https://www.datacamp.com.

As mentioned above, DataCamp will be used as a supplementary resource for this course and we strongly encourage you to use it. At the end of each chapter of these lecture notes, we will include links directing you to appropriate courses within DataCamp that we believe complement the material given.

Chapter 1

RStudio and R Basics (Revision)

R is a language and programming environment for statistical computing and graphics (graphs and plots), which offers an 'Open Source' (freely available) alternative for implementation of the S language, which is the usual language of choice when it comes to statistical computing. In other words, R is a freely available software environment which runs on Windows, MacOS and LINUX, that allows the user to conduct mathematical calculations, data manipulation, statistical computations and create graphical output.

RStudio is known as an *integrated development environment* (IDE) for R, which essentially provides much more user friendly access to R and its features. The figures below show the two environments separately. The first is the original R environment and the second is RStudio. Even from these simple graphics, you can see that RStudio provides a much more detailed user face, with a number of different 'panels' (discussed in more details later) for a range of different commands.

1.1 How to install R and RStudio

Installing R and RStudio has to be done in two separate steps:

1. Firstly, we need to install the original R software for your specific operating system (Windows, Mac or LINUX) from https://cran.ma.imperial.ac.uk/. Once this is installed, you are able to open R and you should be met with a screen similar to the left hand side in Figure:1.1, above. At this point, you are now able to use R and all its features completely. However, as mentioned in the previous section, it is usually preferable to work with RStudio due to its user friendly interface.

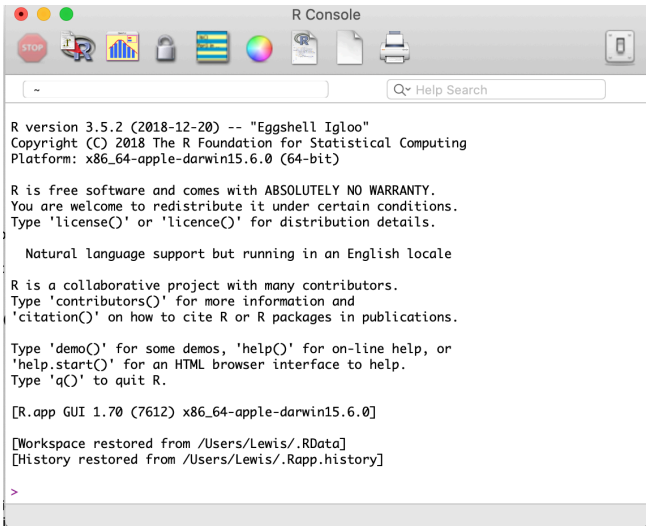


Figure 1.1: R Environment.

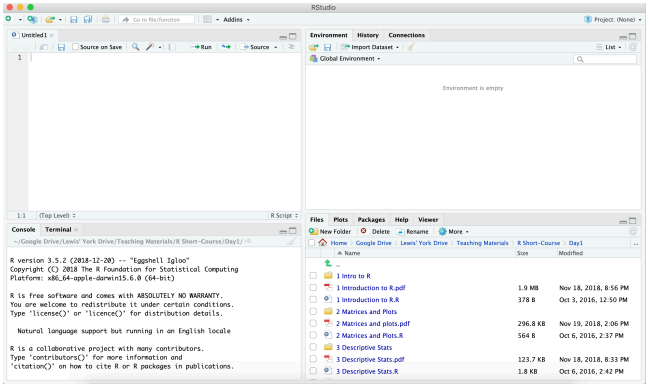


Figure 1.2: RStudio IDE.

2. To download the free version of RStudio, visit [https://rstudio.com/products/rstudio/download/] and download 'RStudio Desktop (Free)'. Once downloaded, you will be able to open RStudio and should see a similar screen to that of Figure:1.2. Keep in mind that the image(s) above may be running older versions than the one you are using. Once you have downloaded RStudio, I recommend you only ever use R through this platform, so there is no need to open the original R software.

Note that in order to use R through the RStudio environment, you must first download the original R software. However, you can use R without downloading RStudio but I do not recommend this!

If you are using a university computer, you do not have to worry about the steps above as R and RStudio are already installed and can be found within the list of installed in programmes.

1.2 RStudio interface explained

When you open RStudio, you will notice that the environment has a number of different 'panels'. You may find that your environment looks slightly different to the one in the figure above and may only have one larger panel on the left hand side rather than two separate ones. This difference will be explained later. To avoid confusion, your screen should look like the figure given below.

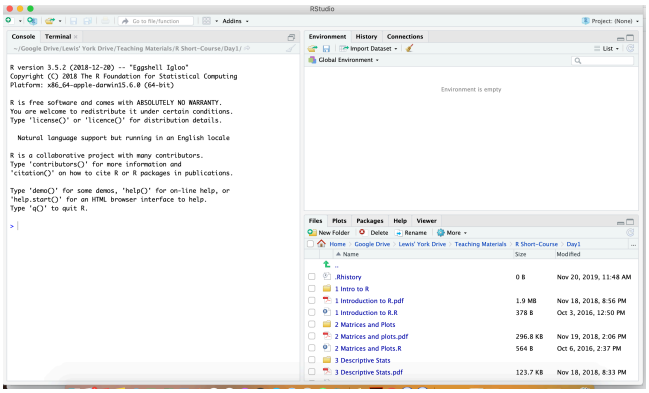


Figure 1.3: Orginal RStudio View.

Let us discuss each of the panels and some of their associated tabs, in a little more detail:

- **Console (Left panel)** - The console is the panel you will interact with the most, as this is where you can type commands which can be 'Run' to produce some output.
- **Environment (Top panel, Tab 1)** - The environment tab lists all active

objects that have been 'assigned' (see below) and stored for later use. This is especially helpful when writing a long programme for which many variables need to be stored, as it allows you to refer back to previously defined objects.

- **History (Top right panel, Tab 2)** - The history tab shows a list of all commands that have been run within the console so far. Again, this can prove useful when writing long programmes which may require re-use of certain commands or to double check what has already been run.

- **Files (Bottom right panel, Tab 1)** - The file tab shows the folder of your 'Working Directory'. That is, the folder in which R is directed to look for data sets etc. This tab looks similar to the equivalent folder in your PC/Mac folder window.

- **Plots (Bottom right panel, Tab 2)** - The plots tabs allows you to view all of the graphs/plots you have created within that session. This proves helpful when you want to compare a number of plots.

- **Packages (Bottom right panel, Tab 3)** - The packages tab provides access to a list of 'Packages' or 'Add-ons' needed to run certain functions. When RStudio is first started up, it will only have access to its basic packages which contain fundamental functions and tools. In order to conduct more sophisticated analysis or calculations it is usually required for you to install an extra package which contains these tools.

- **Help (Bottom right panel, Tab 4)** - The help tab can be used to find additional information about certain functions, tools or commands within RStudio. You will find this to be a very important part of your programming experience and will be used constantly. We will discuss later on how to access help via a shortcut through the console.

1.3 Mathematical calculations

Now that we understand a little more about the setup of R and RStudio, we want to discuss what we can actually do in R. As previously mentioned, R is most notably used to conduct mathematical calculations, data manipulation, statistical computations and create graphical output but let us discuss each of these in a little more detail and give some practical examples you can try for yourself.

In its most basic form, R can be used as a large scale calculator. In contrast to an actual calculator, it can perform a variety of calculations quickly and easily, which would otherwise take a great deal of time, e.g. series summations and matrix multiplcation. In fact, there are many calculations that can be performed in R which would not be possible even with a scientific calculator.

1.3.1 Basic numerical calculations

If you simply type `5*3` into the 'console' (see above) and press enter you should receive the solution as an output which again appears in the console below your input:

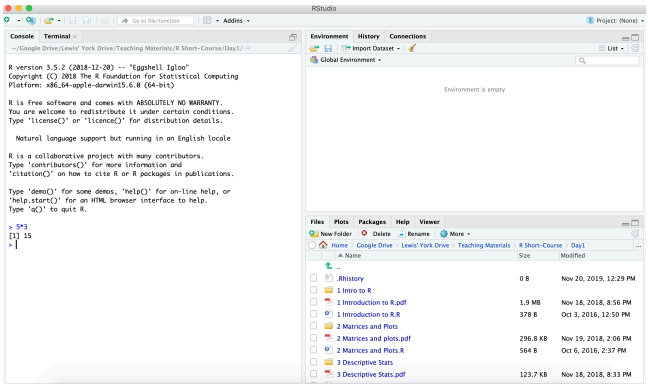


Figure 1.4: Basic Multiplication.

In a similar way, you can perform a variety of other basic mathematical calculations:

```
7+3
```

```
## [1] 10
```

```
9/3
```

```
## [1] 3
```

```
15-2
```

```
## [1] 13
```

```
6^3
```

```
## [1] 216
```

```
sqrt(100)
```

```
## [1] 10
```

Notice that some calculations, like the square root above, require knowledge of certain 'functions' e.g. `sqrt()` of which there exists hundreds in R's base packages for you to use. Knowing what each of them are and how they work is part of the programming experience and will take time. We will talk more about 'functions', and how you can create your own in a later chapter.

Some other useful examples of pre-defined variables and functions are `pi`, `exp()` and `log()` which allow use of π, e^x, $\ln(x)$ in calculations, respectively. For

example, if we wanted to calculate $e^{\ln(\pi)}$:

```
exp(log(pi))
```

```
## [1] 3.141593
```

1.3.2 More complicated calculations

Imagine that you want to find the sum of all the integers from 1 to 1000. To do this on a basic calculator would require you to physically type each integer in turn, adding them as you go along (assuming you do not know the series summation formula). However, in R, you can compute this with one simple function, i.e. `sum()` with argument `1:1000`, which creates the sequence of numbers from 1 to 1000. To see this in action before performing this particular calculation, type `1:10` in the console and press enter:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

As you can see, the output is the required sequence of integers from 1 to 10. Returning to our original summation calculation, by inputting `sum(1:1000)` and pressing enter, R will first create the sequence from 1 to 1000 in a similar to above, then sum all of these values:

```
sum(1:1000)
```

```
## [1] 500500
```

Before we move on to discuss more advanced calculations that can be handled with R, let us take a moment to highlight the disadvantages of writing code directly in the console itself and introduce something known as an 'R script'. In addition, we will also discuss how we can 'assign' values to variables which we can then recall at any point for use in calculation.

1.4 R script

So far, we have executed each line of code directly into the console itself, one line at a time, pressing enter and producing output each time. Although this works and produces the necessary output, it has numerous disadvantages. Firstly, if you make a mistake in the line of code, you cannot simply amend it and re-run it. Instead, you have to 're-type' the code again on a new line without the mistake. Secondly, it requires you to execute every line of code once you have completed it. If you are writing a programme with hundreds of lines, this will become very frustrating especially if something goes wrong half way through and you have to re-write the entire code again. Finally, you cannot easily save your written code within the console to be re-opened and continued at a later date. In order to avoid all of these problems, from now on we type all of our code into an 'R script', from which we can execute the code into the console.

To open an R script, click the icon which looks like a blank piece of paper with the small green plus sign in the top right hand corner of your screen, then click R Script:

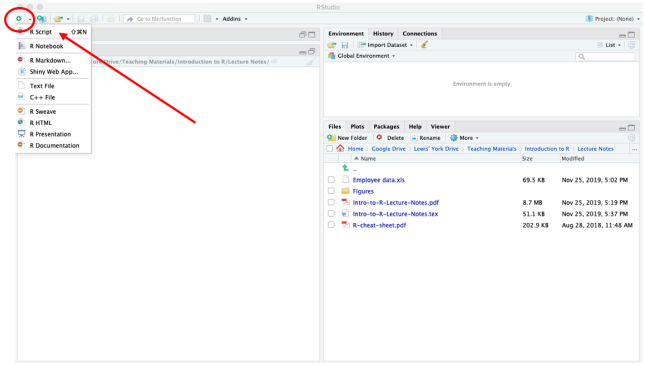


Figure 1.5: Opening an R script.

At this point, a new (blank) panel should open in the top left of your screen. This panel will now become the panel which you type all of your code (you no longer type into the console panel). Once you have typed your line of code, you can `execute` it (run it into the console) by simply highlighting the relevant code then clicking on the `Run` button as seen in Figure:@ref{fig:Script2} below.

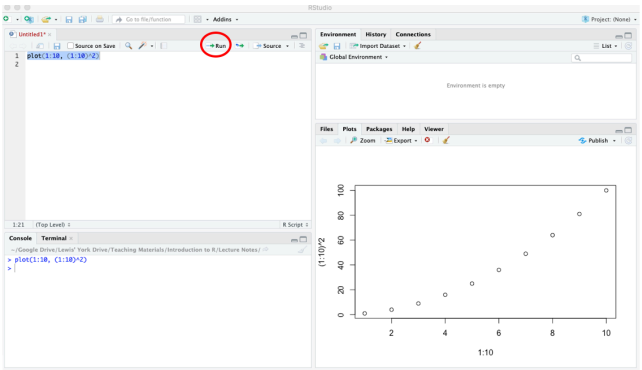


Figure 1.6: Executing code from a script in R.

Note you can also simply go to the start or end of the line and press `Run`, you do not actually need to highlight it. This is only necessary if you want to `Run` more than one line at a time.

By executing code from the script, you avoid all the previously discussed problems. That is, if you have a made a mistake in your code, which you will notice once executed, you now simply amend this in the script and re-run it which is

much simpler than re-writing the entire code. Moreover, you do not actually have to execute any code until you desire. Think of the script as a notebook which you can keep typing in and can run code from whenever you wish. Finally, and most importantly, you can save the script file and re-open this at a later date to continue working on and/or send to a colleague. You do this in the normal way as if saving a standard document in Word or other software.

1.5 Assigning variables

Recall the earlier example where we calculated the sum of values from 1 to 1000. Although relatively straight forward, typing this code out each time we would like to use the result becomes tedious and is, in fact, unnecessary in R. Instead, R allows us to 'assign' a value, vector, matrix, function etc., to a variable so we can recall that particular quantity at any point by simply typing the variable itself. For example, instead of repeatedly typing `sum(1:1000)` or the result itself `{r} sum(1:1000)`, we could 'assigned' this to the variable x using the 'assignment operator' `<-`, which allowed us to reuse the value later on by simply typing x:

```
x <- sum(1:1000)
x
```

```
## [1] 500500
```

Note, however, that when we used the assignment operator it did not print the output itself, which would have happened if we had simply ran the code without assignment. This is the reason we then typed the variable x in the next line of the console, as this will now `print` as output whatever quantity is saved to the variable x, in this case the sum of values from 1 to 1000. If you would actualy like to do both things at the same time, i.e, assign and print the output, you should put the assignment code in brackets:

```
(x <- sum(1:1000))
```

```
## [1] 500500
```

```
x
```

```
## [1] 500500
```

Finally, when a variable is assigned, the variable name and the type of quantity that has been assigned to it, will be stored in the 'environment' tab/panel. In this case, the variable x was assigned and the quantity assigned to them takes the form of a 'numeric' (num) value.

1.6 Vectors and Matrices

As we have already briefly seen within the summation calculations above, R can also easily create collections of values in a single object, known as a vector, which can then be used in a variety of calculations, including vector and/or matrix type calculations themselves.

1.6.1 Vectors

There are in fact a number of different ways to create 'vectors' of values in R, so let us discuss some of the most common.

1. The most general way is to use the 'combine' or 'concatenate' function `c()`. This function combines a series of individual values and then glues them together to form a vector

```
c(1, 2, 5, 9, 15)
```

```
## [1]  1  2  5  9 15
```

```
c(-3, 3, -1, 0, 10, 5, 2, -100, 25)
```

```
## [1]   -3    3   -1    0   10    5    2 -100   25
```

Although this is the most general method, it does require you to type out each value individually, not ideal if you want a vector containing 1000+ values.

2. We have already seen another example of how to create a vector using the semi-colon syntax `1:1000`. However, this is quite specific and only works for creating vectors which form a series of increasing/decreasing values:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
20:5
```

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

The more general version of this method is to create a 'sequence' of values with an initial starting point, an end value and specifying the increments between the values:

```
seq(from=5, to=50, by = 5)
```

```
##  [1]  5 10 15 20 25 30 35 40 45 50
```

3. The third way requires a little more thought and experience but will become second nature once you get going. It relies on you understanding how R deals with vectors in calculations, which you can then take advantage of (see below).

1.6.2 Vector calculations

Using vectors in calculations is just as simple as with scalar values, but will not necessarily produce the output you might first expect in some cases. Let us start by looking at some simply addition and subtraction of vectors which we assign as different variables:

```
a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
b <- 11:20
a+b
```

```
##  [1] 12 14 16 18 20 22 24 26 28 30
```

```
b-a
```

```
##  [1] 10 10 10 10 10 10 10 10 10 10
```

Notice that the calculations in the above have been done 'element-wise'. This is a very important observation as it is a characteristic of R vector calculations that will come in handy throughout your coding life and should be utilised as much as possible. Let us look at a few more examples:

```
a*b
```

```
##  [1]  11  24  39  56  75  96 119 144 171 200
```

```
b/a
```

```
##  [1] 11.000000  6.000000  4.333333  3.500000  3.000000  2.666667  2.428571
##  [8]  2.250000  2.111111  2.000000
```

```
a^2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
a^b
```

```
##  [1] 1.000000e+00 4.096000e+03 1.594323e+06 2.684355e+08 3.051758e+10
##  [6] 2.821110e+12 2.326305e+14 1.801440e+16 1.350852e+18 1.000000e+20
```

Once again, these have all been calculated element-wise! What happens if the vectors are not of the same length? In this case, R will automatically loop around the shorter vector and start using the values again from the start until it has used enough to match the length of the second vector. Let us take a look at a quick example to see how this works in practice:

```
vec1 <- c(1,2,3,4)
vec2 <- c(1,2,3,4,5,6,7)
length(vec1)
```

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

```
length(vec2)
```

```
## [1] 7
```

```
vec1 + vec2
```

```
## Warning in vec1 + vec2: longer object length is not a multiple of shorter object
## length
```

```
## [1]  2  4  6  8  6  8 10
```

This is a perfect example of why you need to be very careful when writing code. Just because you have (possibly) made a mistake, R will not always realise you have and execute a calculation anyway.

1.6.3 Vector strings

R is not all about numerical values. As it is a tool for statistical analysis, data can come in many shapes and sizes including words (known in R as character strings) or logical values, i.e, `TRUE` or `FALSE`. We will talk more about the latter values in the next few weeks but it is worth discussing 'string' here.

A 'character string' is simply a word or combination of letters that you would like R to understand as such. To create or include a string, you need to use quotation marks:

```
"Hello World"
```

```
## [1] "Hello World"
```

Once you put quotation marks around something, R automatically recognises this as a string and will not try to perform and type of operation to this. This is even possible with numerical values:

```
"10 is a numerical value"
```

```
## [1] "10 is a numerical value"
```

As a small example, try adding together the strings "10" and "11" in R. Notice that because we have defined the values as strings, R cannot perform addition with them:

```
str("10")
```

```
##  chr "10"
```

```
str(10)
```

```
##  num 10
```

In exactly the same way as we have seen above, it is possible to create vectors of strings. This is very helpful when you want to name a bunch of objects, rows/columns in data tables or when they represent data points themselves, e.g., geographical regions etc.

```
c("York", "London", "Liverpool", "Birmingham")
```

```
## [1] "York"       "London"     "Liverpool"  "Birmingham"
```

1.6.4 Vector extraction

One final tool of note for vectors is the method of extracting certain values. For example, let us again consider the vector of values from 1 to 1000. Now assume that you want to 'extract' the first 10 values from this vector, i.e. the values 1 to 10. To extract values from a vector, you can use square brackets `[]` immediately after the brackets to inform R of which elements you want to extract:

```
x <- 1:1000
x[1:10] # Note that this extracts the first 10 elements, not the elements with value 1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
y <- seq(from = 10, to = 20, by = 0.5)
y[1:10]
```

```
##  [1] 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5
```

```
z <- y[c(1,3,5,7,9)] # This extracts the 1st, 3rd, 5th, 7th and 9th elements
z
```

```
## [1] 10 11 12 13 14
```

```
y[-(1:10)] # The negative sign means extract everything except the specified elements
```

```
##  [1] 15.0 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0
```

Notice the comments in the above code? This can be done using the hashtag symbol and is a habit I would strongly recommend you start to implement from the off. I have given more information about this in the supplementary chapter (Additional Tips) at the end of these notes.

To give you a little more context to how/where this might be helpful, take a look at the following simple example about with respect to heights of individuals in a given classroom:

Example 1.1. Assume that the height (in cm) of a 80 individuals in a given classroom were measured and recorded in the variable `height_data` given below:

```
height_data
```

```
##  [1] 175.9109 148.9805 173.7578 164.0041 176.0585 175.5764 189.5720 165.2801
##  [9] 159.4315 145.2800 149.0911 176.8081 165.4725 161.4941 173.8868 144.3101
## [17] 159.3412 158.2476 179.3953 164.3868 191.3619 142.8295 151.7678 138.8184
## [25] 192.7071 155.3275 181.9161 139.5167 173.4458 179.0451 146.9677 171.2390
## [33] 178.7697 148.8816 174.7356 177.6438 170.9471 171.6344 161.5625 163.7828
```

```
## [41] 162.2952 198.7534 162.1619 167.3586 154.7708 155.6980 170.0592 176.8990
## [49] 201.4863 154.0795 165.9555 180.3997 174.8742 150.0994 176.5720 151.4947
## [57] 184.5251 158.3298 146.0928 166.3630 170.2236 160.8860 191.4309 148.1925
## [65] 191.1103 166.8840 161.9347 168.8053 175.4741 169.0576 141.3637 160.2115
## [73] 182.6158 168.3202 178.0508 186.1188 168.0577 170.0972 157.0554 169.8845
```

Now assume that we wanted to find out the average height of the 20 smallest individuals in the classroom:

```
(height_sorted <- sort(height_data))
```

```
##  [1] 138.8184 139.5167 141.3637 142.8295 144.3101 145.2800 146.0928 146.9677
##  [9] 148.1925 148.8816 148.9805 149.0911 150.0994 151.4947 151.7678 154.0795
## [17] 154.7708 155.3275 155.6980 157.0554 158.2476 158.3298 159.3412 159.4315
## [25] 160.2115 160.8860 161.4941 161.5625 161.9347 162.1619 162.2952 163.7828
## [33] 164.0041 164.3868 165.2801 165.4725 165.9555 166.3630 166.8840 167.3586
## [41] 168.0577 168.3202 168.8053 169.0576 169.8845 170.0592 170.0972 170.2236
## [49] 170.9471 171.2390 171.6344 173.4458 173.7578 173.8868 174.7356 174.8742
## [57] 175.4741 175.5764 175.9109 176.0585 176.5720 176.8081 176.8990 177.6438
## [65] 178.0508 178.7697 179.0451 179.3953 180.3997 181.9161 182.6158 184.5251
## [73] 186.1188 189.5720 191.1103 191.3619 191.4309 192.7071 198.7534 201.4863
```

```
smallest.20 <- height_sorted[1:20]
mean(smallest.20)
```

```
## [1] 148.5309
```

1.6.5 Exercises

1. In R, create two vectors containing the numbers (5, 6, 7, 8) and (2, 3, 4). Assign these vectors to the variables u and v respectively.

i. Without using R, write down or think about what you expect the following results to produce:

a. add u and v
b. subtract v from u
c. multiply u by v
d. divide u by v
e. raise u to the power of v

ii. Using R, check if your initial thoughts were correct.

2. Create the vector of values $(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5)$ using the following two methods:

i. Using the `seq()` function
ii. Using only the semi-colon syntax and element-wise calculations.

3. Use R to create a vector containing all the square numbers from 1 up to and including 10,000 (100^2).

4. The vectors `LETTERS` and `letters` are already pre-built into R's base-package and contain the capital and lower-case versions of the letters from the English alphabet (Try it by simply running either `LETTERS` or `letters` into R).

i. Create a vector containing the first 10 letters of the English alphabet in Capitals.
ii. Now, using your solution from the previous part, create a new vector of the form:

$$(A, A, A, B, B, B, C, C, C, \ldots, J, J, J).$$

[**Hint: Try looking into the `rep()` function and how it works**]

1.6.6 Matrices

In the previous section, we discussed how to create vectors of values. However, R can just as easily deal with matrices and matrix calculations; a life-saver compared to doing them by hand as you may have had to do so far in other modules. As with vectors, there are actually a number of different ways to create matrices in R, but let us begin by looking at the `matrix()` function and using the 'Help' command, via the question mark symbol, i.e., `?matrix()` (alternatively via the help tab) for more information. Doing so shows that the general form of the `matrix()` function is given by

```
matrix(data = , nrow = , ncol = , byrow = , dimnames = )
```

where each of the arguments are defined as follows:

- **data** - This is a vector of data that is used to create the elements of the matrix itself
- **nrow** - This specifies the number of rows desired for the matrix
- **ncol** - This specifies the number of columns desired for the matrix.
- **byrow** - This argument instructs R on how to fill the matrix using the data vector. If it takes the value of `TRUE`, then the elements will be filled row-wise, i.e. will first fill all the first row, then move down to second row etc, and if `FALSE`, the vice-versa.
- **dimnames** - This argument allows you to assign names to the rows and columns of the matrix.

Note that if either `nrow` or `ncol` is not given, then R will try to guess the required value(s) and will fill any unspecified elements by repeating the original data vector until filled.

Example 1.2. Consider the following two matrices

$$A = \begin{pmatrix} 3 & 4 \\ 6 & 2 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1 & 5 \\ 4 & 6 \end{pmatrix}.$$

These can created in R using the `matrix()` function as follows:

```
(A <- matrix(c(3,6,4,2), nrow = 2, ncol = 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]    3    6
## [2,]    4    2
```

```
(B <- matrix(c(1,4,5,6), nrow = 2, ncol = 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    5    6
```

1.6.7 Matrix Calculations

Now that you have your matrices created and assigned as variables A and B, you can use them in calculations:

```
A+B
```

```
##      [,1] [,2]
## [1,]    4   10
## [2,]    9    8
```

```
B-A
```

```
##      [,1] [,2]
## [1,]   -2   -2
## [2,]    1    4
```

```
A*B
```

```
##      [,1] [,2]
## [1,]    3   24
## [2,]   20   12
```

```
A/B
```

```
##      [,1]      [,2]
## [1,]  3.0 1.5000000
## [2,]  0.8 0.3333333
```

** BE CAREFUL!** Notice that all the calculations have been done element-wise again. As with the vectors, this turns out to be a very helpful tool within R although it might not appear so just now.

If you want to apply 'matrix-multiplication' you have to use a slightly different command:

```
A%*%B
```

```
##      [,1] [,2]
## [1,]   33   48
```

```
## [2,]   14   28
```

1.6.8 Matrix operations

There are, of course, an array of other calculations you may apply when working with matrices e.g, determinant, inverse, transpose etc. Rather than showing each of these in turn, in this section we simply provide a table of the different matrix/vector operations that can be used in R, with a brief description of what each of them are used for. We suggest that you try these out for yourself in order to familiarise yourself and understand how they work and don't forget to use the 'Help' function if you're unsure. Once you have mastered these operations, have a go at the exercises in the next section.

In the following table, `A` and `B` denote matrices, whilst `x` and `b` denote vectors:

Operation	Description
`A + B`	Element-wise sum
`A - B`	Element-wise subtraction
`A * B`	Element-wise multiplication
`A %*% B`	Matrix multiplication
`t(A)`	Transpose
`diag(x)`	Creates diagonal matrix with elements of `x` on the main diagonal
`diag(A)`	Returns a vector containing the elements of the main diagonal of `A`
`diag(k)`	If k is a scalar, this creates a (kxk) identity matrix
`solve(A)`	Inverse of `A` where `A` is a square matrix
`solve(A, b)`	Solves for vector `x` in the equation $A\vec{x}\vec{b}$ (i.e. $\vec{x} = A^{-1}\vec{b}$)
`cbind(A,B,...)`	Combines matrices(vectors) horizontally and returns a matrix
`rbind(A,B,...)`	Combines matrices(vectors) vertically and returns a matrix
`rowMeans(A)`	Returns vector of individual row means
`rowSums(A)`	Returns vector of individual row sums
`colMeans(A)`	Returns vector of individual column means
`colSums(A)`	Returns vector of individual column sums

** Recall that vectors are just particular cases of matrices with either one row or one column. Therefore, it is no surprise that you can create a vector using the matrix function. To do this, simply set the `nrow` or `ncol` argument equal to 1, depending on format of vector you want (row or column vector).**

The only slight restriction to simply using the `c()` function, is that R will always saves the vector as a column vector. We point out here that this might not be so clear when you first define the vector in R, as the output given in the console looks like the form of a row vector. To overcome this, you can simply turn the column vector (default when using combine function in R) into a row vector by performing the transpose (see table above) of the original vector.

1.6.9 Matrix extraction

In a similar way to how you we can extract values from vectors, we can extract values from matrices, this is also done with the square brackets `[]`, however it now takes two different arguments, one for the specified row(s) and the other for the column(s) which you would like to extract.

```
A
```

```
##      [,1] [,2]
## [1,]    3    6
## [2,]    4    2
```

```
A[1,1]
```

```
## [1] 3
```

```
A[2,1]
```

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

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

```
## [1] 3 4
```

```
A[,1] # The blank space mean every row
```

```
## [1] 3 4
```

1.6.10 Exercises

Using R, create the following matrices and vectors

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 8 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix} \qquad D = \begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}$$

1. Using the objects defined above, perform the following operations and check if the result is what you would expect:

i. $A + B$ element-wise sum
ii. $A \times B$ element-wise multiplication
iii. $A \times B$ matrix multiplication
iv. $B \times D$ matrix multiplication

v. $B \times \vec{b}$ matrix multiplication

2. Compute the transpose of matrix A and of matrix D.
3. Create a matrix with the elements 1, 2, 3, 4 in the main diagonal and zeros in the off diagonal elements.
4. Define a vector which consists of elements from the main diagonal of matrix B.
5. Build an identity matrix with dimension 10.
6. Compute the inverse of matrix A. Check if $A \times A^{-1} = I_3$.
7. Find the solution $\vec{x} = (x_1, x_2, x_3)^\top$, where

$$\begin{cases} 6.5 &= x_1 + x_2 + x_3 \\ 9 &= 0.5x_1 + 2x_2 + x_3 \\ 11 &= 3x_1 + x_2 + 2x_3 \end{cases}$$

8. Combine the matrices A and D, horizontally.
9. Combine the matrix A and the transpose of vector b vertically.
10. Compute the mean for each row of matrix A. Do the same for each column of matrix A.
11. Compute the sum for each row of matrix B. Do the same for each column of matrix B.

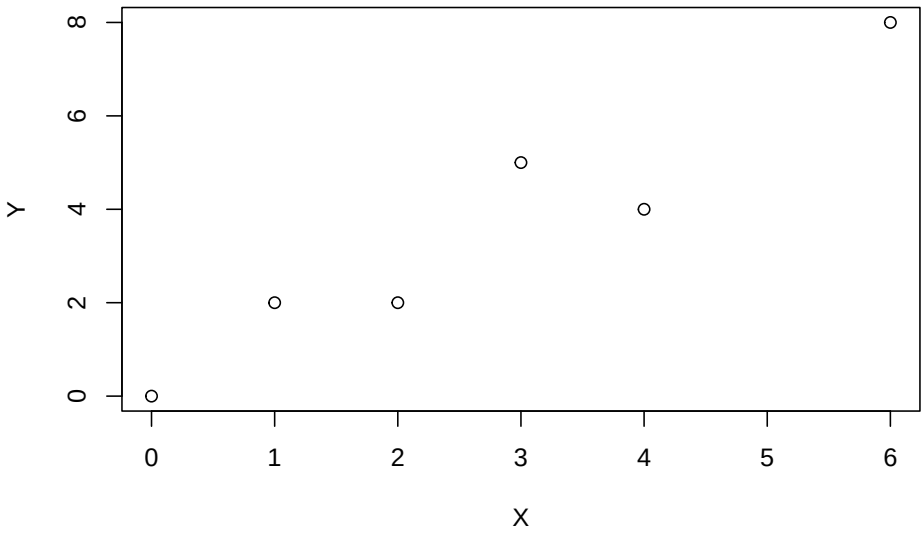
1.7 Plotting graphs

One of R's major strengths is the ease with which well-designed, publication-quality plots can be produced and can include mathematical symbols and formulae where needed. The basic plotting function in R, located in its basic packages, is the so-called `plot()` function. In its simplest form, the `plot()` function allows you to plot two variables, say X and Y, against each other as a scatter plot. For example, imagine we wanted to plot the following points (x, y):

$$(0,0),\ (1,2),\ (2,2),\ (3,5),\ (4,4),\ (6,8),$$

as a scatter plot. Then, we could do this as follows:

```
X <- c(0,1,2,3,4,6)
Y <- c(0,2,2,5,4,8)
plot(X, Y)
```



From the R plot above, you can see that R has simply taken the two vectors (`X` and `Y`) and plotted the values pairwise (as required) to create a basic scatter plot. That being said, the plot itself looks very basic and is not particularly aesthetic. This is because we have used the very basic structure for the `plot()` function. However, with a little alteration, this can be adapted to create something a little more exciting:

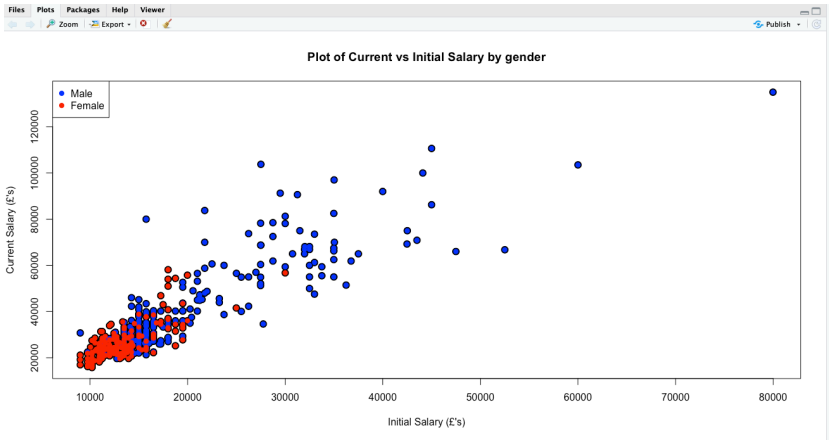


Figure 1.7: Example of Plot using plot().

The example above provides a small insight into the very basics of the plotting tools available in R. Let us know look at this function a little more closely. Using `?plot()` we find that the `plot()` function has the general form:

```
plot(x, y, main = , xlab = , ylab= , type= , pch= , col= , cex= ,
bty =  )
```

where each of the arguments are defined as follows:

- `x` - Points to be plotted along the x-axis
- `y` - Corresponding points to plotted against the y-axis. Note that these values match-up element-wise the x values to create co-ordinate pairs (x, y)
- `main` - Takes a character string and gives the plot a main title
- `xlab` - Takes a character string and labels the x-axis
- `ylab` - Takes a character string and labels the y-axis
- `type` - Takes a number of different character strings to define the type of plot desired, i.e. line, point etc. (see table below)
- pch‘ - Takes a values and defines the shape each point should take, i.e. circle, square etc. (see table below)
- `col` - Sets the colours of the points/lines in the plot
- `cex` - Takes a value and defines the size of the points
- `bty` - Takes a character string and sets the type of axes for the plot

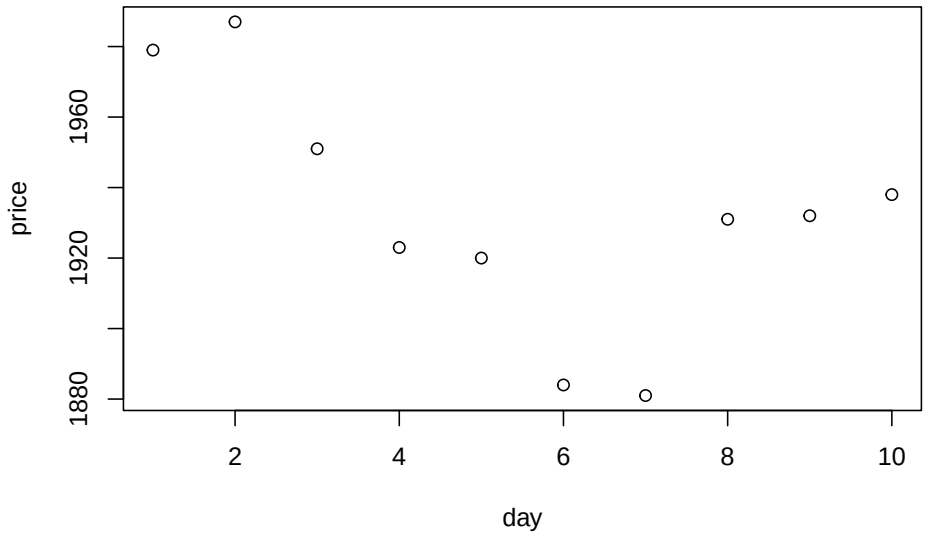
A number of other arguments can be used to change the layout and format of the plot but will not be discussed here. If you are interested, search for the `par()` function in the ‘Help’ tab.

Example 1.3. Consider the followings prices on the equity index S&P500 for the last weeks:

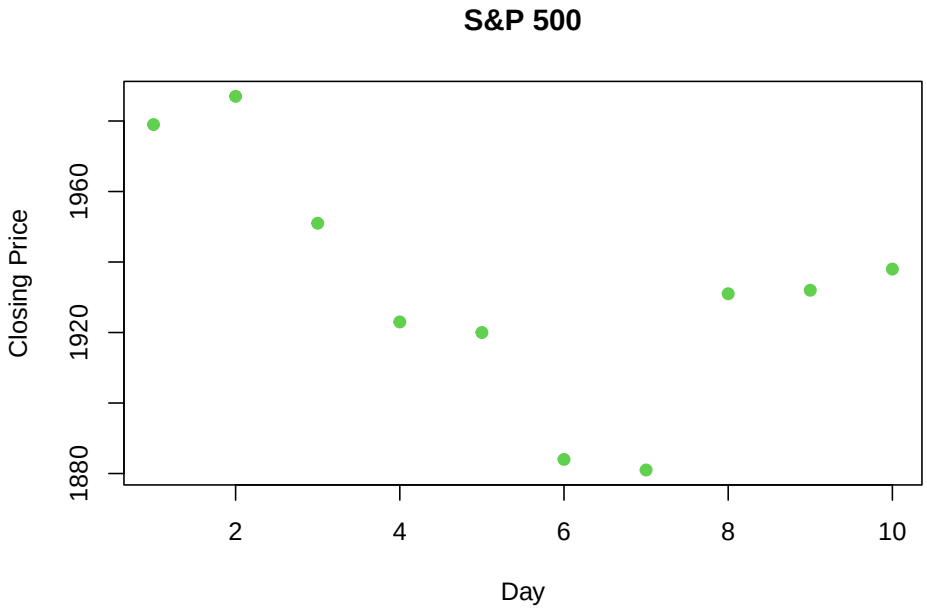
```
day <- c(1:10)
price <- c(1979,1987,1951,1923,1920,1884,1881,1931,1932,1938)
```

Using a combination of all the arguments in the above list, we can produce the following plots:

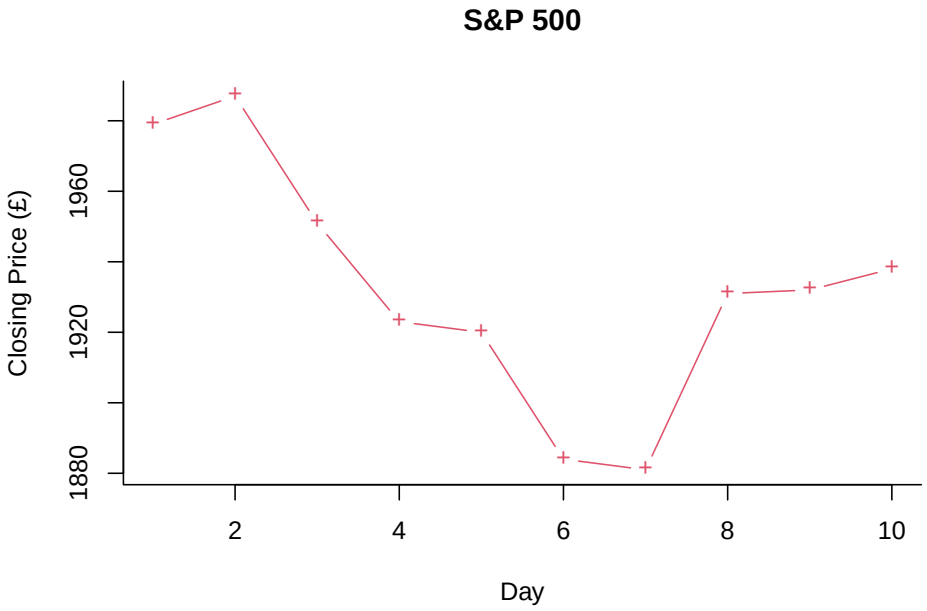
```
plot(day, price)
```



```
plot(day, price,
     main="S&P 500",
     xlab="Day",
     ylab="Closing Price",
     pch=19,
     col=3,
     type="p")
```




```
plot(day, price,
     main="S&P 500",
     xlab="Day",
     ylab="Closing Price (£)",
     pch="+",
     col=2,
     type="b",
     bty="L")
```



The table below gives a non-exhaustive list of some of the different options you can make when choosing arguments for your plots. To find more, search online:

`type`	Description
"p"	points
"l"	lines
"b"	both
"c"	lines part alone of "b"
"h"	histogram like vertical lines
"s"	stair steps

`pch`	Description
0	square
1	circle
2	triangle
4	plus
5	cross
6	diamond

`bty`	Description
"o"	full box
"n"	no axes

bty	Description
"7"	top and right axes
"L"	bottom and left axes
"C"	top, bottom and right axes
"U"	bottom, left and right axes

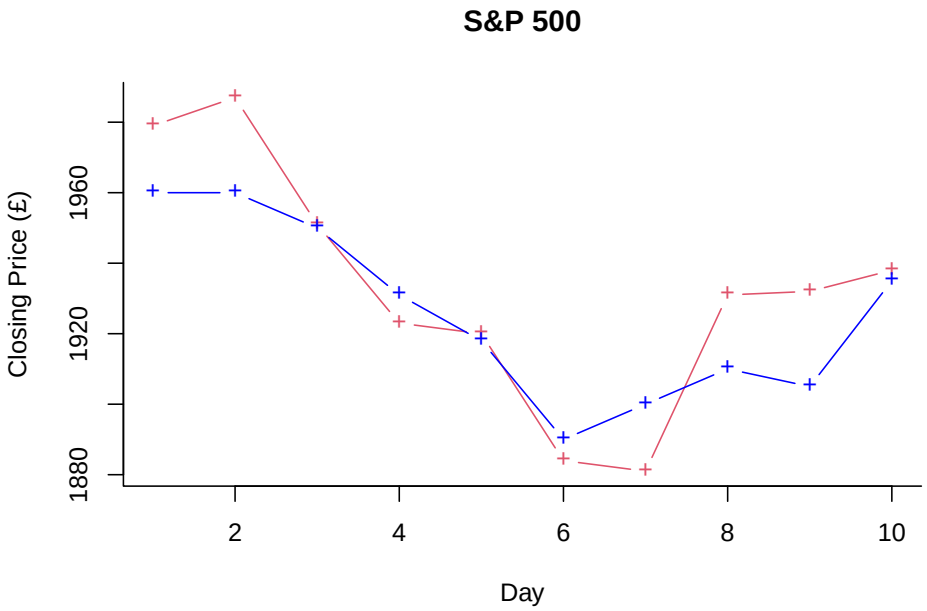
1.7.1 Adding to plots (lines, points etc.)

There will be many occasions where you wish to add another set of points, or some other plot to your original. This is usually the case when comparing two different sets of data or, for example, when wanting to draw a regression line through your data points. Again, R has a variety of pre-defined functions that allow you to do this with ease. However, those who are new to R will make the common mistake of trying to add a second plot to the original by using the `plot()` function for a second time. The `plot()` function (seen above) does not simply plot points or lines. The source code underpinning the `plot()` function first instructs R to create a separate window/panel, create a set of axes (designed based on the choice of `bty` as argument) create some axis labels then, finally, add the points or lines. Therefore, by executing the function again, you will find you produce a completely new plot rather than adding to the previous.

In order to add more graphics to the original plot, we instead have to use the functions `points()`, `lines()` and `abline()`. The `points()` and `lines()` functions work in a very similar to that of the `plot()` function in the sense that they take similar arguments. The only difference now, is that the function does not first create axes etc., but will simply plot the points/lines onto the most recent plot that was created. Note here that since the `points()` function can take `type` as an argument, it is actually possible to create line plots with this function (`type = "l"`) instead of using the lines function. Remember, there are many ways to create the same output in R, it is down to you to decide which you prefer to use. Let's add to the S&P example above, by also adding the FTSE prices during the same time period:

```
plot(day, price,
     main="S&P 500",
     xlab="Day",
     ylab="Closing Price (£)",
     pch="+",
     col=2,
     type="b",
     bty="L")
ftse <- c(1960, 1960, 1950, 1931, 1918, 1890, 1900, 1910, 1905, 1935)
points(day, ftse,
       col = "blue",
       type="b",
```

```
      pch="+")
```



The `abline()` function, on the other hand, is slightly different. This function is used simply to create straight lines on your current plot. Using `?abline()` we see it takes the form

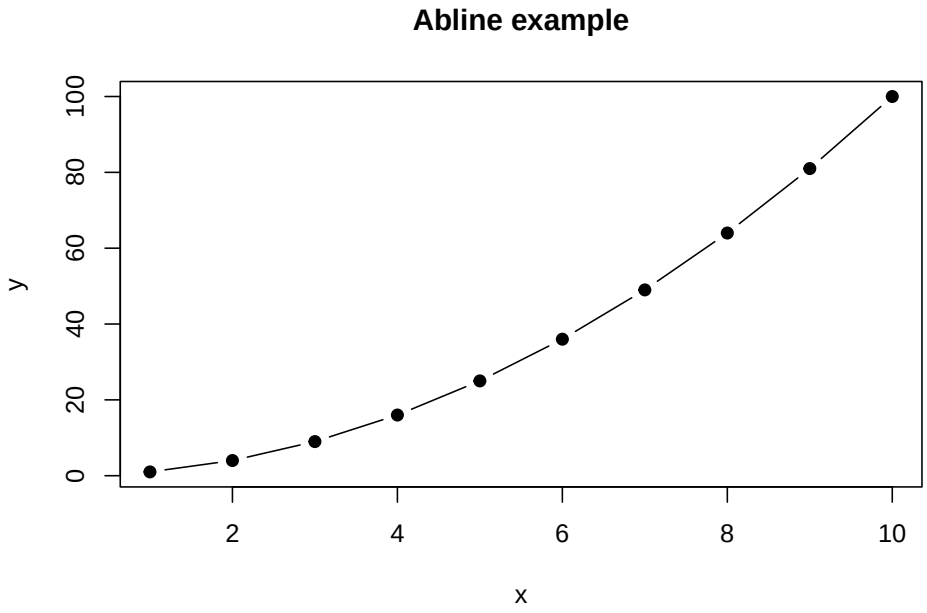
```
abline(a, b, h= , v = , ... )
```

The arguments in this case are no longer data points like in the previous plotting functions but correspond to co-ordinates:

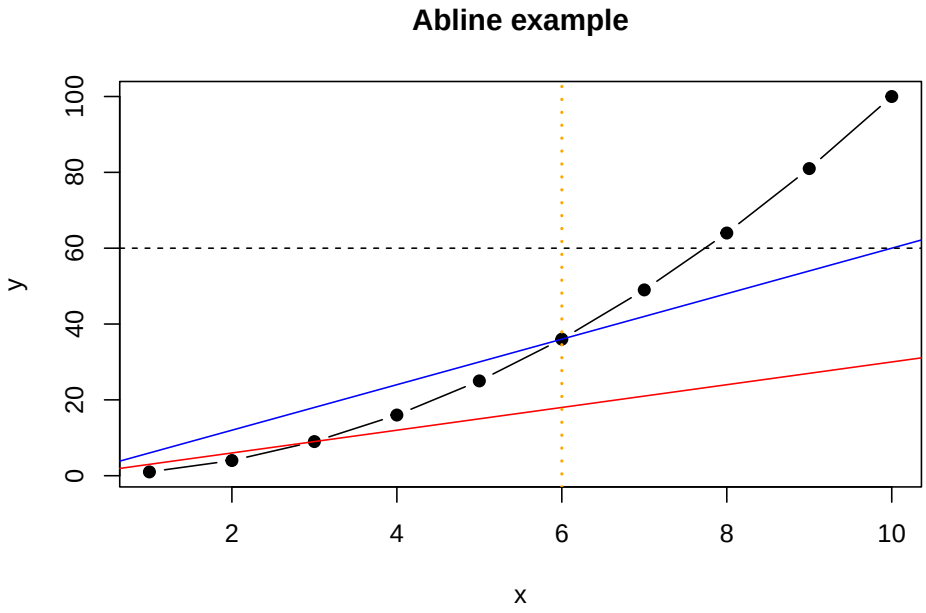
- `a` - The value of the intercept for the straight line
- `b` - The value for the gradient of the straight line
- `h` - The y co-ordinate (intercept) for a horizontal line
- `v` - The x co-ordinate for the vertical line.

In addition to these arguments, you can also format the line type, width etc., but we will not discuss these again as they are simply aesthetic parameters which you can easily search for online.

```
plot(1:10, (1:10)^2,
     main="Abline example",
     ylab="y",
     xlab="x",
     type="b",
     pch=19,)
```




```
plot(1:10, (1:10)^2,
     main="Abline example",
     ylab="y",
     xlab="x",
     type="b",
     pch=19,)
abline(a=0,b=3, col = "red")
abline(a=0, b=6, col = "blue")
abline(h=60, lty = 2)
abline(v=6, lty = 3, lwd = 2, col = "orange")
```



In addition to the plot function and its variety of options, we can implement other plotting functions such as `hist()` and `boxplot()`, which will be discussed in a later chapter in more details, to produce the best graphical representation of your data possible. Finally, although we discuss graphics using the basic plotting commands here, it is worth pointing out the popularity of a completely different package and set of functions, known as `ggplot2`, which makes the plotting experience even more exciting. We will not actually discuss this in these lecture notes, however, it is strongly advised that you familiarise yourself with this package and its associated functions using DataCamp. In fact, there are three excellent courses devoted to the subject which will be linked at the end of these notes.

1.7.2 Exercises

Consider the following formula to calculate the number of mortgage payment terms required to pay off a mortgage as a function of the principle amount (P), the monthly repayments (M) and the monthly interest (i):

$$n = \frac{\ln\left(\frac{i}{\frac{M}{P}-i} + 1\right)}{\ln(1+i)}$$

Using R, solve the following problems:

1. Calculate the number of payments n for a mortgage with principle balance of £200,000, monthly interest rate of 0.5% and monthly payments of £2000.

2. Now construct a vector, named n, of length 6 with the results of this calculation (in years) for a series of monthly payment amounts: $(2000, 1800, 1600, 1200, 1000)$.
3. Does the last value of n surprise you? Can you explain it?
4. Create a line plot for the values of n (excluding the last) against the different payment amounts. Give the plot a title, appropriate label names and make the points appear in blue.

1.8 DataCamp Courses

- https://www.datacamp.com/courses/free-introduction-to-r (R Basics - Recommended)
- https://www.datacamp.com/courses/data-visualization-in-r (Plotting Data)
- https://www.datacamp.com/courses/data-visualization-with-ggplot2-1 (Plotting Data using ggplot - Recommended)
- https://www.datacamp.com/courses/data-visualization-with-ggplot2-2
- https://www.datacamp.com/courses/data-visualization-with-ggplot2-part-3

Chapter 2

Conditionals and IF Statements

In R, conditional statements or arguments are used to compare or analyse values/data based on certain conditions. In general, this is done with the use of 'relational operators' (=, >, <, >=, <=, !=) and 'logical operators' (OR, AND, AND/OR).

2.1 Relational Operators

The most basic of the 'relational operators' is the equality operator (==), which can be used to check if two objects (values, vectors, matrices etc.) are equal:

```
4 == 3+1
```

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

```
5^2 == 25
```

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

```
8 %% 5 == 3  # The double percentage sign here resembles modulo arithmetic, i.e. 8 mod 5
```

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

This can also be performed on vectors on an element by element basis (as usual):

```
1:10 == c(1,2,3,4,5,6,7,8,9,10)
```

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

```
1:10 == c(0,2,3,4,5,6,7,8,9,10)
```

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

Unsurprisingly, it also works on matrices on an element by element basis as well:

```
matrix(5, nrow = 3, ncol = 3)
```

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

```
matrix(1:9, nrow = 3) == matrix(5, nrow = 3, ncol = 3)
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE FALSE FALSE
```

```
diag(5, nrow = 3, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    5    0    0
## [2,]    0    5    0
## [3,]    0    0    5
```

```
diag(5, nrow = 3, ncol = 3) == 5 * diag(1, nrow =3)
```

```
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE
```

Notice that this equality operator uses a double equal sign (==) rather than a single =. This is due to the fact the single equality sign is already used for assignments (similar to <-). This can be confusing, can easily cause errors and is the main reason I always suggest using <- for variable assignment.

Conversely, you can use the not equal operator (!=) in a similar way

```
3 != 5
```

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

```
seq(1, 10, by = 1) != 1:10
```

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

Note - In general, the (!) symbol negates any type of relational operator or Boolean value in R, e.g.

```
!TRUE
```

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

```
!FALSE
```

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

In a similar way, you should easily be able to understand how the rest of the relational operators work, i.e. (`<`, `>`, `<=`, `>=`). In the following example(s), I will introduce you to one of the many pre-programmed data sets that form part of the base package data sets, i.e, mtcars; we will discuss data sets in more details in the next few weeks.

```
mtcars
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Exercise 2.1. Assume we want to analyse the `hp` (horsepower) variable (col-

umn) only. Based on what we discussed last week regarding vector/matrix extraction, how can we extract `hp` data only?

Solution

```
mtcars[,4]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

An alternative method of extraction for data sets (data frames) is to use the `$` extraction command based on the column/variable name. Note that this only works on data frames and not general matrices, whereas the square bracet extraction works for both:

```
(HP <- mtcars$hp)
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

```
HP > 200
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [25] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

What do you think will happen if we execute the code `sum(HP>200)` and `mean(HP>200)`? Have a think about this then check out the solution when you're ready.

Solution

```
sum(HP > 200)
```

```
## [1] 7
```

```
mean(HP > 200)
```

```
## [1] 0.21875
```

In both of these case, the conditional statement(s) have produced a vector of `TRUE` and `FALSE` Boolean values. In R, these are understood as being values of 1 and 0 resepectively. Hence, it is then possible to take the `sum()` or the `mean()` over the Boolean values themselves.

The above gives an examples of how R understands the Boolean values (`TRUE`/`FALSE`) as 1 and 0, respectivley and also give you an idea of how powerful such simple lines of conditional code can be when used in the right way.

Exercise 2.2. Can you create a vector of all square numbers from 1 to 100 and count how many of these values are divisible by 3? Moreover, can you determine what percentage of them are NOT divisible by 5?

In the next few weeks, we will look in more details at how we can use these relational operators (along with the logical operators discussed below) to conditionally extract data/values from a data.frame. This is a very helpful skill to learn for data handling and manipulation.

2.2 Logical Operators

'Logical operators' are used to check whether multiple conditions have been satisfied at the same time (AND) or at least one of them (OR). The key to understanding how these work in R, is understanding how logical operators work in theory.

Let us begin with the logical operator 'AND' which, in R, is denoted via `&` or `&&` (I will explain the difference later). For an AND statement/condition to evaluate to `TRUE`, both conditions in the statement must be `TRUE`. That is, the condition on the left is `TRUE` 'AND' the condition on the right is `TRUE`

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
FALSE & FALSE
```

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

```
pi
```

```
## [1] 3.141593
```

```
pi > 3
```

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

```
pi < 4
```

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

```
pi > 3 & pi < 4
```

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

```
5 < 10 & 5 < 3
```

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

It is actually possible to have more than two arguments and include different relational operators as well.What do we think the following expression will evaluate to, `TRUE` or `FALSE`?

```
pi > 0 & pi < 5 & !(pi %% 2 == 0)
```

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

As with relational operators, logical operators can also be used in vector form, where the & operator evaluates on a term by term basis, e.g.

```
c(1,2,3) < c(2,3,4) & c(2,3,4) < c(3,4,5) # Think about this one a little!
```

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

In fact, this sort of logical/relational operation can also be computed on other objects than just numerical values, i.e. 'character strings':

```
"Red" == "Red"
```

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

```
"Red" == "Blue"
```

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

```
"Red" == "red"
```

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

```
c(1, 2, 3) < c(2, 3, 4) & "Red" == "Blue" # How has this worked? The left hand side is
```

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

```
c(1, 2, 3) < c(2, 1, 4) & "Red" == "Red"
```

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

In contrast to `&` which evaluates on a term by term basis, the double `&&` reads from left to right and only evaluates the first values of each vector

```
c(1, 2, 3) < c(2, 1, 4) && "Red" == "Red"
```

```
## Warning in c(1, 2, 3) < c(2, 1, 4) && "Red" == "Red": 'length(x) = 3 > 1' in
## coercion to 'logical(1)'
```

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

```
c(5, 2, 3) < c(2, 1, 4) && "Red" == "Red"
```

```
## Warning in c(5, 2, 3) < c(2, 1, 4) && "Red" == "Red": 'length(x) = 3 > 1' in
## coercion to 'logical(1)'
```

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

The second logical operator is the so called OR operator, denoted by `|` and `||`, which evaluates to `TRUE` as long as 'at least one statement is `TRUE`', e.g.

```
TRUE | TRUE
```

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

```
TRUE | FALSE
```

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

```
FALSE | TRUE
```

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

```
FALSE | FALSE
```

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

```
F | F | T | F #etc.
```

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

The same ideas as were discussed above for `&` work also for `|`, i.e. `|` evaluates element-wise, whilst `||` only evaluates the first element of a vector.

Exercise 2.3. With all this in mind, how can we calculate the number of cars in the mtcars data set that have horsepower greater than 200, mpg at most 30, are automatic but do not have 6 cylinders?

Exercise 2.4. The set of data VADeaths contains the death rates (measured per 1000 population per year), in Virginia, USA, in 1940. The structure of this data set is a matrix (not a data frame) with the rows denoting age ranges and the columns sex/area.

i. How can we find out this information (and possibly more) about the data set? ii. Extract the two columns containing the female data, either together or separately.
ii. Using conditional arguments, determine how many age groups have a death rate larger than 20 for rural females and a death rate less than 30 from Urban females.

2.3 IF Statements

'IF' Statements are extremely popular and powerful tools in programming that are used to execute certain commands, based on given conditions. In most cases, the conditions used within IF statements are built up from combinations of the relational and logical operators seen above.

In general, an IF statement has the following form:

```
if ( condition ){ command } else { command }
```

To see how an IF statement works in practice, let us look at a simple example to check if a number is odd or even

```
x <- 8

if (x %% 2 == 0){
  print("This number is even")
} else {
  print("This number is odd")
}
```

```
## [1] "This number is even"
```

You can actually make the output even better in this example by asking it to print out the value of x that has been given by using the paste function `paste()`. Notice the variable x is not in quotation marks but the 'words' are.

```
x <- 14

if (x %% 2 == 0){
  print(paste(x, "is an even number"))
} else {
  print(paste(x,"is an odd number"))
}
```

```
## [1] "14 is an even number"
```

This is quite a simple example but it is very possible to have more complicated and longer IF statements that contain more conditional possibilities. If this is the case, you can simply extend the IF statement by adding `elseif` instead of just else. Finally, once you have finished with all conditions, you finish with else. For example

```
x <- 7

if (x < 0) {
  print(paste(x, "is a negative number"))
} else if (x > 0) {
  print(paste(x, "is a positive number"))
} else {
  print(paste(x, "is Zero"))
}
```

```
## [1] "7 is a positive number"
```

Exercise 2.5. Can you create an IF statement which tells you whether a number (x) is divisible by another number (y), where both x and y can be changed (not fixed)? Hint: Use the modulus operator `%%`.

Looking back at the previous two examples regarding even/odd and positive/negative numbers, we can actually combine these two statements by using logical operators within the IF conditions:

```
x <- 4

if (x < 0 & x %% 2== 0) {
    print(paste(x, "is a negative even number"))
  } else if (x < 0) {
    print(paste(x,"is a negative odd number"))
} else if (x > 0 & x %% 2 == 0) {
    print(paste(x, "is a positive even number"))
  } else if (x > 0){
    print(paste(x, "is a positive odd number"))
} else {
  print(paste(x, "is Zero"))
}
```

```
## [1] "4 is a positive even number"
```

In fact, you could do this an alternative way by 'nesting' IF statements inside one another to make several 'layers'. There is no right or wrong way to do these but through experience you will see either can be used depending on the situation.

```
x <- 3

if (x < 0) {
  if (x %% 2 == 0){
  print(paste(x, "is a negative even number"))
  } else {
    print(paste(x,"is a negative odd number"))}
} else if (x > 0) {
  if (x %% 2 == 0){
  print(paste(x, "is a positive even number"))
  } else {
    print(paste(x, "is a positive odd number"))
  }
} else {
  print(paste(x, "is Zero"))
}
```

```
## [1] "3 is a positive odd number"
```

What happens if we let x be a vector?

Note - The IF statement will technically work in the sense it will print something out, but it will not do quite what we expect. This is because in an IF statement, the conditions or 'test statements' can only be single elements and thus, R will only consider the first element of the vector. With this in mind, it is important to note that if you use a logical operator in an IF statement, it is always best to

use the double version, i.e. `&&` or `||`.

That being said, it is possible to bypass such a problem using the `ifelse()` function. The `ifelse()` function allows us to create an IF statement which only has one condition but can be applied to a vector element-wise.

```
x <- c(1, 2, 3)
ifelse(x %% 2 == 0, "Even", "Odd")
```

```
## [1] "Odd"  "Even" "Odd"
```

Note - This only works for quite simple statements.

It is possible to use a more complicated IF statement on a vector as we tried above but to do so we have to introduce the idea of FOR loops, which we will discuss next week!

2.4 Exercises

1. Create an R script that calculates the square root of a value, x. If the value contained in x is negative it should return `NA` as output.
2. Create an R script that returns the maximum value out of the elements of a numeric vector of length 2 (two elements), without using the min, max or sort functions.
3. Use the command `x <- rexp(20, rate = 0.5)` to create a vector containing 20 simulations of an Exponential random variable with mean 2. Return the number of values that are larger than the mean of the vector x. You are allowed to use the `mean()` function.

Chapter 3

Loops

3.1 For Loops

'For loops', sometimes just known as 'Loops' are one of the most useful tools in programming and you will find, once you understand how to implement them, that they become your best friends. That being said, it is very common that people like them so much that they are used when they are not necessary, as we will see later.

Simply put, a 'for loop' allows us to 'loop' through all the elements of a given object (usually a vector or matrix) and perform a command or operation for each element. When combined with 'IF statements', 'for loops' become very powerful and flexible and allow you to perform almost any task.

Let us start by understanding how a basic 'for loop' is constructed, then we will consider some simple examples. The general form of a for loop is as follows:

```
for (i in x) { command in terms of i }
```

That is, i will take the first value of the object x, perform the command in the brackets with this given value of i, then i will loop to the second value of x and so on. For example:

```
for (i in c(1,2,3,4,5)) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

This works perfectly but notice that we could also do this using what we called 'vectorised calculation', which takes advantage of how R deals with vectors on an element-by-element basis:

```
(1:5)^2
```

```
## [1]  1  4  9 16 25
```

As another example, consider the following:

```
(x <- seq(from = 10, to = 100, by = 5))
```

```
##  [1]  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

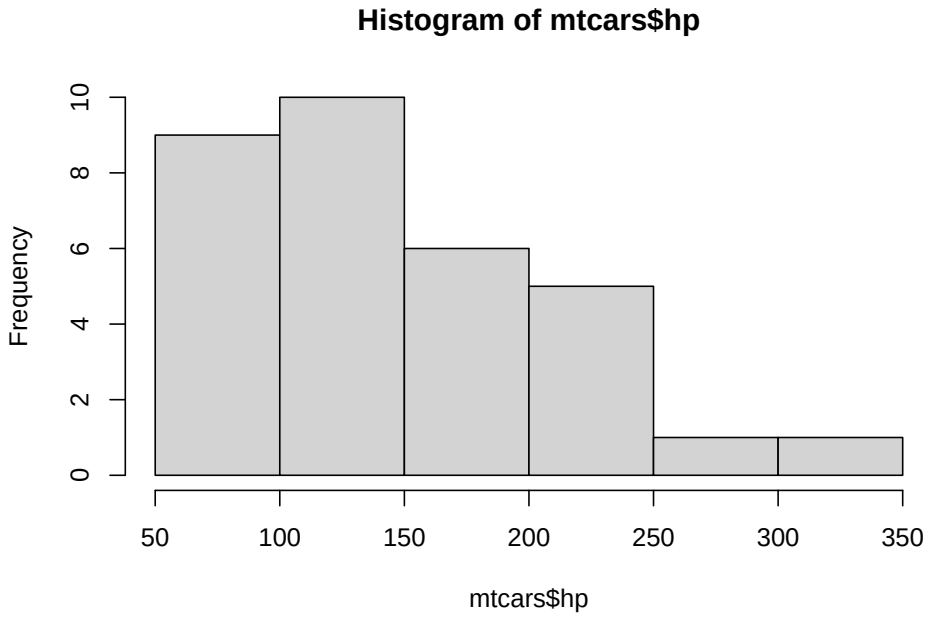
```
for (i in x){
  print(i %% 2 == 0)
}
```

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

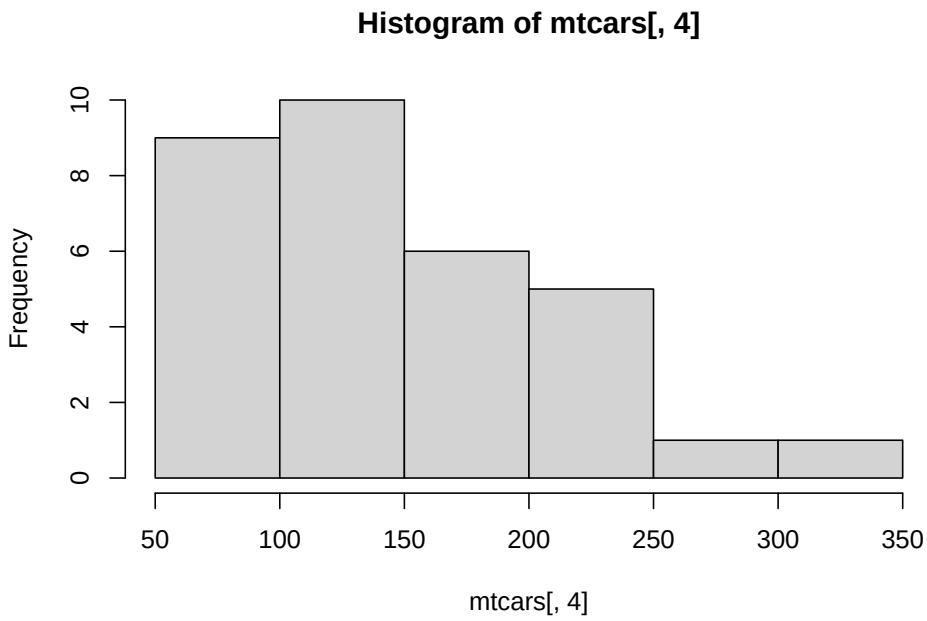
Again, was this necessary or could we have used vectorised calculations again? If possible, you should always use the vectorised calculation version of a command as this saves times and processing power. That being said,there are many situations where 'for loops' are necessary, not just useful.

Let us return to our mtcars data set seenin the previous chapter and consider a problem regarding plotting histograms of the data:

```
hist(mtcars$hp)
```

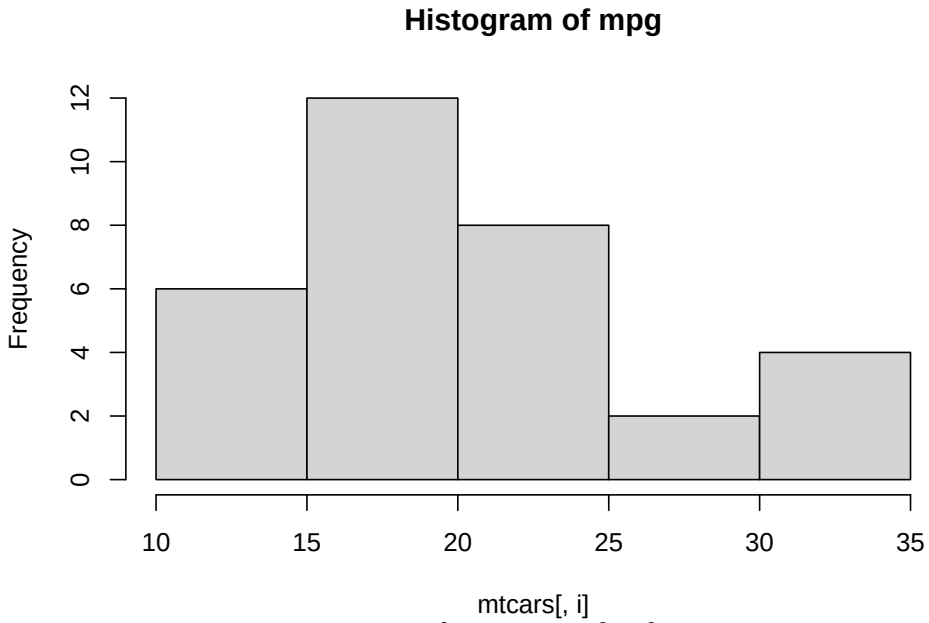


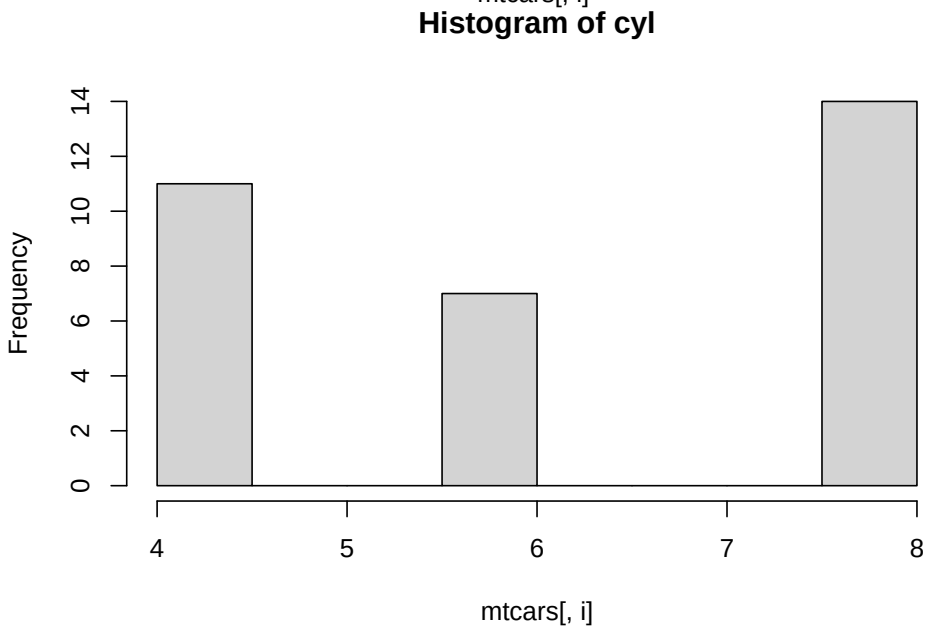

```
hist(mtcars[,4])
```

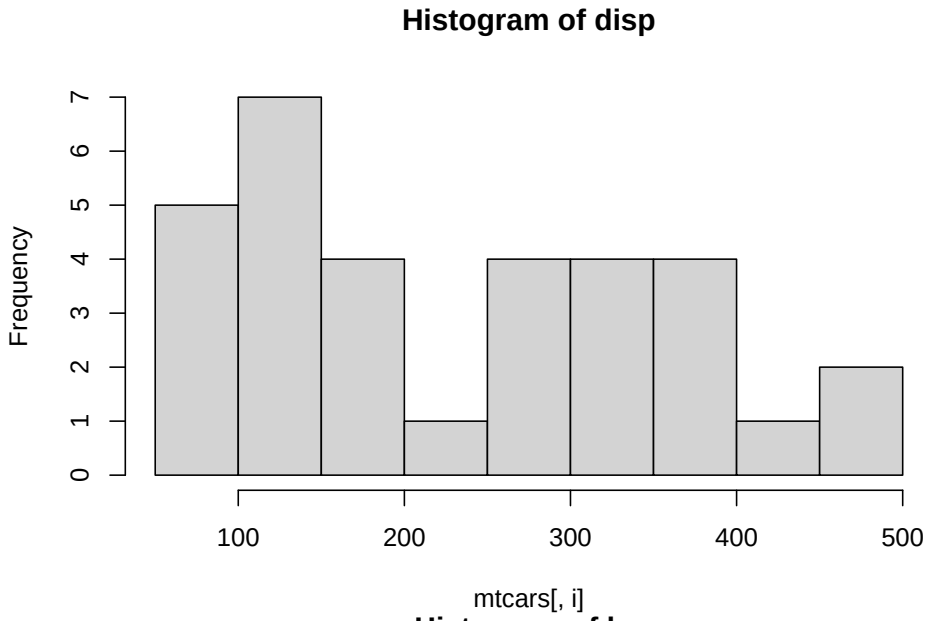


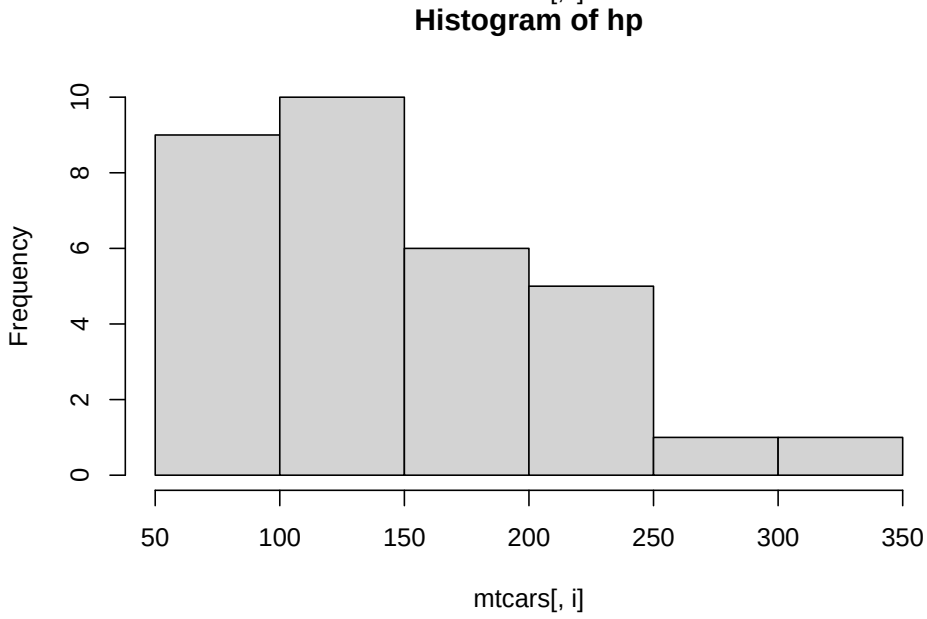
Now, imagine you wanted a histogram for every variable. Executing the code `hist(mtcars)` wouldn't work as the input necessary for this function should be in the form of a single vector of values. However, to overcome this hurdle, we could make use of 'for loops':

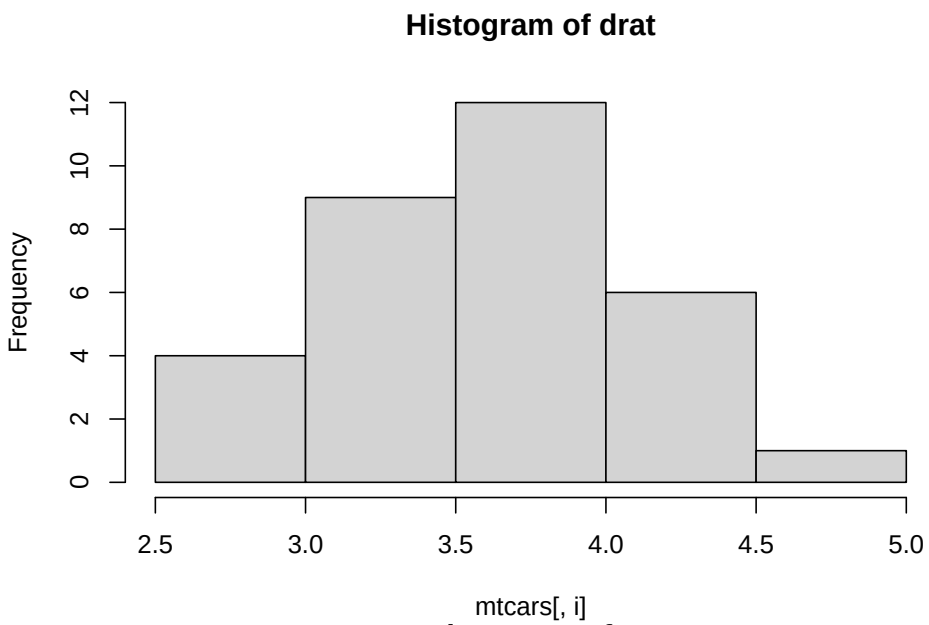
```
for (i in 1:ncol(mtcars)){
  hist(mtcars[,i], main = paste("Histogram of", colnames(mtcars)[i]))
}
```

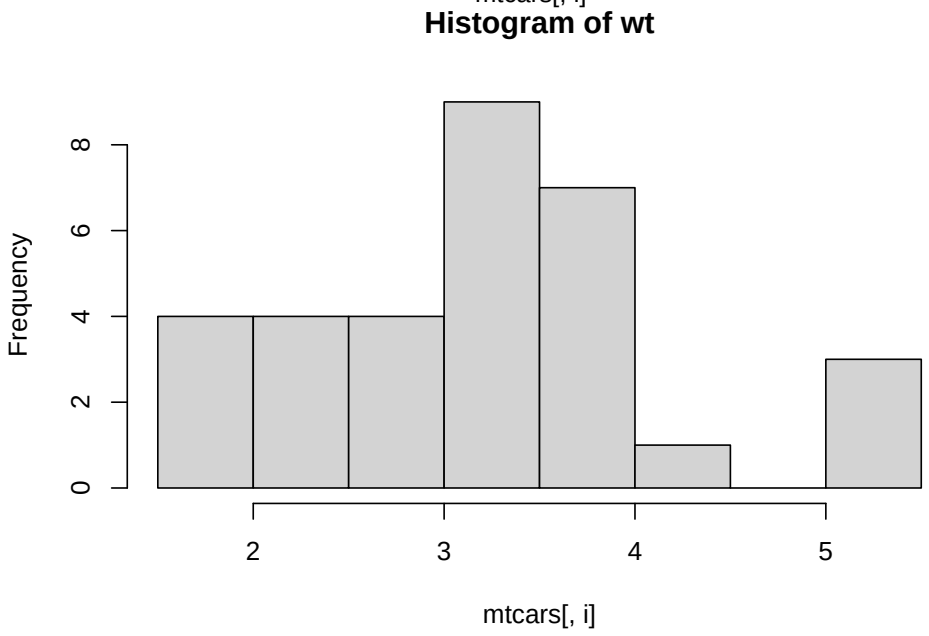


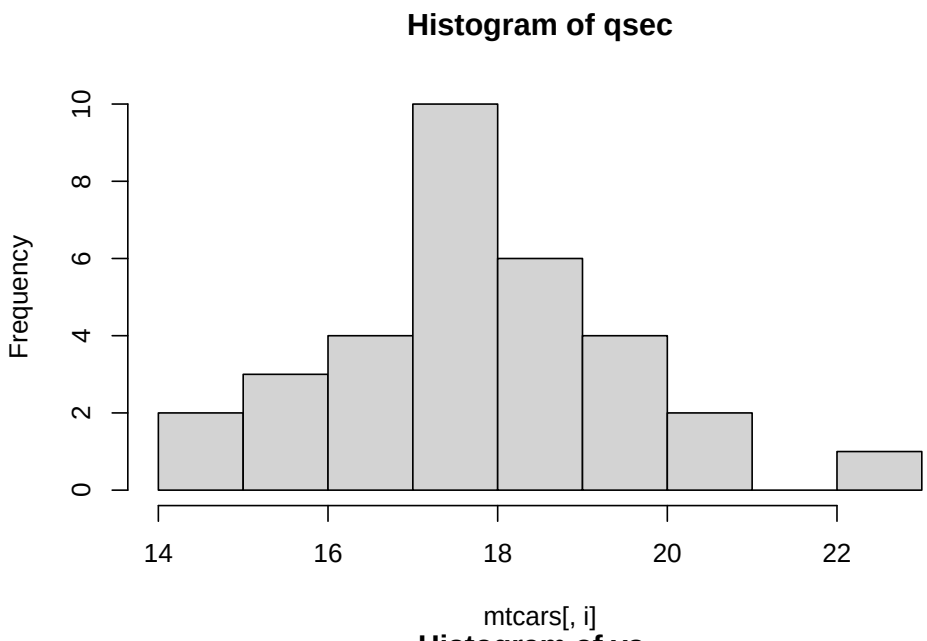


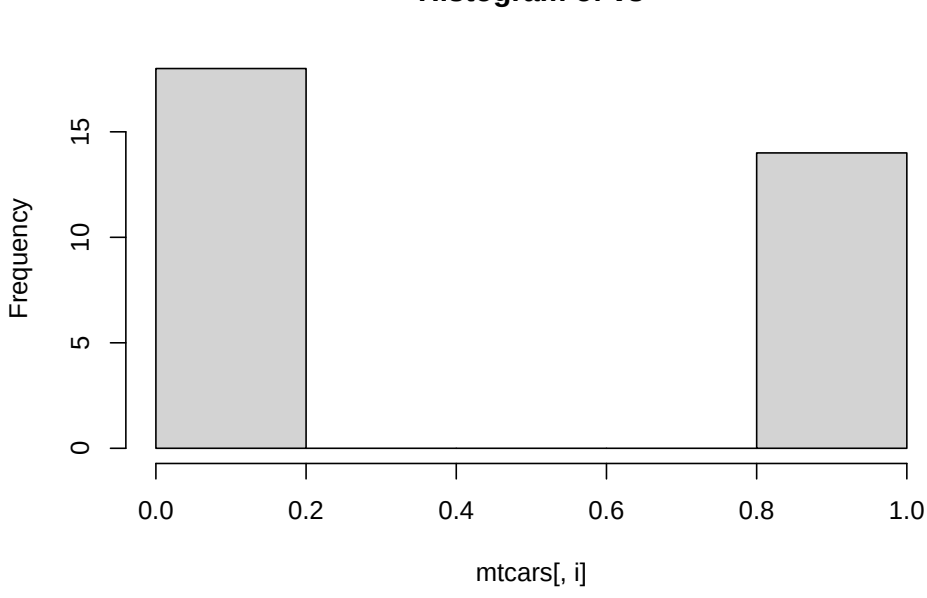

Histogram of disp
Frequency
0
1
2
3
4
5
6
7
100
200
300
400
500
mtcars[, i]

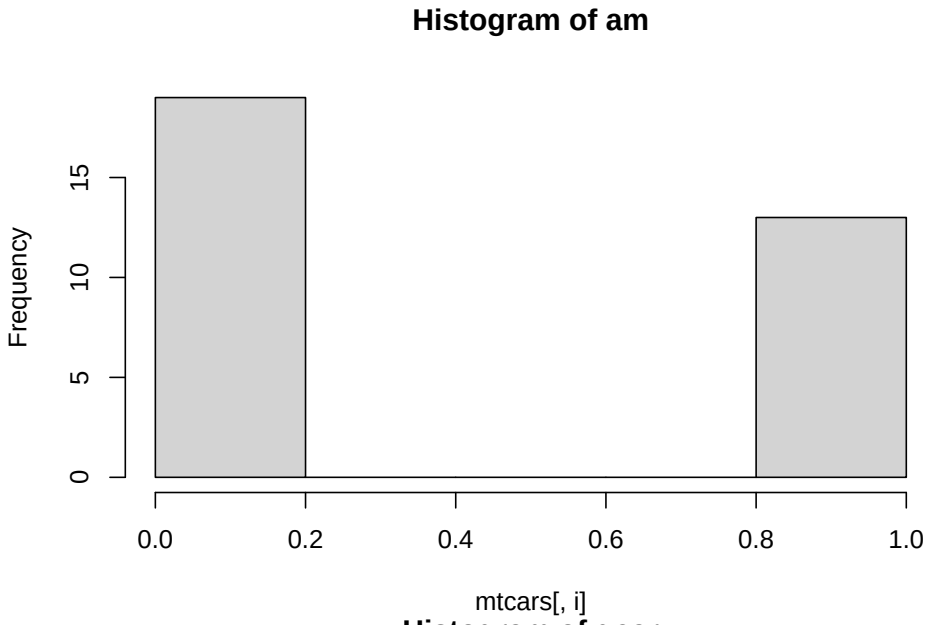

Histogram of hp
Frequency
0
2
4
6
8
10
50
100
150
200
250
300
350
mtcars[, i]

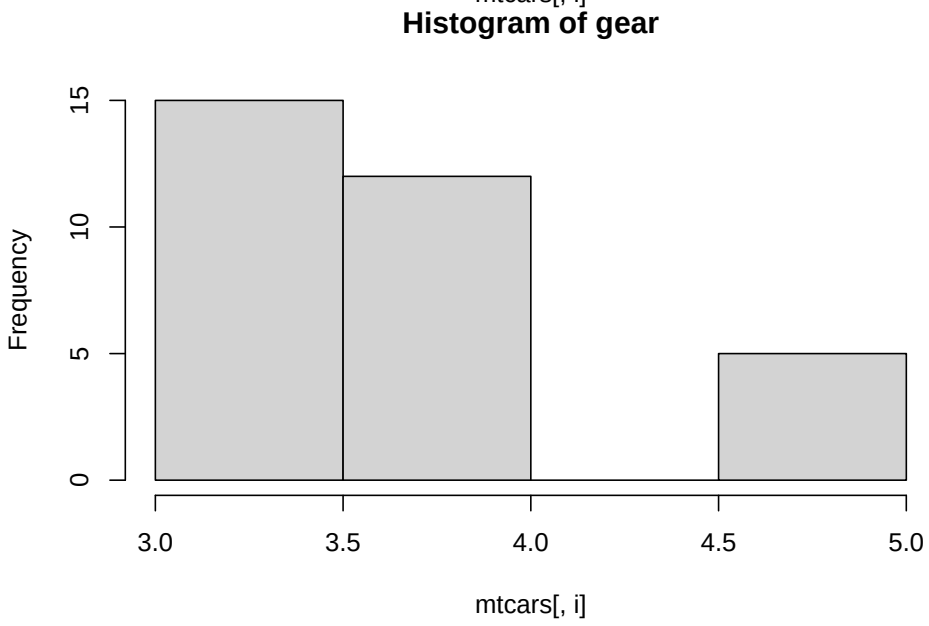

Histogram of drat
Frequency
0
2
4
6
8
10
12
2.5
3.0
3.5
4.0
4.5
5.0
mtcars[, i]

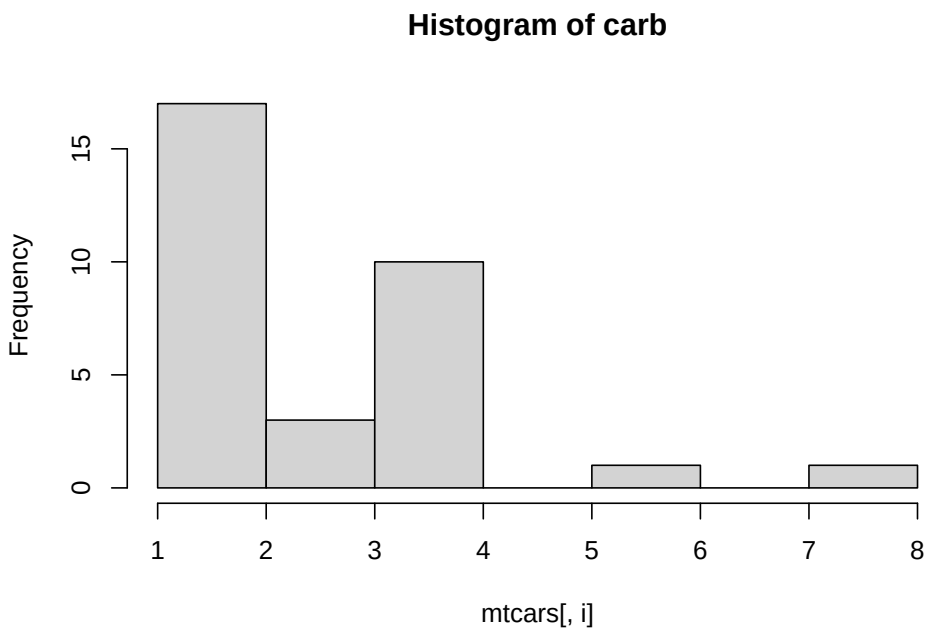

Histogram of wt
Frequency
0
2
4
6
8
2
3
4
5
mtcars[, i]


Histogram of qsec
Frequency
0
2
4
6
8
10
14
16
18
20
22
mtcars[, i]


Histogram of vs
Frequency
0
5
10
15
0.0
0.2
0.4
0.6
0.8
1.0
mtcars[, i]

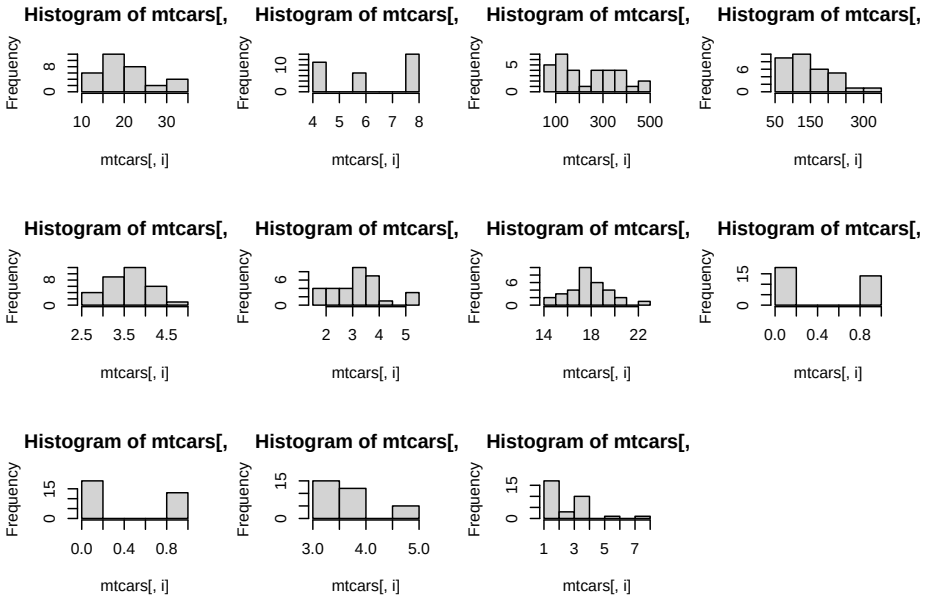

Histogram of am
Frequency
0
5
10
15
0.0
0.2
0.4
0.6
0.8
1.0
mtcars[, i]


Histogram of gear
Frequency
0
5
10
15
3.0
3.5
4.0
4.5
5.0
mtcars[, i]



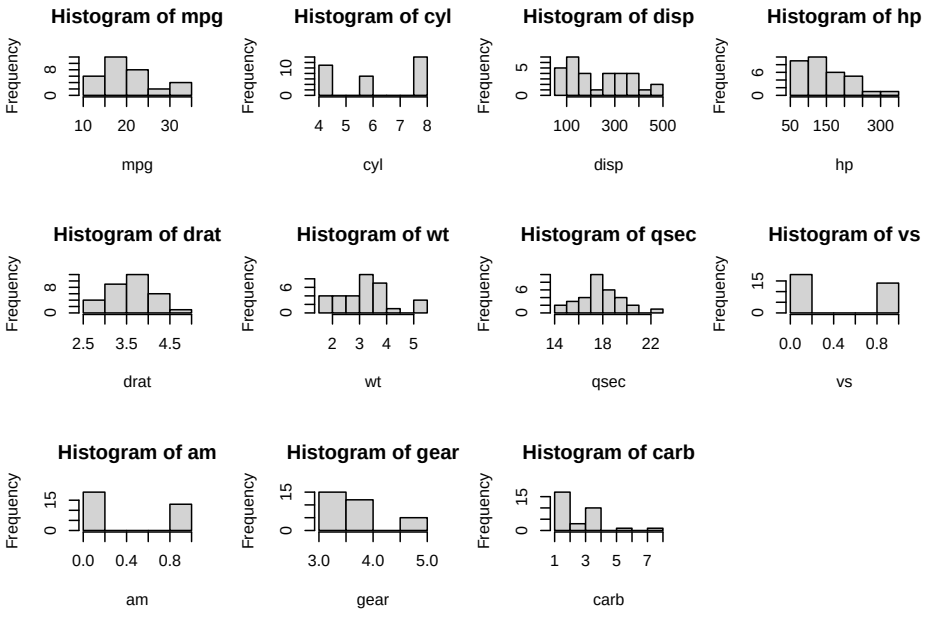
The above is great, but it would be nice to have them all on one screen together.
Note that the code below is not really linked to for loops but is still worth mentioning here.

```
par(mfrow = c(3,4)) # Changes the plot frame to fit 3 rows and 4 columns of separate plots.
for(i in 1:ncol(mtcars)){
  hist(mtcars[,i])
}
```



This is much better but I would like the individual titles and axis labels to reflect the variable name:

```
par(mfrow = c(3,4))
for(i in 1:ncol(mtcars)){
  hist(mtcars[,i], main = paste("Histogram of", colnames(mtcars)[i]), xlab = paste(coln
}
```



Even this very simply example starts to show you the value and versatility of for loops.

Now, as mentioned above, it is also possible to combine 'for loops' with IF statements. For example, the code below counts the number of even numbers in a vector of values:

```
x <- c(2,5,3,9,8,11,6)

count <- 0
for (i in x) {
  if(i %% 2 == 0){
    count <- count+1
  }
}
print(count)
```

```
## [1] 3
```

Exercise 3.1. Is there a quicker and easier way to achieve what has been done above without 'for loops'?

Solution

```
sum(x %% 2 == 0)
```

```
## [1] 3
```

Exercise 3.2. Can you write a 'for loop' that prints out the names of the cars in the `mtcars` data set which have 8 cylinders? Note, the car names can be found using the `rownames(mtcars)` command.

Solution

```
for(i in 1:nrow(mtcars)){
  if(mtcars$cyl[i]==8){
    print(rownames(mtcars)[i])
  } else {}
}
```

```
## [1] "Hornet Sportabout"
## [1] "Duster 360"
## [1] "Merc 450SE"
## [1] "Merc 450SL"
## [1] "Merc 450SLC"
## [1] "Cadillac Fleetwood"
## [1] "Lincoln Continental"
## [1] "Chrysler Imperial"
## [1] "Dodge Challenger"
## [1] "AMC Javelin"
```

```
## [1] "Camaro Z28"
## [1] "Pontiac Firebird"
## [1] "Ford Pantera L"
## [1] "Maserati Bora"
```

In fact, there is actually another way this can be done using conditional extraction which we will talk more about next week.

Exercise 3.3. Remember our IF statement from last week that didn't work correctly because we used a vector in the conditional statement? i.e.

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

if (x < 0) {
  if (x %% 2 == 0){
    print(paste(x, "is a negative even number"))
  } else {
    print(paste(x,"is a negative odd number"))}
} else if (x > 0) {
  if (x %% 2 == 0){
    print(paste(x, "is a positive even number"))
  } else {
    print(paste(x, "is a positive odd number"))
  }
} else {
  print(paste(x, "is Zero"))
}
```

Can you now apply the idea of a 'for loop' to get this to work correctly?

Solution

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

for(i in x){
if (i < 0) {
  if (i %% 2 == 0){
    print(paste(i, "is a negative even number"))
  } else {
    print(paste(i,"is a negative odd number"))}
} else if (i > 0) {
  if (i %% 2 == 0){
    print(paste(i, "is a positive even number"))
  } else {
    print(paste(i, "is a positive odd number"))
  }
} else {
  print(paste(i, "is Zero"))
```

```
  }
}
```

```
## [1] "1 is a positive odd number"
## [1] "2 is a positive even number"
## [1] "3 is a positive odd number"
```

3.1.1 Matrices

So far, we have seen how we can Loop through a vector of values to perform certain tasks, but it is also possible to do this over a matrix of values. The only difference is that this requires two loops (one for each index - row and column). For example:

```
(M <- matrix(round(runif(9,min = 0, max = 100)), nrow = 3, ncol = 3)) # This creates a 3x3 matrix
```

```
##      [,1] [,2] [,3]
## [1,]   48   18   17
## [2,]    1   25    8
## [3,]   66   21   96
```

```
for(i in 1:nrow(M)){
  for(j in 1:ncol(M)){
    print(paste("Element [", i,",",j,"] of M is equal to",M[i,j]))
    }
}
```

```
## [1] "Element [ 1 , 1 ] of M is equal to 48"
## [1] "Element [ 1 , 2 ] of M is equal to 18"
## [1] "Element [ 1 , 3 ] of M is equal to 17"
## [1] "Element [ 2 , 1 ] of M is equal to 1"
## [1] "Element [ 2 , 2 ] of M is equal to 25"
## [1] "Element [ 2 , 3 ] of M is equal to 8"
## [1] "Element [ 3 , 1 ] of M is equal to 66"
## [1] "Element [ 3 , 2 ] of M is equal to 21"
## [1] "Element [ 3 , 3 ] of M is equal to 96"
```

Another very important technique that you will need when working with 'for loops' is how to store values in a new vector (matrix) as you finish each loop. This is something that you will use a lot when working through your R based assessments in your Actuarial modules, since you will be working with larger data sets and need to make calculations which then need to be saved for use later on.

As a simple example let us see how we could use a 'for loop' to generate some random values and save them in a vector if they satisfy some condition.

Before we start, let us note how you can add a value to an already existing vector

```
(x <- c(1, 3, 5, 7, 9))
```

```
## [1] 1 3 5 7 9
```

```
(x <- c(x, 11))
```

```
## [1]  1  3  5  7  9 11
```

In the above line of code, x has been over-written as the vector which contain all the values of the original vector x but then also includes 11 as well. This type of idea of over-writing a given value using itself has been seen already (count variable at the start of this session) and is a very common technique.

```
vec <- c()
for (i in 1:20){
  rand <- rnorm(1, mean = 0, sd = 1) # This generates a standard normal random variabl
  if(rand > 0){
    vec <- c(vec,rand)
  }
}
vec
```

```
##  [1] 0.4897344 1.3745253 0.9537248 1.8231894 0.3634430 1.3206939 1.9156477
##  [8] 1.3641093 0.2786678 0.4536634 0.3400607
```

Alternatively, you could actually save each value in the vector as a particular element, e.g.

```
vec <- c()
vec
```

```
## NULL
```

```
for (i in 1:20){
  rand <- rnorm(1, mean = 0, sd = 1)
  if(rand > 0){
    vec[i] <- rand
  }
}
vec
```

```
##  [1]         NA 0.56492605         NA 0.70239215 1.11076403 1.26745307
##  [7] 1.09352848 1.38469120         NA         NA         NA 0.37496581
## [13]         NA         NA 0.05720042
```

In fact, you could have easily set this up to store all the values in a Matrix rather than a vector

```
(mat <- matrix(c(rep(NA, 16)), nrow = 4))
```

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

```
## [1,]   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA
```

```
for (i in 1:4){
  for (j in 1:4){
  rand <- rnorm(1, mean = 0, sd = 1)
  if(rand > 0){
    mat[i,j] <- rand
  }
  }
}
mat
```

```
##            [,1] [,2]       [,3]      [,4]
## [1,]         NA   NA 0.09028679 1.3999926
## [2,]         NA   NA         NA        NA
## [3,] 0.05985197   NA 1.41950570 0.3116453
## [4,]         NA   NA         NA        NA
```

3.2 While Loops

The final tool we will consider in the area of loops, is the so-called 'WHILE loop'. A While loop is similar to a for loop but instead of simply looping through different values of a specified vector (i in 1:10) it will continue to loop whilst a certain condition holds and will only stop when this condition is no longer satisfied. For example:

```
i <- 1
while (i < 6) {
  print(i)
  i <- i+1
}
```

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

WARNING - Be very careful when using while loops. If you do not write them correctly they can result in your code running infinitely. As an example, try seeing what happens if you forget to increment i to add one each time.

```
i <- 1
while (i < 6) {
```

```
  print(i)
}
```

While loops are very helpful when the number of loops required is unknown. For example, imagine we wanted to find the smallest integer for which the sum of all positive integers up to this value was greater than 1000. This can easily be done using a while loop.

```
i <- 1
sum <- 0

while(sum < 1000){
  sum <- sum + i
  if (sum < 1000){
  i <- i + 1
  } else {
    print(i)
  }
}
```

```
## [1] 45
```

```
sum(1:44)
```

```
## [1] 990
```

```
sum(1:45)
```

```
## [1] 1035
```

Exercise 3.4. Create a variable called `speed` and assign this a rounded random uniform distributed value between 50 - 60, i.e. `round(runif(1, 50, 60))`. Using a while loop, create a code that prints "Your speed is ?? - Slow Down" if speed is greater than 30 then takes 7 off the speed variable. If speed is less than or equal to 30 it should print out "Your speed is ?? - Thank you for not speeding".

Solutions

```
speed <- round(runif(1, 50, 60))
while(speed > 30){
  print(paste("Your speed is", speed, "- Slow Down!"))
  speed <- speed - 7
}
```

```
## [1] "Your speed is 58 - Slow Down!"
## [1] "Your speed is 51 - Slow Down!"
## [1] "Your speed is 44 - Slow Down!"
## [1] "Your speed is 37 - Slow Down!"
```

```
print(paste("Your speed is", speed, "- Thank you for not speeding."))
```

```
## [1] "Your speed is 30 - Thank you for not speeding."
```

I appreciate this is a lot to take in for those who are not familiar with programming but I assure these ideas become second nature with a little practice. We will use them in a larger exercise in the last session so you can see how and when these things would all be used in a practical example. However, for now, I highly recommend that you complete the exercises in DataCamp on conditional statements and loops (Intermediate R) for extra practice.

There are other versions and common commands used in loops, namely `break`, `next` and `repeats`, but I will leave these for you to explore in your own time (ideally via DataCamp). You will need these for the exercises below.

3.3 Exercises

1. Use the command `x <- rexp(20, rate = 0.5)` to create a vector containing 20 simulations of an Exponential random variable with mean 2. Using a loop, return the number of values that are larger than the sample mean of the vector x. You are allowed to use the `mean()` function.
2. Complete Problem 1 again without the use of loops, only 'vectorised calculations'.
3. Write a `while()` loop which prints out the odd numbers from 1 through 7.
4. Using `for()` loops, generate and print the first 20 values of the famous Fibonacci sequence (starting with $0, 1$). Recall, the Fibonacci sequence is obtained by evaluating the next number in the sequence as the sum of the previous two numbers in the sequence.
5. By altering your code in the previous question, use a `while()` loop to determine how many values the Fibonacci sequence contains before its value exceeds 100,000.
6. Use a `while()` loop to determine the smallest value of x such that
$$\prod_{n=1}^{x} n > 10^6.$$
7. Using a `for()` loop, simulate the flip of a fair coin twenty times, keeping track of the individual outcomes (1 = Heads, 0 = Tails) in a vector.

[Hint: You can simulate random numbers that follow given distributions. For example, normal random numbers using rnorm(), exponential using rexp() as seen in Problem 1 or binomial random values using rbinom(). Moreover, the Bernoulli distribution with success

parameter $p \in [0,1]$, which gives a value of 0 or 1, is nothing but a binomial distribution with parameters $n = 1$ and p.]

8. Can you solve the previous problem again without the use of `for()` loops?
9. Using `for()` loops, fill a 5×5 matrix with simulated values from the Poisson distribution having parameter $\lambda = 5$ (`rpois(n, lambda = 5)`). Do this again but without using loops, only 'vectorised calculations'.

Advanced Extension: Can you modify the above to only fill the matrix with simulated values between 1 and 8? Hint: You will have to use the `repeat` and `break` commands.

10. Use a `while()` loop to simulate a stock price path starting at 100 with random normally distributed percentage jumps having mean 0 and standard deviation of 0.01, each day. How many days does it take for the stock price to exceed 150 or drop below 50? Plot the path of the stock price over time using the `plot()` function.

[**Recall: You can simulate a normal random value using the following command `rnorm(n, mean = , sd = )` where n is the number of values you want to simulate**]

Appendix A

Additional Tips

Commenting

Imagine writing a 500 line R code which analyses financial and claim severity data for your company. Within this code, you have assigned a variety of variables, produced countless plots and created numerous different data frames containing the necessary information. Now imagine either of the following scenarios:

- You need to go back through the code to find a particular plot and/or function that computed a certain value of interest that lies somewhere in the middle of your code;
- You have been re-assigned to a different task and have to send your source code to another colleague to take over.

In either case, you will encounter a major problem when it comes to going through lines upon lines of code to find what your looking for, or even remember what you have done. Therefore, to avoid this problem, we insist on the use of commenting when writing R script. To add comments into your script you can simply use the hash tag symbol `#`. R understands that anything on a line following the hash tag is only a comment and will not be executed when run into the console. For example, look at the following code:

As you can see from Figure:@ref{fig:comment1}, when we ran the line of code from the script into the console, R throws up an error. This is due to the fact that R is trying to understand the text following the `curve()` function as a command to be executed but cannot match it to any known functions and/or variables. On the other hand, if we add the commenting symbol `#`, look what happens:

In this case, R executed the initial command/function `curve()` but then did not consider anything after the hash tag as it knows it is simply a comment.

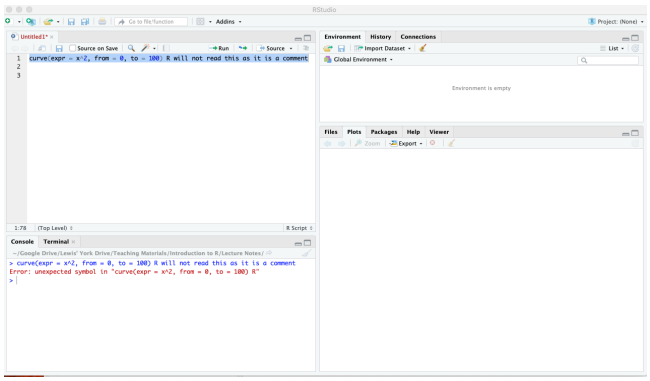


Figure A.1: Error when adding text to code line.

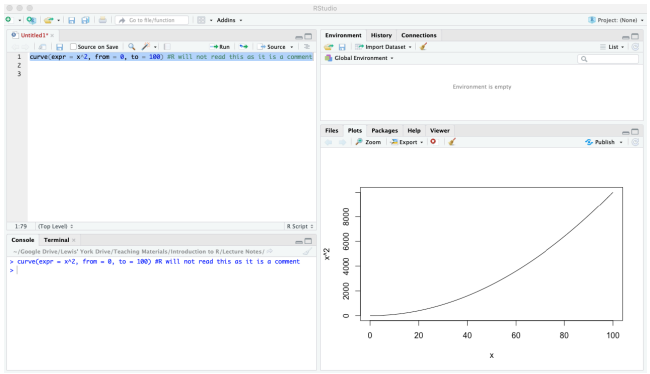


Figure A.2: Comment using hashtag symbol.

This is a very helpful feature that we strongly recommend you use at all times and get into the habit of using early. It will save you a lot of time and hard work further down the line.

Help

The final feature we want to mention in this introductory chapter is the help functionality. Towards the start of this chapter, we discussed the 'Help' tab on the bottom right of the screen and briefly explained how it works. In summary, you can click the 'Help' tab and search for a particular function, e.g. `plot()`. Doing so will bring up an information page detailing the `plot()` function, its possible arguments and some worked examples:

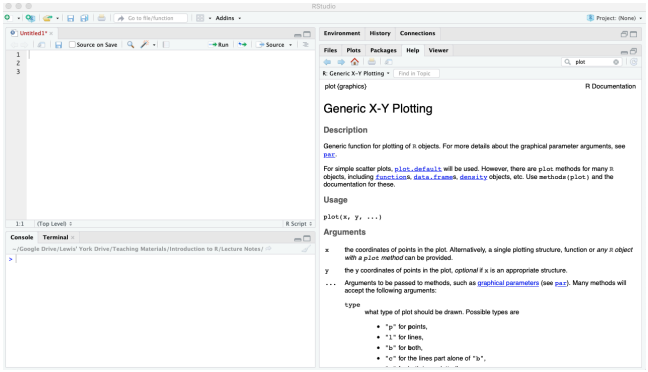


Figure A.3: Using the 'Help' tab for the 'plot()' function.

An alternative way to access this help page is to use the `?` symbol within the script/console. For example, if you know the name of the function you want some more information about, e.g. `plot()`, instead of going into the 'Help' tab, you can simply `?plot()` in your script then run it into the console or, equivalently, type it directly into the console itself. In either case you will produce the same screen as seen in Figure:A.3 above. Finally, if you do not know the function names itself you can conduct a broader search of a specific word using the double question mark symbol, i.e. `??`. For example, assume you wanted to compute the variance but did not already know the associated function was `var()`. Then, you could type `??variance`, which would bring up a list of information pages containing any relevance to variance and you can browse through these as you wish until you find an appropriate function:

Although these 'Help' tools will indeed prove very helpful throughout your programming journey, you cannot emphasise enough the power and ease of simply using a search engine to find answers. There are so many different packages, containing different functions, each of which has several arguments with a list of possible value, it is impossible to learn them. Therefore, browsing the internet

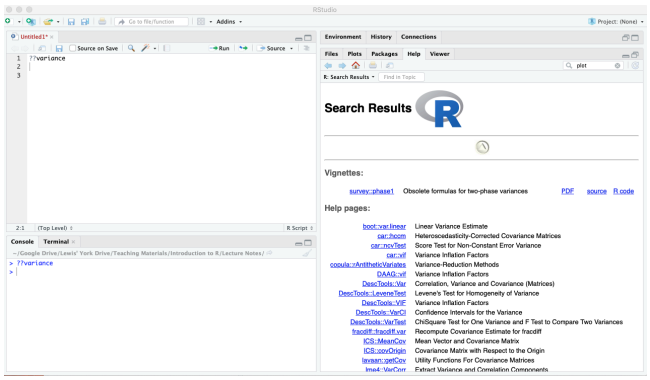


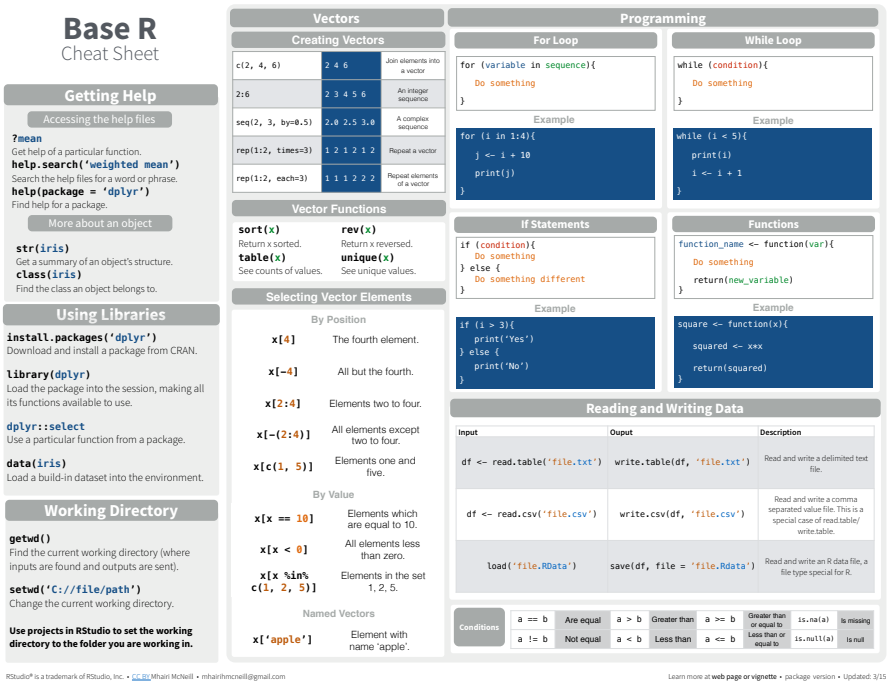
Figure A.4: Using the double question mark symbol for 'Help'.

to search for these functions and how bets to use them is another vital tool at your disposal.

Appendix B

Cheat Sheets

R Cheat Sheets



Base R
Cheat Sheet

Getting Help

Accessing the help files

`?mean`
Get help of a particular function.
`help.search('weighted mean')`
Search the help files for a word or phrase.
`help(package = 'dplyr')`
Find help for a package.

More about an object

`str(iris)`
Get a summary of an object's structure.
`class(iris)`
Find the class an object belongs to.

Using Libraries

`install.packages('dplyr')`
Download and install a package from CRAN.

`library(dplyr)`
Load the package into the session, making all its functions available to use.

`dplyr::select`
Use a particular function from a package.

`data(iris)`
Load a build-in dataset into the environment.

Working Directory

`getwd()`
Find the current working directory (where inputs are found and outputs are sent).

`setwd('C://file/path')`
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	2 4 6	Join elements into a vector
`2:6`	2 3 4 5 6	An integer sequence
`seq(2, 3, by=0.5)`	2.0 2.5 3.0	A complex sequence
`rep(1:2, times=3)`	1 2 1 2 1 2	Repeat a vector
`rep(1:2, each=3)`	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

`sort(x)` Return x sorted.
`rev(x)` Return x reversed.
`table(x)` See counts of values.
`unique(x)` See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 5.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){
  Do something
}
```

Example

```
for (i in 1:4){
  j <- i + 10
  print(j)
}
```

While Loop

```
while (condition){
  Do something
}
```

Example

```
while (i < 5){
  print(i)
  i <- i + 1
}
```

If Statements

```
if (condition){
  Do something
} else {
  Do something different
}
```

Example

```
if (i > 3){
  print('Yes')
} else {
  print('No')
}
```

Functions

```
function_name <- function(var){
  Do something
  return(new_variable)
}
```

Example

```
square <- function(x){
  squared <- x*x
  return(squared)
}
```

Reading and Writing Data

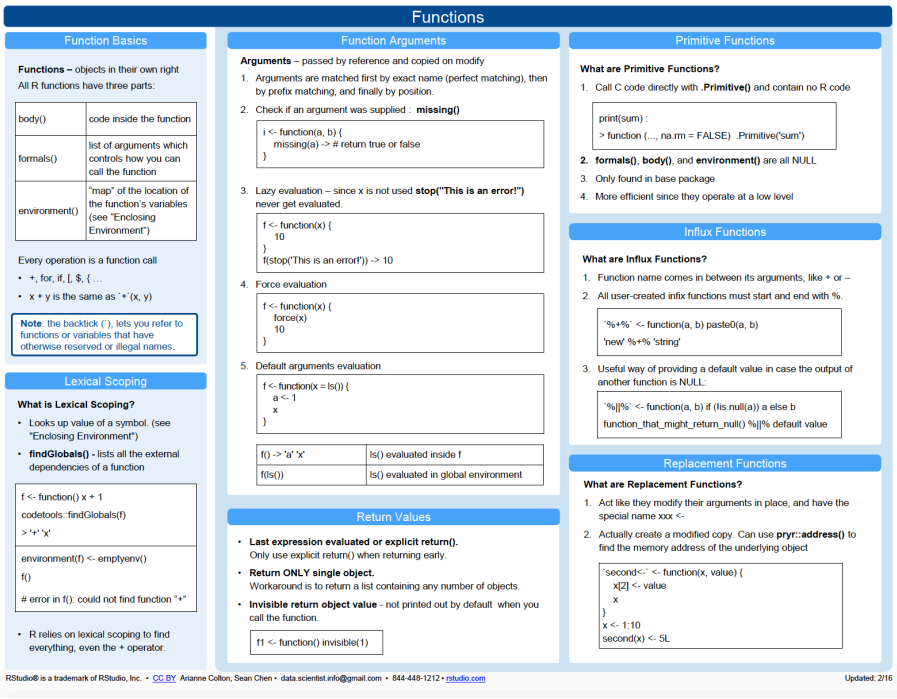
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

Learn more at web page or vignette • package version • Updated: 3/15



Functions

Function Basics

Functions – objects in their own right
All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	"map" of the location of the function's variables (see "Enclosing Environment")

Every operation is a function call

- +, for, if, [, $, { …
- x + y is the same as `+`(x, y)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see "Enclosing Environment")
- **findGlobals()** - lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+"
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'	ls() evaluated inside f
f(ls())	ls() evaluated in global environment

Return Values

- **Last expression evaluated or explicit return().** Only use explicit return() when returning early.
- **Return ONLY single object.** Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE)  .Primitive("sum")
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Influx Functions

What are Influx Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

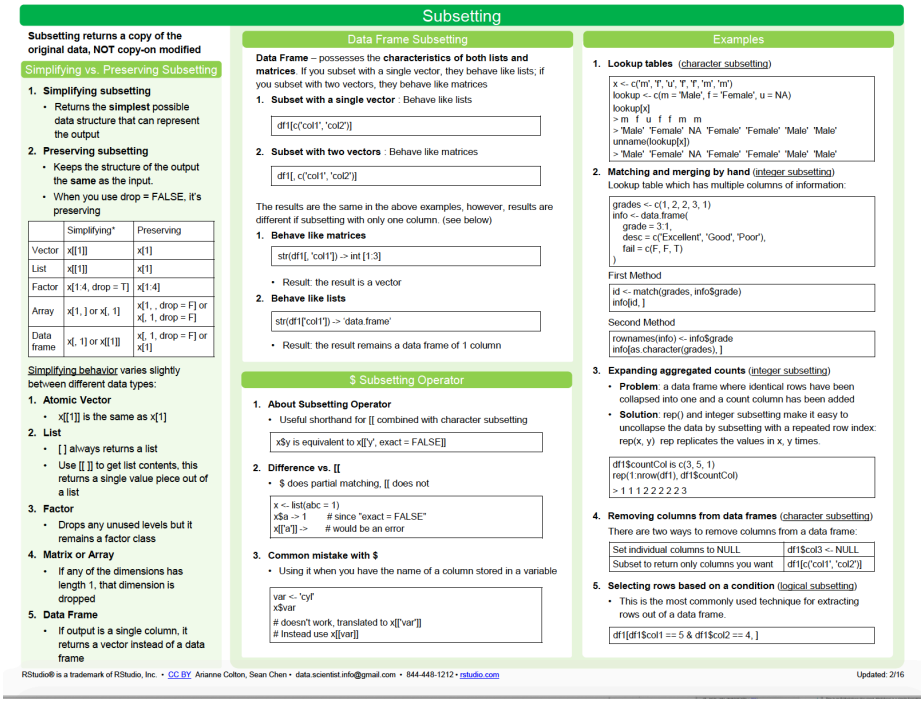
```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```

Subsetting

Subsetting returns a copy of the original data, NOT copy-on modified

Simplifying vs. Preserving Subsetting

1. **Simplifying subsetting**
   - Returns the **simplest** possible data structure that can represent the output
2. **Preserving subsetting**
   - Keeps the structure of the output the **same** as the input.
   - When you use drop = FALSE, it's preserving

	Simplifying*	Preserving
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]
Factor	x[1:4, drop = T]	x[1:4]
Array	x[1,] or x[, 1]	x[1, , drop = F] or x[, 1, drop = F]
Data frame	x[, 1] or x[[1]]	x[, 1, drop = F] or x[1]

Simplifying behavior varies slightly between different data types:

1. **Atomic Vector**
   - x[[1]] is the same as x[1]
2. **List**
   - [ ] always returns a list
   - Use [[ ]] to get list contents, this returns a single value piece out of a list
3. **Factor**
   - Drops any unused levels but it remains a factor class
4. **Matrix or Array**
   - If any of the dimensions has length 1, that dimension is dropped
5. **Data Frame**
   - If output is a single column, it returns a vector instead of a data frame

Data Frame Subsetting

Data Frame – possesses the **characteristics of both lists and matrices**: If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

1. **Subset with a single vector** : Behave like lists

```
df1[c('col1', 'col2')]
```

2. **Subset with two vectors** : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

1. **Behave like matrices**

```
str(df1[, 'col1']) -> int [1:3]
```

   - Result: the result is a vector

2. **Behave like lists**

```
str(df1['col1']) -> 'data.frame'
```

   - Result: the result remains a data frame of 1 column

$ Subsetting Operator

1. **About Subsetting Operator**
   - Useful shorthand for [[combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

2. **Difference vs. [[**
   - $ does partial matching, [[does not

```
x <- list(abc = 1)
x$a -> 1      # since "exact = FALSE"
x[['a']] ->   # would be an error
```

3. **Common mistake with $**
   - Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var
# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

Examples

1. **Lookup tables** (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m   f   u   f   f   m   m
> 'Male'   'Female'  NA  'Female'  'Female'  'Male'  'Male'
unname(lookup[x])
> 'Male'   'Female'  NA  'Female'  'Female'  'Male'  'Male'
```

2. **Matching and merging by hand** (integer subsetting)
Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

3. **Expanding aggregated counts** (integer subsetting)
   - **Problem**: a data frame where identical rows have been collapsed into one and a count column has been added
   - **Solution**: rep() and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: rep(x, y) rep replicates the values in x, y times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

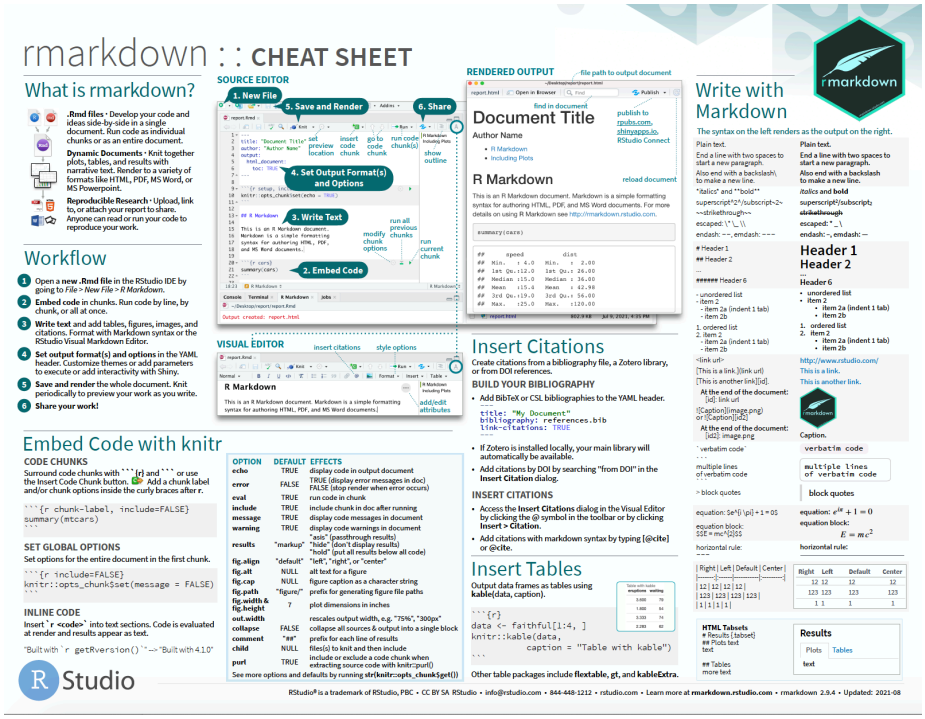
4. **Removing columns from data frames** (character subsetting)
There are two ways to remove columns from a data frame:

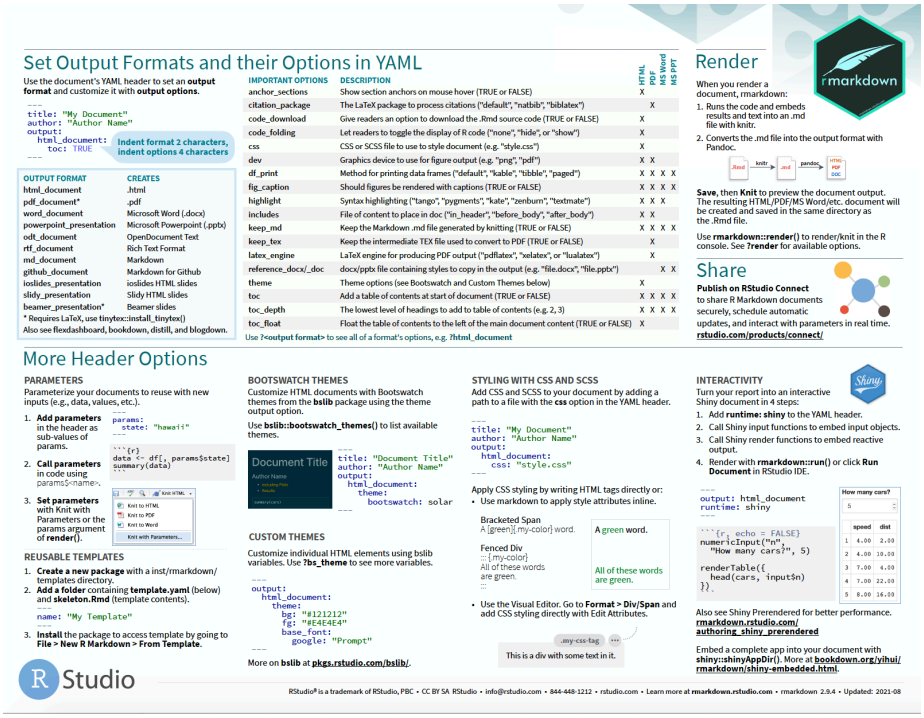
Set individual columns to NULL	df1$col3 <- NULL
Subset to return only columns you want	df1[c('col1', 'col2')]

5. **Selecting rows based on a condition** (logical subsetting)
   - This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```

RMarkdown Cheat Sheets





RMarkdown Guides

Learning how to create documents in RMarkdown for the first time can be quite daunting, especially in this module which requires extensive mathematical formulae, plots and code to be included. To help you with this, I have listed a number of very useful online guides that I use myself when creating RMarkdown documents (including these notes). Be aware that some of them contain alot of information so be careful to only read through the relevant sections:

1. R Markdown: The Definitive Guide
2. R Markdown Cookbook
3. Authoring Books with R Markdown