Basic R for Health Data Science

2025-03-22

1. Working Directories

The working directory is the default location where R will look for files you want to load and where it will put any files you save.

You can use the `getwd()` function in the Console which returns the file path of the current working directory.

```r
getwd() # returns the file path of the current working directory.
```

```
## [1] "C:/R works/R workshop"
```

You can also set your working directory using the `setwd()` function at the start of every R script if you are not using RStudio Project.

```r
setwd("C:/R works/R workshop") # manually set your working directory
```

However, the problem with `setwd()` is that it uses an absolute file path which is specific to the computer you are working on.

2. Basic operations

Before we continue, here are a few things to bear in mind as you work through this Chapter:

- R is case sensitive i.e. A is not the same as a and anova is not the same as Anova.
- Anything that follows a # symbol is interpreted as a comment and ignored by R

R is like a calculator, we can make mathematical operations, for example:

```r
2 + 2
```

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

The [1] in front of the result tells you that the observation number at the beginning of the line is the first observation.

There are a huge range of mathematical functions in R, some of the most useful include; `log()`, `log10()`, `exp()`, `sqrt()`.

```r
log(1)     # logarithm to base e
```

```
## [1] 0
```

```r
log10(1)  # logarithm to base 10
```

```
## [1] 0
```

```r
exp(1)     # natural antilog
```

```
## [1] 2.718282
```

```r
sqrt(4)    # square root
```

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

```r
4^2        # 4 to the power of 2
```

```
## [1] 16
```

```r
pi         # not a function but useful
```

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

2.1 Objects

R is a object-oriented programming language, this means that we create objects that contain information. In R you can achieve the same results using different approaches, for example, to create an object we just type a name for the object and assign it a value using the operators = or <-. We can make operations with objects of the same type, for example:

```r
x = 2 # create a new object with the = operator

y <- 2 # create a new object with the <- operator

z <- "R is cool" # you can also create an object which contains character string

x + y # make a operation with the objects
```

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

You can store more than one value using vectors, to create a vector of numbers we use `c()`. For example, we will store a sequence of numbers from 5 to 10 using 2 different approaches and then ask R if the objects are the same.

tip: using the keys "alt" + "-" will automatically add the operator <-. Choosing which assign operator to use is a matter of preference, I personally think that is easier reading code with the operator <-, but a lot of people uses =.

```r
x <- c(5, 6, 7, 8, 9, 10) # create a sequence form 5 to 10
x
```

```
## [1]  5  6  7  8  9 10
```

```r
y = 5:10 # create the same sequence but with a different approach
y
```

```
## [1]  5  6  7  8  9 10
```

```r
x == y # ask R if the objects have the same information
```

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

```r
z <- seq(from = 1, to = 5, by = 0.5) # another way to generate vectors of sequences
z
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
w <- rep(2, times = 10)   # repeats 2, 10 times
w
```

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

```r
v <- rep("abc", times = 3)    # repeats 'abc' 3 times
v
```

```
## [1] "abc" "abc" "abc"
```

```r
p <- rep(1:5, each = 3)   # repeats each element of the series 3 times
p
```

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

```r
t <- rep(c(3, 1, 10, 7), each = 3) # repeats each element of the series 3 times
t
```

```
##  [1]  3  3  3  1  1  1 10 10 10  7  7  7
```

When we have a vector, we can ask R specific values inside an object.

```r
# Here we ask the 3rd value from our sequence
x[3]
```

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

```r
# Now we multiply the 3rd value of the x sequence times the 5th value of the y sequence
x[3] * y[5]
```

```
## [1] 63
```

```r
mean(x)   # returns the mean of x
```

```
## [1] 7.5
```

```r
var(x)    # returns the variance of x
```

```
## [1] 3.5
```

```r
sd(x)     # returns the standard deviation of x
```

```
## [1] 1.870829
```

```r
length(x) # returns the number of elements in x
```

```
## [1] 6
```

2.2 Functions in R

R has a lot of base functions, but we can define new functions. When using R studio, the key Tab will help us to auto complete, this can help us a lot when we don't remember the exact name of the functions available. The best part of programming with R is that it has a very active community. Since its open source, anyone can create functions and compile them in a package (or library). We can download these packages and access new functions.

Functions in R require arguments, which we can see in the function documentation or if we press the key Tab when we are inside the function.

```r
# To get the sum of a vector of numbers inside an object we use sum()
sum(x)
```

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

We can put functions inside function, for example, to get the square root of a sum of the numbers in x we can use:

```r
sqrt(sum(x))
```

```
## [1] 6.708204
```

Making functions in R is not as complicated as it sounds and can be very useful when we need to do repetitive work. To define a function we need to include the arguments that we want for the function and what are we doing with those arguments. For example, the following function has only one argument which is a name (string) and just pastes some text before and after:

```
F1 <- function(name){
  x <- paste("Hola", name, "! welcome to the R world!") # paste the name with some text
  print(x)
}
# trying the function (Put your name)
F1(name = "Boat")
```

```
## [1] "Hola Boat ! welcome to the R world!"
```

Besides storing numbers in the objects in R, we can store text, matrices, tables, spatial objects, images, and other types of objects.

2.3 Vectors

The function we will learn about is the `c()` function. The `c()` function is short for concatenate and we use it to join together a series of values and store them in a data structure called a vector

```
my_vec <- c(2, 3, 1, 6, 4, 3, 3, 7) # create a vector

my_vec
```

```
## [1] 2 3 1 6 4 3 3 7
```

2.3.1 Extracting elements

Positional index

To extract elements based on their position we simply write the position inside the `[ ]`. For example, to extract the 3rd value of my_vec

```
my_vec[3] # extract the 3rd value
```

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

```
# if you want to store this value in another object
val_3 <- my_vec[3]
val_3
```

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

We can also extract more than one value by using the `c()` function inside the square brackets. Here we extract the 1st, 5th, 6th and 8th element from the my_vec object

```r
my_vec[c(1, 5, 6, 8)]
```

```
## [1] 2 4 3 7
```

Or we can extract a range of values using the : notation. To extract the values from the 3rd to the 8th elements.

```r
my_vec[3:8]
```

```
## [1] 1 6 4 3 3 7
```

Logical index

Another really useful way to extract data from a vector is to use a logical expression as an index.

```r
my_vec[my_vec > 4]
```

```
## [1] 6 7
```

Here, the logical expression is my_vec > 4 and R will only extract those elements that satisfy this logical condition. In this case only the 4th and 8th elements return a TRUE as their value is greater than 4.

```r
my_vec > 4
```

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

So what R is actually doing under the hood is equivalent to

```r
my_vec[c(FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE)]
```

```
## [1] 6 7
```

and only those element that are TRUE will be extracted.

In addition to the < and > operators you can also use composite operators to increase the complexity of your expressions. For example the expression for 'greater or equal to' is >=. To test whether a value is equal to a value we need to use a double equals symbol == and for 'not equal to' we use != (the ! symbol means 'not').

```r
my_vec[my_vec >= 4]        # values greater or equal to 4
```

```
## [1] 6 4 7
```

```r
my_vec[my_vec < 4]         # values less than 4
```

```
## [1] 2 3 1 3 3
```

```r
my_vec[my_vec <= 4]          # values less than or equal to 4
```

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

```r
my_vec[my_vec == 4]          # values equal to 4
```

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

```r
my_vec[my_vec != 4]          # values not equal to 4
```

```
## [1] 2 3 1 6 3 3 7
```

We can also combine multiple logical expressions using Boolean expressions. In R the & symbol means AND and the | symbol means OR. For example, to extract values in my_vec which are less than 6 AND greater than 2

```r
val26 <- my_vec[my_vec < 6 & my_vec > 2]

val26
```

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

or extract values in my_vec that are greater than 6 OR less than 3.

```r
val63 <- my_vec[my_vec > 6 | my_vec < 3]

val63
```

```
## [1] 2 1 7
```

2.3.2 Replacing elements

We can change the values of some elements in a vector using our `[ ]` notation in combination with the assignment operator <-. For example, to replace the 4th value of our my_vec object from 6 to 500

```r
my_vec[4] <- 500

my_vec
```

```
## [1]   2   3   1 500   4   3   3   7
```

We can also replace more than one value or even replace values based on a logical expression.

```r
# replace the 6th and 7th element with 100
my_vec[c(6, 7)] <- 100

my_vec
```

```
## [1]   2   3   1 500   4 100 100   7
```

```r
# replace element that are less than or equal to 4 with 1000
my_vec[my_vec <= 4] <- 1000

my_vec
```

```
## [1] 1000 1000 1000  500 1000  100  100    7
```

2.3.3 Ordering elements

In addition to extracting particular elements from a vector we can also order the values contained in a vector. To sort the values from lowest to highest value we can use the `sort()` function.

```r
vec_sort <- sort(my_vec)

vec_sort
```

```
## [1]    7  100  100  500 1000 1000 1000 1000
```

To reverse the sort, from highest to lowest, we can either include the decreasing = TRUE argument when using the `sort()` function

```r
vec_sort2 <- sort(my_vec, decreasing = TRUE)

vec_sort2
```

```
## [1] 1000 1000 1000 1000  500  100  100    7
```

or first sort the vector using the `sort()` function and then reverse the sorted vector using the `rev()` function. This is another example of nesting one function inside another function.

```r
vec_sort3 <- rev(sort(my_vec))

vec_sort3
```

```
## [1] 1000 1000 1000 1000  500  100  100    7
```

Whilst sorting a single vector is fun, perhaps a more useful task would be to sort one vector according to the values of another vector. To do this we should use the `order()` function in combination with `[ ]`. To demonstrate this let's create a vector called height containing the height of 5 different people and another vector called p.names containing the names of these people (so Joanna is 180 cm, Charlotte is 155 cm etc).

```r
height <- c(180, 155, 160, 167, 181)

height
```

```
## [1] 180 155 160 167 181
```

```
p.names <- c("Joanna","Charlotte","Helen","Karen","Amy")

p.names
```

```
## [1] "Joanna"    "Charlotte" "Helen"     "Karen"     "Amy"
```

Our goal is to order the people in p.names in ascending order of their height. The first thing we'll do is use the `order()` function with the height variable to create a vector called height_ord.

```
height_ord <- order(height)

height_ord
```

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

OK, what's going on here? The first value, 2, (remember ignore [1]) should be read as 'the smallest value of height is the second element of the height vector'. If we check this by looking at the height vector above, you can see that element 2 has a value of 155, which is the smallest value. The second smallest value in height is the 3rd element of height, which when we check is 160 and so on. The largest value of height is element 5 which is 181. Now that we have a vector of the positional indices of heights in ascending order (height_ord), we can extract these values from our p.names vector in this order.

```
names_ord <- p.names[height_ord]

names_ord
```

```
## [1] "Charlotte" "Helen"     "Karen"     "Joanna"    "Amy"
```

You're probably thinking 'what's the use of this?' Well, imagine you have a dataset which contains two columns of data and you want to sort each column. If you just use `sort()` to sort each column separately, the values of each column will become uncoupled from each other. By using the `order()` on one column, a vector of positional indices is created of the values of the column in ascending order. This vector can be used on the second column, as the index of elements which will return a vector of values based on the first column.

2.3.4 Vectorisation

One of the great things about R functions is that most of them are vectorised. This means that the function will operate on all elements of a vector without needing to apply the function on each element separately. For example, to multiple each element of a vector by 5 we can simply use

```
# create a vector
my_vec2 <- c(3, 5, 7, 1, 9, 20)

# multiply each element by 5
my_vec2 * 5
```

```
## [1]  15  25  35   5  45 100
```

Or we can add the elements of two or more vectors

```r
# create a second vector
my_vec3 <- c(17, 15, 13, 19, 11, 0)

# add both vectors
my_vec2 + my_vec3
```

```
## [1] 20 20 20 20 20 20
```

```r
# multiply both vectors
my_vec2 * my_vec3
```

```
## [1] 51 75 91 19 99  0
```

However, you must be careful when using vectorisation with vectors of different lengths as R will quietly recycle the elements in the shorter vector rather than throw a wobbly (error).

```r
# create a third vector
my_vec4 <- c(1, 2)

# add both vectors - quiet recycling!
my_vec2 + my_vec4
```

```
## [1]  4  7  8  3 10 22
```

2.3.5 Missing data

In R, missing data is usually represented by an NA symbol meaning 'Not Available'. From an R perspective missing data can be problematic as different functions deal with missing data in different ways. For example, let's say we collected air temperature readings over 10 days, but our thermometer broke on day 2 and again on day 9 so we have no data for those days.

```r
temp  <- c(7.2, NA, 7.1, 6.9, 6.5, 5.8, 5.8, 5.5, NA, 5.5)

temp
```

```
##  [1] 7.2  NA 7.1 6.9 6.5 5.8 5.8 5.5  NA 5.5
```

We now want to calculate the mean temperature over these days using the `mean()` function.

```r
mean_temp <- mean(temp)

mean_temp
```

```
## [1] NA
```

What's happened here? Why does the `mean()` function return an NA? Actually. If a vector has a missing value then the only possible value to return when calculating a mean is NA. R doesn't know that you perhaps want to ignore the NA values. Use help("mean") to get more information on mean function.

```
help("mean")
```

```
## starting httpd help server ... done
```

```
mean_temp <- mean(temp, na.rm = TRUE)

mean_temp
```

```
## [1] 6.2875
```

It's important to note that the NA values have not been removed from our temp object, rather the `mean()` function has just ignored them. The problem is that not all functions will have an na.rm = argument, they might deal with NA values differently. However, the good news is that every help file associated with any function will always tell you how missing data are handled by default.

2.4 Getting help

To access R's built-in help facility to get information on any function simply use the `help()` function. For example, to open the help page for our friend the `mean()` function.

```
help("mean")
```

or you can use the equivalent shortcut.

```
?mean
```

The `help()` function is useful if you know the name of the function. If you're not sure of the name, but can remember a key word then you can search R's help system using the `help.search()` function.

```
help.search("mean")
```

Or you can use the equivalent shortcut.

```
??mean
```

Another useful function is `apropos()`. This function can be used to list all functions containing a specified character string. For example, to find all functions with mean in their name

```
apropos("mean")
```

```
##  [1] ".colMeans"     ".rowMeans"     "colMeans"      "kmeans"
##  [5] "mean"          "mean.Date"     "mean.default"  "mean.difftime"
##  [9] "mean.POSIXct"  "mean.POSIXlt"  "mean_temp"     "rowMeans"
## [13] "weighted.mean"
```

An extremely useful function is `RSiteSearch()` which enables you to search for keywords and phrases in function help pages and vignettes for all CRAN packages, and in CRAN task views. This function allows you to access the https://www.r-project.org/search.html search engine directly from the Console with the results displayed in your web browser.

```
RSiteSearch("regression")
```

```
## A search query has been submitted to https://search.r-project.org
## The results page should open in your browser shortly
```

Exercise 1

1.1 Find the natural log, log to the base 10, square root and the exponential of 12.43.

1.2 Use the concatenate function `c()` to create a vector called `weight` containing the weight (in kg) of 10 children: 69, 62, 57, 59, 59, 64, 56, 66, 67, 66.

1.3 Calculate the mean, standard deviation, range of weights and the number of children of your weight vector. Next, extract the weights for the first five children using Positional indexes and store these weights in a new variable called first_five. Remember, you will need to use the square brackets `[ ]` to extract elements from a variable.

1.4 Use the `c()` function again to create another vector called `height` containing the height (in cm) of the same 10 children: 112, 102, 83, 84, 99, 90, 77, 112, 133, 112. Next, use the `summary()` function to summarise these data in the `height` object. Now, let's extract all the heights of children less than or equal to 99 cm and assign to a variable called shorter_child.

1.5 Now you can use the information in your weight and height variables to calculate the body mass index (BMI) for each child. The BMI is calculated as weight (in kg) divided by the square of the height (in meters). Store the results of this calculation in a variable called bmi.

3. Data in R

In this Chapter we'll go over the main data types in R and focus on some of the most common data structures. We will also cover how to import data into R from an external file, how to manipulate (wrangle) and summarise data and finally how to export data from R to an external file.

3.1 Data types

R has six basic types of data; numeric, integer, logical, complex and character. The final data type is raw which we won't cover as it's not useful 99.99% of the time.

- Numeric data are numbers that contain a decimal. Actually they can also be whole numbers but we'll gloss over that.
- Integers are whole numbers (those numbers without a decimal point).
- Logical data take on the value of either TRUE or FALSE. There's also another special type of logical called NA to represent missing values.
- Character data are used to represent string values. You can think of character strings as something like a word (or multiple words). A special type of character string is a factor, which is a string but with additional attributes (like levels or an order). We'll cover factors later.

R is (usually) able to automatically distinguish between different classes of data by their nature and the context in which they're used although you should bear in mind that R can't actually read your mind and you may have to explicitly tell R how you want to treat a data type. You can find out the type (or class) of any object using the `class()` function.

```
num <- 2.2
class(num)
```

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

```
char <- "hello"
class(char)
```

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

```
logi <- TRUE
class(logi)
```

```
## [1] "logical"
```

Alternatively, you can ask if an object is a specific class using using a logical test. The `is.class()` family of functions will return either a TRUE or a FALSE.

```
is.numeric(num)
```

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

```
is.character(num)
```

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

```
is.character(char)
```

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

```
is.logical(logi)
```

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

It can sometimes be useful to be able to change the class of a variable using the `as.class()` family of coercion functions, although you need to be careful when doing this as you might receive some unexpected results.

```
# coerce numeric to character
class(num)
```

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

```
num_char <- as.character(num)
num_char
```

```
## [1] "2.2"
```

```r
class(num_char)
```

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

```r
class(char)
```

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

```r
char_num <- as.numeric(char)
```

```
## Warning: NAs introduced by coercion
```

3.2 Data structures

Now that you've been introduced to some of the most important classes of data in R, let's have a look at some of main structures that we have for storing these data.

3.2.1 Scalars and vectors

Perhaps the simplest type of data structure is the vector. Vectors that have a single value (length 1) are called scalars. Vectors can contain numbers, characters, factors or logicals, but the key thing to remember is that all the elements inside a vector must be of the same class.

3.2.2 Matrices and arrays

Another useful data structure used in many disciplines such as population ecology, theoretical and applied statistics is the matrix. A matrix is simply a vector that has additional attributes called dimensions. Arrays are just multidimensional matrices.

A convenient way to create a matrix or an array is to use the `matrix()` and array() functions respectively. Below, we will create a matrix from a sequence 1 to 16 in four rows (nrow = 4) and fill the matrix row-wise (byrow = TRUE) rather than the default column-wise. When using the `array()` function we define the dimensions using the dim = argument, in our case 2 rows, 4 columns in 2 different matrices.

```r
my_mat <- matrix(1:16, nrow = 4, byrow = TRUE)

my_mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

```r
my_array <- array(1:16, dim = c(2, 4, 2))

my_array
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
```

Sometimes it's also useful to define row and column names for your matrix but this is not a requirement. To do this use the `rownames()` and `colnames()` functions.

```
rownames(my_mat) <- c("A", "B", "C", "D")

colnames(my_mat) <- c("a", "b", "c", "d")

my_mat
```

```
##    a  b  c  d
## A  1  2  3  4
## B  5  6  7  8
## C  9 10 11 12
## D 13 14 15 16
```

Once you've created your matrices you can do useful stuff with them and as you'd expect, R has numerous built in functions to perform matrix operations. Some of the most common are given below. For example, to transpose a matrix we use the transposition function `t()`.

```
my_mat_t <- t(my_mat)

my_mat_t
```

```
##   A B  C  D
## a 1 5  9 13
## b 2 6 10 14
## c 3 7 11 15
## d 4 8 12 16
```

To extract the diagonal elements of a matrix and store them as a vector we can use the `diag()` function.

```
my_mat_diag <- diag(my_mat)

my_mat_diag
```

```
## [1]  1  6 11 16
```

3.2.3 Lists

The next data structure we will quickly take a look at is a list. Whilst vectors and matrices are constrained to contain data of the same type, lists are able to store mixtures of data types. This makes for a very flexible data structure which is ideal for storing irregular or non-rectangular data.

To create a list we can use the `list()` function. Note how each of the three list elements are of different classes (character, logical, and numeric) and are of different lengths.

```
list_1 <- list(c("black", "yellow", "orange"),
               c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
               matrix(1:6, nrow = 3))
list_1
```

```
## [[1]]
## [1] "black"  "yellow" "orange"
##
## [[2]]
## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
##
## [[3]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Elements of the list can be named during the construction of the list

```
list_2 <- list(colours = c("black", "yellow", "orange"),
               evaluation = c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
               time = matrix(1:6, nrow = 3))
list_2
```

```
## $colours
## [1] "black"  "yellow" "orange"
##
## $evaluation
## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
##
## $time
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

or after the list has been created using the names() function.

```
names(list_1) <- c("colours", "evaluation", "time")
list_1
```

```
## $colours
## [1] "black"  "yellow" "orange"
```

```
##
## $evaluation
## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
##
## $time
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

3.2.4 Data frames

By far the most commonly used data structure to store data in is the data frame. A data frame is a powerful two-dimensional object made up of rows and columns which looks superficially very similar to a matrix. However, whilst matrices are restricted to containing data all of the same type, data frames can contain a mixture of different types of data. Typically, in a data frame each row corresponds to an individual observation and each column corresponds to a different measured or recorded variable.

We can construct a data frame from existing data objects such as vectors using the `data.frame()` function. As an example, let's create three vectors p.height, p.weight and p.names and include all of these vectors in a data frame object called dataf.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")

dataf <- data.frame(height = p.height, weight = p.weight, names = p.names)
dataf
```

```
##   height weight     names
## 1    180     65    Joanna
## 2    155     50 Charlotte
## 3    160     52     Helen
## 4    167     58     Karen
## 5    181     70       Amy
```

```
dim(dataf)   # 5 rows and 3 columns
```

```
## [1] 5 3
```

Another really useful function which we use all the time is `str()` which will return a compact summary of the structure of the data frame object (or any object for that matter).

```
str(dataf)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ height: num  180 155 160 167 181
##  $ weight: num  65 50 52 58 70
##  $ names : chr  "Joanna" "Charlotte" "Helen" "Karen" ...
```

The `str()` function gives us the data frame dimensions and also reminds us that dataf is a data.frame type object. It also lists all of the variables (columns) contained in the data frame, tells us what type of data the variables contain and prints out the first five values.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")

dataf <- data.frame(height = p.height, weight = p.weight, names = p.names,
                                        stringsAsFactors = TRUE)
str(dataf)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ height: num  180 155 160 167 181
##  $ weight: num  65 50 52 58 70
##  $ names : Factor w/ 5 levels "Amy","Charlotte",..: 4 2 3 5 1
```

3.3 Importing data

R can import data in different formats. The most common are excel files (.csv, .xls y .xlsx), text files .txt and spatial data .shp, which we will talk about more in detail later. To read .xls, .xlsx and .shp files we will need to install some libraries. To install a new library you need to be connected to the internet and use the function `install.packages()` to install the library. Once it has been installed, you can load the library using the function `library()`.

```
# If we dont have the library installed, we use:
# install.packages("readxl")
library(readxl) # load the library

# Import the excel file
data <- read.csv("Data/data1.csv")
```

The most popular format for tables in R are called data.frame, when we import the data from a .csv o .xlsx. We can examine what kind of object it is using the function class(), an object can have more than one type of class.

```
class(data)
```

```
## [1] "data.frame"
```

```
str(data)
```

```
## 'data.frame':    600 obs. of  12 variables:
##  $ patient_no: int  4 6 7 15 17 20 23 30 32 35 ...
##  $ gender    : int  2 2 2 2 2 2 1 2 1 2 ...
##  $ age       : int  52 52 50 52 50 52 52 53 50 50 ...
##  $ bmi       : num  21.2 30.1 27.6 32 27.9 ...
##  $ food1     : int  7 4 3 4 6 6 1 6 3 2 ...
##  $ food2     : int  1 1 2 1 1 5 1 6 3 2 ...
##  $ food3     : int  3 3 5 3 5 3 1 2 3 2 ...
##  $ food4     : int  1 1 3 1 1 3 1 2 3 2 ...
##  $ food5     : int  1 3 2 2 5 3 1 3 3 2 ...
##  $ food6     : int  1 2 3 1 1 3 1 3 3 2 ...
##  $ food7     : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ CRC       : int  0 0 0 0 0 0 0 0 0 0 ...
```

3.4 Wrangling data frames

3.4.1 Ordering data frames

Remember when we used the function `order()` to order one vector based on the order of another vector. This comes in very handy if you want to reorder rows in your data frame.

```
age_ord <- data[order(data$age), ]

head(age_ord, 10)
```

```
##    patient_no gender age   bmi food1 food2 food3 food4 food5 food6 food7 CRC
## 3           7      2  50 27.62     3     2     5     3     2     3     1   0
## 5          17      2  50 27.93     6     1     5     1     5     1     1   0
## 9          32      1  50 23.83     3     3     3     3     3     3     1   0
## 10         35      2  50 25.11     2     2     2     2     2     2     1   0
## 12         40      2  50 30.89     8     8     1     8     8     8     1   1
## 13         44      1  50 34.78     4     3     4     3     4     3     1   0
## 14         48      2  50 22.89     6     5     4     3     2     4     4   1
## 17         56      2  50 21.23     5     1     5     1     5     5     1   0
## 18         57      2  50 26.06     1     1     1     1     1     1     1   0
## 19         63      2  50 23.28     7     5     5     4     5     4     1   0
```

We can also order by descending order of a variable using the decreasing = TRUE argument.

```
bmi_ord <- data[order(data$bmi, decreasing = TRUE), ]

head(bmi_ord, 10)
```

```
##     patient_no gender age   bmi food1 food2 food3 food4 food5 food6 food7 CRC
## 543       1676      2  53 40.37     7     1     7     1     2     1     1   0
## 510       1518      2  54 38.59     7     5     7     5     3     3     1   0
## 196        595      2  51 37.65     4     4     2     2     2     4     4   1
## 124        356      2  50 36.96     3     3     3     3     1     1     1   0
## 476       1393      2  52 35.82     5     5     5     5     5     1     1   0
## 398       1125      2  54 35.70     6     3     5     5     4     3     2   0
## 389       1113      2  55 35.16     6     1     6     1     2     1     1   0
## 281        818      1  52 35.05     2     2     2     2     2     1     1   0
## 13          44      1  50 34.78     4     3     4     3     4     3     1   0
## 142        420      2  50 34.17     6     4     6     5     3     2     1   0
```

What if we wanted to order data by ascending order of age but descending order of bmi? We can use a simple trick by adding a - symbol before the data$bmi variable when we use the `order()` function.

```
age_revbmi_ord <- data[order(data$age, -data$bmi), ]

head(age_revbmi_ord, 10)
```

```
##     patient_no gender age   bmi food1 food2 food3 food4 food5 food6 food7 CRC
## 124        356      2  50 36.96     3     3     3     3     1     1     1   0
## 13          44      1  50 34.78     4     3     4     3     4     3     1   0
## 142        420      2  50 34.17     6     4     6     5     3     2     1   0
## 222         660       2  50 33.33     5     1     3     1     2     3     1   1
## 186         572       2  50 31.05     8     1     3     1     3     1     1   0
## 12           40       2  50 30.89     8     8     1     8     8     8     1   1
## 103         289       2  50 29.82     7     1     4     1     4     1     1   0
## 168         517       2  50 29.78     6     6     5     5     4     3     1   0
## 234         700       2  50 29.47     6     1     4     2     4     2     1   0
## 55          162       2  50 29.30     4     4     4     4     1     1     1   0
```

3.4.2 Adding columns and rows

Sometimes it's useful to be able to add extra rows and columns of data to our data frames. There are multiple ways to achieve this depending on your circumstances. To simply append additional rows to an existing data frame we can use the `rbind()` function and to append columns the `cbind()` function. Let's create a couple of test data frames to see this in action using the `data.frame()` function.

```
# rbind for rows
df1 <- data.frame(id = 1:4, height = c(120, 150, 132, 122),
                        weight = c(44, 56, 49, 45))

df1
```

```
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45
```

```
df2 <- data.frame(id = 5:6, height = c(119, 110),
                        weight = c(39, 35))

df2
```

```
##   id height weight
## 1  5    119     39
## 2  6    110     35
```

```
df3 <- data.frame(id = 1:4, height = c(120, 150, 132, 122),
                        weight = c(44, 56, 49, 45))

df3
```

```
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45
```

```
df4 <- data.frame(location = c("UK", "CZ", "CZ", "UK"))


df4
```

```
##   location
## 1       UK
## 2       CZ
## 3       CZ
## 4       UK
```

We can use the `rbind()` function to append the rows of data in df2 to the rows in df1 and assign the new data frame to df_rcomb.

```
df_rcomb <- rbind(df1, df2)

df_rcomb
```

```
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45
## 5  5    119     39
## 6  6    110     35
```

```
df_ccomb <- cbind(df3, df4)

df_ccomb
```

```
##   id height weight location
## 1  1    120     44       UK
## 2  2    150     56       CZ
## 3  3    132     49       CZ
## 4  4    122     45       UK
```

Another situation when adding a new column to a data frame is useful is when you want to perform some kind of transformation on an existing variable.

```
# log10 transformation
df_rcomb$height_log10 <- log10(df_rcomb$height)

df_rcomb
```

```
##   id height weight height_log10
## 1  1    120     44     2.079181
## 2  2    150     56     2.176091
## 3  3    132     49     2.120574
## 4  4    122     45     2.086360
## 5  5    119     39     2.075547
## 6  6    110     35     2.041393
```

3.4.3 Merging data frames

Instead of just appending either rows or columns to a data frame we can also merge two data frames together.

```r
data2 <- read.csv("Data/data2.csv")

data4 <- read.csv("Data/data4.csv")

merge_df <- merge(x = data2, y = data4)

head(merge_df, 10)
```

```
##    pat_no drug hbvlog hbeag afp liverparenchyma liversurface_ lesion HCC
## 1       2    1   1.30     2 3.0               1             0      0   0
## 2       6    2   1.30     2 4.4               0             0      0   0
## 3       7    2   1.30     1 3.3               0             0      1   1
## 4       9    2   1.30     2 1.1               2             0      1   1
## 5      14    1   1.30     1 2.6               0             0      0   0
## 6      19    2   1.30     2 1.4               0             0      0   0
## 7      20    2   1.53     2 2.2               0             0      0   0
## 8      23    1   1.30     1 2.8               0             0      0   0
## 9      47    2   1.30     1 4.6               1             0      0   0
## 10     53    2   1.30     2 2.7               1             0      0   0
##    patient_no age fh dm smoking  bmi fit hematocrit coloscopy
## 1           1  58  2  1       2 32.6   2       43.1         3
## 2           1  58  2  1       2 32.6   2       43.1         3
## 3           1  58  2  1       2 32.6   2       43.1         3
## 4           1  58  2  1       2 32.6   2       43.1         3
## 5           1  58  2  1       2 32.6   2       43.1         3
## 6           1  58  2  1       2 32.6   2       43.1         3
## 7           1  58  2  1       2 32.6   2       43.1         3
## 8           1  58  2  1       2 32.6   2       43.1         3
## 9           1  58  2  1       2 32.6   2       43.1         3
## 10          1  58  2  1       2 32.6   2       43.1         3
```

combining 2 data frames based on similar variable

```r
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 4.4.2

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
new_df <- left_join(data2, data4, by = c("pat_no" = "patient_no"))

head(new_df, 10)
```

```
##    pat_no drug hbvlog hbeag afp liverparenchyma liversurface_ lesion HCC age fh
## 1       2    1   1.30     2 3.0               1             0      0   0  59  2
## 2       6    2   1.30     2 4.4               0             0      0   0  50  2
## 3       7    2   1.30     1 3.3               0             0      1   1  NA NA
## 4       9    2   1.30     2 1.1               2             0      1   1  57  2
## 5      14    1   1.30     1 2.6               0             0      0   0  NA NA
## 6      19    2   1.30     2 1.4               0             0      0   0  57  1
## 7      20    2   1.53     2 2.2               0             0      0   0  NA NA
## 8      23    1   1.30     1 2.8               0             0      0   0  62  2
## 9      47    2   1.30     1 4.6               1             0      0   0  NA NA
## 10     53    2   1.30     2 2.7               1             0      0   0  50  2
##    dm smoking  bmi fit hematocrit coloscopy
## 1   2       2 20.0   2       39.1         3
## 2   2       2 16.9   2       43.1         3
## 3  NA      NA   NA  NA         NA        NA
## 4   2       2 25.2   2       44.5         3
## 5  NA      NA   NA  NA         NA        NA
## 6   2       2 26.0   2       38.3         3
## 7  NA      NA   NA  NA         NA        NA
## 8   2       2 26.7   2       39.8         3
## 9  NA      NA   NA  NA         NA        NA
## 10  2       2 23.4   2       40.7         3
```

3.4.4 Summarising data frames

A really useful starting point is to produce some simple summary statistics of all of the variables in our data frame using the `summary()` function.

```
summary(data)
```

```
##    patient_no         gender           age             bmi       
##  Min.   :   4.0   Min.   :1.000   Min.   :50.00   Min.   :14.68  
##  1st Qu.: 456.8   1st Qu.:2.000   1st Qu.:51.00   1st Qu.:21.48  
##  Median : 878.5   Median :2.000   Median :53.00   Median :23.91  
##  Mean   : 888.1   Mean   :1.852   Mean   :52.64   Mean   :24.29  
##  3rd Qu.:1291.5   3rd Qu.:2.000   3rd Qu.:54.00   3rd Qu.:26.44  
##  Max.   :1872.0   Max.   :2.000   Max.   :55.00   Max.   :40.37  
##      food1           food2           food3           food4          food5     
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.00   Min.   :1.00  
##  1st Qu.:4.000   1st Qu.:1.000   1st Qu.:3.000   1st Qu.:1.00   1st Qu.:1.00  
##  Median :6.000   Median :2.000   Median :5.000   Median :2.00   Median :2.00  
##  Mean   :5.242   Mean   :3.083   Mean   :4.417   Mean   :2.63   Mean   :2.58  
##  3rd Qu.:6.000   3rd Qu.:5.000   3rd Qu.:6.000   3rd Qu.:4.00   3rd Qu.:3.00  
##  Max.   :8.000   Max.   :8.000   Max.   :8.000   Max.   :8.00   Max.   :8.00  
##      food6           food7            CRC         
##  Min.   :1.000   Min.   :1.000   Min.   :0.00000  
##  1st Qu.:1.000   1st Qu.:1.000   1st Qu.:0.00000  
##  Median :1.000   Median :1.000   Median :0.00000  
##  Mean   :1.777   Mean   :1.228   Mean   :0.08333  
##  3rd Qu.:2.000   3rd Qu.:1.000   3rd Qu.:0.00000  
##  Max.   :8.000   Max.   :8.000   Max.   :1.00000  
```

If we wanted to summarise a smaller subset of variables in our data frame we can use our indexing skills in combination with the `summary()` function.

```r
summary(data[, 2:4])
```

```
##      gender           age              bmi
##  Min.   :1.000   Min.   :50.00   Min.   :14.68
##  1st Qu.:2.000   1st Qu.:51.00   1st Qu.:21.48
##  Median :2.000   Median :53.00   Median :23.91
##  Mean   :1.852   Mean   :52.64   Mean   :24.29
##  3rd Qu.:2.000   3rd Qu.:54.00   3rd Qu.:26.44
##  Max.   :2.000   Max.   :55.00   Max.   :40.37
```

And to summarise a single variable

```r
summary(data$bmi)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   14.68   21.48   23.91   24.29   26.44   40.37
```

`table()` function can be used to build contingency tables of different combinations of factor levels.

```r
table(data$gender)
```

```
##
##   1   2
##  89 511
```

We can extend this further by producing a table of counts for each combination of gender and age factor levels.

```r
table(data$gender, data$age)
```

```
##
##      50  51  52  53  54  55
##   1  12  16  16  21  17   7
##   2  59  77  94 107  88  86
```

We can even build more complicated contingency tables using more variables using `xtabs()` function.

```r
xtabs(~ gender + age + CRC, data = data)
```

```
## , , CRC = 0
##
##       age
## gender 50 51 52 53 54 55
##      1 10 14 14 18 14  6
##      2 54 74 88 98 80 80
##
## , , CRC = 1
##
##       age
## gender 50 51 52 53 54 55
##      1  2  2  2  3  3  1
##      2  5  3  6  9  8  6
```

We can also summarise our data for each level of a factor variable. To do this we will use the `mean()` function and apply this to the age variable for each level of gender using the `tapply()` function.

```
tapply(data$age, data$gender, mean)
```

```
##        1        2 
## 52.40449 52.67710
```

3.4.5 Exporting data

Here is an example of how you can export you data in R for .csv file.

```
write.csv(df1, file = 'Data/df1.csv')
```

Exercise 2

Time for a quick description of the 'whaledata.csv' dataset to get your bearings. These data were collected during two research cruises in the North Atlantic in May and October 2003. During these two months the research vessel visited multiple stations (areas) and marine mammal observers recorded the number of whales at each of these stations. The time the vessel spent at each station was also recorded along with other site specific variables such as the latitude and longitude, water depth and gradient of the seabed. The researchers also recorded the ambient level of sub-surface noise with a hydrophone and categorised this variable into 'low', 'medium' or 'high'. The structure of these data is known as a rectangular dataset with no missing cells. Each row is an individual observation and each column a separate variable. The variable names are contained in the first row of the dataset (aka a header)

2.1 Now let's import the 'whaledata.csv' file into R. To do this you will use the workhorse function of data importing, `read.csv()` and assign it a name "whale"

2.2 Use the `head()` function to display the first 5 rows of your dataframe. Again, this is likely to just fill up your console. Another option is to use the `str()` function to display the structure of the dataset and a neat summary of your variables. How many observations does this dataset have? How many variables are in this dataset? What type of variables are month and water.noise?

2.3 You can get another useful summary of your dataframe by using the `summary()` function. This will provide you with some useful summary statistics for each variable. Which variables have missing values and how many?

2.4 Another useful way to manipulated your dataframes is to sort the rows based on the value of a variable. Use the `order()` function to sort all rows in the whale dataframe in ascending order of depth. Store this sorted dataframe in a variable called whale.depth.sort.

2.5 Now for something a little more complicated. Sort all rows in the whale dataframe by ascending order of depth within each level of water noise. The trick here is to remember that you can order by more than one variable when using the `order()` function.

2.6 Repeat the previous ordering in Question 2.5 but this time order by descending order of depth within each level of water noise.

2.7 Knowing how many observations are present for each factor level is useful to determine whether you have an adequate sample size. Use the `table()` function to determine the number of observations for each level of water noise. Next use the same function to display the number of observations for each combination of water noise and month.

4. Graphics with base R

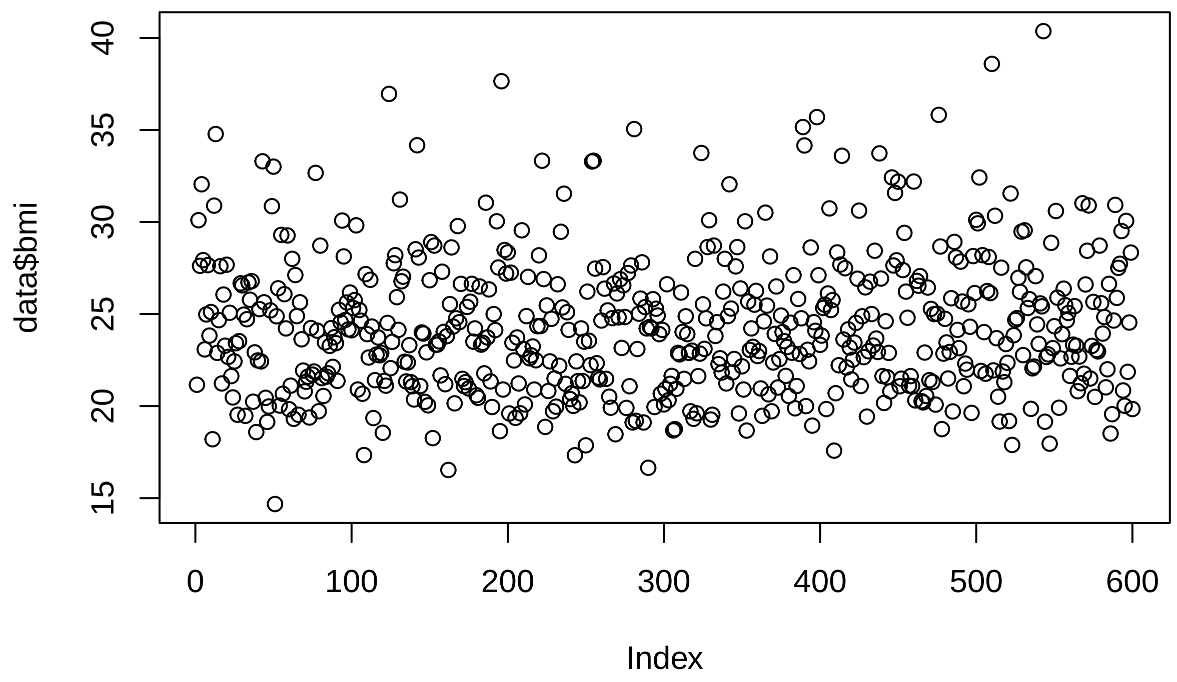
4.1 Simple base R plots

There are many functions in R to produce plots ranging from the very basic to the highly complex. We'll introduce you to most of the common methods of graphing data and describe how to customise your graphs later on.

4.1.1 Scatterplots

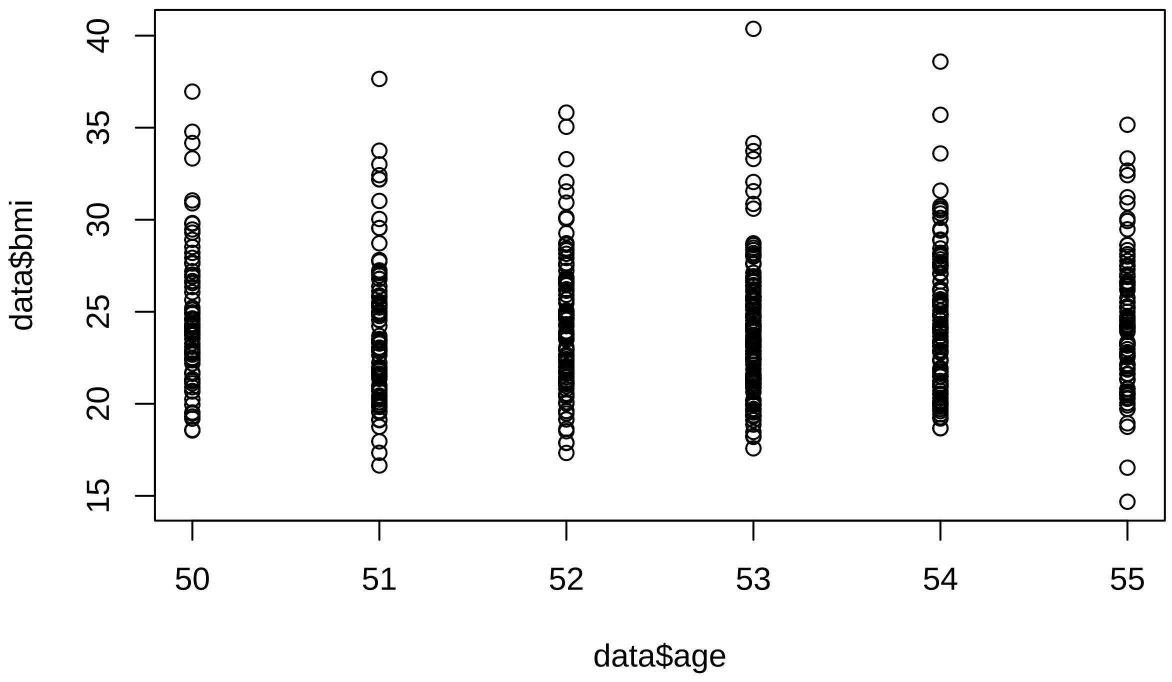
The most common high level function used to produce plots in R is the `plot()` function. For example, let's plot the bmi of patients from our data.

```r
plot(data$bmi)
```

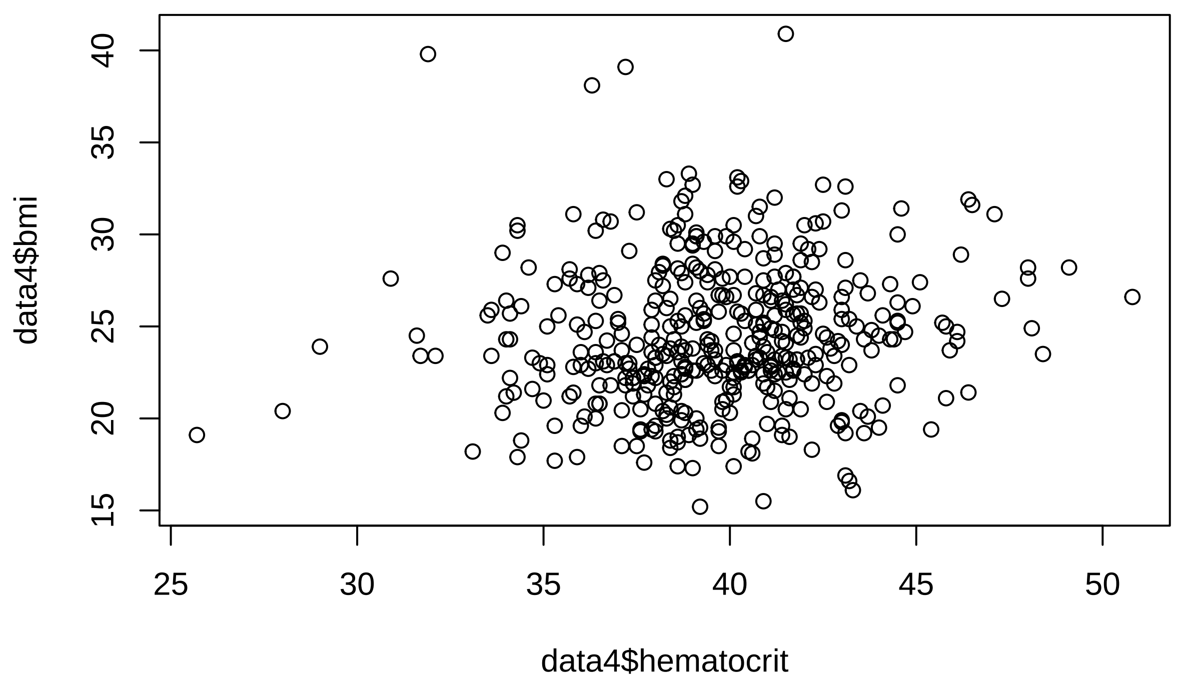


To plot a scatterplot of one numeric variable against another numeric variable we just need to include both variables as arguments when using the plot() function.

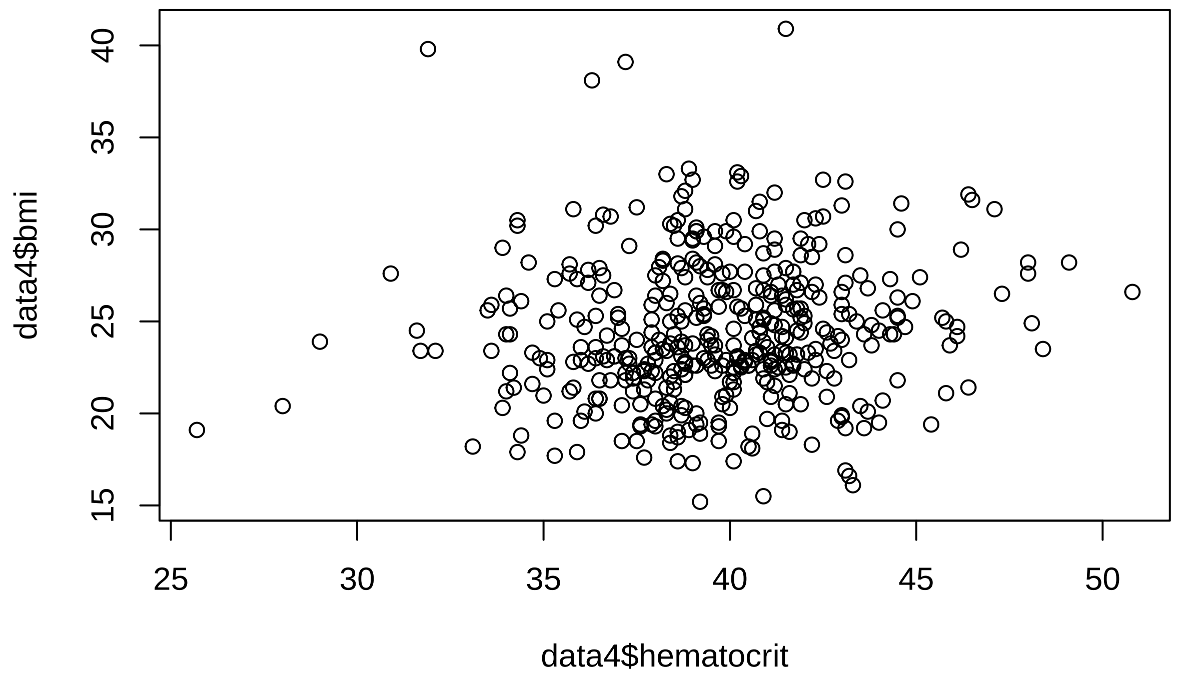
```r
plot(x = data$age, y = data$bmi)
```



```r
plot(x = data4$hematocrit, y = data4$bmi)
```



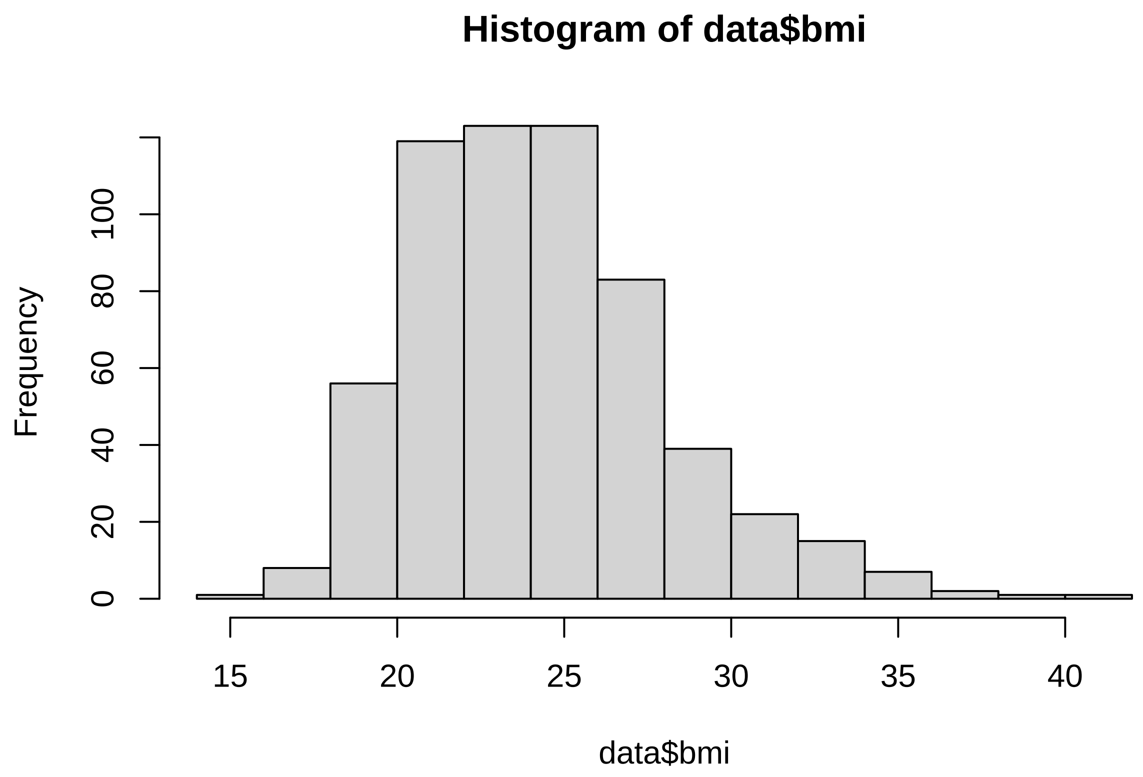
```r
plot(data4$bmi ~ data4$hematocrit)
```



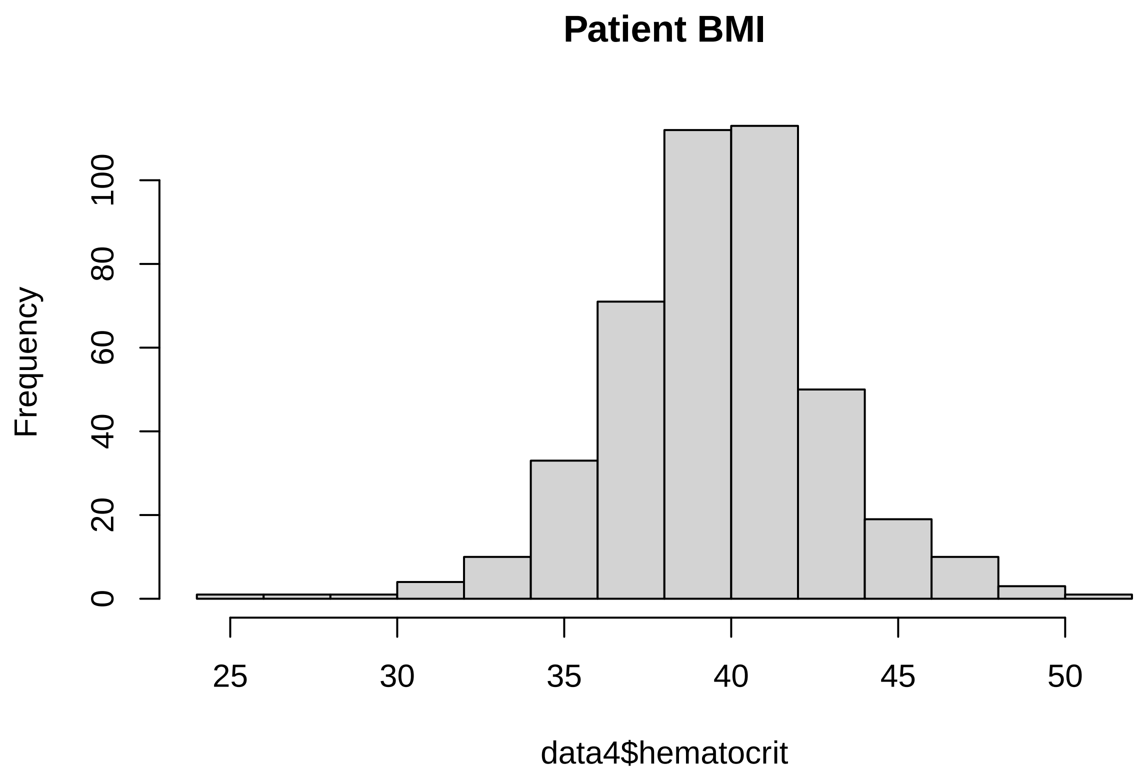
4.1.2 Histograms

Frequency histograms are useful when you want to get an idea about the distribution of values in a numeric variable. The `hist()` function takes a numeric vector as its main argument.

```
hist(data$bmi)
```



```r
hist(data4$hematocrit, main = "Patient BMI")
```

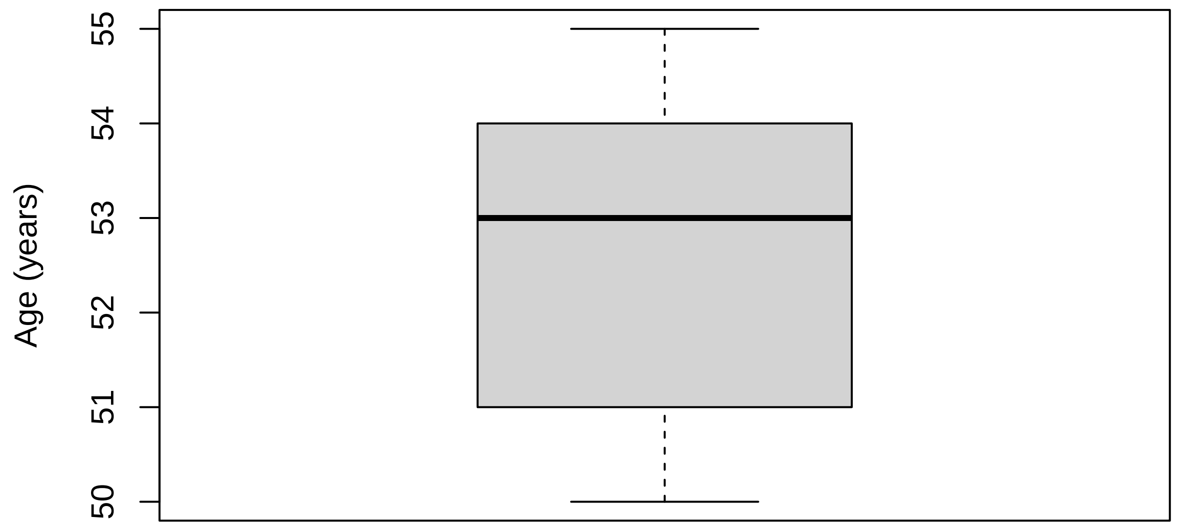


4.1.3 Box plots

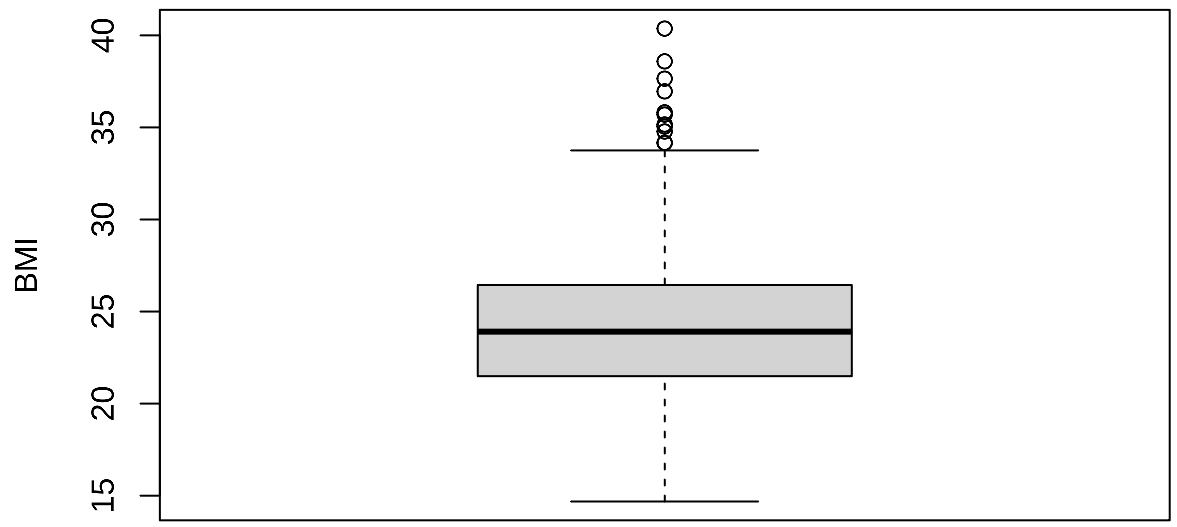
Boxplots (or box-and-whisker plots) are very useful when you want to graphically summarise the distribution of a variable, identify potential unusual values and compare distributions between different groups.

To create a boxplot in R we use the `boxplot()` function.

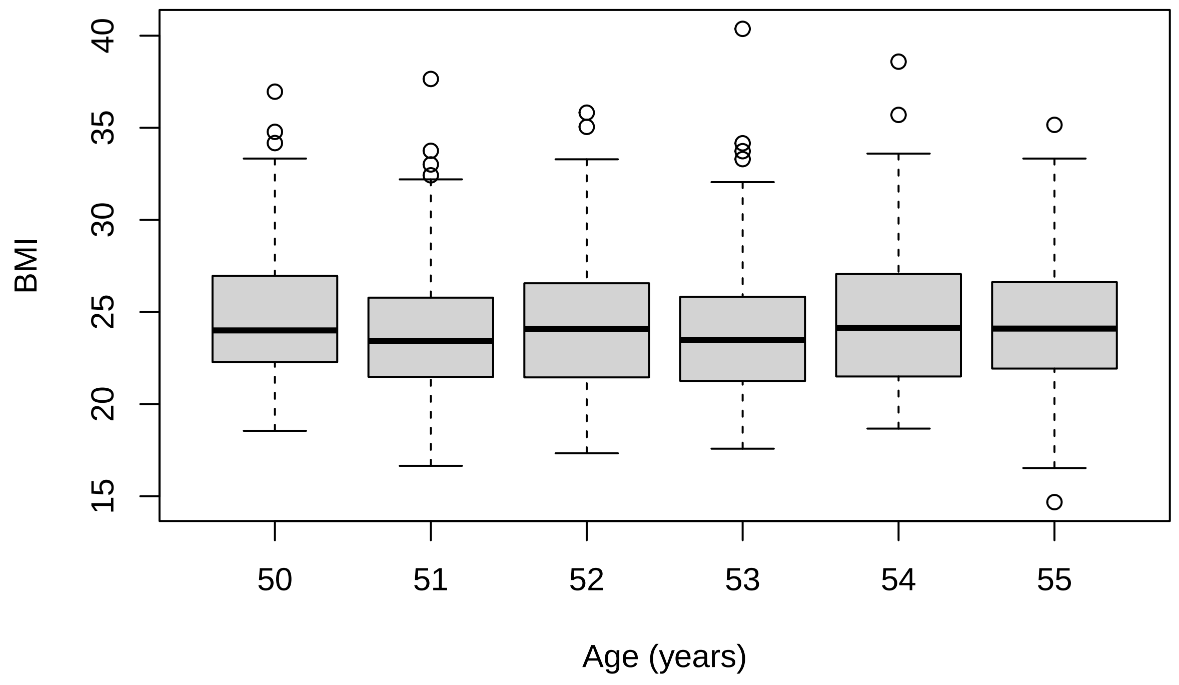
```r
boxplot(data$age, ylab = "Age (years)")
```



```r
boxplot(data$bmi, ylab = "BMI")
```



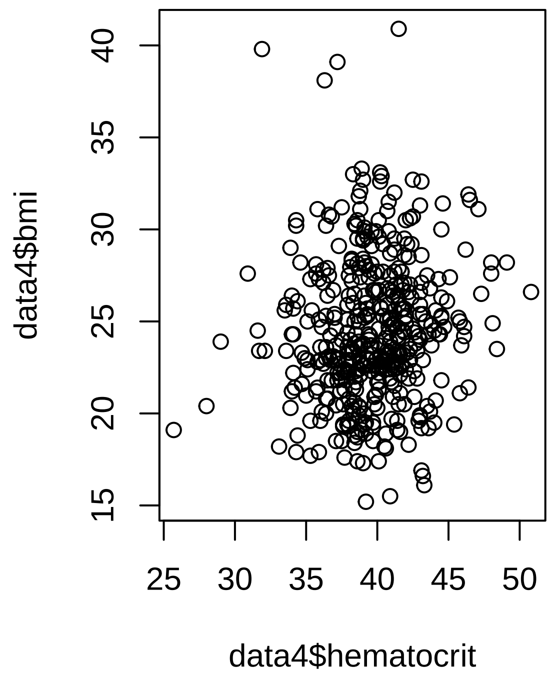
```
boxplot(bmi ~ age, data = data,
        ylab = "BMI", xlab = "Age (years)")
```

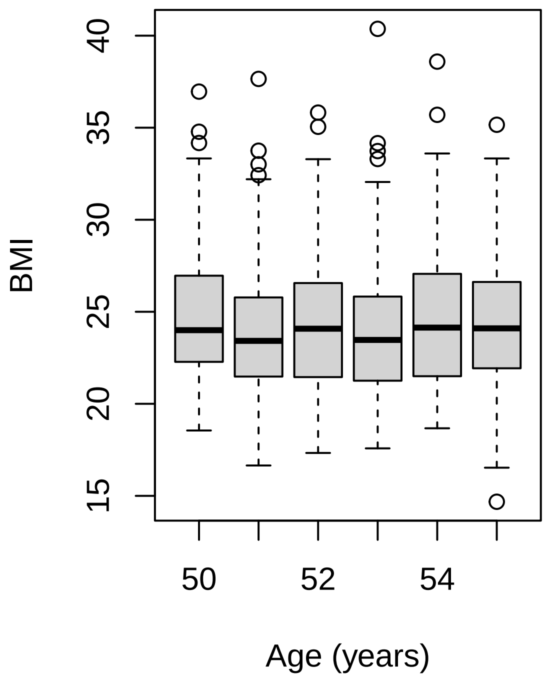


4.2 Multiple graphs

There are a number of different methods for plotting multiple graphs within the same graphics device. However these functions rely on plotting multiple graphs in different panels within the same plot. If you want to plot separate plots within the same graphics device you'll need a different approach. One of the most common methods is to use the main graphical function `par()` to split the plotting device up into a number of defined sections using the mfrow = argument. With this method, you first need to specify the number of rows and columns of plots you would like and then run the code for each plot. For example, to plot two graphs side by side we would use par(mfrow = c(1, 2)) to split the device into 1 row and two columns.

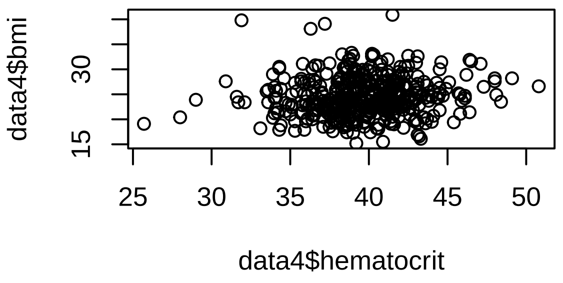
```r
par(mfrow = c(1, 2))

plot(x = data4$hematocrit, y = data4$bmi)

boxplot(bmi ~ age, data = data,
        ylab = "BMI", xlab = "Age (years)")
```

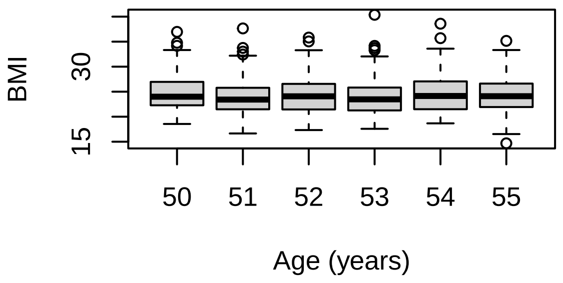


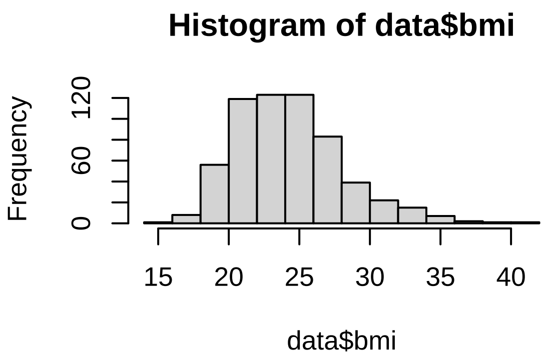


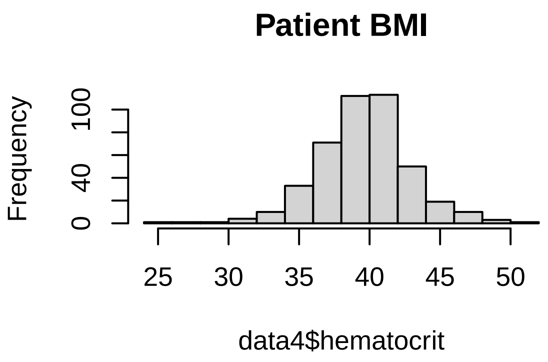
Or if we wanted to plot four plots we can split our plotting device into 2 rows and 2 columns.

```r
par(mfrow = c(2, 2))

plot(x = data4$hematocrit, y = data4$bmi)

boxplot(bmi ~ age, data = data,
        ylab = "BMI", xlab = "Age (years)")

hist(data$bmi)

hist(data4$hematocrit, main = "Patient BMI")
```









4.3 Exporting plots

```
# Open a PDF device
pdf("bmi_histogram.pdf")

# Plot the histogram
hist(data$bmi)

# Close the PDF device
dev.off()
```

```
## pdf
##   2
```

Exercise 3

These data were originally collected as part of a study published in Aquatic Living Resources1 in 2005. The aim of the study was to investigate the seasonal patterns of investment in somatic and reproductive tissues in the long finned squid *Loligo forbesi* caught in Scottish waters. Squid were caught monthly from December 1989 - July 1991 (month and year variables). After capture, each squid was given a unique specimen code, weighed (weight) and the sex determined (sex - only female squid are included here). The size of individuals was also measured as the dorsal mantle length (DML) and the mantle weight measured without internal organs (eviscerate.weight). The gonads were weighed (ovary.weight) along with the accessory reproductive organ (the nidamental gland, nid.weight, nid.length). Each individual was also assigned a categorical measure of maturity (maturity.stage, ranging from 1 to 5 with 1 = immature, 5 = mature). The digestive gland weight (dig.weight) was also recorded to assess nutritional status of the individual.

3.1 Import the 'squid.csv' file into R using the `read.csv()` function and assign it to a variable named squid. Use the str() function to display the structure of the dataset and the `summary()` function to summarise the dataset. How many observations are in this dataset? How many variables? The year, month and maturity.stage variables were coded as integers in the original dataset. Here we would like to code them as factors. Create a new variable for each of these variables in the squid dataframe and recode them as factors. Use the `str()` function again to check the coding of these new variables.

3.2 When exploring your data it is often useful to visualize the distribution of continuous variables. Create histograms using `hist()` function for the variables; DML, weight, eviscerate.weight and ovary.weight.Then plot all the histograms in only one image using `par()` function.

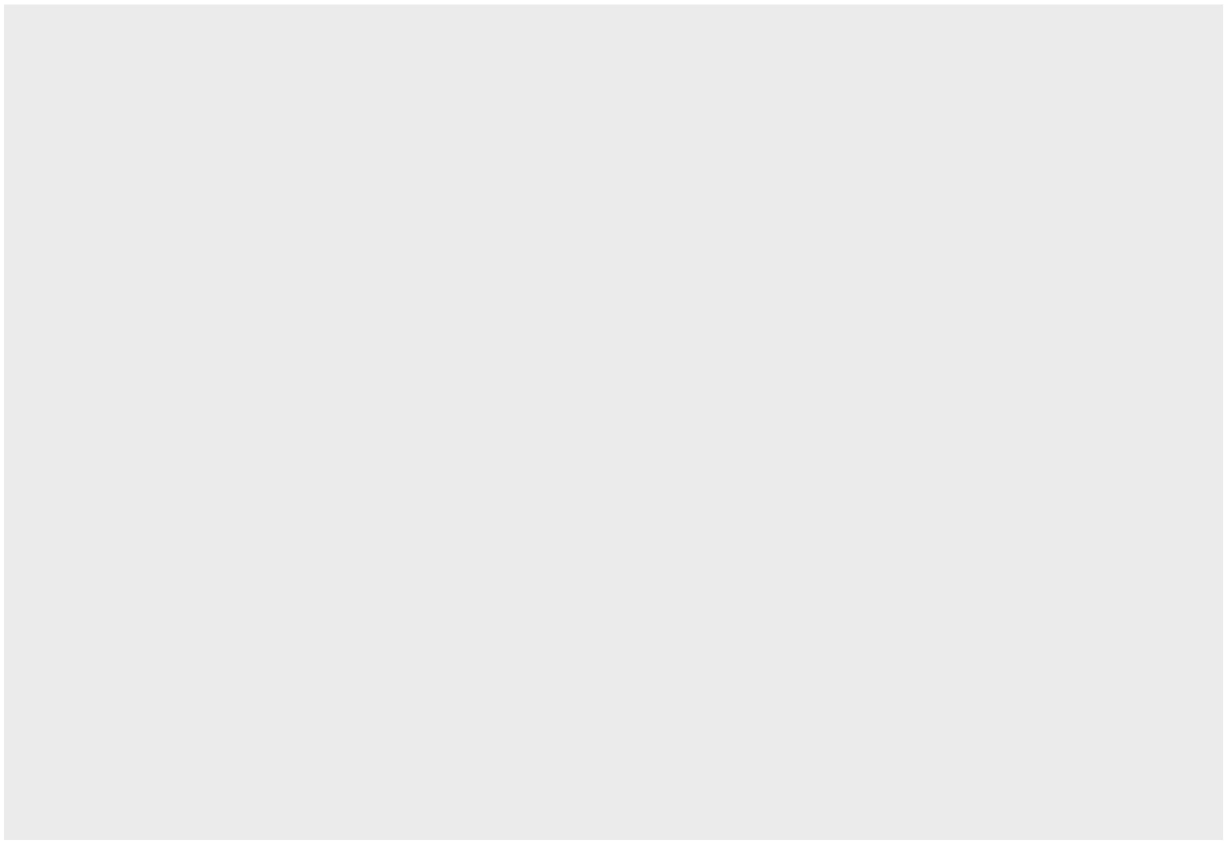
3.3 Scatterplots are great for visualising relationships between two continuous variables. Plot the relationship between DML on the x axis and weight on the y axis. How would you describe this relationship? Is it linear? One approach to linearising relationships is to apply a transformation on one or both variables. Try transforming the weight variable with either a natural log (log()) or square root (sqrt()) transformation. I suggest you create new variables in the squid dataframe for your transformed variables and use these variables when creating your new plots. Which transformation best linearises this relationship?

5. Graphics with ggplot

```r
# install.packages("ggplot2")
library(ggplot2)
library(dplyr)
```

Using the function `ggplot()` without data looks like this

```r
ggplot()
```



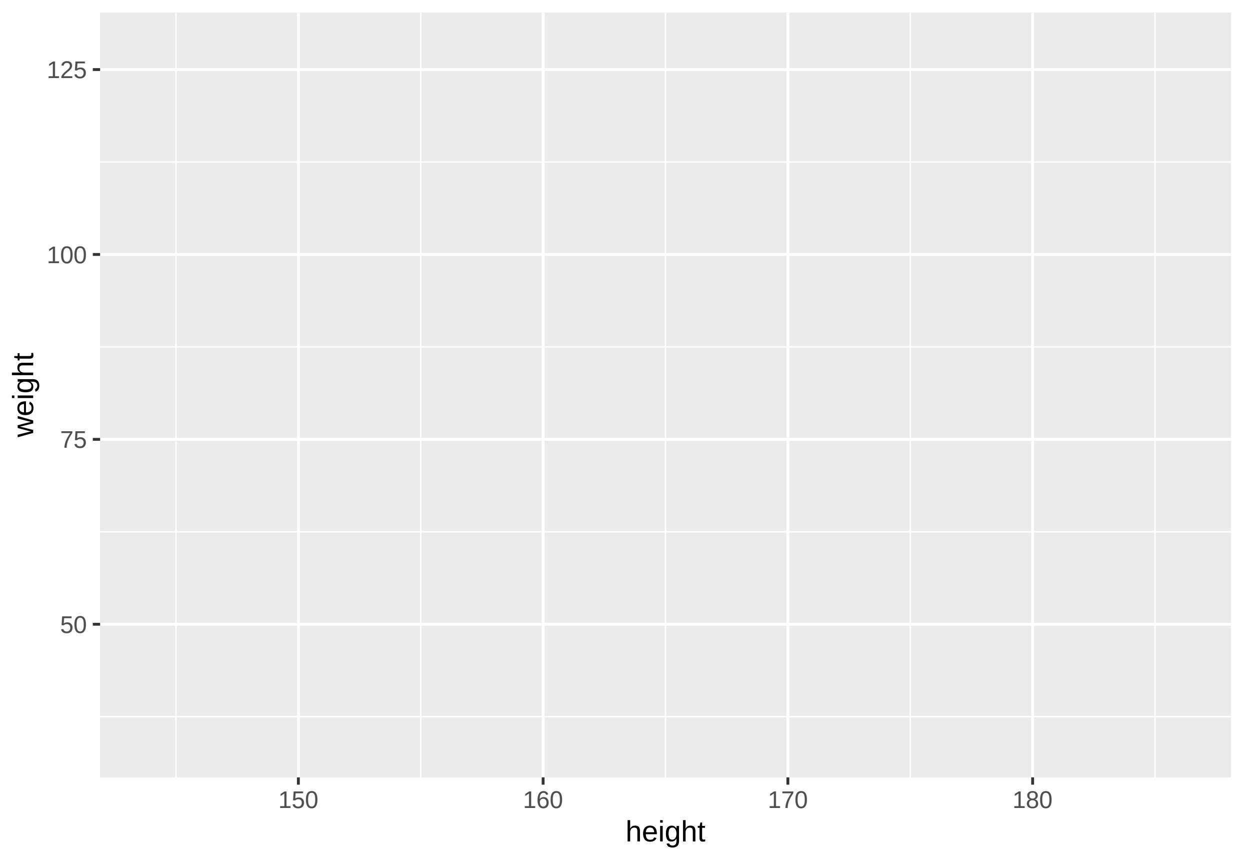
Import our data on liver disease

```r
data6 <- read.csv("Data/data6.csv")

str(data6)
```

```
## 'data.frame':    600 obs. of  10 variables:
##  $ patient_no  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ gender      : int  1 1 1 2 2 1 1 2 1 2 ...
##  $ age         : int  57 23 23 59 54 57 39 46 61 33 ...
##  $ weight      : num  66 98 62 51 72.3 57.2 65 58.4 63 43 ...
##  $ height      : num  166 175 171 157 155 155 160 155 168 153 ...
##  $ peanuts     : int  0 0 0 0 0 1 0 0 0 0 ...
##  $ smoking     : int  1 0 0 0 1 0 0 0 0 0 ...
##  $ alc         : int  1 1 1 1 0 0 1 0 0 1 ...
##  $ dm          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ liverdisease: int  0 0 0 0 0 1 0 0 0 0 ...
```

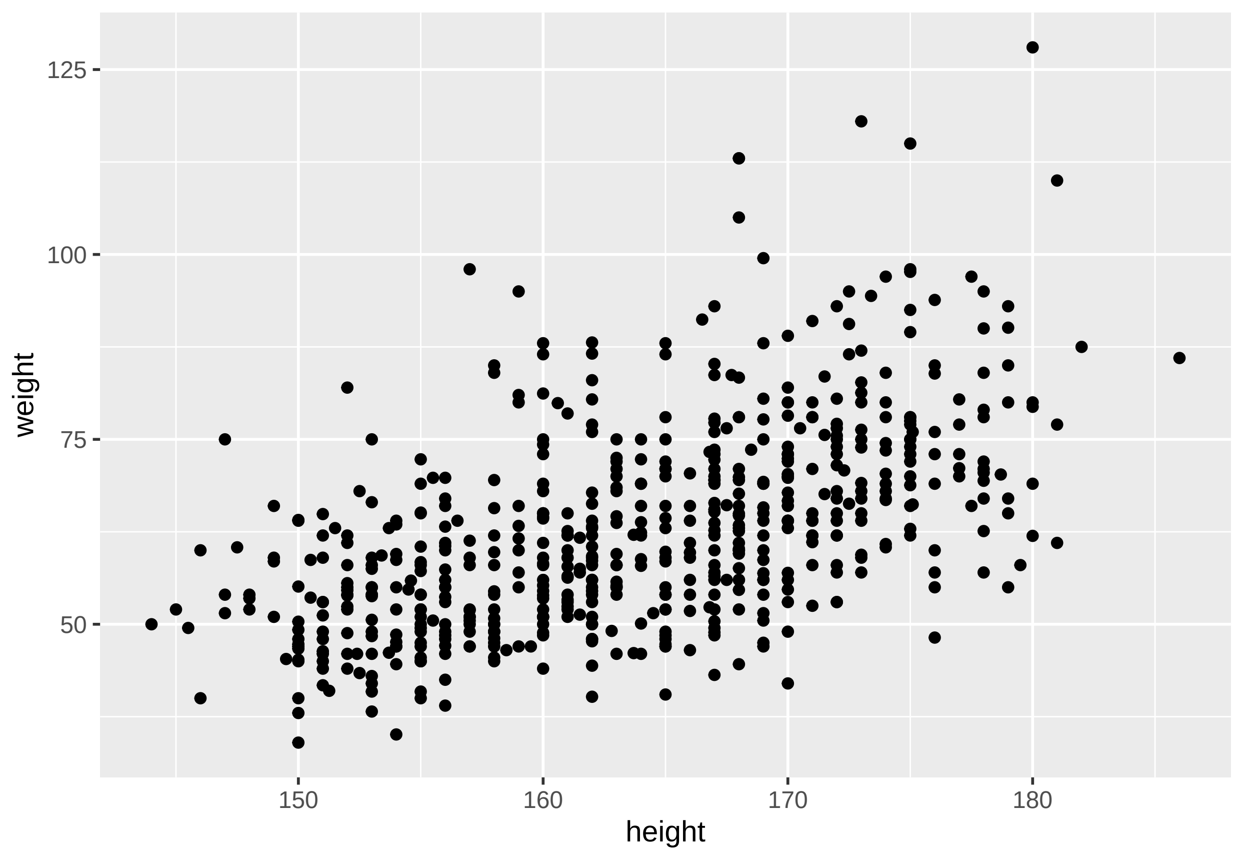
Now lets Include the aesthetics for x and y axes as well as specifying the dataset

```r
ggplot(mapping = aes(x = height, y = weight), data = data6)
```



That's already much better. At least it's no longer a blank grey canvas. We've now told ggplot2 what we want as our x and y axes as well as where to find that data. But what's missing here is where we tell ggplot2 how to display that data. This is now the time to introduce you to 'geoms' or geometry layers.

```r
ggplot(mapping = aes(x = height, y = weight), data = data6) +
  geom_point()
```

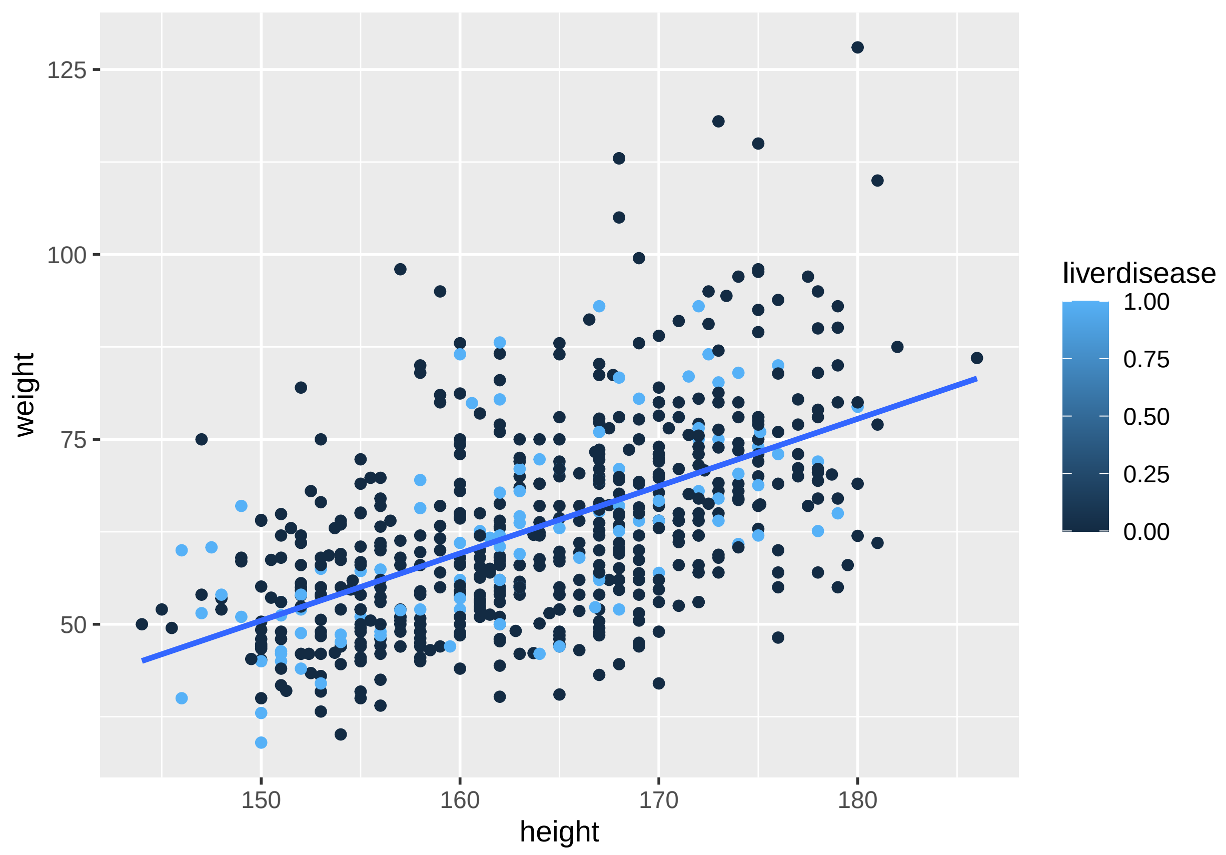


Here you can see that grouping with liver disease factor looks kinda weird because we haven't specify that the liver disease data is a binary data

```r
ggplot(mapping = aes(x = height, y = weight, colour = liverdisease), data = data6) +
  geom_point() +
    geom_smooth(method = "lm", se = F) # se = standard error
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

```
## Warning: The following aesthetics were dropped during statistical transformation:
## colour.
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a `group` aesthetic or to convert a numerical
##   variable into a factor?
```



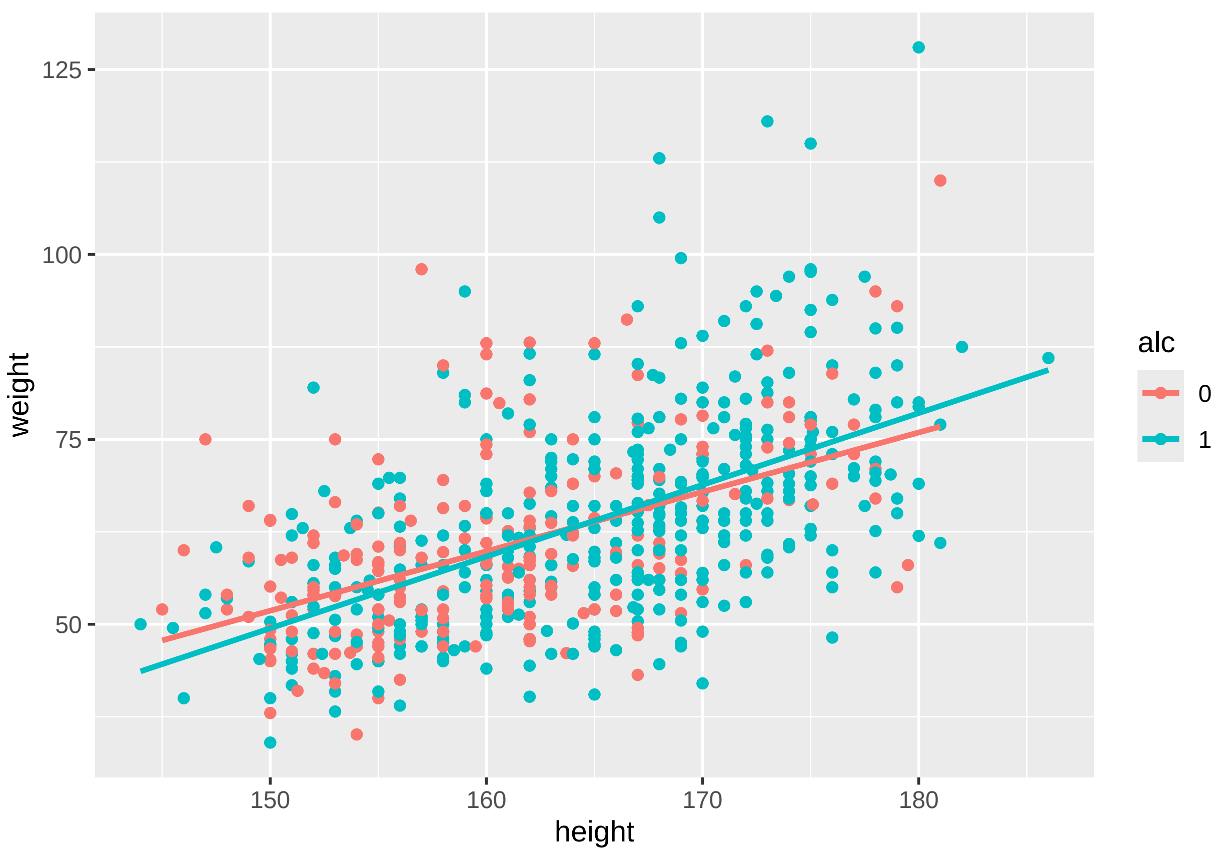
So we tell R the liver disease variable is a character data containing 2 levels, yes and no.

```
data6 <- data6 %>%
  mutate(liverdisease = as.character(liverdisease)) %>%
  mutate(alc = as.character(alc))
```

Let's plot this again

```
ggplot(mapping = aes(x = height, y = weight, colour = alc), data = data6) +
  geom_point() +
    geom_smooth(method = "lm", se = F)
```

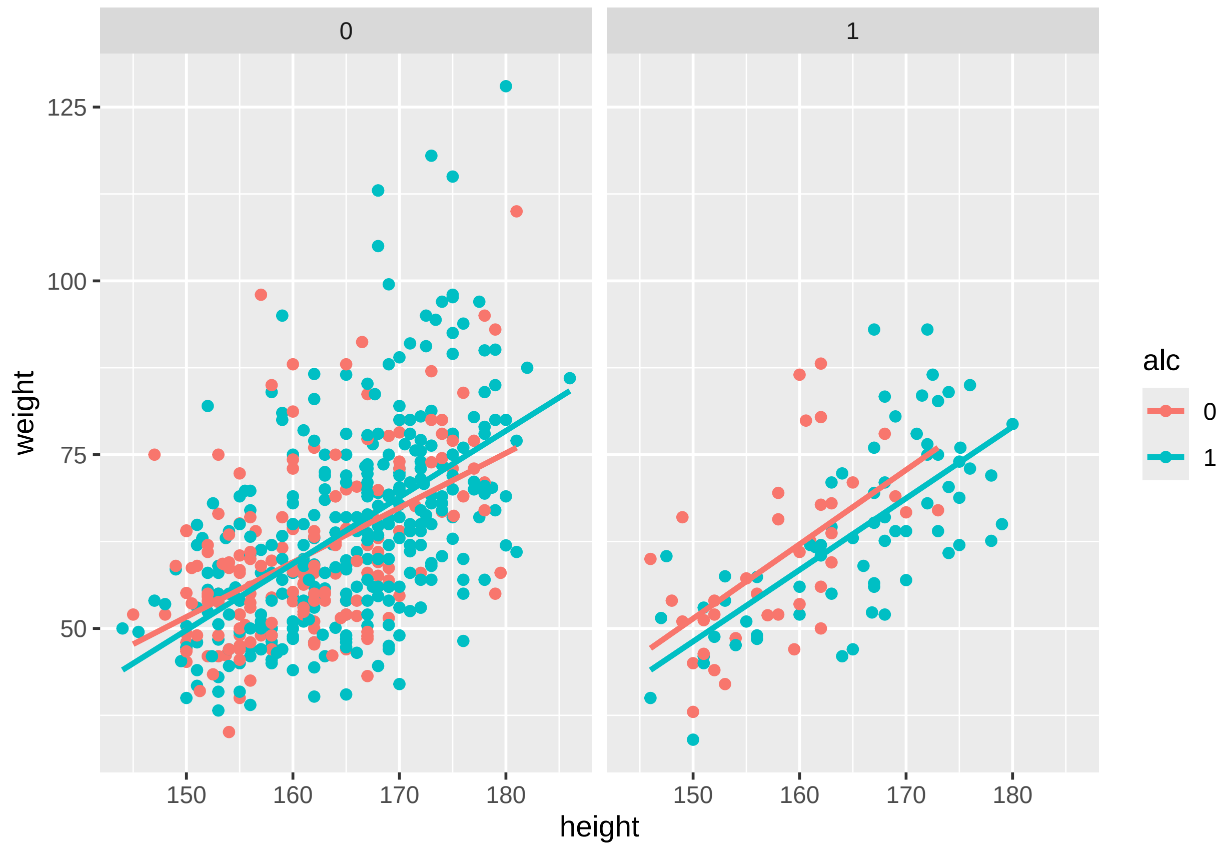
```
## `geom_smooth()` using formula = 'y ~ x'
```



It looks much better right? However, we can still make this easier to visualiz using the facet_wrap() function

```r
ggplot(mapping = aes(x = height, y = weight, colour = alc), data = data6) +
  geom_point() +
  geom_smooth(method = "lm", se = F) +
  facet_wrap(~ liverdisease)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



6. Simple Statistics in R

6.1 One and two sample tests

The main functions for these types of tests are the `t.test()`. this test can be applied to one and two sample analyses as well as paired data.

```
data(trees)

str(trees)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
summary(trees)
```

```
##      Girth           Height       Volume
##  Min.   : 8.30   Min.   :63   Min.   :10.20
##  1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
##  Median :12.90   Median :76   Median :24.20
##  Mean   :13.25   Mean   :76   Mean   :30.17
##  3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
##  Max.   :20.60   Max.   :87   Max.   :77.00
```

If we wanted to test whether the mean height of black cherry trees in this sample is equal to 70 ft (mu = 70), assuming these data are normally distributed, we can use a `t.test()` to do so.

```
t.test(trees$Height, mu = 70)
```

```
##
##  One Sample t-test
##
## data:  trees$Height
## t = 5.2429, df = 30, p-value = 1.173e-05
## alternative hypothesis: true mean is not equal to 70
## 95 percent confidence interval:
##  73.6628 78.3372
## sample estimates:
## mean of x
##        76
```

The function `t.test()` also has a number of additional arguments which can be used for one-sample tests. You can specify that a one sided test is required by using either alternative = "greater" or alternative = "less arguments which tests whether the sample mean is greater or less than the mean specified. For example, to test whether our sample mean is greater than 70 ft.

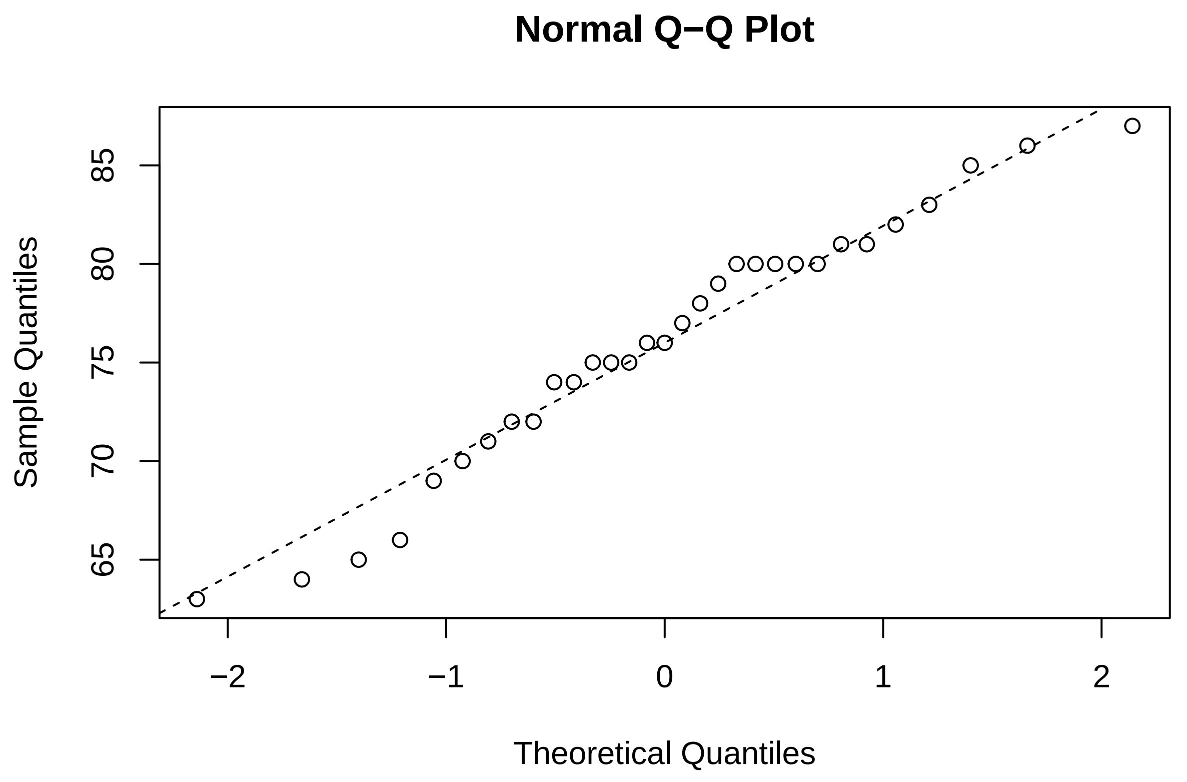
```
t.test(trees$Height, mu = 70, alternative = "greater")
```

```
##
##  One Sample t-test
##
## data:  trees$Height
## t = 5.2429, df = 30, p-value = 5.866e-06
## alternative hypothesis: true mean is greater than 70
## 95 percent confidence interval:
##  74.05764      Inf
## sample estimates:
## mean of x
##        76
```

In our one sample test it's always a good idea to examine your data for departures from normality, rather than just assuming everything is OK. Perhaps the simplest way to assess normality is the 'quantile-quantile plot'.

To construct a Q-Q plot you need to use both the `qqnorm()` and `qqline()` functions. The lty = 2 argument changes the line to a dashed line.

```
qqnorm(trees$Height)
qqline(trees$Height, lty = 2)
```



If you insist on performing a specific test for normality you can use the function `shapiro.test()` which performs a Shapiro – Wilks test of normality.

```
shapiro.test(trees$Height)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  trees$Height
## W = 0.96545, p-value = 0.4034
```

In the example above, the p value = 0.4 which suggests that there is no evidence to reject the null hypothesis and we can therefore assume these data are normally distributed.

6.2 Correlation

In R, the Pearson's product-moment correlation coefficient between two continuous variables can be estimated using the `cor()` function. Using the trees data set again, we can determine the correlation coefficient of the association between tree Height and Volume.

```
data(trees)
str(trees)
```

```
## 'data.frame':    31 obs. of  3 variables:
```

```
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
cor(trees$Height, trees$Volume)
```

```
## [1] 0.5982497
```

or we can produce a matrix of correlation coefficients for all variables in a data frame

```
cor(trees)
```

```
##            Girth    Height    Volume
## Girth  1.0000000 0.5192801 0.9671194
## Height 0.5192801 1.0000000 0.5982497
## Volume 0.9671194 0.5982497 1.0000000
```

The function `cor()` will return the correlation coefficient of two variables, but gives no indication whether the coefficient is significantly different from zero. To do this you need to use the function cor.test().

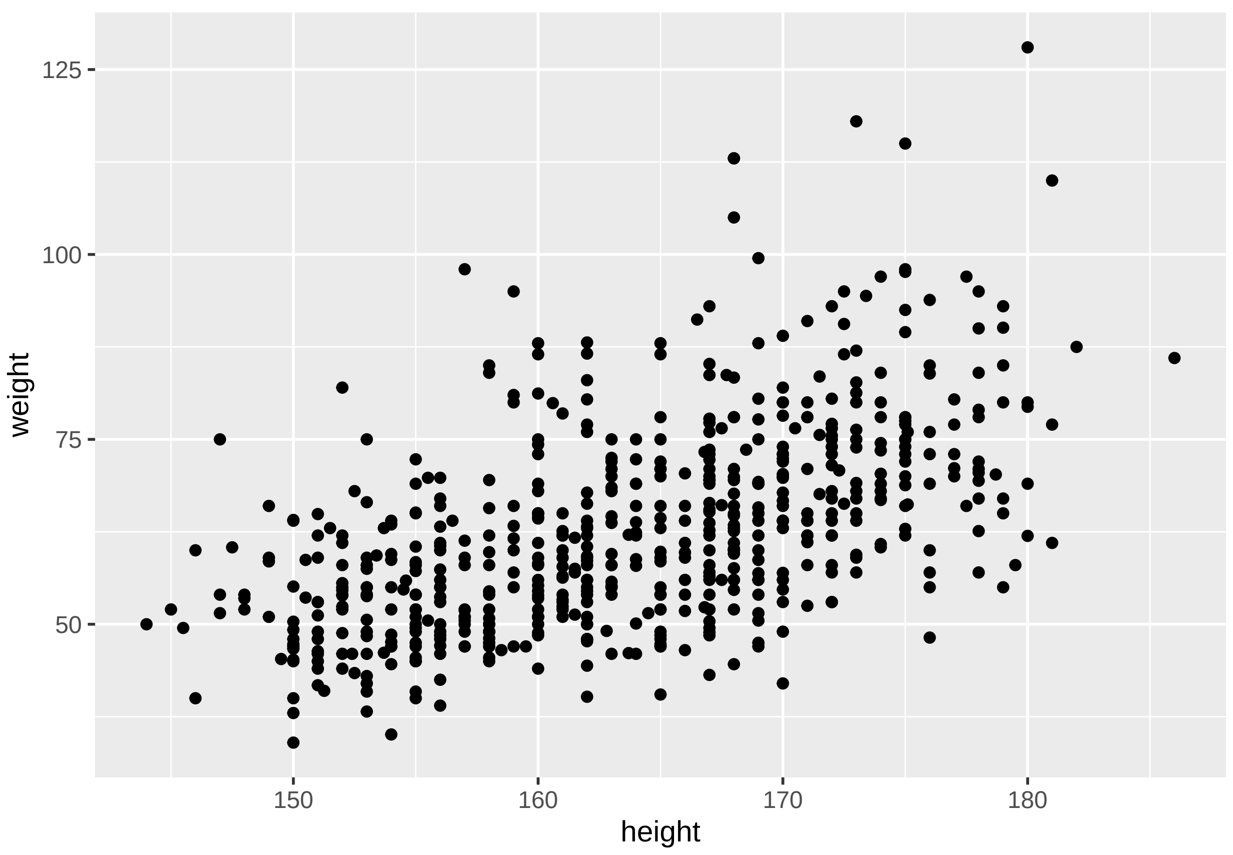
```
cor.test(trees$Height, trees$Volume)
```

```
##
##  Pearson's product-moment correlation
##
## data:  trees$Height and trees$Volume
## t = 4.0205, df = 29, p-value = 0.0003784
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.3095235 0.7859756
## sample estimates:
##       cor
## 0.5982497
```

6.3 Simple linear modelling

Next, let's investigate the relationship between the height and weight variables by plotting a scatter plot.

```
ggplot(mapping = aes(x = height, y = weight), data = data6) +
  geom_point()
```



The plot does suggest that there is a positive relationship between the smoking index and mortality index.

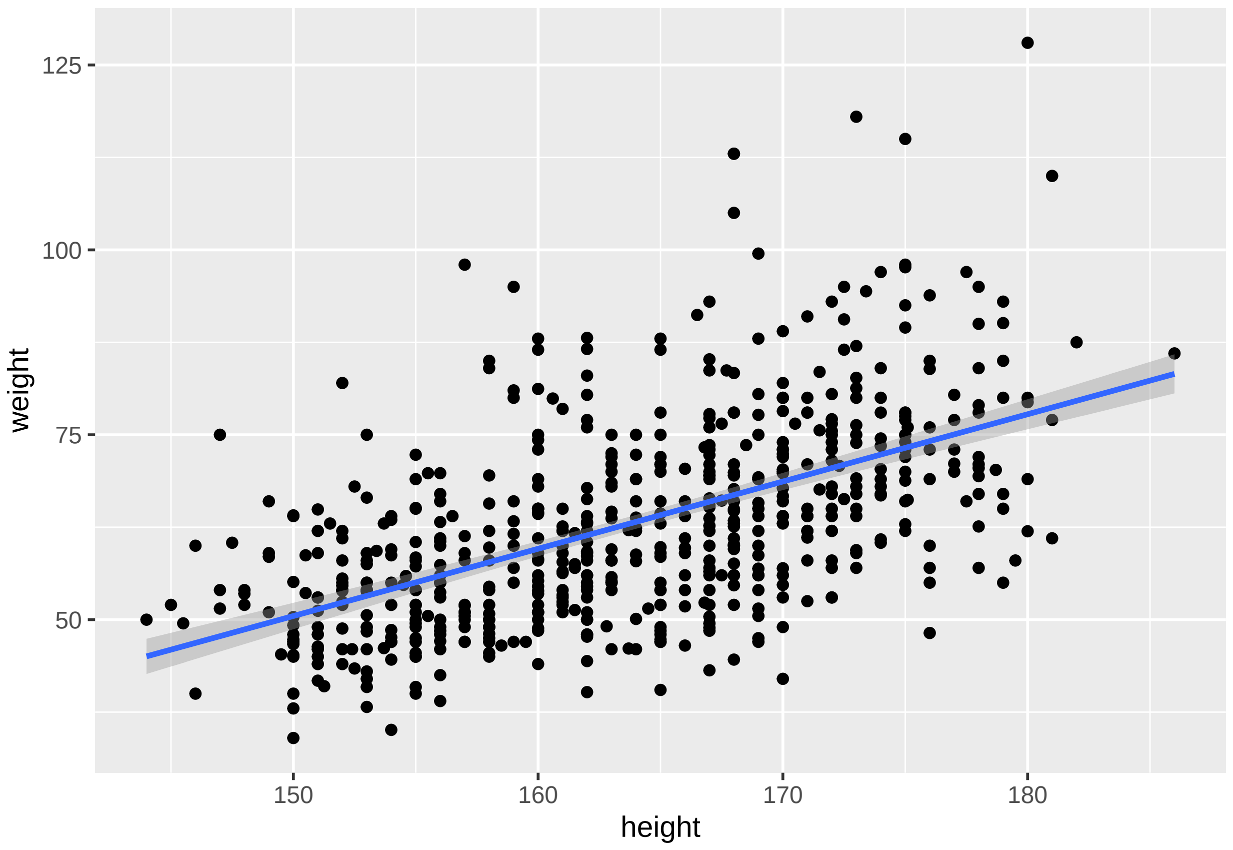
To fit a simple linear model to these data we will use the `lm()` function and include our model formula weight ~ height and assign the results to an object called lm_df.

```r
lm_df <- lm(weight ~ height, data = data6)

summary(lm_df)
```

```
##
## Call:
## lm(formula = weight ~ height, data = data6)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -26.680  -7.619  -1.441   5.696  50.226
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -85.91950    9.25411  -9.284   <2e-16 ***
## height        0.90941    0.05644  16.113   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.64 on 598 degrees of freedom
## Multiple R-squared:  0.3027, Adjusted R-squared:  0.3016
## F-statistic: 259.6 on 1 and 598 DF,  p-value: < 2.2e-16
```

```
ggplot(mapping = aes(x = height, y = weight), data = data6) +
  geom_point() +
    geom_smooth(method = "lm", se = TRUE)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Exercise 4

4.1 Import the ‘prawn’ data into R and assign to a variable with a name. These data were collected from an experiment to investigate the difference in growth rate of the giant tiger prawn, *Penaeus monodon*, fed either an artificial or natural diet. Have a quick look at the structure of this dataset and plot the growth rate versus the diet using an appropriate plot. How many observations are there in each diet treatment?

4.2 You want to compare the difference in growth rate between the two diets using a two sample t-test. Before you conduct the test, you need to assess the normality and equal variance assumptions. Use the function `shapiro.test()` to assess normality of growth rate for each diet separately (Hint: use your indexing skills to extract the growth rate for each diet GRate[diet==‘Natural’] first) and plot a qq plot to assess the normality. Are your data normally distributed?.

4.3 Conduct a two sample t-test using the `t.test()` function. Use the argument var.equal = TRUE to perform the t-test assuming equal variances. What is the null hypothesis you want to test? Do you reject or fail to reject the null hypothesis? What is the value of the t statistic, degrees of freedom and p value? How would you summarise these summary statistics in a report?

4.4 Import the TemoraBR.csv dataset into R and as usual assign it a name. These data are from an experiment that was conducted to investigate the relationship between temperature (temp) and the beat rate (Hz) beat_rate of the copepod *Temora longicornis* which had been acclimatised at three different temperature regimes (acclimitisation_temp). Examine the structure of the dataset. How many variables are there? and what type of variables are they?

4.5 Now we want to explore the correlation between the temperature and beat rate using `cor()`function. What relationship do you see from these two variables? Can you visualize these two variables with `ggplot()` function?

4.6 Now, let's fit a simple linear model to these data we will use the `lm()` function and include our model formula beat_rate ~ temp and assign the results to an object called lm_temora. Can you predict how the beat rate will change when there is an increase in one unit of temperature?