R Programming

Robin Evans
`robin.evans@stats.ox.ac.uk`

Michaelmas 2014

This version: November 5, 2014

Administration

The course webpage is at

`http://www.stats.ox.ac.uk/~evans/teaching.htm`

Lectures are at 10am on Mondays and Wednesdays, and practicals at 9am on Tuesdays and Thursdays; in reality, there will be rather a lot of overlap between these two formats.

Please bring your own laptop to use during all classes, and ensure that you have `R` working (see below). If you don't have access to a laptop, let me know and we will try to provide one.

I will hold office hours each week during Michaelmas term on **Wednesdays between 12pm and 1pm**; my office is on the first floor of 2 SPR, room 204. I'm very happy to help with any difficulties or problems you are having with `R`, but **please take steps to help yourselves first** (see below for a list of resources).

Software

You should install `R` on your own computer at the first opportunity. Visit

`http://cran.r-project.org/`

for details. Ensure you have the latest version (as of the start of Michaelmas 2014, this was version 3.1.1). Try to spend some time getting used to the basics of the software, including arithmetic operations and functions. There are many excellent online tutorials for this purpose.

Resources

A strength of `R` is its **help files**, which we will discuss. These are accessed with the `?` and `??` commands.

The **internet** has almost all the answers, and knows much more about `R` than I do. If you have a problem, it's extremely likely that someone will have had the same difficulty already, and posted a question on an internet forum.

Books are useful, though not required. Here are a some of them with brief comments.

1. Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S.* Springer-Verlag. 4th edition.

The classic text.

2. Chambers (2010) - Software for Data Analysis: Programming with R, Springer.

One of few books with information on more advanced programming (S4, overloading).

3. Wickham, H. (2014) Advanced R. Chapman and Hall.

A great new book on the more advanced features: a good follow up to this class.

4. Crawley, M. (2007) The R Book. Wiley.

Very thorough.

5. Fox, J. (2002) A R and S-PLUS Companion to Applied Regression. Sage.

Does what it says.

6. Ligges, U. (2009) Programmieren mit R. Third edition. Springer.

In German(!)

7. Rizzo, M. L. (2008) Statistical Computing with R. CRC/Chapman & Hall.

More computational – different examples to the other books.

8. Braun, W. J. and Murdoch, D. J. (2007) A First Course in Statistical Programming with R. CUP.

Detailed and well written, but at a rather low level. A bit redundant given the above.

9. Maindonald J. and Braun, W. J. (2003) Data Analysis and Graphics using R Second or third edition CUP.

Advanced statistical graphics

10. Spector, P. (2008) Data Manipulation with R. Springer

Especially for data manipulation.

11. Dalgaard, P. (2009) Introductory Statistics with R. Second Edition. Springer.

Probably redundant given the above.

Getting the Most out of the Class

Learning `R` has much in common with learning a natural language: it's easy to get going with a few simple phrases, though you'll find some idiosyncrasies in the syntax, and occasional aspects are downright illogical. Once you've mastered these few difficulties, the only barrier to fluency is the vast vocabulary of `R`: even in the basic packages there are many commands which you will never use or understand, but the more you learn the more elegantly you will be able to express yourself. There is a smaller core of 'everyday' language which we will focus on, and which you will be expected to understand in exams and practical assessments.

These lecture notes are intended for reference, and will (by the end of the course) contain sections on all the major topics we cover. Lectures will not follow the notes exactly, so be prepared to take your own notes; the practical classes will complement the lectures, and you can be examined on anything we study in either.

Don't copy and paste the commands from this guide into `R`; you will find it very hard to remember the details of the language and will have to look everything up when you come to code something yourself.

Make sure you **try the exercises**, and understand the code involved in each one; if something doesn't make sense, use `R`'s help functions, ask a classmate, try using internet resources, or ask me for help (preferably in that order). Some exercises are marked with an asterisk (*), which means they are a little more advanced than is necessary for the class.

If you find any mistakes or omissions in these notes, I'd be very grateful to be informed.

1 Introduction

1.1 What R is good at

Statistics for relatively advanced users: R has thousands of packages, designed, maintained, and widely used by statisticians.

Statistical graphics: try doing some of our plots in Stata and you won't have much fun.

Flexible code: R has a rather liberal syntax, and variables don't need to be declared as they would in (for example) C++, which makes it very easy to code in. This also has disadvantages in terms of how safe the code is.

Vectorization: R is designed to make it very easy to write functions which are applied pointwise to every element of a vector. This is extremely useful in statistics.

R is powerful: if a command doesn't exist already, you can code it yourself.

1.2 What R is not so good at

Statistics for non-statisticians: there is a steep learning curve, which puts some people off. *Try Stata, SAS or SPSS (if you must).*

Numerical methods, such as solving partial differential equations; *try Matlab.*

Analytical methods, such as algebraically integrating a function. *Try Mathematica or Maple.*

Precision graphics, such as might be useful in psychology experiments. *Try Matlab.*

Optimization. Though it does have some very easy to use methods built-in.

Low-level, high-speed or critical code; *use C, C++, Java or similar.* (However note that such code can be called from R to give the 'best of both worlds'.

1.3 General Properties

R makes it extremely easy to code complex mathematical or statistical procedures, though the programs may not run all that quickly. You can interface R with other languages (C, C++, Fortran) to provide fast implementations of subroutines, but writing this code (and making it portable) will typically take longer. Where the advantage falls in this trade-off will depend upon what you're doing; for most things you will encounter during your degree, `R` is sufficiently fast.

`R` is open source and widely adopted by statisticians, biostatisticians, and geneticists. There is a huge wealth of existing libraries so you can often save time by using these, though it is sometimes easier to start from scratch than to adapt someone else's function to meet your needs. Contributing new packages to the central repository (CRAN) is easy: even your lecturer has managed it. As a result, `R` packages are not build to very high standards (but see Bioconductor).

`R` is portable, and works equally well on Windows, OS X and Linux.

1.4 Interfaces

For Windows and OS X, the standard `R` download comes with an `R` GUI, which is adequate for simple tasks. You can also run `R` from the command line in any operating system.

There are a number of more powerful interfaces which you may like to try. Here's a few.

RStudio. Very popular, with a nice interface and well thought out, especially for more advanced usage: can be a bit buggy, so make sure you update it regularly. Available on all platforms.

Emacs with ESS. (Emacs Speaks Statistics) is available on all platforms, and is very powerful when you get used to it. Has a habit of freezing in my experience, though.

TinnR. Alternative Windows interface.

I intend to demonstrate a few of these different approaches during class.

2 Basic Arithmetic and Objects

R has a command line interface, and will accept simple commands to it. This is marked by a > symbol, called the **prompt**. If you type a command and press return, R will evaluate it and print the result for you.

```
> 6 + 9

[1] 15

> x <- 15
> x - 1

[1] 14
```

The expression x <- 15 creates a variable called x and gives it the value 15. This is called **assignment**; the variable on the left is assigned to the value on the right. The left hand side must contain only contain a single variable.

```
> x + 4 <- 15        # doesn't work
```

Assignment can also be done with = (or ->).

```
> x = 5
> 5*x -> x
> x

[1] 25
```

The operators = and <- are identical, but many people prefer <- because it is not used in any other context, but = is, so there is less room for confusion.

2.1 Vectors

The key feature which makes R very useful for statistics is that it is **vectorized**. This means that many operations can be performed point-wise on a vector. The function c() is used to create vectors:

```
> x <- c(1, -1, 3.5, 2)
> x

[1]  1.0 -1.0  3.5  2.0
```

Then if we want to add 2 to everything in this vector, or to square each entry:

```
> x + 2

[1] 3.0 1.0 5.5 4.0

> x^2

[1]  1.00  1.00 12.25  4.00
```

This is very useful in statistics:

```
> sum((x - mean(x))^2)

[1] 10.69
```

Exercise 2.1. The weights of five people before and after a diet programme are given in the table.

Before	78	72	78	79	105
After	67	65	79	70	93

Read the 'before' and 'after' values into two different vectors called `before` and `after`. Use `R` to evaluate the amount of weight lost for each participant. What is the average amount of weight lost?

***Exercise 2.2.** How would you write a function equivalent to `sum((x - mean(x))^2)` in a language like C or Java?

Some useful vectors can be created quickly with `R`. The colon operator is used to generate integer sequences

```
> 1:10

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

```
> -3:4

[1] -3 -2 -1  0  1  2  3  4

> 9:5

[1] 9 8 7 6 5
```

More generally, the function `seq()` can generate any arithmetic progression.

```
> seq(from=2, to=6, by=0.4)

 [1] 2.0 2.4 2.8 3.2 3.6 4.0 4.4 4.8 5.2 5.6 6.0

> seq(from=-1, to=1, length=6)

[1] -1.0 -0.6 -0.2  0.2  0.6  1.0
```

Sometimes it's necessary to have repeated values, for which we use `rep()`

```
> rep(5,3)

[1] 5 5 5

> rep(2:5,each=3)

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

> rep(-1:3, length.out=10)

 [1] -1  0  1  2  3 -1  0  1  2  3
```

We can also use `R`'s vectorization to create more interesting sequences:

```
> 2^(0:10)

 [1]    1    2    4    8   16   32   64  128  256  512 1024
```

```
> 1:3 + rep(seq(from=0,by=10,to=30), each=3)

 [1]  1  2  3 11 12 13 21 22 23 31 32 33
```

The last example demonstrates **recycling**, which is also an important part of vectorization. If we perform a binary operation (such as `+`) on two vectors of different lengths, the shorter one is used over and over again until the operation has been applied to every entry in the longer one. If the longer length is not a multiple of the shorter length, a warning is given.

```
> 1:10 * c(-1,1)

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

> 1:7 * 1:2

Warning:  longer object length is not a multiple of shorter object
length

[1]  1  4  3  8  5 12  7
```

Exercise 2.3. Create the following vectors in `R` using `seq()` and `rep()`.

(i) $1, 1.5, 2, 2.5, \ldots, 12$

(ii) $1, 8, 27, 64, \ldots, 1000$.

(iii) $1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \ldots, -\frac{1}{100}$.

(iv) $1, 0, 3, 0, 5, 0, 7, \ldots, 0, 49$.

(v) $1, 3, 6, 10, 15, \ldots, \sum_{i=1}^{n} i, \ldots, 210$ [look up `?cumsum`].

(vi) * $1, 2, 2, 3, 3, 3, 4, \ldots, 9, \underbrace{10, \ldots, 10}_{\text{10 times}}$. [Hint: type `?seq`, and read about the `times` argument.]

Exercise 2.4. The ith term in the Taylor expansion of $\log(1+x)$ is $(-1)^{i+1}x^i/i$. Create a vector containing the first 100 terms for $x = 0.5$. *[Write out the first few entries by hand if that helps.]*

Let

$$r_n(x) = \log(1+x) - \sum_{i=1}^{n} \frac{(-1)^{i+1}x^i}{i}.$$

Evaluate $r_n(1)$ for $n = 10, 100, 1000, \ldots, 10^6$.

2.2 Subsetting

It's frequently necessary to extract some of the elements of a larger vector. In R you can use square brackets to select an individual element or group of elements:

```
> x <- c(5,9,2,14,-4)
> x[3]

[1] 2

> # note indexing starts from 1
> x[c(2,3,5)]

[1]  9  2 -4

> x[1:3]

[1] 5 9 2

> x[3:length(x)]

[1]  2 14 -4
```

There are two other methods for getting subvectors. The first is using a logical vector (i.e. containing TRUE and FALSE) of the same length:

```
> x > 4

[1]  TRUE  TRUE FALSE  TRUE FALSE

> x[x > 4]

[1]  5  9 14
```

or using negative indices to specify which elements should not be selected:

```
> x[-1]

[1]  9  2 14 -4

> x[-c(1,4)]

[1]  9  2 -4
```

(Note that this is rather different to what other languages such as C or Python would interpret negative indices to mean.)

Exercise 2.5. The built-in vector `LETTERS` contains the uppercase letters of the alphabet. Produce a vector of (i) the first 12 letters; (ii) the odd 'numbered' letters; (iii) the (English) consonants.

2.3 Logical Operators

As we see above, the comparison operator `>` returns a logical vector indicating whether or not the left hand side is greater than the right hand side. Here we demonstrate the other comparison operators:

```
> x <= 2   # less than or equal to

[1] FALSE FALSE  TRUE FALSE  TRUE

> x == 2   # equal to

[1] FALSE FALSE  TRUE FALSE FALSE

> x != 2   # not equal to

[1]  TRUE  TRUE FALSE  TRUE  TRUE
```

Note the double equals sign `==`, to distinguish between assignment and comparison.

We may also wish to combine logical vectors. If we want the elements of `x` within a range, we can use the following:

```
> (x > 0) & (x < 10)    # 'and'

[1]  TRUE  TRUE  TRUE FALSE FALSE
```

The `&` operator does a pointwise 'and' comparison between the two sides. Similarly, the vertical bar `|` does pointwise 'or', and the unary `!` operator performs negation.

```
> (x == 5) | (x > 10)

[1]  TRUE FALSE FALSE  TRUE FALSE

> !(x > 5)

[1]  TRUE FALSE  TRUE FALSE  TRUE
```

Exercise 2.6. The function `rnorm()` generates normal random variables. For instance, `rnorm(10)` gives a vector of 10 i.i.d. standard normals. Generate 20 standard normals, and store them as `x`. Then obtain subvectors of

(i) the entries in `x` which are less than 1;

(ii) the entries between $-\frac{1}{2}$ and 1;

(iii) the entries whose absolute value is larger than 1.5.

2.4 Character Vectors

As you might have noticed in the exercise above, vectors don't have to contain numbers. We can equally create a **character vector**, in which each entry is a string of text. Strings in `R` are contained within double quotes `"`:

```
> x <- c("Hello", "how do you do", "lovely to meet you", 42)
> x

[1] "Hello"              "how do you do"      "lovely to meet you"
[4] "42"
```

Notice that you cannot mix numbers with strings: if you try to do so the number will be converted into a string. Otherwise character vectors are much like their numerical counterparts.

```
> x[2:3]

[1] "how do you do"      "lovely to meet you"

> x[-4]

[1] "Hello"              "how do you do"      "lovely to meet you"

> c(x[1:2], "goodbye")

[1] "Hello"         "how do you do" "goodbye"
```

2.5 Matrices

Matrices are much used in statistics, and so play an important role in R. To create a matrix use the function `matrix()`, specifying elements by column first:

```
> matrix(1:12, nrow=3, ncol=4)


     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

This is called **column-major order**. Of course, we need only give one of the dimensions:

```
> matrix(1:12, nrow=3)
```

unless we want vector recycling to help us:

```
> matrix(1:3, nrow=3, ncol=4)


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

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

Sometimes it's useful to specify the elements by row first

```
> matrix(1:12, nrow=3, byrow=TRUE)
```

There are special functions for constructing certain matrices:

```
> diag(3)


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

> diag(1:3)


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

> 1:5 %o% 1:5


     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

The last operator performs an **outer product**, so it creates a matrix with (i, j)-th entry $x_i y_j$. The function `outer()` generalizes this to any function f on two arguments, to create a matrix with entries $f(x_i, y_j)$. (More on functions later.)

```
> outer(1:3, 1:4, "+")


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

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

Matrix multiplication is performed using the operator %*%, which is quite distinct from scalar multiplication *.

```
> A <- matrix(c(1:8,10), 3, 3)
> x <- c(1,2,3)
> A %*% x    # matrix multiplication


     [,1]
[1,]   30
[2,]   36
[3,]   45


> A*x      # NOT matrix multiplication


     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   30
```

Standard functions exist for common mathematical operations on matrices.

```
> t(A)      # transpose


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


> det(A)    # determinant


[1] -3


> diag(A)   # diagonal


[1]  1  5 10
```

```
> solve(A)  # inverse

        [,1]    [,2] [,3]
[1,] -0.6667 -0.6667    1
[2,] -1.3333  3.6667   -2
[3,]  1.0000 -2.0000    1
```

Exercise 2.7. Construct the matrix

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 2 & 6 \\ -3 & -1 & -3 \end{pmatrix}$$

Show that $B \times B \times B$ is a scalar multiple of the identity matrix, and find the scalar.

Matrices can be subsetted much the same way as vectors, although of course they have two indices. Row number comes first:

```
> A[2,1]

[1] 2

> A[2,2:ncol(A)]

[1] 5 8

> A[,1:2]      # blank indices give everything

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

> A[c(),1:2]   # empty indices give nothing!

     [,1] [,2]
```

Notice that, where appropriate, R automatically reduces a matrix to a vector or scalar when you subset it. You can override this using the optional drop argument.

```
> A[2,2:ncol(A),drop=FALSE]    # returns a matrix

     [,1] [,2]
[1,]    5    8
```

You can stitch matrices together using the `rbind()` and `cbind()` functions. These employ vector recycling:

```
> cbind(A, t(A))

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

> rbind(A, 1, 0)

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

Exercise 2.8. Construct the following matrices:

(a)

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 1 & -1 & 1 & \cdots & -1 \\ 1 & -1 & 1 & \cdots & -1 \\ \vdots & \vdots & & & \vdots \\ 1 & -1 & 1 & \cdots & -1 \end{pmatrix} \qquad \text{(dimensions } 15 \times 10\text{).}$$

(c) The 5×15 matrix with three 1s in shifting positions:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & & & & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 \end{pmatrix} \qquad \text{(dimensions } 5 \times 15\text{).}$$

[Hint: use column subsetting.]

(d)

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & 9 & 10 \\ 2 & 3 & 4 & \cdots & 10 & 11 \\ 3 & 4 & 5 & & & \vdots \\ \vdots & \vdots & & \ddots & & 17 \\ 9 & 10 & & & 17 & 18 \\ 10 & 11 & \cdots & 17 & 18 & 19 \end{pmatrix};$$

[Look at the `outer()` function.]

(e)

$$\begin{pmatrix} 1 & 2 & 3 & 4 & \cdots & 9 \\ 2 & 3 & 4 & & & 1 \\ 3 & 4 & \ddots & & & \vdots \\ 4 & & & \ddots & & 6 \\ \vdots & & & & 6 & 7 \\ 9 & 1 & \cdots & 6 & 7 & 8 \end{pmatrix};$$

[The modular arithmetic operator `%%` may be useful here.]

(f)

$$\begin{pmatrix} I_5 & \mathbf{1} \\ \mathbf{0} & -I_6 \end{pmatrix}$$

where I_k is the $k \times k$-identity matrix, and $\mathbf{1}$ and $\mathbf{0}$ are matrices with all entries 1 and 0 respectively.

Exercise 2.9. Solve the following system of simultaneous equations using matrix methods.

$$\begin{aligned} a + 2b + 3c + 4d + 5e &= -5 \\ 2a + 3b + 4c + 5d + e &= 2 \\ 3a + 4b + 5c + d + 2e &= 5 \\ 4a + 5b + c + 2d + 3e &= 10 \\ 5a + b + 2c + 3d + 4e &= 11 \end{aligned}$$

Don't just create your matrix by hand!

Exercise 2.10. In this section we've seen that the behaviour of the function `diag()` depends upon its inputs. Can you think of some examples where this might cause a problem?

2.6 Lists

Other than vectors and matrices, the main object for holding data in R is a list[1]. These are a bit like vectors, except that each entry can be any other R object, even another list.

```
> x <- list(1:3, TRUE, "Hello", list(1:2, 5))
```

Here x has 4 elements: a numeric vector, a logical, a string and another list. We can select an entry of x with double square brackets:

```
> x[[3]]


[1] "Hello"
```

To get a sub-list, use single brackets:

```
> x[c(1,3)]


[[1]]
[1] 1 2 3

[[2]]
[1] "Hello"
```

Notice the difference between x[[3]] and x[3].

We can also **name** some or all of the entries in our list, by supplying argument names to list():

```
> x <- list(y=1:3, TRUE, z="Hello")
> x


$y
[1] 1 2 3

[[2]]
[1] TRUE

$z
[1] "Hello"
```

[1] Technically speaking, lists are also a kind of vector in R, but not every object in them has to have the same type; ordinary logical, numeric or character vectors are known as **atomic vectors**.

Notice that the `[[1]]` has been replaced by `$y`, which gives us a clue as to how we can recover the entries by their name. We can still use the numeric position if we prefer:

```
> x$y

[1] 1 2 3

> x[[1]]

[1] 1 2 3
```

The function `names()` can be used to obtain a character vector of all the names of objects in a list.

```
> names(x)

[1] "y" ""  "z"
```

You've seen most standard `R` objects now: almost all the more complicated ones are just lists! We'll see this in the next section.

3 Data

R comes with many datasets built-in, particularly in the MASS package. A **package** is a collection (or library) of functions, datasets, and other objects; most packages are not loaded automatically, so you have to do it yourself:

```
> library(MASS)
```

You can now access various datasets from this package. Try looking at the dataset called hills.

```
> head(hills)
```

To find out what the data represent, use the help function ?hills.

3.1 Data Frames

The object hills is something called a **data frame**. A data frame is a series of records represented by rows (in this case one for each race), each containing values in several fields (in this case dist, climb, time).

You can check that hills is a data frame by inspecting its class(es)

```
> class(hills)

[1] "data.frame"
```

or more reliably by using an is() command

```
> is(hills, "data.frame")

[1] TRUE
```

We'll talk more about classes later in the course.

Data frames share many of the characteristics of matrices. We can select rows or columns in the same way:

```
> hills[3,]

             dist climb  time
Craig Dunain    6   900 33.65
```

```
> hills[hills$dist >= 12,]


              dist climb   time
Bens of Jura    16  7500 204.62
Lairig Ghru     28  2100 192.67
Seven Hills     14  2200  98.42
Two Breweries   18  5200 170.25
Moffat Chase    20  5000 159.83
```

However, they also behave like lists indexed by the columns:

```
> hills$time


 [1]  16.08  48.35  33.65  45.60  62.27  73.22 204.62  36.37  29.75  39.75
[11] 192.67  43.05  65.00  44.13  26.93  72.25  98.42  78.65  17.42  32.57
[21]  15.95  27.90  47.63  17.93  18.68  26.22  34.43  28.57  50.50  20.95
[31]  85.58  32.38 170.25  28.10 159.83
```

The truth is that, like almost all complicated objects in `R`, data frames are lists with some additional structure. Formally speaking, they are not matrices, but they do behave similarly in certain circumstances.

Exercise 3.1. How do the results of the following commands differ from what we would expect if hills were a matrix?

```
> hills[1,]
> hills[3]
> hills %*% c(1,2,4)
> mean(hills)
```

3.2 Manipulating Data using `with()`

We often want to use functions on the columns of a data frame, and it quickly becomes inconvenient to repeatedly type (for example) `hills$` before every such event. For example, the command below will give a scatter plot of the race times against climbs, amongst only those races less than 10 miles long.

```
> plot(hills$climb[hills$dist < 10], hills$time[hills$dist < 10])
```

The `with()` function allows us to refer to the names of objects in a data frame (or, in fact, any list) without having to keep referring to the data frame itself. For example, the command above becomes

```
> with(hills, plot(climb[dist < 10], time[dist < 10]))
```

If you just type `climb` or `dist` on their own, R won't know what object you're referring to. Technically `with()` alters the **scope** of the expression being evaluated (i.e. the code given in the second argument) so that it can 'see' the columns of the data frame as objects. We'll learn a bit more about scope when we talk about functions later on.

Exercise 3.2. Using `with()`, find the mean of the average speeds (in miles per hour) for races which are between 5 and 10 miles long

3.3 Creating Data Frames

The command `data.frame()` is used to create a data frame, each argument representing a column.

```
> books <- data.frame(author=c("Ripley", "Cox", "Snijders", "Cox"),
+                     year=c(1980, 1979, 1999, 2006),
+                     publisher=c("Wiley", "Chapman", "Sage", "CUP"))
> books


    author year publisher
1   Ripley 1980     Wiley
2      Cox 1979   Chapman
3 Snijders 1999      Sage
4      Cox 2006       CUP
```

Exercise 3.3. (a) Create a small data frame representing a database of films. It should contain the fields `title`, `director`, `year`, `country`, and at least three films.

(b) Create a second data frame of the same format as above, but containing just one new film.

(c) Merge the two data frames using `rbind()`.

(d) Try sorting the titles using `sort()`: what happens?

3.4 Factors

There are two main types of data which you will encounter this year: numerical and categorical. We've seen how to create numerical vectors already.

Suppose we have the heights of 100 individuals, the first 50 male and the rest female.

```
> set.seed(1442)        # fixes the random numbers
> height = round(rnorm(100, mean=rep(c(170,160),each=50), sd=10))
> sex = rep(c("M", "F"), each=50)
> head(sex)

[1] "M" "M" "M" "M" "M" "M"
```

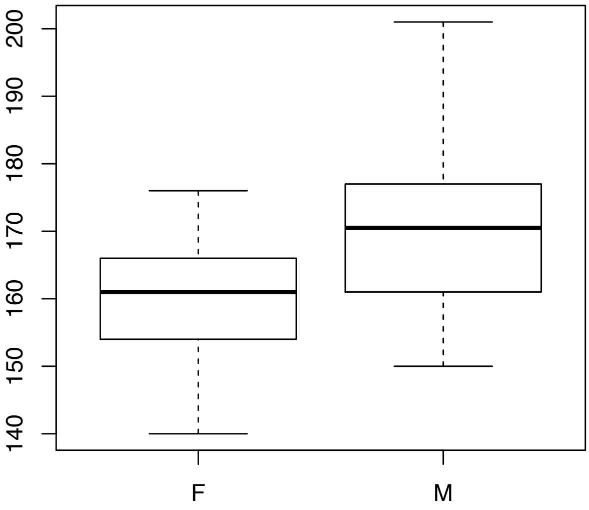
We can tell `R` to treat `sex` as a categorical variable:

```
> Sex = as.factor(sex)
> head(Sex)

[1] M M M M M M
Levels: F M
```

Note that it is displayed slightly differently. The new variable `Sex` is called a **factor**; a factor is a categorical variable which takes various discrete **levels**, in this case `M` and `F` for male and female.

`R` knows to do sensible things with factors:

```
> plot(Sex, height)
```



What happens if you try to plot `sex` against `height` instead? The distinction between categorical and non-categorical data is especially important if we have numbered groups.

The information in a factor is stored as a vector of integers:

```
> as.integer(Sex)


  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Just as a data frame is really a list, a factor is really a vector of integers (for levels) together with some extra information giving each level a names. The additional information is contained within a list of **attributes**. You can view this list directly.

```
> attributes(Sex)


$levels
[1] "F" "M"
```

```
$class
[1] "factor"
```

The attributes in this case are its **class** (you'll see this in many objects) and a vector of the level names. The class tells R that this object should be treated as a factor so that, for example, it will be displayed to you in the right way.

You may find that sometimes data are stored as a factor when you don't want them to be (see the exercise in the previous section). You can turn a factor back in to a character vector easily enough:

```
> as.character(Sex)

  [1] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M"
 [18] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M"
 [35] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "F"
 [52] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
 [69] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
 [86] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
```

Exercise 3.4. (a) Sample the numbers $\{1, 2, 3\}$ uniformly with replacement 50 times; use this to create a factor with levels `Yes`, `No` and `Maybe`.

(b) Create a subvector by removing the `Maybe` entries from the factor above. What levels does the new factor have?

(c) Use the command `droplevels()` to remove the level `Maybe`.

Exercise 3.5. Take a look at the `birthwt` data from the `MASS` package. How is `race` stored in these data? Is this sensible?

Define a factor based on `race`:

```
> Race = factor(birthwt$race)
```

Compare the effect of the commands `summary()`, `plot()` and `mean()` on each of `Race` and `birthwt$race`. Which do you find more useful?

3.5 Row and Column Names

The labels above and to the left of the values in `hills` are not part of the data itself, but can be accessed:

```
> names(hills)

[1] "dist"  "climb" "time"

> row.names(hills)

 [1] "Greenmantle"      "Carnethy"         "Craig Dunain"
 [4] "Ben Rha"          "Ben Lomond"       "Goatfell"
 [7] "Bens of Jura"     "Cairnpapple"      "Scolty"
[10] "Traprain"         "Lairig Ghru"      "Dollar"
[13] "Lomonds"          "Cairn Table"      "Eildon Two"
[16] "Cairngorm"        "Seven Hills"      "Knock Hill"
[19] "Black Hill"       "Creag Beag"       "Kildcon Hill"
[22] "Meall Ant-Suidhe" "Half Ben Nevis"   "Cow Hill"
[25] "N Berwick Law"    "Creag Dubh"       "Burnswark"
[28] "Largo Law"        "Criffel"          "Acmony"
[31] "Ben Nevis"        "Knockfarrel"      "Two Breweries"
[34] "Cockleroi"        "Moffat Chase"
```

As we saw above, in a data frame the column names can be used for indexing (e.g. `hills$time`); the row names cannot be used in this way.

This additional information is stored as **attributes**, which are in a separate list[2] attached to the object `hills`:

```
> attributes(hills)
```

We could add an attribute to `hills` if we wanted:

```
> attributes(hills) <- c(attributes(hills), list(type="races"))
> attributes(hills)
```

Note that `type(hills)` doesn't access `hills$type`, the most important attributes such as `names` and `class` happen to have functions named after them, which can be used to extract relevant information.

3.6 Reading in Data

Whilst `R` provides many interesting datasets, it is often necessary to load data externally. The main commands for doing this are `read.table()` and `read.csv()`.

[2] Actually they're not stored as a list (see `?attributes`), but they behave very similarly.

Here is an example using the `smoking.dat` dataset, available on the course website.

```
> dat <- read.table("smoking.dat", header=TRUE)
> head(dat)

  STATE   CIG BLAD  LUNG  KID LEUK
1    AL 18.20 2.90 17.05 1.59 6.15
2    AZ 25.82 3.52 19.80 2.75 6.61
3    AR 18.24 2.99 15.98 2.02 6.94
4    CA 28.60 4.46 22.07 2.66 7.06
5    CT 31.10 5.11 22.83 3.35 7.20
6    DE 33.60 4.78 24.55 3.36 6.45

> class(dat)

[1] "data.frame"
```

What happens if `header=TRUE` is omitted?

When you specify the file name, be sure to use the double quotes (`"`) around it. You also need to give the correct **path** to the file. `R` will automatically look for the file in its **working directory**. You can check what this is:

```
> getwd()

[1] "/data/redcrest/evans/Dropbox/Teaching/R Programming/2014"
```

Then if your file is in a subfolder called `files`, you need to write (for example)

```
> dat <- read.table("files/smoking.dat", header=TRUE)
```

In some systems you can use `file.choose()` to get the full path to a file. In particular this works well on `R` GUI for Windows or OS X. For example:

```
> dat <- read.table(file.choose(), header=TRUE)
```

will bring up a window for you to choose the file from.

***Exercise 3.6.** Look at the documentation for `read.table()`. Use the function to read in only lines 11 to 20 (Indiana up to Minnesota).

4 Functions

Everything which is done in R is done by functions. A function in a programming langauge is much like its mathematical equivalent: it has some inputs called **arguments**, and an output called the **return value**. In R a function can only return a single object. If you type a function's name at the console, you can see its structure:

```
> setdiff

function (x, y)
{
    x <- as.vector(x)
    y <- as.vector(y)
    unique(if (length(x) || length(y))
        x[match(x, y, 0L) == 0L]
    else x)
}
<bytecode: 0x2a49e98>
<environment: namespace:base>
```

There are two important parts to the function: the **signature**, which in this case is `function(x, y)`, and the **body**, which is the code between the curly brackets. Broadly speaking, when a function is **called**, it takes the information in the arguments, applies the code in the body to them, and then spits out the final expression in the function. In this case that's the complex looking expression `unique( )`.

4.1 Arguments

You can look at the arguments for a function by typing `args`:

```
> args(setdiff)

function (x, y)
NULL
```

Arguments are a little complicated in R. You'll notice that they have a name: the arguments of `setdiff()` are called `x` and `y`. However, you don't usually have to specify an argument by name, because arguments also have a position:

```
> a <- c(1,4,5,7)
> b <- c(1,2,5,9)
> setdiff(a, b)   # everything in 'a' and not in 'b'

[1] 4 7
```

It assumes that the first argument supplied should be `x`, and the second `y`. You can override this by specifying the name, and then the order doesn't matter:

```
> setdiff(y=b, x=a)

[1] 4 7

> setdiff(b, a)

[1] 2 9
```

If you specify *some* of the argument names but not all, then it will use the ordering to deduce the others.

```
> setdiff(y=b, a)  # 'a' must be the argument 'x'

[1] 4 7
```

Most functions don't require all of their arguments to be specified.

```
> x <- rnorm(10)
> y <- x + rnorm(10)
> lm(y ~ x)


Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.0212       0.7415


> args(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

The function `lm()` (which fits a linear model) only requires the single argument `formula` to run; the other arguments are **optional**: some of them have default values which are shown in the signature (such as `model = TRUE`), whereas others simply alter the behaviour of the function when specified (such as `data`).

`R` also uses **partial matching** for arguments, so as long as you give enough of the argument's name to make in unambiguous which one you mean, it will work:

```
> ## these two will do the same thing
> lm(form = y ~ x)
> lm(formula = y ~ x)
```

4.2 Writing Functions

To define your own function you just have to construct something in the same format as above:

```
> square = function(x) {
+   x^2
+ }
> square(4)

[1] 16
```

Objects which are created inside a function do not exist outside it:

```
> mean2 <- function(x) {
+   n <- length(x)
+   sum(x)/n
+ }
>
> mean2(1:10)

[1] 5.5
```

```
> n

Error:  object 'n' not found
```

Clearly an object called `n` was used inside the function above, but it was only inside the function's namespace. Most functions in `R` do not have **side effects**: they return a value, but do not change any of the objects which you can reach at the console. In order to *use* a function, you usually have to assign its output to something.

```
> x <- mean2(1:10)
> x

[1] 5.5
```

Exercise 4.1. The logit function is defined as

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right), \qquad 0 < x < 1.$$

Write an `R` function in one argument to implement this. How does your function behave for values of `x` such as 0, 1, or 2?

Exercise 4.2. Recall that the Taylor expansion of $\log(1 + x)$ is

$$\log(1+x) = \sum_{i=1}^{\infty}(-1)^{i+1}\frac{x^i}{i}$$

Write a function with arguments `x` and `n`, which calculates the Taylor approximation to $\log(1 + x)$ using n terms.

How many terms do you need to get within 10^{-6} of the correct solution when $x = 0.99$?

Exercise 4.3. Given real vectors x, y of length n, the least squares slope $(\alpha, \beta)^T$ is given by

$$\beta = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$
$$\alpha = \bar{y} - \bar{x}\beta.$$

Write a function which takes two arguments, `x` and `y`, and returns a vector of length 2 containing α and β. Verify that your function gives the correct answer using `R`'s built-in function `lm()` [the syntax is `lm(y~x)`].

4.3 `for()` Loops

The most common way to execute a block of code multiple times is with a `for()` loop. What's going on in the code below?

```
> factorial2 = function(n) {
+   out = 1

+   for (i in 1:n) {
+     out = out*i
+   }

+   out
+ }
> factorial2(10)


[1] 3628800
```

You may have seen `for()` loops in other languages. The syntax in `R` is `for (i in x)` for some vector (or list) `x`, where `i` will take each value in `x`. Most commonly, `x` is a vector of the first `n` natural numbers.

`i` is a dummy variable, and can be called whatever you like, though it retains its value outside the loop.

```
> for (sillyname in 1:4) print(sillyname)


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


> sillyname


[1] 4
```

Exercise 4.4. Write a function to perform matrix-vector multiplication. It should take a matrix `A` and a vector `b` as arguments, and return the vector *Ab*. Use **two** loops to do this, rather than `%*%` or any vectorization.

I generally recommend using `seq_len()` or `seq_along()` in `for()` loops, because it always behaves the way you want (and runs quicker than `seq()`):

```
> n = 0
> 1:n  # not a sequence of length n=0

[1] 1 0

> seq(n)  # ditto

[1] 1 0

> seq_len(n)  # better!

integer(0)

> for (i in seq_len(n)) print(i)
> for (i in seq(n)) print(i)

[1] 1
[1] 0
```

4.4 Conditional Code

It's extremely common to need code to do different things depending upon the number given to it. Let's write a short function to find the absolute value of a number.

```
> abs2 = function(x) {
+   if (x < 0) out = -x
+   else out = x

+   out
+ }
> abs2(-4)

[1] 4
```

The `if()` function will only execute the code which follows if the expression in parentheses evaluates to `TRUE` [3]. When the expression is `FALSE` the code

[3] Actually, any non-zero number will act the same way as `TRUE`, but it's safer to only use logicals.

following the `else` statement will be used instead. There is no need to include an `else`, in which case the program will do nothing if the condition in `FALSE`.

Take care not to allow the logical expression following the `if()` to be a vector, or `R` will spit out a warning.

```
> abs2(c(1,-3))  # won't work properly.

Warning:  the condition has length > 1 and only the first element
will be used

[1]  1 -3
```

If you only need to assign a single value (whether a number, logical, or string) based on a condition, you can use `ifelse()`:

```
> ifelse(TRUE, 94, "hello")

[1] 94

> ifelse(FALSE, 94, "hello")

[1] "hello"
```

4.5 `while()` loops

Here is a short function to check whether an integer is prime.

```
> isPrime = function(n) {
+   i = 2
+   if (n < 2) return(FALSE)

+   while (i < sqrt(n)) {
+     if (n %% i == 0) return(FALSE)
+     i = i+1
+   }
+   return(TRUE)
+ }
> isPrime(10)
```

```
[1] FALSE

> isPrime(37)

[1] TRUE
```

This illustrates several points. First, we don't need to wait until reaching the end of a function to return a value; we can use the **`return`** keyword instead.

The other feature is the **`while()`** loop. This will keep running until the expression in the parenthesis becomes false.

4.6 Writing for Speed

Loops are slow in `R`! It is usually much better to try to 'vectorize' a function rather than calling it lots of times with a loop. Of course, for squaring this is already built in to `R`.

```
> system.time(for (i in 1:1e6) i^2)

   user  system elapsed 
  0.177   0.012   0.196 

> system.time(seq_len(1e6)^2)

   user  system elapsed 
  0.025   0.001   0.026 
```

We can write a second function to do matrix-vector multiplication (see Exercise 4.4), but this time replacing the inner loop by a vectorized function to take dot products.

```
> mult2 = function(A, b) {
+   n1 = nrow(A)
+   n2 = ncol(A)

+   out = numeric(n1)

+   for (i in 1:n1) {
+     out[i] = sum(A[i,]*b)
+   }
+   out
+ }
```

Now suppose we create a large matrix, and look at the difference in timing.

```
> A = matrix(rnorm(1e6), 1e3, 1e3)
> b = rnorm(1e3)
> system.time(mult(A,b))

   user  system elapsed
  1.498   0.039   1.572

> system.time(mult2(A,b))

   user  system elapsed
  0.013   0.005   0.018

> system.time(colSums(t(A)*b))  # can you see why this works?

   user  system elapsed
  0.012   0.000   0.012

> system.time(A %*% b)

   user  system elapsed
  0.004   0.000   0.004
```

The difference is dramatic. The moral of this is that it's usually better to use a built-in function, and almost always better to vectorize. The reason `%*%` is so fast is that `R` calls underlying FORTRAN routines which have been optimized over decades.

4.7 Recursion

Functions can recurse, which means they call themselves; here is a function which calculates the entry F_n in the Fibonacci sequence with $F_0 = F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$:

```
> fib = function(n) {
+   if (n < 2) return(1)
+   else return(fib(n-1) + fib(n-2))
+ }
> fib(10)

[1] 89
```

Recursion can be very slow though, so try to avoid it if possible. You can also use `Recall()` instead of writing the function's name in order to recurse.

Exercise 4.5. The number of moves required to complete a *Towers of Hanoi* puzzle with k pieces is $H_k = 2H_{k-1} + 1$ if $k > 1$, with $H_1 = 1$. Write a recursive function to evaluate H_k.

Exercise 4.6. Write a function to calculate a Fibonacci sequence using a loop instead of a recursion. Compare the execution time to `fib()` (above) for calculating the 30th term in the sequence using `system.time()`.

4.8 Scope

When a function is called, the code inside it is run in a separate **environment** to the code you run directly at the command line. This means it's possible for a variable inside a function to have the same name as something at the command line without causing any problems:

```
> x <- 3
> f = function(y) {
+   x <- 5
+   x + y
+ }
> f(4)

[1] 9

> x      # still the same value as before

[1] 3
```

However, if a function fails to find a variable withing its own environment, then it will look to the **parent environment** for such a value: this is either the function which called the current function, or the **global environment** (i.e. the one you use at the command line).

```
> x <- 3
> g = function(y) {
+   x + y
+ }
> g(4)

[1] 7
```

Whilst this sort of behaviour can sometimes seem helpful, it is much better to avoid writing confusing code like this. You are strongly recommended to write functions which only require the information in their own arguments to run.

This is the same principle used by the functions `with()` and `subset()`: they create an environment for the data frame (or list) you give as their first argument; if any names supplied don't match columns within the data frame, `R` searches in the global environment:

```
> conv = 1.609
> with(hills, mean(dist/conv))  # where do conv and dist come from?

[1] 4.679
```

Exercise 4.7. What will happen if I create an object called `dist` before running the commands above?

5 Graphics

Graphics and 'data visualization' are an integral part of statistics, and R makes it easy to produce common plots quickly, as well as giving a powerful interface for more esoteric output. The basic command is the **generic** function plot(). This will try to do the most sensible thing for the kind of data you provide.

We have already seen that plot(x) and plot(x,y) will produce different plots depending upon the class of the inputs. This is very partially summarized in the following table:

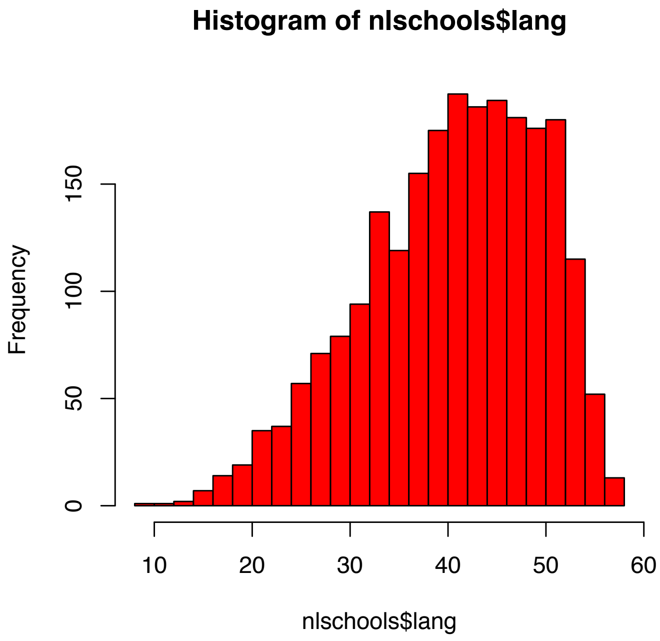
	y		
x	(missing)	numeric	factor
numeric	series plot	scatter plot	spine plot
factor	bar chart	box plots	spine plot

In fact there are many more plotting methods, most of which you will rarely use.

5.1 One dimensional plots

For graphical summaries of one dimensional data we have already seen boxplots and (in the practical) a time series for random walks. Among the most useful is the histogram:

```
> hist(nlschools$lang, breaks=25, col=2)
```

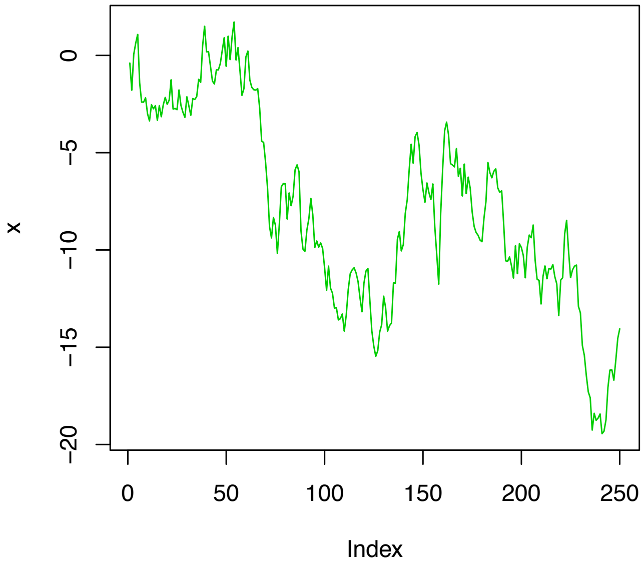


Note that the optional argument breaks chooses (approximately) how many bins the histogram should have, and col alters the colour of the bars. Of course, all plots should have properly labelled axes and a title, which can be easily added.

```
> hist(nlschools$lang, breaks=25, col=2, xlab="Score",
+      main="Language test scores of Dutch 8th grade pupils")
```

Even the simple plot command for a single numeric vector comes with a large range of options.

```
> x = cumsum(rnorm(250))
> plot(x, type="l", col=3)
```

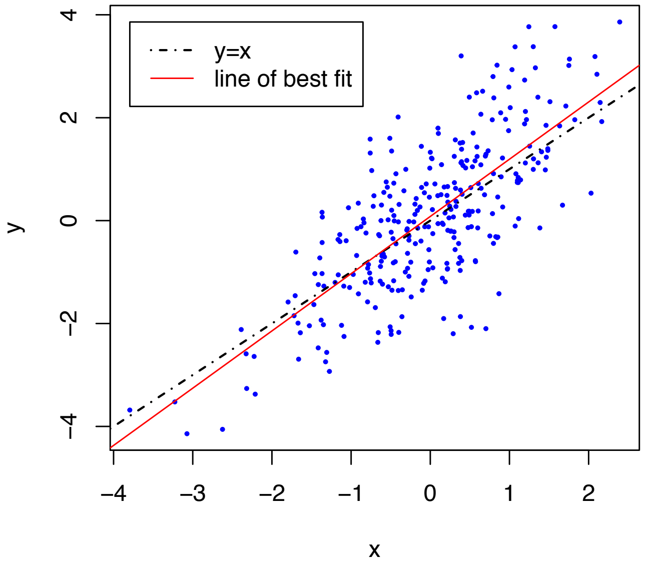


Try this with `type="b"` or `type="h"` and see what happens. You can only find out about a few of the graphics options with the documentation for `plot()`. Try looking at `?par` to find the real detail.

5.2 Adding to Plots

Consider the following simple scatter plot, augmented with the line $y = x$.

```
> x = rnorm(300)
> y = x + rnorm(300)
> plot(x,y, pch=20, col=4, cex=0.5)
> abline(a=0, b=1, lty=4, lwd=1.5)
```



What does `cex=0.5` do?

The function `abline()` plots lines of the form $y = a + bx$ where a and b are specified. As you can see, the width and appearance of the line is adjusted with the options `lty` (line type), `lwd` (line width) and `col` (colour).

You can also use `abline()` in conjunction with output from a simple linear model. Adding the following gives the red line:

```
> abline(lm(y ~ x), col=2)
```

5.3 Legends

The `legend()` command can be used to provide additional information about your plots. The basic syntax is

```
> legend(x=-4, y=4, legend=c("y=x","line of best fit"),
+                          lty=c(4,1), lwd=c(1.5,1), col=1:2)
```

where (x, y) is the top-left hand corner of the box, `legend` is a character vector of annotations, and the other options are used to describe what to display. Many other options are available via the help file.

5.4 Formulae and Boxplots

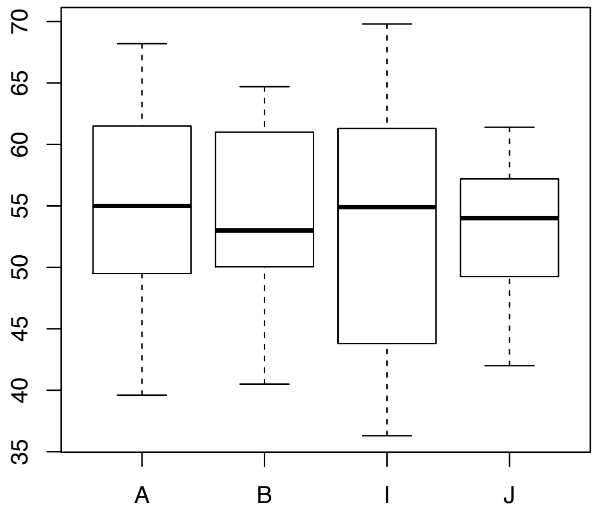
R contains a special object type called a **formula** which can be used to represent statistical models compactly. Try typing some algebraic expression at the console separated by the tilde operator ~ (this is on the # key on a UK keyboard, and left of 1 on a US keyboard).

```
> x ~ a + b*c


x ~ a + b * c
```

The formula object can be used to express relationships between variables under the convention that the left-hand side 'is modelled by' the right-hand side.

This can be used, for example, when producing plots:

```
> data(genotype)
> head(genotype)


  Litter Mother   Wt
1      A      A 61.5
2      A      A 68.2
3      A      A 64.0
4      A      A 65.0
5      A      A 59.7
6      A      B 55.0


> boxplot(Wt ~ Litter, data=genotype)
```



The function interprets the formula as requiring that the left-hand side be summarized in a way which is broken down by the right. Note that `Wt` and `Litter` are contained within `genotype`, and are not recognized at the console[4], but the argument `data=genotype` ensures that the `boxplot()` function knows where to look for `genotype$Wt`.

5.5 Lattice Graphics

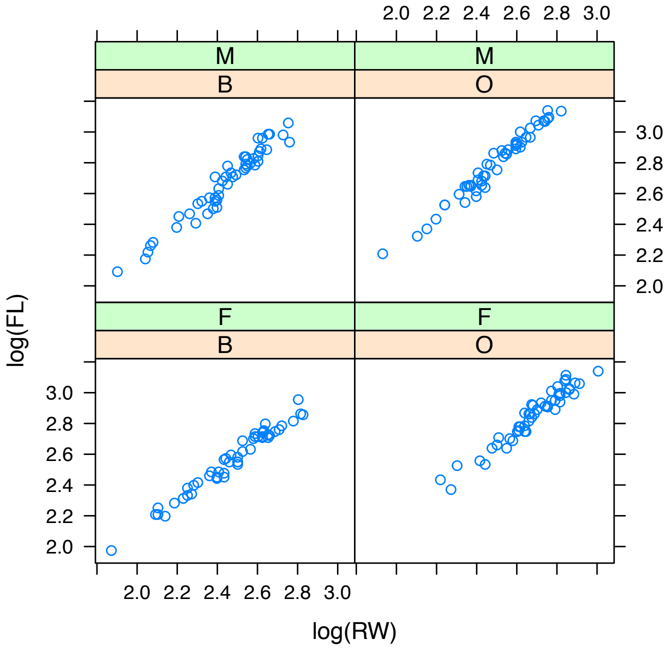
The plots above are all found in the `base` package of `R`, which is to say that they are all preloaded functions. A very popular and powerful extension to `R`'s graphics capabilities is made using the package `lattice`. The range of plots which can be produced even using `lattice`'s default methods is staggering, and we will show only a few small examples here.

The basic command is `xyplot()`, whose first argument is usually a formula.

```
> library(lattice)
> head(crabs)

  sp sex index   FL  RW   CL   CW  BD
```

[4] That is, they are not in the global environment

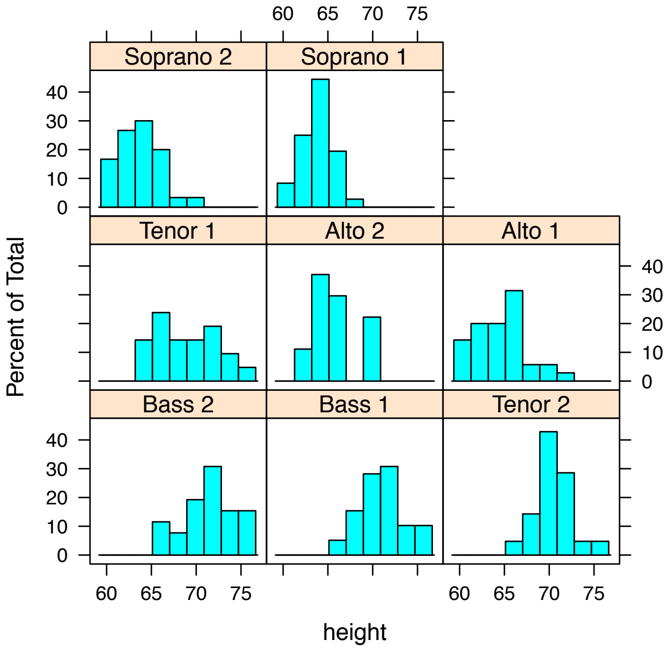
```
1  B   M     1  8.1 6.7 16.1 19.0 7.0
2  B   M     2  8.8 7.7 18.1 20.8 7.4
3  B   M     3  9.2 7.8 19.0 22.4 7.7
4  B   M     4  9.6 7.9 20.1 23.1 8.2
5  B   M     5  9.8 8.0 20.3 23.0 8.2
6  B   M     6 10.8 9.0 23.0 26.5 9.8

> form <- log(FL) ~ log(RW) | sp*sex
> form

log(FL) ~ log(RW) | sp * sex

> xyplot(form, data=crabs)
```



The formula `form` in this case has three parts. The left-hand side `log(FL)` is to be plotted against the right `log(FL)`; since both these variables are continuous, we will obtain a scatter plot. The conditioning bar ‘`|`’ indicates that we wish the information to be broken down by the third term, `sp*sex` (i.e. by species and by sex). Hence `lattice` produces four separate scatter plots, each with the same axes.

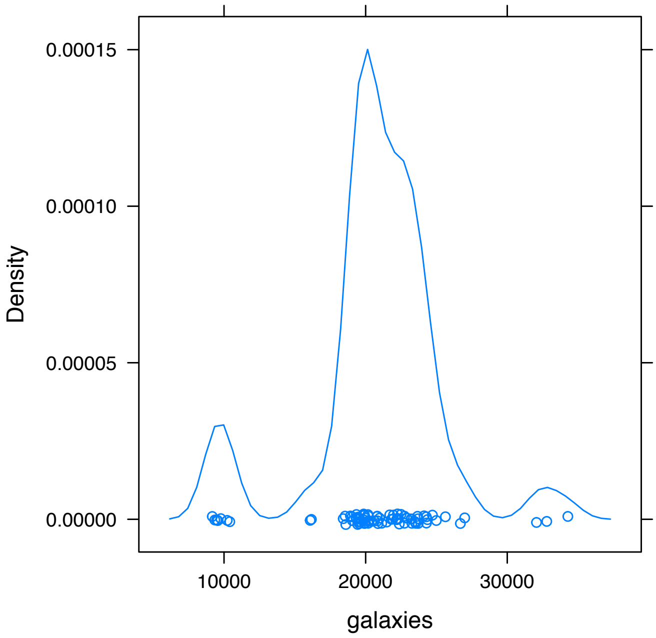
The most common use of the `lattice` package is to produce these **trellis plots** for representing multivariate data. A few more examples you might find useful:

```
> histogram(~ height | voice.part, data=singer)
```



Notice that the command is `histogram()`, not `hist()`, and the plotting options are different.

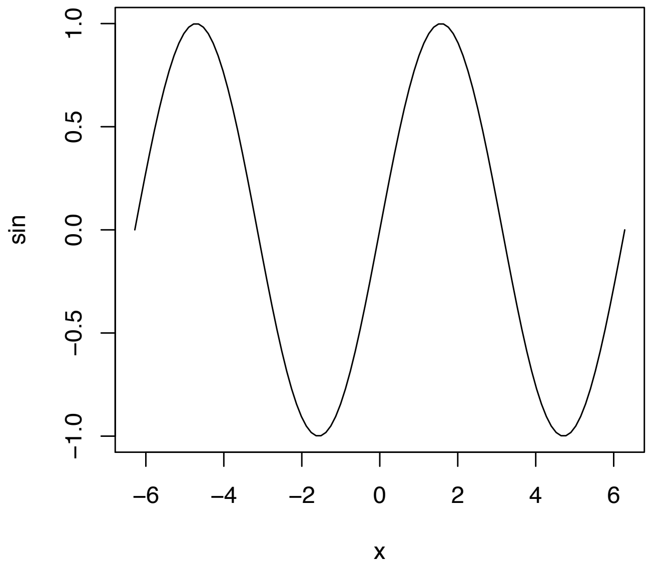
```
> library(MASS)
> densityplot(galaxies)
```



5.6 Function Plots

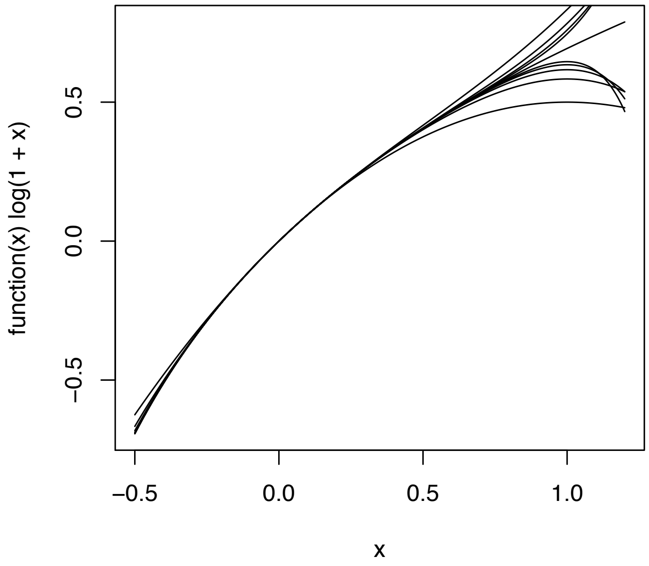
When applied to a vectorized function of one argument, `plot()` will produce a graph in the specified range.

```
> plot(sin, -2*pi, 2*pi)
```



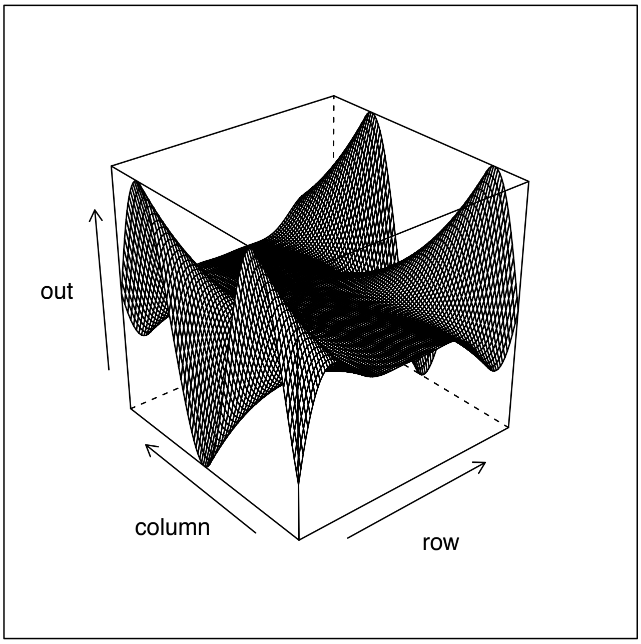
For non-vectorized functions, this has to be done manually:

```
> func = function(x, n=10) {
+   idx = 1:n
+   return(sum((-1)^(idx+1)*x^idx/idx))
+ }
>
> plot(function(x) log(1+x), -0.5, 1.2)
>
> for (i in 2:10) {
+   x <- seq(from=-0.5, to=1.2, length=1000)
+   y <- sapply(x, func, n=i)
+   points(x, y, type="l")
+ }
```



Using the `lattice` package you can also produce surface plots for functions with two arguments using wireframe plots.

```
> func = function(x,y) (x^3-x)*sin(x+y)
> xs = ys = seq(-5, 5, length.out=100)
> out = outer(xs, ys, func)
> wireframe(out)
```



See also `contourplot()` and `levelplot()`.

5.7 Customized Plots

To draw a plot from scratch, use the `plot.new()` command:

```
> set.seed(1328)  # to get the same values as me
> x = rnorm(100)  # generate data
>
> plot.new()
> plot.window(xlim=c(-3,3), ylim=c(-0.1,0.5))
> axis(side=1, pos=-0.1)
> hist(x, breaks=15, add=TRUE, freq=FALSE, col=2)
> plot(dnorm, -3, 3, add=TRUE)
> points(x, rep(-0.05,100), pch="|")
> title(main="Normal random variables")
```

`plot.window()` is used to control the range of the plot, the `axis()` function draws on the axes, and `title()` is used to annotate with text. Try each of the above commands in turn to see what they do.

5.8 Exporting Plots

You will likely need to use R plots in LaTeX documents for your practicals and projects. If you are using a GUI such as the default R interface on OS X or Windows, then select the window and go to File > Save As. I recommend saving plots in PDF format, as this makes it easiest to integrate with a LaTeX document. Other interfaces such as RStudio make it similarly easy to create plots.

You can also save plots from the command line. The way to do this is to tell R to send your plot commands to a file, instead of to the screen. This means you won't be able to see your plot whilst you produce it (but presumably you'll have already checked what it looks like!) Type pdf("yourfilename.pdf") to start, then run your chosen plot commands. Then finish with dev.off() to go back to the default state. For example:

```
> pdf("plotfile.pdf")
> plot(hills)
> dev.off()
```

6 The apply Family of Functions

Much coding involves the repeated application of the same function to several different pieces of data in a vector or list. For this reason, R has a series of functions for performing such tasks, which results in much simpler and easier to understand code.

6.1 apply()

If we want to perform a function on every row or column in a matrix (or on an array, see next section), we can use the apply() function. The syntax is apply(x, d, f), which will apply the function f to each of the dth dimension of x. If d=1 this corresponds to rows, and d=2 to the columns of a matrix.

```
> A = cbind(1:10, (1:10)^2, (1:10)^3)
> apply(A, 2, sum)

[1]   55  385 3025
```

It will also work for 'matrix-like' objects, such as data frames (although see also sapply() below).

```
> library(MASS)
> apply(hills, 2, mean)

    dist    climb     time
   7.529 1815.314   57.876

> apply(hills, 2, sd)

    dist    climb     time
   5.524 1619.151   50.041
```

Exercise 6.1. Using apply(), write a function which, given an $(I \times J)$-matrix $X = (x_{ij})$ computes the magnitude of each row, that is

$$\sqrt{x_{i1}^2 + x_{i2}^2 + \cdots + x_{iJ}^2}, \qquad \text{for each } i = 1, \dots, I$$

and returns the results as a vector.

Exercise 6.2. Write a function which, given an $(I \times J)$-matrix X, returns a vector of length I with entries

$$y_i = \frac{\bar{x}_{i+}}{s_i}$$

where $\bar{x}_{i+}$ and s_i are respectively the sample mean and sample standard deviation of entries in the ith row of X. [The mean divided by the standard deviation is sometimes called the *coefficient of variation*.]

What happens if you use `apply()` with a function like `range()`, which returns more than one value?

Exercise 6.3. Take a look at the data set `EuStockMarkets` (this is in the `datasets` package, which should be already loaded). Find the mean absolute change in returns from one day to the next for each stock (that is, the average of $|x_{i+1} - x_i|$ over all days i). [Hint: recall the `diff()` function.]

Bonus:* Think of a more sensible measure of the volatility than this and implement it [hint: one that doesn't depend upon the scale].

Note that `apply()` does **not** run substantially faster than writing a loop to do the same thing, it is simply easier to code up and to read.

For the particular task of sums or means of rows or columns in a matrix, `R` contains special functions `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`. These are all much faster than the equivalent `apply()` commands.

```
> # 2000 x 2000 random matrix
> x = matrix(rnorm(4e6), 2000, 2000)
> system.time(apply(x,1,sum))

   user  system elapsed
  0.121   0.004   0.127


> system.time(rowSums(x))

   user  system elapsed
   0.01    0.00    0.01
```

Exercise 6.4. Write a function to renormalize the columns of a matrix so that they sum to 1.

Exercise 6.5. Write a function to perform the same task as in Exercise 6.1, but this time using `rowSums()`. Compare the speed of these two methods using `system.time()`.

6.2 `sapply()` and `lapply()`

If we want to apply a function to every entry in a list or vector, we can use `lapply()`. The syntax is just `lapply(x, f)` for a vector or list `x` and a function `f`. It returns a list of the same length as `x` containing the results.

The following example uses `lapply()` twice, first on a vector, and then on the resulting list.

```
> mu = c(-2,-1,0,1,2)
> out = lapply(mu, function(x) rnorm(100, mean=x))
> lapply(out, mean)


[[1]]
[1] -2.169

[[2]]
[1] -1.141

[[3]]
[1] 0.1753

[[4]]
[1] 1.105

[[5]]
[1] 2.008
```

Note that in the previous example, we didn't want the final answer as a list, so we might use the `unlist()` command to turn the results into a vector. The `sapply()` function does this automatically when appropriate, but is otherwise the same as `lapply()`.

```
> sapply(out, mean)


[1] -2.1688 -1.1414  0.1753  1.1050  2.0080
```

If the function being applied returns something more complicated than a single number, you should use `lapply()` instead.

6.3 `replicate()`

Sometimes we wish to repeat exactly the same operation multiple times, without having a different input (this is typically used in conjunction with the generation of random numbers).

```
> out = replicate(20, rnorm(100), simplify=FALSE)
```

Now `out` is a list of 20 independent data sets, each consisting of 100 standard normal random variables.

Exercise 6.6. Suppose we wish to investigate the distribution of the maximum of 10 Poisson random variables with parameter $\lambda = 5$. Generate 1000 independent data sets consisting of such Poisson random variables (see the command `rpois()`), find the maximum of each, and plot as a histogram.

6.4 `tapply()`

In the same vein as the previous examples, we may wish to evaluate a function on data in a vector or list which are 'grouped' according to the levels of some other factor. For example,

```
> library(MASS)
> head(genotype)


  Litter Mother   Wt
1      A      A 61.5
2      A      A 68.2
3      A      A 64.0
4      A      A 65.0
5      A      A 59.7
6      A      B 55.0


> with(genotype, tapply(Wt, Mother, mean))


    A     B     I     J
55.40 58.70 53.36 48.68
```

returns a vector of means. If the function provided gives more than a single value, then `tapply()` will adapt accordingly:

```
> with(genotype, tapply(Wt, Mother, summary))


$A
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   36.3    49.0    58.2    55.4    62.1    68.2
```

```
$B
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   42.0    55.2    59.8    58.7    63.5    69.8

$I
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   39.7    48.9    54.2    53.4    57.5    61.8

$J
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   39.6    42.9    50.0    48.7    53.5    61.0
```

It is also possible to provide more than one grouping in the form of a list or data frame, in which case the data are broken down by both:

```
> tapply(genotype$Wt, genotype[,1:2], mean)


      Mother
Litter     A     B     I     J
     A 63.68 52.40 54.12 48.96
     B 52.33 60.64 53.92 45.90
     I 47.10 64.37 51.60 49.43
     J 54.35 56.10 54.53 49.06
```

Exercise 6.7. Find the heaviest rats born to each mother in the `genotype()` data.

6.5 mapply()

Sometimes it may be useful to apply a function of several arguments repeatedly, where more than one argument can change.

```
> mapply(seq, from=c(1,4,-3), to=c(2,9,0), by=0.5)


[[1]]
[1] 1.0 1.5 2.0

[[2]]
 [1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0

[[3]]
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0
```

Shorter arguments are recycled as necessary.

7 Arrays and Tables

The data sets we have seen so far were best expressed as data frames, with a single row corresponding to each observation. For discrete data sets, this is not always the most useful or efficient way to represent the information, and it may be more useful to use a **contingency table**.

Consider the data set `occupationalStatus` in the `datasets` package. This contains information on the occupational status of 3,498 father and son pairs, grouped from 1 (highest status) to 8 (lowest). One way to present these data would be a data frame with 2 columns and 3,498 rows; however, since there are only 64 possible distinct entries in the data frame, we can represent the data rather more compactly as a matrix:

```
> occupationalStatus

      destination
origin   1   2   3   4   5   6   7   8
     1  50  19  26   8   7  11   6   2
     2  16  40  34  18  11  20   8   3
     3  12  35  65  66  35  88  23  21
     4  11  20  58 110  40 183  64  32
     5   2   8  12  23  25  46  28  12
     6  12  28 102 162  90 554 230 177
     7   0   6  19  40  21 158 143  71
     8   0   3  14  32  15 126  91 106
```

Here each row represents the occupational status of the father, and each column that of the son.

Exercise 7.1. (a) What is the probability of a son having the same occupational status as his father? *[Hint: investigate what* `diag(x)` *does if* `x` *is a matrix.]*

(b) Renormalize the data so that each row sums to 1. In the new data set the ith row represents the conditional distribution of a son's occupational status given that his father has occupational status i.

(c) What is the probability that a son has occupational status between 1 and 3, given that his father has status 1?

What if the father has occupational status 8?

(d) Calculate the vector **y** where y_i is the probability that the son has occupational status between 1 and 3, *given that* his father has occupational status i.

7.1 Higher Dimensions

Of course, if we have a data set consisting of more than two pieces of categorical information about each subject, then a matrix is not sufficient. The generalization of matrices to higher dimensions is the **array**. Arrays are defined much like matrices, with a call to the `array()` command. Here is a $2 \times 3 \times 3$ array:

```
> arr = array(1:18, dim=c(2,3,3))
> arr


, , 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
```

Each 2-dimensional **slice** defined by the last co-ordinate of the array is shown as a 2×3 matrix. Note that we no longer specify the number of rows and columns separately, but use a single vector `dim` whose length is the number of dimensions. You can recover this vector with the `dim()` function.

```
> dim(arr)


[1] 2 3 3
```

Note that a 2-dimensional array is identical to a matrix. Arrays can be subsetted and modified in exactly the same way as a matrix, only using the appropriate number of co-ordinates:

```
> arr[1,2,3]


[1] 15


> arr[,2,]


     [,1] [,2] [,3]
[1,]    3    9   15
[2,]    4   10   16


> arr[1,1,] = c(0,-1,-2)    # change some values
> arr[,,1,drop=FALSE]


, , 1

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

Notice that dimensions are dropped by default.

I have placed some data on housing in a 4-way array on the course website. To load the data navigate to the appropriate directory and run the command

```
> house = dget("housing.dat")
```

The whole table is quite large and difficult to understand, but if we focus on some of the 2-way marginal distributions it could help.

```
> margin.table(house, 1:2)


        Infl
Sat       Low Medium High
  Low     282    206   79
  Medium 170    189   87
  High    175    264  229


> margin.table(house, c(1,4))
```

```
        Cont
Sat       Low High
  Low     262  305
  Medium 178  268
  High   273  395
```

This is equivalent to

```
> apply(house, 1:2, sum)

        Infl
Sat       Low Medium High
  Low     282    206   79
  Medium 170    189   87
  High   175    264  229
```

Exercise 7.2. Create a new data set in which the levels `Atrium` and `Terrace` are combined in to a single level called `Other`.

7.2 Tables

Given a large data set of discrete items, a useful summary is just to count the number of items in each category; this can be done using the `table()` command.

```
> library(MASS)
> head(cabbages)

  Cult Date HeadWt VitC
1  c39  d16    2.5   51
2  c39  d16    2.2   55
3  c39  d16    3.1   45
4  c39  d16    4.3   42
5  c39  d16    2.5   53
6  c39  d16    4.3   50

> table(cabbages$Date)


d16 d20 d21
 20  20  20
```

```
> table(cabbages[,1:2])


     Date
Cult  d16 d20 d21
  c39  10  10  10
  c52  10  10  10
```

For univariate data this gives a vector of counts, and for multi-way data a contingency table in the form of an array. Note that the vector is named, which makes it look slightly more complex than it actually is, but it behaves just like a normal vector.

```
> tab = table(cabbages$Date)
> names(tab)


[1] "d16" "d20" "d21"
```

7.3 Binning Data

If we have continuous data it may be useful to summarize them using a **discretization**; that is, we group together some of the data into intervals.

```
> head(Nile)


[1] 1120 1160  963 1210 1160 1160


> bins = c(0,seq(from=700,to=1300,by=100),Inf)
> bins


[1]    0  700  800  900 1000 1100 1200 1300  Inf


> disNile = cut(Nile, bins)
> head(disNile)


[1] (1.1e+03,1.2e+03] (1.1e+03,1.2e+03] (900,1e+03]       (1.2e+03,1.3e+03]
[5] (1.1e+03,1.2e+03] (1.1e+03,1.2e+03]
8 Levels: (0,700] (700,800] (800,900] (900,1e+03] ... (1.3e+03,Inf]


> table(disNile)
```

```
disNile
          (0,700]         (700,800]         (800,900]       (900,1e+03]
                6                20                25                19
  (1e+03,1.1e+03] (1.1e+03,1.2e+03] (1.2e+03,1.3e+03]     (1.3e+03,Inf]
               12                11                 6                 1
```

The `cut()` command turns the data into a factor defined by the edges of the 'bins' you provide. The default labels are rather unwieldy, but this can be changed:

```
> disNile = cut(Nile, bins, labels=bins[-9])
> table(disNile)


disNile
   0  700  800  900 1000 1100 1200 1300
   6   20   25   19   12   11    6    1
```

8 Strings

A string is a piece of text enclosed in double quotes ("), such as `"Hello!"`. It may include spaces, numbers and punctuation. You can send a string to the screen using the `cat()` function.

```
> cat("Hello")

Hello
```

Note that this is quite different from the `print()` function, which is discussed in Section 9. So if we write a function and wish to provide information to the person using it, this is useful:

```
> myFunc = function(x) {
+   cat("Squaring the number\n")
+   x^2
+ }
>
> myFunc(3)

Squaring the number
[1] 9
```

Note the `\n` at the end of our string: this tells the command to start a new line, otherwise it looks rather ugly (try it). The symbol \ is an **escape character**, which allows us to perform special tasks with it. For example, we can't use the double quotes symbol `"`, because `R` will interpret this to mean that we have finished our string. Instead we write `\"` to represent a double quote, and similarly we have to use `\\` to represent `\`.

For example:

```
> cat("She said \"Hello\" to the man.\n")

She said "Hello" to the man.

> cat("This is a backslash: \\ - how does it make you feel?\n")

This is a backslash: \ - how does it make you feel?
```

Note that the function `cat()` also works with numbers and can have several arguments:

```
> cat(57, "clouds", "\n")

57 clouds
```

By default `cat()` separates the arguments with a space, but you can change this:

```
> cat(57, "clouds", "\n", sep="")

57clouds

> cat(57, "clouds", "\n", sep=".")

57.clouds.
```

Exercise 8.1. Write a function which takes a positive integer `n` and write the numbers $1, 2, \ldots, n$ on the screen, separated by a comma and a space and finishing with a full stop. Like this:

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

Exercise 8.2. Write a function `withBox()` with a single string argument `str`, which prints `str` inside a box of asterisks like this:

```
> withBox("Hello")

*********
* Hello *
*********
```

The function `nchar()` is useful for giving the length of a string.

8.1 The `paste()` function

It's often useful to be able to stick strings together, for which purpose we have the function `paste()`.

```
> paste("Hello", "there")

[1] "Hello there"
```

If given a vector `paste()` uses vector recycling, which can be rather useful:

```
> paste("Plan", LETTERS[1:5])

[1] "Plan A" "Plan B" "Plan C" "Plan D" "Plan E"
```

Like `cat()` we also have a `sep=` option and can work with numbers:

```
> paste("x", 1:10, sep="")

 [1] "x1"  "x2"  "x3"  "x4"  "x5"  "x6"  "x7"  "x8"  "x9"  "x10"
```

You can also make `paste()` concatenate all its output into a single string with the `collapse=` optional argument:

```
> paste(LETTERS[1:10])

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

> paste(LETTERS[1:10], collapse=" ")

[1] "A B C D E F G H I J"
```

Exercise 8.3. Write a function which, given a positive integer `n`, produces a string of the form `1 < 2 < ...  < n`. So, for example,

```
> f(5)

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

[Bonus: do it with a function using only one line of code.]

Exercise 8.4. Write a function which, given a vector `x` of positive integers, returns a list of the same length as `x`, and the ith entry of the list is a character vector of length `x[i]`. The entries in the 1st element of the list should be `"a1"`, `"a2"`, and so on, and in the 2nd should be `"b1"`, `"b2"` for $i = 2$, etc.

For example:

```
> listfunc(c(1,4,2))

[[1]]
[1] "a1"

[[2]]
[1] "b1" "b2" "b3" "b4"

[[3]]
[1] "c1" "c2"
```

[You can do this in one line of code using `mapply()`!]

8.2 Other Manipulation

Other potentially useful functions are:

```
> nchar("How long is this string?")

[1] 24

> strsplit("separate words are fun", split=" ")

[[1]]
[1] "separate" "words"    "are"      "fun"

> substr("I don't want all of this string", 14, 24)

[1] "all of this"
```

9 Classes and Methods

You may by now have noticed that certain functions, such as `plot()` or `summary()`, appear to behave very differently when applied to different types of object.

```
> x <- rnorm(100)
> y <- x + rnorm(100)
> mod1 <- lm(y ~ x)
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.8400 -0.6890  0.0079  0.0766  0.8910  2.5900

> summary(mod1)



Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-2.2756 -0.7760  0.0506  0.7570  2.1564

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.00477    0.09791    0.05     0.96
x            1.22301    0.09248   13.23   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.977 on 98 degrees of freedom
Multiple R-squared:  0.641,Adjusted R-squared:  0.637
F-statistic:  175 on 1 and 98 DF,  p-value: <2e-16
```

You might be forgiven for wondering whether these functions have endless lines of code telling them what to do in dozens of different circumstances. In fact these are **generic** functions.

```
> summary


function (object, ...)
```

```
UseMethod("summary")
<bytecode: 0x7fbdc5b5bce0>
<environment: namespace:base>
```

When a generic function is called, it looks for a suitable **method** to use on the arguments passed to it. You can look at the different methods available using `methods()`

```
> methods(summary)
```

Which method is chosen depends upon the object's **class**; in other words, what type of object it is.

```
> class(mod1)

[1] "lm"
```

When you call summary with an object of class `lm`, it looks for a method with the name `summary.lm()`. So if you call

```
> summary.lm(mod1)
```

the result is the same as above.

9.1 Classes and Attributes

The class of an object is one of its **attributes**. Attributes are a named list of bits of information about an object that are stored alongside it. The only difference between the list of attributes and any other list is that each entry must have a name, and the name must be unique. You can see the attributes of an object using the `attributes()` command:

```
> attributes(hills)

$names
[1] "dist"  "climb" "time"

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

```
$row.names
 [1] "Greenmantle"      "Carnethy"         "Craig Dunain"
 [4] "Ben Rha"          "Ben Lomond"       "Goatfell"
 [7] "Bens of Jura"     "Cairnpapple"      "Scolty"
[10] "Traprain"         "Lairig Ghru"      "Dollar"
[13] "Lomonds"          "Cairn Table"      "Eildon Two"
[16] "Cairngorm"        "Seven Hills"      "Knock Hill"
[19] "Black Hill"       "Creag Beag"       "Kildcon Hill"
[22] "Meall Ant-Suidhe" "Half Ben Nevis"   "Cow Hill"
[25] "N Berwick Law"    "Creag Dubh"       "Burnswark"
[28] "Largo Law"        "Criffel"          "Acmony"
[31] "Ben Nevis"        "Knockfarrel"      "Two Breweries"
[34] "Cockleroi"        "Moffat Chase"

$type
[1] "races"
```

You can also access and set specific attributes by name using `attr()`:

```
> attr(hills, "class")

[1] "data.frame"

> attr(hills, "row.names")[4] = "Big Hill"
> row.names(hills)

 [1] "Greenmantle"      "Carnethy"         "Craig Dunain"
 [4] "Big Hill"         "Ben Lomond"       "Goatfell"
 [7] "Bens of Jura"     "Cairnpapple"      "Scolty"
[10] "Traprain"         "Lairig Ghru"      "Dollar"
[13] "Lomonds"          "Cairn Table"      "Eildon Two"
[16] "Cairngorm"        "Seven Hills"      "Knock Hill"
[19] "Black Hill"       "Creag Beag"       "Kildcon Hill"
[22] "Meall Ant-Suidhe" "Half Ben Nevis"   "Cow Hill"
[25] "N Berwick Law"    "Creag Dubh"       "Burnswark"
[28] "Largo Law"        "Criffel"          "Acmony"
[31] "Ben Nevis"        "Knockfarrel"      "Two Breweries"
[34] "Cockleroi"        "Moffat Chase"
```

This is in addition to the special functions we have already seen: `dimnames()`, `dim()`, `names()`, `rownames()`, etc. You can also create entirely new attributes:

```
> attr(hills, "quality") = "Great!"
> attributes(hills)[4]


$type
[1] "races"
```

9.2 Setting Classes and Methods

Class is an attribute, so you can set it just as easily as any other:

```
> x <- list(temp=19.5, wind=12, wind.dir="SSW", rain=20, summ="showers")
> class(x) <- "weather"
> attr(x, "class")


[1] "weather"
```

When you type an object's name at the console, `R` searches for a `print()` method in order to decide how to display it to you. Since there is no such method for objects of class `weather`, it just uses `print.default()`; we can create a new method by just naming a function `print.weather()`.

```
> print.weather = function(object) {
+   cat("Weather report\n")
+   cat("Temperature ", object$temp, "C\n", sep="")
+   cat("Wind ", object$wind, " kts, from ", object$wind.dir, "\n", sep="")
+   cat("Rain ", object$rain, " mm: ", object$summ, "\n", sep="")
+   object
+ }
```

Now:

```
> x


Weather report
Temperature 19.5C
Wind 12 kts, from SSW
Rain 20 mm: showers
```

Similarly we can write functions `summary.weather()`, `plot.weather()`, or whatever we think is appropriate.

Exercise 9.1. Write a new summary method for factors, which prints `"Factor Summary"` to the screen, before giving a table of counts.

10 Debugging

When constructing a complicated program containing several functions, it is inevitable that we make mistakes. Sometimes these errors will result in code which does not run at all, resulting in an error message from R.

```
> "a" + 1

Error:  non-numeric argument to binary operator
```

On other occasions, code may run but fail to give the correct answer. This is more dangerous, since we may not realise, but it also makes it harder to find the source of the problem. Getting warning messages from R is a sign that something is not working as you intended: **do not ignore these**!

```
> 1:3 + 1:2

Warning:  longer object length is not a multiple of shorter object
length

[1] 2 4 4
```

In any of these cases, you will need to discover what is wrong, and figure out how to fix it.

10.1 Prevention

There are few steps you can take in your programming style to reduce the likelihood of making mistakes in the first place. These are:

- **Make your code modular, and don't repeat yourself.** Try to construct your code in the *spirit* of R: you should use self contained functions to perform specific tasks, and call those functions when necessary. If your program needs to perform the same task more than once, then have it call the same code each time: this way it's easier to detect and cure any mistakes.

- **Write simple code when possible.** R has lots of useful built in features which allow complicated code to be written very succinctly and clearly, such as by using the `apply()` family. Try to make use of such features when possible. In the examination for this course, **you will lose marks** for failing to use a function like `lapply()` in a straightforward situation.

- **Comment your code.** You'll have noticed by now that anything followed by a # symbol in R is ignored by the computer. This is to allow you to put comments in the code, so that someone reading it later can understand what it is intended to do (often this is just you sometime later, after you've forgotten what you were doing when you first wrote the code). It's also helpful to give the variables and functions you use sensible and descriptive names.

  ```
  > ## generate random numbers
  > set.seed(241)
  > rndSteps <- rnorm(N)
  > ## generate cumulative sum for random walk
  > rndWalk <- cumsum(rndSteps)
  ```

  In an exam context, comments are a useful way to indicate to the examiner what you are trying to do.

- **Think about special cases.** The most common errors arise from failing to think about what will happen when (for example) your input is of length 1 or 0; one particularly common example is in the indexing of arrays:

  ```
  > # function to sum over subset of columns of a matrix
  > row_sums <- function(x, cols) {
  +   apply(x[,cols], 1, sum)
  + }
  > x <- matrix(1:9, 3, 3)
  > row_sums(x, 2:3)

  [1] 11 13 15

  > row_sums(x, 2)

  Error:  dim(X) must have a positive length
  ```

  Of course, this is easily prevented by adding drop=TRUE to the subset command.

10.2 Modularity

R is fundamentally a **functional language**, which means that all operations are performed by functions, which are generally self-contained. It's good practice to keep your code **modular**, in the sense of being made up of distinct functions which perform small tasks, and that you can test (and debug) individually.

Most fundamentally, if you have a complex instruction which isn't working, try breaking it down into pieces to see whether each piece is doing what you intend.

```
> if(any(x > 3) && y != 2) {
+     print("Hello")
+ }

Error:  missing value where TRUE/FALSE needed

> any(x > 3)

[1] TRUE

> y != 2

logical(0)

> y

NULL
```

In more complicated code, it is important to write separate functions which are responsible for individual parts of your program. These can then be tested for errors separately, reducing the complexity of the problem. This also makes it much easier to reuse bits of code in other circumstances.

```
> g = function(y) {
+   if (y < 0) warning("Some warning")
+   return(y)
+ }
>
> h = function(z) {
+   stop("Some error message")
+   z
+ }
>
> f = function(x) {
+   # this is a function which calls some other functions
+   out = g(x) + h(x)
+ }
```

It is also easy to trace errors down to the particular function in which they occur, but harder to search within that code for problems. If an error occurs, immediately calling the `traceback()` function will show the series of functions which were called in the run up to the problem (the **call stack**).

```
> f(2)

Error:  Some error message

> traceback()

3: stop("Some error message") at #3
2: h(x) at #3
1: f(2)
```

This shows that the error occurred in the function `h()`. We can fix the code and call the function again.

```
> h = function(z) {
+   z
+ }
> f(-2)

Warning:  Some warning
```

We now obtain a warning. We cannot use `traceback()` with a warning, but we can tell `R` to turn any warnings into errors for the purposes of debugging:

```
Warning:  Some warning

No traceback available
```

```
> options(warn=2)
> f(-2)

Error in g(x) :  (converted from warning) Some warning
```

```
> traceback()

7: doWithOneRestart(return(expr), restart)
6: withOneRestart(expr, restarts[[1L]])
5: withRestarts({
       .Internal(.signalCondition(simpleWarning(msg, call), msg,
           call))
       .Internal(.dfltWarn(msg, call))
   }, muffleWarning = function() NULL)
4: .signalSimpleWarning("Some warning",
       quote(g(x)))
3: warning("Some warning") at #2
2: g(x) at #3
1: f(-2)
```

It is recommended that you add in your own stop() commands to deal with problems which you think are likely to arise (such as bad user input), and to provide an informative error message. This is much better than having the code fail at a lower level in some internal R function, leaving the reason for failure as a potential mystery.

10.3 Error Recovery

An extremely useful feature of R is error recovery, which is built in to the interface. Set it up using:

```
> options(error=recover)
```

Now if any function calls an error, R will pause its execution and show you the call stack. You then have the option to 'look inside' any of those functions, and inspect the objects within them. This allows you to see if they're doing what you expect.

After choosing a function to work with (by typing the relevant number), R brings up a **browser**, which looks like the ordinary command line, but with a slightly different prompt. Typing commands at this prompt allows you to inspect objects within the particular function you selected. Use the escape character Q to exit[5].

To revert to normal working, use:

[5] If you have an object called Q you'll need to use print(Q) to look at it.

```
> options(error=NULL)
```

10.4 Other Functions

The function `debug()` allows you to inspect what a function does line by line: this is particularly useful if your code is not doing what you expect, but does not actually result in an error. Try calling:

```
> debug(g)
> f(2)
```

`R` will bring up the browser as soon as `g()` is called (in this case from within `f()`. The browser will be inside `g()`'s environment, and you can inspect the objects within it on that basis. In addition, pressing 'Enter' without typing any commands will step through one line of `g()`'s code at a time, allowing you to check what is happening as the function progresses. The special command `Q` can, again, be used to break out of this.

To turn off this feature use the function `undebug()`. If you only wish the console to appear this way on one occasion, you can also use `debugonce()` instead of `debug()`.

You can also temporarily edit a function so as to see what it is doing in a more hands-off fashion. Use the function `trace()` with the `edit=TRUE` option.

```
> trace(g, edit=TRUE)
```

This brings up an editor containing a version of your function. You can do whatever you like to this and it will not permanently alter your code; inserting `print()` statements is a good way to use this. You can revert to the original version of your function using `untrace()`.

11 Arithmetic Subtleties*

Computers store most numbers in the form $l \times 2^k$, for some number $1 \leq l < 2$, and an integer $k \in \{-52, -51, \ldots, 52\}$. This can lead to rounding errors, as the following illustrates[6].

```
> 0.3 - 0.1 - 0.2


[1] -2.776e-17
```

This can cause problems when making comparisons, so the command `all.equal()` can be used to avoid this problem.

```
> x = 0.3 - 0.1 - 0.2
> x == 0


[1] FALSE


> all.equal(x, 0)


[1] TRUE
```

If things are different it returns an explanation, so use `isTRUE(all.equal(...))` in a conditional statement:

```
> if (isTRUE(all.equal(x, 0))) print("We've got nothing!")


[1] "We've got nothing!"
```

Very small or large numbers may not respond as you expect in ordinary arithmetic.

```
> 2^-1074


[1] 4.941e-324


> 2^-1075


[1] 0
```

[6] See `http://www.smbc-comics.com/index.php?db=comics&id=2999`

```
> (2^-1074)/1.5

[1] 4.941e-324

> 2^2000

[1] Inf
```

Large numbers may be replaced with `Inf`, which behaves as you might expect infinity to.

```
> Inf/2

[1] Inf

> (-1)*Inf

[1] -Inf

> 0*Inf

[1] NaN

> Inf - Inf

[1] NaN

> exp(-Inf)

[1] 0
```

11.1 Integers and Doubles

`R` does distinguish between integers and ordinary numbers (doubles), but it is not easy to tell the difference, and integers are often coerced into doubles without the user noticing. This can cause problems

```
> i1 = 0L
> i2 = 0
> (i1 == i2)

[1] TRUE

> identical(i1, i2)

[1] FALSE

> i3 = 1e40 + 1
> i3-1e40

[1] 0
```

12 List of the Most Useful Functions in R

```
# libraries
library(), require(), install.packages()

# basic objects
`=`, `<-`, `<<-`, is(), as()
c(), list(), unlist(), matrix(), array(), data.frame(), `~`
cbind(), rbind()
head(), tail(), rev()
# attributes
attributes(), attr()
length(), dim(), ncol(), nrow()
names(), dimnames(), rownames(), colnames()

cat(), print(), print.default(), paste(),
# logic
`!`, `&`, `|`, `&&`, `||`, `==`
any(), all(), ifelse()

apply(), lapply(), sapply(), replicate(), tapply(), mapply()

seq(), seq_along(), seq_len(), `:`
rep(), rep.int()
unique(), duplicated()
match(), which(), `%in%`, which.max(), which.min()
sort(), order(), rank()

factor(), is.factor(), as.factor(),
levels(), nlevels(), droplevels()
table(), cut()

t(), `%*%`, det(), solve(), diag()
sum(), prod(), max(), min()
pmax(), pmin()
cumsum(), cumprod(), diff()
colSums(), rowSums(), colMeans(), rowMeans()

# accessing objects
setwd(), getwd(), ls(), rm()
with(), attach(), detach(),
dput(), dget()
scan(), read.table(), read.csv()
```

```
# loops
if(), for(), while(), next, break, switch(),

# functions
args(), missing(), Recall(), return()
warning(), stop()

# performance and debugging
system.time(),
debug(), debugonce(), undebug(), trace(), untrace()

# statistics
mean(), sd(), var(), cor(), cov()
runif(), rnorm(), rgamma(), rpois(), rbinom(), rexp(),
sample(), set.seed()

# graphics
plot.new(), plot.window(), axis(), legend()
points(), lines(), abline(),
plot(), hist(), boxplot(), pairs()

# arithmetic
abs(), ceiling(), floor(), round(), signif(),
exp(), log(),
`+`, `-`, `*`, `/`, `%%`,

# numerical methods
integrate(), nlm(), optim()
eigen(), svd(), qr()
```