# 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. Likewise, there is no obvious order in which different people will want to learn the different components of the R language. I suggest that you quickly scan down the following bullet points, which represent the order in which I have chosen to present the introductory material, and if you are relatively experienced in statistical computing, you might want to skip directly to the relevant section. I strongly recommend that beginners work thorough the material in the order presented, because successive sections build upon knowledge gained from previous sections. This chapter is divided into the following sections:

Other essential material is elsewhere: beginners will want to master data input (Chapter 3), dataframes (Chapter 4) and graphics (Chapter 5).

## 2.1 Calculations

The screen prompt `>` is an invitation to put R to work. The convention in this book is that material that you need to type into the command line after the screen prompt is shown in red in Courier New font. Just press the Return key to see the answer. You can use the command line as a calculator, like this:

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

```
[1] 1.749795
```

Each line can have at most 8192 characters, but if you want to see a lengthy instruction or a complicated expression on the screen, 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 `+, as follows:`

```
> 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 `>` will be omitted. The output from R is shown in blue 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.

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
```

For very big numbers or very small numbers R uses the following scheme (called exponents):

| | |
|---|---|
| `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. |

### 2.1.1 Complex numbers in R

Complex numbers consist of a real part and an imaginary part, which is identified by lower-case `i` like this:

```
z <- 3.5-8i
```

The elementary trigonometric, logarithmic, exponential, square root and hyperbolic functions are all implemented for complex values. The following are the special R functions that you can use with complex numbers. Determine the real part:

```
Re(z)
```

```
[1] 3.5
```

Determine the imaginary part:

```
Im(z)
```

```
[1] -8
```

Calculate the modulus (the distance from `z` to 0 in the complex plane by Pythagoras; if `x` is the real part and `y` is the imaginary part, then the modulus is $\sqrt{x^2 + y^2}$):

```
Mod(z)
```

```
[1] 8.732125
```

Calculate the argument (`Arg(x+ yi)= atan(y/x)`):

```
Arg(z)
```

```
[1] -1.158386
```

Work out the complex conjugate (change the sign of the imaginary part):

```
Conj(z)
```

```
[1] 3.5+8i
```

Membership and coercion are dealt with in the usual way (p. 30):

```
is.complex(z)
```

```
[1] TRUE
```

```
as.complex(3.8)
```

```
[1] 3.8+0i
```

### 2.1.2 Rounding

Various sorts of rounding (rounding up, rounding down, rounding to the nearest integer) can be done easily. Take the number 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 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
```

The hard part is deciding how you want to round negative numbers, because the concept of up and down is more subtle (remember that –5 is a bigger number than –6). You need to think, instead, of whether you want to round towards zero or away from zero. For negative numbers, rounding up means rounding *towards zero* so do not be surprised when the value of the positive part is different:

```
ceiling(-5.7)

[1] -5
```

With floor, negative values are rounded *away* from zero:

```
floor(-5.7)

[1] -6
```

You can simply strip off the decimal part of the number using the function `trunc`, which returns the integers formed by truncating the values in `x` towards zero:

```
trunc(5.7)

[1] 5

trunc(-5.7)

[1] -5
```

There is an R function called `round` that you can use by specifying 0 decimal places in the second argument:

```
round(5.7,0)

[1] 6

round(5.5,0)

[1] 6

round(5.4,0)

[1] 5

round(-5.7,0)

[1] -6
```

The number of decimal places is not the same as the number of significant digits. You can control the number of significant digits in a number using the function `signif`. Take a big number like 12 345 678 (roughly

12.35 million). Here is what happens when we ask for 4, 5 or 6 significant digits:

```
signif(12345678,4)
```

```
[1] 12350000
```

```
signif(12345678,5)
```

```
[1] 12346000
```

```
signif(12345678,6)
```

```
[1] 12345700
```

and so on. Why you would want to do this would need to be explained.

### 2.1.3 Arithmetic

The screen prompt in R is a fully functional calculator. You can add and subtract using the obvious `+` and `-` symbols, while division is achieved with a forward slash `/` and multiplication is done by using an asterisk `*` like this:

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

```
[1] 0
```

Notice from this example that multiplication ($5 \times 2$) is done *before* the additions and subtractions. Powers (like squared or cube root) use the caret symbol `^` and are done before multiplication or division, as you can see from this example:

```
3^2 / 2
```

```
[1] 4.5
```

All the mathematical functions you could ever want are here (see Table 2.1). The `log` function gives logs to the base e (e = 2.718 282), 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`:

```
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
```

**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)` | $x! = x \times (x - 1) \times (x - 2) \times \cdots \times 3 \times 2$ |
| `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 less than $x$ |
| `ceiling(x)` | smallest integer greater than $x$ |
| `trunc(x)` | closest integer to $x$ between $x$ and 0, e.g. 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 |

The trigonometric functions in R measure angles in radians. A circle is $2\pi$ radians, and this is 360°, so a right angle (90°) is $\pi/2$ radians. R knows the value of $\pi$ 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. 23).

### 2.1.4 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 (which it is), then ask:

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

```
[1] TRUE
```

Note the use of 'double equals' to test for equality (this is explained in detail on p. 26).

### 2.1.5 Variable names and assignment

There are three important things to remember when selecting names for your variables in R:

- Variable names in R are case sensitive, so `y` is not the same as `Y`.
- 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`).

In terms of your work–life balance, make your variable names as short as possible, so that you do not spend most of your time typing, and the rest of your time correcting spelling mistakes in your ridiculously long variable names.

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` where there is a space between the 'less than' and 'minus' symbol. In R, this is actually a question, asking 'is `x` less than minus 5?' and, depending on the current value of `x`, would evaluate to the answer either `TRUE` or `FALSE`.

### 2.1.6 Operators

R uses the following operator tokens:

| | |
|---|---|
| `+ - * / %/% %% ^` | arithmetic (plus, minus, times, divide, integer quotient, modulo, power) |
| `>= < <= == !=` | relational (greater than, greater than or equals, less than, less than or equals, equals, not equals) |
| `! & \|` | logical (not, and, or) |
| `~` | model formulae ('is modelled as a function of') |
| `<- ->` | assignment (gets) |
| `$` | list indexing (the 'element name' operator) |
| `:` | create a sequence |

Several of these operators have different meaning inside model formulae. Thus `*` indicates the main effects plus interaction (rather than multiplication), : indicates the interaction between two variables (rather than generate a sequence) and ^ means all interactions up to the indicated power (rather than raise to the power). You will learn more about these ideas in Chapter 9.

### 2.1.7 Integers

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that small integer data can be represented exactly and compactly. The range of integers is from −2 000 000 000 to +2 000 000 000 (`-2*10^9` to `+2*10^9`, which R could portray as `-2e+09` to `2e+09`).

Be careful. Do not try to change the class of a vector by using the `integer` function. Here is a numeric vector of whole numbers that you want to convert into a vector of integers:

```
x <- c(5,3,7,8)
is.integer(x)
```

```
[1] FALSE
```

```
is.numeric(x)
```

```
[1] TRUE
```

Applying the integer function to it replaces all your numbers with zeros; definitely not what you intended.

```
x <- integer(x)
x
```

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

Make the numeric object first, then convert the object to integer using the `as.integer` function like this:

```
x <- c(5,3,7,8)
x <- as.integer(x)
is.integer(x)
```

```
[1] TRUE
```

The integer function works as `trunc` when applied to real numbers, and removes the imaginary part when applied to complex numbers:

```
as.integer(5.7)
```

```
[1] 5
```

```
as.integer(-5.7)
```

```
[1] -5
```

```
as.integer(5.7 -3i)
```

```
[1] 5
Warning message:
imaginary parts discarded in coercion
```

### 2.1.8 Factors

Factors are categorical variables that have a fixed number of levels. A simple example of a factor might be a variable called gender with two levels: ‘female’ and ‘male’. If you had three females and two males, you could create the factor like this:

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

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

```
mode(gender)
```

```
[1] "numeric"
```

More often, you will create a dataframe by reading your data from a file using `read.table`. When you do this, all variables containing one or more character strings will be converted automatically into factors. Here is an example:

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

```
  Growth.rate Water Detergent Daphnia
1    2.919086  Tyne    BrandA  Clone1
2    2.492904  Tyne    BrandA  Clone1
3    3.021804  Tyne    BrandA  Clone1
4    2.350874  Tyne    BrandA  Clone2
5    3.148174  Tyne    BrandA  Clone2
6    4.423853  Tyne    BrandA  Clone2
```

This dataframe contains a continuous response variable (`Growth.rate`) and three categorical explanatory variables (`Water`, `Detergent` and `Daphnia`), all of which are factors. In statistical modelling, factors are associated with analysis of variance (all the explanatory variables are categorical) and analysis of covariance (some of the explanatory variables are categorical and some are continuous).

There are some important functions for dealing with factors. You will often want to check that a variable is a factor (especially if the factor levels are numbers rather than characters):

```
is.factor(Water)
```

```
 [1] TRUE
```

To discover the *names* of the factor levels, we use the `levels` function:

```
levels(Detergent)
```

```
[1] "BrandA" "BrandB" "BrandC" "BrandD"
```

To discover the *number* of levels of a factor, we use the `nlevels` function:

```
nlevels(Detergent)
```

```
[1] 4
```

The same result is achieved by applying the `length` function to the levels of a factor:

```
length(levels(Detergent))
```

```
[1] 4
```

By default, factor levels are treated in alphabetical order. If you want to change this (as you might, for instance, in ordering the bars of a bar chart) then this is straightforward: just type the factor levels in the order that you want them to be used, and provide this vector as the second argument to the `factor` function.

Suppose we have an experiment with three factor levels in a variable called `treatment`, and we want them to appear in this order: ‘nothing’, ‘single’ dose and ‘double’ dose. We shall need to override R’s natural tendency to order them ‘double’, ‘nothing’, ‘single’:

```
frame <- read.table("c:\\temp\\trial.txt",header=T)
attach(frame)
tapply(response,treatment,mean)
```

```
 double nothing single
     25      60     34
```

This is achieved using the `factor` function like this:

```
treatment <- factor(treatment,levels=c("nothing","single","double"))
```

Now we get the order we want:

```
tapply(response,treatment,mean)
```

```
nothing single double
     60     34     25
```

Only `==` and `!=` can be used for factors. Note, also, that a factor can only be compared to another factor with an identical set of levels (not necessarily in the same ordering) or to a character vector. For example, you cannot ask quantitative questions about factor levels, like `>` or `<=`, even if these levels are numeric.

To turn factor levels into numbers (integers) use the `unclass` function like this:

```
as.vector(unclass(Daphnia))
```

```
 [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1
[39] 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3
```

**Table 2.2.** Logical and relational 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)` |

## 2.2 Logical operations

A crucial part of computing involves asking questions about things. Is one thing bigger than other? Are two things the same size? Questions can be joined together using words like 'and' 'or', 'not'. Questions in R typically evaluate to `TRUE` or `FALSE` but there is the option of a 'maybe' (when the answer is not available, `NA`). In R, `<` means 'less than', `>` means 'greater than', and `!` means 'not' (see Table 2.2).

### 2.2.1 TRUE and T with FALSE and F

You can use `T` for TRUE and `F` for FALSE, but you should be aware that `T` and `F` might have been allocated as variables. So this is obvious:

```
TRUE == FALSE
```

```
[1] FALSE
```

```
T == F
```

```
[1] FALSE
```

This, however, is not so obvious:

```
T <- 0
T == FALSE
```

```
[1] TRUE
```

```
F <- 1
TRUE == F
```

```
[1] TRUE
```

But now, of course, `T` is not equal to `F`:

```
T != F
```

```
[1] TRUE
```

To be sure, always write TRUE and FALSE in full, and never use T or F as variable names.

### 2.2.2 Testing for equality with real numbers

There are international standards for carrying out floating point arithmetic, but on your computer these standards are beyond the control of R. Roughly speaking, integer arithmetic will be exact between $-10^{16}$ and $10^{16}$, but for fractions and other real numbers we lose accuracy because of round-off error. This is only likely to become a real problem in practice if you have to **subtract** similarly sized but very large numbers. A dramatic loss in accuracy under these circumstances is called ‘catastrophic cancellation error’. It occurs when an operation on two numbers increases *relative error* substantially more than it increases *absolute error*.

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 an accuracy of 53 binary digits. 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
```

In fact, they are not the same. We can see by how much the two values differ by subtraction:

```
x * x - 2
```

```
[1] 4.440892e-16
```

This is not a big number, but it is not zero either. So how do we test for equality of real numbers? The best advice is not to do it. Try instead to use the alternatives ‘less than’ with ‘greater than or equal to’, or conversely ‘greater than’ with ‘less than or equal to’. Then you will not go wrong. Sometimes, however, you really do want to test for equality. In those circumstances, do not use double equals to test for equality, but employ the `all.equal` function instead.

### 2.2.3 Equality of floating point numbers using `all.equal`

The nature of floating point numbers used in computing is the cause of some initially perplexing features. You would imagine that since 0.3 minus 0.2 is 0.1, and the logic presented below would evaluate to TRUE. Not so:

```
x <- 0.3 - 0.2
y <- 0.1
x == y
```

```
[1] FALSE
```

The function called `identical` gives the same result.

```
identical(x,y)
```

```
[1] FALSE
```

The solution is to use the function called `all.equal` which allows for insignificant differences:

```
all.equal(x,y)
```

```
[1] TRUE
```

Do not use `all.equal` directly in `if` expressions. Either use `isTRUE(all.equal(....))` or `identical` as appropriate.

### 2.2.4 Summarizing differences between objects using `all.equal`

The function `all.equal` is very useful in programming for checking that objects are as you expect them to be. Where differences occur, `all.equal` does a useful job in describing all the differences it finds. Here, for instance, it reports on the difference between `a` which is a vector of characters and `b` which is a factor:

```
a <- c("cat","dog","goldfish")
b <- factor(a)
```

In the `all.equal` function, the object on the left (`a`) is called the 'target' and the object on the right (`b`) is 'current':

```
all.equal(a,b)
```

```
[1] "Modes: character, numeric"
[2] "Attributes: < target is NULL, current is list >"
[3] "target is character, current is factor"
```

Recall that factors are stored internally as integers, so they have `mode = numeric`.

```
class(b)
```

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

```
mode(b)
```

```
[1] "numeric"
```

The reason why 'current is list' in line [2] of the output is that factors have two attributes and these are stored as a list – namely, their levels and their class:

```
attributes(b)
```

```
$levels
[1] "cat" "dog" "goldfish"

$class
[1] "factor"
```

The `all.equal` function is also useful for obtaining feedback on differences in things like the lengths of vectors:

```
n1 <- c(1,2,3)
n2 <- c(1,2,3,4)
all.equal(n1,n2)
```

```
[1] "Numeric: lengths (3, 4) differ"
```

It works well, too, for multiple differences:

```
n2 <- as.character(n2)
all.equal(n1,n2)
```

```
[1] "Modes: numeric, character"
[2] "Lengths: 3, 4"
[3] "target is numeric, current is character"
```

Note that 'target' is the first argument to the function and 'current' is the second. If you supply more than two objects to be compared, the third and subsequent objects are simply ignored.

### 2.2.5 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 2.2) 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)
```

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`.

### 2.2.6 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
```

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
[7]  0.0000000
```

Logical arithmetic is particularly useful in generating simplified factor levels during statistical modelling. Suppose we want to reduce a five-level factor (a, b, c, d, e) 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 equals sign without a space in between (`==`). 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. |

## 2.3 Generating sequences

An important way of creating vectors is to generate a sequence of numbers. The simplest sequences are in steps of 1, and the colon operator is the simplest way of generating such sequences. All you do is specify the first and last values separated by a colon. Here is a sequence from 0 up to 10:

```
0:10
```

```
 [1]  0 1 2 3 4 5 6 7 8 9 10
```

Here is a sequence from 15 down to 5:

```
15:5
```

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

To generate a sequence in steps other than 1, you use the `seq` function. There are various forms of this, of which the simplest has three arguments: `from, to, by` (the initial value, the final value and the increment). If the initial value is smaller than the final value, the increment should be positive, like this:

```
seq(0, 1.5, 0.1)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5
```

If the initial value is larger than the final value, the increment should be negative, like this:

```
seq(6,4,-0.2)
```

```
 [1] 6.0 5.8 5.6 5.4 5.2 5.0 4.8 4.6 4.4 4.2 4.0
```

In many cases, you want to generate a sequence to match an existing vector in length. Rather than having to figure out the increment that will get from the initial to the final value and produce a vector of exactly the appropriate length, R provides the `along` and `length` options. Suppose you have a vector of population sizes:

```
N <- c(55,76,92,103,84,88,121,91,65,77,99)
```

You need to plot this against a sequence that starts at 0.04 in steps of 0.01:

```
seq(from=0.04,by=0.01,length=11)
```

```
 [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

But this requires you to figure out the length of `N`. A simpler method is to use the `along` argument and specify the vector, `N`, whose length has to be matched:

```
seq(0.04,by=0.01,along=N)
```

```
 [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

Alternatively, you can get R to work out the increment (0.01 in this example), by specifying the start and the end values (`from` and `to`), and the name of the vector (`N`) whose length has to be matched:

```
seq(from=0.04,to=0.14,along=N)
```

```
 [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

An important application of the last option is to get the $x$ values for drawing smooth lines through a scatterplot of data using predicted values from a model (see p. 207).

Notice that when the increment does not match the final value, then the generated sequence stops short of the last value (rather than overstepping it):

```
seq(1.4,2.1,0.3)
```

```
[1] 1.4 1.7 2.0
```

If you want a vector made up of sequences of unequal lengths, then use the `sequence` function. Suppose that most of the five sequences you want to string together are from 1 to 4, but the second one is 1 to 3 and the last one is 1 to 5, then:

```
sequence(c(4,3,4,4,4,5))
```

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

### 2.3.1 Generating 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
[13] 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 appears 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 three 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 a more complicated 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
```

Here is the most complex case with character data rather than numbers: each element of the series is repeated an irregular number of times:

```
rep(c("cat","dog","gerbil","goldfish","rat"),c(2,3,2,1,3))
```

```
[1] "cat"    "cat"      "dog" "dog" "dog" "gerbil"
[7] "gerbil" "goldfish" "rat" "rat" "rat"
```

This is the most general, and also the most useful form of the `rep` function.

### 2.3.2 Generating 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: ‘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
[13]  1  1  1  2  2  2  3  3  3  4  4  4

Levels:   1  2  3  4
```

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

```
Temp <- gl(2, 2, 24, labels = c("Low", "High"))
Soft <- gl(3, 8, 24, labels = c("Hard","Medium","Soft"))
M.user <- gl(2, 4, 24, labels = c("N", "Y"))
Brand <- gl(2, 1, 24, labels = c("X", "M"))
```

```
data.frame(Temp,Soft,M.user,Brand)
```

```
Temp Soft M.user Brand
1    Low  Hard   N X
2    Low  Hard   N M
3    High Hard   N X
4    High Hard   N M
5    Low  Hard   Y X
6    Low  Hard   Y M
7    High Hard   Y X
8    High Hard   Y M
9    Low  Medium N X
```

```
10   Low  Medium N  M
11   High Medium N  X
12   High Medium N  M
13   Low  Medium Y  X
14   Low  Medium Y  M
15   High Medium Y  X
16   High Medium Y  M
17   Low  Soft   N  X
18   Low  Soft   N  M
19   High Soft   N  X
20   High Soft   N  M
21   Low  Soft   Y  X
22   Low  Soft   Y  M
23   High Soft   Y  X
24   High Soft   Y  M
```

## 2.4 Membership: Testing and coercing in R

The concepts of membership and coercion may be unfamiliar. Membership relates to the class of an object in R. Coercion changes the class of an object. For instance, a logical variable has class `logical` and mode `logical`. This is how we create the variable:

```
lv <- c(T,F,T)
```

We can assess its membership by asking if it is a logical variable using the `is.logical` function:

```
is.logical(lv)
```

```
[1] TRUE
```

It is not a factor, and so it does not have levels:

```
levels(lv)
NULL
```

But we can coerce it be a two-level factor like this:

```
(fv <- as.factor(lv))
```

```
[1] TRUE FALSE TRUE
Levels: FALSE TRUE
```

```
is.factor(fv)
```

```
[1] TRUE
```

We can coerce a logical variable to be numeric: `TRUE` evaluates to 1 and `FALSE` evaluates to zero, like this:

```
(nv <- as.numeric(lv))
```

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

This is particularly useful as a shortcut when creating new factors with reduced numbers of levels (as we do in model simplification).

**Table 2.3.** 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 unless you overwrite its name.

| 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` |

In general, the expression `as(object, value)` is the way to coerce an object to a particular class. Membership functions ask `is.something` and coercion functions say `as.something`.

Objects have a type, and you can test the type of an object using an `is.type` function (Table 2.3). For instance, mathematical functions expect numeric input and text-processing 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 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
```

When vectors are created by calculation from other vectors, the new vector will be as long as the longest vector used in the calculation and the shorter variable will be 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 the longer vector (`A`) is not a multiple of the shorter vector (`B`).

## 2.5 Missing values, infinity and things that are not numbers

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
```

### 2.5.1 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.

You may want to discover which values in a vector are missing. Here is a simple case:

```
y <- c(4,NA,7)
```

The missing value question should evaluate to `FALSE TRUE FALSE`. There are two ways of looking for missing values that you might think should work, but do not. These involve treating `NA` as if it was a piece of text and using double equals (`==`) to test for it. So this does not work:

```
y == NA
```

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

because it turns *all* the values into `NA` (definitively not what you intended). This does not work either:

```
y == "NA"
```

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

It correctly reports that the numbers are not character strings, but it returns `NA` for the missing value itself, rather than `TRUE` as required. This is how you do it properly:

```
is.na(y)
```

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

To produce a vector with the `NA` stripped out, use subscripts with the not `!` operator like this:

```
y[! is.na(y)]
```

```
[1] 4 7
```

This syntax is useful in editing out rows containing missing values from large dataframes. Here is a very simple example of a dataframe with four rows and four columns:

```
y1 <- c(1,2,3,NA)
y2 <- c(5,6,NA,8)
y3 <- c(9,NA,11,12)
y4 <- c(NA,14,15,16)
```

```
full.frame <- data.frame(y1,y2,y3,y4)
```

```
reduced.frame <- full.frame[!is.na(full.frame$y1),]
```

so the new `reduced.frame` will have fewer rows than `full.frame` when the variable in `full.frame` called `full.frame$y1` contains one or more missing values.

```
reduced.frame
```

```
  y1 y2  y3  y4
1  1  5   9  NA
2  2  6  NA  14
3  3 NA  11  15
```

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
```

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, like this:

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

```
[1]  7  8
```

However, the result is achieved more simply using the which function 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
```

Be very careful when doing this, because most missing values are not genuine zeros.

## 2.6 Vectors and subscripts

A vector is a variable with one or more values of the same type. For instance, the numbers of peas in six pods were 4, 7, 6, 5, 6 and 7. The vector called peas is one object of length = 6. In this case, the class of the object is numeric. The easiest way to create a vector in R is to concatenate (link together) the six values using the concatenate function, c, like this:

```
peas <- c(4, 7, 6, 5, 6, 7)
```

We can ask all sorts of questions about the vector called peas. For instance, what type of vector is it?

```
class(peas)
```

```
[1] "numeric"
```

How big is the vector?

```
length(peas)
```

```
[1] 6
```

The great advantage of a vector-based language is that it is very simple to ask quite involved questions that involve all of the values in the vector. These vector functions are often self-explanatory:

```
mean(peas)
```

```
[1] 5.833333
```

```
max(peas)
```

```
[1] 7
```

```
min(peas)
```

```
[1] 4
```

Others might be more opaque:

```
quantile(peas)
```

```
  0%  25%  50%  75% 100%
4.00 5.25 6.00 6.75 7.00
```

Another way to create a vector is to input data from the keyboard using the function called `scan`:

```
peas <- scan()
```

The prompt appears `1:` which means type in the first number of peas (4) then press the return key, then the prompt `2:` appears (you type in 7) and so on. When you have typed in all six values, and the prompt `7:` has appeared, you just press the return key to tell R that the vector is now complete. R replies by telling you how many items it has read:

```
1: 4
2: 7
3: 6
4: 5
5: 6
6: 7
7:
```

```
Read 6 items
```

For more realistic applications, the usual way of creating vectors is to read the data from a pre-prepared computer file (as described in Chapter 3).

### 2.6.1 Extracting elements of a vector using subscripts

You will often want to use some but not all of the contents of a vector. To do this, you need to master the use of subscripts (or indices as they are also known). In R, subscripts involve the use of square brackets []. Our vector called `peas` shows the numbers of peas in six pods:

```
peas
```

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

The first element of `peas` is 4, the second 7, and so on. The elements are indexed left to right, 1 to 6. It could not be more straightforward. If we want to extract the fourth element of peas (which you can see is a 5) then this is what we do:

```
peas[4]
```

```
[1] 5
```

If we want to extract several values (say the 2nd, 3rd and 6th) we use a vector to specify the pods we want as subscripts, either in two stages like this:

```
pods <- c(2,3,6)
peas[pods]
```

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

or in a single step, like this:

```
peas[c(2,3,6)]
```

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

You can drop values from a vector by using negative subscripts. Here are all but the first values of `peas`:

```
peas[-1]
```

```
[1] 7 6 5 6 7
```

Here are all but the last (note the use of the `length` function to decide what is last):

```
peas[-length(peas)]
```

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

We can use these ideas to write a function called `trim` to remove (say) the largest two and the smallest two values from a vector called `x`. First we have to `sort` the vector, then remove the smallest two values (these will have subscripts 1 and 2), then remove the largest two values (which will have subscripts `length(x)` and `length(x)-1`):

```
trim <- function(x) sort(x)[-c(1,2,length(x)-1,length(x))]
```

We can use `trim` on the vector called `peas`, expecting to get 6 and 6 as the result:

```
trim(peas)
```

```
[1] 6 6
```

Finally, we can use sequences of numbers to extract values from a vector. Here are the first three values of `peas`:

```
peas[1:3]
```

```
[1] 4 7 6
```

Here are the even-numbered values of `peas`:

```
peas[seq(2,length(peas),2)]
```

```
[1] 7 5 7
```

or alternatively:

```
peas[1:length(peas) %% 2 == 0]
```

```
[1] 7 5 7
```

using the modulo function `%%` on the sequence 1 to 6 to extract the even numbers 2, 4 and 6. Note that vectors in R could have length 0, and this could be useful in writing functions:

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

```
[1]  0
```

### 2.6.2 Classes of vector

The vector called `peas` contained numbers: in the jargon, it is of class `numeric`. R allows vectors of six types, so long as all of the elements in one vector belong to the same class. The classes are logical, integer, real, complex, string (or character) or raw. You will use numeric, logical and character variables all the time. Engineers and mathematicians will use complex numbers. But you could go a whole career without ever needing to use integer or raw.

### 2.6.3 Naming 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
```

### 2.6.4 Working with 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. 30). 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 do not 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 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 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 reverse-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
[13]  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
[13]  3   3   2
```

So the answer to our problem is $11 + 10 + 9 = 30$. But how to compute this? 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.

You will often want to find out which value in a vector is the maximum or the minimum. This is a question about indices, and the answer you want is an integer indicating which element of the vector contains the maximum (or minimum) out of all the values in that vector. Here is the vector:

```
x <- c(2,3,4,1,5,8,2,3,7,5,7)
```

So the answers we want are 6 (the maximum) and 4 (the minimum). The slow way to do it is like this:

```
which(x == max(x))
```

```
[1] 6
```

```
which(x == min(x))
```

```
[1] 4
```

Better, however, to use the much quicker built-in functions `which.max` or `which.min` like this:

```
which.max(x)
```

```
[1] 6
```

```
which.min(x)
```

```
[1] 4
```

## 2.7 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. The most important vector functions are listed in Table 2.4. Here is a numeric vector:

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

Some vector functions produce a single number:

```
mean(y)
```

```
[1] 6.333333
```

**Table 2.4.** 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 *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* |

Others produce two numbers:

```
range(y)
```

```
[1]  2 11
```

here showing that the minimum was 2 and the maximum was 11. Other functions produce several numbers:

```
fivenum(y)
```

```
[1]  2.0  4.0  6.0  8.5  11.0
```

This is Tukey's famous five-number summary: the minimum, the lower hinge, the median, the upper hinge and the maximum (the hinges are explained on p. 346).

Perhaps the single most useful vector function in R is `table`. You need to see it in action to appreciate just how good it is. Here is a huge vector called `counts` containing 10 000 random integers from a negative binomial distribution (counts of fungal lesions on 10 000 individual leaves, for instance):

```
counts <- rnbinom(10000,mu=0.92,size=1.1)
```

Here is a look at the first 30 values in `counts:`

```
counts[1:30]
```

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

The question is this: how many zeros are there in the whole vector of 10 000 numbers, how many 1s, and so on right up to the largest value within counts? A formidable task for you or me, but for R it is just:

```
table(counts)
```

```
counts
   0    1    2   3   4   5  6  7  8 9 10 11 13
5039 2574 1240 607 291 141 54 29 11 9  3  1  1
```

There were 5039 zeros, 2574 ones, and so on up the largest counts (there was one 11 and one 13 in this realization; you will have obtained different random numbers on your computer).

### 2.7.1 Obtaining tables of means using `tapply`

One of the most important functions in all of R is `tapply`. It does not sound like much from the name, but you will use it time and again for calculating means, variances, sample sizes, minima and maxima. With weather data, for instance, we might want the 12 monthly mean temperatures rather than the whole-year average. We have a response variable, `temperature`, and a categorical explanatory variable, `month`:

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

```
[1] "temperature" "lower" "rain" "month" "yr"
```

The function that we want to apply is `mean`. All we do is invoke the `tapply` function with three arguments: the response variable, the categorical explanatory variable and the name of the function that we want to apply:

```
tapply(temperature,month,mean)
```

```
       1         2         3         4         5         6
```

```
7.930051 8.671136 11.200508 13.813708 17.880847 20.306151
       7         8         9        10        11       12
22.673854 23.104924 19.344211 15.125976 10.720702 8.299830
```

It is easy to apply other functions in the same way: here are the monthly variances

```
tapply(temperature,month,var)
```

and the monthly minima

```
tapply(temperature,month,min)
```

If R does not have a built in function to do what you want (Table 2.4), then you can easily write your own. Here, for instance, is a function to calculate the standard error of each mean (these are called anonymous functions in R, because they are unnamed):

```
tapply(temperature,month,function(x) sqrt(var(x)/length(x)))
```

```
        1         2         3         4         5         6
0.1401489 0.1414445 0.1358934 0.1476242 0.1673197 0.1596439

        7         8         9        10        11        12
0.1539661 0.1516091 0.1309294 0.1155612 0.1291703 0.1398438
```

The `tapply` function is very flexible. It can produce multi-dimensional tables simply by replacing the one categorical variable (`month`) by a `list` of categorical variables. Here are the monthly means calculated separately for each year, as specified by `list(yr,month)`. The variable you name first in the list (`yr`) will appear as the row of the results table and the second will appear as the columns (`month`):

```
tapply(temperature,list(yr,month),mean)[,1:6]
```

```
             1         2         3        4        5        6
1987  3.170968  6.871429  8.132258 14.92667 15.60645 17.73667
1988  8.048387  8.248276  9.959375 12.74483 17.31935 18.71667
1989  8.841935  9.482143 11.919355 11.09333 20.40323 21.23667
1990  9.445161 11.028571 12.487097 13.80000 20.16129 18.51667
1991  6.980645  4.817857 12.022581 13.14333 15.58065 16.88000
1992  6.964516  8.686207 11.477419 13.35000 20.45806 22.21667
1993 10.119355  6.985714 11.209677 14.17000 17.79355 21.10000
1994  8.825806  7.217857 11.806452 12.61667 16.23226 20.86000
1995  8.309677 10.439286 10.667742 14.79667 18.74063 19.94483
1996  7.019355  6.065517  8.487097 13.99667 14.38710 21.93667
1997  4.932258 10.178571 13.370968 15.00667 18.17419 19.93000
1998  8.759375 11.242857 11.719355 12.55333 19.43226 19.35000
1999  9.523333  8.485714 11.790323 14.65000 18.94839 20.00667
2000  8.229032 10.324138 11.900000 12.59000 18.22581 20.63333
2001  7.067742  9.121429  9.012903 12.65667 18.96452 20.52667
2002  9.067742 11.396429 12.319355 15.68667 16.81290 19.67667
2003  8.012903  8.171429 13.425806 15.69000 17.36452 22.80000
2004  8.261290  8.993103 10.354839 15.17000 17.98065 21.73667
2005  9.116129  7.032143 10.787097 13.78333 17.12258 22.00000
```

The subscripts `[,1:6]` simply restrict the output to the first six months. You can see at once that January (month 1) 1993 was exceptionally warm and January 1987 exceptionally cold.

There is just one thing about `tapply` that might confuse you. If you try to apply a function that has built-in protection against missing values, then `tapply` may not do what you want, producing `NA` instead of the numerical answer. This is most likely to happen with the mean function because its default is to produce `NA` when there are one or more missing values. The remedy is to provide an extra argument to `tapply`, specifying that you want to see the average of the non-missing values. Use `na.rm=TRUE` to remove the missing values like this:

```
tapply(temperature,yr,mean,na.rm=TRUE)
```

You might want to trim some of the extreme values before calculating the mean (the arithmetic mean is famously sensitive to outliers). The `trim` option allows you to specify the fraction of the data (between 0 and 0.5) that you want to be omitted from the left- and right-hand tails of the sorted vector of values before computing the mean of the central values:

```
tapply(temperature,yr,mean,trim=0.2)
```

```
    1987     1988     1989     1990     1991     1992     1993
13.46000 13.74500 14.99726 15.16301 13.92237 14.32091 14.28000
```

### 2.7.2 The aggregate function for grouped summary statistics

Suppose that we have two response variables (`y` and `z`) and two explanatory variables (`x` and `w`) that we might want to use to summarize functions like mean or variance of `y` and/or `z`. The `aggregate` function has a formula method which allows elegant summaries of four kinds:

| | |
|---|---|
| one to one | `aggregate(y ~ x, mean)` |
| one to many | `aggregate(y ~ x + w, mean)` |
| many to one | `aggregate(cbind(y,z) ~ x, mean)` |
| many to many | `aggregate(cbind(y,z) ~ x + w, mean)` |

This is very useful for removing pseudoreplication from dataframes. Here is an example using a dataframe with two continuous variables (`Growth.rate` and `pH`) and three categorical explanatory variables (`Water`, `Detergent` and `Daphnia`):

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

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

Here is one-to-one use of `aggregate` to find mean growth rate in the two water samples:

```
aggregate(Growth.rate~Water,data,mean)
```

```
  Water Growth.rate
1  Tyne    3.685862
2  Wear    4.017948
```

Here is a one-to-many use to look at the interaction between `Water` and `Detergent`:

```
aggregate(Growth.rate~Water+Detergent,data,mean)
```

```
  Water Detergent Growth.rate
1  Tyne    BrandA    3.661807
2  Wear    BrandA    4.107857
3  Tyne    BrandB    3.911116
4  Wear    BrandB    4.108972
5  Tyne    BrandC    3.814321
6  Wear    BrandC    4.094704
7  Tyne    BrandD    3.356203
8  Wear    BrandD    3.760259
```

Finally, here is a many-to-many use to find mean `pH` as well as mean `Growth.rate` for the interaction between `Water` and `Detergent`:

```
aggregate(cbind(pH,Growth.rate)~Water+Detergent,data,mean)
```

```
  Water Detergent       pH  Growth.rate
1  Tyne    BrandA 4.883908     3.661807
2  Wear    BrandA 5.054835     4.107857
3  Tyne    BrandB 5.043797     3.911116
4  Wear    BrandB 4.892346     4.108972
5  Tyne    BrandC 4.847069     3.814321
6  Wear    BrandC 4.912128     4.094704
7  Tyne    BrandD 4.809144     3.356203
8  Wear    BrandD 5.097039     3.760259
```

### 2.7.3 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
```

Thus 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, not vectors.

### 2.7.4 Summary information from vectors by groups

The vector function tapply is one of the most important and useful vector functions to master. 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. 528). 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. 245).

### 2.7.5 Addresses within vectors

There is an important function called which for finding addresses within vectors. The vector y looks like this:

```
y <- c(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 do not 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
```

### 2.7.6 Finding closest values

Finding the value in a vector that is closest to a specified value is straightforward using `which`. The vector `xv` contains 1000 random numbers from a normal distribution with mean = 100 and standard deviation = 10:

```
xv <- rnorm(1000,100,10)
```

Here, we want to find the value of `xv` that is closest to 108.0. The logic is to work out the difference between 108 and each of the 1000 random numbers, then find which of these differences is the smallest. This is what the R code looks like:

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

```
[1]  332
```

The closest value to 108.0 is in location 332 within `xv`. 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
```

Now we can write a function to return the closest value to a specified value (`sv`) in any vector (`xv`):

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

and run it like this:

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

```
[1]  108.0076
```

### 2.7.7 Sorting, ranking and ordering

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

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

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

We apply the three different functions to the vector called `Price`:

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

Then we 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, a price of 325, happens to be 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 $(8 + 9)/2 = 8.5$. The lowest price is 95, indicated by a rank of 1.

### *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 numbers in this column are subscripts between 1 and 12. 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 value of `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. 166. 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
```

Make sure you understand why some of the brackets are round and some are square.

### 2.7.8 Understanding the difference between `unique` and `duplicated`

The difference is best seen with a simple example. Here is a vector of names:

```
names <- c("Williams","Jones","Smith","Williams","Jones","Williams")
```

We can see how many times each name appears using `table`:

```
table(names)
```

```
names
 Jones  Smith  Williams
     2      1         3
```

It is clear that the vector contains just three different names. The function called `unique` extracts these three unique names, creating a vector of length 3, unsorted, in the order in which the names are encountered in the vector:

```
unique(names)
```

```
[1] "Williams" "Jones" "Smith"
```

In contrast, the function called `duplicated` produces a vector, of the same length as the vector of names, containing the logical values either `FALSE` or `TRUE`, depending upon whether or not that name has appeared already (reading from the left). You need to see this in action to understand what is happening, and why it might be useful:

```
duplicated(names)
```

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

The first three names are not duplicated (`FALSE`), but the last three are all duplicated (`TRUE`). We can mimic the `unique` function by using this vector as subscripts like this:

```
names[!duplicated(names)]
```

```
[1] "Williams" "Jones" "Smith"
```

Note the use of the NOT operator (`!`) in front of the `duplicated` function. There you have it: if you want a shortened vector, containing only the unique values in names, then use `unique`, but if you want a vector of the same length as names then use `duplicated`. You might use this to extract values from a different vector (salaries, for instance) if you wanted the mean salary, ignoring the repeats:

```
salary <- c(42,42,48,42,42,42)
mean(salary)
```

```
[1] 43
```

```
salary[!duplicated(names)]
```

```
[1] 42 42 48
```

```
mean(salary[!duplicated(names)])
```

```
[1] 44
```

Note that this is not the same answer as would be obtained by omitting the duplicate salaries, because two of the people (Jones and Williams) had the same salary (42). Here is the wrong answer:

```
mean(salary[!duplicated(salary)])
```

```
[1] 45
```

### 2.7.9 Looking for runs of numbers within vectors

The function called `rle`, which stands for 'run length encoding' 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 0 1 1 1 0 1 1 2 1 1 0 1 0 2 1 1 2 0 2 0 1 0 0 0 2 0
[36]  1 0 4 0 0 1 0 1 0 1 0 2 1 1 1 0 1 0 1 0 0 0 0 0 0 2 0 0 0 0 1 0 0 0
[71]  2 1 1 1 1 0 1 0 1 0 0 1 1 0 1 0 2 1 1 2 0 1 0 1 0 0 0 1 1 0 1 2 2 0 1
[106] 0 0 0 0 0 0 1 0 0 2 1 2 0 2 0 2 2 1 1 0 2 0 1 1 2 2 2 1 1 1 1 0 0 0 1
[141] 0 2 1 4 0 0 2 1 0 1
```

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 2 1 1 2 3 1 2 1 ...
 values : num [1:93] 1 0 2 1 0 1 0 1 2 1 ...
```

The object produced by `rle` is a list of two vectors: the lengths of the runs and the values that did the running. 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 (this is 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 then 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.

## 2.7.10 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 us 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"
```

## 2.8 Matrices and arrays

An array is a multi-dimensional object. The dimensions of an array are specified by its `dim` attribute, which gives the maximal indices in each dimension. So for a three-dimensional array consisting of 24 numbers in a sequence 1:24, with dimensions 2 × 4 × 3, we write:

```
y <- 1:24
dim(y) <- c(2,4,3)
y
```

```
, , 1

     [,1]  [,2]  [,3]  [,4]
[1,]    1     3     5     7
[2,]    2     4     6     8

, , 2

     [,1]  [,2]  [,3]  [,4]
[1,]    9    11    13    15
[2,]   10    12    14    16

, , 3

     [,1]  [,2]  [,3]  [,4]
[1,]   17    19    21    23
[2,]   18    20    22    24
```

This produces three two-dimensional tables, because the third dimension is 3. This is what happens when you change the dimensions:

```
dim(y) <- c(3,2,4)
y
```

```
, , 1

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

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12

, , 3

     [,1] [,2]
[1,]   13   16
[2,]   14   17
[3,]   15   18

, , 4

     [,1] [,2]
[1,]   19   22
[2,]   20   23
[3,]   21   24
```

Now we have four two-dimensional tables, each of three rows and two columns. Keep looking at these two examples until you are sure that you understand exactly what has happened here.

A matrix is a two-dimensional array containing numbers. A dataframe is a two-dimensional list containing (potentially a mix of) numbers, text or logical variables in different columns. When there are two subscripts [5,3] to an object like a matrix or a dataframe, the first subscript refers to the row number (5 in this example; the rows are defined as **margin** number 1) and the second subscript refers to the column number (3 in this example; 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.

### 2.8.1 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 column-wise. 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
```

### 2.8.2 Naming the rows and columns of matrices

At first, matrices have numbers naming their rows and columns (see above). Here is a $4 \times 5$ 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. 87) 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
```

### 2.8.3 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), calculated over all the rows (blank then comma),

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

```
[1]  2
```

or the variance of the bottom row, calculated over all of the columns (a blank in the second position),

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

```
[1]  0.7
```

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. 61). Remember that columns are margin number 2 and rows are margin number 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 example, 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 number 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.

### 2.8.4 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 × 3 × 1 × 4 is created by subscripting it will be coerced into a 2 × 3 × 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]
```

### 2.8.5 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. 254).

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 column-wise 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
```

### 2.8.6 Applying functions with `apply`, `sapply` and `lapply`

The `apply` function is used for applying functions to the rows or columns of matrices or dataframes. For example, here is a matrix with four rows and six columns:

```
(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. Often you want to apply a function across one of the margins of a matrix. Margin 1 refers to the rows and margin 2 to 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 six rows and four columns (i.e. it has been transposed).

You can supply your own function definition (here $x^2 + x$) 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 (rather than to the margin of a matrix) then use sapply . 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 complicated 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. 715) 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)
```

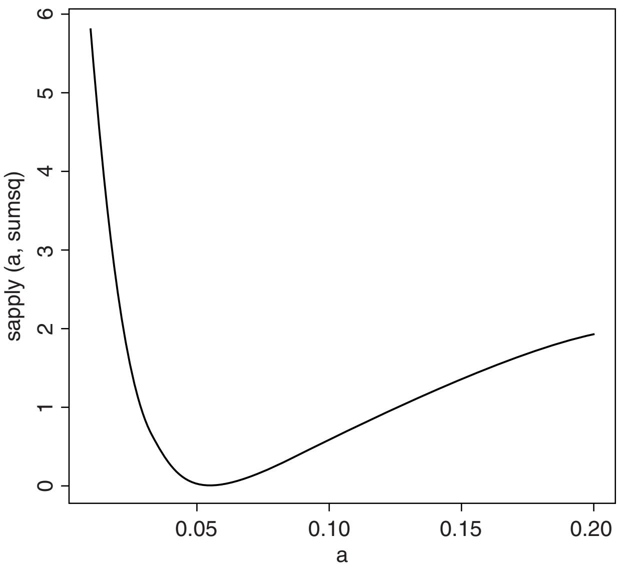
```
Coefficients:

(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")
```



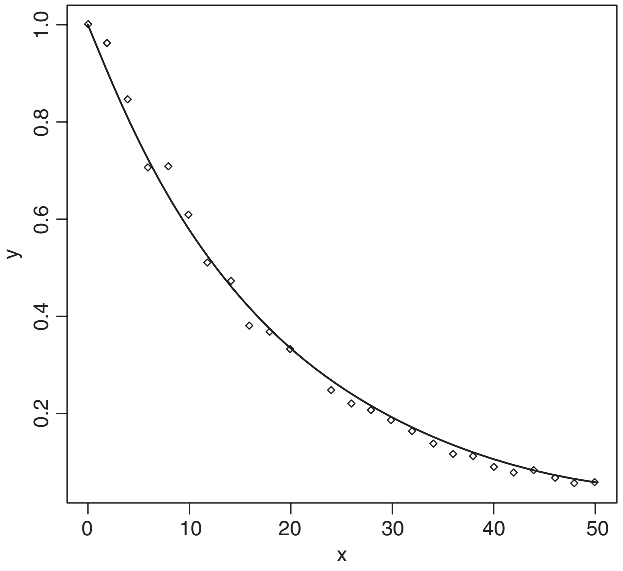
This shows that the least-squares estimate of a is indeed close to 0.06 (this is the value of a associated with the minimum deviance). To extract the minimum value of a we use min with subscripts (square brackets):

```
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 we 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. Sums of squares are not the only way of doing statistics, just the conventional way.

### 2.8.7 Using the `max.col` function

The task is to work out the number of plots on which a species is dominant in the Park Grass dataframe. This involves scanning each row of a matrix and reporting on the column number that contains the maximum value.

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

```
 [1] "AC"      "AE"   "AM"  "AO"  "AP"  "AR"    "AS"
 [8] "AU"      "BH"   "BM"  "CC"  "CF"  "CM"    "CN"
[15] "CX"      "CY"   "DC"  "DG"  "ER"  "FM"    "FP"
[22] "FR"      "GV"   "HI"  "HL"  "HP"  "HS"    "HR"
[29] "KA"      "LA"   "LC"  "LH"  "LM"  "LO"    "LP"
[36] "OR"      "PL"   "PP"  "PS"  "PT"  "QR"    "RA"
[43] "RB"      "RC"   "SG"  "SM"  "SO"  "TF"    "TG"
[50] "TO"      "TP"   "TR"  "VC"  "VK"  "plot"  "lime"
[57] "species" "hay"  "pH"
```

The species names are represented by two-letter codes (so, for example, 'AC' is *Agrostis capillaris*). We define the dominant as the species that has the maximum biomass on a given plot. The first task is to create a dataframe that contains only the species abundances (we do not want the plot numbers, or the treatments, or the values of any covariates). For the Park Grass data, the first 54 columns contain species abundance values, so we select all of the rows in the first 54 columns like this:

```
species <- data[,1:54]
```

Now we use the function `max.col` to go through all of the 89 rows, and for each row return the column number that contains the maximum biomass:

```
max.col(species)
```

```
 [1] 22 22 22  1 32 32 22  1 22 22 22  1 22 22  1  1 22 22 22  4  2  2 51 2  1
[26]  1 22 22  1  1  2  5  1  4  2  2  1  4 22 22 22  4  2  2 25 25  2  2 5 25
[51] 32  1 22 22  2  2  1  1 51  2  2 27  2  2  2  2 35 51 51  1  2  2  1 1 32
[76] 32  1  1  1  1  1  1 14  1  2  1  1  2  2
```

To get the identity of the dominant, we then extract the name of this column, using the index returned by `max.col` as a subscript to the object called `names(species)`:

```
names(species)[max.col(species)]
```

Finally, we use `table` to count up the total number of plots on which each species was dominant. The code looks like this:

```
table(names(species)[max.col(species)])
```

```
AC AE AO AP CN FR HL HS LH LP TP
26 23  4  2  1 19  3  1  5  1  4
```

So AC was dominant on more plots than any other species, with AE in second place and FR in third. The total number of species that were dominant on one or more plots is given by determining the length of this table:

```
length(table(names(species)[max.col(species)]))
```

```
[1] 11
```

So the number of species that were present in the system, but never attained dominance was
$54 - 11 = 43$:

```
length(names(species))-length(table(names(species)[max.col(species)]))
```

```
[1] 43
```

There is no such function as 'min.col', but you can easily emulate it by using `max.col` with the negatives of your data. It makes no sense to do it with this example, because several species are absent from every plot, and the function would just pick one of the absent species at random. But, anyway,

```
max.col(-species)
```

picks out the identity (the column number) of one of the zeros from each row of the dataframe. In a case where there was a unique minimum in each row, then this would find it.

### 2.8.8 Restructuring a multi-dimensional array using aperm

There are circumstances where you may want to reorder the dimensions of an array. Here is an example of an array with three dimensions: two sexes, three ages and four income groups. For simplicity and ease of illustration the values in the array are just the numbers 1 to 24 in order ($2 \times 3 \times 4 = 24$):

```
data <- array(1:24, 2:4)
```

The second argument to the `array` function specifies the number of levels in dimensions 1, 2, and 3 using the sequence-generator `2:4` to produce the numbers 2, 3 and 4. This is what the array looks like:

```
data

, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

There are four sub-tables, each with 2 rows and 3 columns. Now we give names to the factor levels in each of the three dimensions: these are called the `dimnames` attributes and are allocated as `lists` like this:

```
dimnames(data)[[1]] <- list("male","female")
dimnames(data)[[2]] <- list("young","mid","old")
dimnames(data)[[3]] <- list("A","B","C","D")
dimnames(data)

[[1]]
[1] "male" "female"

[[2]]
[1] "young" "mid" "old"

[[3]]
[1] "A" "B" "C" "D"
```

You can see the advantage of naming the dimensions by comparing the output of the array with (below) and without names (above):

```
data

, , A

       young  mid old
male       1    3   5
female     2    4   6

, , B

      young  mid old
male      7    9  11
female    8   10  12

, , C

       young  mid old
male      13   15  17
female    14   16  18

, , D

       young  mid old
male      19   21  23
female    20   22  24
```

Suppose, however, that we want the four income groups (A–D) to be the columns in each of the sub-tables, and the separate sub-tables to represent the two genders. This is a job for `aperm`. We need to specify the order 'age then income then gender' in terms of the order of their dimensions (row, column, sub-table, namely 2 then 3 then 1) like this:

```
new.data <- aperm(data,c(2,3,1))
new.data

, , male

      A  B  C  D
young 1  7 13 19
mid   3  9 15 21
old   5 11 17 23

, , female

      A  B  C  D
young 2  8 14 20
mid   4 10 16 22
old   6 12 18 24
```

This will be tricky to see at first, but you should persevere, because `aperm` is a very useful function.

## 2.9 Random numbers, sampling and shuffling

When debugging a program it is often useful to be able to get the same string of random numbers as you had last time. Use the `set.seed` function to control this:

```
set.seed(375)
runif(3)
```

```
[1] 0.9613669 0.6918535 0.7302684
```

```
runif(3)
```

```
[1] 0.9228566 0.1603804 0.9642799
```

```
runif(3)
```

```
[1] 0.52880907 0.08660864 0.29075809
```

If you reset the seed with the same value, you get the same random numbers as last time:

```
set.seed(375)
runif(3)
```

```
[1] 0.9613669 0.6918535 0.7302684
```

You might want to obtain part of the same series of random numbers, and we use `.Random.seed` like this:

```
current<-.Random.seed
runif(3)
```

```
[1] 0.9228566 0.1603804 0.9642799
```

```
runif(3)
```

```
[1] 0.52880907 0.08660864 0.29075809
```

```
runif(3)
```

```
[1] 0.02590182 0.85520652 0.31350305
```

Resetting `.Random.seed` recreates the same series of random numbers:

```
.Random.seed<-current
runif(3)
```

```
[1] 0.9228566 0.1603804 0.9642799
```

Randomization is central to a great many scientific and statistical procedures. Generating random numbers from a variety of probability distributions is explained in Chapter 7 (p. 272). Here we are concerned with randomizing (shuffling or sampling from) the elements of a vector, as we might use when planning a designed experiment (e.g. allocating treatments to individuals). There are two ways of sampling:

- sampling without replacement (where all of the values in the vector appear in the output, but in a randomized sequence; i.e. the values have been shuffled);
- sampling with replacement (where some values are omitted, and other values appear more than once in the output).

### 2.9.1 The sample function

The default `sample` 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. The vector `y` looks like this:

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

Here are two different shufflings of `y`:

```
sample(y)
```

```
[1]    8    8    9    9    2    10    6    7    3    11    5    4
[13]   6    3    4
```

```
sample(y)
```

```
[1]    9    3    9    8    8    6    5    11    4    6    4    7
[13]   3    2    10
```

The order of the values is different each time `sample` is invoked, but the same numbers are shuffled in every case, and all the numbers in the original vector appear once in the output (so if there are two 9s in the original data, there will be two 9s in the shuffled vector). This is called *sampling without replacement*. You can specify the size of the sample you want as an optional second argument. Suppose we want five random elements from `y`, in any one sample:

```
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. 570). 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
[13]   3    9    3
```

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

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

```
[1]    3     7    10    6    8    2    5    11    4    6    3    9
[13]   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
```

Thus, the four random numbers in the first trial were 8, 7, 4 and 9 (i.e. column 1). To learn more about `sapply`, see p. 63.

## 2.10 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 include 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.

```
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 quite 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
```

### 2.10.1 Creating the binary representation of a number

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
```

### 2.10.2 Loop avoidance

It is good R programming practice to avoid using loops wherever possible. The use of vector functions (p. 41) 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. 39) 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)
```

Next 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"
[9] "small" "small" "small" "big"   "big" "small" "big" "big"
[17]"small" "big"   "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. 838).

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]        -Inf 0.6931472      -Inf 0.0000000      -Inf 0.0000000
[7]   0.0000000      -Inf 0.6931472 1.6094379 1.3862944 0.0000000
[13]  1.3862944      -Inf 0.0000000 0.0000000 0.6931472 0.6931472
[19]  0.0000000      -Inf
```

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

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

```
[1]          NA 0.6931472        NA 0.0000000        NA 0.0000000
[7]   0.0000000        NA 0.6931472 1.6094379 1.3862944 0.0000000
[13]  1.3862944        NA 0.0000000 0.0000000 0.6931472 0.6931472
[19]  0.0000000        NA
```

### 2.10.3 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))
```

```
user system elapsed
 0.03 0.00 0.03
```

As you see, this operation took just 0.03 seconds to solve using the vector function `max` to look at the 10 million numbers in *x*. Using a loop, however, took more than 9 seconds:

```
pc <- proc.time()

cmax <- x[1]
for (i in 2:10000000) {
if(x[i]>cmax) cmax <- x[i] }

proc.time()-pc
```

```
user system elapsed
 9.39 0.13 9.51
```

The functions system.time and proc.time produce a vector of three numbers, showing the user, system and total elapsed times for the currently running R process. It is the third number (elapsed time in seconds, 9.51 in this case) that is typically the most useful.

### 2.10.4 Do not 'grow' data sets by concatenation or recursive function calls

Here is an extreme example of what *not* to do. We want to create a vector containing 100 000 numbers in sequence from 1 to 100 000. First, the quickest way using the built-in sequence generator which is invoked by the colon symbol (:)

```
test1 <- function(){
y <- 1:100000
}
```

Now we obtain the same result using a loop, where we tell R in advance how long the final vector is going to be, using the numeric function. This is called preallocation.

```
test2 <- function(){
y <- numeric(100000)
for (i in 1:100000) y[i] <- i
}
```

Finally, the most inefficient way. Each time we go round the loop we concatenate the new value onto the right-hand end of the vector that has been created up to this point. We start with a NULL vector, then build it up, one step at a time, which looks like a neat idea, but is extremely inefficient, because changing the size of a vector takes roughly the same size as setting a vector up from scratch, and we change the length of our vector 100 000 times in this example. This ill-advised procedure is called re-dimensioning.

```
test3 <- function(){
y <- NULL
for (i in 1:100000) y <- c(y,i)
}
```

To compare the efficiency of the three methods, we shall work out how long each takes to complete the task. The function called proc.time determines how much real time and computer processing unit (CPU) time (in seconds) the currently running R process has already taken:

```
proc.time()
```

```
 user system elapsed
 53.15 5.14 2483.00
```

The user time is the CPU time charged for the execution of user instructions of the calling process, the system time is the CPU time charged for execution by the system on behalf of the calling process, and the elapsed time includes other stuff that the computer is doing, unrelated to your R session.

The function system.time calls the function proc.time, then evaluates your expression, and then calls proc.time once more, returning the difference between the two proc.time calls as its output. We can compare the efficiencies of our three different functions using system.time like this:

```
system.time(test1())
```

```
user system elapsed
   0      0       0
```

```
system.time(test2())
```

```
user system elapsed
0.16   0.02    0.17
```

```
system.time(test3())
```

```
user system elapsed
8.95   0.02    8.97
```

The first method is so lightening fast that it does not even register on the clock. The loop using a pre-determined vector length is also very fast (0.16 seconds). In contrast, the last method, where we grew the vector at each iteration, is staggeringly slow (8.95 seconds). The moral: *do not grow vectors by repeated concatenation*.

### 2.10.5 Loops for producing time series

Wherever we can, we use vectorized functions in R because this leads to compact, efficient and easily readable code. Sometimes, however, we need to resort to using loops. Suppose we are interested in the dynamics of a population which is governed by two parameters: the per capita reproductive rate ($\lambda$) and the maximum supportable population ($N_{\max}$), which for convenience we shall set to 1.0). Next year's population $N(t+1)$ is given by this year's population, $N(t)$, multiplied by lambda, multiplied again by the fraction of $N_{\max}$ that is currently unrealized (i.e. $(N_{\max} - N(t))/N_{\max} = 1 - N(t)$ in the current case). Thus, we have a difference equation

$$N(t+1) = \lambda N(t)[1 - N(t)]$$

To simulate the dynamics of this population in R, we start by writing the difference equation as a function (call it `next.year` for instance):

```
next.year <- function(x) lambda * x * (1 - x)
```

So if we begin with a population of $N = 0.6$ and set $\lambda = 3.7$ we can predict next year's population like this:

```
lambda <- 3.7
next.year(0.6)
```

```
[1] 0.888
```

The population has increased by 48% (0.888 / 0.6 = 1.48). What happens in the second year?

```
next.year(0.888)
```

```
[1] 0.3679872
```

The population crashes to less than half its previous value (0.367 987 2/0.888 = 0.4144). We could go on repeating these calculations, modelling year after year, but this is an obvious case where using a loop would be the best solution. Let us assume that we want to model the population over 20 years. It is good practice in cases like this to define a vector to contain the 20 population sizes at the outset (*preallocation*) using `numeric` like this:

```
N <- numeric(20)
```

We set the initial population size (0.6) like this:

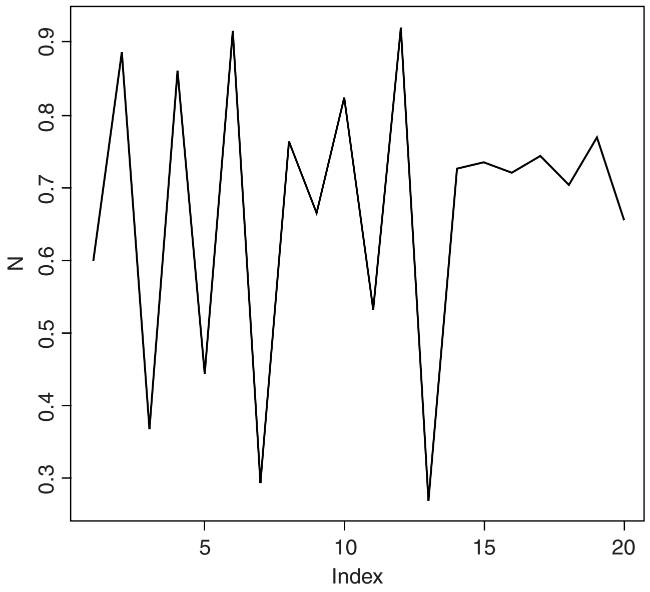
```
N[1] <- 0.6
```

Now if we run through a loop to simulate the years 2 through 20 using an index called `t` (for time), we can invoke the function called `next.year` repeatedly, employing `t` as a subscript like this:

```
for (t in 2:20) N[t] <- next.year(N[t-1])
```

Finally, we might want to plot a time series of the population dynamics over the course of 20 years.

```
plot(N,type="l")
```



This famous difference equation is known as the quadratic map, and it played a central role in the development of chaos theory (May, 1974). For large values of $\lambda$ (as we used in the example above), the function is capable of producing series of numbers that are, to all intents and purposes, random. This led to a definition of **chaos** as behaviour that exhibited *extreme sensitivity to initial conditions*: tiny differences in initial population size would lead to radically different time series in population dynamics.

## 2.11 Lists

Lists are extremely important objects in R. You will have heard of the problems of 'comparing apples and oranges' or how two things are 'as different as chalk and cheese'. You can think of lists as a way of getting around these problems. Here are four completely different objects: a numeric vector, a logical vector, a vector of character strings and a vector of complex numbers:

```
apples <- c(4,4.5,4.2,5.1,3.9)
oranges <- c(TRUE, TRUE, FALSE)
chalk <- c("limestone", "marl","oolite", "CaC03")
cheese <- c(3.2-4.5i,12.8+2.2i)
```

We cannot bundle them together into a dataframe, because the vectors are of different lengths:

```
data.frame(apples,oranges,chalk,cheese)
```

```
Error in data.frame(apples, oranges, chalk, cheese) :
arguments imply differing number of rows: 5, 3, 4, 2
```

Despite their differences, however, we can bundle them together in a single list called items:

```
items <- list(apples,oranges,chalk,cheese)
items
```

```
[[1]]
[1] 4.0 4.5 4.2 5.1 3.9

[[2]]
[1] TRUE TRUE FALSE

[[3]]
[1] "limestone" "marl" "oolite" "CaC03"

[[4]]
[1] 3.2-4.5i 12.8+2.2i
```

Subscripts on vectors, matrices, arrays and dataframes have one set of square brackets [6], [3,4] or [2,3,2,1], but subscripts on lists have double square brackets [[2]] or [[i,j]]. If we want to extract the chalk from the list, we use subscript [[3]]:

```
items[[3]]
```

```
[1] "limestone" "marl" "oolite" "CaC03"
```

If we want to extract the third element within chalk (oolite) then we use single subscripts *after* the double subscripts like this:

```
items[[3]][3]
```

```
[1] "oolite"
```

R is forgiving about failure to use double brackets on their own, but not when you try to access a component of an object within a list:

```
items[3]
```

```
[[1]]
[1] "limestone" "marl" "oolite" "CaC03"
```

```
items[3][3]
```

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

There is another indexing convention in R which is used to extract named components from lists using the element names operator `$`. This is known as 'indexing tagged lists'. For this to work, the elements of the list must have names. At the moment our list called `items` has no names:

```
names(items)
```

```
NULL
```

You can give names to the elements of a list in the function that creates the list by using the equals sign like this:

```
items <- list(first=apples,second=oranges,third=chalk,fourth=cheese)
```

Now you can extract elements of the list by name

```
items$fourth
```

```
[1] 3.2-4.5i 12.8+2.2i
```

### 2.11.1 Lists and `lapply`

We can ask a variety of questions about our new list object:

```
class(items)
```

```
[1] "list"
```

```
mode(items)
```

```
[1] "list"
```

```
is.numeric(items)
```

```
[1] FALSE
```

```
is.list(items)
```

```
[1] TRUE
```

```
length(items)
```

```
[1] 4
```

Note that the length of a list is the number of items in the list, not the lengths of the individual vectors within the list.

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 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(items,length)
```

```
$first
[1] 5

$second
[1] 3

$third
[1] 4

$fourth
[1] 2
```

This shows that `items` consists of four vectors, and shows that there were 5 elements in the first vector, 3 in the second 4 in the third and 2 in the fourth. But 5 of what, and 3 of what? To find out, we apply the function class to the list:

```
lapply(items,class)

$first
[1] "numeric"

$second
[1] "logical"

$third
[1] "character"

$fourth
[1] "complex"
```

So the answer is there were 5 numbers in the first vector, 3 logical variables in the second, 4 character strings in the third vector and 2 complex numbers in the fourth.

Applying numeric functions to lists will only work for objects of class numeric or complex, 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 items:

```
lapply(items,mean)

$first
[1] 4.34

$second
[1] 0.6666667

$third
[1] NA

$fourth
[1] 8-1.15i

Warning message:
In mean.default(X[[3L]], ...) :
 argument is not numeric or logical: returning NA
```

We get a warning message pointing out that the third vector cannot be coerced to a number (it is not numeric, complex or logical), so NA appears in the output. The second vector produces the answer 2/3 because logical false (FALSE) is coerced to numeric 0 and logical true (TRUE) is coerced to numeric 1.

The summary function works for lists:

```
summary(items)

  Length  Class      Mode
 first 5 -none-   numeric
second 3 -none-   logical
 third 4 -none- character
fourth 2 -none-   complex
```

but the most useful overview of the contents of a list is obtained with `str`, the structure function:

```
str(items)

List of 4
 $ first : num [1:5] 4 4.5 4.2 5.1 3.9
 $ second: logi [1:3] TRUE TRUE FALSE
 $ third : chr [1:4] "limestone" "marl" "oolite" "CaCO3"
 $ fourth: cplx [1:2] 3.2-4.5i 12.8+2.2i
```

### 2.11.2 Manipulating and saving lists

Saving lists to files is tricky, because lists typically have different numbers of items in each row so we cannot use `write.table`. Here is a dataframe on species presence (1) or absence (0), with species' Latin binomials in the first column as the row names:

```
data<-read.csv("c:\\temp\\pa.csv",row.names=1)
data

                       Carmel Derry Daneswall Erith Foggen Highbury Slatewell Uppington York
Bartsia alpina              0     0         1     0      0        0         1         0    0
Cleome serrulata            1     1         0     0      0        1         0         0    0
Conopodium majus            0     0         0     0      0        0         0         1    1
Corydalis sempervirens      1     0         0     1      0        1         0         0    0
Nitella flexilis            1     0         0     0      0        0         0         0    1
Ranunculus baudotii         1     0         1     1      0        0         0         1    0
Rhododendron luteum         1     1         1     1      1        0         1         1    1
Rodgersia podophylla        0     1         0     0      0        1         0         0    0
Tiarella wherryi            0     0         1     1      1        0         0         0    0
Veronica opaca              1     0         0     0      0        1         1         1    0
```

There are two kinds of operations you might want to do with a dataframe like this:

- produce lists of the sites at which each species is found;
- produce lists of the species found in any given site.

We shall do each of these tasks in turn.

The issue is that the numbers of place names differ from species to species, and the numbers of species differ from place to place. However, it is easy to create a list showing the column numbers that contain locations for each species:

```
sapply(1:10,function(i) which(data[i,]>0))

[[1]]
[1] 3 7

[[2]]
[1] 1 2 6

[[3]]
[1] 8 9
```

```
[[4]]
[1] 1 4 6

[[5]]
[1] 1 9

[[6]]
[1] 1 3 4 8

[[7]]
[1] 1 2 3 4 5 7 8 9

[[8]]
[1] 2 6

[[9]]
[1] 3 4 5

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

This indicates that *Bartsia alpina* (the first species) is found in locations 3 and 7 (Daneswall and Slatewell). If we save this list (calling it spp for instance), then we can extract the column names at which each species is present, using the elements of spp as subscripts on the column names of data, like this:

```
spp<-sapply(1:10,function(i) which(data[i,]>0))
sapply(1:10, function(i)names(data)[spp[[i]]] )
```

```
[[1]]
[1] "Daneswall" "Slatewell"

[[2]]
[1] "Carmel"    "Derry"     "Highbury"

[[3]]
[1] "Uppington" "York"

[[4]]
[1] "Carmel"    "Erith"     "Highbury"

[[5]]
[1] "Carmel"    "York"

[[6]]
[1] "Carmel"    "Daneswall" "Erith"     "Uppington"

[[7]]
[1] "Carmel"    "Derry"     "Daneswall" "Erith"   "Foggen" "Slatewell" "Uppington" "York"

[[8]]
[1] "Derry"     "Highbury"

[[9]]
[1] "Daneswall" "Erith"     "Foggen"
```

```
[[10]]
[1] "Carmel"    "Highbury"  "Slatewell"   "Uppington"
```

This completes the first task.

The second task is to get species lists for each location. We apply a similar method to extract the appropriate species (this time from `rownames(data)`) on the basis that the presence score for this site is `data[,j] > 0`:

```
sapply(1:9, function (j) rownames(data)[data[,j]>0] )
[[1]]
[1] "Cleome serrulata" "Corydalis sempervirens" "Nitella flexilis" "Ranunculus baudotii"
[5] "Rhododendron luteum" "Veronica opaca"
[[2]]
[1] "Cleome serrulata" "Rhododendron luteum" "Rodgersia podophylla"
[[3]]
[1] "Bartsia alpina" "Ranunculus baudotii" "Rhododendron luteum" "Tiarella wherryi"
[[4]]
[1] "Corydalis sempervirens" "Ranunculus baudotii" "Rhododendron luteum" "Tiarella wherryi"
[[5]]
[1] "Rhododendron luteum" "Tiarella wherryi"
[[6]]
[1] "Cleome serrulata" "Corydalis sempervirens" "Rodgersia podophylla" "Veronica opaca"
[[7]]
[1] "Bartsia alpina" "Rhododendron luteum" "Veronica opaca"
[[8]]
[1] "Conopodium majus" "Ranunculus baudotii" "Rhododendron luteum" "Veronica opaca"
[[9]]
[1] "Conopodium majus" "Nitella flexilis" "Rhododendron luteum"
```

Because the species lists for different sites are of different lengths, the simplest solution is to create a separate file for each species list. We need to create a set of nine file names incorporating the site name, then use `write.table` in a loop:

```
spplists<-sapply(1:9, function (j) rownames(data)[data[,j]>0] )
for (i in 1:9) {
slist<-data.frame(spplists[[i]])
names(slist)<-names(data)[i]
fn<-paste("c:\\temp\\",names(data)[i],".txt",sep="")
write.table(slist,fn)
}
```

We have produced nine separate files. Here, for instance, are the contents of the file called `c:\\temp\\Carmel.txt` as viewed in a text editor like Notepad:

```
"Carmel"
"1" "Cleome serrulata"
"2" "Corydalis sempervirens"
"3" "Nitella flexilis"
```

```
"4" "Ranunculus baudotii"
"5" "Rhododendron luteum"
"6" "Veronica opaca"
```

Perhaps the simplest and best solution is to turn the whole presence/absence matrix into a dataframe. Then both tasks become very straightforward. Start by using `stack` to create a dataframe of place names and presence/absence information:

```
newframe<-stack(data)
head(newframe)
```

```
  values    ind
1      0 Carmel
2      1 Carmel
3      0 Carmel
4      1 Carmel
5      1 Carmel
6      1 Carmel
```

Now extract the species names from the row names, repeat the list of names nine times, and add the resulting vector species names to the dataframe:

```
newframe<-data.frame(newframe, rep(rownames(data),9))
```

Finally, give the three columns of the new dataframe sensible names:

```
names(newframe)<-c("present","location","species")
head(newframe)
```

```
  present location                species
1       0   Carmel         Bartsia alpina
2       1   Carmel       Cleome serrulata
3       0   Carmel       Conopodium majus
4       1   Carmel Corydalis sempervirens
5       1   Carmel       Nitella flexilis
6       1   Carmel    Ranunculus baudotii
```

Unlike the lists, you can easily save this object to file:

```
write.table(newframe,"c:\\temp\\spplists.txt")
```

Now it is simple to do both our tasks. Here a location list for `species = Bartsia alpina`:

```
newframe[newframe$species=="Bartsia alpina" & newframe$present==1,2]
```

```
[1] Daneswall Slatewell
```

and here is a species list for `location = Carmel`:

```
newframe[newframe$location=="Carmel" & newframe$present==1,3]
```

```
[1] Cleome serrulata    Corydalis sempervirens  Nitella flexilis Ranunculus baudotii
[5] Rhododendron luteum Veronica  opaca
```

Lists are great, but dataframes are better. The cost of the dataframe is the potentially substantial redundancy in storage requirement. In practice, with relatively small dataframes, this seldom matters.

## 2.12 Text, character strings and pattern matching

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

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

Numbers can be coerced to characters (as in `b` above), but non-numeric characters cannot be coerced to 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 (`nchar`) in the strings that comprise 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 (e.g. if it was input using `read.table`) 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
```

### 2.12.1 Pasting character strings together

You can amalgamate individual 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-character 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]  "abc123 a 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"
```

You may need to think about why there are three lines of output.

In this next example, we have four fields of information and we want to paste them together to make a file path for reading data into R:

```
drive <- "c:"
folder <- "temp"
file <- "file"
extension <- ".txt"
```

Now use the function `paste` to put them together:

```
paste(drive, folder, file, extension)
```

```
[1] "c: temp file .txt"
```

This has the essence of what we want, but it is not quite there yet. We need to replace the blank spaces `" "` that are the default separator with `""` no space, and to insert slashes `"\\"` between the drive and the directory, and the directory and file names:

```
paste(drive, "\\",folder, "\\",file, extension,sep="")
```

```
[1] "c:\\temp\\file.txt"
```

### 2.12.2 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 within a character string. Here is the code to extract the first, the first and second, the first, second and third, . . . , the first 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).

### 2.12.3 Counting things within strings

Counting the total number of characters in a string could not be simpler; just use the `nchar` function directly, like this:

```
nchar(phrase)
```

```
[1] 43
```

So there are 43 characters including the blanks between the words. To count the numbers of separate individual characters (including blanks) you need to split up the character string into individual characters (43 of them), using `strsplit` like this:

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

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

You could use `NULL` in place of `split=character(0)` (see below). The `table` function can then be used for counting the number of occurrences of each of the characters:

```
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 the string called `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 (so long as there are no leading or trailing blanks in the string):

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

```
9
```

What about the lengths of the words within phrase? Here are the separate words:

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

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

We work out their lengths using lapply to apply the function nchar to each element of the list produced by strsplit. Then we use table to count how many words of each length are present:

```
table(lapply(strsplit(phrase, " "), nchar))
```

```
3 4 5
4 2 3
```

showing there were 4 three-letter words, 2 four-letter words and 3 five-letter words.

This is how you reverse a character string. The logic is that you need to break it up into individual characters, then reverse their order, then paste them all back together again. It seems long-winded until you think about what the alternative would be:

```
strsplit(phrase,NULL)
```

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

```
lapply(strsplit(phrase,NULL),rev)
```

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

```
sapply(lapply(strsplit(phrase, NULL), rev), paste, collapse="")
```

```
[1] "god yzal eht revo spmuj xof nworb kciuq eht"
```

The collapse argument is necessary to reduce the answer back to a single character string. Note that the word lengths are retained, so this would be a poor method of encryption.

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
```

### 2.12.4 Upper- and lower-case text

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"
```

### 2.12.5 The `match` function and relational databases

The `match` function answers the question: 'Where (if at all) do the values in the second vector appear in the first vector?' It is a really important function, but it 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`. It works like this. Where does 5 (from the first position in the first vector) appear in the second vector? Answer: it does not (`NA`). Then, where does 8 (the second element of the first vector) appear in the second vector? Answer: in position number 1. And so on. Why would you ever want to use this? The answer turns out to be very general and extremely useful in data management.

Large and/or complicated databases are always best stored as relational databases (e.g. Oracle or Access). In these, data are stored in sets of two-dimensional spreadsheet-like objects called tables. Data are divided into small tables with strict rules as to what data they can contain. You then create relationships between the tables that allow the computer to look from one table to another in order to assemble the data you want for a particular application. The relationship between two tables is based on fields whose values (if not their variable names) are common to both tables. The rules for constructing effective relational databases were first proposed by Dr E.F. Codd of the IBM Research Laboratory at San Jose, California, in an extremely influential paper in 1970:

- All data are in tables.
- There is a separate table for each set of related variables.

- The order of the records within tables is irrelevant (so you can add records without reordering the existing records).
- The first column of each table is a unique ID number for every row in that table (a simple way to make sure that this works is to have the rows numbered sequentially from 1 at the top, so that when you add new rows you are sure that they get unique identifiers).
- There is no unnecessary repetition of data so the storage requirement is minimized, and when we need to edit a record, we only need to edit it once (the last point is very important).
- Each piece of data is 'granular' (meaning as small as possible); so you would split a customer's name into title (Dr), first name (Charles), middle name (Urban), surname (Forrester), and preferred form of address (Chuck), so that if they were promoted, for instance, we would only need to convert Dr to Prof. in the title field.

These are called the 'normalization rules' for creating bullet-proof databases. The use of Structured Query Language (SQL) in R to interrogate relational databases is discussed in Chapter 3 (p. 154). Here, the only point is to see how the `match` function relates information in one vector (or table) to information in another.

Take a medical example. You have a vector containing the anonymous identifiers of nine patients (subjects):

```
subjects <- c("A", "B", "G", "M", "N", "S", "T", "V", "Z")
```

Suppose you wanted to give a new drug to all the patients identified in the second vector called `suitable.patients`, and the conventional drug to all the others. Here are the suitable patients:

```
suitable.patients <- c("E", "G", "S", "U", "Z")
```

Notice that there are several suitable patients who are not part of this trial (E and U). This is what the `match` function does:

```
match(subjects, suitable.patients)
```

```
[1] NA NA 2 NA NA 3 NA NA 5
```

For each of the individuals in the first vector (subjects) it finds the subscript in the second vector (suitable patients), returning `NA` if that patient does not appear in the second vector. The key point to understand is that the vector produced by `match` is *the same length as the first vector* supplied to `match`, and that the numbers in the result are *subscripts within the second vector*. The last bit is what people find hard to understand at first.

Let us go through the output term by term and see what each means. Patient A is not in the suitable vector, so `NA` is returned. The same is true for patient B. Patient G is suitable, so we get a number in the third position. That number is a 2 because patient G is the second element of the vector called `suitable.patients.` Neither patient M nor N is in the second vector, so they both appear as `NA`. Patient S is suitable and so produces a number. The number is 3 because that is the position of S with the second vector.

To complete the job, we want to produce a vector of the drugs to be administered to each of the subjects. We create a vector containing the two treatment names:

```
drug <- c("new", "conventional")
```

Then we use the result of the match to give the right drug to the right patient:

```
drug[ifelse(is.na(match(subjects, suitable.patients)),2,1)]
```

```
[1] "conventional" "conventional" "new" "conventional"
```

```
[5] "conventional" "new" "conventional" "conventional"
[9] "new"
```

Note the use of `ifelse` with `is.na` to produce a subscript 2 (to use with drug) for the unsuitable patients, and a 1 when the result of the match is not `NA` (i.e. for the suitable patients). You may need to work through this example several times (but it is well worth mastering it).

### 2.12.6 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 ˆ 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 `$` 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"
```

### 2.12.7 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"
```

(Five 'anythings' is shown by '.' then curly brackets {5} then *y*.) Which are the countries with four 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:

```
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"
```

### 2.12.8 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"
```

Here we convert all the characters to upper case:

```
gsub("(\\w*)", "\\U\\1",text, perl=TRUE)
```

```
[1] "ARM" "LEG" "HEAD" "FOOT" "HAND" "HINDLEG" "ELBOW"
```

### 2.12.9 Locations of a pattern within a vector 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
```

### 2.12.10 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. 91):

```
stock[match(requests,stock)][!is.na(match(requests,stock))]
```

```
[1]  "van"  "car"  "car"
```

but this is 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 also that the match must be perfect for this to work ('car' with 'car' is not the same as 'car' with 'cars').

### 2.12.11 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 (Portable Operating System Interface) 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"
```

### 2.12.12 `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`.

### 2.12.13 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)`).

## 2.13 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] "2014-01-24 16:24:54 GMT"
```

This description of date and time is strictly hierarchical from left to right: the longest time scale (years) comes first, then month, then day, separated by hyphens, then there is a blank space, followed by the time, with hours first (using the 24-hour clock), then minutes, then seconds, separated by colons. Finally, there is a character string explaining the time zone (GMT stands for Greenwich Mean Time). This representation of the date and time as a character string is user-friendly and familiar, but it is no good for calculations. For that, we need a single numeric representation of the combined date and time. The convention in R is to base this on seconds (the smallest time scale that is accommodated in `Sys.time`). You can always aggregate upwards to days or year, but you cannot do the reverse. The baseline for expressing today's date and time in seconds is 1 January 1970:

```
as.numeric(Sys.time())
```

```
[1] 1390580694
```

This is fine for plotting time series graphs, but it is not much good for computing monthly means (e.g. is the mean for June significantly different from the July mean?) or daily means (e.g. is the Monday mean significantly different from the Friday mean?). To answer questions like these we have to be able to access a broad set of categorical variables associated with the date: the year, the month, the day of the week, and so forth. To accommodate this, R uses the POSIX system for representing times and dates:

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

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

You can think of the class POSIXct, with suffix 'ct', as *continuous time* (i.e. a number of seconds), and POSIXlt, with suffix 'lt', as *list time* (i.e. a list of all 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. Naturally, you can easily convert to one representation to the other:

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

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

```
sec min hour mday mon year wday yday isdst
 54  24   16   24   0  114    5   23     0
```

Here you see the nine components of the list. The time is represented by the number of seconds (sec), minutes (min) and hours (on the 24-hour clock). Next comes the day of the month (mday, starting from 1), then the month of the year (mon, starting at January = 0), 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. Finally, there is a logical variable isdst which asks whether daylight saving time is in operation (0 = FALSE in this case). 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).

### 2.13.1 Reading time data from files

It is most likely that your data files contain dates in Excel format, for example 03/09/2014 (a character string showing day/month/year separated by slashes).

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

When you read character data into R using read.table, the default option is to convert the character variables into factors. Factors are of mode numeric and class factor:

```
mode(date)
```

```
[1] "numeric"
```

```
class(date)
```

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

For our present purposes, the point is that the data are not recognized by R as being dates. To convert a factor or a character string into a POSIXlt object, we employ an important function called 'strip time', written strptime.

### 2.13.2 The strptime function

To convert a factor or a character string into dates using the strptime function, we provide a format statement enclosed in double quotes to tell R exactly what to expect, in what order, and separated by what kind of symbol. For our present example we have day (as two digits), then slash, then month (as two digits), then slash, then year (with the century, making four digits).

```
Rdate <- strptime(as.character(date),"%d/%m/%Y")
```

```
class(Rdate)
```

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

It is always a good idea at this stage to add the R-formatted date to your dataframe:

```
data <- data.frame(data,Rdate)
```

```
head(data)
```

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

Now, at last, we can do things with the date information. We might want the mean value of x for each day of the week. The name of this object is Rdate$wday:

```
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 lowest mean is on Mondays (wday = 1) and the highest on Fridays (wday = 5).

It is hard to remember all the format codes for strip time, but they are roughly mnemonic and they are always preceded by a percent symbol. Here is the full list of format components:

| | |
|---|---|
| %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 (0–366) |
| %m | Month as decimal number (0–11) |
| %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 |
| `%y` | Year without century |
| `%Z` | Time zone as a character string (output only) |

Note the difference between the upper case for year %Y (this is the unambiguous year including the century, 2014), and the potentially ambiguous lower case %y (it is not clear whether 14 means 1914 or 2014).

There is a useful function called `weekdays` (note the plural) for turning the day number into the appropriate name:

```
y <- strptime("01/02/2014",format="%d/%m/%Y")
weekdays(y)
```

```
[1] "Saturday"
```

which is converted from

```
y$wday
```

```
[1] 6
```

because the days of the week are numbered from Sunday = 0.

Here is another kind of date, with years in two-digit form (`%y`), and the months as abbreviated names (`%b`) using 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"
```

Here is yet another possibility with year, then month in full, then week of the year, then day of the week abbreviated, all separated by a single blank space:

```
yet.another.date <- c("2016 January 2 Mon","2017 February 6 Fri","2018
  March 10 Tue")
strptime(yet.another.date,"%Y %B %W %a")
```

```
[1] "2016-01-11" "2017-02-10" "2018-03-06"
```

The system is clever in that it knows the date of the Monday in week number 2 of January in 2016, and of the Tuesday in week 10 of 2018 (the information on month is redundant in this case):

```
yet.more.dates <- c("2016 2 Mon","2017 6 Fri","2018 10 Tue")
strptime(yet.more.dates,"%Y %W %a")
```

```
[1] "2016-01-11" "2017-02-10" "2018-03-06"
```

### 2.13.3 The `difftime` function

The function `difftime` calculates a difference of two date-time objects and returns an object of class `difftime` with an attribute indicating the units. You can use various arithmetic operations on a `difftime` object including `round, signif, floor, ceiling, trunc, abs, sign` and certain logical operations. You can create a `difftime` object like this:

```
as.difftime(yet.more.dates,"%Y %W %a")
```

```
Time differences in days
[1] 1434 1830 2219
attr(,"tzone")
[1] ""
```

or like this:

```
difftime("2014-02-06","2014-07-06")
```

```
Time difference of -149.9583 days
```

```
round(difftime("2014-02-06","2014-07-06"),0)
```

```
Time difference of -150 days
```

### 2.13.4 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 from a date-time object, but you cannot add two date-time objects. Subtraction of two date-time objects is equivalent to using `difftime`. 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 2015 and 22 October 2018:

```
y2 <- as.POSIXlt("2015-10-22")
y1 <- as.POSIXlt("2018-10-22")
```

Now you can do calculations with the two dates:

```
y1-y2
```

```
Time difference of 1096 days
```

### 2.13.5 The `difftime` and `as.difftime` functions

Working out the time difference between two 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. For instance, how many days elapsed between 15 August 2013 and 21 October 2015?

```
difftime("2015-10-21","2013-8-15")
```

```
Time difference of 797 days
```

If you want only the number of days to use in calculation, then write

```
as.numeric(difftime("2015-10-21","2013-8-15"))
```

```
[1] 797
```

If you have times but no dates, then you can use `as.difftime` to create appropriate objects for calculations:

```
t1 <- as.difftime("6:14:21")
```

```
t2 <- as.difftime("5:12:32")
```

```
t1-t2
```

```
Time difference of 1.030278 hours
```

You will often want to create POSIXlt objects from components stored in different vectors within a dataframe. For instance, here is a dataframe with the hours, minutes and seconds from an experiment with two factor levels in four separate columns:

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

Because the times are not in POSIXlt format, you need to paste together the hours, minutes and seconds into a character string, using 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
```

which gives the answer in decimal hours.

### 2.13.6 Generating sequences of dates

You may want to generate sequences of dates by years, months, weeks, days of the month or days of the week. Here are four sequences of dates, all starting on 4 November 2015, the first going in increments of one day:

```
seq(as.POSIXlt("2015-11-04"), as.POSIXlt("2015-11-15"), "1 day")
```

```
 [1] "2015-11-04 GMT" "2015-11-05 GMT" "2015-11-06 GMT" "2015-11-07 GMT"
 [5] "2015-11-08 GMT" "2015-11-09 GMT" "2015-11-10 GMT" "2015-11-11 GMT"
 [9] "2015-11-12 GMT" "2015-11-13 GMT" "2015-11-14 GMT" "2015-11-15 GMT"
```

the second with increments of 2 weeks:

```
seq(as.POSIXlt("2015-11-04"), as.POSIXlt("2016-04-05"), "2 weeks")
```

```
 [1] "2015-11-04 GMT" "2015-11-18 GMT" "2015-12-02 GMT" "2015-12-16 GMT"
 [5] "2015-12-30 GMT" "2016-01-13 GMT" "2016-01-27 GMT" "2016-02-10 GMT"
 [9] "2016-02-24 GMT" "2016-03-09 GMT" "2016-03-23 GMT"
```

the third with increments of 3 months:

```
seq(as.POSIXlt("2015-11-04"), as.POSIXlt("2018-10-04"), "3 months")
```

```
 [1] "2015-11-04 GMT" "2016-02-04 GMT" "2016-05-04 BST" "2016-08-04 BST"
 [5] "2016-11-04 GMT" "2017-02-04 GMT" "2017-05-04 BST" "2017-08-04 BST"
 [9] "2017-11-04 GMT" "2018-02-04 GMT" "2018-05-04 BST" "2018-08-04 BST"
```

the fourth with increments of years:

```
seq(as.POSIXlt("2015-11-04"), as.POSIXlt("2026-02-04"), "year")
```

```
 [1] "2015-11-04 GMT" "2016-11-04 GMT" "2017-11-04 GMT" "2018-11-04 GMT"
 [5] "2019-11-04 GMT" "2020-11-04 GMT" "2021-11-04 GMT" "2022-11-04 GMT"
 [9] "2023-11-04 GMT" "2024-11-04 GMT" "2025-11-04 GMT" "2026-11-04 GMT"
```

If you specify a number, rather than a recognized character string, in the `by` part of the sequence function, then the number is assumed to be a number of seconds, so this generates the time as well as the date:

```
seq(as.POSIXlt("2015-11-04"), as.POSIXlt("2015-11-05"), 8955)
```

```
 [1] "2015-11-04 00:00:00 GMT" "2015-11-04 02:29:15 GMT"
 [3] "2015-11-04 04:58:30 GMT" "2015-11-04 07:27:45 GMT"
 [5] "2015-11-04 09:57:00 GMT" "2015-11-04 12:26:15 GMT"
 [7] "2015-11-04 14:55:30 GMT" "2015-11-04 17:24:45 GMT"
 [9] "2015-11-04 19:54:00 GMT" "2015-11-04 22:23:15 GMT"
```

As with other forms of `seq,` you can specify the length of the vector to be generated, instead of specifying the final date:

```
seq(as.POSIXlt("2015-11-04"), by="month", length=10)
```

```
[1] "2015-11-04 GMT" "2015-12-04 GMT" "2016-01-04 GMT" "2016-02-04 GMT"
[5] "2016-03-04 GMT" "2016-04-04 BST" "2016-05-04 BST" "2016-06-04 BST"
[9] "2016-07-04 BST" "2016-08-04 BST"
```

or you can generate a vector of dates to match the length of an existing vector, using `along=` instead of `length=`:

```
results <- runif(16)
seq(as.POSIXlt("2015-11-04"), by="month", along=results )
```

```
[1]  "2015-11-04 GMT" "2015-12-04 GMT" "2016-01-04 GMT" "2016-02-04 GMT"
[5]  "2016-03-04 GMT" "2016-04-04 BST" "2016-05-04 BST" "2016-06-04 BST"
[9]  "2016-07-04 BST" "2016-08-04 BST" "2016-09-04 BST" "2016-10-04 BST"
[13] "2016-11-04 GMT" "2016-12-04 GMT" "2017-01-04 GMT" "2017-02-04 GMT"
```

You can use the `weekdays` function to extract the days of the week from a series of dates:

```
weekdays(seq(as.POSIXlt("2015-11-04"), by="month", along=results ))
```

```
[1]   "Wednesday" "Friday"    "Monday"    "Thursday" "Friday" "Monday"
[7]   "Wednesday" "Saturday"  "Monday"    "Thursday" "Sunday" "Tuesday"
[13]  "Friday"    "Sunday"    "Wednesday" "Saturday"
```

Suppose that you want to find the dates of all the Mondays in a sequence of dates. This involves the use of logical subscripts (see p. 39). The subscripts evaluating to TRUE will be selected, so the logical statement you need to make is `wday == 1`. (because Sunday is `wday == 0`). You create an object called `y` containing the first 100 days in 2016 (note that the start date is 31 December 2015), then convert this vector of dates into a POSIXlt object, a list called `x`, like this:

```
y <- as.Date(1:100,origin="2015-12-31")
x <- as.POSIXlt(y)
```

Now, because `x` is a list, you can use the `$` operator to access information on weekday, and you find, of course, that they are all 7 days apart, starting from the 4 January 2016:

```
x[x$wday==1]
```

```
[1]  "2016-01-04 UTC" "2016-01-11 UTC" "2016-01-18 UTC" "2016-01-25 UTC"
[5]  "2016-02-01 UTC" "2016-02-08 UTC" "2016-02-15 UTC" "2016-02-22 UTC"
[9]  "2016-02-29 UTC" "2016-03-07 UTC" "2016-03-14 UTC" "2016-03-21 UTC"
[13] "2016-03-28 UTC" "2016-04-04 UTC"
```

Suppose you want to list the dates of the first Monday in each month. This is the date with `wday == 1` (as above) but only on its first occurrence in each month of the year. This is slightly more tricky, because several months will contain five Mondays, so you cannot use `seq` with `by = "28 days"` to solve the problem (this would generate 13 dates, not the 12 required). Here are the dates of all the Mondays in the year of 2016:

```
y <- as.POSIXlt(as.Date(1:365,origin="2015-12-31"))
```

Here is what we know so far:

```
data.frame(monday=y[y$wday==1],month=y$mo[y$wday==1])[1:12,]
```

```
       monday month
1  2016-01-04     0
2  2016-01-11     0
3  2016-01-18     0
4  2016-01-25     0
```

```
5  2016-02-01     1
6  2016-02-08     1
7  2016-02-15     1
8  2016-02-22     1
9  2016-02-29     1
10 2016-03-07     2
11 2016-03-14     2
12 2016-03-21     2
```

You want a vector to mark the 12 Mondays you require: these are those where `month` is not duplicated (i.e. you want to take the first row from each month). For this example, the first Monday in January is in row 1 (obviously), the first in February in row 5, the first in March in row 10, and so on. You can use the not duplicated function `!duplicated` to tag these rows

```
wanted <- !duplicated(y$mo[y$wday==1])
```

Finally, select the 12 dates of the first Mondays using `wanted` as a subscript like this:

```
y[y$wday==1][wanted]
```

```
[1] "2016-01-04 UTC" "2016-02-01 UTC" "2016-03-07 UTC" "2016-04-04 UTC"
[5] "2016-05-02 UTC" "2016-06-06 UTC" "2016-07-04 UTC" "2016-08-01 UTC"
[9] "2016-09-05 UTC" "2016-10-03 UTC" "2016-11-07 UTC" "2016-12-05 UTC"
```

Note that every month is represented, and none of the dates is later than the 7th of the month as required.

### 2.13.7 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 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

attr(,"tzone")
[1] ""
```

You can compute the differences between successive rows using subscripts, like this:

```
duration[1:15]-duration[2:16]
```

```
Time differences in hours

 [1] -0.8863889 0.2225000 0.2988889 -0.3283333 0.1380556
 [6] -0.2675000 1.2544444 -0.4194444 0.6708333 -0.8194444
[11] -0.7330556 0.7866667 0.5416667 -0.3591667 0.4719444
```

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 you want to put the first of the differences. You should be guided by whether the change in time between rows 1 and 2 is 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 (–0.8864). Because we are working with differences (see p. 785) 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 you 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 you 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
```

You need to take care when doing things with differences. For instance, is it really appropriate that the difference in row 8 is between the last measurement on treatment A and the first measurement on treatment B? Perhaps what you really want are the time differences within the treatments, so you need to insert another `NA` in row number 8? If so, then:

```
times$diffs[8] <- NA
```

### 2.13.8 Regression using dates and times

Here is an example where the number of individual insects was monitored each month over the course of 13 months:

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

```
  survivors       date
1       100 01/01/2011
2        52 01/02/2011
3        28 01/03/2011
4        12 01/04/2011
5         6 01/05/2011
6         5 01/06/2011
```

The first job, as usual, is to use `strptime` to convert the character string `"01/01/2011"` into a date-time object:

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

You can see that the object called `dl` is of class `POSIXlt` and mode `list`:

```
class(dl)
```

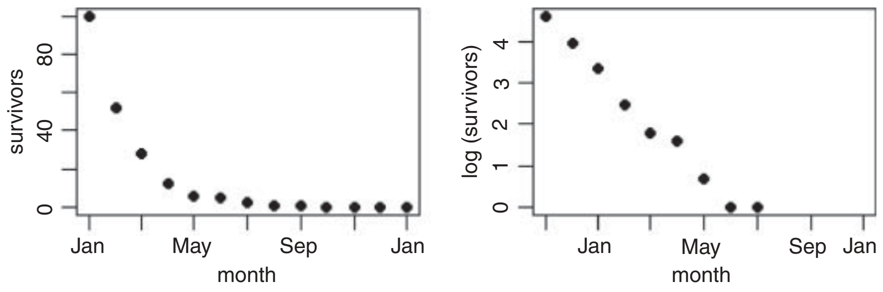
```
[1] "POSIXlt" "POSIXt"
```

```
mode(dl)
```

```
[1] "list"
```

We start by looking at the data using `plot` with the date `dl` on the *x* axis:

```
windows(7,4)
par(mfrow=c(1,2))
plot(dl,survivors,pch=16,xlab ="month")
plot(dl,log(survivors),pch=16,xlab ="month")
```



Inspection of the relationship suggests an exponential decay in numbers surviving, so we shall analyse a model in which `log(survivors)` is modelled as a function of time. There are lots of zeros at the end of the time series (once the last of the individuals was dead), so we shall use `subset` to leave out all of the zeros from the model. Let us try to do the regression analysis of `log(survivors)` against date:

```
model <- lm(log(survivors)~dl,subset=(survivors>0))
```

```
Error in model.frame.default(formula = log(survivors) ~ dl,
subset = (survivors > :
 invalid type (list) for variable 'dl'
```

Oops. Why did that not work? The answer is that you cannot have a list as an explanatory variable in a linear model, and as we have just seen, `dl` is a list. We need to convert from a list (`class = POSIXlt`) to a continuous numeric variable (`class = POSIXct`):

```
dc <- as.POSIXct(dl)
```

Now the regression works perfectly when we use the new continuous explanatory variable `dc`:

```
model <- lm(log(survivors)~dc,subset=(survivors>0))
```

You would get the same effect by using `as.numeric(dl)` in the model formula. We can use the output from this model to add a regression line to the plot of `log(survivors)` against time using:

```
abline(model)
```

You need to take care in reporting the values of slopes in regressions involving date-time objects, because the slopes are rates of change of the response variable *per second*. Here is the summary:

```
summary(model)
```

```
Call:
lm(formula = log(survivors) ~ dc, subset = (survivors > 0))

Residuals:
     Min       1Q   Median       3Q      Max
-0.27606 -0.18306 0.04492 0.13760 0.39277

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.040e+02  1.531e+01   19.86 2.05e-07 ***
dc          -2.315e-07  1.174e-08  -19.72 2.15e-07 ***

Residual standard error: 0.2383 on 7 degrees of freedom
Multiple R-squared:      0.9823, Adjusted R-squared: 0.9798
F-statistic:             389 on 1 and 7 DF, p-value: 2.152e-07
```

The slope is $-2.315 \times 10^{-7}$; the change in `log(survivors)` *per second*. It might be more useful to express this as the monthly rate. So, with 60 seconds per minute, 60 minutes per hour, 24 hours per day, and (say) 30 days per month, the appropriate rate is

```
 -2.315E-07 * 60 * 60 * 24 * 30
```

```
[1] -0.600048
```

We can check this out by calculating how many survivors we would expect from 100 starters after two months:

```
100*exp(-0.600048 * 2)
```

```
[1] 30.11653
```

which compares well with our observed count of 28 (see above).

### 2.13.9 Summary of dates and times in R

The key thing to understand is the difference between the two representations of dates and times in R. They have unfortunately non-memorable names.

- POSIXlt gives a **list** containing separate vectors for the year, month, day of the week, day within the year, and suchlike. It is very useful as a categorical explanatory variable (e.g. to get monthly means from data gathered over many years using `date$mon`).
- POSIXct gives a **vector** containing the date and time expressed as a continuous variable that you can use in regression models (it is the number of seconds since the beginning of 1970).

You can use other functions like `date`, but I do not recommend them. If you stick with POSIX you are less likely to get confused.

## 2.14 Environments

R is built around a highly sophisticated system of naming and locating objects. When you start a session in R, the variables you create are in the global environment `.GlobalEnv`, which is known more familiarly as the user's workspace. This is the first place in which R looks for things. Technically, it is the first item on the search path. It can also be accessed by `globalenv().`

An environment consists of a **frame**, which is collection of named objects, and a **pointer** to an enclosing environment. The most common example is the frame of variables that is local to a function call; its **enclosure** is the environment where the function was defined. The enclosing environment is distinguished from the parent frame, which is the environment of the **caller** of a function.

There is a strict hierarchy in which R looks for things: it starts by looking in the frame, then in the enclosing frame, and so on.

### 2.14.1 Using `with` rather than `attach`

When you `attach` a dataframe you can refer to the variables within that dataframe by name.

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(dataframe, function( ...))
```

The `with` function evaluates an R expression in an environment constructed from data. You will often use the `with` function with 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
```

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. 450. 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()
```

To see all available data sets (including those in the installed packages), type:

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

### 2.14.2 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.

## 2.15 Writing R functions

You typically write functions in R to carry out operations that require two or more lines of code to execute, and that you do not want to type lots of times. We might want to write simple functions to calculate measures of central tendency (p. 116), work out factorials (p. 71) and such-like.

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 over one or more lines. Generally, the body is a group of expressions contained in curly brackets { }, with each expression on a separate line (if the body fits on a single line, no curly brackets are necessary). Functions are typically assigned to symbols, but they need not be. This will only begin to mean anything after you have seen several examples in operation.

### 2.15.1 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
```

### 2.15.2 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 here 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. 18). 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` (`y[1] = 3`) has been dropped (using the negative subscript):

```
med(y)
```

```
[1] 4
```

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

```
[1] 4.5
```

You could write the same function in a single (long) line by using `ifelse` instead of `if`. You need to remember that the second argument in `ifelse` is the action to be performed when the condition is true, and the third argument is what to do when the condition is false:

```
med <- function(x) ifelse(length(x)%%2==1, sort(x)[ceiling(length(x)/2)],
      (sort(x)[length(x)/2]+sort(x)[1+ length(x)/2])/2 )
```

Again, you will not be surprised that there is a built-in function for calculating medians, and helpfully it is called `median`.

## 2.15.3 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 ($\Pi$) to represent multiplication, and $\hat{y}$ (pronounced *y*-hat) to represent the geometric mean, then

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

Let us 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 will 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 in this case, 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 here is a function to calculate geometric mean of a vector of numbers *x*:

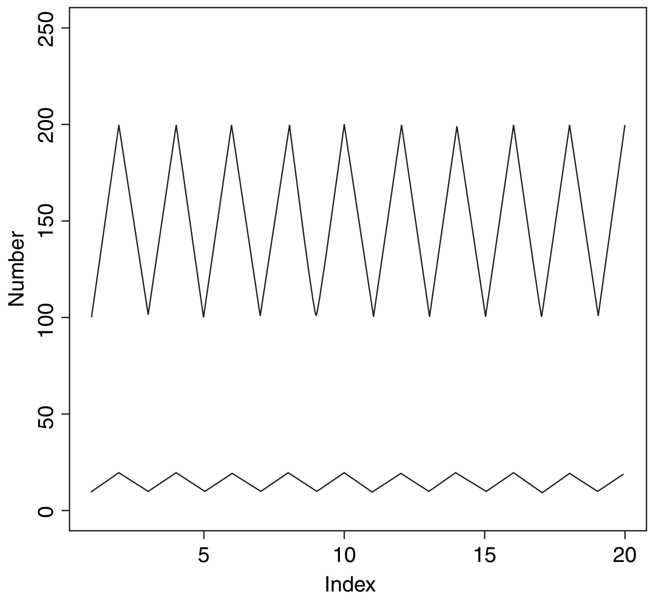
```
geometric <- function (x) exp(mean(log(x)))
```

We can test it with the insect data:

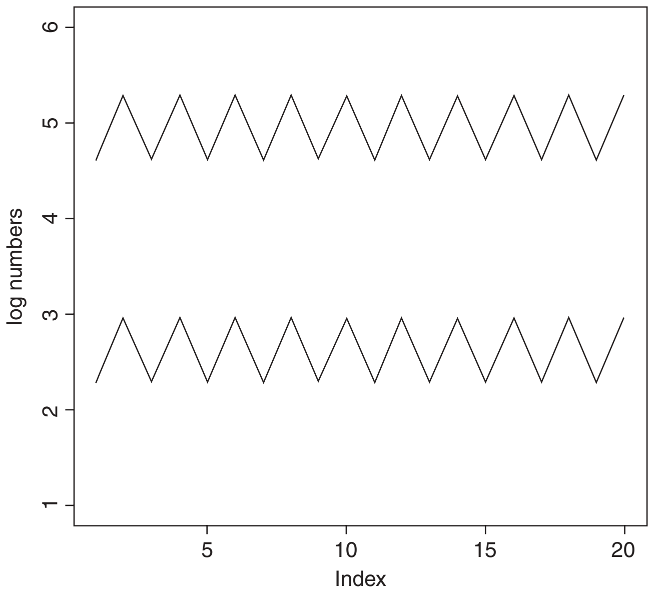
```
geometric(insects)
```

```
[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. 193).

### 2.15.4 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 is 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

$$\tilde{y}=\frac{1}{\left(\sum(1/y)\right)/n}=\frac{n}{\sum(1/y)}.$$

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
```

### 2.15.5 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':

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

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).

### 2.15.6 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 <- 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 10.254 55. 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
```

### 2.15.7 Variance ratio test

How do we know if two variances are significantly different from one another? One of several sensible ways to do this is to carry out Fisher's $F$ test, which is simply the ratio of the two variances (see p. 287). Here is a function to print the $p$ value (p. 347) 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 (`df1` = 9) and the number of degrees of freedom in the denominator (`df2` = 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
```

### 2.15.8 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':

$$\text{CI} = t_{\alpha/2,\text{d.f.}} \times se.$$

The R function `qt` gives the value of Student's $t$ with $1 - \alpha/2 = 0.975$ and degrees of freedom d.f. = `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 could 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)
```

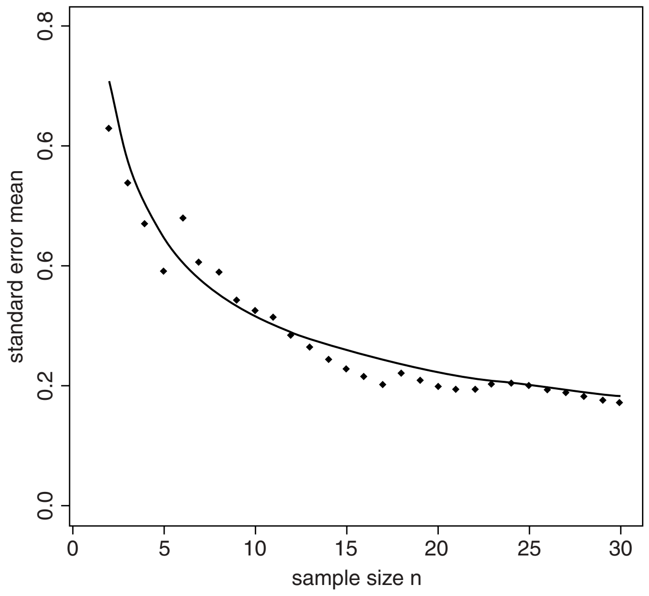
Now in a loop take samples of size 2, 3, 4, ..., 30:

```
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 $n > 24$.

### 2.15.9 Deparsing: A graphics function for 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. 204). 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`).

The process of deparsing turns an unevaluated expression into a character string. One of the important uses of deparsing is in functions that produce output that you want to label with the particular names of the variables that were passed to the function. For instance, if the function is written in terms of a continuous response variable `y` and a categorical explanatory variable x, you might want to label the axes of a plot produced by the function with, say, `clipping` and `biomass` in place of `x` and `y.` For instance, if the function is written in terms of a continuous response variable `yv,` you might want to label the axes of a plot produced by the function with, say, `biomass` in place of `yv.` Inside the `error.bars` function, the `barplot` function uses the `deparse` function to create the appropriate text for `ylab.`

```
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. 426):

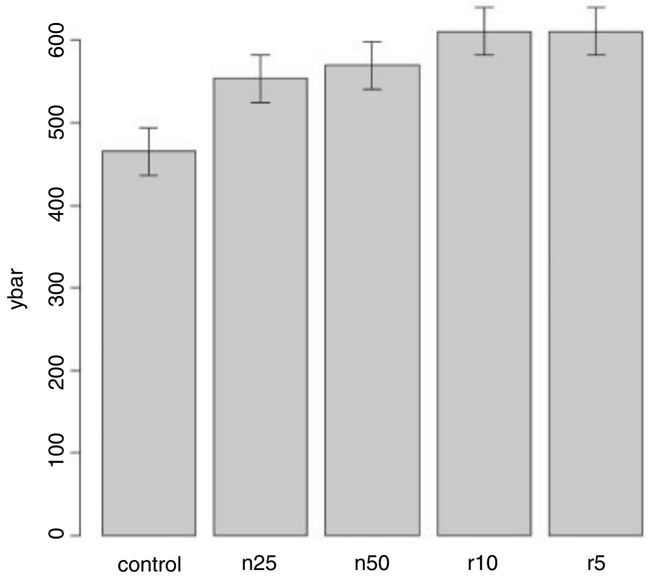
```
comp <- read.table("c:\\temp\\competition.txt",header=T)
attach(comp)
names(comp)
```

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

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

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

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

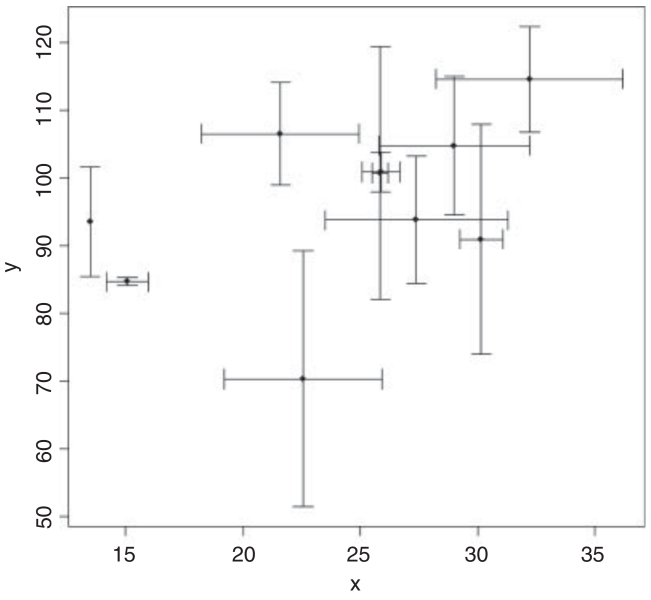


Here is a function to plot error bars on a scatterplot in both the *x* and *y* directions:

```
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)
```



### 2.15.10 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 (see pp. 115–119 for explanations of the separate functions). 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
```

### 2.15.11 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.

### 2.15.12 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.

### 2.15.13 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 blue 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")
```

This 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.

### 2.15.14 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 three 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).

### 2.15.15 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.

## 2.15.16 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, tapply, sapply` and `lapply` (p. 63).

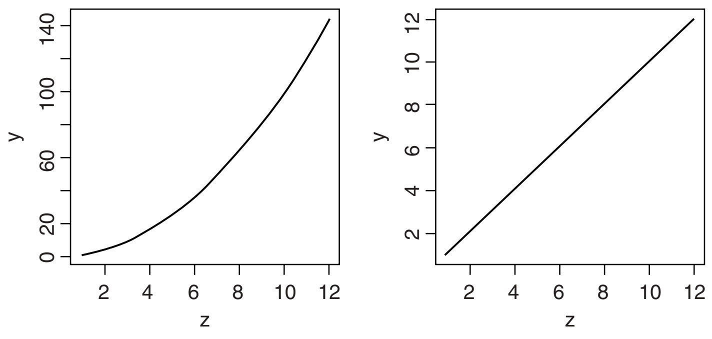
## 2.15.17 Flexible handling of arguments to functions

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). We rescale the `windows` (width then height in inches) so that the graphs come out looking roughly square rather than elongated:

```
windows(7,4)
par(mfrow=c(1,2))
plotx2(12)
plotx2(12,1:12)
```



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.

### 2.15.18 Structure of an object: `str`

Here is one of the simplest objects in R – a vector of length 7 containing real numbers:

```
(y <- seq(0.9,0.3,-0.1))
```

```
[1] 0.9 0.8 0.7 0.6 0.5 0.4 0.3
```

We can ask R about the structure of the object called `y` using `str`:

```
str(y)
```

```
num [1:7] 0.9 0.8 0.7 0.6 0.5 0.4 0.3
```

We discover that it is `numeric` (in both class and mode), a vector of length 7 `[1:7]`, and (because the vector is short) we see all of the values listed. For longer vectors we would see the first few values, depending on what would fit on a single printed line (as affected by the number of decimal places displayed).

What about a slightly more complicated object? Here is a dataframe with two columns:

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

```
'data.frame': 109 obs. of 2 variables:
 $ condition: Factor w/ 5 levels "better","much.better",..: 4 1 1 4 4 4 1 5 4 1 ...
 $ treatment: Factor w/ 3 levels "drug.A","drug.B",..: 1 2 2 3 2 2 1 1 2 2 ...
```

We learn that `data` is a dataframe with 109 rows and 2 columns, then we get detailed information on each of the columns in turn. The first is a variable called `condition` which is a factor with five levels (the first two levels of which (in alphabetical order) are `better` and `much.better`). The second variable is called `treatment` and is a factor with three levels. The numbers are the integer representations of the factor levels in the first 10 rows of the dataframe. Because we can see only factor levels 1 and 2, we would need to do more work to discover what factor level 4 of condition, or level 3 of treatment, actually represented:

```
levels(data$condition);levels(data$treatment)
```

```
[1] "better" "much.better" "much.worse" "no.change" "worse"
[1] "drug.A" "drug.B"      "placebo"
```

We often want to know about the structure of model objects. Here is the simplest case, with a linear regression model (see p. 450 for details):

```
reg <- read.table("c:\\temp\\tannin.txt",header=T)
reg.model <- lm(growth~tannin,data=reg)
str(reg.model)
```

```
List of 12
 $ coefficients : Named num [1:2] 11.76 -1.22
 ..- attr(*, "names")= chr [1:2] "(Intercept)" "tannin"
 $ residuals : Named num [1:9] 0.244 -0.539 -1.322 2.894 -0.889 ...
 ..- attr(*, "names")= chr [1:9] "1" "2" "3" "4" ...
 $ effects : Named num [1:9] -20.67 -9.42 -1.32 2.83 -1.01 ...
 ..- attr(*, "names")= chr [1:9] "(Intercept)" "tannin" "" "" ...
 $ rank : int 2
 $ fitted.values: Named num [1:9] 11.76 10.54 9.32 8.11 6.89 ...
 ..- attr(*, "names")= chr [1:9] "1" "2" "3" "4" ...
 $ assign : int [1:2] 0 1
 $ qr :List of 5
 ..$ qr : num [1:9, 1:2] -3 0.333 0.333 0.333 0.333 ...
 .. ..- attr(*, "dimnames")=List of 2
 .. .. ..$ : chr [1:9] "1" "2" "3" "4" ...
 .. .. ..$ : chr [1:2] "(Intercept)" "tannin"
 .. ..- attr(*, "assign")= int [1:2] 0 1
 ..$ qraux: num [1:2] 1.33 1.26
 ..$ pivot: int [1:2] 1 2
 ..$ tol : num 1e-07
 ..$ rank : int 2
 ..- attr(*, "class")= chr "qr"
 $ df.residual : int 7
 $ xlevels : Named list()
 $ call : language lm(formula = growth ~ tannin, data = reg)
 $ terms :Classes 'terms', 'formula' length 3 growth ~ tannin
 .. ..- attr(*, "variables")= language list(growth, tannin)
 .. ..- attr(*, "factors")= int [1:2, 1] 0 1
 .. .. ..- attr(*, "dimnames")=List of 2
 .. .. .. ..$ : chr [1:2] "growth" "tannin"
 .. .. .. ..$ : chr "tannin"
 .. ..- attr(*, "term.labels")= chr "tannin"
 .. ..- attr(*, "order")= int 1
 .. ..- attr(*, "intercept")= int 1
 .. ..- attr(*, "response")= int 1
 .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
 .. ..- attr(*, "predvars")= language list(growth, tannin)
 .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
 .. .. ..- attr(*, "names")= chr [1:2] "growth" "tannin"
 $ model :'data.frame': 9 obs. of 2 variables:
 ..$ growth: int [1:9] 12 10 8 11 6 7 2 3 3
 ..$ tannin: int [1:9] 0 1 2 3 4 5 6 7 8
 ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 growth ~
     tannin
 .. .. ..- attr(*, "variables")= language list(growth, tannin)
 .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
 .. .. .. ..- attr(*, "dimnames")=List of 2
 .. .. .. .. ..$ : chr [1:2] "growth" "tannin"
 .. .. .. .. ..$ : chr "tannin"
 .. .. ..- attr(*, "term.labels")= chr "tannin"
 .. .. ..- attr(*, "order")= int 1
 .. .. ..- attr(*, "intercept")= int 1
 .. .. ..- attr(*, "response")= int 1
 .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
 .. .. ..- attr(*, "predvars")= language list(growth, tannin)
 .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
 .. .. .. ..- attr(*, "names")= chr [1:2] "growth" "tannin"
 - attr(*, "class")= chr "lm"
```

There are 12 elements in the list representing the structure of this linear model object: coefficients, residuals, effects, rank, fitted values, assign, qr, residual degrees of freedom, xlevels, call, terms and model. Each of these, in turn, is broken down into components; for instance, the two coefficients are numbers (11.76 and –1.22), and their names are (`Intercept`) and `tannin`. You should work down the list and see if you can figure out why each row is an important part of the model.

For more complicated models, the structure is even more involved. Here is the structure of a generalized linear model with a binary response and binomial errors:

```
data <- read.table("c:\\temp\\spino.txt",header=T)
attach(data)
y <- factor(1+(condition=="better")+(condition=="much.better"))
model <- glm(y~treatment,binomial)
summary(model)

Call:
glm(formula = y ~ treatment, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-0.9741 -0.9741 -0.7747 1.3953 1.6431
```

```
Coefficients:
                  Estimate Std. Error z value Pr(>|z|)
(Intercept)        -0.6131     0.3444  -1.780    0.075 .
treatmentdrug.B     0.1141     0.4617   0.247    0.805
treatmentplacebo   -0.4367     0.5581  -0.783    0.434
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance:     139.67 on 108 degrees of freedom
Residual deviance: 138.54 on 106 degrees of freedom
AIC:               144.54
```

We have carried out a one-way analysis of deviance with a two-level response (improved or not) and a three-level factor as explanatory variable (`treatment`). There was no significant difference between `drug.B` and the placebo, nor between either of these and `drug.A` (the intercept). Here is the structure of the object called `model`:

```
str(model)
```

As you will see, this is a very large object, comprising a list with 30 components covering all aspects of the model: the coefficients, fitted values, effects plus all the details of the family and the model formula. I recommend you work your way slowly down the whole list and try to understand why each of the rows represents an essential piece of information about the model.

## 2.16 Writing from R to file

You often want to save an object that you have created in R.

### 2.16.1 Saving your work

To save your current R session, so that you can load it again later and continue your work where you left off, use `save` like this:

```
save(list = ls(all=TRUE), file= "c:\\temp\\session")
```

Then, on another occasion, when you want to restore the data, use `load` like this:

```
load(file= "c:\\temp\\session")
```

### 2.16.2 Saving history

It is very useful to be able to see all of the lines of R code that one has typed during a particular session. You may want to copy the lines into a text editor to make minor alterations, or you may simply want to paste multiple lines back into R to repeat certain operations. To see all of your lines of input code just type:

```
history(Inf)
```

This opens a window called R History through which you can scroll, highlight and copy using Ctrl + C. You could then open a new Untitled R Editor window (File > New Script) and paste the selected lines of code using Ctrl + V. Alternatively, you might want to save the entire history to file, for use on a subsequent occasion:

```
savehistory(file = "c:\\temp\\session18.txt")
```

To retrieve the history for use on another occasion use:

```
loadhistory(file = "c:\\temp\\session18.txt")
```

Then you can access it by `history(Inf)` in the new session.

### 2.16.3 Saving graphics

For speed and simplicity, you can click on a graph (the bar on top of the R Graphics Device goes darker blue) then press Ctrl + C (to copy the graph), then switch to a word processor and paste using Ctrl + V. For publication-quality graphics, however, you will want to save each figure in a separate file as a PDF or PostScript file. There are a great many options (see `?pdf` and `?postscript` for details) but the basics are very simple. Here we set the graphics device to produce a PDF:

```
pdf("c:\\temp\\fig1.pdf")
```

Now, any plot directives are sent to this file. To switch off writing graphics to file, type:

```
dev.off()
```

### 2.16.4 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 the temp directory 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. But 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. Here is a matrix of 1000 rows and 100 columns made up of random integers from a Poisson distribution with mean 0.75:

```
xmat <- matrix(rpois(100000,0.75),nrow=1000)
write.table(xmat,"c:\\temp\\table.txt",col.names=F,row.names=F)
```

The function adds 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
```

```
  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, ...) use

```
write.table(unclass(nbtable),"c:\\temp\\table.txt",col.names=F,row.names=F)
```

### 2.16.5 Pasting into an Excel spreadsheet

Writing a vector from R to the Windows Clipboard uses the function `writeClipboard(x)` where `x` is a single vector of **characters**, so you need to build up a spreadsheet in Excel by pasting (Ctrl + V) 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 press Ctrl + V in the second column, and so on.

### 2.16.6 Writing an Excel readable file from R

Suppose you want to transfer an entire dataframe called `data` to Excel (rather than one column, as above):

```
data <- read.table("c:\\temp\\worms.txt",header=T)
write.table(data,"clipboard",sep="\t",col.names=NA)
```

Then, in Excel, just press Ctrl + V or click on the Paste icon (the Clipboard). Your variable names will appear in the first row of the spreadsheet, with (unheaded) row numbers in the leftmost column.

## 2.17 Programming tips

- Know exactly what you are trying to achieve.
- Keep it simple.

- Clever is good, but clear is better.
- Test each line as you go along, to make sure it does what you want it to do.
- Put plenty of comments in the code, using # for documentation.
- Use variable names and function names that are self-explanatory.
- Do not use `attach` in programs.
- Use `with`, or refer to variables within named dataframes.
- Try different ways of doing the same thing, and select the fastest method.
- Use indents (tabs) to improve clarity of loops and `if` statements.
- Build up the program from small, independently tested functions.
- Stop tinkering once it works effectively.