R Module 1

A Primer on R for Windows

(Version 3.2.1)

Certificate in EnvIroStats (Non-Award)

This document is part of an online Certificate in EnviroStats (Non-Award) by the University of Canberra. Course enquiries can be directed to the address below. Expressions of interest in the course can be made online through:

http://aerg.canberra.edu.au/envirostats

Copies of this publication are available from:

The Institute for Applied Ecology
University of Canberra ACT 2601
Australia

Email: **georges@aerg.canberra.edu.au**

Correct citation:

Georges, A. (2015). *Biometry: Statistics for Ecology and Natural Resource Management. Module 1: A Primer on R for Windows (Version 2.5.1).* Flexible Delivery Development Unit, Centre for the Enhancement of Learning, Teaching and Scholarship (CELTS), University of Canberra, ACT 2601, Australia. ISBN: ####.

SPONSORED BY:

Australian Government
Australian Centre for International Agricultural Research

WorldFish CENTER

Australian Government
Fisheries Research and Development Corporation

Materials development team:

Authors:	Arthur Georges and Peter Caley, 2007
Instructional designer:	Peter Donnan, 2002
Editor:	Loretta Barnard, 2002
Graphic Design:	Peter Delgado, 2002
Desktop Publishing:	Kristi McDonald, 2004 Sue Bebbington, 2004
FDDU Project Manager:	Deborah Veness, 2002
Dynamic Web Page Design:	TCNI Software Solutions PO Box 47 LATHAM ACT 2615 Australia

First prepared in January, 2004 (Version 1.0).
Reprinted June 2007.
Revised and reprinted, June 2007 (Version 1.2)

Published by Technology & Educational Design Services

(TEDS)
University of Canberra
ACT 2601, AUSTRALIA

Contents

What is R?

R is a statistical computing language based on an earlier implementation of a programming language called S. S is still available in the commercial form of S-plus, whereas R is in the public domain.

S was originally designed at Bell Laboratories in 1976, as an alternative to the FORTRAN language and statistical subroutines available at that time. S offered a more interactive approach to programming, its source version was made public in 1981, it became available in commercial form in 1984, and it has maintained its popularity particularly among the statistical community.

R was created by Ross Ihaka and Robert Gentleman (hence the name R) at the University of Auckland, New Zealand, and is now developed by the R Development Core Team. R is considered by its developers to be an implementation of the S programming language.

Many statistical packages on the market, such as SAS, SPSS and Statistica are regarded as fourth generation statistical programming languages. The R programming language is a hybrid between a third generation language such as C or FORTRAN and a fourth generation language such as SAS. This provides for much greater flexibility for the analyist, but demands much more in terms of programming skills. This will become quickly evident as you move through this module.

R supports a wide variety of statistical and numerical techniques, with comparable benchmark results to Octave and its proprietary counterpart MATLAB. R is also provides the analyst with a very wide range of packages, which are user-submitted program **libraries**, for specific functions or specific areas of study. As a result, R is one of the most comprehensive statistical analysis systems on the market. R has exceptionally good graphical capacity, and can be used to produce publication-quality graphs.

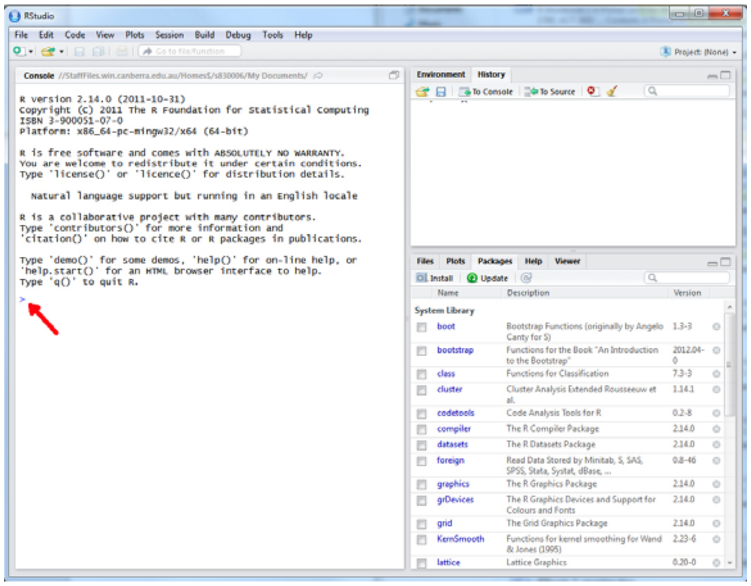
The versatility of R has led to many different styles in the way the program is used. A programmer will use R in a very different way from someone using R to undertake statistical analyses. This module presents only one style. As your skills develop, and you learn more of the capabilities and options of the programming language, you will develop your own style.

Lesson 1: The Operating Environment

The Command Line Interface

Rather than using the standard distribution of R, we will be using an implementation with a graphical user interface called R-studio.

When you first start R-studio, a graphical user interface opens with many features to assist you (Figure 1-1). After some introductory text appears in the Console, a **Command Prompt** is presented (>), and the system awaits instructions.


RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Project: (None)
Console //StaffFiles.win.canberra.edu.au/HomesS/s830006/My Documents/
Environment History
To Console To Source
Files Plots Packages Help Viewer
Install Update

```
R version 2.14.0 (2011-10-31)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-mingw32/x64 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

	Name	Description	Version
	System Library		
☐	boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-3
☐	bootstrap	Functions for the Book "An Introduction to the Bootstrap"	2012.04-0
☐	class	Functions for Classification	7.3-3
☐	cluster	Cluster Analysis Extended Rousseeuw et al.	1.14.1
☐	codetools	Code Analysis Tools for R	0.2-8
☐	compiler	The R Compiler Package	2.14.0
☐	datasets	The R Datasets Package	2.14.0
☐	foreign	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...	0.8-46
☐	graphics	The R Graphics Package	2.14.0
☐	grDevices	The R Graphics Devices and Support for Colours and Fonts	2.14.0
☐	grid	The Grid Graphics Package	2.14.0
☐	KernSmooth	Functions for kernel smoothing for Wand & Jones (1995)	2.23-6
☐	lattice	Lattice Graphics	0.20-0

Figure 1-1. R as it appears when it first starts. The R Console window and two other windows are visible. The Program Editor and R Graphics windows do not appear until required.

Before we move on, there are a couple of little tricks here that are worth mentioning. The first is that the Console can be cleared of text using control-L (^l). The second tip is that the up-arrow will recall previously submitted commands, which will save you a lot of typing. Try these as you go along.

The simplest way of using R is to supply instructions to this command prompt.

```
> beetles <- c(15.2,12.1,17.8,13.9,16.4,15.1)
```

There is a lot to this simple command. What we are doing here is creating a list of values, referred to as a **vector** in R terminology. In this case, the data are lengths of beetle elytra. The concatenate function `c()` is used to create the vector which is then assigned to the **object** `beetles` using the **assignment operator** <-. The object `beetles` is called an object because it is a self-contained entity that can be used in subsequent calculations.

Instructions to the command line are terminated with a return (↵) or a semi-colon (;). R instructions are case sensitive, so the objects `beetles`, `Beetles` and BEETLES are all considered as separate objects. It is wise to adopt a consistent practice, such as always using lower case unless upper case is demanded by the R syntax.

Spaces matter, sometimes. So you will need to watch that.

If you instruct R to undertake some action, and do not assign it to an object, then R will direct the results of the instructions to the screen. For example, requesting R to create the vector of beetle elytra without assigning it to `beetles` will result in the vector being listed on the screen.

```
> c(15.2,12.1,17.8,13.9,16.4,15.1)
[1] 15.2 12.1 17.8 13.9 16.4 15.1
```

You can view the contents of an object simply by giving its name in response to the command prompt.

```
> beetles
[1] 15.2 12.1 17.8 13.9 16.4 15.1
```

As an object, `beetles` can be used in subsequent calculations. For example,

```
> mean(beetles)
[1] 15.08333
```

R programs often comprise a series of nested instructions, and the same result could have been obtained by using

```
> mean(c(15.2,12.1,17.8,13.9,16.4,15.1))
[1] 15.08333
```

This is the advantage of an object-oriented approach to programming.

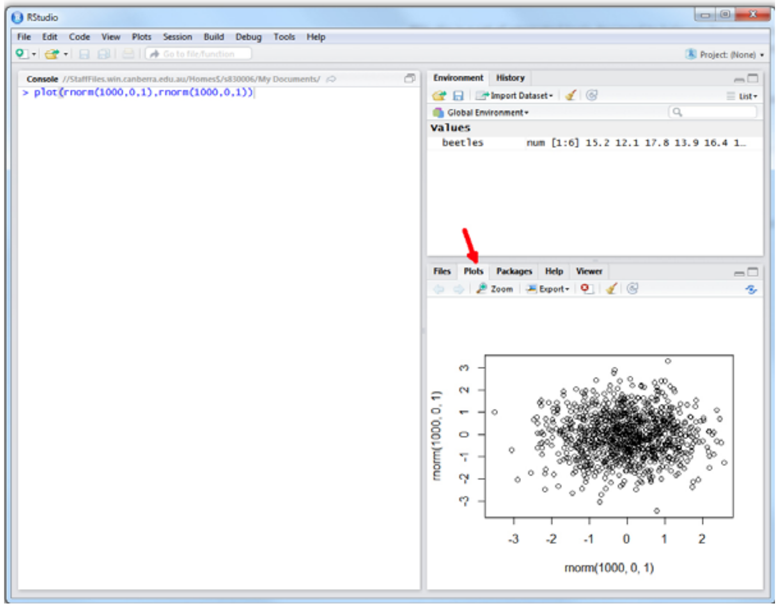
The Graphics Interface

When a command requires more sophisticated output, R will open a purpose-built window. The most useful of these is the graphics window.

A scatter plot of 1000 pairs of coordinates drawn at random from a bivariate standard normal distribution (mean=0, stdev=1) is made by combining the plot() function with the rnorm() function as follows:

```
> plot(rnorm(1000,0,1),rnorm(1000,0,1))
```

The result is shown in in Figure 1-2. This scatter plot can be saved to a file or copied to the clipboard by right-clicking on the graphics window and choosing the desired outcome.



```
> plot(rnorm(1000,0,1),rnorm(1000,0,1))
```

```
Values
  beetles        num [1:6] 15.2 12.1 17.8 13.9 16.4 1…
```

Figure 1-2. R as it appears after activating the graphics window.

You can pull the graphics out into its own window with the [Zoom] tab, or export the image in one of the standard formats using the [Export] tab.

The R Editor Interface

Using the Command Line Interface is great for a quick analysis, but it is essentially a calculator mode. Once you have done the calculations, you walk away only with the results. In more substantial analyses, we need to better manage the set of programming instructions needed to do the job. We do this using the **R editor**, which can be accessed from the file menu [File>New File>R Script] or by typing control-N (^N).

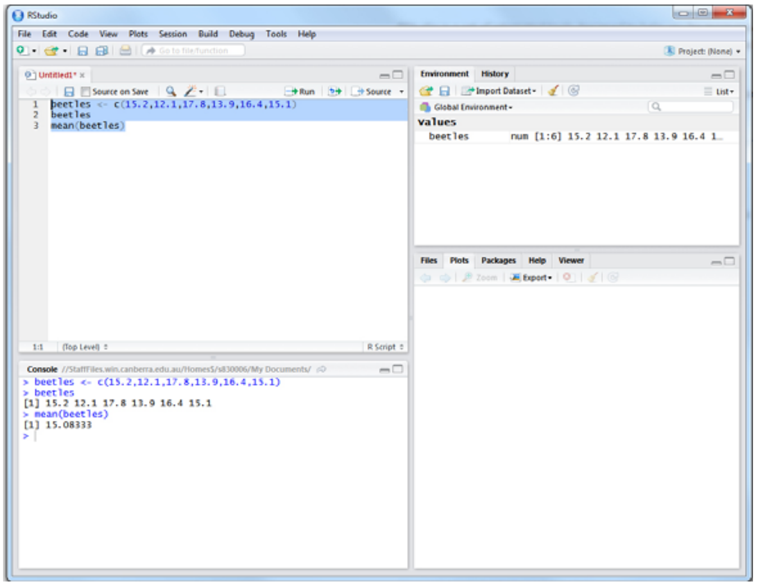
The idea is to type all instructions in the R editor for progressive submission or for submission as a block. At the end of the process, you have a complete program listing that can be saved to disk for later use.

The R editor is not all that sophisticated. Each instruction is typed in on its own line. A line can be submitted for execution by placing the cursor on it and typing control-r (^r). Alternatively, blocks of instructions can be highlighted and submitted in the same way. This allows progressive debugging of the program as it is constructed.

It is wise to include abundant comments as part of your programs, so that you can understand them later or pass them to others in a comprehensible form. Comments are preceeded by the # character and terminated by a return (↵).

Our simple R program, with comments added is shown in Figure 1-3.

```
# Creating and displaying a list of beetle measures
beetles <- c(15.2,12.1,17.8,13.9,16.4,15.1)
beetles
mean(beetles)
```



```
beetles <- c(15.2,12.1,17.8,13.9,16.4,15.1)
beetles
mean(beetles)
```

Environment — values: beetles num [1:6] 15.2 12.1 17.8 13.9 16.4 1…

Console //StaffFiles.win.canberra.edu.au/Homes5/s830006/My Documents/

```
> beetles <- c(15.2,12.1,17.8,13.9,16.4,15.1)
> beetles
[1] 15.2 12.1 17.8 13.9 16.4 15.1
> mean(beetles)
[1] 15.08333
> 
```

Figure 1-3. R as it appears after submitting a simple program.

Accessing Packages

Not many users of R program all of the scripts that they require to undertake a task. This would be like reinventing the wheel. Instead, it is possible to access scripts written by those who have come before you, and who have made those scripts available as a package. A complete list of available packages can be obtained from the *Comprehensive R Archive Network* known as CRAN (**https://cran.r-project.org/**).

You will generally identify the packages you require after some investigative work on the web, by talking to colleagues or taking pointers from the literature. For example, the package `reshape2` is useful for rearranging data, and can be accessed using the R-studio menus (Packages>Install Packages) or using the statement

```
> install.packages("reshape2", lib="C:/Program Files/R/R-
3.2.1/library")
```

You may need administrator rights to install packages. Packages are only installed once, not every time you require them.

The directory where packages are stored is called the library. R comes with a standard set of packages. A list of installed packages can be obtained using

```
> library()
```

or by examining the list using the Packages menu.

Apart from those included in the standard implementation of R, installed packages are loaded for use in a session with the library function.

```
> library(reshape2)
```

A list of loaded packages is obtained with

```
> search()
```

Error Handling and Help

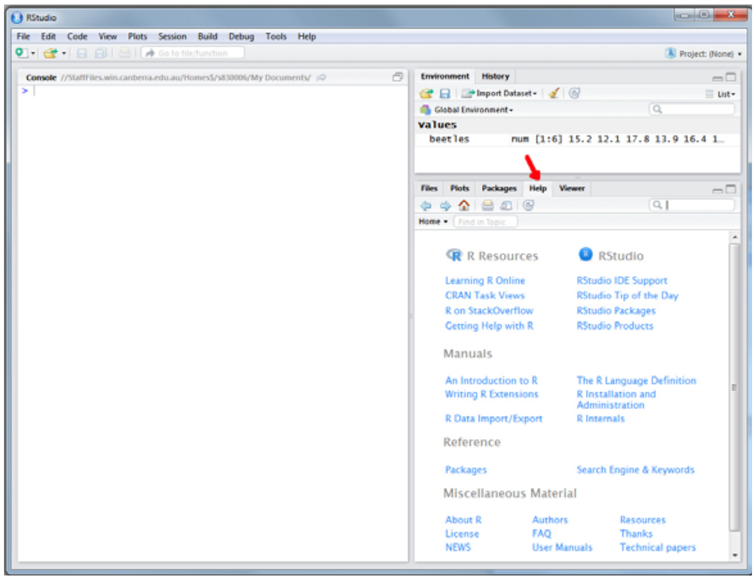
When you make an error in the syntax of commands given to R, the program will respond with some form of diagnostic message. Sometimes these are self-explanatory, sometimes they are not.

Help files

Fortunately, R has very extensive help documentation. If you know the exact name of the function you want help on (e.g. `sort`), help can be obtained using

```
> ?sort
```

More extensive help can be obtained from the help files. These are accessed via the [Help] tab of R-studio. Here you can use the Search Engine and Keywords link to access a wide range of information on the operations of R.



```
Values
  beetles        num [1:6] 15.2 12.1 17.8 13.9 16.4 1…
```

Figure 1-4. Useful information available using the [Help] tab.

Vignettes

Some packages in R have what are called **vignettes**. These are how-to guides for topics, and usually offer gentle introductions and examples. Alternatively, you can view vignettes from any loaded package by going to the 'Vignettes' menu and selecting the required package name. This will give a list of all available vignettes for you to open. Sometimes this menu doesn't appear until you load a package which has a vignette.

The Web

The web and Google are good places to turn for assistance. An excellent quick reference to R can be found on http://www.statmethods.net/

Managing Your Workspace

Starting a New Session

R facilitates the management of workflow by defining a **workspace** to hold all of your objects – vectors, dataframes, user-defined functions and the like. A workspace can be saved at the end of a session, and reloaded at a later time when you want to continue the analysis.

Managing workflow can be difficult in R, and we need some basic rules to minimize confusion.

- Identify discrete projects or analyses and create a separate Windows directory for each one. This way you will avoid having a jumble of objects from many analyses in your workspace.
- Tidy up after each session, by removing all unwanted and temporary objects, before saving your workspace.
- Use standard file naming conventions, such as filename.R for R programs and filename.dat for raw data files.

Once you have started R, you need to start a new project using File>New Project. R will prompt you for a directory in which to save all temporary and working files, and the project image if you choose to save it later.

R may ask you to save existing work before opening a new project. You should do this if you have important work that has been executed in a previously open project.

Resuming a Session

To open an existing project use File>Open Project. This will prompt you for the working directory, scan it for a previously saved workspace, load that workspace if it exists and pass control back to you via the command prompt.

Either way, with an existing or a new project, you can now begin a new analysis, or continue an existing analysis confident that the objects, datafiles, and other associated elements of your analysis are a self-contained unit.

Terminating a Session

Exit a session by exiting from R, at which time you will be asked whether or not you wish to save your workspace.

Managing your Objects

The active objects associated with your workspace are listed when you select the [Global Environment] tab in R-studio.

In addition, there are a number of useful functions for managing your workspace.

`> setwd("c://R_analysis")`	
	Sets the default directory for files, and action that can also be done from the R-studio menus.
`> ls()`	provides a list of objects in your current workspace.
`> rm(`*`object`*`)`	deletes an object from your workspace.
`> rm(list=ls())`	deletes all objects from your workspace.
`> sessionInfo()`	provides information about your session.

Setting a default directory

The best way to manage your work is to ensure that the files associated with each project is in a separate directory on disk. To set the default directory use

```
> setwd("C:/Users/username/Documents/R_demo")
```

R will then look in the directory R_demo when locating a file to read, and to write a file. Note the direction of the backslashes in the file specification.

Where have we come?

The above lesson was designed to give you an overview of the operation of R through the graphical user interface. Having completed this lesson, you should now be familiar the following concepts.

- R has available a Graphical User Interface (GUI) called R-studio, and within it, the R Console, R Editor Window, Graphics Output Window, and various Help Windows.
- R establishes a workspace. Managing the objects in that workspace is challenging for the new user of R, but proficiency will come with practice.

What comes next?

Next we will cover the data structures that are central to the R programming language, including **vectors, factors, arrays, lists** and **dataframes**.

Lesson 2: Basic Data Structures

R as a Programming Language

Workflow and Control Structures

R is a programming language combined with the features of a statistical package. As with most programming languages, R comprises:

- a set of **keywords** and **operators** that are combined by the programmer to form statements or instructions to be executed. R is unusual in that many of its commands comprise function calls, and so its keyword set is largely made up of the names of functions.
- a clearly defined **sequence** in which these instructions are executed.
- mechanisms for **branching**, subject to some condition being met (Figure 1-2).
- mechanisms for **looping**, that is, repeating sequence of statements while a condition is met until a condition is met, or some fixed number of times (Figure 1-2).
- Mechanisms for isolating blocks of code, to be called on repeatedly (and possibly recursively) as required. In R, these are called **functions**.

Figure 1-2. A sample R Program. The program is executed in sequence one statement at a time from the top. The "for" statement results in the enclosed code being repeated for each line of data. Branching is provided with an "if" statement.

```
# Read the data from the clipboard, space delimited
# Note: Cut from Excel
     forest <- read.delim("clipboard")
# View the contents of the dataframe forest to confirm
     forest
# Make dataframe forest the default
     attach(forest)
# calculate timber yield separately for each species
     for (i in 1:length(species)){
        if (species[i] == "radiata") {
          yield <- density*3.1416*diameter*height*0.85
        }
        else {
          yield <- density*3.1416*diameter*height*0.62
        } # Terminate the if statement
     } # Terminate the for loop
 # Add yield to the dataframe turtle
     cbind(forest,yield)
 # Examine forest to confirm
     forest
```

Input/Output (IO)

In addition, programming languages often have unique ways of assembling and managing data and handle **input and output** (I/O) in unique ways that must be mastered.

Data Structures

Finally, a clear understanding of **data structures** used by R and the way R passes data and results to and from the files, the screen and other applications is essential for using R.

We will learn more about all of these concepts as we move through the lessons of this Module, but for now, let's focus on **data structures**.

Objects

What are Objects?

We used the term object earlier and it is appropriate now to define this more precisely. An object in the context of R is a self-contained collection of data, data attributes, code for manipulating the data, and object descriptors that allow objects to recognise each other and interact appropriately.

Scalars, vectors, lists, matricies, arrays, dataframes, and even functions are objects in R, each able to interact with others in clearly defined ways. So a vector of data, an object, can be defined as follows.

```
> beetles <- c(15.2,12.1,17.8,13.9,16.4,15.1)
```

You can view the contents of an object simply by typing its name.

```
> beetles
 [1] 15.2 12.1 17.8 13.9 16.4 15.1
```

and pass it to another object, in this case, a function

```
> mean(beetles)
 [1] 15.08333
```

which recognises the object beetles as class “numeric” and acts upon it accordingly.

Similarly, the object mean(beetles) can be passed to another function

```
> log(mean(beetles))
 [1] 0.6931472
```

in a nested statement.

You can create your own objects, and indeed do this whenever you create a vector, for example. In this sense, the R programming language is object oriented.

The class of an object can be determined using

```
> class(beetles)
[1] "numeric"
```

We start with one of the simplest but most fundamental of data structures, the vector.

Vectors

What is a Vector?

It is not possible to program in R without a good grasp of one data structure in particular, so we introduce it early. It is a vector.

A vector is one of the simplest data structures in R. It is simply an ordered set of values. It is an ordered set in the sense that values in the set have a specific position in the array, and when the vector is accessed it passes the elements across one at a time in the order that they appear in the array.

One way to create a vector is to use the `c()` function, for example

```
> width <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Data Types

Vectors come in various forms, or classes, by which we mean that vectors can be numeric, integer, character or logical, but cannot contain an admixture of these types.

The most common classes are `character`, containing character strings of varying length, `numeric`, containing real numbers, `integer`, containing integers, and `logical`, containing TRUE or FALSE.

You can determine the class of a vector with the `class()` function.

```
> class(width)
[1] "numeric"
```

Coercion

Data of one type can be implicitly 'coerced' into another type during calculations. For example, data containing 1's and 0's will be treated as containing TRUE and FALSE respectively if used in a context where true and false values are expected. Similarly, the number 10 in one context may be viewed as the string "10" in another context. In this sense, R has dynamic not static data types. This requires careful handling.

Data can be explicitly converted from one type, or class in R terminology, using the functions as.numeric(), as.character() or as.logical().

```
> width
[1] 10.4  5.6  3.1  6.4 21.7
> width.txt <- as.character(width)
> width.txt
[1] "10.4" "5.6"  "3.1"  "6.4"  "21.7"
```

Unrecognised implicit coercion is a source of many syntax issues for the beginner, so explicit coercion is recommended, where possible, when starting out.

Sequences

The sequence function `seq()` is a versatile way of making vectors comprising incremental series. For example, the series of numbers from 1 to 10 in 0.2 unit intervals can be generated with

```
> series <- seq(1,10,0.2)
> series
 [1]  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6  2.8  3.0  3.2  3.4  3.6
[15]  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.4  5.6  5.8  6.0  6.2  6.4
[29]  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.2  8.4  8.6  8.8  9.0  9.2
```

```
[43]  9.4  9.6  9.8 10.0
```

Referencing Vector Values

You can view the contents of a vector by typing its name.

```
> width
[1] 10.4  5.6  3.1  6.4 21.7
```

Elements of the vector can be accessed individually in a variety of ways. The most direct is to refer to the values by their position in the vector. For example,

```
> width[4]
[1] 6.4
```

Note the use of square brackets for indexing a vector. You can also pick out the first, third and fourth values by listing their positions in the vector.

```
> width[c(1,3,4)]
[1] 10.4  3.1  6.4
```

Note here the nested use of the concatenate function `c()` which passes the listed values to the vector pointer successively so that the corresponding values of the vector are returned successively.

If the values we want are consecutive, we can use the colon (:) in short-hand notation to specify the concatenation.

```
> width[1:4]
[1] 10.4  5.6  3.1  6.4
```

You can select all values except those specified with negative signs.

```
> width[c(-2,-5)]
[1] 10.4  3.1  6.4
```

A value in a vector can be replaced using an assignment statement

```
> width[1] <- 10
> width
[1] 10.0  5.6  3.1  6.4 21.7
```

So you should by now have the gist. A vector is a data structure that holds an ordered set of numbers, each of which can be addressed by its position in the ordered set using square brackets.

Vector arithmetic

The normal rules of arithmetic apply to vectors in the sense that they apply to each element of the vector (note that it is not at all like vector arithmetic in the mathematical sense).

```
> width <- width + 10
```

will add 10 to each value of the vector. Similar actions occur with the other arithmetic operators. The functions `log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()`, and so on, all have their usual meaning, and when applied to a vector, are applied to each value of the vector.

It is possible to include two vectors in calculations, in which case their values will be included in the equation as matched pairs. For example, if we have two vectors with the same number of values, `length` and `width`, then the assignment

```
> area <- length*width
```

will yield a new vector with the values calculated by multiplying the first value of `length` with the first value of `width`, the second value of `length` with the second value of `width`, and so on. Calculations involving missing values will yield missing values (`NA`) and calculations resulting in undefined or indeterminate values will yield an R specific code `NaN`.

The standard arithmetic operators apply, and include and include the usual addition (+), subtraction (-), division (/), multiplication (*) and exponentiation (^).

Working with vectors of differing sizes is difficult, and we probably should not go there.

Factors

What is a Factor?

Factors are special types of vectors whose values have labels associated with them. For example, we might create a character vector containing a combined sex and maturity code for animals caught on a particular day.

```
> sexcode <- c("F", "F", "M", "J", "F", "M", "J", "J", "F")
> sexcode
[1] "F" "F" "M" "J" "F" "M" "J" "J" "F"
```

Converting this to a factor causes the values, in alphabetical order, to be assigned numbers, and those numbers to be assigned labels.

```
> sexcode <- factor(sexcode)
> sexcode
[1] F F M J F M J J F
Levels: F J M
```

Outwardly, this is a subtle difference to be sure. The numbers are hidden behind the scenes, and re-caste with their character values by R when they are printed out. That they are stored as numbers is evident when we print the object `sexcode` out without reference to its class.

```
> unclass(sexcode)
[1] 1 1 3 2 1 3 2 2 1
attr(,"levels")
[1] "F" "J" "M"
```

The subtle difference between a character vector and a factor will become evident in analyses that involve discrete factors, such as analysis of variance. Confusing factors with vectors is a cause of much frustration for those new to R, and you are advised to regularly check the class of the objects you are using in calculations.

Matricies

What is a Matrix?

A matrix is a table with rows and columns. Like vectors, the matrix can contain values only if they are of the same type. Matricies are usually numeric.

For example, a 2-dimensional matrix containing 18 values could be established from a vector as follows.

```
> count <- seq(1:18)
> count
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

```
> a <- array(count, c(9,2)) # 9 rows, 2 columns
> a
      [,1] [,2]
 [1,]    1   10
 [2,]    2   11
 [3,]    3   12
 [4,]    4   13
 [5,]    5   14
 [6,]    6   15
 [7,]    7   16
 [8,]    8   17
 [9,]    9   18
```

Note that the array function fills the matrix from left to right, that is, it cycles through the rows before moving to the next column. This is important, lest you come terribly unstuck with computations.

Referencing matrix values

Because matricies are vectors with more complex indexing, it should come as no surprise that you can access the contents of the array as if it were a vector.

```
> a[15]
[1] 15
```

More conveniently, the contents of an array would be accessed using the two index values

```
> a[6,2]
[1] 15
```

You can pull out a whole row with

```
> a[7,]
[1] 7 16
```

or a whole column with

```
> a[,2]
[1] 10 11 12 13 14 15 16 17 18
```

Assigning Row and Column Names

Row and column names can be added using the `rownames()` and `colnames()` functions.

```
> rownames(a) <- c("AA45620", "AA35432", "AA75418",
     "AA25767",    "AA87556",    "AA45666", "AA45667",
     "AA45668", "AA45669")
> colnames(a) <- c("BRI", "CBR")
> a
        BRI CBR
AA45620   1  10
AA35432   2  11
AA75418   3  12
AA25767   4  13
AA87556   5  14
AA45666   6  15
AA45667   7  16
AA45668   8  17
AA45669   9  18
```

Once you have named the rows and columns, the above values can be referenced by name. The row and column names should be unique if they are to be used in referencing. The statements in the previous section can be alternatively represented by

```
> a["AA45666","CBR"]
[1] 15
```

```
a["AA45667",]
BRI CBR
  7  16
> a[,"CBR"]
AA45620 AA35432 AA75418 AA25767 AA87556 AA45666 AA45667 AA45668 AA45669
     10      11      12      13      14      15      16      17      18
```

Adding New Rows and Columns

```
> AA45213 <- c(32, 14)
> a <- rbind(a, AA45213)
> a
        BRI CBR
AA45620   1  10
AA35432   2  11
AA75418   3  12
AA25767   4  13
AA87556   5  14
AA45666   6  15
AA45667   7  16
AA45668   8  17
AA45669   9  18
AA45213  32  14
```

The function rbind() can be used in the same way to append a second matrix with the same number of columns. We first create the second matrix with the same row and column names and the same number of columns.

```
a2 <-  array(c(3, 24, 68, 5, 17, 26, 18, 97), c(4,2))
rownames(a2) <- c("AA65634", "AA75417", "AA25438", "AA75769")
colnames(a2) <- c("BRI", "CBR")
a2
        BRI CBR
AA65634   3  17
AA75417  24  26
AA25438  68  18
AA75769   5  97
```

and then append matrix a2 to the end of matrix a.

```
> a <- rbind(a, a2)
> a
        BRI CBR
AA45620   1  10
AA35432   2  11
AA75418   3  12
AA25767   4  13
AA87556   5  14
AA45666   6  15
AA45667   7  16
AA45668   8  17
AA45669   9  18
AA65634   3  17
AA75417  24  26
AA25438  68  18
AA75769   5  97
```

In an analogous way, you can join two matricies with the same number of rows side by side using `cbind()`.

Deleting Rows and Columns

A row can be deleted from a matrix by setting it to NULL.

```
a["AA25438",] <- NULL
```

and in a similar way, columns can be deleted.

Matrix Arithmetic

As with vectors, the normal rules of arithmetic apply to matricies in the sense that they apply to each element of the matrix. For example,

```
> a <- a*10
```

will multiply all values in the matrix by 10, and so on.

If two matricies have the same dimensions, their elements can be added, subtracted, multiplied or used in any equation to generate a new matrix of the same size. The arithmetic operations apply to the elements of the two matricies in pairwise fashion.

In addition, there is a suite of functions for undertaking matrix algebra, but we will not cover those here.

Arrays

What is an Array?

A vector is an array with one dimension, a matrix is an array with two dimensions, but arrays can refer to higher dimensional block data structures.

As an example, a 3x5x2 array has three dimensions referring, for example, to data from 3 sites at 5 times under 2 conditions (burnt and unburnt).

Arrays are defined using the `array()` function, in this case a 3x3x2 dimensional array.

```
> b <- array(count, c(3,3,2))
> b
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
Site_3      12      15      18
```

As with matricies, or higher level arrays, the array function cycles through rows, columns, then slices etc, varying through the first array index first, then the second, then the third and so on.

Referencing array values

Because arrays are vectors with more complex indexing, it should come as no surprise that you can access the contents of the array as if it were a vector.

```
> b[15]
[1] 15
```

More conveniently, the contents of an array would be accessed using the three index values

```
> b[3,2,2]
[1] 15
```

You can pull out a whole row with

```
> b[2,,2]
[1] 11 14 17
```

or a whole column with

```
> b[,3,2]
[1] 16 17 18
```

or a whole slice with

```
> b[,,1]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Adding Row and Column Names

When the dimension of the array is greater than two, names can be assigned to rows, columns and levels of higher dimensions by passing a list of character vectors as follows.

```
> dimnames(b) <- list(c("Site_1", "Site_2", "Site_3"),
     c("Winter", "Spring", "Summer"), c("Burnt", "Unburnt"))
> b
, , Burnt

       Winter Spring Summer
Site_1      1      4      7
Site_2      2      5      8
Site_3      3      6      9

, , Unburnt

       Winter Spring Summer
Site_1     10     13     16
Site_2     11     14     17
```

Once you have named the rows and columns, the above references can be referenced by name. The row and column names should be unique if they are to be used in referencing. The statements in the previous section can be alternatively represented by

```
> b["Site_3","Spring","Unburnt"]
[1] 15
```

```
> b["Site_2",,"Unburnt"]
Winter Spring Summer
    11     14     17
```

```
> b[,"Summer","Unburnt"]
Site_1 Site_2 Site_3
    16     17     18
```

```
> b[,,"Burnt"]
       Winter Spring Summer
Site_1      1      4      7
Site_2      2      5      8
Site_3      3      6      9
```

Arrays will be further discussed in the context of conditional branching and dataframes, after we consider the simplest form taken by raw data.

Where have we come?

You should now be aware that R programming, like any programming, uses a set of **keywords** and **operators** combined to form statements or instructions. These instructions are arranged in a clearly defined **sequence** in which they are executed. The language includes mechanisms for **branching**, **looping**, and for isolating blocks of code, **functions**, to be called on repeatedly as required.

In addition, R has specific **data structures** and means for **input** of data to those structures and for **output** of results.

Specifically, this lesson provided an overview of the simplest of data structures used in R programming. Having completed this lesson, you should now be familiar the following concepts.

- R is a programming language. As such it has a vocabulary of keywords, many of them function names provided in the base function library.
- R works with objects, and at this stage you know of the objects called vectors, factors and arrays.
- Vectors are simply ordered lists of numbers, character strings or logical values. Vectors can contain values that are numeric, character or logical, but these cannot be admixed in the one vector.
- Factors are vectors where the values take on one of several specified factor levels.
- Arrays are matricies of 2 dimensions or blocks of data of 3 or more dimensions.

Where to Now?

Next we learn of more complex datastructures that are a special feature of R, dataframes and lists.

Lesson 3: Dataframes and Lists

Raw Data

Some Sample Data

When first using R, the raw data are best arranged as a rectangular block made up of a series of rows (referred to as observations or entities) and fields (referred to as variables or attributes). Matters are much simpler if the raw data are arranged in this form, because this form is amenable to being input to an array, or as we will see, a special data structure called a **dataframe**.

The data in Table 1-1 are from a study of the pig-nosed turtle *Carettochelys insculpta* in Kakadu National Park.

The data are arranged so that each row contains data collected from an individual turtle. The measurements for each turtle are lined up to form columns of values.

Note that there is an admixture of integer, numeric and character data. This distinguishes dataframes from matricies.

The first variable contains the number of the tag attached to the turtle. The second variable contains the sex of each individual, MALE for mature males, FEMALE for mature females. Juvenile individuals cannot be reliably sexed, so no sex has been recorded. The third variable contains shell lengths in cm, the fourth variable contains head widths in cm and the last variable contains body weights in kg.

Table 1-1. Measurements of the pig-nosed turtle, Carettochelys insculpta.

Tag Number	Sex	Carapace Length (cm)	Head Width (cm)	Weight (kg)
10	MALE	41.0	7.15	7.60
11	FEMALE	46.4	8.18	11.00
2		24.3	4.42	1.65
15		28.7	4.89	2.18
16		32.0	5.37	3.00
3	FEMALE	42.8	7.32	8.60
4	MALE	40.0	6.60	6.50
5	FEMALE	45.0	8.05	10.90
12	FEMALE	44.0	7.55	8.90
13		28.0	4.85	1.97
6	FEMALE	40.0	6.53	6.20
8		32.0	5.35	2.90
9	MALE	35.0	5.74	3.90
17	FEMALE	35.1	6.04	4.50
19	MALE	42.3	6.77	7.80
22	FEMALE	48.1	8.55	12.80
105	MALE	44.0	7.10	9.00
14	MALE	43.0	6.60	7.20
7	FEMALE	48.0	8.67	13.50
1		29.2	5.10	2.38
104	MALE	44.0	7.35	9.00

Data Format

These data need to be converted to a form suitable for input to R. The simplest way is to prepare the data as a block of numbers (a two-dimensional array), delimited by spaces, with single word names at the head of each column. **Missing values** are represented by the special R keyword NA.

Table 1-1. Measurements of the pig-nosed turtle, Carettochelys insculpta presented as a text file suitable for reading into R. The even spacing of the columns is optional, to assist readability.

```
idno sex    length hwidth weight
10   MALE   41.0   7.15   7.60
11   FEMALE 46.4   8.18  11.00
2    NA     24.3   4.42   1.65
15   NA     28.7   4.89   2.18
16   NA     32.0   5.37   3.00
3    FEMALE 42.8   7.32   8.60
4    MALE   40.0   6.60   6.50
5    FEMALE 45.0   8.05  10.90
12   FEMALE 44.0   7.55   8.90
13   NA     28.0   4.85   1.97
6    FEMALE 40.0   6.53   6.20
8    NA     32.0   5.35   2.90
9    MALE   35.0   5.74   3.90
17   FEMALE 35.1   6.04   4.50
19   MALE   42.3   6.77   7.80
22   FEMALE 48.1   8.55  12.80
105  MALE   44.0   7.10   9.00
14   MALE   43.0   6.60   7.20
7    FEMALE 48.0   8.67  13.50
1    NA     29.2   5.10   2.38
104  MALE   44.0   7.35   9.00
```

These data would normally be held in a text file or in an excel spreadsheet and saved as comma delimited (csv format). They can also be part of your script.

Data Frames

What is a Dataframe?

The **dataframe** is, as with a vector, a type of R object, but one that is more complex. It is the primary means by which we will manage our datasets.

Dataframes are special R objects designed to hold data in a form that comprises a block of observations or **entities** (rows) each with values for their **attributes**, sometimes referred to as factors or variables (columns). Names can be given to both the columns (attributes) and the rows (entities).

A dataframe differs from a matrix or array in that it can contain an admixture of data types, and so helps us manage the associations between character vectors (`sex` for example) and numeric data (`length, head width` and `weight`).

Importing Data to a Dataframe

The `read.table()` function is used to import data into a dataframe (see Input and Output). This can be done directly for small datasets

```
> turtle <- read.table(header=TRUE, text="
           idno sex length hwidth weight
            10   MALE   41.0   7.15   7.60
           11   FEMALE 46.4   8.18  11.00
           2    NA     24.3   4.42   1.65
           15   NA     28.7   4.89   2.18
           16   NA     32.0   5.37   3.00
           3    FEMALE 42.8   7.32   8.60
           4    MALE   40.0   6.60   6.50
           5    FEMALE 45.0   8.05  10.90
           12   FEMALE 44.0   7.55   8.90
           13   NA     28.0   4.85   1.97
           6    FEMALE 40.0   6.53   6.20
           8    NA     32.0   5.35   2.90
           9    MALE   35.0   5.74   3.90
           17   FEMALE 35.1   6.04   4.50
           19   MALE   42.3   6.77   7.80
           22   FEMALE 48.1   8.55  12.80
           105  MALE   44.0   7.10   9.00
           14   MALE   43.0   6.60   7.20
           7    FEMALE 48.0   8.67  13.50
           1    NA     29.2   5.10   2.38
           104  MALE   44.0   7.35   9.00
        ")
```

The first step is to examine the contents of the dataframe to see if it has been input correctly.

```
> turtle
1  idno    sex length hwidth weight
2    10   MALE   41.0   7.15   7.60
3    11 FEMALE   46.4   8.18  11.00
4     2   <NA>   24.3   4.42   1.65
5    15   <NA>   28.7   4.89   2.18
6    16   <NA>   32.0   5.37   3.00
7     3 FEMALE   42.8   7.32   8.60
8     4   MALE   40.0   6.60   6.50
9     5 FEMALE   45.0   8.05  10.90
10   12 FEMALE   44.0   7.55   8.90
11   13   <NA>   28.0   4.85   1.97
12    6 FEMALE   40.0   6.53   6.20
13    8   <NA>   32.0   5.35   2.90
14    9   MALE   35.0   5.74   3.90
15   17 FEMALE   35.1   6.04   4.50
16   19   MALE   42.3   6.77   7.80
17   22 FEMALE   48.1   8.55  12.80
18  105   MALE   44.0   7.10   9.00
19   14   MALE   43.0   6.60   7.20
20    7 FEMALE   48.0   8.67  13.50
21    1   <NA>   29.2   5.10   2.38
22  104   MALE   44.0   7.35   9.00
```

Referencing Data in a Dataframe

Index Referencing

The data in a dataframe comprises only the data and not the column headings, even though these were in the original dataset. Hence, `turtle[1,1]` holds the value `10`.

This means that we can apply all that we have learned on referencing data within a matrix. We can reference values directly using index numbers.

```
> turtle[6,2]
[1] FEMALE
Levels: FEMALE MALE
```

The character variable `sex` has been stored in the dataframe as a factor. It is not a character vector, but a factor with the factor levels `MALE` and `FEMALE`.

You can pull out a whole row with

```
> turtle[2,]
   idno    sex length hwidth weight
11   11 FEMALE   46.4   8.18     11
```

or a whole column with

```
> turtle[,2]
[1] MALE FEMALE <NA> <NA> <NA> FEMALE MALE FEMALE FEMALE <NA> FEMALE <NA>
MALE FEMALE MALE FEMALE MALE MALE FEMALE <NA> MALE
Levels: FEMALE MALE
```

Label Referencing

The labels in the first row of the datafile have been assigned to column labels because we had `header=TRUE` in the script that read the data. The row labels have been assigned the values “1” to “22”. It would be sensible in this case to assign the turtle identity number to the rows.

```
> rownames(turtle)<- turtle[,1]
> turtle
    idno    sex length hwidth weight
10    10   MALE   41.0   7.15   7.60
11    11 FEMALE   46.4   8.18  11.00
2      2   <NA>   24.3   4.42   1.65
15    15   <NA>   28.7   4.89   2.18
16    16   <NA>   32.0   5.37   3.00
3      3 FEMALE   42.8   7.32   8.60
4      4   MALE   40.0   6.60   6.50
5      5 FEMALE   45.0   8.05  10.90
12    12 FEMALE   44.0   7.55   8.90
13    13   <NA>   28.0   4.85   1.97
6      6 FEMALE   40.0   6.53   6.20
8      8   <NA>   32.0   5.35   2.90
9      9   MALE   35.0   5.74   3.90
17    17 FEMALE   35.1   6.04   4.50
19    19   MALE   42.3   6.77   7.80
22    22 FEMALE   48.1   8.55  12.80
105  105   MALE   44.0   7.10   9.00
14    14   MALE   43.0   6.60   7.20
7      7 FEMALE   48.0   8.67  13.50
1      1   <NA>   29.2   5.10   2.38
104  104   MALE   44.0   7.35   9.00
```

Now we can conveniently reference the data as we did with a matrix earlier, using the row and column names.

```
> turtle["105", "weight"]
[1] 9
```

```
> turtle[“3”,”sex”]
[1] FEMALE
Levels: FEMALE MALE
```

```
> turtle[“11”,]
   idno    sex length hwidth weight
11   11 FEMALE   46.4   8.18     11
```

```
> turtle[,”sex”]
[1] MALE FEMALE <NA> <NA> <NA> FEMALE MALE FEMALE FEMALE <NA> FEMALE <NA>
MALE FEMALE MALE FEMALE MALE MALE FEMALE <NA> MALE
Levels: FEMALE MALE
```

Note that R implicitly coerced the integers that are in the dataset as `idno` into characters for the column labels when R executed the rownames statement above.

We could have done this explicitly with

```
> rownames(turtle)<- as.character(turtle[,1])
```

Shorthand Conventions

R has a shorthand convention for referring to data columns in a dataframe.

```
> turtle$length
 [1] 41.0 46.4 24.3 28.7 32.0 42.8 40.0 45.0 44.0 28.0 40.0 32.0 35.0 35.1
42.3 48.1 44.0 43.0 48.0 29.2 44.0
```

refers to the turtle lengths as a variable in the dataframe. This is equivalent to

```
> turtle[,”length”]
```

or

```
> turtle[,3]
```

The convenience of this is carried further if you attach your dataframe to your workspace

```
> attach(turtle)
```

Now if you wish to refer to the length data in computations, you can simply use

```
> length
[1] 41.0 46.4 24.3 28.7 32.0 42.8 40.0 45.0 44.0 28.0 40.0 32.0 35.0 35.1
42.3 48.1 44.0 43.0 48.0 29.2 44.0
```

This greatly increases readability of R statements.

Dataframes can be detached using

```
> detach(turtle)
```

So in summary, the benefit of using dataframes is that you can mix data of varying types (classes) in the one matrix, assign column and row names, use all of the referencing conventions that apply to matricies, but in addition, take advantage of the shorthand referencing of variables using the $ operator and `attach()` and `detach()` functions.

How R Handles Missing Values

What symbol represents missing data?

You will have noticed in the example above that the keyword `NA` is recognised by R as a missing value – the sex of juveniles is unknown. Some R commands take into account missing values, such as `summary()`, while others do not, such as `mean()`.

Do not use the string "NA" to represent missing data in your program statements. Use the keyword `NA`. So an appropriate assignment of the fourth value of the vector `weight` to missing is

```
 weight[4] <- NA
```

and not

```
 weight[4] <- "NA"
```

Identifying missing data

The function `is.na()` is used to determine which values of a variable are missing and which have data. For example,

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

Many R functions have an option to specify whether or not to exclude missing values from computations, for example

```
   mean(weight,na.rm=TRUE)
```

Importing data

The read.table function

We have seen one way of importing data to a dataframe, using the `read.table()`function

```
turtle <- read.table(header=TRUE, text="
               idno sex length hwidth weight
               10   MALE   41.0   7.15   7.60
               11   FEMALE 46.4   8.18  11.00
               2    NA     24.3   4.42   1.65
               15   NA     28.7   4.89   2.18
               16   NA     32.0   5.37   3.00
               3    FEMALE 42.8   7.32   8.60
               4    MALE   40.0   6.60   6.50
               5    FEMALE 45.0   8.05  10.90
               12   FEMALE 44.0   7.55   8.90
               13   NA     28.0   4.85   1.97
               6    FEMALE 40.0   6.53   6.20
               8    NA     32.0   5.35   2.90
               9    MALE   35.0   5.74   3.90
               17   FEMALE 35.1   6.04   4.50
               19   MALE   42.3   6.77   7.80
               22   FEMALE 48.1   8.55  12.80
               105  MALE   44.0   7.10   9.00
               14   MALE   43.0   6.60   7.20
               7    FEMALE 48.0   8.67  13.50
               1    NA     29.2   5.10   2.38
               104  MALE   44.0   7.35   9.00
                 ")
```

This may be fine for small datasets, but is not convenient in most cases. Data usually are kept in a separate file on disk, usually in an Excel spreadsheet. There they can be readily accessed by a variety of computer packages, including R.

File specification

R does not recognise the *drive:\path\filename.ext* conventions of the Windows environment. R uses forward slashes in place of reverse slashes, that is *drive://path/filename.ext.* For example

```
turtle <- read.table(“c://caretto.csv”, sep=”,”, header=TRUE)
```

This statement assigns the data in the raw datafile `caretto.csv` to the dataframe `turtle`.

After reading the data in, you can verify the contents of the dataframe by using

```
> turtle
1  idno    sex length hwidth weight
2    10   MALE   41.0   7.15   7.60
3    11 FEMALE   46.4   8.18  11.00
4     2   <NA>   24.3   4.42   1.65
5    15   <NA>   28.7   4.89   2.18
6    16   <NA>   32.0   5.37   3.00
7     3 FEMALE   42.8   7.32   8.60
8     4   MALE   40.0   6.60   6.50
9     5 FEMALE   45.0   8.05  10.90
10   12 FEMALE   44.0   7.55   8.90
11   13   <NA>   28.0   4.85   1.97
12    6 FEMALE   40.0   6.53   6.20
13    8   <NA>   32.0   5.35   2.90
14    9   MALE   35.0   5.74   3.90
15   17 FEMALE   35.1   6.04   4.50
16   19   MALE   42.3   6.77   7.80
17   22 FEMALE   48.1   8.55  12.80
18  105   MALE   44.0   7.10   9.00
19   14   MALE   43.0   6.60   7.20
20    7 FEMALE   48.0   8.67  13.50
21    1   <NA>   29.2   5.10   2.38
22  104   MALE   44.0   7.35   9.00
```

Adding Rows and Columns to a Dataframe

The `rbind()` function can be used to join two like dataframes together, one to follow the other, or to bind a vector to a dataframe as an additional row. The vector needs to be of the same length as the number of columns in the dataframe, and to have data of appropriate type in each column. Character variables in a dataframe are often considered factors and their values as factor levels. Any character values added may need to have compatible factor levels.

There is an equivalent command called `cbind()` which can be used to add vectors to a dataframe as new variables. These vectors need to be the same length as the variables already in the dataframe.

Assigning names to variables

Sometimes data are held in a file without variable names. The raw data file `caretto_nn.csv` contains the same data as `caretto.csv`, but without the variable names heading up each column. We can still read these data in and give the variables names subsequently.

```
turtle <- read.table("caretto_nn.csv", sep=”,”)
names(turtle) <- c("idno","sex","length","hwidth","weight")
```

You can again verify this by examining the contents of the dataframe `turtle`.

Reading from Excel

R can access data directly from Microsoft Excel spreadsheets, which is convenient, but the simplest method is to save your files as comma delimited prior to reading them in with R as illustrated above.

Alternatively, you can copy data then read it from the clipboard. Make sure that the data are in block form (a rectangular matrix comprising observations as rows and attributes as columns). Make sure the column names conform to the format expected by R for object names (no spaces).

Select only the cells containing data (don't select any extra blank rows or columns) and copy them to the clipboard. The contents of the clipboard can then be transferred to an R dataframe using the `read.delim()` function.

```
mydata <- read.delim("clipboard")
```

The object `mydata` will contain the excel data including column names and will be included in your workspace.

Exporting data

The write.table function

We use the `write.table()`function to export data to a file, tab delimited for example.

```
write.table(turtle, "mydata.txt", sep="\t")
```

or as a comma delimited csv file

```
write.table(turtle, "mydata.csv", sep=",")
```

The new data file will appear in your default directory.

Lists

What is a List?

Lists are vectors of objects of various types. For example, a list may comprise an ordered series of scalers, vectors, arrays, factors and dataframes.

For example, we could create a list called cabinet, and assign to it the various objects we have created so far.

```
> cabinet <- list(lengths=beetles, counts=b, sex=sexcode)
> cabinet
$lengths
[1] 15.2 12.1 17.8 13.9 16.4 15.1

$counts
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18


$sex
[1] F F M J F M J J F
Levels: F J M
```

Referencing Contents of a List

You can extract an object from a list using

```
> cabinet[[3]]
```

```
[1] F F M J F M J J F
Levels: F J M
```

but a more convenient way is to take advantage of the list labels.

```
> cabinet$lengths
[1] 15.2 12.1 17.8 13.9 16.4 15.1
```

Accessing values from an object within a list is now intuitive, I hope.

```
> cabinet$lengths[4]
[1] 13.9
```

Where have we come?

You should now be aware that R programming, like any programming, uses a set of **keywords** and **operators** combined to form statements or instructions. These instructions are arranged in a clearly defined **sequence** in which they are executed. The language includes mechanisms for **branching**, **looping**, and for isolating blocks of code, **functions**, to be called on repeatedly as required.

In addition, R has specific **data structures** and means for **input** of data to those structures and for **output** of results.

Specifically, this lesson provided an overview of the data structures commonly used in R programming. Having completed this lesson, you should now be familiar the following concepts.

- R is a programming language. As such it has a vocabulary of keywords, many of them function names provided in the base function library.
- R works with objects, and at this stage you need to know of the objects called vectors, factors, arrays, lists and dataframes.
- Vectors are simply ordered lists of numbers, character strings or logical values. Vectors can contain values that are numeric, character or logical, but these cannot be admixed in the one vector.
- Factors are vectors where the values take on one of several specified factor levels.
- Arrays are matricies of 2 dimensions or blocks of data of 3 or more dimensions.
- Lists are like vectors but the elements can be objects of any kind, an admixture of vectors (numeric, character or logical), arrays, lists or dataframes.
- Dataframes are special structures for working with data comprised of a set of observations or entities (rows) each with values for each of a set of variables or attributes (columns).

Finally, you have some simple tools for importing data from a file and for exporting data to a file.

Lesson 2 was an overview. The best way of learning is by doing, and it is when you start using these data structures to solve programming challenges that you will see their value.

Where to Now?

Now it is time to move on to learn some programming.

Lesson 4: Programming basics

Once the data are read into a dataframe, we can proceed to analyse them with the fundamental tools provided as part of the R base library or with the tools from any additional libraries we have chosen to load.

Keywords and Operators

Keywords

R has remarkably few reserved keywords, but use of these as object or user-defined function names must be avoided. They include

```
TRUE FALSE NULL NA NaN Inf
if else repeat while for in next break function pi
```

It is not necessary, but wise to avoid using function names defined in loaded packages as object names or as the names of your own functions. Some people avoid the confusion by using uppercase names for all of their objects and personal functions.

Assignment statements

The most common R statement used in programming is the assignment statement. For example, assignment statements can be used to create new variables:

```
 lglength <- log10(length)
```

This means literally, take the contents of object `length`, log the values (base 10) and place the logged values in new object `lglength`.

Operators

The standard operators recognized by R are

Operator	**Description**
+	addition
-	subtraction
*	multiplication
/	division
^ **or** **	exponentiation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to [NOTE!!]
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y

Note that the logical operator for equality is not = but ==.

Branching

if-else Statements

As with any programming language, R has statements that allow for branching, that is, executing statements provided some condition is met. In fact R has an abundance of approaches to this.

One way is to use IF-THEN-ELSE constructs:

```
if (sex == "MALE"){
        condition <- 0.00140*length^2.91 - weight
        }
else if(sex == "FEMALE"){
        condition <- 0.00135*length^2.85 - weight
        }
else {
        condition <- NA
}
```

Conditional Indexing

A second approach is to access the dataframe as an array `turtle[i,j]`, putting conditions on the index variables `i` or `j`.

```
turtle[sex=="FEMALE",]
```

will print out only those rows for which `sex=="FEMALE"`. The conditional statement can be quite complex

```
turtle[sex=="female" & length>40.0,].
```

We can select only those observations for which an indicator variable takes on particular values,

```
turtle[is.element(idno, c(17,19, 22,105)),]
```

and of course, these restrictive references to the data can be used inside other functions,

```
summary(turtle[is.element(idno, c(17,19, 22,105)),])
```

Subsetting

A third approach is to split the data into subsets for further analysis

```
males <- subset(turtle, sex == "MALE")
females <- subset(turtle, sex == "FEMALE")
```

A forth approach is to apply functions such as `summary()`separately to each sex with the `tapply()` function

```
tapply(length, sex, summary)
```

Other approaches

There are still other appoaches using the `by()` function or the `by` option within functions. It will take time to become familiar with these different options and when they are each most efficiently applied.

Looping

Loops

As with any programming language, R has statements that allow for looping, that is, repeating blocks of code. This is possible with one of the DO-WHILE, DO-UNTIL OR DO-FOR constructs in R, for example

```
for (i in 1:length(species)){
        if (species[i] == "radiata") {
                yield <- density*3.1416*diameter*height*0.85
        }
        else {
                yield <- density*3.1416*diameter*height*0.62
        } # Terminate the if statement
} # Terminate the for loop
```

Functionals

Seasoned R programmers draw upon an alternative approach to using the above constructs. They use instead functionals (so called because of the analogy with mathematical functions of functions).

The most frequently used is `lapply()`..`lapply()`takes a function, applies it to each element in a list, and returns the results in the form of a list. For example, to round all the length values in dataframe turtle, you could use

```
for (i in 1:length(turtle$length)){
  turtle$length <- round(turtle$length)
}
```

or

```
turtle$length <- lapply(turtle$length, round)
```

This is rather a trivial example, but applied to more complex functions, including those of your own, `lapply()` is a concise and powerful tool and an alternative to conventional looping that is more efficient and improves code readability.

Analysis

Once our data manipulation is completed, we can get down to analysis. We can plot the data.

```
plot(length, weight)
```

and undertake all manner of statistical analyses.

```
t.test(weight ~ sex)
```

which brings us to the end of these introductory lessons.

Where have we come?

The above lesson was designed to give you an overview of the operation of R. It sets the scope of what you need to learn to develop R programming skills. Having completed this lesson, you should now be familiar the following concepts.

- R functions are used to construct R commands.

- As a programming language, R has the means for manipulating data through assignment statements, for repeating segments of program code with `for` and `while` constructs, and for selectively applying segments of code with `if-else` constructs or selection within R functions like `subset()`and `tarray()`.

More detail on programming in R can be found in the very readable text Venables, WN and Ripley, BD (2002). Modern Applied Statistics with S. Fourth Edition. Springer, New York. Chapters 1-4.

Where to next?

The lessons up to this point provided an overview. There is only so much that can be done by reading about R. The best way of learning is by doing, and it is now time to run through what you have learned with some step-by-step examples.

Lesson 5: Practical programming

Getting Started

Firing up R-studio

If R-studio is properly installed, you should then be able to run it by double-clicking on the relevant icon on the desktop or in an Applications Group window. Consult the computer systems officer at your institution for details of how to invoke R.

Download the data for the course and place the files in a directory of your choice.

Double-click on the R icon on your desktop.

Throughout this module, your action is required only when you encounter instructions inside an Activity Box like this one.

R-studio windows

Console, Environment and Files

Once R has commenced running, the program will display three windows (refer to Figure 1-1) – the Console, an environment window and two other windows with environment and file display information.

Setting your Working Directory

You first need to set your working directory as the default. Move the files view to your working directory, then select “Files>More>Set As Working Directory”.

This directory will be linked to this analysis when you later save an image of your workspace.

There are a number of other fairly self-evident buttons on the Menu Bars. The Console has the greeting message, and awaits your commands.

Program Editor

The Program Editor does not open unless requested and is used to create R programs.

Open an EDITOR window by selecting "File>New File>R Script" from the File Menu on the top Menu Bar.

Workflow

To demonstrate how the R-studio windows system operates, an R program to perform a simple task has been provided. You must first import this program to the Editor, so that you may peruse it and ultimately submit it for execution.

Select Open Script from the File Menu to locate and select the file `demo1.R`

Note that several lines of code, an R program, have appeared in the R Editor. Do not worry about what the lines mean at this stage.

An R program must be submitted before it is executed.

Place the cursor on the first line of the program and submit the program line by line with control-r (^r).

Note that the progress of the analysis in the R Console. Refer to the output.You should be able to see a listing of the data, a statistical summary, some correlation analyses, and a bivariate graph.

You should now be familiar with the R-studio graphical user interface, and how the various windows relate to each other.

Creating a data file

Input your data

You can create data files before running R by using your favourite editor, spreadsheet or word processor (in text only mode).

Recall the data in Table 1-2, from a study of the pig-nosed turtle, *Carettochelys insculpta*, in Kakadu National Park.

Table 1-2. Measurements of the pig-nosed turtle, Carettochelys insculpta.

Tag Number	Sex	Carapace Length (cm)	Head Width (cm)	Weight (kg)
10	MALE	41.0	7.15	7.60
11	FEMALE	46.4	8.18	11.00
2		24.3	4.42	1.65
15		28.7	4.89	2.18
16		32.0	5.37	3.00
3	FEMALE	42.8	7.32	8.60
4	MALE	40.0	6.60	6.50
5	FEMALE	45.0	8.05	10.90
12	FEMALE	44.0	7.55	8.90
13		28.0	4.85	1.97
6	FEMALE	40.0	6.53	6.20
8		32.0	5.35	2.90
9	MALE	35.0	5.74	3.90
17	FEMALE	35.1	6.04	4.50
19	MALE	42.3	6.77	7.80
22	FEMALE	48.1	8.55	12.80
105	MALE	44.0	7.10	9.00
14	MALE	43.0	6.60	7.20
7	FEMALE	48.0	8.67	13.50
1		29.2	5.10	2.38
104	MALE	44.0	7.35	9.00

Use your favourite editor to type the data in in a form suitable for reading into R. Remember, missing values in R are represented by the keyword `NA`.

You might wish to line the data up in columns for ease of reading and correcting, but R requires only that the numbers and character strings be separated from each other by one or more blanks.

Use a suitable editor to create the data file in a form conducive to reading in to R. Do not include variable names in the file.

Save your data

Once you have checked your typing and corrected any errors, you can save the data in a disk file in the directory in which you have chosen to store your files. Call it `mydata.dat`.

Save the contents to the file mydata.dat.

Note

You should follow the convention of using .DAT as the extension to all files containing raw data.

Close the external editor and return to R.

Creating a program

Creating an R script or program

The usual way to create programs is in the R Editor, rather than an external editor. The following is a sample R program.

```
# Read in the data
 turtle <- read.table("mydata.dat")
# Add variable names
 names(turtle) <- c("idno","sex","length","hwidth","weight")
# Create two new variables for later use
 turtle$lglength <- log10(turtle$length)
 turtle$lgweight <- log10(turtle$weight)
# Calculate summary statistics
 summary(turtle)
```

Move to the R Editor and type in the above R program. For the sake of this exercise, do not submit the program statements for execution quite yet.

Note

Be very careful to balance the quotes when you use them. Omission of one of the quotes will lead to an obscure error message that will persist until the second quote is submitted or the command terminated with an escape (Esc).

Again, you don't need to worry at all about what the program means at this stage.

You can use your favourite editor or word processor to create an R program, provided that you can save it in text mode. You can then open the file into the Editor window.

Save your program

It is prudent to save a copy of your program on your data disk before submitting the program to R for execution.

Make sure that you are in the R Editor, then save the contents to the file `myprog.R`.

Note

You should follow the convention of using .R as the extension to all disk files containing R programs. It will save a great deal of confusion.

You now have two files on disk – your raw data file `mydata.dat`, and your R program, `myprog.R`. You can confirm this by examining the relevant directory using the Files tab on the Files Window toolbar.

Executing a program

Line by line execution

An R program must be submitted for execution. This is done line by line with control-r (^r) or by highlighting segments of the program and typing ^r.

During execution, the program will remain in the R Editor for editing and re-submission, if things go awry. This way, as you amend the program to fix errors, you build up a working copy of the final program. It is prudent to save a copy of your amended program periodically.

Submit your program for execution line by line, using ^r.

Lets now submit the first few lines of the program, those that set up the dataframe.

```
# Read in the data
 turtle <- read.table("mydata.dat")
# Add variable names
 names(turtle) <- c("idno","sex","length","hwidth","weight")
```

Highlight the above segment of your program as it appears in the R Editor, and use ^r to submit it for execution.

Later, you can execute the whole program using the [Run] tab on the Editor toolbar.

Quite a number of actions are taken when you submit a program for execution. First, a log of the progress of the program will appear in the Console. You should take note of any errors, as these indicate a fatal problem with your program. Warnings should also be heeded, as they indicate that the syntax is correct, but the analysis itself may have problems.

Second, a number of objects are typically created, depending upon the instructions received by your program. These reside in your workspace and will appear in the Environment Window under “Global Environment”. Among these objects will be your dataframe under the name you assigned it in the `read.table()` function.

Examining your workspace

Alternatively, you can confirm that the dataframe turtle, specified in the `read.table()` assignment of your program has indeed been created by examining the list of objects in your workspace.

```
 ls()
[1] "turtle"
```

Submit the above command to the R Console

You can peruse the contents of the dataframe you have created, by submitting its name to the command prompt.

```
 turtle
   idno    sex length hwidth weight
1    10   MALE   41.0   7.15   7.60
2    11 FEMALE   46.4   8.18  11.00
3     2   <NA>   24.3   4.42   1.65
4    15   <NA>   28.7   4.89   2.18
5    16   <NA>   32.0   5.37   3.00
6     3 FEMALE   42.8   7.32   8.60
7     4   MALE   40.0   6.60   6.50
8     5 FEMALE   45.0   8.05  10.90
9    12 FEMALE   44.0   7.55   8.90
10   13   <NA>   28.0   4.85   1.97
11    6 FEMALE   40.0   6.53   6.20
12    8   <NA>   32.0   5.35   2.90
13    9   MALE   35.0   5.74   3.90
14   17 FEMALE   35.1   6.04   4.50
15   19   MALE   42.3   6.77   7.80
16   22 FEMALE   48.1   8.55  12.80
17  105   MALE   44.0   7.10   9.00
18   14   MALE   43.0   6.60   7.20
19    7 FEMALE   48.0   8.67  13.50
20    1   <NA>   29.2   5.10   2.38
21  104   MALE   44.0   7.35   9.00
```

Submit the above command to the R Console

Creating new variables

The next block of program code creates two new variables by transforming existing variables in our dataframe turtle.

```
# Create two new variables for later use
 turtle$lglength <- log10(turtle$length)
 turtle$lgweight <- log10(turtle$weight)
```

Highlight the above segment of your program as it appears in the R Editor, and use ^r to submit it for execution.

What we are asking here is for R to take each of the values of the vector `length` in dataframe `turtle`, log it to base 10, and place the transformed values in the new vector `lglength`, also within the dataframe `turtle`. Ditto for weight.

Again, examine the dataframe to see the outcome of your instructions. The additional variables have been added to the dataframe.

```
 turtle
   idno    sex length hwidth weight lglength  lgweight
1    10   MALE   41.0   7.15   7.60 1.612784 0.8808136
2    11 FEMALE   46.4   8.18  11.00 1.666518 1.0413927
3     2   <NA>   24.3   4.42   1.65 1.385606 0.2174839
4    15   <NA>   28.7   4.89   2.18 1.457882 0.3384565
5    16   <NA>   32.0   5.37   3.00 1.505150 0.4771213
6     3 FEMALE   42.8   7.32   8.60 1.631444 0.9344985
7     4   MALE   40.0   6.60   6.50 1.602060 0.8129134
8     5 FEMALE   45.0   8.05  10.90 1.653213 1.0374265
9    12 FEMALE   44.0   7.55   8.90 1.643453 0.9493900
10   13   <NA>   28.0   4.85   1.97 1.447158 0.2944662
11    6 FEMALE   40.0   6.53   6.20 1.602060 0.7923917
12    8   <NA>   32.0   5.35   2.90 1.505150 0.4623980
13    9   MALE   35.0   5.74   3.90 1.544068 0.5910646
14   17 FEMALE   35.1   6.04   4.50 1.545307 0.6532125
15   19   MALE   42.3   6.77   7.80 1.626340 0.8920946
16   22 FEMALE   48.1   8.55  12.80 1.682145 1.1072100
17  105   MALE   44.0   7.10   9.00 1.643453 0.9542425
18   14   MALE   43.0   6.60   7.20 1.633468 0.8573325
19    7 FEMALE   48.0   8.67  13.50 1.681241 1.1303338
20    1   <NA>   29.2   5.10   2.38 1.465383 0.3765770
21  104   MALE   44.0   7.35   9.00 1.643453 0.9542425
```

Submit the above command to the R Console

Working with your data

Now we are ready to calculate some summary statistics.

```
# Calculate summary statistics
 summary(turtle)
      idno            sex         length          hwidth          weight
 Min.   :  1.00   FEMALE:8   Min.   :24.30   Min.   :4.42   Min.   : 1.650
 1st Qu.:  6.00   MALE  :7   1st Qu.:32.00   1st Qu.:5.37   1st Qu.: 3.000
 Median : 11.00   NA's  :6   Median :41.00   Median :6.60   Median : 7.200
 Mean   : 19.19              Mean   :38.71   Mean   :6.58   Mean   : 6.737
 3rd Qu.: 16.00              3rd Qu.:44.00   3rd Qu.:7.35   3rd Qu.: 9.000
 Max.   :105.00              Max.   :48.10   Max.   :8.67   Max.   :13.500
    lglength         lgweight
 Min.   :1.386   Min.   :0.2175
 1st Qu.:1.505   1st Qu.:0.4771
 Median :1.613   Median :0.8573
 Mean   :1.580   Mean   :0.7502
 3rd Qu.:1.643   3rd Qu.:0.9542
 Max.   :1.682   Max.   :1.1303
```

Highlight the above segment of your program as it appears in the R Editor, and use ^r to submit it for execution.

Executing programs in R in this way allows progressive debugging and refinement of the program code which resides in the R editor. You should now save the program again for future use.

Go to the R Editor and save the program.

When things go wrong

There is a dreaded shadow that hangs over all who engage in computing — SYNTAX. If you don't get it right, the program will not work. Finding out why it will not work is not always easy, though many simple mistakes will be immediately evident.

In the example you have just run, there should have been no errors. Had there been an error, you would need to examine the content of the Console for an error message. Several problems are quite common.

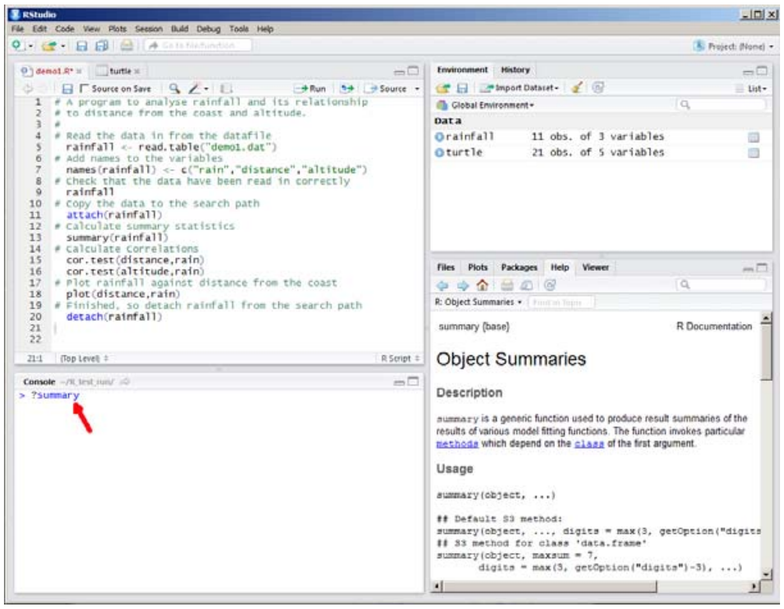
Incorrect syntax for function calls

R functions are very particular in what they expect as arguments, and often the order of the arguments is quite important. When a function fails, the first step is to call up the help files using the `?` prefix. An example is

```
?summary
```

This command opens up the help files in a new window. The help information contains full details of the syntax for the `summary()` function, which should enable you to identify and rectify the problem with your program.

Submit the above command to the R Console



```
# A program to analyse rainfall and its relationship
# to distance from the coast and altitude.
#
# Read the data in from the datafile
  rainfall <- read.table("demo1.dat")
# Add names to the variables
  names(rainfall) <- c("rain","distance","altitude")
# Check that the data have been read in correctly
  rainfall
# Copy the data to the search path
  attach(rainfall)
# Calculate summary statistics
  summary(rainfall)
# Calculate Correlations
  cor.test(distance,rain)
  cor.test(altitude,rain)
# Plot rainfall against distance from the coast
  plot(distance,rain)
# Finished, so detach rainfall from the search path
  detach(rainfall)
```

```
> ?summary
```

```
Data
rainfall      11 obs. of 3 variables
turtle        21 obs. of 5 variables
```

summary (base) R Documentation

Object Summaries

Description

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular methods which depend on the class of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits = max(3, getOption("digits
## S3 method for class 'data.frame'
summary(object, maxsum = 7,
        digits = max(3, getOption("digits")-3), ...)
```

Failure to Balance Quotes

String values are usually identified as such by enclosing them in quotes. If you fail to supply the terminal quote, a very common error, then R passes all that follows the initial quote into the string variable. Usually this is evident because the Console responds with a + rather than the usual command prompt >. You need to break out of this using the Esc key, locate and rectify the error, and resumit the code.

Failure to Balance Brackets

Many R commands comprise nested function calls, and so you have brackets within brackets. These two need to be balanced. Blocks of code also are contained within brackets ({ and }), and these need to be balanced. R will give you an explicit error message when you fail to balance brackets.

Mixing Up Characters with Numbers

Mis-spelling variable names and function names is a very common mistake, and one that requires attention to detail. A common mistake is to use a lowercase L (l) in place of the number one (1) or the letter O in place of the number 0. Look for this when your program does not work for reasons that are not immediately obvious.

Mixing Up Variable Names with Function Names.

It is best to avoid using the pre-existing names of R functions as names for your variables. You will find that we have done this in this Module – using `length` as the name of a variable when there exists a function `length()` that returns the number of values in a vector. This can lead to considerable confusion – `length(length)` for example – and is best avoided. One common way to address this issue is to use upper case for variable names and lower case for function names, though this is a nightmare for touch-typists.

Object Does Not Exist

This is a common error and can arise for a reason as simple as mis-spelling its name, or for more complex reasons such as the object not falling on the search path.

The first step is to check that the object exists by typing its name and submitting it to the R Console. This will verify its existence and display its contents.

If the object is part of a larger object, such as a dataframe, R might not be able to find it. You can identify this problem by typing its full specification, such as `turtle$weight` or by attaching the relevant data frame to the search path and trying again. More on this later.

Debugging

Debugging a program is what programmers do, but for those not used to programming, it can be a real block. The idea is to be systematic about your approach to identifying and rectifying bugs in your programs.

The best way is to run each line of your program one at a time, checking the progress of the analysis. Bugs often beget bugs, so it is important to start at the top of the program.

Bugs derived from incorrect syntax are the easiest to resolve. Those involving logical errors take a bit more mental energy.

Writing R Programs I

The Dataframe

You now know how to enter, edit and save data, to enter an R program, to submit it for execution, and to modify and re-submit R programs if they do not initially work. Your next step is to learn how to write sensible R programs.

Introducing the DataFrame

The program lines

```
turtle <- read.table("mydata.dat")
names(turtle) <- c("idno","sex","length","hwidth","weight")
turtle
```

create a special R object called a DataFrame, import the raw data into the DataFrame, assign names to columns of data in the DataFrame, then display the contents of the Dataframe.

Open a new script in the Editor, type in the above code, and submit it for execution.

The Dataframe can be considered to be an aggregation of vectors, with each column of data representing a vector. Clearly, these vectors need not be of the same type. For example, the variable `sex` is of type `factor` and the variable `length` is of type `numeric`.

You can refer to the variables in the dataset as vectors by adding the dataframe name as a prefix and separating the two with a dollar sign ($).

```
turtle$idno
[1]  10  11   2  15  16   3   4   5  12  13   6   8   9  17  19  22 105  14   7
[20]   1 104
```

Submit the above command to the Console

This is very convenient, as all manner of vector manipulations can be applied to variables within the dataframe.

Attaching Dataframes to the Search Path

Refering to variables with the dataframe name as a prefix can be tiresome, and you can avoid this by copying the dataframe to the search path used by R in locating objects.

Verify that the object `turtle$sex` exists by entering its name in the R Console. Also verify that the object `sex` is unrecognised.

The object `sex` is not recognised because the contents of the dataframe turtle are not recognised by R, they are not in R's search path. You can examine the objects that are in the search path using the command

```
search()
```

Submit the above command to the R Console.

The following output should appear in the R Console. By default, the search path includes the Global Environment at the top, and the R libraries at the bottom. The Global Environment is where all explicitly defined objects are placed by default.

```
search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

The command

```
attach(turtle)
```

will enable you to use the variable name `sex` in place of `turtle$sex`. You will find that attaching dataframes to the search path is pretty much essential.

If we now attach the dataframe `turtle` to the search path, all vectors contained within that dataframe will be recognised by R.

Attach the dataframe turtle to the search path, using `attach()` function described above.

Verify that the object `sex` is now recognised.

The attach function placed the dataframe `turtle` in the search path for R, and so when R encountered the command `sex`, it found the object sex in the dataframe `turtle`, and listed its contents.

Confirm that the search path now contains the dataframe turtle using the `search()` function.

Note that attach copies the dataframe to a location in the search path, so any variables you add to the dataframe will require you to attach it again if you want to access them without the $ prefix.

You can remove a dataframe from the search path using

```
detach(turtle)
```

Although we will not do it now, it is important to match each `attach()` call with a `detach()` call, lest you want a very cluttered search path. Failing to manage the search path is a major source of confusion for the novice user of R.

Accessing a Dataframe as an Indexed Array

Using the square brackets, you can access any particular value in the dataframe. The syntax is

```
dataframe[row, column]
```

For example, to access the value of the fifth row and the second column in the dataset, use

```
turtle[5,2]
```

or alternatively

```
turtle[5,"sex"]
```

The command

```
turtle[,"sex"]
```

is the same as `turtle$sex` and will print all values of the vector `sex`. Similarly,

```
turtle[5,]
```

will print all variables for observation 5. This is all very convenient, once you get used to it.

Exercise

Use array indexing to determine

(a) The sex of the 8th individual in the data set.
(b) The length of the 5th individual.
(c) All the data for the 3rd individual.
(d) Use the `mean()` function to get the mean weights

Accessing data within a dataframe is very flexible indeed, because you can use logical conditions as the indexing variable for rows (usually) or columns. For example,

```
turtle[sex=="FEMALE",]
     idno    sex length hwidth weight lglength  lgweight
2      11 FEMALE   46.4   8.18   11.0 1.666518 1.0413927
NA     NA   <NA>     NA     NA     NA       NA        NA
NA.1   NA   <NA>     NA     NA     NA       NA        NA
NA.2   NA   <NA>     NA     NA     NA       NA        NA
6       3 FEMALE   42.8   7.32    8.6 1.631444 0.9344985
8       5 FEMALE   45.0   8.05   10.9 1.653213 1.0374265
9      12 FEMALE   44.0   7.55    8.9 1.643453 0.9493900
NA.3   NA   <NA>     NA     NA     NA       NA        NA
11      6 FEMALE   40.0   6.53    6.2 1.602060 0.7923917
NA.4   NA   <NA>     NA     NA     NA       NA        NA
14     17 FEMALE   35.1   6.04    4.5 1.545307 0.6532125
16     22 FEMALE   48.1   8.55   12.8 1.682145 1.1072100
19      7 FEMALE   48.0   8.67   13.5 1.681241 1.1303338
NA.5   NA   <NA>     NA     NA     NA       NA        NA
```

will access data only for females. Note that it is a little untidy, in that all rows for which `sex != "FEMALE"` has the data set to missing (`NA`). A cleaner approach might be

```
turtle[is.element(sex, "FEMALE"),]
   idno    sex length hwidth weight lglength  lgweight
2    11 FEMALE   46.4   8.18   11.0 1.666518 1.0413927
6     3 FEMALE   42.8   7.32    8.6 1.631444 0.9344985
8     5 FEMALE   45.0   8.05   10.9 1.653213 1.0374265
9    12 FEMALE   44.0   7.55    8.9 1.643453 0.9493900
11    6 FEMALE   40.0   6.53    6.2 1.602060 0.7923917
```

```
14   17 FEMALE   35.1   6.04    4.5 1.545307 0.6532125
16   22 FEMALE   48.1   8.55   12.8 1.682145 1.1072100
19    7 FEMALE   48.0   8.67   13.5 1.681241 1.1303338
```

You can use more complicated conditions, such as

```
turtle[sex="FEMALE" & length>40,]
```

or

```
turtle[is.element(idno, c(17,19,104)),]
```

By now you would realize that you can wrap these references to the dataframe inside instructions to act upon the data they yield.

```
summary(turtle[is.element(sex, "FEMALE"),])
```

```
      idno           sex         length          hwidth    
 Min.   : 3.00   FEMALE:8   Min.   :35.10   Min.   :6.040  
 1st Qu.: 5.75   MALE  :0   1st Qu.:42.10   1st Qu.:7.122  
 Median : 9.00              Median :44.50   Median :7.800  
 Mean   :10.38              Mean   :43.67   Mean   :7.611  
 3rd Qu.:13.25              3rd Qu.:46.80   3rd Qu.:8.273  
 Max.   :22.00              Max.   :48.10   Max.   :8.670  
     weight         lglength        lgweight     
 Min.   : 4.50   Min.   :1.545   Min.   :0.6532  
 1st Qu.: 8.00   1st Qu.:1.624   1st Qu.:0.8990  
 Median : 9.90   Median :1.648   Median :0.9934  
 Mean   : 9.55   Mean   :1.638   Mean   :0.9557  
 3rd Qu.:11.45   3rd Qu.:1.670   3rd Qu.:1.0578  
 Max.   :13.50   Max.   :1.682   Max.   :1.1303  
```

As an aside, note that the summary command knows about the existence of `MALE` in the dataframe even though we have requested an analysis on females only. This is because R has created `sex` as a factor rather than a character vector, and the factor levels for `sex` have been passed to the summary function in addition to the values of `sex`.

Try out some of the above approaches for yourself by submitting the statements to the R Console

Writing R Programs II

Assignment Statements

Assignment – Creating New Objects

We can also apply arithmetic to the rows and columns now that we know they are vectors. For example, we can create two new variables that might be useful in later analyses. Two new variables (or objects), `lglength` and `lgweight`, are created by taking the logarithm to base 10 of `length` and `weight` respectively using the `log10()` function.

```
turtle$lglength <- log10(turtle$length)
turtle$lgweight <- log10(turtle$weight)
```

Type the above statements in the R Editor and submit it for execution.

If we examine the dataframe turtle, we will see that the two new variables are included.

```
turtle
```

Submit the above command to the R Console.

The output:

```
   idno    sex length hwidth weight lglength  lgweight
1    10   MALE   41.0   7.15   7.60 1.612784 0.8808136
2    11 FEMALE   46.4   8.18  11.00 1.666518 1.0413927
3     2   <NA>   24.3   4.42   1.65 1.385606 0.2174839
4    15   <NA>   28.7   4.89   2.18 1.457882 0.3384565
5    16   <NA>   32.0   5.37   3.00 1.505150 0.4771213
6     3 FEMALE   42.8   7.32   8.60 1.631444 0.9344985
7     4   MALE   40.0   6.60   6.50 1.602060 0.8129134
8     5 FEMALE   45.0   8.05  10.90 1.653213 1.0374265
9    12 FEMALE   44.0   7.55   8.90 1.643453 0.9493900
10   13   <NA>   28.0   4.85   1.97 1.447158 0.2944662
11    6 FEMALE   40.0   6.53   6.20 1.602060 0.7923917
12    8   <NA>   32.0   5.35   2.90 1.505150 0.4623980
13    9   MALE   35.0   5.74   3.90 1.544068 0.5910646
14   17 FEMALE   35.1   6.04   4.50 1.545307 0.6532125
15   19   MALE   42.3   6.77   7.80 1.626340 0.8920946
16   22 FEMALE   48.1   8.55  12.80 1.682145 1.1072100
17  105   MALE   44.0   7.10   9.00 1.643453 0.9542425
18   14   MALE   43.0   6.60   7.20 1.633468 0.8573325
19    7 FEMALE   48.0   8.67  13.50 1.681241 1.1303338
20    1   <NA>   29.2   5.10   2.38 1.465383 0.3765770
21  104   MALE   44.0   7.35   9.00 1.643453 0.9542425
```

Remember, to access the new log variables without using the `turtle$` prefix the modified dataframe turtle must be copied to reside on the search path once more.

```
attach(turtle)
```

Submit the above command to the R Console.

Math Functions and Transformations

Any arithmetic expression can be used in an assignment to create contents for an existing or new object. There are a myriad of mathematical functions that that can be included in assignment statements. Some of the more useful ones follow.

abs()	Take the absolute of values in a vector or other object
min()	Extract the minimum value from a vector or other object
max()	Extract the maximum value from a vector or other object
asin()	Take the arcsin of values in a vector or other object (answer in radians)
cos()	Take the cosine of values in a vector or other object
exp()	Raises e to the power of values in a vector or other object
log()	Takes the natural log of values in a vector or other object
log10 ()	Takes the log to base 10 of values in a vector or other object
sin ()	Takes the sine of values in a vector or other object
sqrt ()	Takes the square root of values in a vector or other object
tan ()	Takes the tangent of values in a vector or other object

A full list of mathematical functions can be obtained using the R help facility (using the HELP menu on the Menu Bar).

For example, transforming counts of macroinvertebrates using a standard square root transformation is effected by

```
count <- sqrt(count+0.5)
```

The standard arithmetic operators apply, and include and include the usual addition (+), subtraction (-), division (/), multiplication (*) and exponentiation (^).

If fish size changes with time exponentially in accordance with the Von Bertalanffy growth equation

$$L = 3.00 - 2.5e^{0.138t}$$

we can easily code this in R as

```
length <- 3.00 - 2.5*exp(0.138*time)
```

Assignment statement chains

More complicated equations are possible by splitting them over several lines. For example, the Sharpe-DeMichele growth model relating embryonic growth with temperature is given by

$$\frac{ds}{dt} = \frac{RHO_{25}\dfrac{T}{298.15}\exp\left[\dfrac{H_A}{r}\left(\dfrac{1}{298.15}-\dfrac{1}{T}\right)\right]}{1+\exp\left[\dfrac{H_L}{r}\left(\dfrac{1}{T_L}-\dfrac{1}{T}\right)\right]+\exp\left[\dfrac{H_H}{r}\left(\dfrac{1}{T_H}-\dfrac{1}{T}\right)\right]}$$

and can be coded in R as follows, using the temporary intermediary variables `c1`, `c2` and `c3` which are subsequently discarded with a `rm()` statement. Temperature T is in degrees Kelvin, and `r` in the equation above is the Gas Constant 1.987.

```
degk <- temp+273.15
c1 <- exp((1/298.15-1/degk)*ha/1.987)
c2 <- exp((1/tl-1/degk)*hl/1.987)
c3 <- exp((1/th-1/degk)*hh/1.987)
rate <- (rho25*degk*c1/298.15)/(1+c2+c3)
rm(c1, c2, c3, degk)
```

This code assumes that the parameters `temp, ha, hl, hh, th, tl` and `rho25` have been assigned values earlier in the program.

Exercise

Retrun your attention to our dataframe `turtle`.

Construct a body condition index as the difference between the actual weight of a turtle and its weight predicted from linear body measurements according to the formula

Predicted Weight = 0.0014*length^2.91

The new vector of values for body condition should be contained in the dataframe `turtle`. Confirm that it is. Calculate summary statistics for this new variable.

Selectively Deleting Rows

Use of indexed reference to the contents of a dataframe as part of assignment statements can be used to selectively delete rows from a dataframe.

```
turtle.new <- turtle[!is.na(sex),]
```

will remove all data for which `sex` is missing. I think you can see the possibilities.

Selectively Deleting Columns

The same approach can be used to select columns for retention.

```
turtle.new <- turtle[,c("sex","length")]
```

will remove all variables except `sex` and `length`.

```
turtle.new <- turtle[,!names(turtle)=="sex"]
```

will remove the variable `sex`.

Writing R Programs III
Functions

What are Functions?

You will realize by now that most of the commands used in R are functions of one sort or another.

Functions in R are not functions in the mathematical sense, but rather are the equivalent to subroutines or subprograms in other languages. A function is a discrete block of code that takes data, manipulates it and returns the results of those manipulations. For example, the function `sort()` can be used to re-order a data vector.

```
weight <- c(10.4, 5.6, 3.1, 6.4, 21.7)
sort(weight)
```

```
[1]  3.1  5.6  6.4 10.4 21.7
```

Information on a particular function can be obtained by typing a question mark followed by the function name, for example,

```
?sort
```

Functions are extremely useful elements of R programming. You can create your own, modify those already available in R, or collect your functions into a library and make them available to others.

Creating Your Own Functions

We can define our own functions very easily. To define a function called `echo`, we might use

```
echo <- function(x) {print(x)}
```

This is using the function statement to define a new function that takes a single argument x. The value of x is then passed to the statements that make up the body of the function, inside the curly brackets. In this case the body of the function is a single statement, `print(x)`. We then assign the function to the object `echo`, which can be subsequently called on as follows

```
   echo("Hello Cocky!")
[1] "Hello Cocky!"
```

Enter and execute the above function definition, then use it to echo some simple statements.

It is possible to put any number of statements inside the curly brackets, and so build quite sophisticated functions.

There are many functions built into the base library of R. A full listing of them can be obtained via the web-base help page (select Help from the Console Menu Bar), under the link entitled "Packages".

Exercise

Retrun your attention to our dataframe `turtle`.

Construct a function to calculate the coefficient of variation for a vector. Recall that the coefficient of variation is the standard deviation divided by the mean. Use the `sum()`, `length()` and `sqrt()` functions in R to do the job.

If you are rusty, you might need to Google the equation for the standard deviation.

Apply your new function to calculate the coefficient of variation for `length` and `weight`. Which of the two is the most variable?

Writing R Programs IV

Controlling Program Flow

The statements making up R programs are executed in the order in which they are given, that is, from top to bottom. However, you are able to control the flow of execution of the program with conditional statements. R has all the standard conditional statements and looping statements of a structured programming language – IF-THEN-ELSE constructs, DO-WHILE constructs, DO-UNTIL constructs and DO-FOR constructs.

IF-THEN-ELSE constructs

The general form of an `if` statement is

```
if (logical expression 1){
     program block 1
     }
else if(logical expression 2){
     program block 2
     }
else {
     program block 3
}
```

A logical expression is constructed from the conventional logical operators. These include equal to (==), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), not equal to (`!=`), the "and" operator (&), the negation operator (`!`) and the "or" operator (`|`). For example

```
if (sex == "FEMALE" | sex == "NA") {cat(idno)}
```

There can be as many `else if` statements embedded in this program element as desired. This structure enables you to execute different sets of program instructions depending on the value taken (T or F) by the series of logical expressions, with clearly defined terminal action if none of the conditions are met. Nice neat structured code.

The indenting is optional, but greatly increases readability.

The structure can be simplified to

```
if (logical expression 1){
     program block 1
     }
else {
     program block 3
}
```

or further simplified to

```
if (logical expression){
     program block 1
     }
```

depending on the number of conditions that are to be treated differently by the program code.

Looping Constructs

R has two primary mechanisms for looping, that is, for repeating segments of code. You can repeat the code **for** a specified range of an index variable or repeat it **while** some condition prevails.

The syntax for the repeat `for` loop is

```
for (i in 1:200) {
     program block
}
```

which will execute 200 times while progressively incrementing the variable `i` which would typically be involved in the calculations undertaken in the program block.

The while loop has the following syntax

```
while (logical expression) {
     program block
}
```

The combination of progressive execution of your program from top to bottom, conditional statements like `if` and `else if`, and looping with repeat `for` and repeat `while` constructs, provides you with full control over the flow of execution of your instructions and a great deal of versatility in your programs.

Data Subsetting

The above constructs for selection and looping are pretty fundamental to any third generation programming language. However R has a number of special features that reduce the need for such constructs. For example, you can take the dataframe turtle and split it into two for separate treatment, rather than using an if-else construct.

We can divide our turtle dataframe into two new dataframes on the basis of a logical condition as follows.

```
males <- subset(turtle, sex=="MALE")
females <- subset(turtle, sex=="FEMALE")
```

Enter and execute the above statements.

The contents can be examined in the usual way.

```
males
```

```
   idno  sex length hwidth weight
1    10 MALE   41.0   7.15    7.6
7     4 MALE   40.0   6.60    6.5
13    9 MALE   35.0   5.74    3.9
15   19 MALE   42.3   6.77    7.8
17  105 MALE   44.0   7.10    9.0
18   14 MALE   43.0   6.60    7.2
21  104 MALE   44.0   7.35    9.0
```

```
females
```

```
   idno    sex length hwidth weight
2    11 FEMALE   46.4   8.18   11.0
6     3 FEMALE   42.8   7.32    8.6
8     5 FEMALE   45.0   8.05   10.9
9    12 FEMALE   44.0   7.55    8.9
11    6 FEMALE   40.0   6.53    6.2
14   17 FEMALE   35.1   6.04    4.5
16   22 FEMALE   48.1   8.55   12.8
19    7 FEMALE   48.0   8.67   13.5
```

Submit the above command to the R Console.

Calculations can then be applied to males and females separately.

```
mean(males$weight); mean(females$weight)
```

```
[1] 7.285714
[1] 9.55
```

Type the above statements in the R Editor and submit for execution.

We can then combine the two dataframes and view the results

```
turtle <- rbind(males, females)
turtle
```

```
   idno    sex length hwidth weight condition
1    10   MALE   41.0   7.15    7.6  61.47630
7     4   MALE   40.0   6.60    6.5  57.78692
13    9   MALE   35.0   5.74    3.9  39.68791
15   19   MALE   42.3   6.77    7.8  67.84470
17  105   MALE   44.0   7.10    9.0  75.83505
18   14   MALE   43.0   6.60    7.2  72.14531
21  104   MALE   44.0   7.35    9.0  75.83505
```

```
2     11 FEMALE   46.4    8.18    11.0  63.15666
6      3 FEMALE   42.8    7.32     8.6  50.30995
8      5 FEMALE   45.0    8.05    10.9  57.05628
9     12 FEMALE   44.0    7.55     8.9  54.84029
11     6 FEMALE   40.0    6.53     6.2  42.37859
14    17 FEMALE   35.1    6.04     4.5  28.97335
16    22 FEMALE   48.1    8.55    12.8  69.36510
19     7 FEMALE   48.0    8.67    13.5  68.17920
```

Type the above statements in the R Editor and submit for execution.

and do some cleaning up of the workspace,

```
   rm(males); rm(females); ls()
```

Type the above statements in the R Editor and submit for execution.

A more satisfactory way of subseting the analysis is to use the `by()` function.

```
by(turtle, sex, summary)
sex: FEMALE
      idno            sex         length          hwidth     
 Min.   : 3.00   FEMALE:8   Min.   :35.10   Min.   :6.040  
 1st Qu.: 5.75   MALE  :0   1st Qu.:42.10   1st Qu.:7.122  
 Median : 9.00              Median :44.50   Median :7.800  
 Mean   :10.38              Mean   :43.67   Mean   :7.611  
 3rd Qu.:13.25              3rd Qu.:46.80   3rd Qu.:8.273  
 Max.   :22.00              Max.   :48.10   Max.   :8.670  
     weight          lglength         lgweight     
 Min.   : 4.50   Min.   :1.545   Min.   :0.6532  
 1st Qu.: 8.00   1st Qu.:1.624   1st Qu.:0.8990  
 Median : 9.90   Median :1.648   Median :0.9934  
 Mean   : 9.55   Mean   :1.638   Mean   :0.9557  
 3rd Qu.:11.45   3rd Qu.:1.670   3rd Qu.:1.0578  
 Max.   :13.50   Max.   :1.682   Max.   :1.1303  
--------------------------------------------------
sex: MALE
      idno             sex         length          hwidth     
 Min.   :  4.00   FEMALE:0   Min.   :35.00   Min.   :5.740  
 1st Qu.:  9.50   MALE  :7   1st Qu.:40.50   1st Qu.:6.600  
 Median : 14.00              Median :42.30   Median :6.770  
 Mean   : 37.86              Mean   :41.33   Mean   :6.759  
 3rd Qu.: 61.50              3rd Qu.:43.50   3rd Qu.:7.125  
 Max.   :105.00              Max.   :44.00   Max.   :7.350  
     weight          lglength         lgweight     
 Min.   :3.900   Min.   :1.544   Min.   :0.5911  
 1st Qu.:6.850   1st Qu.:1.607   1st Qu.:0.8351  
 Median :7.600   Median :1.626   Median :0.8808  
 Mean   :7.286   Mean   :1.615   Mean   :0.8490  
 3rd Qu.:8.400   3rd Qu.:1.638   3rd Qu.:0.9232  
 Max.   :9.000   Max.   :1.643   Max.   :0.9542  
```

Type the above statements in the R Editor and submit it for execution.

A related function for selectively analysing data in a dataframe is `tapply()`. This function takes on the form

```
   tapply(vector, factor, function)
```

where `vector` is the object to which the `function` is to be applied separately for each level of the `factor`. For example,

```
tapply(weight, sex, mean)
```

will apply the function `mean()` to the weights of each sex.

Type the above statements in the R Editor and submit it for execution.

The output is as follows.

```
  FEMALE     MALE
9.550000 7.285714
```

Exercise

Use the `if()` construct and your new function to calculate the coefficient of variation for the carapace lengths of juveniles. Remember, juveniles are those turtles for which `sex` is missing.

Repeat the analysis using the `subset()` approach.

Repeat the analysis again using the `by()` function.

Repeat the analysis again using the `tapply()` approach.

Where have we come?

The objective of the step-through exercises we have just done was to reinforce the key concepts introduced in lesson 1. In particular, you can now appreciate more that

- R is a programming language, with all the usual features for branching and looping.
- The raw data that is to be subject to analysis is typically constructed as a file comprising rows (observations or entities) and columns (variables or attributes). It is constructed using an application outside the R environment, such as Notepad.
- The raw data are read into a dataframe which is essentially a two-dimensional array. Elements of the dataframe can be accessed using array index notation in square brackets. Columns of the dataframe can be referenced and manipulated as named vectors.
- All manner of calculations can be applied to a dataframe by using the column vectors in assignment statements. The usual arithmetic operators apply, and there is a myriad of functions available for manipulating these vectors.

You have also learnt the value of the R Windows.

- The R Editor Window can be used for creating data sets and for creating your R programs.
- R programs need to be submitted for execution via the R Console.

- The results of the analysis appear in the R Console or Graphics Window.
- When things go wrong, error messages appear in the R Console.
- Help is available for R functions using the ? operator, or the online help facility.

You should now have a working understanding of workflow using R.

- When you start R, a workspace is established and a working directory is identified to hold your raw datafiles and your workspace image.
- The analysis proceeds under your instructions, which are submitted through the R Console.
- R maintains a search path to which you attach your dataframe to assist in working with vectors contained in that dataframe.
- Saving your workspace image at the end of a session enables you to resume work at a later date.

Where to now?

Now we have the basics under our belt, let's move on to the Lesson 3 to try a variety of analyses.

Lesson 6: Some Elementary Statistical Analyses

R as a Statistical Analysis System

The CRAN Repository

R is more than a programming language. R is also a mature statistical anaylsis system. Many people have taken the opportunity to develop analysis capability using R, and have made this capability available to the general scientific public in the form of packages.

All R functions are stored in packages. Only when a package is loaded are its contents available.

The standard `base` packages are considered part of the R source code. They contain the basic functions that allow R to work, and the datasets and standard statistical and graphical functions that are described in this Module.

There are literally hundreds of additional contributed packages for R, written by many different authors. Some of these packages implement specialized statistical methods, others give access to data or hardware, and others are designed to complement textbooks. Most are available for download from CRAN (http://CRAN.R-project.org/ and its mirrors).

To see which packages are installed at your site, issue the command

```
library()
```

or click on the [Packages] tab in R-studio.

To see which packages are currently loaded, use

```
search()
```

This will display the list of packages and objects currently in use, that is, those that will be searched for commands.

To download a package from the CRAN site, select the [Packages] tab on the R-studio Window. Then select [Install], type in the package name and, if available, select to install. This will add the new package to those existing in the library of packages within your installed copy of R, but will not load the package.

To load a package that is available from within your installed copy of R, use

```
library(boot)
```

This example will load the `boot` package containing bootstrapping functions provided by Davison & Hinkley (1997) *Bootstrap Methods and their Application*. Cambridge University Press.

You can confirm that it has been loaded with

```
search()
```

As we move through the more advanced Modules using R, new packages will be identified and loaded.

Getting Started

If you are not continuing directly from the previous lesson, you will need to start R again by double-clicking on the relevant icon on the desktop or in an Applications Group window.

Double-click on the R icon on your desktop and start a session with the `start.session()` command.

If you are not continuing directly from the previous lesson, you will need also to refamiliarise R with the dataset. If you have the relevant program saved, read it into the Editor. Otherwise, you will need to type it in again.

```
# Read in the data
   turtle <- read.table("mydata.dat")
# Add variable names
   names(turtle) <- c("idno","sex","length","hwidth","weight")
# Create new variables as the log to base 10 of length and
# hdwidth
   turtle$lglength <- log10(turtle$length)
   turtle$lgweight <- log10(turtle$weight)
# Copy the dataframe turtle to the search path
   attach(turtle)
```

Highlight the above segment of your program as it appears in the R Editor, and use ^r to submit it for execution.

Descriptive statistics

The summary function

Descriptive statistics are a useful place to start an analysis. They can be obtained by using the `summary()` command as follows:

```
   summary(turtle)
```

Type the above statements in the R Editor and submit it for execution.

The output is as follows.

```
      idno             sex         length          hwidth          weight     
 Min.   :  1.00   FEMALE:8   Min.   :24.30   Min.   :4.42   Min.   : 1.650  
 1st Qu.:  6.00   MALE  :7   1st Qu.:32.00   1st Qu.:5.37   1st Qu.: 3.000  
 Median : 11.00   NA's  :6   Median :41.00   Median :6.60   Median : 7.200  
 Mean   : 19.19              Mean   :38.71   Mean   :6.58   Mean   : 6.737  
 3rd Qu.: 16.00              3rd Qu.:44.00   3rd Qu.:7.35   3rd Qu.: 9.000  
 Max.   :105.00              Max.   :48.10   Max.   :8.67   Max.   :13.500  
    lglength         lgweight     
 Min.   :1.386   Min.   :0.2175  
 1st Qu.:1.505   1st Qu.:0.4771  
 Median :1.613   Median :0.8573  
 Mean   :1.580   Mean   :0.7502  
 3rd Qu.:1.643   3rd Qu.:0.9542  
 Max.   :1.682   Max.   :1.1303  
```

Other useful functions

Other functions are available to produce statistics such as the minimum, maximum, range, standard deviation and so on (Table 1-5).

Table 1-5. Useful summary statistics functions in R. In all cases x refers to a vector.

Function	Operation
`min(x)`	Minimum of *x*
`max(x)`	Maximum of *x*
`range(x)`	Range of *x*
`mean(x)`	Mean of *x*
`median(x)`	Median of *x*
`sd(x)`	Standard deviation of *x*
`var(x)`	Variance of *x*
`length(x)`	Number of elements in *x*
`quantile(x, p)`	The quantiles of *x* corresponding to probability *p*
`summary(x)`	Minimum, maximum, median and mean of *x*

Subsetting

Alternatively, you might want descriptive statistics calculated separately for each sex, in which case the `tapply()` function is most appropriate. The `tapply()` function has the following form:

```
tapply(vector, factor, function)
```

where `vector` is the object to which the `function` is to be applied separately for each level of the `factor`. For example,

```
tapply(weight, sex, summary)
```

will apply the function `summary()` to the weights of each sex.

Type the above statements in the R Editor and submit it for execution.

The output is as follows.

```
$FEMALE
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4.50    8.00    9.90    9.55   11.45   13.50

$MALE
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.900   6.850   7.600   7.286   8.400   9.000
```

Try some of the other functions listed in Table 1-5 on some of the other numeric variables in the dataframe.

Histograms, barcharts

Histograms

Size distributions are an important biological characteristic of populations of animals with indeterminate growth, such as turtles. To obtain a size distribution for the Kakadu population of pig-nosed turtles, the following step is appropriate:

```
hist(length)
```

We can tart this up a bit by specifying the intervals and the colour.

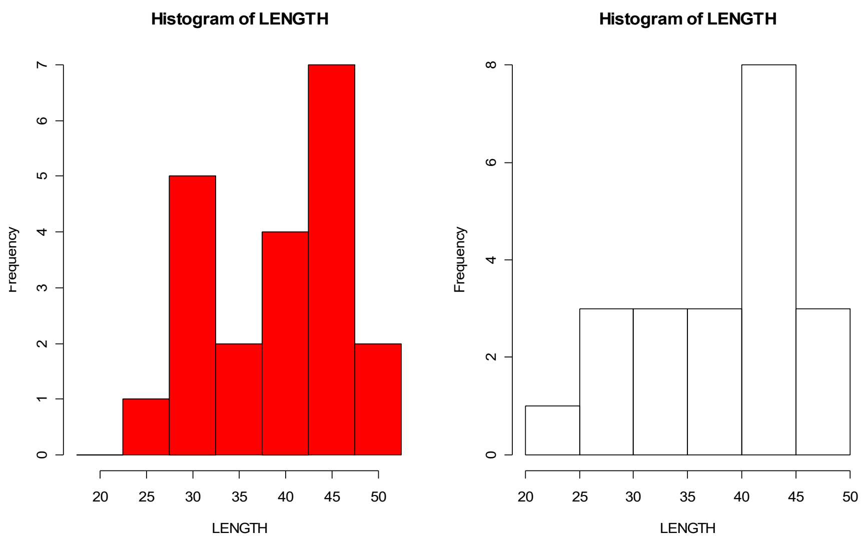
```
hist(length, breaks=seq(17.5,52.5,5), col="red")
```

This step will produce a histogram of the variable `length` based on the frequency of individuals falling in each of the intervals specified by the `breaks` option. In this case each interval will be 5.0 cm wide, and centered on 20 cm, 25 cm, 30 cm etc. `Col=2` specifies the colour of the bars.

Type the above statements in the R Editor and submit it for execution.

Note that your graph may differ from that shown owing to differences in screen attributes.

Figure 1-6 Size distribution for pig-nosed turtles from Kakadu National Park. Length is in cm. The histogram on the left was produced with function hist() with user-defined breaks. The histogram on the right is the same data using the default settings in function hist().



Bar Charts

Histograms are fine for continuous data, but for discrete data it is customary to construct barcharts (the columns of a barchart are separated by a space, Figure 1-7). Bar charts in R are achieved using the `plot()` function. In the turtle data set the variable `sex` is discrete, and the following program is appropriate:

```
plot(sex, type="h", col="red")
```

Type the above statements in the R Editor and submit it for execution.

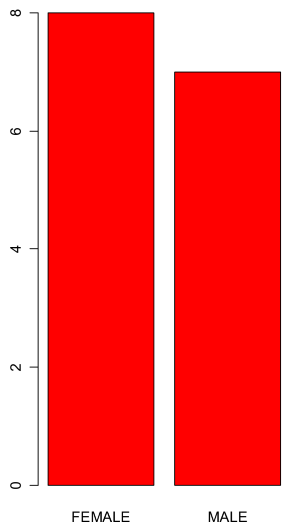


Figure 1.7. A barchart showing the frequency of males and females in a population of pig-nosed turtles from Kakadu National Park. The graph was produced using plot().

T-tests

To perform a student's T-test with R, the data must be in the form of a measurement variable occupying one data column and a breakdown variable occupying another. The breakdown variable must have only two values, not counting missing values.

In the example at hand, we might choose to compare the body weights of males (`sex` coded as MALE) with those of females (`sex` coded as FEMALE). Because juveniles were coded with the missing value code 'NA' the t.test() function will ignore them. Here is the appropriate program:

```
t.test(weight ~ sex)
```

Type the above statements in the R Editor and submit it for execution.

The output is as follows.

```
        Welch Two Sample t-test

data:  weight by sex
t = 1.761, df = 11.235, p-value = 0.1054
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.5584373  5.0870088
sample estimates:
mean in group FEMALE   mean in group MALE
            9.550000             7.285714
```

The `t.test()` function has lots of options, and you might like to look at those by typing

```
?t.test
```

One option is to perform a paired T-test, but the turtle data set does not provide the opportunity to perform a paired T-test.

Scatterplots

The plot statement

Moving on to the bi-variate procedures, scatterplots are an important prelude to both correlation and regression analyses. Before performing a linear regression or correlation analysis, it is important to be sure that the relationship between the two variables under consideration is roughly linear. Consider the relationship between `length` and `weight` using a `plot()` command:

The syntax of the `plot()` function is

```
plot(x,y)
```

so to plot `weight` as a function of `length`, we need

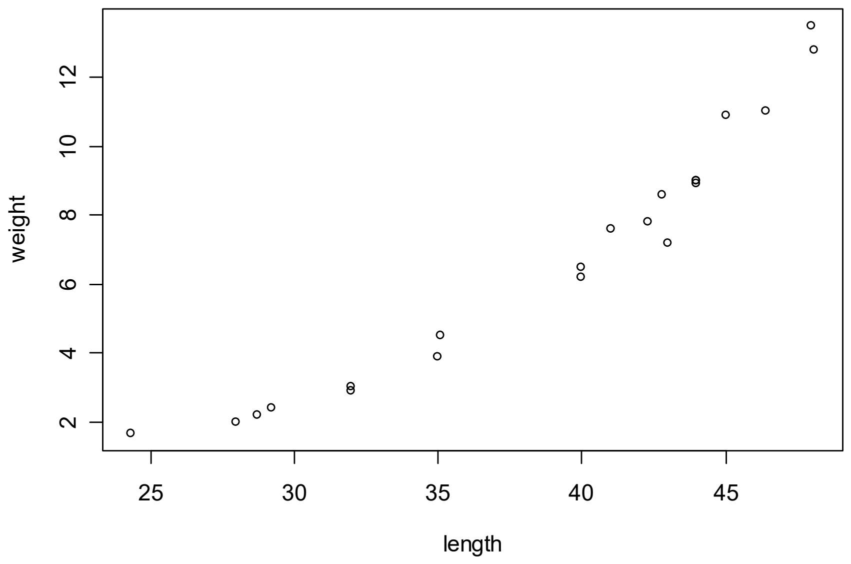
```
plot(length,weight)
```

Type the above statements in the R Editor and submit it for execution.

This statement will produce the plot shown in Figure 1-8, with `weight` on the vertical axis and `length` on the horizontal axis.

As expected, the relationship is not linear, weight being more of a function of body volume than of body shell length. It is for this reason that the transformations of `length` and `weight` were undertaken earlier.

Figure 1-8. Relationship between body weight and carapace length for the pig-nosed turtle from Kakadu National Park. Length is in cm and weight is in kg. The scatterplot was produced with plot().



The extent to which the transformations linearise the relationship between body weight and length can be judged from the following analysis:

```
plot(lglength,lgweight)
```

Type the above statements in the R Editor and submit it for execution.

The output is shown in Figure 1-9. With the possible exception of the left-most point, the relationship between logged shell length and logged body weight appears linear.

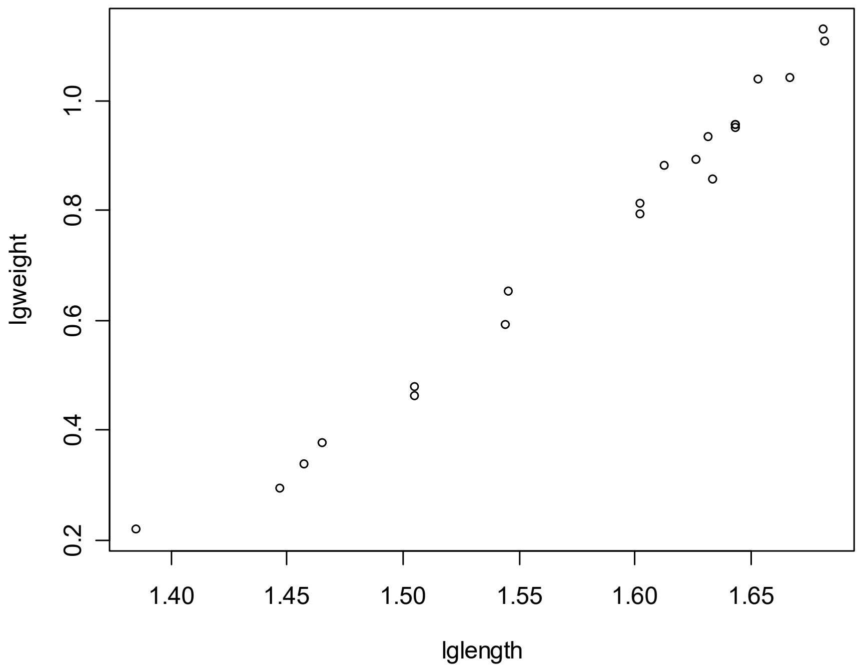


Figure 1-9. Log-linear relationship between body weight and carapace length for the pig-nosed turtle from Kakadu National Park. Length is in cm and weight is in kg. Both variables have been transformed by logs to base 10. The scatterplot was produced with plot().

Correlations

The next step might be to calculate correlations between our response variable weight and the linear measurement of length, after logging.

```
cor.test(lglength,lgweight)
```

which yields the following output

```
        Pearson's product-moment correlation

data:  lglength and lgweight
t = 33.2983, df = 19, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9788252 0.9966334
sample estimates:
      cor
0.9915406
```

The correlation coefficient of 0.99 is significant with a p value well less than 0.0001.

Type the above statements in the R Editor and submit it for execution.

Simple linear regression

As Aboriginal residents of Kakadu National Park regularly eat turtles, one can often obtain shells of the species that are the remains of a meal. In order to estimate the weight of a turtle from its shell length, a predictive regression of weight on shell length is required.

Because of curvilinearity in the relationship between the two untransformed variables, a linear regression of `lgweight` on `lglength` is appropriate:

```
lm(lgweight~lglength)
```

The `lm()` function is designed as a general tool for undertaking linear modelling. In this case, with a single response variable and a single predictor, `lm()` does a simple linear regression. The output is as follows.

```
Call:
lm(formula = lgweight ~ lglength)

Coefficients:
(Intercept)     lglength
     -4.392        3.255
```

More details on the analysis can be obtained by combining the `summary()` function with the `lm()` function

```
summary(lm(lgweight~lglength))
```

The output is as follows.

```
Call:
lm(formula = lgweight ~ lglength)

Residuals:
      Min        1Q    Median        3Q       Max
-0.067357 -0.023797 -0.002945  0.016399  0.099567

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.39213    0.15466   -28.4   <2e-16 ***
lglength     3.25492    0.09775    33.3   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03817 on 19 degrees of freedom
Multiple R-Squared: 0.9832,     Adjusted R-squared: 0.9823
F-statistic:  1109 on 1 and 19 DF,  p-value: < 2.2e-16
```

Note now that we have not only the parameter estimates, but tests of their significance, and the usual R^2 value.

Type the above statements in the R Editor and submit it for execution.

For simple linear regression, the last two lines of the output suffice for a quick interpretation of the analysis. The predictive relationship is

```
LGWEIGHT = 3.25492*LGLENGTH – 4.39
```

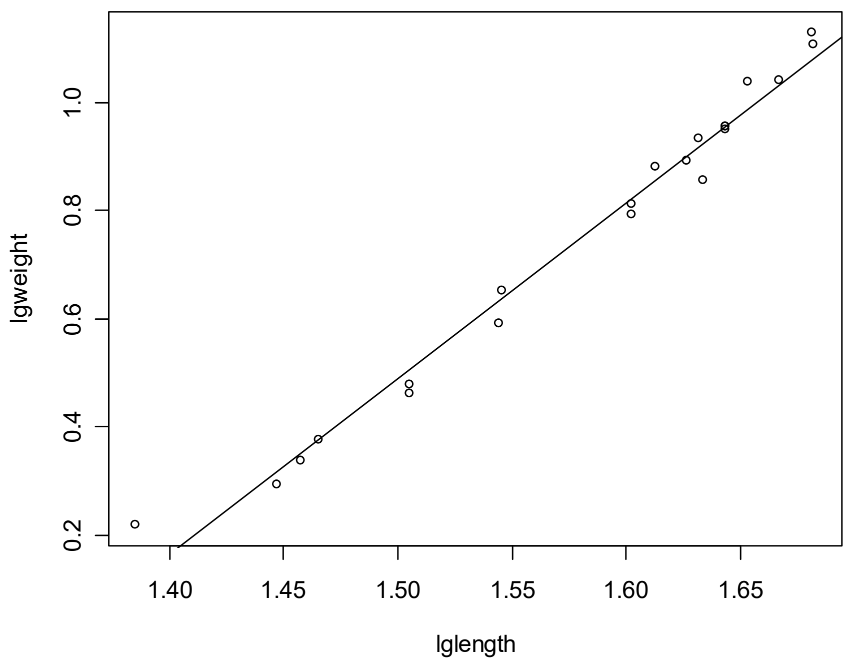
The regression coefficient (slope) is significant ($t = 33.3$, $p < 0.0001$).

A graph of the relationship with the regression line can be be produced using a combination of plot() and abline() (Figure 1-7).

```
plot(lglength,lgweight)
abline(lm(lgweight~lglength))
```

The `abline()` function is used to add a line of best fit generated by the model specified in the `lm()` function. This program yields the output shown in Figure 1-10.

Figure 1-10. Log-linear relationship between body weight and carapace length for the pig-nosed turtle from Kakadu National Park. Both variables have been transformed by logs to base 10. The scatterplot and least-squares line was produced with plot() with the abline() function.



Finishing up

The preceding analyses of the turtle data set have exposed you to R commands for some commonly used basic statistical techniques. Before you instruct the computer to exit from R, you may wish to produce a printout of your program.

Tidy up the program listing in the R Editor window by ensuring there are no elements remaining of the program that did not work.

Print the contents, and then save the program to disk.

Before you leave R, it’s worthwhile to try the online help facility. Detailed help is available on a wide range of R options. Try obtaining help on topics that strike your interest.

> Select Html Help under the HELP Menu of the R Console and peruse the help files.

The basic introduction to R is now complete, so exit from the R environment. You should clean up your workspace by perusing its contents with the `ls()` command and removing unwanted objects with the `rm()` command. Select to save your workspace image, in case you want to revisit this module.

> Clean up your workspace and exit from R by choosing File_Exit from the Menu Bar. Elect to save your workspace, when prompted.

Where have we come?

Having completed this module, you have the basic knowledge and skills to undertake simple statistical analyses in R. In particular, you will appreciate that:

- R is a programming language that uses objects as its primary constructs, and in particular, vectors, dataframes and functions
- R has a windows interface including an R Console for entering commands directly and an R Editor for creating and submitting programs. Graph windows are for displaying graphical output, and Help windows appear when help is requested.
- Data can be accessed from separate raw data file, usually created using a third-party editor or an Excel spreadsheet.
- Once the data are read into R, there are a very large number of functions for analysing those data, both in the base library and in a very many specialized libraries available on the web.

It needs to be said that this module has introduced only a very small part of the capability of R, and has been designed to be a minimal introduction, to get you up to speed as quickly as possible.

Your capacity for using R will grow with use, and you need to keep a good notebook to record new procedures and to share these with your colleagues.