THE UNIVERSITY OF
WESTERN AUSTRALIA

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of The University of Western Australia pursuant to Part VB of the *Copyright Act* 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Document reference:

Author	Maindonald, John Hilary & Braun, John
Chapter title	A brief introduction to R
Book title	Data analysis and graphics using R : an example-based approach
Place	New York
Publisher	Cambridge University Press
Date	2003
Pages	1-28

Downloaded from *Course Materials Online* at the University of Western Australia Library

1

A Brief Introduction to R

This first chapter is intended to introduce readers to the basics of R. It should provide an adequate basis for running the calculations that are described in later chapters.

In later chapters, the R commands that handle the calculations are, mostly, confined to footnotes. Sections are included at the ends of several of the chapters that give further information on the relevant features in R. Most of the R commands will run without change in S-PLUS.

1.1 A Short R Session

1.1.1 R *must be installed!*

An up-to-date version of R may be downloaded from http://cran.r-project.org/ or from the nearest mirror site. Installation instructions are provided at the web site for installing R in Windows, Unix, Linux, and various versions of the Macintosh operating system. Various contributed packages are now a part of the standard R distribution, but a number are not; any of these may be installed as required. Data sets that are mentioned in this book have been collected into a package that we have called *DAAG*. This is available from the web pages http://cbis.anu.edu.au/DAAG and http://www.stats.uwo.ca/DAAG.

1.1.2 Using the console (or command line) window

The command line prompt (>) is an invitation to start typing in commands or expressions. R evaluates and prints out the result of any expression that is typed in at the command line in the console window (multiple commands may appear on the one line, with the semicolon (`;`) as the separator). This allows the use of R as a calculator. For example, type in `2+2` and press the **Enter** key. Here is what appears on the screen:

```
> 2+2
[1] 4
>
```

The first element is labeled `[1]` even when, as here, there is just one element! The > indicates that R is ready for another command.

In a sense this chapter, and much of the rest of the book, is a discussion of what is possible by typing in statements at the command line. Practice in the evaluation of arithmetic

Table 1.1: *The contents of the file* ACTpop.txt.

Year	ACT
1917	3
1927	8
1937	11
1947	17
1957	38
1967	103
1977	214
1987	265
1997	310

expressions will help develop the needed conceptual and keyboard skills. Here are simple examples:

```
> 2*3*4*5              # * denotes 'multiply'
[1] 120
> sqrt(10)             # the square root of 10
[1] 3.162278
> pi                   # R knows about pi
[1] 3.141593
> 2*pi*6378            # Circumference of Earth at Equator (km);
                       # radius is 6378 km
[1] 40074.16
```

Anything that follows a # on the command line is taken as a comment and ignored by R.

There is also a continuation prompt that appears when, following a carriage return, the command is still not complete. By default, the continuation prompt is + (in this book we will omit both the prompt (>) and the continuation prompt (+), whenever command line statements are given separately from output).

1.1.3 Reading data from a file

Our first goal is to read a text file into R using the `read.table()` function. Table 1.1 displays population in thousands, for Australian Capital Territory (ACT) at various times since 1917. The command that follows assumes that the reader has entered the contents of Table 1.1 into a text file called ACTpop.txt.

When an R session is started, it has a working directory where, by default, it looks for any files that are requested. The following statement will read in the data from a file that is in the working directory (the working directory can be changed during the course of an R session; see Subsection 1.3.2):

```
ACTpop <- read.table("ACTpop.txt", header=TRUE)
```

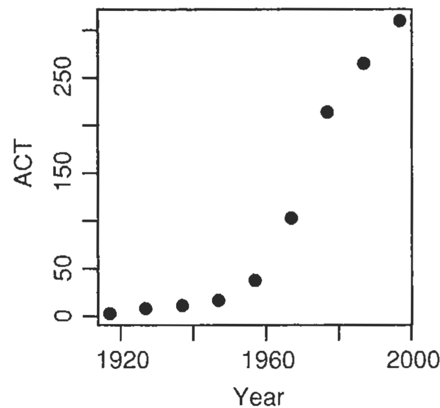

Figure 1.1: ACT population, in thousands, at various times between 1917 and 1997.

This reads in the data, and stores them in the data frame `ACTpop`. The `<-` is a left angle bracket (<) followed by a minus sign (-). It means "the values on the right are assigned to the name on the left". Note the use of `header=TRUE` to ensure that R uses the first line to get header information for the columns.

Type in `ACTpop` at the command line prompt, and the data will be displayed almost as they appear in Table 1.1 (the only difference is the introduction of row labels in the R output). The object `ACTpop` is an example of a data frame. Data frames are the usual way for organizing data sets in R. More information about data frames can be found in Section 1.5.

Case is significant for names of R objects or commands. Thus, `ACTPOP` is different from `ACTpop`. (For file names on Microsoft Windows systems, the Windows conventions apply, and case does not distinguish file names. On Unix systems letters that have a different case are treated as different.)

We now plot the ACT population between 1917 and 1997 by using

```
plot(ACT ~ Year, data=ACTpop, pch=16)
```

The option `pch=16` sets the plotting character to solid black dots. Figure 1.1 shows the graph.

We can make various modifications to this basic plot. We can specify more informative axis labels, change the sizes of the text and of the plotting symbol, add a title, and so on. More information is given in Section 1.8.

1.1.4 Entry of data at the command line

Table 1.2 gives, for each amount by which an elastic band is stretched over the end of a ruler, the distance that the band traveled when released. We can use `data.frame()` to input these (or other) data directly at the command line. We will assign the name `elasticband` to the data frame:

```
elasticband <- data.frame(stretch=c(46,54,48,50,44,42,52),
                          distance=c(148,182,173,166,109,141,166))
```

Table 1.2: *Distance (cm) versus stretch (mm), for elastic band data.*

Stretch	Distance
46	148
54	182
48	173
50	166
44	109
42	141
52	166

The constructs `c(46,54,48,50,44,42,52)` and `c(148,182,173,166,109, 141,166)` join ("concatenate") the separate numbers together into a single vector object. See later, Subsection 1.4.1. These vector objects then become columns in the data frame.

1.1.5 Online help

Before getting deeply into the use of R, it is well to take time to master the help facilities. Such an investment of time will pay dividends. R's help files are comprehensive, and are frequently upgraded. Type in `help(help)` to get information on the help features of the system that is in use. To get help on, e.g., `plot()`, type in

```
help(plot)
```

Different R implementations offer different choices of modes of access into the help pages (thus Microsoft Windows systems offer a choice between a form of help that displays the relevant help file, html help, and compiled html help. The choice between these different modes of access is made at startup. See `help(Rprofile)` for details).

Two functions that can be highly useful in searching for functions that perform a desired task are `apropos()` and `help.search()`. We can best explain their use by giving specific examples. Thus try

```
apropos("sort")          # Try, also, apropos ("sor")
    # This lists all functions whose names include the
    # character string "sort".
help.search("sort")  # Note that the argument is "sort"
    # This lists all functions that have the word 'sort' as
    # an alias or in their title.
```

Note also `example()`. This initiates the running of examples, if available, of the use of the function specified by the function argument. For example:

```
example(image)
  # for a 2 by 2 layout of the last 4 plots, precede with
  # par(mfrow=c(2,2))
  # to prompt for each new graph, precede with par(ask=T)
```

Much can be learned from experimenting with R functions. It may be helpful to create a simple artificial data set with which to experiment. Another possibility is to work with a subset of the data set to which the function will, finally, be applied. For extensive experimentation it is best to create a new workspace where one can work with copies of any user data sets and functions.

Among the abilities that are documented in the help pages, there will be some that bring pleasant and unexpected surprises. There may be insightful and helpful examples. There are often references to related functions. In most cases there are technical references that give the relevant theory. While the help pages are not intended to be an encyclopedia on statistical methodology, they contain much helpful commentary on the methods whose implementation they document. It can help enormously, before launching into use of an R function, to check the relevant help page!

1.1.6 Quitting R

One exits or quits R by using the `q()` function:

```
q()
```

There will be a message asking whether to save the workspace image. Clicking **Yes** (the safe option) will save all the objects that remain in the working directory – any that were there at the start of the session and any that were added (and not removed) during the session.

Depending on the implementation, alternatives may be to click on the **File** menu and then on **Exit**, or to click on the × in the top right hand corner of the R window. (Under Linux, clicking on × exits from the program, but without asking whether to save the workshop image.)

Note: In order to quit from the R session we had to type `q()`. This is because `q` is a function. Typing `q` on its own, without the parentheses, displays the text of the function on the screen. Try it!

1.2 The Uses of R

R has extensive capabilities for statistical analysis, that will be used throughout this book. These are embedded in an interactive computing environment that is suited to many different uses. Here we draw attention to abilities, beyond simple one-line calculations, that are not primarily statistical.

R *will give numerical or graphical data summaries*

An important class of R object is the data frame. R uses data frames to store rectangular arrays in which the columns may be vectors of numbers or factors or text strings. Data frames are central to the way that all the more recent R routines process data. For now, think of data frames as rather like a matrix, where the rows are observations and the columns are variables.

As a first example, consider the data frame `cars` that is in the *base* package. This has two columns (variables), with the names `speed` and `dist`. Typing in `summary(cars)` gives summary information on these variables.

```
> data(cars) # Gives access to the data frame cars
> summary(cars)
     speed           dist
 Min.   : 4.0   Min.   :  2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   : 42.98
 3rd Qu.:19.0   3rd Qu.: 56.00
 Max.   :25.0   Max.   :120.00
```

Thus, we can immediately see that the range of speeds (first column) is from 4 mph to 25 mph, and that the range of distances (second column) is from 2 feet to 120 feet.

R has extensive abilities for graphical presentation

The main R graphics function is `plot()`, which we have already encountered. In addition, there are functions for adding points and lines to existing graphs, for placing text at specified positions, for specifying tick marks and tick labels, for labeling axes, and so on. Some details are given in Section 1.8.

R is an interactive programming language

Suppose we want to calculate the Fahrenheit temperatures that correspond to Celsius temperatures 25, 26, ..., 30. Here is a good way to do this in R:

```
> celsius <- 25:30
> fahrenheit <- 9/5*celsius+32
> conversion <- data.frame(Celsius=celsius, Fahrenheit=fahrenheit)
> print(conversion)
  Celsius Fahrenheit
1      25       77.0
2      26       78.8
3      27       80.6
4      28       82.4
5      29       84.2
6      30       86.0
```

1.3 The R Language

R is a functional language that uses many of the same symbols and conventions as the widely used general purpose language C and its successors C++ and Java. There is a language core that uses standard forms of algebraic notation, allowing the calculations that were described in Subsection 1.1.2. Functions – supplied as part of R or one of its packages, or written by the user – allow the limitless extension of this language core.

1.3.1 R *objects*

All R entities, including functions and data structures, exist as objects that can be operated on as data. Type in `ls()` (or `objects()`) to see the names of all objects in the workspace. One can restrict the names to those with a particular pattern, e.g., starting with the letter "*p*". (Type in `help(ls)` and `help(grep)` for more details. The pattern-matching conventions are those used for `grep()`, which is modeled on the Unix grep command. For example, `ls(pattern="p")` lists all object names that include the letter "p". To get all object names that start with the letter "p", specify `ls(pattern="^p")`.) As noted earlier, typing the name of an object causes the printing of its contents.

It is often possible and desirable to operate on objects – vectors, arrays, lists and so on – as a whole. This largely avoids the need for explicit loops, leading to clearer code. Section 1.2 gave an example.

1.3.2 Retaining objects between sessions

Upon quitting an R session, we have recommended saving the workspace image. It is easiest to allow R to save the image, in the current working directory, with the file name `.RData` that R uses as the default. The image will then be loaded automatically when a new session is started in that directory. The objects that were in the workspace at the end of the earlier session will again be available.

Many R users find it convenient to work with multiple workspaces, typically with a different working directory for each different workspace. As a preliminary to loading a new workspace, it will usually be desirable to save the current workspace, and then to clear all objects from it. These operations may be performed from the menu, or alternatively there are the commands `save.image()` for saving the current workspace, `rm(list=ls())` for clearing the workspace, `setwd()` for changing the working directory, and `load()` for loading a new workspace. For details, see the help pages for these functions. See also Subsection 12.9.1.

One should avoid cluttering the workspace with objects that will not again be needed. Before saving the current workspace, type `ls()` to get a complete list of objects. Then remove unwanted objects using

```
rm(<obj1>, <obj2>,...)
```

where the names of objects that are to be removed should appear in place of `<obj1>`, `<obj2>`, ... For example, to remove the objects `ACTpop` and `elasticband` from the workspace image before quitting, type

```
rm(ACTpop, elasticband)
q()
```

In general, we have left it to the reader to determine which objects should be removed once calculations are complete.

1.4 Vectors in R

Vectors may have mode "logical", "numeric", "character" or "list". Examples of vectors are

```
> c(2, 3, 5, 2, 7, 1)
[1] 2 3 5 2 7 1

> 3:10    # The numbers 3, 4,..., 10
[1] 3 4 5 6 7 8 9 10

> c(T, F, F, F, T, T, F)
[1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE

> c("Canberra", "Sydney", "Canberra", "Sydney")
[1] "Canberra" "Sydney"   "Canberra" "Sydney"
```

The first two vectors above are numeric, the third is logical, and the fourth is a character vector. Note the use of the global variables `F(=FALSE)` and `T(=TRUE)` as a convenient shorthand when logical values are entered.

1.4.1 Concatenation – joining vector objects

The `c` in `c(2, 3, 5, 2, 7, 1)` is an abbreviation for "concatenate". The meaning is: "Collect these numbers together to form a vector." We can concatenate two vectors. In the following, we form numeric vectors `x` and `y`, that we then concatenate to form a vector `z`:

```
> x <- c(2, 3, 5, 2, 7, 1)
> x
[1] 2 3 5 2 7 1
> y <- c(10, 15, 12)
> y
[1] 10 15 12
> z <- c(x, y)
> z
[1] 2 3 5 2 7 1 10 15 12
```

1.4.2 Subsets of vectors

Note three common ways to extract subsets of vectors.

1. Specify the indices of the elements that are to be extracted, e.g.,

```
> x <- c(3, 11, 8, 15, 12)  # Assign to x the values 3,
                            # 11, 8, 15, 12
> x[c(2,4)]                 # Elements in positions 2
[1] 11 15                   # and 4 only
```

2. Use negative subscripts to omit the elements in nominated subscript positions (take care not to mix positive and negative subscripts):

```
> x[-c(2,3)]   # Remove the elements in positions 2 and 3
[1] 3 15 12
```

3. Specify a vector of logical values. The elements that are extracted are those for which the logical value is `TRUE`. Thus, suppose we want to extract values of `x` that are greater than 10.

```
> x > 10
[1] FALSE TRUE FALSE TRUE TRUE
> x[x > 10]
[1] 11 15 12
```

1.4.3 Patterned data

Use, for example, `5:15` to generate all integers in a range, here between 5 and 15 inclusive.

```
> 5:15
[1]  5  6  7  8  9 10 11 12 13 14 15
```

Conversely, `15:5` will generate the sequence in the reverse order.

The function `seq()` is more general. For example:

```
> seq(from=5, to=22, by=3)  # The first value is 5.  The final
                            # value is <= 22
[1] 5 8 11 14 17 20
```

The function call can be abbreviated to

```
seq(5, 22, 3)
```

To repeat the sequence (2, 3, 5) four times over, enter

```
> rep(c(2,3,5), 4)
[1] 2 3 5 2 3 5 2 3 5 2 3 5
```

Patterned character vectors are also possible

```
> c(rep("female", 3), rep("male", 2))
[1] "female" "female" "female" "male" "male"
```

1.4.4 Missing values

The missing value symbol is `NA`. As an example, we may set

```
y <- c(1, NA, 3, 0, NA)
```

Note that any arithmetic operation or relation that involves `NA` generates an `NA`. Specifically, be warned that `y[y==NA] <- 0` leaves `y` unchanged. The reason is that all elements of

`y==NA` evaluate to `NA`. This does not identify an element of `y`, and there is no assignment. To replace all NAs by 0, use the function `is.na()`, thus

```
> y[is.na(y)] <- 0
> y
[1] 1 0 3 0 0
```

The functions `mean()`, `median()`, `range()`, and a number of other functions, take the argument `na.rm=T`; i.e. remove NAs, then proceed with the calculation. By default, these and related functions will fail when there are NAs. By default, the `table()` function ignores NAs.

1.4.5 Factors

A factor is stored internally as a numeric vector with values 1, 2, 3, . . . , k. The value k is the number of levels. The levels are character strings.

Consider a survey that has data on 691 females and 692 males. If the first 691 are females and the next 692 males, we can create a vector of strings that holds the values thus:

```
gender <- c(rep("female",691), rep("male",692))
```

We can change this vector to a factor, by entering

```
gender <- factor(gender)
```

Internally, the factor `gender` is stored as 691 1s, followed by 692 2s. It has stored with it a table that holds the information

1	female
2	male

In most contexts that seem to demand a character string, the 1 is translated into `female` and the 2 into `male`. The values `female` and `male` are the levels of the factor. By default, the levels are chosen to be in sorted order for the data type from which the factor was formed, so that `female` precedes `male`. Hence:

```
> levels(gender)
[1] "female" "male"
```

Note that if `gender` had been an ordinary character vector, the outcome of the above `levels` command would have been `NULL`. The order of the factor levels determines the order of appearance of the levels in graphs and tables that use this information. To cause `male` to come before `female`, use

```
gender <- factor(gender, levels=c("male", "female"))
```

This syntax is available both when the factor is first created, and later to change the order in an existing factor. Take care that the level names are correctly spelled. For example, specifying `"Male"` in place of `"male"` in the `levels` argument will cause all values that were `"male"` to be coded as missing.

One advantage of factors is that the memory required for storage is less than for the corresponding character vector when there are multiple values for each factor level, and the levels are long character strings.

1.5 Data Frames

Data frames are fundamental to the use of the R modeling and graphics functions. A data frame is a generalization of a matrix, in which different columns may have different modes. All elements of any column must, however, have the same mode, i.e. all numeric, or all factor, or all character, or all logical.

Included with our data sets is `Cars93.summary`, created from the `Cars93` data set in the Venables and Ripley *MASS* package, and included in our *DAAG* package. In order to access it, we need first to install it and then to load it into the workspace, thus:

```
> library(DAAG)           # load the DAAG package
> data(Cars93.summary)    # copy Cars93.summary into the workspace
> Cars93.summary
        Min.passengers Max.passengers No.of.cars abbrev
Compact              4              6         16      C
Large                6              6         11      L
Midsize              4              6         22      M
Small                4              5         21     Sm
Sporty               2              4         14     Sp
Van                  7              8          9      V
```

Notice that, before we could access `Cars93.summary`, we had first to use `data(Cars93.summary)` to copy it into the workspace. This differs from the S-PLUS behavior, where such data frames in packages that have been loaded are automatically available for access.

The data frame has row labels (accessed using `row.names(Cars93.summary)`) `Compact`, `Large`,.... The column names (accessed using `names(Cars93.summary)`) are `Min.passengers` (i.e. the minimum number of passengers for cars in this category), `Max.passengers`, `No.of.cars`, and `abbrev`. The first three columns are numeric, and the fourth is a factor. Use the function `class()` to check this, e.g. enter `class(Cars93.summary$abbrev)`.

There are several ways to access the columns of a data frame. Any of the following will pick out the fourth column of the data frame `Cars93.summary` and store it in the vector `type` (also allowed is `Cars93.summary[4]`. This gives a data frame with the single column `abbrev`).

```
type <- Cars93.summary$abbrev
type <- Cars93.summary[,4]
type <- Cars93.summary[,"abbrev"]
type <- Cars93.summary[[4]] # Take the object that is stored
                            # in the fourth list element.
```

In each case, one can view the contents of the object `type` by entering `type` at the command line, thus:

```
> type
[1] C   L   M   Sm Sp V
Levels:   C L M Sm Sp V
```

It is often convenient to use the `attach()` function:

```
> attach(Cars93.summary)
  # R can now access the columns of Cars93.summary directly
> abbrev
[1] C   L   M   Sm Sp V
Levels:   C L M Sm Sp V
> detach("Cars93.summary")
  # Not strictly necessary, but tidiness is a good habit!
  # In R, detach(Cars93.summary) is an acceptable alternative
```

Detaching data frames that are no longer in use reduces the risk of a clash of variable names, e.g., two different attached data frames that have a column with the name `abbrev`, or an `abbrev` both in the workspace and in an attached data frame.

In Windows versions, use of `edit()` allows access to a spreadsheet-like display of a data frame or of a vector. Users can then directly manipulate individual entries or perform data entry operations as with a spreadsheet. For example,

```
Cars93.summary <- edit(Cars93.summary)
```

To close the spreadsheet, click on the **File** menu and then on **Close**.

1.5.1 Variable names

The `names()` function can be used to determine variable names in a data frame. As an example, consider the New York air quality data frame that is included with the base R package. To determine the variables in this data frame, type

```
data(airquality)
names(airquality)
[1] "Ozone"    "Solar.R" "Wind"     "Temp"     "Month"    "Day"
```

The `names()` function serves a second purpose. To change the name of the `abbrev` variable (the fourth column) in the `Cars93.summary` data frame to `code`, type

```
names(Cars93.summary)[4] <- "code"
```

If we want to change all of the names, we could do something like

```
names(Cars93.summary) <- c("minpass", "maxpass", "number", "code")
```

1.5.2 Applying a function to the columns of a data frame

The `sapply()` function is a useful tool for calculating statistics for each column of a data frame. The first argument to `sapply()` is a data frame. The second argument is the name of a function that is to be applied to each column. Consider the `women` data frame.

```
> data(women)
> women     # Display the data
   height weight
1       58    115
2       59    117
3       60    120
................
15      72    164
```

In order to compute averages of each column, type

```
> sapply(women, mean)  # Apply mean() to each of the columns
height weight
  65.0  136.7
```

1.5.3* Data frames and matrices

The numerical values in the data frame `women` might alternatively be stored in a matrix with the same dimensions, i.e., 15 rows × 2 columns. More generally, any data frame where all columns hold numeric data can alternatively be stored as a matrix. This can speed up some mathematical and other manipulations when the number of elements is large, e.g., of the order of several hundreds of thousands. For further details, see Section 12.7. Note that:

- The `names()` function cannot be used with matrices.
- Above, we used `sapply()` to extract summary information about the columns of the data frame `women`. If `women` had been a matrix with the same numerical values in the same layout, the result would have been quite different, and uninteresting – the effect is to apply the function `mean` to each individual element of the matrix.

1.5.4 Identification of rows that include missing values

Many of the modeling functions will fail unless action is taken to handle missing values. Two functions that are useful for checking on missing values are `complete.cases()` and `na.omit()`. The following code shows how we can identify rows that hold missing values.

```
> data(possum)
> possum[!complete.cases(possum), ]
     case site Pop sex age hdlngth skullw totlngth taill
BB36   41    2 Vic   f   5    88.4   57.0       83  36.5
BB41   44    2 Vic   m  NA    85.1   51.5       76  35.5
BB45   46    2 Vic   m  NA    91.4   54.4       84  35.0
     footlgth earconch  eye  chest belly
BB36       NA     40.3 15.9   27.0  30.5
BB41     70.3     52.6 14.4   23.0  27.0
BB45     72.8     51.2 14.4   24.5  35.0
```

The function `na.omit()` omits any rows that contain missing values. For example

```
newpossum <- na.omit(possum)    # Has three fewer rows than possum
```

1.6 R Packages

The recommended R distribution includes a number of packages in its library. Note in particular *base*, *eda*, *ts* (time series), and *MASS*. We will make frequent use both of the *MASS* package and of our own *DAAG* package. *DAAG*, and other packages that are not included with the default distribution, can be readily downloaded and installed.

Installed packages, unless loaded automatically, must then be *load*ed prior to use. The *base* package is automatically loaded at the beginning of the session. To load any other installed package, use the `library()` function. For example,

```
library(MASS)                  # Loads the MASS package
```

1.6.1 Data sets that accompany R *packages*

Type in `data()` to get a list of data sets (mostly data frames) in all packages that are in the current search path. To get information on the data sets that are included in the *base* package, specify

```
data(package="base")       # NB. Specify 'package', not 'library'.
```

Replace `"base"` by the name of any other installed package, as required (type in `library()` to get the names of the installed packages).

In order to bring any of these data frames into the working directory, the user must specifically request it. (Ensure that the relevant package is loaded.) For example, to access the data set `airquality` from the *base* package, type in

```
data(airquality)    # Load airquality into the working directory
```

Such objects should be removed (`rm(airquality)`) when they are not for the time being required. They can be loaded again as occasion demands.

1.7* Looping

A simple example of a `for` loop is[1]

```
> for (i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

[1] Other looping constructs are

```
repeat <expression>            # Place break somewhere inside
while (x > 0) <expression>     # Or (x < 0), or etc.
```

Here `<expression>` is an R statement, or a sequence of statements that are enclosed within braces.

Here is a possible way to estimate population growth rates for each of the Australian states and territories:

```
data(austpop)                    # population figures for all
                                 # Australian states
growth.rates <- numeric(8)       # numeric(8) creates a numeric
                                 # vector with 8 elements, all set
                                 # equal to 0
for (j in seq(2,9)) {
    growth.rates[j-1] <- (austpop[9, j]-austpop[1, j])/
    austpop[1, j]}
growth.rates <- data.frame(growth.rates)
row.names(growth.rates) <- names(austpop[c(-1,-10)])
  # We have used row.names() to name the rows of the data frame
```

The result is

```
     growth.rates
NSW          2.30
Vic          2.27
Qld          3.98
SA           2.36
WA           4.88
Tas          1.46
NT          36.40
ACT        102.33
```

Avoiding loops – `sapply()`

The above computation can also be done using the `sapply()` function mentioned in Subsection 1.5.2:

```
> sapply(austpop[,-c(1,10)], function(x){(x[9]-x[1])/x[1]})
   NSW   Vic    Qld    SA    WA   Tas     NT    ACT
  2.30  2.27   3.98  2.36  4.88  1.46  36.40 102.33
```

Note that in contrast to the example in Subsection 1.5.2, we now have an *inline* function, i.e. one that is defined on the fly and does not have or need a name. The effect is to assign the columns of the data frame `austpop[,-c(1,10)]`, in turn, to the function argument `x`. With `x` replaced by each column in turn, the function returns `(x[9]-x[1])/x[1]`.

In R there is often a better alternative, perhaps using one of the built-in functions, to writing an explicit loop. Loops can incur severe computational overhead.

1.8 R Graphics

The functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`, `identify()`, etc. form a suite that plot graphs and add features to the graph. To see some of the possibilities that R offers, enter

```
demo(graphics)
```

Press the **Enter** key to move to each new graph.

1.8.1 The function `plot( )` *and allied functions*

The basic command is

```
plot(y ~ x)
```

or

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

where `x` and `y` must be the same length.

Readers may find the following plots interesting (note that `sin()` expects angles to be in radians. Multiply angles that are given in degrees by $\pi/180$ to get radians):

```
plot((0:20)*pi/10, sin((0:20)*pi/10))
plot((1:50)*0.92, sin((1:50)*0.92))
```

Readers might show the second of these graphs to their friends, asking them to identify the pattern! By holding with the left mouse button on the lower border until a double sided arrow appears and dragging upwards, the vertical dimension of the graph sheet can be shortened. If sufficiently shortened, the pattern becomes obvious. The eye has difficulty in detecting patterns of change where the angle of slope is close to the horizontal or close to the vertical.

Then try this:

```
par(mfrow=c(3,1))    # Gives a 3 by 1 layout of plots
plot((1:50)*0.92,    sin((1:50)*0.92))
par(mfrow=c(1,1))
```

Here are two further examples.

```
attach(elasticband)         # R now knows where to find stretch
                            # and distance
plot(stretch, distance)     # Alternative: plot(distance ~ stretch)
detach(elasticband)

attach(austpop)
plot(year, ACT, type="l")   # Join the points ("l" = "line")
detach(austpop)
```

Fine control – parameter settings

When it is necessary to change the default parameter settings, use the `par()` function. We have already used `par(mfrow=c(m, n))` to get an m by n layout of graphs on a page. Here is another example:

```
par(cex=1.25, mex=1.25)
```

increases the text and plot symbol size 25% above the default. Adding `mex=1.25` makes room in the margin to accommodate the increased text size.

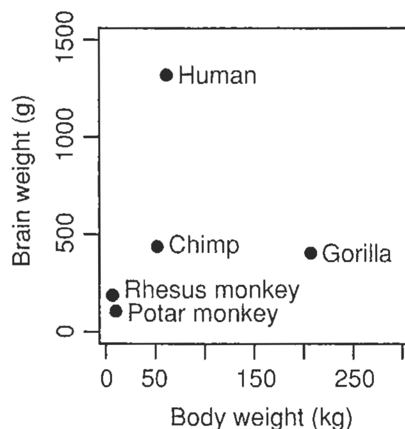

Figure 1.2: Brain weight versus body weight, for the `primates` data frame.

It is good practice to store the existing settings, so that they can be restored later. For this, specify, e.g.,

```
oldpar <- par(cex=1.25, mex=1.25)  # Use par(oldpar) to restore
                                   # earlier settings
```

The size of the axis annotation can be controlled, independently of the setting of `cex`, by specifying a value for `cex.axis`. Similarly, `cex.labels` may be used to control the size of the axis labels.

Type in `help(par)` to get a list of all the parameter settings that are available with `par()`.

Adding points, lines, text and axis annotation

Use the `points()` function to add points to a plot. Use the `lines()` function to add lines to a plot. The `text()` function places text anywhere on the plot. (Actually these functions are identical, differing only in the default setting for the parameter `type`. The default setting for `points()` is `type = "p"`, and that for `lines()` is `type = "l"`. Explicitly setting `type = "p"` causes either function to plot points, `type = "l"` gives lines.) The function `mtext(text, side, line, ...)` adds text in the margin of the current plot. The sides are numbered 1 (x-axis), 2 (y-axis), 3 (top) and 4 (right vertical axis). The `axis()` function gives fine control over axis ticks and labels.

Use of the `text()` function to label points

In Figure 1.2 we have put labels on the points.

We begin with code that will give a crude version of Figure 1.2. The function `row.names()` extracts the row names, which we then use as labels. We then use the function `text()` to add text labels to the points.

```
data(primates)               # The DAAG package must be loaded
attach(primates)             # Needed if primates is not already
                             # attached.
```

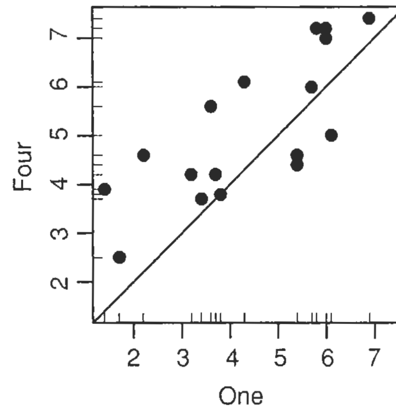

Figure 1.3: Each of 17 panelists compared two milk samples for sweetness. One of the samples had one unit of additive, while the other had four units of additive.

```
plot(Bodywt, Brainwt, xlim=c(0, 300))
  # Specify xlim so that there is room for the labels
text(x=Bodywt, y=Brainwt, labels=row.names(primates), adj=0)
  # adj=0 implies left adjusted text
detach(primates)
```

The resulting graph would be adequate for identifying points, but it is not a presentation quality graph. We now note the changes that are needed to get Figure 1.2. In Figure 1.2 we use the `xlab` (x-axis) and `ylab` (y-axis) parameters to specify meaningful axis titles. We move the labeling to one side of the points by including appropriate horizontal and vertical offsets. We multiply `chw <- par()$cxy[1]` by 0.1 to get an horizontal offset that is one tenth of a character width, and similarly for `chh <- par()$cxy[2]` in a vertical direction. We use `pch=16` to make the plot character a heavy black dot. This helps make the points stand out against the labeling.

Here is the R code for Figure 1.2:

```
attach(primates)
plot(x=Bodywt, y=Brainwt, pch=16, xlab="Body weight (kg)",
     ylab="Brain weight (g)", xlim=c(0,300), ylim=c(0,1500))
chw <- par()$cxy[1]
chh <- par()$cxy[2]
text(x=Bodywt+chw, y=Brainwt+c(-.1, 0, 0,.1, 0)*chh,
     labels=row.names(primates), adj=0)
detach(primates)
```

Where `xlim` and/or `ylim` is not set explicitly, the range of data values determines the limits. In any case, the axis is by default extended by 4% relative to those limits.

Rug plots

The function `rug()` adds vertical bars, showing the distribution of data values, along one or both of the x- and y-axes of an existing plot. Figure 1.3 has rugs on both the x- and y-axes. Data were from a tasting session where each of 17 panelists assessed the sweetness of each of two milk samples, one with four units of additive, and the other with one unit of

additive. The code that produced Figure 1.3 is

```
data(milk)                    # From the DAAG package
xyrange <- range(milk)
plot(four ~ one, data = milk, xlim = xyrange, ylim =
     xyrange, pch = 16)
rug(milk$one)
rug(milk$four, side = 2)
abline(0, 1)
```

Histograms and density plots

We mention these here for completeness. They will be discussed in Subsection 2.1.1.

The use of color

Try the following:

```
theta <- (1:50)*0.92
plot(theta, sin(theta), col=1:50, pch=16, cex=4)
points(theta, cos(theta), col=51:100, pch=15, cex=4)
palette()                     # Names of the 8 colors in the default
                              # palette
```

Points are in the eight distinct colors of the default palette, one of which is "white". These are recycled as necessary.

The default palette is a small selection from the built-in colors. The function `colors()` returns the 657 names of the built-in colors, some of them aliases for the same color. The following repeats the plots above, but now using the first 100 of the 657 built-in colors.

```
theta <- (1:50)*0.92
plot(theta, sin(theta), col=colors()[1:50], pch=16, cex=4)
points(theta, cos(theta), col=colors()[51:100], pch=15, cex=4)
```

1.8.2 Identification and location on the figure region

Two functions are available for this purpose. Draw the graph first, then call one or other of these functions:

- `identify()` labels points;
- `locator()` prints out the co-ordinates of points.

In either case, the user positions the cursor at the location for which co-ordinates are required, and clicks the left mouse button. Depending on the platform, the identification or labeling of points may be terminated by pointing outside of the graphics area and clicking, or by clicking with a button other than the first. The process will anyway terminate after some default number `n` of points, which the user can set. (For `identify()` the default setting is the number of data points, while for `locator()` the default is 500.)

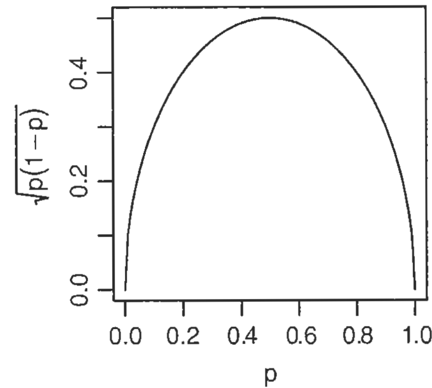

Figure 1.4: The y-axis label is a mathematical expression.

As an example, identify two of the plotted points on the `primates` scatterplot:

```
attach(primates)
plot(Bodywt, Brainwt)
identify(Bodywt, Brainwt, n=2) # Now click near 2 plotted points
detach(primates)
```

1.8.3 Plotting mathematical symbols

Both `text()` and `mtext()` allow replacement of the text string by a mathematical expression. In `plot()`, either or both of `xlab` and `ylab` can be an algebraic expression. Figure 1.4 was produced with

```
p <- (0:100)/100
plot(p, sqrt(p*(1-p)), ylab=expression(sqrt(p(1-p))), type="l")
```

Type `help(plotmath)` to get details of available forms of mathematical expression. The final plot from `demo(graphics)` shows some of the possibilities for plotting mathematical symbols. There are brief further details in Section 12.10

1.8.4 Row by column layouts of plots

There are several ways to do this. Here, we will demonstrate two of them.

Multiple plots, each with its own margins

As noted in earlier sections, the parameter `mfrow` can be used to configure the graphics sheet so that subsequent plots appear row by row, one after the other in a rectangular layout, on the one page. For a column by column layout, use `mfcol`. The following example gives a plot that displays four different transformations of the `Animals` data.

```
par(mfrow=c(2,2))          # 2 by 2 layout on the page
library(MASS)              # Animals is in the MASS package
data(Animals)              # Needed if Animals is not already loaded
```

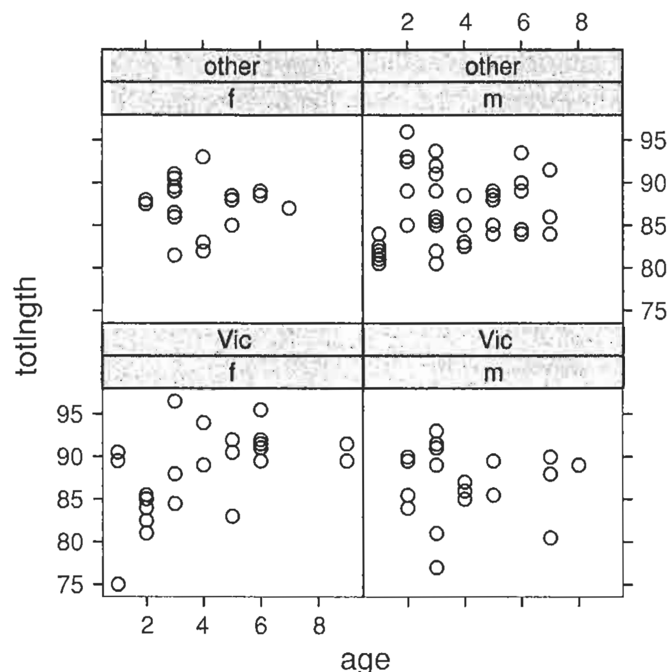

Figure 1.5: Total length of possums versus age, for each combination of population (the Australian state of Victoria or other) and sex (female or male). Further details of these data are in Subsection 2.1.1.

```
attach(Animals)
plot(body, brain)
plot(sqrt(body), sqrt(brain))
plot((body)^0.1, (brain)^0.1)
plot(log(body), log(brain))
detach("Animals")
par(mfrow=c(1,1))          # Restore to 1 figure per page
```

Multiple panels – the `lattice` *function* `xyplot()`

The function `xyplot()` in the *lattice* package gives a rows by columns (x by y) layout of panels in which the axis labeling appears in the outer margins. Figure 1.5 is an example. Enter

```
> library(lattice)
> data(possum)                      # DAAG must be loaded
> table(possum$Pop, possum$sex)     # Graph reflects layout of this
                                    # table
          f  m
  Vic    24 22
  other  19 39
> xyplot(totlngth ~ age | sex*Pop, data=possum)
```

Note that, as we saw in Subsection 1.5.4, there are missing values for `age` in rows 44 and 46 that `xyplot()` has silently omitted. The factors that determine the layout of the panels, i.e., `sex` and `Pop` in Figure 1.5, are known as *conditioning* variables.

There will be further discussion of the *lattice* package in Subsection 2.1.5. It has functions that offer a similar layout for many different types of plot. To see further examples of the use of `xyplot()`, and of some of the other *lattice* functions, type in

```
example(xyplot)
```

Further points to note about the *lattice* package are:

- The *lattice* package implements trellis style graphics, as used in Cleveland (1993). This is why functions that control stylistic features (color, plot characters, line type, etc.) have *trellis* as part of their name.
- Lattice graphics functions cannot be mixed with the graphics functions discussed earlier in this subsection. It is not possible to use `points()`, `lines()`, `text()`, etc., to add features to a plot that has been created using a *lattice* graphics function. Instead, it is necessary to use functions that are special to *lattice* – `grid.points()`, `grid.lines()`, `grid.text()`, etc.
- For inclusion, inside user functions, of statements that will print *lattice* graphs, see the note near the end of Subsection 2.1.5. An explicit print statement is typically required, e.g.

  ```
  print(xyplot(totlngth ~ age | sex*Pop, data=possum))
  ```

1.8.5 Graphs – additional notes

Graphics devices

On most systems, `x11()` will open a new graphics window. See `help(x11)`. On Macintosh systems that do not have an X11 driver, use `macintosh()`. See `help(Devices)` for a list of devices that can be used for writing to a file or to hard copy. Use `dev.off()` to close the currently active graphics device.

The shape of the graph sheet

It is often desirable to control the shape of the graph page. For example, we might want the individual plots to be rectangular rather than square. The function `x11()` sets up a graphics page on the screen display. It takes arguments `width` (in inches), `height` (in inches) and `pointsize` (in $\frac{1}{72}$ of an inch). The setting of `pointsize` (default = 12) determines character heights.[2]

Plot methods for objects other than vectors

We have seen how to plot a numeric vector `y` against a numeric vector `x`. The plot function is a generic function that also has special methods for "plotting" various

[2] It is the relative sizes of these parameters that matter for screen display or for incorporation into Word and similar programs. Once pasted (from the clipboard) or imported into Word, graphs can be enlarged or shrunk by pointing at one corner, holding down the left mouse button, and pulling.

different classes of object. For example, one can give a data frame as the argument to `plot`. Try

```
data(trees)             # Load data frame trees (base package)
plot(trees)             # Has the same effect as pairs(trees)
```

The `pairs()` function will be important when we come to discuss multiple regression. See Subsection 6.1.4, and later examples in that chapter.

Good and bad graphs

There is a difference!

Draw graphs so that they are unlikely to mislead, make sure that they focus the eye on features that are important, and avoid distracting features. In scatterplots, the intention is typically to draw attention to the points. If there are not too many of them, drawing them as heavy black dots or other symbols will focus attention on the points, rather than on a fitted line or curve or on the axes. If they are numerous, dots are likely to overlap. It then makes sense to use open symbols. Where there are many points that overlap, the ink will be denser. If there are many points, it can be helpful to plot points in a shade of gray.[3]

Where the horizontal scale is continuous, patterns of change that are important to identify should have an angle of slope in the approximate range 20° to 70°. (This was the point of the sine curve example in Subsection 1.8.1.)

There are a huge choice and range of colors. Colors, or gray scales, can often be used to useful effect to distinguish groupings in the data. Bear in mind that the eye has difficulty in focusing simultaneously on widely separated colors that appear close together on the same graph.

1.9 Additional Points on the Use of R in This Book

Functions

Functions are integral to the use of the R language. Perhaps the most important topic that we have left out of this chapter is a description of how users can write their own functions. User-written functions are used in exactly the same way as built-in functions. Subsection 12.2.2 describes how users may write their own functions. Examples will appear from time to time through the book.

An incidental advantage from putting code into functions is that the workspace is not then cluttered with objects that are local to the function.

Setting the number of decimal places in output

Often, calculations will, by default, give more decimal places of output than are useful. In the output that we give, we often reduce the number of decimal places below what R gives by default. The `options()` function can be used to make a global change to the number

[3]
```
## Example of plotting with different shades of gray
plot(1:4, 1:4, pch=16, col=c(''gray20",''gray40",''gray60",''gray40"), cex=2)
```

of significant digits that are printed. For example:

```
> sqrt(10)
[1] 3.162278
options(digits=2)           # Change until further notice, or until
                            # end of session.
> sqrt(10)
[1] 3.2
```

Notice that `options(digits=2)` expresses a wish, which R will not always obey! Rounding will sometimes introduce small inconsistencies. For example, in Section 4.5, with results rounded to two decimal places,

$$\sqrt{\frac{372}{12}} = 5.57$$

$$\sqrt{2} \times \sqrt{\frac{372}{12}} = 7.88.$$

Note however that $\sqrt{2} \times 5.57 = 7.87$

Other option settings

Type in `help(options)` to get further details. We will meet another important option setting in Chapter 5. (The output that we present uses the setting `options(show.signif.stars=FALSE)`, where the default is `TRUE`. This affects output in Chapter 5 and later chapters.)

Cosmetic issues

In our R code, we write, e.g., `a <- b` rather than `a<-b`, and `y ~ x` rather than `y~x`. This is intended to help readability, perhaps a small step on the way to literate programming. Such presentation details can make a large difference when others use the code.

Where output is obtained with the simple use of `print()` or `summary()`, we have in general included this as the first statement in the output.

** Common sources of difficulty*

Here we draw attention, with references to relevant later sections, to common sources of difficulty. We list these items here so that readers have a point of reference when it is needed.

- In the use of `read.table()` for the entry of data that have a rectangular layout, it is important to tune the parameter settings to the input data set. Check Subsection 12.3.1 for common sources of difficulty.
- Character vectors that are included as columns in data frames become, by default, factors. There are implications for the use of `read.table()`. See Subsection 12.3.1 and Section 12.4.
- Factors can often be treated as vectors of text strings, with values given by the factor levels. There are several, potentially annoying, exceptions. See Section 12.4.
- The handling of missing values is a common source of difficulty. See Section 12.5.

- The syntax `elasticband[,2]` extracts the second column from the data frame `elasticband`, yielding a numeric vector. Observe however that `elasticband[2, ]` yields a data frame, rather than the numeric vector that the user may require. Specify `unlist(elasticband[2, ])` to obtain the vector of numeric values in the second row of the data frame. See Subsection 12.6.1. For another instance (use of `sapply()`) where the difference between a numeric data frame and a numeric matrix is important, see Subsection 12.6.6.
- It is inadvisable to assign new values to a data frame, thus creating a new local data frame with the same name, while it is attached. Use of the name of the data frame accesses the new local copy, while the column names that are in the search path are for the original data frame. There is obvious potential for confusion and erroneous calculations.
- Data objects that individually or in combination occupy a large part of the available computer memory can slow down all memory-intensive computations. See further Subsection 12.9.1 for comment on associated workspace management issues. See also the opening comments in Section 12.7. Note that most of the data objects that are used for our examples are small and thus will not, except where memory is very small, make much individual contribution to demands on memory.

Variable names in data sets

We will refer to a number of different data sets, many of them data frames in our *DAAG* package. When we first introduce the data set, we will give both a general description of the columns of values that we will use, and the names used in the data frame. In later discussion, we will use the name that appears in the data frame whenever the reference is to the particular values that appear in the column.

1.10 Further Reading

An important reference is the R Core Development Web Team's (2001a) *Introduction to* R. This document, which is regularly updated, is included with the R distributions. It is available from CRAN sites as an independent document. (For a list of sites, go to http://cran.r-project.org.) Books that include an introduction to R include Dalgaard (2002) and Fox (2002).

See also documents, including Maindonald (2001), that are listed under **Contributed Documentation** on the CRAN sites. For a careful detailed account of the R and S languages, see Venables and Ripley (2000). There is a large amount of detailed technical information in Venables and Ripley (2000 and 2002).

Books and papers that set out principles of good graphics include Cleveland (1985 and 1993), Tufte (1997), Wainer (1997), Wilkinson et al. (1999). See also the brief comments in Maindonald (1992).

References for further reading

Cleveland, W.S. 1985. *The Elements of Graphing Data.* Wadsworth.

Cleveland, W.S. 1993. *Visualizing Data.* Hobart Press.

Dalgaard, P. 2002. *Introductory Statistics with* R. Springer-Verlag.

Fox, J. 2002. *An* R *and* S-PLUS *Companion to Applied Regression.* Sage Books.

Maindonald J.H. 1992. Statistical design, analysis and presentation issues. *New Zealand Journal of Agricultural Research* 35: 121–141.

Maindonald, J.H. 2001. *Using* R *for Data Analysis and Graphics.* Available as a pdf file at http://wwwmaths.anu.edu.au/~johnm/r/usingR.pdf

R Core Development Team 2001a. *An Introduction to* R. This document is available from CRAN sites. For a list, go to http://cran.r-project.org

Tufte, E.R. 1997. *Visual Explanations.* Graphics Press.

Venables, W.N. and Ripley, B.D. 2000. S *Programming.* Springer-Verlag.

Venables, W.N. and Ripley, B.D. 2002. *Modern Applied Statistics with* S-PLUS, 4th edn., Springer-Verlag, New York. See also "R" Complements to Modern Applied Statistics with S-PLUS, available from http://www.stats.ox.ac.uk/pub/MASS4/.

Wainer, H. 1997. *Visual Revelations.* Springer-Verlag.

Wilkinson, L. and Task Force on Statistical Inference 1999. Statistical methods in psychology journals: guidelines and explanation. *American Psychologist* 54: 594–604.

1.11 Exercises

1. Using the data frame `elasticband` from Subsection 1.1.4, plot `distance` against `stretch`.

2. The following table gives the size of the floor area (ha) and the price ($A000), for 15 houses sold in the Canberra (Australia) suburb of Aranda in 1999.

```
   area sale.price
1   694 192.0
2   905 215.0
3   802 215.0
4  1366 274.0
5   716 112.7
6   963 185.0
7   821 212.0
8   714 220.0
9  1018 276.0
10  887 260.0
11  790 221.5
12  696 255.0
13  771 260.0
14 1006 293.0
15 1191 375.0
```

   Type these data into a data frame with column names `area` and `sale.price`.

   (a) Plot `sale.price` versus `area`.
   (b) Use the `hist()` command to plot a histogram of the sale prices.
   (c) Repeat (a) and (b) after taking logarithms of sale prices.

3. The `orings` data frame gives data on the damage that had occurred in US space shuttle launches prior to the disastrous Challenger launch of January 28, 1986. Only the observations in rows 1, 2, 4, 11, 13, and 18 were included in the pre-launch charts used in deciding whether to proceed with the launch.

   Create a new data frame by extracting these rows from `orings`, and plot `total` incidents against `temperature` for this new data frame. Obtain a similar plot for the full data set.

4. Create a data frame called `Manitoba.lakes` that contains the lake's `elevation` (in meters above sea level) and `area` (in square kilometers) as listed below. Assign the names of the lakes using the `row.names()` function. Then obtain a plot of lake area against elevation, identifying each point by the name of the lake. Because of the outlying observation, it is again best to use a logarithmic scale.

```
               elevation  area
Winnipeg             217 24387
Winnipegosis         254  5374
Manitoba             248  4624
SouthernIndian       254  2247
Cedar                253  1353
Island               227  1223
Gods                 178  1151
Cross                207   755
Playgreen            217   657
```

   One approach is the following:

```
chw <- par()$cxy[1]/3
chh <- par()$cxy[2]/3
plot(log(area) ~ elevation, pch=16, xlim=c(170,270),
     ylim=c(6,11))
text(x=elevation-chw, y=log(area)+chh,
     labels=row.names(Manitoba.lakes), adj=1)
text(x=elevation+chw, y=log(area)+chh,
     labels=Manitoba.lakes[,2], adj=0)
title(''Manitoba's Largest Lakes")
```

5. The following code extracts the lake areas from the `Manitoba.lakes` data frame and attaches the lake names to the entries of the resulting vector.

```
area.lakes <- Manitoba.lakes[[2]]
names(area.lakes) <- row.names(Manitoba.lakes)
```

   Look up the help for the R function `dotchart()`. Use this function to display the data in `area.lakes`.

6. Using the `sum()` function, obtain a lower bound for the area of Manitoba covered by water.

7. The second argument of the `rep()` function can be modified to give different patterns. For example, to get four 2s, then three 3s, then two 5s, enter

```
rep(c(2,3,5), c(4,3,2))
```

(a) What is the output from the following command?

```
rep(c(2,3,5), 4:2)
```

(b) Obtain a vector of four 4s, four 3s and four 2s.

(c) The argument `length.out` can be used to create a vector whose length is `length.out`. Use this argument to create a vector of length 50 that consists of the repeated pattern `3 1 1 5 7`

8. The ^ symbol denotes exponentiation. Consider the following.

```
1000*((1+0.075)^5 - 1)   # Interest on $1000, compounded
                         # annually at 7.5% p.a. for five years
```

(a) Type in the above expression.

(b) Modify the expression to determine the amount of interest paid if the rate is 3.5% p.a.

(c) What happens if the exponent 5 is replaced by `seq(1, 10)`?

9. Run the following code:

```
gender <- factor(c(rep("female", 91), rep("male", 92)))
table(gender)
gender <- factor(gender, levels=c("male", "female"))
table(gender)

gender <- factor(gender, levels=c("Male", "female"))
                              # Note the mistake
                              # The level was "male", not "Male"
table(gender)
rm(gender)                    # Remove gender
```

Explain the output from the final `table(gender)`.

10. The following code uses the `for()` looping function to plot graphs that compare the relative population growth (here, by the use of a logarithmic scale) for the Australian states and territories.

```
oldpar <- par(mfrow=c(2,4))
for (i in 2:9){
plot(austpop[,1], log(austpop[, i]), xlab="Year",
     ylab=names(austpop)[i], pch=16, ylim=c(0,10))}
par(oldpar)
```

Can this be done without looping? [Hint: The answer is "yes", although this is not an exercise for novices.]