WEEK 1: INTRODUCTION TO R AND BASIC DESCRIPTION OF DATA

Juanjo Medina

Table of Contents

1.1 Introduction

This course provides an introduction to data analysis using R, a programming language and environment -or as some argue the lingua franca of statistics. You can learn more about the history of R clicking here and if you Google "why r" you will find many answers to the question -such as this.

These handouts contains various hyper-links to videos that explain or expand some of the ideas and concepts introduced. It will be handy if you have speakers or headphones to listen to the videos.

In order to follow this handout at home you would need to install both R and R Studio in your local machine -click here for instructions using Windows or here for instructions using a Mac. Alternatively, if you are a university student, your university is likely to have these programs already installed in the computers in their labs.

In this session we are going to learn some of the basic grammar elements in R. Before we start there are four thoughts I want to share:

- If you have never learnt a programming language, it is likely that this may feel a bit intimidating and arcane at first. But as with the learning of anything else, **practice will increase your confidence, understanding, and skills**. And if you practice a lot, speaking R will feel as natural to you as speaking your mother tongue. It is paramour that you ignore your fear. As Roosevelt once said: "the only thing we have to fear is...fear itself a nameless, unreasoning, unjustified terror which paralyzes needed efforts to convert retreat into advance."

- It is important that you **moderate your expectations**. You would not expect to be the main act at Glastonbury after just a few guitar lessons. But even if you never sell a million records, you can get pretty good at guitar playing by practising. Or going back to the language metaphor, you may not be able to write like Shakespeare, but as long as you can communicate with others in a functional way that's what really matters.

- **Get used to making mistakes!** You will make loads of them and there's absolutely no shame on that. As Patrick Burns, author of the helpful R Inferno and Impatient R, suggests: "An important tool to get around in R is to have a hacking attitude to try things with the idea that they probably won't work". Making mistakes is normal and an essential part of the learning process. We learnt by receiving good feedback when making mistakes. My job as an instructor is to give feedback to my students and yours, as a learner, to ask others for help.

- **Embrace frustration** You will get frustrated. But "frustration is not just natural, it's actually a positive thing that you should watch for. Frustration is your brain's way of being lazy; it's trying to get you to quit and go do something easy or fun. If you want to get physically fitter, you need to push your body even though it complains. If you want to get better at programming, you'll need to push your brain. *Recognise when you get frustrated and see it as a good thing; you're now stretching yourself*" (Wickham, 2014: vii-viii).

- **Don't aim to memorise everything.** There are over 5000 **packages** in R (modules that expand the functionality of R). Each of them create additional **functions** that allow you to do new cool stuff with your data. Nobody expects you to know by heart all of these functions and what they do. Memorising them would be as useless as memorising a dictionary. Instead, as you use R more and more your memory of basic functions (those you use the most) will improve. But in the day of searchable help files, Google, and resources such as Quick-R or the R-Codebook (providing quick "cheat-sheets" for common commands), the need to fill your memory with these functions is considerably reduced.

1.2 Interacting with R and creating objects

Unlike other programs for data analysis you may have used in the past (Excel, SPSS), you need to interact with R by means of writing down instructions and asking R to evaluate those instructions. R is an *interpreted* programming language.

If you open the standard R interface you will be confronted with a command prompt in the main console. You can write the R language in the **command line** directly, but most typically you will write in a **script**, a text file, from which you can also run commands and that has the advantage that can be saved.

In these handouts you will see bits of these instructions (I will often refer to this as "code") inside greyed boxes and the result of executing those instructions in white boxes. If you are looking at this document and have an open session of R, you should be able to reproduce the results by cutting it and pasting it into your R console (or script).

At the R prompt in the main console we can type **expressions**. R expressions could be as simple as asking R to evaluate how much is 3+5.

```
3 + 5
```

```
## [1] 8
```

Once you press enter, the R engine will evaluate this expression and do something with it. In this case, it will tell you that the result of adding 3 and 5 is 8. Using R simply as a calculator would be akin to killing flies with nuclear missiles. Typically we want to do slightly more interesting things with it.

We use R for analysing and visualising data. And data in R is stored as **objects**. R is what we call an *object-oriented language*. **Everything that exists in R is an object**.

If you are used to other programs such as Excel or SPSS the first mental barrier you need to cross in order to understand R is that (unlike in those programs), data in R can come in many shapes. Excel or SPSS files come in a tabular spreadsheet form, with rows typically representing cases and columns representing characteristics or attributes of those cases (what we also call variables). Typically, you can only work with one of these data tables at the time. R can work with this sort of spreadsheet data structure. However, it can also work with data that comes in many other shapes. What is more it can work with several of these objects simultaneously. This is part of what makes R so flexible and special.

We can create objects in R by asking R to put things inside of named objects. For that we use the **assignment operator**. In R the `<-` symbol is the assignment operator (you could also use the `=` sign as an assignment operator and some people do, *but I would suggest you don't*).

This assignment operator is what assigns value to a symbol. So, for example, if we type the following expression in the prompt:

```
x <- 5
```

We are simply telling R to create a *numeric object*, called `x`, with one element (5) or of length 1. It is numeric because we are putting a number inside this object. It may help you at this stage to think of objects as boxes, things where you store stuff and the assignment operator as the tool you use to tell R what goes inside.

When a complete expression is entered at the prompt in the main console and we press *return*, this expression is then evaluated by the R engine and the result is returned. This result may be auto-printed in the main console or not (and the latter may take some get used to if you've used before something like SPSS). If you press return after entering the previous expression you won't see much happening in the main console (although if you are using R Studio you may have noticed that a new object, `x`, appears in the global environment window in R Studio).

However, you can see the content of the object `x` either by auto-printing by typing the following:

```
x
```

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

Or alternatively, you can use a **function**. R uses functions to perform operations. **Everything you do in R is the result of running a function**. You can think of functions as preprogrammed routines that ask R to do a particular something. Here we can use the `print()` function to see what it is inside this object.

```
print(x)
```

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

When writing expressions in R is very important you understand that **R is case sensitive**. This could drive you nuts if you are not careful. More often than not if you write an expression asking R to do something and R returns an error message, chances are that you have used lower case when upper case was needed (or vice-versa). So always check for the right spelling. For example, see what happens if I ask the following:

```
print(X)
```

You will get the following message: `"Error in print(X) : object 'X' not found"`. R is telling us that `X` does not exist. There isn't an object " `X` (upper case), but there is an object `x` (lower case). When you get an error message or implausible results, you want to look back at your code to figure out what is the problem. This process is called **debugging**. Very often the solution will simply involve correcting the spelling.

Let's look at another example. We are going to create another object `y` containing a sequence of integer values from 1 to 20.

```
y <- 1:30 # The : operator is used to create integer sequences
y #Auto printing of Y
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
```

So here we now have an integer object with a length of 30, much longer than the one we created above. The first line has a 1 because that's the first element in this object, the second line starts with a number 24 because the first element in that line is the 24th element in the `y` object.

You must have noticed in the code that there was a number sign `#` followed by some text. Whenever the R engine sees the number sign it knows that what follows is not code to be executed. You can use this sign to include *annotations* when you are coding. These annotations are a helpful reminder to yourself (and others reading your code) of **what** the code is doing and (even more important) **why** you are doing it. It is good practice to often use annotations. You can use these annotations in your code to explain your reasoning and to create "scannable" headings in your code. That way after you save your script you will be able to share it with others or return to it at a later point and understand what you were doing when you first created it -see here for further details on annotations and in how to save a script when working with the basic R interface.

In sum, everything that exist in R is an object and everything you do in R is the result of running a function. In fact, the numbers we used in the sum operation earlier are numeric objects (R does not uses scalars as other programming languages) and the sum operator is in fact a function for summation. If you don't believe try executing the following:

```
"+"(3,5)
```

1.3 Vectors, factors and data frames

Most commonly, when you use variables in R, you create **vectors**. What is a vector? An atomic vector is simply a set of elements *of the same class* (typically: character, numeric, integer, or logical -as in True/False). It is the basic data structure in R. Typically you will use the `c()` function (c stands for concatenate) to create vectors.

The code below exemplifies how to create vectors of different classes (numeric, logical, etc.). Notice how the listed elements (to simplify there are two elements in each vector below) are separated by commas:

```
my_1st_vector <- c(0.5, 0.6) #creates a numeric vector with two elements
my_2nd_vector <- c(1L, 2L) #creates an integer vector
my_3rd_vector <- c(TRUE, FALSE) #creates a logical vector
my_4th_vector <- c(T, F) #creates a logical vector using abbreviations of True and False, but you shoul
my_5th_vector <- c("a", "b", "c") #creates a character vector
my_6th_vector <- c(1+0i, 2+4i) #creates a complex vector (we won't really use this class)
```

The beauty of an object oriented statistical language is that one you have these objects you can use them as **inputs** in functions, use them in operations, or to create other objects. This makes R very flexible:

```
class(my_1st_vector) #to figure out the class of the vector
```

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

```
length(my_1st_vector) #to figure out the lenght of the vector
```

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

```
my_1st_vector + 2 #Add a constant to each element of the vector
```

```
## [1] 2.5 2.6
```

```
my_7th_vector <- my_1st_vector + 1 #Create a new vector that contains the elements of my1stvector plus
my_1st_vector + my_7th_vector #Adds the two vectors and auto-print the results (note how the sum was do
```

```
## [1] 2.0 2.2
```

When you create objects you will place them in your working memory or workspace. Each R session will be associated to a workspace (curiously called "global environment"). Think of it as a warehouse where you are placing a bunch of boxes (your objects). R works using your RAM. That's where all these objects get stored, which means you need good RAM for very large datasets. In R Studio you can visualise the objects you have created during a session in the **Global Environment** screen. But if you want to produce a list of what's there you can use the `ls()` function (the results you get my differ from the ones below depending on what you actually have in your global environment).

```
ls() #list all objects in your global environment
```

```
## [1] "my_1st_vector" "my_2nd_vector" "my_3rd_vector" "my_4th_vector"
## [5] "my_5th_vector" "my_6th_vector" "my_7th_vector" "x"
## [9] "y"
```

If you want to delete a particular object you can do so using the `rm()` function.

```
rm(x) #remove x from your global environment
```

It is also possibly to remove all objects at once:

```
rm(list = ls()) #remove all objects from your global environment
```

If you mix in a vector elements that are of a different class (for example numerical and logical), R will **coerce** to the minimum common denominator, so that every element in the vector is of the same class. So, for example, if you input a number and a character, it will coerce the vector to be a character vector -see the example below and notice the use of the `class()` function to identify the class of an object.

```
my_8th_vector <- c(0.5, "a")
class(my_8th_vector) #The class() function will tell us the class of the vector
```

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

An important thing to understand in R is that categorical (ordered, also called ordinal, or unordered, also called nominal)[1] data are typically encoded as **factors** . A factor is simply an integer vector that can contain only predefined values, and is used to store categorical data. Factors are treated specially by many data analytic and visualisation functions. This makes sense because they are essentially different from quantitative variables.

Although you can use numbers to represent categories, *using factors with labels is better than using integers to represent categories* because factors are self-describing (having a variable that has values "Male" and "Female" is better than a variable that has values "1" and "2"). When R reads data in other formats (e.g., comma separated), by default it will automatically convert all character variables into factors. If you rather keep these variables as simple character vectors you need to explicitly ask R to do so.

Factors can also be created with the `factor()` function concatenating a series of *character* elements. You will notice that is printed differently from a simply character vector and that it tells us the levels of the factor (look at the second printed line).

```
the_smiths <- factor(c("Morrisey", "Marr", "Rourke", "Joyce")) #create a new factor
the_smiths #auto-print the factor
```

```
## [1] Morrisey Marr     Rourke   Joyce
## Levels: Joyce Marr Morrisey Rourke
```

[1]If you are confused by this terminology watch this video

```r
#Alternatively for similar result using the as.factor() function
the_smiths_bis <- c("Morrisey", "Marr", "Rourke", "Joyce") #create a character vector
the_smiths_f <- as.factor(the_smiths_bis) #create a factor
the_smiths_f #auto-print factor
```

```
## [1] Morrisey Marr     Rourke   Joyce
## Levels: Joyce Marr Morrisey Rourke
```

Factors in R can be seen as vectors with a bit more information added. This extra information consists of a record of the distinct values in that vector, called **levels**. If you want to know the levels in a given factor you can use the `levels()` function:

```r
levels(the_smiths)
```

```
## [1] "Joyce"    "Marr"     "Morrisey" "Rourke"
```

Notice that the levels appear printed by alphabetical order. There will be situations when this is not the most convenient order. We will discuss in these tutorials how to reorder your factor levels when you need to.

You may have noticed the various names I have used to designate objects (`my_1st_vector`, `the_smiths`, etc.). You can use almost any names you want for your objects. Objects in R can have names of any length consisting of letters, numbers, underscores ("_") or the period (".") and should begin with a letter. In addition, when naming objects:

- Some names are forbidden. These include words such as FALSE and TRUE, logical operators, and programming words like Inf, for, else, break, function, and words for special entities like NA and NaN.
- You want to use names that do not correspond to a specific function. We have seen, for example, that there is a function called `print()`, you don't want to call an object "print" to avoid conflicts. To avoid this use nouns instead of verbs for naming your variables and data.
- You don't want them to be too long (or you will regret it every time you need to use that object in your analysis: your fingers will bleed from typing).
- You want to make them as intuitive to interpret as possible.
- You want to follow consistent naming conventions. R users are terrible about this. But we could make it better if we all aim to follow similar conventions. In these handouts you will see I follow the `underscore_separated` convention -see here for details.

One of the most common objects you will work with in this course are **data frames**. Data frames can be created with the `data.frame()` function. Data frames are multiple vectors of possibly different classes (e.g., numeric, factors), but of the same length (e.g., all vectors, or variables, have the same number of rows). This is what in other programmes for data analysis are represented as data sets, the tabular spreadsheets I was referring to earlier.

```r
#We create a dataframe called mydata.1 with two variables, an integer vector called foo and a logical v
mydata_1 <- data.frame(foo = 1:4, bar = c(T,T,F,F))
mydata_1
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

Or alternatively for the same result:

```
x <- 1:4
y <- c(T, T, F, F)
mydata_2 <- data.frame (foo = x, bar = y)
mydata_2
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

As you can see in R, as in any other language, there are multiple ways of saying the same thing. Programmers aim to produce code that has been optimised: it is short and quick. It is likely that as you develop your R skills you find increasingly more efficient ways of asking R how to do things.

Every object in R can have **attributes**. These are: names; dimensions (for matrices and arrays: number of rows and columns) and dimensions names; class of object (numeric, character, etc.); length (for a vector this will be the number of elements in the vector); and other user-defined. You can access the attributes of an object using the `attributes()` function.

```
attributes(mydata_1)
```

```
## $names
## [1] "foo" "bar"
##
## $row.names
## [1] 1 2 3 4
##
## $class
## [1] "data.frame"
```

By now you must have also noticed the common structure of **functions** in R. You can think of functions as executable commands that R will evaluate. You will have noticed functions have a name followed by a bracket and that you can pass *arguments* to the function by including them within the brackets. In the previous example we were using a function called *attributes* and we passed the argument `mydata_1`.

A function in R can take any number of arguments. You can obtain help about functions in R (and the specific arguments they can take) by using the `help()` function. This will give you access to the help files as a html file. These help files may look cryptic at first, but the more you use R the easier it gets to understand how they work. Don't underestimate the examples at the end of these files. As a beginner they may not mean much to you, but when you get more fluency with R executing those examples will help you in terms of understanding what the functions do.

```
help(attributes)
#Or alternatively
?attributes
```

A final word in code presentation and **coding conventions** before we carry on. Code is a form of communication and it is important that you write it in a way that others will read it clearly. As Hadley Wickham[2] has noted: "Good coding style is like using correct punctuation when writing: you can manage without it, but it sure makes things easier to read." Apart from using the "#" sign to make annotations, there are other basic conventions you should also follow:

[2] A "semi-legendary" R programmer and part of the R Studio team. We all want to know as much R as he does.

- Every comma should be followed by a space
- Every mathematical operator (+, -, =, *, /, etc.) should be surrounded by spaces
- Parentheses do not need spaces
- Lines should be at most 80 characters. If you have to break up a line, indent the following piece.

You may want to look at the style guide used by Google programmers using R for further details, but the basic conventions listed above will possibly suffice for now.

1.4 Data filtering

R capabilities for data manipulation are ridiculously rich. We could devote entire sessions just to discuss those. Here I am just going to introduce some very basic ones around filtering. For more detail (or if you want to do something I have not explained), look at: the R Cookbook; Quick R; this twotorial; or this guide to the fast and excellent `dplyr` package; and for even more detail (than you possibly want at this stage) you can consult a preprint copy of Rob Muenchen's book.

Filtering allows you to extract a vector's element that satisfy certain condition. You will be commonly doing filtering, as statistical analysis often focuses on data that satisfies certain conditions. Going through some of these data manipulation commands now will help you to understand a bit better some of the code that we will be using during the course. Let's start by creating a small data frame:

```
#First we create a numeric vector and a factor (notice they have the same lenght of 5)
y <- c(89, 78, 75, 73, 72)
x <- factor(c("Man City", "Liverpool", "Chelsea", "Arsenal", "Everton"))
#Then use use the date.frame function to put this two vectors in a data frame object.
Premiership13_14 <-data.frame(teams = x, points = y)
Premiership13_14
```

```
##       teams points
## 1  Man City     89
## 2 Liverpool     78
## 3   Chelsea     75
## 4   Arsenal     73
## 5   Everton     72
```

While we are doing our data analysis we will often only want to work with part of a vector or data frame. We can do **subsetting** in a variety of ways. If I just want to access the first element in the `x` vector I could use the squared brackets and ask for:

```
x[1]
```

```
## [1] Man City
## Levels: Arsenal Chelsea Everton Liverpool Man City
```

Similarly we can use the concatenate function `c()` within the square brackets if we want indices of particular values:

```
y[c(1, 2, 4)] #For the 1st, 2nd and 4th elements of the y vector
```

```
## [1] 89 78 73
```

We can also look at specific rows and columns in a data frame:

```
#This will return the first row and columns 1 to 2 of the named data frame (anything to the left of the
Premiership13_14[1, 1:2]
```

```
##      teams points
## 1 Man City     89
```

We can access a particular column in the data frame, typically our variables, using the **dollar sign operator**:

```
#This will return all the values in the teams column that exist in the Premiership13_14 data frame
Premiership13_14$teams
```

```
## [1] Man City  Liverpool Chelsea   Arsenal   Everton
## Levels: Arsenal Chelsea Everton Liverpool Man City
```

We can also subset by particular logic operations. Here it is an example. Suppose I want to identify all rows with a team name that equals to Man City. In that case we can use the variable name team and check using the `==` (which in R means equal) operator if the row equals to Man City. This particular vector will return true only when Man City appears:

```
Premiership13_14[Premiership13_14$teams == "Man City", ] #We are asking for rows where the condition is
```

```
##      teams points
## 1 Man City     89
```

Since we did not excluded any column auto printing this object returns information for every variable in the data frame as it applies to Man City.

You can also perform other type of operations regarding numerical vectors. Say we want to subset the rows with less than 75 points gained through the footballing season last year (remember we only included the first 5 top teams!). Then you would write:

```
Premiership13_14[Premiership13_14$points < 75, ]
```

```
##     teams points
## 4 Arsenal     73
## 5 Everton     72
```

The number 4 and 5 that you see indicate that those are the respective row numbers for those two cases. You will have noticed again a comma within those square brackets, whatever you put after the comma deals with columns rather than rows. Since we don't specify anything there we get all the columns.

You can also filter with the `subset()` function. When applied to vectors, the difference between this function and the ordinary filtering we have covered so far lies in the manner in which missing data is handled. Let's see an example:

```
z <- (c(6, 1:3, NA,12)) #This creates a vector with 6 values, one of which is NA. NA is the label R use
z
```

```
## [1]  6  1  2  3 NA 12
```

If we use ordinary filtering NA values (missing data) will be included in our selection:

```
z[z > 5]
```

```
## [1]  6 NA 12
```

But if we use the `subset()` function the NA values are excluded from the selection:

```
subset(z, z > 5) #As arguments you need to specify the vector (z) and then the filtering condition (z >
```

```
## [1]  6 12
```

It is important to remember this difference for you may have particular reasons to exclude or include NA values in what you are planning to do.

For other logical and arithmetic operators that you can use in R for filtering (and how to write them) in R please look here. For further details you may also want to see Roger Peng's video on subsetting. The `dplyr` package is truly quite something if you are interested in filtering data frames. It uses more intuitive language and adds some very helpful functions, but in this section I thought it was important to cover the more traditional approach of filtering with R. In later sessions we will introduce some of the features of `dplyr`.

1.5: Loading data and packages

You can then, as we have seen, create data into R in a variety of ways. In data analysis, however, it is very common that you will work with data already formatted (and collected by somebody else: government, a professional survey organisation you may have contracted for fieldwork, other researchers, etc.). We live in a time where data is everywhere; some people call it the "data revolution that will change the way we live". And there are, indeed, a variety of places where you can obtain datasets for secondary analysis such as the UK Data Archive or the ICPSR website (feel free to Google these, they may give you ideas for your MA dissertation or future PhD research).

For most analysis then, the first step will involve importing an already existing data frame into R. In this course every time we introduce one of these datasets you will be provided with some background information about the dataset in question: a short **codebook** telling you how the data was collated and some information about the variables included. This information is often called **metadata**, data about data. The first thing you always do in data analysis is to have a look at this codebook to understand what is in it and how the data was generated. With long dry codebooks this may be as inviting as reading a telephone guide, but I can assure you it will save you time and prevent you from making mistakes later on (as I learnt the hard way!).

For the most part the data we use in the course have already being formatted and cleaned to make your life easier. In fact, so have the codebooks. But in real life data, even pre-processed data, tends to be messy. The day to day of data analysis implies spending the bulk of the time (up to 90%) in what some informally call **data munging** before you can start exploring your data for interesting patterns. Pre-processing data may involve: fixing variable names; creating new variables; merging data sets; reshaping data sets; dealing with missing data; transforming variables; checking on and removing inconsistent values; etc. You can find here a short discussion about data carpentry of this type. Using R means that you will be induced to have a much better (and essential) record of all these operations (by saving the scripts performing them).

The sanitised datasets we use in this course will be mostly loaded from the internet. One of the cool things about R is that it can easily read data directly from the internet if you provide a URL for it -click here for a video "twotorial" demonstration. You will be able to download this data from my public repositories in GitHub, a cloud storage facility used by programmers and data analysts [3].

Data may come in a variety of formats (Excel files, comma separated, tab separated, SPSS, STATA, JSON, etc.) [4]. There are functions that help **importing** into R files in these various formats. We are going to start with a simple case. We are going to read a textual comma separated file into R.

In particular, we are going to read a dataset which is a Teaching Unrestricted version of the *British Crime Survey for 2007-2008*. The codebook with information about this dataset is here. This file is in comma separated format and therefore we will be using the `read.csv()` function.

```
##R in Windows have some problems with https addresses, that's why we need to do this first:
urlfile<-'https://raw.githubusercontent.com/jjmedinaariza/LAWS70821/master/BCS0708.csv'
#We create a data frame object reading the data from the remote .csv file
BCS0708<-read.csv(url(urlfile))
```

Note that R requires forward slashes (/) not back slashes () when specifying a file location even if the file is on your hard drive. Another way of getting this sort of file would be using the following approach:

```
#Download the data into your working directory
download.file('https://raw.githubusercontent.com/jjmedinaariza/LAWS70821/master/BCS0708.csv', "BCS0708.
#Read the data in the csv file into an object called BCS0708
BCS0708 <- read.csv("BCS0708.csv")
```

Once in your computer you can load the file into your working environment every time you need it for a different project using the `load()` function.

One of the beauties of R is that anybody can expand its functionality. There are over 5000 **packages** containing collections of functions that expand the functionality of R. Watch this tutorial for how to install and load a package into R. In this course we use packages that are not part of the basic installation of R, which means you will have to install them in your local machine (or in you are one of my students in Manchester the p:drive when you are using a computer at any of the University computer clusters). You can see the packages that are installed locally in the packages tab in R Studio. Or you could use the `library()` function with no arguments.

```
library()
```

After you have a package *installed* in your local machine, you need to *require* it. It is important for you to remember these are two different operations. You install once, you require or load the package every time you start a new session and want to use the package. Think of a new pair of shoes. You need to buy them only once, but every time you want to use them, you have to put them on. It is the same with packages.

To know what packages are currently active in your session, you can use the `search()` function:

```
search()
```

Packages typically include collections of new functions, but often also include datasets for demonstration purposes. This means that if you have installed and loaded a package which contains a particular dataset, you can also easily access it. In this course, we will use data from existing packages as well.

[3] I use GiT and GitHub as my systems of version control. We don't have the time to cover version control systems in this course. But for the purpose of this course you simply need to know that all the data will be stored in GitHub and that you will be able to load the data directly from it. Version control and GiT in particular is a helpful tool for a data scientist. RStudio allows you to use GiT as your system of version control. If you are interested you can find a great tutorial for GiT and GitHub here.

[4] The following video "twotorials" (in two minutes or less) provide information in how to read into R in a variety of formats: csv files; SPSS, STATA, and SAS; and Excel files - here for a different way to read Excel.

The first of this is the `Boston` dataset from the `MASS` package that is part of the base installation of R (you do not need to install it). To get the data from a package in your global environment you will see often people may load the package (if it's not already loaded) using the `library()` function and then use the `data()` function. But you don't need to do that, you could instead specify the package where the data is stored as an argument in your `data()` function.

```
data(Boston, package="MASS")
```

If you then run the code below you will get the help file that contains information (metadata) about this particular dataset. You can think of it as the codebook for this dataset. You will see that this is a dataset that contains information about crime and other characteristics in various parts of Boston (USA).

```
help(Boston, package="MASS")
```

You can see all data that are loaded in a session by using the `data()` function with no arguments. You can use Use `data(package = .packages(all.available = TRUE))` to list the data sets in all *available* packages. That is, in all packages installed in your local machine.

```
data()
```

This will list data frames that are part of R base installation or loaded packages, not other data frames that you may have created or loaded. For the latter you need to use the `ls()` function as we explained earlier.

1.6: A first look at the data

We are going to work with the British Crime Survey data for the remainder of this session. And we want to start by having a sense for what the data look like.

Data are often too big to look at the whole thing. It is almost always impossible to eyeball the entire dataset and see what you have in terms of interesting patterns or potential problems. It is often a case of information overload and we want to be able to extract what is relevant and important about it. Summarising the data is the first step in any analysis and it is also used for finding out potential problems with the data. Regarding the latter you want to look out for: missing values; values outside the expected range (e.g., someone aged 200 years); values that seem to be in the wrong units; mislabelled variables; or variables that seem to be the wrong class (e.g., a quantitative variable encoded as a factor).

If you simply type the name of the new dataset BCS0708 and press return for auto-printing, you will see why we need data summaries. It is hard to eyeball large datasets like this trying to see the whole thing. So lets start by the basic things you always look first in a datasets. First you ask for the dimensions, the number of rows and columns, using the `dim()` function:

```
dim(BCS0708)
```

```
## [1] 11676    35
```

We can see that the dataset has 11676 observations or rows and 35 columns or variables. Looking at this information will help you to diagnose whether there was any trouble getting your data into R (e.g., imagine you know there should be more cases or more variable). You may also want to have a look at the names of the columns using the `names()` function. We will see the names of the variables.

```r
names(BCS0708)
```

```
##  [1] "rowlabel"  "sex"       "age"       "livharm1"  "ethgrp2"  
##  [6] "educat3"   "work"      "yrsarea"   "resyrago"  "tenure1"  
## [11] "rural2"    "rubbcomm"  "vandcomm"  "poorhou"   "tcemdiqu2"
## [16] "tcwmdiqu2" "causem"    "walkdark"  "walkday"   "homealon" 
## [21] "tcviolent" "tcsteal"   "wburgl"    "wmugged"   "wcarstol" 
## [26] "wfromcar"  "wraped"    "wattack"   "winsult"   "wraceatt" 
## [31] "crimerat"  "tcarea"    "tcneigh"   "bcsvictim" "tcindwt"
```

As you may notice, these names are hard to interpret. You need to look at the codebook to figure out what each of those variables is actually measuring. The bad news is that real life datasets in social science are much larger than this and have many more variables (hundreds or more). The good news is that typically you will only need a small handful of them and will only require to deeply familiarise yourself with that smaller subset.

Then you may want to look at the class of each individual column. You can do this using the following code. It is a tricky way of doing it, we are using the `sapply()` function to ask to the class of the first row of all the columns and then inferring the class from this first row (which will be the same class in all rows, remember what we said about coercion earlier on):

```r
sapply(BCS0708[1, ], class)
```

```
##  rowlabel       sex       age  livharm1   ethgrp2   educat3      work 
## "integer"  "factor" "integer"  "factor"  "factor"  "factor"  "factor" 
##   yrsarea  resyrago   tenure1    rural2  rubbcomm  vandcomm   poorhou 
##  "factor"  "factor"  "factor"  "factor"  "factor"  "factor"  "factor" 
## tcemdiqu2 tcwmdiqu2    causem  walkdark   walkday  homealon tcviolent 
## "integer" "integer"  "factor"  "factor"  "factor"  "factor" "numeric" 
##   tcsteal    wburgl   wmugged  wcarstol  wfromcar    wraped   wattack 
## "numeric"  "factor"  "factor"  "factor"  "factor"  "factor"  "factor" 
##   winsult  wraceatt  crimerat    tcarea   tcneigh bcsvictim   tcindwt 
##  "factor"  "factor"  "factor" "numeric" "numeric"  "factor" "numeric"
```

As you can see many variables are classed as factors. This is common with survey data. Many of the questions in social surveys measure the answers as categorical variables (e.g., these are nominal or ordinal level measures) which are then encoded as factors in R. You should know that the default options for `read.csv()` will encode any character attribute in your file as a factor.

Another useful function is `str()`, which will return: the name of the variable; the class of each column; the number of levels or categories (if it is a factor); and the values for the first few cases in the dataset. You can also use the `head()` function if you just want to visualise the values for the first few cases in your dataset. The next code for example ask for the values for the first two cases.

```r
head(BCS0708, 2)
```

```
##   rowlabel    sex age  livharm1 ethgrp2                      educat3 work
## 1 61302140 female  36   married   white                         none  yes
## 2 61384060   male  44 separated   white apprenticeship or a/as level  yes
##                          yrsarea resyrago
## 1 10 years but less than 20 years     <NA>
## 2             less than 12 months       no
##                                         tenure1 rural2        rubbcomm
## 1 buying it with the help of a mortgage or loan  urban            <NA>
## 2                                       rent it  urban not very common
##            vandcomm           poorhou tcemdiqu2 tcwmdiqu2     causem
## 1              <NA>              <NA>         1        NA   e. drugs
## 2 not at all common not at all common         3        NA f. alcohol
##       walkdark        walkday     homealon  tcviolent  tcsteal
## 1  very unsafe or very unsafe a bit unsafe         NA       NA
## 2 a bit unsafe    fairly safe  fairly safe -0.3892744 2.139811
##           wburgl          wmugged       wcarstol     wfromcar
## 1 fairly worried not very worried            <NA>         <NA>
## 2   very worried     very worried fairly worried very worried
##               wraped          wattack          winsult         wraceatt
## 1   not very worried not very worried   fairly worried not very worried
## 2 not at all worried not very worried not very worried not very worried
##              crimerat   tcarea   tcneigh             bcsvictim  tcindwt
## 1 a little more crime 1.117700  2.212788 not a victim of crime 1.763460
## 2                <NA> 1.791787 -1.024336 not a victim of crime 3.844527
```

In the same way you could look at the last two cases in your dataset using `tail()`:

```r
tail(BCS0708, 2)
```

It is good practice to do this to ensure R has read the data correctly and there's nothing terribly wrong with your dataset. It can also give you a first impression for what the data looks like. If you are used to spreadsheet-like views of data, you can use the `View()` function, which should open this view in R Studio.

```r
View(BCS0708)
```

One thing you may also want to do is to see if there are any **missing values**. For that we can use the `is.na()` function. Missing values in R are coded as NA. The code below, for example, asks for NA values for the variable "educat3"" in the `BCS0708` object for cases 1 to 10:

```r
is.na(BCS0708$educat3[1:10])
```

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

R is telling me that none of those elements are missing. More typically, you can ask the count of NA values for a particular variable:

```r
sum(is.na(BCS0708$educat3)) #This is asking R to sum how many cases are TRUE NA in this variable. When
```

```
## [1] 58
```

You can use a bit of a hack to get the proportion of missing cases instead of the count:

```r
mean(is.na(BCS0708$educat3))
```

```
## [1] 0.004967455
```

This code is exploiting the mathematical fact that the mean of binary outcomes (0 or 1) gives you the proportion of 1s in your data.

If you see more than 5% of the cases declared as NA, you need to start thinking about the implications of this. Beware of formulaic application of rules of thumb such as this though!

There is a whole field of statistics devoted to doing analysis when missing data is a problem. R has extensive capabilities for dealing with missing data -see for example here. For the purpose of this introductory course, we only explain how to do analysis that ignore missing data. You would cover techniques for dealing with this sort of issues in more advanced courses -such as this offered by my lovely colleagues at CCSR.

The `any()` function is a useful one if you want to check that a particular value exist for a given variable. So, say we want to know whether we have anybody younger than 17 in the dataset, we could type the following:

```
any(BCS0708$age < 17)
```

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

1.7 Numerical summaries of central tendency and variability

In this section we start exploring some statistical summaries for our data. If you have not watched the required video for this week explaining these measures, it is convenient you watch it now -just click here. You may also find these other videos on measures of central tendency and the standard deviation useful.

Let's start with the *mean*. If you want to obtain the mean of a quantitative variable you can use the following expression:

```
mean(BCS0708$age)
```

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

NA? What's going on? Often this will happen to you. You get an unexpected result. Why may this be happening? You could use the `View()` function to visualise the data. If you view the data you will see that nothing seems odd with the variable. If you use the `str()` function you will see "age"" is a numeric vector, so it's not as if you are asking something unreasonable (e.g., computing the mean for a categorical variable). The writing of the code also seems ok (we are using lower and upper case correctly).

When these things happen, you will need to go through the mental process of eliminating likely explanations such as the ones we have gone through here. Typically, next step is to look at the help files for the function you are using.

If you look at the help files you will notice that there is a default argument for the `mean()` function. The default specifies the following *na.rm=FALSE*. This means that the NA values are not removed before computation proceeds. Of course, you get NA because it is mathematically impossible to perform an operation with a NA value. What is 2+NA? Nothing. I can only imagine this is the default to alert you to the fact you should not ignore missing data problems. Let's try again modifying the default. As you will see now it will work:

```
mean(BCS0708$age, na.rm = TRUE) #Specifying this argument will ensure that only cases with valid values
```

```
## [1] 50.42278
```

Another function you may want to use with numeric variables is `summary()`:

```
summary(BCS0708$age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   16.00   36.00   49.00   50.42   65.00  101.00      15
```

This gives you the five number summary (minimum, first quartile, median, third quartile, and maximum, plus the mean and the count of NA values). Notice how when using this function you did not need to change any default settings to remove NA values.

There are multiple ways of getting results in R. Particularly for basic and intermediate-level statistical analysis many core functions and packages can give you the answer that you are looking for. For example, there are a variety of packages that allow you to look at summary statistics using functions defined within those packages. You will need to install these packages before you can use them.

Once installed you can activate packages with either the `library()` or the `require()` functions. It is a matter of some debate which to use. Some people think that there are good reasons to prefer `library()`, even if `require()` uses a more explicit language as to what you are doing. In these handouts we will use `library()`.

You could use `favstats()` function from the mosaic package:

```
library(mosaic)
favstats(~age, data = BCS0708) #five number summary, mean, standard deviation, and number of cases
```

```
##  min Q1 median Q3 max     mean      sd     n missing
##   16 36     49 65 101 50.42278 18.5389 11661      15
```

The `describe()` function from the Hmisc package:

```
library(Hmisc)
describe(BCS0708$age) #n,missing, mean, various percentiles, the 5 lowest and highest values
```

```
## BCS0708$age
##       n missing  unique    Info    Mean     .05     .10     .25     .50
##   11661      15      84       1   50.42      21      26      36      49
##     .75     .90     .95
##      65      76      81
##
## lowest :  16  17  18  19  20, highest:  95  97  98  99 101
```

The `stat.desc()` function from the pastecs package:

```
library(pastecs)
stat.desc(BCS0708$age) #Which gives you among other things the standard error, standard deviation, and
```

```
##      nbr.val     nbr.null       nbr.na          min          max
## 1.166100e+04 0.000000e+00 1.500000e+01 1.600000e+01 1.010000e+02
##        range          sum       median         mean      SE.mean
## 8.500000e+01 5.879800e+05 4.900000e+01 5.042278e+01 1.716785e-01
## CI.mean.0.95          var      std.dev     coef.var
## 3.365187e-01 3.436907e+02 1.853890e+01 3.676691e-01
```

Or the `describe()` function from the psych package:

```
library(psych)
```

```
##
## Attaching package: 'psych'
##
## The following object is masked from 'package:boot':
##
##     logit
##
## The following object is masked from 'package:Hmisc':
##
##     describe
##
## The following objects are masked from 'package:mosaic':
##
##     logit, rescale
##
## The following object is masked from 'package:car':
##
##     logit
##
## The following object is masked from 'package:ggplot2':
##
##     %+%
```

```
describe(BCS0708$age) #Which gives you a trimmed mean, as well as measures of skew and kurtosis
```

```
##   vars     n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 11661 50.42 18.54     49   50.17 20.76  16 101    85 0.11     -0.9
##     se
## 1 0.17
```

What package to use? Well, that's very much a matter of personal preference and what your specific needs are. As you have seen these packages give you slightly different sets of summaries. You may also inform your choice considering what other functions those packages include. It could be that one of those packages use functions for types of analysis you will use often. In that case it may be sensitive to rely on that particular package.

I want to alert you to something, however. You may have noticed that both the `psych` and the `Hmisc` package both include a `describe()` function. Packages are user-produced and there's no policing of the names people use for their functions. With 5000+ packages is unavoidable that some people will use the same names for some of their functions. When this happens R "masks" the first of these functions in use.

So in this session, if we use `describe()` again, we will use the function that was created as part of the `psych` package (since that was the last loaded package and by loading it we masked the describe function of the previous package). If we want to revert to the `describe()` function as designed in the `Hmisc` package you may need to load that package again so that R masks the `describe()` function from the `psych` package. It is important you keep an eye on this "masked" messages when you load packages!!!

Sometimes you want to produce **summary statistics by group**. The `psych` package is good for this:

```
#Since we just loaded this package we can continue using its functions
describeBy(BCS0708$age, BCS0708$livharm1) #age descriptives for various categories of marital status
```

```
## group: cohabiting
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 1032 38.27 12.86     36   36.93 11.86  18  95    77 0.95     0.72
##    se
## 1 0.4
## --------------------------------------------------------
## group: divorced
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 1055 54.28 12.61     55   54.14 13.34  24  95    71 0.09    -0.49
##     se
## 1 0.39
## --------------------------------------------------------
## group: married
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 5551 52.97 14.79     53   52.74 16.31  18  94    76 0.11    -0.84
##    se
## 1 0.2
## --------------------------------------------------------
## group: separated
##   vars   n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 323 47.66 13.64     45   46.64 13.34  24  86    62 0.63    -0.17
##     se
## 1 0.76
## --------------------------------------------------------
## group: single
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 2372 34.65 16.63     30   32.31 14.83  16  92    76 1.09     0.48
##     se
## 1 0.34
## --------------------------------------------------------
## group: widowed
##   vars    n  mean    sd median trimmed   mad min max range  skew kurtosis
## 1    1 1320 75.09 10.24     76   75.77 10.38  37 101    64 -0.68     0.54
##     se
## 1 0.28
```

If you look at the *mean* for each of the marital status groups, you can see that there is a relationship in this sample. Single people, for example, tend to be younger. Earth shattering discovery! Don't worry, we will look at more interesting examples in subsequent weeks. As I mentioned earlier, we will cover `dplyr` in later sections, but it is important at least to mention here that this package is also very good for breaking down your data frame in such a way that would allow you to produce summary statistics by group.

Now that you have a bit of more familiarity with R you can possibly understand why I earlier suggested to use short names for your data frames. Imagine that your data frame was called `british_crime_survey_2007_2008`. Then you would need to type the following:

```
#Don't try to execute this, it will tell you there is no object called British_Crime_Survey_2007_2008
describeBy(british_crime_survey_2007_2008$age, british_crime_survey_2007_2008$livharm1)
```

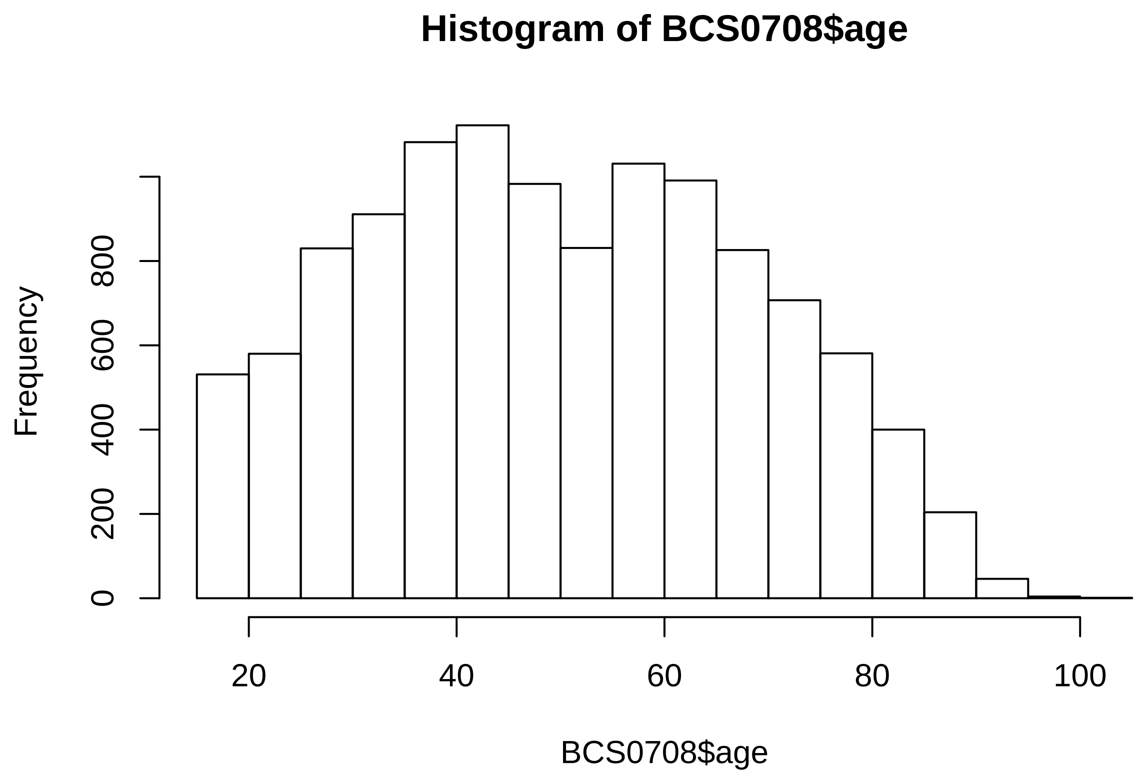
Yet, even if you have use a short name to designate your objects, you may still have to do more typing that your, let's face it, lazy nature will want. If you have to use as inputs various variables and the function require that you use the `name_of_dataset$name_of_variable` formulation as a way to identify your variables, it can be a bit tedious to do so. An incredibly helpful way of getting around that is using the `with()` function.

```
#Ok, we don't save so much typing with this example, but imagine you have 10 variables!
with(BCS0708, describeBy(age, livharm1))
```

```
## group: cohabiting
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 1032 38.27 12.86     36   36.93 11.86  18  95    77 0.95     0.72
##    se
## 1 0.4
## --------------------------------------------------------
## group: divorced
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 1055 54.28 12.61     55   54.14 13.34  24  95    71 0.09    -0.49
##     se
## 1 0.39
## --------------------------------------------------------
## group: married
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 5551 52.97 14.79     53   52.74 16.31  18  94    76 0.11    -0.84
##    se
## 1 0.2
## --------------------------------------------------------
## group: separated
##   vars   n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 323 47.66 13.64     45   46.64 13.34  24  86    62 0.63    -0.17
##     se
## 1 0.76
## --------------------------------------------------------
## group: single
##   vars    n  mean    sd median trimmed   mad min max range skew kurtosis
## 1    1 2372 34.65 16.63     30   32.31 14.83  16  92    76 1.09     0.48
##     se
## 1 0.34
## --------------------------------------------------------
## group: widowed
##   vars    n  mean    sd median trimmed   mad min max range  skew kurtosis
## 1    1 1320 75.09 10.24     76   75.77 10.38  37 101    64 -0.68     0.54
##     se
## 1 0.28
```

Next week we will devote the entire session to cover graphics as a way of visualising data. But it is important you understand that although we separate the treatment of this topic for practical reasons, in practice visualising the data is one of the very first things you do (even before you produce numerical summaries). So, if you are working with a numerical variable, you may want to produce a histogram to quickly look at the full distribution:

```
hist(BCS0708$age)
```



1.8 Summarising categorical variables

How about characterising qualitative variables? You can use the `table()` function to produce a frequency distribution. Let's look at how safe people feel walking along in the area where they live when is dark:

```r
table(BCS0708$walkdark)
```

```
##
## a bit unsafe  fairly safe    very safe  very unsafe
##         2604         4718         3002         1301
```

You can see the modal category is fairly safe. Notice that R by default does not print the missing values. If you want them printed when using this function then you have to specifically ask for them with the `useNA` argument:

```r
table(BCS0708$walkdark, useNA = "ifany")
```

```
##
## a bit unsafe  fairly safe    very safe  very unsafe         <NA>
##         2604         4718         3002         1301           51
```

We may also be interested in expressing these quantities in percentage values. In order to do so, we need to engage in a programming trick. First we create a new object and we use the results from this object to compute those quantities. We do this using the following instructions:

```
.Table <- table(BCS0708$walkdark) ##creates a new object called .Table containing the results we saw be
round(100*.Table/sum(.Table), 2)  ##percentages for ethnic4 created using the object called .Table we j
```

```
##
## a bit unsafe  fairly safe    very safe  very unsafe
##        22.40        40.58        25.82        11.19
```

```
remove(.Table) ##removes this object from working memory
```

When producing frequency distribution tables, it is particularly useful (if you want to save yourself the work of formatting the tables in a nice way) to use the sjPlot package. You can use the `sjt.frq()` function for this. This online tutorial explains some of the features of this function in greater detail.

```
library(sjPlot)
sjt.frq(BCS0708$walkdark, variableLabels = c("How safe you feel walking alone at dark?"))
```

How safe you feel walking alone at dark?

value

N

raw %

valid %

cumulative %

a bit unsafe

2604

22.30

22.40

22.40

fairly safe

4718

40.41

40.58

62.98

very safe

3002

25.71

25.82

88.81

very unsafe

1301

11.14

11.19

100.00

missings

51

0.44

total N=11676 · valid N=11625 · $\bar{x}$=2.26 · =0.93

1.9 Saving your work and further resources

After you spend some time working with R you may want to save your progress. Equally if you create, alter or obtain data (e.g., from an external repository) you may want to store the data in a file saved in either your P: drive, a USB memory stick (whichever you prefer to work from).

The first thing you want to do is to create a folder in either your P: drive, the hard drive of your home computer, or a USB drive for these datasets and the exercises that you will be performing for this course. You may call this folder "R Exercises" or something like that. In future you may want to have a folder for every research project you are working on.

Whatever folder or location you use for the materials of this course, it is convenient you set this folder as your **working directory** whenever you start your session, otherwise every time you need to use a data frame you will have to explain R where to find it. When you start your R session, R will be working from a pre-specified working directory. Which working directory this is will depend on the machine you are using, and whether you are working from a previously saved project in R Studio.

To find out what working directory you are currently working from, you can use the following code:

```r
getwd()
```

In order to use a more convenient directory (like the one you may want to create for this course), you can use the `setwd()` function. This is how I set the working directory for my R related teaching materials for this course:

```r
setwd("C:/Users/jjmedina/Dropbox/Teaching/1 Manchester courses/70821 Intro to Statistics/R Materials/RS
```

The argument, what you find between parentheses, is simply the particular location that *in my case* I want to use as such. You will have to adapt this to whatever location you want to work from. In order for R to find and save in the right place your data and any programs ("scripts"), or graphics that you produce *it is important that you start any R session setting up this working directory.*

Once you have created a folder and have set it up as the working directory you can save your progress. You can do this using the following code:

```r
save.image("FirstWeek.RData")
```

This will create an **image** with all the various objects that are in your workspace in the named file. You can use other name rather than "FirstWeek", but it is important that you make sure you specify that this is a .RData type of file. This will tell R this file is a workspace file. In any case, if you close your R Studio session you will be automatically asked whether you want to save the current workspace image. Watch this video if any of this is not clear.

You can also save particular objects in your working space, say one of your data frames, as separate file. The following code will save the `BCS0708` file as a .RData file. Note, of course, that you can also specify a suitable physical location.

```r
save("BCS0708", file = "C:/Users/jjmedina/Documents/BCS0708.RData")
```

Once the file is saved as a .RData file you can then load it simply with the `load()` function.

```r
load("C:/Users/jjmedina/Documents/BCS0708.RData")
```

However, some people would advise to save data files as such. So rather than creating a .RData file a preferable option may be to use the `write.csv()` option, which will create a comma separated file with your data frame. See this for details.

If you want to write your data frame into a different format, there are a number of ways for doing so. I won't be expecting you to do so, but you can learn how to do it: ; for dbf, SPSS, SAS, STATA; and for Excel.

The many expressions that you may have been using are better stored within a script. This video, from the excellent series produced by MarinStats (Mike Marin at University of British Columbia), explains how to work with scripts within R Studio. I highly recommend that you watch it.

Also, in order to consolidate what we have covered today, you may also want to watch his other short videos (typically 6 minutes) that demonstrate the sort of stuff we have covered today. Specifically his videos on:

- Getting started with R
- Creating vectos and matrices
- Importing data into R
- Getting started with Data in R
- Working with variables and data in R
- Subsetting data
- Customising the look of R Studio

Alternatively you may find as helpful reference the online handouts produced by Andy Teucher. They are more parsimonious than this handout and can be used as a cheat-sheet for the type of functions we introduced today. If you want further practice you may enjoy the interactive modules in Try R. And for further documentation, the Beginner's Guide to R guide of Computerworld magazine, is pretty handy and you will find there as well a very comprehensive annotated list of additional learning resources for R.

Important terms introduced today (if you are not clear about them, please read the document again or ask me about their meaning: + Code + Object-oriented language + Expressions + Assignment operator + Debugging + Annotations + Functions + Arguments + Vectors + Vector class + Data frames + Global environment + Coding conventions + Sub-setting + Logical operators + Codebook + Metadata + Data munging + Missing data + Summary statistics + Central tendency + Variability + Mean, Median, Mode + Standard Deviation + Range + Quantiles + Interquartile range + Quantitative variables + Categorical variables + Frequency table + Working directory

```r
sessionInfo() #This function provides information about the version of R I used to produce the html fil
```

```
## R version 3.2.2 (2015-08-14)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 8 x64 (build 9200)
##
## locale:
## [1] LC_COLLATE=English_United Kingdom.1252
## [2] LC_CTYPE=English_United Kingdom.1252
## [3] LC_MONETARY=English_United Kingdom.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United Kingdom.1252
##
## attached base packages:
## [1] grid      stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] sjPlot_1.8.3     psych_1.5.8      pastecs_1.3-18   boot_1.3-17
##  [5] Hmisc_3.16-0     Formula_1.2-1    survival_2.38-3  mosaic_0.11
##  [9] mosaicData_0.9.1 car_2.1-0        ggplot2_1.0.1    lattice_0.20-33
## [13] dplyr_0.4.3
##
## loaded via a namespace (and not attached):
##  [1] reshape2_1.4.1      splines_3.2.2       colorspace_1.2-6
##  [4] htmltools_0.2.6     yaml_2.1.13         mgcv_1.8-7
##  [7] nloptr_1.0.4        foreign_0.8-65      DBI_0.3.1
## [10] RColorBrewer_1.1-2  plyr_1.8.3          stringr_1.0.0
## [13] sjmisc_1.1          MatrixModels_0.4-1  munsell_0.4.2
## [16] gtable_0.1.2        evaluate_0.7.2      latticeExtra_0.6-26
## [19] knitr_1.11          SparseM_1.7         quantreg_5.19
## [22] pbkrtest_0.4-2      parallel_3.2.2      proto_0.3-10
## [25] Rcpp_0.12.1         acepack_1.3-3.3     scales_0.3.0
## [28] formatR_1.2         lme4_1.1-9          gridExtra_2.0.0
## [31] mnormt_1.5-3        digest_0.6.8        stringi_0.5-5
## [34] tools_3.2.2         magrittr_1.5        lazyeval_0.1.10
## [37] cluster_2.0.3       ggdendro_0.1-17     tidyr_0.3.1
## [40] MASS_7.3-43         Matrix_1.2-2        assertthat_0.1
## [43] minqa_1.2.4         rmarkdown_0.8       R6_2.1.1
## [46] rpart_4.1-10        nnet_7.3-10         nlme_3.1-121
```