4

Data Objects and Functions

Different types of data objects:

Vectors	These collect together elements of the same mode. (Possible modes are "logical", "integer", "numeric", "complex", "character" and "raw")
Factors	Factors identify category levels in categorical data. Modeling functions know how to represent factors. (Factors do not quite manage to be vectors! Why?)
Data frame	A list of columns – same length; modes may differ. Data frames are a device for organizing data.
Lists	Lists group together an arbitrary set of objects (Lists are recursive; elements of lists are lists.)
`NA`s	Use `is.na()` to check for `NA`s.

Data objects and functions are two of several types of objects (others include model objects, formulae, and expressions) that are available in R. Users can create and work with such objects in a user workspace. All can, if the occasion demands, be treated as data!

We start this chapter by noting data objects that may appear as columns of a data frame.

4.1 Column Data Objects – Vectors and Factors

Column objects is a convenient name for one-dimensional data structures that can be included as columns in a data frame. This includes vectors[1], factors, and dates.

[1] Strictly, the vectors that we discuss here are *atomic* vectors. Their elements are not, as happens with lists, wrappers for other language objects.

4.1.1 Vectors

Examples of vectors are

Common vector modes are logical, numeric and character. The 4 lines of code create vectors that are, in order: numeric, numeric, logical, character.

```
c(2,3,5,2,7,1)
3:10 # The numbers 3, 4,.., 10
c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
c("fig","mango","apple","prune")
```

Use `mode()` to show the storage mode of an object, thus:

```
x <- c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
mode(x)
```

```
[1] "logical"
```

The missing value symbol is `NA`. Subsection 4.1.3 will discuss issues that arise when one or more vector elements is an `NA`.

Subsets of Vectors

There are four common ways to extract subsets of vectors.

1. Specify the subscripts of elements that are to be extracted:

```
x <- c(3,11,8,15,12)    # Assign to x the values
x[c(2,4)]               # Extract elements 2 and 4
```

```
[1] 11 15
```

Negative numbers may be used to omit elements:[2]

```
x <- c(3,11,8,15,12)
x[-c(2,3)]
```

```
[1]  3 15 12
```

2. 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   # Values are logical (TRUE or FALSE)
```

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

```
x[x > 10]
```

```
[1] 11 15 12
```

```
"John" %in% c("Jeff", "Alan", "John")
```

```
[1] TRUE
```

3. Where elements have names, these can be used to extract elements:

```
altitude <- c(Cambarville=800, Bellbird=300,
              "Allyn River"=300,
              "Whian Whian"=400,
              Byrangery=200, Conondale=400,
              Bulburin=600)
##
## Names can be used to extract elements
altitude[c("Cambarville", "Bellbird")]
```

```
Cambarville    Bellbird
        800         300
```

4. Use `subset()`, with the vector as the first argument, and a logical statement that identifies the elements to be extracted as the second argument. For example:

```
subset(altitude, altitude>400)
```

```
Cambarville    Bulburin
        800         600
```

[2] Mixing of positive and negative subscripts is not allowed.

Arithmetic relations that may be used for extraction of subsets are >=, ==, != and `%in%`. The first four compare magnitudes, == tests for equality, != tests for inequality, and `%in%` tests whether any element matches.

4.1.2 Factors

Factors are column objects whose elements are integer values 1, 2, ..., *k*, where *k* is the number of levels. They are distinguished from integer vectors by having the class `factor` and a `levels` attribute.

For example, create a character vector `fruit`, thus:

```
fruit <- c("fig","mango","apple","plum", "fig")
```

This might equally well be stored as a factor, thus:

```
fruitfac <- factor(fruit)
```

Internally, the factor is stored as the integer vector 2, 3, 1, 4, 2. These numbers are interpreted according to the attributes table:

1	2	3	4
"apple"	"fig"	"mango"	"plum"

By default, the levels are taken in alphanumeric order.

The function `factor()`, with the `levels` argument specified, can be used both to specify the order of levels when the factor is created, or to make a later change to the order.[3] For example, the following orders levels according to stated glycemic index:

```
glycInd <- c(apple=40, fig=35, mango=55, plum=25)
## Take levels in order of stated glycInd index
fruitfac <- factor(fruit,
                   levels=names(sort(glycInd)))
levels(fruitfac)
```

```
[1] "plum"  "fig"   "apple" "mango"
```

```
unclass(fruitfac)  # Examine stored values
```

```
[1] 2 4 3 1 2
attr(,"levels")
[1] "plum"  "fig"   "apple" "mango"
```

Incorrect spelling of the level names generates missing values, for the level that was mis-spelled. Use the `labels` argument if you wish to change the level names, but be careful to ensure that the label names are in the correct order.

In most places where the context seems to demand it, the integer levels are translated into text strings, thus:

```
fruit <- c("fig","mango","apple", "plum","fig")
fruitfac <- factor(fruit)
fruitfac == "fig"
```

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

Section 8.5 has detailed examples of the use of factors in model formulae.

Factors are an economical way to store vectors of repetitive text strings. By default, when a vector of text strings becomes a column in a data frame, it is incorporated as a factor.

Thus 1 is interpreted as `"apple"`; 2:`"fig"`; 3:`"mango"`; 4:`"plum"`.

[3] Where counts are tabulated by factor level, or *lattice* or other graphs have one panel per factor level, these are in order of the levels.

Mis-spelt name, example:

```
trt <- c("A","A","Control")
trtfac <- factor(trt,
  levels=c("control","A"))
table(trtfac)
```

```
trtfac
control       A
      0       2
```

Ordered factors

In addition to factors, note the existence of ordered factors, created using the function `ordered()`. For ordered factors, the order of levels implies a relational ordering. For example:

```
windowTint <- ordered(rep(c("lo","med","hi"), 2),
                      levels=c("lo","med","hi"))
windowTint
```

```
[1] lo  med hi  lo  med hi
Levels: lo < med < hi
```

```
sum(windowTint > "lo")
```

```
[1] 4
```

Subsetting of factors

Consider the factor `fruitfac` that was created earlier:

```
fruitfac <- factor(c("fig","mango","apple","plum", "fig"))
```

We can remove elements with levels `fig` and `plum` thus:

```
ff2 <- fruitfac[!fruitfac %in% c("fig","plum")]
ff2
```

```
[1] mango apple
Levels: apple fig mango plum
```

```
table(ff2)
```

```
ff2
apple   fig mango  plum
    1     0     1     0
```

The levels `fig` and `plum` remain, but with the table showing 0 values for these levels. Use the function `droplevels()` to remove levels that are not present in the data:

```
droplevels(ff2)
```

```
[1] mango apple
Levels: apple mango
```

Note also:

```
table(droplevels(ff2))
```

```
apple mango
    1     1
```

Why is a factor not a vector?

Two factors that have different levels vectors are different types of object. Thus, formal concatenation of factors with different levels vectors is handled by first coercing both factors to integer vectors. The integer vector that results is not, in most circumstances, meaningful or useful.

Vectors can be concatenated (joined). Two or mare factors can be sensibly concatenated only if they have identical levels vectors.

4.1.3 Missing Values, Infinite Values and NaNs

Any arithmetic or logical operation with `NA` generates an `NA`. The consequences are more far-reaching than might be immediately obvious. Use `is.na()` to test for a missing value:

Failure to understand the rules for calculations with `NA`s can lead to unwelcome surprises.

```
is.na(c(1, NA, 3, 0, NA))
```

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

An expression such as `c(1, NA, 3, 0, NA) == NA` returns a vector of `NA`s, and cannot be used to test for missing values.

```
c(1, NA, 3, 0, NA) == NA
```

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

As the value is unknown, it might or might not be equal to 1, or to another `NA`, or to 3, or to 0.

Note that different functions handle `NA`s in different ways. Functions such as `mean()` and `median()` accept the argument `na.rm=TRUE`, which causes observations that have `NA`s to be ignored. The `plot()` function omits `NA`s, infinities and `NaN`s. For use of `lowess()` to put a smooth curve through the plot, `NA`s must first be removed. By default, `table()` ignores `NA`s.

The modeling function `lm()` accepts any of the arguments `na.action=na.omit` (omit), `na.action=na.exclude` (omit `NA`s when fitting; replace by `NA`s when fitted values and residuals are calculated), and `na.action=na.fail`.

Problems with missing values are a common reason why calculations fail. Infinite values and `NaN`s are a further potential source of difficulty.

`Inf` and `NaN`

The expression `1/0` returns `Inf`. The expression `log(0)` returns `-Inf`, i.e., smaller than any real number. The expressions `0/0` and `log(-1)` both return `NaN`.

Note that `sqrt(-1+0i)` returns `0+1i`. R distinguishes between the real number `-1` and the complex number `-1+0i`.

`NA`s in subscripts?

It is best to ensure that `NA`s do not appear, when there is an assignment, in subscript expressions on either side of the expression.

4.2 Data Frames, Matrices, Arrays and Lists

Data frames: Data frames are lists of column objects. The requirement that all of the column objects have the same length gives data frames a row by column rectangular structure. Different columns can have different column classes — commonly numeric or character or factor or logical or date.

Data frames with all columns numeric can sometimes be handled in the same way as matrices. In other cases, a different syntax may be needed, or conversion from one to the other. Proceed with care!

Matrices – vectors with a Dimension: When printed, matrices appear in a row by column layout in which all elements have the same mode – commonly numeric or character or logical.

Internally, matrices are stored as one long vector in which the columns are stacked one above the other. The first element in the dimension attribute gives the number of rows in each column.

Arrays and tables: Matrices are two-dimensional arrays. Arrays more generally can have an arbitrary number of dimensions. Tables have a structure that is identical to that of arrays.

The data frame `travelbooks` will feature in the subsequent discussion. Look back to Section 1.7 to see how it can be entered.

4.2.1 *Data frames versus matrices and tables*

Modeling functions commonly return larger numeric objects as matrices rather than data frames. The principal components function `prcomp()` returns scores as a matrix, as does the linear discriminant analysis function `lda()` from the *MASS* package.

Computations that can be performed with matrices are typically much faster than their equivalents with data frames. See Section 6.4.

Functions are available to convert data frames into matrices, and vice versa. For example:

```
travelmat <- as.matrix(travelbooks[, 1:4])
  # From data frame to matrix
newtravelbooks <- as.data.frame(travelmat)
  # From matrix to data frame
```

In comparing data frames with matrices, note that:

- Both for data frames and for matrices or two-way tables, the function `dim()` returns number of rows by number of columns, thus:

  ```
  travelmat <- as.matrix(travelbooks[, 1:4])
  dim(travelmat)
  ```

  ```
  [1] 6 4
  ```

Alternatively, do:

```
attr(travelmat, "dim")
```

```
[1] 6 4
```

- For a matrix, `length()` returns the number of elements. For a data frame it returns the number of columns.

  ```
  c(dframelgth=length(travelbooks),
    matlgth=length(travelmat))
  ```

  ```
  dframelgth    matlgth
           6         24
  ```

A data frame is a list of columns. The function `length()` returns the list length.

- The notation that uses single square left and right brackets to extract subsets of data frames, introduced in Section 1.6 works in just the same way with matrices. For example

  ```
  travelmat[, 4]
  travelmat[, "weight"]
  travelmat[, 1:3]
  travelmat[2,]
  ```

Negative indices can be used to omit rows and/or columns.

- Use of the subscript notation to extract a row from a data frame returns a data frame, whereas extraction of a column yields a column vector. Thus:
  - Extraction of a row from a data frame, for example `travelbooks["Canberra - The Guide", ]` or `travelbooks[6, ]`, yields a data frame, i.e., a special form of list.
  - `travelbooks$volume` (equivalent to `travelbooks[,1]` or `travelbooks[,"volume"]`)) is a vector.
- For either a data frame or a matrix, the function `rownames()` can be used to extract row names, and the function `colnames()` to extract column names. For data frames, `row.names()` is an alternative to `rownames()`, while `names()` is an alternative to `colnames()`.

Use `unlist(travelbooks[6, ])` to turn row from the data frame into a vector. All elements are coerced to a common mode, in this case numeric. Thus the final element becomes 1.0 (the code that is stored), rather than `Guide` which was the first level of the factor `type`.

Note also a difference in the mechanisms for adding columns. The following adds new columns `area` (area of page), and `density` (`weight` to `volume` ratio) to the data frame `travelbooks`:

```
travelbooks$area <- with(travelbooks, width*height)
travelbooks$density <- with(travelbooks,
                           weight/volume)
names(travelbooks)    # Check column names
```

```
[1] "thickness" "width"     "height"    "weight"
"volume"    "type"
[7] "area"      "density"
```

Columns are added to the data frame as necessary.

For matrices, use `cbind()`, which can also be used for data frames, to bind in new columns.

4.2.2 Inclusion of character vectors in data frames

When data frames are created, whether by use of `read.table()` or another such function to input data from a file, or by use of the function `data.frame()` to join columns of data together into a data frame, character vectors are converted into factors. Thus, the final column (`type`) of `travelbooks` became, by default, a factor.[4] To prevent such type conversions, specify `stringsAsFactors=FALSE` in the call to `read.table()` or `data.frame()`.

[4] This assumes that the global option `stringsAsFactors` is `FALSE`. To check, interrogate `options()$stringsAsFactors`.

4.2.3 Factor columns in data frame subsets

The data frame `ais` (*DAAG*) has physical charateristics of athletes, divided up thus between ten different sports:

```
with(ais, table(sport))
```

```
sport
 B_Ball   Field     Gym Netball     Row    Swim  T_400m T_Sprnt  Tennis
     25      19       4      23      37      22      29      15      11
 W_Polo
     17
```

Figure 7.2.1 in Subsection 7.9 limits the data to swimmers and rowers. For this, at the same time removing all levels except `Row` and `Swim` from the factor `sport`, do:

```
rowswim <- with(ais, sport %in% c("Row", "Swim"))
aisRS <- droplevels(subset(ais, rowswim))
xtabs(~sport, data=aisRS)
```

If redundant levels were left in place, the graph would show empty panels for each such level.

```
sport
 Row Swim
  37   22
```

Contrast the above with:

```
xtabs(~sport, data=subset(ais, rowswim))
```

```
sport
 B_Ball   Field     Gym Netball     Row    Swim  T_400m T_Sprnt  Tennis
      0       0       0       0      37      22       0       0       0
 W_Polo
      0
```

4.2.4 *Handlng rows that include missing values*

The function `na.omit()` omits rows that contain one or more missing values. The argument may be a data frame or a matrix. The function `complete.cases()` identifies such rows. Thus:

```
test.df <- data.frame(x=c(1:2,NA), y=1:3)
test.df
```

```
   x y
1  1 1
2  2 2
3 NA 3
```

```
## complete.cases()
complete.cases(test.df)
```

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

```
## na.omit()
na.omit(test.df)
```

```
  x y
1 1 1
2 2 2
```

4.2.5 Arrays — some further details

A matrix is a two-dimensional array. More generally, arrays can have an arbitrary number of dimensions.

Tables, which will be the subject of the next subsection, have a very similar structure to arrays.

Removal of the dimension attribute

The dimension attribute of a matrix or array can be changed or removed, thus:

```
travelvec <-  as.matrix(travelbooks[, 1:4])
dim(travelvec) <- NULL  # Columns of travelmat are stacked into one
                        # long vector
travelvec
```

```
 [1]    1.3    3.9    1.2    2.0    0.6    1.5   11.3   13.1   20.0   21.1
[11]   25.8   13.1   23.9   18.7   27.6   28.5   36.0   23.4  250.0  840.0
[21]  550.0 1360.0  640.0  420.0
```

```
  # as(travelmat, "vector") is however preferable
```

Note again that the $ notation, used with data frames and other list objects to reference the contents of list elements, is not relevant to matrices.

4.2.6 Lists

A list is a collection of arbitrary objects. As noted above, a data frame is a specialized form of list. Consider for example the list

Elements of lists are themselves lists. Distinguish `rcanberra[4]`, which is a sub-list and therefore a list, from `rcanberra[[4]]` which extracts the contents of the fourth list element.

```
rCBR <- list(society="ssai", branch="Canberra",
             presenter="John",
             tutors=c("Emma", "Chris", "Frank"))
```

First, extract list length and list names:

```
length(rCBR)       # rCBR has 4 elements
names(rCBR)
```

```
[1] 4
```

```
[1] "society"   "branch"    "presenter" "tutors"
```

The following extracts the 4th list element:

```
rCBR[4]           # Also a list,  name is 'tutors'
```

```
$tutors
[1] "Emma"  "Chris" "Frank"
```

Alternative ways to extract the contents of the 4^{th} element are:

List elements can be accessed by name. Thus, to extract the contents of the 4th list element, alternatives to `rcanberra[[4]]` are `rcanberra[["tutors"]]` or `rcanberra$tutors`.

```
rCBR[[4]]         # Contents of 4th list element
```

```
[1] "Emma"  "Chris" "Frank"
```

```
rCBR$tutors       # Equivalent to rCBR[["tutors"]]
```

```
[1] "Emma"  "Chris" "Frank"
```

Model objects are lists

As noted in Subsection 3.4.2, the various R modeling functions all return their own particular type of model object, either a list or as an S4 object.

Recall again, also, that data frames are a specialized form of list, with the restriction that all columns must all have the same length.

4.3 Functions

Different Kinds of Functions:

Generic	The 'class' of the function argument determines the action taken. E.g., `print()`, `plot()`, `summary()`
Modeling	For example, `lm()` fits *linear* models. Output may be stored in a model object.
Extractor	These extract information from model objects. Examples include `summary()`), `coef()`), `resid()`), and `fitted()`
User	Use, e.g., to automate and document computations
Anonymous	These are user functions that are defined at the point of use, and do not need a name.

The above list is intended to include the some of the most important types of function. These categories may overlap.

The language that R implements has many of the features of a functional language. Functions have accordingly featured throughout the earlier discussion. Here will be noted functions that are commonly important.

Functions for working with dates are discussed in Section 4.3.9 immediately following.

4.3.1 Built-In Functions

Common useful functions

```
## Use with any R object as argument
print()             # Prints a single R object
length()            # Number of elements in a vector or of a list

## Concatenate and print R objects [does less coercion than print()]
cat()               # Prints multiple objects, one after the other

## Use with a numeric vector argument
mean()              # If argument has NA elements, may want na.rm=TRUE
median()            # As for mean(), may want na.rm=TRUE
range()             # As for mean(), may want na.rm=TRUE
unique()            # Gives the vector of distinct values
diff()              # Vector of first differences
                    # N. B. diff(x) has one less element than x
cumsum()            # Cumulative sums, c.f., also, cumprod()

## Use with an atomic vector object
sort()              # Sort elements into order, but omitting NAs
order()             # x[order(x)] orders elements of x, with NAs last
rev()               # reverse the order of vector elements
any()               # Returns TRUE if there are any missing values
as()                # Coerce argument 1 to class given by argument 2
                    # e.g. as(1:6, "factor")
is()                # Is argument 1 of class given by argument 2?
                    # is(1:6, "factor") returns FALSE
                    # 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

For data manipulation, note:

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

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.

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 `NA`s, 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
0.00408 0.95165
```

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

```
 mean    sd
0.383 1.294
```

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.

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

Here is an example:

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

Code that will input molclock1:

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

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:

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

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

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:

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

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

```
Time difference of 335 days
```

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

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

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

```
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
```

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"))))
```

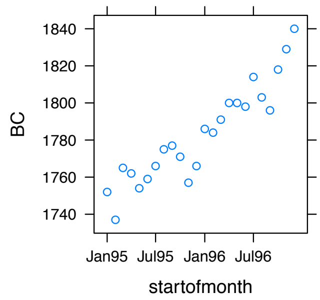

Figure 4.1: Canadian worker force numbers, with dates used to label the x-axis. See Figure 7.12 in Subsection 7.2.6 for data from all Canadian provinces.

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 NAs, 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 for a data frame, `print()` calls the method `print.data.frame()`.

For a factor, `print()` it calls `print.factor()`, and so on. Ordered factors 'inherit' the print method for factors. For objects with no 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

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.

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

Example – a spatial class

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

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

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.

[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.

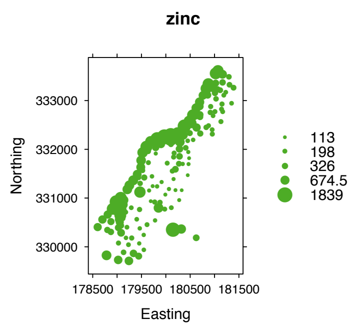

Figure 4.2: Bubble plot for `zinc` concentrations. Areas of bubbles are proportional to concentrations.

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 11.4 has further details.

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

. In base R, and in R packages, there is a wide variety of functions that can be used for data input. This includes data entry abilities that are aimed at specific specialized types of data.

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

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. Refer back to Subsection 2.4.3 for further details. Note that input 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.

5.1.1 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.

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.

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

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

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).

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

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.

5.1.2 *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.

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.

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.

5.1.3 The `memisc` package: input from SPSS and Stata

The *memisc* package has effective abilities for examining and inputting data from various SPSS and Stata formats, including **.sav**, **.por**, and Stata **.dta** data types. It allows users 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 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//RtmpZDkSBa/file11f21a
        with 67 variables and 662 observations
```

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

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.

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

```
## 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:

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

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

```
======================================================

   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.

[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.

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

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.

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':

[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).

```
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

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.

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.

```
## 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.CO2.OTHX.ZS"
[2,] "EN.CO2.MANF.ZS"
[3,] "EN.CO2.ETOT.ZS"
[4,] "EN.CO2.BLDG.ZS"
     name
[1,] "CO2 emissions from other sectors, excluding residential buildings and commercial and public se
[2,] "CO2 emissions from manufacturing industries and construction (% of total fuel combustion)"
[3,] "CO2 emissions from electricity and heat production, total (% of total fuel combustion)"
[4,] "CO2 emissions from residential buildings and commercial and public services (% of total fuel c
```

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

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:

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

```
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:

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

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

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.