

Getting Started with R and RStudio

Rhondda Jones & Robin Gilliver

© 2017

Copyright vests in the authors

About this content

This content has been provided to be used in the Statistical Methods for Data Scientists subject. Apart from being reformatted to an online delivery and some minor modifications, the content has been created by the authors Rhondda Jones and Robin Gilliver and they retain their copyright. However, the authors are not responsible for the delivery of the content in the Statistical Methods for Data Science subject.

The authors can be contacted about this material via the email: jones_gilliver@bigpond.com.

Preface

The first version of R was written largely for teaching purposes by two New Zealand statisticians, Ross Ihaka and Robert Gentleman, in 1995: it evolved from the S language developed by John Chambers at the AT&T Bell laboratories. It was an instant hit! By now, R has become the fastest growing analytical software available, in terms of both its usage and its capacity. The ability to use it effectively is now an essential and marketable skill for most quantitatively-oriented professions, from bioinformatics to data science. Because it is free and open-source, R has also become the environment where new analytical methods first appear. It has a huge and growing online user community, and numerous websites devoted to assisting and coaching new users. Welcome to the R community! We hope you enjoy the journey.

1 Installation and startup

R is a specialist computer language designed mainly for data analysis – although, like any other computer language, it can do many other things too. It is best used within an integrated development environment, or IDE, which makes writing **R** code much easier. Here we introduce you both to **R** itself and to **RStudio**, the best-known IDE for **R**. This manual is intended to be used actively – that is, you should try each of the procedures and commands described here for yourself, making sure that you can generate the right outputs. For each coding section, it is also good practice to invent and test a few similar additional commands to make sure that you really do understand the syntax which is described. In our experience, it takes about 12 hours of active use of R and RStudio to become reasonably fluent with them.

1.1 Installing R and RStudio

R is free and can be downloaded from the CRAN website:

<https://cran.r-project.org/>

This page provides links to let you download installers for the latest R versions for Windows, Mac, and Linux machines.

RStudio can be downloaded from:

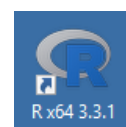
<https://www.rstudio.com/products/rstudio/download3/>

This website has links to download either personal (free) or commercial versions of RStudio. Most users will use the desktop personal license. As with **R**, you can choose an installer appropriate to your computer.

You should install **R** before installing RStudio. In each case, double-click on the downloaded installer and follow the installation instructions provided. At the end of this process, you should have icons for both **R** and RStudio on your desktop.

1.2 Starting R and RStudio

Once you have **R** installed, its desktop icon looks as shown on the right.



For RStudio, the desktop icon will be:



You can double-click on the icon in the usual way to start either program. Starting RStudio will also start **R** automatically – but initially, start **R** on its own to see how to use it without RStudio.

1.2.1 Using R with the RGui

If you open **R** rather than RStudio, the program window (RGui) will look as shown below.

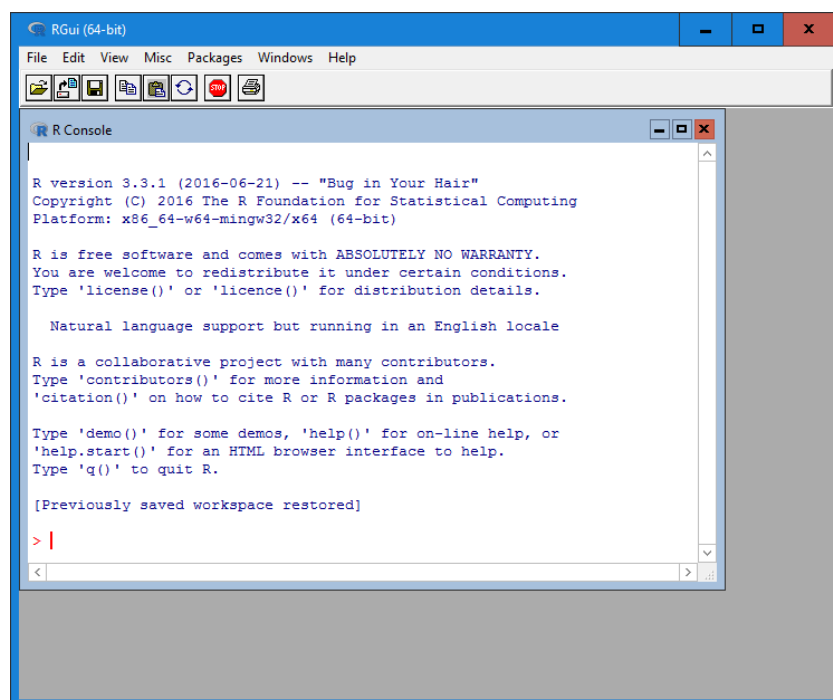


Figure 1-1: The RGui window containing the R console window.



Open a script

Load a workspace

Save a workspace

Copy

Paste

Copy and paste

Stop current computation

Print

The toolbar at the top includes a number of useful functions whose purpose is listed to the left.

We will not dwell on them here because their functions are superseded by the capabilities of RStudio.

The inner Console window executes **R** commands that you type in manually. When you first open **R**, it will provide various items of information, and then display a prompt (>) to type in a command.

After entering a command, you then press the **Enter** key on your keyboard to request that the command be executed. When **R** finishes executing a command, it will display a new prompt. If you press the

Enter key after typing an incomplete command, **R** will display a plus sign (+) to indicate that it expects additional input for the command. Pressing the **Esc** key on your keyboard will cancel the current command and give you a > ready for new input.

To see how this works, enter a few simple commands. At the prompt (>) type in any arithmetic expression, for example:

```
> (3 + 9) * 5
```

Press the Enter key. R responds with the answer – in this case

Arithmetic operators :

+	add
-	subtract
*	multiply
/	divide
^	take to power
%%	modulus (remainder after integer division)
%/%	integer divide

```
[1] 60
```

Notice that the command appears in red and the response in blue. Depending on the command, there can be multiple values in the response: the number at the beginning of the output line is the index number for the first value in the line.

Now enter the same calculation again, but with an error – omit the closing bracket after the 9.

```
> (3 + 9 * 5
```

When you press the **Enter** key, R responds with

```
+
```

This time **R** notices the absence of a closing bracket and indicates that it expects further input. We can give it a closing bracket and press Enter

```
+ )
```

```
[1] 48
```

but the answer is wrong because the final bracket should not be at the end. Commands can be retrieved using the up and down arrows on your keyboard. In this case, press the up arrow twice to retrieve the command with the mistake, and then correct it. Then try the following, and more of your own ...

```
> 12/8
```

```
[1] 1.5
```

```
> 12%%8
```

```
[1] 4
```

```
> 12%/%8
```

```
[1] 1
```

```
> 12^2
```

```
[1] 144
```

```
> (50 + 6) / 2^3
```

```
[1] 7
```

Order of operations

Normal algebraic rules apply—
i.e. according to the precedence
list below and left to right for
operators with similar
precedence

1. Inside parentheses () before outside
2. Take to power ^
3. Multiply and divide
* / %% %/%
4. Add and subtract + -

You can type more than one command on a line if you separate the commands with a semicolon.

```
> 12*3 ; 12-4
```

```
[1] 36
```

```
[1] 8
```

You can also evaluate logical expressions

For example

```
10^2 == 99
```

```
[1] FALSE
```

This expression is read as “Ten squared is equal to 99” and **R** correctly tells you that this statement is false. Logical expressions can be made quite complex using “and” (&& for single values) and “or” (|| for single values). So the first expression below is read as “10 is equal to 8 OR 5 squared is equal to 25” and evaluates as true.

```
(10==8) || (5^2==25)
```

```
[1] TRUE
```

```
(10==8) && (5^2==25)
```

```
[1] FALSE
```

Logical (comparison) operators:

==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
	or (for vectors)
&	and (for vectors)
	or (for single values only)
&&	and (for single values only)

Be careful not to use '=' (assignment)
where you should use '==' (the logical
operator "is equal to").

This is one of the most common
command errors, even for
experienced users.

Try more examples of your own.

To save the result of any expression, assign the expression to a name, for example, enter:

```
> x = (50 + 6) / 2^3  
>
```

This time, instead of displaying the result, R has stored it in a variable called x. To display it, use:

```
> x  
[1] 7
```

R accepts three assignment operators:

<- (a ‘less-than’ symbol followed immediately by a hyphen)
-> (a hyphen followed immediately by a ‘greater-than’ symbol)
= an equals sign.

Thus the three commands

```
Lizzie <- "My cat"  
"My cat" -> Lizzie  
Lizzie = "My cat"
```

are all equivalent—they all say “Assign the character string "My cat" to the name Lizzie”. In this manual we will use the last of these options. The assignment command creates an **R** object called Lizzie which has the value "My cat" and stores it in your workspace. Assignment to a previously used name will destroy the old object and create a new one.

An assignment command will not automatically display the value of the new object. To display the value, either enter the name separately, or place the assignment in brackets (which will assign and display in the one operation ...).

```
height = 172  
height  
[1] 172
```

```
(height = 172)  
[1] 172
```

To clear the Console window, right click on it and select **Clear All**.

At this point we will move on to using RStudio as an environment for **R**, so close the RGui window.

R names

Names are case-sensitive, can be any length, and can include letters, numbers, and the period symbol. They must start with a letter and must not contain spaces.

So

joannel

Joannel

Jo.annel

are all valid (and different)

R names.

1.2.2 Using R with RStudio



Start **RStudio** by double-clicking its startup icon

When you start **RStudio** for the first time, it should resemble the screenshot below, with a large window to the left, and two smaller windows to the right.

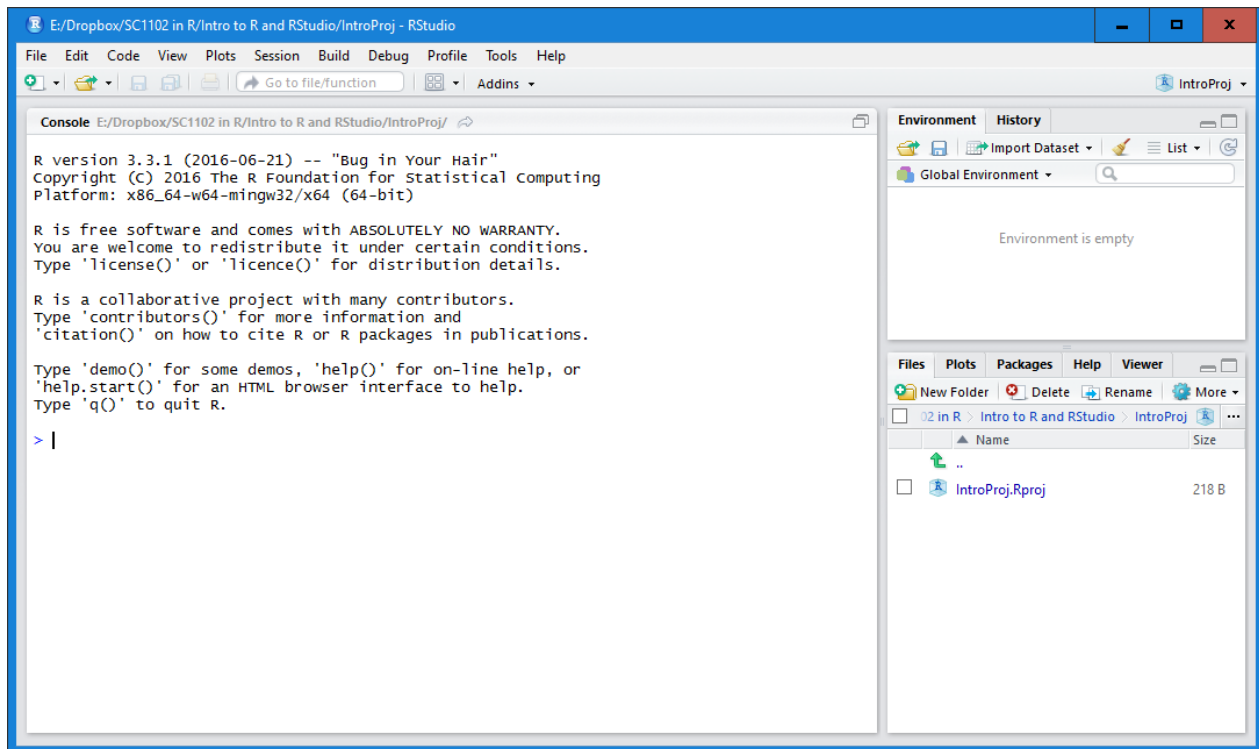


Figure 1-2: The RStudio windows before creating a script file.

The left-hand window is the **R** console for whichever version of **R** you have previously installed. Notice that it is helpfully labelled **Console** at the top, and displays the directory where your **R** workspace is located. The console window displays commands and the results of their execution, just as the ordinary **R** console does.

The layout described here is for **RStudio** version 1.0.136. Earlier versions label some windows differently .

The top right-hand window has two tabs – **Environment** and **History**. The **Environment** tab gives you the contents of your current **R** workspace – which will be empty at this point. The **History** tab keeps track of all the commands you have used – and again, is empty when you first open **RStudio**.

The bottom right-hand window has five tabs. The first one, **Files**, gives you the files within the folder where your workspace is located. The second, **Plots**, will display plots you create in your **R** session. The third, **Packages**, gives you a list of the packages you currently have access to, and the fourth, **Help**, lets you search the available help files. The fifth (**Viewer**) can display local web content.

You will find the right-hand windows very helpful. But the most useful part of **RStudio** is not visible yet: to obtain it, we need a new window which appears when you create a script, and gives you access to the source editor of **RStudio**.

1.3 Creating, editing and running scripts

The key advantage of using an IDE like **RStudio** is the ability to edit and test both individual commands and sets of commands within a specialized script editor.

To create a script file to hold your commands, press

File > New File > R Script

A new window is created above the Console window, so that **RStudio** now looks as shown below:

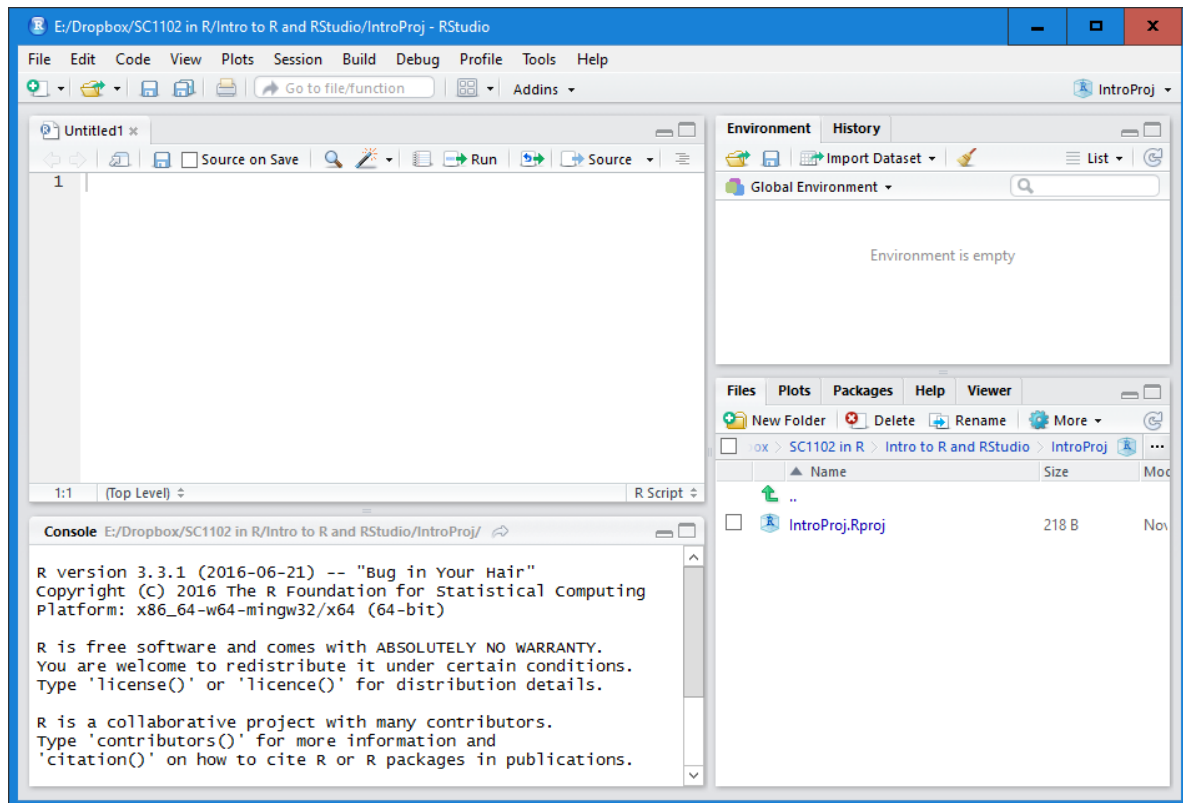


Figure 1-3: The RStudio windows including a script window.

Notice that the script window tab is labelled “Untitled1”, This is currently the name of the script. To change the name to something more descriptive and save it, press

File > Save as...

enter **Introduction** for the **File name:** and press **Save** . Notice that it is saved as **Introduction.R** – RStudio expects any file containing an **R** script to end in **.R**.

To illustrate how the script window works, we will enter, edit, and execute a few simple commands. In the script window, type

```
x = (50 + 6) / 2^3
```

As you type the line, notice that **RStudio** helpfully matches brackets as you go. It will do the same for quote marks.

With the cursor still on the line, press the **Run** button at the top of the script window.



Notice that the command is echoed in the Console window and has been successfully executed.

Also, in the **Environment** window on the top right, `x` is now listed under **Values**, together with its value (7).

Now select `2^3` within the typed line, and press **Run** again. The selected expression is again echoed, and this time (because the `x =` part of the command was not selected), the expression is evaluated and the result is shown in the Console window. Both commands are echoed under the **History** tab.

The general rule is that when you press **Run**, if nothing is selected, RStudio runs the line in the script where the cursor is located. If the line is an incomplete command, it will keep reading lines until the command is completed. If something is selected – part of a line, or several lines – it runs whatever is selected.

Now close RStudio by clicking the red Close button on the top right. RStudio will ask you if you want to save the data and script you have created in the session.

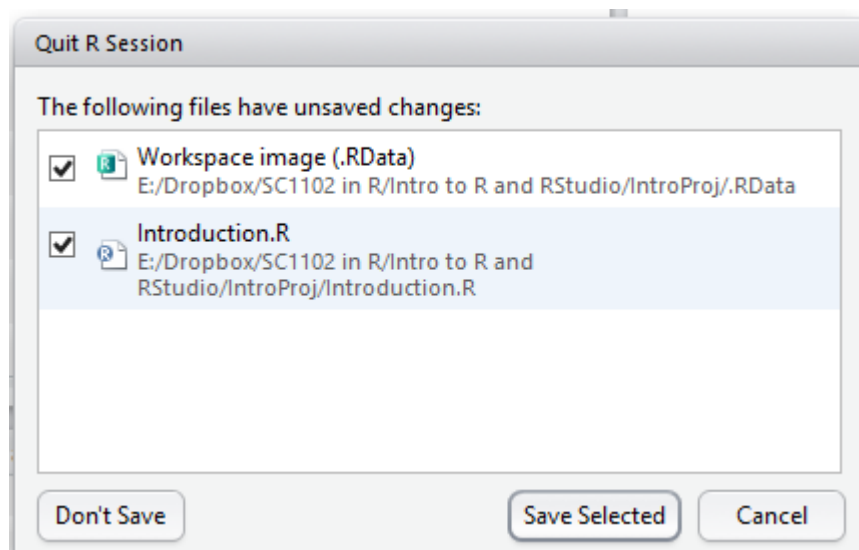


Figure 1-4 Quit dialogue for RStudio

If you press the **Save Selected** button, both the script and any saved data will appear the next time you open RStudio – press it and reopen RStudio to check.

2 Introduction to R functions

A **function** is a set of R operations which are given a name and executed as a group for a particular purpose. Most statistical operations can be executed with a single command which calls a ready-made function.

Functions have the form :

somename ()

usually with one or more values inside the brackets. The quantities inside the brackets are called the **arguments** of the function— they are the data on which the function operates. For example, the function to calculate square roots is

sqrt(x)

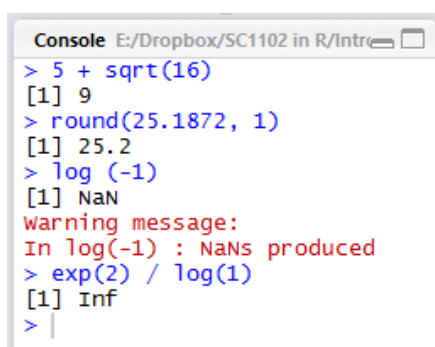
where **x** is either a number or the name of a variable which contains one or more numbers.

```
sqrt(25)
[1] 5
```

Any arithmetic expression you type in for R to evaluate can include any of the R functions with appropriate arguments in parentheses (). If the expression cannot be evaluated, R will respond with a missing value (**NaN** - for "not a number") and a warning message. Division by zero gives **Inf** (for "infinity").

Type the expressions below on the left into the RStudio script window and run each one. The Console window should then appear as shown below on the right.

```
5 + sqrt(16)
round(25.1872, 1)
log (-1)
exp(2) / log(1)
```



```
Console E:/Dropbox/SC1102 in R/Intro
> 5 + sqrt(16)
[1] 9
> round(25.1872, 1)
[1] 25.2
> log (-1)
[1] NaN
warning message:
In log(-1) : NaNs produced
> exp(2) / log(1)
[1] Inf
> |
```

For the rest of this manual, to demonstrate a command we will generally provide the command as entered in the script window in black, followed by the output produced on the console in blue – eg

```
5 + sqrt(16)
[1] 9
```

Where a command is too long to fit on one line of this manual, we will indent the second and subsequent lines slightly.

Each of the arguments used by a function has a name, and for some of them, R will provide a default value. For example, the full definition of the **round()** function - found by running the

Parentheses must always be included when you use a function in a command

A function may have no arguments if:

- it has a value which is not affected by user input— e.g. **date()**, which gives the current date and time, or
- there is a default argument which the function can use if an argument is not supplied - e.g. **detach()**

But even if there are no arguments, the parentheses have to be there, because their existence tells **R** that the name belongs to a function.

More about functions—including how to write your own—can be found in later sections.

command **?round** is

```
round(x, digits = 0)
```

x has no default value, because that is data that you must provide - the number(s) to be rounded. But **digits** has a default value of 0: this means that if you do not specify a value, **R** will round **x** to the nearest whole number, with no digits after the decimal place. If you provide a value - as we did in the earlier example, it replaces the default.

You can specify the argument name within the command: if you provide argument names, arguments can be in any order. If you do not provide the name, then arguments must be in the same order as in the function definition.

So in the **round()** example we could have entered:

```
round(digits=1, x=25.1872)  
[1] 25.2
```

There is an enormous number of R functions – to create and modify data, or to analyze it, search for it, model it, or plot it in a variety of ways. If you can't find a function to do exactly what you want, it is possible to write your own. Most people have a set of functions that they use frequently and remember, and use Google to search for those they don't use very often.

2.1 Functions for creating and labelling vectors

So far we have applied functions to single values, as in the **round()** example above. But most R functions are designed to operate on groups of values, which R calls **vectors**. A vector is a set of values of the same data type (eg all numeric, or all character) which has a single name for the whole set. This means that you can do calculations which apply to all elements of the vector with a single command.

2.1.1 *Creating vectors with the **c()** function.*

This simply combines a set of values into a vector, and is the most common way to create vectors manually.

```
myData = c(25, 30, 12, 15, 8, 9, 32)  
myData  
[1] 25 30 12 15 8 9 32  
weights = c(125, 147)  
weights  
[1] 125 147
```

You can also use it to add values to an existing vector

```
myData = c(myData, 16, 22)  
myData  
[1] 25 30 12 15 8 9 32 16 22
```

The **c()** function can also create a vector of character strings.

```
myNames = c("Mary", "Fred")  
myNames  
[1] "Mary" "Fred"
```

2.1.2 Naming vector elements

Individual elements of a vector can be given names with the **names()** function. So to attach “Mary” and “Fred” to the values in the **weights** vector we created earlier:

```
names(weights) = myNames
weights
Mary Fred
125 147
```

You can also give names to each value when a vector is created using the **c()** function.

```
roles = c(Jane="boss", Bill="foreman", Tom="analyst")
roles
      Jane      Bill      Tom
"boss" "foreman" "analyst"
```

2.1.3 Accessing, displaying and changing vector elements

Each element of a vector has an index number, starting with 1. Use square brackets and the index number to display, change, or refer to an individual element of the vector.

```
myData
[1] 25 30 12 15 8 9 32 16 22

myData[3]
[1] 12

myData[3] = 16
myData
[1] 25 30 16 15 8 9 32 16 22
```

You can select more than one element using a vector of index numbers inside the square brackets:

```
myData[c(1,3,5,7,9)]
[1] 25 16 8 32 22
```

Another way to select elements is by using a logical expression:

```
myData[myData > 20]
[1] 25 30 32 22
```

2.1.4 Creating vectors with sequential or repeating data: **seq()** and **rep()**

Vectors in which the values are a sequence or a repeating pattern are very important in setting up analyses. Three useful functions for creating these kinds of vectors are the sequence operator ‘:’, and the **seq()** and **rep()** functions. These are illustrated below.

```
my.series = 1:10
my.series
[1] 1 2 3 4 5 6 7 8 9 10

my.second.series = 4:-4
```

```

my.second.series
[1] 4 3 2 1 0 -1 -2 -3 -4
my.third.series = seq(10, -10, by = -2)
my.third.series
[1] 10 8 6 4 2 0 -2 -4 -6 -8 -10
my.fourth.series <- rep(c("A", "B"), times=4)
my.fourth.series
[1] "A" "B" "A" "B" "A" "B" "A" "B"
my.fifth.series <- rep(c("A", "B"), each = 4)
my.fifth.series
[1] "A" "A" "A" "A" "B" "B" "B" "B"

```

2.2 Some basic calculations and plots using vectors

Vectors are the most important data structure which **R** provides, and using them expands the capacity of **R** immensely.

2.2.1 Create a set of four vectors to illustrate vector calculations

To illustrate a range of simple calculations and plots, create the five vectors shown below:

- A vector called **bodyWeights** containing 500 individuals from a hypothetical population of adult males whose weights are Normally distributed with a mean of 80 and a standard deviation of 10.

```
bodyWeights = rnorm(n=500, mean=80, sd=10)
```

- A labelled vector containing the numbers of described species in different vertebrate groups

```
vertebrateSpecies = c(Fish=32900, Amphibia=7302,
  Reptiles=10038, Birds=10425, Mammals=5513)
```

- A vector called **Year** which holds a sequence of 10 years from 2006 to 2015

```
Year = 2006:2015
```

- A vector called **Density** which holds the 10 values: 5, 3, 7, 8, 6, 9, 8, 10, 14, 11, representing hypothetical densities of seedlings in each of the years listed in the **Year** vector.

```
Density = c(5, 3, 7, 8, 6, 9, 8, 10, 14, 11)
```

Notice that these four vectors and some or all of the values they hold are now listed under **Values** in RStudio's **Environment** window.

2.2.2 Missing values

The standard way to indicate that a value is missing is with an NA (for “not available”). For example, suppose that we had a second set of densities for the years 2006 to 2015, but were missing data for the years 2008 and 2010 (that is, for the third and fifth values in the vector).

```
Density2 = c(6, 6, NA, 7, NA, 8, 10, 11, 10, 15)
```

2.2.3 Basic vector calculations, pie charts, and bar plots

Once a vector has been created, functions or arithmetic operations applied to a vector act on all elements of the vector.

For example, to express the numbers of vertebrate species in thousands

```
vertebrateSpecies / 1000
```

Fish	Amphibia	Reptiles	Birds	Mammals
32.900	7.302	10.038	10.425	5.513

To calculate the total number of described vertebrate species

```
sum(vertebrateSpecies)
[1] 66178
```

And then, to estimate the percentage of species in each group, rounded to 1 decimal place:

```
percentageSpecies = round(100 * vertebrateSpecies /
  sum(vertebrateSpecies), 1)
```

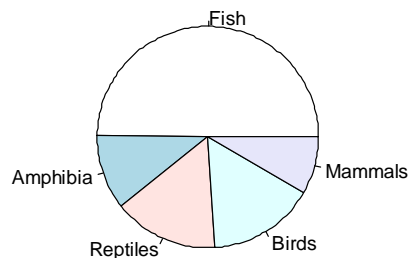
```
percentageSpecies
```

Fish	Amphibia	Reptiles	Birds	Mammals
49.7	11.0	15.2	15.8	8.3

With a labelled vector, it is also straightforward to draw pie plots or bar plots to display the data.

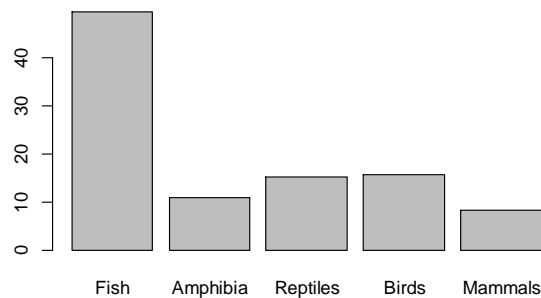
When you run each plot command, the plot should appear in the Plots tab on the bottom right window of RStudio .

```
pie(vertebrateSpecies)
```



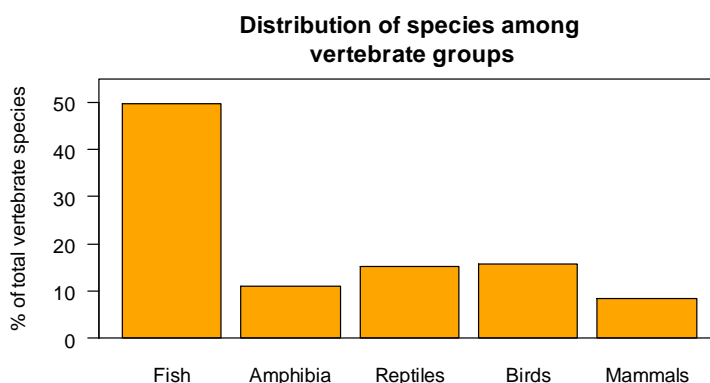
For the bar plot, you will probably need to make the plot window wider to allow all the bar labels to be displayed – this can be done by dragging the corner of the window after the plot has been drawn.

```
barplot(percentageSpecies)
```



This plot could do with a bit of improvement – below we modify the command extensively to produce the final product. You should try adding each new argument one at a time to see what each does. Note the use of `\n` in the **main** argument to force part of the title to be on a new line.

```
barplot(percentageSpecies, las=1, col="orange",
        ylab="% of total vertebrate species",
        main="Distribution of species among\n vertebrate groups",
        ylim=c(0,55))
box()
```



Colors

To have a look at the range of colors you can use, run the command

```
colors()
```

There are several useful buttons above the **RStudio** plot window. Try them out. The **Zoom** button gives you the plot in a separate window whose shape and size it is easier to change. The **Export** button allows you to either save the plot or copy it to the Clipboard to transfer to another document – eg a Word document or a PowerPoint slide. If you have drawn several plots, the arrows allow you to move between them. The remaining two buttons delete either the current plot or all the plots you have drawn.

2.2.4 Calculating descriptive statistics and plotting distributions

To demonstrate the calculation of descriptive statistics we use the **bodyWeights** vector, where the mean and standard deviation of the population from which the sample was drawn are specified as 80 and 10 respectively. Because you will have created a different random sample, the results you get when you run these commands will be slightly different from those shown here.

```
mean(bodyWeights) # calculates the mean
```

Comments

The symbol `#` signals a comment to R. Everything between the `#` and the end of a line is ignored, e.g. in the command

```
mean(myData) # the mean
               what is executed is just
mean(myData)
```

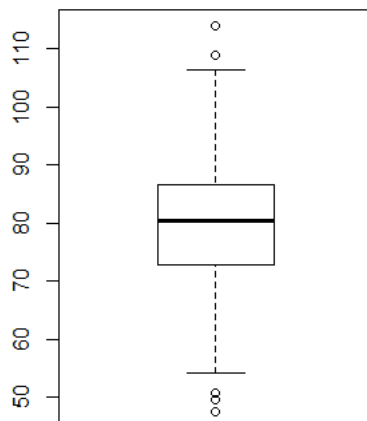
```

[1] 79.8727
median(bodyWeights)
[1] 80.28216
sd(bodyWeights) # the standard deviation
[1] 10.42242
range(bodyWeights)
[1] 47.45433 114.05530
var(bodyWeights) # the variance
[1] 108.6269
length(bodyWeights) # the number of elements
[1] 500
summary(bodyWeights) # gives min, max, median, mean, quartiles
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
47.45  72.83   80.28   79.87  86.66  114.10

```

There are two standard ways to plot a distribution of numbers like the **bodyWeights** data: a boxplot and a histogram.

boxplot(bodyWeights)



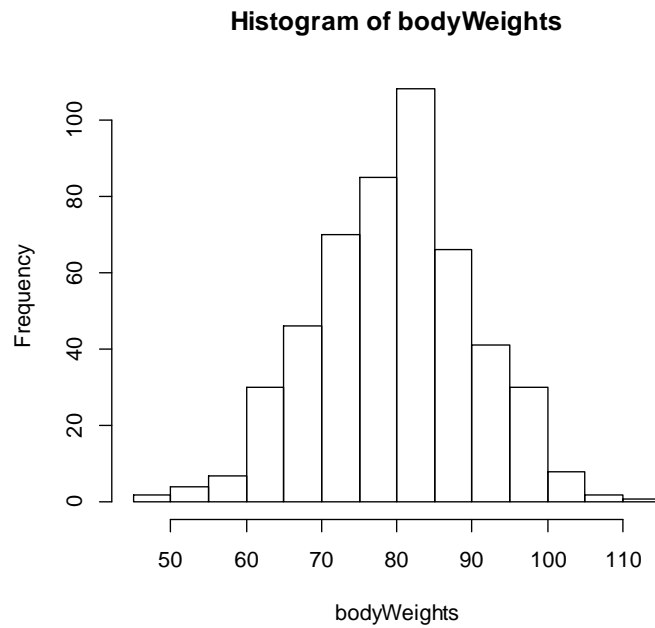
Using the default arguments for the boxplot function:

- The heavy horizontal line represents the median value
- The box includes the inter-quartile range (that is, it includes 50% of the data).
- The whiskers extend a further 1.5 times the interquartile range
- Any remaining values are shown as individual points (to identify them as possible outliers)

To examine arguments which modify these conventions (e.g. the **range** argument which defines the extent of the whiskers) use the command

?boxplot

hist(bodyWeights)

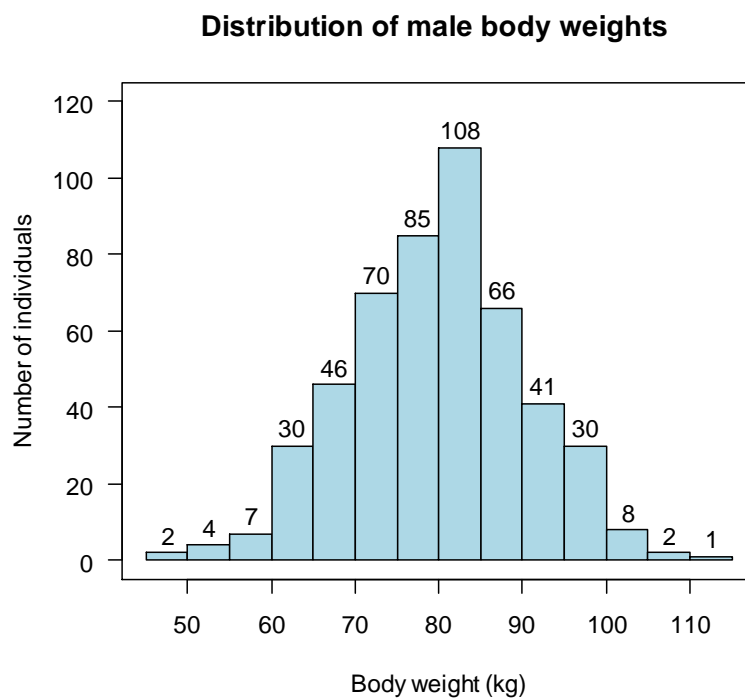


Now edit the `hist(bodyWeights)` command in the script window to change some of the attributes of the plot, and run the edited command.

```
hist(bodyWeights, las=1, col="lightblue", xlab="Body weight (kg)",
      ylab="Number of individuals",
      main="Distribution of male body weights",
      labels=T, ylim=c(0,120))
```

Then put a box around the plot to obtain the final result below.

```
box()
```



2.2.5 Descriptive statistics when there are missing values

When a vector contains missing values, commands to calculate summary statistics like means and standard deviations will generally need to be modified. For example, to calculate the mean of the **Density2** vector, which has two missing values, we might use:

```
mean(Density2)
[1] NA
```

R returns a missing value – because two values are missing, the mean of all 10 values in **Density2** cannot be calculated. If you want the mean of the remaining values, the command needs to specify that missing values should be removed before the calculation is made.

```
mean(Density2, na.rm=TRUE)
[1] 9.125
```

The function **is.na()** returns TRUE for values which are missing and FALSE otherwise:

```
is.na(Density2)
[1] FALSE FALSE TRUE FALSE TRUE FALSE FALSE
FALSE FALSE FALSE
```

To find the indices of missing values:

```
which(is.na(Density2))
[1] 3 5
```

To find the indices of values which are NOT missing:

```
which(!is.na(Density2))
[1] 1 2 4 6 7 8 9 10
```

Key arguments in plot()

The initial part of this plot command:

Density~Year

is a **formula** which can be read as “Density (on the y-axis) is a function of Year (on the x-axis)”.

The **type** argument specifies what kind of plot – “b” means “both points and lines”. “l” would plot only lines, and “p” would plot only points. There are other options too = see **?plot**

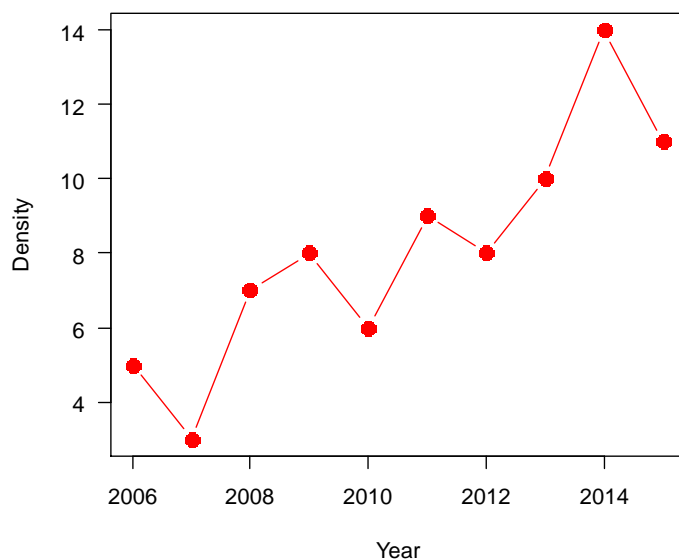
pch specifies the plotting character – in this case a solid circle – there are a lot of options, see **?points**

cex changes the size of the plotting character – the default is 1, so >1 expands it and <1 contracts it.

2.2.6 Calculations and plots with two numeric vectors

Here we use the two associated vectors **Year** and **Density**, which each has 10 observations. Initially draw a line plot to show how the density changes with year.

```
plot(Density~Year, type="b", pch=16,
     col="red", cex=1.5, las=1)
```



Key arguments in `abline()`

`lwd` changes the thickness of the line.

`lty` specifies what sort of line (in this case a dashed line)

Key arguments in `text()`

`x` and `y` specify the coordinates of the middle of the string relative to the x & y axes.

`labels` specifies what is to be written

The plot suggests that density increases with year – that is, there is a positive correlation between **Year** and **Density**. To calculate the correlation coefficient (a measure of the strength of the correlation) use

```
cor(Year, Density)
[1] 0.8699182
```

Since the maximum possible value of the correlation coefficient is +1, the correlation appears quite strong.

To calculate the intercept and slope of the straight line which best describes the relationship between **Density** and **Year**:

```
lm(Density~Year)

Call:
lm(formula = Density ~ Year)

Coefficients:
(Intercept)      Year
-1807.442      0.903
```

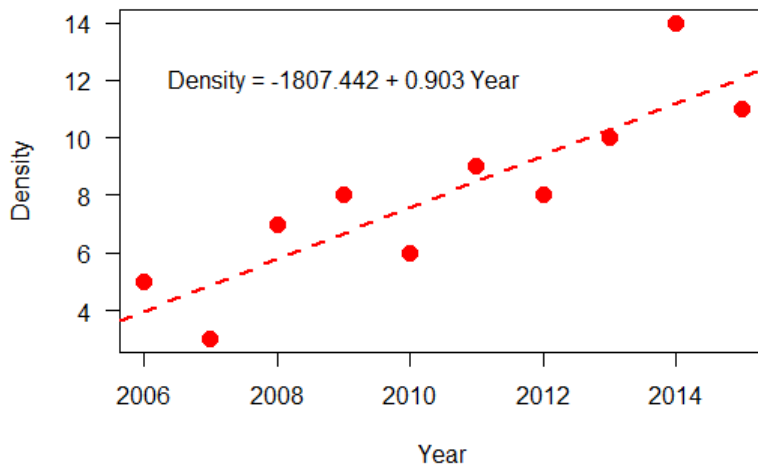
To draw the regression line superimposed on an existing scatterplot (ie a plot which uses points instead of both points and lines):

```
plot(Density~Year, type="p", pch=16,
     col="red", cex=1.5, las=1)
abline(-1807.442, 0.903, lwd=2, lty=2,
     col="red")
```

We can then create a character string to write the equation, and use the `text()` command to superimpose that on the plot too, giving the final result below.

```
text(x=2008, y=12,
```

```
labels="Density = -1807.442 + 0.903 Year")
```



Exercises 2

1. Calculate the mean, standard deviation, median, range, and number of elements for both **Density** and **Density2**.
2. Draw a bar plot with red bars to show the values of **Density2** for each year. (Hint: you can use the **names** argument within the **barplot()** command to label the bars.)
3. Draw a scatterplot of **Density2** against **Density**, and draw the straight line which best describes the relationship on the plot. Include the equation which describes the line.
4. Calculate the correlation coefficient for the relationship between **Density** and **Density2**. Hint: check the **use** argument on the help page for **cor()** (via **?cor**).

3 Data in R

3.1 Data types

Data in **R** come in a variety of different types, some of them (like dates) quite complex. The data types you will probably use most often are:

- Integer
- Numeric (Double)
- Character
- Factor
- Logical

The properties of each are outlined below: – we have used the first three in earlier sections.

Integer: integer data are whole numbers – eg 12, -103, 0 The **Year** vector created earlier was an integer variable.

Numeric (Double): numeric data with a decimal part - eg 12.15, -103.0, 0.007 “Double” is short for “double precision”, which means that each value is accurate up to 16 places – so 0.007 is actually stored with another 13 zeroes after the 7. The **bodyWeights** variable created earlier stored its data as doubles.

Character: character data are character strings, used most often to label or name individual values or data rows. When you surround a value with quotation marks, it is created as character data even if what is in quotes is a number, for example

```
textStrings = c("alpha", "Catch the ball!", "27.6")
textStrings
[1] "alpha" "Catch the ball!" "27.6"
```

Factor: when a variable is a category which is used to group data into subsets, it should be stored as a factor variable. Factor data has two parts – an integer which defines the level of the factor, and a label which is a text string used to describe each level. For example, consider the **bodyWeights** vector holding 500 hypothetical male body weights we created in chapter 2 via the command

```
bodyWeights = rnorm(n=500, mean=80, sd=10)
```

Add 500 hypothetical female body weights with a mean of 65 and a standard deviation of 10 to the original **bodyWeights** vector via the command below:

```
bodyWeights = c(bodyWeights, rnorm(n=500, mean=65, sd=10))
```

Notice that in the **Environment** window, the **bodyWeights** vector now has 1000 values: the first 500 are the original 500 males, and the second 500 are the females. We now create a vector called **gender** which can be used to specify which individuals are male and which are female:

```
gender = rep(c("Male", "Female"), each = 500)
```

Check the vector in the **Environment** window: notice that it has been created as a character variable. It can be converted to a factor with the command:

```
gender = factor(gender)
```

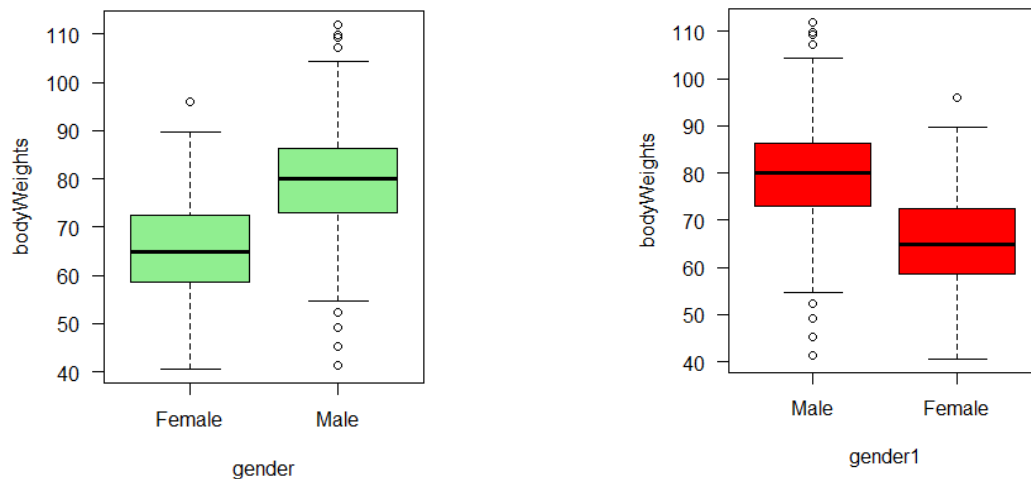
Notice that the **Environment** window now tells us that **gender** is a Factor with 2 levels, and that “Female” is listed before “Male”. This means that **Female** is level 1 for the factor, and **Male** is level 2 (note that the early values of **gender** are all 2, as they should be). In general, unless you specify otherwise, the factor levels will be defined in alphabetical order: in this case “female” is defined as level 1 because F comes before M in the alphabet. The order can be changed: the command below creates a new **gender1** vector with “Male” as the

first level.

```
gender1 = factor(gender, levels =c("Male", "Female"))
```

The order of levels determines how values are plotted and analyzed, as illustrated in the two plots generated below.

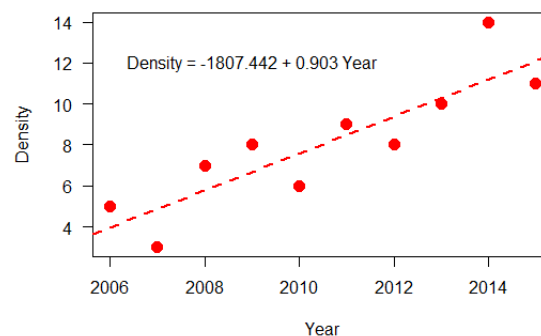
```
plot(bodyWeights ~ gender, col="lightgreen", las=1)
plot(bodyWeights ~ gender1, col="red", las=1)
```



Logical: logical variables can have only two possible values – TRUE and FALSE. Like factor variables, there is also a number associated with them (FALSE is 0 and TRUE is 1). Logical variables are often created by comparisons of some kind.

For example, consider the relationship between **Density** and **Year** which we evaluated at the end of Chapter 2, producing the regression equation and plot shown again below.

```
Year = 2006:2015
Density = c(5, 3, 7, 8, 6, 9, 8, 10, 14, 11)
lm(Density~Year)
plot(Density~Year, type="p", pch=16, col="red", cex=1.5, las=1)
abline(-1807.442, 0.903, lwd=2, lty=2, col="red")
text(x=2008, y=12,
      labels="Density = -1807.442 + 0.903 Year")
```



We can use the equation to obtain the fitted values for each year:

```
fitted = -1807.442 + 0.903*Year
```

The fitted points should of course sit exactly on the regression line, as we can demonstrate by plotting them on the same graph:

```
points(fitted ~ Year, cex=1.5, col="black")
```

So which years had densities higher than expected (ie above the fitted line)? We can identify them by creating a logical variable which compares **Density** with **fitted**:

```
isHigh = Density > fitted
isHigh
[1] TRUE FALSE TRUE TRUE FALSE TRUE FALSE FALSE TRUE FALSE
```

If **isHigh = TRUE**, the relevant year had a density above the line. We can show those years using the **which()** function to find the indices of the **TRUE** values:

```
which(isHigh)
[1] 1 3 4 6 9
Year[which(isHigh)]
[1] 2006 2008 2009 2011 2014
```

3.2 Data structures: data frames

A data structure is something which can hold a set of data – so vectors are one type of data structure. However, an experiment or survey normally involves several variables which are associated in some way, so we want to keep them together in something like a spreadsheet – that is, a two-dimensional structure where each variable is a column in the spreadsheet, and each “case” is a row. The **R** data structure to do this is called a *data frame*.

3.2.1 Data frames

A data frame combines a set of vectors which are all the same length. Its vectors can be of different data types, so a single data frame might include vectors of categorical data (the factor data type), of logical data, and of numerical data. While each vector can only contain data of one type, the data frame as a whole can include data of several different types.

Usually, you will start with the data in a spreadsheet of some kind, and then read the whole data set into **R** – we will demonstrate how to do that shortly.

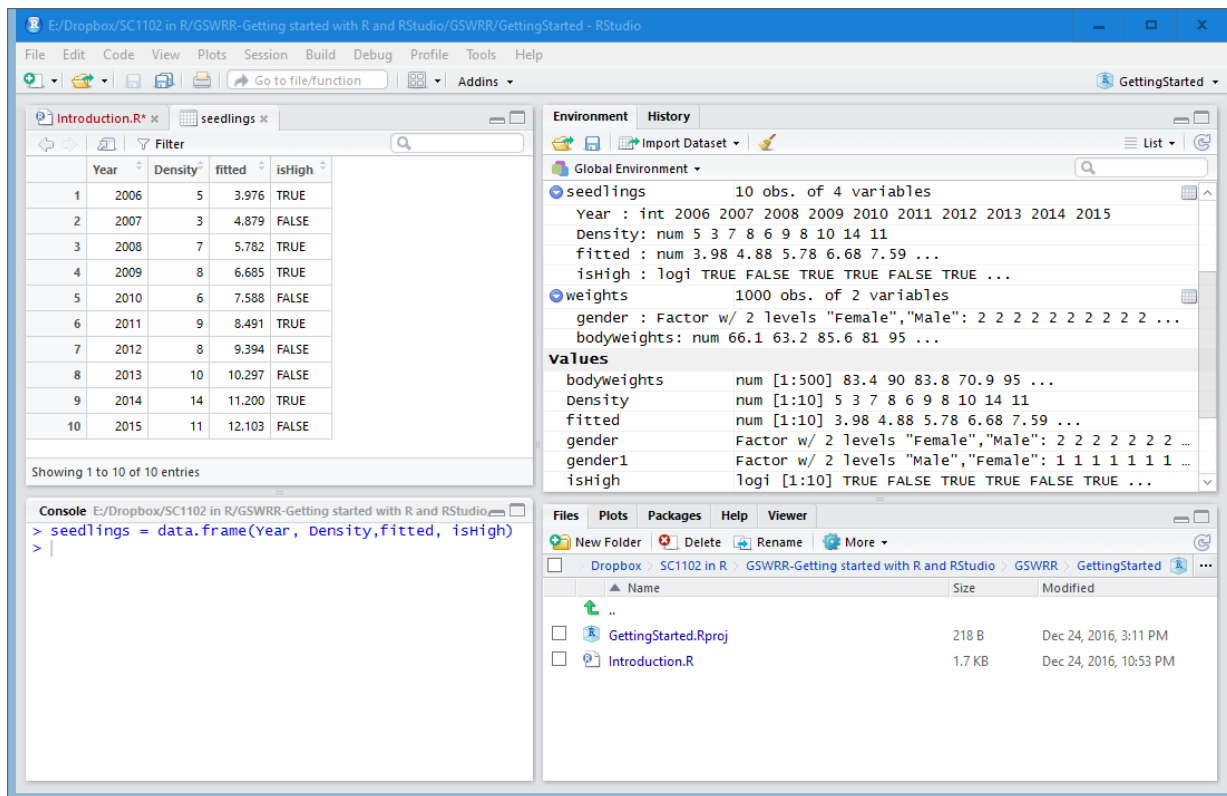
However, data frames can also be created manually by creating the individual vectors and then copying them into a data frame with the **data.frame()** function. So, for example, we might want to copy the **Year**, **Density**, **fitted** and **isHigh** vectors created in Section 3.1 into a data frame called **seedlings**:

```
seedlings = data.frame(Year, Density, fitted, isHigh)
```

Similarly we can combine the **bodyWeights** and **gender** vectors into a data frame called **weights**:

```
weights = data.frame(gender, bodyWeights)
```

In the **RStudio Environment** window, notice that there are still “loose” **Year**, **Density**, **bodyWeights** and **gender** vectors under the **Values** heading, but there is now also a **Data** heading which lists the **seedlings** and **weights** data frames: click the little blue arrows to the left of **seedlings** and **weights** to display their vectors. Then click the grid symbol to the right of **seedlings** to show the whole data frame in a new tab to the left, as in the screenshot below.



You can refer to (and change) any value in the data frame by specifying its row index and column index in square brackets using **dataframeName[row index, column index]**. So to display the value in the first row and first column:

```
seedlings[1, 1]
[1] 2006
```

Similarly you can obtain blocks of values by specifying vectors of row and column numbers: for example

```
seedlings[1:3, c(1,3)]
  Year fitted
1 2006  3.976
2 2007  4.879
3 2008  5.782
```

Leaving out the row (or the column) number requests a whole column (or row) – notice that the comma still has to be provided.

```
seedlings[, 1]
[1] 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015

seedlings[1, ]
  Year Density fitted isHigh
1 2006      5  3.976  TRUE
```

Once vectors have been copied into a data frame, it is good practice to go back to the script tab and remove the originals from the workspace to avoid confusion:

```
rm(Year, Density, fitted, isHigh)
```

Notice that the vectors have disappeared from the **Values** part of the environment window, but they are still in the **seedlings** data frame.

To use vectors stored within a data frame, the name of the data frame must be specified within the command. There are four common ways to do this:

1. Specify the name of the data frame whenever you refer to one of its vectors, separating the two names with a \$ sign.

```
plot(seedlings$Density ~ seedlings$Year, pch=16)
```

2. Specify the vectors via their column numbers (leaving out the row numbers requests the whole column):

```
plot(seedlings[,2] ~ seedlings[,1], pch=16)
```

3. Use the **with()** function. This instructs R to look inside the specified data frame for the vector names. This is often the most economical approach.

```
with(seedlings, plot(Density ~ Year, pch=16) )
```

4. Commands which include a formula will usually allow the data frame to be specified in the command itself.

```
plot(Density ~ Year, pch=16, data=seedlings)
```

3.2.2 *Importing an existing data set into a data frame*

A sample data set is shown in the green box to the right, which contains the results of a hormone treatment on the height and biomass of two varieties of a plant species. There are eighteen rows, each representing an individual plant. The columns specify two categorical vectors (**variety** and **treatment**) and two numeric vectors (**height.cm** and **biomass.gm**)

All columns in the data set must be the same length, and data in a single column must all be of the same type (eg either all numeric, or all categorical).

This data set is provided as a csv file (that is, as comma-separated variables in a text file) called

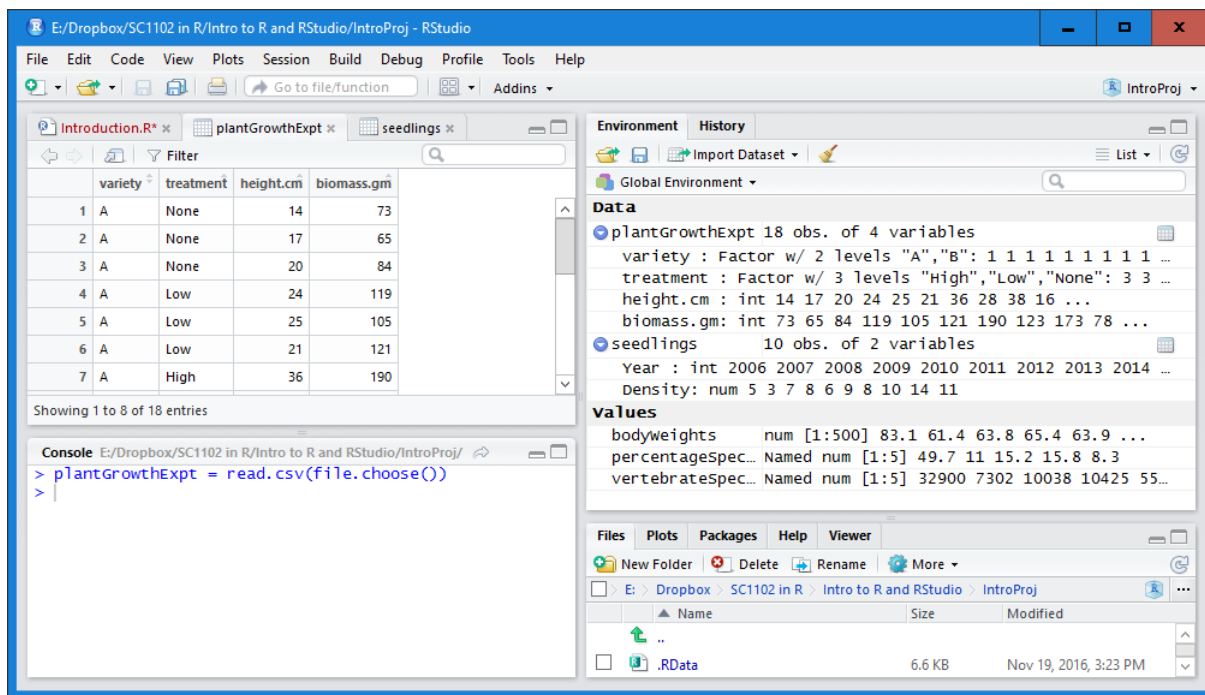
plantGrowthExpt.csv – notice that this file contains **only** the variable names and the data itself – no other headings, and no empty rows. Files can be saved as csv files from Excel and from most other statistical packages. To import it into R, use the command:

```
plantGrowthExpt = read.csv(file.choose())
```

When you run the command, you will get a **Select file** window which lets you navigate to the file, select it, and press **Open** to load it into **R**. Then inspect the file in the **Environment** window. By clicking the little grid button to the right of the file name, display it in a script window tab, as shown below. You can also obtain this display with the command

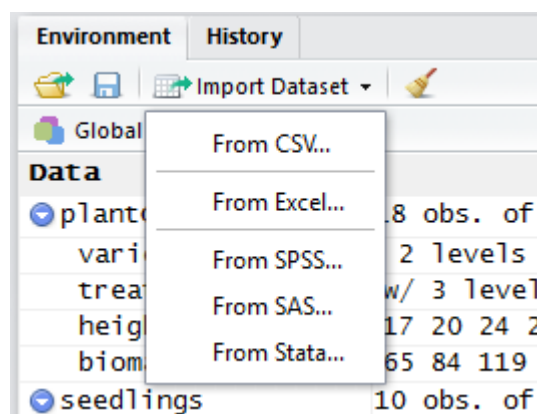
```
view(plantGrowthExpt)
```

variety	treatment	height.cm	biomass.gm
A	None	14	73
A	None	17	65
A	None	20	84
A	Low	24	119
A	Low	25	105
A	Low	21	121
A	High	36	190
A	High	28	123
A	High	38	173
B	None	16	78
B	None	22	115
B	None	24	180
B	Low	27	158
B	Low	34	196
B	Low	21	161
B	High	32	213
B	High	30	174
B	High	39	262



Notice that **RStudio** is telling you that **PlantGrowthExpt** has 18 observations of 4 variables, and that the first two variables – variety and treatment – are factor variables – that is, they define categories. Variety has two factor levels, A and B, and treatment has three – High, Low, and None. In general, when you use **read.csv()** to import a dataset into a data frame, it will assume that any non-numeric variables should be imported as factor variables rather than as character variables.

For importing **csv** files like this one, using the **read.csv()** function is probably the simplest option, and we will continue to use it in this manual. However, **RStudio** provides support for importing files saved in several different formats. To look at the options available, click the **Import Dataset** button at the top of the **Environment** window.



(The first time you use any of these import options, **RStudio** will require you to load additional libraries from the internet which underpin the import process.)

The other two data structures commonly used in **R** – arrays and lists - are less often needed by beginning **R** users, and we only deal with them briefly here.

Exercises 3

1. Remove the “loose” **bodyWeights** and **gender** vectors from the workspace.
2. The resources folder contains a csv file called **TullySugar.ONI.csv**. Import it into an R data frame called **TullySugar**. How many rows and columns does it have? What are the names of its variables, and what is their data type? Display the first few, and the last few, rows of the data frame.
3. Using the **weights** data frame, draw a boxplot to show the distribution of **bodyWeight** values for males and females.

4 Manipulating and describing data

This chapter uses the `plantGrowthExpt`, `weights`, and `seedlings` data frames created earlier to illustrate some options for manipulating data and summarizing it numerically. The next two chapters will review a range of ways to display data graphically.

The simplest option for describing the data held in a data frame is the `summary()` function, which provides summary statistics for each vector in the data frame:

```
summary(plantGrowthExpt)
```

variety	treatment	height.cm	biomass.gm
A:9	High:6	Min. :14.0	Min. : 65.0
B:9	Low :6	1st Qu.:21.0	1st Qu.:107.5
	None:6	Median :24.5	Median :140.5
		Mean :26.0	Mean :143.9
		3rd Qu.:31.5	3rd Qu.:178.5
		Max. :39.0	Max. :262.0

```
summary(weights)
```

gender	bodyWeights
Female:500	Min. : 39.44
Male :500	1st Qu.: 64.07
	Median : 72.50
	Mean : 73.04
	3rd Qu.: 82.10
	Max. :117.98

```
summary(seedlings)
```

Year	Density	fitted	isHigh
Min. :2006	Min. : 3.00	Min. : 3.976	Mode :logical
1st Qu.:2008	1st Qu.: 6.25	1st Qu.: 6.008	FALSE:5
Median :2010	Median : 8.00	Median : 8.040	TRUE :5
Mean :2010	Mean : 8.10	Mean : 8.040	NA's :0
3rd Qu.:2013	3rd Qu.: 9.75	3rd Qu.:10.071	
Max. :2015	Max. :14.00	Max. :12.103	

Notice that for factor and logical variables, using `summary()` provides the number of observations for each level. For numeric variables (both integer and double), it provides the minimum, maximum, first and third quartiles, and the median and mean.

This is straightforward, but doesn't begin to cover all the ways you might want to manipulate and summarize the data. For example, you might want to apply a log transform to the biomass data, or calculate mean body weights for males and females separately. In fact, most data analysts spend more time getting their data into the right form than they do on formal analysis.

Below we look at a number of techniques for manipulating, transforming and summarizing data. Initially (section 4.1) we look at some techniques available in the **R** base package. Then, in section 4.2 we look at the options provided by a specialist **R** package, `dplyr`, written specifically to make data manipulation and description easier, faster, and more flexible.

4.1 Simple data transformations using base R

4.1.1 *Numeric transforms*

The `plantGrowthExpt` data frame includes both the height and the biomass of each plant. Suppose we want to look at how robust (or spindly) each plant is, and decide that the biomass/height ratio is an appropriate measure. To create a new variable (`bh.ratio`) for this measure within the `plantGrowthExpt` data frame:

```
plantGrowthExpt$bh.ratio = with(plantGrowthExpt,  
                                biomass.gm / height.cm)
```

Similarly, to create a variable which is the square root of the biomass:

```
plantGrowthExpt$sqrt.biom = with(plantGrowthExpt,  
                                 sqrt(biomass.gm))
```

R includes a wide range of functions which may be used to transform data. The table below gives a brief list of some of the most commonly used. In each case **x** is a number or vector of numbers.

Transform	What it does	When it is used
log(x) log10(x)	Calculate natural and base 10 logarithms	x must be positive and greater than 0 Often used to make right-skewed data more symmetric and closer to a Normal distribution,
sqrt(x)	Calculate the square root	x must be positive Sometimes used to normalize somewhat right-skewed data.
asin(x)	Calculate the arcsine	x must lie between 0 and 1 Sometimes used to normalize proportional data where values are not too close to 0 or 1.
1/x	Calculate the reciprocal	Sometimes used to normalize very right-skewed data.
x^2	Calculate the square	Sometimes used to normalize somewhat left-skewed data where all values have the same sign
exp(x)	Calculate the exponential	May be used to normalize left-skewed data or to convert log-transformed values back to their original form
scale(x, center=TRUE, scale=TRUE)	With the default arguments, calculate the z-score	Rescales the data so that it has a mean of 0 and a standard deviation of 1.

4.1.2 Logical transforms

It is often useful to create a logical variable which simply takes the value **TRUE** or **FALSE**. For example in the **plantGrowthExpt** data, we might want to create a variable **isLarge** which has the value **TRUE** if the plant's biomass is greater than the mean biomass for the whole group, and **FALSE** otherwise. This is straightforward using a logical expression:

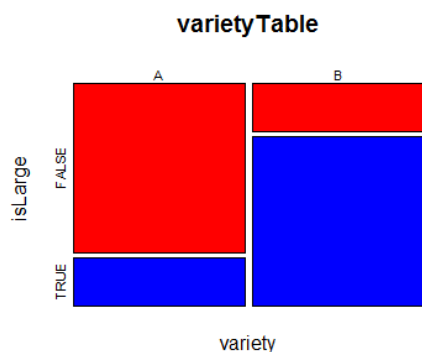
```
plantGrowthExpt$isLarge = with(plantGrowthExpt,  
                               biomass.gm > mean(biomass.gm))
```

Then we might check whether large plants are similarly distributed between varieties: the result suggests that larger plants may be more likely for variety B than for variety A.

```
varietyTable = with(plantGrowthExpt, table(variety, isLarge))  
varietyTable  
      isLarge  
variety FALSE TRUE  
      A      7    2  
      B      2    7
```

Use a mosaic plot to display the table.

```
mosaicplot(varietyTable, col=c("red","blue"))
```



4.1.3 Creating categories from numeric data

To demonstrate this we use the **TullySugar.ONI** data frame loaded from a csv file in Exercise 3.2 – if it is not already loaded, load it now. It contains three vectors:

- **Year** (from 1945 to 1999),
- **Cane.yield** (measuring sugarcane yield in the Tully region for each year)
- **ONI.SON**. This third variable represents the Oceanic Niño Index (ONI) during September-October-November each year. The index represents the deviation from the long-term average of the equatorial mid-Pacific sea surface temperature at this time of year. So positive values mean the water is warmer than average, and negative values mean that it is cooler. The ONI is used to define the climatic phase – values of 0.5 or higher are defined as El Niño, values of -0.5 or lower are defined as La Niña, and values between -0.5 and +0.5 are defined as Neutral.

The command below creates a variable called **phase** which specifies the climatic phase for each year. It uses the **cut()** function to match the right phase label with each **ONI.SON** value.

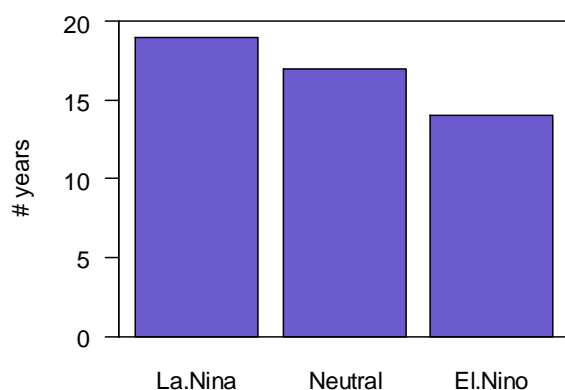
```
TullySugar.ONI$phase = with(TullySugar.ONI,  
                             cut(ONI.SON, breaks=c(-Inf,-0.49, 0.49, Inf),  
                                labels=c("La.Nina", "Neutral", "El.Nino")))
```

Notice the use of **-Inf** and **Inf** to represent minus and plus infinity. The breaks define three categories – all values up to -0.49, values between -0.49 and 0.49, and all values greater than 0.49. Because the **ONI.SON** values are reported with only one decimal place, setting the break points with an additional decimal place means that we don't have to worry about whether values which exactly equal a break point are inside or outside a particular category (because breaks are defined so that no value will ever exactly equal a break point). Notice also that the categories are created as factor data, and that the order of the levels is not alphabetic – they are in order of increasing size of the cut variable.

Then the commands below find how many years were in each phase and display the results in a bar plot

```
with(TullySugar.ONI, table(phase))
La.Nina Neutral El.Nino
  19      17      14

with(TullySugar.ONI, barplot(table(phase), col="slateblue",
                             ylab="# years", las=1, ylim=c(0,20)))
box()
```



Exercises 4a

- In the **seedlings** data frame, create two new variables:
 - One called **residual** which is the difference between the actual density and the fitted value.
 - A second called **logDensity** which is the log-transformed density.
 Use the **summary()** function to find the minimum, maximum, mean and median of both of the two new variables
- Create a variable **highRatio** within the **plantGrowthExpt** data frame which takes the value **TRUE** if **bh.ratio** is above its overall median, and **FALSE** if it is below the median. Produce a table to examine whether the frequency of **TRUE** and **FALSE** values differs between varieties,
- Within the **weights** dataframe, create a new variable called **weightclass** with three weight categories (Low, Medium, and High) for values <65, 65-85, and >85 kg. Produce a table and plot to show how many males and how many females are in each weight class.

5 Plotting data with base R

R has a very extensive plotting and plot-editing capability, and is particularly well suited to exploratory data analysis. Here we expand on the introduction to plots given in chapter 2 to cover some of the more common plotting functions which you are likely to need.

For the first few plots demonstrated below, we show you both the plot with default options and the code changes needed to put it in its final form.

5.1 Displaying frequency distributions

5.1.1 Bar graphs for categorical data: `barplot()`

We demonstrated a basic `barplot` command in section 2.3: check its arguments using `?barplot`.

Gross Domestic Product (GDP) is the value of goods and services produced by a country in a year.

This table shows the 2014 per-capita GDP (in \$US) of countries in Oceania, as estimated by the International Monetary Fund.

We want to put them in ascending order (using the `arrange()` function in `dplyr`), and then display the table as a bar plot, with the bar lengths proportional to the per-capita GDP and each bar labeled with the country.

Because some country names are long, they will be easier to read if the bar plot is horizontal.

Country	per capita GDP
Australia	47600
Federated States of Micronesia	3000
Fiji	9400
Kiribati	1800
Marshall Islands	3200
New Zealand	37100
Palau	15300
Papua New Guinea	5400
Samoa	2000
Solomon Islands	1653
Tonga	5300
Tuvalu	3600
Vanuatu	2600

The data shown in the table above can be found in the file `OceaniaGDP.csv`. Read the file into a data frame, check the structure of the data in the environment window and examine the first few rows, and then use `arrange()` to sort the countries in order of per capita GDP:

```
OceaniaGDP = read.csv(file.choose())
head(OceaniaGDP)
```

```
      Country per.capita.GDP
1    Australia      47600
2 Federated States of Micronesia    3000
3         Fiji      9400
4     Kiribati      1800
5 Marshall Islands      3200
```

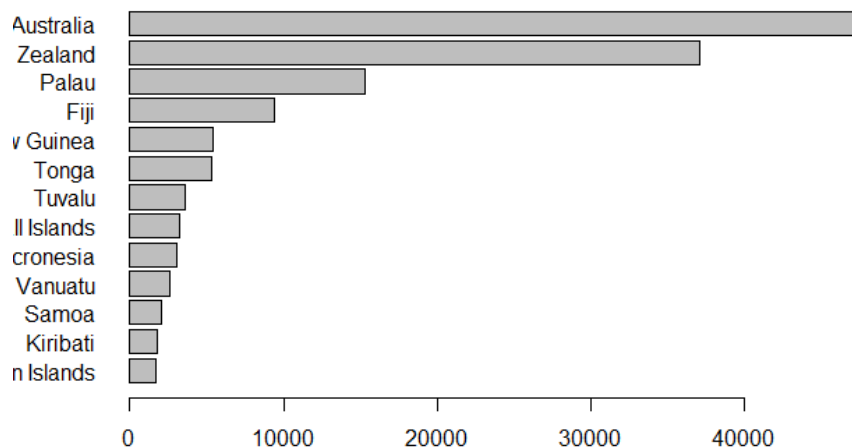
```
library(dplyr)
OceaniaGDP = arrange(OceaniaGDP, per.capita.GDP)
```

The **names** argument in **barplot()** can be used to specify a vector holding the bar labels – it needs a vector of character strings, but the **Country** variable has been read as factor data, so the code below creates a character vector called **labels**.

```
labels = as.character(OceaniaGDP$Country)
```

Below is a first try at the bar plot, using the **las=1** argument to force the y-axis labels to be horizontal.

```
barplot(OceaniaGDP$ per.capita.GDP, horiz=TRUE, names=labels, las=1)
```



The result is less than satisfactory! Things to change include:

1. The left margin of the plot is much too narrow for the labels.
2. The x-axis should extend to 50000 so that it extends a little past the largest bar.
3. The colour is a little uninspired.
4. The x-axis should have a label to say what the numbers mean.
5. It might be good to have a title for the plot.

The first of these is the trickiest since margins have to be set as a global graphical parameter using the **par()** function. (Use **?par** to check the function's numerous arguments.) The required argument here is **mar** (for margins), which takes a vector of four values, one for each margin, starting at the bottom and moving clockwise. The default settings are 5,4,4,2, so we need to increase the second one – to 14 should be enough. When the **par()** settings are changed, they stay changed until you change them back, so it is useful to store the old settings in order to restore them later.

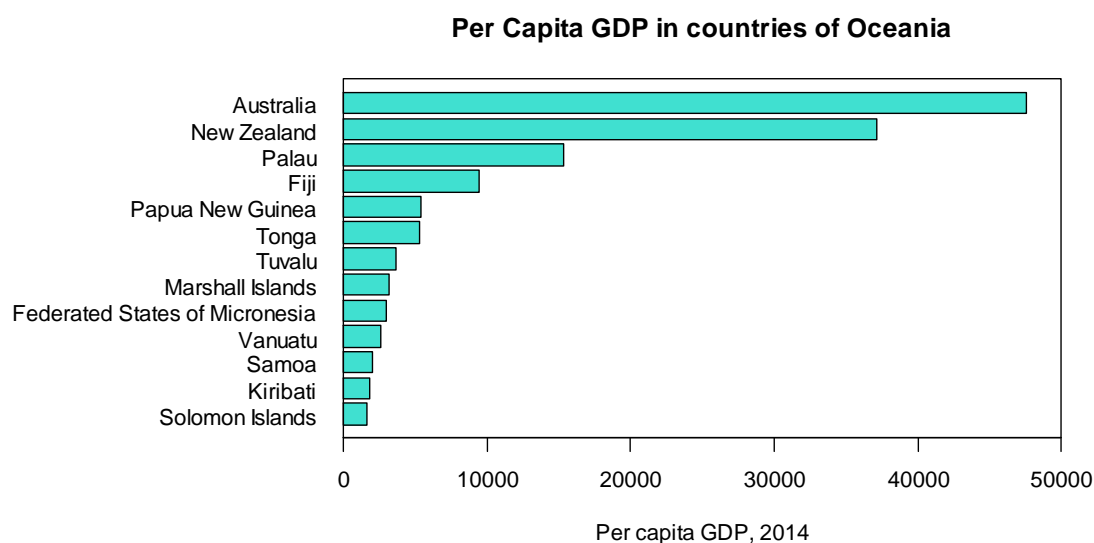
```
opar = par()      ## store the default graphics settings in opar
par(mar=c(5,14,4,2)) ## increase the left margin
```

The remaining changes can be handled via arguments within the **barplot()** command.

```
barplot(OceaniaGDP$per.capita.GDP, horiz=TRUE, names=labels, las=1,
        xlim=c(0, 50000), col="turquoise", xlab="Per capita GDP, 2014",
        main = "Per Capita GDP in countries of Oceania")
```


Finally, add a frame for the plot and restore the default graphics settings

```
box ()
par (opar)
```



5.1.2 Stacked bar plots for proportions: `barplot()`

The most common way to display proportional data is a stacked bar plot. The original data will usually be counts: although you can stack the raw counts, it is usually better to convert them to proportions or percentages (so that they add up to 1 or to 100%) before creating the plot.

Below is a table of counts resulting from a poll about Australian voter intentions in three towns. We will construct a stacked-bar plot which compares the percentages of each voting intention in each town.

Location	LNP	Labour	Green	Other
Town A	2246	1877	482	205
Town B	1125	1318	404	130
Town C	978	525	215	101

In **R**, a stacked bar plot requires a matrix for data entry: the values in each column of the matrix are stacked. Since we want each bar to represent a town, the rows in the table above will need to become columns in the matrix. The code below creates a vector of percentages for each town, uses `cbind()` to combine them into a matrix called **VoterIntention**, and then uses the party affiliations as row names for the matrix.

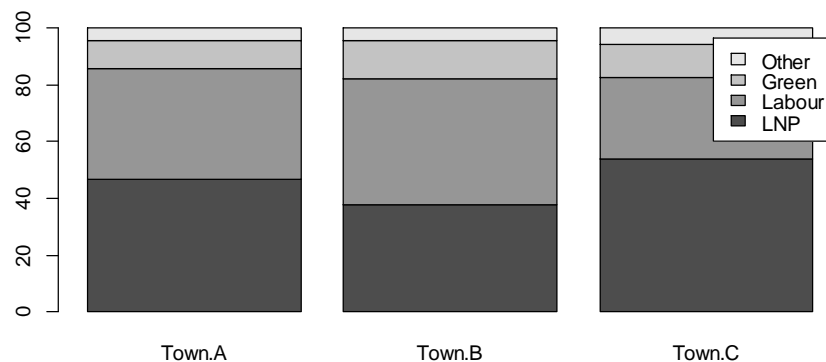
```
Town.A = c(2246, 1877, 482, 205) ## create vector of voter counts
Town.A = 100*Town.A/sum(Town.A) ## convert counts to percentages
Town.B = c(1125, 1318, 404, 130)
Town.B = 100*Town.B/sum(Town.B)
Town.C = c(978, 525, 215, 101)
Town.C = 100*Town.C/sum(Town.C)
VoterIntention = cbind(Town.A, Town.B, Town.C)
rownames(VoterIntention) = c("LNP", "Labour", "Green", "Other")
```

VoterIntention

	Town.A	Town.B	Town.C
LNP	46.684681	37.789721	53.765805
Labour	39.014758	44.272758	28.862012
Green	10.018707	13.570709	11.819681
Other	4.281854	4.366812	5.552501

Then we can try the default plot. Setting the **legend.text** argument to TRUE requests the plot to include a legend and to use the row names as labels for the legend.

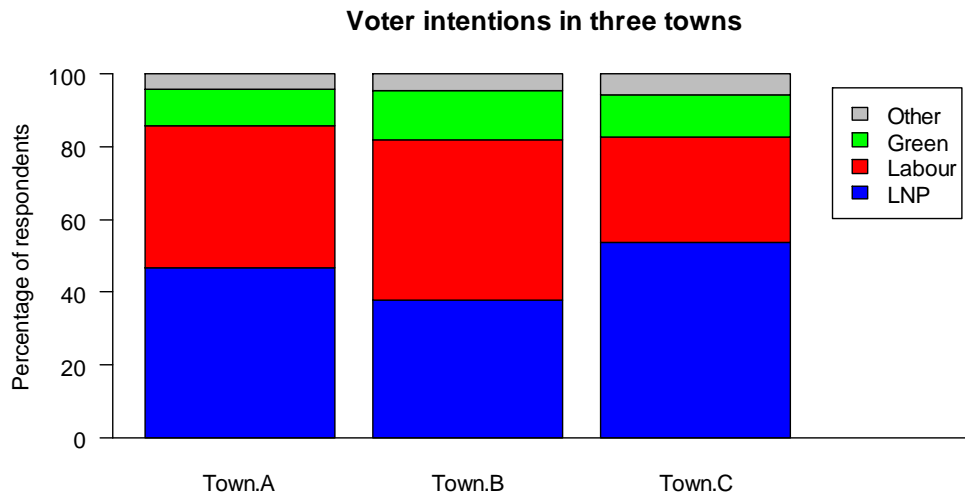
```
barplot(VoterIntention, legend.text=TRUE)
```



Not quite right yet. We need to:

1. Make space for the legend. Since the legend is within the plot area, not in the margin, changing the margins won't help this time. However we can extend the x-axis to make space within the plot area (bars are 1 unit apart: trial and error suggests x-axis limits of about 0.2 and 4.4 are OK).
2. Make the y-axis values horizontal.
3. Provide a y-axis label.
4. Change the colour scheme.
5. Add a title for the plot,
6. Add a horizontal line at y=0 to serve as an x-axis.

```
barplot(VoterIntention, legend.text=TRUE, xlim=c(0.2,4.4),
        las=1, ylab="Percentage of respondents",
        col=c("blue","red","green","grey"),
        main="Voter intentions in three towns")
abline(h=0)
```



5.1.3 Box plots for numerical data: `boxplot()`

Boxplots and histograms are the most common ways to display and compare the distributions of sets of numerical values.

- Boxplots allow a simple comparison of the median, interquartile range, and total range of a numerical variable;
- Histograms split the data values into consecutive intervals (bins) in order to plot frequencies in each interval.

To demonstrate how these are drawn and used to compare distributions, we use a sample of blood haemoglobin concentrations for human males living in different locations. In this section we use boxplots, in the next, we will use histograms.

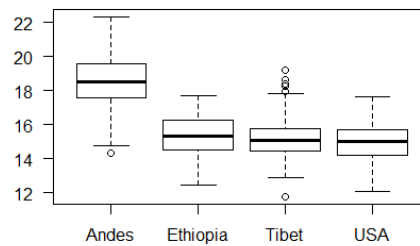
Three of the locations are at high altitude (Tibet, Ethiopia, and the Andes) and one is at sea-level (USA). The data are available in the file **haemoglobin.csv**. Import the data into a new R data frame called **haemoglobin**, check its structure in the **Environment** window, and examine its first few rows .

```
haemoglobin = read.csv(file.choose())
head(haemoglobin)
```

```
  group  Hb
1 Ethiopia 12.47
2 Ethiopia 12.68
3 Ethiopia 12.54
4 Ethiopia 12.55
5 Ethiopia 12.77
6 Ethiopia 12.57
```

The **Environment** window tells us that group is a factor variable with four levels. Generating the default boxplot is straightforward.

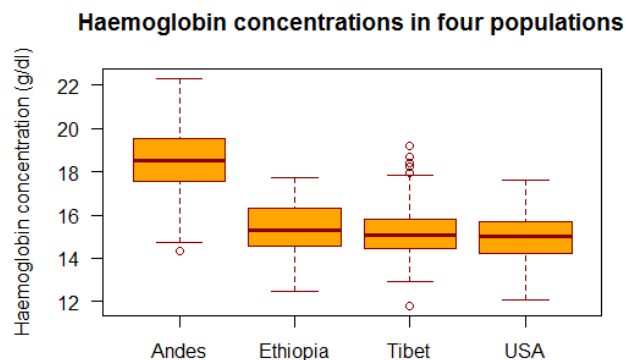
```
boxplot(Hb ~ group, data = haemoglobin, las=1)
```



The upper and lower boundaries of the boxes show the interquartile range. The whiskers extend an additional 1.5 times the interquartile range. More distant points are plotted as individual outliers. This is the most conventional form of boxplot, but other forms are possible. Run `?boxplot` to investigate alternative options.

Modifications are also straightforward. The command below adds a y-axis label and a plot title, and changes the colour scheme.”

```
boxplot(Hb ~ group, data = haemoglobin, las=1,
        ylab="Haemoglobin concentration (g/dl)",
        main="Haemoglobin concentrations in four populations",
        col="orange", border="darkred")
```

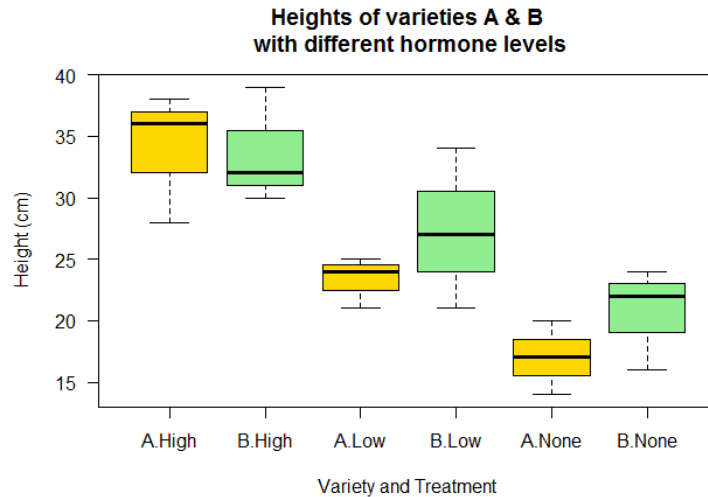


The plot makes it very obvious that the sample from the Andes tends to have higher values than the other three locations.

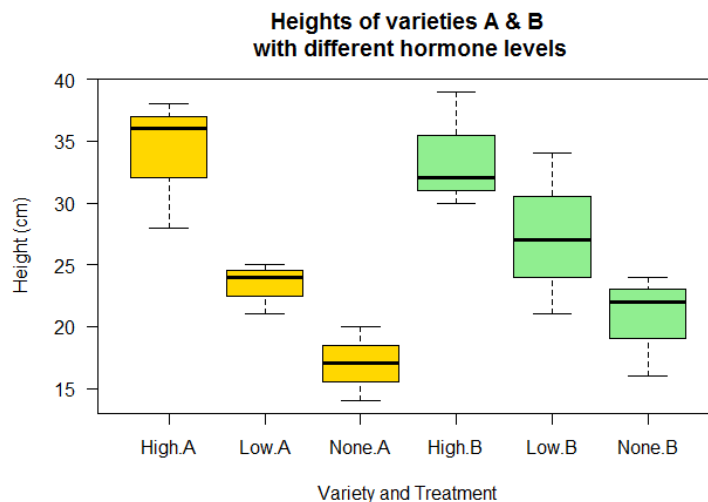
It is also possible to examine the effects of two factors simultaneously. Below we load and examine the `plantGrowthExpt` data set and then plot the effect of variety and hormone treatment on plant height. via two different layouts. Each layout draws six groups in two colours. In the first, we want two colours to alternate, so we need only specify only two, and the pair will repeat. In the second, we want two sets of three, so we need to specify values for all six.

```
plantGrowthExpt = read.csv(file.choose())
head(plantGrowthExpt, 4)
  variety treatment height.cm biomass.gm
1      A      None        14         73
2      A      None        17         65
3      A      None        20         84
4      A      Low         24        119
```

```
boxplot(height.cm~variety*treatment, data=plantGrowthExpt,
        col=c("gold","lightgreen"),las=1,
        xlab="Variety and Treatment", ylab="Height (cm)",
        main="Heights of varieties A & B\n with different hormone levels")
```



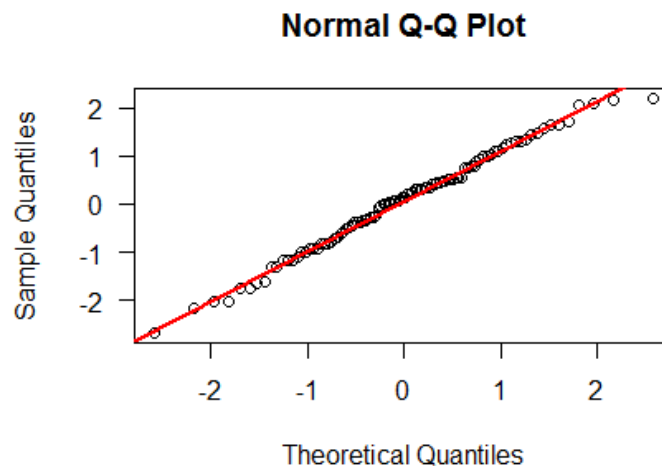
```
boxplot(height.cm~treatment*variety, data=plantGrowthExpt,
        col=rep(c("gold","lightgreen"), each=3), las=1,
        xlab="Variety and Treatment", ylab="Height (cm)",
        main="Heights of varieties A & B\n with different hormone levels")
```



5.1.5 Diagnostic plots

R provides a variety of plots designed to evaluate whether the assumptions of particular tests are met. We will look at one common example here. Many commonly-used statistical tests assume that data are Normally distributed. The `qqnorm()` and `qqline()` functions can provide a quick visual check of this assumption. A quantile-quantile (Q-Q) plot plots the quantiles of one data set against the quantiles of another. If they come from the same distribution, the points will lie close to a straight line with a slope of 1, through the origin. So to check if a set of data are Normally distributed we plot the quantiles of the data against what the quantiles should be if the data really are Normally distributed. Substantial curvature of the points away from the line would indicate a failure of the normality assumption

```
## Generate a set of 100 randomly chosen data points from a Normal
## distribution, and then check their normality with a QQ plot
trialdata = rnorm(100)
qqnorm(trialdata, las=1)
qqline(trialdata, col = "red", lwd=2)
```



5.2 Showing relationships between two variables

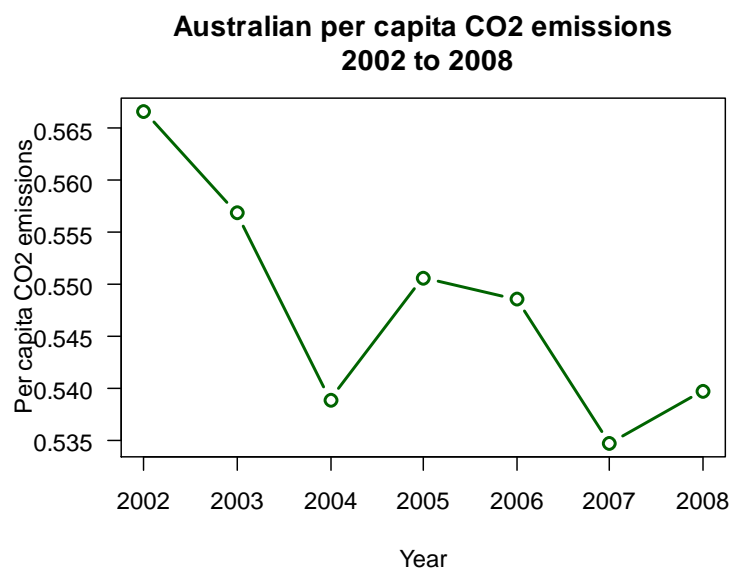
5.2.1 *Line plots for showing data sequences through time*

We will plot the per capita CO2 emissions for Australia for the years from 2002 to 2008. The data (from the World Bank) are provided as a csv data file called **perCapitaCO2.Australia.csv**: load it into a data frame called **CO2Australia**. Note that it has two vectors, **Year** and **CO2**.

```
CO2Australia = read.csv(file.choose())
```

We provided code to produce a line plot in section 2.3. Equivalent code for this data set is shown below (note the use of `\n` to spread the title over two lines:

```
plot(CO2 ~ Year, data=CO2Australia, type="b",
     lwd=2, las=1, col="darkgreen", cex=1.2,
     ylab = "Per capita CO2 emissions",
     main = "Australian per capita CO2 emissions\n 2002 to 2008")
```



The y-axis label is a little cramped, the code below expands the y-axis margin slightly to accommodate it, and uses the `title()` function to add the axis label (`title()` allows the distance between the axis label and the axis line to be modified). (The problem could also be fixed by reducing the size of all tick labels using `cex.axis=0.8` within the `plot()` command – try it!)

```
opar = par()
par(mar=c(5,6,4,2))
plot(CO2 ~ Year, data=CO2Australia, type="b",
      lwd=2, las=1, col="darkgreen",
      ylab="",
      main = "Australian per capita CO2 emissions\n 2002 to 2008")
title(ylab = "Per capita CO2 emissions", line=4)
par(opar) ## warning messages can be ignored
```

