PHCM9795 Foundations of Biostatistics

Pilot Notes for R

16 July, 2022

Contents

Introduction

These notes provide an introduction to R and instructions on how to conduct the analyses introduced in Foundations of Biostatistics.
These notes are currently under development, with sections being added and revised as the course progresses.
This is the first year that R has been offered as an option. I am keen to receive feedback about the notes and your experience learning R. Please get in touch if anything is unclear, or you have any questions or suggestions.

Changelog

2022-07-16
[Added]

- Module 8: first release.

2022-07-14
[Added]

- Module 7: first release.

2022-07-05
[Added]

- Module 6: first release.

2022-06-20
[Added]

- Module 5: first release.

2022-06-14
[Changed]

- Section 2.12 - corrected the `pnorm(q, mean, sd, lower.tail=FALSE)` documentation to state that the it is the probablity of obtaining **more than** q that is calculated.
- Section 3.1 - recommendation to use `t.test()` to calculate a 95% confidence interval for a mean, and not the `descriptives()` function as `descriptives()` uses a z-value instead of a t-value.

2022-06-10
[Added]

- Section 2.10 - Added instructions on labelling groups using the `cut()` function

2022-06-09
[Added]

- Section 2.8 - Summarising a single column of data using the `descriptives()` function from `jmv` package.

2022-06-07
[Changed]

- Section 2.6: Use the `<-` operator instead of `=`

2022-06-05
[Changed]

- Module 1: Typos

2022-05-30
[Changed]

- Module 1: Typo in R Preferences (Section 1.3.1)

[Added]

- Section 1.12: Instructions to plot a histogram with relative frequencies (i.e. percents) instead of frequencies

2022-05-27
[Changed]

- Module 1: Fixed bar-charts that were not plotted correctly

2022-05-27
[Added]

- Section 1.2.1: Added a note about using the "patched" version of R 4.2.0 for Windows
- Section 1.14: Instructions for creating two-way tables using the `contTables()` function in the `jmv` package

2022-05-23
[Added]

- Section 1.9: Explicit instructions to install `jmv` and `summarytools` when working in Module 1

[Changed]

- Section 1.9: Changed location of `pbc.dat` from `examples` to `activities` folder for consistency

2022-05-19
Initial release

Module 1

Introduction to R and RStudio

Learning outcomes

By the end of this Module, you will be able to:

- understand the difference between R and RStudio
- navigate the RStudio interface
- input and import data into R
- use R to summarise data
- perform basic data transformations
- understand the difference between saving R data and saving R output
- copy R output to a standard word processing package

Part 1: An introduction to R

"R is a language and environment for statistical computing and graphics." Link. It is an open-source programming language, used mainly for statistics (including biostatistics) and data science.
The aim of these notes is to introduce the R language within the RStudio environment, and to introduce the commands and procedures that are directly relevant to this course. There is so much more to R than we can cover in these notes. Relevant information will be provided throughout the course, and we will provide further references that you can explore if you are interested.

1.1 R vs RStudio

At its heart, R is a programming language. When you install R on your computer, you are installing the language and its resources, as well as a very basic interface for using R. You can write and run R code using the basic R app, but it's not recommended.
RStudio is an "Integrated Development Environment" that runs R while also providing useful tools to help you write code and analyse data. You can think of R as an engine which does the work, and RStudio as a car that uses the engine, but also provides useful tools like GPS navigation and reversing cameras that help you drive.
Note: even though we recommend that you use RStudio, you still need install R. **RStudio will not run without R installed.**

R: Don't run this	RStudio: Run this instead

1.2 Installing R and RStudio

1.2.1 To install R on your computer

1. Download the R installer from:

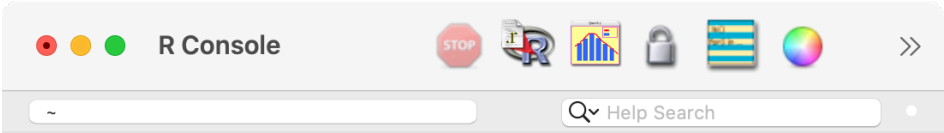
    a. for Windows: https://cran.r-project.org/bin/windows/base/
    b. for MacOS: https://cran.r-project.org/bin/macosx/

    - **Note for Windows users:** as at May 27, 2022, R Version 4.2.0 has compatability issues with RStudio. You should download and install R from https://cran.r-project.org/bin/windows/base/rpatched.html

2. Install R by running the installer and following the installation instructions. The default settings are fine.

    - **Note for macOS:** if you are running macOS 10.8 or later, you will need to install an additional application called XQuartz, which is available at https://www.xquartz.org/. Download the latest installer (XQuartz-2.8.1.dmg as of April 2022), and install it in the usual way.

3. Open the R program. You should see a screen as below:



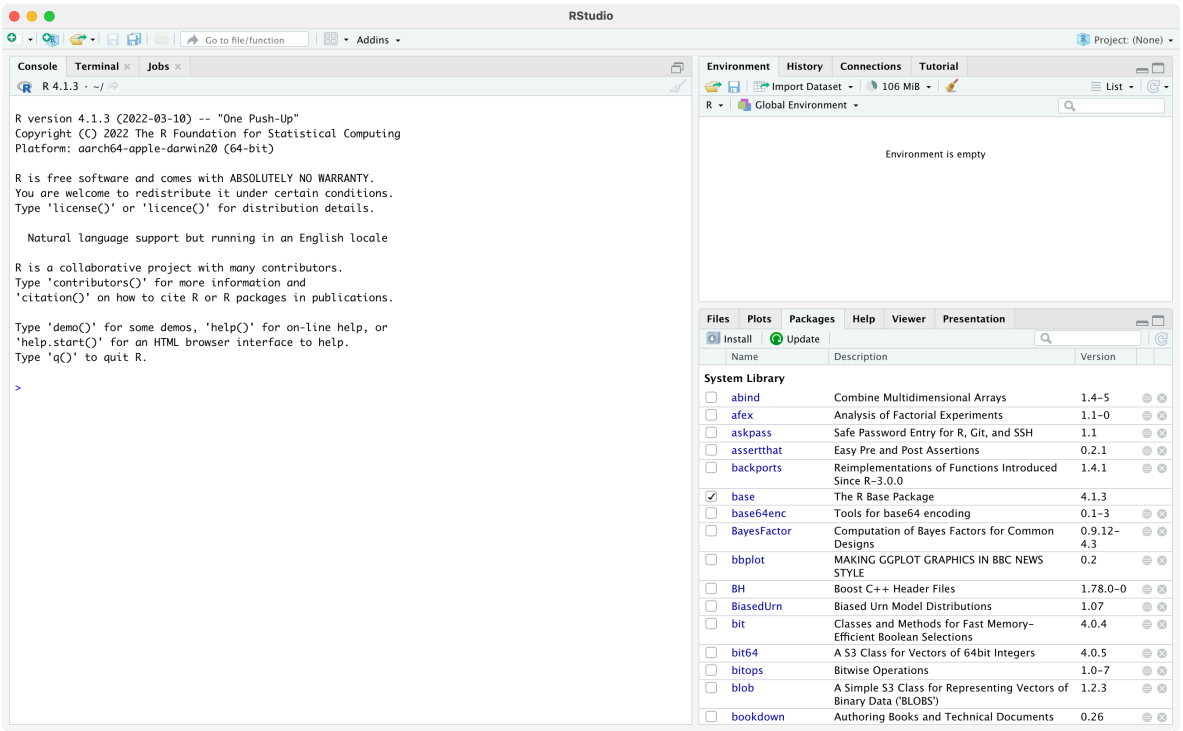

```
R version 4.1.3 (2022-03-10) -- "One Push-Up"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.77 (8052) aarch64-apple-darwin20]

> |
```

Near the bottom of the R screen, you will find the “>” symbol which represents the command line. If you type `1 + 2` into the command line and then hit enter you should get:
`[1] 3`
This is R performing your calculation, with the `[1]` indicating that the solution to `1 + 2` is a single number (the number 3).

At this point, close R - we will not interact with R like this in the future. You can close R by typing `quit()` at the command prompt, followed by the return key, or in the usual way of closing an application in your operating system. There is no need to save anything here if prompted.

1.2.2 To install RStudio on your computer

1. Make sure you have already installed R, and verified that it is working.
2. Download the RStudio desktop installer at: https://www.rstudio.com/products/rstudio/download. Ensure that you select the RStudio Desktop (Free) installer in the first column.
3. Install RStudio by running the installer and following the installation instructions. The default settings are fine.
4. Open RStudio, which will appear as below:



Locate the command line symbol ">" at the bottom of the left-hand panel. Type `1 + 2` into the command line and hit enter, and you will see:

```
[1] 3
```

This confirms that RStudio is running correctly, and can use the R language to correctly calculate the sum between 1 and 2!

RStudio currently comprises three window panes, and we will discuss these later.

1.3 Recommended setup

I will provide a recommended setup for R and RStudio in this section. You are free to use alternative workflows and setup, but this setup works well in practice.

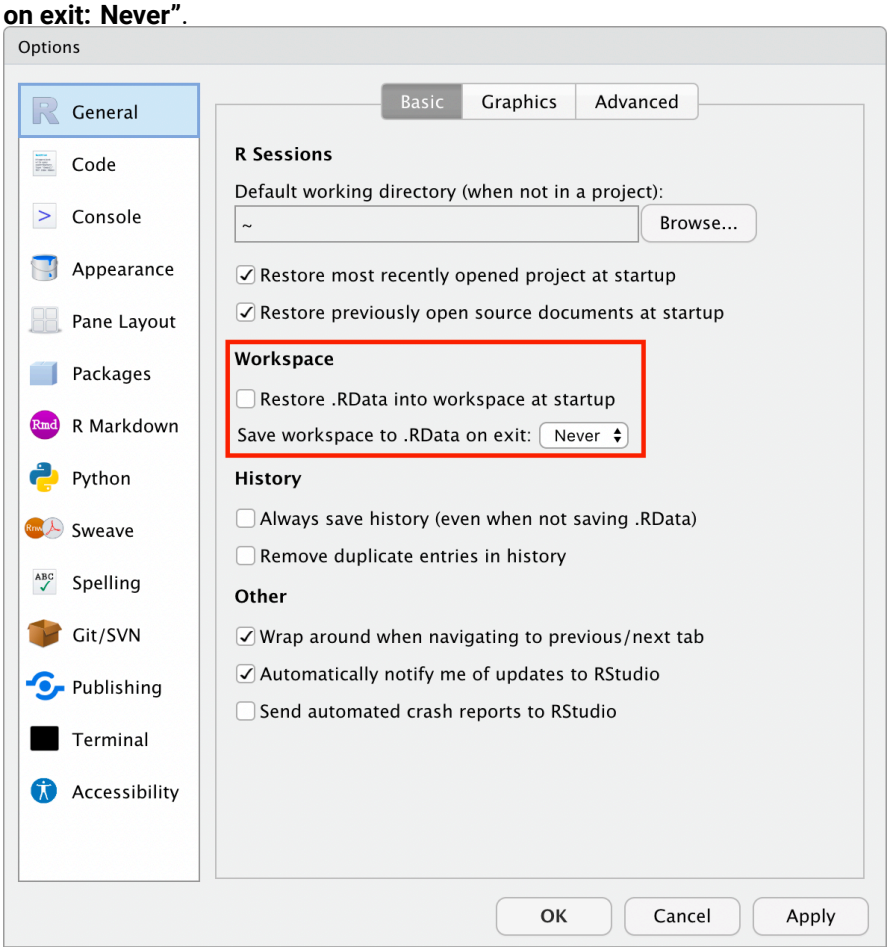
1.3.1 RStudio preferences

By default, RStudio will retain data, scripts and other objects when you quit your RStudio session. Relying on this can cause headaches, so I recommend that you set up RStudio so that it does not preserve your workspace between sessions. Open the RStudio options:

- Mac: **RStudio > Preferences**

- Windows: **Tools > Options**

and **deselect "Restore .RData into workplace at startup"**, and choose: "**Save workspace to .RData on exit: Never**".

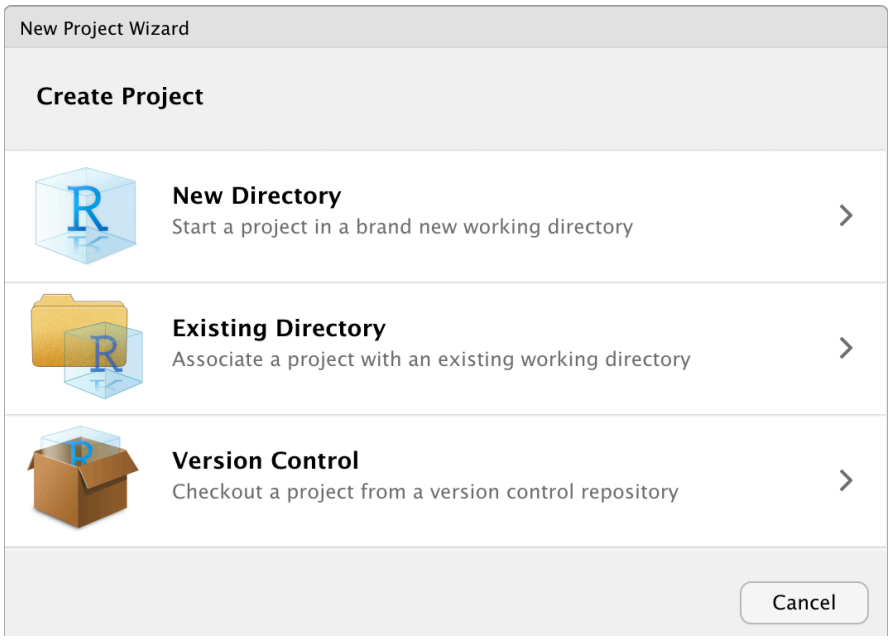


1.3.2 Set up a project

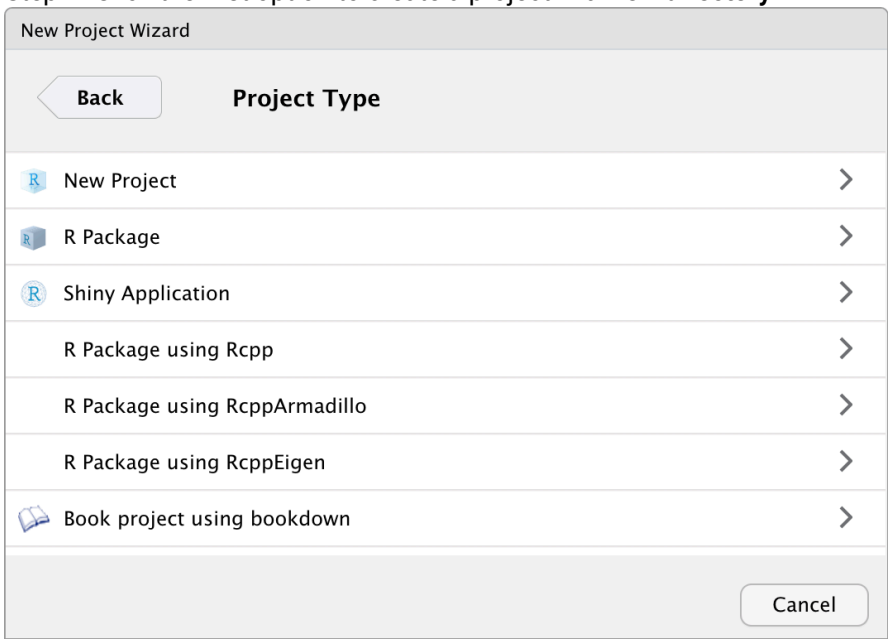
A project in RStudio is a folder that RStudio recognises as a place to store R scripts, data files, figures that are common to an analysis project. Setting up a folder allows much more simple navigation and specification of data files and output. More detail can be found in Chapter 8 of the excellent text: R for Data Science. Using projects is not necessary, but I recommend working with projects from day one.

We will create a project called **PHCM9795** to store all the data you will use and scripts that you will write in this course. First, think about where you want to store your project folder: this could be somewhere in your *Documents* folder.

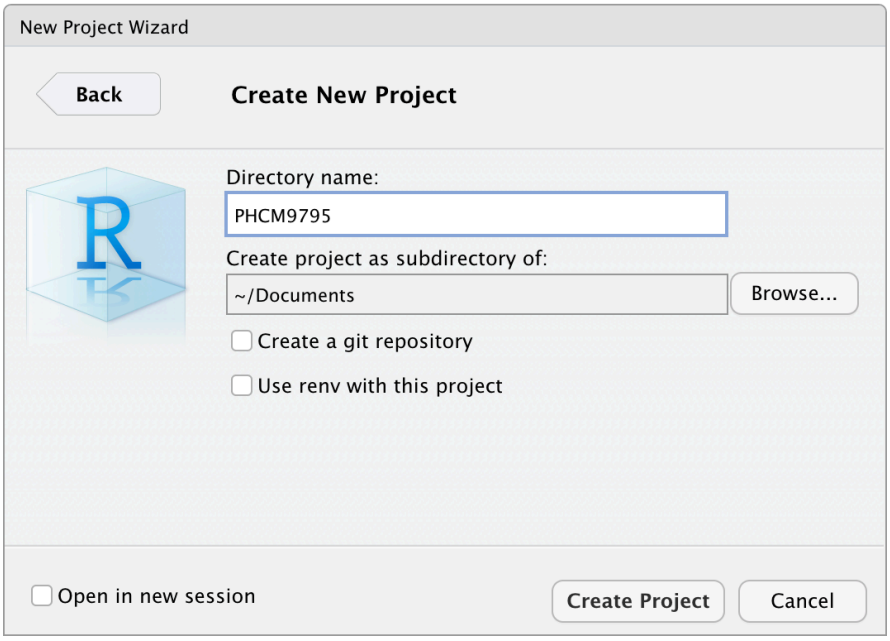
Step 1: Choose **File > New Project...** in RStudio to open the **Create Project** dialog box:



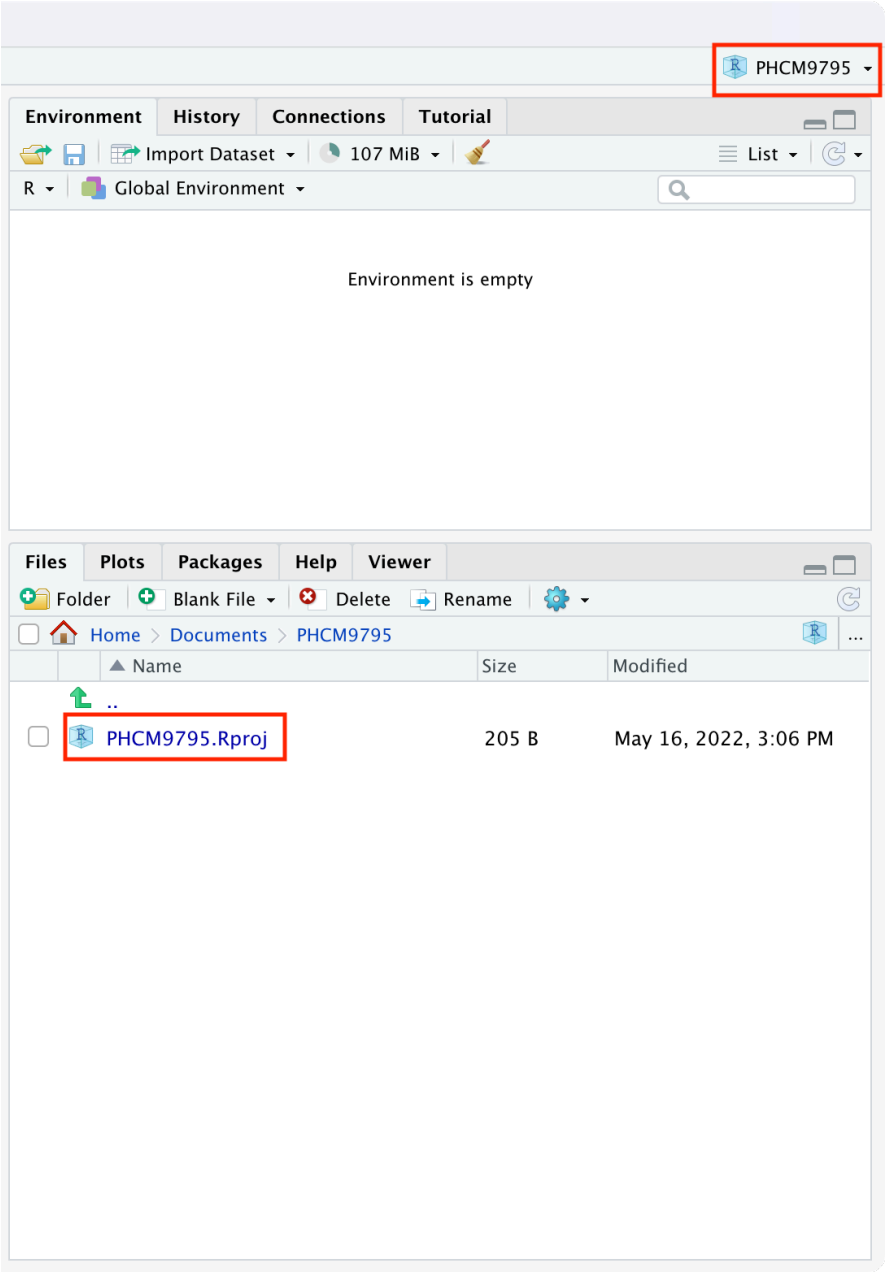
Step 2: Click the first option to create a project in a **New directory**



Step 3: Click the first option: **New Project**. Give the project a name, by typing PHCM9795 in the "Directory name", and choose where you want to store the project by clicking the **Browse** button.



Step 4: Click **Create** to create your project.
You will now have a new folder in your directory, which contains only one file: PHCM9795.Rproj, and the two right-hand panes of RStudio will appear as below:



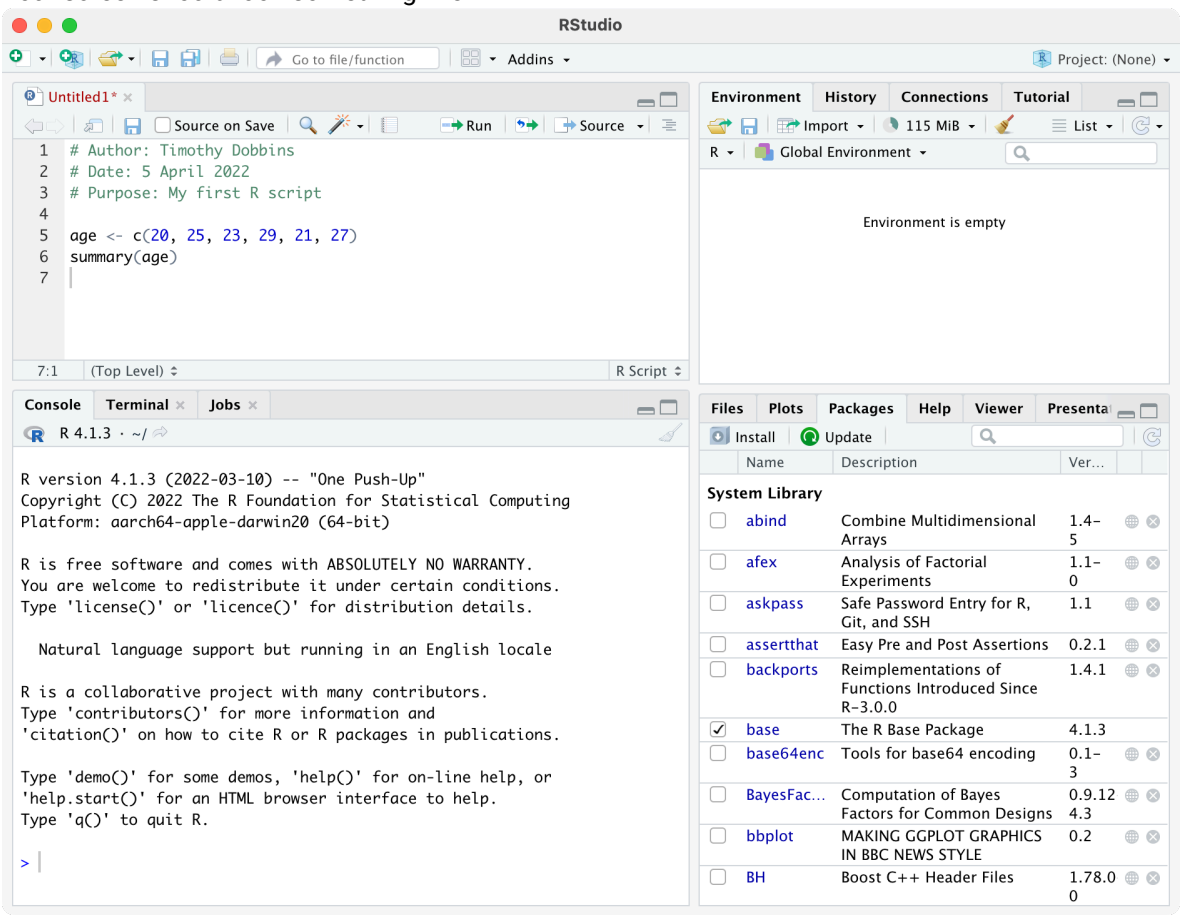
The top-right menu bar is showing that you are working within the PHCM9795 project, and the bottom-right window is showing the contents of that window: the single PHCM9795.Rproj file. We will add some more files to this project later.

1.4 A simple R analysis

In this very brief section, we will introduce R by calculating the average of six ages.
To begin, open a new R Script by choosing **File > New file > R Script** . A script (or a program) is a collection of commands that are sequentially processed by R. You can also type Ctrl+Shift+N in Windows, or Command+Shift+N in MacOS to open a new script in RStudio, or click the **New File** button at the top of the RStudio window.
You should now see four window panes, as below. In the top-left window, type the following (replacing my name with yours, and including today's date):

```
# Author: Timothy Dobbins
# Date: 5 April 2022
# Purpose: My first R script

age <- c(20, 25, 23, 29, 21, 27)
summary(age)
```

Note: R is case-sensitive, so you should enter the text exactly as written in these notes.
Your screen should look something like:



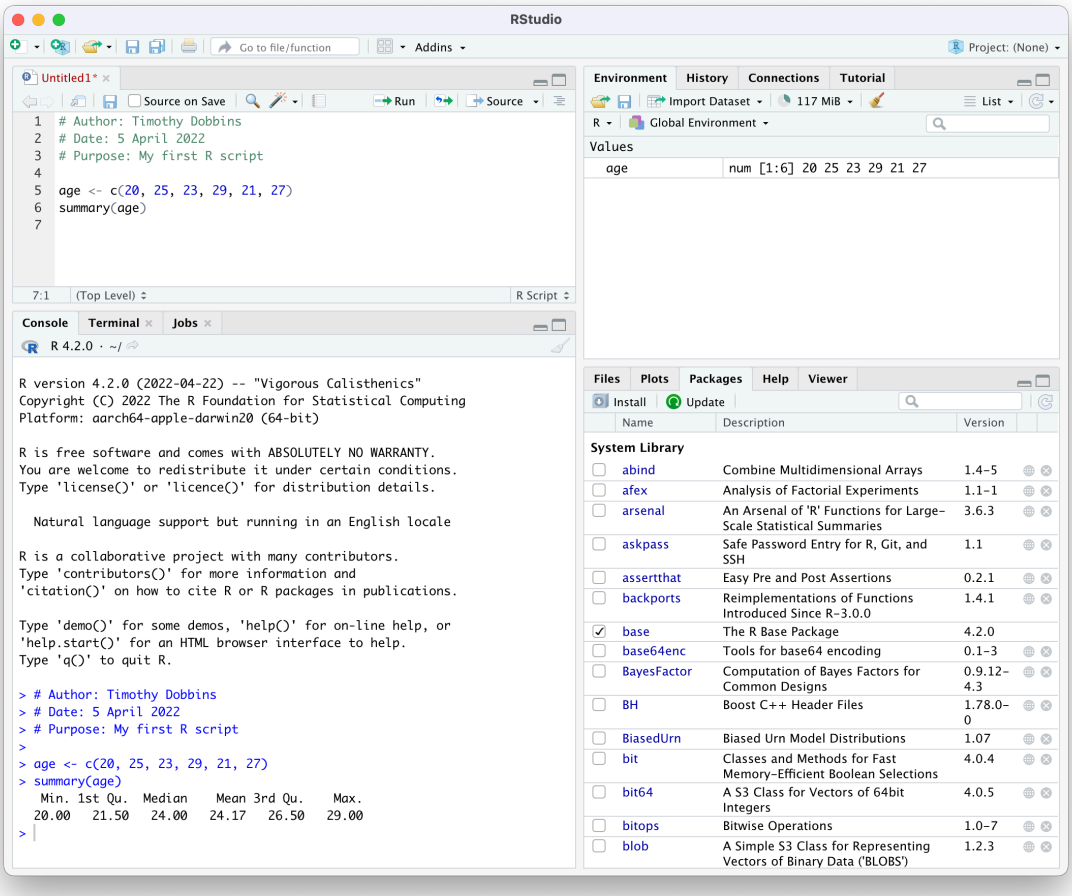
To run your script, choose **Code > Run Region > Run All**. You will see your code appear in the bottom-left window, with the following output:

```
> # Author: Timothy Dobbins
> # Date: 5 April 2022
> # Purpose: My first R script
>
> age <- c(20, 25, 23, 29, 21, 27)

> summary(age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  20.00   21.50   24.00   24.17   26.50   29.00
```

We will explain the key parts of this script later, but for now, you have entered six ages and calculated the mean age (along with five other summary statistics).
Save your script within the PHCM9795 project by using **File > Save As**, using the name `my_first_analysis.R`.

1.5 The RStudio environment

Now that we have seen a simple example of how to use R within RStudio, let's describe the RStudio environment. Let's assume that you have just run your first R script, and you have four windows as below:



The top-left window is call the **Source** window, and is where you write and edit your R scripts. Scripts can be saved by clicking **File > Save As** or by clicking on the symbol of a floppy disk at the top of the script. The file will have an extension of .R, for example script.R. Remember to give your script a meaningful title and remember to periodically save as you go.
In RStudio, the name of the script will be black when it has been saved, and will change to red if you have any unsaved changes.
The **Console** window, at the bottom left, contains the command line which is indicated with the symbol >. You can type commands here, but anything executed directly from the console is not saved and therefore is lost when the session ends (when you exit RStudio). You should always run your commands from a script file which you can save and use again later. When you run commands from a script, the output and any notes/errors are shown in the console. The Terminal and Jobs tabs will not be used in this course.
The **Environment** window at the top-right shows a list of objects that have been created during your session. When you close your RStudio session these objects will disappear. We will not use the History or Connections tabs in this course.
The bottom right corner contains some useful tabs, in particular the **Help** tab. When you are troubleshooting errors or learning how to use a function, the Help tab should be the first place you visit. Here you can search the help documents for all the packages you have installed. Whenever you create plots in R, these will be shown in the **Plots** tab. The **Packages** tab contains a list of installed

packages and indicates which ones are currently in use (we will learn about packages later). Packages which are loaded, i.e. in use, are indicated with a tick. Some packages are in use by default when you begin a new session. You can access information about a package by clicking on its name. The **Files** tab provides a shortcut to access your files. The Viewer tab will not be used in this course.

1.6 Some R basics

While we use R as a statistics package, R is a programming language. In order to use R effectively, we need to define some basics.

1.6.1 Scripts

While R can be run completely from the command line, issuing commands one-by-one, it is most commonly run using **scripts**. A script is simply a list of commands that are processed in order. The simple analysis we conducted earlier is a very simple script. Some things to know about R scripts:

- anything appearing after a # is a comment, and is ignored by R. The first three lines of our script are there for ourselves (either as writers of code, or readers of code). I include comments at the beginning of each of my scripts to describe:
  - who wrote the script (useful if someone else uses your script and wants to ask questions about it);
  - when the script was written;
  - what the script does. This last point may seem odd, but it's useful to describe what this script does, and why it might differ to other scripts being used in the analysis. This is particularly useful if your scripts become long and complex.
- **R is case-sensitive**. So `age`, `AGE` and `Age` could refer to three separate variables (please don't do this!)
- use blank lines and comments to separate sections of your script

1.6.2 Objects

If you do some reading about R, you may learn that R is an "object-oriented programming language". When we enter or import data into R, we are asking R to create **objects** from our data. These objects can be manipulated and transformed by **functions**, to obtain useful insights from our data.
Objects in R are created using the **assignment operator**. The most common form of the assignment operator looks like an arrow: `<-` and is typed as the `<` and `-` symbols. The simplest way of reading `<-` is as the words "is defined as". Note that it possible to use `->` and even `=` as assignment operators, but their use is less frequent.
Let's see an example:

```
x <- 42
```

This command creates a new object called `x`, which is defined as the number 42 (or in words, "`x` is defined as 42"). Running this command gives no output in the console, but the new object appears in the top-right **Environment** panel. We can view the object in the console by typing its name:

```
# Print the object x
x
```

```
## [1] 42
```

Now we see the contents of `x` in the console.
This example is rather trivial, and we rarely assign objects of just one value. In fact, we created an object earlier, called `age`, which comprised six values.

1.6.3 Data structures

There are two main structures we will use to work with data in this course: **vectors** and **data frames**. A **vector** is a combination of data values, all of the same type. For example, our six ages that we entered earlier is a vector. You could think of a vector as a column of data (even though R prints vectors as rows!) And technically, even an object with only one value is a vector, a vector of size 1. The easiest way of creating a vector in R is by using the `c()` function, where c stands for 'combine'. In our previous Simple Analysis in R (Section 1.4), we wrote the command:

```
age <- c(20, 25, 23, 29, 21, 27)
```

This command created a new object called `age`, and *combined* the six values of age into one vector. Just as having a vector of size 1 is unusual, having just one column of data to analyse is also pretty unusual. The other structure we will describe here is a **data frame** which is essentially a collection of vectors, each of the same size. You could think of a data frame as being like a spreadsheet, with columns representing variables, and rows representing observations.
There are other structures in R, such as matrices and lists, which we won't discuss in this course. And you may come across the term **tibble**, which is a type of data frame.

1.6.4 Functions

If objects are the nouns of R, functions are the verbs. Essentially, functions transform objects. Functions can transform your data into summary statistics, graphical summaries or analysis results. For example, we used the `summary()` function to display summary statistics for our six ages.
R functions are specified by their arguments (or inputs). The arguments that can be supplied for each function can be inspected by examining the help notes for that function. To obtain help for a function, we can submit `help(summary)` (or equivalently `?summary`) in the console, or we can use the **Help** tab in the bottom-right window of RStudio. For example, the first part of the help notes for `summary` appear as:

summary {base} R Documentation

Object Summaries

Description

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular `methods` which depend on the `class` of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits, quantile.type = 7)
## S3 method for class 'data.frame'
summary(object, maxsum = 7,
        digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor'
summary(object, maxsum = 100, ...)
```

The help notes in R can be quite cryptic, but the **Usage** section details what inputs should be specified for the function to run. Here, `summary` requires an object to be specified. In our case, we specified `age`, which is our object defined as the vector of six ages.
Most help pages also include some examples of how you might use the function. These can be found at the very bottom of the help page.

Examples

Run examples

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)

lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

The `summary()` function is quite simple, in that it only requires one input, the object to be summarised. More complex functions might require a number of inputs. For example, the help notes for the `descriptives()` function in the `jmv` package show a large number of inputs can be specified:

descriptives {jmv} R Documentation

Descriptives

Description

Descriptives are an assortment of summarising statistics, and visualizations which allow exploring the shape and distribution of data. It is good practice to explore your data with descriptives before proceeding to more formal tests.

Usage

```
descriptives(data, vars, splitBy = NULL, freq = FALSE,
  desc = "columns", hist = FALSE, dens = FALSE, bar = FALSE,
  barCounts = FALSE, box = FALSE, violin = FALSE, dot = FALSE,
  dotType = "jitter", boxMean = FALSE, boxLabelOutliers = TRUE,
  qq = FALSE, n = TRUE, missing = TRUE, mean = TRUE,
  median = TRUE, mode = FALSE, sum = FALSE, sd = TRUE,
  variance = FALSE, range = FALSE, min = TRUE, max = TRUE,
  se = FALSE, ci = FALSE, ciWidth = 95, iqr = FALSE,
  skew = FALSE, kurt = FALSE, sw = FALSE, pcEqGr = FALSE,
  pcNEqGr = 4, pc = FALSE, pcValues = "25,50,75", formula)
```

There are two things to note here. First, notice that the first two inputs are listed with no = symbol, but all other inputs are listed with = symbols (with values provided after the = symbol). This means that everything apart from `data` and `vars` have **default** values. We are free to not specify values for these inputs if we are happy with the defaults provided. For example, by default the variance is not calculated (as `variance = FALSE`). To obtain the variance as well as the standard deviation, we can change this default to `variance = TRUE`:

```
# Only the standard deviation is provided as the measure of variability
descriptives(data=pbc, vars=age)

# Additionally request the variance to be calculated
descriptives(data=pbc, vars=age, variance=TRUE)
```

Second, for functions with multiple inputs, we can specify the input name and its value, or we can ignore the input name and specify just the input values **in the order listed in the Usage section**. So the following are equivalent:

```
# We can specify that the dataset to be summarised is pbc,
#   and the variable to summarise is age:
descriptives(data=pbc, vars=age)

# We can omit the input name, as long as we keep the inputs in the correct order -
```

```
#   that is, dataset first, variable second:
descriptives(pbc, age)

# We can change the order of the inputs, as long as we specify the input name:
descriptives(vars=age, data=pbc)
```

In this course, we will usually provide all the input names, even when they are not required. As you become more familiar with R, you will start to use the shortcut method.

1.6.4.1 The curse of inconsistency

As R is an open-source project, many people have contributed to its development. This has led to a frustrating part of R: some functions require a single object to be specified, but some require you to specify a data frame and select variables for analysis. Let's see an example.
The help for `summary()` specifies the usage as: `summary(object, ...)`. This means we need to specify a single object to be summarised. An object could be a single column of data (i.e. a vector), or it could be a data frame. If we have a data frame called `pbc` which contains many variables, the command `summary(pbc)` would summarise every variable in the data frame.
What if we only wanted to summarise the age of the participants in the data frame? To select a single variable from a data frame, we can use the following syntax: `dataframe$variable`. So to summarise just age from this data frame, we would use: `summary(pbc$age)`.
Compare this with the `descriptives()` function in the `jmv` package. We saw earlier that the two required inputs for `descriptives()` are `data` (the data frame to be analysed) and `vars` (the variables to be analysed). So to summarise `age` from the `pbc` data frame, we would specify `descriptives(data=pbc, vars=age)`.
This inconsistency will seem maddening at first, and will continue to be maddening! Reading the **usage** section of the help pages is a useful way to determine whether you should specify an object (like `pbc$age`) or a data frame and a list of variables.

1.6.5 Packages

A **package** is a collection of functions, documentation (and sometimes datasets) that extend the capabilities of R. Packages have been written by R users to be freely distributed and used by others. R packages can be obtained from many sources, but the most common source is CRAN: the Comprehensive R Archive Network.
A useful way of thinking about R is that R is like a smartphone, with packages being like apps which are downloaded from CRAN (similar to an app-store). When you first install R, it comes with a basic set of packages (apps) installed. You can do a lot of things with these basic packages, but sometimes you might want to do things differently, or you may want to perform some analyses that can't be done using the default packages. In these cases, you can install a package.
Like installing an app on a smartphone, you only need to *install* a package once. But each time you want to use the package, you need to *load* the package into R.

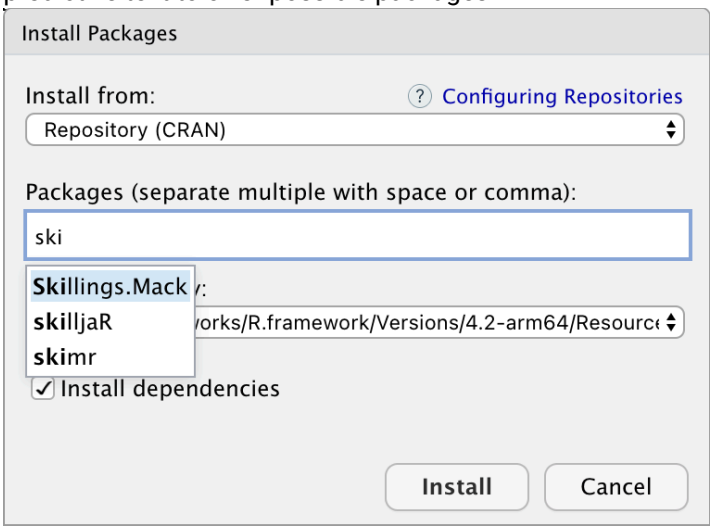
1.6.6 How to install a package

There are a couple of ways to install a package. You can use the `install.packages()` function if you know the exact name of the package. Let's use an example of installing the `skimr` package, which gives a very nice, high-level overview of any data frame. We can install `skimr` by typing the following into the console:

```
install.packages("skimr")
```

Note the use of the quotation marks.
Alternatively, RStudio offers a graphical way of installing packages that can be accessed via **Tools > Install Packages**, or via the **Install** button at the top of the **Packages** tab in the bottom-right window.

You can begin typing the name of the package in the dialog box that appears, and RStudio will use predictive text to offer possible packages:



While writing code is usually the recommended way to use R, installing packages is an exception. Using **Tools > Install Packages** is perfectly fine, because you only need to install a package once.

1.6.7 How to load a package

When you begin a new session in RStudio, i.e. when you open RStudio, only certain core packages are automatically loaded. You can use the `library()` function to load a package that you has previously been installed. For example, now that we have installed `skimr`, we need to load it before we can use it:

```
library(skimr)
```

Note that quotation marks are not required for the `library()` function (although they can be included if you really like quotation marks!).

Installing vs loading packages

Package installation:

- use the `install.packages()` function (note the 's') or **Tools > Install packages**
- the package name must be surrounded by quotation marks
- only needs to be done once

Package loading

- use the `library()` function
- the package name does not need to be surrounded by quotation marks
- must be done for each R session

1.7 What is this thing called the tidyverse?

If you have done much reading about R, you may have come across the tidyverse:

> "The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures." https://www.tidyverse.org/

Packages in the tidyverse have been designed with a goal to make using R more consistent by defining a "grammar" to manipulate data, examine data and draw conclusions from data. While the tidyverse is a common and powerful set of packages, we will not be teaching the tidyverse in this course for two main reasons:

1. The data we provide have been saved in a relatively tidy way, and do not need much manipulation for analyses to be conducted. The cognitive load in learning the tidyverse in this course is greater than the benefit that could be gained.
2. There are many resources (online, in print etc) that are based on `base R`, and do not use the tidyverse. It would be difficult to understand these resources if we taught only tidyverse techniques. In particular, the `dataframe$variable` syntax is an important concept that should be understood before moving into the tidyverse.

In saying all of this, I think the tidyverse is an excellent set of packages, which I frequently use. At the completion of this course, you will be well equipped to teach yourself tidyverse using many excellent resources such as: Tidyverse Skills for Data Science and R for Data Science.

Part 2: Obtaining basic descriptive statistics

In this exercise, we will analyse data to complete a descriptive table from a research study. The data come from a study in primary biliary cirrhosis, a condition of the liver, from Modeling Survival Data: Extending the Cox Model Therneau and Grambsch [2010]. By the end of this exercise, we will have completed the following table.

Table 1.2: Summary of 418 participants from the PBC study (Therneau and Grambsch, 2000)

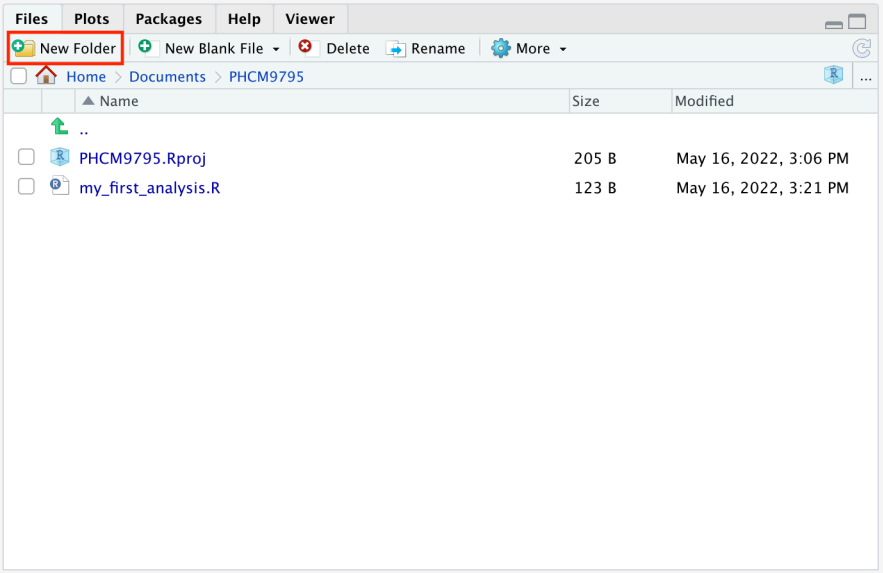
Characteristic		**Summary**
Age (years)		Mean (SD) or Median [IQR]
Sex	Male	n (%)
	Female	n (%)
AST* (U/ml)		Mean (SD) or Median [IQR]
Serum bilirubin		Mean (SD) or Median [IQR]
Stage	I	n (%)
	II	n (%)
	III	n (%)
	IV	n (%)
Vital status at study end	Alive: no transplant	n (%)
	Alive: transplant	n (%)
	Deceased	n (%)

* asparate aminotransferase

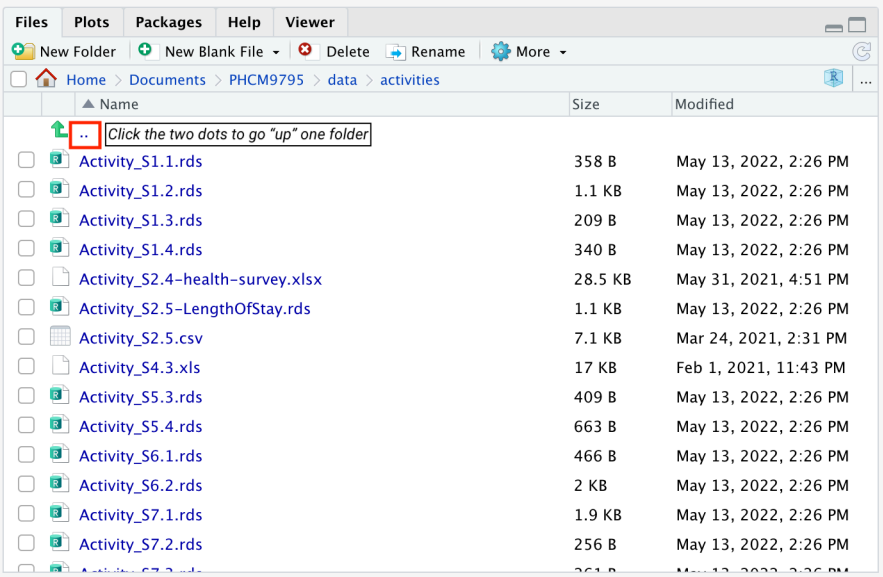
This table is available in Table1.docx, saved on Moodle.

1.8 Set up your data

We created a project in the previous step. We will now create a folder to store all the data for this course. Storing the data within the project makes life much easier!
Create a new folder by clicking the **New Folder** icon in the **Files** tab at the bottom-right:



Call the new folder **data**.
Click on this folder to open it, and then create two new folders: **activities** and **examples**.
Download the "Data sets: for learning activities" from Moodle, and use Windows Explorer or MacOS Finder to save these data sets in **activities**. Save the "Data sets: example data from course notes" into the **examples** folder.
Your **activities** folder should look like:



Click the two dots next to the up-arrow at the top of the folder contents to move back up the folder structure. Note that you need to click the dots, and not the up-facing green arrow!

1.9 Reading a data file

Typing data directly into R is not common; we usually read data that have been previously saved. In this example, we will read an `.rds` file using the `readRDS()` function, which has only one input: the

location of the file.
1 - Confirm that the `pbc.rds` file is in the `activities` sub-folder within the `data` folder (as per the previous steps).
2 - Install the packages: `jmv`, `skimr` and `summarytools` using **Tools > Install packages**, or by typing into the console:

```
install.packages("jmv")
install.packages("skimr")
install.packages("summarytools")
```

3 - Load the `skimr` package, and use the `readRDS()` function to read the file into R, assigning it to a data frame called `pbc`. Because we set up our project, we can locate our data easily by telling R to use the file: `"data/activities/pbc.rds"`, which translates as: the file `pbc.rds` which is located in the `activities` sub-folder within the `data` folder.

```
library(skimr)

pbc <- readRDS("data/activities/pbc.rds")
```

4 - We can now use the `summary()` function to examine the pbc dataset:

```
summary(pbc)
```

```
##        id              time          status            trt
##  Min.   :  1.0   Min.   :  41   Min.   :0.0000   Min.   :1.000
##  1st Qu.:105.2   1st Qu.:1093   1st Qu.:0.0000   1st Qu.:1.000
##  Median :209.5   Median :1730   Median :0.0000   Median :1.000
##  Mean   :209.5   Mean   :1918   Mean   :0.8301   Mean   :1.494
##  3rd Qu.:313.8   3rd Qu.:2614   3rd Qu.:2.0000   3rd Qu.:2.000
##  Max.   :418.0   Max.   :4795   Max.   :2.0000   Max.   :2.000
##                                                  NA's   :106
##       age             sex           ascites            hepato
##  Min.   :26.28   Min.   :1.000   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:42.83   1st Qu.:2.000   1st Qu.:0.00000   1st Qu.:0.0000
##  Median :51.00   Median :2.000   Median :0.00000   Median :1.0000
##  Mean   :50.74   Mean   :1.895   Mean   :0.07692   Mean   :0.5128
##  3rd Qu.:58.24   3rd Qu.:2.000   3rd Qu.:0.00000   3rd Qu.:1.0000
##  Max.   :78.44   Max.   :2.000   Max.   :1.00000   Max.   :1.0000
##                                  NA's   :106       NA's   :106
##     spiders           edema             bili             chol
##  Min.   :0.0000   Min.   :0.0000   Min.   : 0.300   Min.   : 120.0
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.: 0.800   1st Qu.: 249.5
##  Median :0.0000   Median :0.0000   Median : 1.400   Median : 309.5
##  Mean   :0.2885   Mean   :0.1005   Mean   : 3.221   Mean   : 369.5
##  3rd Qu.:1.0000   3rd Qu.:0.0000   3rd Qu.: 3.400   3rd Qu.: 400.0
##  Max.   :1.0000   Max.   :1.0000   Max.   :28.000   Max.   :1775.0
##  NA's   :106                                        NA's   :134
##     albumin          copper          alkphos             ast
##  Min.   :1.960   Min.   :  4.00   Min.   :  289.0   Min.   : 26.35
##  1st Qu.:3.243   1st Qu.: 41.25   1st Qu.:  871.5   1st Qu.: 80.60
##  Median :3.530   Median : 73.00   Median : 1259.0   Median :114.70
##  Mean   :3.497   Mean   : 97.65   Mean   : 1982.7   Mean   :122.56
##  3rd Qu.:3.770   3rd Qu.:123.00   3rd Qu.: 1980.0   3rd Qu.:151.90
##  Max.   :4.640   Max.   :588.00   Max.   :13862.4   Max.   :457.25
##                  NA's   :108      NA's   :106       NA's   :106
##       trig           platelet        protime          stage
```

```
##  Min.   : 33.00   Min.   : 62.0   Min.   : 9.00   Min.   :1.000
##  1st Qu.: 84.25   1st Qu.:188.5   1st Qu.:10.00   1st Qu.:2.000
##  Median :108.00   Median :251.0   Median :10.60   Median :3.000
##  Mean   :124.70   Mean   :257.0   Mean   :10.73   Mean   :3.024
##  3rd Qu.:151.00   3rd Qu.:318.0   3rd Qu.:11.10   3rd Qu.:4.000
##  Max.   :598.00   Max.   :721.0   Max.   :18.00   Max.   :4.000
##  NA's   :136      NA's   :11      NA's   :2       NA's   :6
```

An alternative to the `summary()` function is the `skim()` function in the `skimr` package, which produces summary statistics as well as rudimentary histograms:

```
skim(pbc)
```

```
── Data Summary ────────────────────────
                           Values
Name                       pbc
Number of rows             418
Number of columns          20
_________________________
Column type frequency:
  numeric                  20
_________________________
Group variables            None

── Variable type: numeric ──────────────────────────────────────────────────────────────────────────────────────────────
   skim_variable n_missing complete_rate      mean        sd     p0     p25     p50     p75     p100 hist
 1 id                    0         1      210.      121.      1     105.    210.    314.     418
 2 time                  0         1     1918.     1105.     41    1093.   1730    2614.    4795
 3 status                0         1        0.830     0.956   0       0       0       2        2
 4 trt                 106         0.746    1.49      0.501   1       1       1       2        2
 5 age                   0         1       50.7      10.4    26.3    42.8    51.0    58.2     78.4
 6 sex                   0         1        1.89      0.307   1       2       2       2        2
 7 ascites             106         0.746    0.0769    0.267   0       0       0       0        1
 8 hepato              106         0.746    0.513     0.501   0       0       1       1        1
 9 spiders             106         0.746    0.288     0.454   0       0       0       1        1
10 edema                 0         1        0.100     0.253   0       0       0       0        1
11 bili                  0         1        3.22      4.41    0.3     0.8     1.4     3.4     28
12 chol                134         0.679  370.      232.    120     250.    310.    400     1775
13 albumin               0         1        3.50      0.425   1.96    3.24    3.53    3.77     4.64
14 copper              108         0.742   97.6      85.6     4      41.2    73      123      588
15 alkphos             106         0.746 1983.     2140.    289     872.   1259    1980    13862.
16 ast                 106         0.746  123.      56.7    26.4    80.6   115.    152.     457.
17 trig                136         0.675  125.      65.1    33      84.2   108     151      598
18 platelet             11         0.974  257.      98.3    62     188.    251     318      721
19 protime               2         0.995   10.7      1.02    9      10      10.6    11.1     18
20 stage                 6         0.986    3.02     0.882   1       2       3       4        4
```

The `summary()` and `skim()` functions are useful to give a quick overview of a dataset: how many variables are included, how variables are coded, which variables contain missing data and a crude histogram showing the distribution of numeric variables.

1.10 Summarising continuous variables

One of the most flexible functions for summarising continuous variables is the `descriptives()` function from the `jmv` package. The function is specified as `descriptives(data=, vars=)` where:

- `data` specifies the dataframe to be analysed
- `vars` specifies the variable(s) of interest, with multiple variables combined using the `c()` function

We can summarise the three continuous variables in the pbc data: age, AST and serum bilirubin, as shown below.

```
library(jmv)

descriptives(data=pbc, vars=c(age, ast, bili))
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                          age         ast         bili
##
##    N                          418         312          418
##    Missing                      0         106            0
##    Mean                  50.74155    122.5563     3.220813
##    Median                51.00068    114.7000     1.400000
##    Standard deviation    10.44721    56.69952     4.407506
##    Minimum               26.27789    26.35000    0.3000000
##    Maximum               78.43943    457.2500     28.00000
##
```

By default, the `descriptives` function presents the mean, median, standard deviation, minimum and maximum. We can request additional statistics, such as the quartiles (which are called the percentiles, or pc, in the descriptives function):

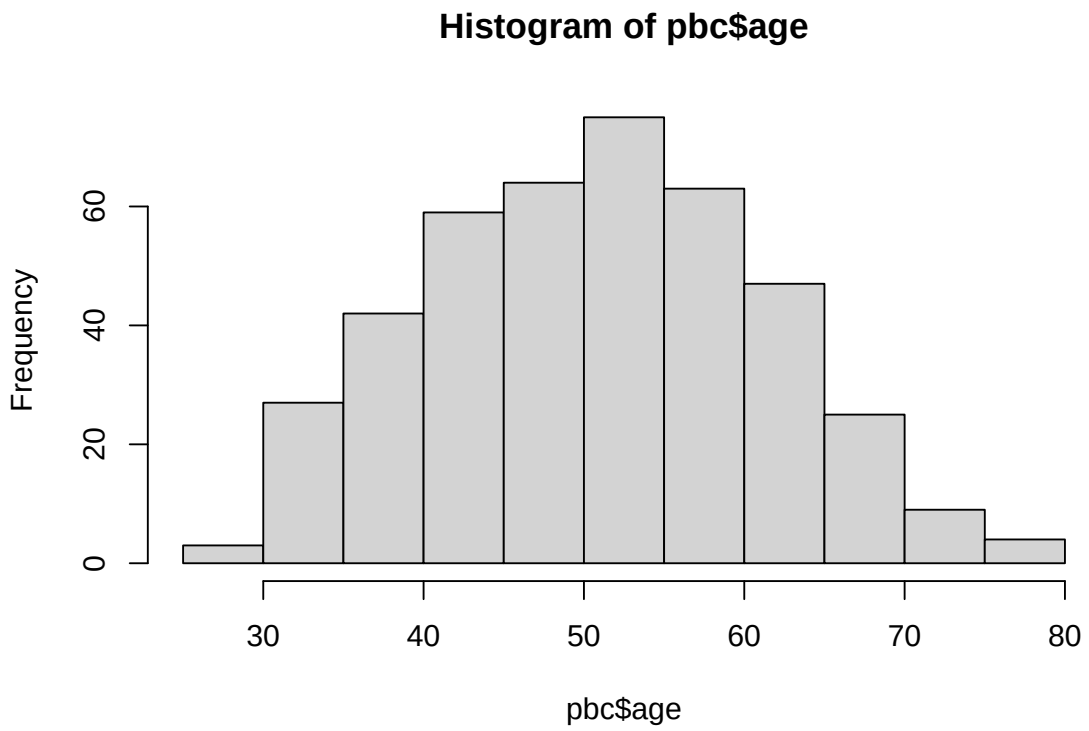
```
descriptives(data=pbc, vars=c(age, ast, bili), pc=TRUE)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                          age         ast         bili
##
##    N                          418         312          418
##    Missing                      0         106            0
##    Mean                  50.74155    122.5563     3.220813
##    Median                51.00068    114.7000     1.400000
##    Standard deviation    10.44721    56.69952     4.407506
##    Minimum               26.27789    26.35000    0.3000000
##    Maximum               78.43943    457.2500     28.00000
##    25th percentile       42.83231    80.60000    0.8000000
##    50th percentile       51.00068    114.7000     1.400000
##    75th percentile       58.24093    151.9000     3.400000
##
```

1.11 Producing a histogram

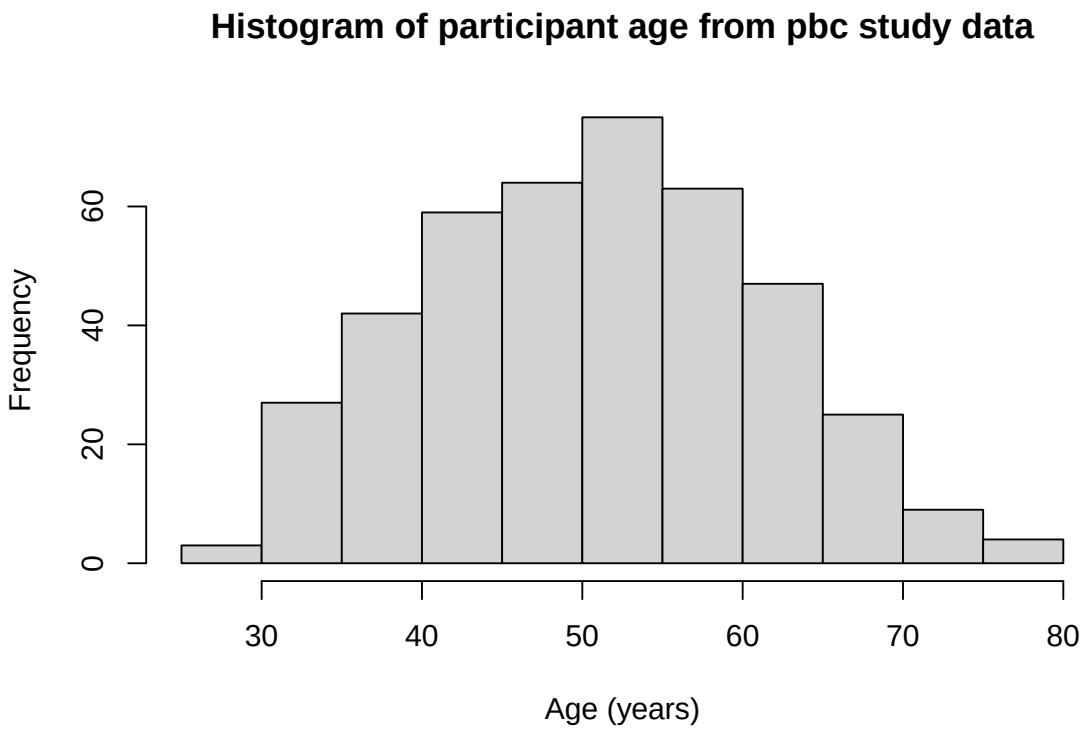
We can use the `hist()` function to produce a histogram, specifying the dataframe to use and the variable to be plotted as `dataframe$variable`:

```
hist(pbc$age)
```



The histogram function does a remarkably good job of choosing cutpoints and binwidths, and these rarely need to be changed. However, the labelling of the histogram should be improved by using `xlab=" "` and `main=" "` to assign labels for the x-axis and overall title respectively:

```
hist(pbc$age, xlab="Age (years)",
     main="Histogram of participant age from pbc study data")
```



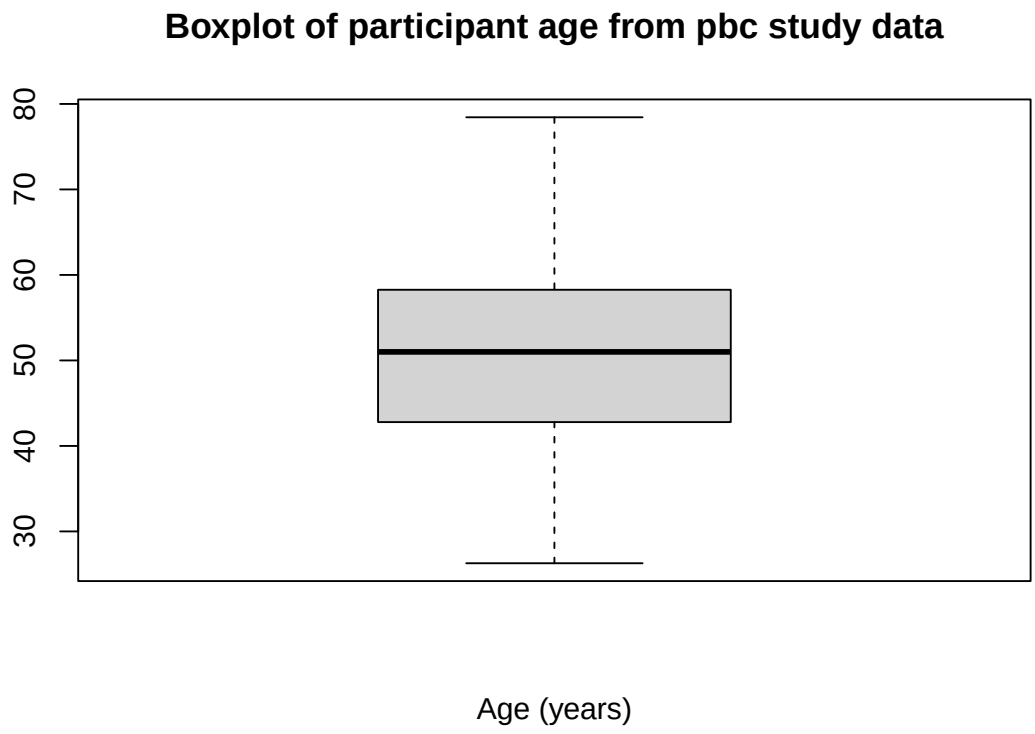
By default, the `hist()` function plots a **frequency histogram**, with counts on the y-axis. We can tweak the histogram using the following code to plot a histogram of the **relative frequencies**:

```
h <- hist(pbc$age, plot=FALSE)
h$density <- h$counts/sum(h$counts)*100
plot(h, freq=FALSE,
     xlab="Age (years)",
     ylab="Relative frequency (%)",
     main="Histogram of participant age from pbc study data")
```

1.12 Producing a boxplot

The `boxplot` function is used to produce boxplots, again specifying the dataframe to use and the variable to be plotted as `dataframe$variable`. Labels can be applied in the same way as the histogram:

```
boxplot(pbc$age, xlab="Age (years)", main="Boxplot of participant age from pbc study data")
```



1.13 Producing a one-way frequency table

We have three categorical variables to summarise in Table 1: sex, stage and vital status. These variables are best summarised using one-way frequency tables.

```
library(summarytools)

freq(pbc$sex)
```

```
## Frequencies
## pbc$sex
## Type: Numeric
##
##               Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
## ----------- ------ --------- -------------- --------- --------------
##           1     44     10.53          10.53     10.53          10.53
##           2    374     89.47         100.00     89.47         100.00
##        <NA>      0                               0.00         100.00
##       Total    418    100.00         100.00    100.00         100.00
```

1.13.1 Defining categorical variables as factors

You will notice that the table above, in its current form, is uninterpretable as the 1 and 2 categories are not labelled. In this course, all variables including categorical variables tend to be numerically coded. To define a categorical variable as such in R, we define it as a **factor** using the factor function: `factor(variable=, levels=, labels=)`
We specify:

- levels: the values the categorical variable can take
- labels: the labels corresponding to each of the levels (entered in the same order as the levels)

To define our variable sex as a factor, we use:

```
pbc$sex <- factor(pbc$sex, levels=c(1, 2), labels=c("Male", "Female"))
```

We can confirm the coding by re-running a frequency table:

```
freq(pbc$sex)
```

```
## Frequencies
## pbc$sex
## Type: Factor
##
##                Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
## ------------ ------ --------- -------------- --------- --------------
##         Male     44     10.53          10.53     10.53          10.53
##       Female    374     89.47         100.00     89.47         100.00
##         <NA>      0                                0.00         100.00
##        Total    418    100.00         100.00    100.00         100.00
```

Task: define `stage` and `status` (Vital Status) as factors, and produce one-way frequency tables. Refer to the file `pbc_info.txt` to view the labels for each variable. For example, for Stage:

```
pbc$stage <- factor(pbc$stage, levels=c(1, 2, 3, 4), labels=c("Stage 1", "Stage 2", "Stage 3", "Stage
freq(pbc$stage)
```

```
## Frequencies
## pbc$stage
## Type: Factor
##
##                 Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
## ------------- ------ --------- -------------- --------- --------------
##       Stage 1     21      5.10           5.10      5.02           5.02
##       Stage 2     92     22.33          27.43     22.01          27.03
##       Stage 3    155     37.62          65.05     37.08          64.11
##       Stage 4    144     34.95         100.00     34.45          98.56
##          <NA>      6                                1.44         100.00
##         Total    418    100.00         100.00    100.00         100.00
```

1.14 Producing a two-way frequency table

To produce tables summarising two categorical variables, we can use the `contTables()` function within the `jmv` package. The minimal inputs to include are `data`: the name of the data frame to be analysed, `rows`: the variable representing the rows of the table, and `cols`: the name of the columns of the table.
For example, to produce a two-way table showing stage of disease by sex using the pbc data frame, we use:

```
contTables(data=pbc, rows=sex, cols=stage)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    sex       Stage 1    Stage 2    Stage 3    Stage 4    Total
##
##    Male            3          8         16         17       44
##    Female         18         84        139        127      368
##    Total          21         92        155        144      412
##
##
##
##  x² Tests
##
##          Value        df    p
##
##    x²    0.8779873     3    0.8307365
##    N           412
##
```

[The bottom part of the output, χ^2 Tests, can be ignored for now]
You may notice in the above that the number of observations is now 412. This is because there are missing observations for either sex or stage: which is it, and how would you determine this?
From the cross-tabulation, you can see the individual frequencies of participants in each of the categories in each cell. For example, there are 3 male participants who have Stage 1 disease. You can also read the totals for each row and column. For example, there are 44 males, and 144 participants have Stage 4 disease.
You can also add percentages into your table using `pcCol=TRUE` to include column percents, and `pcRow=TRUE` for row percents. For example, to calculate the relative frequencies (i.e. percentages) of sex within each stage, we would request **column percents** with the option: `pcCol=TRUE`.

```
contTables(data=pbc, rows=sex, cols=stage, pcCol=TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    sex                      Stage 1     Stage 2     Stage 3     Stage 4     Total
##
##    Male     Observed              3           8          16          17         44
##           % within column   14.28571    8.69565   10.32258    11.80556   10.67961
##
##    Female   Observed             18          84         139         127        368
##           % within column   85.71429   91.30435   89.67742    88.19444   89.32039
##
##    Total    Observed             21          92         155         144        412
##           % within column  100.00000  100.00000  100.00000   100.00000  100.00000
##
##
##
##  x² Tests
##
##          Value        df    p
```

```
##
##    x²    0.8779873     3    0.8307365
##    N           412
##
```

We can see that the 3 male participants with Stage 1 disease represent 14% of those with Stage 1 disease.

1.15 Saving data in R

There are many ways to save data from R, depending on the type of file you want to save. The recommendation for this course is to save your data using the `.rds` format, using the `saveRDS()` function, which takes two inputs: `saveRDS(object, file)`. Here, `object` is the R object to be saved (usually a data frame), and `file` is the location for the file to be saved (file name and path, including the `.rds` suffix).
It is not necessary to save our PBC data, as we have made only minor changes to the data that can be replicated by rerunning our script. If you had made major changes and wanted to save your data, you could use:

```
saveRDS(pbc, file="pbc_revised.rds")
```

1.16 Copying output from R

It is important to note that saving your data or your script in R will not save your output. The easiest way to retain the output of your analyses is to copy the output from the Console into a word processor package (e.g. Microsoft Word) before closing R.
Unfortunately, by default, R is not ideal for creating publication quality tables. There are many packages that will help in this process, such as R Markdown, bookdown[1], huxtable, gt and gtsummary, but their use is beyond the scope of this course. R Markdown for Scientists provides an excellent introduction to R Markdown.

> Task: Complete Table 1 using the output generated in this exercise. You should decide on whether to present continuous variables by their means or medians, and present the most appropriate measure of spread. Include footnotes to indicate if any variables contain missing observations.

Part 3: Creating other types of graphs

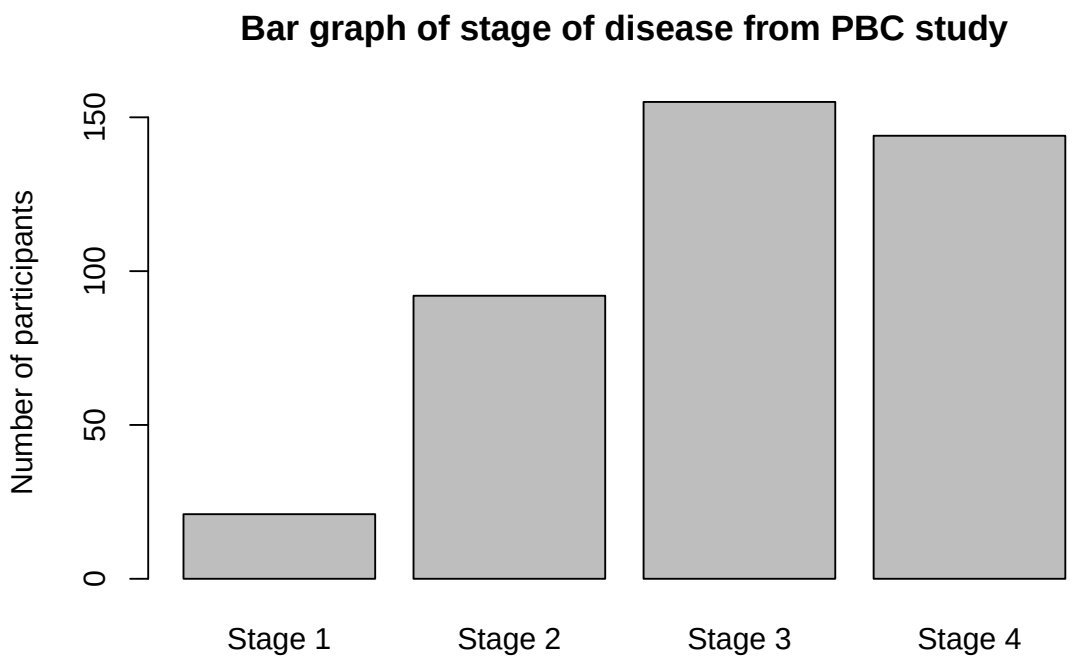
The `plot()` function, also known as *base graphics*, is the default method of plotting data in R that can produce publication-quality graphics with minimal coding. There are alternative packages for plotting, with `ggplot2` being one of the most well known. We will present instructions for base graphics in this course, but excellent documentation for `ggplot2` can be found at the ggplot2: Elegant Graphics for Data Analysis website, written by the package authors.

1.17 Bar graphs

The simplest way to use the `plot()` function is by specifying an object to be plotted. As with the `hist()` function, to plot a single variable from a data frame, we must define it using: `dataframe$variable`.
Here we will create the bar chart shown in Figure 1.3 of the statistics notes using the `pbc.rds` dataset. The x-axis of this graph will be the stage of disease, and the y-axis will show the number of participants in each category.

[1] these R notes and the PHCM9795 course notes have been written using bookdown

```
plot(pbc$stage,
     main="Bar graph of stage of disease from PBC study",
     ylab="Number of participants")
```



Note that stage is a categorical variable, that has been defined as a factor (in Section 1.13.1). You **must define categorical data as factors** to plot them in a bar graph.

1.17.1 Clustered bar graph

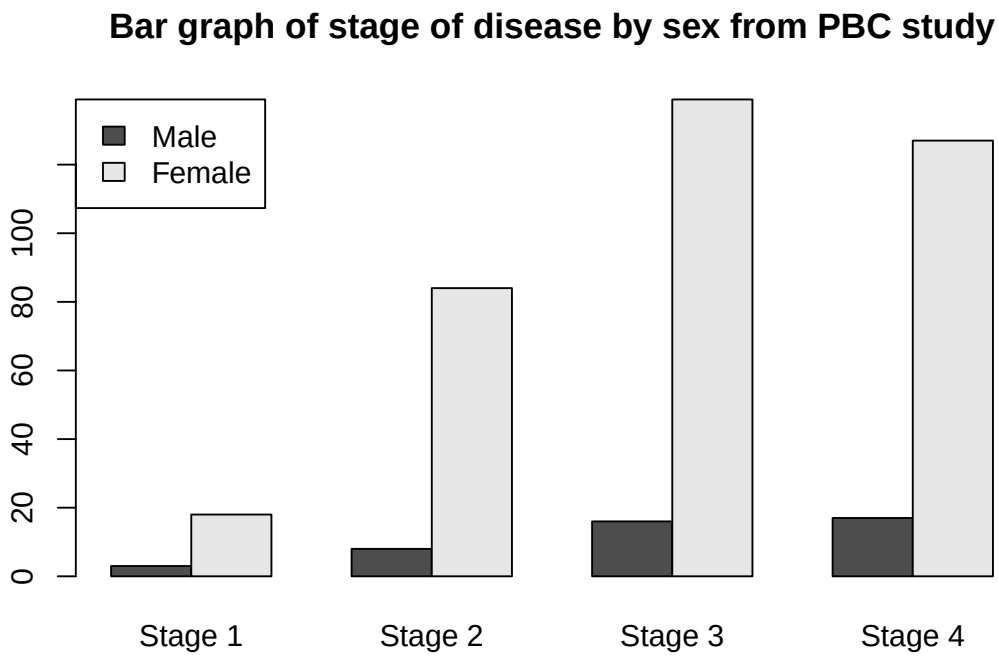
To create a clustered bar chart as shown in Figure 1.4 of the statistics notes, we need to do a bit of manipulation. We first need to tabulate the data using the `table()` function. We want to plot stage of disease broken down by sex, so we specify `sex` as the first variable, and `stage` as the second variable for the `table()` command.

```
counts <- table(pbc$sex, pbc$stage)
counts
```

```
##
##          Stage 1 Stage 2 Stage 3 Stage 4
##   Male         3       8      16      17
##   Female      18      84     139     127
```

After tabulating the data, we use the `barplot()` function to plot the summarised data. We specify the main title using `main=" "`, specify that the stages be plotted separately by sex (`beside=TRUE`), specify the legend be defined by sex, and position the legend in the top-left of the graph:

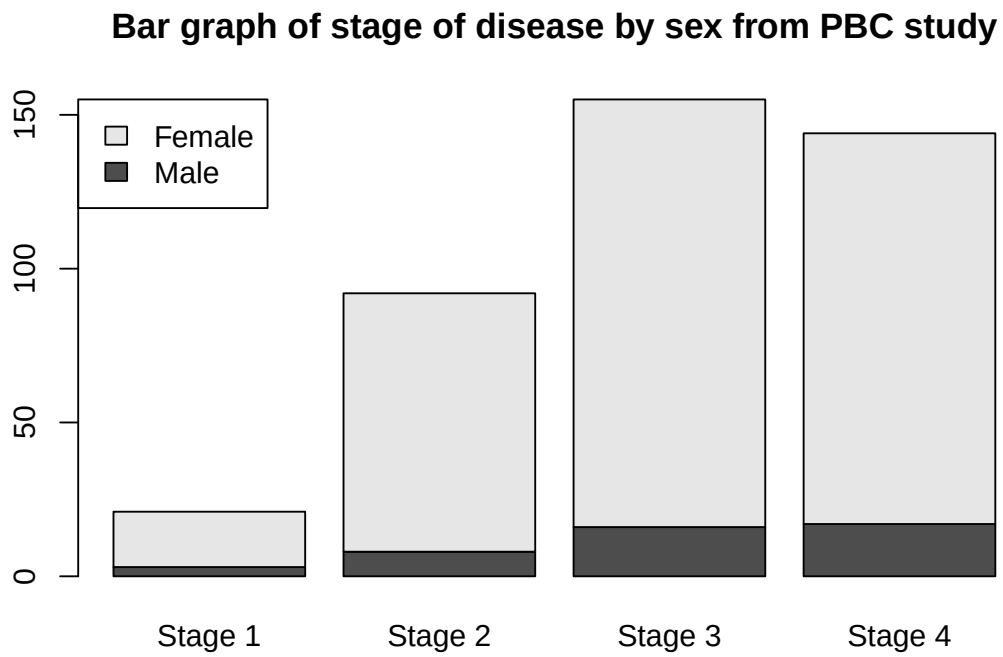
```
barplot(counts, main="Bar graph of stage of disease by sex from PBC study",
        beside=TRUE, legend = rownames(counts), args.legend = list(x = "topleft"))
```



1.17.2 Stacked bar graph

A stacked bar graph can be constructed as for the clustered bar graph, but we specify `beside=FALSE`:

```
barplot(counts, main="Bar graph of stage of disease by sex from PBC study",
        beside=FALSE, legend = rownames(counts), args.legend = list(x = "topleft"))
```



1.17.3 Stacked bar graph of relative frequencies

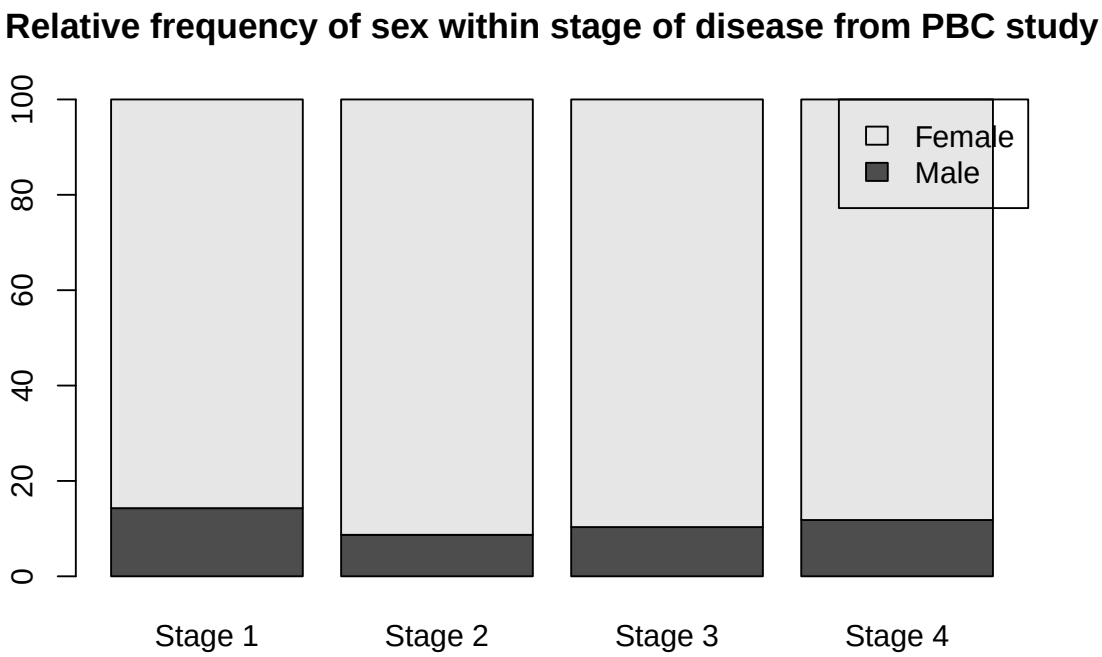
To plot relative frequencies, we need to transform our table of frequencies (`counts`) into proportions, by using the `prop.table()` function. The prop.table() function takes two arguments: a table of counts, and `margin`, which defines whether we want proportions calculated by row (`margin=1`) or column (`margin=2`).
We want to calculate the relative frequency of sex within each stage category. From our `counts` table above, this equates to calculating *column* proportions, so we specify `margin=2`. We also multiply the resulting table by 100 to obtain percentages (rather than proportions):

```
percent <- prop.table(counts, margin=2)*100
percent
```

```
##
##            Stage 1   Stage 2   Stage 3   Stage 4
##   Male   14.285714  8.695652 10.322581 11.805556
##   Female 85.714286 91.304348 89.677419 88.194444
```

After calculating the percentages, we use `barplot()` again, similar to the stacked bar graph:

```
barplot(percent,
        main="Relative frequency of sex within stage of disease from PBC study",
        legend = rownames(counts), beside=FALSE, args.legend = list(x = "topright"))
```



1.18 Creating line graphs

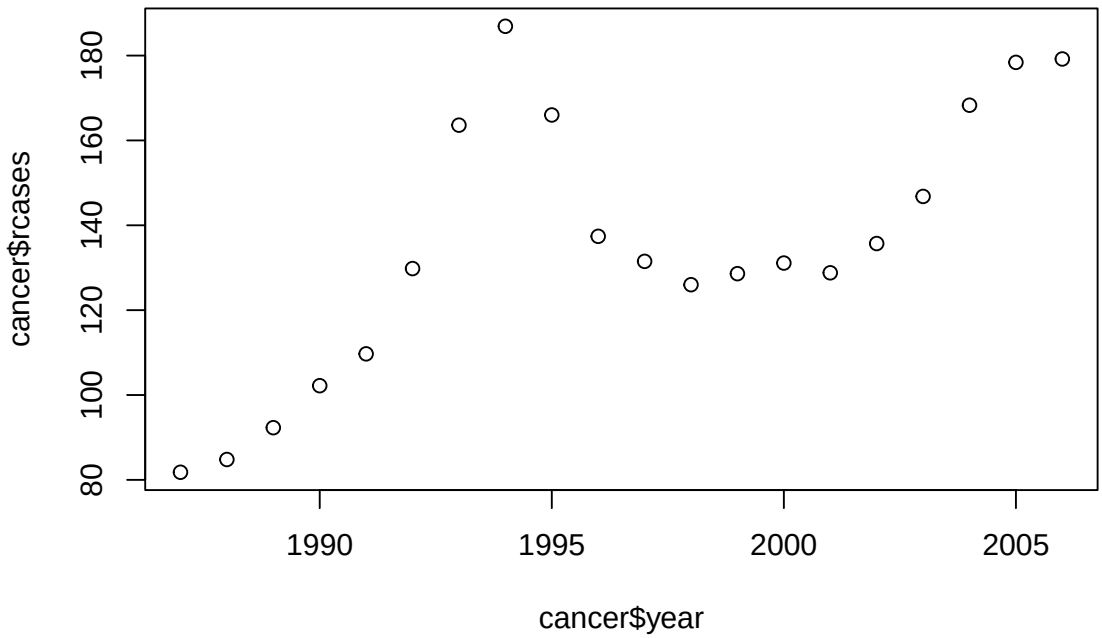
To demonstrate the graphing of aggregate data , we use the data on new cases and deaths from prostate cancer in males in NSW. This data has been entered as `Example_1.2.rds`.

```
cancer <- readRDS("data/examples/Example_1.2.rds")
summary(cancer)
```

```
##       year          ncases        ndeaths          rcases         rdeaths
##  Min.   :1987   Min.   :1567   Min.   : 645.0   Min.   : 81.8   Min.   :31.10
##  1st Qu.:1992   1st Qu.:2804   1st Qu.: 788.2   1st Qu.:121.9   1st Qu.:34.67
##  Median :1996   Median :3790   Median : 868.0   Median :131.3   Median :36.55
##  Mean   :1996   Mean   :3719   Mean   : 855.0   Mean   :135.4   Mean   :37.09
##  3rd Qu.:2001   3rd Qu.:4403   3rd Qu.: 921.0   3rd Qu.:164.2   3rd Qu.:40.38
##  Max.   :2006   Max.   :6158   Max.   :1044.0   Max.   :186.9   Max.   :43.80
```

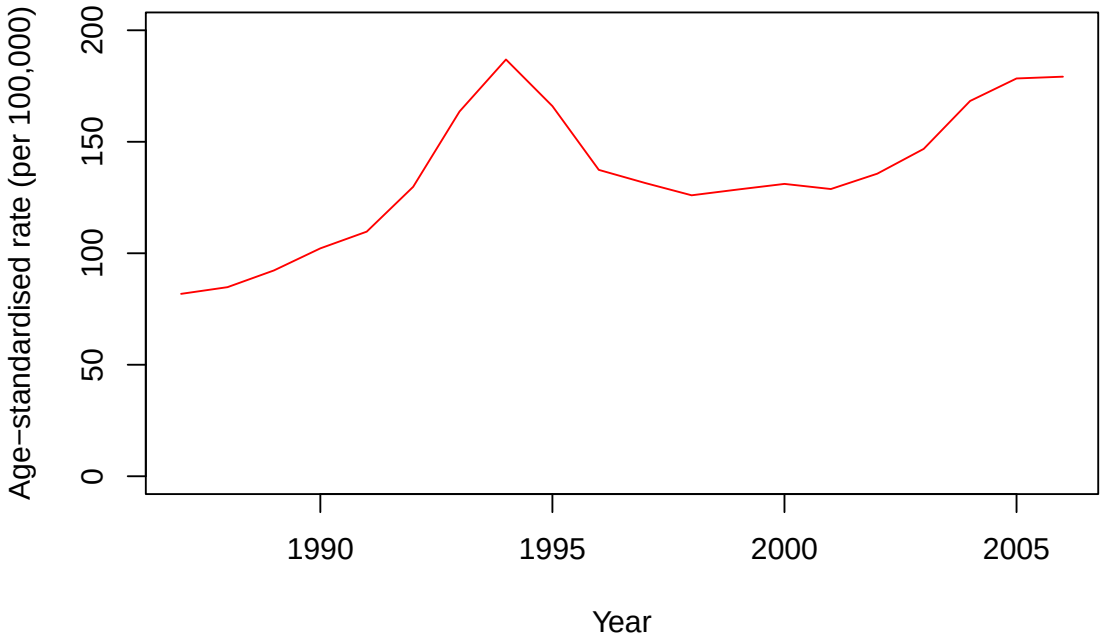
We begin by plotting cancer cases (as the *y* variable) against year (the *x* variable).

```
plot(x=cancer$year, y=cancer$rcases)
```



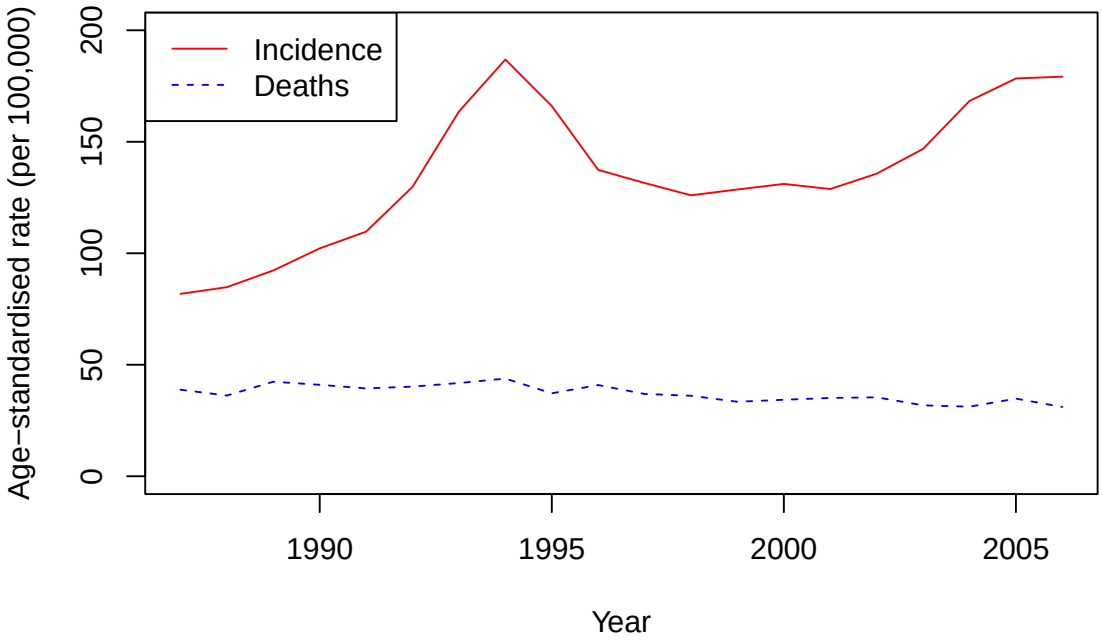
Let's define the plot to be joined by lines (`type="l"`), in the colour red (`col="red"`), providing meaningful labels for the x-axis and y-axis, and changing the scale of the y-axis to be between 0 and 200 (`ylim=c(0,200)`):

```
plot(x=cancer$year, y=cancer$rcases,
     type="l", col = "red",
     xlab = "Year",
     ylab = "Age-standardised rate (per 100,000)", ylim=c(0,200))
```



We can now add a second line to the plot using the `lines()` function, specifying a dashed line (`lty=2`), and add a legend to the plot:

```
plot(x=cancer$year, y=cancer$rcases, type="l", col = "red",
     xlab = "Year", ylab = "Age-standardised rate (per 100,000)",
     ylim=c(0,200))

lines(cancer$year, cancer$rdeaths, col = "blue", type = "l", lty = 2)

legend("topleft", legend=c("Incidence", "Deaths"),
       col=c("red", "blue"), lty = 1:2)
```



Note: coding for graphs is not always straightforward. Two excellent resources for creating graphs in R are: R Graphics Cookbook and The R Graph Gallery.

Module 2

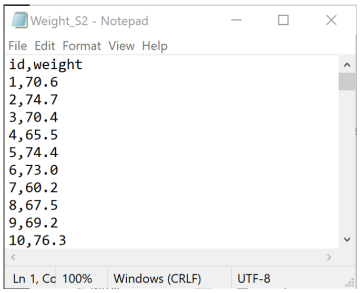
Probability and probability distributions

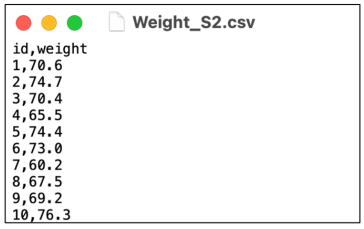
2.1 Importing data into R

We have described previously how to import data that have been saved as R .rds files. It is quite common to have data saved in other file types, such as Microsoft Excel, or plain text files. In this section, we will demonstrate how to import data from other packages into R.
There are two useful packages for importing data into R: `haven` (for data that have been saved by Stata, SAS or SPSS) and `readxl` (for data saved by Microsoft Excel). Additionally, the `labelled` package is useful in working with data that have been labelled in Stata.

2.1.1 Importing plain text data into R

A `csv` file, or a "comma separated variables" file is commonly used to store data. These files have a very simple structure: they are plain text files, where data are separated by commas. csv files have the advantage that, as they are plain text files, they can be opened by a large number of programs (such as Notepad in Windows, TextEdit in MacOS, Microsoft Excel - even Microsoft Word). While they can be opened by Microsoft Excel, they can be opened by many other programs: the csv file can be thought of as the lingua-franca of data.
In this demonstration, we will use data on the weight of 1000 people entered in a csv file called `weight_s2.csv` available on Moodle.
To confirm that the file is readable by any text editor, here are the first ten lines of the file, opened in Notepad on Microsoft Windows, and TextEdit on MacOS.





We can use the `read.csv` function:

```
sample <- read.csv("data/examples/Weight_s2.csv")
```

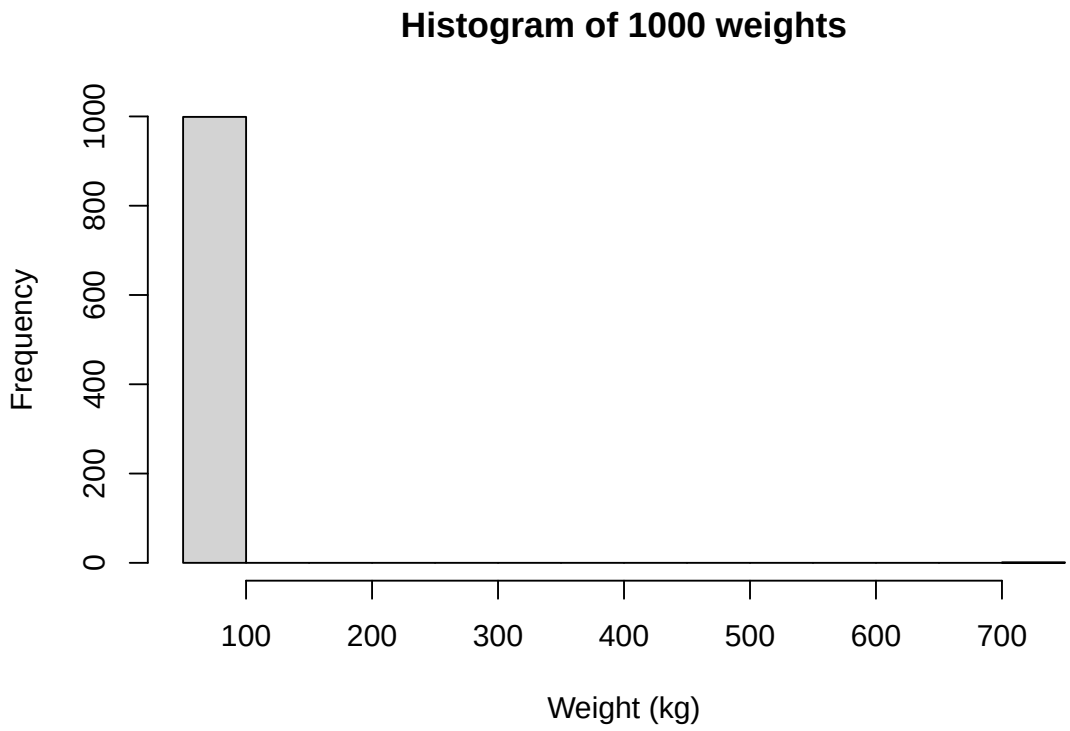
Here, the `read.csv` function has the default that the first row of the dataset contains the variable names. If your data do not have column names, you can use `header=FALSE` in the function.

Note: there is an alternative function `read_csv` which is part of the `readr` package (a component of the `tidyverse`). Some would argue that the `read_csv` function is more appropriate to use because of an issue known as `strings.as.factors`. The `strings.as.factors` default was removed in R Version 4.0.0, so it is less important which of the two functions you use to import a `.csv` file. More information about this issue can be found here and here.

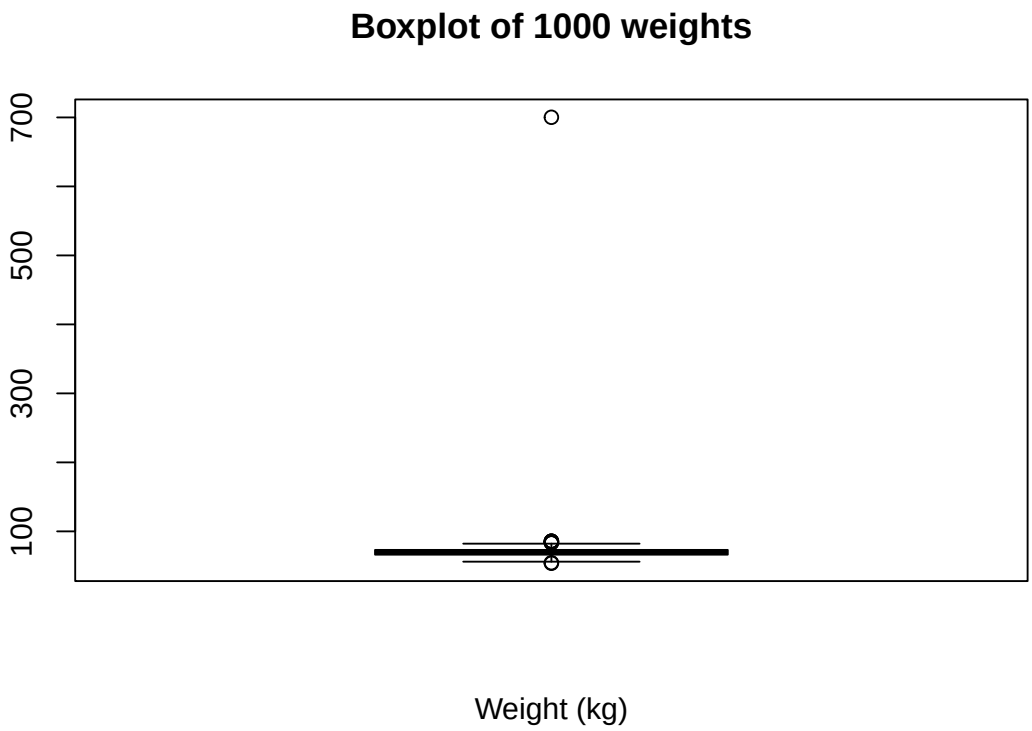
2.2 Checking your data for errors in R

Before you start describing and analysing your data, it is important to make sure that no errors have been made during the data entry process. Basically, you are looking for values that are outside the range of possible or plausible values for that variable.
If an error is found, the best method for correcting the error is to go back to the original data e.g. the hard copy questionnaire, to obtain the original value, entering the correct value into R If the original data is not available or the original data is also incorrect, the erroneous value is often excluded from the dataset.
For continuous variables, the easiest methods are to examine a boxplot and histogram. For example, a boxplot and histogram for the weight variable we just imported appear as:

```
hist(sample$weight, xlab="Weight (kg)", main="Histogram of 1000 weights")
```




```
boxplot(sample$weight, xlab="Weight (kg)", main="Boxplot of 1000 weights")
```



There is a clear outlying point shown in the boxplot. Although not obvious, the same point is shown in the histogram as a bar around 700 with a very short height.
We can identify any outlying observations in the dataset using the `subset` function. You will need to decide if these values are a data entry error or are biologically plausible. If an extreme value or "outlier", is biologically plausible, it should be included in all analyses.
For example, to list any observations from the `sample` dataset with a weight larger than 200:

```
subset(sample, weight>200)
```

```
##    id weight
## 58 58  700.2
```

We see that there is a very high value of 700.2kg. A value as high as 700kg is likely to be a data entry error (e.g. error in entering an extra zero) and is not a plausible weight value. Here, **you should check your original data**.
You might find that the original weight was recorded in medical records as 70.2kg. You can change this in R by writing code.
Note: many statistical packages will allow you to view a spreadsheet version of your data and edit values in that spreadsheet. This is not best practice, as corrected observations may revert to their original values depending on whether the edited data have been saved or not. By using code-based recoding, the changes will be reproduced the next time the code is run.
We will use an `ifelse` statement to recode the incorrect weight of 700.2kg into 70.2kg. The form of the `ifelse` statement is as follows:
`ifelse(test, value_if_true, value_if_false)`
Our code will create a new column (called `weight_clean`) in the `sample` dataframe. We will test whether `weight` is equal to 700.2; if this is true, we will assign `weight_clean` to be 70.2, otherwise `weight_clean` will equal the value of `weight`.
Putting it all together:

```
sample$weight_clean = ifelse(sample$weight==700.2, 70.2, sample$weight)
```

Note: if an extreme value lies within the range of biological plausibility it should not be removed from analysis.
Once you have checked your data for errors, you are ready to start analysing your data.

2.2.1 What on earth: == ?

In R, the test of equality is denoted by two equal signs: `==`. So we would use `==` to test whether an observation is equal to a certain value. Let's see an example:

```
# Test whether 6 is equal to 6
6 == 6
```

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

```
# Test whether 6 is equal to 42
6 == 42
```

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

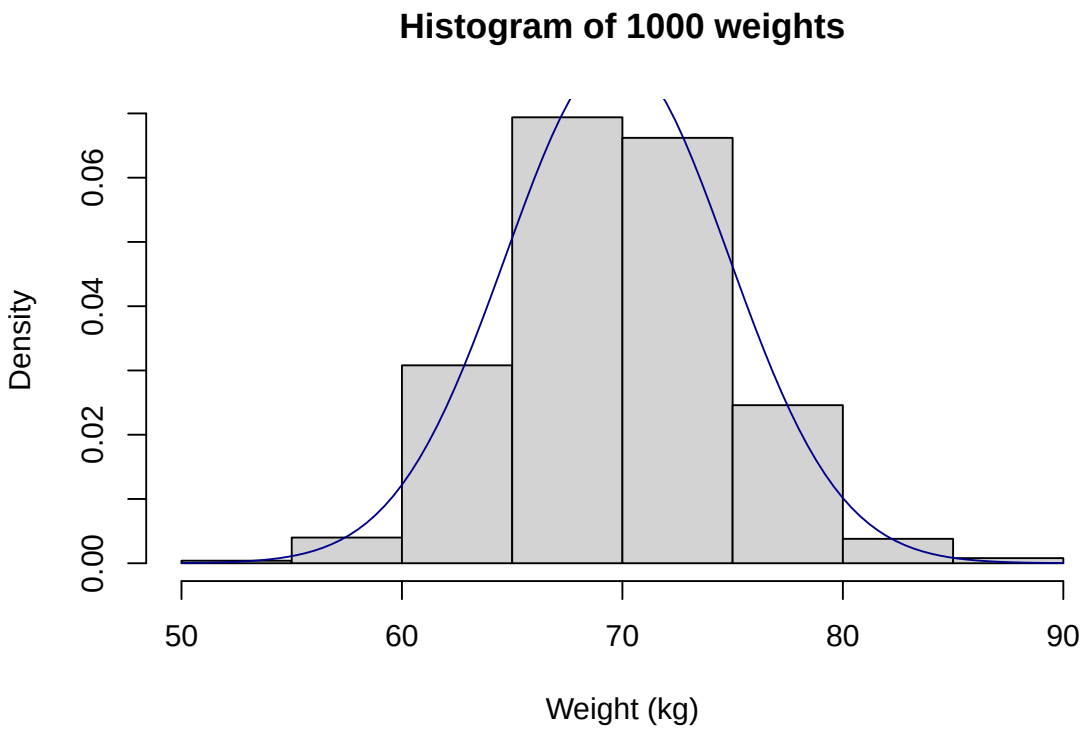
You can read the `==` as "is equal to". So the code `sample$weight == 700.2` is read as: "is the value of weight from the data frame sample equal to 700.2?". In our `ifelse` statement above, if this condition is true, we replace `weight` by 70.2; if it is false, we leave weight as is.

2.3 Overlaying a Normal curve on a histogram

It can be useful to produce a histogram with an overlayed Normal curve to assess whether our sample appears approximately Normally distributed. We can do this by plotting a histogram using the `hist()` function. As we're overlaying a probability distribution, we request the histogram be plotted on a probability scale, rather than a frequency scale, using `probability=TRUE`.
We then request a curve be overlayed using the `curve()` function:

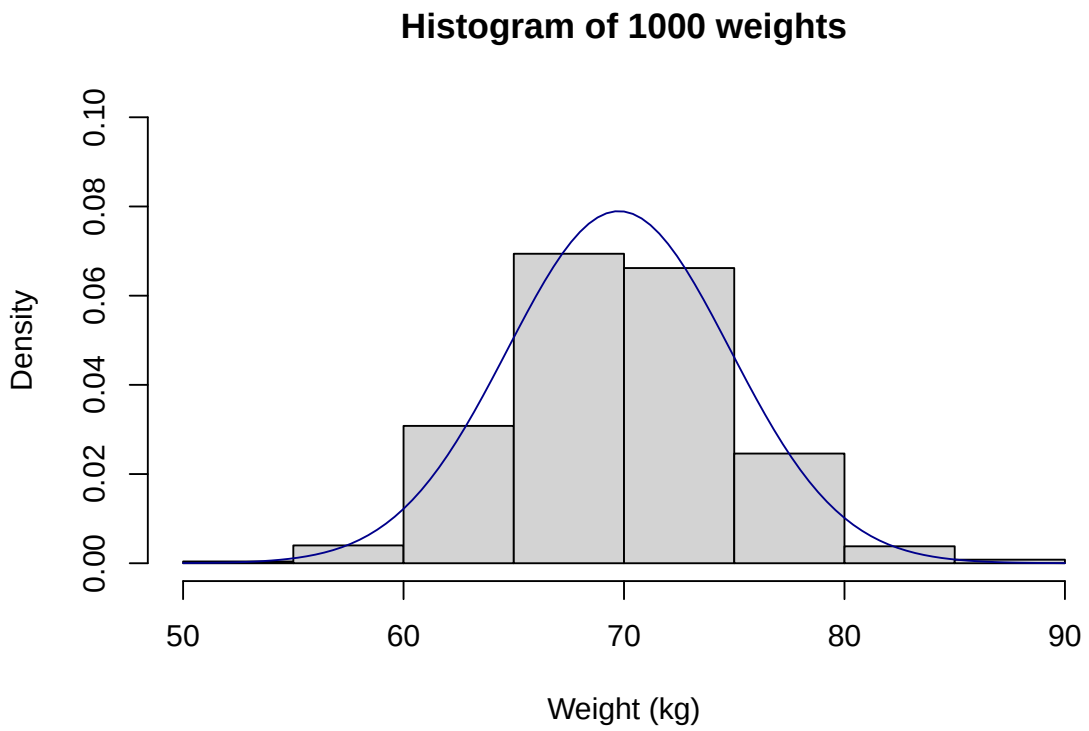
- the curve should be based on the Normal distribution (`dnorm`);
  - with a mean equal to the mean of the cleaned weight: `mean(sample$weight_clean))`;
  - and a standard deviation equal to the standard deviation of the cleaned weight: `sd(sample$weight_clean))`
- using a dark-blue colour;
- and added to the previous histogram (rather than plotting the curve by itself): `add=TRUE`

```
hist(sample$weight_clean,
     xlab="Weight (kg)",
     main="Histogram of 1000 weights",
     probability = TRUE)

curve(dnorm(x,
            mean=mean(sample$weight_clean),
            sd=sd(sample$weight_clean)),
      col="darkblue",
      add=TRUE)
```



Notice that the top of the curve is chopped off. We can plot the whole curve by extending the y-axis of the histogram to 0.1:

```
hist(sample$weight_clean,
     xlab="Weight (kg)",
     main="Histogram of 1000 weights",
     probability = TRUE,
     ylim=c(0,0.1))

curve(dnorm(x,
            mean=mean(sample$weight_clean),
            sd=sd(sample$weight_clean)),
      col="darkblue",
      add=TRUE)
```



2.4 Descriptive statistics for checking normality

All the descriptive statistics including *skewness* and *kurtosis* discussed in this module can be obtained using the `descriptives` function from the `jmv` package. In particular, skewness and kurtosis can be requested in addition to the default statistics by including: `skew=TRUE, kurt=TRUE`:

```
library(jmv)

descriptives(data=sample, vars=weight_clean, skew=TRUE, kurt=TRUE)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                            weight_clean
##
##    N                               1000
##    Missing                            0
##    Mean                         69.76450
##    Median                       69.80000
##    Standard deviation           5.052676
##    Minimum                      53.80000
##    Maximum                      85.80000
##    Skewness                   0.07360659
##    Std. error skewness        0.07734382
##    Kurtosis                   0.05418774
##    Std. error kurtosis         0.1545343
```

```
##
```

2.5 Importing Excel data into R

Another common type of file that data are stored in is a Microsoft Excel file (.xls or .xlsx). In this demonstration, we will import a selection of records from a large health survey, stored in the file `health-survey.xlsx`.
The health survey data contains 1140 records, comprising:

- sex: 1 = respondent identifies as male; 2 = respondent identifies as female
- height: height in meters
- weight: weight in kilograms

To import data from Microsoft Excel, we can use the `read_excel()` function in the `readxl` package.

```
library(readxl)

survey <- read_excel("data/examples/health-survey.xlsx")
summary(survey)
```

```
##       sex           height          weight
##  Min.   :1.00   Min.   :1.220   Min.   : 22.70
##  1st Qu.:1.00   1st Qu.:1.630   1st Qu.: 68.00
##  Median :2.00   Median :1.700   Median : 79.40
##  Mean   :1.55   Mean   :1.698   Mean   : 81.19
##  3rd Qu.:2.00   3rd Qu.:1.780   3rd Qu.: 90.70
##  Max.   :2.00   Max.   :2.010   Max.   :213.20
```

We can see that sex has been entered as a numeric variable. We should transform it into a factor so that we can assign labels to each category:

```
survey$sex <- factor(survey$sex, level=c(1,2), labels=c("Male", "Female"))

summary(survey$sex)
```

```
##   Male Female
##    513    627
```

We also note that height looks like it has been entered as meters, and weight as kilograms.

2.6 Generating new variables

Our health survey data contains information on height and weight. We often summarise body size using BMI: body mass index which is calculated as: $\frac{\text{weight (kg)}}{(\text{height (m)})^2}$
We can create a new column in our data frame in many ways, such as using the following approach:
`dataframe$new_column <- <formula>`
For example:

```
survey$bmi <- survey$weight / (survey$height^2)
```

We should check the construction of the new variable by examining some records. The `head()` and `tail()` functions list the first and last 6 records in any dataset. We can also examine a histogram and boxplot:

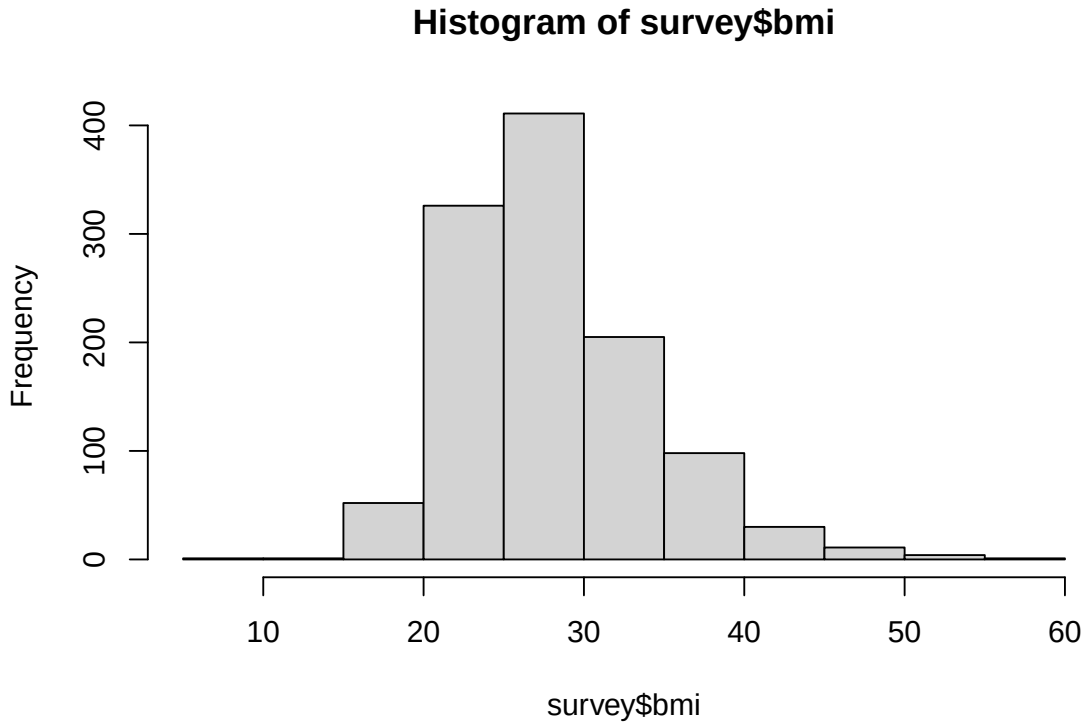
```
head(survey)
```

```
## # A tibble: 6 x 4
##   sex    height weight   bmi
##   <fct>   <dbl>  <dbl> <dbl>
## 1 Male     1.63   81.7  30.8
## 2 Male     1.63   68    25.6
## 3 Male     1.85   97.1  28.4
## 4 Male     1.78   89.8  28.3
## 5 Male     1.73   70.3  23.5
## 6 Female   1.57   85.7  34.8
```

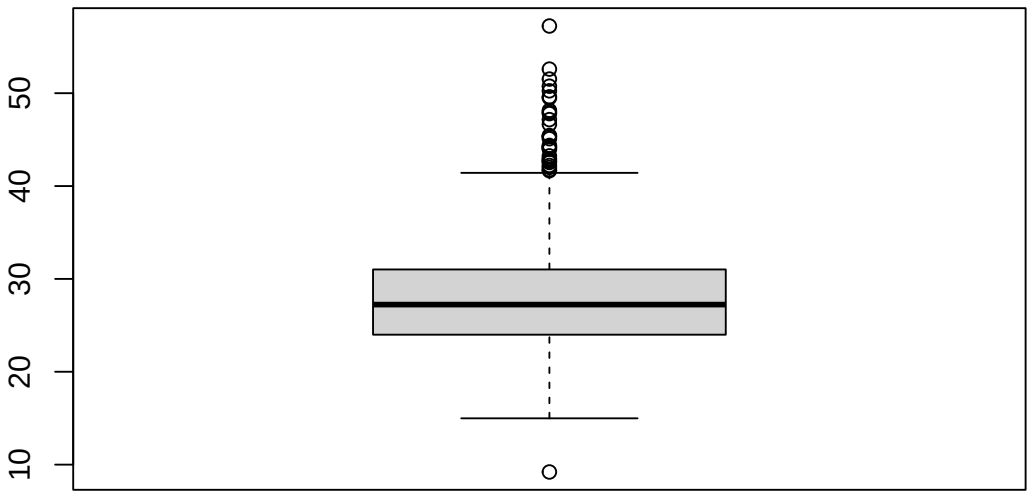
```
tail(survey)
```

```
## # A tibble: 6 x 4
##   sex    height weight   bmi
##   <fct>   <dbl>  <dbl> <dbl>
## 1 Female   1.65   95.7  35.2
## 2 Male     1.8    79.4  24.5
## 3 Female   1.73   83    27.7
## 4 Female   1.57   61.2  24.8
## 5 Male     1.7    73    25.3
## 6 Female   1.55   91.2  38.0
```

```
hist(survey$bmi)
```



```
boxplot(survey$bmi)
```



In the general population, BMI ranges between about 15 to 30. It appears that BMI has been correctly generated in this example. We should investigate the very low and some of the very high values of BMI, but this will be left for another time.

2.7 Summarising data by another variable

We will often want to calculate the same summary statistics by another variable. For example, we might want to calculate summary statistics for BMI for males and females separately. We can do this in in the `descriptives` function by defining sex as a `splitBy` variable:

```
library(jmv)
descriptives(data=survey, vars=bmi, splitBy = sex)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                          sex       bmi
##
##    N                     Male          513
##                          Female        627
##    Missing               Male            0
##                          Female          0
##    Mean                  Male      28.29561
##                          Female    27.81434
```

```
##    Median                Male      27.39592
##                          Female    26.66667
##    Standard deviation    Male      5.204975
##                          Female    6.380523
##    Minimum               Male      16.47519
##                          Female    9.209299
##    Maximum               Male      57.23644
##                          Female    52.59516
##
```

2.8 Summarising a single column of data

In Module 1, we started with a very simple analysis: reading in six ages, and them using `summary()` to calculate descriptive statistics. We then went on to use the `decriptives()` function in the `jmv` package as more flexible way of calculating descriptive statistics. Let's revisit this analysis:

```
# Author: Timothy Dobbins
# Date: 5 April 2022
# Purpose: My first R script
library(jmv)

age <- c(20, 25, 23, 29, 21, 27)

# Use "summary" to obtain descriptive statistics
summary(age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   20.00   21.50   24.00   24.17   26.50   29.00
```

```
# Use "descriptives" to obtain descriptive statistics
descriptives(age)
```

```
## Error: Argument 'data' must be a data frame
```

The `summary()` function has worked correctly, but the `descriptives()` function has given an error: `Error: Argument 'data' must be a data frame`. What on earth is going on here?
The error gives us a clue here - the `descriptives()` function requires a data frame for analysis. We have provided the object `age`: a **vector**. As we saw in Section 1.6.3, a vector is a single column of data, while a data frame is a collection of columns.
In order to summarise a vector using the `descriptives()` function, we must first convert the vector into a data frame using `as.data.frame()`. For example:

```
# Author: Timothy Dobbins
# Date: 5 April 2022
# Purpose: My first R script
library(jmv)

age <- c(20, 25, 23, 29, 21, 27)

# Use "summary" to obtain descriptive statistics
summary(age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   20.00   21.50   24.00   24.17   26.50   29.00
```

```
# Create a new data frame from the vector age:

age_df <- as.data.frame(age)

# Use "descriptives" to obtain descriptive statistics for age_df
descriptives(age_df)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                          age
##
##    N                           6
##    Missing                     0
##    Mean                24.16667
##    Median              24.00000
##    Standard deviation   3.488075
##    Minimum             20.00000
##    Maximum             29.00000
##
```

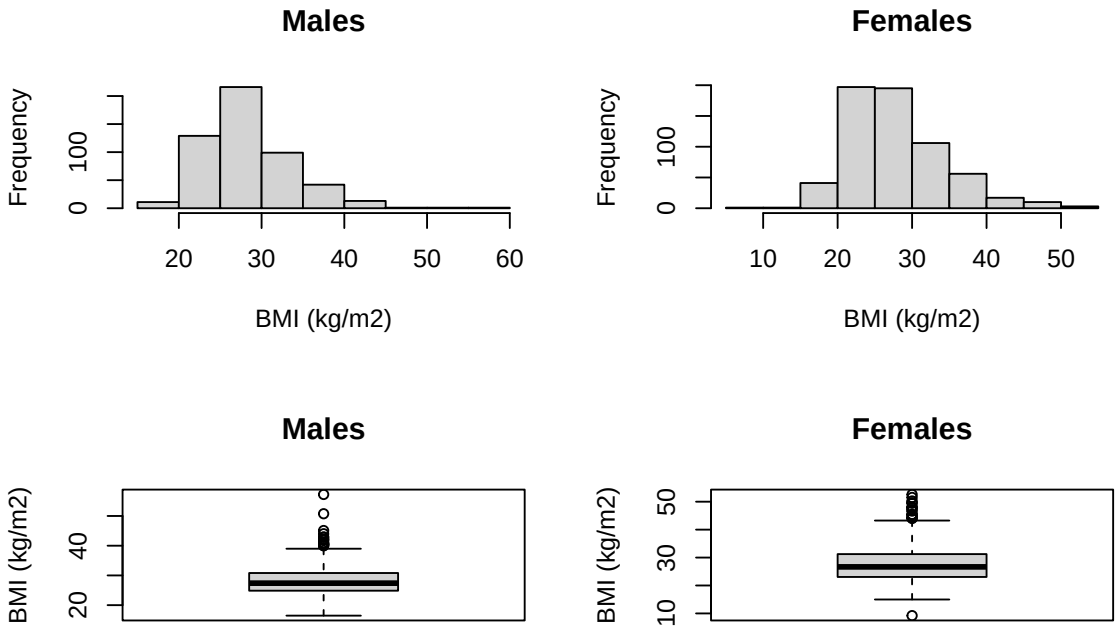
2.9 Plotting data by another variable

Unfortunately, it is not straight-forward to create separate plots for every level of another variable. We will demonstrate by plotting BMI by sex using our health survey data.
The following steps are not the most efficient way of doing this, but are easy to follow and understand. We first begin by creating two new data frames, for males and females separately, using the `subset()` function:

```
survey_males <- subset(survey, sex=="Male")
survey_females <- subset(survey, sex=="Female")
```

Note that we use the **label** for sex, not the underlying numeric value, as sex is a **factor**.
We can now create hisotgrams and boxplots of BMI for males and females separately. To place the graphs next to each other in a single figure, we can use the `par` function, which sets the *graphics parameters*. Essentially, we want to tell R to split a plot window into a matrix with *nr* rows and *nc* columns, and we fill the cells by rows (`mfrow`) or columns (`mfcols`).
For example, to plot four figures in a single plot, filled by rows, we use `par(mfrow=c(2,2))`.
When we are done plotting multiple graphs, we can reset the graphics parameters by submitting `par(mfrow=c(1,1))`.

```
# Set the graphics parameters to plot 2 rows and 2 columns:
par(mfrow=c(2,2))

# Specify each plot separately
hist(survey_males$bmi, xlab="BMI (kg/m2)", main="Males")
hist(survey_females$bmi, xlab="BMI (kg/m2)", main="Females")

boxplot(survey_males$bmi, ylab="BMI (kg/m2)", main="Males")
boxplot(survey_females$bmi, ylab="BMI (kg/m2)", main="Females")
```




```
# Reset graphics parameters
par(mfrow=c(1,1))
```

2.10 Recoding data

One task that is common in statistical computing is to recode variables. For example, we might want to group some categories of a categorical variable, or to present a continuous variable in a categorical way.
In this example, we can recode BMI into the following categories as suggested by the World Health Organisation [footnote]:

- Underweight: BMI < 18.5
- Normal weight: 18.5 $\leq$ BMI < 25
- Pre-obesity: 25 $\leq$ BMI < 30
- Obesity Class I: 30 $\leq$ BMI < 35
- Obesity Class II: 35 $\leq$ BMI < 40
- Obesity Class III: BMI $\geq$ 40

The quickest way to recode a continuous variable into categories is to use the `cut` command which takes a continuous variable, and "cuts" it into groups based on the specified "cutpoints"

```
survey$bmi_cat <- cut(survey$bmi,
                      breaks = c(0, 18.5, 25, 30, 35, 40, 100))
```

Notice that lower (BMI=0) and upper (BMI=100) bounds have been specified, as both a lower and upper limit must be defined for each group.
If we examine the new `bmi_cat` variable:

```
summary(survey$bmi_cat)
```

```
##  (0,18.5] (18.5,25]   (25,30]   (30,35]   (35,40]  (40,100]
##        18       362       411       205        97        47
```

we see that each group has been labelled (a, b]. This notation is equivalent to: greater than a, and less than or equal to b. The `cut` function excludes the lower limit, but includes the upper limit. Our BMI ranges have been defined to include the lower limit, and exclude the upper limit (for example, greater than or equal to 30 and less than 35).
We can specify this recoding using the `right=FALSE` option:

```
survey$bmi_cat <- cut(survey$bmi,
                      breaks = c(0, 18.5, 25, 30, 35, 40, 100),
                      right=FALSE)

summary(survey$bmi_cat)
```

```
##  [0,18.5) [18.5,25)   [25,30)   [30,35)   [35,40)  [40,100)
##        18       362       411       201       101        47
```

Finally, we can specify labels for the groups using the `labels` option:

```
survey$bmi_cat <- cut(survey$bmi,
                      breaks = c(0, 18.5, 25, 30, 35, 40, 100),
                      right=FALSE,
                      labels = c("Underweight", "Normal", "Pre-obesity",
                                 "Obesity Class I", "Obesity Class II",
                                 "Obesity Class III"))

summary(survey$bmi_cat)
```

```
##       Underweight            Normal       Pre-obesity   Obesity Class I
##                18               362               411               201
##  Obesity Class II Obesity Class III
##               101                47
```

2.11 Computing binomial probabilities using R

There are two R functions that we can use to calculate probabilities based on the binomial distribution: `dbinom` and `pbinom`:

- `dbinom(x, size, prob)` gives the probability of obtaining `x` successes from `size` trials when the probability of a success on one trial is `prob`;
- `pbinom(q, size, prob)` gives the probability of obtaining `q` **or fewer** successes from `size` trials when the probability of a success on one trial is `prob`;
- `pbinom(q, size, prob, lower.tail=FALSE)` gives the probability of obtaining **more than** `q`successes from `size` trials when the probability of a success on one trial is `prob`.

To do the computation for part (a) in Worked Example 2.1, we will use the `dbinom` function with:

- *x* is the number of successes, here, the number of smokers (i.e. k=3);
- *size* is the number of trials (i.e. n=6);
- and *prob* is probability of drawing a smoker from the population, which is 19.8% (i.e. p=0.198).

Replace each of these with the appropriate number into the formula:

```
dbinom(x=3, size=6, prob=0.198)
```

```
## [1] 0.08008454
```

To calculate the upper tail of probability in part (b), we use the `pbinom(lower.tail=FALSE)` function. Note that the `pbinom(lower.tail=FALSE)` function **does not include q**, so to obtain 4 or more successes, we need to enter `q=3`:

```
pbinom(q=3, size=6, prob=0.198, lower.tail=FALSE)
```

```
## [1] 0.01635325
```

For the lower tail for part (c), we use the `pbinom` function:

```
pbinom(q=2, size=6, prob=0.198)
```

```
## [1] 0.9035622
```

2.12 Computing probabilities from a Normal distribution

We can use the `pnorm` function to calculate probabilities from a Normal distribution:

- `pnorm(q, mean, sd)` calculates the probability of observing a value of `q` or less, from a Normal distribution with a mean of `mean` and a standard deviation of `sd`. Note that if `mean` and `sd` are not entered, they are assumed to be 0 and 1 respectively (i.e. a standard normal distribution.)
- `pnorm(q, mean, sd, lower.tail=FALSE)` calculates the probability of observing a value of more than `q`, from a Normal distribution with a mean of `mean` and a standard deviation of `sd`.

To obtain the probability of obtaining 0.5 or greater from a standard normal distribution:

```
pnorm(0.5, lower.tail=FALSE)
```

```
## [1] 0.3085375
```

To calculate the worked example: Assume that the mean diastolic blood pressure for men is 77.9 mmHg, with a standard deviation of 11. What is the probability that a man selected at random will have high blood pressure (i.e. diastolic blood pressure greater than or equal to 90)?

```
pnorm(90, mean=77.9, sd=11, lower.tail=FALSE)
```

```
## [1] 0.1356661
```

Module 3

Precision: R notes

3.1 Calculating a 95% confidence interval of a mean

3.1.1 Individual data

To demonstrate the computation of the 95% confidence interval of a mean we have used data from `Example_1.3.rds` which contains the weights of 30 students:

```
library(jmv)

students <- readRDS("data/examples/Example_1.3.rds")

summary(students)
```

```
##      weight         gender
##  Min.   :60.00   Male  :16
##  1st Qu.:67.50   Female:14
##  Median :70.00
##  Mean   :70.00
##  3rd Qu.:74.38
##  Max.   :80.00
```

The mean and its 95% confidence interval can be obtained many ways in R. We will use the `t.test()` function installed in R to calculate the confidence interval:

```
t.test(students$weight)
```

```
##
##  One Sample t-test
##
## data:  students$weight
## t = 76.029, df = 29, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  68.11694 71.88306
## sample estimates:
## mean of x
##        70
```

The output of the `t.test()` function gives us the sample mean (70.0 kg) as well as the 95% confidence interval around the mean: 68.1 to 71.9 kg.

Note: the `descriptives()` function within the `jmv` package also calculates a 95% confidence interval around the mean. **It is recommended not to use this function** as it currently (as of June 2022) uses a *z* value to calculate the confidence interval, rather than a *t* value.

3.1.2 Summarised data

For Worked Example 3.2 where we are given the sample mean, sample standard deviation and sample size. R does not have a built-in function to calculate a confidence interval from summarised data, but we can write our own.
Note: writing your own functions is beyond the scope of this course. You should copy and paste the code provided to do this.

```
### Copy this section
ci_mean <- function(n, mean, sd, width=0.95, digits=3){
  lcl <- mean - qt(p=(1 - (1-width)/2), df=n-1) * sd/sqrt(n)
  ucl <- mean + qt(p=(1 - (1-width)/2), df=n-1) * sd/sqrt(n)

  print(paste0(width*100, "%", " CI: ", format(round(lcl, digits=digits), nsmall = digits),
               " to ", format(round(ucl, digits=digits), nsmall = digits) ))

}
### End of copy

ci_mean(n=30, mean=70, sd=6, width=0.95)
```

```
## [1] "95% CI: 67.760 to 72.240"
```

```
ci_mean(n=30, mean=70, sd=6, width=0.99)
```

```
## [1] "99% CI: 66.981 to 73.019"
```

Module 4

Hypothesis testing

4.1 One sample t-test

We will use data from `Example_4.1.rds` to demonstrate how a one-sample t-test is conducted in R.

```
bloodpressure <- readRDS("data/examples/Example_4.1.rds")

summary(bloodpressure)
```

```
##       dbp
##  Min.   : 24.00
##  1st Qu.: 64.00
##  Median : 72.00
##  Mean   : 72.41
##  3rd Qu.: 80.00
##  Max.   :122.00
##  NA's   :35
```

To test whether the mean diastolic blood pressure of the population from which the sample was drawn is equal to 71, we can use the t.test command:

```
t.test(bloodpressure$dbp, mu=71)
```

```
##
##  One Sample t-test
##
## data:  bloodpressure$dbp
## t = 3.0725, df = 732, p-value = 0.002202
## alternative hypothesis: true mean is not equal to 71
## 95 percent confidence interval:
##  71.50732 73.30305
## sample estimates:
## mean of x
##  72.40518
```

The output gives a test statistic, degrees of freedom and a P values from the two-sided test. The mean of the sample is provided, as well as the 95% confidence interval.

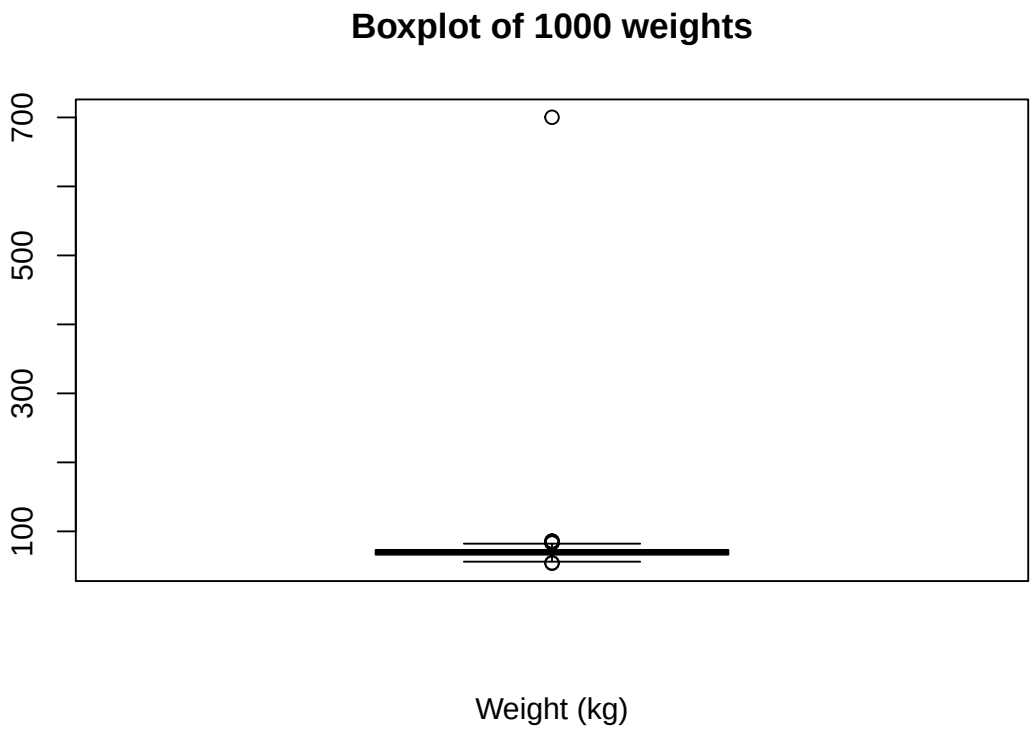
Module 5

Comparing two means

5.1 Setting an observation to missing

Setting an incorrect observation to missing is straightforward in Stata by using the Data Editor. While RStudio allows browsing a data set as a spreadsheet, it will not let a user replace an observation with a missing value: this should be done using code.
A missing value in R is denoted `NA`, and this is consistent for any variable type: continuous, string (i.e. character) and even a factor.
Recall the `weights` data used in Module 2. In viewing a boxplot of `weight`, we saw an obvious outlier of 700.2kg for ID 58:

```
library(jmv)

sample <- read.csv("data/examples/Weight_s2.csv")

boxplot(sample$weight, xlab="Weight (kg)", main="Boxplot of 1000 weights")
```




```
subset(sample, weight>200)
```

```
##    id weight
## 58 58  700.2
```

We previously set this value to 70.2kg using an `ifelse()` command. Here, let's create a new, cleaned weight variable, and set the incorrect value to missing:

```
sample$weight_clean = ifelse(sample$weight==700.2, NA, sample$weight)
```

Our code will create a new column (called weight_clean) in the `sample` dataframe. We will test whether weight is equal to 700.2; if this is true, we will assign weight_clean to be NA (i.e. missing), otherwise weight_clean will equal the value of weight.
Let's view the data from ID 58, and summarise the cleaned weight variable using `descriptives()` and a boxplot:

```
subset(sample, sample$id==58)
```

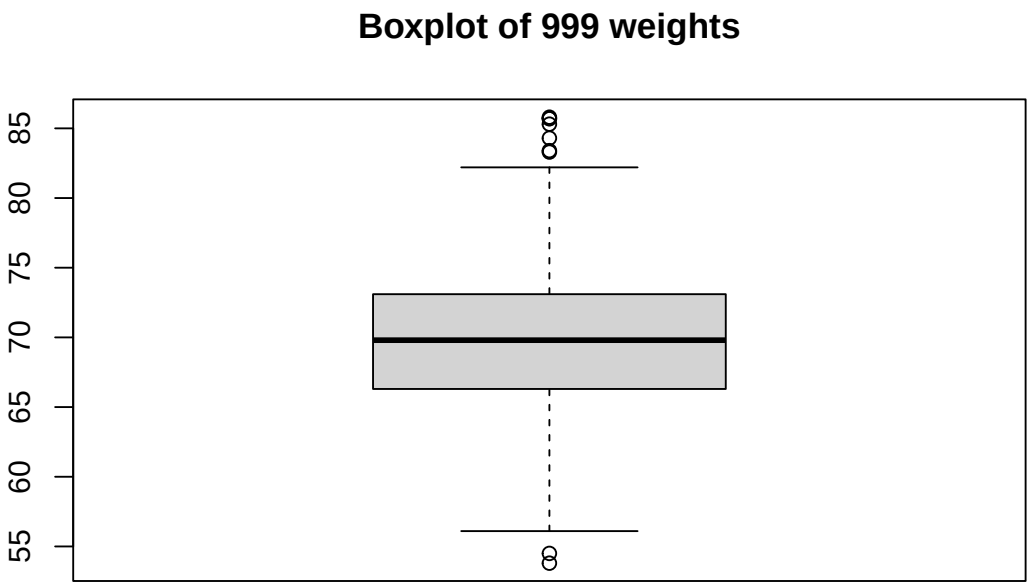
```
##    id weight weight_clean
## 58 58  700.2           NA
```

```
descriptives(data=sample, vars=weight_clean)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
```

```
##
##                          weight_clean
##
##    N                              999
##    Missing                          1
##    Mean                       69.76406
##    Median                     69.80000
##    Standard deviation         5.055188
##    Minimum                    53.80000
##    Maximum                    85.80000
##
```

```
boxplot(sample$weight_clean, xlab="Weight (kg)", main="Boxplot of 999 weights",
        sub="(Excluding 1 observation of 700.2kg)")
```



5.2 Checking data for the independent samples t-test

5.2.1 Producing histograms and boxplots by a second variable

We have seen how to create histograms and boxplots separated by a second variable in Module 2 (Section 2.9). We will demonstrate using the birthweight data in `Example_5.1.rds`.

```
library(jmv)

bwt <- readRDS("data/examples/Example_5.1.rds")

summary(bwt)
```

```
##     gender    birthweight
##  Female:56   Min.   :2.750
##  Male  :44   1st Qu.:3.257
##              Median :3.450
##              Mean   :3.514
##              3rd Qu.:3.772
##              Max.   :4.250
```

```
summary(bwt$gender)
```

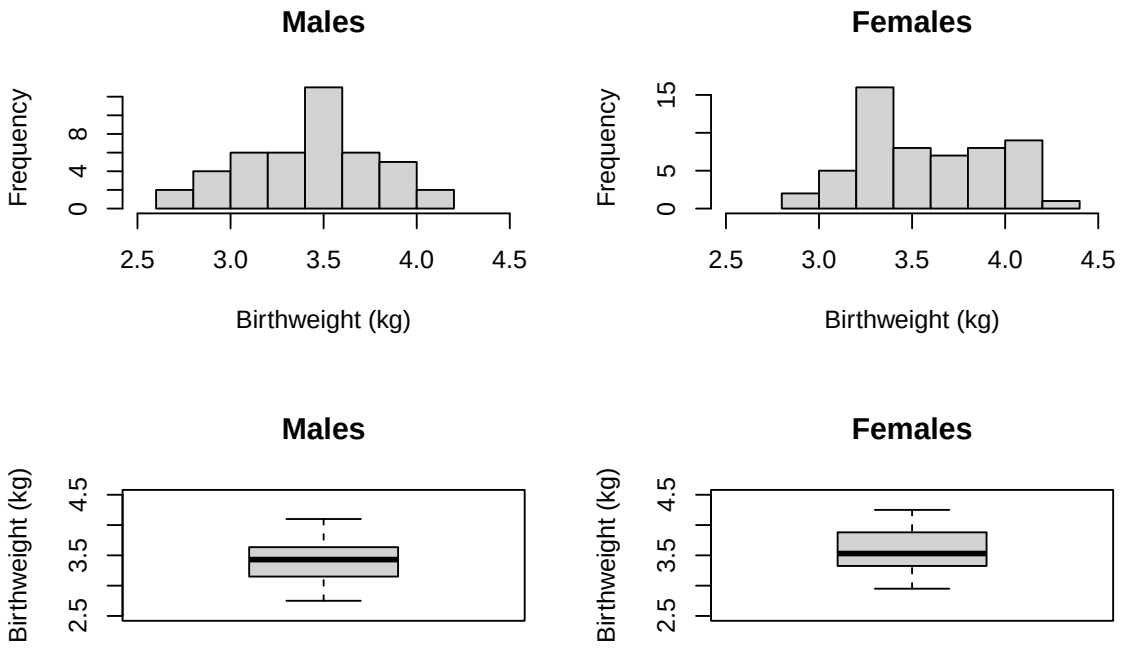
```
## Female   Male
##     56     44
```

We can create subsets of the birthweight data, subsetted for males and females separately. Note here that `gender` is a factor, so we need to select based on the factor labels, not the underlying numeric code.

```
bwt_m <- subset(bwt, bwt$gender=="Male")
bwt_f <- subset(bwt, bwt$gender=="Female")
```

We can now create histograms and boxplots for males and females separately, in the usual way, using the `par` function to set the graphics parameters to display graphs in a 2-by-2 grid:

```
par(mfrow=c(2,2))
hist(bwt_m$birthweight, xlim=c(2.5, 4.5), xlab="Birthweight (kg)", main="Males")
hist(bwt_f$birthweight, xlim=c(2.5, 4.5), xlab="Birthweight (kg)", main="Females")

boxplot(bwt_m$birthweight, ylim=c(2.5, 4.5), ylab="Birthweight (kg)", main="Males")
boxplot(bwt_f$birthweight, ylim=c(2.5, 4.5), ylab="Birthweight (kg)", main="Females")
```



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

When we are done plotting multiple graphs, we can reset the plot window by submitting `par(mfrow=c(1,1))`.

5.2.2 Producing split summary statistics

The `descriptives` function within the `jmv` function allows summary statistics to be calculated within subgroups using the `splitBy` argument:

```
descriptives(data=bwt, vars=birthweight, splitBy=gender)
```

```
##
##  DESCRIPTIVES
##
##  Descriptives
##
##                          gender    birthweight
##
##    N                     Female             56
##                          Male               44
##    Missing               Female              0
##                          Male                0
##    Mean                  Female       3.587411
##                          Male         3.421364
##    Median                Female       3.530000
##                          Male         3.430000
##    Standard deviation    Female      0.3629788
##                          Male        0.3536165
##    Minimum               Female       2.950000
##                          Male         2.750000
##    Maximum               Female       4.250000
##                          Male         4.100000
##
```

5.3 Independent samples t-test

We can use the `ttestIS()` (t-test, independent samples) function from the `jmv` package to perform the independent samples t-test. We include the `meanDiff=TRUE` and `ci=TRUE` options to obtain the difference in means, with its 95% confidence interval. We can request a Welch's test (which does not assume equal variances) by the `welchs=TRUE` option:

```
ttestIS(data=bwt, vars=birthweight, group=gender, meanDiff=TRUE, ci=TRUE)
```

```
##
##  INDEPENDENT SAMPLES T-TEST
##
##  Independent Samples T-Test
##
##                      Statistic   df       p        Mean difference   SE difference   Lower
##
##   birthweight   Student's t    2.296556   98.00000   0.0237731       0.1660471     0.07230265
##
```

```
ttestIS(data=bwt, vars=birthweight, group=gender, meanDiff=TRUE, ci=TRUE, welchs=TRUE)
```

```
##
##  INDEPENDENT SAMPLES T-TEST
##
##  Independent Samples T-Test
##
##                      Statistic   df       p        Mean difference   SE difference   Lower      Upp
##
##   birthweight   Student's t    2.296556   98.00000   0.0237731       0.1660471     0.07230265   0.022
##              Welch's t     2.303840   93.54377   0.0234458       0.1660471     0.07207403   0.0229332
##
```

There is no built-in function to calculate an independent t-test from summarised data, nor is there a function within `jmv`. We can use the `tsum.test()` function within the `BSDA` package, with the following syntax:

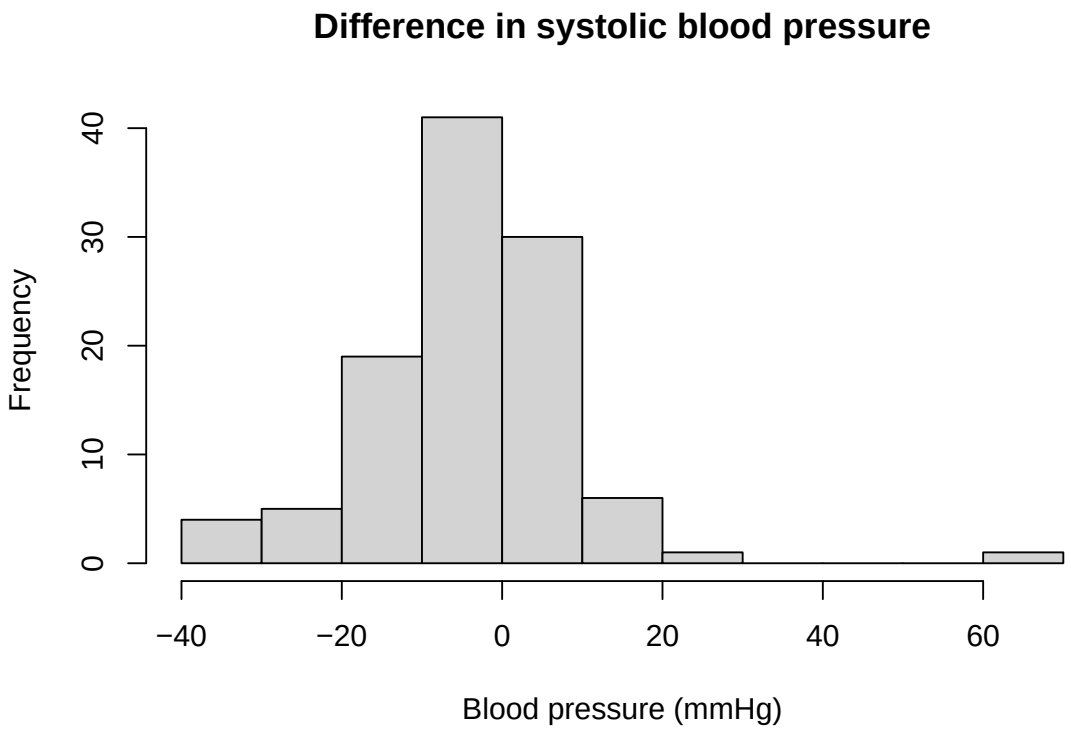
```
tsum.test(mean.x=, s.x=, n.x=,
          mean.y=, s.y=, n.y=,
          mu=0, alternative="two.sided", var.equal = TRUE)
```

Here we specify the mean, standard deviation and sample size for the first group (on the first line) and the second group (on the second line). We can relax the assumption of equal variances using `var.equal=FALSE`.

5.4 Checking the assumptions for a Paired t-test

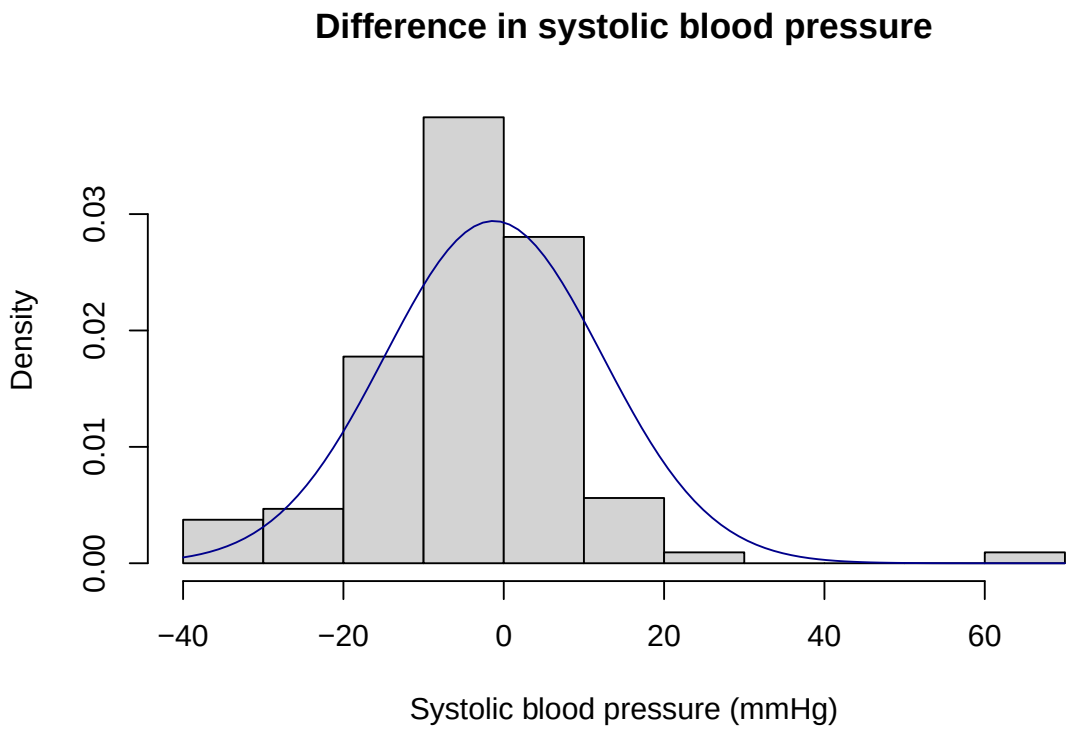
Before performing a paired t-test, you must check that the assumptions for the test have been met. Using the dataset `Example_5.2.rds` to show that the difference between the pair of measurements between the sites is normally distributed, we first need to compute a new variable of the differences and examine its histogram.

```
sbp <- readRDS("data/examples/Example_5.2.rds")
sbp$diff = sbp$sbp_dp - sbp$sbp_tp
hist(sbp$diff, xlab="Blood pressure (mmHg)", main="Difference in systolic blood pressure")
```



We might want to plot a Normal curve over this distribution, as we did in Module 2:

```
hist(sbp$diff,
     xlab="Systolic blood pressure (mmHg)",
     main="Difference in systolic blood pressure",
     probability = TRUE)

curve(dnorm(x,
            mean=mean(sbp$diff),
            sd=sd(sbp$diff)),
      col="darkblue",
      add=TRUE)
```



While there is a large difference in blood pressure (around 60 mmHg) that warrents further checking, the curve is roughly symmetric with an approximately Normal distribution.

5.5 Paired t-Test

To perform a paired t-test we will use the dataset `Example_5.2.rds`. We can perform a paired t-test using the `ttestPS()` function within the `jmv` package, where we defined the paired observations as: 'pairs=list(list(i1 = 'variable1', i2 = 'variable2'))

```
ttestPS(data=sbp, pairs=list(list(i1 = 'sbp_dp', i2 = 'sbp_tp')), meanDiff=TRUE, ci=TRUE)
```

```
##
##  PAIRED SAMPLES T-TEST
##
##  Paired Samples T-Test
##
##                          statistic    df       p        Mean difference   SE difference   Lower
##
##   sbp_dp   sbp_tp   Student's t   -0.9621117   106.0000   0.3381832       -1.261682       1.311368   -
##
```

The syntax of the ttestPS function is a little cumbersome. The `t.test` function can be used as an alternative:

```
t.test(sbp$sbp_dp, sbp$sbp_tp, paired=TRUE)
```

```
##
##  Paired t-test
##
```

```
## data:  sbp$sbp_dp and sbp$sbp_tp
## t = -0.96211, df = 106, p-value = 0.3382
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -3.861596  1.338232
## sample estimates:
## mean difference
##       -1.261682
```

Module 6

Summary statistics for binary data

6.1 95% confidence intervals for proportions

We can use the `BinomCI(x=, n=, method=)` function within the `DescTools` package to compute 95% confidence intervals for proportions. Here we specify `x`: the number of successes, `n`: the sample size, and optionally, the `method` (which defaults to Wilson's method).

```
library(DescTools)

BinomCI(x=47, n=215, method='wald')
```

```
##            est    lwr.ci    upr.ci
## [1,] 0.2186047 0.1633595 0.2738498
```

```
BinomCI(x=47, n=215, method='wilson')
```

```
##            est    lwr.ci    upr.ci
## [1,] 0.2186047 0.1685637 0.2785246
```

6.2 Significance test for single proportion

We can use the `binom.test` function to perform a significance test for a single proportion: `binom.test(x=, n=, p=)`. Here we specify `x`: the number of successes, `n`: the sample size, and `p`: the hypothesised proportion (which defaults to 0.5 if nothing is entered).

```
binom.test(x=54, n=300, p=0.2)
```

```
##
##  Exact binomial test
##
## data:  54 and 300
## number of successes = 54, number of trials = 300, p-value = 0.4273
## alternative hypothesis: true probability of success is not equal to 0.2
## 95 percent confidence interval:
##  0.1382104 0.2282394
## sample estimates:
## probability of success
##                   0.18
```

Note that the `binom.test` function also produces a 95% confidence interval around the estimated proportion. This confidence interval is based on the inferior Wald method: *the confidence interval derived from the Wilson method is preferred*.

6.3 Computing a relative risk and its 95% confidence interval

We will use Worked Example 6.4 to demonstrate calculating a relative risk and its 95% CI:

```
library(jmv)

drug <- readRDS("data/examples/Example_6.4.rds")

summary(drug)
```

```
##      group       side_effect
##  Placebo:50   No nausea:81
##  Active :50   Nausea   :19
```

By using the `head()` function to view the first six lines of data, we see that both `group` and `side_effect` have been entered as factors. Notice the order in which the factor levels are presented: `group` has the `Placebo` level defined as the first level, and the `Active` level defined as the second; `side_effect` has `No nausea` defined as the first level, and the `Nausea` level defined as the second.
We will use `jmv` to calculate relative risks, odds ratios and risk differences. To calculate these estimates correctly, **we must define the positive exposure and positive outcome to be the first level of a factor**. When defining an exposure for example, we should define the active treatment or the positive exposure as the first category. When defining an outcome, we should define the category of interest (e.g. disease, or side effect) as the first category.
In this example, we will define `Active` as the first level in the `group` factor, and `Nausea` to be the first level of the `side_effect` factor.
We can do this using the `relevel()` function, which re-orders the levels of a factor so that the level specified is defined as the first level, and the others are moved down:

```
# Define "Active" as the first level of group:
drug$group <- relevel(drug$group, ref="Active")


# Define "Nausea" as the first level of side_effect:
drug$side_effect <- relevel(drug$side_effect, ref="Nausea")
```

Upon re-leveling the factors, we can check that the levels of interest have been defined as the first levels:

```
summary(drug)
```

```
##      group       side_effect
##  Active :50   Nausea   :19
##  Placebo:50   No nausea:81
```

To construct the 2-by-2 table and calculate a relative risk, we use the `contTables()` function in `jmv`. We request the row-percents using `pcRow = TRUE` and the relative risk and confidence interval using `relRisk = TRUE`:

```
contTables(data=drug,
           rows=group, cols=side_effect,
           pcRow=TRUE, relRisk = TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    group                       Nausea       No nausea    Total
##
##    Active     Observed             15           35          50
##               % within row    30.00000     70.00000    100.00000
##
##    Placebo    Observed              4           46          50
##               % within row     8.00000     92.00000    100.00000
##
##    Total      Observed             19           81         100
##               % within row    19.00000     81.00000    100.00000
##
##
##
##  x² Tests
##
##           Value       df    p
##
##    x²     7.862248     1    0.0050478
##    N          100
##
##
##
##  Comparative Measures
##
##                      Value         Lower       Upper
##
##    Relative risk     3.750000      1.337540    10.51370
##
##      Rows compared
```

If you only have the cross-tabulated data (i.e. aggregated), you will need to enter your data into a new data frame. For example, to recreate the above analyses, we can re-write the 2-by-2 table as follows:

Group	side_effect	Number
Active	Nausea	15
Active	No nausea	35
Placebo	Nausea	4
Placebo	No nausea	46

We can enter these data in a dataframe, comprising three vectors, as follows:

```
drug_aggregated <- data.frame(
  group = c("Active", "Active", "Placebo", "Placebo"),
  side_effect = c("Nausea", "No nausea", "Nausea", "No nausea"),
  n = c(15, 35, 4, 46)
)
```

We need to define group and side_effect as factors. Here we must define the `levels` **in the order we want the categories to appear in the table**. Note that as group and side_effect are entered as text

variables, we can omit `labels` command when defining the factors, and the factor will be labelled using the text entry:

```
drug_aggregated$group <- factor(drug_aggregated$group, levels=c("Active", "Placebo"))
drug_aggregated$side_effect <- factor(drug_aggregated$side_effect, levels=c("Nausea", "No nausea"))
```

We can calculate the relative risk using the summarised data in the same was done previously. However, we need to include the number of observations in each cell using the `counts` command:

```
contTables(data=drug_aggregated,
           rows=group, cols=side_effect, count=n,
           pcRow=TRUE, relRisk = TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    group                      Nausea       No nausea    Total
##
##    Active     Observed              15           35           50
##               % within row     30.00000     70.00000    100.00000
##
##    Placebo    Observed               4           46           50
##               % within row      8.00000     92.00000    100.00000
##
##    Total      Observed              19           81          100
##               % within row     19.00000     81.00000    100.00000
##
##
##
##  x² Tests
##
##          Value       df    p
##
##    x²    7.862248     1    0.0050478
##    N          100
##
##
##
##  Comparative Measures
##
##                     Value         Lower       Upper
##
##    Relative risk    3.750000      1.337540    10.51370
##
##      Rows compared
```

6.4 Computing a difference in proportions and its 95% confidence interval

We can use the `contTables` function to obtain a difference in proportions and its 95% CI, by specifying `diffProp=TRUE`:

```
contTables(data=drug,
           rows=group, cols=side_effect,
           pcRow=TRUE, diffProp=TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    group                       Nausea       No nausea    Total
##
##    Active     Observed              15           35           50
##               % within row     30.00000     70.00000    100.00000
##
##    Placebo    Observed               4           46           50
##               % within row      8.00000     92.00000    100.00000
##
##    Total      Observed              19           81          100
##               % within row     19.00000     81.00000    100.00000
##
##
##
##  x² Tests
##
##           Value       df    p
##
##    x²     7.862248     1    0.0050478
##    N           100
##
##
##
##  Comparative Measures
##
##                                    Value          Lower         Upper
##
##    Difference in 2 proportions    0.2200000      0.07238986    0.3676101
##
##      Rows compared
```

6.5 Computing an odds ratio and its 95% confidence interval

We can use the `contTables` function to obtain an odds ratio and its 95% CI, by specifying `odds=TRUE`. Here we will use the summarised HPV data from Module 6.

```
hpv <- data.frame(
  hpv = c("HPV +", "HPV +", "HPV -", "HPV -"),
  cancer = c("Case", "Control", "Case", "Control"),
  n = c(57, 14, 43, 186)
)

hpv$cancer <- factor(hpv$cancer, levels=c("Case", "Control"))
hpv$hpv <- factor(hpv$hpv, levels=c("HPV +", "HPV -"))

contTables(data=hpv,
```

```
          rows=hpv, cols=cancer, count=n,
          odds = TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    hpv      Case    Control    Total
##
##    HPV +      57         14       71
##    HPV -      43        186      229
##    Total     100        200      300
##
##
##
##  x² Tests
##
##           Value       df    p
##
##    x²    92.25660     1    < .0000001
##    N           300
##
##
##
##  Comparative Measures
##
##                  Value       Lower       Upper
##
##    Odds ratio    17.61130    8.992580    34.49041
##
```

Note that 95% confidence intervals for the odds ratio based on `jmv` may differ from those calculated by Stata. At this stage, the method used by `jmv` to calculate the confidence interval is not documented.

Module 7

Hypothesis testing for categorical data

7.1 Pearson's chi-squared test

7.1.1 Individual data

We will demonstrate how to use R to conduct a Pearson chi-squared test using Worked Example 7.1.

```
library(jmv)

nausea <- readRDS("data/examples/Example_7.1.rds")

head(nausea)
```

```
##     group side_effect
## 1 Placebo      Nausea
## 2 Placebo      Nausea
## 3 Placebo      Nausea
## 4 Placebo      Nausea
## 5 Placebo   No nausea
## 6 Placebo   No nausea
```

```
str(nausea$group)
```

```
##  Factor w/ 2 levels "Placebo","Active": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "label")= chr "Group"
```

```
str(nausea$side_effect)
```

```
##  Factor w/ 2 levels "No nausea","Nausea": 2 2 2 2 1 1 1 1 1 1 ...
##  - attr(*, "label")= chr "Side effect"
```

The columns `group` and `side_effect` have been entered as factors, with "Placebo" and "No nausea" as the first levels. We should use the `relevel()` command to re-order the factor levels.

```
nausea$group <- relevel(nausea$group, ref="Active")
nausea$side_effect <- relevel(nausea$side_effect, ref="Nausea")

str(nausea$group)
```

```
##  Factor w/ 2 levels "Active","Placebo": 2 2 2 2 2 2 2 2 2 2 ...
```

```
str(nausea$side_effect)
```

```
##  Factor w/ 2 levels "Nausea","No nausea": 1 1 1 1 2 2 2 2 2 2 ...
```

After confirming the factors are appropriately defined, we can construct our 2-by-2 table and view the expected frequencies.

```
contTables(data=nausea,
           rows=group, cols=side_effect,
           exp=TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    group                  Nausea       No nausea    Total
##
##    Active     Observed           15           35           50
##               Expected     9.500000     40.50000     50.00000
##
##    Placebo    Observed            4           46           50
##               Expected     9.500000     40.50000     50.00000
##
##    Total      Observed           19           81          100
##               Expected    19.000000     81.00000    100.00000
##
##
##
##  x² Tests
##
##          Value       df    p
##
##    x²    7.862248     1    0.0050478
##    N           100
##
```

After confirming that there are no cells with small expected frequencies, we can interpret the chi-square test. The last section reports the chi-squared test statistic which has a value of 7.86 with 1 degree of freedom and a P-value of 0.005.
If there are small values of expected frequencies, Fisher's exact test can be requested using `fisher = TRUE`:

```
contTables(data=nausea,
           rows=group, cols=side_effect,
           fisher = TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
```

```
##    group      Nausea    No nausea    Total
##
##    Active         15           35       50
##    Placebo         4           46       50
##    Total          19           81      100
##
##
##
##  x² Tests
##
##                            Value       df    p
##
##    x²                      7.862248     1    0.0050478
##    Fisher's exact test                      0.0094886
##    N                            100
##
```

7.1.2 Summarised data

When you only have the summarised date (for example, the cross-tabulated data), you need to enter the summarised data manually. As we did in Module 6, the 2-by-2 table can be entered as four lines of data:

```
drug_aggregated <- data.frame(
  group = c("Active", "Active", "Placebo", "Placebo"),
  side_effect = c("Nausea", "No nausea", "Nausea", "No nausea"),
  n = c(15, 35, 4, 46)
)
```

The `contTables()` function is used in the usual way, specifying `count=n`.

7.2 Chi-squared test for tables larger than 2-by-2

Use the data in `Example_7.2.rds`. We use similar steps as described above for a 2-by-2 table.

```
allergy <- readRDS("data/examples/Example_7.2.rds")

head(allergy)
```

```
##   id asthma hdmallergy catallergy infection    sex maternalasthma
## 1  1     No        Yes         No       Yes Female             No
## 2  2    Yes         No         No        No Female             No
## 3  3    Yes         No         No        No Female             No
## 4  4     No         No         No        No   Male             No
## 5  4    Yes        Yes        Yes        No Female             No
## 6  5    Yes        Yes        Yes        No Female             No
##   allergy_severity
## 1 Moderate allergy
## 2     Non-allergic
## 3     Non-allergic
## 4     Non-allergic
## 5 Moderate allergy
## 6 Moderate allergy
```

```
contTables(data=allergy,
           rows=allergy_severity, cols=sex,
           pcCol=TRUE)
```

```
##
##  CONTINGENCY TABLES
##
##  Contingency Tables
##
##    allergy_severity                      Female       Male         Total
##
##    Non-allergic        Observed               150          137          287
##                        % within column     61.98347     53.10078     57.40000
##
##    Slight allergy      Observed                50           70          120
##                        % within column     20.66116     27.13178     24.00000
##
##    Moderate allergy    Observed                27           32           59
##                        % within column     11.15702     12.40310     11.80000
##
##    Severe allergy      Observed                15           19           34
##                        % within column      6.19835      7.36434      6.80000
##
##    Total               Observed               242          258          500
##                        % within column    100.00000    100.00000    100.00000
##
##
##
##  x² Tests
##
##           Value       df    p
##
##    x²     4.308913     3    0.2299813
##    N          500
##
```

7.3 McNemar's test for paired proportions

To perform this test in R, we will use the dataset `Example_7.3.rds`.

```
drug <- readRDS("data/examples/Example_7.3.rds")

head(drug)
```

```
##   druga drugb
## 1   Yes   Yes
## 2   Yes   Yes
## 3   Yes   Yes
## 4   Yes   Yes
## 5   Yes   Yes
## 6   Yes   Yes
```

As usual, we should check that the variables being tabulated are factors, with the first level of the factor being the outcome of interest. We can use the `relevel()` function to re-order levels as necessary.

```
str(drug$druga)
```

```
##  Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
##  - attr(*, "label")= chr "Response to Drug A"
```

```
str(drug$drugb)
```

```
##  Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
##  - attr(*, "label")= chr "Response to Drug B"
```

```
drug$druga <- relevel(drug$druga, ref="Yes")
drug$drugb <- relevel(drug$drugb, ref="Yes")

str(drug$druga)
```

```
##  Factor w/ 2 levels "Yes","No": 1 1 1 1 1 1 1 1 1 1 ...
```

```
str(drug$drugb)
```

```
##  Factor w/ 2 levels "Yes","No": 1 1 1 1 1 1 1 1 1 1 ...
```

We can use the `contTablesPaired()` function within the `jmv` library to conduct McNemar's test of paired proportions:

```
contTablesPaired(data=drug, rows=druga, cols=drugb)
```

```
##
##  PAIRED SAMPLES CONTINGENCY TABLES
##
##  Contingency Tables
##
##    druga    Yes    No    Total
##
##    Yes       21    20       41
##    No        14     5       19
##    Total     35    25       60
##
##
##
##  McNemar Test
##
##           Value       df    p
##
##    x²    1.058824     1    0.3034837
##    N           60
##
```

Note that `contTablesPaired()` does not calculate an exact P-value.
To estimate the proportion in each of the paired samples, its difference, and the 95% confidence interval of the difference, we can use the `mcNemarDiff()` function which is stored in Microsoft Teams and here.

```
### Copied from Microsoft Teams
mcNemarDiff <- function(data, var1, var2, digits = 3) {
  if (!requireNamespace("epibasix", quietly = TRUE)) {
    stop("This function requires epibasix to be installed")
  }

  tab <- table(data[[var1]], data[[var2]])
  p1 <- (tab[1, 1] + tab[1, 2]) / sum(tab)
  p2 <- (tab[1, 1] + tab[2, 1]) / sum(tab)
  pd <- epibasix::mcNemar(tab)$rd
  pd.cil <- epibasix::mcNemar(tab)$rd.CIL
  pd.ciu <- epibasix::mcNemar(tab)$rd.CIU
  print(paste0(
    "Proportion 1: ",
    format(round(p1, digits = digits), nsmall = digits),
    "; Proportion 2: ", format(round(p2, digits = digits), nsmall = digits)
  ))
  print(paste0(
    "Difference in paired proportions: ",
    format(round(pd, digits = digits), nsmall = digits),
    "; 95% CI: ", format(round(pd.cil, digits = digits), nsmall = digits),
    " to ", format(round(pd.ciu, digits = digits), nsmall = digits)
  ))
}
### End copy

mcNemarDiff(data = drug, var1 = "druga", var2 = "drugb", digits = 2)
```

```
## [1] "Proportion 1: 0.68; Proportion 2: 0.58"
## [1] "Difference in paired proportions: 0.10; 95% CI: -0.11 to 0.31"
```

In this study of 60 participants, where each participant received both drugs, 41 (68%) responded to Drug A and 35 (58%) responded to Drug B. The difference in the proportions responding is estimated as 10% (95% CI -11% to 31%). There is no evidence that the response differed between the two drugs (McNemar's chi-squared = 1.06 with 1df, P=0.30).

Module 8

Correlation and simple linear regression

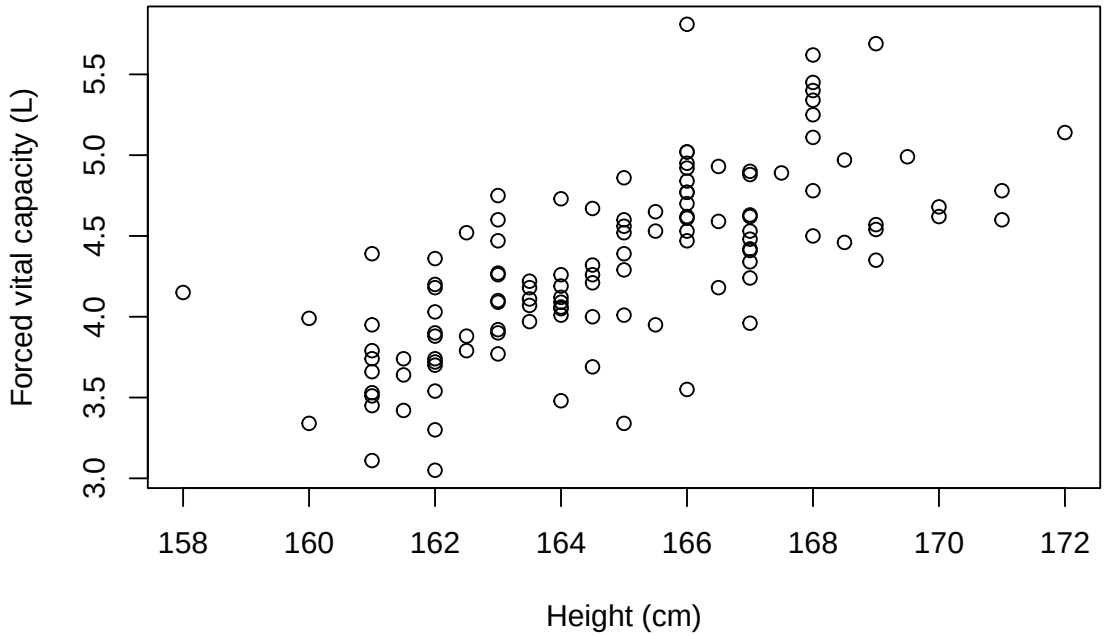
We will demonstrate using Stata for correlation and simple linear regression using the dataset `Example_8.1.rds`.

```
lung <- readRDS("data/examples/Example_8.1.rds")
```

8.1 Creating a scatter plot

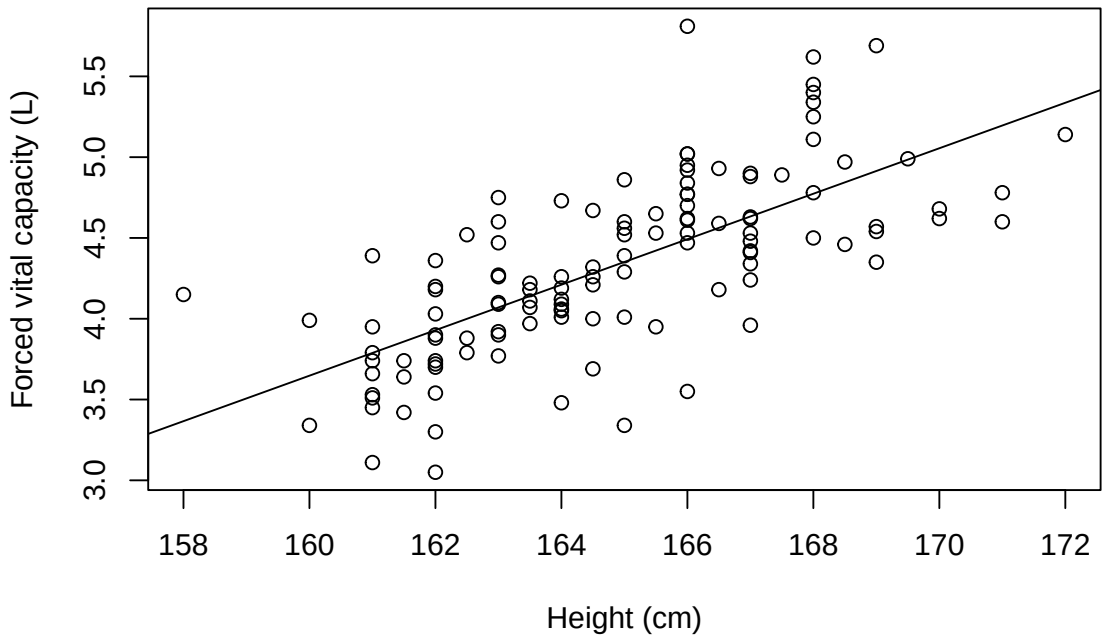
We can use the `plot` function to create a scatter plot to explore the association between height and FVC, assigning meaningful labels with the `xlab` and `ylab` commands:

```
plot(x=lung$Height, y=lung$FVC,
     xlab="Height (cm)",
     ylab="Forced vital capacity (L)")
```



To add a fitted line, we can use the `abline()` function which adds a straight line to the plot. The equation of this straight line will be determined from the estimated regression line, which we specify with the `lm()` function, which fits a *linear model*.
The basic syntax of the `lm()` function is: `lm(y ~ x)` where `y` represents the *outcome* variable, and `x` represents the *explanatory* variable. Putting this all together:

```
plot(x=lung$Height, y=lung$FVC,
     xlab="Height (cm)",
     ylab="Forced vital capacity (L)")

abline(lm(lung$FVC ~ lung$Height))
```



Calculating a correlation coefficient

We can use the `cor.test(x, y)` function to calculate a Pearson's correlation coefficient:

```
cor.test(lung$Height, lung$FVC)
```

```
##
##  Pearson's product-moment correlation
##
## data:  lung$Height and lung$FVC
## t = 10.577, df = 118, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.5924715 0.7794090
## sample estimates:
##      cor
## 0.697628
```

8.2 Fitting a simple linear regression model

We can use the `lm` function to fit a simple linear regression model, specifying the model as `y ~ x` where `y` represents the *outcome* variable, and `x` represents the *explanatory* variable. Using `Example_8.1.rds`, we can quantify the relationship between FVC and height:

```
lm(lung$FVC ~ lung$Height)
```

```
##
## Call:
```

```
## lm(formula = lung$FVC ~ lung$Height)
##
## Coefficients:
## (Intercept)  lung$Height
##    -18.8735       0.1408
```

The default output from the `lm` function is rather sparse. We can obtain much more useful information by defining the linear regression model as an object, then using the `summary()` function:

```
model <- lm(lung$FVC ~ lung$Height)
summary(model)
```

```
##
## Call:
## lm(formula = lung$FVC ~ lung$Height)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.01139 -0.23643 -0.02082  0.24918  1.31786
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -18.87347    2.19365  -8.604 3.89e-14 ***
## lung$Height   0.14076    0.01331  10.577  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3965 on 118 degrees of freedom
## Multiple R-squared:  0.4867, Adjusted R-squared:  0.4823
## F-statistic: 111.9 on 1 and 118 DF,  p-value: < 2.2e-16
```

Finally, we can obtain 95% confidence intervals for the regression coefficients using the `confint` function:

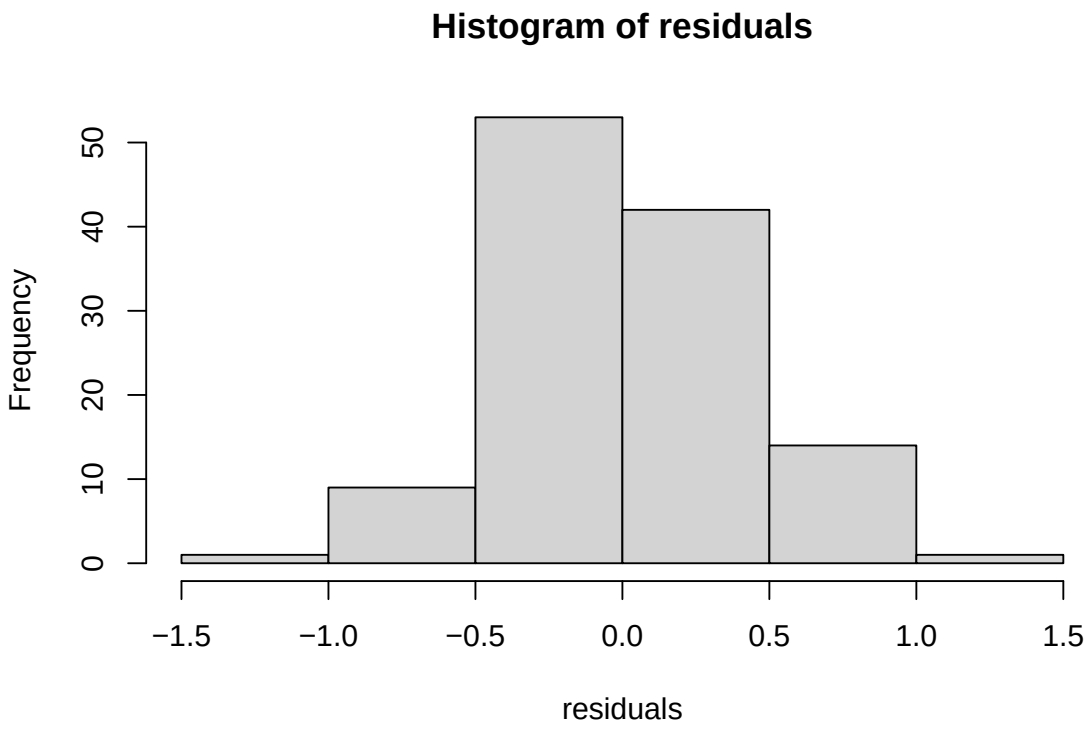
```
confint(model)
```

```
##                   2.5 %      97.5 %
## (Intercept) -23.2174967 -14.5294444
## lung$Height   0.1144042   0.1671092
```

8.3 Plotting residuals from a simple linear regression

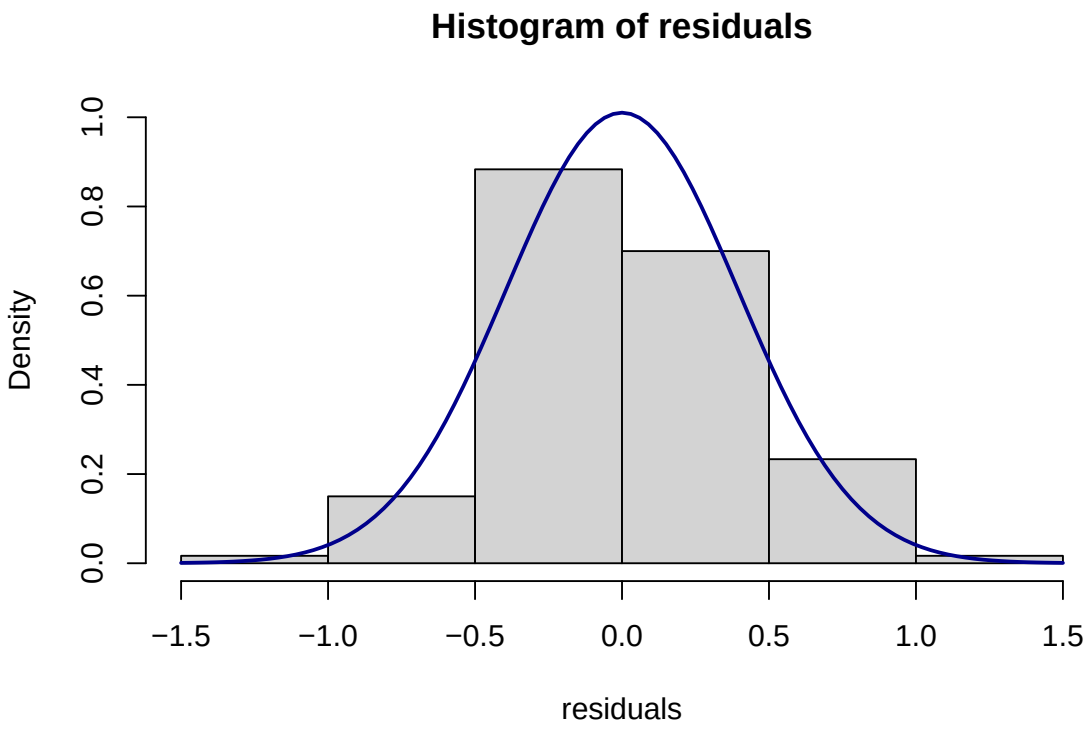
We can use the `resid` function to obtain the residuals from a saved model. These residuals can then be plotted using a histogram in the usual way:

```
residuals <- resid(model)
hist(residuals)
```



A Normal curve can be overlaid if we plot the residuals using a probability scale.

```
hist(residuals, probability = TRUE,
     ylim = c(0, 1))

curve(dnorm(x, mean=mean(residuals), sd=sd(residuals)),
      col="darkblue", lwd=2, add=TRUE)
```


Histogram of residuals
Density
0.0
0.2
0.4
0.6
0.8
1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
residuals

Bibliography

Terry M. Therneau and Patricia M. Grambsch. *Modeling Survival Data: Extending the Cox Model*. Springer, New York Berlin Heidelberg, December 2010. ISBN 978-1-4419-3161-0.