Exercise 1: Data Manipulation, Exploration and Visualisation in R

Data, Algorithms and Meaning (*Week of 19/03/2018*)

R Introduction

Installation

R is a free software environment for statistical computing and graphics. It has become one of the major tools for data scientists worldwide, largely due to its huge range of powerful packages, state of the art methods, and huge user community.

Make sure that you first have R installed. You can download R on mac, windows or linux here: http://cran.csiro.au

We recommend RStudio as an IDE for R (Integrated Development Environment). You can download it here: https://www.rstudio.com/products/rstudio/download/

There is a wealth of information and tutorials online regarding R and Rstudio. Feel free to browse the following links:

- A (very) short introduction to R: https://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf
- Introduction to R and RStudio video: https://www.youtube.com/watch?v=lVKMsaWju8w
- RStudio introduction video: https://www.youtube.com/watch?v=jPk6-3prknk

The rest of this section will cover the basics of package management, working directories and reading / writing data from disk, basic essentials to working with R.

Package Management

At the time of writing, R has 10,164 additional packages that can be installed. Some of these packages are extremely useful, while others are incredibly specific on their application. This huge amount of extended functionality is not installed by default when you first install R. Instead, you need to manually install each package when you wish to use it. There are two steps in this process. To illustrate, we will install Hadley Wickam's celebrated ggplot2 package.

The first step is to install the package to your local package directory using **`install.packages("")`**:

```
install.packages("ggplot2")
```

This downloads and unpacks the relevant files to your local package repository. You only ever need to run this once on your machine. There is an additional step before we can use this package: we must load it to memory.

```
library(ggplot2)
require(ggplot2)
```

Both of these commands are equivalent. Note that when we install a package, we must pass the package name in double quotes, but this is not required to load the package to memory. We will return to the ggplot2 package later in the second exercise.

Working Directory

We need to tell R explicitly where on our hard drive we want to work. The **`setwd()`** command will do this. For example,

```
# Mac and Linux
setwd("~/DAM/Pre-work")
# Windows
setwd("C:\\User\\(username)\\Documents\\DAM\\Pre-work") # edit path as needed!
```

To check that we have set our working directory correctly, we can run **`getwd()`**:

```
getwd()
```

To list all the files in our working directory, we can run **`list.files()`**:

```
list.files()
```

```
## [1] "diamonds.csv"        "README.md"           "R_exercise_1.pdf"
## [4] "R_exercise_1.Rmd"    "R_exercise_2.pdf"    "R_exercise_2.Rmd"
## [7] "R_exercise_2_v2.pdf" "R_exercise_2_v2.Rmd" "Rstudio.png"
```

Notice that the list of files in our working directory is stored as a vector. This can come in handy if we wish to import a whole set of files, one after the other. Also note that your list of files will be different!

R Basics

In this exercise, we are going to introduce some of R's core concepts. Open up RStudio. You should see a screen like that shown in Figure 1. For this exercise, you should type the R commands given in the console, then press ENTER to execute the command.

Values, Objects and Vectors

In the console, type the number 1 as below and press enter. You should see the output displayed after **##** below:

```
1
```

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

What you have just done is told R to display the value 1, as a number. R prints the value `1`, next to the index `[1]`. What is this index about? The most basic data object in R is a vector. When you told R to display the number `1`, it interpreted this as "display the number 1 as an element in a 1x1 vector". The index `[1]` indicates the position of the value being displayed in the vector object.

To create a vector with more than one element, we use the concatenate command, **`c()`**. For example, to create a 3 element vector of numbers 1, 2 and 3, we run

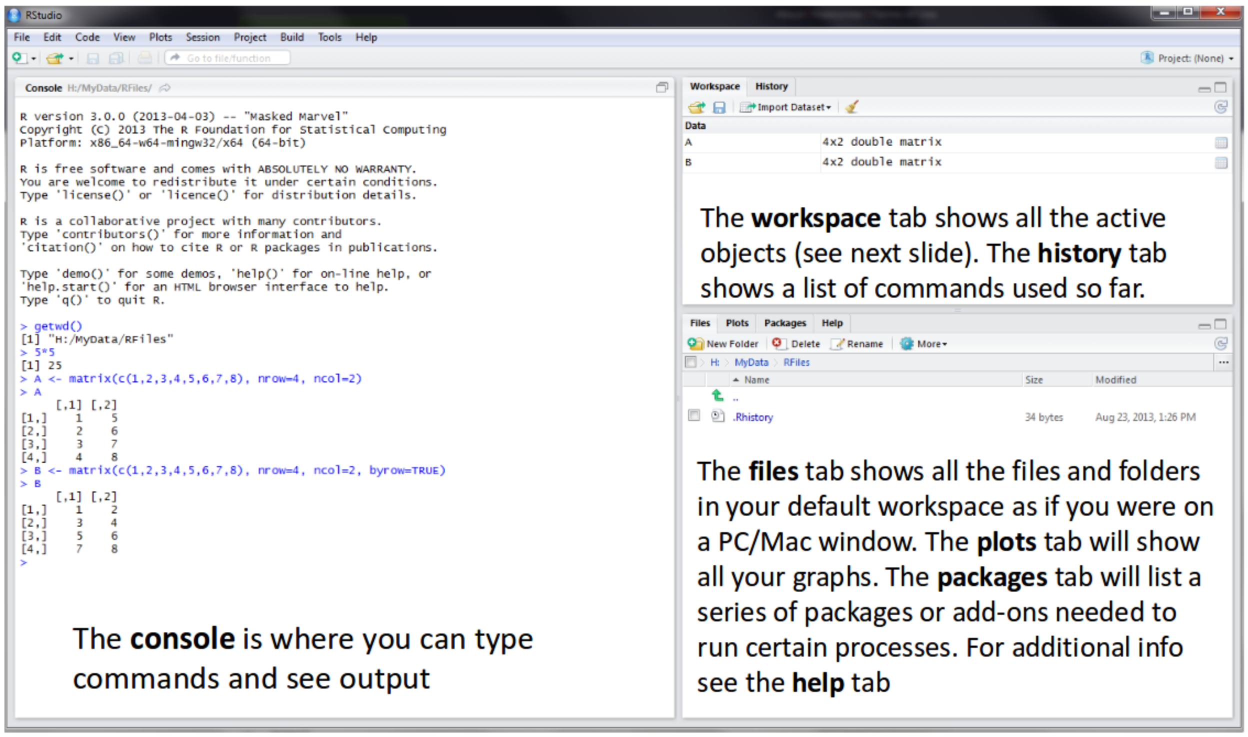
```
c(1, 2, 3)
```

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

The index only gets displayed for each row of data output, which is why we only see `[1]`. We can also create a vector of characters by using double quotes:

```
c("a", "b", "c")
```

```
## [1] "a" "b" "c"
```



```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

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.

> getwd()
[1] "H:/MyData/RFiles"
> 5*5
[1] 25
> A <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2)
> A
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> B <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2, byrow=TRUE)
> B
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
>
```

The **console** is where you can type commands and see output

Data		
A	4x2 double matrix	
B	4x2 double matrix	

The **workspace** tab shows all the active objects (see next slide). The **history** tab shows a list of commands used so far.

Name	Size	Modified
..		
.Rhistory	34 bytes	Aug 23, 2013, 1:26 PM

The **files** tab shows all the files and folders in your default workspace as if you were on a PC/Mac window. The **plots** tab will show all your graphs. The **packages** tab will list a series of packages or add-ons needed to run certain processes. For additional info see the **help** tab

Figure 1: Rstudio GUI - three key quadrants

Why is it important that R works with vectors? Because R will execute commands over all elements of the vector. For instance, adding the number 1 to the vector (1, 2 ,3) gives the following:

```r
c(1, 2, 3) + 1
```

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

R interprets this command as adding the vector of 1's to the vector (1, 2, 3). We can also execute other basic mathematical operations over a vector in a similar way. E.g.,

```r
c(1, 2, 3) * 2
```

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

Here R will multiple each element in the vector by 2. For those who remember their linear algebra, we can also do vector and matrix multiplication easily in R by using the '%' notation. To refresh your memories, the dot product is the sum of the product of the corresponding elements of each vector. For example, the dot product of vectors (1, 2, 3) is below (the dot product calculation is illustrated in the comment).

```r
c(1, 2, 3) %*% c(1, 2, 3) # = 1*1 + 2*2 + 3*3
```

```
##      [,1]
## [1,]   14
```

Finally, there are some great functions to generate useful vectors in R. The **seq()** function generates a sequence of numbers from a starting number, to an ending number, by a fixed interval. E.g.

```r
seq(from = 1, to = 20, by = 1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

The inputs to a function in R are also known as 'arguments'. To call a function you simply type the function name, followed by parentheses, with arguments given within the parantheses. In the example above, we have explicity named the arguments (`from = 1, to = 20, ...`). When using a function for the first time (or even one you've used many times but forgotten some of the arguments), it is handy to bring up the associated help file. To bring up the help documentation for a function in R, you can either run **`help()`**, or put a **`?`** in front of the function name. Both are equivalent:

```
help(seq)
?seq
```

There is a commonly used shorthand for **`seq()`** where the **`by`** increment is always 1:

```
1:20
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

We can also create a vector of a value replicated any number of times using **`rep()`**:

```
rep(x = 1, times = 20)
```

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

Data Frames

The most commonly used object in R is the data frame. Put simply, a data frame is a matrix of vectors arranged as columns. You can think of a data frame as a spreadsheet or a database table, where every column must be of the same data type. Here is an example of a data frame,

```
data.frame(1:10, seq(1, 20, 2), rep("letters", 10))
```

```
##    X1.10 seq.1..20..2. rep..letters...10.
## 1      1             1            letters
## 2      2             3            letters
## 3      3             5            letters
## 4      4             7            letters
## 5      5             9            letters
## 6      6            11            letters
## 7      7            13            letters
## 8      8            15            letters
## 9      9            17            letters
## 10    10            19            letters
```

Note that the arguments to functions like **`seq()`** and **`rep()`** do not need to be named explicitly (e.g. `from = ...`). However, for functions if you do not name them then you must input them in the same order as the documentation. In this case, since we are defining a new data frame, R has given each column (or variable) a name in line with the vector definition. We usually rename the variables in our data frames to something nicer. There are a two equivalent ways to do this:

Using **`setNames(data_frame)`** with our data frame as the input, we can assign a name to each vector input to the data frame.

```
setNames(object = data.frame(1:10, seq(1, 20, 2), rep("letters", 10)),
         nm = c("seq_1","seq_2", "letters"))
```

We can assign variable names when we create the data frame by naming each of our column vectors:

```
data.frame(seq_1 = 1:10, seq_2 = seq(1, 20, 2), letters = rep("letters", 10))
```

```
##    seq_1 seq_2 letters
## 1      1     1 letters
## 2       2     3 letters
## 3       3     5 letters
## 4       4     7 letters
## 5       5     9 letters
## 6       6    11 letters
## 7       7    13 letters
## 8       8    15 letters
## 9       9    17 letters
## 10     10    19 letters
```

Both of these methods will give us the same result.

Built in Data Sets

R comes with a number of data sets preloaded. To view these data sets, type the following command into the console.

```
data()
```

You should see a table of data sets appear, with the data set name in the left column, and a short description in the right column. Because these data sets are built into the core R environment, simply typing the name of the data set will display the results in the console. For example, type the following into the console and press enter:

```
AirPassengers
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1949 112 118 132 129 121 135 148 148 136 119 104 118
## 1950 115 126 141 135 125 149 170 170 158 133 114 140
## 1951 145 150 178 163 172 178 199 199 184 162 146 166
## 1952 171 180 193 181 183 218 230 242 209 191 172 194
## 1953 196 196 236 235 229 243 264 272 237 211 180 201
## 1954 204 188 235 227 234 264 302 293 259 229 203 229
## 1955 242 233 267 269 270 315 364 347 312 274 237 278
## 1956 284 277 317 313 318 374 413 405 355 306 271 306
## 1957 315 301 356 348 355 422 465 467 404 347 305 336
## 1958 340 318 362 348 363 435 491 505 404 359 310 337
## 1959 360 342 406 396 420 472 548 559 463 407 362 405
## 1960 417 391 419 461 472 535 622 606 508 461 390 432
```

The data set called `AirPassengers` is a time series of air passenger numbers per month. There are many more data sets built in which we will work on. The command **`data()`** only lists a subset. To list all the data, we add an argument to the function `data()` as follows:

```
data(package = .packages(all.available = TRUE))
```

Viewing Data

Listed in this new output is a data set called `mtcars`, described as 'Motor Trend Car Road Tests'. Let's have a look at the top 6 rows of this data set (using the **`head()`** function):

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

This data set has a variety of data types, including the car name as a character string, numeric values, categorical values and ordinal values. Given that we have a number of different variables each with a different data type, this data set is probably stored as a data frame. To see that this data is actually stored as a data frame, we can examine the structure of the `mtcars` data set using the **`str()`** function:

```
str(mtcars)
```

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

The first line of output tells us that this data set is a `'data.frame'`, with `32 observations` (rows) and `11 variables` (columns). The rest of the output gives each variable name (preceded by a `$`, which we'll come back to), the data type (`num`), and the top 10 observations. Normally someone would now ask 'What happened to the car names?'. Good question! In this data set, they are actually stored as row names, rather than a specific variable. Just as variables have names, in R rows have names too. It's easy to list them both:

```
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```
colnames(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

We can also produce a **`summary()`** of the data set. This function produces a number of descriptive statistics for each variable:

```
summary(mtcars)
```

```
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

Data Types

Once we've examined the data frame, the next step in any analysis is to ensure that each variable is stored in the correct data type. Even though all of our variables are numbers and are stored as `num`, we don't necessarily want to work with them all as numeric variables. For instance, the column `cyl` refers to the number of cylinders of the vehicle. This is rightly an ordinal variable, meaning that it is categorical with distinct values, but its order has meaning. On the other hand, `am` refers to the transmission of the vehicle. It is 1 if the car is an automatic, 0 if manual. This is a binary flag, so is categorical.

In R, the most common data types that you will come across are numeric (`num`), integers (`int`), character strings (`char`), `date` and `factor`. Sometimes, if a variable is numeric or integer but is actually ordinal, we can choose to represent it as a number that we can calculate on, or as a factor (i.e. categorical variable). In the `mtcars` example, we will choose to store all ordinal variables as factors for reasons that will become clear.

We will now modify the data to our desired data types. Firstly, let's copy the data set to our working environment. We will store the `mtcars` data to a new data frame called `data`. There are two ways to do this in R, both very similar: We can use the operator `=`, or `<-`. The main difference between the two operators is that using `<-` has direction, i.e. we are assigning the value on the right hand side to the object on the left hand side. We can also reverse this operation by using `->`, which assigns the value on the left to the object on the right Which one you use is up to you, but for the rest of this tutorial we will use the `=` operator (requires only typing one character):

```r
data <- mtcars # LHS assignment
mtcars -> data # RHS assignment
data = mtcars # LHS assigment - requires only one keystroke, not three :)
```

We have now created a data frame called `data` in our working environment. To refer to a specific variable of a data.frame, we can use the `$` operator after the data frame name. We can then assign new values, so long as we are assigning the same number of values. We are going to convert the variable `cyl` to a factor, making it categorical. We can do this with the `as.factor()` function:

```r
data$cyl = as.factor(data$cyl)
```

You can check that the change was made by running the `str()` function on this variable:

```r
str(data$cyl)
```

```
##  Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
```

R now tells us that the variable is a factor with 3 levels (i.e. values) "4", "6", "8". These values are encoded as increasing integers starting from 1; values "4", "6", "8" become codes 1, 2, 3. Note that R treats the levels of a factor variable as character strings.

Now we will convert the other categorical variables to factors:

```
data$vs = as.factor(data$vs)
data$am = as.factor(data$am)
data$gear = as.factor(data$gear)
data$carb = as.factor(data$carb)
```

Lastly, let's create a new variable for the car name. Up until now we have just been overwriting existing variables, but creating a new variable is easy - we simply assign values to a new variable name on the data frame **data** as follows. Note that the following works since **rownames(data)** has the same number of values as the data frame has rows:

```
data$carname = rownames(data)
```

We can now remove the row names by assigning the value of NULL:

```
rownames(data) = NULL
```

To check that everything has worked, we can run the **str()** or **summary()** functions on our data. Let's run **summmary()**, where we see that for factors the function gives us a count of each level:

```
summary(data)
```

```
##       mpg        cyl         disp             hp              drat      
##  Min.   :10.40   4:11   Min.   : 71.1   Min.   : 52.0   Min.   :2.760  
##  1st Qu.:15.43   6: 7   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080  
##  Median :19.20   8:14   Median :196.3   Median :123.0   Median :3.695  
##  Mean   :20.09          Mean   :230.7   Mean   :146.7   Mean   :3.597  
##  3rd Qu.:22.80          3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920  
##  Max.   :33.90          Max.   :472.0   Max.   :335.0   Max.   :4.930  
##        wt             qsec       vs     am     gear   carb  
##  Min.   :1.513   Min.   :14.50   0:18   0:19   3:15   1: 7  
##  1st Qu.:2.581   1st Qu.:16.89   1:14   1:13   4:12   2:10  
##  Median :3.325   Median :17.71                 5: 5   3: 3  
##  Mean   :3.217   Mean   :17.85                        4:10  
##  3rd Qu.:3.610   3rd Qu.:18.90                        6: 1  
##  Max.   :5.424   Max.   :22.90                        8: 1  
##    carname         
##  Length:32         
##  Class :character  
##  Mode  :character  
##                    
##                    
## 
```

Subsetting

To take a subset of a vector in R, we use square brackets to the right of our object (**vector[ ]**). Within the square brackets, we enter the vector element indices we wish to subset. This index always starts from 1. For example, lets create a vector called x with values of 1 to 10, incremented by 1. We will then use the square bracket notation to extract the 3rd, 4th, 5th and 9th values. Note how we can create a concatenated vector of indices (c(3:5, 9)):

```
x = 1:10
x[c(3:5, 9)]
```

```
## [1] 3 4 5 9
```

When working with data frames, we commonly wish to extract subsets of rows or columns. There are multiple ways of doing this in R. Firstly, the rows and columns of a data frame have a numeric index associated with each. Similarly to vectors, we use the square bracket notation to subset by indices. To subset data frames, we must always have a comma within the square brackets which denote the row and column indices. The syntax is to put rows as the first input, columns as the second (i.e. `data[row index, column index]`).

For example, to view the first 6 rows of the first column of our data frame, we would run:

```
head(data[, 1])
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

Note that this is the same as running `head(data$mpg)`, but without having to specify the name of the first column. To view the first 3 columns, we can use the sequence function:

```
head(data[, 1:3])
```

```
##    mpg cyl disp
## 1 21.0   6  160
## 2 21.0   6  160
## 3 22.8   4  108
## 4 21.4   6  258
## 5 18.7   8  360
## 6 18.1   6  225
```

To view the first 3 rows of the first 3 columns, we would simply execute:

```
data[1:3, 1:3]
```

```
##    mpg cyl disp
## 1 21.0   6  160
## 2 21.0   6  160
## 3 22.8   4  108
```

The other common way to subset data frames is to explicitly use the row or column names. Remember that in R, every value is defined as a vector. If we want to subset by multiple rows or columns, we must give their names as a vector (using `c()` or another function which outputs a vector). For instance,

```
data[1:3, c("mpg", "cyl", "disp")]
```

```
##    mpg cyl disp
## 1 21.0   6  160
## 2 21.0   6  160
## 3 22.8   4  108
```

We removed the rownames of our data frame earlier, so R defaults to numeric indices.

Reading and Writing Data

Let's read in some data from disk. You should download the file **diamonds.csv** from UTS Canvas. The two main functions to read in data are `read.csv()` and `read.table()`. The key difference is that `read.csv()` can only read csv files, while `read.table()` can handle other formats, such as txt files, but also requires more arguments. We can read our file in one of two ways. Note that we are assigning the data to an object called `data` (`data = read...`):

```r
diamonds_data = read.csv("diamonds.csv", header = TRUE)
diamonds_data = read.table("diamonds.csv", sep = ",", header = TRUE)
```

The first argment is the file name. We can actually specify the full path to the file on our hard drive, but if our file is stored in our working directory we can just provide the file name. The argument `header = TRUE` tells R that our file has a row of column headings. In `read.table()`, the argument `sep = ","` is telling R that our column delimiter is a comma; if our file was tab delimited, we would use `sep = "\t"` instead.

To export data, we can similarly use the following functions:

```r
write.csv(diamonds_data, file = "diamonds.csv")
write.table(diamonds_data, file = "diamonds.csv", sep = ",")
```

We can also write to other formats, such as excel. However, we generally need to install additional packages to do this.

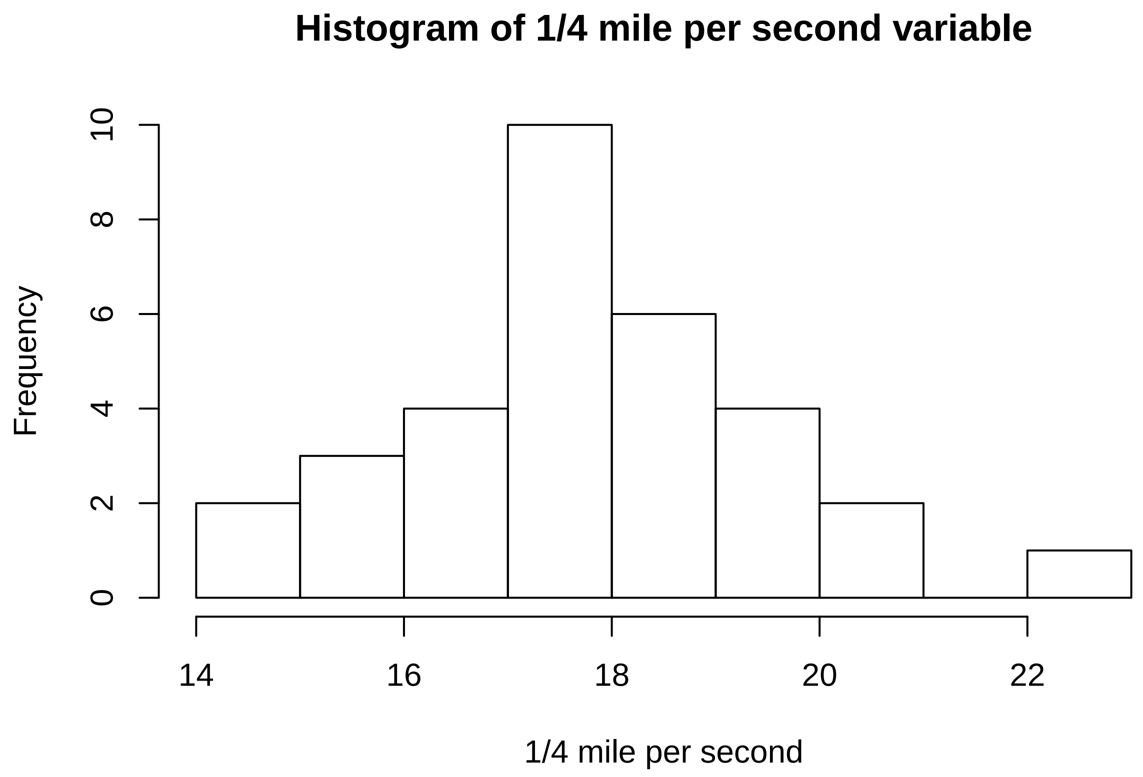
Visualisation

One of the key advantages of R over other data science languages is its extensive visualisation libraries. There are several systems within R for plotting data. The two most commonly used are the base R plotting functions, and the ggplot packages. We will start by exploring the base R plotting functions first.

Base R Plots

Let's start by exploring the histogram plotting function, **`hist()`**. A histogram plots the distribution of a variable. Values are binned together into ranges, and the freqency of values in each range are counted. Let's create a histogram plot on the variable `qsec` (1/4 mile time). Note that only the first argument (`data$qsec`) is required, **`main`** renames the title and **`xlab`** renames the **`x`** axis. The line spacing is for readability, it does not affect the output.

```r
hist(data$qsec,
     main = "Histogram of 1/4 mile per second variable",
     xlab = "1/4 mile per second")
```



Since we didn't specify any additional arguments, R has chosen to group the `qsec` variable into bins of value 2 units each, and counted the number of vehicles in each bin (frequency). The histogram shows that the `qsec` variable is centred around a value of just below 18 miles per second, and the distribution looks quite symmetric (i.e. the mean and the median are similar). We can confirm this by running the **`summary()`** command on the same variable. Note that a histogram is a plot of the distribution of a variable, while the measures in the **`summary()`** command aim to describe the distribution with summary statistics

```
summary(data$qsec)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   14.50   16.89   17.71   17.85   18.90   22.90
```

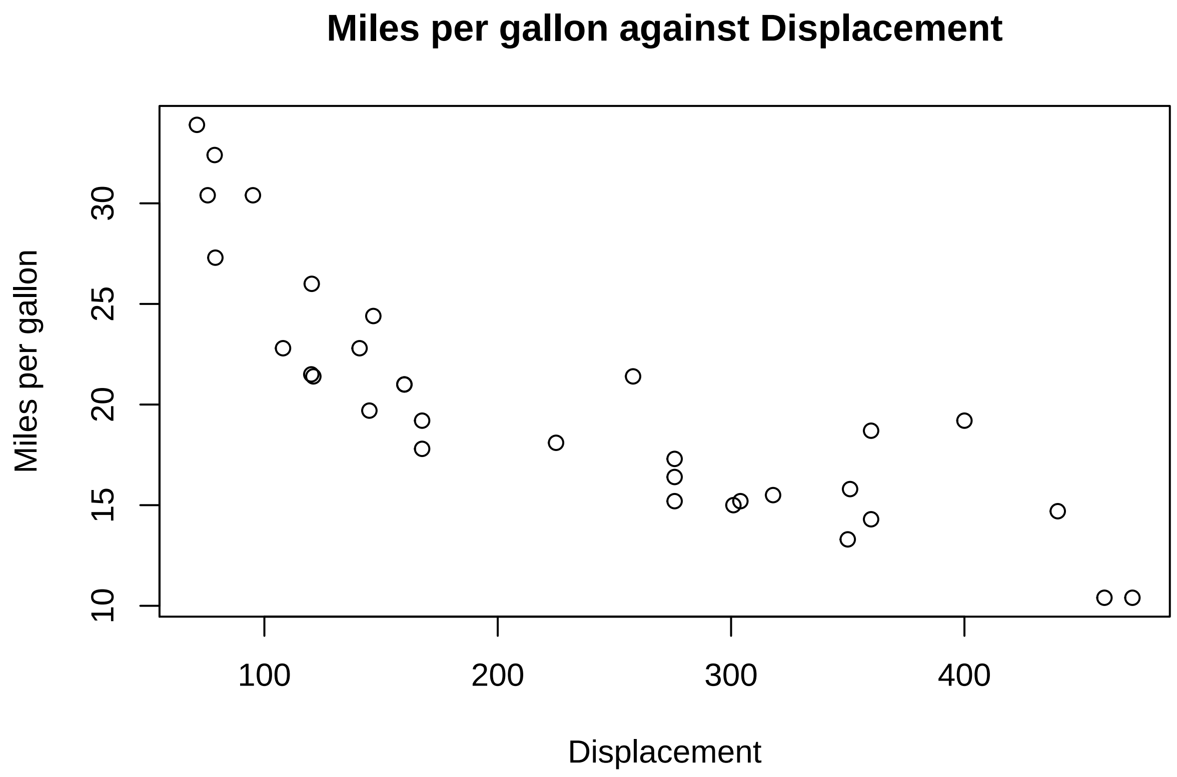
As expected, the mean and the median are both very close and are just below 18.

Histograms are good for plotting the distribution of a single variable, but commonly we want to plot one variable against another one (or another two, three, . . .). The simplest way to do this is to create a scatterplot using the **`plot()`** function. Lets start by looking up the help file:

```
?plot
```

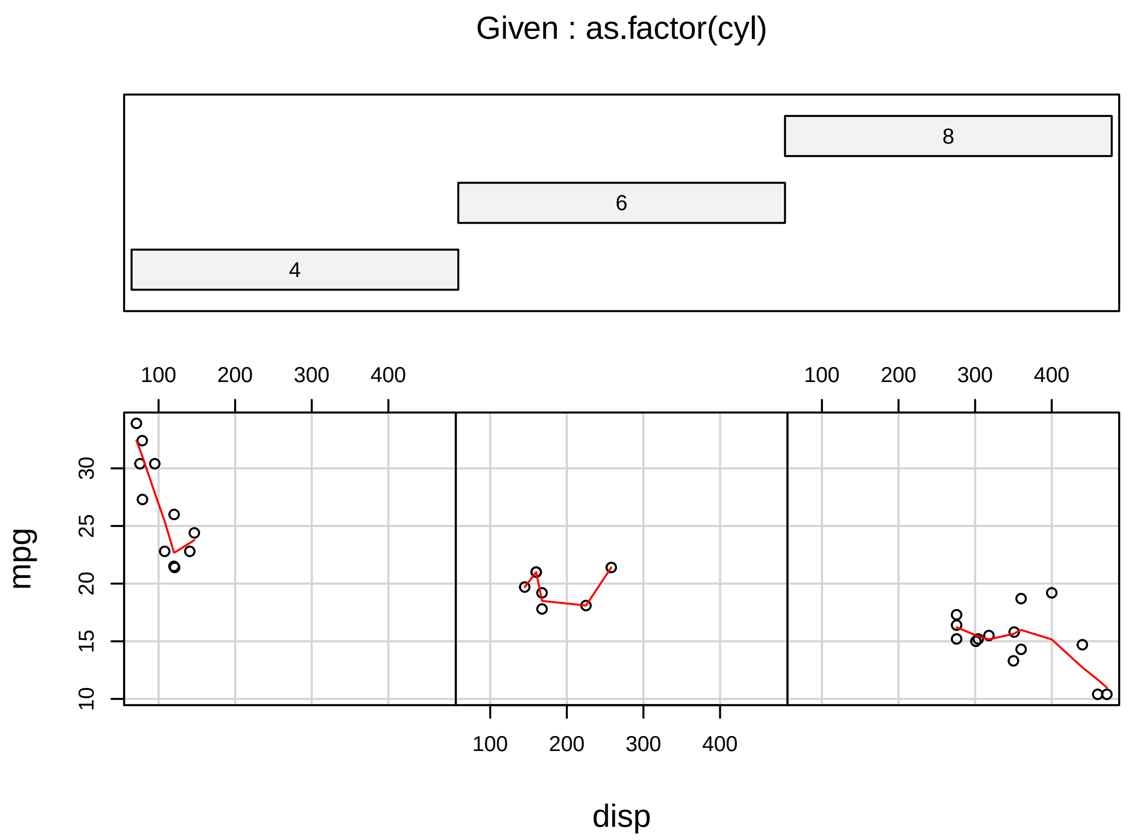
Under the **Usage** section, we see that **`plot()`** takes two arguments, **`x`** and **`y`**. There are also some optional arguments that can modify the type of plot. The default is to produce a scatterplot. Let's try this with the variables **`mpg`** (miles per gallon) and **`disp`** (engine displacement). Note that we can always specify explicity which argument we are giving R (e.g. `x = ..., y = ...`), or we can get lazy and not specify the argument as long as the order in which we supply the arguments is the same as the order in the function's documentation.

```
plot(x = data$disp, y = data$mpg,
     main = "Miles per gallon against Displacement",
     xlab = "Displacement",
     ylab = "Miles per gallon")
```



From the plot, it is clear that there is a negative relationship between miles per gallon and displacement. That is, as engine displacement increases, miles per gallon decreases. To better understand why, we should do some more visualisation. Commonly there might be another variable that is said to 'interact' with a relationship between two other variables. This means that there is actually a three way relationship or correlation rather than a correlation simply between two variables. For instance, we could hypothesise that the number of cylinders might interact with displacement and miles per gallon. In R, there is a function called **`coplot()`** that can create a faceted plot split by the levels of a factor. Let's try it:

```r
coplot(formula = mpg ~ disp | as.factor(cyl), data = data,
       panel = panel.smooth, rows = 1)
```

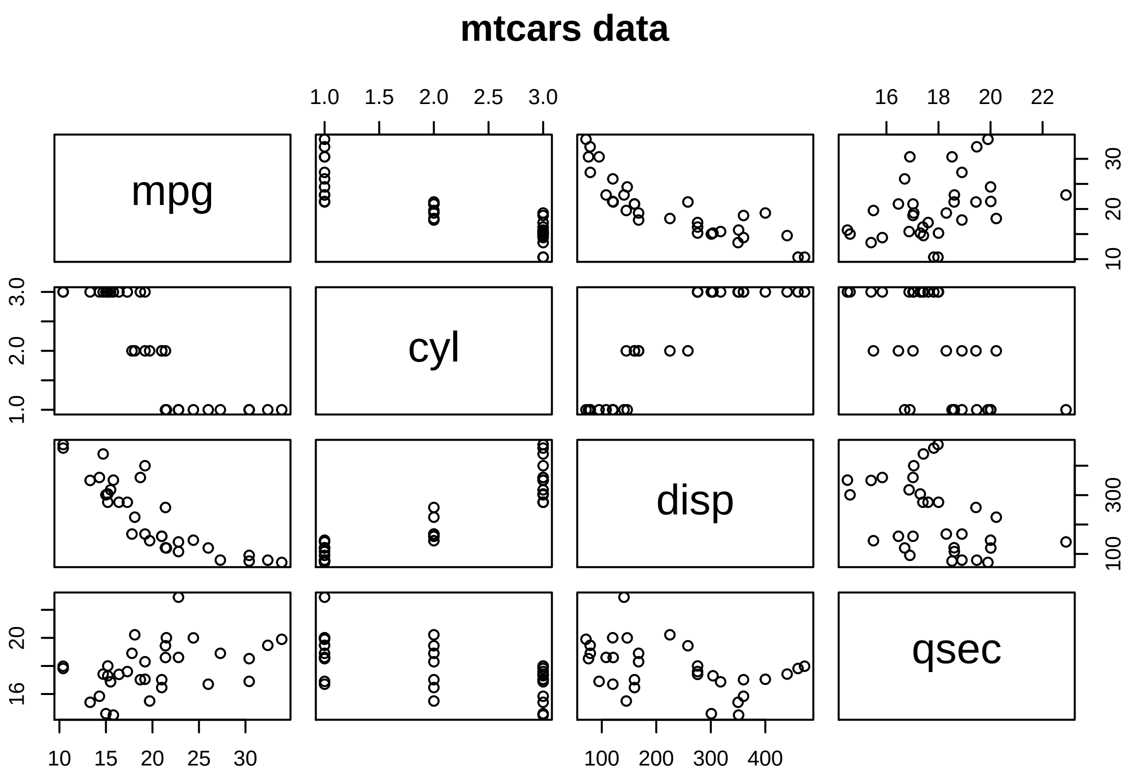


The formula argument specifies how we want the plot to be built. This argument is common to many R functions, but can be called differently for each. Always check the documentation first to verify the syntax. **`coplot()`** also takes a data argument, which means that we can just give variable names to the formula, not data.frame$variable. Finally, the panel argument is given a parameter of **`panel.smooth`**, which imputes a line of best fit to the data. We will discuss these lines in much more detail later in the course.

From this plot, it is clear that the engine displacement directly depends on the number of cylinders.

The last base R visualisation we will demonstrate is the **`pairs()`** function. This function creates individual scatter plots of all pair-wise combinations of variables in the data frame given. Below is an example using a subset of our data frame.

```
pairs(data[, c("mpg","cyl","disp","qsec")], main = "mtcars data")
```


mtcars data
mpg
cyl
disp
qsec
1.0
1.5
2.0
2.5
3.0
16
18
20
22
10
20
30
100
300
10
15
20
25
30
100
200
300
400