```
                  # is(TRUE, "logical") returns TRUE
is.na()           # Returns TRUE if the argument is an NA

## Information on an R object
str()             # Information on an R object
args()            # Information on arguments to a function
mode()            # Gives the storage mode of an R object
                  # (logical, numeric, character, . . ., list)

## Create a vector
numeric()         # numeric(5) creates a numeric vector, length 5,
                  # all elements 0.
                  # numeric(0) (length 0) is sometimes useful.
character()       # Create character vector; c.f. also logical()
```

The function `mean()`, and a number of other functions, takes the argument `na.rm=TRUE`; i.e., remove `NA`s, then proceed with the calculation. For example

```
mean(c(1, NA, 3, 0, NA), na.rm=T)
```

```
[1] 1.333
```

Note that the function `as()` has, at present, no method for coercing a matrix to a data frame. For this, use `as.data.frame()`.

Functions in different packages with the same name

For example, as well as *lattice* function `dotplot()` the graphics package has a defunct function `dotplot()`. To be sure of getting the *lattice* function *dotplot()*, refer to it as `lattice::dotplot`.

4.3.2 *Functions for data summary and/or manipulation*

4.3.3 *Functions for creating and working with tables*

4.3.4 *Tables of Counts*

Use either `table()` or `xtabs()` to make a table of counts. Use `xtabs()` for cross-tabulation, i.e., to determine totals of numeric values for each table category.

For data manipulation, note:

- the apply family of functions (Subsection 4.3.7).
- data manipulation functions in the *reshape2* and *plyr* packages (Chapter 6).

The `table()` function

For use of `table()`, specify one vector of values (often a factor) for each table margin that is required. For example:

```
library(DAAG)         # possum is from DAAG
with(possum, table(Pop, sex))
```

```
       sex
Pop      f  m
  Vic   24 22
  other 19 39
```

NAs in tables

By default, `table()` ignores NAs. To show information on NAs, specify `exclude=NULL`, thus:

```
library(DAAG)
table(nswdemo$re74==0, exclude=NULL)
```

```
FALSE  TRUE  <NA>
  119   326   277
```

The `xtabs()` function

This more flexible alternative to `table()` uses a table formula to specify the margins of the table:

```
xtabs(~ Pop+sex, data=possum)
```

```
       sex
Pop      f  m
  Vic   24 22
  other 19 39
```

A column of frequencies can be specified on the left hand side of the table formula. In order to demonstrate this, the three-way table `UCBAdmissions` (*datasets* package) will be converted into its data frame equivalent. Margins in the table become columns in the data frame:

Manipulations with data frames are in general conceptually simpler than manipulations with tables. For tables that are not unreasonably large, it is in general a good strategy to first convert the table to a data frame and make that the starting point for further calculations.

```
UCBdf <- as.data.frame.table(UCBAdmissions)
head(UCBdf, n=3)
```

```
     Admit Gender Dept Freq
1 Admitted   Male    A  512
2 Rejected   Male    A  313
3 Admitted Female    A   89
```

The following then forms a table of total admissions and rejections in each department:

```
xtabs(Freq ~  Admit+Dept, data=UCBdf)
```

```
          Dept
Admit        A   B   C   D   E   F
  Admitted 601 370 322 269 147  46
  Rejected 332 215 596 523 437 668
```

Information on data objects

The function `str()` gives basic information on the data object that is given as argument.

```
library(DAAG)
str(possumsites)
```

```
'data.frame':   7 obs. of  3 variables:
 $ Longitude: num  146 149 151 153 153 ...
 $ Latitude : num  -37.5 -37.6 -32.1 -28.6 -28.6 ...
 $ altitude : num  800 300 300 400 200 400 600
```

4.3.5 *Utility functions*

```
dir()              # List files in the working or other specified directory
sessionInfo()      # Print version numbers for R and for attached packages
system.file()      # By default, show path to 'package="base"'
R.home()           # Path to R home directory
.Library           # Path to the default library
.libPaths()        # Get/set paths to library directories
```

Section A has further details.

4.3.6 *User-defined functions*

The function `mean()` calculates means, The function `sd()` calculates standard deviations. Here is a function that calculates mean and standard deviation at the same time:

Note also that functions can be defined at the point of use. Such functions do not need a name, and are called anonymous functions. Section 4.3.4 has an example.

```
mean.and.sd <- function(x){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, sd = sdev)    # return value
}
```

The parameter `x` is the argument that the user must supply. The body of the function is enclosed between curly braces. The value that the function returns is given on its final line. Here the return value is a vector that has two named elements.

The following calculates the mean and standard deviation of heterozygosity estimates for seven different *Drosophila* species.[5]

[5] Data are from Lewontin, R. 1974. *The Genetic Basis of Evolutionary Change*.

```
hetero <- c(.43,.25,.53,.47,.81,.42,.61)
mean.and.sd(hetero)
```

```
  mean     sd
0.5029 0.1750
```

It is useful to give the function argument a default value, so that it can be run without user-supplied parameters, in order to see what it does. A possible choice is a set of random normal numbers, perhaps generated using the `rnorm()` function. Here is a revised function definition. Because the function body has been reduced to a single line, the curly braces are not needed.

Note that a different set of random numbers will be returned, giving a different mean and SD, each time that the function is run with its default argument.

```
mean.and.sd <- function(x = rnorm(20))
                  c(mean=mean(x), sd=sd(x))
mean.and.sd()
```

```
   mean      sd
-0.1563  0.8558
```

```
mean.and.sd()
```

```
  mean      sd
0.1434 0.8270
```

4.3.7 *The* `apply` *family of functions*

`apply()`, `sapply()` and friends

`apply()`	Use `apply()` to apply a function across rows or columns of a matrix (or data frame)
`sapply()` & friends	`sapply()` and `lapply()` apply functions in parallel across columns of a data frame, or across elements of a list, or across elements of a vector.

For the `apply` family of functions, specify as the `FUN` argument any function that will not generate an error. Obviously, `log("Hobart")` is not allowed!

Note also the function `tapply()`, which will not be discussed here.

`apply()`: The function `apply()` is intended for use with matrices or, more generally, with arrays. It has three mandatory arguments, a matrix or data frame, the dimension (1 for rows; 2 for columns) or dimensions, and a function that will be applied across that dimension of the matrix or data frame.

Here is an example:

If used with a data frames, the data frame is first coerced to `matrix`.

Code that will input `molclock1`:

```
library(DAAG)
datafile("molclock1")
molclock <-
  read.table("molclock1.txt")
```

```
apply(molclock, 2, range)
```

The following tabulates admissions, in the three-way table `UCBAdmissions`, according to `sex`:

```
apply(UCBAdmissions, c(1,2), sum)
```

```
          Gender
Admit       Male Female
  Admitted 1198    557
  Rejected 1493   1278
```

`sapply()` and `lapply()`: Use `sapply()` and `lapply()` to apply a function (e.g., `mean()`, `range()`, `median()`) in parallel to all columns of a data frame. They take as arguments the name of the data frame, and the function that is to be applied.

Warning: Use `apply()` with `COLUMN=2`, to apply a function to all columns of a matrix. If `sapply()` or `lapply()` is given a matrix as argument, the function is applied to each element (the matrix is treated as a vector).

The function `sapply()` returns the same information as `lapply()`. But whereas `lapply()` returns a list, `sapply()` tries if possible to simplify the result to give a vector or matrix or array.

Here is an example of the use of `sapply()`:

```
sapply(molclock, range)
```

```
     Gpdh  Sod  Xdh AvRate  Myr
[1,]  1.5 12.6 11.5   11.9   55
[2,] 40.0 46.0 31.7   24.9 1100
```

A third argument `na.rm=TRUE` can be supplied to the function `sapply`. This argument is then automatically passed to the function that is given in the second argument position.

Use of `na.rm=TRUE`:

```
sapply(molclock, range,
       na.rm=TRUE)
```

```
     Gpdh  Sod  Xdh AvRate  Myr
[1,]  1.5 12.6 11.5   11.9   55
[2,] 40.0 46.0 31.7   24.9 1100
```

More generally, the first argument to `sapply()` or `lapply()` can be any vector.

`sapply()` – Application of a user function

We will demonstrate the use of `sapply()` to apply a function that counts the number of `NA`s to each column of a data frame. A suitable function can be defined thus:

```
countNA <- function(x)sum(is.na(x))
```

An alternative is to define a function[6] in place, without a name, that counts number of `NA`s. The alternatives are:

[6] This is called an *anonymous* function.

Use function defined earlier:

```
library(MASS)
sapply(Pima.tr2[, 1:5], countNA)
```

```
npreg   glu    bp  skin   bmi
    0     0    13    98     3
```

Define function at place of call:

```
sapply(Pima.tr2[, 1:5],
       function(x)sum(is.na(x)))
```

```
npreg   glu    bp  skin   bmi
    0     0    13    98     3
```

4.3.8 Functions for working with text strings

The functions `paste()` and `paste0()` join text strings. The function `sprintf()`, primarily designed for formatting output for printing, usefully extends the abilities of `paste()` and `paste0()`.

For `paste()`, the default is to use a space as a separator; `paste0()` omits the space.

Other simple string operations include `substring()` and `nchar()` (number of characters). Both of these, and `strsplit()` noted in the next paragraph, can be applied to character vectors.

The function `strsplit()`, used to split strings, has an argument `fixed` that by default equals `FALSE`. The effect is that the argument `split`, which specifies the character(s) on which the string will split, is assumed to be a regular expression. See `help(regexp)` for details. For use of a `split` character argument, call `strsplit()` with `fixed=FALSE`.

Other functions that accept an argument `fixed` include the search functions `grep()` and `regexpr()`, and the search and replace functions `sub()` and `gsub()`.

Bird species in the dataset `cuckoos` (*DAAG*) are:

```
(spec <- levels(cuckoos$species))
```

```
[1] "hedge.sparrow" "meadow.pipit"  "pied.wagtail"
[4] "robin"         "tree.pipit"    "wren"
```

Now replace the periods in the names by spaces:

```
(specnam <- sub(".", " ", spec, fixed=TRUE))
```

```
[1] "hedge sparrow" "meadow pipit"  "pied wagtail"
[4] "robin"         "tree pipit"    "wren"
```

Regular expression substitution:

```
specnam <- sub("\\.",
               " ", spec)
```

In regular expressions enter a period (`"."`) as `"\\."`

For string matching, use `match()`, `pmatch()` and `charmatch()`. For matching with regular expressions, note `grep()` and `regexpr()`. For string substitution, use `sub()` and `gsub()`.

See `help(regex)` for information on the use of regular expressions.

Web pages with information on string manipulation in R include:

http://www.stat.berkeley.edu/classes/s133/R-6.html

http://en.wikibooks.org/wiki/R_Programming/Text_Processing

The first is an overview, with the second more detailed.

The package *stringr*, due to Hadley Wickham, provides what may be a more consistent set of functions for string handling than are available in base R.

For strings representing biological sequences, install the well-documented Bioconductor package *Biostrings*.

4.3.9 Functions for Working with Dates (and Times)

Use `as.Date()` to convert character strings into dates. The default format has year, then month, then day of month, thus:

```
# Electricity Billing Dates
dat <- c("2003-08-24","2003-11-23","2004-02-22",
         "2004-05-03")
dd <- as.Date(dat)
```

Good starting points for learning about dates in R are the help pages `help(Dates)`, `help(as.Date)` and `help(format.Date)`.

Use `format()` to set or change the way that a date is formatted. The following is a selection of the available symbols:

%d:	day, as number
%a:	abbreviated weekday name (%A: unabbreviated)
%m:	month (00-12)
%b:	month abbreviated name (%B: unabbreviated)
%y:	final two digits of year (%Y: all four digits)

The default format is `"%Y-%m-%d"`. The character `/` can be used in place of `-`. Other separators (e.g., a space) must be explicitly specified, using the `format` argument, as in the examples below.

Date objects can be subtracted:

```
as.Date("1960-12-1") - as.Date("1960-1-1")
```

```
Time difference of 335 days
```

Subtraction yields a time difference object. If necessary, use `unclass()` to convert this to a numeric vector.

There is a `diff()` method for date objects:

```
dd <- as.Date(c("2003-08-24","2003-11-23",
                "2004-02-22", "2004-05-03"))
diff(dd)
```

```
Time differences in days
[1] 91 91 71
```

Use `unclass()` to turn a time difference object into an integer vector:

```
unclass(diff(dd))
```

Formatting dates for printing: Use `format()` to fine tune the formatting of dates for printing.

See `help(format.Date)`.

```
dec1 <- as.Date("2004-12-1")
format(dec1, format="%b %d %Y")
```

```
[1] "Dec 01 2004"
```

```
format(dec1, format="%a %b %d %Y")
```

```
[1] "Wed Dec 01 2004"
```

Such formatting may be used to give meaningful labels on graphs. Figure 4.1 provides an example:

```
## Labeling of graph: data frame jobs (DAAG)
library(DAAG); library(lattice)
fromdate <- as.Date("1Jan1995", format="%d%b%Y")
startofmonth <- seq(from=fromdate, by="1 month",
                    length=24)
atdates <- seq(from=fromdate, by="6 month",
               length=4)
xyplot(BC ~ startofmonth, data=jobs,
       scale=list(x=list(at=atdates,
                         labels=format(atdates,
                                       "%b%y"))))
```

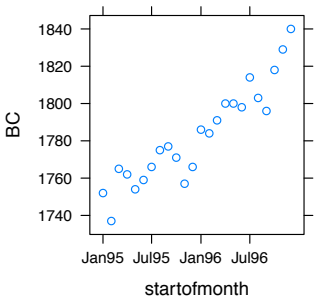

Figure 4.1: Canadian worker force numbers, with dates used to label the x-axis. See Figure 7.12 in Subsection 7.2.6 for data from all Canadian provinces.

Conversion of dates to and from integer number of days: By default, dates are stored in integer numbers of days. Use `julian()` to convert a date into its integer value, by default using January 1 1970 as origin. Use the argument `option` to specify some different origin:

```
dates <- as.Date(c("1908-09-17", "1912-07-12"))
julian(dates)
```

```
[1] -22386 -20992
attr(,"origin")
[1] "1970-01-01"
```

```
julian(dates, origin=as.Date("1908-01-01"))
```

```
[1]  260 1654
attr(,"origin")
[1] "1908-01-01"
```

Note also `weekdays()`, `months()`, and `quarters()`:

```
dates <- as.Date(c("1908-09-17", "1912-07-12"))
weekdays(dates)
```

```
[1] "Thursday" "Friday"
```

```
months(dates)
```

```
[1] "September" "July"
```

```
quarters(dates)
```

```
[1] "Q3" "Q3"
```

Regular sequences of dates: Use the function `help(seq.Date)`.

Given a vector of 'event' times, the following function can be used to count the number of events in each of a regular sequence of time intervals:

```
intervalCounts <- function(date, from=NULL, to=NULL, interval="1 month"){
  if(is.null(from))from <- min(date)
  if(is.null(to))to <- max(date)
  dateBreaks <- seq(from=from, to=to, by=interval)
  dateBreaks <- c(dateBreaks, max(dateBreaks)+diff(dateBreaks[1:2]))
  cutDates <- cut(date, dateBreaks, right=FALSE)
  countDF <- data.frame(Date=dateBreaks[-length(dateBreaks)],
                        num=as.vector(table(cutDates)))
  countDF
}
```

The following counts the number of events by year:

```
dates <- c("1908-09-17", "1912-07-12", "1913-08-06", "1913-09-09", "1913-10-17")
dates <- as.Date(dates)
(byYear <- intervalCounts(dates, from=as.Date("1908-01-01"), interval='1 year'))
```

```
        Date num
1 1908-01-01   1
2 1909-01-01   0
3 1910-01-01   0
4 1911-01-01   0
5 1912-01-01   1
6 1913-01-01   3
```

Further useful functions for working with dates: Note also `date()` which returns the current date and time, and `Sys.Date()` which returns the date. For information on functions for working with times, see `help(ISOdatetime)`.

The CRAN Task View for Time Series Analysis has notes on classes and methods for times and dates, and on packages that give useful functionality

4.3.10 Summaries of Information in Data Frames

A common demand is to obtain a tabular summary of information in each of several columns of a data frame, broken down according to the levels of one or more grouping variables. Consider the data frame `nswdemo` (*DAAG*). Treatment groups are control (`trt==0`) and treatment (`trt==1`) group, with variables `re74` (1974 income), `re75` (1975) and `re78` (1978),

The following calculates the number of zeros for each of the three variables, and for rach of the two treatment categories:

```
## Define a function that counts zeros
countzeros <- function(x)sum(!is.na(x) & x==0)
aggregate(nswdemo[, c("re74", "re75", "re78")],
          by=list(group=nswdemo$trt),
          FUN=countzeros)
```

The data frame is split according to the grouping elements specified in the by argument. The function is then applied to each of the columns in each of the splits.

```
  group re74 re75 re78
1     0  195  178  129
2     1  131  111   67
```

Now find the proportion, excluding `NA`s, that are zero. The result will be printed out with improved labeling of the rows:

```
## countprop() counts proportion of zero values
countprop <- function(x){
    sum(!is.na(x) & x==0)/length(na.omit(x))}
```

```
prop0 <-
  aggregate(nswdemo[, c("re74","re75","re78")],
            by=list(group=nswdemo$trt),
            FUN=countprop)
## Now improve the labeling
rownames(prop0) <- c("Control", "Treated")
round(prop0,2)
```

```
        group re74 re75 re78
Control     0 0.75 0.42 0.30
Treated     1 0.71 0.37 0.23
```

The calculation can alternatively be handled by two calls to the function `sapply()`, one nested within the other, thus:

The argument `z` in the 'in place' function is a data frame. The argument `x` to `countprop()` is a column of a data frame.

```
prop0 <-
  sapply(split(nswdemo[, c("re74","re75","re78")],
               nswdemo$trt),
         FUN=function(z)sapply(z, countprop))
round(t(prop0), 2)
```

```
  re74 re75 re78
0 0.75 0.42 0.30
1 0.71 0.37 0.23
```

4.4 *Classes and Methods (Generic Functions)

Key language constructs:	
Classes	Classes make generic functions (methods) possible.
Methods	Examples are `print()`, `plot()`, `summary()`, etc.

There are two implementation of classes and methods, the original S3 implementation, and the newer S4 implementation that is implemented in the *methods* package. Here, consider the simpler S3 implementation.

All objects have a class. Use the function `class()` to get this information.

For many common tasks there are generic functions – `print()`, `summary()`, `plot()`, etc., whose action varies according to the class of object to which they are applied.

Thus `print()` calls a method thus: factor: `print.factor()`; data frame: `print.data.frame()`; and so on. Ordered factors "inherit" the print method for factors. For objects without an explicit print method, `print.default()` is called.

To get details of the S3 methods that are available for a generic function such as `plot()`, type, e.g., `methods(plot)`. To get a list of the S3 methods that are available for objects of class `lm`, type, e.g., `methods(class="lm")`

4.4.1 *S4 methods

Packages that use S4 classes and methods include *lme4*, Bioconductor packages, and most of the spatial analysis packages.

The S4 conventions and mechanisms extend the abilities available under S3, build in checks that are not available with S3, and are more conducive to good software engineering practice.

Example – a spatial class

The *sp* package defines, among other possibilities, spatial data classes `SpatialPointsDataFrame` and `SpatialGridDataFrame`.

The *sp* function `bubble()`, for plotting spatial measurement data, accepts a spatial data object as argument.[7] The function `coordinates()` can be used, given spatial coordinates, to turn a data frame or matrix into an object of one of the requisite classes.

Data from the data frame `meuse`[8], from the *sp* package, will be used for an example. A first step is to create an object of one of the classes that the function `bubble()` accepts as argument, thus:

```
library(sp)
data(meuse)
class(meuse)
```

```
[1] "data.frame"
```

```
coordinates(meuse) <- ~ x + y
class(meuse)
```

```
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

This has created an object of the class `SpatialPointsDataFrame`.

Code that creates the plot, shown in Figure 4.2, is:

```
bubble(meuse, zcol="zinc", scales=list(tck=0.5),
       maxsize=2, xlab="Easting", ylab="Northing")
```

The function `bubble()` uses the abilities of the `lattice` package. It returns a `trellis` graphics object.

The coordinates can be extracted using `coordinates(meuse)`. Remaining columns from the original data frame are available from the data frame `meuse@data`.

Use `slotNames()` to examine the structure of the object:

```
slotNames(meuse)
```

```
[1] "data"        "coords.nrs"  "coords"
[4] "bbox"        "proj4string"
```

Typing `names(meuse)` returns the column names for the `data` slot. The effect is the same as that of typing `names(meuse@data)`. To get a list of the S4 methods that are available for a generic function, use `showMethods()`. Section 12.4 has further details.

Classes defined in the *sp* package are widely used across R spatial data analysis packages.

[7] Each point (location) is shown as a bubble, with area proportional to a value for that point.

[8] Data are from the floodplain of the river Meuse, in the Netherlands. It includes concentrations of various metals (`cadmium`, `copper`, `lead`, `zinc`), with Netherlands topographical map coordinates.

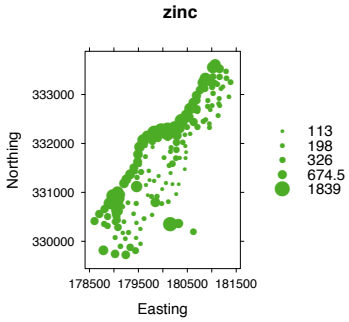

Figure 4.2: Bubble plot for `zinc` concentrations. Areas of bubbles are proportional to concentrations.

Note that `meuse@data` is shorthand for `slot(meuse, "data")`.

4.5 Common Sources of Surprise or Difficulty

Character vectors, when incorporated as columns of a data frame, become by default factors.

Factors can often be treated as vectors of text strings, with values given by the factor levels. Watch however for contexts where the integer codes are used instead.

Use `is.na()` to check for missing values. Do not try to test for equality with `NA`. Refer back to Section 4.1.3.

If there is a good alternative, avoid the attaching of data frames. If you do use this mechanism, be aware of the traps.

Assignment of new values to an attached data frame creates a new local data frame with the same name. The new local copy remains in the workspace when the data frame is detached.

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. Use the function `unlist()` to extract the vector of numeric values.

4.6 Summary

Important R data structures are vectors, factors, data frames and lists. Vector modes include numeric, logical, character or complex.

Factors, used for categorical data, can be important in the use of many of R's modeling functions. Ordered factors are appropriate for use with ordered categorical data.

Use `table()` for tables of counts, and `xtabs()` for tables of counts or totals.

R allows the use of infinite Values (`Inf` or `-Inf`) and `NaN`s (not a number) in calculations. Introduce such quantities into your calculations only if you understand the implications.

A matrix is a vector that is stacked column upon column into a rectangular array that has dimensions given by its dimension attribute. A data frame is, by contrast, a list of columns.

Matrices are in some (not all) contexts handled similarly to data frames whose elements are all of one type (typically all numeric).

Calculations with matrices are likely to be much faster than with data frames.

Lists are "non-atomic" vectors. Use the function `c()` (concatenate) to join lists, just as for "atomic" vectors.

Modeling functions typically output a *model object* that has a list structure. This holds information from the model fit, in a form from which generic model functions can then extract commonly required forms of output.

Generic functions that may be used with model objects typically include `print()`, `summary()`, `fitted()`, `coef()` and `resid()`.

4.7 Exercises

1. Find an R function that will sort a vector. Give an example.

2. Modify the function `mean.and.sd()` so that it outputs, in addition to mean and standard deviation, the number of vector elements.

3. *What is the mode of: (i) a factor; (ii) a dataframe?; (iii) a list that is not necessarily a dataframe? Apply the function `mode()` to objects of each of these classes. Explain what you find.

4. The attempt to assign values to an expression whose subscripts include missing values generates an error. Run the following code and explain the error that results:

```
y <- c(1, NA, 3, 0, NA)
y[y > 0]
y[y > 0] <- c(11, 12)
```

5. 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
```

The output from the final `table(gender)` is

```
gender
  Male female
     0     91
```

Explain the numbers that appear.

6. In the data set `nswpsdi1` (`DAAGxtras`), do the following for each of the two levels of `trt`:

(a) Determine the numbers for each of the levels of `black`;

(b) Determine the numbers for each of the levels of `hispanic`; item Determine the numbers for each of the levels of `marr` (married).

7. Sort the rows in the data frame `Acmena` in order of increasing values of `dbh`.

[Hint: Use the function `order()`, applied to `age` to determine the order of row numbers required to sort rows in increasing order of age. Reorder rows of `Acmena` to appear in this order.]

```
Acmena <- subset(rainforest, species=="Acmena smithii")
ord <- order(Acmena$dbh)
acm <- Acmena[ord, ]
```

Sort the row names of `possumsites` (*DAAG*) into alphanumeric order. Reorder the rows of `possumsites` in order of the row names.

8(a) Create a `for` loop that, given a numeric vector, prints out one number per line, with its square and cube alongside.

(b) Look up `help(while)`. Show how to use a `while` loop to achieve the same result.

(c) Show how to achieve the same result without the use of an explicit loop.

9. Here are examples that illustrate the use of `paste()` and `paste0()`:

```
paste("Leo", "the", "lion")
paste("a", "b")
paste0("a", "b")
paste("a", "b", sep="")
paste(1:5)
paste(1:5, collapse="")
```

What are the respective effects of the parameters `sep` and `collapse`?

10. The following function calculates the mean and standard deviation of a numeric vector.

```
meanANDsd <- function(x){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, sd = sdev) # The function returns this vector
}
```

Modify the function so that: (a) the default is to use `rnorm()` to generate 20 random normal numbers, and return the standard deviation; (b) if there are missing values, the mean and standard deviation are calculated for the remaining values.

11. Try the following:

```
class(2)
class("a")
class(cabbages$HeadWt)     # cabbages is in the datasets package
class(cabbages$Cult)
```

Now do `sapply(cabbages, class)`, and note which columns hold numerical data. Extract those columns into a separate data frame, perhaps named `numtinting`.

[Hint: `cabbages[, c(2,3)]` is not the correct answer, but it is, after a manner of speaking, close!]

12. Functions that may be used to get information about data frames include `str()`, `dim()`, `row.names()` and `names()`. Try each of these functions with the data frames `allbacks`, `ant111b` and `tinting` (all in *DAAG*).

For getting information about each column of a data frame, use `sapply()`. For example, the following applies the function `class()` to each column of the data frame `ant111b`.

```
library(DAAG)
sapply(ant111b, class)
```

For columns in the data frame `tinting` that are factors, use `table()` to tabulate the number of values for each level.

5
Data Input and Storage

*5.1 *Data Input from a File*

.

Use of the RStudio menu is recommended. This is fast, and allows a visual check of the data layout before input proceeds. If input options are incorrectly set, these can be changed as necessary before proceeding. The code used for input is shown. In those rare cases where input options are required for which the menu does not make provision, the command line code can be edited as needed, before proceeding.

Most data input functions allow import from a file that is on the web — give the URL when specifying the file. Another possibility is to copy the file, or a relevant part of it, to the clipboard. For reading from and writing to the clipboard under Windows, see `http://bit.ly/2sxyOhG`. For MacOS, see `http://bit.ly/2t1nX0I`

5.1.1 Managing input is from the RStudio menu

Data input that is initiated from the RStudio menu uses functions from the package `readr` for input of tabular data. The function `readr::read_table()` replaces `read.table()`, `readr::read_csv()` replaces `read.csv()`, and similarly for other `read.table()` aliases.

It uses the function `readxl::readxl()` for Excel spreadsheet data. There is provision, also, using functions from the package `haven`, to import data from SPSS (POR and SAV files), from SAS (XPT and SAS files), and from Stata (DTA files).

Output is in all cases to a tibble, which is a specialized form of data frame. Character columns are not automatically converted to factors, column names are not converted into valid R identifiers, and row names are not set. For subsequent processing, there are important differences between tibbles and data frames that users need to note.

It is important to check, when data have been entered, that data values appear sensible. Do minimal checks on: ranges of variable values, the mode of the input columns (numeric or factor, or . . .). Scatterplot matrices are helpful both for checking variable ranges and for identifying impossible or unusual combinations of variable values.

See `vignette("semantics", package="haven")` for details of the way that labelled data and missing values are handled, for input from SPSS, SAS, and Stata.

5.1.2 Input using the `read.table()` family of functions

There are several aliases for `read.table()` that have different settings for input defaults. Note in particular `read.csv()`, for reading in comma delimited **.csv** files such as can be output from Excel spreadsheets. See `help(read.table)`. Recall that

Non-default option settings can however, for very large files, severely slow data input.

For factor columns check that the levels are as expected.

- Character vectors are by default converted into factors. To prevent such type conversions, specify `stringsAsFactors=FALSE`.
- Specify `heading=TRUE`[1] to indicate that the first row of input has column names. Use `heading=FALSE` to indicate that it holds data. [If names are not given, columns have default names `V1`, `V2`, ...]
- Use the parameter `row.names`, then specifying a column number, to specify a column for use to provide row names.

[1] By default, if the first row of the file has one less field than later rows, it is taken to be a header row. Otherwise, it is taken as the first row of data.

Issues that may complicate input

Where data input fails, consider using `read.table()` with the argument `fill=TRUE`, and carefully check the input data frame. Blank fields will be implicitly added, as needed, so that all records have an equal number of identified fields.

Carefully check the parameter settings[2] for the version of the input command that is in use. It may be necessary to change the field separators (specify `sep`), and/or the missing value character(s) (specify `na.strings`). Embedded quotes and comment characters (#; by default anything that follows # on the same line is ignored.) can be a source of difficulty.

Where a column that should be numeric is converted to a factor this is an indication that it has one or more fields that, as numbers, would be illegal. For example, a "1" (one) may have been mistyped as an "l" (ell), or "0" (zero) as "O" (oh).

Note options that allow the limiting of the number of input rows. For `read.table()`) and aliases, set `nrows`. For functions from the `readr` package, set `n_max`. For `scan()`, discussed in the next subsection, set `nlines`. All these functions accept the argument `skip`, used to set the number of lines to skip before input starts.

NB also that `count.fields()` counts the number of fields in each record — albeit watch for differences from input fields as detected by the input function.

[2] For text with embedded single quotes, set `quote = ""`. For text with # embedded; change `comment.char` suitably.

Among other possibilities, there may be a non-default missing value symbol (e.g., `"."`), but without using `na.strings` to indicate this.

5.1.3 **The use of scan() for flexible data input*

Data records may for example spread over several rows. There seems no way for `read.table()` to handle this.

The following code demonstrates the use of `scan()` to read in the file **molclock1.txt**. To place this file in your working directory, attach the *DAAG* package and type `datafile("molclock1")`.

```
colnam <- scan("molclock1.txt", nlines=1, what="")
molclock <- scan("molclock1.txt", skip=1,
                 what=c(list(""), rep(list(1),5)))
molclock <- data.frame(molclock, row.names=1)
  # Column 1 supplies row names
names(molclock) <- colnam
```

There are two calls to `scan()`, each time taking information from the file **molclock1.txt**. The first, with `nlines=1` and `what=""`, input the column names. The second, with `skip=1` and `what=c(list(""), rep(list(1),5)))]`, input the several rows of data.

The `what` parameter should be a list, with one list element for each field in a record. The "" in the first list element indicates that the data is to be input as character. The remaining five list elements are set to 1, indicating numeric data. Where records extend over several lines, set `multi.line=TRUE`.

For repeated use with data files that have a similar format, consider putting the code into a function, with the `what` list as an argument.

5.1.4 *The* `memisc` *package: input from SPSS and Stata*

The *memisc* package has highly effective abilities for examining and inputting data from various SPSS formats. These include **.sav**, **.por**, and Stata **.dta** data types. Note in particular the ability to check the contents of the columns of the dataset before importing part or all of the file.

Note also the `haven` package, mentioned above, and the *foreign* package. The *foreign* package has functions that allow input of various types of files from Epi Info, Minitab, S-PLUS, SAS, SPSS, Stata, Systat and Octave. There are abilities for reading and writing some dBase files. For further information, see the R Data Import/Export manual.

An initial step is to use an importer function to create an *importer* object. As of now, *importer* functions are: `spss.fixed.file()`, `spss.portable.file()` (**.por** files), `spss.system.file()` (**.sav** files), and `Stata.file()` (**.dta** files). The importer object has information about the variables: including variable labels, value labels, missing values, and for an SPSS 'fixed' file the columns that they occupy, etc.

Additionally, it has also information from further processing of the file header and/or the file proper that is needed in preparation for importing the file.

Functions that can be used with an importer object include:

- `description()`: column header information;
- `codebook()`: detailed information on each column;
- `as.data.set()`: bring the data into R, as a 'data.set' object;
- `subset()`: bring a subset of the data into R, as a 'data.set' object

The functions `as.data.set()` and `subset()` yield 'data.set' objects. These have structure that is additional to that in data frames. Most functions that are available for use with data frames can be used with data.set class objects.

Use `as.data.frame()` to coerce data.set objects into data frames. Information that is not readily retainable in a data frame format may be lost in the process.

The vignette `anes48` that comes with the *memisc* package illustrates the use of the above abilities.

Example

A compressed version of the file "NES1948.POR" (an SPSS 'portable' dataset) is stored as part of the *memisc* installation. The following does the unzipping, places the file in a temporary directory, and stores the path to the file in the text string `path2file`:

To substitute your own file, store the path to the file in `path2file`.

```
library(memisc)
## Unzip; return path to "NES1948.POR"
path2file <- unzip(system.file("anes/NES1948.ZIP",package="memisc"),
                   "NES1948.POR",exdir=tempfile())
```

Now create an 'importer' object, and get summary information:

```
# Get information about the columns in the file
nes1948imp <- spss.portable.file(path2file)
show(nes1948imp)
```

```
SPSS portable file '/var/folders/00/_kpyywm16hnbs2c0dvlf0mwr0000gq/T//Rtmp9uICLO/file4f0
        with 67 variables and 662 observations
```

There will be a large number of messages that draw attention to duplicate labels.

Use `labels()`) to change labels, or `missing.values()` to set missing value filters, prior to data import.

Before importing, it may be well to check details of what is in the file. The following, which restricts attention to columns 4 to 9 only, indicates the nature of the information that is provided.

```
## Get details about the columns (here, columns 4 to 9 only)
description(nes1948imp)[4:9]
```

```
$v480002
[1] "INTERVIEW NUMBER"

$v480003
[1] "POP CLASSIFICATION"

$v480004
[1] "CODER"

$v480005
[1] "NUMBER OF CALLS TO R"

$v480006
[1] "R REMEMBER PREVIOUS INT"

$v480007
[1] "INTR INTERVIEW THIS R"
```

As there are in this instance 67 columns, it might make sense to look at columns perhaps 10 at a time.

More detailed information is available by using the R function `codebook()`. The following gives the codebook information for column 5:

```
## Get codebook information for column 5
codebook(nes1948imp[, 5])
```

This is more interesting than what appears for columns (1 - 4).

```
------------------------------------------------------------

   nes1948imp[, 5] 'POP CLASSIFICATION'

------------------------------------------------------------

   Storage mode: double
   Measurement: nominal

         Values and labels    N    Percent

   1   'METROPOLITAN AREA'  182   27.5 27.5
   2   'TOWN OR CITY'       354   53.5 53.5
   3   'OPEN COUNTRY'       126   19.0 19.0
```

The following imports a subset of just four of the columns:

```
vote.socdem.48 <- subset(nes1948imp,
              select=c(
                  v480018,
                  v480029,
                  v480030,
                  v480045
                  ))
```

To import all columns, do:

```
socdem.48 <- as.data.set(nes1948imp)
```

For more detailed information, type:

```
## Go to help page for 'importers'
help(spss.portable.file)
```

Look also at the vignette:

```
vignette("anes48")
```

5.2 *Input of Data from a web page

This section notes some of the alternative ways in which data that is available from the web can be input into R. The first subsection below comments on the use of a point and click interface to identify and download data.

A point and click interface is often convenient for an initial look. Rather than downloading the data and then inputting it to R, it may be better to input it directly from the web page. Direct input into R has the advantage that the R commands that are used document exactly what has been done.[3]

Note that the functions `read.table()`, `read.csv()`, `scan()`, and other such functions, are able to read data directly from a file that is available on the web. There is a limited ability to input part only of a file.

Suppose however that the demand is to downlaod data for several of a large number of variables, for a specified range of years, and for a specified geographical area or set of countries. A number of data archives now offer data in one or more of several markup formats that assist selective access. Formats include XML, GML, JSON and JSONP.

The web page: http://www.visualizing.org/data/browse/ has an extensive list of web data sources. The World Bank Development Indicators database will feature prominently in the discussion below.

[3] This may be especially important if a data download will be repeated from time to time with updated data, or if data are brought together from a number of different files, or if a subset is taken from a larger database.

GML, or Geography Markup Language, is based on XML.

A browser interface to World Bank data: The web page http://databank.worldbank.org/data/home.aspx[4] gives a point and click interface to, among other possibilities, the World Bank development indicator database. Clicking on any of 20 country names that are displayed shows data for these countries for 1991-2010, for 54 of the 1262 series that were available at last check. Depending on the series, data may be available back to 1964. Once selections have been made, click on DOWNLOAD to download the data. For input into R, downloading as a **.csv** file is convenient.

Manipulation of these data into a form suitable for a motion chart display was demonstrated in Subsection 6.2.3

[4] Click on COUNTRY to modify the choice of countries. To expand (to 246) countries beyond the 20 that appear by default, click on Add more country. Click on SERIES and TIME to modify and/or expand those choices. Click on Apply Changes to set the choices in place.

Australian Bureau of Meteorology data: Graphs of area-weighted time series of rainfall and temperature measures, for various regions of Australia, can be accessed from the Australian Bureau of Meteorology web page http://www.bom.gov.au/cgi-bin/climate/change/timeseries.cgidemo. Click on Raw data set[5] to download the raw data.

[5] To copy the web address, right click on Raw data set and click on Copy Link Location (Firefox) or Copy Link Address (Google Chrome) or Copy Link (Safari).

Once the web path to the file that has the data has been found, the data can alternatively be input directly from the web. The following gets the annual total rainfall in Eastern Australia, from 1910 through to the present':

```
webroot <- "http://www.bom.gov.au/web01/ncc/www/cli_chg/timeseries/"
rpath <- paste0(webroot, "rain/0112/eaus/", "latest.txt")
totrain <- read.table(rpath)
```

A function to download multiple data series: The following accesses the latest annual data, for total rainfall and average temperature, from the command line:

```
getbom <-
function(suffix=c("AVt","Rain"), loc="eaus"){
        webroot <- "http://www.bom.gov.au/web01/ncc/www/cli_chg/timeseries/"
        midfix <- switch(suffix[1], AVt="tmean/0112/", Rain="rain/0112/")
        webpage <- paste(webroot, midfix, loc, "/latest.txt", sep="")
        print(webpage)
        read.table(webpage)$V2
        }
##
## Example of use
offt = c(seaus=14.7, saus=18.6, eaus=20.5,  naus=24.7, swaus=16.3,
         qld=23.2, nsw=17.3, nt=25.2,sa=19.5, tas=10.4, vic=14.1,
         wa=22.5, mdb=17.7, aus=21.8)
z <- list()
for(loc in names(offt))z[[loc]] <- getbom(suffix="Rain", loc=loc)
bomRain <- as.data.frame(z)
```

The function can be re-run each time that data is required that includes the most recent year.

*Extraction of data from tables in web pages

The function `readHTMLTable()`, from the *XML* package, will prove very useful for this. It does not work, currenty at least, for pages that use https:.

Historical air crash datra: The web page `http://www.planecrashinfo.com/database.htm` has links to tables of aviation accidents, with one table for each year. The table for years up to and including 1920 is on the web page `http://www.planecrashinfo.com/1920/1920.htm`, that for 1921 on the page `http://www.planecrashinfo.com/1921/1921.htm`, and so on through until the most recent year. The following code inputs the table for years up to and including 1920:

```
library(XML)
```

```
url <- "http://www.planecrashinfo.com/1920/1920.htm"
to1920 <- readHTMLTable(url, header=TRUE)
to1920 <- as.data.frame(to1920)
```

The following inputs data from 2010 through until 2014:

```
url <- paste0("http://www.planecrashinfo.com/",
              2010:2014, "/", 2010:2014, ".htm")
tab <- sapply(url, function(x)readHTMLTable(x, header=TRUE))
```

```
## The following less efficent alternative code spells the steps out in more detail
## tab <- vector('list', 5)
## k <- 0
## for(yr in 2010:2014){
##  k <- k+1
##  url <- paste0("http://www.planecrashinfo.com/", yr, "/", yr, ".htm")
##  tab[[k]] <- as.data.frame(readHTMLTable(url, header=TRUE))
## }
```

Now combine all the tables into one:

```
## Now combine the 95 separate tables into one
airAccs <- do.call('rbind', tab)
names(airAccs) <- c("Date", "Location/Operator",
                    "AircraftType/Registration", "Fatalities")
airAccs$Date <- as.Date(airAccs$Date, format="%d %b %Y")
```

The help page `help(readHTMLTable)` gives examples that demonstrate other possibilities.

5.2.1 *Embedded markup — XML and alternetives

Data are are now widely available, from a number of differet web sites, in one or more of several markup formats. Markup code, designed to make the file self-describing, is included with the data. The user does not need to supply details of the data structure to the software reading the data.

Markup languages that may be used include XML, GML, JSON and JSONP. Queries are built into the web address. Alternatives to setting up the query directly may be:

For details of markup use, as they relate to the World Bank Development Indicators database, see `http://data.worldbank.org/node/11`.

- Use a function such as `fromJSON()` in the *RJSONIO* package to set up the link and download the data;
- In a few cases, functions have been provided in R packages that assist selection and downloading of data. For the World Bank Development Indicators database, note `WDI()` and other functions in the *WDI* package.

Download of NZ earthquake data: Here the GML markup conventions are used, as defined by the WFS OGC standard. Details can be found on the website `http://info.geonet.org.nz/display/appdata/Earthquake+Web+Feature+Service`

WFS is Web Feature Service. OGC is Open Geospatial Consortium. GML is Geographic Markup language GML, based on XML.

The **.csv** format is one of several formats in which data can be retrieved.

The following extracts earthquake data from the New Zealand GeoNet website. Data is for 1 September 2009 onwards, through until the current date, for earthquakes of magnitude greater than 4.5.

```
## Input data from internet
from <-
  paste(c("http://wfs-beta.geonet.org.nz/",
          "geoserver/geonet/ows?service=WFS",
          "&version=1.0.0",
```

```
          "&request=GetFeature",
          "&typeName=geonet:quake",
          "&outputFormat=csv",
          "&cql_filter=origintime>='2009-08-01'",
          "+AND+magnitude>4.5"),
        collapse="")
quakes <- read.csv(from)
z <- strsplit(as.character(quakes$origintime),
              split="T")
quakes$Date <- as.Date(sapply(z, function(x)x[1]))
quakes$Time <- sapply(z, function(x)x[2])
```

World Bank data — using the WDI *package* Use the function `WDIsearch()` to search for indicators. Thus, to search for indicators with "CO2" in their name, enter `WDIsearch('co2')`. Here are the first 4 (out of 38) that are given by such a search:

```
library(WDI)
WDIsearch('co2')[1:4,]
```

```
     indicator
[1,] "EN.ATM.CO2E.CP.KT"
[2,] "EN.ATM.CO2E.EG.ZS"
[3,] "EN.ATM.CO2E.FF.KT"
[4,] "EN.ATM.CO2E.FF.ZS"
     name
[1,] "CO2 emissions from cement production (thousand metric tons)"
[2,] "CO2 intensity (kg per kg of oil equivalent energy use)"
[3,] "CO2 emissions from fossil-fuels, total (thousand metric tons)"
[4,] "CO2 emissions from fossil-fuels (% of total)"
```

Use the function `WDI()` to input indicator data, thus:

```
library(WDI)
inds <- c('SP.DYN.TFRT.IN','SP.DYN.LE00.IN', 'SP.POP.TOTL',
 'NY.GDP.PCAP.CD', 'SE.ADT.1524.LT.FE.ZS')
indnams <- c("fertility.rate", "life.expectancy", "population",
             "GDP.per.capita.Current.USD", "15.to.25.yr.female.literacy")
names(inds) <- indnams
wdiData <- WDI(country="all",indicator=inds, start=1960, end=2013, extra=TRUE)
colnum <- match(inds, names(wdiData))
names(wdiData)[colnum] <- indnams
## Drop unwanted "region"
WorldBank <- droplevels(subset(wdiData, !region %in% "Aggregates"))
```

The effect of `extra=TRUE` is to include the additional variables `iso2c` (2-character country code), `country`, `year`, `iso3c` (3-character country code), `region`, `capital`, `longitude`, `latitude`, `income` and `lending`.

The data frame `Worldbank` that results is in a form where it can be used with the *googleVIS* function `gvisMotionChart()`, as described in Section 7.5.1

The function `WDI()` calls the non-visible function `wdi.dl()`, which in turn calls the function `fromJSON()` from the *RJSONIO* package. To see the code for `wdi.dl()`, type `getAnywhere("wdi.dl")`.

5.3 *Creating and Using Databases*

In addition to the *RSQLite*, note the *RMySQL* and *ROracle* packages. All use the interface provided by the *DBI* package.

The *RSQLite* package makes it possible to create an SQLite database, or to add new rows to an existing table, or to add new table(s), within an R session. The SQL query language can then be used to access tables in the database. Here is an example. First create the database:

```
library(DAAG)
library(RSQLite)
driveLite <- dbDriver("SQLite")
con <- dbConnect(driveLite, dbname="hillracesDB")
dbWriteTable(con, "hills2000", hills2000,
             overwrite=TRUE)
dbWriteTable(con, "nihills", nihills,
             overwrite=TRUE)
dbListTables(con)
```

```
[1] "hills2000" "nihills"
```

The database `hillracesDB`, if it does not already exist, is created in the working directory.

Now input rows 16 to 20 from the newly created database:

```
## Get rows 16 to 20 from the nihills DB
dbGetQuery(con,
  "select * from nihills limit 5 offset 15")
```

```
  dist climb   time  timef
1  5.5  2790 0.9483 1.2086
2 11.0  3000 1.4569 2.0344
3  4.0  2690 0.6878 0.7992
4 18.9  8775 3.9028 5.9856
5  4.0  1000 0.4347 0.5756
```

```
dbDisconnect(con)
```

5.4 **File compression:*

The functions for data input in versions 2.10.0 and later of R are able to accept certain types of compressed files. This extends to `scan()` and to functions such as `read.maimages()` in the *limma* package, that use the standard R data input functions.

By way of illustration, consider the files **coral551.spot**, . . . , **coral556.spot** that are in the subdirectory **doc** of the *DAAGbio* package. In a directory that held the uncompressed files, they were created by typing, on a Unix or Unix-like command line:

Severer compression: replace `gzip -9` by `xz -9e`.

```
gzip -9 coral55?.spot
```

The **.zip** files thus created were renamed back to ***.spot*** files.

When saving large objects in image format, specify `compress=TRUE`. Alternatives that may lead to more compact files are `compress="bzip2"` and `compress="xz"`.

Note also the R functions `gzfile()` and `xzfile()` that can be used to create files in a compressed text format. This might for example be text that has been input using `readLines()`.

5.5 Summary

Following input, perform minimal checks that values in the various columns are as expected.

With very large files, it can be helpful to read in the data in chunks (ranges of rows).

Note mechanisms for direct input of web data. Many data archives now offer one or more of several markup formats that facilitate selective access.

6
Data Manipulation and Management

Data analysis has as its end point the use of forms of data summary that will convey, fairly and succinctly, the information that is in the data. The fitting of a model is itself a form of data summary.

Be warned of the opportunities that simple forms of data summary, which seem superficially harmless, can offer for misleading inferences. These issues affect, not just data summary per se, but all modeling. Data analysis is a task that should be undertaken with critical faculties fully engaged.

Data summaries that can lead to misleading inferences arise often, from a unbalance in the data and/or failure to account properly for important variables or factors.

Alternative types of data objects

Column objects: These include (atomic) vectors, factors, and dates.

Date and date-time objects: The creation and manipulations of date objects will be described below.

Data Frames: These are rectangular structures. Columns may be "atomic" vectors, or factors, or other objects (such as dates) that are one-dimensional.

A data frame is a list of column objects, all of the same length.

Matrices and arrays: Matrices[1] are rectangular arrays in which all elements have the same mode. An array is a generalization of a matrix to allow an arbitrary number of dimensions.

[1] Internally, matrices are one long vector in which the columns follow one after the other.

Tables: A table is a specialized form of array.

Lists: A list is a collection of objects that can be of arbitrary class. List elements are themselves lists. In more technical language, lists are *recursive* data structures.

S3 model objects: These are lists that have a defined structure.

S4 objects: These are specialized data structures with tight control on the structure. Unlike S3 objects, they cannot be manipulated as lists. Modeling functions in certain of the newer packages[2] return S4 objects.

[2] These include *lme4*, the Bioconductor packages, and the spatial analysis packages.

6.1 Manipulations with Lists, Data Frames and Arrays

Recall that data frames are lists of columns that all have the same length. They are thus a specialised form of list. Matrices are two-dimensional arrays. Tables are in essence arrays that hold numeric values.

6.1.1 Tables and arrays

The dataset `UCBAdmissions` is stored as a 3-dimensional table. If we convert it to an array, very little changes:

It changes from a table object to a numeric object, which affects the way that it is handled by some functions. In either case, what we have is a numeric vector of length 24 ($= 2 \times 2 \times 6$) that is structured to have dimensions 2 by 2 by 6.

6.1.2 Conversion between data frames and tables

The three-way table `UCBAdmissions` are admission frequencies, by Gender, for the six largest departments at the University of California at Berkeley in 1973. For a reference to a web page that has the details; see the help page for `UCBAdmissions`. Type

```
help(UCBAdmissions)     # Get details of the data
example(UCBAdmissions)
```

Note the margins of the table:

```
str(UCBAdmissions)
```

```
 table [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
 - attr(*, "dimnames")=List of 3
  ..$ Admit : chr [1:2] "Admitted" "Rejected"
  ..$ Gender: chr [1:2] "Male" "Female"
  ..$ Dept  : chr [1:6] "A" "B" "C" "D" ...
```

In general, operations with a table or array are easiest to conceptualise if the table is first converted to a data frame in which the separate dimensions of the table become columns. Thus, the `UCBAdmissions` table will be converted to a data frame that has columns `Admit`, `Gender` and `Dept`. Either use the `as.data.frame.table()` command from base R, or use the `adply()` function from the *plyr* package.

The following uses the function `as.data.frame.table()` to convert the 3-way table `UCBAdmissions` into a data frame in which the margins are columns:

As `UCBAdmissions` is a table (not an array), `as.data.frame(UCBAdmissions)` will give the same result.

```
UCBdf <- as.data.frame.table(UCBAdmissions)
head(UCBdf, 5)
```

```
     Admit Gender Dept Freq
1 Admitted   Male    A  512
```

```
2 Rejected   Male     A  313
3 Admitted Female     A   89
4 Rejected Female     A   19
5 Admitted   Male     B  353
```

Alternatively, use the function `adply()` from the *plyr* package that is described in Section 6.2. Here the `identity()` function does the manipulation, working with all three dimensions of the array:

```
library(plyr)
UCBdf <- adply(.data=UCBAdmissions,
               .margins=1:3,
               .fun=identity)
names(UCBdf)[4] <- "Freq"
```

First, calculate overall admission percentages for females and males. The following calculates also the total accepted, and the total who applied:

```
library(dplyr)
gpUCBgender <- dplyr::group_by(UCBdf, Gender)
AdmitRate <- dplyr::summarise(gpUCBgender,
                              Accept=sum(Freq[Admit=="Admitted"]),
                              Total=sum(Freq),
                              pcAccept=100*Accept/Total)
AdmitRate
```

```
# A tibble: 2 x 4
  Gender Accept Total pcAccept
  <fctr>  <dbl> <dbl>    <dbl>
1   Male   1198  2691    44.52
2 Female    557  1835    30.35
```

Now calculate admission rates, total number of females applying, and total number of males applying, for each department:

```
gpUCBgd <- dplyr::group_by(UCBdf, Gender, Dept)
rateDept <- dplyr::summarise(gpUCBgd,
    Total=sum(Freq),
    pcAccept=100*sum(Freq[Admit=="Admitted"])/Total)
```

Results can conveniently be displayed as follows. First show admission rates, for females and males separately:

```
xtabs(pcAccept~Gender+Dept, data=rateDept)
```

```
        Dept
Gender        A      B      C      D      E      F
  Male   62.061 63.036 36.923 33.094 27.749  5.898
  Female 82.407 68.000 34.064 34.933 23.919  7.038
```

Now show total numbers applying:

```
xtabs(Total~Gender+Dept, data=rateDept)
```

```
        Dept
Gender     A   B   C   D   E   F
  Male   825 560 325 417 191 373
  Female 108  25 593 375 393 341
```

As a fraction of those who applied, females were strongly favored in department A, and males somewhat favored in departments C and E. Note however that relatively many males applied to A and B, where admission rates were high. This biased overall male rates upwards. Relatively many females applied to C, D and F, where rates were low. This biased the overall female rates downwards.

The overall bias arose because males favored departments where admission rates were relatively high.

6.1.3 *Table margins*

For working directly on tables, note the function `margin.table()`. The following retains margin 1 (`Admit`) and margin 2 (`Gender`), adding over `Dept` (the remaining margin):

```
## Tabulate by Admit (margin 2) & Gender (margin 1)
(margin21 <- margin.table(UCBAdmissions,
                          margin=2:1))
```

```
        Admit
Gender   Admitted Rejected
  Male       1198     1493
  Female      557     1278
```

Take margin 2, first, then margin 1, gving a table where rows correspond to levels of `Gender`.

Use the function `margin.table()` to turn this into a table that has the proportions in each row:

```
prop.table(margin21, margin=1)
```

```
        Admit
Gender   Admitted Rejected
  Male     0.4452   0.5548
  Female   0.3035   0.6965
```

6.1.4 *Categorization of continuous data*

The data frame `bronchit`, in the *DAAGviz* package, has observations on 212 men in a sample of Cardiff (Wales, UK) enumeration districts. Variables are `r` (1 if respondent suffered from chronic bronchitis and 0 otherwise), `cig` (number of cigarettes smoked per day) and `poll` (the smoke level in the locality).

The dataset `bronchit` may alternatively be found in the *SMIR* package.

It will be convenient to define a function `props` that calculates the proportion of the total in the first (or other nominated element) of a vector:

```
props <- function(x, elem=1)sum(x[elem])/sum(x)
```

Now use the function `cut()` to classify the data into four categories, and form tables:

```
library(DAAGviz)
catcig <- with(bronchit,
               cut(cig, breaks=c(0,1,10,30),
                   include.lowest=TRUE))
tab <- with(bronchit, table(r, catcig))
round(apply(tab, 2, props, elem=2), 3)
```

The argument `breaks` can be either the number of intervals, or it can be a vector of break points such that all data values lie within the range of the breaks. If the smallest of the break points equals the smallest data value, supply the argument `include.lowest=TRUE`.

```
  [0,1]  (1,10] (10,30]
  0.072   0.281   0.538
```

There is a clear increase in the risk of bronchitis with the number of cigarettes smoked.

This categorization was purely for purposes of preliminary analysis. Categorization for purposes of analysis is, with the methodology and software that are now available, usually undesirable. Tables that are based on categorization can nevertheless be useful in data exploration.

It was at one time common practice to categorize continuous data, in order to allow analysis methods for multi-way tables. There is a loss of information, which can at worst be serious.

6.1.5 *Matrix Computations

Let `X` (n by p), `Y` (n by p) and B (p by k) be numeric matrices. Some of the possibilities are:

```
X + Y              # Elementwise addition
X * Y              # Elementwise multiplication
X %*% B            # Matrix multiplication
solve(X, Y)        # Solve X B = Y for B
svd(X)             # Singular value decomposition
qr(X)              # QR decomposition
t(X)               # Transpose of X
```

Note that if `t()` is used with a data frame, a matrix is returned. If necessary, all values are coerced to the same mode.

Calculations with data frames that are slow and time consuming will often be much faster if they can be formulated as matrix calculations. This is in general become an issue only for very large datasets, with perhaps millions of observations. Section 6.4 has examples. For small or modest-sized datasets, convenience in formulating the calculations is likely to be more important than calculation efficiency.

Section 4.3.7 will discuss the use of `apply()` for operations with matrices, arrays and tables.

6.2 plyr, dplyr & reshape2 *Data Manipulation*

The *plyr* package has functions that together:

- provide a systematic approach to computations that perform a desired operation across one or more dimensions of an array, or of a data frame, or of a list;
- allow the user to choose whether results will be returned as an array, or as a data frame, or as a list.

The *dplyr* package has functions for performing various summary and other operations on data frames. For many purposes, it supersedes the *plyr* package.

The *reshape2* package is, as its name suggests, designed for moving between alternative data layouts.

6.2.1 plyr

The *plyr* package has a separate function for each of the nine possible mappings. The first letter of the function name (one of `a` = array,

d = data frame, l = list) denotes the class of the input object, while the second letter (the same choice of one of three letters) denotes the class of output object that is required. This pair of letters is then followed by ply.

Here is the choice of functions:

	Class of Output Object		
	a (array)	d (data frame)	l (list)
Class of Input Object			
a (array)	aaply	adply	alply
d (data frame)	daply	ddply	dlply
l (list)	laply	ldply	llply

First observe how the function adply can be used to change from a tabular form of representation to a data frame. The dimension names will become columns in the data frame.

```
detach("package:dplyr")
library(plyr)
```

```
dreamMoves <-
   matrix(c(5,3,17,85), ncol=2,
          dimnames=list("Dreamer"=c("Yes","No"),
                        "Object"=c("Yes","No")))
(dfdream <- plyr::adply(dreamMoves, 1:2,
                        .fun=identity))
```

```
  Dreamer Object V1
1     Yes    Yes  5
2      No    Yes  3
3     Yes     No 17
4      No     No 85
```

To get the table back, do:

```
plyr::daply(dfdream, 1:2, function(df)df[,3])
```

```
       Object
Dreamer Yes No
    Yes   5 17
    No    3 85
```

The following calculates sums over the first two dimensions of the table UCBAdmissions:

```
plyr::aaply(UCBAdmissions, 1:2, sum)
```

Here, aaply() behaves exactly like apply().

```
          Gender
Admit       Male Female
  Admitted 1198    557
  Rejected 1493   1278
```

The following calculates, for each level of the column trt in the data frame nswdemo, the number of values of re74 that are zero:

```
library(DAAG, quietly=TRUE)
plyr::daply(nswdemo, .(trt),
      function(df)sum(df[,"re74"]==0, na.rm=TRUE))
```

```
  0   1
195 131
```

To calculate the proportion that are zero, for each of control and treatment and for each of non-black and black, do:

Notice the use of the syntax `.(trt, black)` to identify the columns `trt` and `black`. This is an alternative to `c("trt", "black")`.

```
options(digits=3)
plyr::daply(nswdemo, .(trt, black),
      function(df)sum(df[,"re75"]==0)/nrow(df))
```

```
   black
trt     0     1
  0 0.353 0.435
  1 0.254 0.403
```

The function `colwise()` takes as argument a function that operates on a column of data, returning a function that operates on all nominated columns of a data frame. To get information on the proportion of zeros for both of the columns `re75` and `re78`, and for each of non-black and black, do:

Here, `colwise()` operates on the objects that are returned by splitting up the data frame `nswdemo` according to levels of `trt` and `black`. Note the use of `ddply()`, not `daply()`.

```
plyr::ddply(nswdemo, .(trt, black),
      colwise(function(x)sum(x==0)/length(x),
              .cols=.(re75, re78)))
```

```
  trt black  re75   re78
1   0     0 0.353 0.1529
2   0     1 0.435 0.3412
3   1     0 0.254 0.0847
4   1     1 0.403 0.2605
```

6.2.2 *Use of* dplyr *with Word War 1 cricketer data*

Data in the data frame `cricketer`, extracted by John Aggleton (now at Univ of Cardiff), are from records of UK first class cricketers born 1840 – 1960. Variables are

- Year of birth
- Years of life (as of 1990)
- 1990 status (dead or alive)
- Cause of death: killed in action / accident / in bed
- Bowling hand – right or left

The following creates a data frame in which the first column has the year, the second the number of right-handers born in that year, and the third the number of left-handers born in that year. .

Both *plyr* and *dplyr* have functions `summarise()`. As in the code shown, detach *plyr* before proceeding. Alternatively, or additionally, specify `dplyr::summarise()` rather than `summarise()`

```
library(DAAG)
detach("package:plyr")
library(dplyr)
```

```
names(cricketer)[1] <- "hand"
gpByYear <- group_by(cricketer, year)
```

```
lefrt <- dplyr::summarise(gpByYear,
                          left=sum(hand=='left'),
                          right=sum(hand=='right'))
## Check first few rows
lefrt[1:4, ]
```

```
# A tibble: 4 x 3
   year  left right
  <int> <int> <int>
1  1840     1     6
2  1841     4    16
3  1842     5    16
4  1843     3    25
```

The data frame is split by values of `year`. Numbers of left and right handers are then tabulated.

From the data frame `cricketer`, we determine the range of birth years for players who died in World War 1. We then extract data for all cricketers, whether dying or surviving until at least the final year of Workd War 1, whose birth year was within this range of years. The following code extracts the relevant range of birth years.

Note that a cricketer who was born in 1869 would be 45 in 1914, while a cricketer who was born in 1896 would be 18 in 1914.

```
## Use subset() from base R
ww1kia <- subset(cricketer,
                 kia==1 & (year+life)%in% 1914:1918)
range(ww1kia$year)
```

```
[1] 1869 1896
```

Alternatively, use `filter()` from *dplyr*:

```
ww1kia <- filter(cricketer,
                 kia==1, (year+life)%in% 1914:1918)
```

For each year of birth between 1869 and 1896, the following expresses the number of cricketers killed in action as a fraction of the total number of cricketers (in action or not) who were born in that year:

```
## Use filter(), group_by() and summarise() from dplyr
crickChoose <- filter(cricketer,
                      year%in%(1869:1896), ((kia==1)|(year+life)>1918))
gpByYearKIA <- group_by(crickChoose, year)
crickKIAyrs <- dplyr::summarise(gpByYearKIA,
                                kia=sum(kia), all=length(year), prop=kia/all)
crickKIAyrs[1:4, ]
```

```
# A tibble: 4 x 4
   year   kia   all   prop
  <int> <int> <int>  <dbl>
1  1869     1    37 0.0270
2  1870     2    36 0.0556
3  1871     1    45 0.0222
4  1872     0    39 0.0000
```

For an introduction to *dplyr*, enter:

```
vignette("introduction", package="dplyr")
```

6.2.3 reshape2: `melt()`, `acast()` & `dcast()`

The *reshape2* package has functions that move between a dataframe layout where selected columns are unstacked, and a layout where they are stacked. In moving from an unstacked to a stacked layout, column names become levels of a factor. In the move back from stacked to unstacked, factor levels become column names.

Here is an example of the use of `melt()`:

```
## Create dataset Crimean, for use in later calculations
library(HistData)   # Nightingale is from this package
library(reshape2)   # Has the function melt()
Crimean <- melt(Nightingale[,c(1,8:10)], "Date")
names(Crimean) <- c("Date", "Cause", "Deaths")
Crimean$Cause <- factor(sub("\\.rate", "", Crimean$Cause))
Crimean$Regime <- ordered(rep(c(rep('Before', 12), rep('After', 12)), 3),
                          levels=c('Before', 'After'))
formdat <- format.Date(sort(unique(Crimean$Date)), format="%d %b %y")
Crimean$Date <- ordered(format.Date(Crimean$Date,
                        format="%b %y"), levels=formdat)
```

The dataset is now in a suitable form for creating a Florence Nightingale style wedge plot, in Figure C.3.

The dataset `Crimean` has been included in the *DAAGviz* package.

Reshaping data for Motion Chart display – an example

The following inputs and displays World Bank Development Indicator data that has been included with the package *DAAGviz*:

```
## DAAGviz must be installed, need not be loaded
path2file <- system.file("datasets/wdiEx.csv", package="DAAGviz")
wdiEx <- read.csv(path2file)
print(wdiEx, row.names=FALSE)
```

```
 Country.Name Country.Code     Indicator.Name Indicator.Code    X2010    X2000
    Australia          AUS Labor force, total SL.TLF.TOTL.IN 1.17e+07 9.62e+06
    Australia          AUS  Population, total    SP.POP.TOTL 2.21e+07 1.92e+07
        China          CHN Labor force, total SL.TLF.TOTL.IN 8.12e+08 7.23e+08
        China          CHN  Population, total    SP.POP.TOTL 1.34e+09 1.26e+09
```

A *googleVis* Motion Chart does not make much sense for this dataset as it stands, with data for just two countries and two years. Motion charts are designed for showing how scatterplot relationships, here between forest area and population, have changed over a number of years. The dataset will however serve for demonstrating the reshaping that is needed.

For input to Motion Charts, we want indicators to be columns, and years to be rows. The `melt()` and `dcast()`[3] functions from the *reshape2* package can be used to achieve the desired result. First, create a single column of data, indexed by classifying factors:

[3] Note also `acast()`, which outputs an array or a matrix.

```
library(reshape2)
wdiLong <- melt(wdiEx, id.vars=c("Country.Code",
                "Indicator.Name"),
                measure.vars=c("X2000", "X2010"))
## More simply: wdiLong <- melt(wdiEx[, -c(2,4)])
wdiLong
```

```
  Country.Code     Indicator.Name variable    value
1          AUS Labor force, total    X2000 9.62e+06
2          AUS  Population, total    X2000 1.92e+07
3          CHN Labor force, total    X2000 7.23e+08
4          CHN  Population, total    X2000 1.26e+09
5          AUS Labor force, total    X2010 1.17e+07
6          AUS  Population, total    X2010 2.21e+07
7          CHN Labor force, total    X2010 8.12e+08
8          CHN  Population, total    X2010 1.34e+09
```

Now use `dcast()` to "cast" the data frame into a form where the indicator variables are columns:

If a matrix or array is required, use `acast()` in place of `dcast()`.

```
names(wdiLong)[3] <- "Year"
wdiData <- dcast(wdiLong,
                 Country.Code+Year ~ Indicator.Name,
                 value.var="value")
wdiData
```

```
  Country.Code  Year Labor force, total Population, total
1          AUS X2000           9.62e+06          1.92e+07
2          AUS X2010           1.17e+07          2.21e+07
3          CHN X2000           7.23e+08          1.26e+09
4          CHN X2010           8.12e+08          1.34e+09
```

A final step is to replace the factor `Year` by a variable that has the values 2000 and 2010.

```
wdiData <- within(wdiData, {
   levels(Year) <- substring(levels(Year),2)
   Year <- as.numeric(as.character(Year))
})
wdiData
```

```
  Country.Code Year Labor force, total Population, total
1          AUS 2000           9.62e+06          1.92e+07
2          AUS 2010           1.17e+07          2.21e+07
3          CHN 2000           7.23e+08          1.26e+09
4          CHN 2010           8.12e+08          1.34e+09
```

6.3 Session and Workspace Management

6.3.1 Keep a record of your work

A recommended procedure is to type commands into an editor window, then sending them across to the command line. This makes it possible to recover work on those hopefully rare occasions when the session aborts.

Be sure to save the script file from time to time during the session, and upon quitting the session.

6.3.2 Workspace management

For tasks that make heavy memory demands, it may be important to ensure that large data objects do not remain in memory once they are no longer needed. There are two complementary strategies:

- Objects that cannot easily be reconstructed or copied from elsewhere, but are not for the time being required, are conveniently saved to an image file, using the `save()` function.
- Use a separate working directory for each major project.

Note the utility function `dir()` (get the names of files, by default in the current working directory).

Several image files ("workspaces") that have distinct names can live in the one working directory. The image file, if any, that is called **.RData** is the file whose contents will be loaded at the beginning of a new session in the directory.

Use `getwd()` to check the name and path of the current working directory. Use `setwd()` to change to a new working directory, while leaving the workspace contents unchanged.

The removal of clutter: Use a command of the form `rm(x, y, tmp)` to remove objects (here `x`, `y`, `tmp`) that are no longer required.

As noted in Section 2.2.2, a good precaution can be to make an archive of the workspace before such removal.

Movement of files between computers: Files that are saved in the default binary save file format, as above, can be moved between different computer systems.

Further possibilities – saving objects in text form: An alternative to saving objects[4] in an image file is to dump them, in a text format, as dump files, e.g.

```
volume <- c(351, 955, 662, 1203, 557, 460)
weight <- c(250, 840, 550, 1360, 640, 420)
dump(c("volume", "weight"), file="books.R")
```

[4] Dumps of S4 objects and environments, amongs others, cannot currently be retrieved using `source()`. See `help(dump)`.

The objects can be recreated [5] from this "dump" file by inputting the lines of **books.R** one by one at the command line. This is what, effectively, the command `source()` does.

```
source("books.R")
```

[5] The same checks are performed on dump files as if the text had been entered at the command line. These can slow down entry of the data or other object. Checks on dependencies can be a problem. These can usually be resolved by editing the R source file to change or remove offending code.

For long-term archival storage, dump (**.R**) files may be preferable to image files. For added security, retain a printed version. If a problem arises (from a system change, or because the file has been corrupted), it is then possible to check through the file line by line to find what is wrong.

6.4 Computer Intensive Computations

Computations may be computer intensive because of the size of datasets. Or the computations may themselves be demanding, even for data sets that are of modest size.

Note that using all of the data for an analysis or for a plot is not always the optimal strategy. Running calculations separately on different subsets may afford insights that are not otherwise available. The subsets may be randomly chosen, or they may be chosen to reflect, e.g., differences in time or place.

Computation will be slow where computationally intensive calculations are implemented directly in R code, rather than passed to

The computationally intensive parts of regression calculations with `lm()` work with matrices, making these relatively efficient.

efficient compiled code that is called from R. Matrix calculations are passed to highly efficient compiled code.

Where it is necessary to look for ways to speed up computations, it is important to profile computations to find which parts of the code are taking the major time. Really big improvements will come from implementing key parts of the calculation in C or Fortran rather than in an application oriented language such as R or Python. Python may do somewhat better than R.

The relatively new Julia language appears to offer spectacular improvements on both R and Python, with times that are within a factor of 2 of the Fortran or C times. See http://julialang.org/.

There can be big differences between the alternatives that may be available in R for handling a calculation. Some broad guidelines will now be provided, with examples of how differences in the handling of calculations can affect timings.

Use matrices, where possible, in preference to data frames: Most of R's modeling functions (regression, smoothing, discriminant analysis, etc.) are designed to work with data frames. Where an alternative available that works with matrices, this will be faster.

Biological expression array applications are among those that are commonly designed to work with data that is in a matrix format. The matrix or matrices may be components of a more complex data structure.

Matrix operations can be more efficient even for such a simple operation as adding a constant quantity to each element of the array, or taking logarithms of all elements. Here is an example:

```
xy <- matrix(rnorm(5*10^7), ncol=100)
dim(xy)
```

```
[1] 500000    100
```

```
system.time(xy+1)
```

```
   user  system elapsed
  0.167   0.143   0.311
```

```
xy.df <- data.frame(xy)
system.time(xy.df+1)
```

```
   user  system elapsed
  0.177   0.139   0.317
```

Timings are on a mid 2012 1.8 Ghz Intel i5 Macbook Air laptop with 8 gigabytes of random access memory.

Use efficient coding: Matrix arithmetic can be faster than the equivalent computations that use `apply()`. Here are timings for some alternatives that find the sums of rows of the matrix `xy` above:

	user	system	elapsed
`apply(xy,1,sum)`	0.528	0.087	0.617
`xy %*% rep(1,100)`	0.019	0.001	0.019
`rowSums(xy)`	0.034	0.001	0.035

The bigmemory project: For details, go to http://www.bigmemory.org/. The *bigmemory* package for R "supports the creation, storage, access, and manipulation of massive matrices". Note also the associated packages *biganalytics*, *bigtabulate*, *synchronicity*, and *bigalgebra*.

The data.table package: This allows the creation of `data.table` objects from which information can be quickly extracted, often in a fraction of the time required for extracting the same information from a data frame. The package has an accompanying vignette. To display it (assuming that the package has been installed), type

```
vignette("datatable-intro", package="data.table")
```

On 64-bit systems, massive data sets, e.g., with tens or hundreds of millions of rows, are possible. For such large data objects, the time saving can be huge.

6.5 Summary

`apply()`, and `sapply()` can be useful for manipulations with data frames and matrices. Note also the functions `melt()`, `dcast()` and `acast()` from the *reshape2* package.

Careful workspace management is important when files are large. It pays to use separate working directories for each different project, and to save important data objects as image files when they are, for the time being, no longer required.

In computations with large datasets, operations that are formally equivalent can differ greatly in their use of computational resources.