Introduction to R

QFAB Bioinformatics

QFAB

qcif

Queensland Cyber Infrastructure Foundation Ltd, ABN 13 225 133 729, Axon building, 47 – The University of Queensland - St Lucia, Qld, 4072
(QCIF incorporates QFAB Bioinformatics). 16.01.023-20171023

Contents

Abstract

R is a open-sourced software programming language and software environment for statistical computing and graphics. The R language is widely used among statisticians and data analysts for developing statistical workflows for data analysis. R is an implementation of the S programming language combined with lexical scoping semantics inspired by Scheme. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

Bioconductor is a project to develop innovative software tools for use in computational biology. It is based on the R language. Bioconductor packages provide flexible interactive tools for carrying out a number of different computational tasks. Literate programming and analytical pipeline crafting.

R and Bioconductor may not be as fast as dedicated bioinformatics software for computationally intensive processes such as mapping short sequence reads onto the genome, but the flexibility of having raw access to all of the data, methods and structure is empowering.

Chapter 1

Getting started

R is a statistical environment and programming language for data analysis and graphical display. The software is open-source, freely available and has been compiled ready for use on Windows, Mac and Linux computers. The Bioconductor framework has a considerable number of packages that have been implemented for the analysis and exploration of metabolomics, proteomics, DNA microarray and Next Generation DNA sequence data.

This workshop is intended as an introduction to R that should familiarise you with the concepts that underpin the crafting of workflows in R and some of the key techniques that will aid in the crafting of reusable workflows.

Some of the topics we will cover over the course of the day include:

- Using the RStudio IDE for running a data analysis in R
- The basics of literate programming for studies using R
- Loading, creating, modifying and saving basic -omics data objects according to key formats
- Create graphs and plots of data
- Understanding how to read and understand someone else's R-code

1.1 How to read this book

In the shaded dialog boxes with the thick left border are commands that should be typed into your R *Console* window. This corresponds to **input**, the **output** will sometimes follow below with lines preceeded with two hashmarks (##).

```
print("HELLO WORLD")
```

```
## [1] "HELLO WORLD"
```

Tips, Suggestions and Traps

Comments and salient advice are provided in dialogs such as this. We'll use this format to warn you of traps that you could slip into and to provide some hints.

Warnings and Traps

Warnings are provided in dialogs such as this. We'll use this format to warn you of traps that you could slip into and to provide some hints.

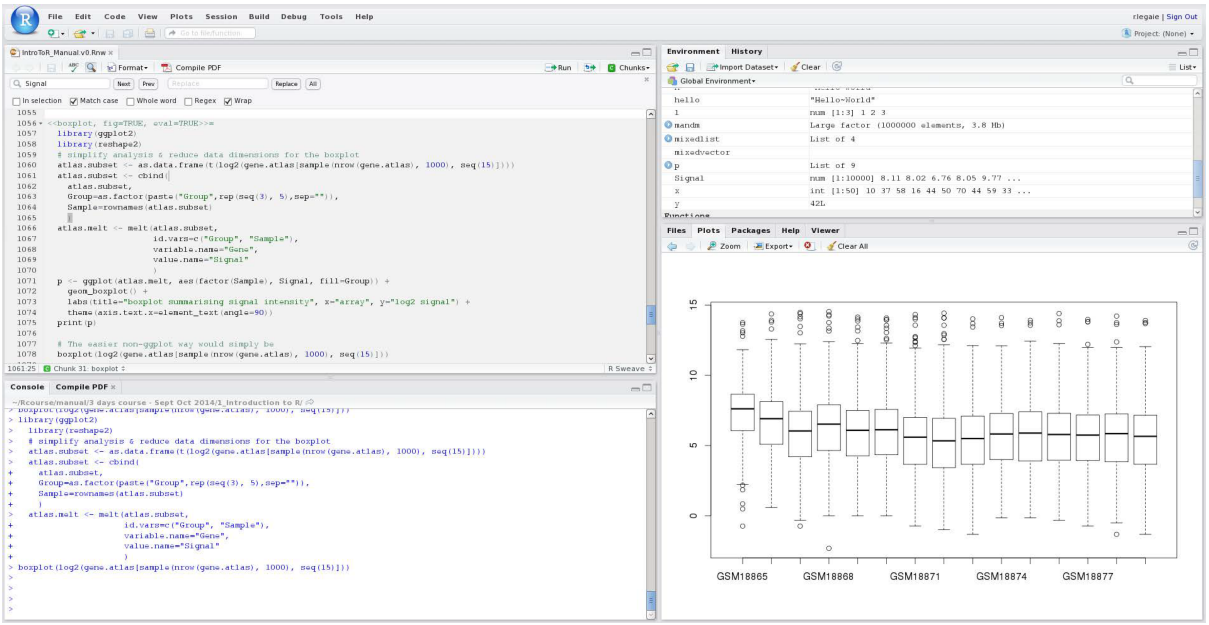


Figure 1.1: A screenshot of the RStudio session used in the preparation of this course material. The document being prepared is displayed in the top left pane. A figure is shown in the bottom right pane and the loaded environment variables are presented in the top right. Having access to all of these information in a single application is simpler than managing an active R console, a text editor and an image viewer.

Time for you to make R work

Exercises are the best way to learn. A document can provide an insight to the process but hands-on interaction with R is the only way to learn. In the exercise boxes are some suggested exercises that will apply the knowledge that is being shared.

1.2 RStudio IDE

RStudio is a free and open source integrated development environment (IDE) for R. An IDE is most typically used in software development where a user is presented with a unified environment where code can be edited, documentation can be read and debuggers can be applied to understand what is happening in the code and why it may not be working as expected.

In bioinformatics (or computational biology) an IDE provides us with an integrated approach to writing R scripts (or markdown documents), interacting with our data as we implement our script and provides us with an overview of the objects that we have created and their content. RStudio also retains a history of the commands that we have typed, a collection of the figures that we have prepared and provides access to method documentation and package vignettes. We will consider all of these aspects during this workshop. The RStudio IDE is great for the preparation of scripts and reports since the code syntax-highlighting helps you discover typos and errors in your code and even spelling mistakes.

RStudio comes in two main flavours: a *Desktop* version that runs straight off the computer that we are sitting at and a *Server* version that can be hosted on a slightly more capable computer in a data centre. The server version is sometimes preferable since it is accessed via a web-browser and can be configured with massive memory and disk allocations that might not be practical for a normal laptop. The server version is platform-agnostic (from the user perspective that is; the server itself needs to run Linux) and the Desktop version can be run on Windows, Mac and Linux computers. An example RStudio session running on the Desktop version is shown in Figure 1.1.

RStudio integrates very cleanly with a number of best scientific working practices (data sharing, reproducible research and research documentation). RStudio allows for experiments to be performed in

projects - this allows you to create a separate workspace for each study that you are performing. Code can be shared between projects easily and Version Control (Subversion and Github) facilitate the sharing of workflows and methods with other researchers.

For this course we recommend that you use our server version that has been preconfigured for the software, packages and data that will be used.

Open up an RStudio server client and create a project

1. Open a web browser (not Internet Explorer) and connect to the server given to you by the trainer. Use the username and password that we have provided to connect and explore the options available in the top bar.
2. Explore the interface
   a) Where is the **Files** tab? What does it show? Can you see the **data** folder?
   b) Click the **data** folder to look inside, keep going until you get to some files
3. Click on the **Packages** tab and look at the list of packages that are installed in R. If you are ahead, scroll down and click on the *methods* name. This should bring you to the **Help** tab, see what additional functions you can do using this package.
   – We will talk about what packages are later in the workshop.
   a) In the top-right corner of the **Help** tab is a search text field. Type in *"print"* and hit enter (if the autofinish starts to pop up click on print). This will show you the `print()` function and what parameters it accepts.

1.3 Alternatives to RStudio

RStudio is not the only way to access R. The standard R installation from CRAN (Comprehensive R Archive Network) provides a basic R graphical user interface called the **R console**. This provides access to the key R functionality and provides interfaces for the preparation of R scripts. The simplest way to access R is perhaps through the command line. With a typical R installation simply typing `"R"` at the console should load an R session which you can directly interact with, as seen in Figure 1.2.

1.4 The R workspace

The **workspace** refers to a R working environment and the collection of objects that have been created by the user. A **session** refers to an instance of a workspace. In a session you populate your workspace with a collection of objects and functions. A session can be saved, this means that you can save the workspace content so that the next time that you use R you can have all of the information loaded and already available.

Session and workspace concepts are best understood within an environment such as RStudio.

1.5 Literate programming

There are two approaches to use R to perform data anaylsis:

- The first approach is done interactively via the **console** window. You can tell you are in this window by the `>` prompt and the blinking cursor. When you are in this window, all commands are executed *immediatedly* as soon as you hit the `[Enter]` key.
- The second approach is by writing scripts (or text file) and then running the script from top to bottom. This is done via the **editor** window (not visible by default, only when you open a script file.) The commands entered in this screeen is *not* executed immediatedly. A script consists of a

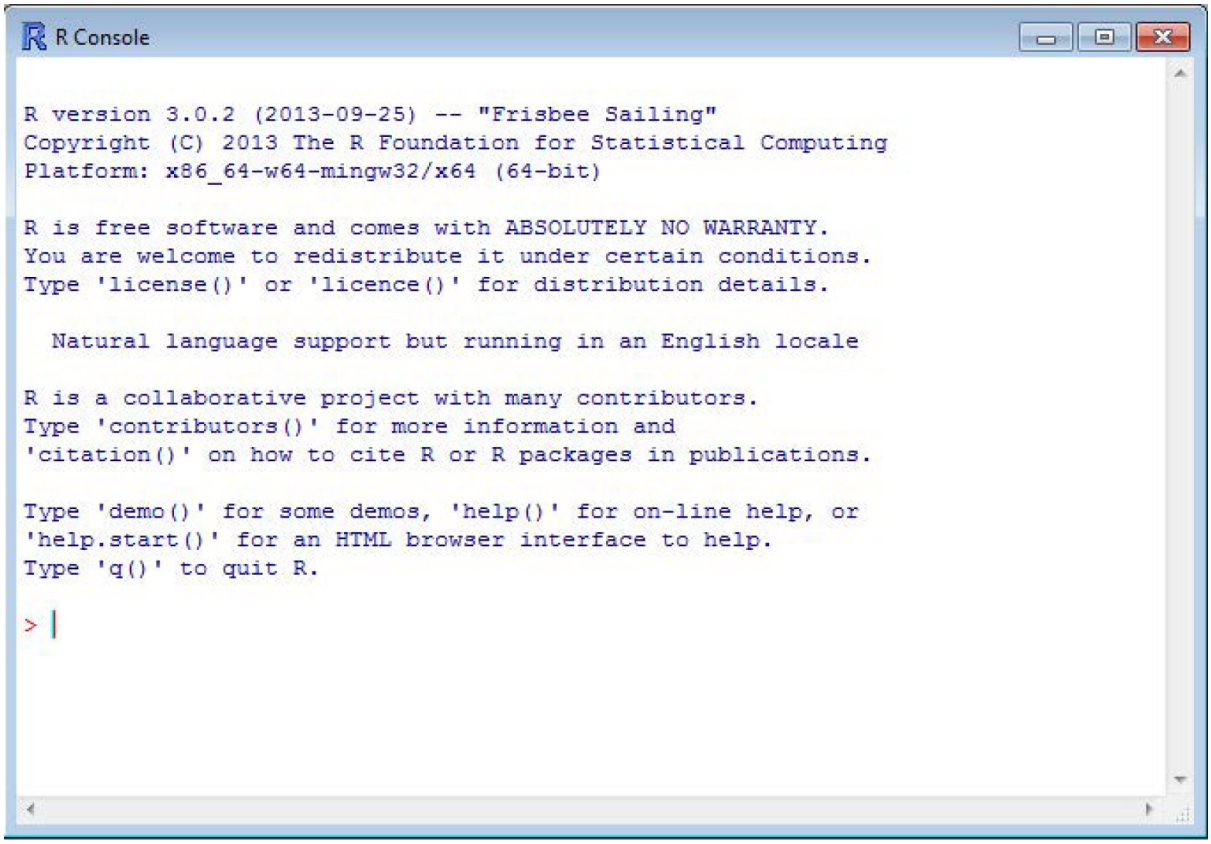


Figure 1.2: R installations typically include a basic R editor and console that provide a basic framework for interacting with the software, preparing R scripts and for the installation and management of packages. This figure shows the R console as installed by default on a Mac computer.

collection of commands that perform a larger task on the data and is generally executed from top to bottom. A quick example will be shown in Section 7.6.4.1.

The preferred mode of operation is up to the user. There are pros and cons using either method. The interactive mode is quick to start and you get immediate feedback about your data. However, it does not support repeated analysis very well as you would constantly have to retype the same lines of code again and again. The scripting method is very useful for repeated analysis but can be slow to set up as mistakes in the command can become hard to debug for large scripts. With practice you will become more familary (and faster) and will develop your own style of working.

When using RStudio, it allows you a third mode of operation which is a hybrid of the two approaches mentioned above. That is, you write your commands in the editor window and run sections of the code (or **chunks**) as required. So the analysis is done semi-interactively.

For this workshop, we recommend that you use the script method to work through the exercises. That is, create a new text file for each section and execute the lines one at a time so you can see what happens with each command. This will allow you to come back to the exercises in your own time and review the content.

Literate programming

1. Try some basic functions in the **Console** window (the large panel to the left)
   a) Type `1 + 1` and hit [ENTER], what is the output and where is it shown?
   b) Type `print("HELLO WORLD")` command in the console and hit [ENTER].
   c) Try some other functions, e.g.
   - `3^2`, which is three squared
   - `11%%2`, which is the modulo function (the remainder left after the first number is divided by the second).
2. Go to **File** and select **New File** > **R Script** and type the same commands in the **editor** window.
   a) Highlight *only* line 1 and click on the **Run** button at the top right of the editor window (or hit **[Ctrl]+[Enter]** on the keyboard)
   b) Where is the output shown?
   c) Highlight all three lines and click on the **Run** or **[Ctrl]+[Enter]**
3. Create a New Project that we will use for this course.
   a) Under the **File** menu, select **New Project**
   b) Click on **New Directory**, then **Empty Project**
   c) Select a name for your project. Generally this should reflect the analysis that you will be performing. Remember, you may want to come back to your projects months or even years in the future.
   d) You should now see an `.Rproj` file in your **Files** tab, with the name of the project you have just chosen. This file will contain your R session.

Below are some helpful tips when using RStudio to help you speed up your analysis.

The up arrow key

When typing in the **console** window, you can recall previous commands by hitting the `[up arrow]` key. This will cycle through the history of your previous commands starting with the most recent. Use the `[up arrow]` and `[down arrow]` keys to move through the commands. Hit the `[esc]` key to escape and return to the `>` prompt.

The history window

The history window on the top right hand corner (next to Environment) holds a history of all the commands you have executed in the past. This is a faster way of navigating your previous commands to rerun them. But this window is only available in these types of development environment software like RStudio. You even: * search your history using the search box * highlight the desired lines and click on either the `To Console` or `To Source` button

1.6 Documentation

As you may now appreciate, there is a massive amount of information tied to R. There are functions already implemented for most of the typical data transformations that you may wish to perform. One of hardest challenges with R is finding the method that does what you need. It's out there!

Function	Description
`?`	is used to find out information about a specific function. E.g. `?t.test` will give you information about how to use the `t.test` function in R.
`apropos()`	will return a vector of all the objects or functions with names containing the specified search string. E.g. `apropos("test")` will find around 50 different statistical test functions, while `apropos("mixed")` will find `mixedList` and `mixedVector` variables that we created earlier (or at least it would if we hadn't just deleted all our objects!)
`help.search()`	searches the help documentation for entries matching the search string. Items will match if the search term is included in the function description or keywords, not just the name as is the case for `apropos`.

```
?t.test
apropos("test")
help.search("topic")
```

Chapter 2

Data variables (objects) in R

As with all other computer languages information needs to be stored in a way that can be recalled, displayed and analysed. A **variable name** is used to point to a data object in memory (or on disk).

The conventions for a variable name:

- should not start with a number,
- should not contain spaces,
- should not be the same as function names as this can be ambiguous,
- should not contain special characters such as `#`, `%`, `&`. These characters require special handling and often lead to errors that require debugging.

2.1 Declaring variables

Declaring a variable (or object) is simple in R, we just need to give it a name and the content for that variable. This can be done either directly in the console or in the editor window and saved as a script.

Remember that code entered into the console window will be executed immediately when you press the [Enter] key. But do not expect output for every line you enter, many R commands do not print anything to the screen. Code in the editor document will only be executed when you run the lines of commands or run the entire script.

Declare an variable

1. Enter the example code below into your RStudio interface. You can type it directly into the console and pressing the **[Enter] key** after each line or in the editor window.
   a) Which method are you using? console or editor?
   b) Did you get any output after executing line [1]?
2. If you entered above code in the console, click on the **Environment** tab in the top-right corner window. Do you see the value `x` listed? If you do not then try again (or select the `x <- 42` line in your editor window and click on **Run**).
   a) Try other commands, e.g. `y <- 555`, do you see a new variable `y` listed in the *Environment* tab?

```
x <- 42
x
```

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

```
print(x)
```

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

Setting variable values and viewing variable contents

Command	Description
`<-`	This marker is used to set a variable's value. The more standard `=` operator will also work but they have different levels of precedence. As best practice it is better to use `<-` in R.
`#`	The hashmark at the start of a line is a way to comment your code. This directs R to ignore the rest of the current line. Commenting your good is very good practice and is also used lots to remove commands that you may use in debugging a more complex workflow.
`print()`	Is the print function which writes out the value of the variable. Simply typing the name of the variable will also write out the value contained within, as seen in the examples above: `x` and `print(x)`

2.2 Best practise and variable naming conventions

Some quick best practice tips:

- Best practice suggests that you should not assign meaningless names to variables; give a variable a succinct but meaningful name that will be of value to understand which information you have saved.
  - `df` may be an obvious name for a data.frame, but `phenotypeDf` or `phenotype.df` may be more memorable later on in your analysis.
  - you can use **camelCase** (or camelNotation), underscore (_) or fullstop (.) for variable names in R.
- You can create thousands of variables in your R workspace but are they all necessary? It is worthwhile performing housekeeping on your data collection if you are creating many variables.
- If you have many variables in your R workspace then you may not be using the most ideal method to store your data. Consider data structures like `data.frame` and `lists` to better structure your data.
- RStudio allows you to remove all objects from your workspace by using the **Clear** button in the **Environment** tab as an alternative to the command: `rm(list=ls())`. (You can follow this with the `gc()` function for *garbage collection* to further ensure that memory has been freed up.)

2.3 Data types

In common with many programming languages, every variable in R is assigned a **data type**. The primary data types in R are `Numeric`, `Integer`, `Character`, `Logical`, and data structures, which will be covered in Section 2.6. The type of a variable determines what functions can be applied to it.

2.3.1 Numeric and Integer

Numerics in R store decimal values. **Integers** are a separate type to numerics and contain *whole number* values only. By default R will store numbers as `numeric`, since most mathematical operations will require decimal point numbers (such as division).

In the previous section we declared a variable called `x`, to which we assigned the value 42. R recognised this as a number and automatically set the type of `x` to be numeric.

The following table shows the types of numeric and integer operations that can be performed:

Operator	Description	Example	Result
+	Addition	`x + 2`	44
-	Subtraction	`x-20`	22
	Multiplication	`x*2.23`	93.66
/	Division	`x/3.5`	12
%/%	Integer division	`x %% 3.5`	0
^	Exponential	`x^2`	1764
%%	Modulus	`x%%2`	0

In addition to the basic operations, there are more complex numeric functions, the following table shows some common examples.

Given `y <- 423.2332` and `z <- -2.34`.

Operator	Description	Result
abs(z)	absolute value	2.34
sqrt(y)	square root	2.34
ceiling(y)	round up to nearest integer	2.34
floor(y)	round down to nearest integer	2.34
round(y,digits=2)	round to numer of n decimal points	2.34
log(y)	natural logarithm	2.34
log10(y)	common logarithm	2.34

2.3.2 Character

A **character** variable is used to represent strings or text values in R. Characters are identified by the surrounding double (" ")or single quotes ('').

```
h <- "Hello world"
print(h)
```

```
## [1] "Hello world"
```

The following table shows some types of character operations that can be performed. Given the example `h <- "Hello world"`.

Operator	Description	Example	Result
`nchar(h)`	number of characters in variable *x*	11	
`substr(x,start,stop)`	extract substring from start position to stop position	`substr(h,2,4)`	ell
`grep(pattern,x)`	search for *pattern* in variable *x*	`grep('ell',h)`	1
`grepl()`	provides a logical response as to whether the pattern exists.	TRUE	
`sub(pattern,rep,x)`	search for *pattern* and replace with *rep* in variable *x*	`sub('ello','i',h)`	Hi world
`strsplit(x,delim)`	split the elements of a character variable *x* using the specified *delim*	`strsplit(h,'o')`	c("Hell", " w","rld")
`paste(...,sep)`	concatenate list of strings together (...) separating them using the *sep* delimiter	`paste('a','b,'c',sep=',')`	a,b,c
`toupper(x)`	change to uppercase	`toupper(h)`	HELLO WORLD
`tolower(y)`	chagne to lowercase	`tolower(h)`	hello world

Some examples:

```
hello <- paste("Hello", "World", sep="~")
print(hello)
nchar(hello)
substr(hello, 3, 7)
strsplit(hello,'~')
```

```
## [1] "Hello~World"
## [1] 11
## [1] "llo~W"
## [[1]]
## [1] "Hello" "World"
```

```
gsub("World", "R", hello)
grep("Hello", hello)
grep("Mouse", hello)
grepl("Stephen", hello)
```

```
## [1] "Hello~R"
## [1] 1
## integer(0)
## [1] FALSE
```

2.3.3 Logical

Logical datatypes can have one of only two values: `TRUE` or `FALSE`. **Note**, these are case-specific, `True` or `true` are not valid logical values. However, the shortcut: `T` and ‘`F` will work.

```
x <- 42
is.even <- x %% 2
is.even
```

```
## [1] 0
```

Setting, retrieving and changing data types

1. Enter the examples from the sections above (numerics to logical) in RStudio. Either into the console or the editor window.
   - Remember, you can use the *Environments* tab to keep track of the variables being created.
   - **Note**, below all the outputs are shown altogether for readability. If you are running them in console, you will see the output immediately as you hit [Enter].
2. Experiment with different values so you understand better how different operations work.

2.4 Determining data type

Sometimes during a long session of analysing data, we might forget to which data type a variable belongs. We can find out the data type directly by using the `class()` function. Alternatively, we test whether a variable is a `numeric`, `integer`, `character` or `logical` using the `is.XXXXX()` function, e.g. `is.numeric(x)`. See the following example:

```
x <- 42
y <- 423.2332
h <- "Hello world"
is.even <- x %% 2

class(is.even)

is.numeric(x)
is.integer(y)
is.character(h)
is.character(y)
is.logical(x)
is.logical(is.even)
```

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

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

2.4.1 Typecasting

If we want to perform integer functions on `x`, we can force it from one type to another, in this case using the `as.integer()` function. This is known as **typecasting**. The `as.integer()` function also forces non-whole numbers into integers by rounding them down to the nearest whole number.

```
xInt <- as.integer(x)
is.integer(xInt)
class(xInt)

as.integer(is.even)
```

```
## [1] TRUE
## [1] "integer"
## [1] 0
```

However, there are times when type casting will fail.

```
age <- "54 years old"
age <- as.integer(age)
```

```
## Warning: NAs introduced by coercion
```

```
age
```

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

Since the variable `age` holds the characters "years old" it cannot be type cast to a number. Instead, R will provide a warning message as seen above, and assign the value `NA` to the variable `age`.

Simple datatypes

1. Now try typecasting `FALSE` to an integer.
2. Next try typecasting in the reverse, that is, converting the digits (1, 0) into TRUE or FALSE. *Hint*: `as.logical()`
3. [**Optional**] If you are ahead, have a look at typecasting other values (e.g. >1 and <0) into logical types. Can you see what is happening?

Writing to variables or the screen

- In the example above, you will notice that the line `y <- as.integer(x)` generates no output, while `as.integer(3.1415)` prints a result to screen. This is because in the first case, the output of `as.integer()` is passed into the variable `y`, while in the second case, no destination is given so by default it prints to screen.
- This also means that the output from the first command can be accessed and used again later by calling `y` but the output from the second is lost and cannot be used by future R commands.

2.5 Functions

Before we move onto data structures in the next section, we will take an aside to look at **functions**. Functions are a group of commands that perform a specific action. By itself it is not a complete executable program but form part of a larger workflow. Like variables that hold data, functions are also given a name so that they can be *reused*. They generally follow the *input* → *process* → *output* model, that is, they take input, do something with the input and produce output. The output can either be printed to the screen or be *returned* and assigned to a variable that holds the new data for further reuse.

So far we have already started using some of the in-built functions in R like `print()`, `str()`, `as.integer()` etc. They take the form of `output_variable <- function_name(input_variable)` where the output_variable is optional. If not output_variable is provided, by default the output from the funciton will be printed to the screen.

2.5.1 Nesting functions

Other than assigning the output of a function to a variable, functions can also be **nested** within other functions. For example, the following code performs an operation and prints the result to the screen:

```
length <- 42
width <- 10
area <- length*width
result <- paste("area of rectangle with length=",length,
                ", width=", width, "is", area)
print(result)
```

can be collapsed to:

```
print(paste("area of rectangle with length=",length,
            ", width=", width, "is", length*width))
```

```
## [1] "area of rectangle with length= 42 , width= 10 is 420"
```

Just like in mathematics, the **order of operations** (or *operator precedence*) is the same in R. Operations are evaluated (or executed) from the innermost brackets first. So in the command above, the inner `c("A","B","C")` is executed first and the output (which is held temporarily in memory) is then passed to the outer function `paste()`.

With this shortcut, you can reduce the resources that a program takes (not to mention the lines of code) and create very complex nested function calls. Keep an eye out for nested functions throughout the workshop.

Break it up first, then reassemble

When new to R and coming across a complex nested function call, the tip is to **break it up** into sections. Break the functions into smaller pieces *starting from the innermost function* and assign them to variables first. Execute each line, one at a time and examine the output. Then when you are comfortable with what each line is doing, put the pieces together again. This will help you understand what each function is doing first before trying to decipher the entire line in one go. Once you become familiar with more functions, reading nested functions will become a breeze!

2.6 Data structures

We do not typically expect to work with atomic variables during an *-omics* analysis. To make the best use of R we want to use hundreds, thousands and millions of data points. We arrange these in one of a number of different data structures: vectors, lists, matrices and data frames.

2.6.1 Vectors

A **vector** is a group of components of the *same* type. Vectors are created using the `c()` function:

```
data <- c(2,3,34,2,43,234,2,342,43,423)
data
length(data)
```

```
##  [1]   2   3  34   2  43 234   2 342  43 423
## [1] 10
```

2.6.1.1 Indexing

Elements in vectors are accessed directly by their position, this is referred to as **indexing**. The first element starts at index 1.

You can access positions consecutively or by jumping around:

Operator	Description
`vector <- c()`	function (for *combine*) is used to join a number of single values together into a vector
`vector[i]`	return element at position i of a vector, positions start from 1
`vector[start:end]`	slicing, get elements from *start* positon to the *end* position
`vector[c(3,5,10)]`	retrieves positions 3, 5, 10 of the vector
`vector[-i]`	return all elements except position i
`length(vector)`	returns the number of elments in the vector

```
data[3:7]
```

```
## [1]  34   2  43 234   2
```

```
data[c(3,5,9)]
```

```
## [1] 34 43 43
```

You can also specify which elements **not** to return by using the `-` sign infront of the index.

```
data[-2]
```

```
## [1]   2  34   2  43 234   2 342  43 423
```

```
data[-c(2,4,8)]
```

```
## [1]   2  34  43 234   2  43 423
```

Note that the above does not delete the elements from the original `data` variable, it just does not show them to screen. If you want to save a copy of the vector without the specified positions then you need to assign it to a new variable:

```
new.data <- data[-c(2,4,8)]
```

2.6.1.2 Mixed vectors

What happens if we try to mix data types in a vector? See the following worked example:

```
mixedVector <- c(shape = "rectangle",
                 width = as.integer(42),
                 length= 3.25)
mixedVector
```

```
##       shape       width      length
## "rectangle"        "42"      "3.25"
```

Above, the variable *mixedVector* holds three elements of different data types, however a vector must consists of the *same* data type. So all elements are automatically cast to the most robust datatype which is a `character`.

The `str()` is a function that describes the structure of an variable:

```
str(mixedVector)
```

```
##  Named chr [1:3] "rectangle" "42" "3.25"
##  - attr(*, "names")= chr [1:3] "shape" "width" "length"
```

Furthermore, in this example, each element has an associated name. We can access the element using its name instead of the position:

```
mixedVector['length']
```

```
## length
## "3.25"
```

2.6.1.3 Combining vector variables

`c()` can be used to combine vectors, not just individual values:

```
c(data, mixedVector)
```

```
##
##         "2"         "3"        "34"         "2"        "43"       "234"
##                                                      shape       width
##         "2"       "342"        "43"       "423" "rectangle"        "42"
##      length
##      "3.25"
```

Using vectors

1. Enter the chunks above into your RStudio interface and review the output.
2. After executing the last code block, check on the values that are in variables `data` and `mixedVector` again.
   a) How many elements does each variable hold? _____
   b) Do any of them hold 6 elements? _____
   c) What do you need to do to retain the combined vector with 6 elements?

3. Access the `mixedVector` element by name using `mixedVector['shape']`. What is the output? _____

Challenge, continue if you are ahead

4. Now enter the next chunk of code below and review the output.
   a) Can you determine the difference between `mixedVector[2]` and `mixedVector[[2]]`?
5. What would the equivalent command to `mixedVector[[2]]` be, if you were accessing the vector by name?
6. Does the following command: `mixedVector['length']*mixedVector['width']` work? Why?

Access some elements of mixedVector by position or name:

```
mixedVector[2]
mixedVector[[2]]
mixedVector['shape']
names(mixedVector)
```

2.6.2 Lists

Lists are similar to vectors in that they are one-dimensional structures for storing data and like vectors can be accessed by position or namespace. They differ from vectors in that they can contain multiple *different* data types. This makes them more flexible for storing data but more limited in the analyses that can be performed on them. Lists are created using the `list()` command, which has a similar format to `c()`.

```
mixedList <- list(shape="rectangle", width=as.integer(42), length=3.25)
str(mixedList)
```

```
## List of 3
##  $ shape : chr "rectangle"
##  $ width : int 42
##  $ length: num 3.25
```

Checking for membership

`%in%` is a way to test the membership of a single element in a list or vector of items. For example, try typing in the command: `"rectangle" %in% mixedList`

Like vectors, elements of a lists can be accessed by their position or their name. The name can also be accessed using the `$` notation. That is, the two lines below are eqivalent.

```
mixedList$shape
```

```
## [1] "rectangle"
```

```
mixedList['shape']
```

```
## $shape
## [1] "rectangle"
```

```
mixedList[['shape']]
```

```
## [1] "rectangle"
```

The `$` notation is only accessible by `Lists` and `Data frames` (Section 2.6.6).

2.6.3 Changing Vectors and Lists

While a list or vector is useful, their utility is much greater when we can add or remove elements from them, or change the value of existing elements.

```
# Add items to list
mixedList <- append(mixedList, c(units='mm'))
mixedList <- append(mixedList, c(area=mixedList$length * mixedList$width))

# Display the names of the lists
names(mixedList)
```

```
## [1] "shape"  "width"  "length" "units"  "area"
```

```
# Display just the second element of the list
mixedList[2]
```

```
## $width
## [1] 42
```

```
# Change the value of the 'units' item
mixedList$units <- 'cm'

# Remove the 'area' item from the list
mixedList$area <- NULL
```

Using Lists

1. Enter the two `mixedList` code chunks into RStudio and review the output.
2. Try and retrieve different elements of the list
   a) Can you get the second **and** fourth elements? *Hint:* use the combine `c()` function when indexing
   b) What about everything *BUT* the second and fourth?
3. [**Optional**] If you are ahead, now try to getting a range of values, that is get the second to the forth elements. *Hint:* try using the colon (:) notation to specify a range.

2.6.4 Factors

Factors are vector objects that contain grouping (classification) information of its components. Functionally they are similar to vectors in that they are a collection of objects of a *single* type. Factors are best applied when there are a limited number of different values. They are often used when categorical values are used in modelling or presenting data, e.g. phenotypic or study class data, such as diabetic, pre-diabetic, healthy.

For the example that we explore in the following exercise, we are going to consider a massive bucket of M&Ms. Assuming that the different colours are randomly mixed with an equal probability we would like to explore a million different sweets.

Using Factors

Use the code below to create a factorised `mandm` vector containing 1 million M&Ms of various colours.

Note: the output is again grouped together at the end for readability. The table below explains the new functions and what they mean.

Create some colours and randomly select a million samples from this list of colours using the `sample()` function, with replacement (meaning duplicate) colours are allowed.

```
colours <- c("red", "yellow", "green", "blue", "orange")
mandm   <- sample(colours, 1000000, replace=TRUE)

object.size(mandm)
length(mandm)
```

```
## 8000280 bytes
## [1] 1000000
```

```
# convert this into a factor datatype
mandm <- as.factor(mandm)
str(mandm)
object.size(mandm)
```

```
##  Factor w/ 5 levels "blue","green",..: 2 3 1 2 3 4 3 5 1 2 ...
## 4000688 bytes
```

`table()` is a very handy method to return the frequency counts:

```
table(mandm)
```

```
## mandm
##   blue  green orange    red yellow
## 200402 199827 200061 199679 200031
```

We can also ask questions like *which positions are 'blue'?*. This is returned by the `which()` function.

```
head(which(mandm == "blue"))
```

```
## [1]  3  9 11 13 19 30
```

```
length(which(mandm == "blue"))
```

```
## [1] 200402
```

What is happening here?

Function	Description
`sample()`	is a really useful method that allows you to sample elements from your data collection. The replace variable defined whether a value that is sampled should be replaced following its selection.

Function	Description
`object.size()`	describes the amount of memory that an object is using - this can be useful to identify really humungous objects that can be cleaned from your workspace.
`length()`	describes the number of elements present in the vector.
`table()`	prepares a tabular summary of values and the number of times that they occur - this can be used to summarise data effectively.
`head()`	shows only the first `n` elements of the vector, here we specified show the top 10 elements.
`which()`	returns a vector of the positions that means the condition, in this example which of the 1 million mandm's are 'white'? Since there are none, this returns an empty vector, represented by `interger(0)`.
`length(which())`	nests the two functions together. It performs the inner `which()` function first, then passes the result to the `length()` function. As the names of the function suggest, this is finding out how many elements in `mandm` are equal to 'blue'?

The real challenge with factors is not creating them or using them but adding novel content to them, at least in terms of new categories. If we find an M&M that is not one of our five original colours, we can't just change the colour of that entry in our vector, we need to specifically add that colour to our factor list first. As a result, factors are suited more for *immutable* or static content. Other data structures provide simpler mechanisms for data manipulation.

Adding factor values

1. Follow the code chunk below to try to change the colour of the first M&M to the value "white". The comments explain what each line is doing. You may find it best entering these commands directly in the console rather than the editor window, so you can see what is happening at each stage.
2. Add another colour option (e.g. brown) and change one or more of the vector elements to that new colour (e.g. elements from position 100 to 200)
3. **Challenge** add yet another colour ("purple") and this time *randomly* change 20 elements to the new colour purple.

```
# First, just try to change the first element to white
mandm[1] <- "white"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "white"): invalid factor level,
## NA generated
```

```
# Check the factor values, have we successfully added white?
levels(mandm)

# Now explicitly add white as a factor value
levels(mandm) <- c(levels(mandm), "white")

# Try again to change the colour of the first M&M
mandm[1] <- "white"

# And check the factor values again
levels(mandm)
table(mandm)
```

2.6.5 Matrices

A **matrix** is a collection of data elements arranged in a two-dimensional rectangular layout, effectively a table. Similar to the vector a matrix can contain only a *single* type of data.

It is often easiest to create a matrix from a vector of data. The next exercise will show you how to do this, follow the code below to generate a variety of matrices. R cannot guess the dimension of the data so you should specify either the number of rows (`nrow`) or columns (`ncol`) that the final data should have.

Common matrix operations:

Operation	Description
`matrix()`	create matrix
`nrow()`	returns number of rows
`ncol()`	returns number of columns
`dim()`	returns the dimension

Creating a matrix

Note: this time the output follows *immediately* after each command, so that you can compare with what you get.

1. Enter the code below to generate a variety of matrices. *Hint:* Use the **Help** tab to search for `matrix` to find out more about the function and what the `nrow` and `byrow` options do.
2. Rather than printing the output directly to screen, store the matrix into a variable named `testMatrix`.

```
matrix(1:12)
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
```

```
matrix(1:12, nrow=4)
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
matrix(1:12, nrow=4, byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

```
letter.mat <- matrix(LETTERS[1:12], nrow=4, byrow=TRUE)
```

Some useful functions

- `LETTERS` is a built-in variable R that returns a vector of characters from A to Z.
- The colon (:) syntax is a quick way of creating a vector of sequential numbers from `start` to `end` e.g. `200:225` will return a vector from 200 to 225.
- The `seq(from,to,by)` function is also very handy when you need to create a regular sequence of numbers that jump by a certain interval, e.g. `seq(20, 50, 2)` will return a vector from 20 to 50, incrementing by 2. You can also do the reverse by entering `seq(50, 20, -2)`.

2.6.5.1 Indexing a matrix

To access the elements of a matrix you use the notation `matrix[row, col]`. Examples as follows:

```
letter.mat[2,]   # returns the 2nd row
letter.mat[,3]   # returns the 3rd column
letter.mat[2,3]  # returns the element at row=2 and column=3
letter.mat[,-2]  # returns all but the 2nd column
letter.mat[3:4,] # returns rows 3 to 4
```

2.6.6 Data frame

The **data frame** is to a matrix what a list is to a vector. Like a matrix it is a two dimensional table for storing data and is made up of equal length rows and columns, but like a list, the columns can contain data of *different* data types. Although within a column, the data must all be of the *same* type.

Common operations for data frames:

Operation	Description
`data.frame()`	creates a data.frame
`nrow()`	returns number of rows
`ncol()`	returns number of columns
`dim()`	returns the dimension
`colnames(x)`	returns the column names of a data frame
`colnames(x) <- c('a','b','c')`	sets the column names for a data frame
`rownames(x)`	returns the rownames of a data frame
`rownames(x) <- c('p1','p2','p3')`	sets the rownames for a data frame

Follow the code chunk below to start exploring data frames. The comments explain what is happening in each stage. The first step tries creating a data frame using the same format as creating a matrix. This doesn't work though, see if you can work out what is actually happening here.

```
# Try creating a data frame using the same terminology as a matrix
data.frame(c(LETTERS[1:12]),nrow=4, byrow=TRUE)

# Now build a data frame from a temporary matrix
as.data.frame(matrix(c(LETTERS[1:12]),nrow=4, byrow=TRUE))
exampleDf <- as.data.frame(matrix(c(LETTERS[1:12]),nrow=4, byrow=TRUE))

# It's often useful to name the columns and rows of a data frame (or matrix)
colnames(exampleDf) <- c("x", "y", "z")
rownames(exampleDf)

# What is the size and structure of our data frame
dim(exampleDf)  # Dimension of the data frame
ncol(exampleDf) # Number of columns
nrow(exampleDf) # Number of rows
str(exampleDf)  # Structure information
```

2.6.6.1 Indexing a data frame

Indexing is the same as matrix using the syntax `[row,col]`:

```
exampleDf[,1]

# Then the contents of the first row
exampleDf[1,]

# Finally, add a fourth column of a different type to the data frame
exampleDf[,4] <- c(1,2,3,4)
exampleDf
class(exampleDf[,3])
class(exampleDf[,4])
```

Just like *lists*, if you have column names in your data frame, you can use the `$` notation:

```
colnames(exampleDf)
```

```
## [1] "x"  "y"  "z"  "V4"
```

```
exampleDf$x
```

```
## [1] A D G J
## Levels: A D G J
```

2.7 Finding, describing and removing objects

In the previous data types section we have created a couple of lists, integers, numerics and a data.frame. This is the beginning of a data analysis workflow. While you are typing at your keyboard it is easy to become distracted and to forget the name of the variable that you created, or to create many temporary variables that you no longer need. Following are a few helpful functions you will commonly use during data transformation and analysis.

Tidying up objects

1. Use the `ls()` function to list all the variables in your workspace.
2. Use `rm()` to get rid of a variable or variables that you no longer want.
   - To remove multiple objects, just enter the names of all those objects separated by commas
3. Delete everything from your workspace using `rm(list=ls())`
4. Review your history (all the commands you have entered into the console in this session) using the RStudio History tab (top right window).

```
ls()
rm(h)
rm(x, y)
rm(list=ls())
history()
```

Chapter 3

File and data input/output (IO)

It is likely that you will not always be able to complete your R analysis in a single sitting. Rather than having to recreate all your variables each time, there are various ways that you can save data for reuse. This section covers how to save your session data, importing and exporting data.

3.1 Saving variable data

If you started a new Project for your work, at the end of the day you can select **File** > **Close Project**. This will save all of your variables, your history, your open files and a range of other information. When you then re-open R, you can double click on the project file (which will have the suffix **.Rproj**) to continue where you left off.

As well as saving your entire project, you can save just specified variables, using the `save()` command. The save function saves the data as a binary object and the file prefix generally used for R binary objects is ".Rdata".

In the following worked example, we will save the `mandm` variable that was created in the previous chapter. We then remove this variable from our workspace using the `rm()` function and check that it has been removed using the `exists()` function.

```
colours <- c("red", "yellow", "green", "blue", "orange")
mandm   <- sample(colours, 1000000, replace=TRUE)
save(mandm,file="chocolate.Rdata")

# Delete 'mandm' from the workspace
rm(mandm)

# Comfirm it is deleted
exists('mandm')
```

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

3.2 Loading variable data

To load in an Rdata (binary) object back into your workspace use the `load()` function. You will need to provide the path to the filename.

```
load("chocolate.Rdata")

exists('mandm')
```

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

```
class(mandm)
```

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

```
dim(mandm)
```

```
## NULL
```

```
frequency <- table(mandm)
```

In the following example, we are saving multiple variables to one file:

```
save(mandm,frequency,file = "chocolate.Rdata")
```

Saving and loading

Enter the code blocks above for saving and loading data. Remember to remove the variable after saving it so that you can test the variable has been loaded properly.

Try the following chucnk of code after saving multiple variables:

```
rm(mandm, frequency)
frequency
load("chocolate.Rdata")
frequency
```

3.3 Current working directory

An aside about directory paths. When working with R, you need to be aware of your **current working directory** (CWD). As the name suggests, this is the directory (folder) in which you are currently working. If no paths are specified, any files you wish to import or export to and from R will be with respect to this directory.

The *Files* tab in the bottom-right corner is *not* always set to your CWD. Think of this as a Windows explorer (or Finder in Mac) built-in with Rstudio. This tab allows you to navigate your file system and check on files. This means that during your session, you may navigate away from your CWD.

To check your CWD use the command:

```
getwd()
```

and to change your CWD to another location use the `setwd(file-path)` command.

When using `save()` it is better to explicitly specifing the path to the folder where you want to save your data to so that no mistakes are made when you move away from your CWD.

3.3.1 File paths

File paths come in two forms:

- **relative filepath** is relative to the current working directory. For example, using a Windows system, if my currently working directory is `C:\Users\john.smith\Documents` and I save a script called "area-rectangle.R" without specifying a specific location, then R will save the file in the location `C:\Users\john.smith\Documents\area-rectangle.R`.
- **absolute filepath** is the fullpath of the location starting with the drive letter in Windows (e.g. C: or D:); or in Linux and Mac starting with the \ symbol, meaning https://aarnet.zoom.us/j/579276844the *root* directory (the top).

R uses the forward slash (/) to specify filepaths by default. If you use the Windows format with the backwards slash (\\), you will get an error. You can use two backward slashes (\\) to escape the error but it's faster and easier to read to use the / slash.

3.4 Importing from text files

Data is most commonly shared in tabular formats (Excel presents data in a tabular format). These are most-often text files that contain columns of data separated with comma, tab or other field-delimiters. In R there are some simple methods for quickly reading in massive amounts of data.

For the rest of this workshop we are going to work with a public dataset measuring the daily minimum and maximum temperatures across a number of cities in Australia. The data has been measured by 112 stations across Australia. The input file is located in folder: `data/Intro_to_R/stations.txt`.

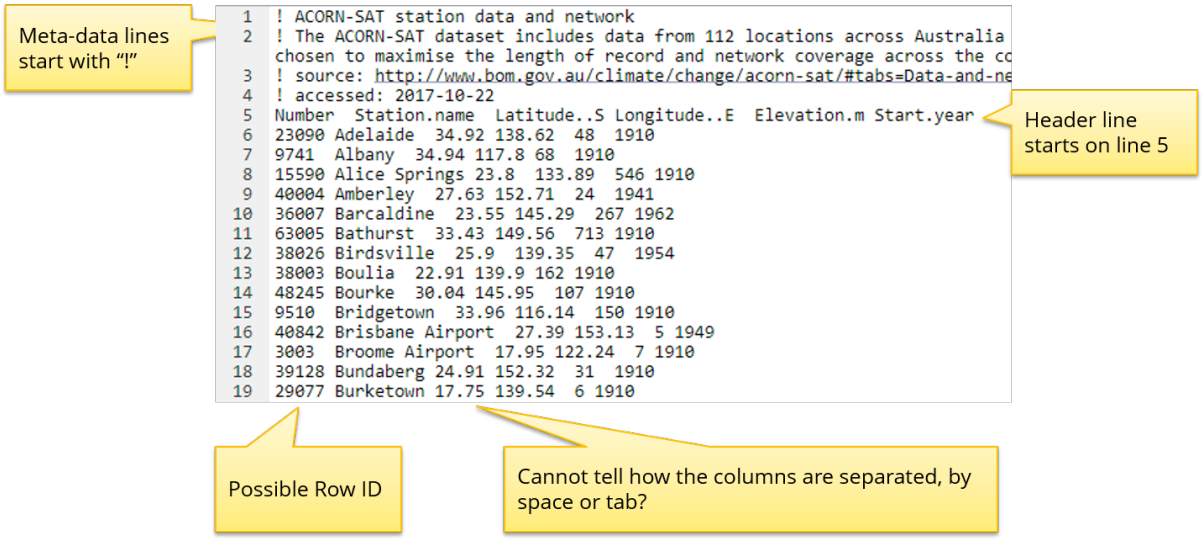


Figure 3.1: Stations.txt file

```
data.path <- "data/Intro_to_R"

# Of course, you will need a different path if the data is on your local P.C.
# e.g. data.path <- "C:/Intro_to_R/Data"
# Create a variable pointing to the file location.

stations.file <- file.path(data.path, "stations.txt")

# Then look at the start and finish of that file
head(readLines(stations.file))
tail(readLines(stations.file))
```

What is happening here?

Function	Description
`readLines()`	reads each line from a target file into a vector. See also `read.table`, `read.csv` functions.
`file.path()`	*DOES NOT* read in the contents of a file. It just point to the location of a files on your system. You need to use `readLines`, `read.table` or similar to access the contents of that file.
`head()`	show first 6 lines of the object
`tail()`	show last 6 lines of object

Using the `head()` and `tail()` commands helps us understand the nature of the file. It seems that the `#` prefix indicates metadata rather than data and the file is tab separated indicated by the ‘’ symbols.

Beware of printing large variables with RStudio Server, you can freeze your session. Judiciously use methods to limit the output:

- read into a variable first
- limit the output
- use `head()` / `tail()` functions

We have started to explore the contents of the expression data file using the code above, but at the moment **have not imported** it into a variable in R. The following exercise will step you through using the `read.table()` function in R to read a text file and import the data as a data.frame object.

When reading a table, R expects the same number of columns for every row, it will by default use the first line to determine the number of columns that are in your dataset. You can specify the separater to use depending on how your data columns are separated e.g. tab-delimited, comma-delimited or by something else altogether.

Reading in tabular data

1. Enter the previous code chunk to set up your `data.path` and take an initial look at the expression data in the variable `stations.file`.
2. Open up the help documentation for `read.table()` function while you perform the following worked example. Read up on the following parameters in detail:

- `sep`
- `comment`
- `header`
- `row.names`

3. Follow the code below to use `read.table()` to import the dataset into R. There should be 112 rows (stations) and 5 columns (variables), use the station number as the rownames since it will be unique.

Pay close attention to the dimension of the dataset after each read as each parameter will affect the read.

Attempt 1:: read.table using default settings:

This will fail because by default the `sep` parameter separates on white-space and the first X lines in the file are all comments, each line does not have the same number of columns as indicated by the error message.

```
stations <- read.table(stations.file)
```

```
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, : line 1
did not have 42 elements
```

Attempt 2:: skip comment lines

We specify the separating delimiter (`sep='\t'`), the comment character ('!') and check the dimension of the data object after reading in the file. Is the number of rows and columns correct?

```
stations <- read.table(stations.file, comment.char = "!", sep='\t')
dim(stations)
```

```
## [1] 113   6
```

We have 1 too many rows and columns. Taking a physical look a the object will tell us why. It is because we haven't assigned the row or column names to the data frame.

```
head(stations)
```

```
##       V1            V2          V3           V4          V5         V6
## 1 Number  Station.name Latitude..S Longitude..E Elevation.m Start.year
## 2  23090      Adelaide       34.92       138.62          48       1910
## 3   9741        Albany       34.94        117.8          68       1910
## 4  15590 Alice Springs        23.8       133.89         546       1910
## 5  40004      Amberley       27.63       152.71          24       1941
## 6  36007    Barcaldine       23.55       145.29         267       1962
```

Attempt 3:: assign header and row names

```
stations <- read.table(stations.file, comment.char="!", sep="\t",
                       header=TRUE, row.names = 1)
dim(stations)
```

```
## [1] 112   5
```

```
head(stations)
```

```
##        Station.name Latitude..S Longitude..E Elevation.m Start.year
## 23090      Adelaide       34.92       138.62          48       1910
## 9741         Albany       34.94       117.80          68       1910
## 15590 Alice Springs       23.80       133.89         546       1910
## 40004      Amberley       27.63       152.71          24       1941
## 36007    Barcaldine       23.55       145.29         267       1962
```

```
## 63005       Bathurst       33.43       149.56         713       1910
```

Much better, this looks right!

Let's save the `stations` variable as a Rdata object so that we don't have to read it in again.

```
save(stations, file='stations.Rdata')
```

Import temperature dataset

Now that you have imported the stations metadata, let's import the actual temperatures for one station across the last 69 years.

Read in the minimum temperatures file: `data/Intro_to_R/minTEmp.040842.csv` into a variable named `minTemp`.

Use the first column (date) as the rownames, there should be 366 rows and 69 variables.

Expected output:

```
##       X1949 X1950 X1951 X1952 X1953
## Jan-1    NA  22.3  19.0  22.4  18.3
## Jan-2    NA  19.3  16.3  20.8  20.8
## Jan-3    NA  23.1  15.4  23.4  16.3
## Jan-4    NA  22.5  15.1  23.1  17.5
## Jan-5    NA  22.4  14.6  20.3  19.8
```

We will be working with this dataset in the following chapter.

3.5 Writing to text file

You can also write tabular data to text files. Let's say we are only interested in keep track of the station number and their name, so we will write this to a text file:

```
write.table(stations[,"Station.name"], file="stationName.tsv", row.names = T,
            col.names = F, quote=F, sep="\t")
```

What is happening here * `file` - specifies the output file path, we are using the current working directory * `row.names` - specifies the row names are written to the file * `col.names` - specifies no column names are written to the file * `quote` - specifies do not use quotation symbols in the output * `sep` - specifies the character to use for separating the columns, in our case we are using tabs

Enter the `write.table()` command above and then examine the output file. You should be able to click on the filename in the "Files" tab to view the file.

Then experiment with different parameter settings, you can keep the output file open to observe how the output changes with each parameter change. For example, try

- `row.names=F`
- `quote=T`
- `sep=','`

If you are often dealing with comma-separated files then you can simple use the `read.csv()` and `write.csv()` functions. These functions have the parameter `sep` set to comma by default.

3.6 Other IO operations

3.6.1 Interactive input

`scan()` is a primitive method that can be used to import data from a variety of sources. In most instances it would be preferable to read data from file using `read.table` or `readLines`. scan is however magnificent for reading in data from the standard input or from other software streams. As a quick demonstration we can read in data as you type it:

```
scan(what=numeric())
```

3.6.2 Reading from a web connection

Most of the time we will be reading information into R that is stored on our local filesystem, but R can also import data directly from the web. The `read.table()` and `readLines()` functions are happy to read in from a web socket. This is great but requires that the data be present on a web-page to download.

For example, the following reads data from Google Trends github account on search results for "Fathers day". Based on the URL we can expect the file is a comma separated file because the filename as in "*.csv", however, we do not know whether there are any comments in the files, if the first line is the heading columns etc. So we will peak into the file first before reading it as a data object.

```
data.URL <- "https://raw.githubusercontent.com/GoogleTrends/data/gh-pages/
20150624_FathersDay.csv"
fathers.txt <- readLines(data.URL)
head(fathers.txt)
```

```
## Warning in readLines(data.URL): incomplete final line found on
## 'https://raw.githubusercontent.com/GoogleTrends/data/gh-pages/
## 20150624_FathersDay.csv'
## [1] "Search interest in dads in each country's respective father's day in 2014,,,,,"
## [2] "country,ISO Code,Date,Holiday,Indexed,Rank"
## [3] "Puerto Rico,PR,06/15/2014,Father's Day,100.00,1"
## [4] "Caribbean Netherlands,BQ,06/01/2014,Global Day of Parents,79.46,2"
## [5] "Curaçao,CW,06/15/2014,Father's Day,75.63,3"
## [6] "Aruba,AW,06/15/2014,Father's Day,74.62,4"
```

Now that we can see the first line is a description and the column headings are in line 2 we will read the data file into a data object:

```
fathers.day <- read.csv(data.URL, header=T, skip=1)
fathers.day[1:5,1:6]
```

```
##                 country ISO.Code       Date               Holiday Indexed
## 1           Puerto Rico       PR 06/15/2014          Father's Day  100.00
## 2 Caribbean Netherlands       BQ 06/01/2014 Global Day of Parents   79.46
## 3               Curaçao       CW 06/15/2014          Father's Day   75.63
## 4                 Aruba       AW 06/15/2014          Father's Day   74.62
## 5                Guyana       GY 06/15/2014          Father's Day   67.42
##   Rank
## 1    1
```

```
## 2    2
## 3    3
## 4    4
## 5    5
```

– The `RCurl` package provides a much more flexible approach to accessing data that is on the web and is worth reviewing if you wish to scrape a web-accessible database in a more automated fashion.

Bioinformatics data

– Of greatest interest however is the ability to download pre-structured biological data from the web. This can be managed using packages such as `bioMart` and `GEOquery`.

Chapter 4

Simple data analysis

4.1 Simple Data cleaning

We have read in our temperature data from the Brisbane Airport station in the previous chapter. Before we do any simple statistical analysis in this chapter we need to first check on the data we uploaded. Data cleaning is an essential step prior to any data analysis, but is beyond the scope of this workshop. (Have a look at our Data preparation, processing and reporting with R workshop.)

One quick way to check our temperature data is the range of the dataset:

```
range(minTemp, na.rm=T)
```

```
## [1]    -2.2 99999.9
```

This tells us we have temperatures ranging from -2.2 degrees to 99999.9 degrees. Alarm bells should be going off in your head for the maximum range. Could this be an error in the reading or something else? Visiting the source of where the data is obtained from may provide more information.

The Bureau of Metorology site tells us that missing data is also encoded with 99999.9, given that this is the case let's change the data and use `NA` for missing data as the value 99999.9 will interfere when we perform simple summary statistics like calculating the average temperature.

```
# create a copy of the original dataset just in case
minTemp.ORI <- minTemp

minTemp[minTemp == 99999.9] <- NA
range(minTemp, na.rm=T)
```

```
## [1] -2.2 28.1
```

Recheck the range after any modification to the data, this is much more realistic. Now we can perform some basic summary functions on the dataset.

4.2 Summary statistics

Below are some other useful methods for a vector or list of numbers:

Operator	Description
`min(x)`	returns minimum value in list or vector
`max(x)`	returns the maximum value in list or vector
`sum(x)`	returns the total sum of a numeric list or vector
`mean(x)`	returns the mean value of list or vector
`sd(x)`	returns the standard deviation of list or vector

Operator	Description
`summary(x)`	returns basic statistic summary of vector
`which.min(x)`	returns the position of the list or vector with the minimum value
`which.max(x)`	returns the position of the list or vector with the maximum value
`range(data)`	returns the minimum and maximum range of a list/vector
`duplicated()`	provides a vector of logicals indicating which elements of a vector have already been seen in that vector. In other words, the first time a value is seen it will return FALSE, and then if it occurs again it will return TRUE)
`unique()`	provides a non-redundant list of all values in a vector; any duplicated values will be output only

We will use some of these functions to explore our temperature data, let's look at the temperature from year 1996:

Examples:

```
temp <- minTemp$X1996

# Find various statistical summaries of that test data. The semicolon
# separates commands on the same line.
min(temp)
max(temp)
mean(temp)
sd(temp)
range(temp)
```

```
## [1] 1.9
## [1] 25.9
## [1] 15.40874
## [1] 5.157272
## [1]  1.9 25.9
```

There is very handy function, `summary()` that returns the descriptive summary statitics for you given a vector of numbers:

```
summary(temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.90   11.70   16.25   15.41   19.10   25.90
```

```
# Find the position of the minimum and maximum values
which.min(temp)
which.max(temp)
```

```
## [1] 200
## [1] 32
```

```
head(duplicated(temp))   # only show the first 6 elements
length(unique(temp))     # number of unique values in `temp`
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
## [1] 171
```

4.2.1 Two-dimensional datasets

While the above can be used on matrices and data frames, the answer will always be a single number. There are times when you have a table of numbers and only want to operate on the rows or columns. This can be done using the following operations:

Operator	Description
`colSums(x)`	sum of column values
`rowSums(x)`	sum of row values
`colMeans(x)`	mean of column values
`rowMeans(x)`	mean of row values

This time instead of working on only one year, let's work on all 69 years worth of data:

- What is the average temperature for each year?
- What is the average temperature for each day?

```
head(colMeans(minTemp))
head(rowMeans(minTemp))
```

```
##    X1949    X1950    X1951    X1952    X1953    X1954
##       NA       NA       NA 15.34098       NA       NA
## Jan-1 Jan-2 Jan-3 Jan-4 Jan-5 Jan-6
##    NA    NA    NA    NA    NA    NA
```

Notice how there are alot of NAs being returned, this is because there are some missing values and R cannot calculate the mean when there are missing values. You can specify the parameter `na.rm=T` to remove the missing valuese (NA) before calculating the mean:

```
head(colMeans(minTemp, na.rm=T))
```

```
##    X1949    X1950    X1951    X1952    X1953    X1954
## 12.12783 15.35397 13.35623 15.34098 14.72247 15.82493
```

```
head(rowMeans(minTemp, na.rm=T))
```

```
##    Jan-1    Jan-2    Jan-3    Jan-4    Jan-5    Jan-6
## 20.53676 20.65441 20.89118 20.79265 20.82941 20.72941
```

Unfortunately there is no `colSd()` or `rowSd()` equivalent to finding the standard deviation of the columns or rows of a matrix. We will see how to do this later in the workshop.

Summary statistics

1. Try the example codes above.
2. Try using `summary()` on the the variable *minTemp*, what do you get?

Challenge if you are ahead try the following:

3. What is the lowest ever temperature recorded by this station? *Hint:* Remember to remove NA values.
4. Using the `which()` function, can you find the row and column position of the lowest recorded temperature? (*Hint:* Look at the help documentation for the `which()` function.)
   a. Using the row and column what is the year, month and date of this lowest temperature?

4.3 Vectorisation

The brilliance of using R for data analysis is the concept of **vectorisation**. Because we generally work with a table of values and want to perform the same operation again either on the whole table, a whole row or a whole column. Vectorisation means a function can be applied to every element of a vector or matrix without having to explicitly loop through each element of the whole variable.

For example, let's say we want to convert our temperatures from Celsius to Kelvin, we can do this simply by one line of code:

```
head(minTemp$X1996 + 273.15)
```

```
## [1] 296.55 296.65 297.85 290.95 293.55 293.65
```

Performing a vectorised function on a vector or matrix does not change the contents of that variable. To perform an in-place calculation, you need to explicitly overwrite the variable with the output of the function:

```
kelvinTemp <- minTemp + 273.15
```

Chapter 5

Packages and functions

5.1 Extending R with packages

One of the reason that R has such a following in the statistical and biological fields is the range of available packages that can be used to extend the utility of the software. **Packages** are collections of functions, data, and compiled code written in R and other languages that are encapsulated and distributed in the package format. Packages are generally domain or analysis specific, thus are only installed on a as needed basis. The directory where packages are stored is called the library.

R comes with a standard set of packages. Others are available for download and installation from repositories that include CRAN, Bioconductor and R-forge. To use a package, you must first install it and then load it into your session using either the require or library functions.

Because of the configuration of the training server we are using today, you can only install packages to *your own personal library* and not a system wide installed. This means that the packages you install will not be available for use by another participant in this class.

The code below outlines the process involved:

```
# Look to see which packages are already installed
library()

# To install one that is not there, use install.packages
# N.B. You will get a message "Would you like to use a personal library instead?"
install.packages("gplots")

# Once it's installed, we need to load it
require("gplots")
# or
library("gplots")

# library() shows the installed packages. search() shows which of those are loaded.
search()

# If you no longer need a package, then you may want to unload it with detach()
# You need to specify "package:" before the name, as per the listing from search()
detach("package:gplots")
```

Using packages

1. Use `library()` and `search()` to see what packages are available on the training server

and loaded into your workspace.

5.2 User defined functions

In this section we cover how to write your own **user-defined functions** (Section 5.2), which as the name suggests are functions you write yourself instead of using the built-in functions.

While there are hundreds of packages that can do everything that you need in R, it is often quicker to write a function to do something specific than it is to find someone else's function. For purposes of scientific reproducibility, for automation and for clean code, wrapping chunks of code into functions is a must!

Crafting a function is pretty simple. We use the `function()` command, and provide it with a name for our new function and some logic for that function to perform.

Below we create a very simple function called `addOrMinus`:

```
addOrMinus <- function(a, b=2, add=TRUE) {
  if (add) {          # If the logical add is TRUE, then carry out this section
    return(a + b)     # Add a and b and return the result
  } else {            # or if add is FALSE, then do this instead
    return(a - b)     # Subtract b from a and return that result
  }
}
```

What is happening here?

This function takes the following three **arguments**:

(i) a *required* numerical value named `a`,
(ii) an *optional* numerical value named `b` and
(iii) an *optional* logical value named `add`.

Arguments `b` and `add` are given **default values** of `2` and `TRUE` respectively by using the equal (=) sign in the function declaration. These defaults values are used in the logic of the function if no other values are provided when the function is called.

You call the function using its name and specifying the input:

```
addOrMinus(1)
```

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

```
addOrMinus(2, 4)
```

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

and you can assign the answer to a new variable:

```
answer <- addOrMinus(2, add=FALSE, b=4)
```

Below is another example calculating the hypotenuse of a triangle:

```
hypotenuse <- function(a,b) {
  return(sqrt(a^2+b^2))
}
hypotenuse(3,4)
```

What is happening here? This function takes two numbers and **returns** the square root of the sum of the squares of those two numbers.

Functions can contain any data-type as input, they can be simple data types (Section 2.3) or even the more complex data structures (Section 2.6). You can have a mixture of required and optional values. By convention, required arguments are listed first, then followed by any optional arguments.

Taking the above `hypotenuse` function, the second number b can become optional by specifying a default value, such as:

```
hypotenuse2 <- function(a,b=3){
  print(paste("a=",a,"and b=",b))
  return(sqrt(a^2+b^2))
}
hypotenuse2(3)
```

```
## [1] "a= 3 and b= 3"
## [1] 4.242641
```

```
hypotenuse2(234,32)
```

```
## [1] "a= 234 and b= 32"
## [1] 236.1779
```

Creating and using functions

1. Try the examples above to create and experiment with the two functions.
2. Create a function that will perform the temperature conversion celsius to fahrenheit: $F = \frac{9}{5}C + 32$. Call the function `celsius.to.fahrenheit(minTemp$X1996)` on temperatures from 1996 of our temperature data. Save this to a new variable `fahrenheit.1996`.
3. Create another function to perform the ooposite conversion, from fahrenheit to celsius: $C = (F - 32)\frac{5}{9}$. Call the function `fahrenheit.to.celsius(fahrenheit.1996)`, did you get the same results?
4. Now try running either `celsius.to.fahrenheit` or `fahrenheit.to.celsius` on the full matrix `minTemp`.

Expected output for 2: Only the first 6 results are shown

```
## [1] "Before conversion:"
## [1] 23.4 23.5 24.7 17.8 20.4 20.5
## [1] "Convert to fahrenheit:"
## [1] 74.12 74.30 76.46 64.04 68.72 68.90
```

5.3 Tips for functions

- Look on CRAN, Bioconductor or other repositories first to see if others have created the function you need. If so, test their function first.
- Like variables, give functions a meaningful and self-explanatory name.
- Keep functions simple and structured, that is, only do one thing.
- Use some form of indentation to keep your code readable so that you can follow what is happening. Have a look at the R coding conventions and links contained within at http://journal.rproject.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf.

- An explicit `return()` is not required in a function. By default a function will return the last executed command. However, it is worthwhile to include a return statement if you have a complicated function that has multiple endpoints and logical decisions. This also makes debugging easier.
  - You can also create and return complex data structures e.g. `return(list(...))`

If you think that writing a function is great, the next step will be to write whole packages that automate, streamline and focus your research. This is not difficult but unfortunately does not fit in the scope of this workshop.

Chapter 6

More data analysis

6.1 apply()

In Chapter 4, we considered the simplicity with which we could perform simple transformations on vectorised data. In these examples a discrete transformation was applied to each value.

If we consider our chunk of gene expression data in the `gene.atlas` variable, we can imagine a number of simple transformations that we might wish to perform such as log2 transformations. There are more cases when we might wish to perform an analysis by row or column within the data. While this could be managed using a for loop to iterate over the data there are a number of simpler ways to access the data. One way to do this is with `apply()`.

`apply()` takes at least three arguments:

- the matrix of input data,
- the MARGIN, whether to perform the calculation by row (1) or column (2) and
- the function to apply to the margin. The function can be a built in function or a user-defined function.

This function can be one that is predefined in R (such as `mean` or `sd` in the exercise below) or one you have created yourself as per the previous function section. In the latter case remember that the function should expect a vector as input and return a single value (e.g. it receives a vector of numbers and returns their mean).

The example below finds the standard deviation (`sd`) of the rows and columns of a randomly generated matrix with 1000 rows and 10 columns. **Remember** if you run this you will get different results because the numbers are randomly generated by using the `sample()` function.

```
dataset <- matrix(sample(10000:30000, 10000, replace=T), ncol=10)

# apply the standard deviation function on each row (MARGIN=1) of the matrix
rowSD <- apply(dataset, MARGIN=1, sd)
head(rowSD)                    # we only show the first 6 values using head()
```

```
## [1] 4873.865 6104.786 5356.709 5996.164 5256.596 6245.991
```

```
# apply the sd function on each column (MARGIN=2) of the matrix
colSD <- apply(dataset, MARGIN=2, sd)
head(colSD)
```

```
## [1] 5839.756 5937.252 5685.356 5746.754 5739.467 5740.089
```

You can also use other functions such as `range()` to find the minimum and maximum of each row/column of a matrix:

```
apply(dataset, MARGIN=2, range)
```

```
##       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
## [1,] 10034 10003 10009 10030 10020 10004 10044 10022 10002 10011
## [2,] 29985 29956 29955 29986 29946 29952 29988 29973 29947 29982
```

Using apply

Try the above examples using `apply()`, then repeat this on our `minTemp` dataset

Hint:

```
daily.SD <- apply(minTemp, MARGIN=1, sd, na.rm=T)
```

Remember we have missing values in our dataset, so we need to specify the `na.rm=T` parameter setting to remove NA values before calculating the standard deviation.

For more information, read the help documentation for `apply` to find out more about the paremeter settings.

Expected results only showing the first 6 elements

```
##    Jan-1    Jan-2    Jan-3    Jan-4    Jan-5    Jan-6
## 2.289750 2.301163 2.392509 2.512555 2.357019 2.487345
##    X1949    X1950    X1951    X1952    X1953    X1954
## 5.864297 4.804218 5.564952 5.720759 5.796398 4.825505
```

Optional exercises

Try the following if you are ahead:

1. Which year had the highest standard deviation? *Hint* save the results from the previous exercise to a variable and use `which.max()`
2. Which year had the lowest standard deviation?
3. Which day had the highest standard deviation across the 69 years?
4. Which day had the lowest standard deviation across the 69 years?

You can also use your own user-defined function in `apply()`. The comments in the following example explains each section:

```
# Create a user-define function that expects numeric data
# calculates the mean temperature before converting it to fahrenheit
# returns the single mean temperature as fahrenheit
mean.celsius.to.fahrenheit <- function(temp){
  mean.temp <- mean(temp,na.rm=T)
  return((9/5)*mean.temp+32)
}

# Call the conversion function to return the
# mean tempature per week (MARGIN=1 is rows)
head(apply(minTemp, MARGIN=1, mean.celsius.to.fahrenheit))
```

```
##    Jan-1    Jan-2    Jan-3    Jan-4    Jan-5    Jan-6
## 68.96618 69.17794 69.60412 69.42676 69.49294 69.31294
```

```
# Call the conversion function to return the
# mean temperature per day of the week (MARGIN=2 is cols)
head(apply(minTemp, MARGIN=2, mean.celsius.to.fahrenheit))
```

```
##    X1949    X1950    X1951    X1952    X1953    X1954
## 53.83009 59.63715 56.04122 59.61377 58.50044 60.48488
```

apply

- `apply` is a crucial method for exploring data within the rows and columns of a `data frame` or `matrix`.
- `lapply` is similar to apply but it returns results as a list rather than a matrix.
- There is also the `dplyr` package, which is even more powerful but is beyond the scope of this workshop. See https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

6.2 For and while loops

There are times when the `apply()` function may not be the most suitable, for example when you are performing actions that do not return data, such as plotting data or running proof-of-concept workflows. This is where `for` and `while` loops are used. However, if you are using them for situations like the examples we saw above, they may be slower than using the `apply` function.

To create a loop, we need to set a looping condition. There are two types in R:

1. a `for()` loop will run on each element of the input variable, and
2. a `while()` loop repeats until an exit condition is reached. **Note**, you must ensure that you update the exit condition so it eventually becomes true, otherwise you will get an **infinite looping** error where you code *never* exists until it uses up all available resources (usually the memory).

We also need to wrap that loop in braces (`{` and `}`) to define the start and end of the code block to be iterated.

6.2.1 For loops

Below is a simple example of a for loop, which prints the column name of every fifth column in our temperature dataset.

```
years <- colnames(minTemp) # get the column names

for(year in years[seq(1,length(years), 5)]) {
  print(year)
}
```

```
## [1] "X1949"
## [1] "X1954"
## [1] "X1959"
## [1] "X1964"
## [1] "X1969"
## [1] "X1974"
## [1] "X1979"
## [1] "X1984"
## [1] "X1989"
## [1] "X1994"
## [1] "X1999"
```

```
## [1] "X2004"
## [1] "X2009"
## [1] "X2014"
```

Something more complex, we loop through every 5th year in our temperature matrix and return the day with the lowest and highest temperature:

```
for (i in seq(1,ncol(minTemp),5)) {
  year    <- gsub("X", "", colnames(minTemp)[i])
  temps   <- minTemp[,i]
  min.day <- which.min(temps)
  max.day <- which.max(temps)
  print(paste(rownames(minTemp)[min.day],"had the lowest min temp &",
              rownames(minTemp)[max.day],"had the highest min temp in",year))
}
```

```
## [1] "Jul-20 had the lowest min temp & Dec-29 had the highest min temp in 1949"
## [1] "Jun-17 had the lowest min temp & Feb-1 had the highest min temp in 1954"
## [1] "Jun-27 had the lowest min temp & Jan-8 had the highest min temp in 1959"
## [1] "Aug-13 had the lowest min temp & Jan-14 had the highest min temp in 1964"
## [1] "Jul-23 had the lowest min temp & Feb-11 had the highest min temp in 1969"
## [1] "Jul-12 had the lowest min temp & Jan-8 had the highest min temp in 1974"
## [1] "Jul-2 had the lowest min temp & Dec-17 had the highest min temp in 1979"
## [1] "Jul-3 had the lowest min temp & Dec-16 had the highest min temp in 1984"
## [1] "Jul-22 had the lowest min temp & Mar-16 had the highest min temp in 1989"
## [1] "Aug-2 had the lowest min temp & Jan-9 had the highest min temp in 1994"
## [1] "Jun-18 had the lowest min temp & Jan-15 had the highest min temp in 1999"
## [1] "Jun-21 had the lowest min temp & Feb-22 had the highest min temp in 2004"
## [1] "Jun-12 had the lowest min temp & Jan-25 had the highest min temp in 2009"
## [1] "Jul-12 had the lowest min temp & Feb-20 had the highest min temp in 2014"
```

6.2.2 While loop

Below is a simple while loop that prints the Fibonacci series until the N^{th} number is divisible by 7:

```
#initialise the numbers
fib.prev <- 0
fib.curr <- 1
while (fib.curr %% 7 != 0) {
  print(fib.curr)

  # calculate new number
  new.curr <- fib.curr + fib.prev

  # update the pervious and current numbers
  fib.prev <- fib.curr
  fib.curr <- new.curr
}
```

```
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 8
## [1] 13
```

Going Loopy

- R objects are often vectors and can be analysed in bulk using normal mathematical and statistical transformations. This applies for the vectorised data in `data.frames` and `matrices`.
- Use `for` and `while` loops for testing ideas and exploring data, but try to avoid them if you can use `apply` (or its variation) to get the same result in production code.

`factors` cannot be used quite as easily in such analyses, so beware when operating on factors.

Chapter 7

Visualising data

In the previous sections we have looked at a number of ways to import data, access data, summarise data. We should now consider how we best present data in a graphical way. R is endowed with a number of native and add-on packages that facilitate the preparation of figures, diagrams and graphs. In this section we will explore plots prepared using the built-in plotting functions, which will give you a quick way to visualise your data.

The plots covered are

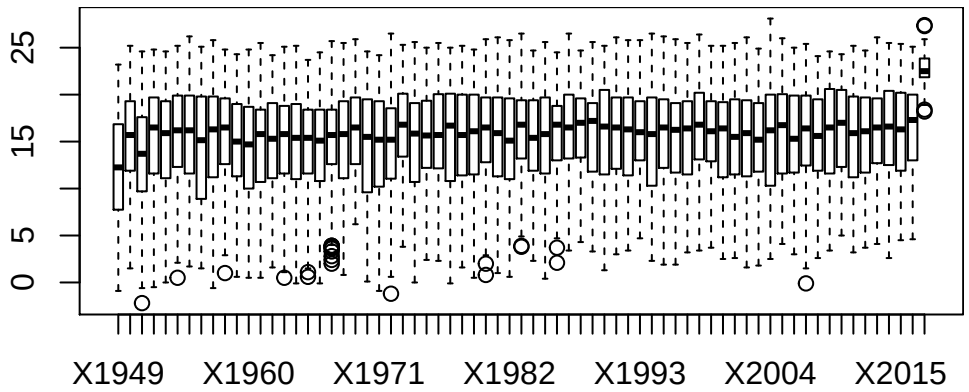
- boxplots
- histograms
- density
- scatterplots
- bar charts

7.1 Boxplots

The boxplot is a widely used plot that can summarise the distribution of data within a collection. We routinely use boxplots to show a trend within data during.

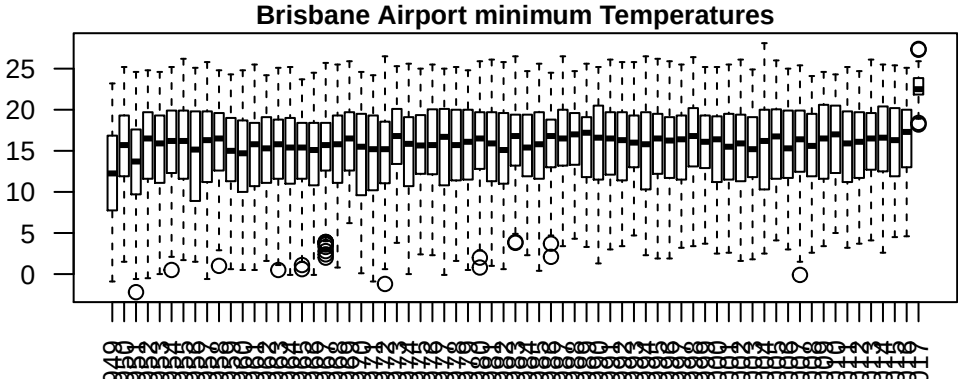
Boxplots are for numeric data only:

```
boxplot(minTemp)
```



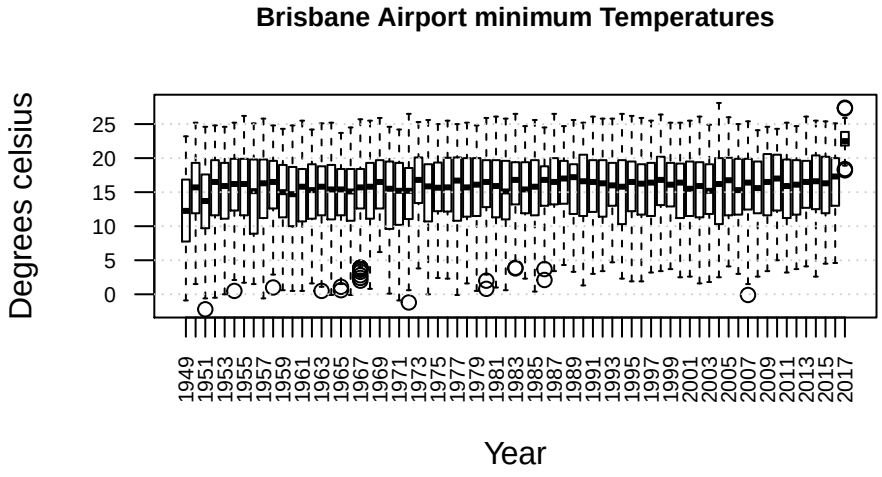
Let's add in the parameters to get a more sophisticated boxplot. It's a good idea to include one parameter at a time to see the effects of the new parameter. We will explain what each of the parameters mean during the worshop. Only the last figure in the following worked example is shown.

```
boxplot(minTemp, las=2)

boxplot(minTemp,
        las=2,
        main="Brisbane Airport minimum Temperatures",
        xlab="Year",
        ylab="Degrees celsius")
```



We will explain what each of the following lines mean during the workshop:

```
years <- gsub("X","",colnames(minTemp))
years[seq(2,length(years),2)] <- NA

boxplot(minTemp,
        las=2,
        main="Brisbane Airport minimum Temperatures",
        xlab="Year",
        ylab="Degrees celsius",
        cex.axis=0.7,
        cex.main=0.8,
        names=years)
abline(h=seq(0,25,5),lwd=1,lty=3,col='grey80')
```



Let's see if there are any seasonal trend in the data by plotting the temperature per season. To make this easier we will create a new variable called `season` that match each row in our `minTemp` dataset. We will explain what each of the following line means during the workshop.

```
months <- rownames(minTemp)
months <- substr(months,1,3)
months <- factor(months, levels=c('Jan','Feb','Mar','Apr','May','Jun',
                                  'Jul','Aug','Sep','Oct','Nov','Dec'))
levels(months)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

```
seasons <- months
levels(seasons) <- c(rep('Summer',2), rep('Autumn',3),
                     rep('Winter',3), rep('Spring',3), 'Summer')
levels(seasons)
```

```
## [1] "Summer" "Autumn" "Winter" "Spring"
```

```
table(months,seasons)
```

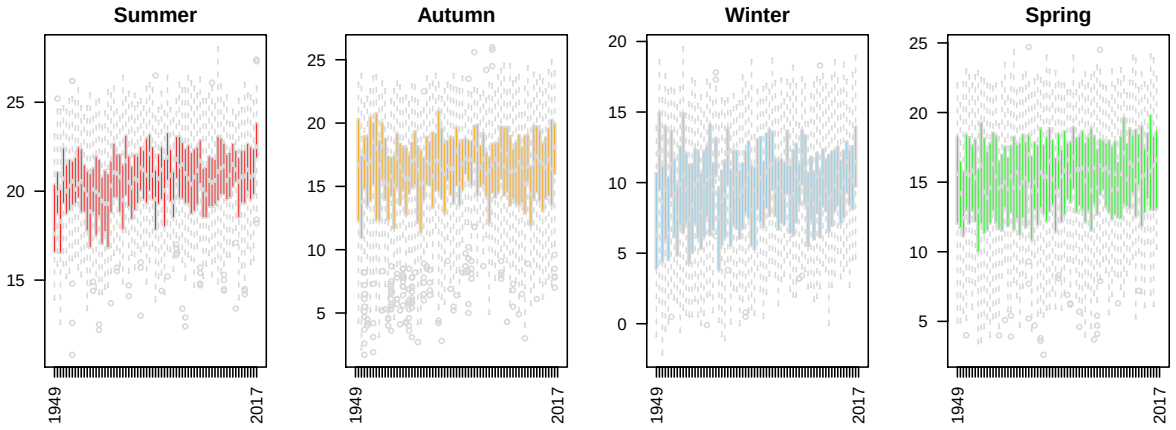
```
##       seasons
## months Summer Autumn Winter Spring
##    Jan     31      0      0      0
##    Feb     29      0      0      0
##    Mar      0     31      0      0
##    Apr      0     30      0      0
##    May      0     31      0      0
##    Jun      0      0     30      0
##    Jul      0      0     31      0
##    Aug      0      0     31      0
##    Sep      0      0      0     30
##    Oct      0      0      0     31
##    Nov      0      0      0     30
##    Dec     31      0      0      0
```

We can use a for loop to create separate plots for each season. To have all plots appear on one use the `par(mfrow)` function to specify a canvas with 1 row and 3 columns.

Other properties:

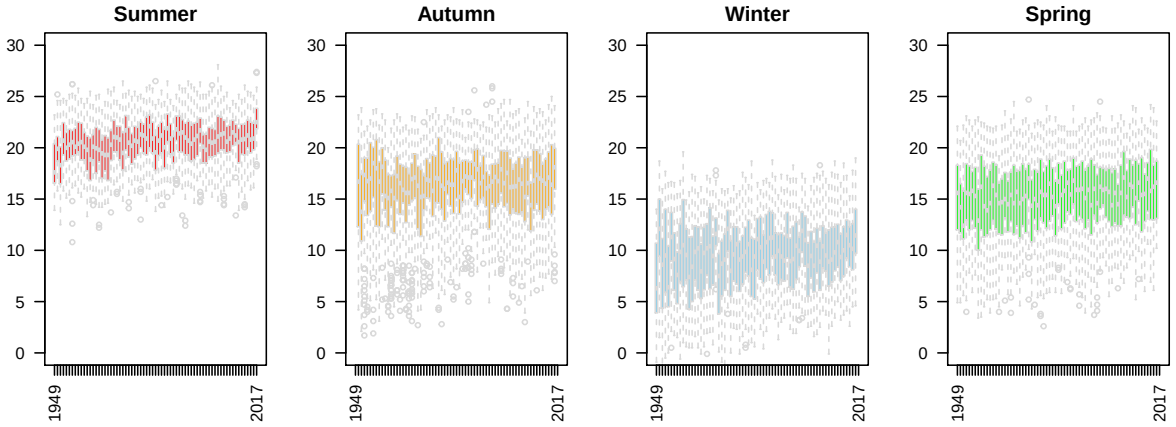
- `mar=c(8,4,4,1)` - is to specify the margins surrounding the plot: (bottom,left,top,right)
- `cex.main`, `cex.lab`, `cex.axis` - affects the font size
- `title()` gives finer control for the `xlab`, `ylab`, `main`, using the `line` parameter, you can adjust the location of the labels close to or away from the borders of the plot

```
season.col <- list(Summer='red',Autumn='orange',Winter='skyblue',Spring='green')
years[-c(1,length(years))] <- NA

par(mfrow=c(1,4),mar=c(4,2,2,2))
for(season in levels(seasons)) {
  keep.rows <- seasons == season
  season.temp <- minTemp[keep.rows,]
  boxplot(season.temp, las=2, col=season.col[[season]],
          names=years, border='grey85', main=season)
}
```



The above plot show there is no particular seasonal trend in minimum temperature. This of course does not seem right logically. Closer attention will reveal that the y-axis are different for each plot, this can lead to mis-interpretation if the reader is not paying close attention. It is better to keep the y-axis the same across comparative figures. You can specify this using the `ylim` parameter.

```
par(mfrow=c(1,4),mar=c(4,2,2,2))
for(season in levels(seasons)) {
  keep.rows <- seasons == season
  season.temp <- minTemp[keep.rows,]
  boxplot(season.temp, las=2, col=season.col[[season]],
          names=years, border='grey85', main=season, ylim=c(0,30))
}
```



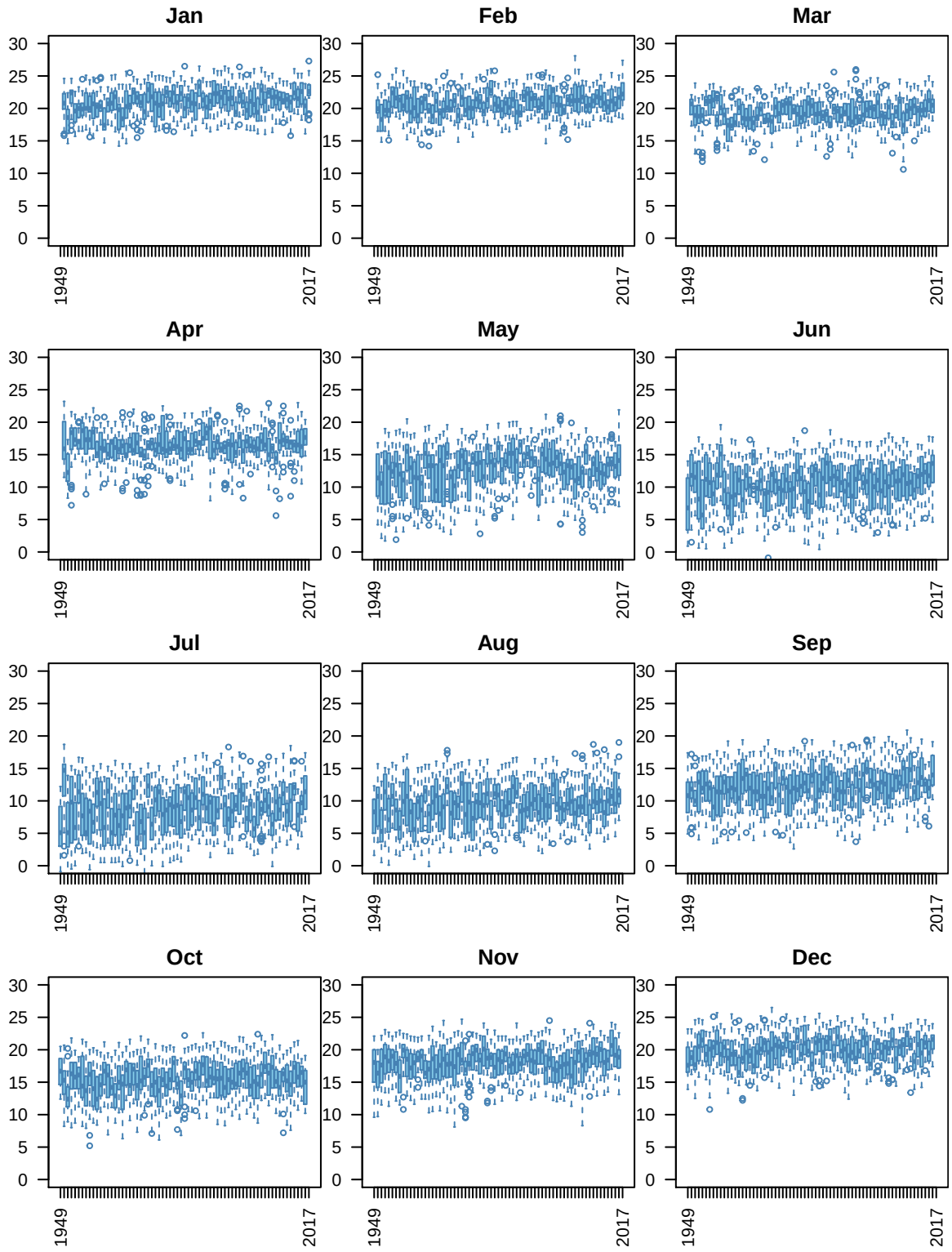
Now it is clearer that there is differences in the sepal and petal length/widths across the different species.

Monthly trend

If you are ahead, how about using the `months` variable to create monthly plots and see if there is any trend.

Hint: use `par(mfrow=c(4,3))` to create a 4 by 3 matrix of plots.

Expected output

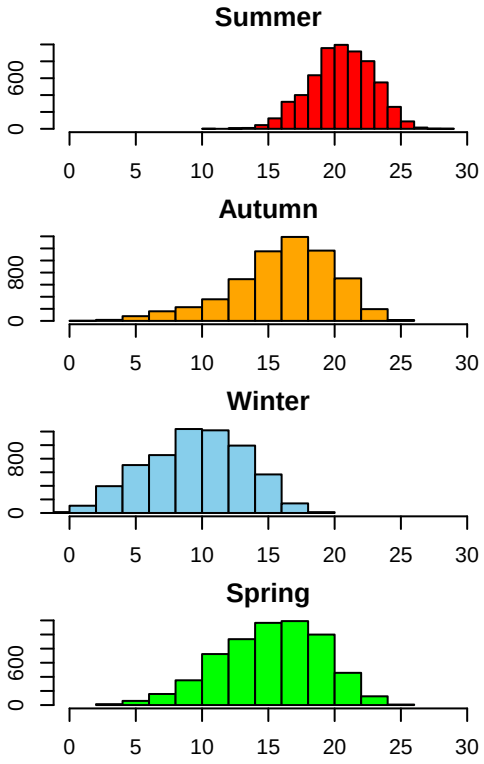


For more information on other graphical parameters, look up `par` which will show you the list of parameters that can be set for plotting.

7.2 Histograms

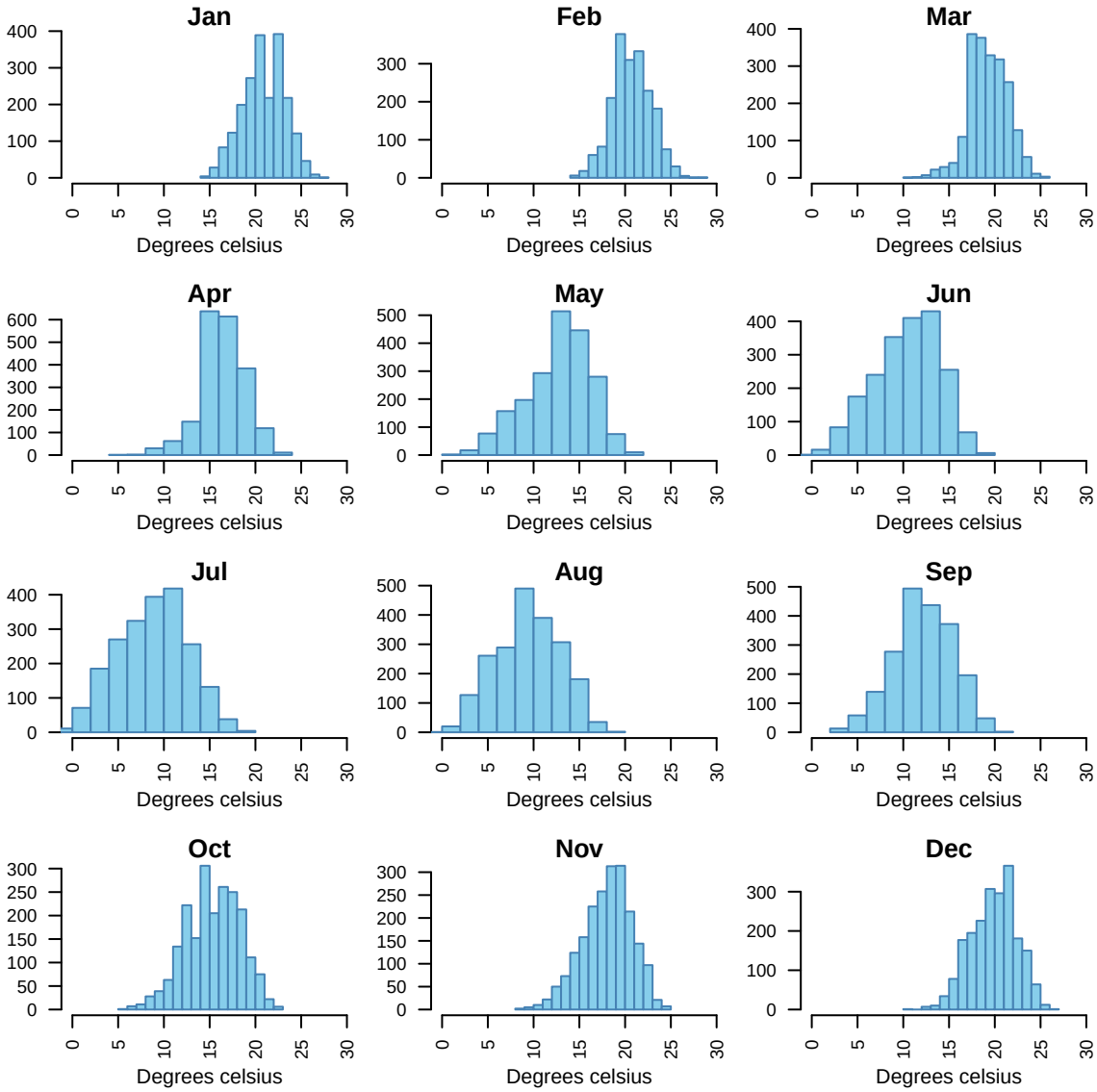
A histogram shows the distribution of data for typically a single vector. This time we will plot the distribution for each season, again using the `for` loop to generate the plots. Note how we also keep the x-axis scale the same across the plots, this time using the `xlim` parameter.

```
par(mfrow=c(4,1),mar=c(2,2,2,0))
for(season in levels(seasons)) {
  keep.rows <- seasons == season
  temp <- as.matrix(minTemp[keep.rows,])
  hist(temp, main=season, col=season.col[[season]], xlim=c(0,30))
}
```



Try creating a histogram for temperatures by month.

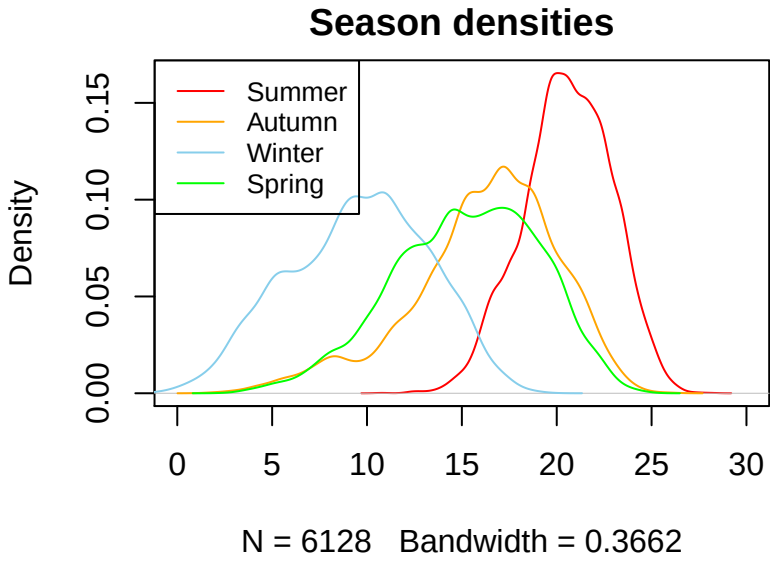
Expected output


Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Degrees celsius

7.3 Density plots

Another alternative for histogram plot is to use the density plot. See `plot(density(x))` where x is the vector of numbers like the one used in `hist()` above. Since density plots are lines, we can plot multiple densities onto the one graph:

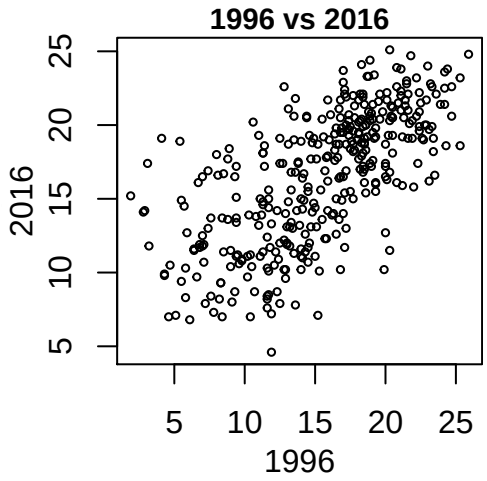
```
par(mar=c(4,4,2,0))

first.season <- levels(seasons)[1]
for(season in levels(seasons)) {
  keep.rows <- seasons == season
  temp <- as.matrix(minTemp[keep.rows,])

  if (season == first.season) {
    plot(density(temp, na.rm=T),xlim=c(0,30), col=season.col[[season]],
         main="Season densities")
  } else {
    lines(density(temp, na.rm=T),xlim=c(0,30), col=season.col[[season]])
  }
}
legend('topleft',
       legend=levels(seasons),
       lty=1,
       cex=0.8,
       col=as.character(season.col))
```



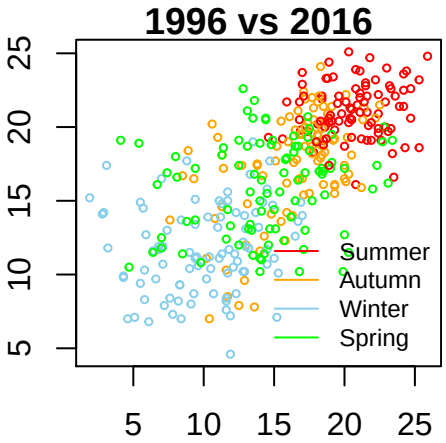
7.4 Scatterplots

Scatterplots are widely used for plotting data where an object has mutiple variables. This is a quick way to look for any correlation in the dataset.

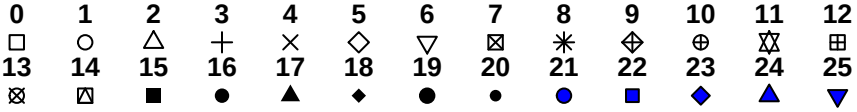
```
plot(minTemp$X1996, minTemp$X2016, cex=0.5,
     main='1996 vs 2016', xlab='1996',ylab='2016')
```




```
plot(minTemp$X1996, minTemp$X2016, cex=0.5,
     col=as.character(season.col[seasons]),
     main='1996 vs 2016', xlab='1996',ylab='2016')
legend('bottomright', bty='n', legend=levels(seasons),
       lty=1, cex=0.75, col=as.character(season.col))
```



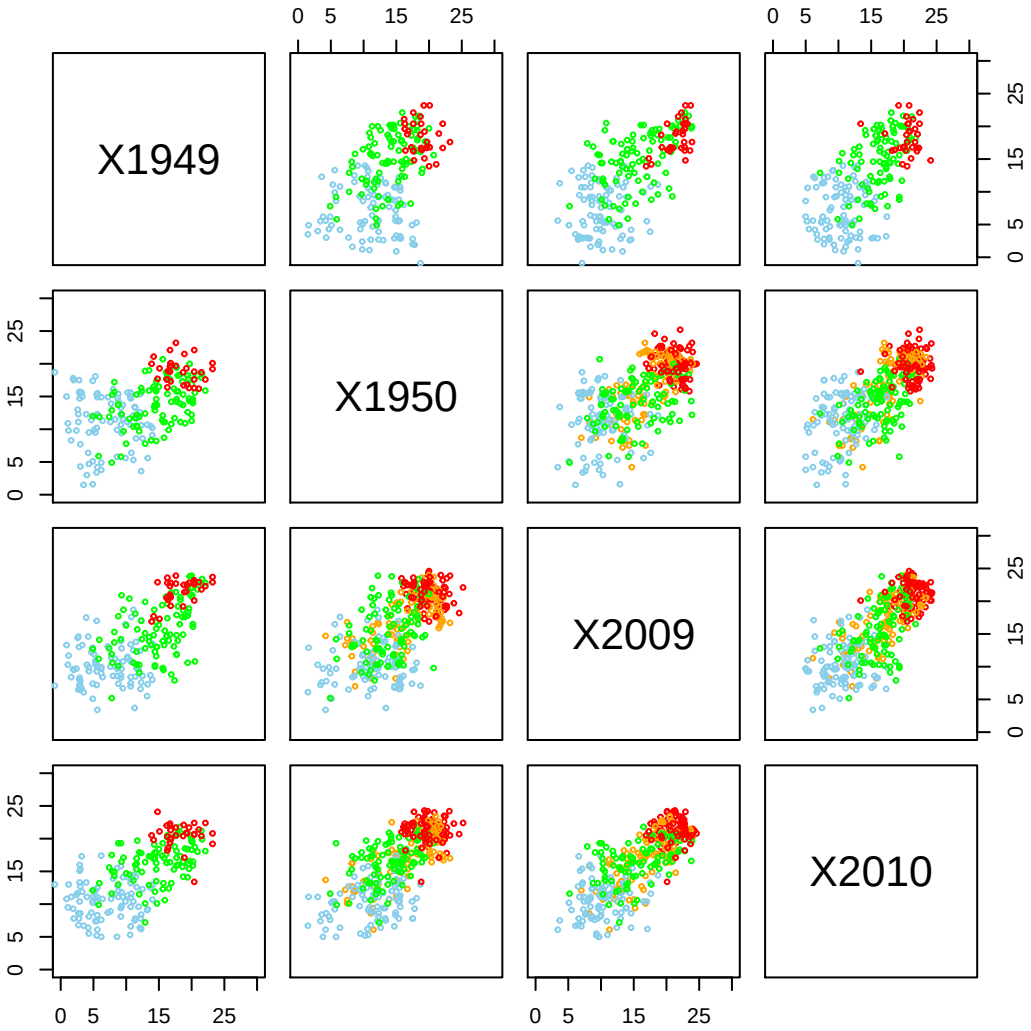
The `pch` parameter is the symbols to use, search in help for pch to get a list of allowable symbols



7.4.1 Scatterplot matrix

You can also generate a scatter plot matrix using the following code. This will make a $N \times N$ plot of all the columns in your dataset. Again this is only useful for numeric columns.

```
plot(minTemp[,c(1:2,61:62)],
     col=as.character(season.col[seasons]),
     cex=0.5, xlim=c(0,30), ylim=c(0,30))
```



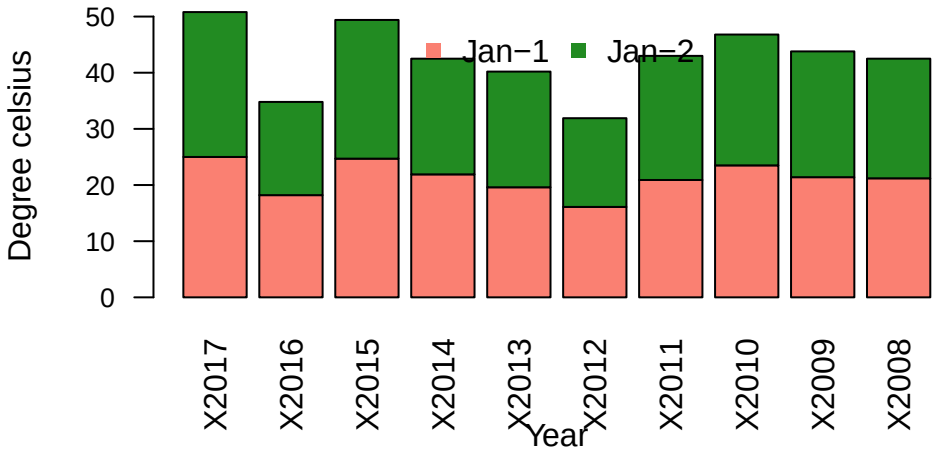
Becareful of generating scatterplot matrix if you have a **BIG** dataset, as this can take up all your computer memory.

7.5 Bar charts

A barchart is superficially similar to a histogram in the bars of data are displayed. Bar charts are ideal for displaying data associated with categorical data.

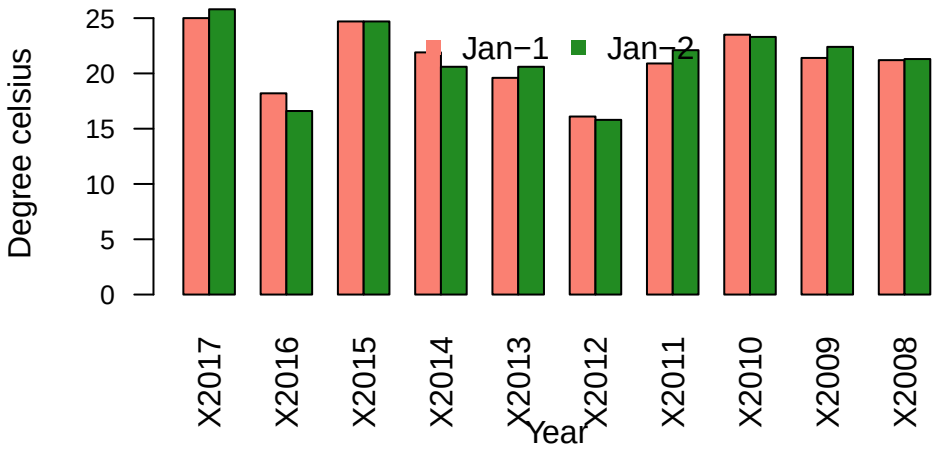
Let's say for example, we want to see the trend for January 1st and 2nd across the last 10 years. The `rev()` function reverses a given vector.

```
last.10.years <- colnames(minTemp)
last.10.years <- rev(last.10.years)[1:10]

temp <- as.matrix(minTemp[1:2,last.10.years])
barplot(temp, las=2, col=c('salmon','forestgreen'),
        xlab='Year', ylab='Degree celsius')
legend('topright',legend=rownames(minTemp)[1:2], horiz = T, cex=0.75,
       col = c('salmon','forestgreen'),pch=15, bty='n')
```



The above is not suitable as it stacks the bars on top of each other, while we can compare Jan-1, we cannot compare Jan-2:

```
barplot(temp, las=2, col=c('salmon','forestgreen'), beside=T,
        xlab='Year', ylab='Degree celsius')
legend('top',legend=rownames(minTemp)[1:2], horiz = T,
       col = c('salmon','forestgreen'),pch=15, bty='n')
```



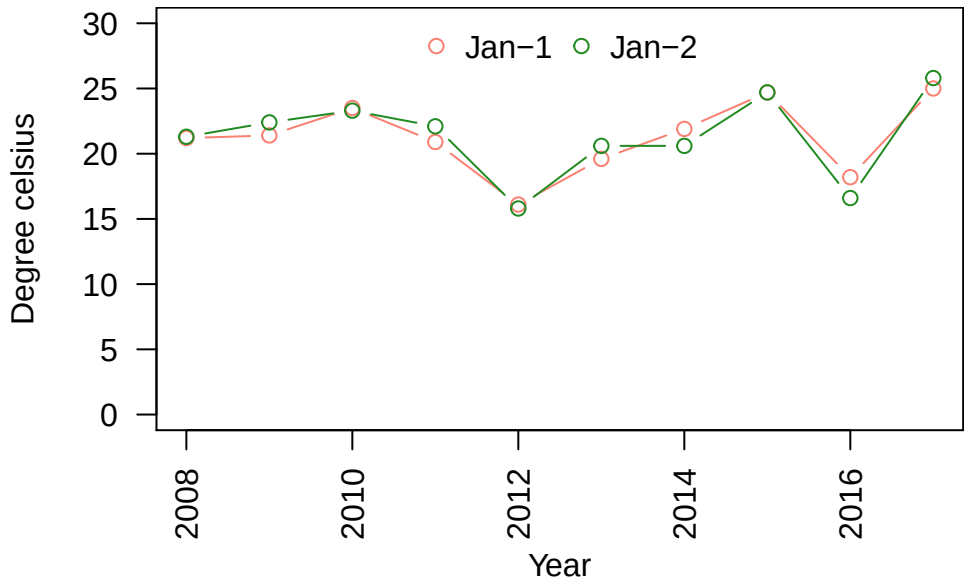
While this is better and allows for comparison between the two days over the last 10 years, when it comes

to timeseries data, it is more suitable to use points and lines to show trend. Replotting the first barplot we saw before using lines and points:

```
last.10.years.n <- as.numeric(gsub("X","",last.10.years))
last.10.years.n
```

```
##  [1] 2017 2016 2015 2014 2013 2012 2011 2010 2009 2008
```

```
plot(last.10.years.n,temp[1,],
     las=2, ylim=c(0,30), type='b',col='salmon',
     xlab='Year', ylab='Degree celsius')
lines(last.10.years.n,temp[2,],col='forestgreen',type='b')
legend('topright',legend=rownames(minTemp)[1:2], horiz = T, cex=0.75,
       col = c('salmon','forestgreen'),pch=1, bty='n')
```



Note that the x-axis in the second plot is in increasing order as we convert the years from characters to numbers.

The second plot is easier to do using the `ggplot2` package but this is beyond the scope of this workshop. See our Data preparation, processing and reporting with R workshop.

Appendix 1

Summary of Object Types

Type	Description
vectors	ordered collection of numeric, character, complex and logical values.
factors	special type vectors with grouping information of its components
data frames	two dimensional structures with different data types
matrices	two dimensional structures with data of same type
arrays	multidimensional arrays of vectors
lists	general form of vectors with different types of elements
functions	piece of code

More information on R and Bioconductor for Genomics

- Thomas Girke's manuals and guides to R http://manuals.bioinformatics.ucr.edu/home
- http://www.r-bloggers.com/

7.6 Going further with R

This section is a quick glance at the other features in R that will help you in your future data analysis. We will quickly look at:

- **Documentation** with R
- **Vignettes** that come with R packages including citing packages
- **SessionInfo** information to reproduce your data analysis, and
- **Other ways to run R**

7.6.1 Vignettes

Just now, we explored some of the help functions built into R. As well as this builtin help, most R packages also come with their own, often extensive, documentation. This is normally in the form of a vignette, a document that provides a task-oriented description of package functionality. Vignettes contain executable examples and are intended to be used interactively. You can also download the vignette for a package that isn't installed on your system by visiting the CRAN, Bioconductor project webpages, or via an internet search.

Using vignettes (optional)

1. The `vignette()` command provides access to vignettes for installed packages.

2. Run `vignette()` at the console to list all the vignettes available.
   - Notice that a package can, but does not always, have a vignette with the package name - for example the `annotate` package has an annotate vignette, but `BiocParallel` only has the `IntroductionToBiocParallel` vignette
3. Type `vignette("biomaRt")` to open the vignette for the biomaRt package, which links to the Biomart gene annotation database.
4. Some vignettes have non-unique names - for example, several packages have an intro vignette.
   a) Try `vignette("intro")`
   b) To get the limma package vignette, use the command `vignette("intro", package = "limma")`

7.6.2 Citations

The R core development team and the very active community of package authors have invested a lot of time and effort in creating R as it is today. Please give credit where credit is due and cite R and R packages when you use them for data analysis.

```
citation("VennDiagram")
citation("limma")
```

7.6.3 SessionInfo

Congratulations - you have now conquered the basics of R and you are ready to start breaking things! One of the most useful aspects of R is the rich ecosystem of supplementary packagess that can aid, facilitate and empower your research. The Bioconductor framework contains hundreds of packages of methods of relevance to biologists working with biological data (population genetics, SNPs, NGS reads, microarrays, mass spectrometry etc). As we discovered in the Vignette section, there are well documented tutorials and guides that will work us through the application of fabulous methods and workflows. You will come across some unexpected 'error' messages.

R packages are written by people like the QFAB bioinformaticians. Well intentioned, but busy. Code is crafted lovingly and tested in the scenarios that are implicit in our daily development. You are likely to be running R on a different computer with combinations of installed packages that are subtly different to those that we wrote the software with. If you identify a "bug" in that the package gives a wrong result, fails with a hairy error message or misbehaves it is worthwhile to let the developer know that you are having a problem. Reproducibility of the error is critical - we would like to know which version of R is being used and all of the packages that are loaded in memory so that the developer can agree that yes, there is a problem and can help in the resolution.

It is trivial to report the data on the R environment that you are working in - the SessionInfo function reports installed packages and their versions.

```
require(limma)
```

```
## Loading required package: limma
```

```
sessionInfo()
```

```
## R version 3.4.1 (2017-06-30)
## Platform: x86_64-redhat-linux-gnu (64-bit)
## Running under: CentOS Linux 7 (Core)
##
```

```
## Matrix products: default
## BLAS/LAPACK: /usr/lib64/R/lib/libRblas.so
##
## locale:
##  [1] LC_CTYPE=en_AU.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_AU.UTF-8        LC_COLLATE=en_AU.UTF-8
##  [5] LC_MONETARY=en_AU.UTF-8    LC_MESSAGES=en_AU.UTF-8
##  [7] LC_PAPER=en_AU.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_AU.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] limma_3.30.13    kableExtra_0.4.0 knitr_1.17
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.12     bookdown_0.5     codetools_0.2-15 digest_0.6.12
##  [5] rprojroot_1.2    R6_2.2.2         backports_1.1.0  magrittr_1.5
##  [9] evaluate_0.10.1  httr_1.3.1       rlang_0.1.2      stringi_1.1.5
## [13] rstudioapi_0.6   xml2_1.1.1       rmarkdown_1.6    tools_3.4.1
## [17] stringr_1.2.0    readr_1.1.1      hms_0.3          yaml_2.1.14
## [21] compiler_3.4.1   rvest_0.3.2      htmltools_0.3.6  tibble_1.3.4
```

7.6.4 Other ways of running R

Today we are using RStudio, but R can be run in a number of different ways; for example, you could embed some R functionality in a computer program written in Python or Java. As discussed in sections 1.2 and 1.3, both R and RStudio provide an interactive environment where you are prompted to enter a single command at the time. In this training course we are providing you with snippets of information that could be used within a data analysis workflow. These are intended to be run interactively.

In a bioinformatics laboratory the scientists who use R craft a mixture of packages and workflow scripts into analytical pipelines. R may be even be called by other software environments. Bioinformaticians will typically use the R console for refining, tuning and modifying existing scripts that they have written. The workflow is run through master scripts. In this section we will look at a simple R script that will create a table of information and write it out to file. We will be having a more complete look at the process of writing data to file in a later section (Section **??**).

The advantage of running R scripts as a batch analysis is that larger and more complex analyses (such as the mapping of short DNA sequence reads) can be run overnight and each of the commands will run successively as prior commands complete.

7.6.4.1 Preparing an R script

An R script is a container for multiple R commands. It is intended to be run largely hands-off.

The RStudio software provides us with a very convenient way to create an R script - in the file dialog there is the option to create file and an R Script is a primary file type. R scripts typically have the extension `.R`. The file is a plain text file and needs to know the packages that should be loaded, the objects that should be set and the working environment where we should be working.

7.6.4.2 Running an R script

The following command is how you run an R script from a terminal window. This is outside of RStudio.

```
R CMD BATCH [options] my_script.R [outfile]
R.exe" CMD BATCH
```

```
--vanilla --slave "c:\my projects\my_script.R"
```

We will demonstrate the next exercise to show you the method to run a script file using RStudio. You can repeat this exercise in your own time.

Running a script in RStudio

- Create a new script file by going to **File** > **New File** > **R Script**.
- Enter some commands in the editor window, for example the code chunk below.
- Save the script with a file name e.g. "area-rectangle.R"
- In RStudio, there are 3 ways you can run this script from top to bottom:
  1. In the **console** window, type in the command: `source("area-rectangle.R")`. What is the output?
  2. In the **editor** window, top right corner click on the **arrow** next to the **Source** button, select **Source**. What is the output?
  3. Repeat step 2 but this time select **Source with Echo**. What is the difference between step 2 and 3?

```
print(date())
print("This is an R script!")
length <- 10.25
width <- 4.35
area.of.rectangle <- length * width
print(paste("The area of a rectangle with length=",
            length,
            "cm by width=",
            width,
            "cm is",
            area.of.rectangle,"square cm."))
```

Use [Tab] to autocomplete

You can use the tab key to autocomplete a variable name, a function name or a filename (if the file is in the working directory). This will save you a lot of time and prevent typing errors when you are analysing your data.