Appendix

Essentials of the R Language

R as a Calculator

The command line after the screen prompt > is an excellent calculator:

```
> log(42/7.3)
[1] 1.749795
```

By default, logs in R are base *e* (natural or Napierian, not base 10 logs), but you can specify any base you want, as a second argument to the log function. Here is log base 2 of 16:

```
> log(16,2)
[1] 4
```

Each line can have many characters, but if you want to evaluate a complicated expression, you might like to continue it on one or more further lines for clarity. The way you do this is 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 line 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 press the Esc key and use the Up arrow to edit the last (incomplete) line.

From here onwards (and throughout the book), the prompt character > is omitted. The material that you should type on the command line is shown in red font in this book. Just press the Return key to see the answer. The output from R is shown in dark blue

Statistics: An Introduction Using R, Second Edition. Michael J. Crawley.

Courier New font, which uses absolute rather than proportional spacing, so that columns of numbers remain neatly aligned on the page and on the screen.

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

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

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

All of the usual calculations can be done directly, and the standard order of precedence applies: powers and roots are evaluated first, then multiplication and division, then finally addition and subtraction. Although there is a named function for square roots, `sqrt`, roots are generally calculated as fractional powers. Exponentiation (powers) uses the caret ('hat') `^` operator (not `**` as in some computing languages like Fortran or GLIM). So the cube root of 8 is

```
8^(1/3)
```

```
[1] 2
```

The exponentiation operator `^` groups *right to left*, but all other operators group left to right. Thus, `2^2^3` is `2^8`, not `4^3`, whereas `1-1-1` is `-1`, not `1`. Use brackets as necessary to override the precedence. For a *t* test, where the value required is $\frac{|y_A - y_B|}{SE_{diff}}$ and the numbers are 5.7, 6.8 and 0.38, you type:

```
abs(5.7-6.8)/0.38
```

```
[1] 2.894737
```

The default number of digits printed out by R is 7 (as above). You can control the number of digits printed using the `digits` option like this:

```
options(digits=3)
abs(5.7-6.8)/0.38
```

```
[1] 2.89
```

Built-in Functions

All the mathematical functions you could ever want are here (Table A.1). We have already met the `log` function; the antilog function to the base *e* is `exp`:

```
exp(1)
```

```
[1] 2.718282
```

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 number it is clearly not exactly zero (so you need to be careful when testing for exact equality of real numbers; see p. 313).

Table A.1 Mathematical functions used in R

Function	Meaning
`log(x)`	log to base e of x
`exp(x)`	antilog of x (=2.7818^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)`	$\Gamma(x)$ $(x-1)!$ 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 six 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

(*continued*)

Table A.1 (*Continued*)

Function	Meaning
`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 on real or complex numbers
`abs(x)`	the absolute value of x, ignoring the minus sign if there is one

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), known in maths as *modulo*:

```
119 %% 13
[1] 2
```

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

```
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, type:

```
15421 %% 7 == 0
[1] TRUE
```

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' <- without a 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` meaning '*x* gets 5' and `x < -5` which is a logical question, asking 'is *x* less than minus 5?' and producing the answer TRUE or FALSE.

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
```

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 writing functions. 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`:

```
3/0
[1] Inf
```

or minus infinity, which is `-Inf` in R:

```
-12/0
[1] -Inf
```

Calculations involving infinity can be evaluated:

```
exp(-Inf)
[1] 0

0/Inf
[1] 0

(0:3)^Inf

[1]  0  1 Inf Inf
```

The syntax `0:3` is very useful. It generaters a series (0, 1, 2, 3 in this case) and the function is evaluated for each of the values to produce a **vector** of answers. In this case, the vector is of `length = 4`.

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). 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)`. 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))
```

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

Note the use of round brackets to get the answer printed as well as allocated. Making an index of the missing values in an array could use the `seq` function:

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

```
[1] 7 8
```

However, 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)
```

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

Alternatively, 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
```

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 formulas. Thus in a model formula `*` indicates the main effects plus interaction, `:` indicates the interaction between two variables and `^` means all interactions up to the indicated power.

Creating a Vector

Vectors are variables with one or more values of the same type: logical, integer, real, complex, string (or character) or raw. 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 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)
```

Alternatively, you can enter then 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
```

Press the Enter key twice 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.csv` or `read.table` (p. 25).

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, 1, . . . , 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 the great strengths of R is its ability to evaluate functions over entire vectors, thereby avoiding the need for loops and subscripts. The most important vector functions are shown in Table A.2.

Table A.2 Vector functions 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*, with degrees of freedom = `length(x) - 1`
`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 *i*th position in each
`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 *i*th 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.csv("c:\\temp\\daphnia.csv")
attach(data)
names(data)
```

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

The response variable is `Growth.rate` and the other three variables are factors. 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.

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. 309). 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 number 1) and the second subscript refers to the *column* number (the columns are margin number 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$resid` (p. 317). This is known as 'indexing tagged lists' using the *element names operator* `$`.

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. 314). Logical TRUE has been coerced to numeric 1 and logical FALSE has been coerced to numeric 0.

That's 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 *x* values that are less than 5, we write:

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

```
[1] 10
```

Let us 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 us 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 the addresses of the elements of *y* that contained values bigger than 5:

```
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
```

Trimming Vectors Using Negative Subscripts

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*:

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

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))])
```

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

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

```
[1] 5.2
```

Logical Arithmetic

Arithmetic involving logical expressions is very useful in programming and in selection of variables (see Table A.3). 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.

Table A.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)

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
```

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. Suppose that we need four 9s, one 15, four 21s and two 83s. We use the concatenation function `c` to create two vectors, one for the target numbers and another for the repeats of each element of the target:

```
rep(c(9,15,21,83),c(4,1,4,2))
```

```
[1] 9 9 9 9 15 21 21 21 21 83 83
```

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 regular series of whole numbers use the colon operator (`:` as on p. 296). 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. 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)
```

Suppose also that 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).

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
```

Here, by default, the numbers are entered columnwise. The `class` and `attributes` of *X* indicate that it is a matrix of 3 rows and 3 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)
```

and 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
```

Character Strings

In R, character strings are defined by double quotation marks. 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, and so on (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 eight 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
```

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"
```

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 like x or y), a statement of the form `symbol = expression` (e.g. `pch=16`) or the special formal argument `...` (dot dot 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 do not need 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. 46). Now we have all the tools we need to write a general function to calculate medians. Let us 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 us try it out, first with the odd-numbered vector `y`, then with the even-numbered vector `y[-1]`, after the first element of y (which is `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`.

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*: 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).

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)                    {
if (x == 0) return(0)
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 through 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 `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)
```

Evaluating Functions with `apply`

The `apply` function is used for applying functions to the rows (subscript 1) or columns (subscript 2) of matrices or dataframes:

```
(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
```

Here are the row totals (four of them) across margin 1 of the matrix:

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

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

Here are the column totals (six of them) across margin 2 of the matrix:

```
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
```

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
```

Testing and Coercing in R

Objects have a type, and you can test the type of an object using an `is.type` function (see Table A.4). For instance, mathematical functions expect numeric input and text-processing functions expect character input. Some types of objects can be coerced into other types (again see Table A.4). 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.

Table A.4 Testing and coercing

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`
Raw	`is.raw`	`as.raw`
List	`is.list`	`as.list`
Logical	`is.logical`	`as.logical`
Matrix	`is.matrix`	`as.matrix`
Numeric	`is.numeric`	`as.numeric`
Time series (ts)	`is.ts`	`as.ts`
Vector	`is.vector`	`as.vector`

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 matrices 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 4 March in one case and 3 April in another. 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] "2015-03-23 08:56:26 GMT"
```

The answer is strictly hierarchical from left to right: the longest time scale (year = 2015) comes first, then month (March = 03) then day (the 23rd) separated by hyphens, then there is a blank space and the time hours first (08 in the 24-hour clock) then minutes (56), then seconds (26) separated by colons. Finally, there is a character string explaining the time zone (GMT = Greenwich Mean Time). Now obviously, there is a lot going on here. To accommodate this, R uses the Portable Operating System Interface system (POSIX) for representing times and dates:

```
class(Sys.time())
```

```
[1] "POSIXct" "POSIXt"
```

There are two quite different ways of expressing time, each useful in a different context. For doing graphs and regression analyses where time is the explanatory variable, you want time to be a continuous number. But where you want to do summary statistics (say, monthly means across a long run of years), you want month to be a factor with 12 levels. In R this distinction is handled by the two classes `POSIXct` and `POSIXlt`. The first class, with suffix `ct`, you can think of as *continuous time* (i.e. a number of seconds). The other class, with suffix `lt`, you can think of as *list time* (i.e. a named list of vectors, representing all of the various categorical descriptions of the time, including day of the week and so forth). It is hard to remember these acronyms, but it is well worth making the effort. You can see (above) that `Sys.time` is of both class `ct` and class `t`. Times that are of class `lt` are also of class `t`. What this means is that class `POSIXt` tells you that an object is a time, but not what *sort* of time it is; it does not distinguish between continuous time `ct` and list times `lt`. Naturally, you can easily convert from one representation to the other:

```
time.list <- as.POSIXlt(Sys.time())
class(time.list)
```

```
[1] "POSIXlt" "POSIXt"
```

```
unlist(time.list)
```

```
  sec   min  hour  mday   mon  year  wday  yday  isdst
   26    56     8    23     2   115     1    81      0
```

Here you see the nine components of the list of class `POSIXlt`. The time is represented by the number of seconds (`sec`), minutes (`min`) and hours (`hour` on the 24-hour clock). Next comes the day of the month (`mday`, starting from 1), then the month of the year (`mon`, starting (non-intuitively, perhaps) at January = 0; you can think of it as 'months completed so far'), then the year (starting at 0 = 1900). The day of the week (`wday`) is coded from Sunday = 0

to Saturday = 6. The day within the year (yday) is coded from 0 = January 1 (you can think of it as 'days completed so far'). Finally, there is a logical variable isdst which asks whether daylight saving time is in operation (0 = FALSE in this case because we are on Greenwich Mean Time). The ones you are most likely to use include year (to get yearly mean values), mon (to get monthly means) and wday (to get means for the different days of the week, useful for answering questions like 'are Fridays different from Mondays?').

You can use the element name operator $ to extract parts of the date and time from this object using the names: sec, min, hour, mday, mon, year, wday, yday and isdst. 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):

```
time.list$wday
```

```
[1] 1
```

```
time.list$yday
```

```
[1] 81
```

meaning that today is a Monday and the 82nd day of the year (81 days are completed so far).

It is important to know how to read dates and times into R. Here we illustrate two of the commonest methods:

- Excel dates
- dates stored in separate variables for year, month, day, hour, minute, second

The Excel date convention uses forward slashes to separate the numerical components. The English way to give a date is day/month/year and the American way is month/day/year. Single-figure days and months can have a leading zero. So 03/05/2015 means 3 May in England but 5 March in America. Because of the appearance of the slashes these are character strings rather than numbers, and R will interpret them as factors when they are read from a file into a dataframe. You have to convert this factor into a date–time object using a function called strptime (you can remember this as 'stripping the time' out of a character string). We start by reading the Excel dates into a dataframe:

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

```
  x       date
1 3 15/06/2014
2 1 16/06/2014
3 6 17/06/2014
4 7 18/06/2014
5 8 19/06/2014
6 9 20/06/2014
```

As things stand, the date is not recognized by R:

```
class(date)
```

```
[1] "factor"
```

The `strptime` function takes the name of the factor (`date`) and a format statement showing exactly what each element of the character string represents and what symbols are used as separators (forward slashes in this case). In our example, the day of the month comes first (`%d`) then a slash, then the month (`%m`) then another slash, then the year in full (`%Y`; if the year was just the last two digits '15' then the appropriate code would be `%y` in lower case). We give the translated date a new name like this:

```
Rdate <- strptime(date,"%d/%m/%Y")
class(Rdate)
```

```
[1] "POSIXlt" "POSIXt"
```

That's more like it. Now we can do date-related things. For instance, what is the mean value of *x* for each day of the week? You need to know that the name of the day of the week within the list called `Rdate` is `wday`, then

```
tapply(x,Rdate$wday,mean)
```

```
    0     1     2     3     4     5     6
5.660 2.892 5.092 7.692 8.692 9.692 8.892
```

The value was lowest on Mondays (day 1) and highest on Fridays (day 5).

For cases where your data file has several variables representing the components of date or a time (say, hours, minutes and seconds separately), you use the `paste` function to join the components up into a single character string, using the appropriate separators (hyphens for dates, colons for times). Here is the data file:

```
time <- read.csv("c:\\temp\\times.csv")
attach(time)
head(time)
```

```
  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
```

Create a vector of times *y* using `paste` with a colon separator:

```
y <- paste(hrs,min,sec,sep=":")
y
```

```
 [1] "2:23:6"  "3:16:17" "3:2:56"  "2:45:0"  "3:4:42"  "2:56:25"
 [7] "3:12:28" "1:57:12" "2:22:22" "1:42:7"  "2:31:17" "3:15:16"
[13] "2:28:4"  "1:55:34" "2:17:7"  "1:48:48"
```

You then need to convert these character strings into something that R will recognize as a date and/or a time. If you use `strptime` then R will automatically add today's date to the

POSIXct object. Note that "%T" is a shortcut for "%H:%M:%S". For more details and a full list of codes, see *The R Book* (Crawley, 2013).

```
strptime(y,"%T")
```

```
 [1] "2014-01-22 02:23:06" "2014-01-22 03:16:17" "2014-01-22 03:02:56"
 [4] "2014-01-22 02:45:00" "2014-01-22 03:04:42" "2014-01-22 02:56:25"
     ....
```

When you only have times rather than dates and times, then you may want to use the as.difftime function rather than as.POSIXct to create the variables to work with:

```
(Rtime <- as.difftime(y))
```

```
Time differences in hours
 [1] 2.385000 3.271389 3.048889 2.750000 3.078333 2.940278 3.207778
 [8] 1.953333 2.372778 1.701944 2.521389 3.254444 2.467778 1.926111
[15] 2.285278 1.813333
```

This converts all of the times into decimal hours in this case (it would convert to minutes if there were no hours in the examples). Here is the mean time for each of the two experiments (A and B):

```
tapply(Rtime,experiment,mean)
```

```
       A        B
2.829375 2.292882
```

Calculations with Dates and Times

You can subtract one date from another, but you can*not* add two dates together. The following calculations are allowed for dates and times:

- time + number
- time – number
- time1 – time2
- time1 'logical operation' time2

where the logical operations are one of ==, !=, <, <=, > or >=.

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("2018-10-22")
y1 <- as.POSIXlt("2015-10-22")
```

Now you can do calculations with the two dates:

```
y2-y1
```

```
Time difference of 1096 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("2018-10-22 09:30:59")
y4 <- as.POSIXlt("2018-10-22 12:45:06")
y4-y3
```

```
Time difference of 3.235278 hours
```

Alternatively, you can do these calculations using the `difftime` function:

```
difftime("2018-10-22 12:45:06","2018-10-22 09:30:59")
```

```
Time difference of 3.235278 hours
```

Here is an example of logical operation on date time objects. We want to know whether `y4` was later than `y3`:

```
y4 > y3
```

```
[1] TRUE
```

Understanding the Structure of an R Object Using `str`

We finish with a simple but important function for understanding what kind of object is represented by a given name in the current R session. We look at three objects of increasing complexity: a vector of numbers, a list of various classes and a linear model.

```
x <- runif(23)
str(x)
```

```
 num [1:23] 0.971 0.23 0.645 0.697 0.537 ...
```

We see that `x` is a number (`num`) in a vector of length 23 (`[1:23]`) with the first five values as shown (0.971, . . .).

In this example of intermediate complexity the variable called `basket` is a list:

```
basket <- list(rep("a",4),c("b0","b1","b2"),9:4,gl(5,3))
basket
```

```
[[1]]
[1] "a" "a" "a" "a"

[[2]]
[1] "b0" "b1" "b2"

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

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

The first element is a character vector of length 4 (all "a"), while the second has three different character strings. The third is a numeric vector and the fourth is a factor of length 12 with four levels. Here is what you get from str:

```
str(basket)
```

```
List of 4
 $ : chr [1:4] "a" "a" "a" "a"
 $ : chr [1:3] "b0" "b1" "b2"
 $ : int [1:6] 9 8 7 6 5 4
 $ : Factor w/ 5 levels "1","2","3","4",..: 1 1 1 2 2 2 3 3 3 4 ...
```

First you see the length of the list (4) then the attributes of each of the four components: the first two are character strings (chr), the third contains integers (int) and the fourth is a factor with five levels.

Finally, we look at an example of a complex structure – a quadratic linear model:

```
xv <- seq(0,30)
yv <- 2+0.5*xv+rnorm(31,0,2)
model <- lm(yv~xv+I(xv^2))
str(model)
```

```
List of 12
 $ coefficients : Named num [1:3] 2.51317 0.38809 0.00269
  ..- attr(*, "names")= chr [1:3] "(Intercept)" "xv" "I(xv^2)"
 $ residuals  : Named num [1:31] -1.712 -1.869 4.511 0.436 -0.207 ...
  ..- attr(*, "names")= chr [1:31] "1" "2" "3" "4" ...
 $ effects    : Named num [1:31] -50.965 23.339 -1.068 0.646 0.218 ...
  ..- attr(*, "names")= chr [1:31] "(Intercept)" "xv" "I(xv^2)" "" ...
 $ rank       : int 3
 $ fitted.values: Named num [1:31] 2.51 2.9 3.3 3.7 4.11 ...
  ..- attr(*, "names")= chr [1:31] "1" "2" "3" "4" ...
 $ assign     : int [1:3] 0 1 2
 $ qr         :List of 5
  ..$ qr  : num [1:31, 1:3] -5.57 0.18 0.18 0.18 0.18 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:31] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:3] "(Intercept)" "xv" "I(xv^2)"
  .. ..- attr(*, "assign")= int [1:3] 0 1 2
  ..$ qraux: num [1:3] 1.18 1.24 1.13
  ..$ pivot: int [1:3] 1 2 3
  ..$ tol  : num 1e-07
  ..$ rank : int 3
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 28
 $ xlevels    : Named list()
 $ call       : language lm(formula = yv ~ xv + I(xv^2))
 $ terms      :Classes 'terms', 'formula' length 3 yv ~ xv + I(xv^2)
  .. ..- attr(*, "variables")= language list(yv, xv, I(xv^2))
```

```
  .. ..- attr(*, "factors")= int [1:3, 1:2] 0 1 0 0 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:3] "yv" "xv" "I(xv^2)"
  .. .. .. ..$ : chr [1:2] "xv" "I(xv^2)"
  .. ..- attr(*, "term.labels")= chr [1:2] "xv" "I(xv^2)"
  .. ..- attr(*, "order")= int [1:2] 1 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(yv, xv, I(xv^2))
  .. ..- attr(*, "dataClasses")= Named chr [1:3] "numeric" "numeric"
"numeric"
  .. .. ..- attr(*, "names")= chr [1:3] "yv" "xv" "I(xv^2)"
 $ model      :'data.frame': 31 obs. of 3 variables:
  ..$ yv    : num [1:31] 0.802 1.035 7.811 4.138 3.901 ...
  ..$ xv    : int [1:31] 0 1 2 3 4 5 6 7 8 9 ...
  ..$ I(xv^2):Class 'AsIs' num [1:31] 0 1 4 9 16 25 36 49 64 81 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 yv ~ xv +
I(xv^2)
  .. .. ..- attr(*, "variables")= language list(yv, xv, I(xv^2))
  .. .. ..- attr(*, "factors")= int [1:3, 1:2] 0 1 0 0 0 1
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:3] "yv" "xv" "I(xv^2)"
  .. .. .. .. ..$ : chr [1:2] "xv" "I(xv^2)"
  .. .. ..- attr(*, "term.labels")= chr [1:2] "xv" "I(xv^2)"
  .. .. ..- attr(*, "order")= int [1:2] 1 1
  .. .. ..- attr(*, "intercept")= int 1
  .. .. ..- attr(*, "response")= int 1
  .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. .. ..- attr(*, "predvars")= language list(yv, xv, I(xv^2))
  .. .. ..- attr(*, "dataClasses")= Named chr [1:3] "numeric" "numeric"
"numeric"
  .. .. .. ..- attr(*, "names")= chr [1:3] "yv" "xv" "I(xv^2)"
 - attr(*, "class")= chr "lm"
```

This complex structure consists of a list of 12 objects, covering every detail of the model from the variables, their values, the coefficients, the residuals, the effects, and so on.

Reference

Crawley, M.J. (2013) *The R Book*, 2nd edn, John Wiley & Sons, Chichester.

Further Reading

Chambers, J.M. and Hastie, T.J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole, Pacific Grove, CA.

R Development Core Team (2014) *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Avaialble from http://www.R-project.org.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S-PLUS*, 4th edn, Springer-Verlag, New York.