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.

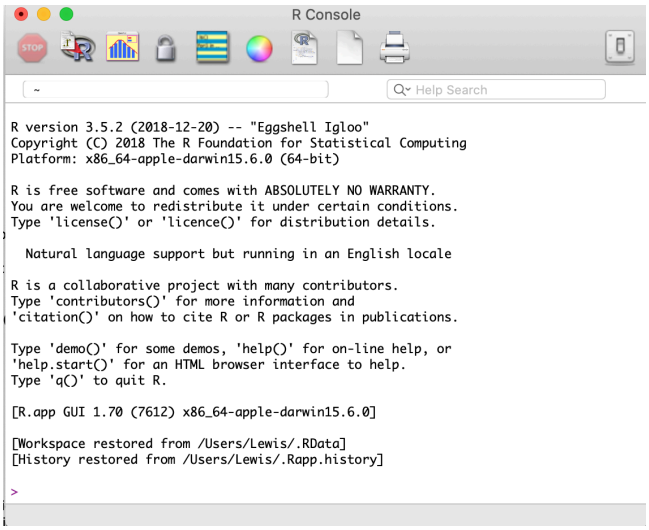


Figure 1.1: R Environment.

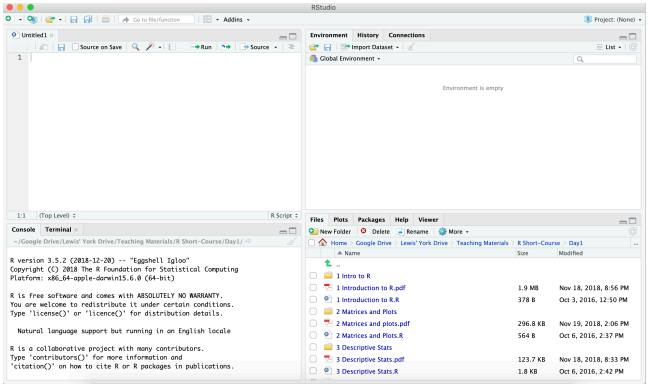


Figure 1.2: RStudio IDE.

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.

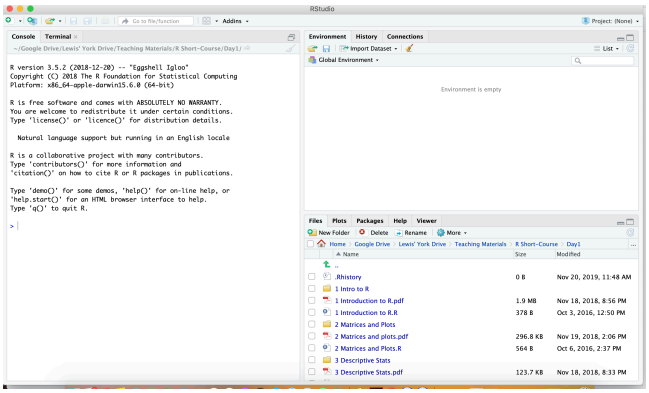


Figure 1.3: Orginal RStudio View.

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:

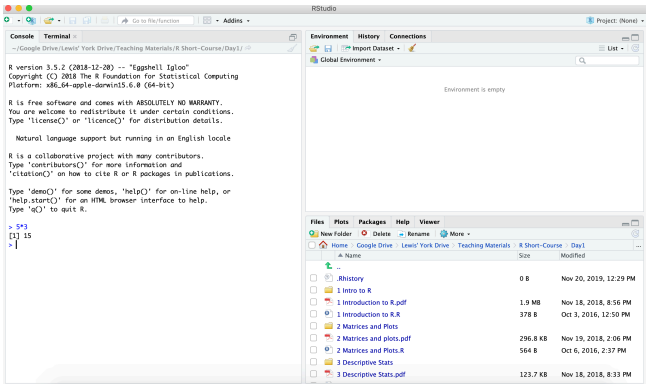


Figure 1.4: Basic Multiplication.

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:

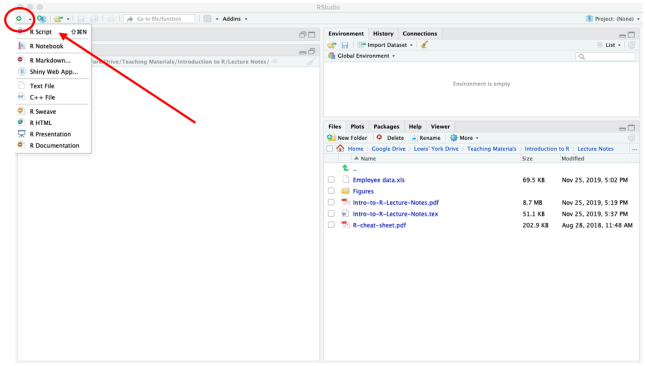


Figure 1.5: Opening an R script.

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.

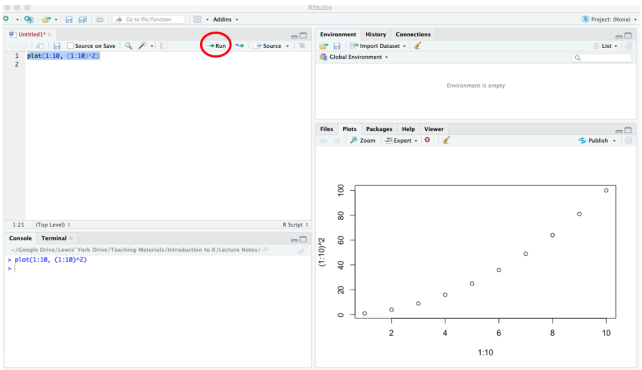


Figure 1.6: Executing code from a script in R.

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] 159.6387 152.1597 173.8923 139.3386 165.5400 183.3106 141.3953 161.2906
##  [9] 144.3464 185.0584 190.6962 188.3825 157.7647 158.6045 157.4677 174.2499
## [17] 154.0448 170.3168 168.9291 165.4991 186.0757 174.0649 199.2892 208.4788
## [25] 159.6440 175.0554 169.5560 167.6970 164.6888 165.8173 150.2013 167.4068
## [33] 130.7729 201.3681 179.2945 151.2633 184.4723 151.1630 172.2308 188.0390
```

```
## [41] 170.0262 175.6983 129.5150 173.5141 194.1047 176.5073 177.0230 165.8140
## [49] 161.5286 162.5032 168.0920 172.3115 184.0901 180.4994 163.5994 166.6334
## [57] 194.0391 162.5077 144.4812 158.8923 175.4724 174.8625 197.2839 186.5450
## [65] 172.8938 180.3151 143.6240 151.7783 195.1111 156.7111 172.2550 165.2559
## [73] 147.3840 204.2713 172.0471 172.9714 160.4252 175.5621 210.7012 163.6677
```

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] 129.5150 130.7729 139.3386 141.3953 143.6240 144.3464 144.4812 147.3840
##  [9] 150.2013 151.1630 151.2633 151.7783 152.1597 154.0448 156.7111 157.4677
## [17] 157.7647 158.6045 158.8923 159.6387 159.6440 160.4252 161.2906 161.5286
## [25] 162.5032 162.5077 163.5994 163.6677 164.6888 165.2559 165.4991 165.5400
## [33] 165.8140 165.8173 166.6334 167.4068 167.6970 168.0920 168.9291 169.5560
## [41] 170.0262 170.3168 172.0471 172.2308 172.2550 172.3115 172.8938 172.9714
## [49] 173.5141 173.8923 174.0649 174.2499 174.8625 175.0554 175.4724 175.5621
## [57] 175.6983 176.5073 177.0230 179.2945 180.3151 180.4994 183.3106 184.0901
## [65] 184.4723 185.0584 186.0757 186.5450 188.0390 188.3825 190.6962 194.0391
## [73] 194.1047 195.1111 197.2839 199.2892 201.3681 204.2713 208.4788 210.7012
```

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

```
## [1] 149.0273
```

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.

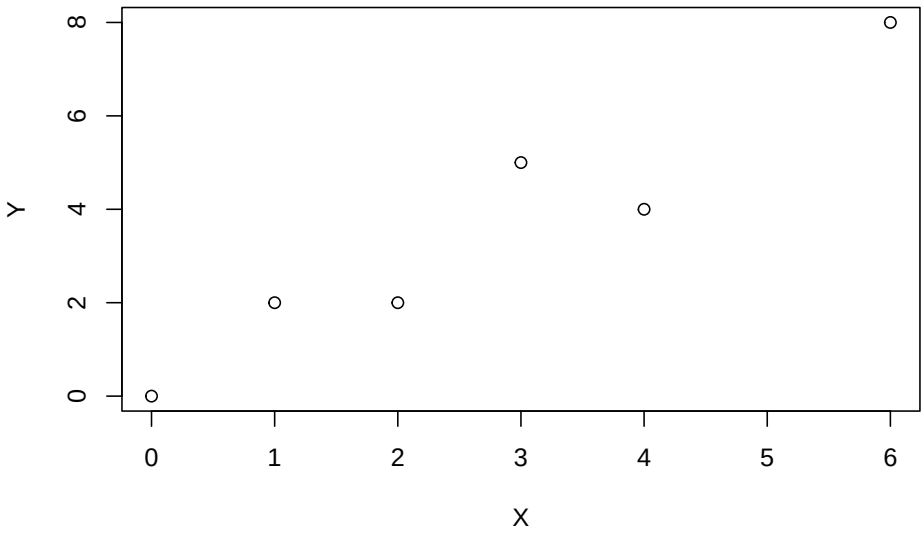
1.7 Plotting graphs

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:

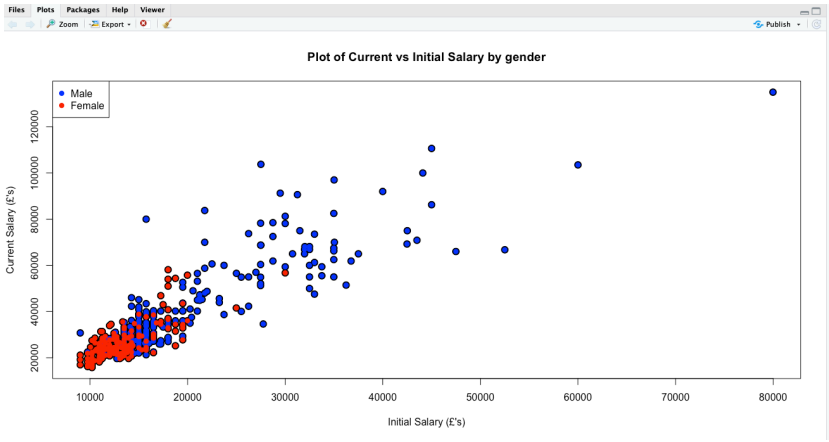


Figure 1.7: Example of Plot using plot().

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

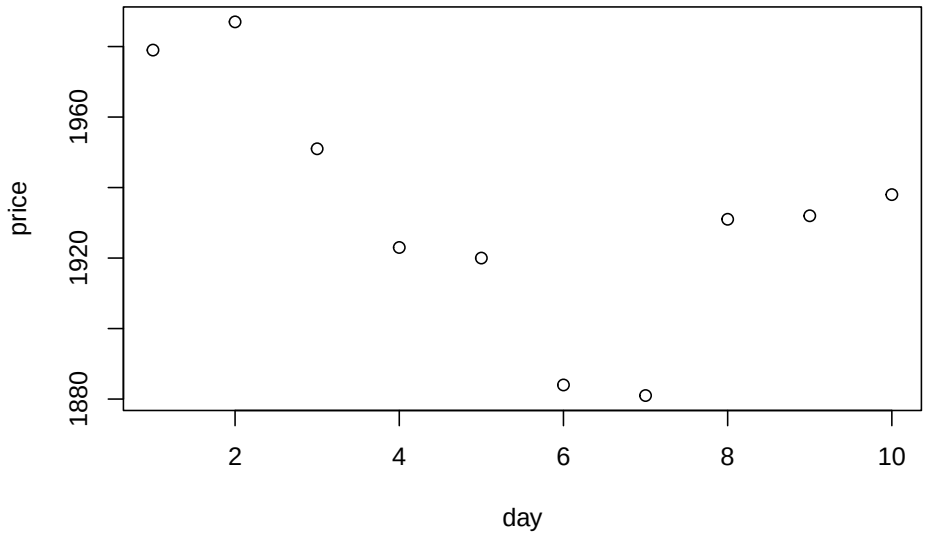
A number of other arguments can be used to change the layout and format of the plot but will not be discussed here. If you are interested, search for the `par()` function in the ‘Help’ tab.

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

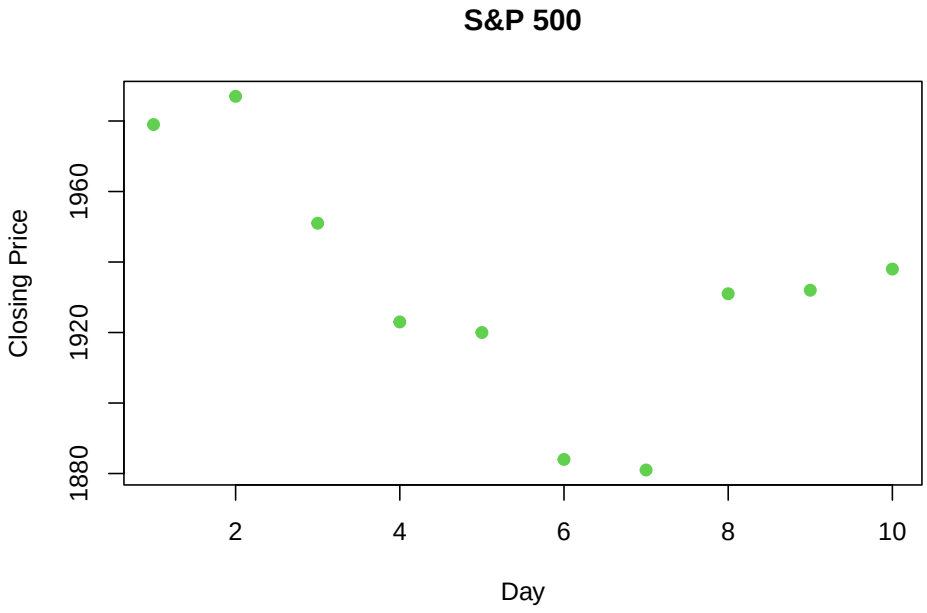
```
day <- c(1:10)
price <- c(1979,1987,1951,1923,1920,1884,1881,1931,1932,1938)
```

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

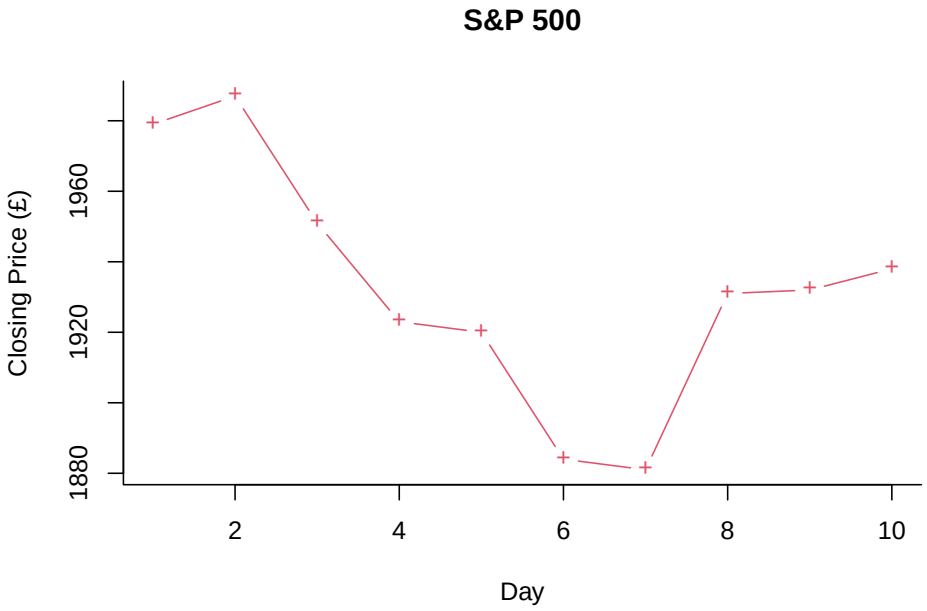
```
plot(day, price)
```



```
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

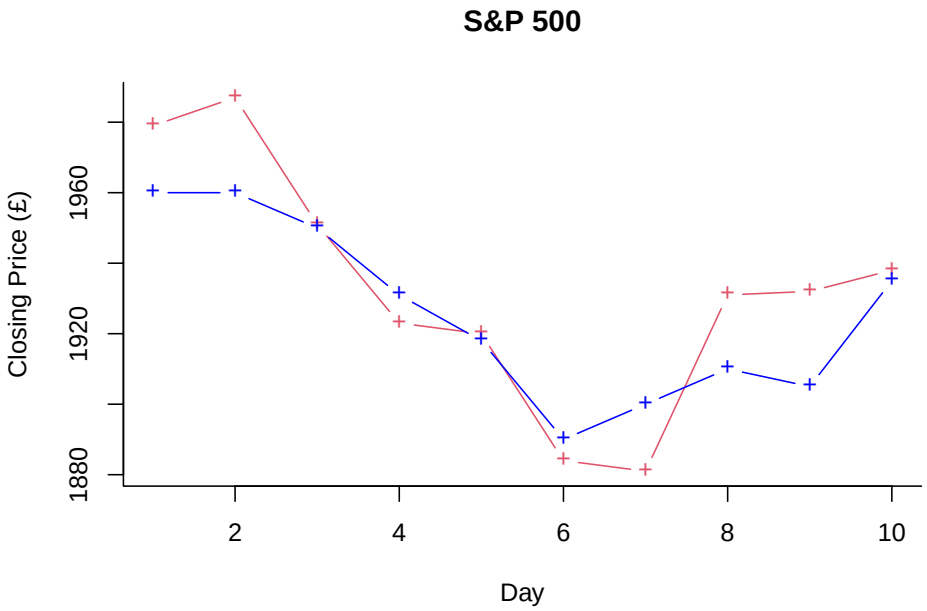
1.7.1 Adding to plots (lines, points etc.)

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

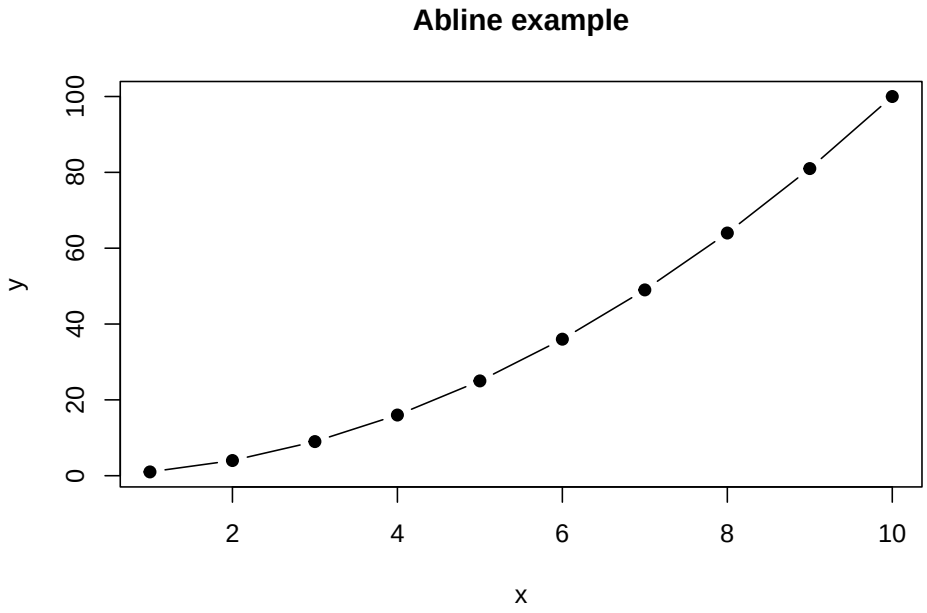
```
abline(a, b, h= , v = , ... )
```

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

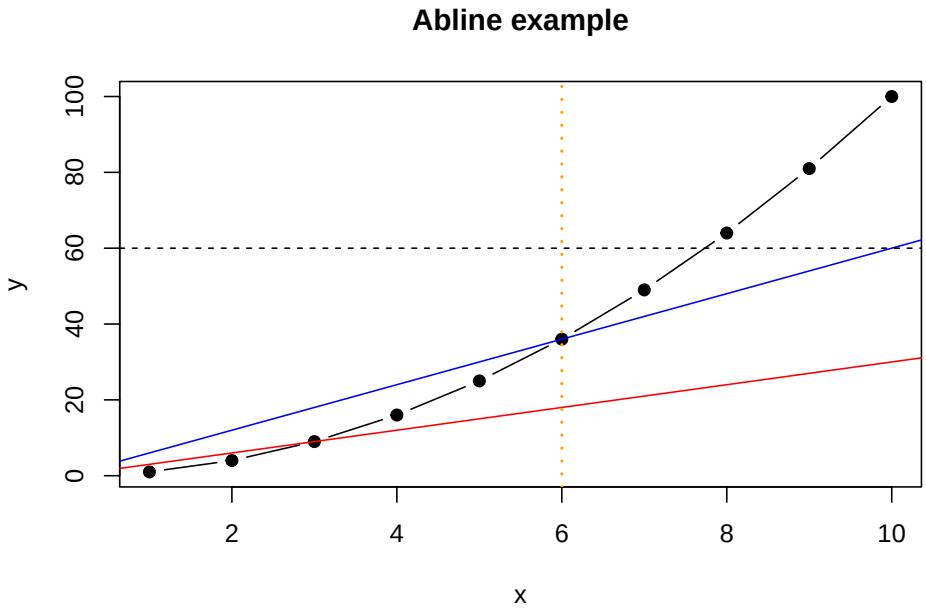
- `a` - The value of the intercept for the straight line
- `b` - The value for the gradient of the straight line
- `h` - The y co-ordinate (intercept) for a horizontal line
- `v` - The x co-ordinate for the vertical line.

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 course(s)

- 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.

2.5 DataCamp course(s)

- https://www.datacamp.com/courses/intermediate-r (Intermediate R Course)

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
```

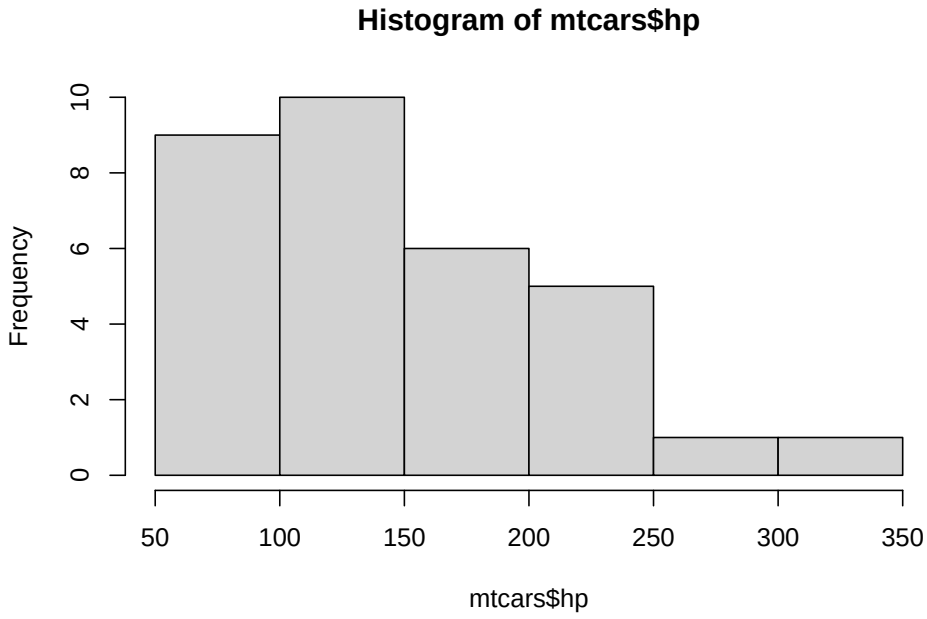
```
for (i in x){
  print(i %% 2 == 0)
}
```

```
## [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
```

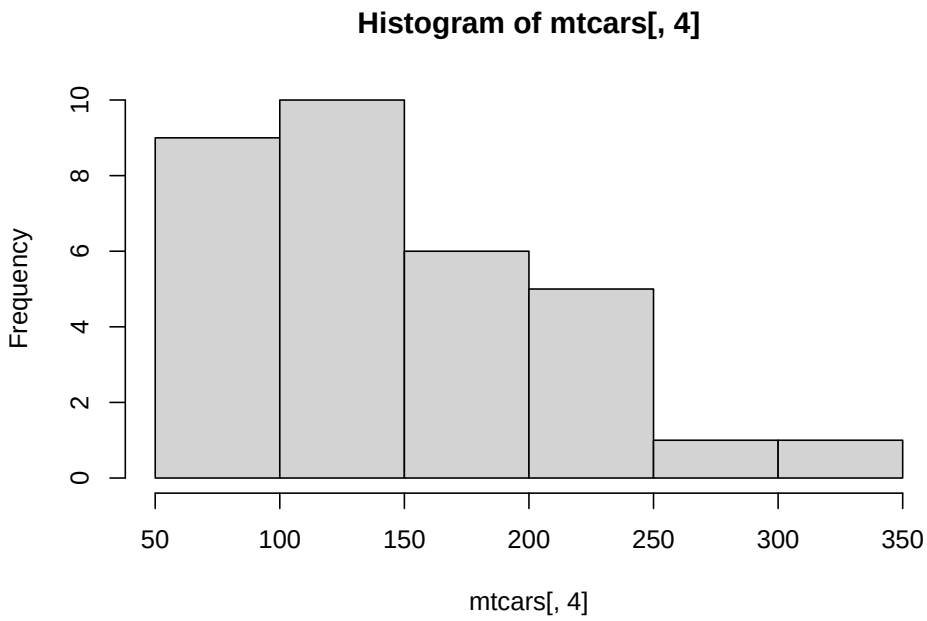
Again, was this necessary or could we have used vectorised calculations again? If possible, you should always use the vectorised calculation version of a command as this saves times and processing power. That being said,there are many situations where 'for loops' are necessary, not just useful.

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)
```

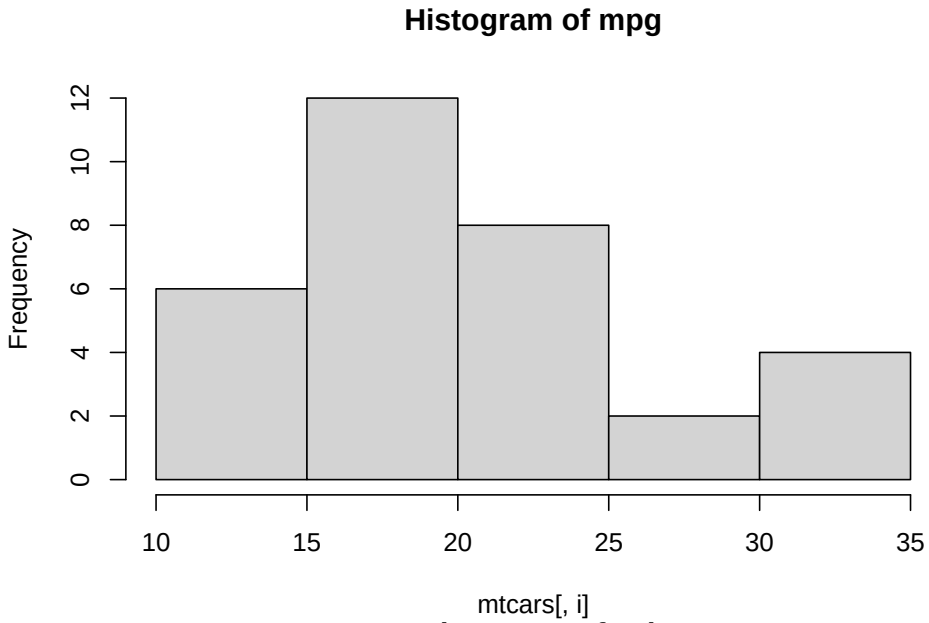


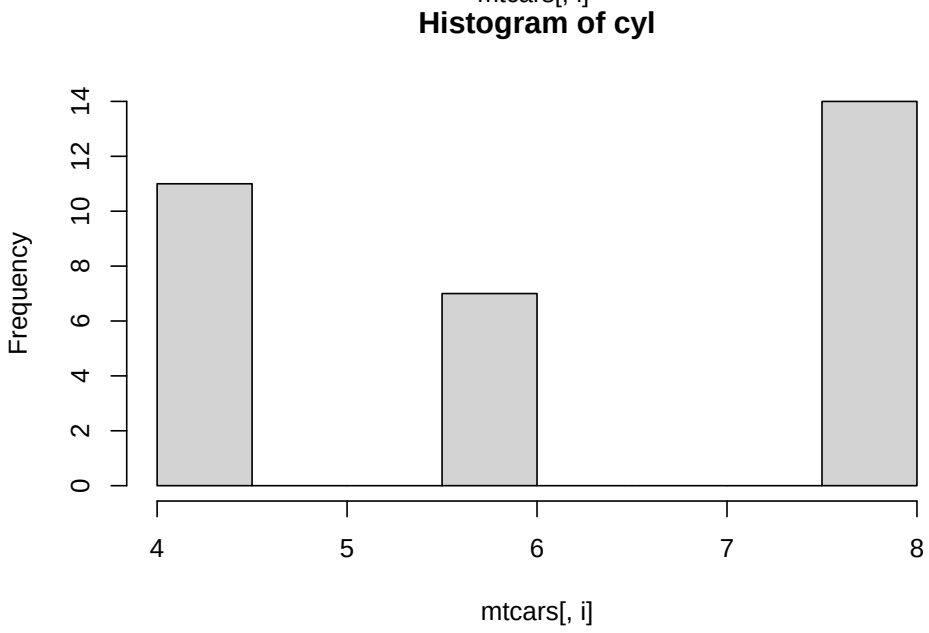

```
hist(mtcars[,4])
```

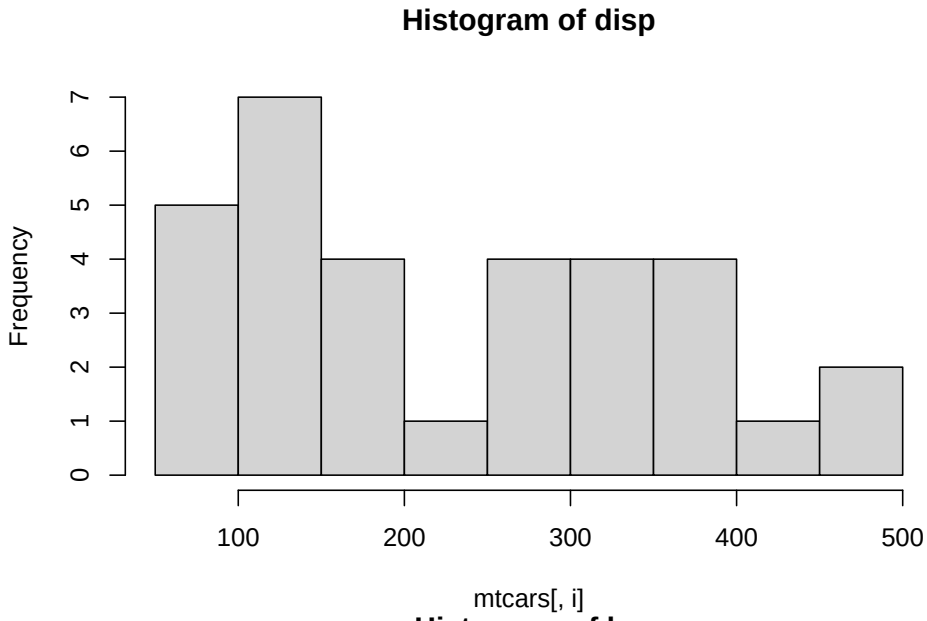


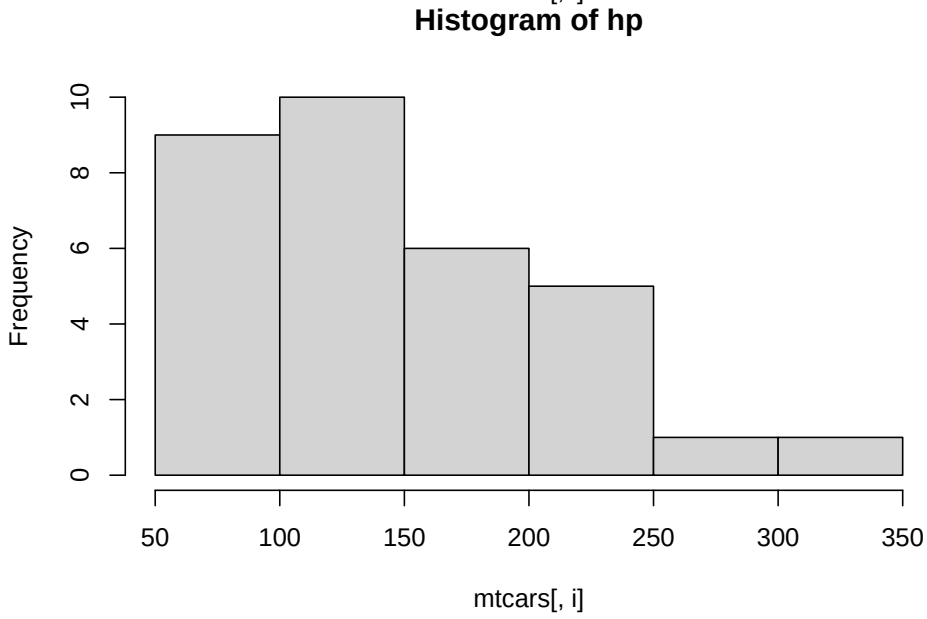
Now, imagine you wanted a histogram for every variable. Executing the code `hist(mtcars)` wouldn't work as the input necessary for this function should be in the form of a single vector of values. However, to overcome this hurdle, we could make use of 'for loops':

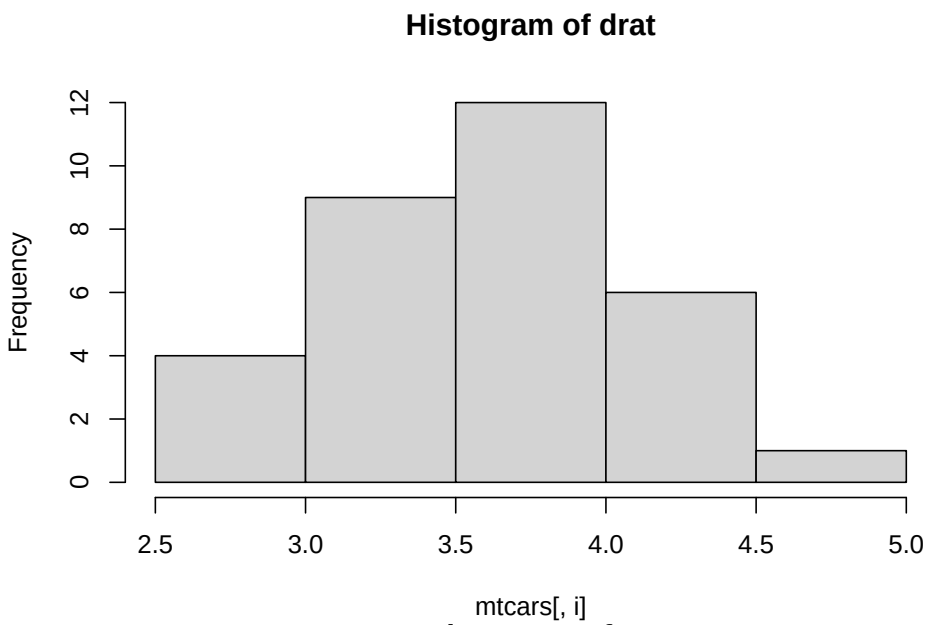
```
for (i in 1:ncol(mtcars)){
  hist(mtcars[,i], main = paste("Histogram of", colnames(mtcars)[i]))
}
```

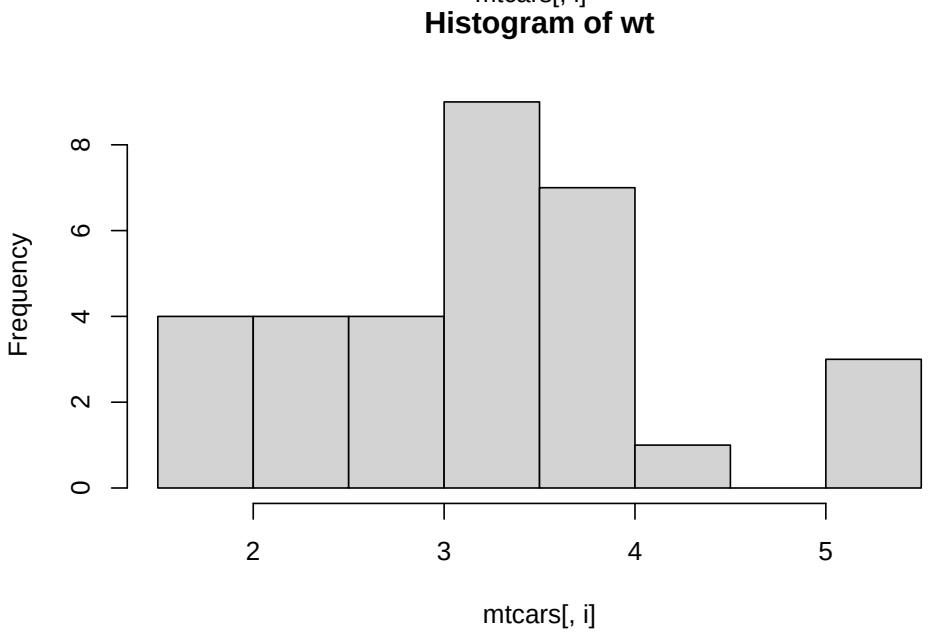


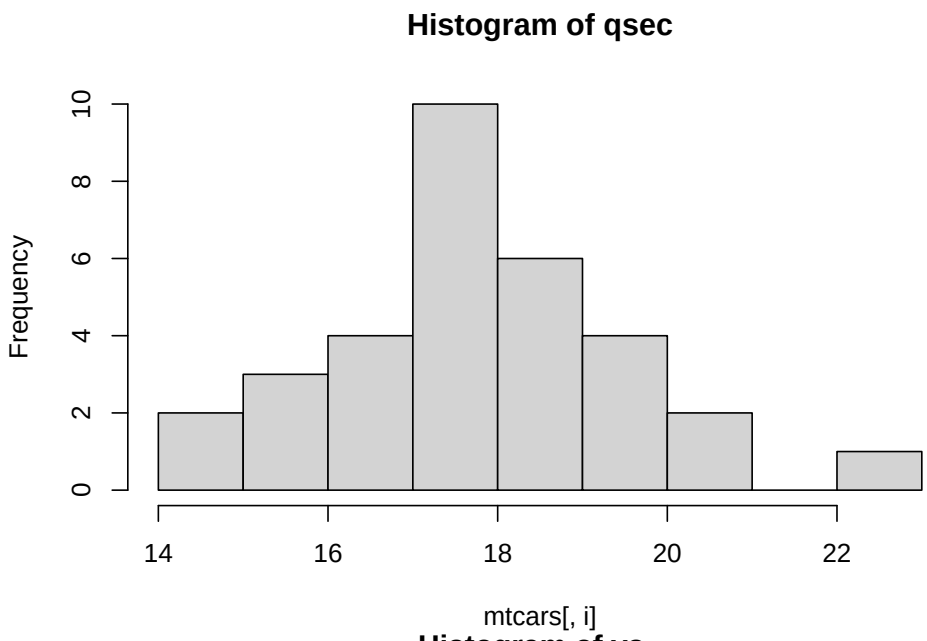


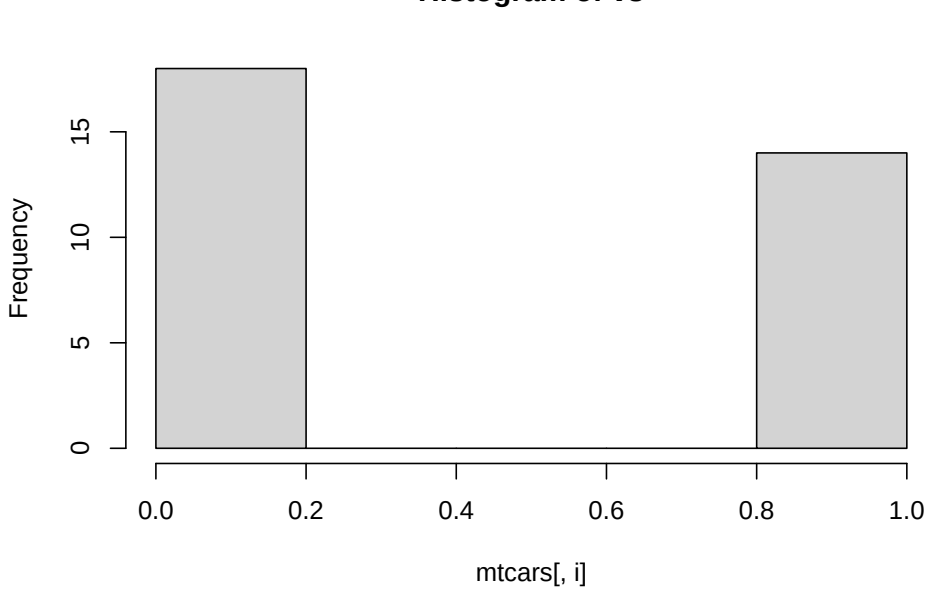

Histogram of disp
Frequency
0
1
2
3
4
5
6
7
100
200
300
400
500
mtcars[, i]

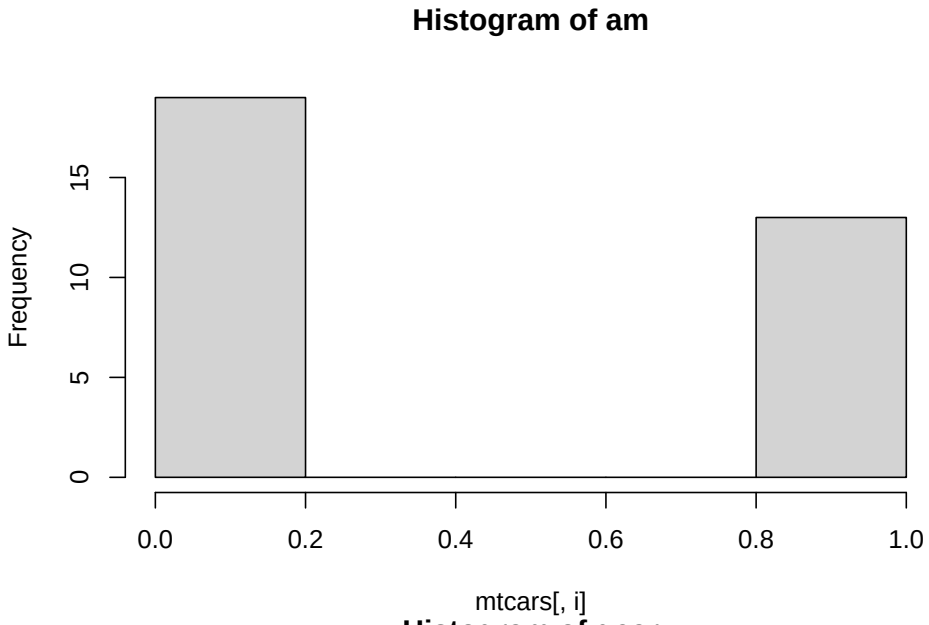

Histogram of hp
Frequency
0
2
4
6
8
10
50
100
150
200
250
300
350
mtcars[, i]

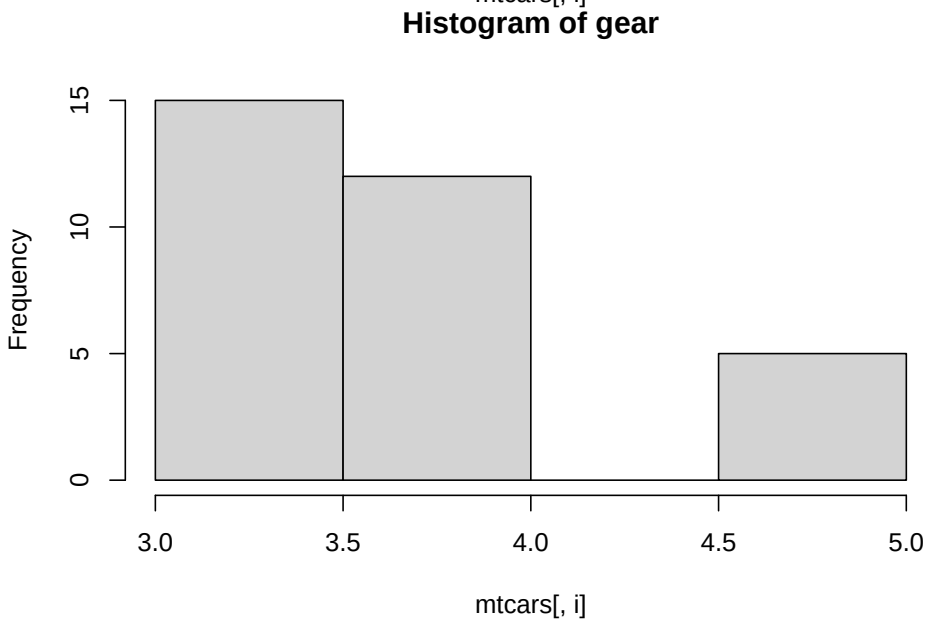

Histogram of drat
Frequency
0
2
4
6
8
10
12
2.5
3.0
3.5
4.0
4.5
5.0
mtcars[, i]

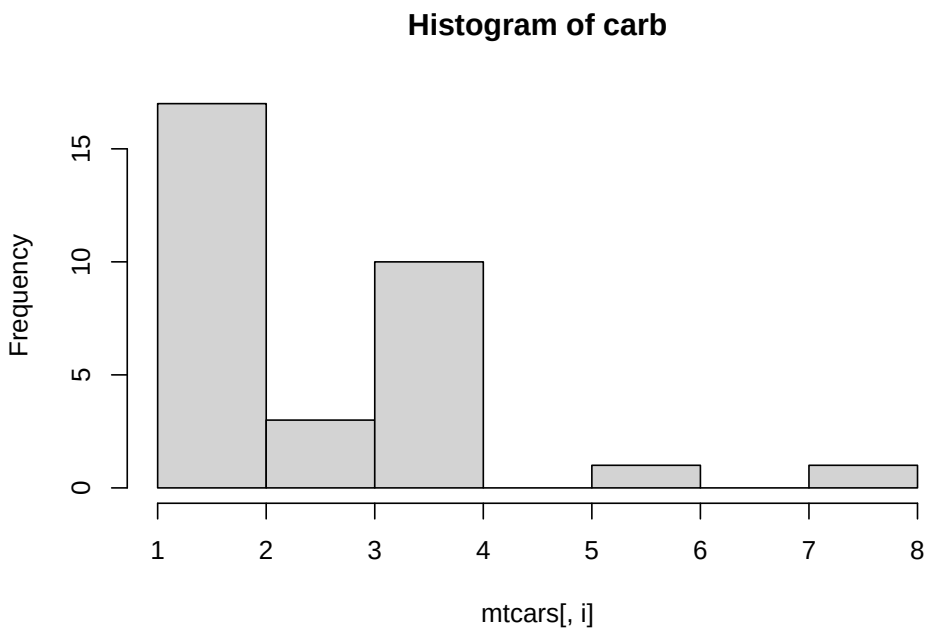

Histogram of wt
Frequency
0
2
4
6
8
2
3
4
5
mtcars[, i]


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]

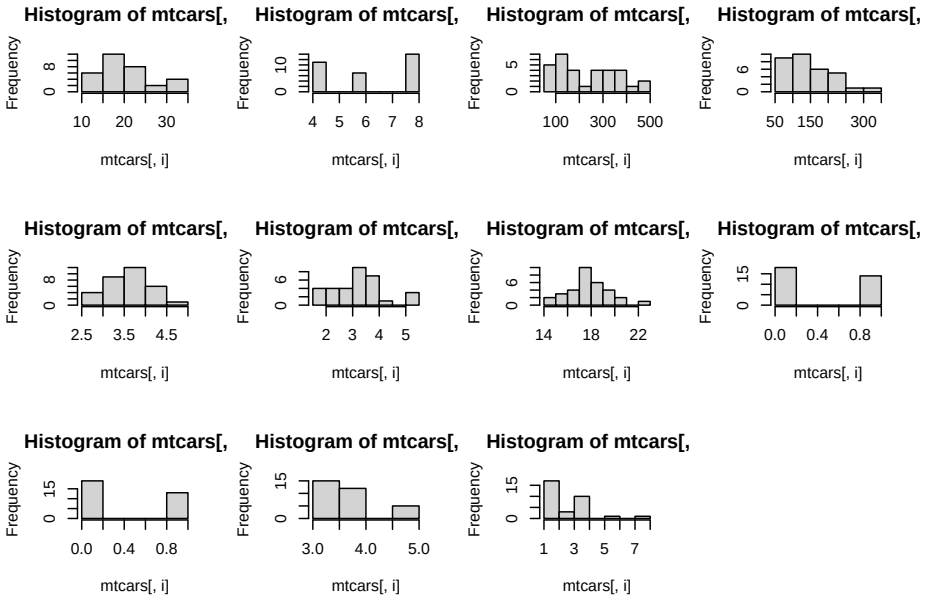

Histogram of am
Frequency
0
5
10
15
0.0
0.2
0.4
0.6
0.8
1.0
mtcars[, i]


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



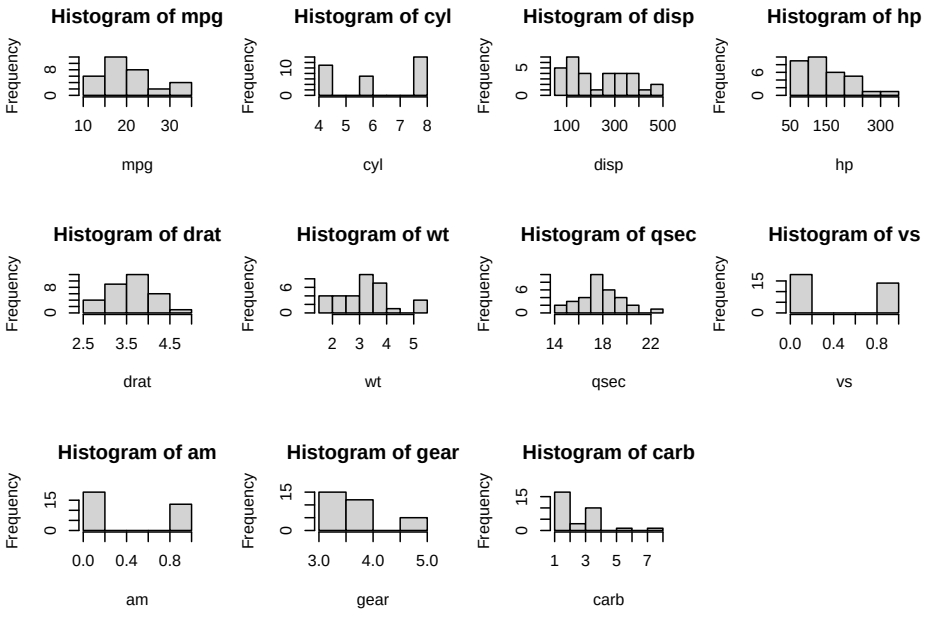
The above is great, but it would be nice to have them all on one screen together.
Note that the code below is not really linked to for loops but is still worth mentioning here.

```
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,]   38   39   32
## [2,]   33   29   57
## [3,]   94   99    1
```

```
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 38"
## [1] "Element [ 1 , 2 ] of M is equal to 39"
## [1] "Element [ 1 , 3 ] of M is equal to 32"
## [1] "Element [ 2 , 1 ] of M is equal to 33"
## [1] "Element [ 2 , 2 ] of M is equal to 29"
## [1] "Element [ 2 , 3 ] of M is equal to 57"
## [1] "Element [ 3 , 1 ] of M is equal to 94"
## [1] "Element [ 3 , 2 ] of M is equal to 99"
## [1] "Element [ 3 , 3 ] of M is equal to 1"
```

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] 0.7072639 1.4447359 0.2036540 0.8807477 0.4590733 1.5047596 0.2908178
## [8] 1.6136259 0.6204394
```

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          NA          NA          NA 1.076445925 0.011270281
##  [7] 0.626182822          NA          NA 1.036040755 0.442182619 0.360372398
## [13]          NA 2.339478099          NA 0.567780546 0.006013551 1.121348261
## [19] 1.037972562
```

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,] 0.1035371   NA         NA 0.97840752
## [2,]        NA   NA 1.35758616 0.98407918
## [3,] 1.2034385   NA 0.09906931 1.35864149
## [4,]        NA   NA 0.56488681 0.05799925
```

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 54 - Slow Down!"
## [1] "Your speed is 47 - Slow Down!"
## [1] "Your speed is 40 - Slow Down!"
```

```
## [1] "Your speed is 33 - Slow Down!"
```

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

```
## [1] "Your speed is 26 - 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**]

3.4 DataCamp course(s)

- https://www.datacamp.com/courses/intermediate-r (Intermediate R Course)

Chapter 4

Functions

In the previous weeks, we have already encountered and worked with some of R's pre-defined functions that you can use on your data/objects to produce certain results. For example, the `mean()` function, `var()` function or even `plot()` function. Each of these 'functions' require one (or more) input variables, then provide some output. Although these functions are readily available for you to use in the base packages, the functions themselves have actually been created from scratch and primarily consist solely of basic programming techniques we have already discussed, e.g. loops, conditional statements etc.

4.1 Creating functions

In this section, we are going to discuss how to create our own functions. There are two main reasons for wanting to create your own functions:

1. To reuse a series of code over and over again without having to re-write the same code (especially if the code is complex and long)
2. For other people to use in their programming (similar to how we have already used some of the functions other people have created)

To create a function in R, we need to following steps:

1. Choose of a name for the function
2. Consider the input variables that will be required for the function
3. Use the following lines of code:

```
functionname <- function(input1, input2, ...){ Commands to execute
for function using input variables listed }
```

As a basic example, let us re-create the `mean()` function ourselves from scratch:

```
mean(1:100) # Remind ourselves how the mean() function works
```

```
## [1] 50.5
```

```
mean_function <- function(x){
  sum(x)/length(x)
}

mean_function(1:100)
```

```
## [1] 50.5
```

```
mean_function(5:5000)
```

```
## [1] 2502.5
```

As you can see in the above, once the function has been created and given a name, in this case `mean_function()` it can now be called and used like any other pre-defined function.

```
(vec <- rexp(100, rate = 1))
```

```
##   [1] 1.20276714 0.61999304 0.30642390 0.68962847 1.22733014 0.16343052
##   [7] 0.10251746 0.42012195 1.33959026 0.06597190 2.00771249 0.03912226
##  [13] 0.23782919 0.87278611 0.94973898 0.14686882 1.80447044 0.79875410
##  [19] 0.11864001 3.84679801 0.90670735 0.24635385 0.14112867 1.11822448
##  [25] 0.19848137 0.04972387 0.76655930 1.21218754 0.53128986 2.59731767
##  [31] 0.16190229 0.76979885 0.37359309 0.12655968 1.04733646 3.42768525
##  [37] 1.00184433 0.13952760 0.15188104 0.16638192 0.32719197 0.67835047
##  [43] 1.08269167 0.78878299 0.91370159 0.16834374 0.08218586 2.63679470
##  [49] 1.13327752 4.27928393 1.44276360 2.57742768 0.65971673 4.76834895
##  [55] 0.03677728 0.15467403 0.34315753 0.36897882 0.22542899 0.47155639
##  [61] 0.03180474 1.71720398 0.38723172 0.10308970 0.46175679 1.52047312
##  [67] 0.23652385 1.02000915 1.61550503 1.17517498 0.53240255 1.40132685
##  [73] 0.81376730 1.42390785 0.47120853 3.98566902 0.42941544 1.62218982
##  [79] 0.17123159 1.48343982 0.22076714 1.15894198 0.42230837 0.02697195
##  [85] 0.50199213 0.32623915 0.14127229 1.68431159 0.88390821 6.25577307
##  [91] 0.25181801 3.04647552 0.11785249 0.97952066 0.02146515 2.71936545
##  [97] 0.27130077 0.92420458 1.14437927 1.61635189
```

```
mean(vec)
```

```
## [1] 0.9855297
```

```
mean_function(vec)
```

```
## [1] 0.9855297
```

As you can see from this simply example, it is even possible to use functions inside functions, e.g. we have used the `sum()` and `length()` functions inside our

newly created function.

Exercise 4.1. Can you create a function called sum_function which sums up all of the values in a vector without using the predefined sum() function?

Solution

```
sum_function <- function(y){
  sum <- 0
  for (i in 1:length(y)){
    sum <- sum + y[i]
  }
  return(sum) # Why this and not print(sum)?
}

sum_function(1:500)
```

```
## [1] 125250
```

Notice that in the above construction of the function we have used `return(sum)`. Why do we do this rather than ask R to `print(sum)`?

```
sum_function_print <- function(x){
  sum <- 0
  for (i in 1:length(x)){
    sum <- sum + x[i]
  }
  print(sum)
}

sum_function(1:10)
```

```
## [1] 55
```

```
sum_function_print(1:10)
```

```
## [1] 55
```

In the above, it does not seem to make any difference if we have used `return(sum)` or `print(sum)` but what if we want to use the function as part of another calculation?

```
sum_function(1:10) + 10
```

```
## [1] 65
```

```
sum_function_print(1:10) + 10
```

```
## [1] 55
```

```
## [1] 65
```

You can see that in the above example, when we have used the version with `print()` in another calculation, we end up with two outputs. This is because we have instructed R that every time the function is executed, it should print the initial result of the function. Then, R also automatically saves the final value for the function and uses it in the remainder of the calculation, resulting in the second output. Unless you specifically want this, you should always use return(), to ensure R only saves the final value to the function. **NOTE: If you only use `return()` R will still display the final output of the function (see the example above).**

Exercise 4.2. Using IF statements, can you create a function that rounds a number to its nearest integer (.5 rounds up)? You **cannot** use the already pre-defined round() function.

Solution

```
round_function <- function(x){
  if(x %% 1 < 0.5){
    return(x-(x%%1))
  } else {
    return(x+1-(x%%1))
  }
}

round_function(9.3)
```

```
## [1] 9
```

```
round_function(9.5)
```

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

Of course, some functions are much more complicated underneath the surface. For example, the `lm()` function executes a full 'linear regression' fitting to a set of data and returns a variety of information about the fitted model:

```
fit <- lm(mtcars$mpg ~ mtcars$hp)
summary(fit)
```

```
##
## Call:
## lm(formula = mtcars$mpg ~ mtcars$hp)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.7121 -2.1122 -0.8854  1.5819  8.2360
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) 30.09886    1.63392  18.421  < 2e-16 ***
## mtcars$hp   -0.06823    0.01012  -6.742 1.79e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.863 on 30 degrees of freedom
## Multiple R-squared:  0.6024, Adjusted R-squared:  0.5892
## F-statistic: 45.46 on 1 and 30 DF,  p-value: 1.788e-07
```

Note: Do not worry about understanding all of this output here, it is included purely for the sake of an example.

Although such functions may look much more complicated, they are still ultimately only made up of combinations of basic commands (albeit many lines of them), the concept is much the same and only requires more thought. As an example of a slightly more complicated function, let us create a prime number calculator:

4.2 Prime number calculator - example

```
prime <- function(number){
  flag <- 0

  if(number == 2){
    flag <- 1
  } else if (number > 2) {
    # check for factors
    flag <- 1
    for(i in 2:(number-1)) {
      if ((number %% i) == 0) {
        flag <- 0
        break
      }
    }
  }

  if(flag == 1) {
    print(paste(number,"is a prime number"))
  } else {
    print(paste(number,"is not a prime number"))
  }
}

prime(7)
```

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

```
prime(986376383)
```

```
## [1] "986376383 is not a prime number"
```

4.3 Multiple Input Variables

In the functions we have created so far, we have only considered one input variable. However, it is possible to create functions with multiple inputs. For example, imagine we wanted to find the accumulated value of an investment over some time period under compound interest. In such a problem, you have three different possible inputs:

1. Initial investment
2. annual interest rate
3. Time (years):

```
Acc_value <- function(initial, interest, years){
  value <- initial*(1+interest)^years
  return(value)
}

Acc_value(100000, 0.05, 25)
```

```
## [1] 338635.5
```

```
Acc_value(100,0.05,10)
```

```
## [1] 162.8895
```

Notice how much easier this is now that we have created a function. Before functions, we would have had to define each variable as a set value, then run the calculation and every time we wanted to calculate it for a new set of values, we would have to change them individually and run it all again, i.e.,

```
initial <- 100000
interest <- 0.05
years <- 25
initial*(1+interest)^years
```

```
## [1] 338635.5
```

Creating functions avoids this tedious problem but also allows us to use them inside other calculations or even other functions (see later).

Before we look at some examples of functions working inside of functions, we note that functions also work with vectors:

```
(x <- seq(0.01, 0.06, by = 0.005))
```

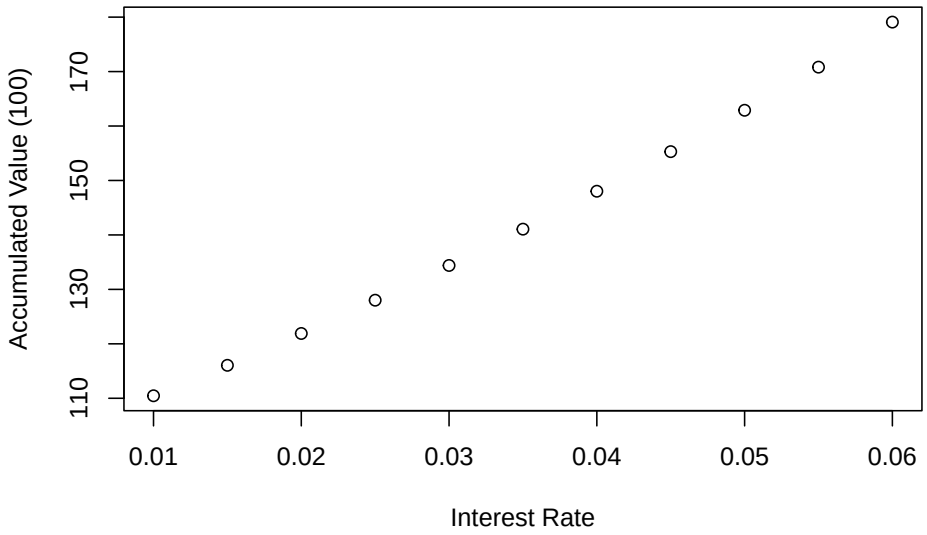
```
##  [1] 0.010 0.015 0.020 0.025 0.030 0.035 0.040 0.045 0.050 0.055 0.060
```

```
Acc_value(100000, x, 25)
```

```
##  [1] 128243.2 145094.5 164060.6 185394.4 209377.8 236324.5 266583.6 300543.4
##  [9] 338635.5 381339.2 429187.1
```

In this case, the function works element-by-element wise in the usual way to create a vector of outputs. To see this implemented inside another function, let us consider the following example of plotting the various accumulated values calculated above using the `plot()` function:

```
plot(x, Acc_value(100,x, 10), ylab = "Accumulated Value (100)", xlab = "Interest Rate")
```



This particular use of the function within the `plot()` function will prove to be very useful for future assessments, where you are typically asked to compare certain quantities under varying conditions (interest rates, terms etc.)

As another example, we recall that there are actually two different types of interest (Simple and Compound). Of course, we could create two separate functions for each of these. However, since these are related it would be nice to have a single function that could deal with both. This can easily by adding a new variable:

```
Acc_value <- function(initial, interest, years, type){
  if (type == "compound"){
  value <- initial*(1+interest)^years
  return (value)
  } else if (type == "simple"){
    value <- initial*(1+(interest*years))
    return (value)
  } else {
```

```
    print("Invalid Interest Type. Must either be 'compound' or 'simple'")
  }
}

Acc_value(100, 0.05, 10, "simple")
```

```
## [1] 150
```

```
Acc_value(100,0.05,10, "compound")
```

```
## [1] 162.8895
```

```
Acc_value(11, 0.05, 10, "comp")
```

```
## [1] "Invalid Interest Type. Must either be 'compound' or 'simple'"
```

An alternative way to do this to let the type variable by a Boolean value as we have seen in other functions:

```
Acc_value <- function(initial, interest, years, compound){
  if (compound == TRUE){
    value <- initial*(1+interest)^years
  } else if (compound == FALSE){
    value <- initial*(1+(interest*years))
  }
  return(value)
}

Acc_value(100, 0.05, 10, compound = FALSE)
```

```
## [1] 150
```

4.3.1 Default options

In some cases, you can have variables within a function that can be changed but more often than not will take a certain value. In this case, you can set a default value for this variable which it will take if not explicitly defined in the function command:

```
Acc_value <- function(initial, interest, years, compound = TRUE){
  if (compound == TRUE){
    value <- initial*(1+interest)^years
  } else if (compound == FALSE){
    value <- initial*(1+(interest*years))
  }
  return(value)
}
```

```
Acc_value(100, 0.05, 10)
```

```
## [1] 162.8895
```

```
Acc_value(100, 0.05, 10, compound = FALSE)
```

```
## [1] 150
```

Finally, just as a nice example of the above application, we could further develop the interest function above:

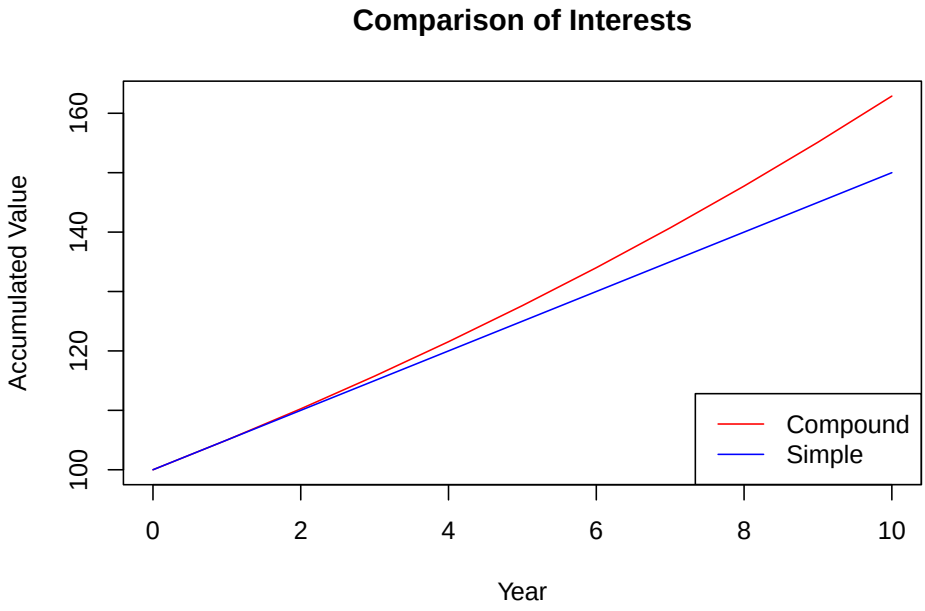
```
Acc_value <- function(initial, interest, years, compound = TRUE, compare = FALSE){

    comp_values <- c(initial)
    for (i in 1:years){
    comp_values <- c(comp_values,initial*(1+interest)^i)
    }
    simp_values <- c(initial)
    for (i in 1:years){
      simp_values <- c(simp_values, initial*(1+(interest*i)))
    }

    if (compare == FALSE){
      if (compound == TRUE){
        return(comp_values[length(comp_values)])
      } else {
        return(simp_values[length(simp_values)])
      }
    }

    if(compare == TRUE){
      x <- 0:years
      plot(x, comp_values, ylab = "Accumulated Value", xlab = "Year", main = "Comparison of Inter
      lines(x, simp_values, type = "l", col = "blue")
      legend("bottomright", legend = c("Compound", "Simple"), col = c("red", "blue"), lty = 1)
    }
}

Acc_value(100, 0.05, 10)
```

```
## [1] 162.8895
```

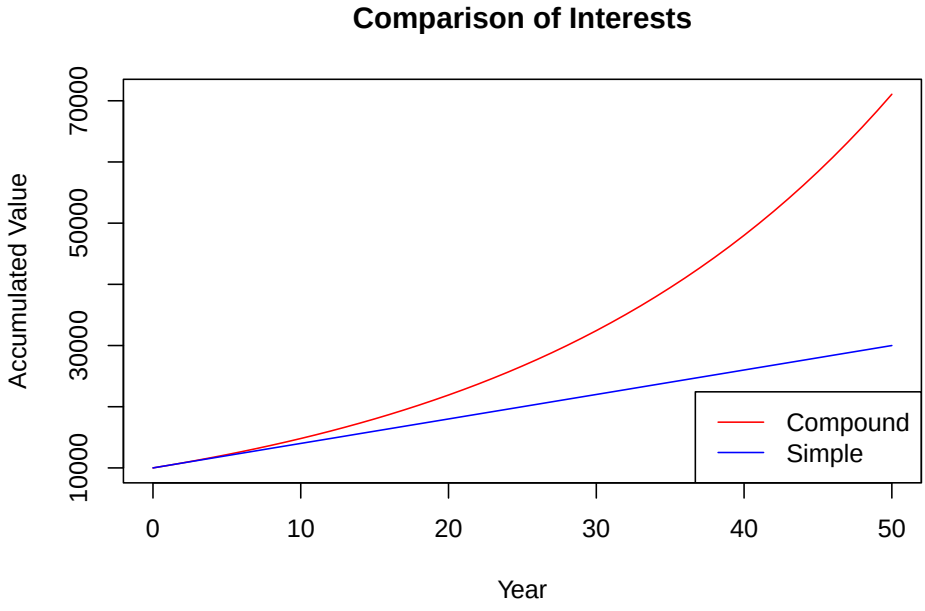
```
Acc_value(100, 0.05, 10, compound = FALSE)
```

```
## [1] 150
```

```
Acc_value(100,0.05, 10, compare = TRUE)
```




```
Acc_value(10000, 0.04, 50, compare = TRUE)
```



Now that you understand the basics of how functions work, try having a go at the following exercises.

4.4 Exercies

1. Recall that in the session, we discussed how to re-create the `sum()` and `mean()` function from programming basics. In a similar way, create a function called `variance` that calculates the variance of a vector of values. You are allowed to use the pre-defined `sum()` and `mean()` functions inside your variance function. Try doing this in two different ways:

i. Using For loops
ii. Using vectorised calculations.

2. Create a function that, given an integer, will calculate how many divisors it has (other than 1 and itself). Make the divisors appear on the screen.

3. From your 'Introduction to Actuarial Science' module, you should have come across the concept of 'discounting' and the 'present value' of money. Create a function in R called `PV` that takes 3 input variables: 1) Final value (F), 2) Annual interest rate and 3) Number of years, which calculates the present value of F.

4. Recall that for geometric summation, we have

$$\sum_{k=0}^{n-1} x^k = \frac{1-x^n}{1-x}.$$

Moreover, if $|x| < 1$, the above summation converges as $n \to \infty$, such that

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

Create a function in R called `GeomSum` that takes two input variables (x and n) and calculates the geometric sum of x from 0 up to n. It should also be possible to include the option that $n = \infty$. **Hint: You may have to include a Boolean value for this but remember, the above limit only exists under a given condition. If this condition is not satisfied, make the function print out a warning message.**

5. Recall from your 'Introduction to Actuarial Science' module that the 'Accumulated Value' of an annuity-due with unit payments is defined by

$$\ddot{s}_{n\rceil} = (1+i)^n + (1+i)^{n-1} + \cdots + (1+i) = \frac{(1+i)^n - 1}{i} \times (1+i)$$

Create a function that takes three input variables representing 1) The value of repeated payments, 2) The annual interest rate and 3) The number of years. The function should create a vector with the accumulated value of the investment after each year and plot it on a basic plot against time (see the R Script for a similar example).

6. Recall the Stock price example from Problem Sheet 2 (Problem 10). Create a function called `Stock` that allows the user to input a starting amount, the standard deviation of percentage change (assume the change is normally distributed $N(0, \sigma^2)$) and the values of an upper and lower barrier. The function should then plot the movement of the stock and print out the number of days it takes to reach either the upper or lower barrier.

4.5 DataCamp course(s)

- https://www.datacamp.com/courses/intermediate-r (Intermediate R Course)

Chapter 5

Data Analysis

In this session we will discuss how to create data frames (objects similar to matrices which are typically used to store data) manually (explaining how this differs from a matrix), methods of extracting and manipulating data effectively and, finally, use of the `apply()` function(s).

5.1 Creating a Data Frame

We have already been introduced to the idea of a data frame in the previous sessions through the 'mtcars' data set:

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

However, in general, a data frame is simply an array of data, where each row denotes a separate data point or observation and each column a different variable. Why are these different from matrices in R?

The main difference between a 'matrix' and a 'dataframe' in R is what can be

stored within them. We have already stated that a matrix (like a vector) can consist of numerical values OR Boolean values OR character strings. However, what I have not yet mentioned is they cannot be a mixture of these things:

```
V1 <- c(1, 2, 3)
str(V1)
```

```
##  num [1:3] 1 2 3
```

```
V2 <- c("Hello", "Red", "UK")
str(V2)
```

```
##  chr [1:3] "Hello" "Red" "UK"
```

```
V3 <- c(T, T, F)
str(V3)
```

```
##  logi [1:3] TRUE TRUE FALSE
```

```
(V <- c(V1, V2, V3))
```

```
## [1] "1"     "2"     "3"     "Hello" "Red"   "UK"    "TRUE"  "TRUE"  "FALSE"
```

```
str(V)
```

```
##  chr [1:9] "1" "2" "3" "Hello" "Red" "UK" "TRUE" "TRUE" "FALSE"
```

```
(M <- matrix(c(V1, V2, V3), nrow = 3, ncol = 3, byrow = FALSE))
```

```
##      [,1] [,2]    [,3]
## [1,] "1"  "Hello" "TRUE"
## [2,] "2"  "Red"   "TRUE"
## [3,] "3"  "UK"    "FALSE"
```

```
str(M)
```

```
##  chr [1:3, 1:3] "1" "2" "3" "Hello" "Red" "UK" "TRUE" "TRUE" "FALSE"
```

As you can see in the above, since a matrix can only consists of objects of the same type, R has automatically re-assigned the objects to match with one of the object types. Again, this is a perfect example of why you need to be so careful when programming as your code may appear to work but the reality is quite different.

Data frames on the other hand, allow us to create a matrix like structure but each column may take on a different variable format. This is ideal when working and storing data. Let us look at how to create a data frame using the above vectors V1, V2 and V3:

```
data <- data.frame(Numeric = V1, Characters = V2, Boolean = V3)
knitr::kable(data)
```

Numeric	Characters	Boolean
1	Hello	TRUE
2	Red	TRUE
3	UK	FALSE

```
str(data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ Numeric   : num  1 2 3
##  $ Characters: chr  "Hello" "Red" "UK"
##  $ Boolean   : logi  TRUE TRUE FALSE
```

What do you notice about the character strings in this data frame? It is important to decide if any 'words' or 'character strings' in your data set are simply words or if they resemble different factors of a given group. In the latter case, you need to tell R this! This is very important when applying a variety of statistical functions to your data.

```
data1 <- data.frame(Numeric = V1, Characters = V2, Boolean = V3, stringsAsFactors = F)
str(data1)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ Numeric   : num  1 2 3
##  $ Characters: chr  "Hello" "Red" "UK"
##  $ Boolean   : logi  TRUE TRUE FALSE
```

```
data2 <- data.frame(Numeric = V1, Characters = V2, Boolean = V3, stringsAsFactors = T)
str(data2)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ Numeric   : num  1 2 3
##  $ Characters: Factor w/ 3 levels "Hello","Red",..: 1 2 3
##  $ Boolean   : logi  TRUE TRUE FALSE
```

```
rownames(data1) <- c("Observation 1", "Observation 2", "Observation 3")
knitr::kable(data1)
```

	Numeric	Characters	Boolean
Observation 1	1	Hello	TRUE
Observation 2	2	Red	TRUE
Observation 3	3	UK	FALSE

Now that we have created our data frame, we can begin to analyse the data in any way we choose! We will discuss a little later on more complex ways in which we may want to do this. For now, let us look at one more example to make sure we understand how data frames are created:

```
Height <- rgamma(20, shape = 70, rate = 0.4) # Simulates/generates Gamma Distributed variables
head(Height)
```

```
## [1] 156.1684 208.7195 171.3375 162.9003 167.7923 156.4146
```

```
Weight <- rnorm(20, mean = 75, sd = 10)
head(Weight)
```

```
## [1] 72.23670 87.27186 80.91796 62.30064 76.78156 89.31763
```

```
Age <- rpois(20, lambda = 20) # Simulates/generates Poisson Distributed variables
Age
```

```
##  [1] 21 21 22 21 11 20 21 18 19 24 17 24 19 11 16 25 17 21 25 10
```

```
Sex<- ifelse(rbinom(20,1,prob=0.5) == 1, "Male", "Female") # How has this worked?

Data <- data.frame(Height_cm = Height, Weight_Kg = Weight, Age_Years = Age, Sex = Sex,
knitr::kable(Data)
```

Height_cm	Weight_Kg	Age_Years	Sex
156.1684	72.23670	21	Female
208.7195	87.27186	21	Female
171.3375	80.91796	22	Female
162.9003	62.30064	21	Male
167.7923	76.78156	11	Female
156.4146	89.31763	20	Male
141.5089	67.20490	21	Female
141.6228	58.53537	18	Male
186.4411	76.92566	19	Male
178.1800	74.25898	24	Female
185.2370	93.34134	17	Male
177.8322	86.00147	24	Female
150.8432	81.08779	19	Male
167.7093	67.20133	11	Male
168.1175	85.01000	16	Female
166.1480	71.82074	25	Male
172.9431	84.84126	17	Female
181.0348	56.74887	21	Male
178.2115	70.12923	25	Male
173.1805	73.17381	10	Male

```
str(Data)
```

```
## 'data.frame':    20 obs. of  4 variables:
##  $ Height_cm: num  156 209 171 163 168 ...
##  $ Weight_Kg: num  72.2 87.3 80.9 62.3 76.8 ...
##  $ Age_Years: int  21 21 22 21 11 20 21 18 19 24 ...
##  $ Sex      : Factor w/ 2 levels "Female","Male": 1 1 1 2 1 2 1 2 2 1 ...
```

```
mean(Data$Height_cm)
```

```
## [1] 169.6171
```

```
mean(Data$Height_cm > 170)
```

```
## [1] 0.5
```

Another helpful tool worth mentioning here when analysing data in a data frame is the `table()` function, which counts the frequency of different observations and displays them in a table format:

```
table(Data$Age_Years)
```

```
##
## 10 11 16 17 18 19 20 21 22 24 25
##  1  2  1  2  1  2  1  5  1  2  2
```

```
table(Data$Age_Years,Data$Sex)
```

```
##
##      Female Male
##   10      0    1
##   11      1    1
##   16      1    0
##   17      1    1
##   18      0    1
##   19      0    2
##   20      0    1
##   21      3    2
##   22      1    0
##   24      2    0
##   25      0    2
```

5.2 Importing data - Excel

In most cases when working with data, the observations will have been collected and stored in another programme which is better built for data collection, e.g. Excel. Fortunately, R can import such data easily and will save the imported values into a data frame type object (known as a tibble) automatically. It is also possible to save it directly as a data frame explicitly if preferred. At this stage, the differences between tibble and a data frame are not important.

To do this, use the following steps:

1. Click the `Import Dataset` option within the Environment window (Top right)
2. Click 'From Excel…' (You may be asked to install some packages here, if so press 'Yes' or 'Okay')
3. Click 'Browse' to enter your files
4. Choose the Excel file containing the Data
5. Edit the Dataset name (if necessary)

6. Choose which sheet you want to import from the Excel file (if necessary)
7. Decide if you need to skip any rows due to format
8. Tick first row as names if appropriate
9. Press import
10. Re-save as a data frame (if necessary)

In this example I will import four different data sets (Male and Female Deaths and Population in UK) from the 'UK(Pop&Death).xls' data set and name them 1) `Male_UK_Death`, 2) `Female_UK_Death`, 3) `Male_UK_Pop` and 4) `Female_UK_Pop`, respectively.

Clearly I cannot show you the above steps explicitly in these notes, but they are the steps that have been used to load the data here. Below is an example of what the corresponding code that is executed looks like when following these steps (you may find that your directory path where you have saved the data is different but that should be it):

```
library(readxl)
Male_UK_Death <- read_excel("UK(Pop&Death).xls", sheet = "UK male deaths")
Female_UK_Death <- read_excel("UK(Pop&Death).xls", sheet = "UK female deaths")
Male_UK_Pop <- read_excel("UK(Pop&Death).xls", sheet = "UK male pop")
Female_UK_Pop <- read_excel("UK(Pop&Death).xls", sheet = "UK female pop")
```

Note: It is also possible to import data from a number of other sources (SPSS, Stata, SAS, .csv files etc.)

5.3 Manipulating and analysing data

In many cases, the data set that we have imported may not be exactly how we want it for our analysis. There are a huge number of things that you may want/need to do to your data to tidy it up or 'clean' the data as it is commonly known. For example, deal with missing data, extract unimportant rows/columns, delete outliers, combine data sets etc.

As an example, let us have a little look at the data sets we created:

Age	1961	1962	1963	1964	1965	1966	1967	1968	1969	1970	1971
0	12047	12707	12408	11989	11325	11105	10254	10319	9894	9713	9366
1	785	727	788	684	671	726	649	644	644	590	531
2	484	423	437	406	407	471	435	439	453	394	374
3	305	300	324	289	354	350	340	381	311	314	279
4	261	258	297	262	280	305	266	289	305	258	255
5	218	217	235	247	255	258	251	259	265	218	248
6	209	202	223	196	230	203	212	230	204	241	220
7	193	187	205	187	202	164	185	196	214	208	209
8	161	148	182	163	183	193	192	153	167	149	192
9	177	151	175	161	170	169	165	192	148	162	186
10	134	141	170	139	153	140	185	142	155	169	173
11	154	159	130	143	145	153	135	158	139	146	147
12	171	158	151	155	163	163	148	154	148	151	137
13	183	205	166	167	182	171	178	143	178	149	178
14	221	230	196	200	178	193	202	210	180	158	199
15	217	276	241	229	235	227	221	228	201	209	211
16	309	301	375	400	369	429	345	335	325	348	345
17	413	396	372	524	524	480	423	405	409	402	402
18	399	449	434	463	567	535	475	410	455	400	399
19	374	424	453	474	442	597	503	410	418	393	445
20	395	386	439	438	421	433	513	433	385	475	419
21	357	337	392	422	404	404	415	479	453	443	446
22	397	381	347	345	374	432	365	367	454	439	396
23	349	365	342	364	345	384	351	355	345	447	440
24	335	358	368	344	354	336	332	340	362	327	390
25	334	340	324	368	345	299	352	311	373	344	339
26	367	305	363	329	340	343	346	309	349	338	352
27	347	301	343	325	341	325	326	301	310	357	340
28	349	314	344	360	349	331	320	313	292	318	334
29	357	373	351	343	363	393	286	342	345	308	309
30	435	376	359	359	372	353	376	307	365	343	319
31	394	385	382	382	338	402	361	354	346	387	347
32	432	402	448	409	405	372	361	374	381	386	389
33	442	430	434	444	412	418	380	354	384	389	392
34	490	491	480	470	478	473	411	421	450	415	403
35	604	545	524	512	535	539	468	413	451	446	485
36	606	583	559	591	603	571	542	503	476	491	509
37	675	647	697	634	630	566	544	559	531	532	517
38	750	766	733	751	716	709	660	660	636	579	528
39	872	874	809	835	771	779	728	700	689	672	665
40	1022	983	964	912	879	885	824	793	850	792	780
41	1111	1106	1031	1009	1052	964	922	888	945	842	889
42	972	1303	1362	1189	1157	1158	1065	1105	985	1038	990
43	990	974	1402	1358	1322	1258	1187	1257	1187	1146	1111
44	1198	1077	1149	1532	1520	1422	1313	1342	1366	1298	1261
45	1488	1385	1269	1279	1739	1724	1571	1448	1575	1482	1458
46	1648	1579	1453	1379	1436	1917	1889	1767	1785	1615	1668
47	1941	1947	1922	1718	1493	1645	2117	2111	2019	1881	1897
48	2117	2166	2156	1850	1877	1656	1716	2322	2461	2201	2082
49	2455	2301	2400	2421	2212	2058	1788	1920	2795	2692	2453
50	2644	2609	2792	2657	2599	2484	2212	1965	2169	2936	2910
51	2977	3017	2924	2841	2992	2805	2518	2504	2254	2350	3183
52	3415	3433	3415	3328	3292	3351	3064	2997	2754	2462	2501

We would like to remove the last two rows of this data set as they don't contain 'raw' data.

```
Male_UK_Death_New <- Male_UK_Death[-c(107, 108),]
knitr::kable(Male_UK_Death_New)
```

Age	1961	1962	1963	1964	1965	1966	1967	1968	1969	1970	1971	1972	197
0	12047	12707	12408	11989	11325	11105	10254	10319	9894	9713	9366	8393	778
1	785	727	788	684	671	726	649	644	644	590	531	556	52
2	484	423	437	406	407	471	435	439	453	394	374	357	36
3	305	300	324	289	354	350	340	381	311	314	279	310	29
4	261	258	297	262	280	305	266	289	305	258	255	237	25
5	218	217	235	247	255	258	251	259	265	218	248	226	23
6	209	202	223	196	230	203	212	230	204	241	220	234	22
7	193	187	205	187	202	164	185	196	214	208	209	209	18
8	161	148	182	163	183	193	192	153	167	149	192	188	17
9	177	151	175	161	170	169	165	192	148	162	186	167	16
10	134	141	170	139	153	140	185	142	155	169	173	147	12
11	154	159	130	143	145	153	135	158	139	146	147	161	16
12	171	158	151	155	163	163	148	154	148	151	137	143	14
13	183	205	166	167	182	171	178	143	178	149	178	159	15
14	221	230	196	200	178	193	202	210	180	158	199	191	19
15	217	276	241	229	235	227	221	228	201	209	211	208	22
16	309	301	375	400	369	429	345	335	325	348	345	287	27
17	413	396	372	524	524	480	423	405	409	402	402	403	41
18	399	449	434	463	567	535	475	410	455	400	399	435	47
19	374	424	453	474	442	597	503	410	418	393	445	446	42
20	395	386	439	438	421	433	513	433	385	475	419	389	46
21	357	337	392	422	404	404	415	479	453	443	446	449	43
22	397	381	347	345	374	432	365	367	454	439	396	446	43
23	349	365	342	364	345	384	351	355	345	447	440	413	42
24	335	358	368	344	354	336	332	340	362	327	390	395	44
25	334	340	324	368	345	299	352	311	373	344	339	439	41
26	367	305	363	329	340	343	346	309	349	338	352	372	45
27	347	301	343	325	341	325	326	301	310	357	340	341	37
28	349	314	344	360	349	331	320	313	292	318	334	318	39
29	357	373	351	343	363	393	286	342	345	308	309	345	39
30	435	376	359	359	372	353	376	307	365	343	319	343	39
31	394	385	382	382	338	402	361	354	346	387	347	360	31
32	432	402	448	409	405	372	361	374	381	386	389	344	36
33	442	430	434	444	412	418	380	354	384	389	392	378	37
34	490	491	480	470	478	473	411	421	450	415	403	421	46
35	604	545	524	512	535	539	468	413	451	446	485	470	50
36	606	583	559	591	603	571	542	503	476	491	509	491	54
37	675	647	697	634	630	566	544	559	531	532	517	527	52
38	750	766	733	751	716	709	660	660	636	579	528	593	54
39	872	874	809	835	771	779	728	700	689	672	665	650	61
40	1022	983	964	912	879	885	824	793	850	792	780	699	70
41	1111	1106	1031	1009	1052	964	922	888	945	842	889	862	83
42	972	1303	1362	1189	1157	1158	1065	1105	985	1038	990	963	99
43	990	974	1402	1358	1322	1258	1187	1257	1187	1146	1111	1117	107
44	1198	1077	1149	1532	1520	1422	1313	1342	1366	1298	1261	1257	119
45	1488	1385	1269	1279	1739	1724	1571	1448	1575	1482	1458	1461	148
46	1648	1579	1453	1379	1436	1917	1889	1767	1785	1615	1668	1684	156
47	1941	1947	1922	1718	1493	1645	2117	2111	2019	1881	1897	1800	179
48	2117	2166	2156	1850	1877	1656	1716	2322	2461	2201	2082	2051	206
49	2455	2301	2400	2421	2212	2058	1788	1920	2795	2692	2453	2429	239
50	2644	2609	2792	2657	2599	2484	2212	1965	2169	2936	2910	2773	266
51	2977	3017	2924	2841	2992	2805	2518	2504	2254	2350	3183	3231	298
52	3415	3433	3415	3328	3292	3351	3064	2997	2754	2462	2501	3696	346

Then, in a similar way, we can do the same with the rest of the data sets as they have same problem

```
Female_UK_Death_New <- Female_UK_Death[-c(107, 108),]
Male_UK_Pop_New <- Male_UK_Death[-c(107, 108),]
Female_UK_Pop_New <- Female_UK_Pop[-c(107, 108),]
```

We have now tidied up our data to be in a format more beneficial to us and we could start to analyse these individually, i.e. Create plots, calculate statistics, fit statistical models etc. For example:

```
Aux <- Male_UK_Death_New[-106,]
summary(lm(Aux$Age ~ Aux$`1961`))
```

```
##
## Call:
## lm(formula = Aux$Age ~ Aux$`1961`)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -81.594 -17.102  -2.754   6.391  62.101
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 4.189e+01  3.686e+00  11.367  < 2e-16 ***
## Aux$`1961`  3.296e-03  7.947e-04   4.147 6.94e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28.33 on 103 degrees of freedom
## Multiple R-squared:  0.1431, Adjusted R-squared:  0.1348
## F-statistic:  17.2 on 1 and 103 DF,  p-value: 6.937e-05
```

However, what if we were not interested in the individual data sets separated by sex and only cared about the overall deaths and population. That is, we want to combine the data sets. In this case, we need to create a new data frame with the relevant data.

```
Overall_Deaths <- data.frame(Male_UK_Death_New$Age, Male_UK_Death_New[,-1] + Female_UK_
colnames(Overall_Deaths) <- colnames(Male_UK_Death_New)
knitr::kable(Overall_Deaths)
```

Age	1961	1962	1963	1964	1965	1966	1967	1968	1969	1970	1971	1972	197
0	20885	21821	21568	20857	19571	19235	18075	17748	17083	16715	16164	14591	134
1	1335	1325	1417	1250	1254	1329	1218	1191	1183	1067	1001	1027	93
2	831	732	774	741	757	807	769	771	766	684	636	672	62
3	572	568	579	533	604	603	552	652	527	528	503	567	52
4	448	447	525	459	511	537	468	518	517	458	428	432	41
5	376	414	405	418	430	419	400	425	433	400	410	381	40
6	322	334	352	322	368	337	342	370	316	389	359	381	36
7	314	300	319	310	323	278	305	321	317	333	333	322	29
8	274	244	292	272	274	302	309	285	277	253	300	321	27
9	273	227	255	251	257	263	273	298	234	251	302	284	25
10	224	222	253	224	243	232	288	238	221	273	293	240	23
11	248	243	199	229	268	259	237	231	232	235	225	239	26
12	273	271	250	254	259	256	246	260	225	263	224	220	22
13	302	307	273	278	285	258	280	259	277	234	281	229	23
14	345	348	293	304	301	291	303	317	268	235	304	289	29
15	333	408	361	353	347	382	347	346	302	309	318	326	36
16	446	419	532	592	534	559	487	477	456	465	493	437	43
17	564	536	511	683	706	666	589	535	571	540	562	576	57
18	550	612	582	622	748	717	668	560	600	545	543	591	64
19	531	575	603	662	641	832	690	573	593	555	607	619	61
20	527	543	582	621	581	616	688	611	552	681	581	556	62
21	506	497	559	597	578	555	583	663	620	622	605	611	61
22	583	542	494	493	545	601	524	526	666	627	575	626	59
23	538	544	511	517	507	562	528	512	515	665	610	597	58
24	528	532	502	495	520	508	516	497	564	493	586	580	63
25	513	523	514	566	532	467	514	498	535	528	519	657	63
26	552	505	551	514	528	534	500	467	536	539	517	548	65
27	507	487	548	509	532	520	502	476	491	528	531	517	58
28	540	531	532	558	564	554	521	495	507	494	524	514	58
29	584	608	593	583	577	593	484	540	523	494	488	556	59
30	690	648	596	594	616	570	593	532	563	561	507	550	58
31	675	683	632	606	560	636	603	580	548	605	559	533	51
32	728	712	777	681	637	607	595	631	667	634	650	572	61
33	772	720	776	753	696	700	628	626	615	647	658	625	62
34	839	847	841	790	807	810	690	707	735	673	676	696	73
35	986	960	915	851	903	906	806	737	770	722	767	777	79
36	1016	1035	1016	952	983	938	925	837	825	818	828	838	86
37	1181	1120	1145	1092	1088	1001	937	983	912	905	920	908	88
38	1326	1360	1303	1273	1244	1185	1105	1129	1092	1012	928	985	92
39	1548	1506	1441	1358	1352	1314	1267	1218	1203	1174	1108	1094	106
40	1780	1677	1611	1559	1544	1510	1410	1379	1409	1370	1350	1210	122
41	1988	1955	1793	1692	1789	1604	1565	1513	1613	1475	1483	1467	141
42	1661	2251	2325	2100	2024	1959	1799	1850	1784	1721	1655	1645	170
43	1741	1728	2446	2364	2223	2172	2034	2071	2060	1962	1872	1869	181
44	2068	1865	1957	2658	2584	2385	2245	2247	2322	2177	2091	2103	197
45	2507	2314	2134	2121	2917	2916	2647	2506	2548	2466	2413	2469	245
46	2876	2665	2498	2371	2456	3218	3192	2940	2933	2743	2780	2769	262
47	3285	3309	3205	2886	2524	2739	3531	3493	3396	3232	3182	3005	299
48	3618	3610	3604	3121	3057	2779	2785	3885	4014	3756	3466	3382	336
49	3984	3902	4159	4021	3681	3413	2982	3190	4553	4347	4052	4027	386
50	4353	4283	4503	4343	4276	4145	3653	3269	3578	4835	4673	4545	431
51	4832	4803	4723	4577	4742	4624	4116	4048	3669	3856	5169	5301	472
52	5527	5441	5499	5246	5298	5372	4957	4841	4526	4065	4186	5921	564

```
Overall_Pop <- data.frame(Male_UK_Pop_New$Age, Male_UK_Pop_New[,-1]+Female_UK_Pop_New[
colnames(Overall_Pop) <- colnames(Male_UK_Pop_New)
knitr::kable(Overall_Pop)
```

Age	1961	1962	1963	1964	1965	1966	1967	1968	1969	1970	1971
0	455627	473726	482306	488283	489919	480889	474641	465237	462142	439335	446688
1	423136	443078	460389	468907	476303	477060	469418	467135	454640	450196	426237
2	415734	422674	442324	459231	467370	473965	475799	470194	466887	452443	445878
3	407249	415923	422224	441682	457804	465365	473046	476777	469903	465152	448233
4	391751	407218	415646	421757	440456	455885	464660	474106	476700	468349	460530
5	378940	391909	407047	414718	420297	438709	454906	464357	472001	472147	456249
6	369959	379056	391490	406028	413339	419025	437676	454933	463478	469768	465500
7	375137	370252	378644	390629	404883	411810	418104	437852	453772	461475	463429
8	370975	375546	369945	377786	389521	403318	410899	418126	436911	452105	452081
9	368537	371430	375256	369125	376799	387898	402460	411341	417259	435540	446766
10	376034	368129	371187	375227	369747	375628	387770	402488	409254	415805	436835
11	390199	375746	368938	371177	375212	368148	374185	387483	401245	409219	420710
12	406065	390072	376581	368955	371150	373918	367125	374048	386565	401545	410482
13	441008	406539	390978	376496	369123	370130	372897	367388	373578	386984	399954
14	470075	442266	407219	390740	376663	368294	369408	373491	367043	374496	389896
15	384260	471839	442392	406797	391051	377424	368875	373566	375023	371585	381802
16	379982	382442	473056	436918	401631	389583	377873	367952	374301	379483	371400
17	379439	377755	379206	474327	430827	406132	391076	379323	367645	376828	380981
18	369743	379965	376585	378150	475371	437582	408591	393643	383392	370993	378637
19	339324	369696	380293	375532	377712	477046	440063	412597	397928	385907	375809
20	323322	339836	368491	379918	375539	381341	476851	440220	412336	397372	386188
21	335319	323638	340145	368483	381137	378703	380729	480231	440000	413043	397181
22	340286	337489	322978	340130	369026	382209	377626	381495	482213	439569	414367
23	339240	340992	338426	321940	340738	370553	381738	378388	381091	483110	443664
24	331743	339777	341611	338439	321871	341575	369412	382347	377989	381433	481655
25	328018	333241	338050	339300	336834	321910	339292	367347	380930	375237	381585
26	321992	328717	332642	336710	339065	336068	320143	336790	363732	379659	375040
27	315201	322580	328278	331669	336251	338862	334672	316354	333162	362499	376659
28	320987	316331	322168	327305	331628	336515	337127	332271	312859	329601	361260
29	329646	321103	315947	320907	327180	331540	334893	334186	329862	309452	330884
30	339445	331024	319473	313283	319303	326264	329226	330570	331003	327444	309905
31	337048	340647	329663	316863	312795	320098	325253	326985	328741	329420	328687
32	337038	339314	339904	327683	316160	313216	318973	323381	325299	327201	329452
33	336196	339015	339395	338125	327274	317326	312165	316995	322219	324294	326932
34	344221	337935	338770	338309	338288	327264	316301	310772	315837	321052	324112
35	354138	345894	337690	338094	338858	337321	325930	314295	308791	313461	320164
36	356423	354440	346081	336742	338436	337168	335971	325009	313282	308151	315744
37	363697	357031	353840	345564	336411	336655	335824	335385	324474	312461	307714
38	369669	363741	356337	352715	345593	335053	335623	336253	335070	323557	311689
39	392608	370049	362426	355215	352005	343218	334252	335343	336272	334414	322237
40	418447	392217	368467	360376	353948	351196	343000	333945	335512	336080	331222
41	406942	417092	391015	367534	359969	354594	351041	343293	333040	334627	335620
42	318057	415361	414795	390105	367179	360873	354342	350677	342132	332141	333003
43	298601	311069	422167	412883	389887	367300	360269	353608	349240	341407	332529
44	330754	295469	305379	426059	411710	389519	366563	359207	351908	347920	339175
45	348932	328397	293349	302361	429990	411374	388508	364573	356337	349775	347362
46	381049	347081	326260	292253	300472	419523	409701	386959	363121	355312	348889
47	387372	380449	345661	325231	290659	301043	420580	406866	385323	362208	356688
48	381093	385281	379630	344225	323436	290086	299566	422340	404430	384504	360365
49	376059	379539	382322	379058	342637	322968	288703	304911	424583	403307	381623
50	378803	373043	378292	379900	376456	341124	321200	286566	292488	427542	401791
51	376886	377169	370601	376405	377386	374901	339134	319202	285319	289469	421280
52	378729	376011	376332	368525	374340	377843	372911	337466	317763	283522	292056

Exercise 5.1. Use these two data sets to create a data frame consisting of the mortality rates for the UK population over each year. That is, the probability of dying in a given year based on your age?

Solution

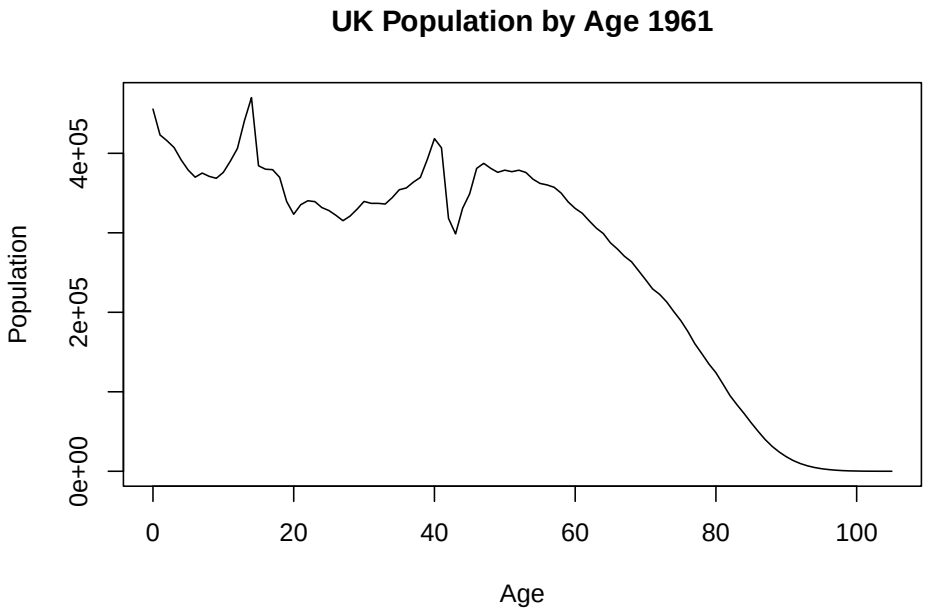
```
Mortality <- data.frame(Male_UK_Death_New$Age, Overall_Deaths[,-1]/Overall_Pop[,-1])
colnames(Mortality) <- colnames(Male_UK_Pop_New)
knitr::kable(Mortality)
```

Age	1961	1962	1963	1964	1965	1966	1967	1968	
0	0.0458379	0.0460625	0.0447185	0.0427150	0.0399474	0.0399988	0.0380814	0.0381483	0.036
1	0.0031550	0.0029904	0.0030778	0.0026658	0.0026328	0.0027858	0.0025947	0.0025496	0.002
2	0.0019989	0.0017318	0.0017498	0.0016136	0.0016197	0.0017027	0.0016162	0.0016397	0.001
3	0.0014045	0.0013656	0.0013713	0.0012068	0.0013193	0.0012958	0.0011669	0.0013675	0.001
4	0.0011436	0.0010977	0.0012631	0.0010883	0.0011602	0.0011779	0.0010072	0.0010926	0.001
5	0.0009922	0.0010564	0.0009950	0.0010079	0.0010231	0.0009551	0.0008793	0.0009152	0.000
6	0.0008704	0.0008811	0.0008991	0.0007930	0.0008903	0.0008042	0.0007814	0.0008133	0.000
7	0.0008370	0.0008103	0.0008425	0.0007936	0.0007978	0.0006751	0.0007295	0.0007331	0.000
8	0.0007386	0.0006497	0.0007893	0.0007200	0.0007034	0.0007488	0.0007520	0.0006816	0.000
9	0.0007408	0.0006112	0.0006795	0.0006800	0.0006821	0.0006780	0.0006783	0.0007245	0.000
10	0.0005957	0.0006030	0.0006816	0.0005970	0.0006572	0.0006176	0.0007427	0.0005913	0.000
11	0.0006356	0.0006467	0.0005394	0.0006170	0.0007143	0.0007035	0.0006334	0.0005962	0.000
12	0.0006723	0.0006947	0.0006639	0.0006884	0.0006978	0.0006846	0.0006701	0.0006951	0.000
13	0.0006848	0.0007552	0.0006982	0.0007384	0.0007721	0.0006971	0.0007509	0.0007050	0.000
14	0.0007339	0.0007869	0.0007195	0.0007780	0.0007991	0.0007901	0.0008202	0.0008487	0.000
15	0.0008666	0.0008647	0.0008160	0.0008678	0.0008874	0.0010121	0.0009407	0.0009262	0.000
16	0.0011737	0.0010956	0.0011246	0.0013549	0.0013296	0.0014349	0.0012888	0.0012964	0.001
17	0.0014864	0.0014189	0.0013476	0.0014399	0.0016387	0.0016399	0.0015061	0.0014104	0.001
18	0.0014875	0.0016107	0.0015455	0.0016448	0.0015735	0.0016386	0.0016349	0.0014226	0.001
19	0.0015649	0.0015553	0.0015856	0.0017628	0.0016971	0.0017441	0.0015680	0.0013888	0.001
20	0.0016300	0.0015978	0.0015794	0.0016346	0.0015471	0.0016154	0.0014428	0.0013879	0.001
21	0.0015090	0.0015357	0.0016434	0.0016202	0.0015165	0.0014655	0.0015313	0.0013806	0.001
22	0.0017133	0.0016060	0.0015295	0.0014494	0.0014769	0.0015724	0.0013876	0.0013788	0.001
23	0.0015859	0.0015953	0.0015099	0.0016059	0.0014879	0.0015167	0.0013831	0.0013531	0.001
24	0.0015916	0.0015657	0.0014695	0.0014626	0.0016156	0.0014872	0.0013968	0.0012999	0.001
25	0.0015639	0.0015694	0.0015205	0.0016681	0.0015794	0.0014507	0.0015149	0.0013557	0.001
26	0.0017143	0.0015363	0.0016564	0.0015265	0.0015572	0.0015890	0.0015618	0.0013866	0.001
27	0.0016085	0.0015097	0.0016693	0.0015347	0.0015822	0.0015345	0.0015000	0.0015046	0.001
28	0.0016823	0.0016786	0.0016513	0.0017048	0.0017007	0.0016463	0.0015454	0.0014897	0.001
29	0.0017716	0.0018935	0.0018769	0.0018167	0.0017636	0.0017886	0.0014452	0.0016159	0.001
30	0.0020327	0.0019576	0.0018656	0.0018960	0.0019292	0.0017471	0.0018012	0.0016093	0.001
31	0.0020027	0.0020050	0.0019171	0.0019125	0.0017903	0.0019869	0.0018539	0.0017738	0.001
32	0.0021600	0.0020984	0.0022859	0.0020782	0.0020148	0.0019380	0.0018654	0.0019513	0.002
33	0.0022963	0.0021238	0.0022864	0.0022270	0.0021267	0.0022059	0.0020118	0.0019748	0.001
34	0.0024374	0.0025064	0.0024825	0.0023351	0.0023855	0.0024751	0.0021815	0.0022750	0.002
35	0.0027842	0.0027754	0.0027096	0.0025171	0.0026648	0.0026859	0.0024729	0.0023449	0.002
36	0.0028505	0.0029201	0.0029357	0.0028271	0.0029045	0.0027820	0.0027532	0.0025753	0.002
37	0.0032472	0.0031370	0.0032359	0.0031601	0.0032341	0.0029734	0.0027902	0.0029310	0.002
38	0.0035870	0.0037389	0.0036567	0.0036091	0.0035996	0.0035368	0.0032924	0.0033576	0.003
39	0.0039429	0.0040697	0.0039760	0.0038230	0.0038409	0.0038285	0.0037906	0.0036321	0.003
40	0.0042538	0.0042757	0.0043722	0.0043260	0.0043622	0.0042996	0.0041108	0.0041294	0.004
41	0.0048852	0.0046872	0.0045855	0.0046037	0.0049699	0.0045235	0.0044582	0.0044073	0.004
42	0.0052223	0.0054194	0.0056052	0.0053832	0.0055123	0.0054285	0.0050770	0.0052755	0.005
43	0.0058305	0.0055550	0.0057939	0.0057256	0.0057017	0.0059134	0.0056458	0.0058568	0.005
44	0.0062524	0.0063120	0.0064084	0.0062386	0.0062763	0.0061229	0.0061245	0.0062554	0.006
45	0.0071848	0.0070463	0.0072746	0.0070148	0.0067839	0.0070884	0.0068132	0.0068738	0.007
46	0.0075476	0.0076783	0.0076565	0.0081128	0.0081738	0.0076706	0.0077910	0.0075977	0.008
47	0.0084802	0.0086976	0.0092721	0.0088737	0.0086837	0.0090984	0.0083955	0.0085851	0.008
48	0.0094937	0.0093698	0.0094935	0.0090667	0.0094516	0.0095799	0.0092968	0.0091987	0.009
49	0.0105941	0.0102809	0.0108783	0.0106079	0.0107431	0.0105676	0.0103290	0.0104621	0.010
50	0.0114915	0.0114813	0.0119035	0.0114320	0.0113586	0.0121510	0.0113730	0.0114075	0.012
51	0.0128209	0.0127343	0.0127442	0.0121598	0.0125654	0.0123339	0.0121368	0.0126816	0.012
52	0.0145935	0.0144703	0.0146121	0.0142351	0.0141529	0.0142175	0.0132927	0.0143451	0.014

Note - We could have done this for the individual sex data sets to obtain a more accurate estimate of mortality based on sex.

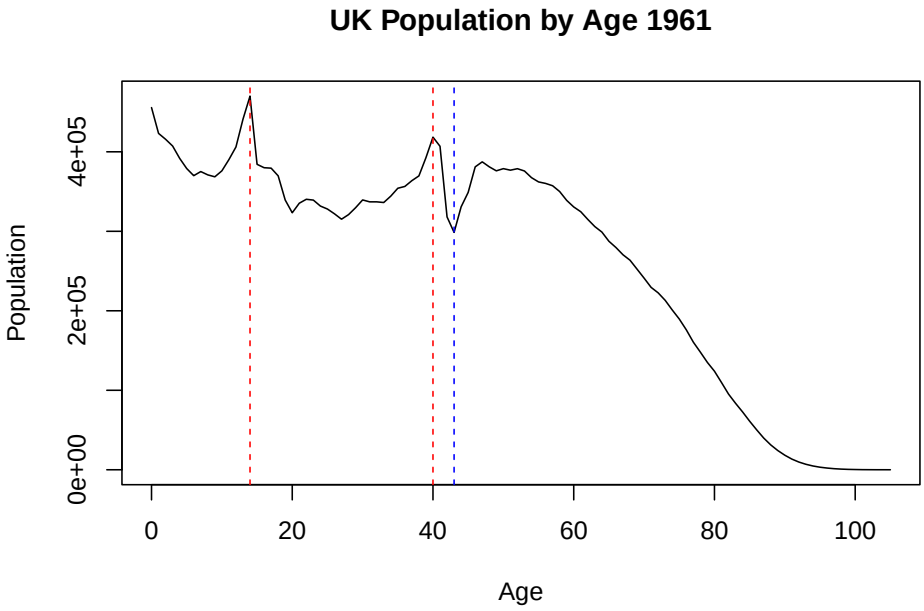
Before we carry on, let us actually have a look at this data visually now that we have created it:

```
plot(0:105, Overall_Pop$`1961`, type = "l", main = "UK Population by Age 1961", xlab =
```



Exercise 5.2. As actuaries, you need to be able to critically analyse data, not just use it. What do you think is the reason for these spikes and falls in the data?

```
plot(0:105, Overall_Pop$`1961`, type = "l", main = "UK Population by Age 1961", xlab =
abline(v = 14, col = "red", lty = 2)
abline(v = 40, col = "red", lty = 2)
abline(v = 43, col = "blue", lty = 2)
```



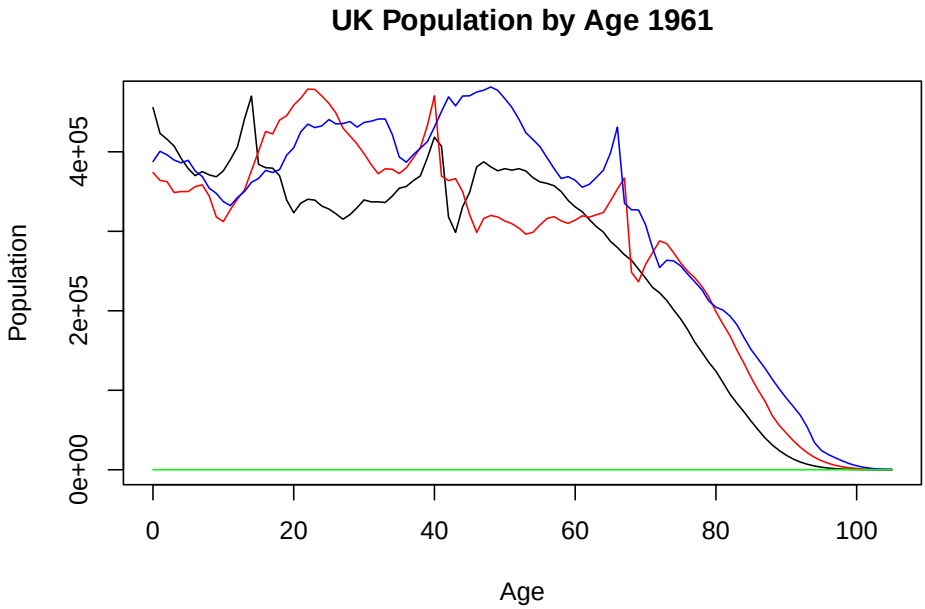
Solution

The simple answer is most likely World War 2. The initial spike in the population at around the age of 14/15 due to the baby boom immediately after the end of the war in 1947, this would result in a surge of 14/15 year olds in 1961. The same thing ocurred after World War 1, which is the likely cause of the second spike at around age 41/42. The sudden drop or fall just after this at around the 43/44 year mark is most likely due to the fact that individuals aged roughly 43/44 in 1961 would have been aged roughly 23/24 at the start of World War 2 in 1942, the prime age for fighting soldiers.

Exercise 5.3. Why would these spikes be of such importance to us as actuaries?

Solution

```
plot(0:105, Overall_Pop$`1961`, type = "l", main = "UK Population by Age 1961", xlab = "Age", yla
lines(0:105, Overall_Pop$`1987`, type = "l", col = "red")
lines(0:105, Overall_Pop$`2013`, type = "l", col= "blue")
lines(0:105, Mortality$`2017`, type = "l", col = "green")
```



These spikes in the different ages will drift with time and eventually cause spikes in the population at retirement age. This is extremely important for pension actuaries when modelling and predicting pension plans and beneift payments.

Exercise 5.4. Can you plot the mortality rates for every single year on one plot? **Hint: Note that the line colours can also be represented by numbers, e.g., col = 1.**

Solution

```
plot(0:105, Mortality[,2], type = "l", xlab = "Age", ylab = "Mortality Rate", main = "U

for(i in 3:(ncol(Mortality)-2)){
  lines(0:105, Mortality[,i], type = "l", col = i)
}
```

5.4 Conditional extraction

There may be circumstances where you want to consider some analysis of a data set, but you may only be interested in particular observations that satisfy certain conditions. For example, assume we wanted to price a pension annuity (regular payments until death), then we would only be interested in the Mortality rates of pensioners (65+). We can extract this information using the conditional arguments we have seen in previous sessions. However, before we can do this, we have to amend the Age values in the Mortality data set to be numerical/integer values rather than characters:

```
class(Mortality$Age)
```

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

```
Mortality$Age <- 0:105
class(Mortality$Age)
```

```
## [1] "integer"
```

```
Mort_pension <- Mortality[Mortality$Age >= 65, ]
knitr::kable(Mort_pension)
```

	Age	1961	1962	1963	1964	1965	1966	1967	
66	65	0.0453250	0.0451678	0.0465511	0.0444692	0.0466920	0.0473058	0.0447622	0
67	66	0.0482433	0.0472162	0.0482025	0.0459599	0.0465396	0.0501376	0.0482375	0
68	67	0.0543559	0.0542023	0.0546142	0.0513501	0.0523997	0.0519464	0.0527789	0
69	68	0.0588200	0.0583787	0.0596780	0.0547545	0.0558814	0.0564531	0.0537941	0
70	69	0.0633813	0.0635610	0.0642519	0.0594366	0.0601774	0.0608078	0.0584258	0
71	70	0.0691646	0.0678830	0.0699610	0.0648189	0.0644687	0.0653506	0.0620893	0
72	71	0.0734503	0.0723687	0.0728671	0.0674723	0.0684603	0.0687176	0.0653036	0
73	72	0.0825980	0.0825967	0.0825036	0.0763458	0.0793749	0.0781386	0.0737164	0
74	73	0.0894636	0.0882420	0.0909511	0.0827134	0.0837402	0.0845020	0.0801360	0
75	74	0.0970311	0.0969414	0.0970082	0.0915101	0.0906133	0.0920960	0.0871653	0
76	75	0.1056129	0.1032938	0.1043559	0.0946088	0.0969340	0.0982225	0.0924882	0
77	76	0.1171314	0.1101415	0.1145954	0.1030168	0.1037891	0.1085443	0.1019910	0
78	77	0.1254875	0.1231202	0.1158537	0.1104062	0.1115323	0.1117554	0.1051725	0
79	78	0.1357077	0.1353843	0.1378214	0.1115317	0.1238244	0.1242391	0.1125151	0
80	79	0.1500385	0.1479939	0.1488987	0.1324086	0.1195239	0.1345043	0.1237140	0
81	80	0.1599393	0.1575593	0.1618862	0.1447187	0.1447990	0.1342436	0.1355124	0
82	81	0.1749119	0.1701382	0.1731900	0.1544345	0.1528577	0.1572606	0.1311799	0
83	82	0.1922043	0.1915158	0.1986889	0.1714408	0.1705369	0.1741823	0.1602726	0
84	83	0.2081499	0.2042565	0.2131486	0.1881321	0.1875583	0.1922412	0.1756620	0
85	84	0.2249288	0.2307439	0.2269647	0.2073820	0.2073161	0.2091217	0.1948028	0
86	85	0.2402017	0.2398683	0.2439453	0.2147270	0.2167415	0.2226554	0.2043676	0
87	86	0.2657572	0.2638814	0.2645337	0.2342807	0.2366666	0.2425583	0.2256615	0
88	87	0.2747994	0.2828091	0.2850552	0.2463805	0.2557907	0.2597608	0.2425246	0
89	88	0.2971728	0.2868150	0.3033020	0.2599041	0.2663857	0.2752678	0.2499675	0
90	89	0.3105857	0.3100156	0.3225794	0.2868291	0.2860159	0.3014150	0.2794725	0
91	90	0.3308378	0.3296326	0.3339968	0.2899460	0.3028748	0.3183755	0.2943025	0
92	91	0.3522008	0.3446657	0.3506511	0.3041082	0.3218931	0.3275193	0.3059018	0
93	92	0.3755806	0.3746327	0.3873755	0.3339905	0.3489631	0.3550581	0.3335813	0
94	93	0.4033785	0.4009334	0.4081246	0.3519504	0.3656893	0.3842217	0.3550345	0
95	94	0.4251106	0.4240805	0.4342789	0.3807297	0.3839836	0.3897949	0.3821761	0
96	95	0.4491763	0.4244910	0.4396092	0.3847242	0.4071392	0.4184549	0.3914789	0
97	96	0.4829995	0.4791571	0.4802691	0.4405475	0.4320446	0.4609404	0.4168357	0
98	97	0.4907201	0.4552023	0.4924839	0.4386792	0.4403727	0.4380215	0.4410811	0
99	98	0.5261876	0.5046948	0.5333333	0.4777070	0.4643216	0.5161290	0.4575972	0
100	99	0.5081967	0.4708171	0.4833948	0.4450172	0.4542429	0.4859375	0.4978355	0
101	100	0.5374150	0.5860927	0.5555556	0.4926686	0.4657895	0.5573333	0.4626506	0
102	101	0.5878378	0.6149068	0.6000000	0.5265957	0.5668203	0.5176991	0.4481328	0
103	102	0.6627907	0.6081081	0.4835165	0.5098039	0.4035088	0.5923077	0.4859155	0
104	103	0.7555556	0.6818182	0.5853659	0.7307692	0.4838710	0.5303030	0.4800000	0
105	104	0.8571429	0.9047619	0.8260870	0.4782609	0.4642857	0.6111111	0.5609756	0
106	105	0.7500000	0.3076923	0.8571429	0.3333333	0.6800000	0.8064516	0.4687500	0

How does this work? This is a combination of conditional statements and extraction. In short, the conditional statement creates a vector of `TRUE/FALSE` values

which are then used in the matrix type extraction technique we discussed in previous weeks. The result is that R will only extract the row(column) numbers corresponding to `TRUE` values. This is known as 'conditional extraction'.

Moreover, due to changes in technology and the NHS, we may decide that mortality rates pre-2000 are not valid enough to be used in our calculations, so we might only want to consider post-2000 values. Can we do this in a similar way? In general you will not be able to extract certain columns using conditional arguments since they represent different variables. However, you can do this by inspection, using matrix extraction and the `which()` function:

```
which(colnames(Mortality) == "2000")
```

```
## [1] 41
```

```
Mort_pension_2000 <- Mort_pension[,c(1,41:ncol(Mort_pension))]
knitr::kable(Mort_pension_2000)
```

	Age	2000	2001	2002	2003	2004	2005	2006	
66	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0
67	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0
68	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0
69	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0
70	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0
71	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0
72	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0
73	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0
74	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0
75	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0
76	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0
77	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0
78	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0
79	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0
80	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0
81	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0
82	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0
83	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0
84	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0
85	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0
86	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0
87	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0
88	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0
89	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0
90	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0
91	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0
92	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0
93	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0
94	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0
95	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0
96	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0
97	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0
98	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0
99	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0
100	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0
101	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0
102	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0
103	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0
104	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0
105	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0
106	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0

In general, it is possible to use the conditional argument format to extract rows from data sets, as long as you are conditioning on elements/variables within the

data rather than column names. To see this, let us briefly revisit the mtcars data set from the previous sessions

```
knitr::kable(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

Assume that you only want to deal with cars that have 6 or 8 cylinders:

```
data <- mtcars[mtcars$cyl >= 6, ]
knitr::kable(data)
```

	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
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 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
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
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

```
data1 <- mtcars[mtcars$cyl >= 6 & mtcars$hp > 100, ]
knitr::kable(data1)
```

	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
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 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
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
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

5.5 Adding data

As well as extracting data from data frame, it is also possible to add new data to an existing data frame. That is, add a new observation (row) or even add a new variable (column) which is possibly even calculated using the rest of the data. There are a number of ways to add a new row to a dataframe but I find the easiest is to use something similar to row extraction but for an undefined row (non-existent row). Let us first create a data frame:

```
new_data <- data.frame(A = c(1,2,3), B = c("Pass", "Fail", "Pass"), C = c("M", "M", "F"))
knitr::kable(new_data)
```

A	B	C
1	Pass	M
2	Fail	M
3	Pass	F

Now, we can add a new row/observation to this existing data frame:

```
new_data[nrow(new_data)+1,] <- c(4, "Fail", "M")
knitr::kable(new_data)
```

A	B	C
1	Pass	M
2	Fail	M
3	Pass	F
4	Fail	M

However, this only really works when adding a single row. If you wanted to add another data frame, you can use the rowbind function `rbind()`:

```
add_data <- data.frame(A = c(5,6,7), B = c("Pass", "Pass", "Pass"), C = c("F", "M", "F
knitr::kable(add_data)
```

A	B	C
5	Pass	F
6	Pass	M
7	Pass	F

```
new_data <- rbind(new_data, add_data)
knitr::kable(new_data)
```

A	B	C
1	Pass	M
2	Fail	M
3	Pass	F
4	Fail	M
5	Pass	F
6	Pass	M
7	Pass	F

With this in mind, let us revisit the Mortality data set for pensioners post 2000:

```
knitr::kable(Mort_pension_2000)
```

	Age	2000	2001	2002	2003	2004	2005	2006	2007	
66	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0.0231199	0
67	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0.0255041	0
68	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0.0272228	0
69	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0.0299645	0
70	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0.0333274	0
71	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0.0357971	0
72	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0.0387563	0
73	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0.0429724	0
74	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0.0471572	0
75	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0.0522401	0
76	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0.0573588	0
77	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0.0634192	0
78	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0.0691717	0
79	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0.0753961	0
80	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0.0843689	0
81	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0.0929846	0
82	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0.1019045	0
83	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0.1115087	0
84	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0.1208632	0
85	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0.1317668	0
86	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0.1436239	0
87	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0.1554006	0
88	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0.1703657	0
89	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0.1641898	0
90	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0.2043887	0
91	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0.2207156	0
92	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0.2322676	0
93	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0.2610809	0
94	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0.2801292	0
95	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0.2981533	0
96	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0.3286202	0
97	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0.3559714	0
98	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0.3695969	0
99	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0.4026568	0
100	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0.4243807	0
101	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0.4600622	0
102	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0.4796304	0
103	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0.5235378	0
104	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0.5634518	0
105	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0.5730594	0
106	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0.5588865	0

If we want to add a new column/variable, you can do this in a very similar way as for rows. As an example, let us assume we want to add the mean of the

mortality rates across years for the different ages.

Exercise 5.5. Can you create a vector of the mean mortality rates for each age using loops?

Solution

```
mean_mort <- c()
for(i in 1:nrow(Mort_pension_2000)){
  mean <- mean(as.numeric(Mort_pension_2000[i,-1]))
  mean_mort <- c(mean_mort, mean)
}
mean_mort
```

```
##  [1] 0.02238190 0.02459383 0.02689333 0.02942466 0.03229551 0.03530337
##  [7] 0.03894055 0.04307048 0.04711833 0.05205196 0.05707330 0.06277159
## [13] 0.06853001 0.07508073 0.08228392 0.09053181 0.09885726 0.10854133
## [19] 0.11906908 0.13048077 0.14266349 0.15546612 0.16949190 0.18490826
## [25] 0.20157107 0.21858396 0.23699153 0.25800491 0.27940554 0.30027083
## [31] 0.32596897 0.35100470 0.37298012 0.39781445 0.42343919 0.45313849
## [37] 0.48834911 0.51654249 0.55021119 0.57178464 0.61909782
```

Now that we have calculated the value, let us add it to the data. This can be done using any of the following methods:

Method 1

```
Mort_pension_2000[,ncol(Mort_pension_2000)+1] <- mean_mort
colnames(Mort_pension_2000)[ncol(Mort_pension_2000)] <- c("Mean")
knitr::kable(Mort_pension_2000)
```

	Age	2000	2001	2002	2003	2004	2005	2006	2007	
66	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0.0231199	0
67	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0.0255041	0
68	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0.0272228	0
69	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0.0299645	0
70	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0.0333274	0
71	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0.0357971	0
72	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0.0387563	0
73	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0.0429724	0
74	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0.0471572	0
75	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0.0522401	0
76	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0.0573588	0
77	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0.0634192	0
78	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0.0691717	0
79	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0.0753961	0
80	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0.0843689	0
81	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0.0929846	0
82	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0.1019045	0
83	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0.1115087	0
84	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0.1208632	0
85	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0.1317668	0
86	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0.1436239	0
87	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0.1554006	0
88	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0.1703657	0
89	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0.1641898	0
90	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0.2043887	0
91	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0.2207156	0
92	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0.2322676	0
93	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0.2610809	0
94	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0.2801292	0
95	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0.2981533	0
96	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0.3286202	0
97	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0.3559714	0
98	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0.3695969	0
99	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0.4026568	0
100	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0.4243807	0
101	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0.4600622	0
102	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0.4796304	0
103	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0.5235378	0
104	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0.5634518	0
105	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0.5730594	0
106	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0.5588865	0

Method 2

```
Mort_pension_2000 <- Mort_pension_2000[,-ncol(Mort_pension_2000)] # This just removes

Mort1 <- cbind(Mort_pension_2000, mean_mort)
colnames(Mort1)[ncol(Mort1)] <- c("Mean")
knitr::kable(Mort1)
```

	Age	2000	2001	2002	2003	2004	2005	2006	2007	
66	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0.0231199	0
67	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0.0255041	0
68	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0.0272228	0
69	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0.0299645	0
70	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0.0333274	0
71	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0.0357971	0
72	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0.0387563	0
73	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0.0429724	0
74	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0.0471572	0
75	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0.0522401	0
76	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0.0573588	0
77	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0.0634192	0
78	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0.0691717	0
79	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0.0753961	0
80	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0.0843689	0
81	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0.0929846	0
82	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0.1019045	0
83	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0.1115087	0
84	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0.1208632	0
85	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0.1317668	0
86	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0.1436239	0
87	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0.1554006	0
88	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0.1703657	0
89	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0.1641898	0
90	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0.2043887	0
91	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0.2207156	0
92	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0.2322676	0
93	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0.2610809	0
94	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0.2801292	0
95	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0.2981533	0
96	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0.3286202	0
97	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0.3559714	0
98	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0.3695969	0
99	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0.4026568	0
100	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0.4243807	0
101	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0.4600622	0
102	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0.4796304	0
103	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0.5235378	0
104	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0.5634518	0
105	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0.5730594	0
106	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0.5588865	0

Method 3

```
Mort2 <- data.frame(Mort_pension_2000, mean_mort)
colnames(Mort2) <- c(colnames(Mort_pension_2000), "Mean")
knitr::kable(Mort2)
```

	Age	2000	2001	2002	2003	2004	2005	2006	2007	
66	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0.0231199	0
67	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0.0255041	0
68	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0.0272228	0
69	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0.0299645	0
70	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0.0333274	0
71	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0.0357971	0
72	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0.0387563	0
73	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0.0429724	0
74	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0.0471572	0
75	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0.0522401	0
76	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0.0573588	0
77	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0.0634192	0
78	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0.0691717	0
79	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0.0753961	0
80	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0.0843689	0
81	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0.0929846	0
82	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0.1019045	0
83	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0.1115087	0
84	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0.1208632	0
85	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0.1317668	0
86	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0.1436239	0
87	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0.1554006	0
88	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0.1703657	0
89	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0.1641898	0
90	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0.2043887	0
91	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0.2207156	0
92	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0.2322676	0
93	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0.2610809	0
94	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0.2801292	0
95	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0.2981533	0
96	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0.3286202	0
97	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0.3559714	0
98	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0.3695969	0
99	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0.4026568	0
100	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0.4243807	0
101	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0.4600622	0
102	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0.4796304	0
103	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0.5235378	0
104	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0.5634518	0
105	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0.5730594	0
106	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0.5588865	0

5.6 The apply family

The final tool I want to talk about in this R module is the family of functions known as the `apply()` functions. Put simply, the `apply()` function, along with the its counterparts `lapply()`, `sapply()` and `vapply()` allow us to 'apply' a particular function on each row and/or column of a data frame (and other objects) without using loops.

The `apply()` function is basically a quicker and more convenient version of a `for()` loop and should always be considered first before loops as they are easier to write, read and are a lot quicker to execute which makes a huge difference when working with larger data sets.

The `apply()` function takes 3 main inputs (it can take more and we will discuss this soon). The first is the object you want to 'apply' the function to, in our case today a data frame. The second is either of the following: `1`, `2`, or `c(1,2)` where `1` indicates you want the function to be applied to the rows of the object,`2` for the columns of the object and `c(1,2)` to individual elements. Finally, the third input is the name of the function we want to apply.

As an example, let us look at finding the means for each row of the `Mort_pension_2000` data frame as we did before using `for()` loops.

```
apply(Mort_pension_2000[,-1], 1, mean)
```

```
##         66         67         68         69         70         71         72
## 0.02238190 0.02459383 0.02689333 0.02942466 0.03229551 0.03530337 0.03894055
##         73         74         75         76         77         78         79
## 0.04307048 0.04711833 0.05205196 0.05707330 0.06277159 0.06853001 0.07508073
##         80         81         82         83         84         85         86
## 0.08228392 0.09053181 0.09885726 0.10854133 0.11906908 0.13048077 0.14266349
##         87         88         89         90         91         92         93
## 0.15546612 0.16949190 0.18490826 0.20157107 0.21858396 0.23699153 0.25800491
##         94         95         96         97         98         99        100
## 0.27940554 0.30027083 0.32596897 0.35100470 0.37298012 0.39781445 0.42343919
##        101        102        103        104        105        106
## 0.45313849 0.48834911 0.51654249 0.55021119 0.57178464 0.61909782
```

Does the output seem a little strange? You should notice that the row names are not the actual ages but the row number the ages were on! Again, be careful. Although, this can be easily rectified:

```
rownames(Mort_pension_2000) <- 65:105
apply(Mort_pension_2000[,-1], 1, mean)
```

```
##         65         66         67         68         69         70         71
## 0.02238190 0.02459383 0.02689333 0.02942466 0.03229551 0.03530337 0.03894055
##         72         73         74         75         76         77         78
## 0.04307048 0.04711833 0.05205196 0.05707330 0.06277159 0.06853001 0.07508073
```

```
##          79          80          81          82          83          84          85
## 0.08228392 0.09053181 0.09885726 0.10854133 0.11906908 0.13048077 0.14266349
##          86          87          88          89          90          91          92
## 0.15546612 0.16949190 0.18490826 0.20157107 0.21858396 0.23699153 0.25800491
##          93          94          95          96          97          98          99
## 0.27940554 0.30027083 0.32596897 0.35100470 0.37298012 0.39781445 0.42343919
##         100         101         102         103         104         105
## 0.45313849 0.48834911 0.51654249 0.55021119 0.57178464 0.61909782
```

Using this simple execution, we could add the mean column to our original data set using the following lines of code:

```
Mort3 <- cbind(Mort_pension_2000, apply(Mort_pension_2000[,-1], 1, mean))
colnames(Mort3)[ncol(Mort3)] <- "Mean"
knitr::kable(Mort3)
```

	Age	2000	2001	2002	2003	2004	2005	2006	
65	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0
66	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0
67	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0
68	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0
69	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0
70	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0
71	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0
72	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0
73	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0
74	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0
75	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0
76	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0
77	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0
78	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0
79	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0
80	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0
81	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0
82	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0
83	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0
84	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0
85	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0
86	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0
87	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0
88	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0
89	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0
90	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0
91	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0
92	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0
93	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0
94	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0
95	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0
96	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0
97	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0
98	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0
99	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0
100	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0
101	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0
102	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0
103	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0
104	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0
105	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0

Exercise 5.6. With this in mind, can you now add the standard deviation for each age to the end of Mort3, using the `apply()` function?

Solution

```
Mort3 <- cbind(Mort3, apply(Mort_pension_2000[,-1], 1, sd))
colnames(Mort3)[ncol(Mort3)] <- "Std. Dev."
knitr::kable(Mort3)
```

	Age	2000	2001	2002	2003	2004	2005	2006	
65	65	0.0278453	0.0266606	0.0261528	0.0259069	0.0246108	0.0241466	0.0236559	0
66	66	0.0308525	0.0290598	0.0288230	0.0279165	0.0274248	0.0265933	0.0255072	0
67	67	0.0335818	0.0323751	0.0317880	0.0311815	0.0294690	0.0292283	0.0280962	0
68	68	0.0369593	0.0347670	0.0347835	0.0340542	0.0327099	0.0312807	0.0310929	0
69	69	0.0405886	0.0388618	0.0381295	0.0378106	0.0357243	0.0340657	0.0324298	0
70	70	0.0439035	0.0428366	0.0409666	0.0397868	0.0391074	0.0369862	0.0362533	0
71	71	0.0487182	0.0467752	0.0465493	0.0449379	0.0422429	0.0420281	0.0402224	0
72	72	0.0538503	0.0517941	0.0509016	0.0492548	0.0473727	0.0459098	0.0440236	0
73	73	0.0589917	0.0576292	0.0554753	0.0544581	0.0515873	0.0494410	0.0479822	0
74	74	0.0657478	0.0622442	0.0619867	0.0603801	0.0565445	0.0557421	0.0535528	0
75	75	0.0700235	0.0685910	0.0682166	0.0666065	0.0630728	0.0599411	0.0583842	0
76	76	0.0753368	0.0739970	0.0743648	0.0726591	0.0688199	0.0675093	0.0637262	0
77	77	0.0822262	0.0802845	0.0797120	0.0796057	0.0753610	0.0743143	0.0708878	0
78	78	0.0884686	0.0859751	0.0865164	0.0851583	0.0833300	0.0811302	0.0780198	0
79	79	0.0939941	0.0934709	0.0935694	0.0949527	0.0884832	0.0887238	0.0863233	0
80	80	0.1014521	0.1011788	0.1021735	0.1016985	0.0966068	0.0958757	0.0937271	0
81	81	0.1063295	0.1088480	0.1101913	0.1102797	0.1055018	0.1053656	0.1018429	0
82	82	0.1222890	0.1132130	0.1191477	0.1203196	0.1145742	0.1126951	0.1110408	0
83	83	0.1324233	0.1318556	0.1213337	0.1305448	0.1233390	0.1237859	0.1203748	0
84	84	0.1437819	0.1424082	0.1407282	0.1322863	0.1355058	0.1340405	0.1315754	0
85	85	0.1539061	0.1554339	0.1543621	0.1597380	0.1358936	0.1448993	0.1425242	0
86	86	0.1653379	0.1628012	0.1676403	0.1703181	0.1628528	0.1465962	0.1561529	0
87	87	0.1785984	0.1769576	0.1786353	0.1864538	0.1759518	0.1762669	0.1533323	0
88	88	0.1947017	0.1923505	0.1958701	0.1964498	0.1924671	0.1907852	0.1852686	0
89	89	0.2068114	0.2100395	0.2129975	0.2161462	0.2013810	0.2123148	0.1998371	0
90	90	0.2243093	0.2212690	0.2269600	0.2328060	0.2191950	0.2216868	0.2157481	0
91	91	0.2428753	0.2400114	0.2450832	0.2472792	0.2393630	0.2416400	0.2393893	0
92	92	0.2640729	0.2587539	0.2676636	0.2711878	0.2606217	0.2588041	0.2555003	0
93	93	0.2827992	0.2850763	0.2903785	0.3007276	0.2752053	0.2874141	0.2785002	0
94	94	0.3029729	0.3004167	0.3075925	0.3142848	0.3072019	0.3026762	0.2968440	0
95	95	0.3358635	0.3271933	0.3288209	0.3444826	0.3229117	0.3298165	0.3196674	0
96	96	0.3569652	0.3484768	0.3558364	0.3638302	0.3485475	0.3496359	0.3434985	0
97	97	0.3773845	0.3765631	0.3718255	0.3979007	0.3734694	0.3743954	0.3691512	0
98	98	0.3934211	0.4040446	0.4016691	0.4150943	0.3906151	0.4081706	0.3906802	0
99	99	0.4137931	0.4158562	0.4317628	0.4304149	0.4187125	0.4292060	0.3977774	0
100	100	0.4414894	0.4612748	0.4568138	0.4742015	0.4351741	0.4492041	0.4425685	0
101	101	0.4839506	0.4816054	0.4794737	0.5020141	0.4841849	0.5053513	0.4691750	0
102	102	0.4859521	0.5068627	0.5466667	0.5572052	0.4880137	0.5332278	0.5067466	0
103	103	0.5476636	0.5266904	0.5415282	0.5927673	0.5462555	0.5581062	0.5188172	0
104	104	0.5795053	0.5646688	0.5987461	0.6223565	0.5662983	0.5661376	0.5024155	0
105	105	0.6736111	0.6032787	0.6231884	0.6123596	0.5840220	0.6641414	0.6545012	0

Just so you can see how this works let's look at the same function applied to columns:

```
apply(Mort_pension_2000[,-1], 2, mean)
```

```
##      2000      2001      2002      2003      2004      2005      2006      2007
## 0.2137402 0.2107915 0.2145128 0.2197029 0.2065787 0.2114458 0.2028491 0.2057311
##      2008      2009      2010      2011      2012      2013      2014      2015
## 0.2130662 0.1988136 0.1993748 0.1943025 0.2038228 0.2058559 0.1969537 0.2119526
##      2016      2017
## 0.2002276 0.2057339
```

Finally, let us try on the individual elements (obviously we will have to use something other than mean here):

```
apply(Mort_pension_2000, c(1,2), class)
```

```
##     Age       2000      2001      2002      2003      2004      2005
## 65  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 66  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 67  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 68  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 69  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 70  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 71  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 72  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 73  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 74  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 75  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 76  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 77  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 78  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 79  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 80  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 81  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 82  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 83  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 84  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 85  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 86  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 87  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 88  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 89  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 90  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 91  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 92  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 93  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 94  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 95  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
## 96  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 97  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 98  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 99  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 100 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 101 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 102 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 103 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 104 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 105 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##     2006      2007      2008      2009      2010      2011      2012
## 65  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 66  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 67  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 68  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 69  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 70  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 71  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 72  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 73  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 74  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 75  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 76  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 77  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 78  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 79  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 80  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 81  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 82  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 83  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 84  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 85  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 86  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 87  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 88  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 89  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 90  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 91  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 92  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 93  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 94  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 95  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 96  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 97  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 98  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 99  "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
## 100 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 101 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 102 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 103 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 104 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
## 105 "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##     2013      2014      2015      2016      2017
## 65  "numeric" "numeric" "numeric" "numeric" "numeric"
## 66  "numeric" "numeric" "numeric" "numeric" "numeric"
## 67  "numeric" "numeric" "numeric" "numeric" "numeric"
## 68  "numeric" "numeric" "numeric" "numeric" "numeric"
## 69  "numeric" "numeric" "numeric" "numeric" "numeric"
## 70  "numeric" "numeric" "numeric" "numeric" "numeric"
## 71  "numeric" "numeric" "numeric" "numeric" "numeric"
## 72  "numeric" "numeric" "numeric" "numeric" "numeric"
## 73  "numeric" "numeric" "numeric" "numeric" "numeric"
## 74  "numeric" "numeric" "numeric" "numeric" "numeric"
## 75  "numeric" "numeric" "numeric" "numeric" "numeric"
## 76  "numeric" "numeric" "numeric" "numeric" "numeric"
## 77  "numeric" "numeric" "numeric" "numeric" "numeric"
## 78  "numeric" "numeric" "numeric" "numeric" "numeric"
## 79  "numeric" "numeric" "numeric" "numeric" "numeric"
## 80  "numeric" "numeric" "numeric" "numeric" "numeric"
## 81  "numeric" "numeric" "numeric" "numeric" "numeric"
## 82  "numeric" "numeric" "numeric" "numeric" "numeric"
## 83  "numeric" "numeric" "numeric" "numeric" "numeric"
## 84  "numeric" "numeric" "numeric" "numeric" "numeric"
## 85  "numeric" "numeric" "numeric" "numeric" "numeric"
## 86  "numeric" "numeric" "numeric" "numeric" "numeric"
## 87  "numeric" "numeric" "numeric" "numeric" "numeric"
## 88  "numeric" "numeric" "numeric" "numeric" "numeric"
## 89  "numeric" "numeric" "numeric" "numeric" "numeric"
## 90  "numeric" "numeric" "numeric" "numeric" "numeric"
## 91  "numeric" "numeric" "numeric" "numeric" "numeric"
## 92  "numeric" "numeric" "numeric" "numeric" "numeric"
## 93  "numeric" "numeric" "numeric" "numeric" "numeric"
## 94  "numeric" "numeric" "numeric" "numeric" "numeric"
## 95  "numeric" "numeric" "numeric" "numeric" "numeric"
## 96  "numeric" "numeric" "numeric" "numeric" "numeric"
## 97  "numeric" "numeric" "numeric" "numeric" "numeric"
## 98  "numeric" "numeric" "numeric" "numeric" "numeric"
## 99  "numeric" "numeric" "numeric" "numeric" "numeric"
## 100 "numeric" "numeric" "numeric" "numeric" "numeric"
## 101 "numeric" "numeric" "numeric" "numeric" "numeric"
## 102 "numeric" "numeric" "numeric" "numeric" "numeric"
## 103 "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
## 104 "numeric" "numeric" "numeric" "numeric" "numeric"
## 105 "numeric" "numeric" "numeric" "numeric" "numeric"
```

You can use ANY function in the `apply()` command, including those which have multiple inputs - in this case, the additional inputs are just input as additional inputs into the `apply()` function itself - and custom created functions like those we created in the previous week. As a final example, let us create a function and use it within `apply()` on the Mortality data:

```
min_max <- function(x, type){
  ordered <- sort(x)
  if(type == "min"){
    return (ordered[1])
  } else if (type == "max"){
    return(ordered[length(ordered)])
  }
}

apply(Mort_pension_2000[,-1], 1, min_max, type = "min")
```

```
##         65         66         67         68         69         70         71
## 0.01883388 0.02047562 0.02189074 0.02462564 0.02695115 0.02853286 0.03254928
##         72         73         74         75         76         77         78
## 0.03682673 0.04034642 0.04410924 0.04970849 0.05459301 0.05976099 0.06544642
##         79         80         81         82         83         84         85
## 0.07276663 0.08133775 0.08917693 0.09896874 0.11004793 0.12088046 0.13334147
##         86         87         88         89         90         91         92
## 0.14586211 0.15333233 0.16418981 0.17963347 0.19209634 0.20916244 0.22064379
##         93         94         95         96         97         98         99
## 0.25244209 0.27842851 0.29249287 0.32927679 0.34647303 0.37148794 0.39777739
##        100        101        102        103        104        105
## 0.42034884 0.46917498 0.48428480 0.49441964 0.50241546 0.55555556
```

```
apply(Mort_pension_2000[,-1], 1, min_max, type = "max")
```

```
##         65         66         67         68         69         70         71
## 0.02784526 0.03085251 0.03358179 0.03695932 0.04058863 0.04390346 0.04871818
##         72         73         74         75         76         77         78
## 0.05385029 0.05899168 0.06574779 0.07002352 0.07533679 0.08222625 0.08846863
##         79         80         81         82         83         84         85
## 0.09495267 0.10217352 0.11027975 0.12228904 0.13242329 0.14378188 0.15973804
##         86         87         88         89         90         91         92
## 0.17031806 0.18645383 0.19644984 0.21614621 0.23280598 0.24830152 0.27118780
##         93         94         95         96         97         98         99
## 0.30072765 0.32529437 0.34906440 0.37129197 0.39790068 0.43152568 0.44901753
##        100        101        102        103        104        105
## 0.49334348 0.53084948 0.55720524 0.60396040 0.62235650 0.67361111
```

There are some very subtle differences between `lapply()`, `sapply()` and `vapply()` that I will not go into here but please make sure to work through the DataCamp courses to understand these. For the sake of this course, the apply() function allows us to do what we want sufficiently! Feel free to now have a go at the final set of exercises below:

I hope you enjoyed this small workshop on R Programming and feel more confident with the basics of what you can do in R. I strongly suggest you continuously test yourself in R and even make up your own problems/challenges, as physically programming is really the only way to remember and improve your programming skills. Also, please do not be afraid to search the web for tips and advice, I personally find this the simplest and quickest way to learn.

I am always more than happy to help with any questions you may have, so do please not hesitate to contact me!

5.7 Exercises

5.8 DataCamp course(s)

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.

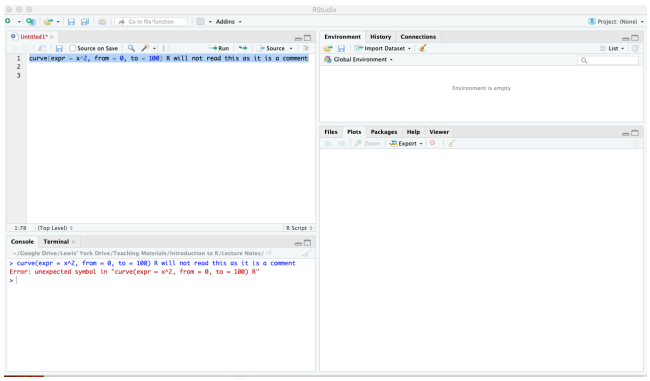


Figure A.1: Error when adding text to code line.

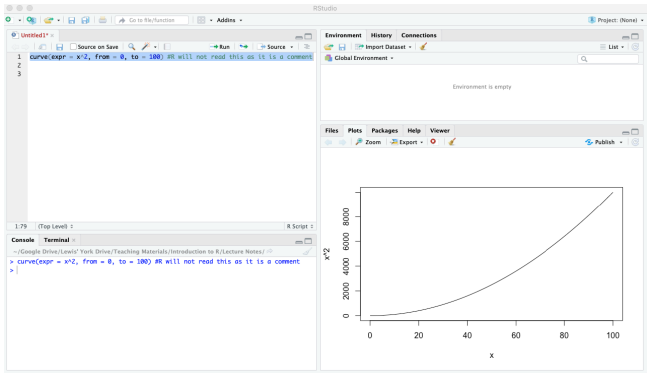


Figure A.2: Comment using hashtag symbol.

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:

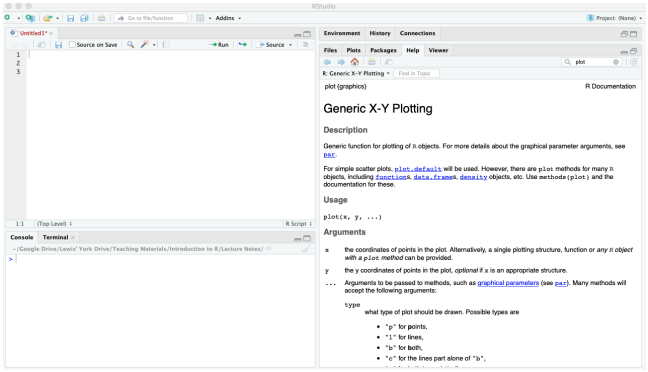


Figure A.3: Using the 'Help' tab for the 'plot()' function.

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

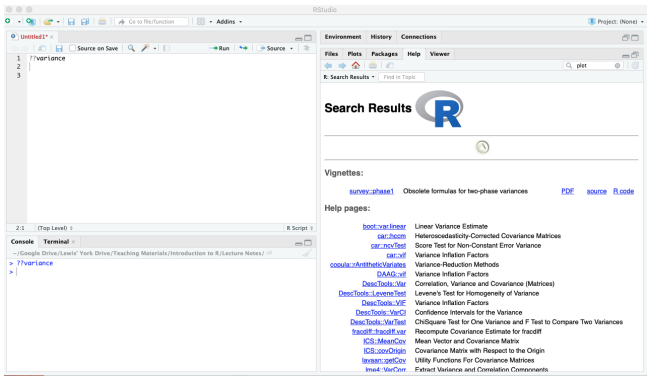


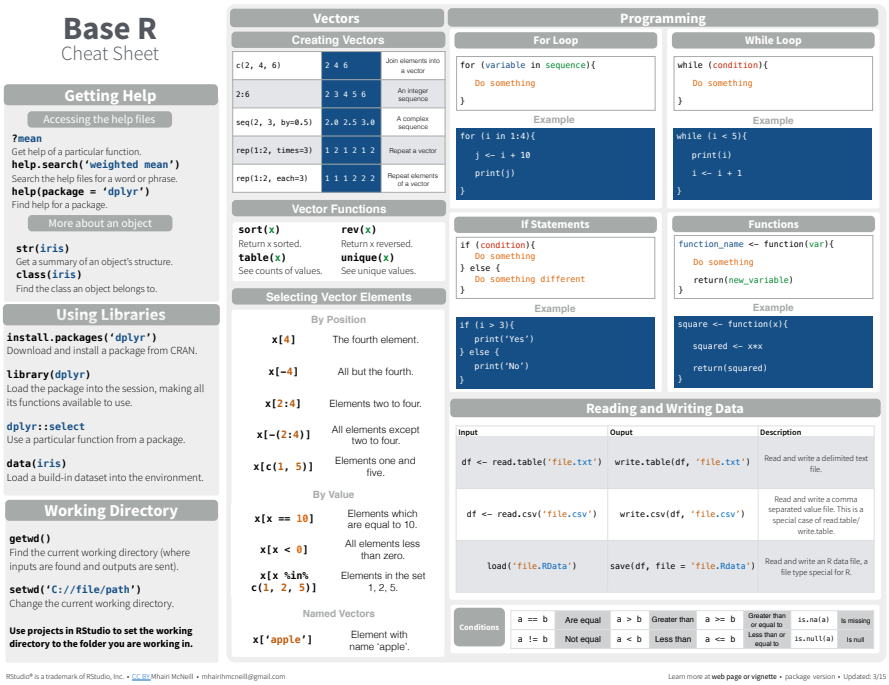
Figure A.4: Using the double question mark symbol for 'Help'.

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

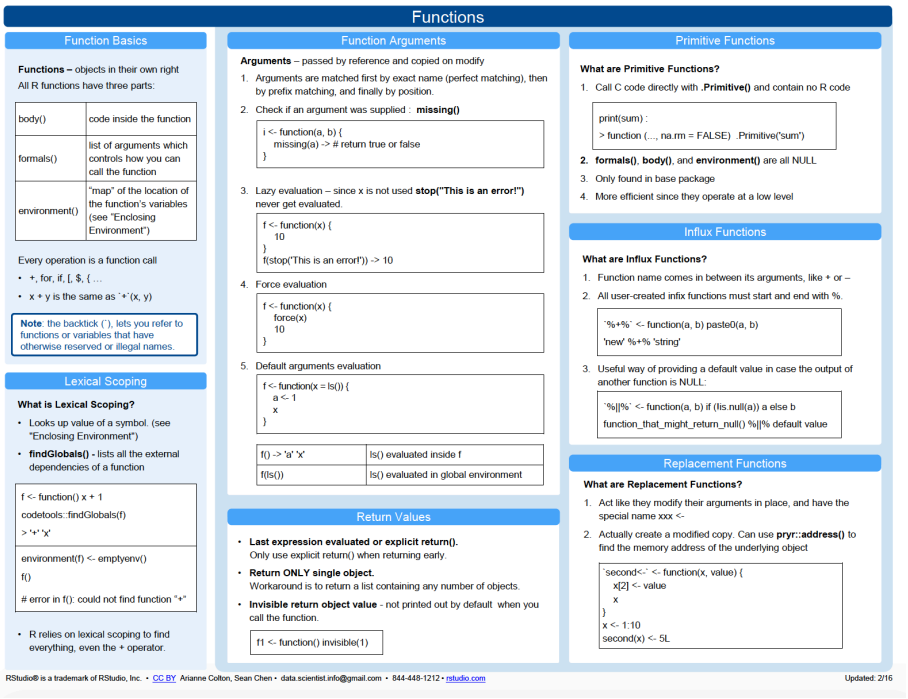
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df,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

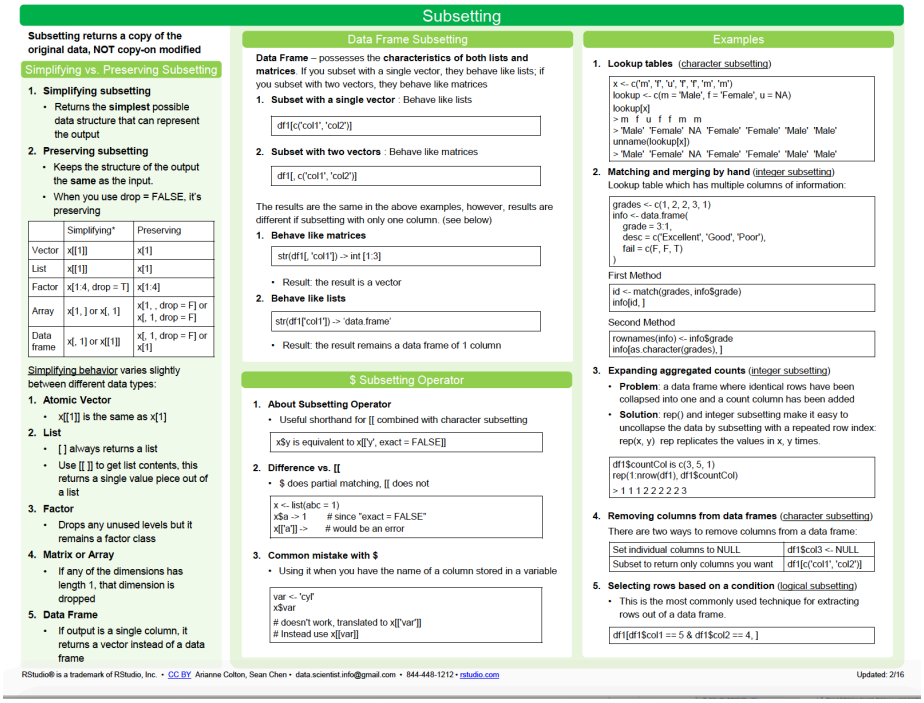
Replacement Functions

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
```

RStudio® is a trademark of RStudio, Inc. • CC BY Arianne Colton, Sean Chen • data.scientist.info@gmail.com • 844-448-1212 • rstudio.com Updated: 2/16



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:

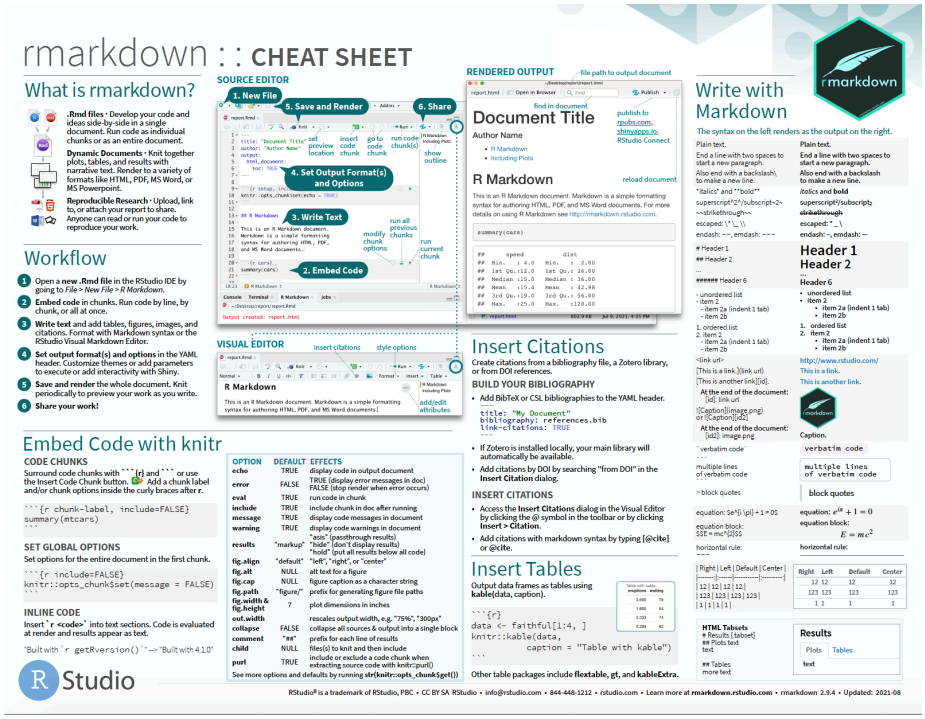
Set individual columns to NULL	df1$col3 <- NULL
Subset to return only columns you want	df1[c('col1', 'col2')]

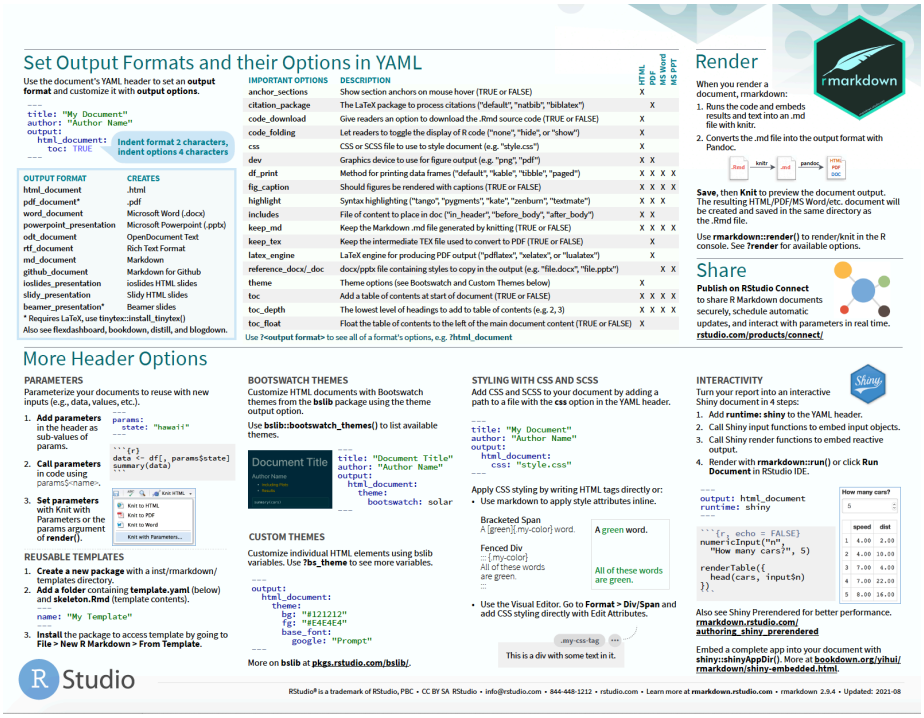
5. **Selecting rows based on a condition** (logical subsetting)
   - This is the most commonly used technique for extracting rows out of a data frame.

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

RStudio® is a trademark of RStudio, Inc. • CC BY Arianne Colton, Sean Chen • data.scientist.info@gmail.com • 844-448-1212 • rstudio.com Updated: 2/16

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