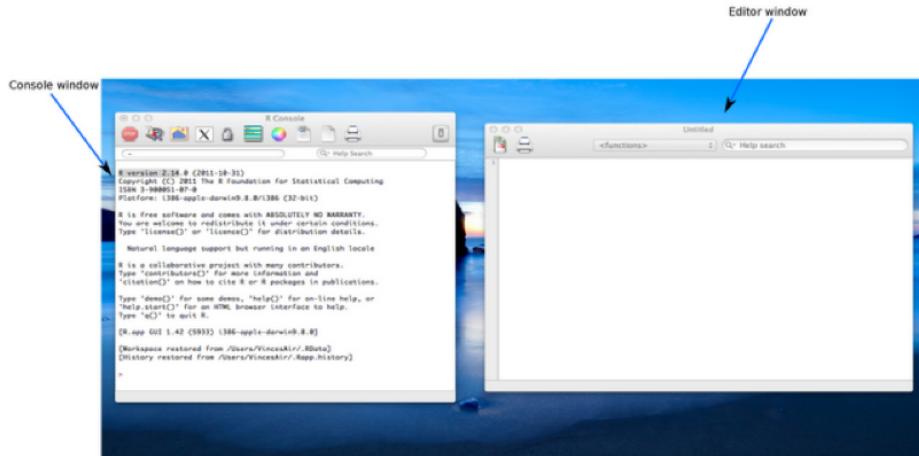


1 Chapter 1 - Introduction

1.1 The Environment

R can be run in a number of different modes, for the purpose of this course we will be focusing on ‘interactive mode’ through the graphical user interface (GUI); ‘batch mode’ is also available but will not be covered here. Note that the screenshots and accompanying screencasts for this course were produced with R version 2.14 running on Mac OSX. The look and feel on other operating systems will differ slightly.



The visible windows are:

1. The editor window
2. The console

We can write commands directly into the console window or we can create a script file and edit it in the editor window, highlighting specific text we wish to run. The second approach has the benefit of being able to save the commands written in the script files, although it takes more time (and in fact the commands we write directly in the console can also be saved to a file).

When writing scripts it is good practice to include comments in our files that help describe what the code does. The way to do this in R is with the # symbol before text. The following code is ignored by R:

```
#2+2
```

Using the highlight + run approach is akin to copying and pasting the text in the interpreter but R scripts can also be run directly (so that they can be run on servers or as routines without the need to have a user interact).

1.2 Objects

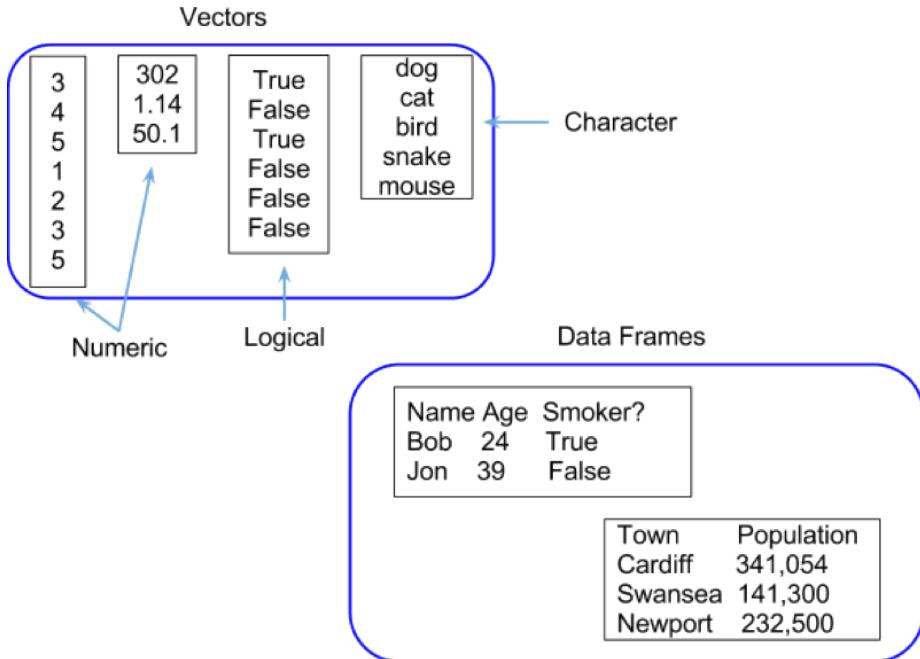
R is an extremely versatile programming language. In particular R is an “object oriented language”. The significance of this is that everything (functions, data files, outputs of a regression analysis) is an “object”. The type of object is called the “class” and what one can do to a class is called a “method”. The advantage of this is that when a new “class” is developed one simply needs to ensure that it has relevant “methods”, to be compatible with other objects.

As an example, various objects in R have a “plot” method, for example the output of a regression analysis can be plotted using the same command as one would use to plot a scatter plot of a data set.

R has a wide range of data types (which are themselves objects). The 2 classes corresponding to data sets we will concentrate on in this course are:

1. Vectors
2. Data frames

Vectors are simply collections of variables of a particular type (“Numeric”, “Character”, “Boolean” etc). In R a type of variable is called a “mode”, representing how it is stored in the computer memory. Data frames are collections of vectors and correspond to data sets. Technically, data frames are lists with dimensions, which are themselves just generic vectors. One might say a collection of equal length vectors (thus allowing the rectangular shape). Some examples of vectors and data frames are shown.



Let's import some data!

1.3 Importing Data

We will consider two approaches to importing data:

1. Direct input
2. Importing an external data set (xls, csv etc...)

In practice you will never use the direct input method but let's take a look for completeness (although it is very useful when wanting to quickly test a few things). This will also give us our first experience of the editor window!

Let us create a data set named `first_data_set` which will include the following data:

```
Name, Age
Bob, 23
Billy, 25
```

To do so write the following code in the editor window:

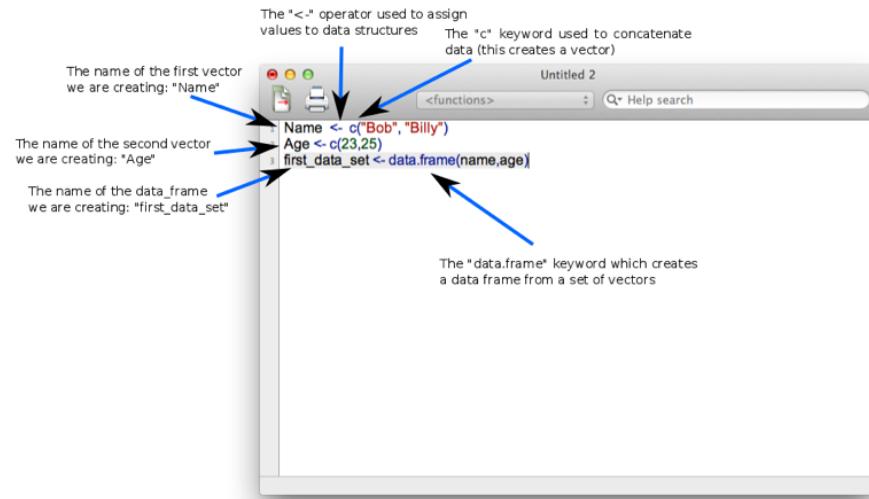
```

Name <- c("Bob","Billy")
Age <- c(23,25)
first_data_set <- data.frame(Name,Age)

```

Let's take a look at the shown screenshot of this. You may notice that some elements of the text are highlighted, this is to emphasise key words (note that this doesn't happen automatically on Windows).

1. The first two lines of code make use of the `<-` operator that assigns an object to a variable.
2. The objects in questions are created using the `c`(ombine) function that creates a vector. We use this to create 2 vectors: Name and Age.
3. Finally we put the 2 vectors into a data frame using the `data.frame` command.

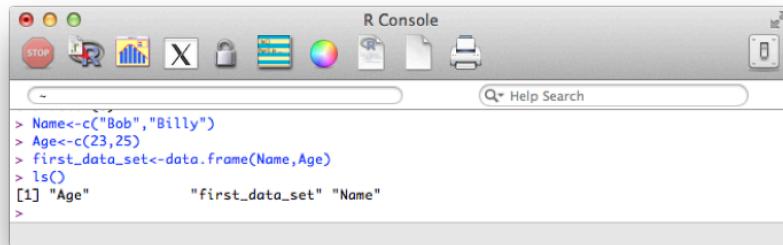


We run this code by highlighting it and pressing `ctrl + r` (.pdf + enter on Mac). Note that when we submit code this way it also appears in the console window. We could have in fact directly type this code into the console window. For those familiar with command line commands the console works in a very similar way. We can press the up arrow repeatedly to cycle through previous commands and use tab to autocomplete.

The data set `first_data_set` is now saved to memory. To view all the data structures in memory we use the simple line of code:

```
ls()
```

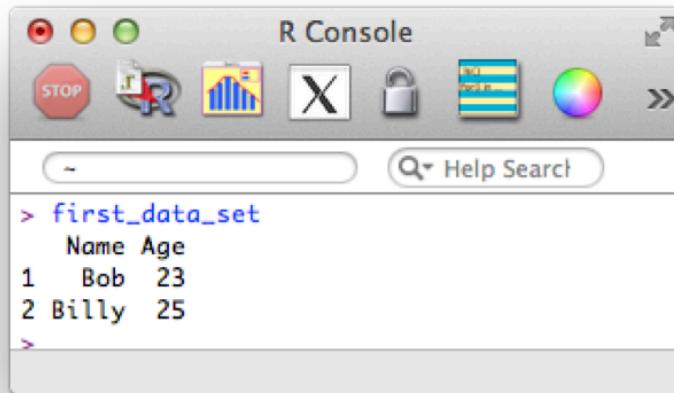
A screenshot of the output is shown. We see that there are actually 3 objects in memory, the two vectors (Name and Age) as well as the data frame (first_data_set).



```
R Console
STOP Help Search
> Name<-c("Bob","Billy")
> Age<-c(23,25)
> first_data_set<-data.frame(Name,Age)
> ls()
[1] "Age"      "first_data_set" "Name"
>
```

To view our data set, we simply type the name (as shown):

```
first_data_set
```



```
R Console
STOP Help Search
> first_data_set
   Name Age
1 Bob  23
2 Billy 25
>
```

Using direct input is of course not at all realistic when trying to import larger data sets.

Often large data sets will be saved in comma-separated values (csv) format which can be read by most (all?) software. We will import the data set shown (here viewed in a simple text editor).

```

Name,Age,Sex,Height in Metres,Weight in Kg,Home Postcode,Savings in Pounds,Random Number
John,15,M,1.72,71,CF24 3AG,1000,336.8041790091
Jo,14,M,1.85,73,CF27 4HL,500,757.197195664
Jill,21,F,1.57,49,SW6 4JL,357,458.5039406084
James,24,M,1.59,58,SW5 3JL,10930,565.9515243624
Jenny,74,F,1.63,70,BR21 4YE,465029,206.1446015723
Juliet,15,F,1.62,45,CF14 7BR,930,977.9322277755
Jackie,3,F,1.65,60,np24 3AG,10,564.2944280989
Julien,37,M,2.01,100,np24 3AG,102930,583.125016652
Joe,19,M,1.92,75,np7 5BD,2930,206.9145319983
Jeremiah,39,M,1.54,70,np15 1AE,84953,41.0423562862
Jim,17,M,1.73,83,CF35 5AS,470320,985.9272670001
Julie,2,F,1.62,45, CF72 8JY,1.5,327.5803755969
Jo,7,F,1.62,51, CF72 9DP,200,50.6616821513

```

We will import this data set into R using the following code:

```
JJJ <- read.csv(file = "~/JJJ.csv", head=TRUE)
```

Let's take a look at the screenshot shown. Note here that we are not using the text editor but directly writing code in the console (this is often how I prefer to use R for short bits of code).

1. `read.csv` - is the command which is used to tell R to read in data from a csv file.
2. `file` - an option tells R where the csv file is located.
3. `head` - an option tells R to read the variable names from the first row of the csv file. Note that this command can be omitted (the default value is `TRUE`).

We have omitted other options (such as `sep` which can be used to change the default separator from `,` to something else).

R version 2.14.0 (2011-10-31)
 Copyright (C) 2011 The R Foundation for Statistical Computing
 ISBN 3-900051-07-0
 Platform: i386-apple-darwin9.8.0/i386 (32-bit)

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

Natural language support but running in an English locale

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

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

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]
 [History restored from /Users/VincentAir/.Rapp.history]

```
JJJ <- read.csv(file = "~/JJJ.csv", head=TRUE)
read.csv(file, header = TRUE, sep = "", quote = "\"", dec = ".", fill = TRUE, comment.char = "")
```

The name of the object created R.

The read.csv command used to import csv files.

The location of the file to be imported.

The head option that tells R to read the name of the variables from the first row.

Running the code (by either pressing enter if using the console or highlighting and running as before is using the editor) gives the required object as shown.

The screenshot shows the R Console window with the following text output:

```

Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]

[History restored from /Users/VincentAir/.Rapp.history]

> JJJ <- read.csv(file "~/JJJ.csv",head=TRUE)
> objects()
[1] "JJJ"
> JJJ
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number
1  John 15   M        1.72          71    CF24 3AG      1000.0     737.0167
2   Jo 14   M        1.85          73    CF27 4HL      500.0      549.9774
3  Jill 21   F        1.57          49     SW6 4JL      357.0      509.8740
4 James 24   M        1.59          58     SWS 3JL    10930.0     749.5984
5 Jenny 74   F        1.63          70    BR21 4YE    465029.0     826.8453
6 Juliet 15   F        1.62          45    CF14 7BR      930.0      361.1762
7 Jackie 3   F        1.65          60    NP24 3AG      10.0       461.7102
8 Julie 37   M        2.01          100   NP24 3AG    102930.0     972.6716
9   Joe 19   M        1.92          75     NP7 5BD      2930.0     360.0188
10 Jeremiah 39   M        1.54          70    NP15 1AE      84953.0     210.9413
11 Jim 17   M        1.73          83    CF35 5AS    470320.0     890.3615
12 Julie 2   F        1.62          45    CF72 8JY      1.5       689.4226
13   Jo 7   F        1.62          51    CF72 9DP      200.0      967.2400
>

```

In the following chapters we will learn how to create new data sets from old data sets and as such it may become necessary to export files to csv.

1.4 Exporting Data Sets

We will export our first data set (`first_dataset`) to csv using the following line of code:

```
write.csv(first_data_set,"~/Desktop/first_data_set.csv")
```

Let's take a look at the screenshot.

1. `write.csv` is the command which is used to tell R to read in data from a csv file.
2. The first command tells R which R object to export.
3. The second command tells R the location of the csv file.

```

R version 2.14.0 (2011-10-31)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-apple-darwin9.8.0/i386 (32-bit)

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

Natural language support but running in an English locale

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

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

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]
[History restored from /Users/VincentAir/.Rapp.history]

> Name <- c("Bob","Billy")
> Age <- c(23,25)
> first_data_set <- data.frame(Name,Age)
> objects()
[1] "Age"           "first_data_set" "Name"
> write.csv(first_data_set,"~/first_data_set.csv")
>

```

2 Chapter 2 - Basic Statistical Procedures

2.1 Procedures

In the previous chapter we were introduced to some very basic aspects of R:

1. what R looks like
2. how to import data into R
3. how to export data into R

In this chapter we will take a closer look at procedures that allow us to analyse and manipulate data. Vectors are the building blocks of all R objects. Single numeric/string variables are in fact vector of size 1. Almost all procedures in R are obtained by applying functions to vectors. Details as to how R handles these operations will be explained in the next chapter (so don't worry about it too much for now).

The procedures we are going to look at in this chapter are:

1. Viewing datasets
2. Summarising the contents of data sets
3. Obtaining summary statistics of data sets
4. Obtaining frequency tables
5. Obtaining linear models
6. Plotting data

2.2 A list of procedures

2.2.1 Utility procedures

We have seen how to view an entire data set (by simply printing the name of the object in question).

We illustrate this once again by considering the MMM data set shown, (imported using `read.csv`).

R Console

Platform: i386-apple-darwin9.8.0/i386 (32-bit)

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

Natural language support but running in an English locale

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

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

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]

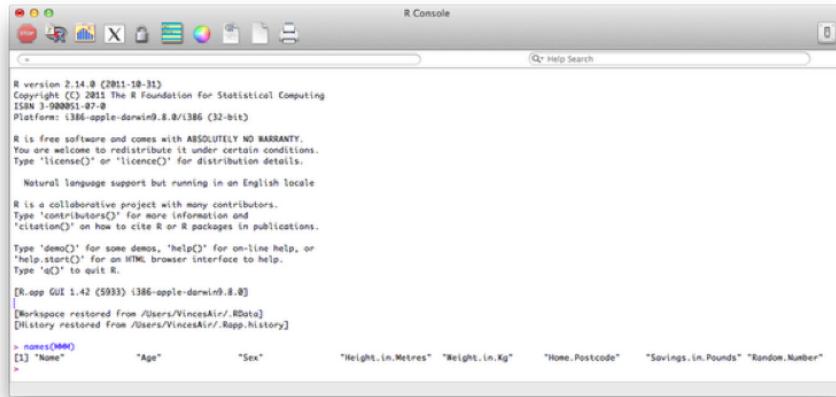
[Workspace restored from /Users/VincentAir/.RData]
[History restored from /Users/VincentAir/.Rapp.history]

```
> ls()
[1] "Age"           "First_data_set" "JJJ"          "Name"
> MMM<-read.csv("MMM.csv")
> ls()
[1] "Age"           "First_data_set" "JJJ"          "Name"
[2] "MMM"
> MMM
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number
1  Malcom  9   Male      1.81       88    CF24 3AG        38     673.12263
2  Mabel  76    F       1.56       58    CF27 4HL      10000    210.71541
3  Manuel 45    M       1.67       41     SW6 4JL       400     814.88402
4   Mark  44   Male      1.76       64     SWS 3JL      64953    31.48134
5   Marc  11    M       1.72       82    BR21 4YE      4512     523.80907
6   Marie  24 Female     1.45       38    CF14 7BR       20     483.87993
7    Mori  26    F       1.61       69    NP24 3AG      10256    582.68096
8  Melody 104    F       1.67       53    NP24 3AG      5078354   337.96374
9  Melody  51    F       1.54       87    NP7 5BD      32156    116.66437
10 Montgomery 19    M       1.88       97    NP15 1AE      56512    483.16678
11   Myer  37    M       1.79       90    CF35 5AS      15648    544.55991
12  Maureen 52    F       1.42       73    CF72 8JY      2000     941.49838
13   Mike  27 Male      1.92      119    CF72 9DP      250     54.01880
> |
```

At times we might not want to open the data set but simply gain some information as to what is in the data set.

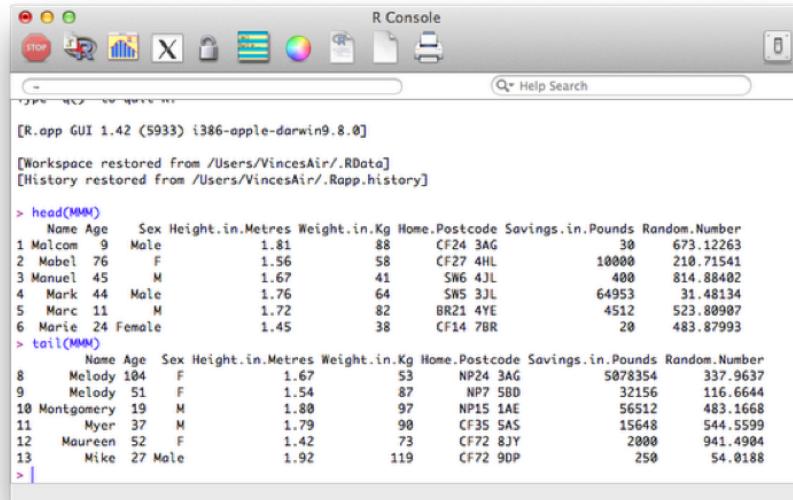
To view only the names of the variables of our data set we use the **name** function as shown.

```
names(MMM)
```



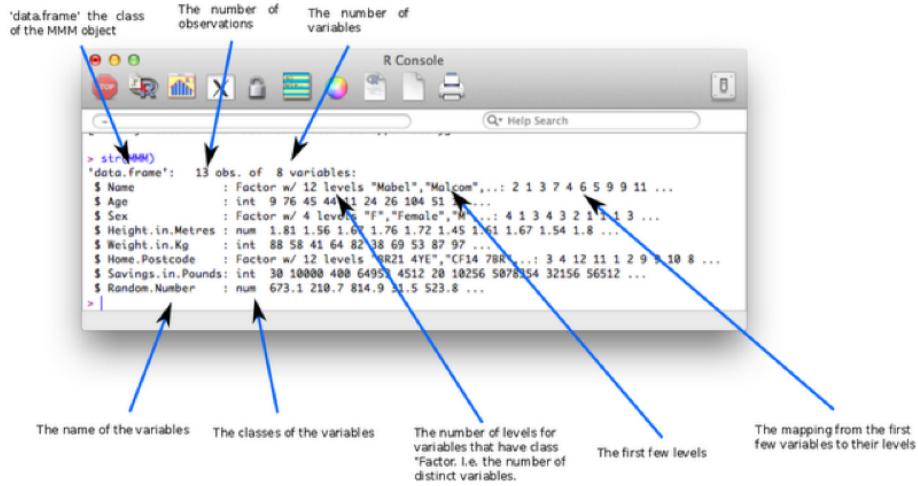
If we had a very large data set then we could quickly view the first/last few entries using the `head/tail` function as shown.

```
head(MMM)
tail(MMM)
```



Finally if we would like to view a description of the overall structure of a data set we can use the `str` function as shown.

```
str(MMM)
```



The class of the imported character variables are **Factors**, this is due to the importation method (`read.csv`) automatically converting the character variables in this form - details about “Factors” are given below. The reason this occurs is the default value of `stringsAsFactors` (used in the `read.csv` function) is TRUE, the following code forces the characters retain their class without conversion.

```
MMM<-read.csv("MMM.csv",stringsAsFactors=FALSE)
```

The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers. This is often a much more efficient way of handling strings.

Natural language support but running in an English locale

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

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

```
[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]
[Workspace restored from /Users/VincesAir/.RData]
[History restored from /Users/VincesAir/.Rapp.history]

> MMM<-read.csv("MMM.csv",stringsAsFactors=FALSE)
> str(MMM)
'data.frame': 13 obs. of 8 variables:
 $ Name      : chr "Malcom" "Mabel" "Manuel" "Mark" ...
 $ Age       : int 9 76 45 44 11 24 26 104 51 19 ...
 $ Sex       : chr "Male" "F" "M" "Male" ...
 $ Height.in.Metres : num 1.81 1.56 1.67 1.76 1.72 1.45 1.61 1.67 1.54 1.8 ...
 $ Weight.in.Kg   : int 88 58 41 64 82 38 69 53 87 97 ...
 $ Home.Postcode : chr "CF24 3AG" "CF27 4HL" "SW6 4JL" "SW5 3JL" ...
 $ Savings.in.Pounds: int 30 10000 400 64953 4512 20 10256 5078354 32156 56512 ...
 $ Random.Number  : num 673.1 210.7 814.9 31.5 523.8 ...
>
```

2.2.2 Descriptive statistics

To gain an initial set of summary statistics of a data frame we can use the `summary` function:

```
summary(MMM)
```

The output of which is shown.

```

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

Natural language support but running in an English locale

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

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

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]

[Workspace restored from /Users/VincentAir/.RData]
[History restored from /Users/VincentAir/.Rapp.history]

> summary(MMM)
   Name      Age     Sex Height.in.Metres Weight.in.Kg   Home.Postcode Savings.in.Pounds Random.Number
Melody :2  Min.   : 9.00   F   :5  Min.   :1.420   Min.   :38.00  NP24 3AG:2   Min.   : 20  Min.   :31.48
Mabel :1  1st Qu.:24.00  Female:1  1st Qu.:1.560   1st Qu.:58.00  BR21 4YE:1  1st Qu.: 400  1st Qu.:210.72
Malcom:1  Median :37.00   M   :4  Median :1.670   Median :73.00  CF14 7BR:1  Median :10000  Median :483.88
Manuel:1  Mean   :40.38   Male :3   Mean   :1.671   Mean   :73.77  CF24 3AG:1  Mean   :405776  Mean   :446.83
Marc  :1  3rd Qu.:51.00   NA   :1  3rd Qu.:1.798   3rd Qu.:88.00  CF27 4HL:1  3rd Qu.: 32156  3rd Qu.:582.68
Mari  :1  Max.   :104.00  NA   :1  Max.   :1.920   Max.   :119.00  CF35 5AS:1  Max.   :5078354  Max.   :941.49
(Other):6
>

```

Recall that most “things” in R are objects and “summary” is a good example of a generic function that works on most objects. If you are faced with a new object (for example the output of a regression analysis) it is sometimes worth trying to apply summary on it to get some initial information.

To obtain a particular summary statistic of a specific variable, we can use functions that apply to vectors and select the vectors from the dataset.

To select a particular column (as a vector) from our dataset, we use the following command:

`MMM$Age`

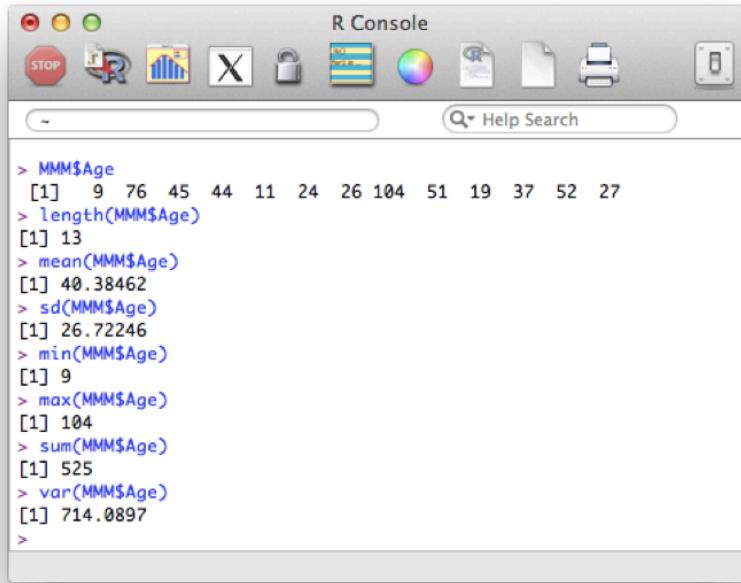
We can then apply various functions to this vector:

```

length(MMM$Age)
mean(MMM$Age)
sd(MMM$Age)
min(MMM$Age)
max(MMM$Age)
sum(MMM$Age)
var(MMM$Age)

```

The output of which is shown.



The screenshot shows the R Console window with the title "R Console". The menu bar includes "File", "Edit", "View", "Search", "Help", and "Console". Below the menu is a toolbar with icons for Stop, Run, Plot, Histogram, X, Lock, Data View, Color, and Print. The main console area contains the following R code and output:

```

> MMM$Age
[1] 9 76 45 44 11 24 26 104 51 19 37 52 27
> length(MMM$Age)
[1] 13
> mean(MMM$Age)
[1] 40.38462
> sd(MMM$Age)
[1] 26.72246
> min(MMM$Age)
[1] 9
> max(MMM$Age)
[1] 104
> sum(MMM$Age)
[1] 525
> var(MMM$Age)
[1] 714.0897
>

```

We can compartmentalise our results using the `by` function. The general syntax for the `by` function is given below:

```
by(data=dataFrame , Indices=grouping variables, FUN= a function)
```

We'll use this to obtain the mean age and height compartmentalised by sex:

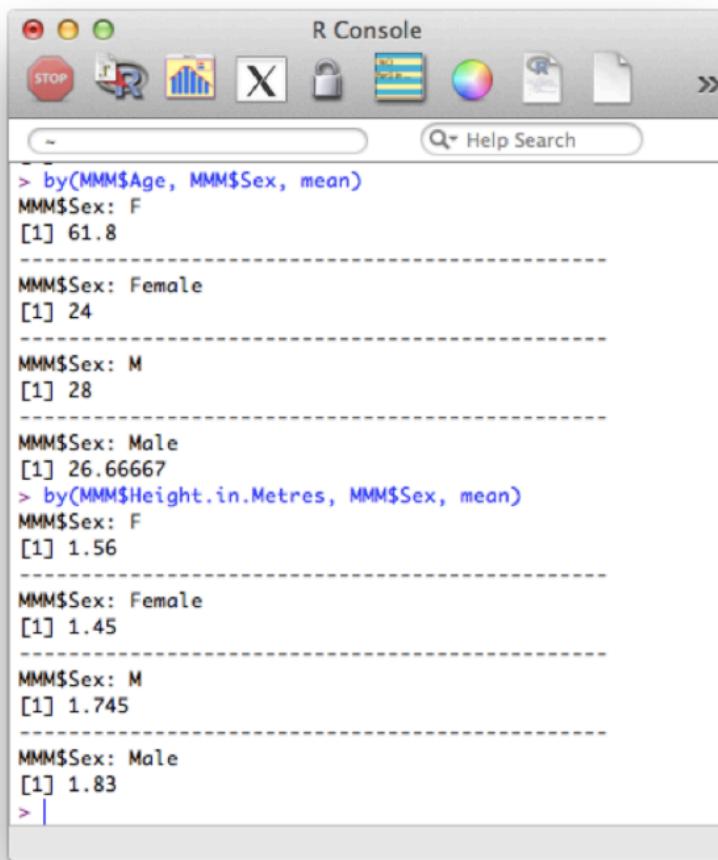
```
by(MMM$Age, MMM$Sex, mean)
by(MMM$Height.in.Metres, MMM$Sex, mean)
```

The output of which is shown.

The above code subsets the data frame by the grouping variable. If we want to just carry out an action on a vector (as above, we're only actually interested in the Age vector or the Height vector) then we can also use the “tapply function”. This applies a function to a vector according to the levels of another vector:

```
tapply(MMM$Age, MMM$Sex, mean)
```

Finally, to reduce the number of keystrokes, we can use the `with` statement. This tells R to evaluate everything within a given data frame. The following code reproduces the above results:



The screenshot shows the R Console window with the title "R Console". The window contains the following R code and its corresponding output:

```
> by(MMM$Age, MMM$Sex, mean)
MMM$Sex: F
[1] 61.8
-----
MMM$Sex: Female
[1] 24
-----
MMM$Sex: M
[1] 28
-----
MMM$Sex: Male
[1] 26.66667
> by(MMM$Height.in.Metres, MMM$Sex, mean)
MMM$Sex: F
[1] 1.56
-----
MMM$Sex: Female
[1] 1.45
-----
MMM$Sex: M
[1] 1.745
-----
MMM$Sex: Male
[1] 1.83
> |
```

Figure 1: Compartmentalising Summary statistics in R

```
with(data=MMM,by(Age,Sex,mean) )  
with(data=MMM,tapply(Age, Sex, mean))
```

Note: the `data=` statement can be omitted.

2.2.3 Frequency Tables

The `table` function allows us to obtain frequency tables of data sets. As an example let us consider the data set shown. The `table` function creates a “table” (a particular type of R object):

```
table(math_tests$Teacher,math_tests$Pass.Fail)
```

ID	Name	Teacher	Pass/Fail
1	Bob	Mr Smith	P
2	Brayden	Mr Evans	F
3	Billy	Mr Smith	P
4	John	Mr Smith	P
5	Jack	Mr Smith	F
6	Julie	Mr Evans	F
7	Jane	Mr Evans	P
8	Jackie	Mr Evans	P
9	Bonnie	Mr Evans	P
10	Bob	Mr Evans	P
11	Juliet	Mr Smith	P
~			
			1,1
			All

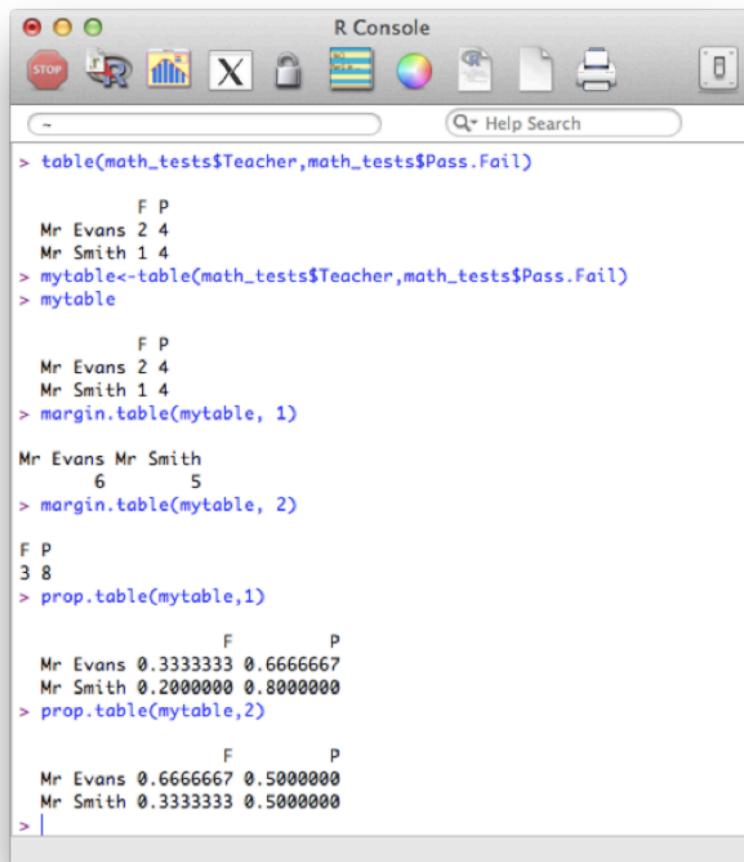
Again we can write this as:

```
with(math_tests,table(Teacher,Pass.Fail))
```

We can save this table as a new object and use the `prop` command to gain row and column totals and proportions:

```
mytable<-table(math_tests$Teacher,math_tests$Pass.Fail)
margin.table(mytable, 1)
margin.table(mytable, 2)
prop.table(mytable,1)
prop.table(mytable,2)
```

The output of all this is shown.



The screenshot shows the R Console window with the following session history:

```
> table(math_tests$Teacher,math_tests$Pass.Fail)
   F P
Mr Evans 2 4
Mr Smith 1 4
> mytable<-table(math_tests$Teacher,math_tests$Pass.Fail)
> mytable

   F P
Mr Evans 2 4
Mr Smith 1 4
> margin.table(mytable, 1)

Mr Evans Mr Smith
       6      5
> margin.table(mytable, 2)

F P
3 8
> prop.table(mytable,1)

      F          P
Mr Evans 0.3333333 0.6666667
Mr Smith 0.2000000 0.8000000
> prop.table(mytable,2)

      F          P
Mr Evans 0.6666667 0.5000000
Mr Smith 0.3333333 0.5000000
> |
```

2.2.4 Correlations

The following lines of code select only the columns from MMM that are numeric. An explanation of this will follow in the next chapter.

```
MMM[, sapply(MMM, is.numeric)]
```

The correlation ‘cor’ function only acts upon numeric vectors (and/or dataframes), hence the selection of solely numeric values first.

```
MMMinum<-MMM[, sapply(MMM, is.numeric)]
cor(MMMinum)
```

The `cor` function however does not give tests of significance. We can obtain significance tests between two variables using `cor.test`:

```
cor.test(MMM$Age, MMM$Height.in.Metres)
```

The output is shown.

```
R Console
> MMMinum<-MMM[, sapply(MMM, is.numeric)]
> cor(MMMinum)
      Age Height.in.Metres Weight.in.Kg Savings.in.Pounds Random.Number
Age   1.0000000 -0.330095895 -0.4209802  0.716202814 -0.2265822
Height.in.Metres -0.33009595  1.000000000  0.6300353  0.001949651 -0.3034282
Weight.in.Kg    -0.4209802  0.630035302  1.0000000 -0.267950062 -0.2555668
Savings.in.Pounds 0.7162028  0.001949651 -0.2679501  1.000000000 -0.1214556
Random.Number   -0.2265822 -0.303420212 -0.2555668 -0.121455617  1.0000000
> cor.test(MMM$Age, MMM$Height.in.Metres)

Pearson's product-moment correlation

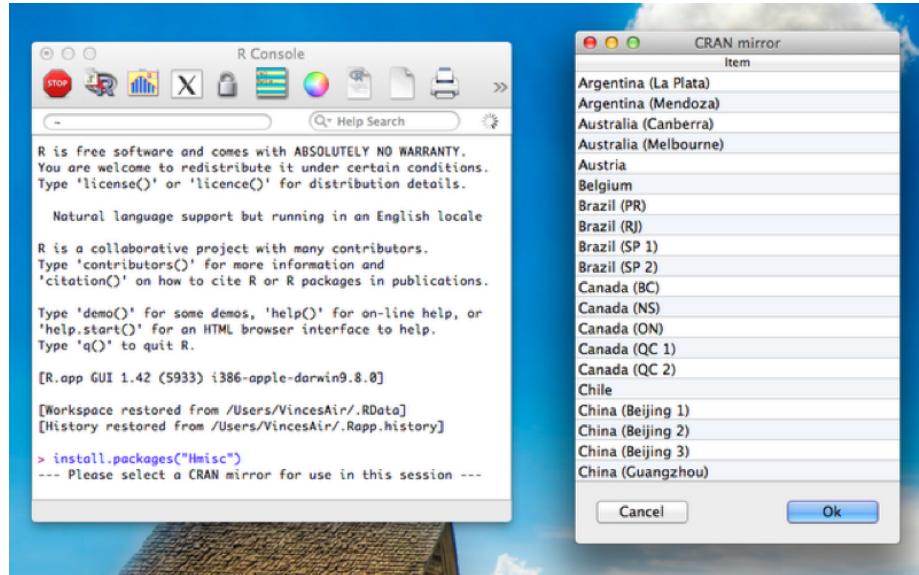
data:  MMM$Age and MMM$Height.in.Metres
t = -1.1598, df = 11, p-value = 0.2707
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
-0.7454925  0.2699957
sample estimates:
        cor
-0.3300959
```

As is often the case in open source software, packages are independently developed and need to be called to be used in R. Above we have shown the very basic approach to obtaining correlations in R, we will now use the `rcorr` function from the `Hmisc` package.

To install the package we use the following code:

```
install.packages("Hmisc")
```

Once that happens a window opens asking us to choose the mirror from which to download. This is shown.



Once that is done to load the package the following code is required:

```
library(Hmisc)
```

or:

```
require(Hmisc)
```

To see the packages currently loaded we can use the following code:

```
search()
```

Note: Packages can also be installed by selecting the 'Packages' tab and selecting the package required for installation.

```

> library(Hmisc)
Loading required package: survival
Loading required package: splines
Hmisc library by Frank E Harrell Jr

Type library(help='Hmisc'), ?overview, or ?Hmisc.Overview'
to see overall documentation.

NOTE:Hmisc no longer redefines [.Factor to drop unused levels when
subsetting. To get the old behavior of Hmisc type dropUnusedLevels().

Attaching package: 'Hmisc'

The following object(s) are masked from 'package:survival':
  untangle.specials

The following object(s) are masked from 'package:base':
  format.pval, round.POSIXt, trunc.POSIXt, units

Warning message:
  package 'Hmisc' was built under R version 2.14.2

```

Using this package, we will use the `rcorr` function that gives the correlation matrix for a data set. Note that the data set must be numeric and in `matrix` form. The following code selects the numeric variables from the MMM data set and converts the result to a matrix:

```

MMMnum<-MMM[,sapply(MMM,is.numeric)]
MMMmat<-as.matrix(MMMnum)

```

Once this is done we can get the correlation matrix using the following code:

```
rcorr(MMMmat)
```

The output is shown.

```

R Console
STOP HELP SEARCH
> MMMnum<-MMM[,sapply(MMM,is.numeric)]
> MMMmat<-as.matrix(MMMnum)
> rcorr(MMMmat)
            Age Height.in.Metres Weight.in.Kg Savings.in.Pounds Random.Number
Age          1.00      -0.33       -0.42        0.72      -0.23
Height.in.Metres -0.33       1.00        0.63        0.00      -0.30
Weight.in.Kg     -0.42        0.63       1.00       -0.27      -0.26
Savings.in.Pounds 0.72        0.00       -0.27        1.00      -0.12
Random.Number    -0.23      -0.30       -0.26      -0.12       1.00
n= 13

P
            Age Height.in.Metres Weight.in.Kg Savings.in.Pounds Random.Number
Age 0.2707      0.1520      0.0059      0.4566
Height.in.Metres 0.2707      0.0210      0.9950      0.3136
Weight.in.Kg     0.1520 0.0210      0.3778      0.3994
Savings.in.Pounds 0.0059 0.9950      0.3778      0.6927
Random.Number    0.4566 0.3136      0.3994
>

```

2.2.5 Linear Models

In this section we very briefly see the syntax for some basic linear models in R.

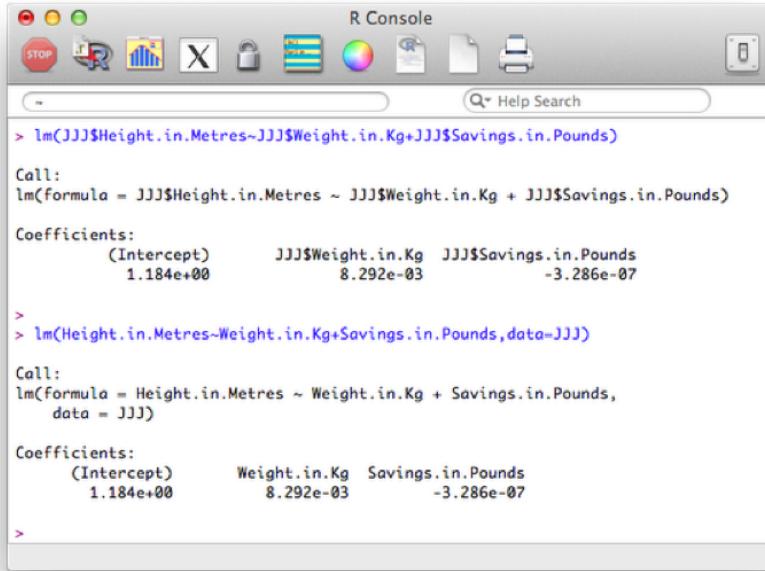
The general syntax for linear regression is as follows in R:

```
lm(outcome~predictors)
```

The following code will be used to investigate whether or not there is a linear model of height with weight and savings and predictors (in two ways, the second is slightly more compact and leaves less room for confusion):

```
lm(JJJ$Height.in.Metres~JJJ$Weight.in.Kg+JJJ$Savings.in.Pounds)
lm(Height.in.Metres~Weight.in.Kg+Savings.in.Pounds,data=JJJ)
```

The results are shown.



The screenshot shows the R Console window with the title "R Console". The window contains the following R code and its output:

```
> lm(JJJ$Height.in.Metres~JJJ$Weight.in.Kg+JJJ$Savings.in.Pounds)
Call:
lm(formula = JJJ$Height.in.Metres ~ JJJ$Weight.in.Kg + JJJ$Savings.in.Pounds)

Coefficients:
(Intercept)      JJJ$Weight.in.Kg  JJJ$Savings.in.Pounds
1.184e+00        8.292e-03       -3.286e-07

> lm(Height.in.Metres~Weight.in.Kg+Savings.in.Pounds,data=JJJ)
Call:
lm(formula = Height.in.Metres ~ Weight.in.Kg + Savings.in.Pounds,
data = JJJ)

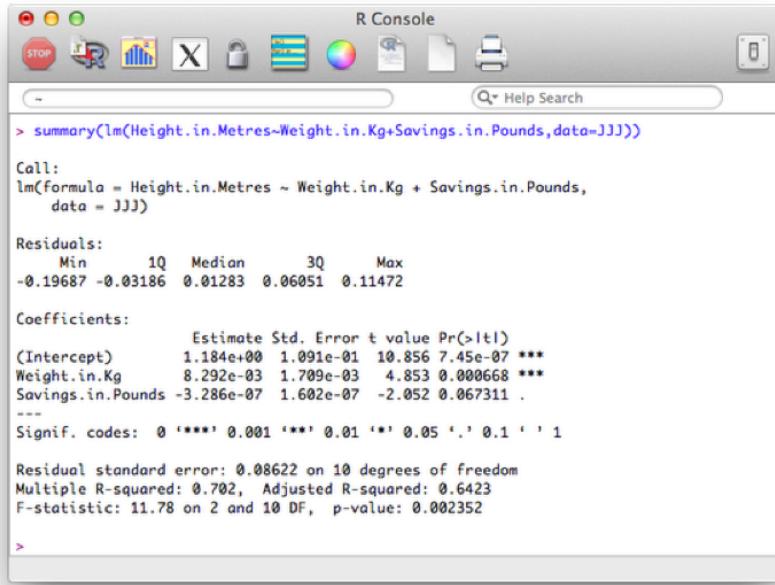
Coefficients:
(Intercept)      Weight.in.Kg  Savings.in.Pounds
1.184e+00        8.292e-03       -3.286e-07

>
```

To get the full set of results from the regression analysis we use the following code:

```
summary(lm(Height.in.Metres~Weight.in.Kg+Savings.in.Pounds,data=JJJ))
```

The output is shown.



The screenshot shows an R console window with the title "R Console". The window contains the following R code and its output:

```

> summary(lm(Height.in.Metres~Weight.in.Kg+Savings.in.Pounds,data=JJJ))

Call:
lm(formula = Height.in.Metres ~ Weight.in.Kg + Savings.in.Pounds,
    data = JJJ)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.19687 -0.03186  0.01283  0.06051  0.11472 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 1.184e+00 1.091e-01 10.856 7.45e-07 ***
Weight.in.Kg 8.292e-03 1.709e-03  4.853 0.000668 ***
Savings.in.Pounds -3.286e-07 1.602e-07 -2.052 0.067311 .  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.08622 on 10 degrees of freedom
Multiple R-squared:  0.702, Adjusted R-squared:  0.6423 
F-statistic: 11.78 on 2 and 10 DF,  p-value: 0.002352

>

```

Looking at the p value we see that the overall model should not be rejected, however the detailed results show that perhaps we could remove savings from the model.

Analysis of variance (ANOVA) can be done easily in R. We shall show this using a new data set (math.csv) shown.

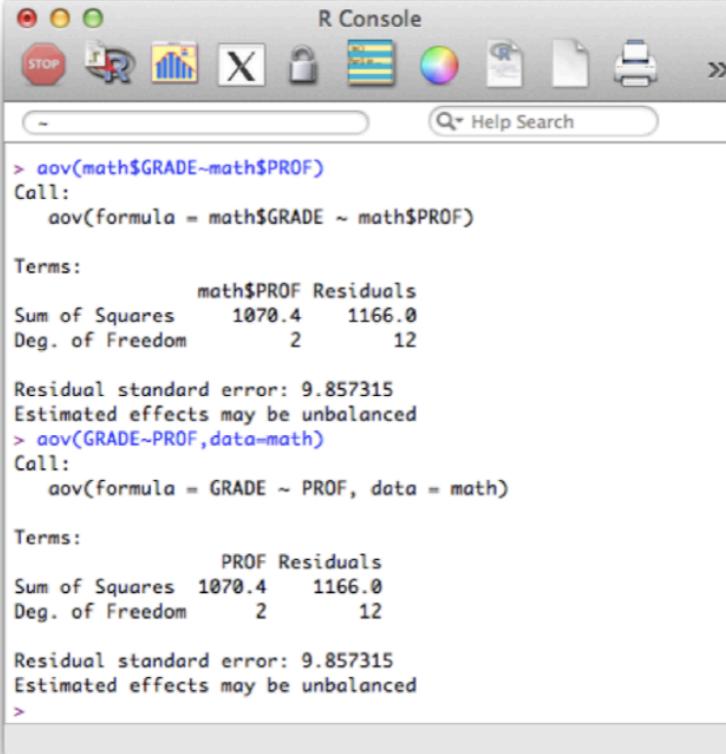
```
1 CALC,PROF,GRADE
2 1,A,65
3 1,B,70
4 1,C,90
5 1,B,85
6 1,B,95
7 1,C,100
8 1,A,75
9 1,C,89
10 2,B,87
11 2,A,73
12 2,C,75
13 2,A,55
14 2,B,79
15 2,C,98
16 2,A,82
```

```
aov(outcome~class,data)
```

We will use the “aov” function to see if the grades obtained by students depend on their teacher (in two ways, the second is slightly more compact):

```
aov(math$GRADE~math$PROF)
aov(GRADE~PROF,data=math)
```

The results are shown.



The screenshot shows the R Console window with the title "R Console". The menu bar includes "STOP", "R", "X", "LOCK", "HELP", "SEARCH", and "">>". Below the menu bar is a toolbar with icons for file operations like Open, Save, Print, and Help. The main area displays the following R code and its output:

```
> aov(math$GRADE~math$PROF)
Call:
  aov(formula = math$GRADE ~ math$PROF)

Terms:
  math$PROF Residuals
Sum of Squares    1070.4    1166.0
Deg. of Freedom      2        12

Residual standard error: 9.857315
Estimated effects may be unbalanced
> aov(GRADE~PROF,data=math)
Call:
  aov(formula = GRADE ~ PROF, data = math)

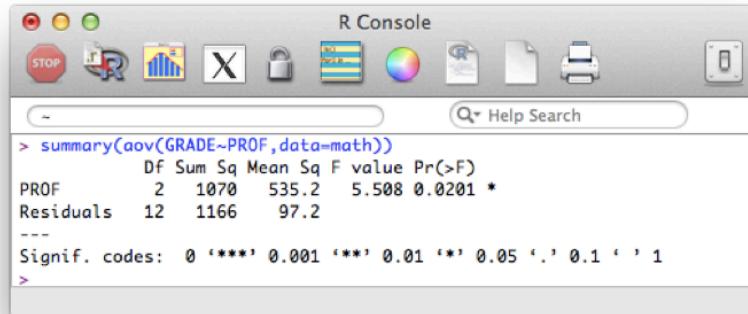
Terms:
  PROF Residuals
Sum of Squares  1070.4    1166.0
Deg. of Freedom   2        12

Residual standard error: 9.857315
Estimated effects may be unbalanced
>
```

To get the full set of results from the ANOVA we use the following code:

```
summary(aov(GRADE~PROF,data=math))
```

The results are shown.



The screenshot shows the R Console window with the title "R Console". The window contains the following R code and its output:

```

> summary(aov(GRADE~PROF,data=math))
   DF Sum Sq Mean Sq F value Pr(>F)
PROF      2    1070    535.2   5.508 0.0201 *
Residuals 12   1166     97.2
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>

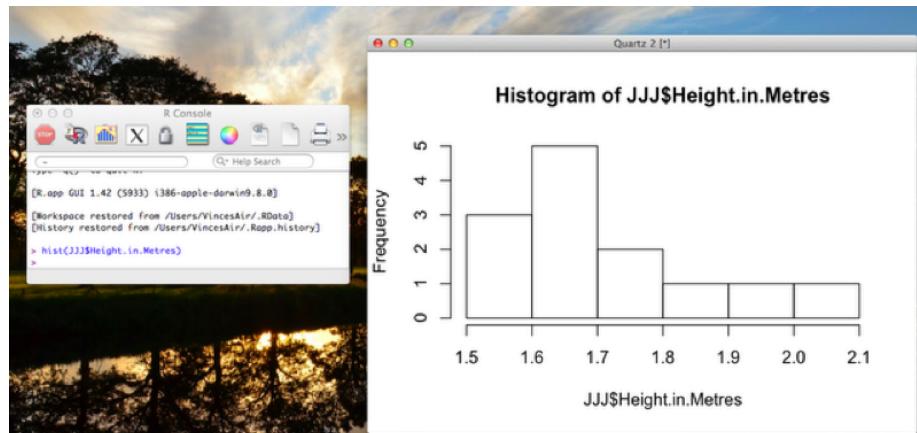
```

2.2.6 Plots and charts

Note that due to the object oriented nature of R, almost all of the above outputs have a “plot” attribute.

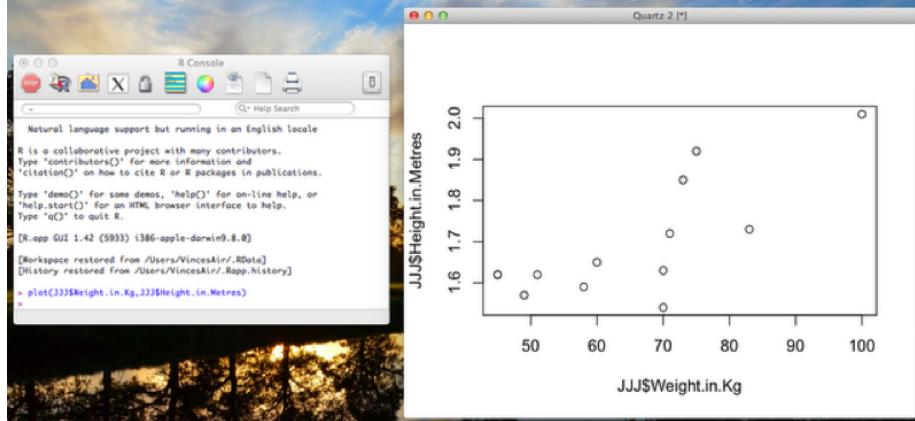
The simplest way to produce a histogram in R is to use the “hist” function. The following code gives a histogram for the height of entries in the JJJ data set as shown.

```
hist(JJJ$Height.in.Metres)
```



The simplest way to produce a scatter plot in R is to use the “plot” function. The following code gives a scatter plot for the height against weight of entries in the JJJ data set as shown.

```
plot(JJJ$Weight.in.Kg, JJJ$Height.in.Metres)
```



There are various other ways to obtain similar graphs, as well as change the look and feel of our graphs. We won't go into this here but you are encouraged to look into it (in particular the ggplot package is widely used).

2.3 Exporting output

All the non graphical outputs from R are objects, as such they can be output to a file (to be copied into another document if need be) using the write statements of Sections 1.4. However to export graphical output, we use any of the following statements (depending on the output format required):

```
pdf("mygraph.pdf")
win.metafile("mygraph.wmf")
png("mygraph.png")
jpeg("mygraph.jpg")
bmp("mygraph.bmp")
postscript("mygraph.ps")
```

Once that command is written we use a normal R command to create a plot and finally we close the output file with the following statement:

```
dev.off()
```

The following code creates a png file entitled "height_v_weight_plot" with the previous scatter plot.

```
png("height_v_weight_plot.png")
plot(JJJ$Weight.in.Kg, JJJ$Height.in.Metres)
dev.off()
```

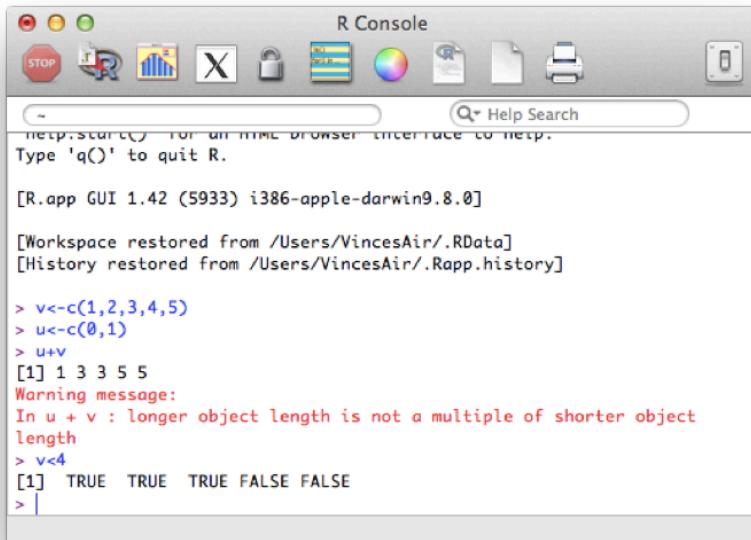
3 Chapter 3 - Manipulating data

3.1 Vectors and data frames

3.1.1 Vectors

When considering R data frames it is important to recall that they are composed of vectors. Even individual scalars and strings are vectors. This is a very powerful tool.

One important notion when handling vectors is the use of ‘recycling’. As all elements are vectors, when performing an operation between two vectors of different length, R automatically repeats (or recycles) the shorter one until it is long enough.

A screenshot of the R Console window. The window has a title bar 'R Console' with standard OS X window controls. Below the title bar is a toolbar with icons for Stop, Help, and other functions. The main area is a text-based console. It shows the following session:

```
http://www.R-project.org for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.42 (5933) i386-apple-darwin9.8.0]

[Workspace restored from /Users/Vincent/RData]
[History restored from /Users/Vincent/.Rapp.history]

> v<-c(1,2,3,4,5)
> u<-c(0,1)
> u+v
[1] 1 3 3 5 5
Warning message:
In u + v : longer object length is not a multiple of shorter object
length
> v<4
[1] TRUE TRUE TRUE FALSE FALSE
> |
```

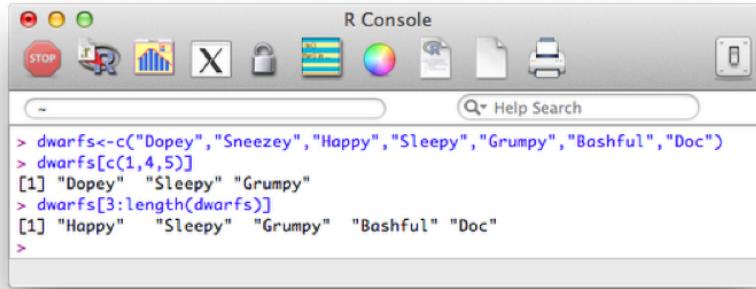
The console shows the creation of two vectors 'v' and 'u', their addition, and a warning message about recycling. The final output is a vector of length 5 with boolean values.

In the previous example, $(u+v)$ we add the elements of both vectors together. R automatically increases the length of u so that the operation becomes $(1,2,3,4,5) + (0,1,0,1,0)$. In the second example we compare the elements of v to 4. R automatically increases the length of the vector containing 4 so that the operation becomes $(1,2,3,4,5) < (4,4,4,4,4)$ which returns a vector of size 5 with boolean (True or False) elements.

This second concept is important when understanding how to select certain variables in R (we saw this briefly in the previous section).

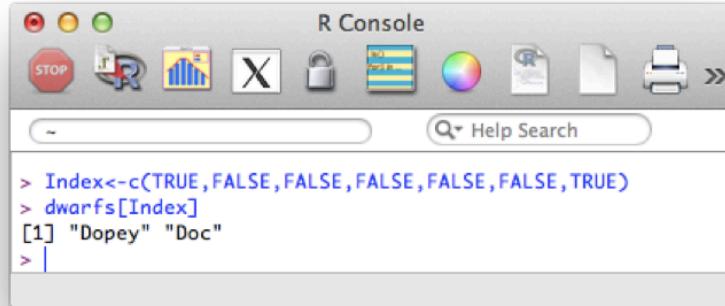
Another important notion in R is that of indexing. We can select elements of a vector by specifying the indices of the elements required:

```
dwarfs<-c("Dopey", "Sneezy", "Happy", "Sleepy", "Grumpy", "Bashful", "Doc")
dwarfs[c(1,4,5)]
dwarfs[3:length(dwarfs)]
```



Both of the previous approaches use a vector of indices to indicate the elements we require. The second approach uses a shorthand to create a vector of elements (containing the integers 3 to 5). Another approach is to simply use a vector of boolean values (True or False) to indicate the positions that are to be selected.

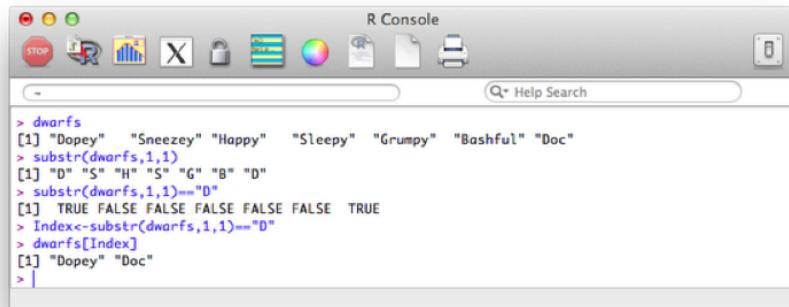
```
Index<-c(TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE)
dwarfs[Index]
```



It should be straightforward to realise that we can combine recycling and indexing to filter vectors:

```
Index<-substr(dwarfs,1,1)=="D"
dwarfs[Index]
```

The first command creates an index set of boolean variables using the `substr` function and recycling (in this case used to take the first character of each element). This allows us to obtain the elements of the vector `dwarfs` with first letter D as shown.



We have seen how to subset vectors using filtering, the same logic applies to data frames.

We can first of all use indexing to obtain the variables we want. For example the following code will select the all the variables apart from the 4th and 5th:

```
MMM[c(1,2,3,6,7,8)]
```

A quicker way is to simply state the variables we want to drop:

```
MMM[c(-4,-5)]
```

The output of the above code is shown.

	Name	Age	Sex	Home.Postcode	Savings.in.Pounds	Random.Number	
1	Malcom	9	Male	CF24 3AG	30	673.12263	
2	Mabel	76	F	CF27 4HL	10000	210.71541	
3	Manuel	45	M	SW6 4JL	400	814.88402	
4	Mark	44	Male	SWS 3JL	64953	31.48134	
5	Marc	11	M	BR21 4YE	4512	523.80907	
6	Marie	24	Female	CF14 7BR	20	483.87993	
7	Mari	26	F	NP24 3AG	10256	582.68096	
8	Melody	104	F	NP24 3AG	5078354	337.96374	
9	Melody	51	F	NP7 5BD	32156	116.66437	
10	Montgomery	19	M	NP15 1AE	56512	483.16678	
11	Myer	37	M	CF35 5AS	15648	544.55991	
12	Maureen	52	F	CF72 8JY	2000	941.49038	
13	Mike	27	Male	CF72 9DP	250	54.01880	

We can also list the names of variables we want to keep:

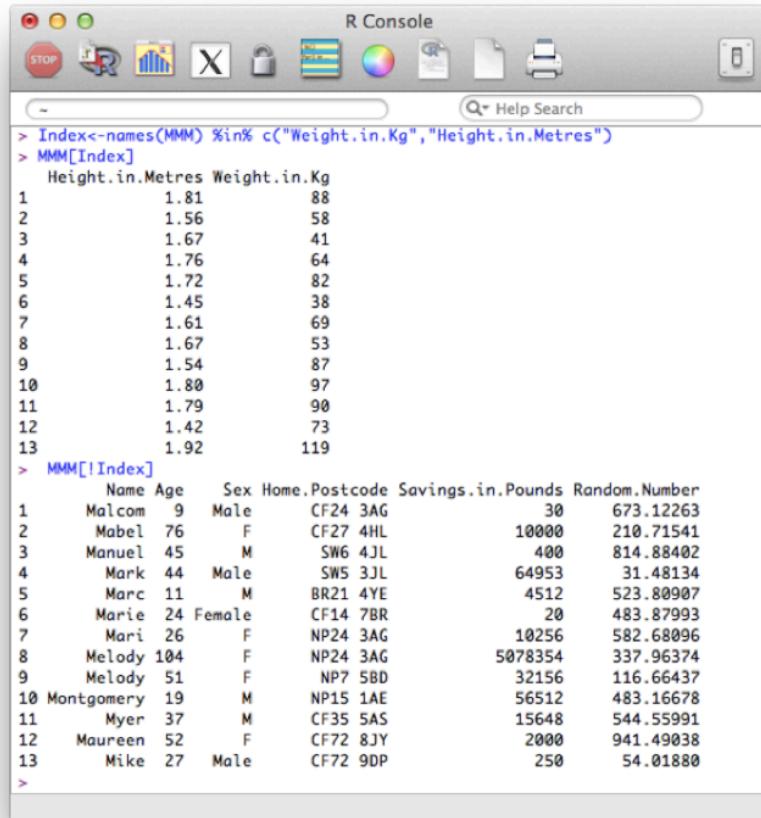
```
MMM[c("Name", "Age", "Sex", "Home.Postcode", "Savings.in.Pounds", "Random.Number")]
```

Finally we can create a vector of booleans that gives the same above result or the opposite result (i.e. drops the variables).

```
Index<-names(MMM) %in% c("Weight.in.Kg","Height.in.Metres")
MMM[Index]
```

```
Index<-names(MMM) %in% c("Weight.in.Kg","Height.in.Metres")
MMM[!Index]
```

Recall the `names` function simply gives a vector containing the names of all the variables in the `MMM` dataset. The `%in%` operator is used to create a vector of booleans by testing if the elements of `names(MMM)` are in the vector `c("Weight.in.Kg", "Height.in.Metres")`. The `!` operator acting on `Index` simply negates the booleans contained in `Index`.



The screenshot shows an R console window with the title "R Console". The window contains the following R code and its resulting output:

```
> Index<-names(MMM) %in% c("Weight.in.Kg","Height.in.Metres")
> MMM[Index]
  Height.in.Metres Weight.in.Kg
1          1.81      88
2          1.56      58
3          1.67      41
4          1.76      64
5          1.72      82
6          1.45      38
7          1.61      69
8          1.67      53
9          1.54      87
10         1.80      97
11         1.79      90
12         1.42      73
13         1.92     119

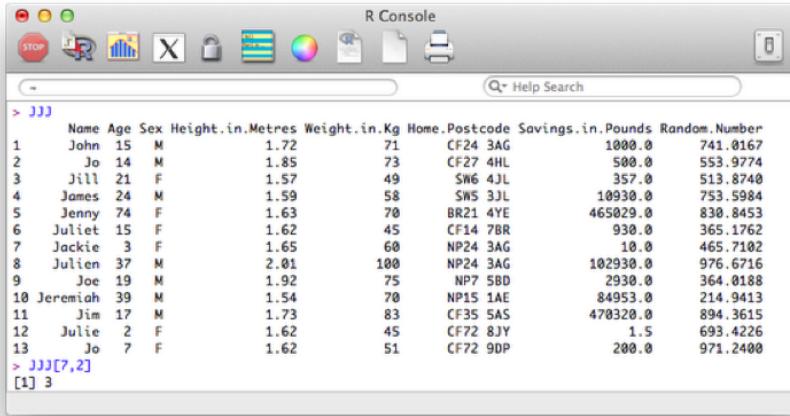
> MMM[!Index]
   Name Age Sex Home.Postcode Savings.in.Pounds Random.Number
1  Malcom  9   Male    CF24 3AG            30    673.12263
2  Mabel  76    F    CF27 4HL           10000   210.71541
3  Manuel 45    M    SW6 4JL            400    814.88402
4  Mark   44   Male    SW5 3JL           64953   31.48134
5  Marc   11    M    BR21 4YE            4512    523.80907
6  Marie  24 Female  CF14 7BR             20    483.87993
7  Mari   26    F    NP24 3AG            10256    582.68096
8  Melody 104   F    NP24 3AG           5078354   337.96374
9  Melody  51    F    NP7 5BD            32156   116.66437
10 Montgomery 19    M    NP15 1AE           56512   483.16678
11  Myer   37    M    CF35 5AS            15648   544.55991
12  Maureen 52    F    CF72 8JY            2000    941.49038
13  Mike   27   Male    CF72 9DP            250    54.01880
```

3.1.2 Selecting Observations

We can select any particular element of a data frame in R using the following syntax:

```
dataframe[i,j]
```

This would give the entry for variable j of observation i as shown.



The screenshot shows the R Console window with the title "R Console". The console displays the following code and output:

```
> JJJ
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number
1  John  15   M      1.72          71    CF24 3AG     1000.0    741.0167
2   Jo   14   M      1.85          73    CF27 4HL      500.0    553.9774
3  Jill  21   F      1.57          49     SW6 4JL      357.0    513.8740
4 James  24   M      1.59          58     SW5 3JL     10930.0    753.5984
5 Jenny  74   F      1.63          70    BR21 4YE     465029.0    830.8453
6 Juliet 15   F      1.62          45    CF14 7BR      930.0    365.1762
7 Jackie  3   F      1.65          60    NP24 3AG      10.0     465.7102
8 Julien 37   M      2.01         100   NP24 3AG    102930.0    976.6716
9   Joe  19   M      1.92          75     NP7 5BD      2930.0    364.0188
10 Jeremiah 39   M      1.54          70    NP15 1AE     84953.0    214.9413
11   Jim  17   M      1.73          83    CF35 5AS    470320.0    894.3615
12 Julie  2   F      1.62          45    CF72 8JY      1.5     693.4226
13   Jo   7   F      1.62          51    CF72 9DP      200.0    971.2400
> JJJ[7,2]
[1] 3
```

If we ignore one of the indices R simply returns all the entries corresponding to that index. For example the following code would return all the observations for the 7th observation of the JJJ data set:

```
JJJ[7,]
```

We can also use this to sort a data set. The “order function” returns a set of indices reflecting the ascending order of a vector, thus to sort the JJJ data set by age we use the following code:

```
JJJ[order(JJJ$Age),]
```

We can use filtering to expand on this and select all observations that obey a particular condition. For example the following code selects entries of JJJ that have age less than or equal to 18:

```
JJJ[JJJ$Age<=18,]
```

3.2 Merging and concatenating data sets

To concatenate two data sets in R we use the `rbind` function (i.e. we bind the two dataframes by rows).

```
MMMJJJ<-rbind(JJJ,MMM)
```

Note that both these data sets need to contain all the variables. If one of the datasets does not contain all the variables then you need to add that variable to it and set its values to NA (missing).

	Name	Age	Sex	Height.in.Metres	Weight.in.Kg	Home.Postcode	Savings.in.Pounds	Random.Number
1	John	15	M	1.72	71	CF27 3AG	1000.0	741.01669
2	Jo	14	M	1.85	73	CF27 4HL	500.0	553.97742
3	Jill	21	F	1.57	49	SN6 4JL	357.0	513.87405
4	James	24	M	1.59	58	SN5 3JL	1030.0	753.59836
5	Jenny	74	F	1.63	70	BR21 4YE	465029.0	838.84534
6	Juliet	15	F	1.62	45	CF14 7BR	930.0	365.17623
7	Jackie	3	F	1.65	60	NP24 3AG	10.0	465.71015
8	Julien	37	M	2.01	100	NP24 3AG	102930.0	976.67164
9	Joe	19	M	1.92	75	NP7 5BD	2930.0	364.01884
10	Jeremiah	39	M	1.54	70	NP15 1AE	84953.0	214.94134
11	Jim	17	M	1.73	83	CF35 SAS	470320.0	894.36153
12	Julie	2	F	1.62	45	CF72 8JY	1.5	693.42257
13	Jo	7	F	1.62	51	CF72 9DP	200.0	971.24004
14	Malcom	9	Male	1.81	88	CF24 3AG	38.0	673.12263
15	Mabel	76	F	1.56	58	CF27 4HL	10000.0	210.71541
16	Manuel	45	M	1.67	41	SN6 4JL	400.0	814.88402
17	Mark	44	Male	1.76	64	SN5 3JL	64953.0	31.48134
18	Marc	11	M	1.72	82	BR21 4YE	4512.0	523.88097
19	Marie	24	Female	1.45	38	CF14 7BR	20.0	483.87993
20	Mari	26	F	1.61	69	NP24 3AG	10256.0	582.68096
21	Melody	104	F	1.67	53	NP24 3AG	5078354.0	337.96374
22	Melody	51	F	1.54	87	NP7 5BD	32156.0	116.66437
23	Montgomery	19	M	1.80	97	NP15 1AE	56512.0	483.16678
24	Myer	37	M	1.79	90	CF35 SAS	15648.0	544.55991
25	Maureen	52	F	1.42	73	CF72 8JY	2000.0	941.49038
26	Mike	27	Male	1.92	119	CF72 9DP	250.0	54.01880

To merge two dataframes in R we use the `merge` function. We'll illustrate this with the following data set:

```
Name<-c("Bob","Ben")
Weight<-c(75,94)
other_data_set<-data.frame(Name,Weight)
```

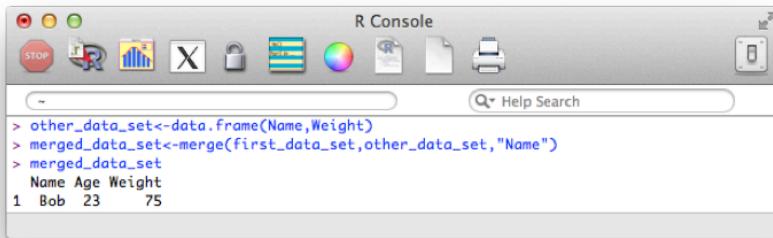
We'll merge this new data set with the data set we created in Chapter 1.

```
merged_data_set<-merge(first_data_set,other_data_set,"Name")
```

(or equivalently:)

```
merged_data_set<-merge(x=first_data_set,y=other_data_set,by="Name")
```

The output is shown.



A screenshot of the R Console window. The title bar says "R Console". The menu bar includes "File", "Edit", "View", "Help", and "Search". Below the menu is a toolbar with icons for Stop, Run, Plot, Histogram, X, Lock, Flag, Color, and Print. The main console area shows the following R code and its output:

```
> other_data_set<-data.frame(Name,Weight)
> merged_data_set<-merge(first_data_set,other_data_set,"Name")
> merged_data_set
  Name Age Weight
1 Bob 23    75
```

Note that the merge statement only selects observations that are present in both files. We can pass further arguments to the merge statement that allow us to select all the values from a particular data set and/or both data sets. These operations are at times called ‘joins’ (and are very common in SQL which we shall see in Chapter 5). The basic merge statement (as above) would be referred to as an ‘inner’ join.

A left outer join (selecting all variables from the first data set):

```
merged_data_set<-merge(first_data_set,other_data_set,"Name",all.x=TRUE)
```

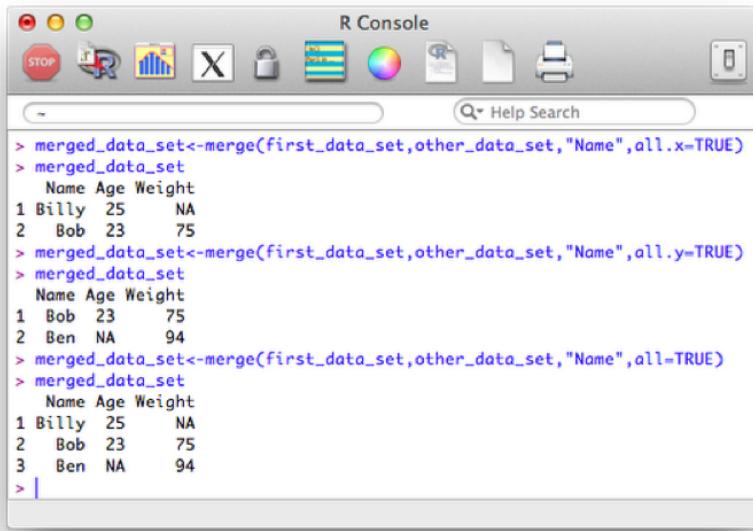
A right outer join:

```
merged_data_set<-merge(first_data_set,other_data_set,"Name",all.y=TRUE)
```

A full outer join:

```
merged_data_set<-merge(first_data_set,other_data_set,"Name",all=TRUE)
```

The output of the above is shown.



The screenshot shows the R Console window with the title "R Console". The window contains the following R code and its output:

```
> merged_data_set<-merge(first_data_set,other_data_set,"Name",all.x=TRUE)
> merged_data_set
  Name Age Weight
1 Billy 25    NA
2 Bob   23    75
> merged_data_set<-merge(first_data_set,other_data_set,"Name",all.y=TRUE)
> merged_data_set
  Name Age Weight
1 Bob   23    75
2 Ben   NA    94
> merged_data_set<-merge(first_data_set,other_data_set,"Name",all=TRUE)
> merged_data_set
  Name Age Weight
1 Billy 25    NA
2 Bob   23    75
3 Ben   NA    94
> |
```

3.3 Creating new variables

Creating new variables using various arithmetic and/or string relationships is straightforward in R. The following code creates a new data set call `MMM_With_BMI` as a copy of the `MMM` data set and then adds a new variable “`BMI`” as a function of the height and weight variables in the `MMM_With_BMI` dataset.

```
MMM_With_BMI<-MMM
MMM_With_BMI$BMI<-MMM$Weight.in.Kg/(MMM$Height.in.Metres^2)
MMM_With_BMI
```

The output is shown.

```

> MMM_With_BMI<-MMM
> MMM_With_BMI$BMI<-MMM$Weight.in.Kg/(MMM$Height.in.Metres^2)
> MMM_With_BMI
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number      BMI
1  Malcolm  9   Male        1.81       88        CF24 3AG          30     673.12263 26.86121
2   Mabel  76    F        1.56       58        CF27 4HL         10000    218.71541 23.83300
3  Manuel  45    M        1.67       41        SW6 4JL          400     814.88402 14.70114
4    Marc  44   Male        1.76       64        SWS 3JL         64953    31.48134 20.66116
5    Marc  11    M        1.72       82        BR21 4YE         4512     523.80907 27.71769
6  Marie  24 Female        1.45       38        CF14 7BR          20     483.87993 18.07372
7   Mari  26    F        1.61       69        NP24 3AG         10256     582.68096 26.61934
8  Melody 104    F        1.67       53        NP24 3AG         5078354    337.96374 19.00391
9  Melody  51    F        1.54       87        NP7 5BD          32156    116.66437 36.68410
10 Montgomery  19    M        1.80       97        NP15 1AE         56512    483.16678 29.93827
11   Myer  37    M        1.79       98        CF35 5AS         15648    544.55991 28.08981
12 Maureen  52    F        1.42       73        CF72 8JY         2000     941.49038 36.20313
13   Mike  27   Male        1.92      119        CF72 9DP          250     54.01880 32.28082
> MMM
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number      BMI
1  Malcolm  9   Male        1.81       88        CF24 3AG          30     673.12263
2   Mabel  76    F        1.56       58        CF27 4HL         10000    218.71541
3  Manuel  45    M        1.67       41        SW6 4JL          400     814.88402
4    Marc  44   Male        1.76       64        SWS 3JL         64953    31.48134
5    Marc  11    M        1.72       82        BR21 4YE         4512     523.80907
6  Marie  24 Female        1.45       38        CF14 7BR          20     483.87993
7   Mari  26    F        1.61       69        NP24 3AG         10256     582.68096
8  Melody 104    F        1.67       53        NP24 3AG         5078354    337.96374
9  Melody  51    F        1.54       87        NP7 5BD          32156    116.66437
10 Montgomery  19    M        1.80       97        NP15 1AE         56512    483.16678
11   Myer  37    M        1.79       98        CF35 5AS         15648    544.55991
12 Maureen  52    F        1.42       73        CF72 8JY         2000     941.49038
13   Mike  27   Male        1.92      119        CF72 9DP          250     54.01880

```

The above code is quite long though, so we can use the `within` function which is similar to the `with` function. It lets R know you are working within a particular data frame.

```
MMM_With_BMI <- within(MMM, BMI <- Weight.in.Kg/(Height.in.Metres^2))
```

The output is shown:

```

> HMM$With_BMI <- within(HMM, BMI <- Weight.in.Kg/(Height.in.Metres^2))
> HMM_With_BMI
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number    BMI
1  Malcolm  9   Male      1.81        88     CF24 3AG       38      673.12263 26.86121
2   Mabel  76    F       1.56        58     CF27 4HL      10000    210.71541 23.83300
3  Manuel  45    M       1.67        41      SW6 4JL       400     814.88402 14.70114
4   Mark  44   Male      1.76        64      SW5 3JL      64953    31.48134 24.66116
5   Marc  11    M       1.72        82     BURSTWEE      4512    523.88091 27.71769
6  Natalie 24 Female     1.45        38     CF14 7QR       98     483.87991 18.8772
7   Matt  26    F       1.61        69     NP24 3AG      18256    582.68096 26.61934
8  Melody 104   F       1.67        53     NP24 3AG      5078354   337.96374 19.00301
9  Melody 51    F       1.54        87     NP7 5BD      32156    116.66437 36.68419
10 Montgomery 19   M       1.80        97     NP15 1AE      56512    483.16678 29.93827
11   Myer  37    M       1.79        98     CF35 SAS      15648    544.55991 28.08901
12 Maureen 52    F       1.42        73     CF72 8JY      2000    941.49038 36.20313
13   Mike  27   Male      1.92       119     CF72 90P      250     54.01880 32.28002
> HMM
   Name Age Sex Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number    BMI
1  Malcolm  9   Male      1.81        88     CF24 3AG       38      673.12263
2   Mabel  76    F       1.56        58     CF27 4HL      10000    210.71541
3  Manuel  45    M       1.67        41      SW6 4JL       400     814.88402
4   Mark  44   Male      1.76        64      SW5 3JL      64953    31.48134
5   Marc  11    M       1.72        82     BURSTWEE      4512    523.88091
6  Natalie 24 Female     1.45        38     CF14 7QR       98     483.87991
7   Matt  26    F       1.61        69     NP24 3AG      18256    582.68096
8  Melody 104   F       1.67        53     NP24 3AG      5078354   337.96374
9  Melody 51    F       1.54        87     NP7 5BD      32156    116.66437
10 Montgomery 19   M       1.80        97     NP15 1AE      56512    483.16678
11   Myer  37    M       1.79        98     CF35 SAS      15648    544.55991
12 Maureen 52    F       1.42        73     CF72 8JY      2000    941.49038
13   Mike  27   Male      1.92       119     CF72 90P      250     54.01880
> |

```

Some of the arithmetic functions available in R are shown.

Symbol	Definition	Example
** or ^	Exponential	$y=x^{**3}$ or $y=x^3$
*	Multiplication	$r=x*y$
/	Division	$d=x/y$
+	Addition	$s=x+y$
-	Subtraction	$t=x-y$

Symbol	Definition	Example
abs	Absolute value	$y=\text{abs}(x)$
floor	Integer (takes the integer part of the argument)	$y=\text{floor}(x)$
log	Natural Log	$y=\text{log}(x)$
log10	Log base 10	$y=\text{log10}(x)$
round	Rounds the argument to the nearest specified level	$y=\text{round}(x,2)$
sqrt	Square root	$t=\text{sqrt}(x)$

We can also do operations on strings, the following code replaces the variable **Sex** with the first character of **Sex** (which gets rid of the Male - M and Female - F issue).

```
MMM_With_BMI$Sex<-substr(MMM_With_BMI$Sex,1,1)
```

Some examples of string functions are shown.

Symbol	Definition	Example
substr	Outputs a substring of length L starting at position N of a string.	y=substr(string,N,L)
toupper	converts a string to upper case	y=toupper(string)
tolower	converts a string to lower case	y=tolower(string)

It's also worth checking the web for other R functions (there is a huge amount of them).

3.3.1 Renaming variables

To rename variables one can use the `rename` function from the `reshape` library (that can be installed as we have seen in previous section).

```
library(reshape)
JJJ<-rename(JJJ,c(Sex="Gender"))
```

The output is shown.

```

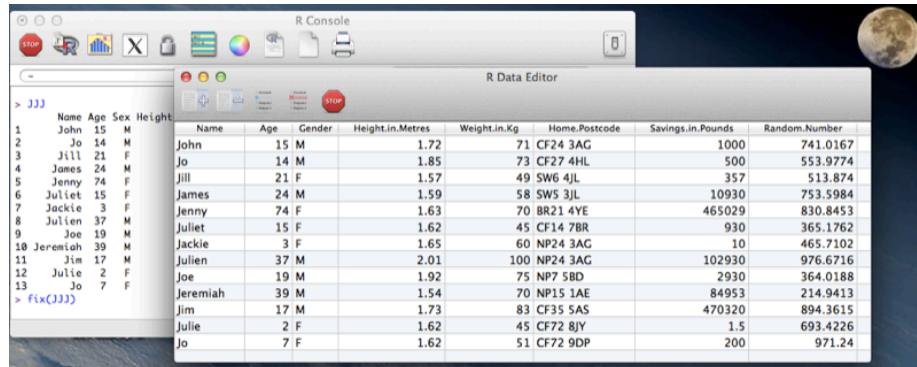
R Console
> library(reshape)
Loading required package: plyr
Attaching package: 'reshape'
The following object(s) are masked from 'package:plyr':
  rename, round_any

> JJJ<-rename(JJJ,c(Sex="Gender"))
> JJJ
   Name Age Gender Height.in.Metres Weight.in.Kg Home.Postcode Savings.in.Pounds Random.Number
1  John  15      M          1.72        71    CF24 3AG       1000.0     741.0167
2   Jo  14      M          1.85        73    CF27 4HL        500.0     553.9774
3  Jill  21      F          1.57        49    SW6 4JL        357.0     513.8740
4 James  24      M          1.59        58    SW5 3JL      10930.0     753.5984
5 Jenny  74      F          1.63        70    BR21 4YE      465029.0     830.8453
6 Juliet 15      F          1.62        45    CF14 7BR        930.0     365.1762
7 Jackie  3      F          1.65        60    NP24 3AG        10.0      465.7102
8 Julien 37      M          2.01       100    NP24 3AG      102930.0     976.6716
9  Joe  19      M          1.92        75    NP7 5BD      2930.0      364.0188
10 Jeremiah 39      M          1.54        70    NP15 1AE      84953.0     214.9413
11 Jim  17      M          1.73        83    CF35 5AS      470320.0     894.3615
12 Julie  2      F          1.62        45    CF72 8JY        1.5      693.4226
13   Jo  7      F          1.62        51    CF72 9DP       200.0      971.2400
>

```

Another option is to use the “fix” function that opens the dataset in a GUI that easily allows for modification of the dataset (including the name of the variables). Note that changes are saved on close of the fix environment.

```
fix(JJJ)
```

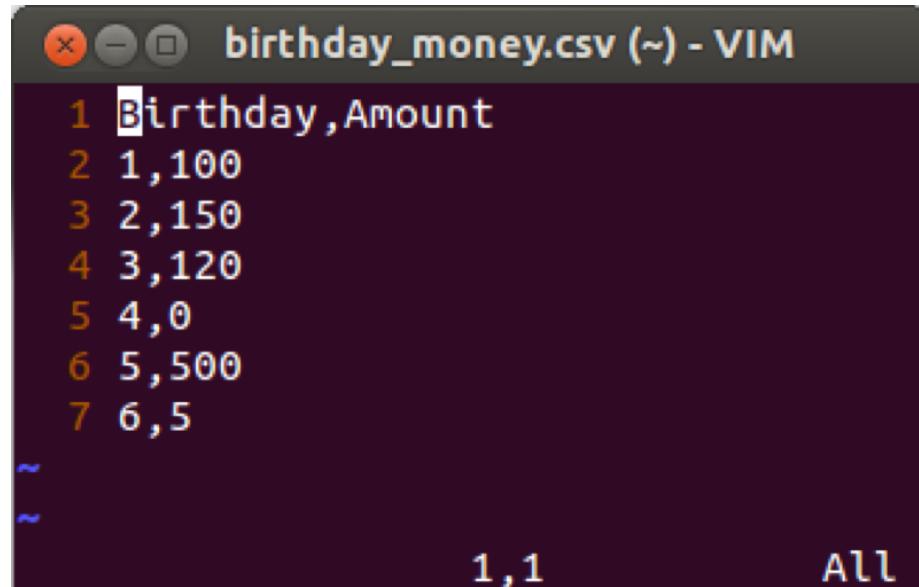


3.3.2 Operations across rows

As discussed previously, the columns of a data frame can be manipulated very easily as they are just vectors. In the next section we will see how to manipu-

late vectors using flow control statements but we will take a quick look at two functions that allow for quick and easy manipulation across rows.

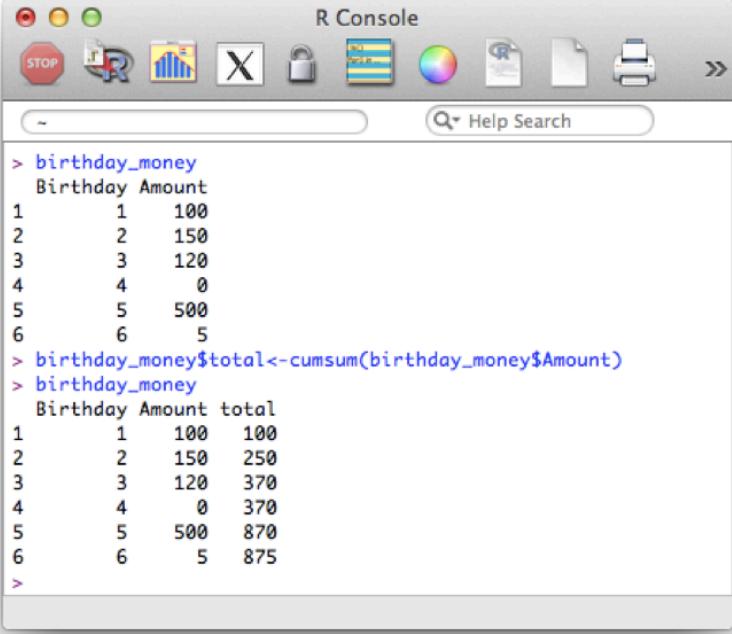
We will demonstrate this using the birthday_money.csv data set as shown.



```
1 Birthday,Amount
2 1,100
3 2,150
4 3,120
5 4,0
6 5,500
7 6,5
~
~
1,1 All
```

Suppose we want to take a cumulative sum of the birthday money, we create a new variable call total using the `cumsum` function that returns the cumulative sum of elements of a vector.

```
birthday_money$total<-cumsum(birthday_money$Amount)
```



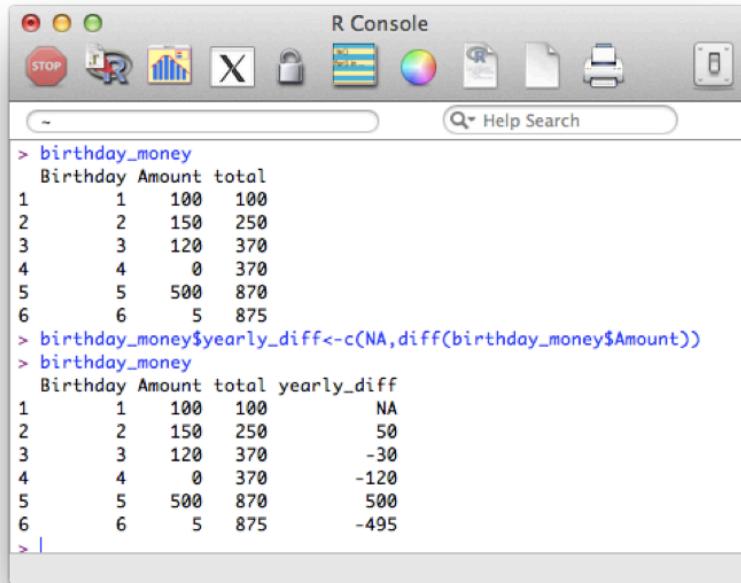
The screenshot shows the R Console window with the title "R Console". The menu bar includes "STOP", "File", "Edit", "View", "Help", and "Search". Below the menu is a toolbar with various icons. The console area displays the following R code and its output:

```
> birthday_money
   Birthday Amount
1          1    100
2          2    150
3          3    120
4          4     0
5          5    500
6          6     5
> birthday_money$total<-cumsum(birthday_money$Amount)
> birthday_money
   Birthday Amount total
1          1    100    100
2          2    150    250
3          3    120    370
4          4     0    370
5          5    500    870
6          6     5    875
>
```

Another similar tool is to use the “diff” function that calculates consecutive differences of elements of a vector:

```
birthday_money$yearly_diff<-c(NA,diff(birthday_money$Amount))
```

Note that we also include a first entry of our column “yearly_diff” as “NA”, this is because the output of diff will be shorter than the length of the original vector.



The screenshot shows the R Console window with the title "R Console". The menu bar includes "STOP", "R", "File", "Edit", "View", "Help", and "Search". Below the menu is a toolbar with icons for Stop, Run, Help, View, and others. The main area contains R code and its output:

```
> birthday_money
   Birthday Amount total
1       1    100    100
2       2    150    250
3       3    120    370
4       4      0    370
5       5    500    870
6       6      5    875
> birthday_money$yearly_diff<-c(NA,diff(birthday_money$Amount))
> birthday_money
   Birthday Amount total yearly_diff
1       1    100    100        NA
2       2    150    250        50
3       3    120    370       -30
4       4      0    370      -120
5       5    500    870        500
6       6      5    875      -495
> |
```

3.4 Handling dates in R

Dates are a particular class in R. When importing dates, they are imported as strings.

```
birthdays.csv (~) - VIM
1 Name,Birthday
2 Malcolm,09/10/1934
3 Mathieu,04/02/1998
4 Jack,02/11/2005
5 Nicolas,03/03/1978
6 Pauline,05/02/1922
7 Pascal,08/04/1954
8 Dimitri,09/03/2002
9 Julien,08/01/2004
10 Penny,10/12/1984
11 Izabela,11/09/1983
12 Paul,11/12/1984
13 Janet,12/12/1994
14 Joanna,12/09/1983
15 Iain,01/07/1985
16 Usain,11/07/1992
17 Bryan,10/09/1986
18 Richie,12/07/1984
19 Dan,02/05/1989
20 Leanne,02/09/1988
21 Juliet,01/12/1982
22 Vince,14/02/1984
23 Zoe,23/09/1983
~
~
~
```

<99C written 1,1 All

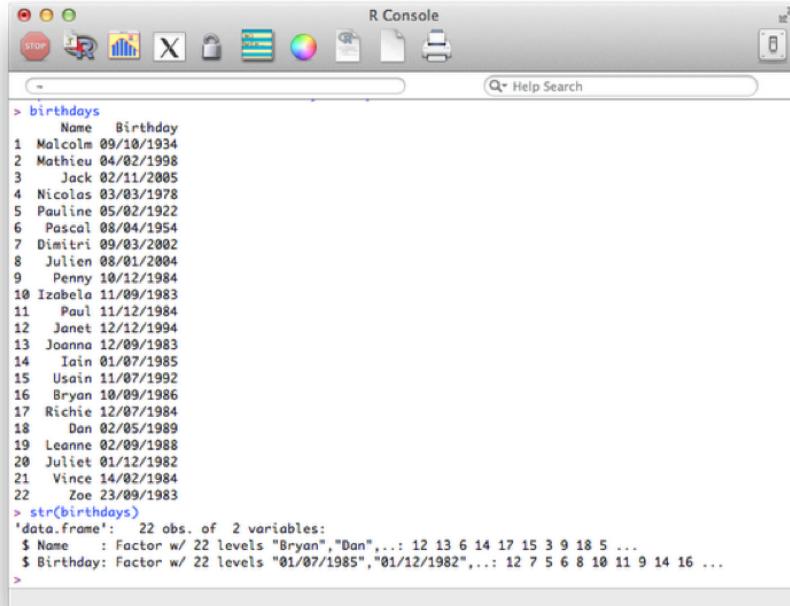
We import the file and create a data frame in the usual way:

```
birthdays<-read.csv("~/birthdays.csv")
```

Using the “str” command to view the structure of our data frame:

```
str(birthdays)
```

The output is shown confirming that the dates are recognized as strings (recall that by default `read.csv` imports strings as `factors`).



```
R Console
> birthdays
  Name Birthday
1 Malcolm 09/10/1934
2 Mathieu 04/02/1998
3 Jack 02/11/2005
4 Nicolas 03/03/1978
5 Pauline 05/02/1922
6 Pascal 08/04/1954
7 Dimitri 09/03/2002
8 Julien 08/01/2004
9 Penny 10/12/1984
10 Izabela 11/09/1983
11 Paul 11/12/1984
12 Janet 12/12/1994
13 Joann 12/09/1983
14 Iain 01/07/1985
15 Usain 11/07/1992
16 Bryan 10/09/1986
17 Richie 12/07/1984
18 Dan 02/05/1989
19 Leanne 02/09/1988
20 Juliet 01/12/1982
21 Vince 14/02/1984
22 Zoe 23/09/1983
> str(birthdays)
'data.frame': 22 obs. of 2 variables:
 $ Name : Factor w/ 22 levels "Bryan","Dan",...: 12 13 6 14 17 15 3 9 18 5 ...
 $ Birthday: Factor w/ 22 levels "01/07/1985","01/12/1982",...: 12 7 5 6 8 10 11 9 14 16 ...
```

In this current format if we tried to carry out any mathematical manipulation of the dates we would not succeed. We can however tell R that certain variables are dates. We do this using the “`as.dates`” function by describing the format our dates are in:

```
birthdays$Birthday<-as.Date(birthdays$Birthday,"%d/%m/%Y")
```

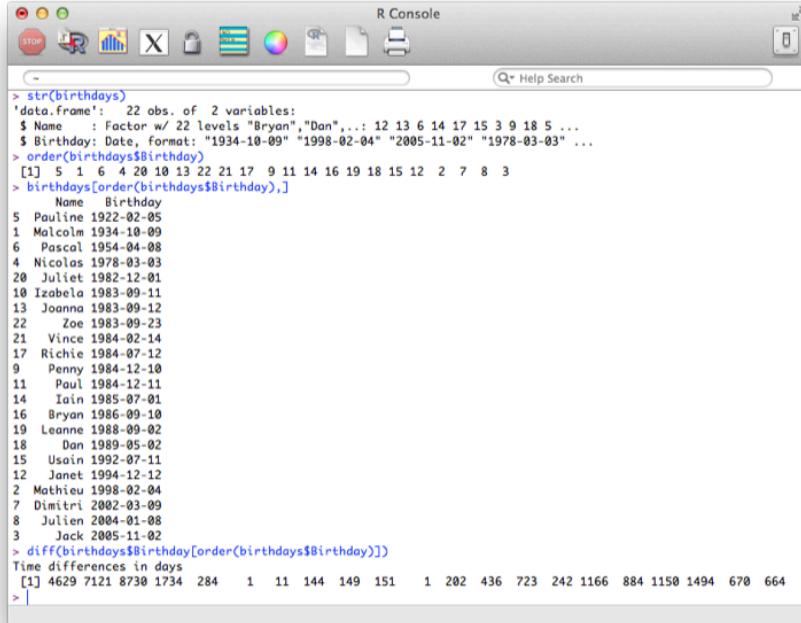
The format is indicated using “%x” where “x” can be of various formats as show.

Symbol	Meaning	Example
d	Day as a number	01-31
a	Abbreviated weekday	Mon
A	Weekday	Monday
m	Month	01-12
b	Abbreviated month	Jan
B	Month	January
y	2 digit year	84
Y	4 digit year	1984

We'll now check the structure of our data frame, re-order (using the `order` function - that returns the indices of the elements of a vector in order) our birthdays and calculate the difference between birthdays (using the `diff` function).

```
str(birthdays$Birthday)
order(birthdays$Birthday)
sorted<- birthdays[order(birthdays$Birthday),]
diff(birthdays$Birthday[order(birthdays$Birthday)])
```

The output of all this is shown.



The screenshot shows an R console window with the title "R Console". The window contains the following R session:

```

> str(birthdays)
'data.frame': 22 obs. of 2 variables:
 $ Name : Factor w/ 22 levels "Bryan","Dan",...
 $ Birthday: Date, format: "1934-10-09" "1998-02-04" "2005-11-02" "1978-03-03" ...
> order(birthdays$Birthday)
[1] 5 1 6 4 20 10 13 22 21 17 9 11 14 16 19 18 15 12 2 7 8 3
> birthdays[order(birthdays$Birthday),]
   Name      Birthday
5  Pauline 1922-02-05
1  Malcolm 1934-10-09
6  Pascal 1954-04-08
4  Nicolas 1978-03-03
20 Juliet 1982-12-01
10 Isabela 1983-09-11
13 Joanna 1983-09-12
22 Zoe 1983-09-23
21 Vince 1984-02-14
17 Richie 1984-07-12
9 Penny 1984-12-10
11 Paul 1984-12-11
14 Iain 1985-07-01
16 Bryan 1986-09-10
19 Leanne 1988-09-02
18 Dan 1989-05-02
15 Usain 1992-07-11
12 Janet 1994-12-12
2 Mathieu 1998-02-04
7 Dimitri 2002-03-09
8 Julien 2004-01-08
3 Jack 2005-11-02
> diff(birthdays$Birthday[order(birthdays$Birthday)])
Time differences in days
[1] 4629 7121 8738 1734 284 1 11 144 149 151 1 202 436 723 242 1166 884 1150 1494 670 664
>

```

4 Chapter 4 Programming

4.1 Flow Control

A huge part of programming (in any language) is the use of so called “conditional statements” that allow for flow control. We do this in R using “if” statements.

There are two types of “if” statements in R. The simple “if” statement as shown below:

```
x<-39
if (x>20) y<-1
```

We can use this in conjunction with an “else” statement:

```
x<-19
if (x>20) y<-1 else y<-0
```

Finally, the if-else call is a function and as such we can rewrite the above code as:

```
y<- if (x>20) 1 else 0
```

Finally we can include multiple commands as outcomes of an if statement by using “{}”:

```
x<-20
if (x==21) {
y<-1
z<-"T"
} else{
y<-0
z<-"F"}
```

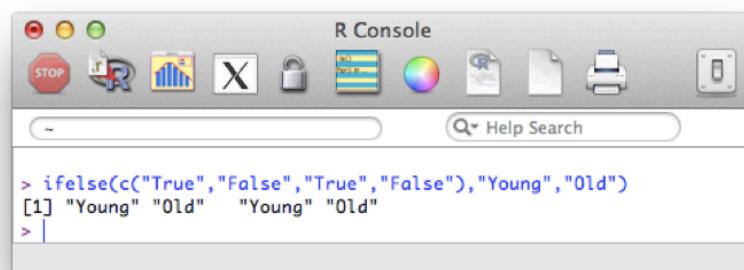
The above statements checks a single value and whilst we'll learn in the next section how to loop over sets of values it is very much worth learning how to use the ‘vectorized’ form of the if statement: the “ifelse” command. The general syntax is given below:

```
ifelse(Boolean_Vector, Outcome_If_True, Outcome_If_False)
```

An example of this is given below:

```
ifelse(c("True", "False", "True", "False"), "Young", "Old")
```

The output is shown.



Using this and our knowledge of filtering we see how we can create new variables using the ifelse statement. The following code creates a new variable “age_group”:

```
MMMJJJ$Age_group<-ifelse(MMM$Age<30,"Young","Old")
MMMJJJ[c("Name","Age_group")]
```

The output is shown.

R Console

```
> MMMJJJ$Age_group<-ifelse(MMM$Age<30,"Young","Old")
> MMMJJJ[c("Name","Age_group")]
   Name Age_group
1   John    Young
2     Jo      Old
3   Jill      Old
4  James      Old
5  Jenny    Young
6  Juliet    Young
7 Jackie    Young
8  Julien      Old
9    Joe      Old
10 Jeremiah  Young
11   Jim      Old
12  Julie      Old
13    Jo    Young
14 Malcom    Young
15  Mabel      Old
16  Manuel      Old
17    Mark      Old
18   Marc    Young
19  Marie    Young
20   Mari    Young
21  Melody      Old
22  Melody      Old
23 Montgomery  Young
24   Myer      Old
25 Maureen      Old
26   Mike    Young
> |
```

Some of the comparison operators that can be used in conjunction with 'if' statements are shown.

Symbol	Mnemonic	Definition
<	Lt	Less than
<=	Le	Less than or equal to
>	Gt	Greater than
>=	Ge	Greater than or equal to
==	Eq	Logical Equal to
!=	Ne	Not equal to

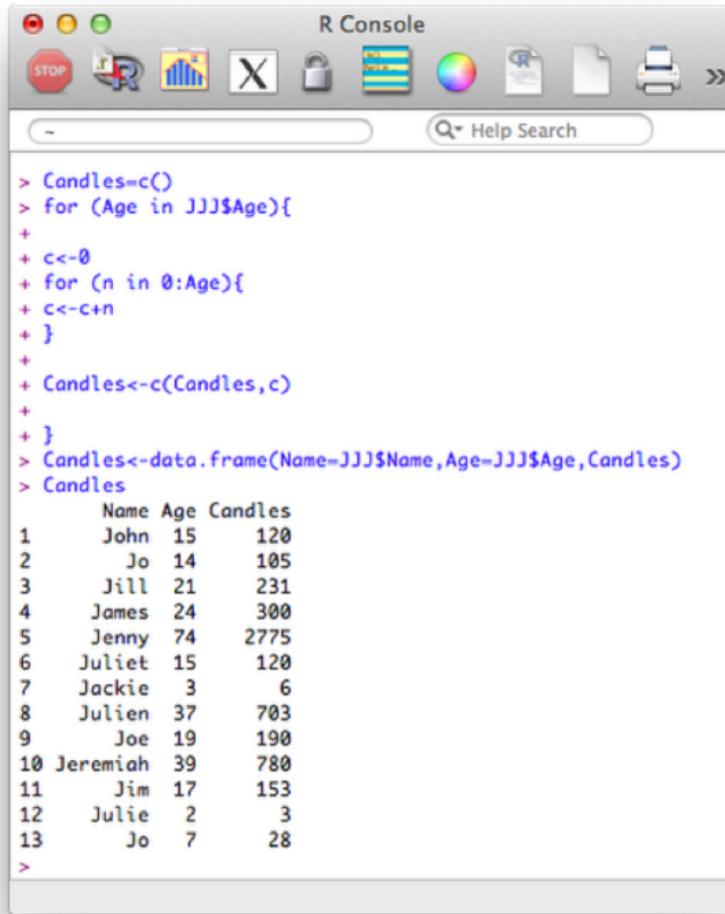
A further important notion in programming is the notion of loops. There are two types of loops that we will consider:

1. for
2. while

The for loop allows us to compute iterative procedures. As with most things in R, the for loop iterates a value over a vector. The following code outputs the total number of birthday candles that would have been used on everyone's birthday cake in the JJJ data set.

```
Candles=c()
for (Age in JJJ$Age){
  c<-0
  for (n in 0:Age){
    c<-c+n
  }
  Candles<-c(Candles,c)
}
Candles<-data.frame(Name=JJJ>Name, Age=JJJ$Age, Candles)
```

The first statement creates an empty vector called "Candles". The first for loop, loops over the age variable in the JJJ data set ("0:age" is in fact a short way of writing a vector of integers from 0 to age). For each of those values of age we use a second for loop to sum the total number of candles and concatenate that value to the vector Candles. Finally we create a new data set Candles by concatenating the various vectors required (note that we're also renaming certain variables here).



The screenshot shows the R Console window with the title "R Console". The menu bar includes "File", "Edit", "View", "Help", and "Search". Below the menu is a toolbar with icons for Stop, Run, Plot, X, Lock, Help, and Print. The main area contains R code and its output:

```
> Candles=c()
> for (Age in JJJ$Age){
+ 
+ c<-0
+ for (n in 0:Age){
+ c<-c+n
+ }
+
+ Candles<-c(Candles,c)
+
+
> Candles<-data.frame(Name=JJJ>Name,Age=JJJ$Age,Candles)
> Candles
   Name Age Candles
1  John  15     120
2    Jo  14     105
3   Jill  21     231
4 James  24     300
5  Jenny  74    2775
6 Juliet  15     120
7 Jackie   3      6
8 Julien  37     703
9   Joe  19     190
10 Jeremiah  39     780
11   Jim  17     153
12 Julie   2      3
13    Jo  7      28
>
```

Note that in general this is not the most efficient way of undertaking things in R. Vectorized versions of the above are much faster (we won't cover these here). Another improvement for the above code is to create the vector Candles initially as a vector of the correct size and type. For example we can create a numeric vector of a certain length using the following code (all initial values will be set to 0):

```
Candles=vector("numeric",length=length(JJJ$Age))
```

Using this the above code would be written as:

```

Candles=vector("numeric",length=length(JJJ$Age))
k<-0
for (Age in JJJ$Age){
k<-k+1
c<-0
for (n in 0:Age){
c<-c+n
}
Candles [k]<-c
}
Candles<-data.frame(Name=JJJ>Name,Age=JJJ$Age,Candles)

```

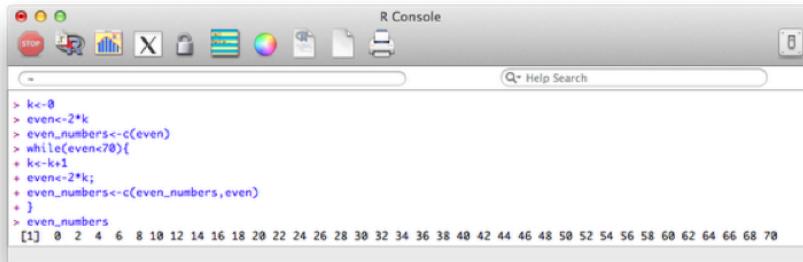
The second type of loop we will consider is the do while loop. This loop checks a condition before carrying out an operation. The following code creates a vector with all even numbers less than 70:

```

k<-0
even<-2*k
even_numbers<-c(even)
while(even<70){
k<-k+1
even<-2*k;
even_numbers<-c(even_numbers,even)
}

```

The output is shown.



4.2 Functions

One of the great capacities of R is the ease with which one can create new functions. The general syntax for this is given by:

```
myfunction <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

The return statement is very important as it indicates the “result” of the function. This can be any R object, a vector, a data frame etc... Note that is can also be omitted, as long as the last command is what you want returned.

The following code creates a function called “f1” that adds 3 to a number if it is even and adds 2 to a number if it is odd.

```
f1 <- function(x){
  r <- if (x%%2==0) x+3 else x+2
  return(r)
}
```

To run the function we would then use it like any other R function. For example the following would return 11.

```
f1(9)
```

(The %% command return the modulo of the first number with respect to the second)

We can also create a function with no arguments that simply replicates shorthand for some code:

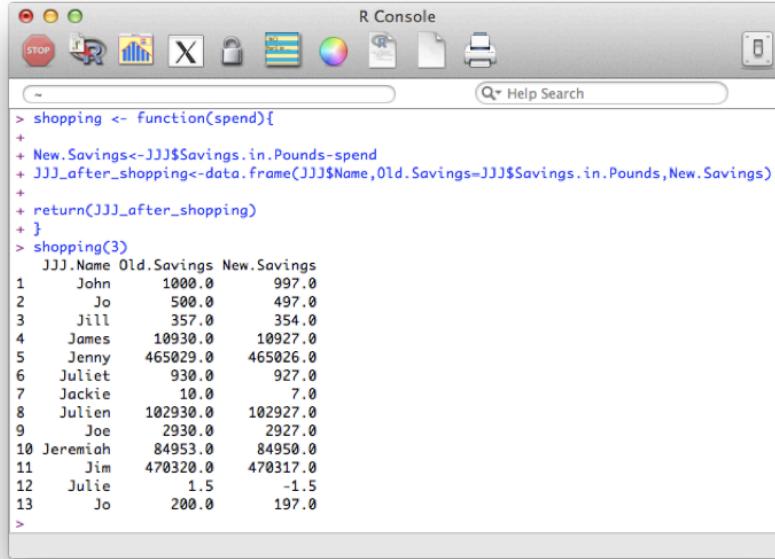
```
My_plot <- function(){
  r<-hist(JJJ$Height.in.Metres)
  return(r)
}
```

```
My_plot()
```

The following code defines a function that creates a new dataset entitled “JJJ_after_shopping” that subtracts a quantity from the savings variable in the JJJ dataset:

```
shopping <- function(spend){
  New.Savings<-JJJ$Savings.in.Pounds-spend
  JJJ_after_shopping<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
  return(JJJ_after_shopping)
}
```

Note that this function makes use of recycling (when creating the New.Savings vector).



```
R Console
> shopping <- function(spend){
+   New.Savings<-JJJ$Savings.in.Pounds-spend
+   JJJ_after_shopping<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
+
+   return(JJJ_after_shopping)
+ }
> shopping(3)
  JJJ.Name Old.Savings New.Savings
1  John      1000.0     997.0
2   Jo       500.0      497.0
3  Jill      357.0      354.0
4 James    10930.0    10927.0
5 Jenny   465029.0    465026.0
6 Juliet     930.0      927.0
7 Jackie     10.0       7.0
8 Julien   102930.0    102927.0
9  Joe      2930.0      2927.0
10 Jeremiah  84953.0    84950.0
11 Jim     470320.0    470317.0
12 Julie      1.5      -1.5
13 Jo        200.0      197.0
>
```

We can of course define functions with multiple arguments as below:

```
shopping <- function(spend,trips){
  New.Savings<-JJJ$Savings.in.Pounds-trips*spend
  JJJ_after_shopping<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
  return(JJJ_after_shopping)
}
```

It's also possible to set certain values as defaults:

```
shopping <- function(spend,trips=1){
  New.Savings<-JJJ$Savings.in.Pounds-trips*spend
  JJJ_after_shopping<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
  return(JJJ_after_shopping)
}
```

4.3 Vectorising

In general the for loops we have seen can be written in a much more efficient way using function and various forms of the apply function (which apply functions over vectors, lists and matrices):

1. apply
2. lapply
3. sapply
4. mapply

Note that an array in R is a very generic data type; it is a general structure of up to eight dimensions. For specific dimensions there are special names for the structures. A zero dimensional array is a scalar or a point; a one dimensional array is a vector; and a two dimensional array is a matrix. The general syntax of the apply function is given below:

```
apply(matrix,margin,function)
```

We have not yet seen matrices but they are relatively simple to understand: 2 dimensional objects. The following code produces a 2 by 3 matrix:

```
mat<-matrix(c(1,2,3,4,5,6),2,3)
mat
```

The “margin” option (either 1,2 or both (1:2)) simply tells R what dimension to apply the required function to, experiment with the following:

```
apply(mat,1,mean)
apply(mat,2,mean)
apply(mat,1:2,mean)
```

We can use the apply function on data frames and vectors but in general it will be easier to use the “lapply” function which simply applies a function to a 1 dimensional object. The lapply function becomes especially useful when dealing with data frames. In R the data frame is considered a list and the variables in the data frame are the elements of the list. We can therefore apply a function to all the variables in a data frame by using the lapply function. Note that unlike in the apply function there is no margin argument since we are just applying the function to each component of the list. The following code simply returns the square roots of a vector.

```
lapply(c(1,2,3,4,5),sqrt)
```

Note that the above returns a list (an R object that we will not pay much attention to here). We can get a vector by using the following:

```
unlist(lapply(c(1,2,3,4,5),sqrt))
```

The “sapply” function is simply a version of lapply that by default returns the most appropriate object type. The following code gives the exact same result as above:

```
sapply(c(1,2,3,4,5),sqrt)
```

Finally, there exists a multivariate example of the above function which allows us to pass multiple arguments to a function. The following code defines a simple function:

```
simple_function<-function(x,y) x/y
```

We can now apply this function so that it takes the consecutive ratios of two vectors:

```
mapply(simple_function,1:4,4:1)
```

With these functions we can drastically improve the performance of R code. The following reproduces code from before:

```
my_sum<-function(x){  
  sum(x)  
}  
  
sapply(JJJ$Age,my_sum(1:x))
```

Note that there is no need to actually define the function we can refer directly to the function object:

```
sapply(JJJ$Age,function(x) sum(1:x))
```

4.4 Handling strings

SAS is a macro language and philosophically macros allow a user to substitute pieces of text for a variable, and evaluate the result. R is not a macro language and thus does the opposite: evaluates the arguments and then uses the values.

The paste command allows us to concatenate strings. The following code outputs the string “Hello-World”. Note that we can use any string as a separator (include the empty string “”).

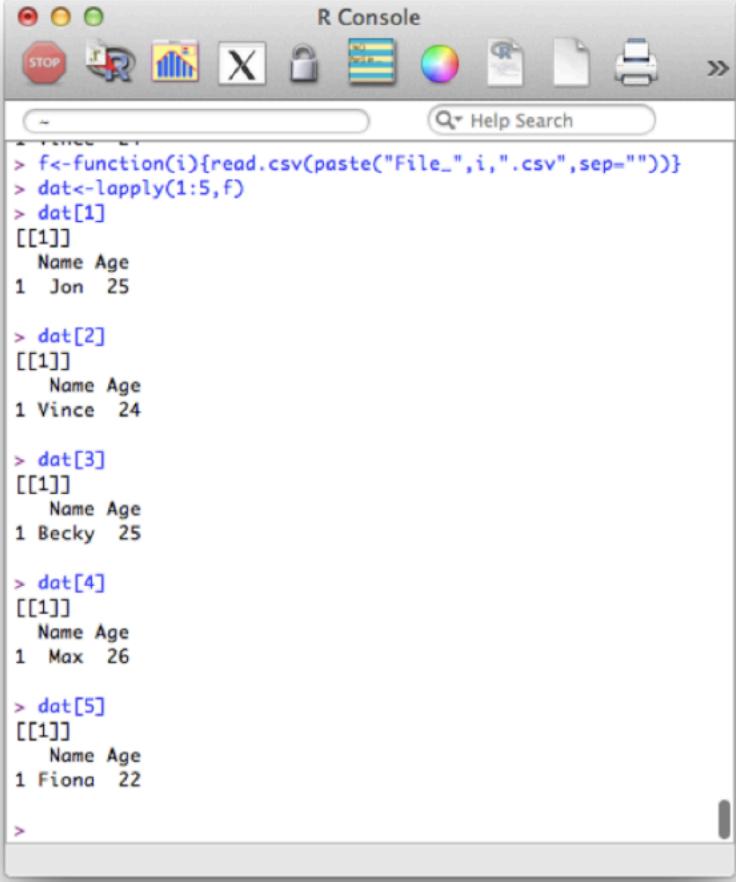
```
x<-"Hello"  
paste(x,"World",sep="-")
```

This immediately allows for quite complex manipulation of data files. For example the following code, imports the 5 datafiles File_1.csv - File_5.csv:

```
f<-function(i){read.csv(paste("File_",i,".csv",sep=""))}
dat<-lapply(1:5,f)
```

Note that this piece of code introduces a new structure a bit more formally. The object “dat” is here a list and we use the “lappy” function (we haven’t seen this yet) to apply the newly created function “f” (that imports data).

The output is shown.



The screenshot shows the R Console window with the title "R Console". The window has a toolbar with various icons for file operations like Open, Save, Print, and Help. Below the toolbar is a search bar labeled "Help Search". The main area contains the R command-line interface. The user has run the following R code:

```
> f<-function(i){read.csv(paste("File_",i,".csv",sep=""))}
> dat<-lapply(1:5,f)
> dat[1]
[[1]]
  Name Age
1 Jon  25

> dat[2]
[[1]]
  Name Age
1 Vince 24

> dat[3]
[[1]]
  Name Age
1 Becky 25

> dat[4]
[[1]]
  Name Age
1 Max  26

> dat[5]
[[1]]
  Name Age
1 Fiona 22
```

The output shows five data frames, each containing two columns: "Name" and "Age". The first data frame has one row with Name "Jon" and Age 25. The second has one row with Name "Vince" and Age 24. The third has one row with Name "Becky" and Age 25. The fourth has one row with Name "Max" and Age 26. The fifth has one row with Name "Fiona" and Age 22.

We can also revisit a previous function (the shopping function) and create a

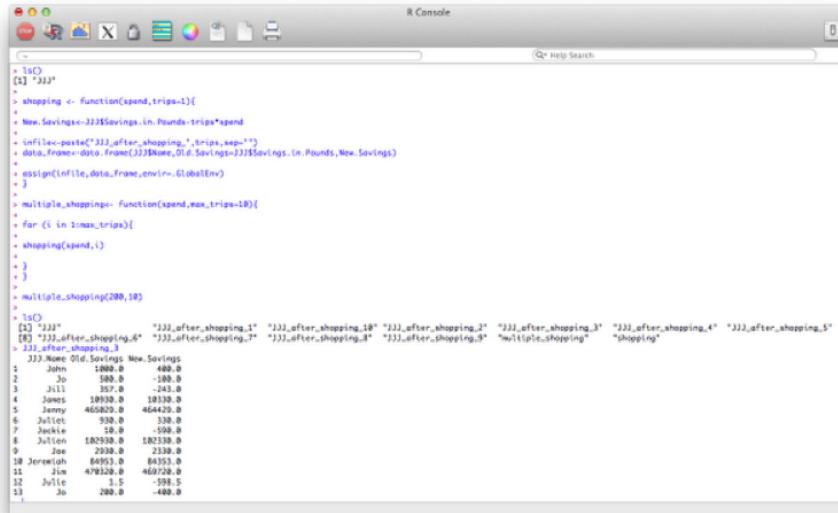
different data set for every value of spend. We'll even make this a bit more complicated and nest functions so that we can repeat this operation for various values of the variable "spend".

```
shopping <- function(spend,trips=1){
  New.Savings<-JJJ$Savings.in.Pounds-trips*spend
  infile<-paste("JJJ_after_shopping_",trips,sep="")
  data_frame<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
  assign(infile,data_frame,envir=.GlobalEnv)
}

multiple_shopping<- function(spend,max_trips=10){

  for (i in 1:max_trips){
    shopping(spend,i)
  }
}
```

Note the extra option that has been passed to the "assign" command "envir=.GlobalEnv". This is to ensure that the data sets created in the function are global (i.e. are still there when the function stops running).



The screenshot shows an R console window with the title 'R Console'. The window displays the following R session:

```

> ls()
[1] "JJJ"
> 
> shopping <- function(spend,trips=1){
+   New.Savings<-JJJ$Savings.in.Pounds-trips*spend
+   infile<-paste("JJJ_after_shopping_",trips,sep="")
+   data_frame<-data.frame(JJJ>Name,Old.Savings=JJJ$Savings.in.Pounds,New.Savings)
+   assign(infile,data_frame,envir=.GlobalEnv)
+ }

> multiple_shopping<- function(spend,max_trips=10){
+   for (i in 1:max_trips){
+     shopping(spend,i)
+   }
+ }

> multiple_shopping(200,10)
> 
> ls()
[1] "JJJ"
[2] "JJJ_after_shopping_1"
[3] "JJJ_after_shopping_2"
[4] "JJJ_after_shopping_3"
[5] "JJJ_after_shopping_4"
[6] "JJJ_after_shopping_5"
[7] "JJJ_after_shopping_6"
[8] "JJJ_after_shopping_7"
[9] "JJJ_after_shopping_8"
[10] "JJJ_after_shopping_9"
[11] "JJJ_after_shopping_10"
[12] "multiple_shopping"
[13] "Old.Savings"
[14] "New.Savings"

1  John      1800.0    400.0
2  Jo       500.0    -100.0
3  Jim      2000.0    -200.0
4  Jones    1800.0    1810.0
5  Jenny   46500.0   464420.0
6  Juliet   930.0     330.0
7  Julie    100.0     -100.0
8  Julien   18250.0   18230.0
9  Jon      2300.0     2300.0
10 Jerome   64933.0   64753.0
11 Jim      47932.0   46973.0
12 Julie    3.5      -598.5
13 Jo       200.0     -400.0

```

4.5 Memory and scripts.

In this section we will take a brief look at how R handles the “workspace”. If you have already quit R you would have seen the prompt “Save workspace image?”. If you answer “yes” then R saves various things to various files (in the current working directory): 1. .Rdata holds all the objects (data frames, vectors, functions etc...) currently in memory (note that this file is written in an R specific file and so can’t be read). 2. .Rhistory holds all the commands used (and so can be recalled).

Being prompted whether or not to save the workspace is helpful (in my opinion) as you can simply open an R session to try a few things and not save (similar to using the work library in SAS). It is possible to save the workspace image as you go (this is worthwhile in case your system happens to crash):

```
save.image()
```

Note: we can leave the argument of the “image” function empty (as above), in which case the file will be saved in the current directory. We can also pass the required location to the “image” function.

It is also possible to save particular objects to particular files as well as load files but we won’t go into that here.

One final aspect to consider is that of running script files from the command line. We do this using the “source” command. Note that this command executes all the code in the script as if it was typed in one after the other. To see this let us write the following code in a text file (saving it as “first_script.r” on the desktop for example):

```
x<-c(1,2,3,4)
y<-c(1,0)
print(x+y)
print(x*y)
```

We then run the script using the following code:

```
source("~/Desktop/first_script.r")
```

Repetitive command that are run often (for example, routine data analysis) can be saved as scripts and called if and when new data is received.

5 Chapter 5 Further packages

In this chapter we will examine three packages in particular:

1. sqldf: a package allowing for the use of sql syntax in R
2. ggplot2: a powerful package for the plotting of data
3. twitteR: a package that allows for the data mining of twitter

5.1 sqldf

5.1.1 Basic SQL

SQL is a language designed for querying and modifying databases. Used by a variety of database management software suites:

1. Oracle
2. Microsoft ACCESS
3. SPSS

SQL uses one or more objects called TABLES where: rows contain records (observations) and columns contain variables.

To use SQL in R we need to use the sqldf package.

The following code creates a data set called test in the work library as a copy of the mmm data set:

```
test<-sqldf("select * from MMM")
```

The “*” command tells R to take all variables from the data set. We can however specify exactly what variables we want:

```
test<-sqldf("select Name,Age,Sex from MMM")
```

We can also create new variables:

```
test<-sqldf("select Name,Age,Sex,Weight_in_Kg/(power(Height_in_Metres,2)) as bmi from MMM")
```

Some of the SQL operators are shown.

<code>abs()</code>	Gives the absolute value
<code>ceil()</code>	Gives the ceiling
<code>floor()</code>	Gives the floor
<code>exp()</code>	Gives the exponential
<code>ln()</code>	Gives the natural log
<code>mod()</code>	Performs modulo arithmetic
<code>power()</code>	Gives the power.

5.1.2 Further sql

In this section we'll take a look at what else R can do with sql. For the purpose of the following examples let's write a new data set:

```
Var1<-c("A", "A", "B", "C", "C")
Var2<-c(1,1,1,2,2)
Var3<-c("A", "A", "A", "B", "C")
Var4<-c(2,2,1,2,1)
Var5<-c("B", "B", "C", "D", "E")

example<-data.frame(Var1,Var2,Var3,Var4,Var5)
```

Some simple SQL code very easily helps us to get rid of duplicate rows (this can be very helpful when handling real data). To do this we use the “distinct” keyword.

```
sqldf("select distinct * from example")
```

We can also select particular variables:

```
sqldf("select distinct Var1,Var2,Var3 from example")
```

We can also use the “where” statement to select variables that obey a particular condition:

```
sqldf("select * from example where Var2<=Var4")
```

We can sort data sets using the “order by” keyword:

```
sqldf("select distinct * from example order by Var1")
```

A very nice application of SQL is in the aggregation of summary statistics. The following code creates a new variable that gives the average value of var2. The value of this variable is the same for all the observations:

```
sqldf("select avg(Var2) as average_of_Var2 from example")
```

We could however get something a bit more useful by aggregating the data using a “group” statement:

```
sqldf("select Var1,avg(Var2) as average_of_Var2 from example group by Var1")
```

5.1.3 Joining tables with SQL

A very common use of SQL is to carry out “joins” which are equivalent to a merger of data sets. There are 4 types of joins to consider:

1. inner join
 1. output table only contains rows common to all tables
 2. variable attributes taken from left most table
2. outer join left
 1. output table contains all rows contributed by the left table
 2. variable attributes taken from left most table
3. outer join right
 1. output table contains all rows contributed by the right table
 2. variable attributes taken from right most table
4. outer join full
 1. output table contains all rows contributed by all tables
 2. variable attributes taken from left most table

To work with these examples let's use the data sets created with the following code:

```

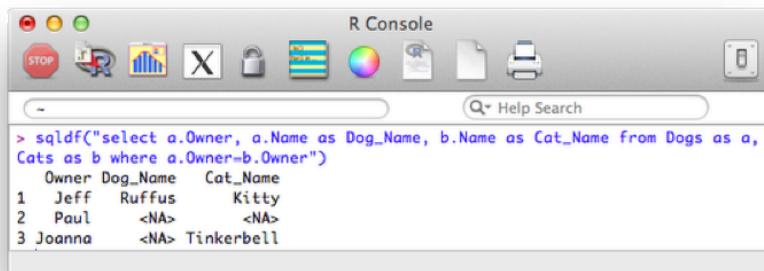
Owner<-c("Jeff","Janet","Paul","Joanna")
Name<-c("Ruffus","Sam",NA,NA)
Dogs<-data.frame(Owner,Name)

Owner<-c("Jeff","Paul","Joanna","Vince")
Name<-c("Kitty",NA,"Tinkerbell","Chick")
Cats<-data.frame(Owner,Name)

```

The following code carries out an inner join of these two datasets also changing the name of the “Name” variable depending on which data set it was from.

```
sqldf("select a.Owner, a.Name as Dog_Name, b.Name as Cat_Name from Dogs as a, Cats as b where a.Owner=b.Owner")
```



The screenshot shows an R console window with a title bar "R Console". The window contains the following text:

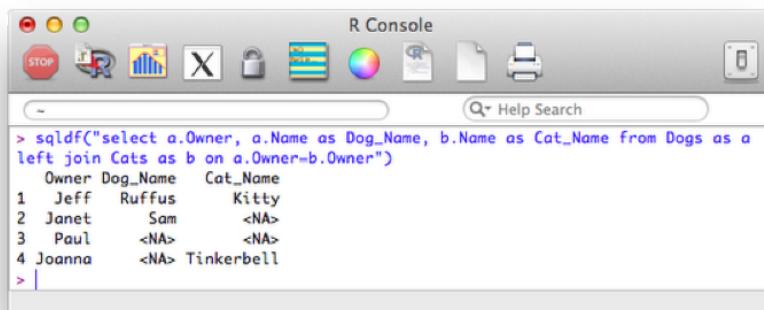
```

> sqldf("select a.Owner, a.Name as Dog_Name, b.Name as Cat_Name from Dogs as a,
  Cats as b where a.Owner=b.Owner")
   Owner Dog_Name   Cat_Name
1  Jeff    Ruffus     Kitty
2  Paul      <NA>      <NA>
3 Joanna     <NA> Tinkerbell

```

The following code carries out a left outer join, the output of which is show.

```
sqldf("select a.Owner, a.Name as Dog_Name, b.Name as Cat_Name from Dogs as a left join Cats as b
```



The screenshot shows an R console window with a title bar "R Console". The window contains the following text:

```

> sqldf("select a.Owner, a.Name as Dog_Name, b.Name as Cat_Name from Dogs as a
  left join Cats as b on a.Owner=b.Owner")
   Owner Dog_Name   Cat_Name
1  Jeff    Ruffus     Kitty
2 Janet      Sam      <NA>
3  Paul      <NA>      <NA>
4 Joanna     <NA> Tinkerbell
> |

```

Right and full outer joins are not yet supported in sqldf however they can

actually be obtained by simply using the “merge” function (as discussed in Chapter 3).

5.2 ggplot2

This is an extremely powerful package that allows for the creation of publication quality plots with ease. There are two basic functions in ggplot2:

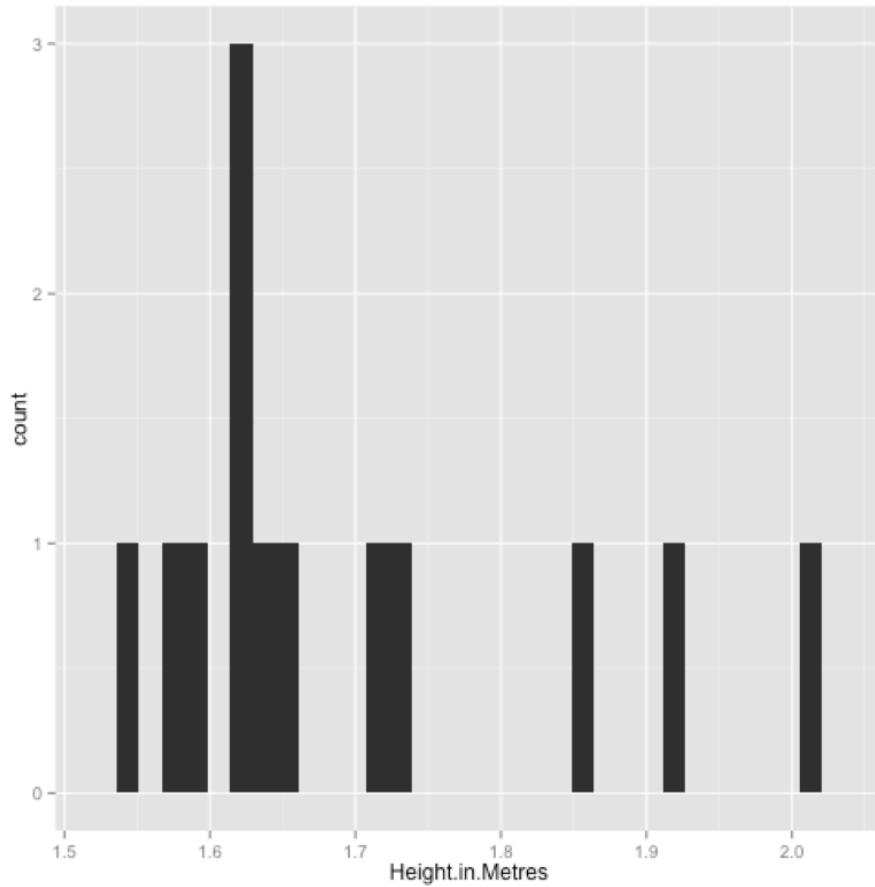
1. qplot which allows us obtain quick graphs
2. ggplot which gives us more control of granularity (we will not go into it here)

5.2.1 Basic plots with qplot

The qplot command is very similar to the plot command in that it will often produce the plot required based on the inputs. To obtain a histogram of the Height.in.Metres variable of the JJJ data set we simply use:

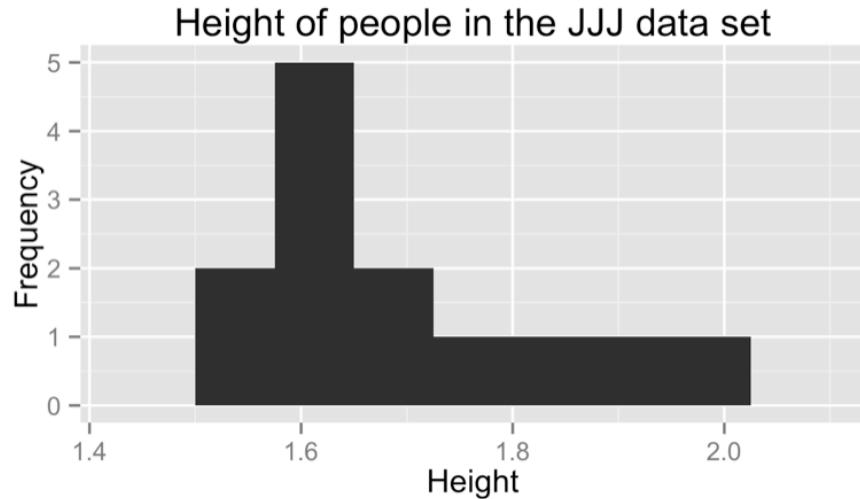
```
qplot(data=JJJ,x=Height.in.Metres)
```

This produces the plot shown.



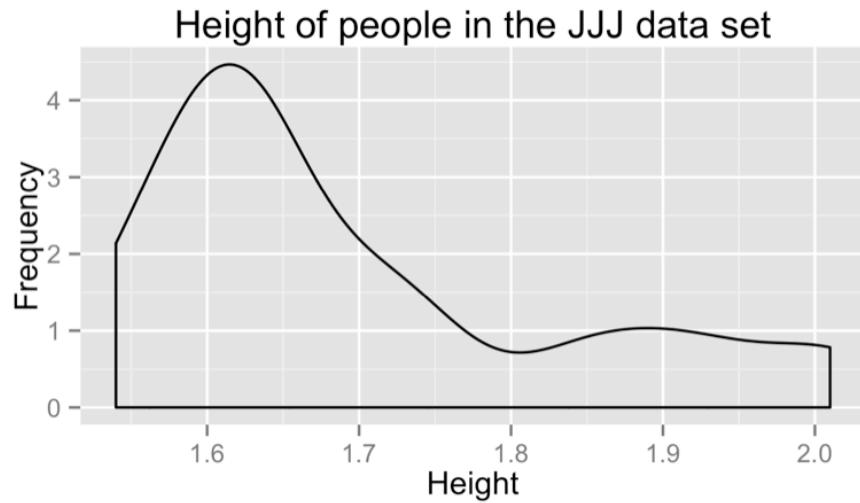
We can improve this by changing the bin width, including a title and changing the labels for the x axis and y axis.

```
qplot(data=JJJ,x=Height.in.Metres,binwidth=.075,main="Height of people in the JJJ data set",xla
```



We can obtain a density plot corresponding to the above by using the “density” option for the “geom” argument as shown:

```
qplot(data=JJJ,x=Height.in.Metres,binwidth=.075,main="Height of people in the JJJ data set",xla
```

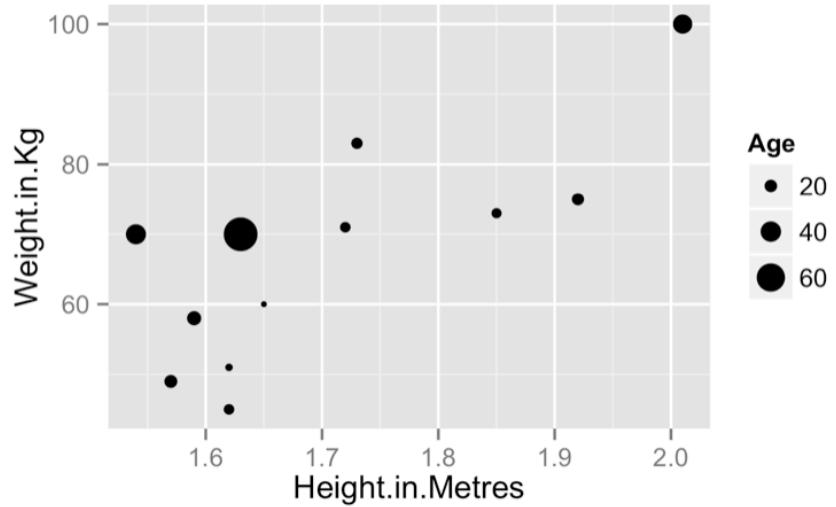


If we pass two vectors to qplot we obtain a scatter plot:

```
qplot(data=JJJ,x=Weight.in.Kg,y=Height.in.Metres)
```

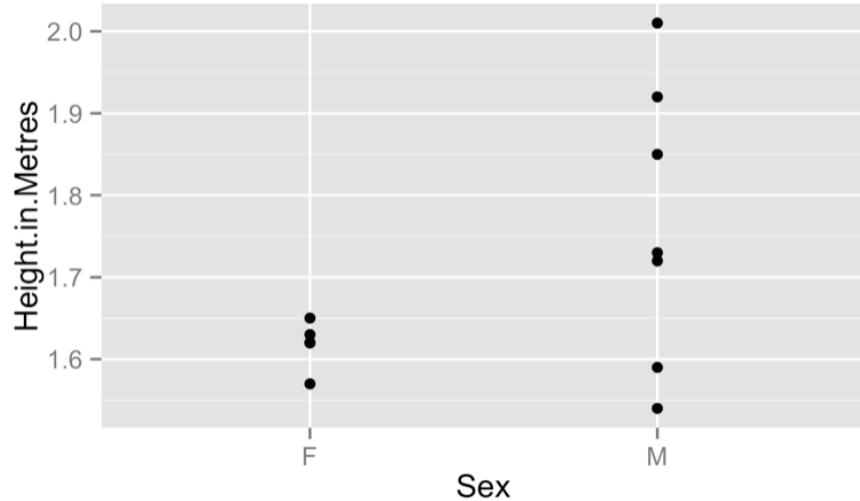
we can also pass qplot a “size” argument to obtain the graph shown:

```
qplot(data=JJJ,x=Height.in.Metres,y=Weight.in.Kg,size = Age)
```



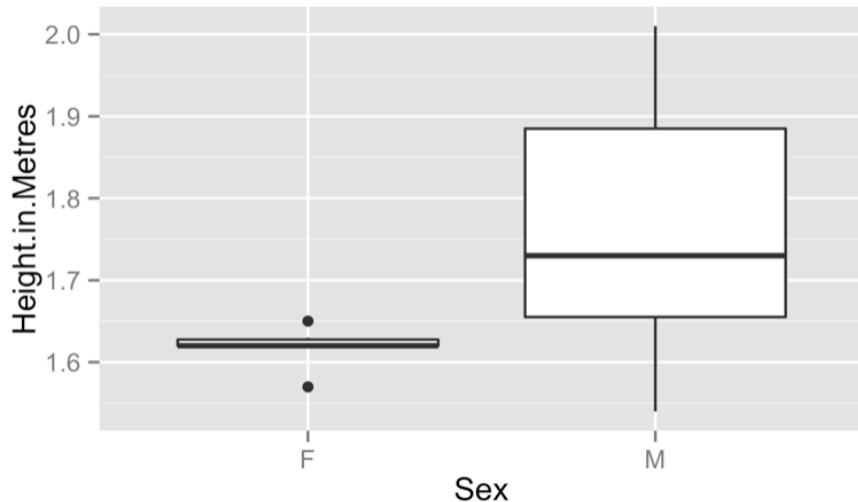
We can of course obtain scatter plots against categorical variables as shown:

```
qplot(data=JJJ,x=Sex,y=Height.in.Metres)
```



We can pass “boxplot” as the “geom” argument to get a boxplot as shown.

```
qplot(data=JJJ,x=Sex,y=Height.in.Metres,geom='boxplot')
```



5.2.2 Advanced features

We can add various features to our scatter plot. The following code just plots a line between all the points:

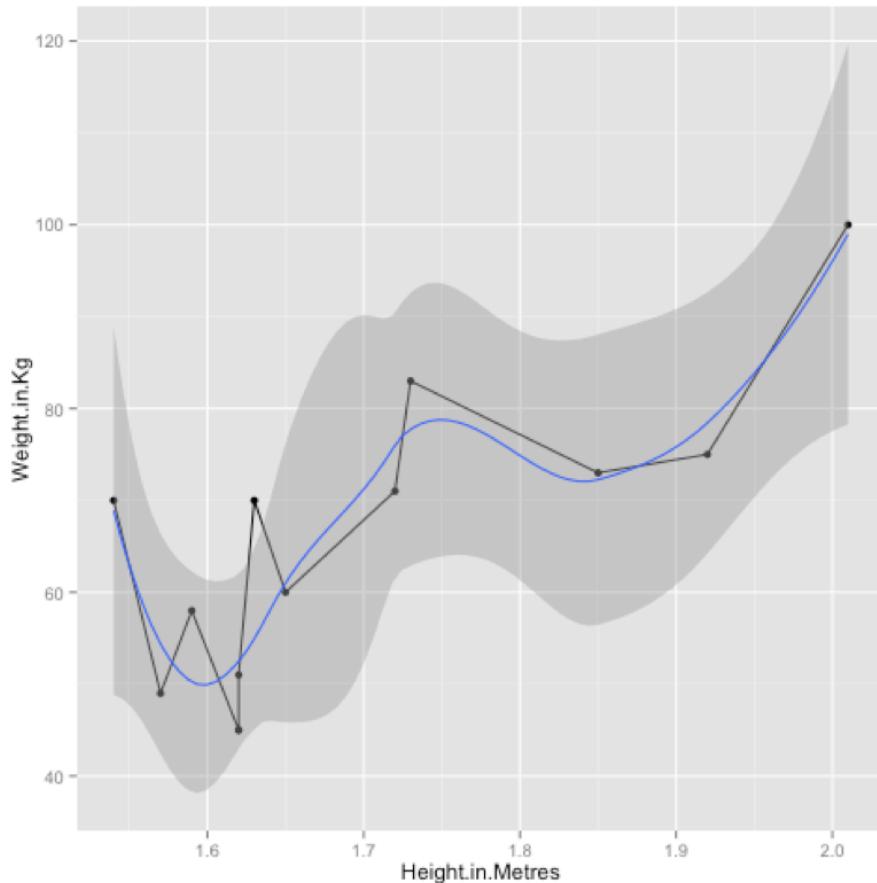
```
qplot(data=JJJ,x=Height.in.Metres,y=Weight.in.Kg,geom="line")
```

We can combine various geom options so as to not just include a line but also the points:

```
qplot(data=JJJ,x=Height.in.Metres,y=Weight.in.Kg,geom=c("point","line"))
```

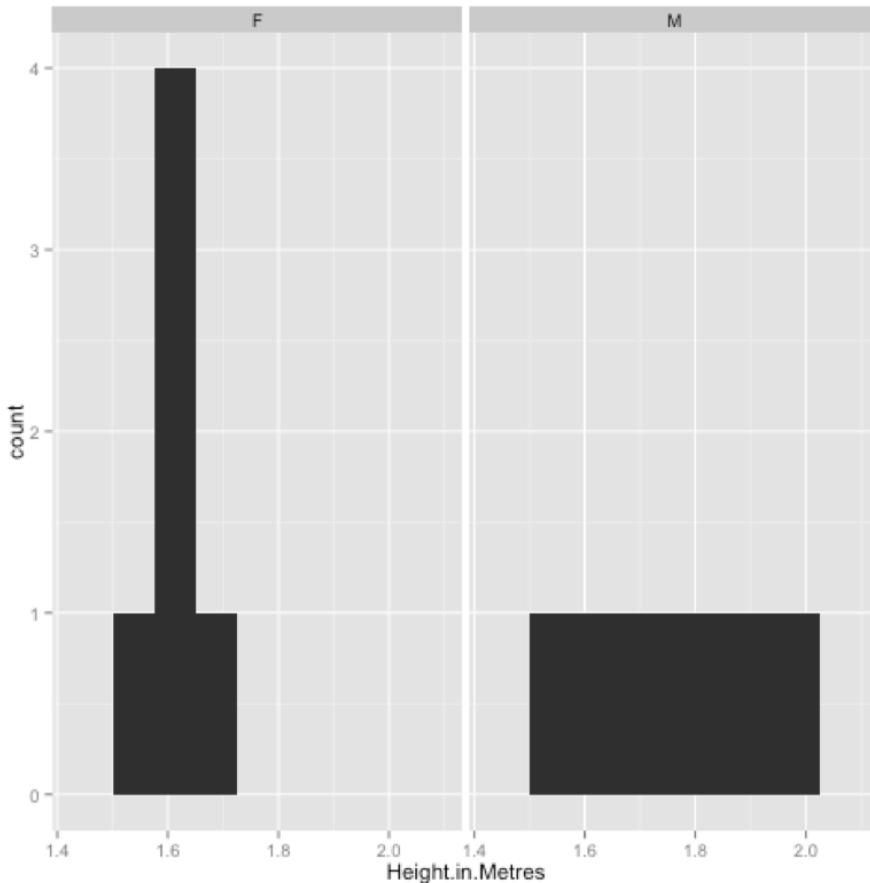
Finally we can also add a smoothed line to our plot as shown:

```
qplot(data=JJJ,x=Height.in.Metres,y=Weight.in.Kg,geom=c("point","line","smooth"))
```



We can very easily obtain a collection of any of the above plots across a categorical variables using the “facets” command as shown:

```
qplot(data=JJJ,x=Height.in.Metres,binwidth=.075,facets=~Sex)
```



We can use the “ggsave” command to save the last plotted graph to file:

```
ggsave(filename="~/Desktop/test.pdf")
```

One final aspect we will take a look at in ggplot2 is that of layers. To do so we will use the following dataset:

```
MMMJJJ_to_plot<-within(MMMJJJ,{data_set<-ifelse(substr(MMMJJJ>Name,1,1)=="M","MMM","JJJ");S...}
```

Firstly we create a plot using qplot and assign it to p (recall that everything in R is an object).

```
p<-qplot(data=MMMJJJ_to_plot,x=Height.in.Metres,y=Weight.in.Kg,facets=~data_set~Sex,color=Sex)
```

To view the plot we simply type “p” (note that we have also included a “color” option):

p

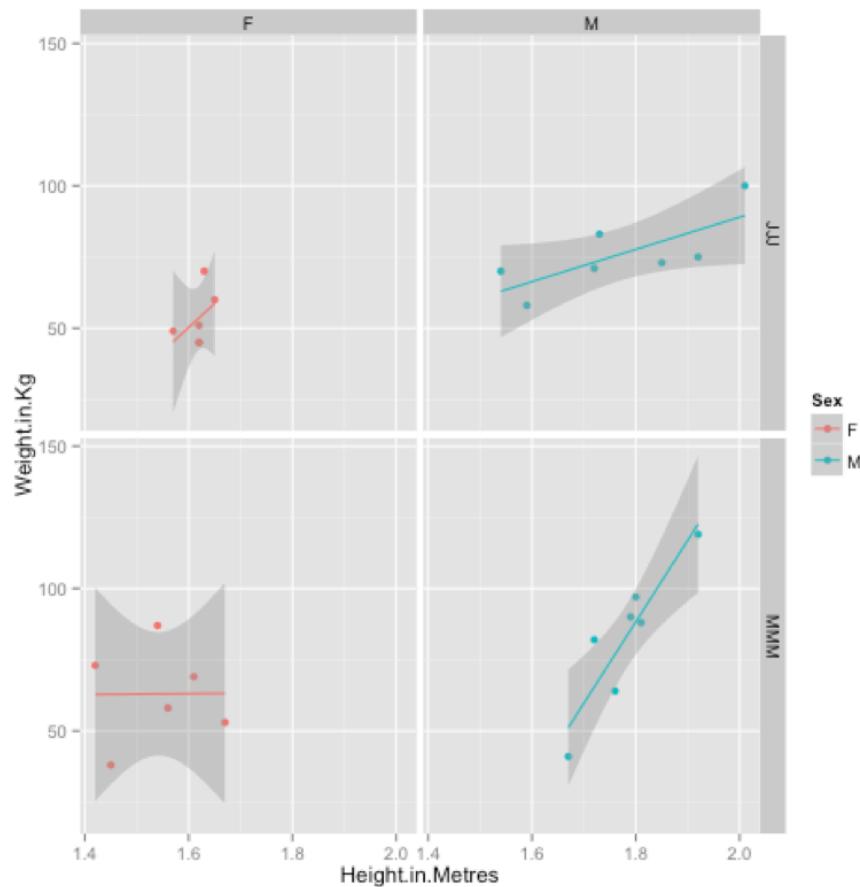
Finally we can add a new layer to this plot by “adding” (+) a linear model to our graph:

```
p<-p+stat_smooth(method="lm")
```

The output of all this is shown.

Finally we can save a particular graph object in ggplot using ggsave:

```
ggsave(p,filename=~"/Desktop/plot.pdf")
```



5.3 twitteR

The last package we will consider is a package that can be used to data mine twitter.

To get the certain trends we use can use the following code:

```
getTrends(period = "daily", date=Sys.Date())
getTrends(period = "daily", date=Sys.Date() - 1)
getTrends(period = "weekly")
```

To obtain tweets for a particular hashtag:

```
searchTwitter("#orms")
```

Finally to obtain tweets from a particular user (the following gives the tweets of the INFORMS as shown):

```
userTimeline("INFORMS")
```



The image shows a Twitter profile page for the account @INFORMS. The header features the INFORMS logo (a white square with 'informs' in black) and the text 'INFORMS' in large, bold, white capital letters. Below the name, it says '@INFORMS FOLLOW YOU'. A bio follows: 'The Institute for Operations Research and the Management Sciences is the largest professional society in the world for professionals in the field of O.R.' The location is listed as 'Hanover, MD' and the website as 'http://www.informs.org'. Below the bio, there's a summary bar with the following statistics: 4,912 TWEETS, 308 FOLLOWING, and 1,993 FOLLOWERS. To the right of these numbers is a button labeled 'Following' with a user icon. The main content area is titled 'Tweets' and contains two recent posts from @INFORMS. The first tweet, dated 15 Dec, is a retweet from @ThaddeusKTSim: 'RT @Slate: Could machine-learning algorithms debunk Twitter rumors before spread? ow.ly/1Qf9ky #orms #analytics'. It includes a 'View summary' link. The second tweet, also dated 15 Dec, is a retweet from @DCWoods: 'RT @DCWoods: I made an LP/MILP solver for iPhone. I blogged about it here: ow.ly/1Qf9hN'. It includes an 'Expand' link.

4,912 TWEETS 308 FOLLOWING 1,993 FOLLOWERS

Tweets

INFORMS @INFORMS 15 Dec
RT @ThaddeusKTSim: RT @Slate: Could machine-learning algorithms debunk Twitter rumors before spread? ow.ly/1Qf9ky #orms #analytics
[View summary](#)

INFORMS @INFORMS 15 Dec
RT @DCWoods: I made an LP/MILP solver for iPhone. I blogged about it here: ow.ly/1Qf9hN
[Expand](#)

```
Dropbox — R — 80x24

> q<-userTimeline("INFORMS")
> q
[[1]]
[1] "INFORMS: RT @ThaddeusKTSim: RT @Slate: Could machine-learning algorithms debunk Twitter rumors before spread? http://t.co/fPwgX06A #orms #analytics"

[[2]]
[1] "INFORMS: RT @DCWoods: I made an LP/MILP solver for iPhone. I blogged about it here: http://t.co/XLcPxEtf"

[[3]]
[1] "INFORMS: RT @or_exchange: New on OR-X: [ANN] Introducing Minimax: an LP/MILP solver for iPhone http://t.co/8I0J0sBA"

[[4]]
[1] "INFORMS: Thanks! RT @nicolasclbr: NCB Operations Research News is out! http://t.co/GU3gs0Cz ▶ Top stories today via @CompSciFact @INFORMS"

[[5]]
[1] "INFORMS: @jfpuget You're welcome!"

[[6]]
[1] "INFORMS: Our thoughts & prayers are heading to those in Sandy Hook."
```