2

Essentials of the R Language

There is an enormous range of things that R can do, and one of the hardest parts of learning R is finding your way around. I suggest that you start by looking down all the chapter names at the front of this book (p. v) to get an overall idea of where the material you want might be found. The Index for this book has the names of R functions in **bold**. The dataframes used in the book are listed by name, and also under 'dataframes'.

Alternatively, if you know the name of the function you are interested in, then you can go directly to R's help by typing the exact name of the function after a question mark on the command line (p. 2).

Screen prompt

The screen prompt > is an invitation to put R to work. You can either use one of the many facilities which are the subject of this book, including the built-in functions discussed on p. 11, or do ordinary calculations:

```
> log(42/7.3)
```

```
[1] 1.749795
```

Each line can have at most 128 characters, so if you want to give a lengthy instruction or evaluate a complicated expression, you can continue it on one or more further lines simply by ending the line at a place where the line is obviously incomplete (e.g. with a trailing comma, operator, or with more left parentheses than right parentheses, implying that more right parentheses will follow). When continuation is expected, the prompt changes from > to +

```
> 5+6+3+6+4+2+4+8+
+       3+2+7
```

```
[1] 50
```

Note that the + continuation prompt does not carry out arithmetic plus. If you have made a mistake, and you want to get rid of the + prompt and return to the > prompt, then either press the Esc key or use the Up arrow to edit the last (incomplete) line.

Two or more expressions can be placed on a single line so long as they are separated by semi-colons:

```
2+3; 5*7; 3-7
```

```
[1] 5
[1] 35
[1] -4
```

From here onwards and throughout the book, the prompt character > will be omitted. The material that you should type on the command line is shown in Arial font. Just press the Return key to see the answer. The output from R is shown in Courier New font, which uses absolute rather than proportional spacing, so that columns of numbers remain neatly aligned on the page or on the screen.

Built-in Functions

All the mathematical functions you could ever want are here (see Table 2.1). The log function gives logs to the base e (e = 2.718282), for which the antilog function is exp

```
log(10)
```

```
[1] 2.302585
```

```
exp(1)
```

```
[1] 2.718282
```

If you are old fashioned, and want logs to the base 10, then there is a separate function

```
log10(6)
```

```
[1] 0.7781513
```

Logs to other bases are possible by providing the log function with a second argument which is the base of the logs you want to take. Suppose you want log to base 3 of 9:

```
log(9,3)
```

```
[1] 2
```

The trigonometric functions in R measure angles in radians. A circle is 2π radians, and this is 360°, so a right angle (90°) is $\pi/2$ radians. R knows the value of π as pi:

```
pi
```

```
[1] 3.141593
```

```
sin(pi/2)
```

```
[1] 1
```

```
cos(pi/2)
```

```
[1] 6.123032e-017
```

Notice that the cosine of a right angle does not come out as exactly zero, even though the sine came out as exactly 1. The e-017 means 'times 10^{-17}'. While this is a very small

Table 2.1. Mathematical functions used in R.

Function	Meaning
log(x)	log to base e of x
exp(x)	antilog of x (e^x)
log(x,n)	log to base n of x
log10(x)	log to base 10 of x
sqrt(x)	square root of x
factorial(x)	$x!$
choose(n,x)	binomial coefficients $n!/(x!\ (n-x)!)$
gamma(x)	$\Gamma(x)$, for real x $(x-1)!$, for integer x
lgamma(x)	natural log of $\Gamma(x)$
floor(x)	greatest integer $< x$
ceiling(x)	smallest integer $> x$
trunc(x)	closest integer to x between x and 0 trunc(1.5) $= 1$, trunc(-1.5) $= -1$ trunc is like floor for positive values and like ceiling for negative values
round(x, digits=0)	round the value of x to an integer
signif(x, digits=6)	give x to 6 digits in scientific notation
runif(n)	generates n random numbers between 0 and 1 from a uniform distribution
cos(x)	cosine of x in radians
sin(x)	sine of x in radians
tan(x)	tangent of x in radians
acos(x), asin(x), atan(x)	inverse trigonometric transformations of real or complex numbers
acosh(x), asinh(x), atanh(x)	inverse hyperbolic trigonometric transformations of real or complex numbers
abs(x)	the absolute value of x, ignoring the minus sign if there is one

number it is clearly not exactly zero (so you need to be careful when testing for exact equality of real numbers; see p. 77).

Numbers with Exponents

For very big numbers or very small numbers R uses the following scheme:

1.2e3 means 1200 because the e3 means 'move the decimal point 3 places to the right'

1.2e-2 means 0.012 because the e-2 means 'move the decimal point 2 places to the left'

3.9+4.5i is a complex number with real (3.9) and imaginary (4.5) parts, and i is the square root of -1.

Modulo and Integer Quotients

Integer quotients and remainders are obtained using the notation %/% (percent, divide, percent) and %% (percent, percent) respectively. Suppose we want to know the integer part of a division: say, how many 13s are there in 119:

```
119 %/% 13
```

```
[1] 9
```

Now suppose we wanted to know the remainder (what is left over when 119 is divided by 13): in maths this is known as **modulo**:

```
119 %% 13
```

```
[1] 2
```

Modulo is very useful for testing whether numbers are odd or even: odd numbers have modulo 2 value 1 and even numbers have modulo 2 value 0:

```
9 %% 2
```

```
[1] 1
```

```
8 %% 2
```

```
[1] 0
```

Likewise, you use modulo to test if one number is an exact multiple of some other number. For instance to find out whether 15 421 is a multiple of 7, ask:

```
15421 %% 7 == 0
```

```
[1] TRUE
```

Rounding

Various sorts of rounding (rounding up, rounding down, rounding to the nearest integer) can be done easily. Take 5.7 as an example. The 'greatest integer less than' function is floor

```
floor(5.7)
```

```
[1] 5
```

and the 'next integer' function is ceiling

```
ceiling(5.7)
```

```
[1] 6
```

You can round to the nearest integer by adding 0.5 to the number then using floor. There is a built-in function for this, but we can easily write one of our own to introduce the notion of function writing. Call it rounded, then define it as a function like this:

```
rounded<-function(x) floor(x+0.5)
```

Now we can use the new function:

```
rounded(5.7)
```

```
[1] 6
```

```
rounded(5.4)
```

```
[1] 5
```

Infinity and Things that Are Not a Number (NaN)

Calculations can lead to answers that are plus infinity, represented in R by `Inf`, or minus infinity, which is represented as `-Inf`:

```
3/0
```

```
[1] Inf
```

```
-12/0
```

```
[1] -Inf
```

Calculations involving infinity can be evaluated: for instance,

```
exp(-Inf)
```

```
[1] 0
```

```
0/Inf
```

```
[1] 0
```

```
(0:3)^Inf
```

```
[1] 0 1 Inf Inf
```

Other calculations, however, lead to quantities that are not numbers. These are represented in R by `NaN` ('not a number'). Here are some of the classic cases:

```
0/0
```

```
[1] NaN
```

```
Inf-Inf
```

```
[1] NaN
```

```
Inf/Inf
```

```
[1] NaN
```

You need to understand clearly the distinction between `NaN` and `NA` (this stands for 'not available' and is the missing-value symbol in R; see below). The function `is.nan` is provided to check specifically for `NaN`, and `is.na` also returns `TRUE` for `NaN`. Coercing `NaN` to logical or integer type gives an `NA` of the appropriate type. There are built-in tests to check whether a number is finite or infinite:

```
is.finite(10)
```

```
[1] TRUE
```

```
is.infinite(10)
```

```
[1] FALSE
```

```
is.infinite(Inf)
```

```
[1] TRUE
```

Missing values NA

Missing values in dataframes are a real source of irritation because they affect the way that model-fitting functions operate and they can greatly reduce the power of the modelling that we would like to do.

Some functions do not work with their default settings when there are missing values in the data, and mean is a classic example of this:

```
x<-c(1:8,NA)
mean(x)
```

```
[1] NA
```

In order to calculate the mean of the non-missing values, you need to specify that the NA are to be removed, using the na.rm=TRUE argument:

```
mean(x,na.rm=T)
```

```
[1] 4.5
```

To check for the location of missing values within a vector, use the function is.na(x) rather than x !="NA". Here is an example where we want to find the locations (7 and 8) of missing values within a vector called vmv:

```
vmv<-c(1:6,NA,NA,9:12)
vmv
```

```
[1]  1  2  3  4  5  6 NA NA  9 10 11 12
```

Making an index of the missing values in an array could use the seq function,

```
seq(along=vmv)[is.na(vmv)]
```

```
[1] 7 8
```

but the result is achieved more simply using which like this:

```
which(is.na(vmv))
```

```
[1] 7 8
```

If the missing values are genuine counts of zero, you might want to edit the NA to 0. Use the is.na function to generate subscripts for this

```
vmv[is.na(vmv)]<- 0
vmv
```

```
[1]  1  2  3  4  5  6  0  0  9 10 11 12
```

or use the ifelse function like this

```
vmv<-c(1:6,NA,NA,9:12)
ifelse(is.na(vmv),0,vmv)
```

```
[1]  1  2  3  4  5  6  0  0  9 10 11 12
```

Assignment

Objects obtain values in R by assignment (‘*x gets* a value’). This is achieved by the **gets arrow** <- which is a composite symbol made up from ‘less than’ and ‘minus’ with no space between them. Thus, to create a scalar constant *x* with value 5 we type

```
x<-5
```

and not `x = 5`. Notice that there is a potential ambiguity if you get the spacing wrong. Compare our `x<-5`, ‘*x* gets 5’, with `x < -5` which is a logical question, asking ‘is *x* less than minus 5?’ and producing the answer `TRUE` or `FALSE`.

Operators

R uses the following operator tokens:

+ - */%% ^	arithmetic
> >= < <= == !=	relational
! & |	logical
~	model formulae
<- ->	assignment
$	list indexing (the ‘element name’ operator)
:	create a sequence

Several of the operators have different meaning inside model formulae. Thus * indicates the main effects plus interaction, : indicates the interaction between two variables and ^ means all interactions up to the indicated power (see p. 332).

Creating a Vector

Vectors are variables with one or more values of the same type: logical, integer, real, complex, string (or character) or raw. A variable with a single value (say 4.3) is often known as a scalar, but in R a scalar is a vector of length 1. Vectors could have length 0 and this can be useful in writing functions:

```
y<-4.3
z<-y[-1]
length(z)
```

```
[1] 0
```

Values can be assigned to vectors in many different ways. They can be generated by R: here the vector called *y* gets the sequence of integer values 10 to 16 using : (colon), the sequence-generating operator,

```
y <- 10:16
```

You can type the values into the command line, using the concatenation function c,

```
y <- c(10, 11, 12, 13, 14, 15, 16)
```

or you can enter the numbers from the keyboard one at a time using scan:

```
y <- scan()
1: 10
2: 11
3: 12
4: 13
5: 14
6: 15
7: 16
8:
Read 7 items
```

pressing the Enter key instead of entering a number to indicate that data input is complete. However, the commonest way to allocate values to a vector is to read the data from an external file, using read.table (p. 98). Note that read.table will convert character variables into factors, so if you do *not* want this to happen, you will need to say so (p. 100). In order to refer to a vector by name with an R session, you need to attach the dataframe containing the vector (p. 18). Alternatively, you can refer to the dataframe name and the vector name within it, using the element name operator $ like this: dataframe$y

One of the most important attributes of a vector is its length: the number of numbers it contains. When vectors are created by calculation from other vectors, the new vector will be as long as the longest vector used in the calculation (with the shorter variable(s) recycled as necessary): here A is of length 10 and B is of length 3:

```
A<-1:10
B<-c(2,4,8)
A*B
[1]  2  8  24  8  20  48  14  32  72  20
Warning message: longer object length is not a multiple of shorter object
length in: A * B
```

The vector B is recycled three times in full and a warning message in printed to indicate that the length of A is not a multiple of the length of B (11×4 and 12×8 have not been evaluated).

Named Elements within Vectors

It is often useful to have the values in a vector labelled in some way. For instance, if our data are counts of 0, 1, 2, . . . occurrences in a vector called counts

```
(counts<-c(25,12,7,4,6,2,1,0,2))
[1]  25  12  7  4  6  2  1  0  2
```

so that there were 25 zeros, 12 ones and so on, it would be useful to name each of the counts with the relevant number 0 to 8:

```
names(counts)<-0:8
```

Now when we inspect the vector called counts we see both the names and the frequencies:

```
counts
```

```
 0  1  2  3  4  5  6  7  8
25 12  7  4  6  2  1  0  2
```

If you have computed a table of counts, and you want to *remove* the names, then use the as.vector function like this:

```
(st<-table(rpois(2000,2.3)))
```

```
  0   1   2   3   4   5   6   7 8 9
205 455 510 431 233 102 43 13 7 1
```

```
as.vector(st)
```

```
[1]  205  455  510  431  233  102  43  13  7  1
```

Vector Functions

One of R's great strengths is its ability to evaluate functions over entire vectors, thereby avoiding the need for loops and subscripts. Important vector functions are listed in Table 2.2.

Table 2.2. Vector functions used in R.

Operation	Meaning
max(x)	maximum value in x
min(x)	minimum value in x
sum(x)	total of all the values in x
mean(x)	arithmetic average of the values in x
median(x)	median value in x
range(x)	vector of min(x) and max(x)
var(x)	sample variance of x
cor(x,y)	correlation between vectors x and y
sort(x)	a sorted version of x
rank(x)	vector of the ranks of the values in x
order(x)	an integer vector containing the permutation to sort x into ascending order
quantile(x)	vector containing the minimum, lower quartile, median, upper quartile, and maximum of x
cumsum(x)	vector containing the sum of all of the elements up to that point
cumprod(x)	vector containing the product of all of the elements up to that point
cummax(x)	vector of non-decreasing numbers which are the cumulative maxima of the values in x up to that point
cummin(x)	vector of non-increasing numbers which are the cumulative minima of the values in x up to that point
pmax(x,y,z)	vector, of length equal to the longest of x, y or z, containing the maximum of x, y or z for the ith position in each

Table 2.2. (Continued)

Operation	Meaning
pmin(x,y,z)	vector, of length equal to the longest of x, y or z, containing the minimum of x, y or z for the ith position in each
colMeans(x)	column means of dataframe or matrix x
colSums(x)	column totals of dataframe or matrix x
rowMeans(x)	row means of dataframe or matrix x
rowSums(x)	row totals of dataframe or matrix x

Summary Information from Vectors by Groups

One of the most important and useful vector functions to master is tapply. The 't' stands for 'table' and the idea is to apply a function to produce a table from the values in the vector, based on one or more grouping variables (often the grouping is by factor levels). This sounds much more complicated than it really is:

```
data<-read.table("c:\\temp\\daphnia.txt",header=T)
attach(data)
names(data)
```

```
[1] "Growth.rate" "Water" "Detergent" "Daphnia"
```

The response variable is Growth.rate and the other three variables are factors (the analysis is on p. 479). Suppose we want the mean growth rate for each detergent:

```
tapply(Growth.rate,Detergent,mean)
```

```
BrandA  BrandB  BrandC  BrandD
  3.88    4.01    3.95    3.56
```

This produces a table with four entries, one for each level of the factor called Detergent. To produce a two-dimensional table we put the two grouping variables in a list. Here we calculate the median growth rate for water type and daphnia clone:

```
tapply(Growth.rate,list(Water,Daphnia),median)
```

```
      Clone1  Clone2  Clone3
Tyne    2.87    3.91    4.62
Wear    2.59    5.53    4.30
```

The first variable in the list creates the rows of the table and the second the columns. More detail on the tapply function is given in Chapter 6 (p. 183).

Using with rather than attach

Advanced R users do not routinely employ attach in their work, because it can lead to unexpected problems in resolving names (e.g. you can end up with multiple copies of the same variable name, each of a different length and each meaning something completely different). Most modelling functions like lm or glm have a data= argument so attach is unnecessary in those cases. Even when there is no data= argument it is preferable to wrap the call using with like this

```
with(data, function(...))
```

The with function evaluates an R expression in an environment constructed from data. You will often use the with function other functions like tapply or plot which have no built-in data argument. If your dataframe is part of the built-in package called datasets (like OrchardSprays) you can refer to the dataframe directly by name:

```
with(OrchardSprays,boxplot(decrease~treatment))
```

Here we calculate the number of 'no' (not infected) cases in the bacteria dataframe which is part of the MASS library:

```
library(MASS)
with(bacteria,tapply((y=="n"),trt,sum))
```

```
placebo  drug  drug+
     12    18     13
```

and here we plot brain weight against body weight for mammals on log-log axes:

```
with(mammals,plot(body,brain,log="xy"))
```

without attaching either dataframe. Here is an unattached dataframe called reg.data:

```
reg.data<-read.table("c:\\temp\\regression.txt",header=T)
```

with which we carry out a linear regression and print a summary

```
with (reg.data, {
                model<-lm(growth~tannin)
                summary(model) } )
```

The linear model fitting function lm knows to look in reg.data to find the variables called growth and tannin because the with function has used reg.data for constructing the environment from which lm is called. Groups of statements (different lines of code) to which the with function applies are contained within curly brackets. An alternative is to define the data environment as an argument in the call to lm like this:

```
summary(lm(growth~tannin,data=reg.data))
```

You should compare these outputs with the same example using attach on p. 388. Note that whatever form you choose, you still need to get the dataframe into your current environment by using read.table (if, as here, it is to be read from an external file), or from a library (like MASS to get bacteria and mammals, as above). To see the names of the dataframes in the built-in package called datasets, type

```
data()
```

but to see all available data sets (including those in the installed packages), type

```
data(package = .packages(all.available = TRUE))
```

Using attach in This Book

I use attach throughout this book because experience has shown that it makes the code easier to understand for beginners. In particular, using attach provides simplicity and brevity, so that we can

- refer to variables by name, so x rather than dataframe$x
- write shorter models, so lm(y~x) rather than lm(y~x,data=dataframe)
- go straight to the intended action, so plot(y~x) not with(dataframe,plot(y~x))

Nevertheless, readers are encouraged to use with or data= for their own work, and to avoid using attach wherever possible.

Parallel Minima and Maxima: pmin and pmax

Here are three vectors of the same length, x, y and z. The parallel minimum function, pmin, finds the minimum from any one of the three variables for each subscript, and produces a *vector* as its result (of length equal to the longest of x, y, or z):

```
x
[1] 0.99822644 0.98204599 0.20206455 0.65995552 0.93456667 0.18836278
y
[1] 0.51827913 0.30125005 0.41676059 0.53641449 0.07878714 0.49959328
z
[1] 0.26591817 0.13271847 0.44062782 0.65120395 0.03183403 0.36938092
pmin(x,y,z)
[1] 0.26591817 0.13271847 0.20206455 0.53641449 0.03183403 0.18836278
```

so the first and second minima came from z, the third from x, the fourth from y, the fifth from z, and the sixth from x. The functions min and max *produce scalar* results.

Subscripts and Indices

While we typically aim to apply functions to vectors as a whole, there are circumstances where we want to select only some of the elements of a vector. This selection is done using **subscripts** (also known as **indices**). Subscripts have square brackets [2] while functions have round brackets (2). Subscripts on vectors, matrices, arrays and dataframes have one set of square brackets [6], [3,4] or [2,3,2,1] while subscripts on lists have double square brackets [[2]] or [[i,j]] (see p. 65). When there are two subscripts to an object like a matrix or a dataframe, the first subscript refers to the row number (the rows are defined as **margin** no. 1) and the second subscript refers to the column number (the columns are margin no. 2). There is an important and powerful convention in R such that *when a subscript appears as a blank it is understood to mean 'all of'*. Thus

- [,4] means all rows in column 4 of an object
- [2,] means all columns in row 2 of an object.

There is another indexing convention in R which is used to extract named components from objects using the $ operator like this: model$coef or model$resid (p. 363). This is known as 'indexing tagged lists' using the element names operator $.

Working with Vectors and Logical Subscripts

Take the example of a vector containing the 11 numbers 0 to 10:

```
x<-0:10
```

There are two quite different kinds of things we might want to do with this. We might want to *add up* the values of the elements:

```
sum(x)
```

```
[1] 55
```

Alternatively, we might want to *count* the elements that passed some logical criterion. Suppose we wanted to know how many of the values were less than 5:

```
sum(x<5)
```

```
[1] 5
```

You see the distinction. We use the vector function sum in both cases. But sum(x) adds up the values of the *x*s and sum(x<5) counts up the number of cases that pass the logical condition '*x* is less than 5'. This works because of *coercion* (p. 25). Logical TRUE has been coerced to numeric 1 and logical FALSE has been coerced to numeric 0.

That is all well and good, but how do you add up the values of just some of the elements of *x*? We specify a logical condition, but we don't want to count the number of cases that pass the condition, we want to add up all the values of the cases that pass. This is the final piece of the jigsaw, and involves the use of *logical subscripts*. Note that when we counted the number of cases, the counting was applied to the entire vector, using sum(x<5). To find the sum of the values of *x* that are less than 5, we write:

```
sum(x[x<5])
```

```
[1] 10
```

Let's look at this in more detail. The logical condition x<5 is either true or false:

```
x<5
```

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

You can imagine false as being numeric 0 and true as being numeric 1. Then the vector of subscripts [x<5] is five 1s followed by six 0s:

```
1*(x<5)
```

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

Now imagine multiplying the values of x by the values of the logical vector

```
x*(x<5)
```

```
[1]  0  1  2  3  4  0  0  0  0  0  0
```

When the function sum is applied, it gives us the answer we want: the sum of the values of the numbers $0+1+2+3+4=10$.

```
sum(x*(x<5))
```

```
[1]  10
```

This produces the same answer as sum(x[x<5]), but is rather less elegant.

Suppose we want to work out the sum of the three largest values in a vector. There are two steps: first sort the vector into descending order. Then add up the values of the first three elements of the sorted array. Let's do this in stages. First, the values of y:

```
y<-c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
```

Now if you apply sort to this, the numbers will be in ascending sequence, and this makes life slightly harder for the present problem:

```
sort(y)
```

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

We can use the reverse function, rev like this (use the Up arrow key to save typing):

```
rev(sort(y))
```

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

So the answer to our problem is $11+10+9=30$. But how to compute this? We can use specific subscripts to discover the contents of any element of a vector. We can see that 10 is the second element of the sorted array. To compute this we just specify the subscript [2]:

```
rev(sort(y))[2]
```

```
[1]  10
```

A range of subscripts is simply a series generated using the colon operator. We want the subscripts 1 to 3, so this is:

```
rev(sort(y))[1:3]
```

```
[1]  11  10  9
```

So the answer to the exercise is just

```
sum(rev(sort(y))[1:3])
```

```
[1]  30
```

Note that we have not changed the vector y in any way, nor have we created any new space-consuming vectors during intermediate computational steps.

Addresses within Vectors

There are two important functions for finding addresses within arrays. The function which is very easy to understand. The vector *y* (see above) looks like this:

```
y
```

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

Suppose we wanted to know which elements of *y* contained values bigger than 5. We type

```
which(y>5)
```

```
[1]  1  4  5  6  7 8 11  13 15
```

Notice that the answer to this enquiry is *a set of subscripts*. We don't use subscripts inside the which function itself. The function is applied to the whole array. To see the values of *y* that are larger than 5, we just type

```
y[y>5]
```

```
[1]  8  7  6  6  8  9  9  10  11
```

Note that this is a shorter vector than *y* itself, because values of 5 or less have been left out:

```
length(y)
```

```
[1]  15
```

```
length(y[y>5])
```

```
[1]  9
```

To extract every *n*th element from a long vector we can use seq as an index. In this case I want every 25th value in a 1000-long vector of normal random numbers with mean value 100 and standard deviation 10:

```
xv<-rnorm(1000,100,10)
xv[seq(25,length(xv),25)]
```

```
 [1] 100.98176  91.69614 116.69185  97.89538 108.48568 100.32891  94.46233
 [8] 118.05943  92.41213 100.01887 112.41775 106.14260  93.79951 105.74173
[15] 102.84938  88.56408 114.52787  87.64789 112.71475 106.89868 109.80862
[22]  93.20438  96.31240  85.96460 105.77331  97.54514  92.01761  97.78516
[29]  87.90883  96.72253  94.86647  90.87149  80.01337  97.98327  92.77398
[36] 121.47810  92.40182  87.65205 115.80945  87.60231
```

Finding Closest Values

Finding the value in a vector that is closest to a specified value is straightforward using which. Here, we want to find the value of *xv* that is closest to 108.0:

```
which(abs(xv-108)==min(abs(xv-108)))
```

```
[1]  332
```

The closest value to 108.0 is in location 332. But just how close to 108.0 is this 332nd value? We use 332 as a subscript on xv to find this out

```
xv[332]
```

```
[1] 108.0076
```

Thus, we can write a function to return the closest value to a specified value (*sv*)

```
closest<-function(xv,sv){
xv[which(abs(xv-sv)==min(abs(xv-sv)))] }
```

and run it like this:

```
closest(xv,108)
```

```
[1] 108.0076
```

Trimming Vectors Using Negative Subscripts

Individual subscripts are referred to in square brackets. So if x is like this:

```
x<- c(5,8,6,7,1,5,3)
```

we can find the 4th element of the vector just by typing

```
x[4]
```

```
[1] 7
```

An extremely useful facility is to use negative subscripts to drop terms from a vector. Suppose we wanted a new vector, z, to contain everything but the first element of x

```
z <- x[-1]
z
```

```
[1] 8 6 7 1 5 3
```

Suppose our task is to calculate a trimmed mean of x which ignores both the smallest and largest values (i.e. we want to leave out the 1 and the 8 in this example). There are two steps to this. First, we sort the vector x. Then we remove the first element using x[-1] and the last using x[-length(x)]. We can do both drops at the same time by concatenating both instructions like this: -c(1,length(x)). Then we use the built-in function mean:

```
trim.mean <- function (x) mean(sort(x)[-c(1,length(x))])
```

Now try it out. The answer should be mean(c(5,6,7,5,3)) $= 26/5 = 5.2$:

```
trim.mean(x)
```

```
[1] 5.2
```

Suppose now that we need to produce a vector containing the numbers 1 to 50 but omitting all the multiples of seven (7, 14, 21, etc.). First make a vector of all the numbers 1 to 50 including the multiples of 7:

```
vec<-1:50
```

Now work out how many numbers there are between 1 and 50 that are multiples of 7

```
(multiples<-floor(50/7))
```

```
[1] 7
```

Now create a vector of the first seven multiples of 7 called *subscripts*:

```
(subscripts<-7*(1:multiples))
```

```
[1]  7 14 21 28 35 42 49
```

Finally, use negative subscripts to drop these multiples of 7 from the original vector

```
vec[-subscripts]
```

```
 [1]  1  2  3  4  5  6  8  9 10 11 12 13 15 16 17 18 19 20 22 23 24 25
[23] 26 27 29 30 31 32 33 34 36 37 38 39 40 41 43 44 45 46 47 48 50
```

Alternatively, you could use modulo seven %%7 to get the result in a single line:

```
vec[-(1:50*(1:50%%7==0))]
```

```
 [1]  1  2  3  4  5  6  8  9 10 11 12 13 15 16 17 18 19 20 22 23 24 25
[23] 26 27 29 30 31 32 33 34 36 37 38 39 40 41 43 44 45 46 47 48 50
```

Logical Arithmetic

Arithmetic involving logical expressions is very useful in programming and in selection of variables. If logical arithmetic is unfamiliar to you, then persevere with it, because it will become clear how useful it is, once the penny has dropped. The key thing to understand is that logical expressions evaluate to either true or false (represented in R by TRUE or FALSE), and that R can coerce TRUE or FALSE into numerical values: 1 for TRUE and 0 for FALSE. Suppose that x is a sequence from 0 to 6 like this:

```
x<-0:6
```

Now we can ask questions about the contents of the vector called x. Is x less than 4?

```
x<4
```

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

The answer is yes for the first four values (0, 1, 2 and 3) and no for the last three (4, 5 and 6). Two important logical functions are all and any. They check an entire vector but return a single logical value: TRUE or FALSE. Are all the x values bigger than 0?

```
all(x>0)
```

```
[1] FALSE
```

No. The first x value is a zero. Are any of the x values negative?

```
any(x<0)
```

```
[1] FALSE
```

No. The smallest x value is a zero. We can use the answers of logical functions in arithmetic. We can count the true values of (x<4), using sum

```
sum(x<4)
```

```
[1] 4
```

or we can multiply (x<4) by other vectors

```
(x<4)*runif(7)
```

```
[1] 0.9433433 0.9382651 0.6248691 0.9786844 0.0000000 0.0000000
    0.0000000
```

Logical arithmetic is particularly useful in generating simplified factor levels during statistical modelling. Suppose we want to reduce a five-level factor called treatment to a three-level factor called t2 by lumping together the levels a and e (new factor level 1) and c and d (new factor level 3) while leaving b distinct (with new factor level 2):

```
(treatment<-letters[1:5])
```

```
[1] "a" "b" "c" "d" "e"
```

```
(t2<-factor(1+(treatment=="b")+2*(treatment=="c")+2*(treatment=="d")))
```

```
[1] 1 2 3 3 1
Levels: 1 2 3
```

The new factor t2 gets a value 1 as default for all the factors levels, and we want to leave this as it is for levels a and e. Thus, we do not add anything to the 1 if the old factor level is a or e. For old factor level b, however, we want the result that $t2 = 2$ so we add 1 (treatment=="b") to the original 1 to get the answer we require. This works because the logical expression evaluates to 1 (TRUE) for every case in which the old factor level is b and to 0 (FALSE) in all other cases. For old factor levels c and d we want the result that $t2 = 3$ so we add 2 to the baseline value of 1 if the original factor level is either c (2*(treatment=="c")) or d (2*(treatment=="d")). You may need to read this several times before the penny drops. Note that 'logical equals' is a double = sign without a space between the two equals signs. You need to understand the distinction between:

x <- y x is assigned the value of y (x *gets* the values of y);

x = y in a function or a list x is set to y unless you specify otherwise;

x == y produces TRUE if x is exactly equal to y and FALSE otherwise.

Evaluation of combinations of TRUE and FALSE

It is important to understand how combinations of logical variables evaluate, and to appreciate how logical operations (such as those in Table 2.3) work when there are missing values, NA. Here are all the possible outcomes expressed as a logical vector called x:

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

```
names(x) <- as.character(x)
```

Table 2.3. Logical operations.

Symbol	Meaning
!	logical NOT
&	logical AND
|	logical OR
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	logical equals (double =)
!=	not equal
&&	AND with IF
||	OR with IF
xor(x,y)	exclusive OR
isTRUE(x)	an abbreviation of identical(TRUE,x)

To see the logical combinations of & (logical AND) we can use the outer function with *x* to evaluate all nine combinations of NA, FALSE and TRUE like this:

```
outer(x, x, "&")
```

```
        <NA>  FALSE   TRUE
 <NA>     NA  FALSE     NA
FALSE  FALSE  FALSE  FALSE
 TRUE     NA  FALSE   TRUE
```

Only TRUE & TRUE evaluates to `TRUE`. Note the behaviour of NA & NA and NA & TRUE. Where one of the two components is `NA`, the result will be `NA` if the outcome is ambiguous. Thus, NA & TRUE evaluates to `NA`, but NA & FALSE evaluates to `FALSE`. To see the logical combinations of | (logical OR) write

```
outer(x, x, "|")
```

```
       <NA>  FALSE  TRUE
 <NA>    NA     NA  TRUE
FALSE    NA  FALSE  TRUE
 TRUE  TRUE   TRUE  TRUE
```

Only FALSE | FALSE evaluates to `FALSE`. Note the behaviour of NA | NA and NA | FALSE.

Repeats

You will often want to generate repeats of numbers or characters, for which the function is rep. The object that is named in the first argument is repeated a number of times as specified in the second argument. At its simplest, we would generate five 9s like this:

```
rep(9,5)
```

```
[1] 9 9 9 9 9
```

You can see the issues involved by a comparison of these three increasingly complicated uses of the rep function:

```
rep(1:4, 2)
```

```
[1] 1 2 3 4 1 2 3 4
```

```
rep(1:4, each = 2)
```

```
[1] 1 1 2 2 3 3 4 4
```

```
rep(1:4, each = 2, times = 3)
```

```
[1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

In the simplest case, the *entire* first argument is repeated (i.e. the sequence 1 to 4 is repeated twice). You often want each *element* of the sequence to be repeated, and this is accomplished with the each argument. Finally, you might want each number repeated and the whole series repeated a certain number of times (here 3 times).

When each element of the series is to be repeated a different number of times, then the second argument must be a vector of the same length as the vector comprising the first argument (length 4 in this example). So if we want one 1, two 2s, three 3s and four 4s we would write:

```
rep(1:4,1:4)
```

```
[1] 1 2 2 3 3 3 4 4 4 4
```

In the most complex case, there is a different but irregular repeat of each of the elements of the first argument. Suppose that we need four 1s, one 2, four 3s and two 4s. Then we use the concatenation function c to create a vector of length 4 c(4,1,4,2)) which will act as the second argument to the rep function:

```
rep(1:4,c(4,1,4,2))
```

```
[1] 1 1 1 1 2 3 3 3 3 4 4
```

Generate Factor Levels

The function gl ('generate levels') is useful when you want to encode long vectors of factor levels: the syntax for the three arguments is this:

```
gl('up to', 'with repeats of', 'to total length')
```

Here is the simplest case where we want factor levels up to 4 with repeats of 3 repeated only once (i.e. to total length = 12):

```
gl(4,3)
```

```
[1] 1 1 1 2 2 2 3 3 3 4 4 4
Levels: 1 2 3 4
```

Here is the function when we want that whole pattern repeated twice:

```
gl(4,3,24)
```

```
[1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4

Levels: 1 2 3 4
```

If the total length is not a multiple of the length of the pattern, the vector is truncated:

```
gl(4,3,20)
```

```
[1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3
Levels: 1 2 3 4
```

If you want text for the factor levels, rather than numbers, use labels like this:

```
gl(3,2,24,labels=c("A","B","C"))
```

```
[1] A A B B C C A A B B C C A A B B C C A A B B C C
Levels: A B C
```

Generating Regular Sequences of Numbers

For regularly spaced sequences, often involving integers, it is simplest to use the colon operator. This can produce ascending or descending sequences:

```
10:18
```

```
[1] 10 11 12 13 14 15 16 17 18
```

```
18:10
```

```
[1] 18 17 16 15 14 13 12 11 10
```

```
-0.5:8.5
```

```
[1] -0.5 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

When the interval is not 1.0 you need to use the seq function. In general, the three arguments to seq are: initial value, final value, and increment (or decrement for a declining sequence). Here we want to go from 0 up to 1.5 in steps of 0.2:

```
seq(0,1.5,0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4
```

Note that seq stops *before* it gets to the second argument (1.5) if the increment does not match exactly (our sequence stops at 1.4). If you want to seq downwards, the third argument needs to be negative

```
seq(1.5,0,-0.2)
```

```
[1] 1.5 1.3 1.1 0.9 0.7 0.5 0.3 0.1
```

Again, zero did not match the decrement, so was excluded and the sequence stopped at 0.1. Non-integer increments are particularly useful for generating x values for plotting smooth curves. A curve will look reasonably smooth if it is drawn with 100 straight line segments, so to generate 100 values of x between min(x) and max(x) you could write

```
x.values<-seq(min(x),max(x),(max(x)-min(x))/100)
```

If you want to create a sequence of the same length as an existing vector, then use along like this. Suppose that our existing vector, x, contains 18 random numbers from a normal distribution with a mean of 10.0 and a standard deviation of 2.0:

```
x<-rnorm(18,10,2)
```

and we want to generate a sequence of the same length as this (18) starting at 88 and stepping down to exactly 50 for x[18]

```
seq(88,50,along=x)
```

```
 [1] 88.00000 85.76471 83.52941 81.29412 79.05882 76.82353
     74.58824 72.35294
 [9] 70.11765 67.88235 65.64706 63.41176 61.17647 58.94118
     56.70588 54.47059
[17] 52.23529 50.00000
```

This is useful when you do not want to go to the trouble of working out the size of the increment but you do know the starting value (88 in this case) and the final value (50). If the vector is of length 18 then the sequence will involve 17 decrements of size:

```
(50-88)/17
```

```
[1] -2.235294
```

The function sequence (spelled out in full) is slightly different, because it can produce a *vector* consisting of sequences

```
sequence(5)
```

```
[1] 1 2 3 4 5
```

```
sequence(5:1)
```

```
[1] 1 2 3 4 5 1 2 3 4 1 2 3 1 2 1
```

```
sequence(c(5,2,4))
```

```
[1] 1 2 3 4 5 1 2 1 2 3 4
```

If the argument to sequence is itself a sequence (like 5:1) then several sequences are concatenated (in this case a sequence of 1 to 5 is followed by a sequence of 1 to 4 followed by a sequence of 1 to 3, another of 1 to 2 and a final sequence of 1 to 1 (= 1). The successive sequences need not be regular; the last example shows sequences to 5, then to 2, then to 4.

Variable Names

- Variable names in R are case-sensitive so x is not the same as X.
- Variable names should not begin with numbers (e.g. $1x$) or symbols (e.g. $\%x$).
- Variable names should not contain blank spaces: use *back.pay* (not *back pay*).

Sorting, Ranking and Ordering

These three related concepts are important, and one of them (order) is difficult to understand on first acquaintance. Let's take a simple example:

```
houses<-read.table("c:\\temp \\houses.txt",header=T)
attach(houses)
names(houses)
```

```
[1] "Location"  "Price"
```

Now we apply the three different functions to the vector called Price,

```
ranks<-rank(Price)
sorted<-sort(Price)
ordered<-order(Price)
```

and make a dataframe out of the four vectors like this:

```
view<-data.frame(Price,ranks,sorted,ordered)
view
```

```
   Price  ranks  sorted  ordered
 1   325   12.0      95        9
 2   201   10.0     101        6
 3   157    5.0     117       10
 4   162    6.0     121       12
 5   164    7.0     157        3
 6   101    2.0     162        4
 7   211   11.0     164        5
 8   188    8.5     188        8
 9    95    1.0     188       11
10   117    3.0     201        2
11   188    8.5     211        7
12   121    4.0     325        1
```

Rank

The prices themselves are in no particular sequence. The ranks column contains the value that is the rank of the particular data point (value of Price), where 1 is assigned to the lowest data point and length(Price) – here 12 – is assigned to the highest data point. So the first element, Price = 325, is the highest value in Price. You should check that there are 11 values smaller than 325 in the vector called Price. Fractional ranks indicate ties. There are two 188s in Price and their ranks are 8 and 9. Because they are tied, each gets the average of their two ranks $(8+9)/2 = 8.5$.

Sort

The sorted vector is very straightforward. It contains the values of Price sorted into ascending order. If you want to sort into descending order, use the reverse order function rev like this: y<-rev(sort(x)). Note that sort *is potentially very dangerous*, because it uncouples values that might need to be in the same row of the dataframe (e.g. because they are the explanatory variables associated with a particular value of the response variable). It is bad practice, therefore, to write x<-sort(x), not least because there is no 'unsort' function.

Order

This is the most important of the three functions, and much the hardest to understand on first acquaintance. The order function returns an integer vector containing the permutation

that will sort the input into ascending order. You will need to think about this one. The lowest value of Price is 95. Look at the dataframe and ask yourself what is the subscript in the original vector called Price where 95 occurred. Scanning down the column, you find it in row number 9. This is the first value in ordered, ordered[1]. Where is the next smallest value (101) to be found within Price? It is in position 6, so this is ordered[2]. The third smallest Price (117) is in position 10, so this is ordered[3]. And so on.

This function is particularly useful in sorting dataframes, as explained on p. 113. Using order with subscripts is a much safer option than using sort, because with sort the values of the response variable and the explanatory variables could be uncoupled with potentially disastrous results if this is not realized at the time that modelling was carried out. The beauty of order is that we can use order(Price) as a subscript for Location to obtain the price-ranked list of locations:

```
Location[order(Price)]
```

```
[1] Reading    Staines      Winkfield  Newbury
[5] Bracknell  Camberley    Bagshot    Maidenhead
[9] Warfield   Sunninghill  Windsor    Ascot
```

When you see it used like this, you can see exactly why the function is called order. If you want to reverse the order, just use the rev function like this:

```
Location[rev(order(Price))]
```

```
[1] Ascot       Windsor    Sunninghill Warfield
[5] Maidenhead  Bagshot    Camberley   Bracknell
[9] Newbury     Winkfield  Staines     Reading
```

The sample Function

This function shuffles the contents of a vector into a random sequence while maintaining all the numerical values intact. It is extremely useful for randomization in experimental design, in simulation and in computationally intensive hypothesis testing. Here is the original *y* vector again:

```
y
```

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

and here are two samples of *y*:

```
sample(y)
```

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

```
sample(y)
```

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

The order of the values is different each time that sample is invoked, but the same numbers are shuffled in every case. This is called *sampling without replacement*. You can specify the size of the sample you want as an optional second argument:

```
sample(y,5)
```

```
[1]  9  4  10  8  11
```

```
sample(y,5)
```

```
[1] 9  3  4  2  8
```

The option replace=T allows for *sampling with replacement*, which is the basis of bootstrapping (see p. 320). The vector produced by the sample function with replace=T is the same length as the vector sampled, but some values are left out at random and other values, again at random, appear two or more times. In this sample, 10 has been left out, and there are now three 9s:

```
sample(y,replace=T)
```

```
[1]  9  6  11  2  9  4  6  8  8  4  4  4  3  9  3
```

In this next case, the are two 10s and only one 9:

```
sample(y,replace=T)
```

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

More advanced options in sample include specifying different probabilities with which each element is to be sampled (prob=). For example, if we want to take four numbers at random from the sequence 1:10 without replacement where the probability of selection (p) is 5 times greater for the middle numbers (5 and 6) than for the first or last numbers, and we want to do this five times, we could write

```
p <- c(1, 2, 3, 4, 5, 5, 4, 3, 2, 1)
```

```
x<-1:10
```

```
sapply(1:5,function(i) sample(x,4,prob=p))
```

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

so the four random numbers in the first trial were 8, 7, 4 and 9 (i.e. column 1).

Matrices

There are several ways of making a matrix. You can create one directly like this:

```
X<-matrix(c(1,0,0,0,1,0,0,0,1),nrow=3)
```

```
X
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

where, by default, the numbers are entered columnwise. The class and attributes of X indicate that it is a matrix of three rows and three columns (these are its dim attributes)

```
class(X)
```

```
[1] "matrix"
```

```
attributes(X)
```

```
$dim
```

```
[1] 3 3
```

In the next example, the data in the vector appear row-wise, so we indicate this with byrow=T:

```
vector<-c(1,2,3,4,4,3,2,1)
V<-matrix(vector,byrow=T,nrow=2)
V
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    4    3    2    1
```

Another way to convert a vector into a matrix is by providing the vector object with two dimensions (rows and columns) using the dim function like this:

```
dim(vector)<-c(4,2)
```

We can check that vector has now become a matrix:

```
is.matrix(vector)
```

```
[1] TRUE
```

We need to be careful, however, because we have made no allowance at this stage for the fact that the data were entered row-wise into vector:

```
vector
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    3
[3,]    3    2
[4,]    4    1
```

The matrix we want is the transpose, t, of this matrix:

```
(vector<-t(vector))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    4    3    2    1
```

Naming the rows and columns of matrices

At first, matrices have numbers naming their rows and columns (see above). Here is a 4×5 matrix of random integers from a Poisson distribution with mean $= 1.5$:

```
X<-matrix(rpois(20,1.5),nrow=4)
X
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    2    5    3
[2,]    1    1    3    1    3
[3,]    3    1    0    2    2
[4,]    1    0    2    1    0
```

Suppose that the rows refer to four different trials and we want to label the rows 'Trial.1' etc. We employ the function rownames to do this. We could use the paste function (see p. 44) but here we take advantage of the prefix option:

```
rownames(X)<-rownames(X,do.NULL=FALSE,prefix="Trial.")
X
```

```
        [,1] [,2] [,3] [,4] [,5]
Trial.1    1    0    2    5    3
Trial.2    1    1    3    1    3
Trial.3    3    1    0    2    2
Trial.4    1    0    2    1    0
```

For the columns we want to supply a vector of different names for the five drugs involved in the trial, and use this to specify the colnames(X):

```
drug.names<-c("aspirin", "paracetamol", "nurofen", "hedex", "placebo")
colnames(X)<-drug.names
X
```

```
        aspirin paracetamol nurofen hedex placebo
Trial.1       1           0       2     5       3
Trial.2       1           1       3     1       3
Trial.3       3           1       0     2       2
Trial.4       1           0       2     1       0
```

Alternatively, you can use the dimnames function to give names to the rows and/or columns of a matrix. In this example we want the rows to be unlabelled (NULL) and the column names to be of the form 'drug.1', 'drug.2', etc. The argument to dimnames has to be a list (rows first, columns second, as usual) with the elements of the list of exactly the correct lengths (4 and 5 in this particular case):

```
dimnames(X)<-list(NULL,paste("drug.",1:5,sep=""))
X
```

```
     drug.1 drug.2 drug.3 drug.4 drug.5
[1,]      1      0      2      5      3
[2,]      1      1      3      1      3
[3,]      3      1      0      2      2
[4,]      1      0      2      1      0
```

Calculations on rows or columns of the matrix

We could use subscripts to select parts of the matrix, with a blank meaning 'all of the rows' or 'all of the columns'. Here is the mean of the rightmost column (number 5),

```
mean(X[,5])
```

```
[1] 2
```

calculated over all the rows (blank then comma), and the variance of the bottom row,

```
var(X[4,])
```

```
[1] 0.7
```

calculated over all of the columns (a blank in the second position). There are some special functions for calculating summary statistics on matrices:

```
rowSums(X)
```

```
[1] 11 9 8 4
```

```
colSums(X)
```

```
[1] 6 2 7 9 8
```

```
rowMeans(X)
```

```
[1] 2.2 1.8 1.6 0.8
```

```
colMeans(X)
```

```
[1] 1.50 0.50 1.75 2.25 2.00
```

These functions are built for speed, and blur some of the subtleties of dealing with NA or NaN. If such subtlety is an issue, then use apply instead (p. 68). Remember that columns are margin no. 2 (rows are margin no. 1):

```
apply(X,2,mean)
```

```
[1] 1.50 0.50 1.75 2.25 2.00
```

You might want to sum groups of rows within columns, and rowsum (singular and all lower case, in contrast to rowSums, above) is a very efficient function for this. In this case we want to group together row 1 and row 4 (as group A) and row 2 and row 3 (group B). Note that the grouping vector has to have length equal to the number of rows:

```
group=c("A","B","B","A")
```

```
rowsum(X, group)
```

```
  [,1] [,2] [,3] [,4] [,5]
A    2    0    4    6    3
B    4    2    3    3    5
```

You could achieve the same ends (but more slowly) with tapply or aggregate:

```
tapply(X, list(group[row(X)], col(X)), sum)
```

```
  1 2 3 4 5
A 2 0 4 6 3
B 4 2 3 3 5
```

Note the use of row(X) and col(X), with row(X) used as a subscript on group.

```
aggregate(X,list(group),sum)
```

```
  Group.1 V1 V2 V3 V4 V5
1       A  2  0  4  6  3
2       B  4  2  3  3  5
```

Suppose that we want to shuffle the elements of each column of a matrix independently. We apply the function sample to each column (margin no. 2) like this:

```
apply(X,2,sample)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    2    1    3
[2,]    3    1    0    1    3
[3,]    1    0    3    2    0
[4,]    1    0    2    5    2
```

```
apply(X,2,sample)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    0    5    2
[2,]    1    1    2    1    3
[3,]    3    0    2    2    3
[4,]    1    0    3    1    0
```

and so on, for as many shuffled samples as you need.

Adding rows and columns to the matrix

In this particular case we have been asked to add a row at the bottom showing the column means, and a column at the right showing the row variances:

```
X<-rbind(X,apply(X,2,mean))
X<-cbind(X,apply(X,1,var))
X
```

```
     [,1] [,2] [,3] [,4] [,5]     [,6]
[1,]  1.0  0.0 2.00 5.00    3 3.70000
[2,]  1.0  1.0 3.00 1.00    3 1.20000
[3,]  3.0  1.0 0.00 2.00    2 1.30000
[4,]  1.0  0.0 2.00 1.00    0 0.70000
[5,]  1.5  0.5 1.75 2.25    2 0.45625
```

Note that the number of decimal places varies across columns, with one in columns 1 and 2, two in columns 3 and 4, none in column 5 (integers) and five in column 6. The default in R is to print the minimum number of decimal places consistent with the contents of the column as a whole.

Next, we need to label the sixth column as ‘variance’ and the fifth row as ‘mean’:

```
colnames(X)<-c(1:5,"variance")
rownames(X)<-c(1:4,"mean")
X
```

```
     1   2    3    4 5 variance
1  1.0 0.0 2.00 5.00 3  3.70000
2  1.0 1.0 3.00 1.00 3  1.20000
3  3.0 1.0 0.00 2.00 2  1.30000
4  1.0 0.0 2.00 1.00 0  0.70000
```

```
mean  1.5  0.5  1.75  2.25  2  0.45625
```

When a matrix with a single row or column is created by a subscripting operation, for example row <- mat[2,], it is by default turned into a vector. In a similar way, if an array with dimension, say, $2 \times 3 \times 1 \times 4$ is created by subscripting it will be coerced into a $2 \times 3 \times 4$ array, losing the unnecessary dimension. After much discussion this has been determined to be a *feature* of R. To prevent this happening, add the option drop = FALSE to the subscripting. For example,

```
rowmatrix <- mat[2, , drop = FALSE]
colmatrix <- mat[, 2, drop = FALSE]
a <- b[1, 1, 1, drop = FALSE]
```

The drop = FALSE option should be used defensively when programming. For example, the statement

```
somerows <- mat[index,]
```

will return a vector rather than a matrix if index happens to have length 1, and this might cause errors later in the code. It should be written as

```
somerows <- mat[index , , drop = FALSE]
```

The sweep function

The sweep function is used to 'sweep out' array summaries from vectors, matrices, arrays or dataframes. In this example we want to express a matrix in terms of the departures of each value from its column mean.

```
matdata<-read.table("c: \\temp \\sweepdata.txt")
```

First, you need to create a vector containing the parameters that you intend to sweep out of the matrix. In this case we want to compute the four column means:

```
(cols<-apply(matdata,2,mean))
```

```
  V1      V2     V3       V4
4.60   13.30   0.44   151.60
```

Now it is straightforward to express all of the data in matdata as departures from the relevant column means:

```
sweep(matdata,2,cols)
```

```
      V1    V2     V3     V4
1   -1.6  -1.3 -0.04 -26.6
2    0.4  -1.3  0.26  14.4
3    2.4   1.7  0.36  22.4
4    2.4   0.7  0.26 -23.6
5    0.4   4.7 -0.14 -15.6
6    4.4  -0.3 -0.24   3.4
7    2.4   1.7  0.06 -36.6
8   -2.6  -0.3  0.06  17.4
9   -3.6  -3.3 -0.34  30.4
10  -4.6  -2.3 -0.24  14.4
```

Note the use of margin = 2 as the second argument to indicate that we want the sweep to be carried out on the columns (rather than on the rows). A related function, scale, is used for centring and scaling data in terms of standard deviations (p. 191).

You can see what sweep has done by doing the calculation long-hand. The operation of this particular sweep is simply one of subtraction. The only issue is that the subtracted object has to have the same dimensions as the matrix to be swept (in this example, 10 rows of 4 columns). Thus, to sweep out the column means, the object to be subtracted from matdata must have the each column mean repeated in each of the 10 rows of 4 columns:

```
(col.means<-matrix(rep(cols,rep(10,4)),nrow=10))
```

```
      [,1] [,2] [,3]  [,4]
 [1,]  4.6 13.3 0.44 151.6
 [2,]  4.6 13.3 0.44 151.6
 [3,]  4.6 13.3 0.44 151.6
 [4,]  4.6 13.3 0.44 151.6
 [5,]  4.6 13.3 0.44 151.6
 [6,]  4.6 13.3 0.44 151.6
 [7,]  4.6 13.3 0.44 151.6
 [8,]  4.6 13.3 0.44 151.6
 [9,]  4.6 13.3 0.44 151.6
[10,]  4.6 13.3 0.44 151.6
```

Then the same result as we got from sweep is obtained simply by

```
matdata-col.means
```

Suppose that you want to obtain the subscripts for a columnwise or a row-wise sweep of the data. Here are the row subscripts repeated in each column:

```
apply(matdata,2,function (x) 1:10)
```

```
      V1  V2  V3  V4
 [1,]  1   1   1   1
 [2,]  2   2   2   2
 [3,]  3   3   3   3
 [4,]  4   4   4   4
 [5,]  5   5   5   5
 [6,]  6   6   6   6
 [7,]  7   7   7   7
 [8,]  8   8   8   8
 [9,]  9   9   9   9
[10,] 10  10  10  10
```

Here are the column subscripts repeated in each row:

```
t(apply(matdata,1,function (x) 1:4))
```

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

Here is the same procedure using sweep:

```
sweep(matdata,1,1:10,function(a,b) b)
```

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

```
sweep(matdata,2,1:4,function(a,b) b)
```

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

Arrays

Arrays are numeric objects with dimension attributes. We start with the numbers 1 to 25 in a vector called array:

```
array<-1:25
is.matrix(array)
```

```
[1] FALSE
```

```
dim(array)
```

```
NULL
```

The vector is not a matrix and it has no (NULL) dimensional attributes. We give the object dimensions like this (say, with five rows and five columns):

```
dim(array)<-c(5,5)
```

Now it does have dimensions and it is a matrix:

```
dim(array)
```

```
[1] 5 5
```

```
is.matrix(array)
```

```
[1] TRUE
```

When we look at array it is presented as a two-dimensional table (but note that it is *not* a table object; see p. 187):

```
array
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

```
is.table(array)
```

```
[1] FALSE
```

Note that the values have been entered into array in columnwise sequence: this is the default in R. Thus a vector is a one-dimensional array that lacks any dim attributes. A matrix is a two-dimensional array. Arrays of three or more dimensions do not have any special names in R; they are simply referred to as three-dimensional or five-dimensional arrays. You should practise with subscript operations on arrays until you are thoroughly familiar with them. Mastering the use of subscripts will open up many of R's most powerful features for working with dataframes, vectors, matrices, arrays and lists. Here is a three-dimensional array of the first 24 lower-case letters with three matrices each of four rows and two columns:

```
A<-letters[1:24]
dim(A)<-c(4,2,3)
A
```

```
, , 1

     [,1] [,2]
[1,]  "a"  "e"
[2,]  "b"  "f"
[3,]  "c"  "g"
[4,]  "d"  "h"

, , 2

     [,1] [,2]
[1,]  "i"  "m"
[2,]  "j"  "n"
```

```
[3,]   "k"   "o"
[4,]   "l"   "p"

, , 3
      [,1] [,2]
[1,]   "q"   "u"
[2,]   "r"   "v"
[3,]   "s"   "w"
[4,]   "t"   "x"
```

We want to select all the letters a to p. These are all the rows and all the columns of tables 1 and 2, so the appropriate subscripts are [„1:2]

```
A[„1:2]
```

```
, , 1

      [,1] [,2]
[1,]   "a"   "e"
[2,]   "b"   "f"
[3,]   "c"   "g"
[4,]   "d"   "h"

, , 2
      [,1] [,2]
[1,]   "i"   "m"
[2,]   "j"   "n"
[3,]   "k"   "o"
[4,]   "l"   "p"
```

Next, we want only the letters q to x. These are all the rows and all the columns from the third table, so the appropriate subscripts are [„3]:

```
A[„3]
```

```
      [,1] [,2]
[1,]   "q"   "u"
[2,]   "r"   "v"
[3,]   "s"   "w"
[4,]   "t"   "x"
```

Here, we want only c, g, k, o, s and w. These are the third rows of all three tables, so the appropriate subscripts are [3„]:

```
A[3„]
```

```
      [,1] [,2] [,3]
[1,]   "c"   "k"   "s"
[2,]   "g"   "o"   "w"
```

Note that when we drop the whole first dimension (there is just one row in A[3„]) the shape of the resulting matrix is altered (two rows and three columns in this example). This is a *feature* of R, but you can override it by saying drop = F to retain all three dimensions:

```
A[3„,drop=F]
```

```
, , 1

     [,1] [,2]
[1,]  "c"  "g"

, , 2

     [,1] [,2]
[1,]  "k"  "o"

, , 3

     [,1] [,2]
[1,]  "s"  "w"
```

Finally, suppose we want all the rows of the second column from table 1, A[,2,1], the first column from table 2, A[,1,2], and the second column from table 3, A[,2,3]. Because we want all the rows in each case, so the first subscript is blank, but we want different column numbers (cs) in different tables (ts) as follows:

```
cs<-c(2,1,2)
ts<-c(1,2,3)
```

To get the answer, use sapply to concatenate the columns from each table like this:

```
sapply (1:3, function(i) A[,cs[i],ts[i]])
```

```
     [,1] [,2] [,3]
[1,]  "e"  "i"  "u"
[2,]  "f"  "j"  "v"
[3,]  "g"  "k"  "w"
[4,]  "h"  "l"  "x"
```

Character Strings

In R, character strings are defined by double quotation marks:

```
a<-"abc"
b<-"123"
```

Numbers can be characters (as in *b*, above), but characters cannot be numbers.

```
as.numeric(a)
```

```
[1] NA
Warning message:
NAs introduced by coercion
```

```
as.numeric(b)
```

```
[1] 123
```

One of the initially confusing things about character strings is the distinction between the length of a character object (a vector) and the numbers of characters in the strings comprising that object. An example should make the distinction clear:

```
pets<-c("cat","dog","gerbil","terrapin")
```

Here, pets is a vector comprising four character strings:

```
length(pets)
```

```
[1] 4
```

and the individual character strings have 3, 3, 6 and 7 characters, respectively:

```
nchar(pets)
```

```
[1] 3 3 6 7
```

When first defined, character strings are not factors:

```
class(pets)
```

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

```
is.factor(pets)
```

```
[1] FALSE
```

However, if the vector of characters called pets was part of a dataframe, then R would coerce all the character variables to act as factors:

```
df<-data.frame(pets)
is.factor(df$pets)
```

```
[1] TRUE
```

There are built-in vectors in R that contain the 26 letters of the alphabet in lower case (letters) and in upper case (LETTERS):

```
letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
[17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
[17] "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

To discover which number in the alphabet the letter n is, you can use the which function like this:

```
which(letters=="n")
```

```
[1] 14
```

For the purposes of printing you might want to suppress the quotes that appear around character strings by default. The function to do this is called noquote:

```
noquote(letters)
```

```
[1] a b c d e f g h i j k l m n o p q r s t u v w x y z
```

You can amalgamate strings into vectors of character information:

```
c(a,b)
```

```
[1] "abc" "123"
```

This shows that the concatenation produces a vector of two strings. It does *not* convert two 3-character strings into one 6-charater string. The R function to do that is paste:

```
paste(a,b,sep="")
```

```
[1] "abc123"
```

The third argument, sep="", means that the two character strings are to be pasted together without any separator between them: the default for paste is to insert a single blank space, like this:

```
paste(a,b)
```

```
[1] "abc 123"
```

Notice that you do *not* lose blanks that are within character strings when you use the sep="" option in paste.

```
paste(a,b,"a longer phrase containing blanks",sep="")
```

```
[1] "abc123a longer phrase containing blanks"
```

If one of the arguments to paste is a vector, each of the elements of the vector is pasted to the specified character string to produce an object of the same length as the vector:

```
d<-c(a,b,"new")
e<-paste(d,"a longer phrase containing blanks")
e
```

```
[1] "abc a longer phrase containing blanks"
[2] "123 a longer phrase containing blanks"
[3] "new a longer phrase containing blanks"
```

Extracting parts of strings

We being by defining a phrase:

```
phrase<-"the quick brown fox jumps over the lazy dog"
```

The function called substr is used to extract substrings of a specified number of characters from a character string. Here is the code to extract the first, the first and second, the first, second and third, . . . (up to 20) characters from our phrase

```
q<-character(20)
for (i in 1:20) q[i]<- substr(phrase,1,i)
q
```

```
[1]  "t"                   "th"                "the"
[4]  "the "                "the q"             "the qu"
[7]  "the qui"             "the quic"          "the quick"
[10] "the quick "          "the quick b"       "the quick br"
[13] "the quick bro"       "the quick brow"    "the quick brown"
[16] "the quick brown "    "the quick brown f" "the quick brown fo"
[19] "the quick brown fox "                    "the quick brown fox "
```

The second argument in substr is the number of the character at which extraction is to begin (in this case always the first), and the third argument is the number of the character at which extraction is to end (in this case, the *i*th). To split up a character string into individual characters, we use strsplit like this

```
strsplit(phrase,split=character(0))
```

```
[[1]]
[1]   "t" "h" "e"  " " "q" "u" "i" "c" "k" " " "b" "r" "o" "w" "n" " "
[17]  "f" "o" "x"  " " "j" "u" "m" "p" "s" " " "o" "v" "e" "r"
[31]   " " "t" "h" "e"  " " "l" "a" "z" "y" " " "d" "o" "g"
```

The table function is useful for counting the number of occurrences of characters of different kinds:

```
table(strsplit(phrase,split=character(0)))
```

```
  a b c d e f g h i j k l m n o p q r s t u v w x y z
8 1 1 1 1 3 1 1 2 1 1 1 1 1 1 4 1 1 2 1 2 2 1 1 1 1 1
```

This demonstrates that all of the letters of the alphabet were used at least once within our phrase, and that there were 8 blanks within phrase. This suggests a way of counting the number of words in a phrase, given that this will always be one more than the number of blanks:

```
words<-1+table(strsplit(phrase,split=character(0)))[1]
words
```

```
9
```

When we specify a particular string to form the basis of the split, we end up with a list made up from the components of the string that *do not contain the specified string*. This is hard to understand without an example. Suppose we split our phrase using 'the':

```
strsplit(phrase,"the")
```

```
[[1]]

[1]  ""   " quick brown fox jumps over "   " lazy dog"
```

There are three elements in this list: the first one is the empty string "" because the first three characters within phrase were exactly 'the' ; the second element contains the part of the phrase between the two occurrences of the string 'the'; and the third element is the end of the phrase, following the second 'the'. Suppose that we want to extract the characters between the first and second occurrences of 'the'. This is achieved very simply, using subscripts to extract the second element of the list:

```
strsplit(phrase,"the")[[1]] [2]
```

```
[1]  " quick brown fox jumps over "
```

Note that the first subscript in double square brackets refers to the number within the list (there *is* only one list in this case) and the second subscript refers to the second element within this list. So if we want to know how many characters there are between the first and second occurrences of the word "the" within our phrase, we put:

```
nchar(strsplit(phrase,"the")[[1]] [2])
```

```
[1]  28
```

It is easy to switch between upper and lower cases using the toupper and tolower functions:

```
toupper(phrase)
```

```
[1]  "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
```

```
tolower(toupper(phrase))
```

```
[1]  "the quick brown fox jumps over the lazy dog"
```

The match Function

The match function answers the question 'Where do the values in the second vector appear in the first vector?'. This is impossible to understand without an example:

```
first<-c(5,8,3,5,3,6,4,4,2,8,8,8,4,4,6)
second<-c(8,6,4,2)
match(first,second)
```

```
[1] NA  1 NA NA NA  2  3  3  4  1  1  1  3  3  2
```

The first thing to note is that match produces a vector of subscripts (index values) and that these are subscripts within the *second* vector. The length of the vector produced by match is the length of the *first* vector (15 in this example). If elements of the first vector do not occur anywhere in the second vector, then match produces NA.

Why would you ever want to use this? Suppose you wanted to give drug A to all the patients in the first vector that were identified in the second vector, and drug B to all the others (i.e. those identified by NA in the output of match, above, because they did *not* appear in the second vector). You create a vector called drug with two elements (A and B), then select the appropriate drug on the basis of whether or not match(first,second) is NA:

```
drug<-c("A","B")
drug[1+is.na(match(first,second))]
```

```
[1] "B" "A" "B" "B" "B" "A" "A" "A" "A" "A" "A" "A" "A"  "A" "A"
```

The match function can also be very powerful in manipulating dataframes to mimic the functionality of a relational database (p. 127).

Writing functions in R

Functions in R are objects that carry out operations on *arguments* that are supplied to them and return one or more values. The syntax for writing a function is

```
function (argument list) body
```

The first component of the function declaration is the keyword function, which indicates to R that you want to create a function. An argument list is a comma-separated list of formal arguments. A formal argument can be a symbol (i.e. a variable name such as *x* or *y*), a statement of the form symbol = expression (e.g. pch=16) or the special formal argument ... (triple dot). The body can be any valid R expression or set of R expressions. Generally, the body is a group of expressions contained in curly brackets { }, with each expression on a separate line. Functions are typically assigned to symbols, but they need not to be. This will only begin to mean anything after you have seen several examples in operation.

Arithmetic mean of a single sample

The mean is the sum of the numbers $\sum y$ divided by the number of numbers $n = \sum 1$ (summing over the number of numbers in the vector called y). The R function for n is length(y) and for $\sum y$ is sum(y), so a function to compute arithmetic means is

```
arithmetic.mean<-function(x)  sum(x)/length(x)
```

We should test the function with some data where we know the right answer:

```
y<-c(3,3,4,5,5)
arithmetic.mean(y)
```

```
[1] 4
```

Needless to say, there is a built-in function for arithmetic means called mean:

```
mean(y)
```

```
[1] 4
```

Median of a single sample

The median (or 50th percentile) is the middle value of the sorted values of a vector of numbers:

```
sort(y)[ceiling(length(y)/2)
```

There is slight hitch here, of course, because if the vector contains an even number of numbers, then there *is* no middle value. The logic is that we need to work out the arithmetic average of the two values of y on either side of the middle. The question now arises as to how we know, in general, whether the vector y contains an odd or an even number of numbers, so that we can decide which of the two methods to use. The trick here is to use modulo 2 (p. 12). Now we have all the tools we need to write a general function to calculate medians. Let's call the function med and define it like this:

```
med<-function(x) {
odd.even<-length(x)%%2
if (odd.even == 0) (sort(x)[length(x)/2]+sort(x)[1+ length(x)/2])/2
else sort(x)[ceiling(length(x)/2)]
}
```

Notice that when the if statement is true (i.e. we have an even number of numbers) then the expression immediately following the if function is evaluated (this is the code for calculating the median with an even number of numbers). When the if statement is false (i.e. we have an odd number of numbers, and odd.even == 1) then the expression following the else function is evaluated (this is the code for calculating the median with an odd number of numbers). Let's try it out, first with the odd-numbered vector y, then with the even-numbered vector y[-1], after the first element of y (y[1] = 3) has been dropped (using the negative subscript):

```
med(y)
```

```
[1] 4
```

```
med(y[-1])
```

```
[1] 4.5
```

Again, you won't be surprised that there is a built-in function for calculating medians, and helpfully it is called median.

Geometric mean

For processes that change multiplicatively rather than additively, neither the arithmetic mean nor the median is an ideal measure of central tendency. Under these conditions, the appropriate measure is the geometric mean. The formal definition of this is somewhat abstract: the geometric mean is the *n*th root of the product of the data. If we use capital Greek pi ($\prod$) to represent multiplication, and $\hat{y}$ (pronounced *y*-hat) to represent the geometric mean, then

$$\hat{y} = \sqrt[n]{\prod y}.$$

Let's take a simple example we can work out by hand: the numbers of insects on 5 plants were as follows: 10, 1, 1000, 1, 10. Multiplying the numbers together gives 100 000. There are five numbers, so we want the fifth root of this. Roots are hard to do in your head, so we'll use R as a calculator. Remember that roots are fractional powers, so the fifth root is a number raised to the power $1/5 = 0.2$. In R, powers are denoted by the ^ symbol:

```
100000^0.2
```

```
[1] 10
```

So the geometric mean of these insect numbers is 10 insects per stem. Note that two of the data were exactly like this, so it seems a reasonable estimate of central tendency. The arithmetic mean, on the other hand, is a hopeless measure of central tendency, because the large value (1000) is so influential: it is given by $(10 + 1 + 1000 + 1 + 10)/5 = 204.4$, and none of the data is close to it.

```
insects<-c(1,10,1000,10,1)
mean(insects)
```

```
[1] 204.4
```

Another way to calculate geometric mean involves the use of logarithms. Recall that to multiply numbers together we add up their logarithms. And to take roots, we divide the logarithm by the root. So we should be able to calculate a geometric mean by finding the antilog (exp) of the average of the logarithms (log) of the data:

```
exp(mean(log(insects)))
```

```
[1] 10
```

So a function to calculate geometric mean of a vector of numbers *x*:

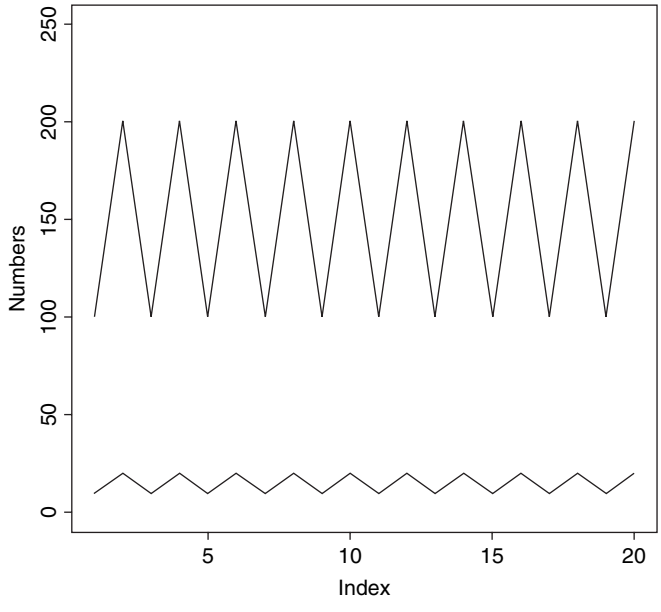
```
geometric<-function (x) exp(mean(log(x)))
```

and testing it with the insect data

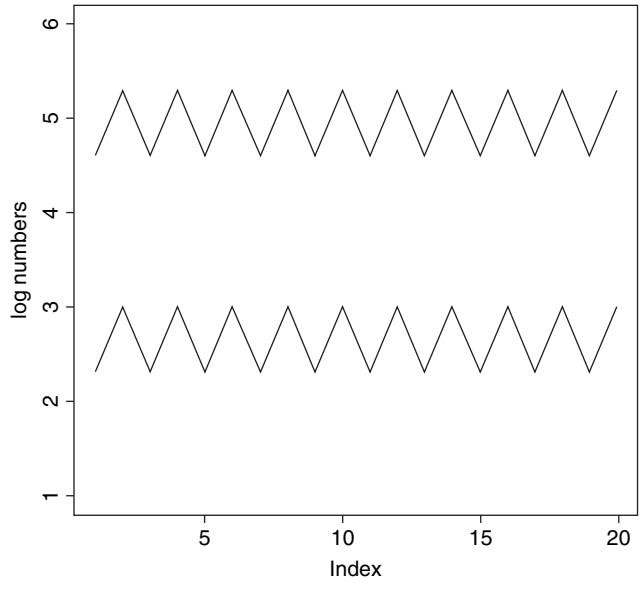
```
geometric(insects)
```

```
[1] 10
```

The use of geometric means draws attention to a general scientific issue. Look at the figure below, which shows numbers varying through time in two populations. Now ask yourself which population is the more variable. Chances are, you will pick the upper line:



But now look at the scale on the *y* axis. The upper population is fluctuating 100, 200, 100, 200 and so on. In other words, it is doubling and halving, doubling and halving. The lower curve is fluctuating 10, 20, 10, 20, 10, 20 and so on. It, too, is doubling and halving, doubling and halving. So the answer to the question is that they are equally variable. It is just that one population has a higher mean value than the other (150 vs. 15 in this case). In order not to fall into the trap of saying that the upper curve is more variable than the lower curve, it is good practice to graph the logarithms rather than the raw values of things like population sizes that change multiplicatively, as below.



Now it is clear that both populations are equally variable. Note the change of scale, as specified using the ylim=c(1,6) option within the plot function (p. 181).

Harmonic mean

Consider the following problem. An elephant has a territory which is a square of side = 2 km. Each morning, the elephant walks the boundary of this territory. He begins the day at a sedate pace, walking the first side of the territory at a speed of 1 km/hr. On the second side, he has sped up to 2 km/hr. By the third side he has accelerated to an impressive 4 km/hr, but this so wears him out, that he has to return on the final side at a sluggish 1 km/hr. So what is his average speed over the ground? You might say he travelled at 1, 2, 4 and 1 km/hr so the average speed is $(1+2+4+1)/4 = 8/4 = 2$ km/hr. But that is wrong. Can you see how to work out the right answer? Recall that velocity is defined as distance travelled divided by time taken. The distance travelled is easy: it's just $4 \times 2 = 8$ km. The time taken is a bit harder. The first edge was 2 km long, and travelling at 1 km/hr this must have taken 2 hr. The second edge was 2 km long, and travelling at 2 km/hr this must have taken 1 hr. The third edge was 2 km long and travelling at 4 km/hr this must have taken 0.5 hr. The final edge was 2 km long and travelling at 1 km/hr this must have taken 2 hr. So the total time taken was $2+1+0.5+2 = 5.5$ hr. So the average speed is not 2 km/hr but $8/5.5 = 1.4545$ km/hr. The way to solve this problem is to use the **harmonic mean**.

The harmonic mean is the reciprocal of the average of the reciprocals. The average of our reciprocals is

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{4}+\frac{1}{1}=\frac{2.75}{4}=0.6875.$$

The reciprocal of this average is the harmonic mean

$$\frac{4}{2.75}=\frac{1}{0.6875}=1.4545.$$

In symbols, therefore, the harmonic mean, $\tilde{y}$ (y-curl), is given by

$$\tilde{y}=\frac{1}{(\Sigma(1/y))/n}=\frac{n}{\Sigma(1/y)}.$$

An R function for calculating harmonic means, therefore, could be

```
harmonic<-function (x) 1/mean(1/x)
```

and testing it on our elephant data gives

```
harmonic(c(1,2,4,1))
```

```
[1] 1.454545
```

Variance

A measure of variability is perhaps the most important quantity in statistical analysis. The greater the variability in the data, the greater will be our uncertainty in the values of parameters estimated from the data, and the less will be our ability to distinguish between competing hypotheses about the data.

The variance of a sample is measured as a function of 'the sum of the squares of the difference between the data and the arithmetic mean'. This important quantity is called the 'sum of squares':

$$SS = \Sigma(y - \bar{y})^2.$$

Naturally, this quantity gets bigger with every new data point you add to the sample. An obvious way to compensate for this is to measure variability as the average of the squared departures from the mean (the 'mean square deviation'.). There is a slight problem, however. Look at the formula for the sum of squares, SS, above and ask yourself what you need to know before you can calculate it. You have the data, y, but the only way you can know the sample mean, $\bar{y}$, is to calculate it from the data (you will never know $\bar{y}$ in advance).

Degrees of freedom

To complete our calculation of the variance we need the **degrees of freedom** (d.f.) This important concept in statistics is defined as follows:

$$\text{d.f.} = n - k,$$

which is the sample size, n, minus the number of parameters, k, estimated from the data. For the variance, we have estimated one parameter from the data, $\bar{y}$, and so there are $n - 1$ degrees of freedom. In a linear regression, we estimate two parameters from the data, the slope and the intercept, and so there are $n - 2$ degrees of freedom in a regression analysis.

Variance is denoted by the lower-case Latin letter s squared: s^2. The square root of variance, s, is called the standard deviation. We always calculate variance as

$$\text{variance} = s^2 = \frac{\text{sum of squares}}{\text{degrees of freedom}}.$$

Consider the following data, y:

```
y<-c(13,7,5,12,9,15,6,11,9,7,12)
```

We need to write a function to calculate the sample variance: we call it variance and define it like this:

```
variance<-function(x) sum((x – mean(x))^2)/(length(x)-1)
```

and use it like this:

```
variance(y)
```

```
[1] 10.25455
```

Our measure of variability in these data, the variance, is thus 10.25455. It is said to be an unbiased estimator because we divide the sum of squares by the degrees of freedom $(n - 1)$ rather than by the sample size, n, to compensate for the fact that we have estimated one parameter from the data. So the variance is *close* to the average squared difference between the data and the mean, especially for large samples, but it is not exactly equal to the mean squared deviation. Needless to say, R has a built-in function to calculate variance called var:

```
var(y)
```

```
[1] 10.25455
```

Variance Ratio Test

How do we know if two variances are significantly different from one another? We need to carry out Fisher's *F* test, the ratio of the two variances (see p. 224). Here is a function to print the *p* value (p. 290) associated with a comparison of the larger and smaller variances:

```
variance.ratio<-function(x,y) {
  v1<-var(x)
  v2<-var(y)
  if (var(x) > var(y)) {
  vr<-var(x)/var(y)
  df1<-length(x)-1
  df2<-length(y)-1}
else { vr<-var(y)/var(x)
  df1<-length(y)-1
  df2<-length(x)-1}
2*(1-pf(vr,df1,df2)) }
```

The last line of our function works out the probability of getting an *F* ratio as big as *vr* or bigger by chance alone if the two variances were really the same, using the cumulative probability of the *F* distribution, which is an R function called pf. We need to supply pf with three *arguments*: the size of the variance ratio (*vr*), the number of degrees of freedom in the numerator (9) and the number of degrees of freedom in the denominator (also 9).

Here are some data to test our function. They are normally distributed random numbers but the first set has a variance of 4 and the second a variance of 16 (i.e. standard deviations of 2 and 4, respectively):

```
a<-rnorm(10,15,2)
b<-rnorm(10,15,4)
```

Here is our function in action:

```
variance.ratio(a,b)
```

```
[1] 0.01593334
```

We can compare our *p* with the *p*-value given by the built-in function called var.test

```
var.test(a,b)
```

```
F test to compare two variances

data: a and b
F = 0.1748, num df = 9, denom df = 9, p-value = 0.01593
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
0.04340939 0.70360673
```

```
sample estimates:
ratio of variances
         0.1747660
```

Using Variance

Variance is used in two main ways: for establishing measures of unreliability (e.g. confidence intervals) and for testing hypotheses (e.g. Student's t test). Here we will concentrate on the former; the latter is discussed in Chapter 8.

Consider the properties that you would like a measure of unreliability to possess. As the variance of the data increases, what would happen to the unreliability of estimated parameters? Would it go up or down? Unreliability would go up as variance increased, so we would want to have the variance on the top (the numerator) of any divisions in our formula for unreliability:

$$\text{unreliability} \propto s^2.$$

What about sample size? Would you want your estimate of unreliability to go up or down as sample size, n, increased? You would want unreliability to go down as sample size went up, so you would put sample size on the bottom of the formula for unreliability (i.e. in the denominator):

$$\text{unreliability} \propto \frac{s^2}{n}.$$

Finally, consider the units in which unreliability is measured. What are the units in which our current measure is expressed? Sample size is dimensionless, but variance is based on the sum of squared differences, so it has dimensions of mean squared. So if the mean was a length in cm, the variance would be an area in cm^2. This is an unfortunate state of affairs. It would make good sense to have the dimensions of the unreliability measure and of the parameter whose unreliability it is measuring the same. That is why all unreliability measures are enclosed inside a big square root term. Unreliability measures are called *standard errors*. What we have just worked out is the *standard error of the mean*,

$$se_{\bar{y}} = \sqrt{\frac{s^2}{n}},$$

where s^2 is the variance and n is the sample size. There is no built-in R function to calculate the standard error of a mean, but it is easy to write one:

```
se<-function(x) sqrt(var(x)/length(x))
```

You can refer to functions from within other functions. Recall that a confidence interval (CI) is 't from tables times the standard error':

$$CI = t_{\alpha/2,df} \times se.$$

The R function qt gives the value of Student's t with $1-\alpha/2=0.975$ and degrees of freedom $df=$length(x)-1. Here is a function called ci95 which uses our function se to compute 95% confidence intervals for a mean:

```
ci95<-function(x) {
  t.value<- qt(0.975,length(x)-1)
  standard.error<-se(x)
  ci<-t.value*standard.error
  cat("95% Confidence Interval = ", mean(x) -ci, "to ", mean(x) +ci,"\n") }
```

We can test the function with 150 normally distributed random numbers with mean 25 and standard deviation 3:

```
x<-rnorm(150,25,3)
ci95(x)
```

```
95% Confidence Interval = 24.76245 to 25.74469
```

If we were to repeat the experiment, we can be 95% certain that the mean of the new sample would lie between 24.76 and 25.74.

We can use the se function to investigate how the standard error of the mean changes with the sample size. First we generate one set of data from which we shall take progressively larger samples:

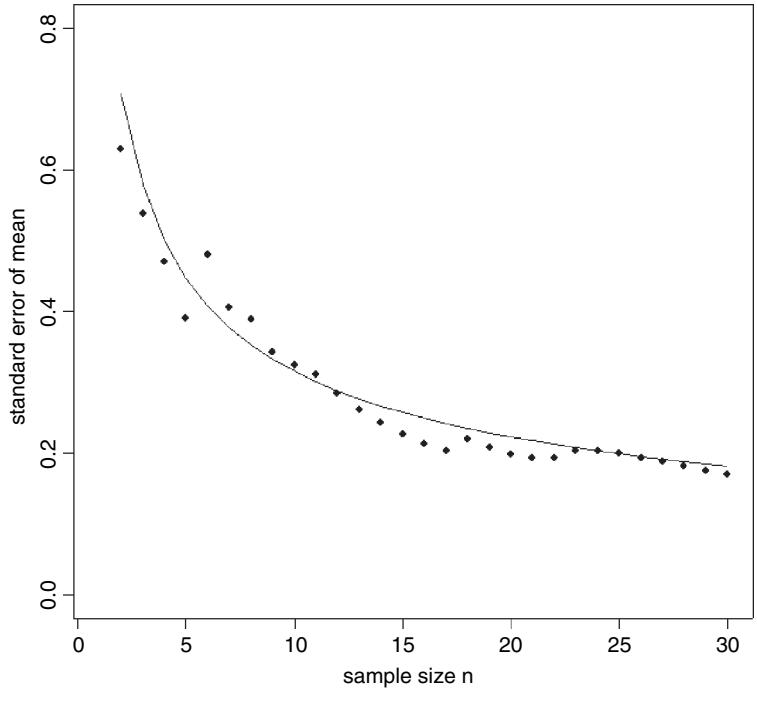
```
xv<-rnorm(30)
```

Now in a loop take samples of size 2, 3, 4, . . . , 30:

```
sem<-numeric(30)
sem[1]<-NA
for(i in 2:30) sem[i]<-se(xv[1:i])
plot(1:30,sem,ylim=c(0,0.8),
  ylab="standard error of mean",xlab="sample size n",pch=16)
```

You can see clearly that as the sample size falls below about $n=15$, so the standard error of the mean increases rapidly. The blips in the line are caused by outlying values being included in the calculations of the standard error with increases in sample size. The smooth curve is easy to compute: since the values in xv came from a standard normal distribution with mean 0 and standard deviation 1, so the average curve would be $1/\sqrt{n}$ which we can add to our graph using lines:

```
lines(2:30,1/sqrt(2:30))
```



You can see that our single simulation captured the essence of the shape but was wrong in detail, especially for the samples with the lowest replication. However, our single sample was reasonably good for $n > 24$.

Error Bars

There is no function in the base package of R for drawing error bars on bar charts, although several contributed packages use the arrows function for this purpose (p. 147). Here is a simple, stripped down function that is supplied with three arguments: the heights of the bars (*yv*), the lengths (up and down) of the error bars (*z*) and the labels for the bars on the x axis (*nn*):

```
error.bars<-function(yv,z,nn){
xv<-
barplot(yv,ylim=c(0,(max(yv)+max(z))),names=nn,ylab=deparse(substitute(yv)
))
g=(max(xv)-min(xv))/50
for (i in 1:length(xv)) {
lines(c(xv[i],xv[i]),c(yv[i]+z[i],yv[i]-z[i]))
lines(c(xv[i]-g,xv[i]+g),c(yv[i]+z[i], yv[i]+z[i]))
lines(c(xv[i]-g,xv[i]+g),c(yv[i]-z[i], yv[i]-z[i]))
}}
```

Here is the error.bars function in action with the plant competition data (p. 370):

```
comp<-read.table("c:\\temp\\competition.txt",header=T)
attach(comp)
```

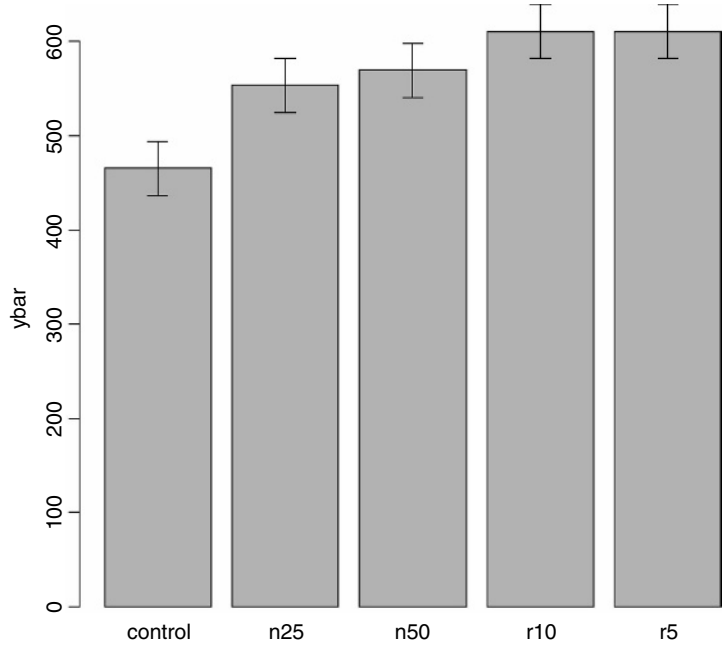
```
names(comp)
```

```
[1] "biomass" "clipping"
```

```
se<-rep(28.75,5)
labels<-as.character(levels(clipping))
ybar<-as.vector(tapply(biomass,clipping,mean))
```

Now the invoke the function with the means, standard errors and bar labels:

```
error.bars(ybar,se,labels)
```

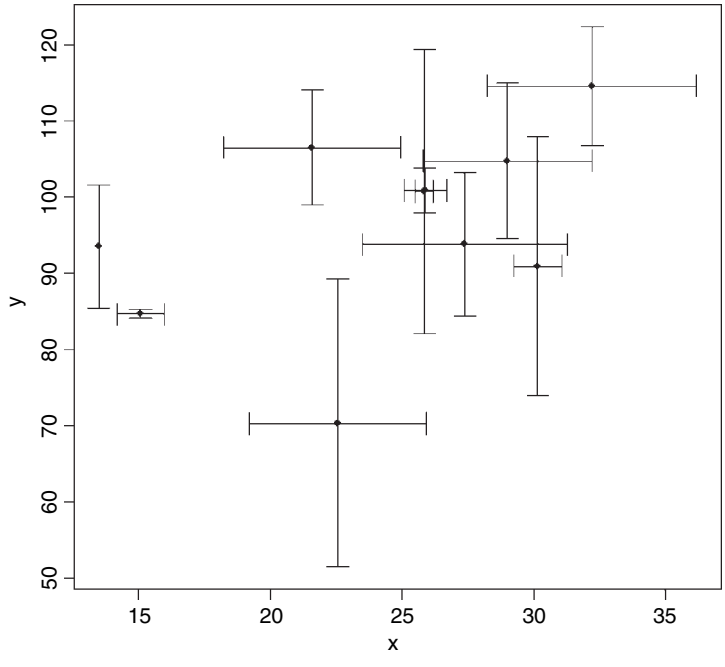


Here is a function to plot error bars on a scatterplot in both the *x* and *y* directions:

```
xy.error.bars<-function (x,y,xbar,ybar){
   plot(x, y, pch=16, ylim=c(min(y-ybar),max(y+ybar)),
        xlim=c(min(x-xbar),max(x+xbar)))
   arrows(x, y-ybar, x, y+ybar, code=3, angle=90, length=0.1)
   arrows(x-xbar, y, x+xbar, y, code=3, angle=90, length=0.1) }
```

We test it with these data:

```
x <- rnorm(10,25,5)
y <- rnorm(10,100,20)
xb <- runif(10)*5
yb <- runif(10)*20
xy.error.bars(x,y,xb,yb)
```



Loops and Repeats

The classic, Fortran-like loop is available in R. The syntax is a little different, but the idea is identical; you request that an index, *i*, takes on a sequence of values, and that one or more lines of commands are executed as many times as there are different values of *i*. Here is a loop executed five times with the values of *i* from 1 to 5: we print the square of each value:

```
for (i in 1:5) print(i^2)
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

For multiple lines of code, you use curly brackets {} to enclose material over which the loop is to work. Note that the 'hard return' (the Enter key) at the end of each command line is an essential part of the structure (you can replace the hard returns by semicolons if you like, but clarity is improved if you put each command on a separate line):

```
j<-k<-0
for (i in 1:5) {
j<-j+1
k<-k+i*j
print(i+j+k) }
```

```
[1] 3
[1] 9
```

```
[1]  20
[1]  38
[1]  65
```

Here we use a for loop to write a function to calculate factorial x (written $x!$) which is

$$x! = x \times (x-1) \times (x-2) \times (x-3) \ldots \times 2 \times 1$$

So $4! = 4 \times 3 \times 2 = 24$. Here is the function:

```
fac1<-function(x) {
      f <- 1
      if (x<2) return (1)
      for (i in 2:x) {
      f <- f*i
      f }}
```

That seems rather complicated for such a simple task, but we can try it out for the numbers 0 to 5:

```
sapply(0:5,fac1)
```

```
[1] 1  1  2  6  24  120
```

There are two other looping functions in R: repeat and while. We demonstrate their use for the purpose of illustration, but we can do much better in terms of writing a compact function for finding factorials (see below). First, the while function:

```
fac2<-function(x) {
f <- 1
t <- x
while(t>1) {
f <- f*t
t <- t-1 }
return(f) }
```

The key point is that if you want to use while, you need to set up an indicator variable (t in this case) and change its value *within* each iteration (t<-t-1). We test the function on the numbers 0 to 5:

```
sapply(0:5,fac2)
```

```
[1] 1 1 2 6 24 120
```

Finally, we demonstrate the use of the repeat function:

```
fac3<-function(x) {
f <- 1
t <- x
repeat {
if (t<2) break
f <- f*t
t <- t-1  }
return(f)  }
```

Because the repeat function contains no explicit limit, you need to be careful not to program an infinite loop. You must have a logical escape clause that leads to a break command:

```
sapply(0:5,fac3)
```

```
[1] 1 1 2 6 24 120
```

It is almost always better to use a built-in function that operates on the entire vector and hence removes the need for loops or repeats of any sort. In this case, we can make use of the cumulative product function, cumprod. Here it is in action:

```
cumprod(1:5)
```

```
[1] 1 2 6 24 120
```

This is already pretty close to what we need for our factorial function. It does not work for 0! of course, because the whole vector would end up full of zeros if the first element in the vector was zero (try 0:5 and see). The factorial of $x > 0$ is the maximum value from the vector produced by cumprod:

```
fac4<-function(x) max(cumprod(1:x))
```

This definition has the desirable side effect that it also gets 0! correct, because when x is 0 the function finds the maximum of 1 and 0 which is 1 which is 0!.

```
max(cumprod(1:0))
```

```
[1] 1
```

```
sapply(0:5,fac4)
```

```
[1] 1 1 2 6 24 120
```

Alternatively, you could adapt an existing built-in function to do the job. $x!$ is the same as $\Gamma(x+1)$, so

```
fac5<-function(x) gamma(x+1)
sapply(0:5,fac5)
```

```
[1] 1 1 2 6 24 120
```

Until recently there was no built-in factorial function in R, but now there is:

```
sapply(0:5,factorial)
```

```
[1] 1 1 2 6 24 120
```

Here is a function that uses the while function in converting a specified number to its binary representation. The trick is that the smallest digit (0 for even or 1 for odd numbers) is always at the right-hand side of the answer (in location 32 in this case):

```
binary<-function(x) {
  i<-0
  string<-numeric(32)
  while(x>0) {
    string[32-i]<-x %% 2
    x<-x%/% 2
    i<-i+1 }
```

```
first<-match(1,string)
string[first:32] }
```

The leading zeros (1 to first − 1) within the string are not printed. We run the function to find the binary representation of the numbers 15 to 17:

```
sapply(15:17,binary)
```

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

[[2]]
[1]  1  0  0  0  0

[[3]]
[1]  1  0  0  0  1
```

The next function uses while to generate the Fibonacci series 1, 1, 2, 3, 5, 8, . . . in which each term is the sum of its two predecessors. The key point about while loops is that the logical variable controlling their operation is altered inside the loop. In this example, we alter n, the number whose Fibonacci number we want, starting at n, reducing the value of n by 1 each time around the loop, and ending when n gets down to 0. Here is the code:

```
fibonacci<-function(n) {
  a<-1
  b<-0
  while(n>0)
    {swap<-a
    a<-a+b
    b<-swap
    n<-n-1 }
  b }
```

An important general point about computing involves the use of the swap variable above. When we replace a by $a + b$ on line 6 we lose the original value of a. If we had not stored this value in swap, we could not set the new value of b to the old value of a on line 7. Now test the function by generating the Fibonacci numbers 1 to 10:

```
sapply(1:10,fibonacci)
```

```
[1] 1 1 2 3 5 8 13 21 34 55
```

Loop avoidance

It is good R programming practice to avoid using loops wherever possible. The use of vector functions (p. 17) makes this particularly straightforward in many cases. Suppose that you wanted to replace all of the negative values in an array by zeros. In the old days, you might have written something like this:

```
for (i in 1:length(y)) { if(y[i] < 0) y[i] <- 0 }
```

Now, however, you would use logical subscripts (p. 21) like this:

```
y [y < 0] <- 0
```

The ifelse function

Sometimes you want to do one thing if a condition is true and a different thing if the condition is false (rather than do nothing, as in the last example). The ifelse function allows you to do this for entire vectors without using for loops. We might want to replace any negative values of y by -1 and any positive values and zero by $+1$:

```
z <- ifelse (y < 0, -1, 1)
```

Here we use ifelse to convert the continuous variable called Area into a new, two-level factor with values 'big' and 'small' defined by the median Area of the fields:

```
data<-read.table("c:\\temp\\worms.txt",header=T)
attach(data)
ifelse(Area>median(Area),"big","small")
```

```
 [1]   "big"   "big" "small" "small"   "big"  "big"  "big" "small" "small"
[10] "small" "small"   "big"   "big" "small"  "big"  "big" "small"   "big"
[19] "small" "small"
```

You should use the much more powerful function called cut when you want to convert a continuous variable like Area into many levels (p. 241).

Another use of ifelse is to override R's natural inclinations. The log of zero in R is `-Inf`, as you see in these 20 random numbers from a Poisson process with a mean count of 1.5:

```
y<-log(rpois(20,1.5))
y
```

```
 [1] 0.0000000 1.0986123 1.0986123 0.6931472 0.0000000 0.6931472 0.6931472
 [8] 0.0000000 0.0000000 0.0000000 0.0000000      -Inf      -Inf      -Inf
[15] 1.3862944 0.6931472 1.6094379      -Inf      -Inf 0.0000000
```

However, we want the log of zero to be represented by NA in our particular application:

```
ifelse(y<0,NA,y)
```

```
 [1] 0.0000000 1.0986123 1.0986123 0.6931472 0.0000000 0.6931472 0.6931472
 [8] 0.0000000 0.0000000 0.0000000 0.0000000        NA        NA        NA
[15] 1.3862944 0.6931472 1.6094379        NA        NA 0.0000000
```

The slowness of loops

To see how slow loops can be, we compare two ways of finding the maximum number in a vector of 10 million random numbers from a uniform distribution:

```
x<-runif(10000000)
```

First, using the vector function max:

```
system.time(max(x))
```

```
[1] 0.13 0.00 0.12   NA   NA
```

As you see, this operation took just over one-tenth of a second (0.12) to solve using the vector function max to look at the 10 million numbers in x. Using a loop, however, took more than 15 seconds:

```
pc<-proc.time()
cmax<-x[1]
```

```
for (i in 2:10000000) {
if(x[i]>cmax) cmax<-x[i] }
proc.time()-pc
```

```
[1]  15.52   0.00  15.89    NA    NA
```

The functions system.time and proc.time produce a vector of five numbers, showing the user, system and total elapsed times for the currently running R process, and the cumulative sum of user (subproc1) and system times (subproc2) of any child processes spawned by it (none in this case, so NA). It is the third number (elapsed time in seconds) that is typically the most useful.

Do not 'grow' data sets in loops or recursive function calls

Here is an extreme example of what *not* to do. We want to generate a vector containing the integers 1 to 1 000 000:

```
z<-NULL
for (i in 1:1000000){
z<-c(z,i) }
```

This took a ridiculous 4 hours 14 minutes to execute. The moral is clear: do not use concatenation c(z,i) to generate iterative arrays. The simple way to do it,

```
z<-1:1000000
```

took 0.05 seconds to accomplish.

The switch Function

When you want a function to do different things in different circumstances, then the switch function can be useful. Here we write a function that can calculate any one of four different measures of central tendency: arithmetic mean, geometric mean, harmonic mean or median (p. 51). The character variable called measure should take one value of Mean, Geometric, Harmonic or Median; any other text will lead to the error message `Measure not included`. Alternatively, you can specify the number of the switch (e.g. 1 for Mean, 4 for Median).

```
central<-function(y, measure) {
  switch(measure,
    Mean = mean(y),
    Geometric = exp(mean(log(y))),
    Harmonic = 1/mean(1/y),
    Median = median(y),
  stop("Measure not included")) }
```

Note that you have to include the character strings in quotes as arguments to the function, but they must not be in quotes within the switch function itself.

```
central(rnorm(100,10,2),"Harmonic")
```

```
[1] 9.554712
```

```
central(rnorm(100,10,2),4)
```

```
[1] 10.46240
```

The Evaluation Environment of a Function

When a function is called or invoked a new *evaluation frame* is created. In this frame the formal arguments are matched with the supplied arguments according to the rules of **argument matching** (below). The statements in the body of the function are evaluated sequentially in this environment frame.

The first thing that occurs in a function evaluation is the matching of the formal to the actual or supplied arguments. This is done by a three-pass process:

- **Exact matching on tags**. For each named supplied argument the list of formal arguments is searched for an item whose name matches exactly.
- **Partial matching on tags**. Each named supplied argument is compared to the remaining formal arguments using partial matching. If the name of the supplied argument matches exactly with the first part of a formal argument then the two arguments are considered to be matched.
- **Positional matching**. Any unmatched formal arguments are bound to unnamed supplied arguments, in order. If there is a . . . argument, it will take up the remaining arguments, tagged or not.
- If any arguments remain unmatched an error is declared.

Supplied arguments and default arguments are treated differently. The supplied arguments to a function are evaluated in the evaluation frame of the calling function. The default arguments to a function are evaluated in the evaluation frame of the function. In general, supplied arguments behave as if they are local variables initialized with the value supplied and the name of the corresponding formal argument. Changing the value of a supplied argument within a function will not affect the value of the variable in the calling frame.

Scope

The **scoping rules** are the set of rules used by the evaluator to find a value for a symbol. A symbol can be either **bound** or **unbound**. All of the formal arguments to a function provide bound symbols in the body of the function. Any other symbols in the body of the function are either local variables or unbound variables. A local variable is one that is defined within the function, typically by having it on the left-hand side of an assignment. During the evaluation process if an unbound symbol is detected then R attempts to find a value for it: the environment of the function is searched first, then its enclosure and so on until the global environment is reached. The value of the first match is then used.

Optional Arguments

Here is a function called charplot that produces a scatterplot of x and y using solid red circles as the plotting symbols: there are two essential arguments (x and y) and two optional (*pc* and *co*) to control selection of the plotting symbol and its colour:

```
charplot<-function(x,y,pc=16,co="red"){
plot(y~x,pch=pc,col=co)}
```

The optional arguments are given their default values using = in the argument list. To execute the function you need only provide the vectors of x and y,

```
charplot(1:10,1:10)
```

to get solid red circles. You can get a different plotting symbol simply by adding a third argument

```
charplot(1:10,1:10,17)
```

which produces red solid triangles (pch=17). If you want to change only the colour (the fourth argument) then you have to specify the variable name because the optional arguments would not then be presented in sequence. So, for navy-coloured solid circles, you put

```
charplot(1:10,1:10,co="navy")
```

To change both the plotting symbol and the colour you do not need to specify the variable names, so long as the plotting symbol is the third argument and the colour is the fourth

```
charplot(1:10,1:10,15,"green")
```

which produces solid green squares. Reversing the optional arguments does not work

```
charplot(1:10,1:10,"green",15)
```

(this uses the letter g as the plotting symbol and colour no. 15). If you specify both variable names, then the order does not matter:

```
charplot(1:10,1:10,co="green",pc=15)
```

This produces solid green squares despite the arguments being out of sequence.

Variable Numbers of Arguments (. . .)

Some applications are much more straightforward if the number of arguments does not need to be specified in advance. There is a special formal name . . . (triple dot) which is used in the argument list to specify that an arbitrary number of arguments are to be passed to the function. Here is a function that takes any number of vectors and calculates their means and variances:

```
many.means <- function (. . . ) {
  data <- list(. . . )
  n<- length(data)
  means <- numeric(n)
  vars <- numeric(n)
  for (i in 1:n) {
    means[i]<-mean(data[[i]])
    vars[i]<-var(data[[i]])
  }
  print(means)
  print(vars)
  invisible(NULL)
}
```

The main features to note are these. The function definition has . . . as its only argument. The 'triple dot' argument . . . allows the function to accept additional arguments of unspecified name and number, and this introduces tremendous flexibility into the structure and behaviour of functions. The first thing done inside the function is to create an object called data out of the list of vectors that are actually supplied in any particular case. The length of this list is the number of vectors, not the lengths of the vectors themselves (these could differ from one vector to another, as in the example below). Then the two output variables (means and vars) are defined to have as many elements as there are vectors in the parameter list. The loop goes from 1 to the number of vectors, and for each vector uses the built-in functions mean and var to compute the answers we require. It is important to note that because data is a list, we use double [[ ]] subscripts in addressing its elements.

Now try it out. To make things difficult we shall give it three vectors of different lengths. All come from the standard normal distribution (with mean 0 and variance 1) but x is 100 in length, y is 200 and z is 300 numbers long.

```
x<-rnorm(100)
y<-rnorm(200)
z<-rnorm(300)
```

Now we invoke the function:

```
many.means(x,y,z)
```

```
[1]  -0.039181830  0.003613744  0.050997841
[1]       1.146587     0.989700     0.999505
```

As expected, all three means (top row) are close to 0 and all 3 variances are close to 1 (bottom row). You can use . . . to absorb some arguments into an intermediate function which can then be extracted by functions called subsequently. R has a form of *lazy evaluation* of function arguments in which arguments are not evaluated until they are needed (in some cases the argument will never be evaluated).

Returning Values from a Function

Often you want a function to return a single value (like a mean or a maximum), in which case you simply leave the last line of the function unassigned (i.e. there is no 'gets arrow' on the last line). Here is a function to return the median value of the parallel maxima (built-in function pmax) of two vectors supplied as arguments:

```
parmax<-function (a,b) {
c<-pmax(a,b)
median(c) }
```

Here is the function in action: the unassigned last line median(c) returns the answer

```
x<-c(1,9,2,8,3,7)
y<-c(9,2,8,3,7,2)
parmax(x,y)
```

```
[1] 8
```

If you want to return two or more variables from a function you should use return with a list containing the variables to be returned. Suppose we wanted the median value of both the parallel maxima and the parallel minima to be returned:

```
parboth<-function (a,b) {
c<-pmax(a,b)
d<-pmin(a,b)
answer<-list(median(c),median(d))
names(answer)[[1]]<-"median of the parallel maxima"
names(answer)[[2]]<-"median of the parallel minima"
return(answer) }
```

Here it is in action with the same x and y data as above:

```
parboth(x,y)
```

```
$"median of the parallel maxima"
[1] 8

$"median of the parallel minima"
[1] 2
```

The point is that you make the multiple returns into a list, then return the list. The provision of multi-argument returns (e.g. return(median(c),median(d)) in the example above) has been deprecated in R and a warning is given, as multi-argument returns were never documented in S, and whether or not the list was named differs from one version of S to another.

Anonymous Functions

Here is an example of an anonymous function. It generates a vector of values but the function is not allocated a name (although the answer could be).

```
(function(x,y){ z <- 2*x^2 + y^2; x+y+z })(0:7, 1)
```

```
[1]  2  5  12  23  38  57  80  107
```

The function first uses the supplied values of x and y to calculate z, then returns the value of $x + y + z$ evaluated for eight values of x (from 0 to 7) and one value of y (1). Anonymous functions are used most frequently with apply, sapply and lapply (p. 68).

Flexible Handling of Arguments to Functions

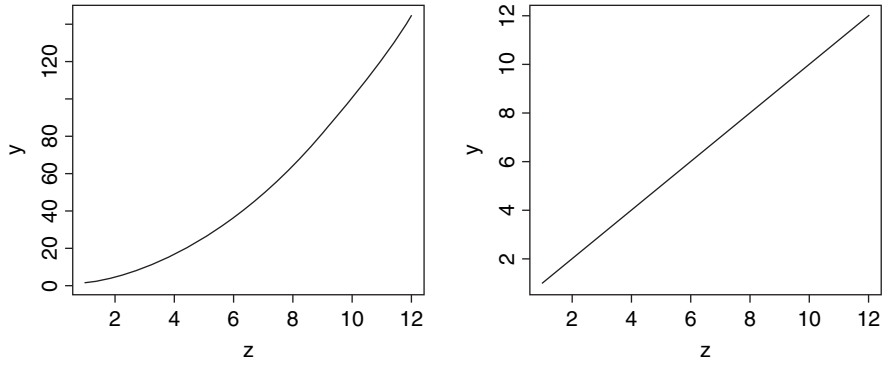
Because of the **lazy evaluation** practised by R, it is very simple to deal with missing arguments in function calls, giving the user the opportunity to specify the absolute minimum number of arguments, but to override the default arguments if they want to. As a simple example, take a function plotx2 that we want to work when provided with either one or two arguments. In the one-argument case (only an integer $x > 1$ provided), we want it to plot z^2 against z for $z = 1$ to x in steps of 1. In the second case, when y is supplied, we want it to plot y against z for $z = 1$ to x.

```
plotx2 <- function (x, y=z^2) {
  z<-1:x
  plot(z,y,type="l") }
```

In many other languages, the first line would fail because z is not defined at this point. But R does not evaluate an expression until the body of the function actually calls for it to be

evaluated (i.e. never, in the case where y is supplied as a second argument). Thus for the one-argument case we get a graph of z^2 against z and in the two-argument case we get a graph of y against z (in this example, the straight line 1:12 vs. 1:12)

```
par(mfrow=c(1,2))
plotx2(12)
plotx2(12,1:12)
```



You need to specify that the type of plot you want is a line (type="l" using lower-case L, not upper-case I and not number 1) because the default is to produce a scatterplot with open circles as the plotting symbol (type="p"). If you want your plot to consist of points with lines joining the dots, then use type="b" (for 'both' lines and points). Other types of plot that you might want to specify include vertical lines from the x axis up to the value of the response variable (type="h"), creating an effect like very slim barplots or histograms, and type="s" to produce a line drawn as steps between successive ranked values of x. To plot the scaled axes, but no lines or points, use type="n" (see p. 137).

It is possible to access the actual (not default) expressions used as arguments inside the function. The mechanism is implemented via promises. You can find an explanation of promises by typing ?promise at the command prompt.

Evaluating Functions with apply, sapply and lapply

apply and sapply

The apply function is used for applying functions to the rows or columns of matrices or dataframes. For example:

```
(X<-matrix(1:24,nrow=4))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

Note that placing the expression to be evaluated in parentheses (as above) causes the value of the result to be printed on the screen. This saves an extra line of code, because to achieve the same result without parentheses requires us to type

```
X<-matrix(1:24,nrow=4)
X
```

Often you want to apply a function across one of the margins of a matrix – margin 1 being the rows and margin 2 the columns. Here are the row totals (four of them):

```
apply(X,1,sum)
```

```
[1]  66  72  78  84
```

and here are the column totals (six of them):

```
apply(X,2,sum)
```

```
[1]  10  26  42  58  74  90
```

Note that in both cases, the answer produced by apply is a vector rather than a matrix. You can apply functions to the individual elements of the matrix rather than to the margins. The margin you specify influences only the shape of the resulting matrix.

```
apply(X,1,sqrt)
```

```
         [,1]     [,2]     [,3]     [,4]
[1,] 1.000000 1.414214 1.732051 2.000000
[2,] 2.236068 2.449490 2.645751 2.828427
[3,] 3.000000 3.162278 3.316625 3.464102
[4,] 3.605551 3.741657 3.872983 4.000000
[5,] 4.123106 4.242641 4.358899 4.472136
[6,] 4.582576 4.690416 4.795832 4.898979
```

```
apply(X,2,sqrt)
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
[1,] 1.000000 2.236068 3.000000 3.605551 4.123106 4.582576
[2,] 1.414214 2.449490 3.162278 3.741657 4.242641 4.690416
[3,] 1.732051 2.645751 3.316625 3.872983 4.358899 4.795832
[4,] 2.000000 2.828427 3.464102 4.000000 4.472136 4.898979
```

Here are the shuffled numbers from each of the rows, using sample without replacement:

```
apply(X,1,sample)
```

```
     [,1] [,2] [,3] [,4]
[1,]    5   14   19    8
[2,]   21   10    7   16
[3,]   17   18   15   24
[4,]    1   22   23    4
[5,]    9    2    3   12
[6,]   13    6   11   20
```

Note that the resulting matrix has 6 rows and 4 columns (i.e. it has been transposed).

You can supply your own function definition within apply like this:

```
apply(X,1,function(x) x^2+x)
```

```
      [,1]  [,2]  [,3]  [,4]
[1,]     2     6    12    20
[2,]    30    42    56    72
[3,]    90   110   132   156
[4,]   182   210   240   272
[5,]   306   342   380   420
[6,]   462   506   552   600
```

This is an anonymous function because the function is not named.

If you want to apply a function to a vector then use sapply (rather than apply for matrices or margins of matrices). Here is the code to generate a list of sequences from 1:3 up to 1:7 (see p. 30):

```
sapply(3:7, seq)
```

```
[[1]]
[1] 1 2 3

[[2]]
[1] 1 2 3 4

[[3]]
[1] 1 2 3 4 5

[[4]]
[1] 1 2 3 4 5 6

[[5]]
[1] 1 2 3 4 5 6 7
```

The function sapply is most useful with complex iterative calculations. The following data show decay of radioactive emissions over a 50-day period, and we intend to use non-linear least squares (see p. 663) to estimate the decay rate a in $y = \exp(-ax)$:

```
sapdecay<-read.table("c:\\temp\\sapdecay.txt",header=T)
attach(sapdecay)
names(sapdecay)
```

```
[1] "x" "y"
```

We need to write a function to calculate the sum of the squares of the differences between the observed (y) and predicted (yf) values of y, when provided with a specific value of the parameter a:

```
sumsq <- function(a,xv=x,yv=y)
  { yf <- exp(-a*xv)
  sum((yv-yf)^2) }
```

We can get a rough idea of the decay constant, a, for these data by linear regression of $\log(y)$ against x, like this:

```
lm(log(y)~x)
```

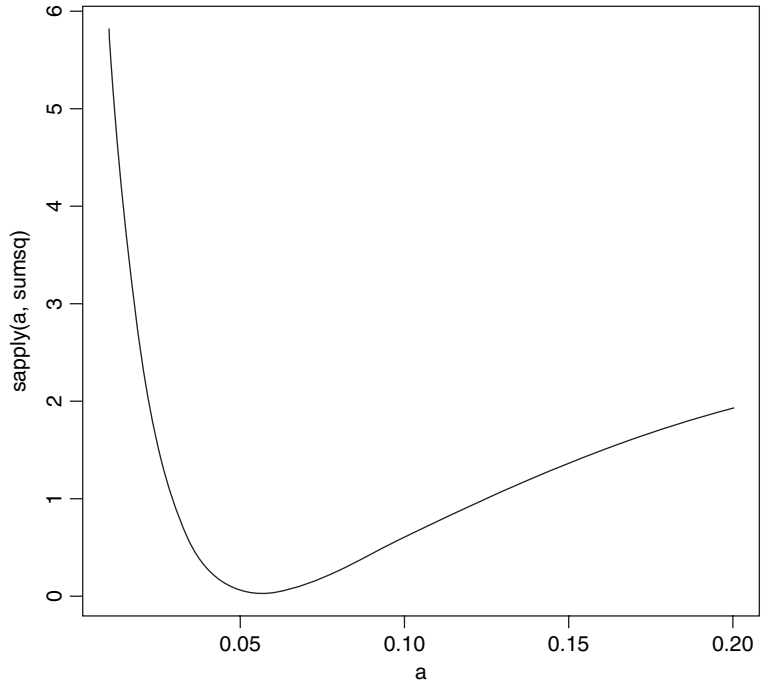
```
Coefficients:
(Intercept)           x
    0.04688     -0.05849
```

So our parameter a is somewhere close to 0.058. We generate a range of values for a spanning an interval on either side of 0.058:

```
a<-seq(0.01,0.2,.005)
```

Now we can use sapply to apply the sum of squares function for each of these values of a (without writing a loop), and plot the deviance against the parameter value for a:

```
plot(a,sapply(a,sumsq),type="l")
```



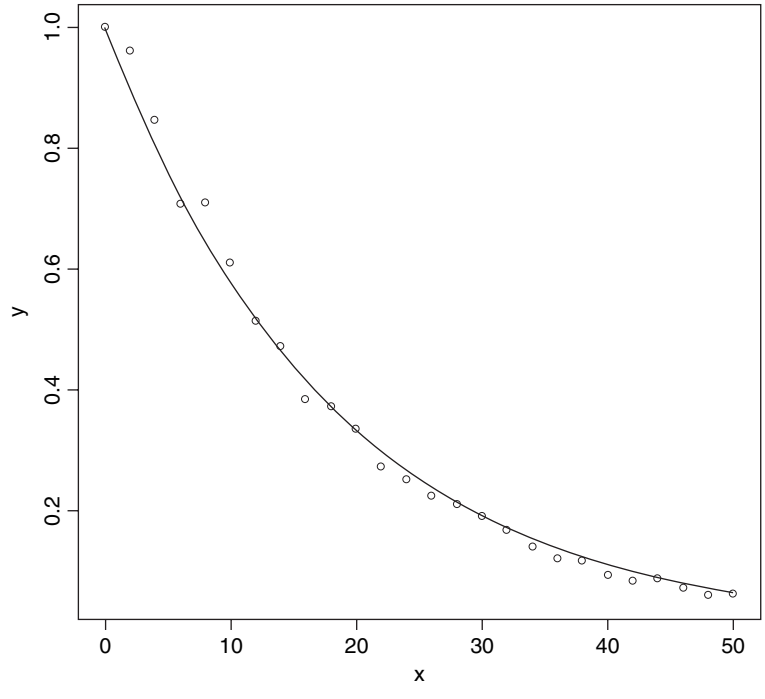
This shows that the least-squares estimate of a is indeed close to 0.06 (this is the value of a associated with the minimum deviance). To extract the minimum value of a we use min with subscripts (square brackets) to extract the relevant value of a:

```
a[min(sapply(a,sumsq))==sapply(a,sumsq)]
```

```
[1] 0.055
```

Finally, we could use this value of a to generate a smooth exponential function to fit through our scatter of data points:

```
plot(x,y)
xv<-seq(0,50,0.1)
lines(xv,exp(-0.055*xv))
```



Here is the same procedure streamlined by using the optimize function. Write a function showing how the sum of squares depends on the value of the parameter a:

```
fa<-function(a) sum((y-exp(-a*x))^2)
```

Now use optimize with a specified range of values for a, here c(0.01,0.1), to find the value of a that minimizes the sum of squares:

```
optimize(fa,c(0.01,0.1))
```

```
$minimum
[1] 0.05538411

$objective
[1] 0.01473559
```

The value of a is that minimizes the sum of squares is 0.055 38 and the minimum value of the sum of squares is 0.0147. What if we had chosen a different way of assessing the fit of the model to the data? Instead of minimizing the sum of the squares of the residuals, we might want to minimize the sum of the absolute values of the residuals. We need to write a new function to calculate this quantity,

```
fb<-function(a) sum(abs(y-exp(-a*x)))
```

then use optimize as before:

```
optimize(fb,c(0.01,0.1))
```

```
$minimum
[1]  0.05596058
```

```
$objective
[1]  0.3939221
```

The results differ only in the fourth digit after the decimal point, and you could not choose between the two methods from a plot of the model.

Lists and lapply

We start by creating a list object that consists of three parts: character information in a vector called *a*, numeric information in a vector called *b*, and logical information in a vector called *c*:

```
a<-c("a","b","c","d")
b<-c(1,2,3,4,4,3,2,1)
c<-c(T,T,F)
```

We create our list object by using the list function to bundle these variables together:

```
list.object<-list(a,b,c)
class(list.object)
```

```
[1] "list"
```

To see the contents of the list we just type its name:

```
list.object
```

```
[[1]]
[1]  "a"  "b"  "c"  "d"

[[2]]
[1]  1  2  3  4  4  3  2  1

[[3]]
[1]  TRUE  TRUE  FALSE
```

The function lapply applies a specified function to each of the elements of a list in turn (without the need for specifying a loop, and not requiring us to know how many elements there are in the list). A useful function to apply to lists is the length function; this asks the question how many elements comprise each component of the list. Technically we want to know the length of each of the vectors making up the list:

```
lapply(list.object,length)
```

```
[[1]]
[1]  4

[[2]]
[1]  8

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

This shows that list.object consists of three vectors ([[1]], [[2]] and [[3]]), and shows that there were four elements in the first vector, eight in the second and three in the third. But four of what, and eight of what? To find out, we apply the function class to the list:

lapply(list.object,class)

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

[[2]]
[1]  "numeric"

[[3]]
[1]  "logical"
```

So the answer is there were 4 characters in the first vector, 8 numbers in the second and 3 logical values in the third vector.

Applying numeric functions to lists will only work for objects of class numeric or objects (like logical values) that can be coerced into numbers. Here is what happens when we use lapply to apply the function mean to list.object:

lapply(list.object,mean)

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

[[2]]
[1]  2.5

[[3]]
[1]  0.6666667

Warning message:
argument is not numeric or logical: returning NA in:
mean.default(X[[1]], ...)
```

We get a warning message pointing out that the first vector cannot be coerced to a number (it is not numeric or logical), so NA appears in the output. The second vector is averaged as expected. The third vector produces the answer 2/3 because logical false (F) is coerced to numeric 0 and logical true (T) is coerced to numeric 1.

Looking for runs of numbers within vectors

The function is called rle, which stands for 'run length encoding' and is most easily understood with an example. Here is a vector of 150 random numbers from a Poisson distribution with mean 0.7:

(poisson<-rpois(150,0.7))

```
  [1]  1  1  0  0  2  1  0  1  0  1  0  0  0  0  2  1  0  0  3  1
       0  0  1  0  2  0  1  1  0  0  0  1  0  0  0  2  1
 [38]  0  0  0  1  0  0  0  2  0  0  0  1  1  0  2  1  0  0  0  2
       0  0  2  3  2  1  0  2  0  0  0  0  0  1  1  0  0
 [75]  0  0  0  1  1  1  0  0  1  0  1  2  2  0  0  2  0  0  0  0
       0  0  0  2  1  0  0  1  0  1  0  1  1  1  2  0  3
[112]  0  0  2  0  0  1  0  1  0  4  0  0  1  0  2  1  0  1  1  0
       0  1  3  3  0  0  1  1  0  1  0  0  0  0  0  1  0
[149]  2  0
```

We can do our own run length encoding on the vector by eye: there is a run of two 1s, then a run of two 0s, then a single 2, then a single 1, then a single 0, and so on. So the run lengths are 2, 2, 1, 1, 1, 1,. . . . The values associated with these runs were 1, 0, 2, 1, 0, 1,. . . . Here is the output from rle:

```
rle(poisson)
```

```
Run Length Encoding
lengths: int [1:93] 2 2 1 1 1 1 1 1 4 1...
values :  num [1:93] 1 0 2 1 0 1 0 1 0 2...
```

The object produced by rle is a list of two vectors: the lengths and the values. To find the longest run, and the value associated with that longest run, we use the indexed lists like this:

```
max(rle(poisson)[[1]])
```

```
[1] 7
```

So the longest run in this vector of numbers was 7. But 7 of what? We use which to find the location of the 7 in `lengths,` then apply this index to `values` to find the answer:

```
which(rle(poisson)[[1]]==7)
```

```
[1]  55
```

```
rle(poisson)[[2]][55]
```

```
[1]  0
```

So, not surprisingly given that the mean was just 0.7, the longest run was of zeros.

Here is a function to return the length of the run and its value for any vector:

```
run.and.value<-function (x) {
a<- max(rle(poisson)[[1]])
b<-rle(poisson)[[2]][which(rle(poisson)[[1]] == a)]
cat("length = ",a," value = ",b, "\n")}
```

Testing the function on the vector of 150 Poisson data gives

```
run.and.value(poisson)
```

```
length = 7  value = 0
```

It is sometimes of interest to know the number of runs in a given vector (for instance, the lower the number of runs, the more aggregated the numbers; and the greater the number of runs, the more regularly spaced out). We use the length function for this:

```
length(rle(poisson)[[2]])
```

```
[1] 93
```

indicating that the 150 values were arranged in 93 runs (an intermediate value, characteristic of a random pattern). The value 93 appears in square brackets `[1:93]` in the output of the run length encoding function.

In a different example, suppose we had n_1 values of 1 representing 'present' and n_2 values of 0 representing 'absent', then the minimum number of runs would be 2 (a solid block of 1s and a sold block of 0s). The maximum number of runs would be $2n + 1$ if they alternated (until the smaller number $n =$ min(n1,n2) ran out). Here is a simple **runs test** based on 1000 randomizations of 25 ones and 30 zeros:

```
n1<-25
n2<-30
y<-c(rep(1,n1),rep(0,n2))
len<-numeric(10000)
for (i in 1:10000) len[i]<-length(rle(sample(y))[[2]])
quantile(len,c(0.025,0.975))
```

```
2.5% 97.5%
  21    35
```

Thus, for these data ($n_1 = 25$ and $n_2 = 30$) an aggregated pattern would score 21 or fewer runs, and a regular pattern would score 35 or more runs. Any scores between 21 and 35 fall within the realm of random patterns.

Saving Data Produced within R to Disc

It is often convenient to generate numbers within R and then to use them somewhere else (in a spreadsheet, say). Here are 1000 random integers from a negative binomial distribution with mean mu= 1.2 and clumping parameter or aggregation parameter (k) size = 1.0, that I want to save as a single column of 1000 rows in a file called nbnumbers.txt in directory 'temp' on the c: drive:

```
nbnumbers<-rnbinom(1000, size=1, mu=1.2)
```

There is general point to note here about the number and order of arguments provided to built-in functions like rnbinom. This function can have two of three optional arguments: size, mean (mu) and probability (prob) (see ?rnbinom). R knows that the unlabelled number 1000 refers to the number of numbers required because of its position, first in the list of arguments. If you are prepared to specify the names of the arguments, then the order in which they appear is irrelevant: rnbinom(1000, size=1, mu=1.2) and rnbinom(1000, mu=1.2, size=1) would give the same output. If optional arguments are not labelled, then their order is crucial: so rnbinom(1000, 0.9, 0.6) is different from rnbinom(1000, 0.6, 0.9) because if there are no labels, then the second argument *must* be size and the third argument *must* be prob.

To export the numbers I use write like this, specifying that the numbers are to be output in a single column (i.e. with third argument 1 because the default is 5 columns):

```
write(nbnumbers,"c:\\temp\\nbnumbers.txt",1)
```

Sometimes you will want to save a table or a matrix of numbers to file. There is an issue here, in that the write function transposes rows and columns. It is much simpler to use the write.table function which does *not* transpose the rows and columns

```
xmat<-matrix(rpois(100000,0.75),nrow=1000)
write.table(xmat,"c:\\temp\\table.txt",col.names=F,row.names=F)
```

but it does add made-up row names and column names unless (as here) you specify otherwise. You have saved 1000 rows each of 100 Poisson random numbers with $\lambda = 0.75$.

Suppose that you have counted the number of different entries in the vector of negative binomial numbers (above):

```
nbtable<-table(nbnumbers)
nbtable
```

```
nbnumbers
  0    1    2   3   4   5   6  7  8  9  11  15
445  248  146  62  41  33  13  4  1  5   1   1
```

and you want write this output to a file. If you want to save both the counts and their frequencies in adjacent columns, use

```
write.table(nbtable,"c:\\temp\\table.txt",col.names=F,row.names=F)
```

but if you only want to export a single column of frequencies (445, 248, .. etc) use

```
write.table(unclass(nbtable),"c:\\temp\\table.txt",col.names=F,row.names=F)
```

Pasting into an Excel Spreadsheet

Writing a vector from R to the Windows clipboard uses the function `writeClipboard(x)` where `x` is a character vector, so you need to build up a spreadsheet in Excel one column at a time. Remember that character strings in dataframes are converted to factors on input unless you protect them by `as.is(name)` on input. For example

```
writeClipboard(as.character(factor.name))
```

Go into Excel and press Ctrl+V, and then back into R and type

```
writeClipboard(as.character(numeric.variable))
```

Then go into Excel and Ctrl+V in the second column, and so on.

Writing an Excel Readable File from R

Suppose you want to transfer the dataframe called `data` to Excel:

```
write.table(data,"clipboard",sep="\t",col.names=NA)
```

Then, in Excel, just type Ctrl+V or click on the Paste icon (the clipboard).

Testing for Equality

You need to be careful in programming when you want to test whether or not two computed numbers are equal. R will assume that you mean 'exactly equal', and what *that* means depends upon machine precision. Most numbers are rounded to 53 binary digits accuracy. Typically therefore, two floating point numbers will *not* reliably be equal unless they were computed by the same algorithm, and not always even then. You can see this by squaring the square root of 2: surely these values are the same?

```
x <- sqrt(2)
x * x == 2
```

```
[1] FALSE
```

We can see by how much the two values differ by subtraction:

```
x * x - 2
```

```
[1] 4.440892e-16
```

Sets: union, intersect and setdiff

There are three essential functions for manipulating sets. The principles are easy to see if we work with an example of two sets:

```
setA<-c("a", "b", "c", "d", "e")
setB<-c("d", "e", "f", "g")
```

Make a mental note of what the two sets have in common, and what is unique to each.

The **union** of two sets is everything in the two sets taken together, but counting elements only once that are common to both sets:

```
union(setA,setB)
```

```
[1] "a" "b" "c" "d" "e" "f" "g"
```

The **intersection** of two sets is the material that they have in common:

```
intersect(setA,setB)
```

```
[1] "d" "e"
```

Note, however, that the **difference** between two sets is order-dependent. It is the material that *is* in the first named set, that *is not* in the second named set. Thus setdiff(A,B) gives a different answer than setdiff(B,A). For our example,

```
setdiff(setA,setB)
```

```
[1] "a" "b" "c"
```

```
setdiff(setB,setA)
```

```
[1] "f" "g"
```

Thus, it should be the case that setdiff(setA,setB) plus intersect(setA,setB) plus setdiff(setB,setA) is the same as the union of the two sets. Let's check:

```
all(c(setdiff(setA,setB),intersect(setA,setB),setdiff(setB,setA))==
  union(setA,setB))
```

```
[1] TRUE
```

There is also a built-in function setequal for testing if two sets are equal

```
setequal(c(setdiff(setA,setB),intersect(setA,setB),setdiff(setB,setA)),
   union(setA,setB))
```

```
[1] TRUE
```

You can use %in% for comparing sets. The result is a logical vector whose length matches the vector on the left

```
setA %in% setB
```

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

```
setB %in% setA
```

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

Using these vectors of logical values as subscripts, we can demonstrate, for instance, that setA[setA %in% setB] is the same as intersect(setA,setB):

```
setA[setA %in% setB]
```

```
[1] "d" "e"
```

```
intersect(setA,setB)
```

```
[1] "d" "e"
```

Pattern Matching

We need a dataframe with a serious amount of text in it to make these exercises relevant:

```
wf<-read.table("c:\\temp\\worldfloras.txt",header=T)
attach(wf)
names(wf)
```

```
[1] "Country" "Latitude" "Area" "Population" "Flora"
[6] "Endemism" "Continent"
```

Country

As you can see, there are 161 countries in this dataframe (strictly, 161 places, since some of the entries, such as Sicily and Balearic Islands, are not countries). The idea is that we want to be able to select subsets of countries on the basis of specified patterns within the character strings that make up the country names (factor levels). The function to do this is grep. This searches for matches to a pattern (specified in its first argument) within the character vector which forms the second argument. It returns a vector of indices (subscripts) within the vector appearing as the second argument, where the pattern was found in whole or in part. The topic of pattern matching is very easy to master once the penny drops, but it hard to grasp without simple, concrete examples. Perhaps the simplest task is to select all the countries containing a particular letter – for instance, upper case R:

```
as.vector(Country[grep("R",as.character(Country))])
```

```
[1]  "Central African Republic"  "Costa Rica"
[3]  "Dominican Republic"        "Puerto Rico"
[5]  "Reunion"                   "Romania"
[7]  "Rwanda"                    "USSR"
```

To restrict the search to countries whose *first* name begins with R use the ^ character like this:

```
as.vector(Country[grep("^R",as.character(Country))])
```

```
[1] "Reunion" "Romania" "Rwanda"
```

To select those countries with multiple names with upper case R as the first letter of their second or subsequent names, we specify the character string as 'blank R' like this:

```
as.vector(Country[grep(" R",as.character(Country)])
```

```
[1]  "Central African Republic"  "Costa Rica"
[3]  "Dominican Republic"        "Puerto Rico"
```

To find all the countries with two or more names, just search for a blank " "

```
as.vector(Country[grep(" ",as.character(Country))])
```

```
 [1]  "Balearic Islands"          "Burkina Faso"
 [3]  "Central African Republic"  "Costa Rica"
 [5]  "Dominican Republic"        "El Salvador"
 [7]  "French Guiana"             "Germany East"
 [9]  "Germany West"              "Hong Kong"
[11]  "Ivory Coast"               "New Caledonia"
[13]  "New Zealand"               "Papua New Guinea"
[15]  "Puerto Rico"               "Saudi Arabia"
[17]  "Sierra Leone"              "Solomon Islands"
[19]  "South Africa"              "Sri Lanka"
[21]  "Trinidad & Tobago"         "Tristan da Cunha"
[23]  "United Kingdom"            "Viet Nam"
[25]  "Yemen North"               "Yemen South"
```

To find countries with names ending in 'y' use the $ (dollar) symbol like this:

```
as.vector(Country[grep("y$",as.character(Country))])
```

```
[1] "Hungary" "Italy" "Norway" "Paraguay" "Sicily" "Turkey"
[7] "Uruguay"
```

To recap: the start of the character string is denoted by ^ and the end of the character string is denoted by $. For conditions that can be expressed as groups (say, series of numbers or alphabetically grouped lists of letters), use square brackets inside the quotes to indicate the range of values that is to be selected. For instance, to select countries with names containing upper-case letters from C to E inclusive, write:

```
as.vector(Country[grep("[C-E]",as.character(Country))])
```

```
 [1]  "Cameroon"                  "Canada"
 [3]  "Central African Republic"  "Chad"
 [5]  "Chile"                     "China"
 [7]  "Colombia"                  "Congo"
 [9]  "Corsica"                   "Costa Rica"
[11]  "Crete"                     "Cuba"
[13]  "Cyprus"                    "Czechoslovakia"
[15]  "Denmark"                   "Dominican Republic"
[17]  "Ecuador"                   "Egypt"
[19]  "El Salvador"               "Ethiopia"
[21]  "Germany East"              "Ivory Coast"
[23]  "New Caledonia"             "Tristan da Cunha"
```

Notice that this formulation picks out countries like Ivory Coast and Tristan da Cunha that contain upper-case Cs in places other than as their first letters. To restrict the choice to first letters use the ^ operator before the list of capital letters:

```
as.vector(Country[grep("^[C-E]",as.character(Country))])
```

```
 [1]  "Cameroon"                  "Canada"
 [3]  "Central African Republic"  "Chad"
 [5]  "Chile"                     "China"
 [7]  "Colombia"                  "Congo"
 [9]  "Corsica"                   "Costa Rica"
[11]  "Crete"                     "Cuba"
[13]  "Cyprus"                    "Czechoslovakia"
[15]  "Denmark"                   "Dominican Republic"
[17]  "Ecuador"                   "Egypt"
[19]  "El Salvador"               "Ethiopia"
```

How about selecting the counties *not* ending with a specified patterns? The answer is simply to *use negative subscripts* to drop the selected items from the vector. Here are the countries that do not end with a letter between 'a' and 't':

```
as.vector(Country[-grep("[a-t]$",as.character(Country))])
```

```
[1] "Hungary"  "Italy"    "Norway"  "Paraguay"  "Peru"     "Sicily"
[7] "Turkey"   "Uruguay"  "USA"     "USSR"      "Vanuatu"
```

You see that USA and USSR are included in the list because we specified lower-case letters as the endings to omit. To omit these other countries, put ranges for both upper- and lower-case letters inside the square brackets, separated by a space:

```
as.vector(Country[-grep("[A-T a-t]$",as.character(Country))])
```

```
[1] "Hungary"  "Italy"    "Norway"   "Paraguay"  "Peru"  "Sicily"
[7] "Turkey"   "Uruguay"  "Vanuatu"
```

Dot . as the 'anything' character

Countries with 'y' as their second letter are specified by ^.y The ^ shows 'starting', then a single dot means one character of any kind, so *y* is the specified second character:

```
as.vector(Country[grep("^.y",as.character(Country))])
```

```
[1] "Cyprus" "Syria"
```

To search for countries with 'y' as third letter:

```
as.vector(Country[grep("^..y",as.character(Country))])
```

```
[1] "Egypt" "Guyana" "Seychelles"
```

If we want countries with 'y' as their sixth letter

```
as.vector(Country[grep("^. {5}y",as.character(Country))])
```

```
[1] "Norway" "Sicily" "Turkey"
```

(5 'anythings' is shown by '.' then curly brackets {5} then *y*). Which are the countries with 4 or fewer letters in their names?

```
as.vector(Country[grep("^. {,4}$",as.character(Country))])
```

```
[1]  "Chad"  "Cuba"  "Iran"  "Iraq"  "Laos"  "Mali"  "Oman"
[8]  "Peru"  "Togo"  "USA"   "USSR"
```

The '.' means 'anything' while the {,4} means 'repeat up to four' anythings (dots) before $ (the end of the string). So to find all the countries with 15 or more characters in their name is just

```
as.vector(Country[grep("^. {15, }$",as.character(Country))])
```

```
[1]  "Balearic Islands"     "Central African Republic"
[3]  "Dominican Republic"   "Papua New Guinea"
[5]  "Solomon Islands"      "Trinidad & Tobago"
[7]  "Tristan da Cunha"
```

Substituting text within character strings

Search-and-replace operations are carried out in R using the functions sub and gsub. The two substitution functions differ only in that sub replaces only the first occurrence of a pattern within a character string, whereas gsub replaces all occurrences. An example should make this clear. Here is a vector comprising seven character strings, called text:

```
text <- c("arm","leg","head", "foot","hand", "hindleg", "elbow")
```

We want to replace all lower-case 'h' with upper-case 'H':

```
gsub("h","H",text)
```

```
[1] "arm" "leg" "Head" "foot" "Hand" "Hindleg" "elbow"
```

Now suppose we want to convert the first occurrence of a lower-case 'o' into an upper-case 'O'. We use sub for this (not gsub):

```
sub("o","O",text)
```

```
[1] "arm" "leg" "head" "fOot" "hand" "hindleg" "elbOw"
```

You can see the difference between sub and gsub in the following, where both instances of 'o' in foot are converted to upper case by gsub but not by sub:

```
gsub("o","O",text)
```

```
[1] "arm" "leg" "head" "fOOt" "hand" "hindleg" "elbOw"
```

More general patterns can be specified in the same way as we learned for grep (above). For instance, to replace the first character of every string with upper-case 'O' we use the dot notation (. stands for 'anything') coupled with ^ (the 'start of string' marker):

```
gsub("^.","O",text)
```

```
[1] "Orm" "Oeg" "Oead" "Ooot" "Oand" "Oindleg" "Olbow"
```

It is useful to be able to manipulate the cases of character strings. Here, we capitalize the first character in each string:

```
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2",text, perl=TRUE)
```

```
[1] "Arm" "Leg" "Head" "Foot" "Hand" "Hindleg" "Elbow"
```

while here we convert all the characters to upper case:

```
gsub("(\\w*)", "\\U\\1",text, perl=TRUE)
```

```
[1] "ARM" "LEG" "HEAD" "FOOT" "HAND" "HINDLEG" "ELBOW"
```

Locations of the pattern within a vector of character strings using regexpr

Instead of substituting the pattern, we might want to know *if* it occurs in a string and, if so, *where* it occurs within each string. The result of regexpr, therefore, is a numeric vector (as with grep, above), but now indicating the position of the (first instance of the) pattern within the string (rather than just *whether* the pattern was there). If the pattern does not appear within the string, the default value returned by regexpr is −1. An example is essential to get the point of this:

```
text
```

```
[1] "arm" "leg" "head" "foot" "hand" "hindleg" "elbow"
```

```
regexpr("o",text)
```

```
[1] -1  -1  -1  2  -1  -1  4
attr(,"match.length")
[1] -1  -1  -1  1  -1  -1  1
```

This indicates that there were lower-case 'o's in two of the elements of text, and that they occurred in positions 2 and 4, respectively. Remember that if we wanted just the subscripts showing which elements of text contained an 'o' we would use grep like this:

```
grep("o",text)
```

```
[1] 4 7
```

and we would extract the character strings like this:

```
text[grep("o",text)]
```

```
[1] "foot" "elbow"
```

Counting how many 'o's there are in each string is a different problem again, and this involves the use of gregexpr:

```
freq<-as.vector(unlist (lapply(gregexpr("o",text),length)))
present<-ifelse(regexpr("o",text)<0,0,1)
freq*present
```

```
[1] 0 0 0 2 0 0 1
```

indicating that there are no 'o's in the first three character strings, two in the fourth and one in the last string. You will need lots of practice with these functions to appreciate all of the issues involved.

The function charmatch is for matching characters. If there are multiple matches (two or more) then the function returns the value 0 (e.g. when all the elements contain 'm'):

```
charmatch("m", c("mean", "median", "mode"))
```

```
[1] 0
```

If there is a unique match the function returns the index of the match within the vector of character strings (here in location number 2):

```
charmatch("med", c("mean", "median", "mode"))
```

```
[1] 2
```

Using %in% and which

You want to know all of the matches between one character vector and another:

```
stock<-c('car','van')
requests<-c('truck','suv','van','sports','car','waggon','car')
```

Use which to find the locations in the first-named vector of any and all of the entries in the second-named vector:

```
which(requests %in% stock)
```

```
[1] 3 5 7
```

If you want to know *what* the matches are as well as *where* they are,

```
requests [which(requests %in% stock)]
```

```
[1] "van" "car" "car"
```

You could use the match function to obtain the same result (p. 47):

```
stock[match(requests,stock)][!is.na(match(requests,stock))]
```

```
[1] "van" "car" "car"
```

but it's more clumsy. A slightly more complicated way of doing it involves sapply

```
which(sapply(requests, "%in%", stock))
```

```
van  car  car
  3    5    7
```

Note the use of quotes around the %in% function. Note that the match must be perfect for this to work ('car' with 'car' is not the same as 'car' with 'cars').

More on pattern matching

For the purposes of specifying these patterns, certain characters are called **metacharacters**, specifically \| () [{^ $ * + ? Any metacharacter with special meaning in your string may be quoted by preceding it with a backslash: \\{$ or * for instance. You might be used to specifying one or more 'wildcards' by * in DOS-like applications. In R, however, the regular expressions used are those specified by POSIX 1003.2, either extended or basic, depending on the value of the extended argument, unless perl = TRUE when they are those of PCRE (see ?grep for details).

Note that the square brackets in these class names [ ] are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list. For example, [[:alnum:]] means [0-9A-Za-z], except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. The interpretation below is that of the POSIX locale.

[:alnum:]	Alphanumeric characters: [:alpha:] and [:digit:].
[:alpha:]	Alphabetic characters: [:lower:] and [:upper:].
[:blank:]	Blank characters: space and tab.
[:cntrl:]	Control characters in ASCII, octal codes 000 through 037, and 177 (DEL).
[:digit:]	Digits: 0 1 2 3 4 5 6 7 8 9.
[:graph:]	Graphical characters: [:alnum:] and [:punct:].
[:lower:]	Lower-case letters in the current locale.
[:print:]	Printable characters: [:alnum:], [:punct:] and space.

[:punct:]	Punctuation characters: ! " # $ % & () *+, - ./: ; <=> ? @ [\] ^ _ ‘ { | } ~.
[:space:]	Space characters: tab, newline, vertical tab, form feed, carriage return, space.
[:upper:]	Upper-case letters in the current locale.
[:xdigit:]	Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.

Most metacharacters lose their special meaning inside lists. Thus, to include a literal], place it first in the list. Similarly, to include a literal ^, place it anywhere but first. Finally, to include a literal -, place it first or last. Only these and \ remain special inside character classes. To recap:

- Dot . matches any single character.
- Caret ^ matches the empty string at the beginning of a line.
- Dollar sign $ matches the empty string at the end of a line.
- Symbols \< and \> respectively match the empty string at the beginning and end of a word.
- The symbol \b matches the empty string at the edge of a word, and \B matches the empty string provided it is not at the edge of a word.

A regular expression may be *followed* by one of several repetition quantifiers:

?	The preceding item is optional and will be matched at most once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{n}	The preceding item is matched exactly *n* times.
{n, }	The preceding item is matched *n* or more times.
{,m}	The preceding item is matched up to *m* times.
{n,m}	The preceding item is matched at least *n* times, but not more than *m* times.

You can use the OR operator | so that "abba|cde" matches either the string "abba" or the string "cde".

Here are some simple examples to illustrate the issues involved.

```
text <- c("arm","leg","head", "foot","hand", "hindleg", "elbow")
```

The following lines demonstrate the "consecutive characters" {n} in operation:

```
grep("o{1}",text,value=T)
```

```
[1] "foot" "elbow"
```

```
grep("o{2}",text,value=T)
```

```
[1] "foot"
```

```
grep("o{3}",text,value=T)
character(0)
```

The following lines demonstrate the use of {n, } "n or more" character counting in words:

```
grep("[[:alnum:]]{4, }",text,value=T)
[1] "head" "foot" "hand" "hindleg" "elbow"
grep("[[:alnum:]]{5, }",text,value=T)
[1] "hindleg" "elbow"
grep("[[:alnum:]]{6, }",text,value=T)
[1] "hindleg"
grep("[[:alnum:]]{7, }",text,value=T)
[1] "hindleg"
```

Perl regular expressions

The perl = TRUE argument switches to the PCRE library that implements regular expression pattern matching using the same syntax and semantics as Perl 5.6 or later (with just a few differences). For details (and there are many) see ?regexp.

Perl is good for altering the cases of letters. Here, we capitalize the first character in each string:

```
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2",text, perl=TRUE)
[1] "Arm" "Leg" "Head" "Foot" "Hand" "Hindleg" "Elbow"
```

while here we convert all the character to upper case:

```
gsub("(\\w*)", "\\U\\1",text, perl=TRUE)
[1] "ARM" "LEG" "HEAD" "FOOT" "HAND" "HINDLEG" "ELBOW"
```

Stripping patterned text out of complex strings

Suppose that we want to tease apart the information in these complicated strings:

```
(entries <-c ("Trial 1 58 cervicornis (52 match)", "Trial 2 60 terrestris (51 matched)",
"Trial 8 109 flavicollis (101 matches)"))
[1] "Trial 1 58 cervicornis (52 match)"
[2] "Trial 2 60 terrestris (51 matched)"
[3] "Trial 8 109 flavicollis (101 matches)"
```

The first task is to remove the material on numbers of matches including the brackets:

```
gsub(" *$", "", gsub("\\(.*\\)$", "", entries))
[1] "Trial 1 58 cervicornis"  "Trial 2 60 terrestris"
[3] "Trial 8 109 flavicollis"
```

The first argument " *$", "", removes the "trailing blanks" while the second deletes everything .* between the left \\(and right \\) hand brackets "\\(.*\\)$" substituting this with nothing "". The next job is to strip out the material in brackets and to extract that material, ignoring the brackets themselves:

```
pos<- regexpr("\\(.*\\)$", entries)
substring(entries, first=pos+1, last=pos+attr(pos,"match.length")-2)
```

```
[1] "52 match" "51 matched" "101 matches"
```

To see how this has worked it is useful to inspect the values of pos that have emerged from the regexpr function:

```
pos
```

```
[1] 25 23 25
attr(,"match.length")
[1] 10 12 13
```

The left-hand bracket appears in position 25 in the first and third elements (note that there are two blanks before 'cervicornis') but in position 23 in the second element. Now the lengths of the strings matching the pattern \\(.*\\)$ can be checked; it is the number of 'anything' characters between the two brackets, plus one for each bracket: 10, 12 and 13.

Thus, to extract the material in brackets, but to ignore the brackets themselves, we need to locate the first character to be extracted (pos+1) and the last character to be extracted pos+attr(pos,"match.length")-2, then use the substring function to do the extracting. Note that first and last are vectors of length 3 (= length(entries)).

Testing and Coercing in R

Objects have a type, and you can test the type of an object using an is.type function (Table 2.4). For instance, mathematical functions expect numeric input and text-processing

Table 2.4. Functions for testing (is) the attributes of different categories of object (arrays, lists, etc.) and for coercing (as) the attributes of an object into a specified form. Neither operation changes the attributes of the object.

Type	Testing	Coercing
Array	is.array	as.array
Character	is.character	as.character
Complex	is.complex	as.complex
Dataframe	is.data.frame	as.data.frame
Double	is.double	as.double
Factor	is.factor	as.factor
List	is.list	as.list
Logical	is.logical	as.logical
Matrix	is.matrix	as.matrix
Numeric	is.numeric	as.numeric
Raw	is.raw	as.raw
Time series (ts)	is.ts	as.ts
Vector	is.vector	as.vector

functions expect character input. Some types of objects can be coerced into other types. A familiar type of coercion occurs when we interpret the TRUE and FALSE of logical variables as numeric 1 and 0, respectively. Factor levels can be coerced to numbers. Numbers can be coerced into characters, but non-numeric characters cannot be coerced into numbers.

```
as.numeric(factor(c("a","b","c")))
```

```
[1] 1 2 3
```

```
as.numeric(c("a","b","c"))
```

```
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

```
as.numeric(c("a","4","c"))
```

```
[1] NA 4 NA
Warning message:
NAs introduced by coercion
```

If you try to coerce complex numbers to numeric the imaginary part will be discarded. Note that is.complex and is.numeric are never both TRUE.

We often want to coerce tables into the form of vectors as a simple way of stripping off their dimnames (using as.vector), and to turn matrixes into dataframes (as.data.frame). A lot of testing involves the NOT operator ! in functions to return an error message if the wrong type is supplied. For instance, if you were writing a function to calculate geometric means you might want to test to ensure that the input was numeric using the !is.numeric function

```
geometric<-function(x){
if(!is.numeric(x)) stop ("Input must be numeric")
exp(mean(log(x))) }
```

Here is what happens when you try to work out the geometric mean of character data

```
geometric(c("a","b","c"))
```

```
Error in geometric(c("a", "b", "c")) : Input must be numeric
```

You might also want to check that there are no zeros or negative numbers in the input, because it would make no sense to try to calculate a geometric mean of such data:

```
geometric<-function(x){
if(!is.numeric(x)) stop ("Input must be numeric")
if(min(x)<=0) stop ("Input must be greater than zero")
exp(mean(log(x))) }
```

Testing this:

```
geometric(c(2,3,0,4))
```

```
Error in geometric(c(2, 3, 0, 4)) : Input must be greater than zero
```

But when the data are OK there will be no messages, just the numeric answer:

```
geometric(c(10,1000,10,1,1))
```

```
[1] 10
```

Dates and Times in R

The measurement of time is highly idiosyncratic. Successive years start on different days of the week. There are months with different numbers of days. Leap years have an extra day in February. Americans and Britons put the day and the month in different places: 3/4/2006 is March 4 for the former and April 3 for the latter. Occasional years have an additional 'leap second' added to them because friction from the tides is slowing down the rotation of the earth from when the standard time was set on the basis of the tropical year in 1900. The cumulative effect of having set the atomic clock too slow accounts for the continual need to insert leap seconds (32 of them since 1958). There is currently a debate about abandoning leap seconds and introducing a 'leap minute' every century or so instead. Calculations involving times are complicated by the operation of time zones and daylight saving schemes in different countries. All these things mean that working with dates and times is excruciatingly complicated. Fortunately, R has a robust system for dealing with this complexity. To see how R handles dates and times, have a look at Sys.time():

```
Sys.time()
```

```
[1] "2005-10-23 10:17:42 GMT Daylight Time"
```

The answer is strictly hierarchical from left to right: the longest time scale (years) comes first, then month then day separated by hyphens (minus signs), then there is a blank space and the time, hours first (in the 24-hour clock) then minutes, then seconds separated by colons. Finally there is a character string explaining the time zone. You can extract the date from Sys.time() using substr like this:

```
substr(as.character(Sys.time()),1,10)
```

```
[1] "2005-10-23"
```

or the time

```
substr(as.character(Sys.time()),12,19)
```

```
[1] "10:17:42"
```

If you type

```
unclass(Sys.time())
```

```
[1] 1130679208
```

you get the number of seconds since 1 January 1970. There are two basic classes of date/times. Class POSIXct represents the (signed) number of seconds since the beginning of 1970 as a numeric vector: this is more convenient for including in dataframes. Class POSIXlt is a named list of vectors closer to human-readable forms, representing seconds, minutes, hours, days, months and years. R will tell you the date and time with the date function:

```
date()
```

```
[1] "Fri Oct 21 06:37:04 2005"
```

The default order is day name, month name (both abbreviated), day of the month, hour (24-hour clock), minute, second (separated by colons) then the year. You can convert Sys.time to an object that inherits from class POSIXlt like this:

```
date<- as.POSIXlt(Sys.time())
```

You can use the element name operator $ to extract parts of the date and time from this object using the following names: sec, min, hour, mday, mon, year, wday, yday and isdst (with obvious meanings except for mday (=day number within the month), wday (day of the week starting at 0 = Sunday), yday (day of the year after 1 January = 0) and isdst which means 'is daylight savings time in operation?' with logical 1 for TRUE or 0 for FALSE). Here we extract the day of the week (date$wday = 0 meaning Sunday) and the Julian date (day of the year after 1 January as date$yday)

```
date$wday
```

```
[1] 0
```

```
date$yday
```

```
[1] 295
```

for 23 October. Use unclass with unlist to view all of the components of date:

```
unlist(unclass(date))
```

```
sec   min   hour   mday   mon   year   wday   yday   isdst
 42    17     10     23     9    105      0    295       1
```

Note that the month of October is 9 (not 10) because January is scored as month 0, and years are scored as post-1900.

Calculations with dates and times

You can do the following calculations with dates and times:

- time + number
- time – number
- time1 – time2
- time1 'logical operation' time2

where the logical operations are one of ==, !=, <, <=, '>' or >=. You can add or subtract a number of seconds or a difftime object (see below) from a date-time object, but you cannot add two date-time objects. Subtraction of two date-time objects is equivalent to using difftime (see below). Unless a time zone has been specified, POSIXlt objects are interpreted as being in the current time zone in calculations.

The thing you need to grasp is that you should convert your dates and times into POSIXlt objects *before* starting to do any calculations. Once they are POSIXlt objects, it is straightforward to calculate means, differences and so on. Here we want to calculate the number of days between two dates, 22 October 2003 and 22 October 2005:

```
y2<-as.POSIXlt("2003-10-22")
y1<-as.POSIXlt("2005-10-22")
```

Now you can do calculations with the two dates:

```
y1-y2
```

```
Time difference of 731 days
```

Note that you cannot *add* two dates. It is easy to calculate differences between times using this system. Note that the dates are separated by hyphens whereas the times are separated by colons:

```
y3<-as.POSIXlt("2005-10-22 09:30:59")
y4<-as.POSIXlt("2005-10-22 12:45:06")
y4-y3
```

```
Time difference of 3.235278 hours
```

The difftime function

Working out the time difference between to dates and times involves the difftime function, which takes two date-time objects as its arguments. The function returns an object of class difftime with an attribute indicating the units. How many days elapsed between 15 August 2003 and 21 October 2005?

```
difftime("2005-10-21","2003-8-15")
```

```
Time difference of 798 days
```

If you want only the number of days, for instance to use in calculation, then write

```
as.numeric(difftime("2005-10-21","2003-8-15"))
```

```
[1] 798
```

For differences in hours include the times (colon-separated) and write

```
difftime("2005-10-21 5:12:32","2005-10-21 6:14:21")
```

```
Time difference of -1.030278 hours
```

The result is negative because the first time (on the left) is before the second time (on the right). Alternatively, you can subtract one date-time object from another directly:

```
ISOdate(2005,10,21)-ISOdate(2003,8,15)
```

```
Time difference of 798 days
```

You can convert character stings into difftime objects using the as.difftime function:

```
as.difftime(c("0:3:20", "11:23:15"))
```

```
Time differences of 3.333333, 683.250000 mins
```

You can specify the format of your times. For instance, you may have no information on seconds, and your times are specified just as hours (format %H) and minutes (%M). This is what you do:

```
as.difftime(c("3:20", "23:15", "2:"), format= "%H:%M")
```

```
Time differences of 3.333333, 23.250000, NA hours
```

Because the last time in the sequence '2:' had no minutes it is marked as NA.

The strptime function

You can 'strip a date' out of a character string using the strptime function. There are functions to convert between character representations and objects of classes POSIXlt and POSIXct representing calendar dates and times. The details of the formats are system-specific, but the following are defined by the POSIX standard for strptime and are likely to be widely available. Any character in the format string other than the % symbol is interpreted literally.

%a Abbreviated weekday name

%A Full weekday name

%b Abbreviated month name

%B Full month name

%c Date and time, locale-specific

%d Day of the month as decimal number (01–31)

%H Hours as decimal number (00–23) on the 24-hour clock

%I Hours as decimal number (01–12) on the 12-hour clock

%j Day of year as decimal number (001–366)

%m Month as decimal number (01–12)

%M Minute as decimal number (00–59)

%p AM/PM indicator in the locale

%S Second as decimal number (00–61, allowing for two 'leap seconds')

%U Week of the year (00–53) using the first Sunday as day 1 of week 1

%w Weekday as decimal number (0–6, Sunday is 0)

%W Week of the year (00–53) using the first Monday as day 1 of week 1

%x Date, locale-specific

%X Time, locale-specific

%Y Year with century

%Z Time zone as a character string (output only)

Where leading zeros are shown they will be used on output but are optional on input.

Dates in Excel spreadsheets

The trick is to learn how to specify the format of your dates properly in strptime. If you had dates (and no times) in a dataframe in Excel format (day/month/year)

```
excel.dates <- c("27/02/2004", "27/02/2005", "14/01/2003",
                 "28/06/2005", "01/01/1999")
```

then the appropriate format would be "%d/%m/%Y" showing the format names (from the list above) and the 'slash' separators / (note the upper case for year %Y; this is the unambiguous year including the century, 2005 rather than the potentially ambiguous 05 for which the format is %y). To turn these into R dates, write

```
strptime(excel.dates,format="%d/%m/%Y")
```

```
[1] "2004-02-27" "2005-02-27" "2003-01-14" "2005-06-28" "1999-01-01"
```

Here is another example, but with years in two-digit form (%y), and the months as abbreviated names (%b) and no separators:

```
other.dates<- c("1jan99", "2jan05", "31mar04", "30jul05")
strptime(other.dates, "%d%b%y")
```

```
[1] "1999-01-01" "2005-01-02" "2004-03-31" "2005-07-30"
```

You will often want to create POSIXlt objects from components stored in vectors within dataframes. For instance, here is a dataframe with the hours, minutes and seconds from an experiment with two factor levels in separate columns:

```
times<-read.table("c:\\temp\\times.txt",header=T)
times
```

```
    hrs  min  sec  experiment
 1    2   23    6           A
 2    3   16   17           A
 3    3    2   56           A
 4    2   45    0           A
 5    3    4   42           A
 6    2   56   25           A
 7    3   12   28           A
 8    1   57   12           A
 9    2   22   22           B
10    1   42    7           B
11    2   31   17           B
12    3   15   16           B
13    2   28    4           B
14    1   55   34           B
15    2   17    7           B
16    1   48   48           B
```

```
attach(times)
```

Because the times are not in POSIXlt format, you need to paste together the hours, minutes and seconds into a character string with colons as the separator:

```
paste(hrs,min,sec,sep=":")
```

```
 [1]  "2:23:6"  "3:16:17" "3:2:56"   "2:45:0"   "3:4:42"  "2:56:25"  "3:12:28"
 [8] "1:57:12" "2:22:22" "1:42:7"  "2:31:17" "3:15:16"   "2:28:4" "1:55:34"
[15]  "2:17:7" "1:48:48"
```

Now save this object as a difftime vector called duration:

```
duration<-as.difftime (paste(hrs,min,sec,sep=":"))
```

Then you can carry out calculations like mean and variance using the tapply function:

```
tapply(duration,experiment,mean)
```

```
       A         B
2.829375  2.292882
```

Calculating time differences between the rows of a dataframe

A common action with time data is to compute the time difference between successive rows of a dataframe. The vector called duration created above is of class difftime and contains 16 times measured in decimal hours:

```
class(duration)
```

```
[1] "difftime"
```

```
duration
```

```
Time differences of 2.385000, 3.271389, 3.048889, 2.750000, 3.078333,
2.940278, 3.207778, 1.953333, 2.372778, 1.701944, 2.521389, 3.254444,
2.467778, 1.926111, 2.285278, 1.813333 hours
```

We can compute the differences between successive rows using subscripts, like this

```
duration[1:15]-duration[2:16]
```

```
Time differences of -0.8863889, 0.2225000, 0.2988889, -0.3283333,
0.1380556, -0.2675000, 1.2544444, -0.4194444, 0.6708333, -0.8194444,
-0.7330556, 0.7866667, 0.5416667, -0.3591667, 0.4719444 hours
```

You might want to make the differences between successive rows into part of the dataframe (for instance, to relate change in time to one of the explanatory variables in the dataframe). Before doing this, you need to decide on the row in which to put the first of the differences. Is the change in time between rows 1 and 2 related to the explanatory variables in row 1 or row 2? Suppose it is row 1 that we want to contain the first time difference (−0.886). Because we are working with differences (see p. 719) the vector of differences is shorter by one than the vector from which it was calculated:

```
length(duration[1:15]-duration[2:16])
```

```
[1] 15
```

```
length(duration)
```

```
[1] 16
```

so we need to add one 'NA' to the bottom of the vector (in row 16).

```
diffs<-c(duration[1:15]-duration[2:16],NA)
diffs
```

```
 [1] -0.8863889  0.2225000  0.2988889 -0.3283333  0.1380556 -0.2675000
 [7]  1.2544444 -0.4194444  0.6708333 -0.8194444 -0.7330556  0.7866667
[13]  0.5416667 -0.3591667  0.4719444          NA
```

Now we can make this new vector part of the dataframe called times:

```
times$diffs<-diffs
times
```

```
    hrs  min  sec  experiment       diffs
1     2   23    6           A  -0.8863889
2     3   16   17           A   0.2225000
3     3    2   56           A   0.2988889
4     2   45    0           A  -0.3283333
5     3    4   42           A   0.1380556
6     2   56   25           A  -0.2675000
7     3   12   28           A   1.2544444
8     1   57   12           A  -0.4194444
9     2   22   22           B   0.6708333
10    1   42    7           B  -0.8194444
11    2   31   17           B  -0.7330556
12    3   15   16           B   0.7866667
13    2   28    4           B   0.5416667
14    1   55   34           B  -0.3591667
15    2   17    7           B   0.4719444
16    1   48   48           B          NA
```

There is more about dates and times in dataframes on p. 126.