Quick R Primer

June 20, 2013

This document is a brief summary of the main R data types and some useful commands.

0.1 Arithmetic

When you start up R, you see a welcome message and the prompt `>`. In my version of R, the text is in blue and the prompt and cursor appear in red. You can type commands at the prompt which R will interpret. For example, if you type `2 + 3` and press return, your R session will look like this:

```
> 2 + 3
[1] 5
>
```

The `[1]` in the output indicates that 5 is the first entry of the vector 2 + 3. This is explained further below. Examples of other arithmetic operations are $-$, $*$, / and ^. Brackets can be used to group operations. For example, `2^(1 + 1)` returns `4` (2 to the power of 1+1.) Note that the amount of whitespace typed on either side of the operator doesn't matter, so that `2+3` is the same as `2+ 3` or `2 + 3`.

0.2 Variables

You can create a variable and assign it a value by using the assignment operator `<-`. For example, the command

```
x <- 2
```

assigns the value 2 to the newly-created variable `x`. You can also use = to assign values, as in most programming languages, but this is not recommended because there are obscure situations in which it can fail.

Variable names in R are case-sensitive, so that `X` is not the same as `x`.

When you assign a value to a variable, R will not react. When you type the name of a variable, it will print out the value assigned to that variable, if it exists. Putting brackets around a variable assignment will cause R to print the value at the same time as it is assigned. An example R session might look like this.

```
> y
Error: object 'y' not found
> y <- 2
> y
[1] 2
> (z <- 3)
[1] 3
>
```

Many other things can be created in R besides numbers. In general, these are called *objects*. An object of any type can be assigned to any variable. In other words, you can give an object any name you like.

0.3 Vectors

A vector is an ordered sequence of numbers. A vector in R can be produced using `c`, which stands for concatenation. For example:

```
> x <- c(2,3)
> x
[1] 2 3
```

Here, we created a vector `x` containing the values 2 and 3, and then looked at it. To look at the first entry by itself, type `x[1]` and to look at the second entry, type `x[2]`. This is why the `[1]` appears when R displays a vector; it is next to the entry which is `x[1]`. Note that the entries are numbered starting with 1, not 0, unlike many other programming languages.

Trying to access an element which doesn't exist does not give an error. Instead, R returns `NA`, which represents a missing value. Negative indices have a different meaning. Typing `x[-1]` returns `x` with entry 1 deleted. Similarly, typing `x[-5]` deletes the fifth entry. Trying to delete an entry which does not exist does not give an error; if there is no fifth entry, then `x[-5]` is the same as `x`.

```
> x <- c(2,3)
> x[2]
[1] 3
> x[-1]
[1] 3
> x[5]
[1] NA
> x[-5]
[1] 2 3
```

Vectors can be produced in different ways. One common way to define a vector is to use the colon operator. The command `a:b` creates a vector containing the numbers going from `a` to `b` in steps of size 1.

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

The command `seq(a, b, r)` produces a vector from `a` to `b` going in steps of size `r`.

```
> seq(0, 1, 0.2)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

New vectors can be constructed by concatenating old ones using `c`.

```
> y <- c(1:3, x, -0.8)
> y
[1] 1.0 2.0 3.0 2.0 3.0 -0.8
```

Note that vectors can also be indexed by other vectors. For example, to create a vector containing the first and third entries of `y`, you can type `y[c(1, 3)]` instead of `c(y[1], y[3])`.

0.4 Vector arithmetic

Arithmetic operations are performed entrywise on vectors.

```
> c(-1,2) * c(3,4)
[1] -3 8
```

Operations such as addition and multiplication can be performed on vectors of different lengths. This is very useful, but can be confusing if you are familiar with linear algebra. When an operation is performed on two vectors of different lengths, the shorter one is *recycled* to have the same length as the longer one. So, for example, if you type

```
c(2, 3) + c(1, 2, 3, 4, 5)
```

R will first replace `c(2, 3)` by `c(2, 3, 2, 3, 2)` and then perform the addition, to obtain

```
[1] 3 5 5 7 7
```

Other arithmetic operations also work componentwise on vectors.

```
> c(2, 7) * c(2, 4, -2)
[1] 4 28 -4
> (1:3)^2
[1] 1 4 9
```

Scalars (i.e. numbers) are really vectors of length one, so multiplying a vector by a scalar works in the way you would expect. Adding a scalar to a vector adds the scalar to each entry of the vector, in accordance with the recycling rule.

There are many useful vector functions built into R. For example, if `x` is a vector,

```
length(x)
```

gives the number of entries of `x` and

```
rev(x)
```

reverses the entries of `x`. It is important to note that applying a function to an object in R never changes the object itself. For example, if `x` is `c(1,2,3)`, then typing `rev(x)` produces `c(3,2,1)`, but `x` itself is not changed. If you want to reverse the entries of `x` and store the result in `x` then you can type

```
x <- rev(x)
```

0.5 Matrices

A matrix can be constructed using the `matrix` command. The user has to supply a vector of entries of the matrix and the number of rows and columns.

```
> X <- matrix(1:4, nrow=2, ncol=2)
> X
     [1,] [2,]
[1,]    1    3
[2,]    2    4
```

This can be abbreviated to

```
> X <- matrix(1:4, 2, 2)
```

or you can also specify just one of `nrow` or `ncol`, since the other is determined by the length of the vector `1:4`. Notice that matrices are stored in *column-major* order by default. If

```
X <- matrix(v, nrow=nrow, ncol=ncol)
```

is a matrix, then the first `nrow` entries of `v` form the first column of the matrix `X`. To create a matrix in row-major order, either use the `byrow=TRUE` option

```
X <- matrix(1:4, nrow=2, ncol=2, byrow=TRUE)
```

or use the transpose `t`.

```
X <- t(matrix(1:4, nrow=2, ncol=2))
```

An entry of a matrix can be picked out in the same way as with vectors, except that now you must specify both the row and the column.

```
> X <- (matrix(c(1,2,1,3,2,3), nrow=2, ncol=3)
> X[2,3]
[1] 3
```

A row or column of the matrix can be picked out by omitting one of the numbers. For example, `X[1,]` returns the first row and `X[,2]` returns the second column. But notice that individual row and columns are always returned as vectors. If you want them to be $1 \times n$ or $n \times 1$ matrices, you can add a `drop=FALSE` to the command.

```
> X <- matrix(1:4, 2,2)
> X[,2]
[1] 3 4
> X[,2 ,drop=FALSE]
     [,1]
[1,]    3
[2,]    4
```

R regards a matrix `X` as a vector plus a pair of numbers giving the number of rows and columns. This pair of numbers is called the *dimension* of `X` and can be obtained by typing `dim(X)`. The default behaviour when picking out a single row or column is to "drop" the dimension attribute. Including `drop=FALSE` tells R not to drop the dimension attribute. *Attributes* are discussed further in Section 0.13.

Multiplication of matrices happens entrywise, just like with vectors. But the matrices have to be of the same dimensions, or R will give an error. Other binary operations, such as addition and division, work on matrices in the same way. If you want to do the usual matrix multiplication, use the operator `%*%` instead of `*`.

```
> X <- matrix(1:4, 2,2)
> Y <- matrix(5:8, 2,2)
> X * Y
     [,1] [,2]
[1,]    5   21
[2,]   12   32
> X %*% Y
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

The operator `%*%` can also be used to do matrix-vector multiplication. For example, to find the result of subtracting the second column of `Y` from the first column.

```
> v <- c(1,-1)
> Y %*% v
     [,1]
[1,]   -2
[2,]   -2
```

Matrices and vectors can be combined using the binary operations we have already seen. The matrix is temporarily treated as a vector, and the result inherits the dimension of the matrix. This may result in a warning message if the length of the vector does not match the number of entries in the matrix. It is easier to explain via examples.

```
> 2*X
     [,1] [,2]
[1,]    2    6
[2,]    4    8
> c(1,2,3)*X
     [,1] [,2]
[1,]    1    9
[2,]    4    4
Warning message:
In c(1, 2, 3) * X :
  longer object length is not a multiple of shorter object length
```

In the second example, first `c(1,2,3)` was recycled to `c(1,2,3,1)` and then the product

```
c(1,2,3,1) * c(1,2,3,4)
```

was computed. The result was finally placed in a 2×2 matrix.

Special cases of this behaviour are: adding a scalar `a` to a matrix `X` adds `a` to each entry of `X`; multiplying a matrix `X` by a scalar `a` multiplies each entry of `X` by `a`. Neither of these operations gives a warning message because they are so commonly used.

0.6 Strings

A string is a piece of text. Strings can be enclosed in single or double quotes and can contain special characters called *escape characters* such as `\n` (newline) `\t` (tab) and `\"` (double quotes). Typing the name of a string in the console causes it to be printed out. A string `s` can also be printed using `print(s)`.

```
> me <- "Tarzan"
> you <- "Jane Porter\n"
> me
[1] "Tarzan"
> print(you)
[1] "Jane Porter\n"
```

The `[1]` appears in the output because `me` and `you` are really regarded as vectors of strings which happen to have length one.

Another function for printing strings is `cat()`. This function does not print the contents of the string literally; it interprets the escape characters in the string. The name `cat` comes from the fact that you can give it more than one string as an argument and the arguments will be con**cat**enated.

```
> cat(you)
Jane Porter
> cat("me", me)
me Tarzan>
```

Notice that there is no new line because the string did not include the `\n` character. To concatenate and print strings without a space in between, the `sep` argument can be used.

```
cat("alpha", "bet", sep="")
```

One use of strings is to give names to a vector. Names can be assigned to a vector using `names`.

```
> heights <- c(1.98, 1.69, 1.1)
> names(heights) <- c("Tarzan", "Jane", "chimp")
> heights
Tarzan   Jane  chimp
```

```
  1.98   1.69   1.10
```

You can now refer to the second entry of `heights` using `heights[2]` or `heights["Jane"]`. In either case, the output will be:

```
Jane
1.69
```

Note that you can create vectors of strings, like `names(heights)` in this example. Arithmetic operations are not defined for strings. Instead, there are special functions for manipulating strings. A common one is `paste`, which is used to concatenate its arguments into a single string.

```
> x <- paste("example", "number", sep=" ")
> x
[1] "example number"
```

(Note that you cannot use `cat` instead of `paste` because `cat` just prints its arguments to the console; it does not *return* a value.)

Can a vector contain both strings and numbers?

```
> c(x, 1)
[1] "example number" "1"
```

Here, R has *coerced* the 1 into a string because a vector is not allowed to contain objects of different types. To combine objects of different types into a single object, you need a *list*.

0.7 Lists

A list is a collection of objects treated as a single object. A list can be thought of as a vector whose entries are allowed to contain things of different types, such as strings and numbers. Lists are sometimes called *generic vectors*. Ordinary vectors are then called *atomic vectors* to distinguish them from lists. It is very useful to know this because R's error messages sometimes use these terms.

Lists are used extensively in R to bundle data together. For example, the function `eigen` computes the eigenvalues and eigenvectors of a matrix. It returns a list.

```
> X <- matrix(1:4, 2, 2)
> e <- eigen(X)
> e
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

The list `e` consists of a vector and a matrix. The components of the list can be accessed using the `$` operator.

```
> e$values
[1]  5.3722813 -0.3722813
> e$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
> e$vals
NULL
```

Typing `e$vals` gives `NULL` because there `e` has no component called `vals`. This is analogous to trying to access an entry of a vector which does not exist, except that `NULL` is mysteriously different to `NA`.

You can also access the entries of a list by numbers enclosed in double brackets. For example, `e[[1]]` is equivalent to `e$values` and `e[[2]]` is equivalent to `e$vectors` in this example.

To create your own list, you can use the `list` function. For example

```
gull1 <- list(species="herring", wingspan=20)
```

creates a list `gull1` containing a string `gull1$species` and a number `gull1$wingspan`.

R provides some useful functions for working with lists, but it is not usual to do arithmetic with them, unlike with vectors. They are most useful for bundling a lot of data together. For example, suppose you write a function to carry out some statistical procedure. You might want the output of the function to consist of the result plus some diagnostic information, such as a p–value from an appropriate hypothesis test. It would be sensible to make the output of this function a list. Most of R's statistical functions work this way. The `lm` function for linear regression is an example.

```
x <- seq(0, 1, length=50)
y <- 5 * x + 1 + rnorm(50)
result <- lm(y~x)
result
```

The above lines of code create a vector `x` containing 50 numbers from 0 to 1. The vector `y` consists of `5*x+1` plus a vector of random numbers drawn from the normal distribution with mean 0 and standard deviation 1. Then `y` is regressed on `x` and the result is stored in `result`. Finally, typing `result` causes `result` to be printed.

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.9588       5.0429
```

(The numbers 0.9588 and 5.0429 will differ from the numbers you get if you copy and paste the code into an R session because the `rnorm` function generates random numbers.)

Notice that the output from printing `result` to the console does not look like the output from the list `e` that was seen above. How do we know that `result` is a list, and what does it contain? You can see the contents of any R object `x` by typing `str(x)`. In this case, typing `str(result)` gives a description of what is in `result`. In particular, the first line of output tells us that `result` is a list of 12 objects. Some of these objects, for example `result$qr`, are themselves lists.

Why doesn't typing `result` tell us what is in `result`? This is because the author of the `lm` function decided that it would be more useful if typing `result` only printed the most relevant information instead of printing everything. In Section 0.13 we will see that user-defined objects can be made to have the same behaviour by using S3 classes.

0.8 Data frames

A *data frame* is a special kind of list. Essentially it is a list of vectors of the same length. This concept comes from statistics. A *data set* in statistics consists of a number of *variables* measured on some *subjects*. The vectors which make up the data frame are supposed to be the measurements of the variables.

For example, suppose we have some seagulls. For each seagull, we record its species, its wingspan in centimetres, its weight in grams and its sex. A table of the results might look like this.

gull	species	wingspan	weight	sex
1	herring	40	850	M
2	herring	38	840	M
3	herring	30	772	F
4	unknown	50	1000	F

To create a data frame in R from these data, enter the following command.

```
seagulls <- data.frame(species=c("herring","herring","herring",NA), wingspan=c(40,38,30,50),
weight=c(850,840,772,1000), sex=c("M","M","F","F"))
```

Notice that you can copy and paste the command directly into the R console, even though it runs over two lines. In general, R will accept multi-line input. If you press return before you have finished typing a command, the command can be continued on the next line. The prompt will change from `>` to `+` to indicate that a command is being continued.

Printing the `seagulls` object produces a very similar table to the one displayed above.

```
  species wingspan weight sex
1 herring       40    850   M
2 herring       38    840   M
3 herring       30    772   F
4    <NA>       50   1000   F
```

The data frame is displayed like a matrix. Remember that it is also a list, so for example `seagulls$wingspan` gives the `wingspan` column. But data frames can also be manipulated using matrix-like syntax. We can also get the second column by typing `seagulls[,2]`. Or if we wanted to know just the species and sexes of the first three gulls, we could type

```
seagulls[1:3, c(1,4)]
```

to select rows 1 to 3 and columns 1 and 4.

```
  species sex
1 herring   M
2 herring   M
3 herring   F
```

We can extract the first gull with

```
gull1 <- seagulls[1,]
```

and we can find its weight using either `gull1$weight` or `gull1[3]`. So is `gull1` a list or a vector? Neither; it's a data frame, as can be checked by typing `str(gull1)`. (Also, it's technically a list because data frames are a special case of lists.)

If you are familiar with SQL, data frames should be reminiscent of SQL tables. There are add-on packages for R which enable data frames to be queried using SQL syntax. R itself also has powerful methods for dealing with data frames. Some of these will be mentioned in Section 0.10.

Notice something odd which happens if we look at `seagulls$sex` or `seagulls$species` in this example.

```
> seagulls$sex
[1] M M F F
Levels: F M
> str(seagulls$sex)
 Factor w/ 2 levels "F","M": 2 2 1 1
```

Didn't we enter the sexes as `c("M","M","F","F")`? Shouldn't `seagull$sex` be a vector of strings? Yes, but now it is no longer a vector of strings; it has been quietly converted into something else; a *factor*.

0.9 Factors

The idea of a factor comes from experimental designs. Suppose we gave each of six plants a high or low dose of fertiliser. We could record the doses in a vector of strings.

```
c("high","high","high","low","low","low")
```

Or we could give each possible *level* a code, say 1 for high and 2 for low. Then we could record the doses in a vector of integers

```
c(1, 1, 1, 2, 2, 2)
```

and note that 1 stands for high and 2 stands for low. This is exactly what a factor is. A factor is a vector whose entries are positive integers in the range $1, 2, \ldots, n$, together with a vector of n *levels* where i stands for level i.

Many R functions take factors as arguments. One obvious question is: why is a factor better than a vector of strings? This is a very good question, since most of the time there is no harm in working with vectors of strings, whereas factors can be annoying. One possible reason is that factors are more robust to errors. For example, consider `seagulls$sex`. This is a factor of length 4, but you can imagine that we might have a much larger number of seagulls. Suppose someone wants to change the sex of the first seagull and accidentally types `D` instead of `F`.

```
> x <- seagulls$sex
> x[1] <- "D"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "D") :
  invalid factor level, NAs generated
```

R warns us that we entered a code which is not one of the allowed levels and an `NA` appears in the factor. The warning can potentially be useful.

On the other hand, working with factors can be a pain. For example, we have seen that creating a data frame with the `data.frame()` function causes every string-valued variable to be converted into a factor. This might be very useful with a variable like sex where there are only a few possible categories, but what if we are inputting people's addresses? Then everyone basically has a unique value, and it doesn't make much sense to give each possible address a numeric code.

You can avoid strings being converted into factors by using the `stringsAsFactors` option when creating a data frame. Here is an example.

```
seagulls <- data.frame(species=c("herring","herring","herring",NA), wingspan=c(40,38,30,50),
weight=c(850,840,772,1000), sex=c("M","M","F","F"), stringsAsFactors=FALSE)
```

This also works with some other functions which create data frames, such as the `read.table()` function which is used for reading tables from text files and storing the result in a data frame.

0.10 Logic

R has two inbuilt constants `TRUE` and `FALSE`. An abbreviation for `TRUE` is `T` and similarly an abbreviation for `FALSE` is `F`. However, you can also use `T` and `F` as variable names, whereas `TRUE` and `FALSE` cannot be used as variable names. When doing logic, it is therefore better to use the names `TRUE` and `FALSE` in case you accidentally overwrite one or both of `T` or `F` during your R session.

Vectors of logical values can be created and the binary operators `&` (and), `|` (or) and `!` (not) are available. They act componentwise on vectors like the other binary operators we have seen.

```
> c(TRUE, FALSE) | c(FALSE, FALSE)
[1]  TRUE FALSE
> c(TRUE, FALSE) & c(FALSE, FALSE)
[1] FALSE FALSE
> !c(TRUE, FALSE)
[1] FALSE  TRUE
```

You can also do arithmetic with logical values, in which case `TRUE` will be treated as 1 and `FALSE` as 0. So, for example, `TRUE + TRUE` evaluates to 2.

Logical vectors can also be created using the comparison operators `>`, `>=`, `<`, `<=`, == (a single = cannot be used to compare two values.)

```
> x <- c(1, 1, 3, 4, 7)
> x<3
[1]  TRUE  TRUE FALSE FALSE FALSE
> (x<3)|(x>=4)
[1]  TRUE  TRUE FALSE  TRUE  TRUE
```

Logical vectors can be used for indexing. Recall that if `x` is a vector, we can write `x[c(1,3)]` to get a vector containing the first and third entries of `x`. If we want to get a vector containing those entries of `x` which satisfy some condition, we can write `x[cond]` where `cond` is the condition. This will produce a vector of those values in `x` for which `cond` evaluates to `TRUE`.

```
> x[(x<3)|(x>=4)]
[1] 1 1 4 7
```

A very useful function related to this is `which`, which returns the indices for which some specified condition is true.

```
> which(x < 3)
[1] 1 2
```

This states that the first and second entries of `x` are less than 3 and the other entries are not less than 3. Notice that `x[which(x < 3)]` is the same as `x[x < 3]`.

The same ideas can be used to index matrices and data frames. For example, if `X` is the 3×3 matrix `matrix(1:9, 3, 3)` then

```
X[c(FALSE, TRUE, TRUE), c(TRUE, TRUE, TRUE)]
```

will return the submatrix consisting of the last two rows of `X`. The second logical vector can be omitted because typing nothing is equivalent to selecting all rows.

```
X[c(FALSE, TRUE, TRUE), ]
```

The logical vectors can be obtained by setting conditions on the rows and columns. Here is an example.

```
> X[X[,1] > 1, X[2,] >= 5]
     [,1] [,2]
[1,]    5    8
[2,]    6    9
```

This command picked out the rows where the first column has an entry > 1, which is the last two rows, and the columns where the second row has an entry ≥ 5, which is the last two columns.

Rows and columns of a matrix whose rows and columns have been named (by using `rownames` and `colnames`) can be referred to by name instead of by number. Similarly, the columns of a data frame can be referred to by name and indexed in the same way as matrices.

Going back to the `seagulls` data frame of Section 0.8, suppose we wanted to know the species and wingspan of all seagulls which were either female or had weight at least 850. It can be done this way.

```
> seagulls[seagulls$sex=="F" | seagulls$weight >= 850, c("species", "wingspan")]
  species wingspan
1 herring       40
3 herring       30
4    <NA>       50
```

This is equivalent to the following SQL query.

```
select species, wingspan from seagulls
where weight >= 850
and sex = "F";
```

In this way, SQL-like searches can be performed on R data frames. Note that if we had written `weight` instead of `seagulls$weight`, R would have given an error because there is no object named `weight` in our environment. To refer to the variables by their names without the `seagull$` bit, you can enclose everything in the `with` command. Here,

```
with(seagulls, seagulls[sex=="F" | weight >= 850, c("species", "wingspan")])
```

gives the same result as before.

0.11 Control flow

So far, we have seen various data types in R. Now we will move on to R programming.

An alternative to typing commands into the prompt at the R console is to write them in a text file, save the file with a .R extension, and then load it into R using the `source` command. For example, if the file is saved as `file1.R` then typing

```
source("file1.R")
```

will cause all the commands in the file to be executed in order. But sometimes it is necessary to execute them in some more complicated way, and this is where control flow is used.

The **for loop** is used for executing a statement or sequence of statements (called a *code block*) repeatedly, possibly with a minor change between repeats. For example,

```
for (i in 1:3){
    print(i^2)
    }
```

causes the following output to be printed to the console.

```
[1] 1
[1] 4
[1] 9
```

The code inside the curly brackets is executed with each value of `i` in turn, so this loop is equivalent to the following.

```
print(1^2)
print(2^2)
print(3^2)
```

The general form of the for loop is

```
for (i in v){ ... }
```

where `v` is a vector and `...` is a set of statements which may or may not involve `i`. Any other name apart from `i` may be used; it is called a *dummy variable*, like the x in $\int f(x)dx$. Commonly `v` is chosen to be `1:n` for some `n`. If there is only one line of code in the loop, the curly brackets can be omitted, so the above loop is equivalent to writing

```
for (i in 1:3) print(i^2)
```

which runs slightly faster (as we will see, typing `{` is actually a function call in R.) However, many people recommend always including the curly brackets in order to avoid making errors.

The **while loop** is more versatile than the for loop. It has the form

```
while (cond){ ... }
```

Where `cond` is a logical statement and `...` represents some commands which will be executed over and over again as long as `cond` evaluates to `TRUE`. Usually the body of the loop will change `cond` in some way, otherwise the loop will continue executing forever.

The above example of a for loop could be executed as a while loop by writing

```
i <- 1
while (i <= 3){
    print(i^2)
    i <- i+1
    }
```

which is equivalent to the following program.

```
i <- 1
print(i^2)
i <- 2
print(i^2)
i <- 3
print(i^2)
i <- 4
```

Here the variable `i`, which is counting the number of times we have been through the loop, is not a dummy variable. A while loop doesn't even have to be executed a fixed number of times. For example, a while loop can be used to write a program which simulates rolling a die until a six is obtained. Here, the command `sample(v, k)` generates a vector of `k` random samples drawn without replacement from the vector `v`.

```
roll <- 1
while (roll != 6){
    roll <- sample(1:6, 1)
    cat(roll, "\n")
    }
```

Note that we could have set `roll` to anything except 6 at the beginning; all that matters is that `roll != 6` is `TRUE` so that the program goes through the loop at least once.

The **if statement** is used to cause a program to branch depending on whether a particular logical statement is true or false. Its general form is

```
if (cond){ ... }
```

which causes the statements `...` to be executed if `cond` is `TRUE`. If `cond` is `FALSE`, they are ignored. An `else` can also be included. The command

```
if (cond){
    ... }else{
    ...
    }
```

causes the first block of code to be executed if `cond` is `TRUE` and the second block to be executed if `cond` is `FALSE`. The way the lines are indented is a matter of taste; remember that R ignores whitespace. However, it is important to note that the `else` should be on the same line as the `}`. If it isn't, then R will interpret the

```
if (cond) {
    ...}
```

as a complete statement by itself. Then it will try to move on to the `else` and give the following maddening error message.

```
Error: unexpected 'else' in "else"
```

Just like with the for and while loops, the curly brackets can be omitted if there is only one statement in a block.

0.12 Functions

?? We have already seen plenty of examples of R functions. An example is the `length` function which takes a vector and returns its length. If the vector is `x`, its length is found by typing `length(x)`. Here, `x` is called the *argument* of the function. A function can have more than one argument. For example, the `c` function can have any number of vectors as arguments. For example, we can type

```
c(c(1,2), c(2,3), c(3,4))
```

to concatenate three vectors. Some functions have optional arguments, for example the `stringsAsFactors` option in the `data.frame` function.

You can extend R by writing your own functions. The syntax to define a function `f` taking the argument `x` is

```
f <- function(x){
    ...
    }
```

where `...` represents some instructions, called the *body* of the function. As an example, here is a function which computes the absolute value of a number `x`. There is already a function `abs` which does this in R, but here is how it can be done independently.

```
f <- function(x){
    if (x>0){
        return(x)
        }
    else{
        return(-x)
        }
    }
```

Here, the `return` tells the function to stop what it is doing and return a value. We can test the function out.

```
> f(-5)
[1] 5
> f(3.2)
[1] 3.2
>
```

We can also assign a variable to the return value of the function. For example, `x <- f(-5)` assigns the value 5 to `x`.

There is nothing to say what kind of object the argument of the function should be. A function in R can take any kind of object as an argument. If you try to apply a function to the wrong kind of object, things can go wrong. For example, trying to apply `f` to a vector may give the wrong answer.

```
> f(c(1,-1))
[1]  1 -1
Warning message:
In if (x > 0) { :
  the condition has length > 1 and only the first element will be used
```

What happens here is that the function needs to evaluate the logical expression `c(1,-1)>0` which evaluates to `c(TRUE, FALSE)`. As the warning message tells us, putting a vector inside an `if` means that only the first element of the vector will be considered. In this case, the first element is `TRUE`, so `f` follows the `TRUE` branch of the `if` statement and returns its argument `x`, which in this case is `c(1,-1)`.

If we wanted to write `f` in such a way that it could be applied to vectors, it could be done like this.

```
f <- function(x){
    x[x<0] <- -x[x<0]
    return(x)
    }
```

This version of `f` picks out those entries of `x` which are negative and replaces them by their negatives, which is all that is required. In R, a function automatically returns the last value it has computed. So it is not actually necessary to include a `return` statement and many people would write the function like this.

```
f <- function(x){
    x[x<0] <- -x[x<0]
    x
    }
```

A function can have more than one argument. For example, here is a function to compute the Newton quotient of a given function `f` at a value `x` with stepsize `h`.

```
Newton.quotient <- function(f, x, h){
    (f(x+h)-f(x))/h
    }
```

If the body of a function only contains one line, then the curly braces can be omitted. So we could write

```
Newton.quotient <- function(f, x, h) (f(x+h)-f(x))/h
```

and test the function on the built-in function `sin`, which differentiates to `cos`.

```
> Newton.quotient(sin, 1, 0.01)
[1] 0.536086
> cos(1)
[1] 0.5403023
```

Arguments to functions can be given default values. Suppose we usually want to evaluate the Newton quotient with $h = 0.01$ but want to retain the option of using another value of h. The function can be rewritten as

```
Newton.quotient <- function(f, x, h=0.01) (f(x+h)-f(x))/h
```

Then we can get the same result as before by typing `Newton.quotient(sin, 1)`. But we could also type `Newton.quotient(sin, 1, 0.001)` or `Newton.quotient(sin, 1, h=0.001)` if we wanted to use $h = 0.001$ instead.

Many functions which are likely to be needed are built in to R. One way of finding out how to do something is by using the R help system. To find information on a topic `topic` you can type

```
??"topic"
```

and R will open a webpage with a list of help pages which mention the topic. If you are offline, the help system can still be access provided that you have a web browser installed. To get the help page for a specific function, you can type `?<name>` where `<name>` is the name of the function. For example

```
?sample
```

brings up a page on the `sample` function explaining which arguments are available, what it does, and giving examples of how to use it. Most functions are well-documented and running the examples can be helpful. A quick way of running these examples in the R console is `example(sample)`.

Some functions have names which include special characters. For example, we have seen that the logical operator 'not' is denoted `!`. But typing `?!` does not produce a help page. Because `!` is a special character, it must be enclosed in backticks. The backtick is a special character; on my keyboard it is in the top left hand corner. The command ``?`!` `` brings up the correct help page.

There are actually many functions in R which have odd names like this, because many things are functions in disguise. For example, `+` is a function; instead of writing `2 + 3` we can write `` `+`(2,3) `` and it is the same thing. This can be the useful for understanding some of R's error messages. We saw the following message earlier.

```
> x[1] <- "D"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "D") :
  invalid factor level, NAs generated
```

which reveals that in typing `x[1] <- "D"` we were actually invoking a function called `[<-.factor` with the arguments `x, 1` and `"D"`. Similarly, `for`, `while` and `if` are actually functions.

If the help page for a function is not helpful enough, or there is no help page available, you can also view a function's source code by typing its name at the prompt. For example:

```
> Newton.quotient
function(f, x, h) (f(x+h)-f(x))/h
```

If the function's source code is very long, it might be better to open it in a separate page using the `page` command. For example, to read the source code for the `?` function, use `page(`?`)`, remembering the backticks.

Many of the most fundamental R functions are not written in R.

```
> abs
function (x)  .Primitive("abs")
```

To find out how `abs` works, it is necessary to search the R source code, which can be downloaded from the internet. The function is actually written in C. Similarly, if we try to look at the `sample` function, R displays the following.

```
> sample
function (x, size, replace = FALSE, prob = NULL)
{
    if (length(x) == 1L && is.numeric(x) && x >= 1) {
        if (missing(size))
            size <- x
        .Internal(sample(x, size, replace, prob))
    }
    else {
        if (missing(size))
            size <- length(x)
        x[.Internal(sample(length(x), size, replace, prob))]
    }
}
<bytecode: 0x0b1ed324>
<environment: namespace:base>
```

It turns out that R is just calling its internal `sample` function, written in C. The last line of output tells us that the `sample` function is located in an *environment* called `namespace:base`. Environments are places where R objects 'live' and there are powerful programming techniques which exploit them.

We have seen how to include optional arguments in a user-defined function but we have not yet seen how to write functions like `c` which take an unspecified number of arguments. This can be done using the ellipsis `...` which stands for "some unspecified number of further arguments".

Here is an example. One of the most useful functions in R is the `plot` function. This function has a lot of optional parameters. A command like `plot(a,b)` where `a` and `b` are numbers will plot an open circle with a black outline at the point (a, b). The axes are automatically chosen so that (a, b) is in the middle of the screen. Further points can be added using `points`, since calling `plot` again will create a new window. This function also has optional parameters. For example,

```
points(a, b, col="blue", pch=20)
```

will add a filled blue circle at the point (a, b) to the existing plot. Now suppose we wish to write a function which will create a new graphics window and fill it with polka dots. Here is how it could be done.

```
polka.dots <- function(x=10, y=10, ...){
    plot(0, xlim=c(1,x-0.5), ylim=c(1,y), type="n", xaxt="n", yaxt="n", xlab="", ylab="")
    for (i in 1:x){
        for (j in 1:y){
            points(i - 0.5*(j%%2), j, ...)
            }
        }
    }
```

The first call to the `plot` function plots something (a zero) but we include `type="n"` to make sure that nothing appears. The commands

```
xaxt="n", yaxt="n", xlab="", ylab=""
```

suppress the axes and axis labels. The `xlim = c(1,x-0.5)` argument specifies that the range of x values in the plot should go from 1 to `x-0.5`. Similarly for `ylim`. Having called `plot`, we call the `points` function xy times to draw the dots. The `j%%2` is the remainder when `j` is divided by 2; this is a binary operator which was not defined above.

The point here is that any extra arguments we pass to the `polka.dots` function are passed on to the `points` function via `...`. For example, typing

```
polka.dots()
```

produces a 10×10 pattern of default dots. But

```
polka.dots(20, 20, col="blue", pch=16)
```

produces a 20×20 pattern of filled blue dots. There are many other arguments which could be included as well. See `?points` for a list of them.

Another interesting property of the `plot` function is that it behaves differently on different objects. If you call `plot` on the vector `1:4`, the result will look quite different from the result you get by calling it on the matrix `matrix(1:4, 2, 2)`. But we have seen that this matrix is really just the vector `1:4` with an extra attribute, its dimension. A function that treats different types of objects differently like this is called a *generic* function. You can create generic functions using S3 classes. These are explained in Section 0.13.

Sometimes a function needs to be used just once and it is undesirable to give it a special name. A good example occurs when using the `lapply` function. This is a very powerful function which applies a given function to each entry of a list. The syntax for `lapply` is

```
lapply(a.list, a.function)
```

For example, suppose we have a list `mylist` containing some numerical vectors and we wish to extract the first entry of each vector in the list and return a list of the results. This can be done in the following way.

```
mylist <- list(c(1,2,3), c(-1,1,0), c(4,5), 12)
f <- function(vect) vect[1]
lapply(mylist, f)
```

Here, the function `f` is left in our workspace, but we do not need to use it again. We can instead declare `f` inside the call to `lapply` like this.

```
mylist <- list(c(1,2,3), c(-1,1,0), c(4,5), 12)
lapply(mylist, function(vect) vect[1])
```

This can be an efficient way of doing things. The function declared inside `lapply` is called an *anonymous* function because it was never given a name.

Here is an example using both `...` and `lapply`. Suppose we want a function which takes any number of arguments $x_1, x_2, \ldots, x_n$. It then rolls an x_i–sided die for each i and returns a list of the resulting rolls.

A little thought shows that we don't yet know how to do this because we cannot extract the individual numbers from the input `...`. Fortunately, this can be done easily by using the command `list(...)`. This converts the intractable `...` into a list. The function can be written like this.

```
rolldice <- function(...){
    sides <- list(...)
    rolls <- lapply(sides, function(x) sample(x,1))
    rolls
    }
```

The `list(...)` trick enables functions like `c`, which take a variable number of arguments, to be written. Note that in the `rolldice` function, we used the syntax `sample(x, 1)`. This is a shorthand for `sample(1:x, 1)`, that is, sampling one random number from the vector `1:x`, as explained in `?sample`.

0.13 S3 classes

R has several systems for object-oriented programming, but S3 classes are the simplest. An *object* is just a collection of data packaged together. We have already seen this idea in R because data can be packaged together into a list. It may be desirable to view objects of the same type as members of a single *class* and then write functions especially for dealing with this class. This is easy to do in R.

First we need to know how to set *attributes* of an object. Attributes are just extra properties which an object might have. It is possible to create whatever attributes you like. Here is an example.

```
x <- factor(c("high", "low", "high", "low"))
attr(x, "notes") <- "levels of fertilizer for experiment 5"
```

We defined a factor `x` and then gave it a new `notes` attribute. This attribute appears if we print `x`

```
> x
[1] high low  high low
attr(,"notes")
[1] levels of fertilizer for experiment 5
Levels: high low
```

The attributes of an object can be viewed with the `attributes` function.

```
> attributes(x)
$levels
[1] "high" "low"

$class
[1] "factor"

$notes
[1] "levels of fertilizer for experiment 5"
```

It turns out that `x` already had two attributes. The first is the `levels`, which every factor has, as we have seen. The second is the `class`, which in this case is `"factor"`. Note that the `attr` function can also be used to make arbitrary changes to these attributes. For example, it is perfectly legal to type `attr(x, "levels") <- 3`. This will change each `"high"` in `x` to a `3` and each `"low"` to an `NA`, as `x` will now have only one level.

The `class` attribute determines how generic functions act on `x`. For example, the `print` function will display the following when we type `print(x)` (or equivalently, just type `x` into the console; this automatically calls `print`.)

```
> print(x)
[1] high low  high low
Levels: high low
```

If the class of `x` is changed, `print` will no longer behave in the same way. Here is what happens if `x` is changed to a character vector.

```
> class(x) <- "character"
> print(x)
[1] "1" "2" "1" "2"
attr(,"levels")
[1] "high" "low"
```

Nothing has been changed except for the `class` attribute, but the output is different. When a function is called on an object with a `class` attribute, say `factor`, R will first search for a function called `print.factor`. If such a function exists, it will be called. Otherwise, R will call the default print function. We have already seen another example of this; remember the error message involving the function `[<-.factor`. This is the version of the function `[<-` which operates on factors.

Functions like `print.factor` and `print.lm` are called *methods*. The first is a method of the class `factor` and the second is a method of the class `lm`. When `print` is called on an object of the appropriate class, R automatically calls the `print` method for that class.

Here is an example showing how to write your own methods.

```
> gull1 <- list(species="herring", wingspan=20)
> gull1
$species
[1] "herring"

$wingspan
[1] 20
```

To get a better way of printing the details of a gull, we can write a `print.gull` function.

```
print.gull <- function(gull){
                cat(gull$species, " gull. Wingspan: ", gull$wingspan, "cm\n", sep="")
                }
```

This doesn't do anything yet, but we can type

```
attr(gull1, "class") <- "gull"
```

or

```
class(gull1) <- "gull"
```

and now printing `gull1` gives the desired output.

```
> gull1
herring gull. Wingspan: 20cm
```

It is still possible to see how `gull1` was defined by using `str`. Another way of seeing the internal structure of an object is by stripping it of its class using the `unclass` function. Or the class of an object `x` can be taken away by using `class(x) <- NULL`. (The attributes of an object are a list, and remember that a component of a list is deleted by setting it to `NULL`.)

Other methods which are commonly written for homemade classes are `plot` and `summary`. Functions like `plot` and `summary` for which methods can be written are called *generic functions*. Not all R functions are generic.

Earlier we saw functions called `Newton.quotient` and `polka.dots`. Are these methods for the `quotient` and `dots` classes? They are not. But if we created a class called `quotient` or `dots` and functions called `Newton` or `polka`, R would call these functions, possibly with undesirable consequences. For this reason, many authorities do not recommend using dots in the names of variables and functions. Unfortunately the use of dots to separate words in the names of variables and functions is a common R programming convention, which allegedly dates back to an ancestral language in which underscores in names were illegal.

Because an object in an S3 class is really just a list, there is nothing to stop its components from being changed, removed or overwritten by someone who is using it. Object-oriented programmers find this behaviour undesirable,

so another class system, called *S4 classes* was added to R in order to have a better way of doing object-oriented programming. It is not as widely used as S3, but it is important to be aware of it, as it is used in some add-on packages, including ones that ship with R, for example the neural net package `nnet`.

An object can have more than one class. For example, suppose we introduce a `bird` class which is more general than `gull`. We can define a `print` method for this new class.

```
print.bird <- function(x) cat("Tweet\n")
```

We can test that this works by creating an object, giving it the class `bird`, and trying to print it.

```
> x <- NA
> class(x) <- "bird"
> x
Tweet
```

Any object with the class `bird` will say `Tweet` if its name is typed at the command prompt.

Suppose we now *add* the class `bird` to `gull1`. What happens?

```
> class(gull1) <- c("gull", "bird")
> gull1
herring gull. Wingspan: 20cm
```

In this case, R still called the function `print.gull`. But if we write the classes in the opposite order, R will call `print.bird` instead.

```
> class(gull1) <- c("bird", "gull")
> gull1
Tweet
```

The rule is that if an object has classes `x1`, `x2`, ..., then R first tries to find a function `print.x1` if it exists, then tries `print.x2`, and so on. So if we remove the `print.bird` function from our R session, then R should behave as though the class of `gull1` is just `gull`. The `rm` function can be used to remove an object.

```
> rm(print.bird)
> gull1
herring gull. Wingspan: 20cm
```

If we remove the `print.gull` method as well, then R will just call the default `print` function.

```
> rm(print.gull)
> gull1
$species
[1] "herring"

$wingspan
[1] 20

attr(,"class")
[1] "bird" "gull"
```

The is how the idea of *inheritance* is implemented with S3 classes. To create an object with belongs to a subclass `special` of a more general class `general`, you simply set its class attribute to `c("special", "general")`. Methods for class `special` will be used when they exist, and otherwise methods for `general` will be used.

The `unclass` function can be useful for seeing the internal structure of an object. We have already seen that `str` will print a summary of how an object is built up out of lists, but `unclass` is an alternative which will print the whole thing.

For example, if you type

```
x <- 1:2
y <- 3:4
unclass(lm(y~x))
```

you will see the whole structure of the object `lm(y~x)` as a list of objects, some of which are lists. Remember that R functions do not change their arguments. If you want to strip an object `x` of its class, you can write `x <- unclass(x)`. The R help (accessed with `?class` or `help(class)`) recommends instead `class(x) <- NULL`.

0.14 Environments

Suppose we define a function

```
f <- function(x) x^2
```

and then try to calculate 2^2 but accidentally use the wrong kind of brackets.

```
> f[2]
Error in f[2] : object of type 'closure' is not subsettable
```

It seems that R is telling us that we cannot take a subset of a function, except that it calls `f` a *closure*.

```
> typeof(f)
[1] "closure"
```

Apparently `f` is a closure? Isn't it a function?

```
> f
function(x) x^2
```

It certainly seems to be a function. So why is R calling it a closure?

It turns out that a closure is just a function which is 'aware' of its enclosing *environment*. An environment is a collection of objects which in turn is contained in some other environment, called its *parent environment*. (The biggest one is defined to be its own parent.)

Environments can be created in R using `new.env`.

```
e <- new.env()
e$a <- 2
e$g <- function(x) x^3
```

Here, an environment `e` was created and then *populated* with a numeric value `a` and a function `g`. Attempting to print `e` will just return a hexadecimal number representing its location in the computer's memory. To see what is contained in the environment, you can use the `ls` function.

```
> e
<environment: 0x0ae1a588>
> ls(e)
[1] "a" "g"
```

We could equally well have created a list containing `a` and `g` as its components. But the environment `e` also has access to its parent environment.

```
> parent.env(e)
<environment: R_GlobalEnv>
```

This reveals that the parent of `e` is the *global environment*, which is the one in which you are working by default at the beginning of an R session. Typing `ls()` without arguments will list all the objects in the global environment.

Calculations within `e` can be carried out by using `with`.

```
> with(e, g(a))
[1] 8
```

In Section 0.8 we saw the use of `with` with data frames. Remember that data frames are a special case of lists; `with` can also be used with lists. For example, if we define a list

```
mylist <- list(a=2, g=function(x) x^3)
```

then

```
with(mylist, g(a))
```

is a shorter way of writing the following command.

```
mylist$g(mylist$a)
```

Here is how to define a closure. First define a function, and then choose its enclosing environment. Functions are assigned to the global environment by default, but their enclosing environment can be changed using `environment`.

```
> h <- function(x) x*a
> environment(h)
<environment: R_GlobalEnv>
> environment(h) <- e
> environment(h)
<environment: 0x0b341898>
```

The function `h` returns a value which depends on `a`. When it is evaluated, `h` will search for a variable called `a` in its enclosing environment and it will use this as the value of `a`. Since `e$a` is 2, `h` should return double its argument.

```
> h(6)
[1] 12
```

If we assign `h` to an environment in which `a` has a different value, we will get a different result.

```
> n <- new.env()
> n$a <- 3
> environment(h) <- n
> h(6)
[1] 18
```

Rather than using the dollar sign, variables can also be assigned to environments using `assign`. For example, `assign("a", 3, envir=n)` is equivalent to `n$a <- 3`. Similarly, `get` can be used to get the value of a variable. The command `get("a", envir=n)` is equivalent to `n$a`.

The functions `assign` and `get` are extremely useful, even if you are only working in the global environment and don't care about environments at all. This is because they enable the user to refer to objects using strings (`"a"` in the above examples.) This is often convenient; for example, when iterating through a collection of objects.

Why care about closures and environments at all? Unless you are writing an R package, you might never need to worry about these details. But here are a couple of useful things that can be done with them.

Firstly, closures let you write functions which return functions. For example, the `Newton.quotient` function from Section **??** could be written in this way.

```
Newton.quotient <- function(F, h) function(x) (F(x+h)-F(x))/h
```

This function takes as arguments a function `F` and a stepsize `h` and returns another function which is an approximation to the derivative of `F`. We could plot the `sin` function and its numerical derivative in this way.

```
x <- seq(-3, 3, 0.01)
Dsin <- Newton.quotient(sin, 0.01)
plot(x, sin(x), "l")
lines(x, Dsin(x), col="blue")
```

The function `Dsin` lives in a newly-created environment, as can be checked with `environment`.

```
> ls(environment(Dsin))
[1] "F" "h"
> environment(Dsin)$F
function (x)  .Primitive("sin")
> environment(Dsin)$h
[1] 0.01
```

We can see that `F` and `h` are still being stored somewhere, as they have to be if `Dsin` is going to be evaluated.

Environments can be used to intercept function calls. For example, suppose we decide that it would be useful if fitting a linear model would automatically make a scatterplot and add the line of best fit when it is called. How can we make it do this without rewriting the whole function? It can be done using environments.

When we type something like `lm(y~x)` at the R prompt, we know that R creates an object of class `lm` and calls `print.lm` on it. We can modify `print.lm` to do what we want.

```
> print.lm
function (x, digits = max(3, getOption("digits") - 3), ...)
{
    cat("\nCall:\n", paste(deparse(x$call), sep = "\n", collapse = "\n"),
        "\n\n", sep = "")
    if (length(coef(x))) {
        cat("Coefficients:\n")
        print.default(format(coef(x), digits = digits), print.gap = 2,
            quote = FALSE)
    }
    else cat("No coefficients\n")
    cat("\n")
    invisible(x)
}
<bytecode: 0x0ad7a944>
<environment: namespace:stats>
```

The `print.lm` function ends with a call to `invisible`, which is a function which returns its argument invisibly. We can create our own version of `print.lm` with a modified `invisible` which will do what we want.

```
myprint.lm <- function(x){
    e <- new.env()
    myprint.lm <- stats::print.lm
    environment(myprint.lm) <- e
    e$invisible <- function(x){
        plot(x$model$x, x$model$y, xlab="", ylab="")
        abline(x$coefficients)
        invisible(x)
        }
    myprint.lm(x)
    }
```

We can check that this works with:

```
x <- 1:50
y <- x + rnorm(50)
myprint.lm(lm(y~x))
```

How does `myprint.lm` work? First, it creates a new environment `e`. It then fetches the `print.lm` function. This function lives in the `stats` package, which is why it has to be fetched using a special double-colon operator. It then creates a copy of the `print.lm` function called `myprint.lm` and places it in the environment `e`. It then creates a function called `invisible` in the environment `e` which produces the required plot and then carries on in exactly the same way as the `invisible` function. Having set up the environment `e`, it calls the function `myprint.lm`. This function behaves exactly like the function `print.lm`, except that when it needs to access an

object, it searches first in the enviroment `e`. In particulat, when it needs to call a function called `invisible`, it finds the function `e$invisible` and calls it, producing the desired plot.

You can even, if desired, override the behaviour of `print.lm` by simply assigning `myprint.lm` to `print.lm` using

```
print.lm <- myprint.lm
```

and then typing `lm(y~x)` will produce the plot automatically.