1 Aims of the Chapter

2 Using functions in R

2.1 Running a function

2.2 Assignment output

2.3 `<-` versus `=`

2.4 Display

2.5 Checking the values of objects you create

3 Exercises A

4 Masking existing objects and functions

5 Vectors

5.1 Numeric vectors

5.2 Creating vectors of integers

5.3 Vector Arithmetic

5.4 Recycling in vectors

5.5 Creating an empty vector of specified length

5.6 Missing values

5.7 Character vectors

5.8 Logical vectors

5.9 Boolean operators (and, or, not)

5.10 Subsetting

6 Exercises B

7 Higher dimensional objects

7.1 Matrices

7.2 Subsetting matrices

7.3 Preserving dimensions when subsetting matrices

7.4 Arrays

7.5 Matrix multuplication

8 Lists

8.1 `[ ]` versus `[[ ]]` (a common mistake with lists)

9 Exercises C

10 References

Chapter 3 - The R Language

1 Aims of the Chapter

This chapter introduces some of the basic vocabulary and syntax of the R language, the basic commands and how they work. By the end of this chapter you should know:

- how to run built-in functions in R;
- how to assign the output of these functions to objects in R;
- when to use `<-` versus `=`;
- how to create and manipulate vectors in R;
- how R uses vectorisation
- how R recycles vectors
- about the different types of missing values in R;
- how to subset a vector in various ways
- how to
    - construct matrices and arrays
    - work with matrices and arrays
    - subset matrices and arrays
    - preserve dimensions when subsetting
- how to
    - construct lists
    - work with lists
    - subset lists
- the difference between the subsetting operators `[ ]` and `[[ ]]`

2 Using functions in R

2.1 Running a function

To evaluate a function, just type its name followed by a list of any arguments in rounded brackets. R functions often have default values that will be assumed if you do not specify different values. Consider the `dnorm` function which returns the probability density of a normally distributed random variable. Looking at the help file for `dnorm` we see what arguments we could specify and what their default values are. We see that the default mean and standard deviation are 0 and 1 respectively. If we do not specify them when we call the function these are the values R will assume. So

```
dnorm(x = 2)
```

```
## [1] 0.05399097
```

returns the density of a random variable with a $N(0, 1)$ distribution, evaluated at the value 2. If we want to evaluate the density of a random variable with a $N(3, 4)$ distribution, evaluated at the value 2 then we need to use

```
dnorm(x = 2, mean = 3, sd = 2)
```

```
## [1] 0.1760327
```

Ignore the `[1]` for now; we'll see what it means later.

The `help` function is an R function which returns no value: it does nothing except produce a side-effect (calling up a help page). Here `help` can take one argument which is the name of the object of interest.

You can run multiple functions in the same line, for example

```
log(exp(3))
```

```
## [1] 3
```

returns 3 as expected. Applying too many functions in on line can however lead to code that is difficult to read.

2.2 Assignment output

The results of an evaluation can be assigned invisibly to a variable, as in

```
x <- rbinom(n = 10, size = 5, prob = 0.2)
```

which creates a vector of 10 independent realisations of a binomial random variable (with 5 trials, and probability of success 0.2), and calls it `x`. If `x` had already been defined by the user, it would have been over-written. To assign numerical values directly, use the format

```
x <- 3
```

to set `x` to the value 3, and

```
x <- c(2, 4, 6)
```

to set `x` to the vector $(2, 4, 6)$. Note that the function `c` performs concatenation, that is it sticks things together, usually in a vector.

2.3 `<-` versus `=`

`<-` and `=` are used in different ways in R. Consider creating a sequence of values using the `seq` function. In the assignment

```
x <- seq(from = 1, to = 10, by = 3)
```

`<-` is used to assign the sequence to the object `x` whilst `=` is used inside the `seq` function to specify argument values. Note that we have now replaced the value of `x` in the working environment with the seq of integers from 1 to 10.

2.4 Display

To see the value of an object in your working environment, just type its name. Try typing `x` and pressing *Return*. This also applies to functions. Just typing the name of a function without `()` results in the function code itself being displayed. Try typing `help` without `()`; this shows the code used in the R `help` function. This is the easiest way to check the syntax of functions you have created. If you then need to edit your function code you should do this in a script file.

2.5 Checking the values of objects you create

Suppose we create a vector `x` using

```
x <- rbinom(n = 10, size = 5, prob = 0.2)
```

When we run this code R doesn't show anything. It has just created the vector `x`. To see what `x` is you could just type `x` in the Console window but this can get a bit tedious. Luckily there is a quicker way: just put `()` around the whole line of code and it will print the value of the object assigned within `()`, in this case `x`

```
(x <- rbinom(n = 10, size = 5, prob = 0.2))
```

```
##  [1] 0 1 2 2 3 1 0 0 3 2
```

3 Exercises A

1. Display the built-in object `pi`.
2. Use `seq` to create a vector called `numbers` containing the values 0, 1, 2, 3, 4.
3. What do you think `numbers + 5` would return? Try it.
4. Using what you observed in Question 3, what is the easiest way to create a vector containing the values of $\exp(j)$ for $j = 0, \ldots, 4$.
5. Use R to calculate the sample variance of the numbers 1, 4, 7, 6.2 and 5.1
6. Sample 100 realisations of a random variable with a standard normal distribution. Use the `hist` function to plot a histogram of these realisations with the frequency on the y axis.
7. Plot the histogram in Question 6 but this time showing the probability density on the y axis (look at the help files for `hist`,).

4 Masking existing objects and functions

Whenever you refer to any object, R searches for it in a series of environments starting with the Workspace, where newly created objects are kept. This procedure means it is very easy to mask an R object, that is, to create a new object with the same name as one already present further down the search path. For example consider the constant π. We could easily change its value by mistake.

```
pi
```

```
## [1] 3.141593
```

```
pi <- 100
pi
```

```
## [1] 100
```

If you mask an existing object by mistake then just rename it and remove the masking object

```
my_pi <- 100
rm(pi)
pi
```

```
## [1] 3.141593
```

```
my_pi
```

```
## [1] 100
```

If you mask an existing function you may get a warning message. Otherwise you may notice the masked function behaving in an odd way. The best way to avoid masking is to think carefully about naming objects and functions.

Masking sometimes happens when you add packages to your search path. The package added may contain functions and some of these function names may exist in your search path already. For example when you add the package `dplyr` to your search path you mask some existing functions in your search path.

```
# install dplyr if you haven't already used it
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

R tells you that some functions in your search path are masked by some of the functions in the `dplyr` library. If this happens you should be explicit about the package that the function you want to use is in. For example, we can call the `filter` function from either the `stats` or `dplyr` package.

```
stats::filter()
dplyr::filter()
```

Using the `package::function` syntax also helps the reader in several ways:

- it helps to see where functions are being called from which can be useful if you are sharing code and are using functions not in the standard packages loaded by R at start up;
- it also means that the person you are sharing the code with doesn't need to load the package into the search path.

For example using `dplyr::filter())` means that the person running the code only needs to have installed the `dplyr` package at some time. They don't need to have loaded `dplyr` into the search path in the current session.

5 Vectors

5.1 Numeric vectors

Vectors are probably the most commonly used data structure in R. Essentially, a scalar is just a vector of length 1. Individual elements of a vector can be accessed by using the subscript operator `[ ]`. For example, `x[3]` is the third element of `x`. This notation, and the idea that the basic object is a vector, explains the default way in which R displays numerical results.

Consider:

```
sqrt(3)
```

```
## [1] 1.732051
```

The [1] in the output indicates that the output starts with the first element of a vector. In this case it happens to be the only element. If a vector is too long to fit on one line, the beginning of each line shows the index of the first element on that line. Try

```
seq(1,100)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

The function `length` gives the number of elements in a vector. Vectors can be created using the `c`, `seq` and `rep` functions. Try, for example

```
num1to4  <- seq(1, 4)
length(num1to4)
```

```
## [1] 4
```

```
other_nums <- c(13, 17, 23)
length(other_nums)
```

```
## [1] 3
```

```
decr_seq <- seq(31, 20, by = -4)
length(decr_seq)
```

```
## [1] 3
```

```
more_nums <- c(num1to4, other_nums, decr_seq)
length(more_nums)
```

```
## [1] 10
```

```
more_nums
```

```
##  [1]  1  2  3  4 13 17 23 31 27 23
```

5.2 Creating vectors of integers

To create a vector containing the integers from 1 to 10 we have seen how `seq` can be used as follows

```
seq(1,10)
```

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

But if the sequence is an **integer** sequence there is a shorter way using

```
1:10
```

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

5.3 Vector Arithmetic

Arithmetic can be done on numeric vectors. For example

```
2 * more_nums
```

```
##  [1]  2  4  6  8 26 34 46 62 54 46
```

multiplies each element of `more_nums` by 2. The important point is that the calculation is done *element by element*. This is referred to as *vectorisation*. It makes it easy and efficient to write R code involving vectors. Addition, multiplication and division of vectors of the **same length** is also carried out element by element. Predict what each of the following would do and check it

```
num1to4 +  num1to4^2
```

```
## [1]  2  6 12 20
```

```
num1to4 +  2
```

```
## [1] 3 4 5 6
```

5.4 Recycling in vectors

When vectors are of **different lengths**, the shorter one is extended by repeating its values until it is as long as the longer vector; this is called *re-cycling*. If the length of the longer vector is not a multiple of that of the shorter vector, R will give a warning since the chances are that what's being done was not intended. To see these effects, try

```
length(num1to4)
```

```
## [1] 4
```

```
length(more_nums)
```

```
## [1] 10
```

```
num1to4 + more_nums
```

```
## Warning in num1to4 + more_nums: longer object length is not a multiple of
## shorter object length
```

```
##  [1]  2  4  6  8 14 19 26 35 28 25
```

This produces a warning since the two vectors are of length 10 and 4 (which are not multiples of one another). Similarly

```
num1to4 ^ c(0, 1, 2)
```

```
## Warning in num1to4^c(0, 1, 2): longer object length is not a multiple of
## shorter object length
```

```
## [1] 1 2 9 1
```

gives an error message because the vectors are of length 4 and 3 (which are not multiples of one another). but

```
num1to4 ^ c(0, 1)
```

```
## [1] 1 2 1 4
```

still uses recycling but there is no warning message because the lengths are 4 and 2. Recycling is a powerful tool in R but **unintentional** recycling is the source of many errors in R.

5.5 Creating an empty vector of specified length

In later sections we will need to create a vector of a certain length so that it's values can then be specified sequentially. There are many ways to do this. Perhaps the simplest is to use the `numeric` function which creates a vector of zeroes of the specified length. Try

```
numeric(6)
```

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

But you could use the `rep` function as in

```
rep(x = 0, times = 6)
```

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

5.6 Missing values

The missing value symbol in R is `NA` , and the symbol `NaN` stands for `not a number'. Some examples of when they appear are

```
num1to4[12]
```

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

```
0/0
```

```
## [1] NaN
```

If I am creating an empty vector and plan to give each element a value, I tend to create a vector of NAs using, for example,

```
rep(NA, 6)
```

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

I do this because, after assigning each element a value, the presence of an NA indicates I may have missed giving some elements a value by mistake.

Care needs to be taken when using vectors since R can do unexpected things. Consider creating a vector of 2s of length 5 using

```
repeat_two <- rep(x = 2, times = 5)
```

What do you think would happen if you assigned the 10th value of `repeat_two` (which is of length 5) the value 50?

```
repeat_two[10] <- 50
repeat_two
```

```
##  [1]  2  2  2  2  2 NA NA NA NA 50
```

So R has extended the vector to be of length 10 and filled in the undefined elements as NAs.

#Warning:# Recycling in R is very useful, but also potentially dangerous. Your code might not give any error messages because R has implemented recycling of vectors, but this may not be what you wanted. Be careful with recycling.

5.7 Character vectors

All elements of a vector must be of the same *type* (numeric, character, logical etc). So you cannot have a vector containing numbers and characters. Character strings are entered with double quotation marks

```
clrs <- c("red", "green", "blue")
```

You can easily join character strings together using the `paste` function. It takes an arbitrary number of arguments, turns them into character strings or character vectors, and joins them together element by element into a character vector.

```
paste(c("A", "B"), 1:2, c("X", "Y"))
```

```
## [1] "A 1 X" "B 2 Y"
```

> Try this:
>
> What would the following code produce?
>
> paste(c(“A”, “B”), 1:3, c(“X”, “Y”, “Z”, “W”))

By default `paste` inserts a space between the joined elements; to suppress this, the argument `sep` is used. For example compare with

```
paste(c("A", "B"), 1:2, c("X", "Y"), sep = "")
```

```
## [1] "A1X" "B2Y"
```

5.8 Logical vectors

Vectors may also be made up of the logical (Boolean) values `TRUE` and `FALSE`. Thus

```
a <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
```

is a logical vector.

Logical vectors are most often the result of *comparison operators* applied to the components of numerical vectors. The comparison operators are

- `<` for *less than*
- `>` for *greater than*
- `<=` for * less than or equal to*
- `>=` for *greater than or equal to*
- `==` for *equal to*
- `!=` for *not equal to*

Try

```
b <- c(2, 1, 0, 1, 2)
(comp1 <- b > 1)
```

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

Note the use of `()` around the code to force R to print the results to the screen. In this example the components of the vector `b` were compared to a single number, but more generally the comparison operators may be used to compare one vector with another vector. In this case the rules work element by element and if the vectors are of different lengths, *re-cycling* is carried out.

```
d <- 1:2
(comp2 <- (b < d))
```

```
## Warning in b < d: longer object length is not a multiple of shorter object
## length
```

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

You may find it useful to write down the result of the steps involved in this calculation:

- extension of `d` to length 5 by recycling
- component-wise comparison with `b`

Note that the re-cycling rule explains what has been done above in the evaluation of `b > 1`; the one-component vector `1` on the right hand side has been extended to length 5 before component-wise comparison with `b`

5.9 Boolean operators (and, or, not)

- `&` for *and* (note: A & B is true if and only if both A **and** B are true)
- `|` for *or* (note: A | B is true if A, B or both are true)
- `!` for *not* (note: !T is F and !F is T)

These options can be applied to logical vectors. Again they work element-by-element and use recycling as necessary.

Try this:

Predict what each line of the following code will produce and verify the result in R

```
a & comp1
!a | comp1
comp1 &  (!a | comp2)
```

Given a set of (one or more) logical vectors, how can we determine if any of their elements true? The function `any` returns a single value giving the answer

```
any(a)
```

```
## [1] TRUE
```

The complementary function `all` tests whether *all* elements are true

```
all(a)
```

```
## [1] FALSE
```

A useful feature of logical vectors is that they may be used in ordinary arithmetic. When they appear in an arithmetical expression, `TRUE` becomes `1` and `FALSE` becomes `0` . Thus the `sum` function (which normally adds together the components of a vector) when applied to the logical vector `a` gives

```
a
```

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

```
sum(a)
```

```
## [1] 3
```

the number of true `TRUE` elements of `a` .

5.10 Subsetting

An individual element of a vector may be accessed using the subset operator `[ ]` ;for example `b[3]` is the third component of the vector `b`

```
b <- c(1, -3, 6, 9, -13, 67)
b[3]
```

```
## [1] 6
```

More generally, we can extract a subset of a vector `b` by `b[subsets]` where `subsets` has one of the following forms

1. positive integers; for example

```
b[2:5] # 2nd to 5th elements of b
```

```
## [1]  -3   6   9 -13
```

```
b[c(1, 3, 5)] # first, third and fifth elements of b
```

```
## [1]   1   6 -13
```

2. negative integers; for example

```
b[c(-1, -3)] # selects all elements of b except the first and third
```

```
## [1]  -3   9 -13  67
```

3. logical; for example

```
a <- c(TRUE,  FALSE,  TRUE,  FALSE,  TRUE)
y <- c(0, 0.5, 2, 3, 0, 0.8)
b[a] # elements of b corresponding to the `TRUE` elements of a
```

```
## [1]   1   6 -13  67
```

```
b[b > 0] # selects all the strictly positive elements of b
```

```
## [1]  1  6  9 67
```

```
b[y < 1] # selects elements of b corresponding to elements of y that are less than 1,
```

```
## [1]   1  -3 -13  67
```

We can combine two logical vectors

```
b[(b > 0) & (y < 1)]
```

```
## [1]  1 67
```

selects elements of `b` for which both conditions are true. Note that the `()` here are not strictly necessary

```
b[b > 0 & y < 1]
```

```
## [1]  1 67
```

but brackets can help to see the order of operations more clearly. This process of extracting specific elements of a vector is called subsetting.

6 Exercises B

1. Predict what each of the following lines of code will produce, whether any would give warnings, and then check them in R

- `1:5 * 2:6`
- `1:5 * 2:3`

- `c(2, 5, 8, 3, 5, 9) > 3.5`
- `c(2, 5, 8, 3, 5, 9) > c(3.5, 5.4)`
- `c(2, 5, 8, 3, 5, 9)[c(2, 5, 8, 3, 5, 9) > 3.5]`
- `c(2, 5, 8, 3, 5)[c(2, 5) > 3.5]`
- `seq(1, 10)[1:10 > 3.4]`

2. Predict what `rep(1:4, 1:4)` will produce, then check the result.
3. The built-in vectors `letters` and `LETTERS` consist of the lower and upper-case letters of the alphabet respectively. Use them and `seq` and `rep` as appropriate to produce the following

- `"a" "b" "a" "b" "a" "b" "a" "b"`
- `"A" "A" "B" "B" "C" "C"`
- `2 4 6 8 2 4 6 8`

4.

- Create the vector `x` consisting of the numbers 17.8, 0.8, 1.2, 11, 7, 0.006, 6.9, 5.3
- Find the mean of `x` and subtract it from each element.
- Display the square roots of the elements of x.
- Display those elements of x that are larger than their square roots (by using subsetting).
- Check the help file for the function `round` and use `round` to display the square roots of the elements rounded to 2 decimal places.
- How much do the squares of the rounded roots differ from the original numbers?

5. Create two related vectors as follows

```
(x <- seq(-1, 1, by = 0.1))
```

```
##  [1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3
## [15]  0.4  0.5  0.6  0.7  0.8  0.9  1.0
```

```
(y <- x^2)
```

```
##  [1] 1.00 0.81 0.64 0.49 0.36 0.25 0.16 0.09 0.04 0.01 0.00 0.01 0.04 0.09
## [15] 0.16 0.25 0.36 0.49 0.64 0.81 1.00
```

Use subsetting ideas to select:

- the first and last elements in `y` ;
- elements in `y` that are bigger than the median of `y` ;
- elements in `y` corresponding to elements in `x` that are less than the mean of `x` .

6. The function `is.na` applied to a vector returns `TRUE` for each element that is `NA` and `FALSE` otherwise. Run each line of the following code to understand what it does.

```
integer_vector <- 1:10
to_replace <- sample(1:10, 5)
some_missing <- integer_vector
some_missing[to_replace] <- NA
```

Use `is.na` in conjunction with subsetting to remove the NAs from the vector `some_missing`

7. Without running the code, say what each of the following return?

- `!c(2, 5, 8, 3, 5, 9) < 5.3`
- `seq(1:6)[!c(2, 5, 8, 3, 5, 9) < 5.3]`
- `seq(1:10)[!c(2, 5, 8, 3, 5, 9) < 5.3]`
- `seq(1, 10)[1:10 > 3.4 | 1:10 < 1.1]`
- `seq(1, 10)[!(1:10 > 3.4 | 1:10 < 1.1)]`
- `seq(1, 10)[1:10 > 3.4 & 1:10 < 7.5]`
- `seq(-5,8)[!seq(12, 27, by = 3)[c(-4,-8)] >= 20]`

Check them in R

8.

- How could you use `is.na` in conjunction with `any` to construct a composite command to detect the presence of any missing values in a given vector `x`?
- How could you obtain the number of missing values in `x`?

7 Higher dimensional objects

7.1 Matrices

The `matrix` function arranges values in a matrix

```
(m <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3))
```

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

by default filling down columns. (An extra argument, `byrow = T`, overrides this, causing values to be filled along rows.) If the vector of values is too short, it is re-cycled.

```
matrix(c(1, 2, 3, 4), nrow = 2, ncol = 3)
```

```
## Warning in matrix(c(1, 2, 3, 4), nrow = 2, ncol = 3): data length [4] is
## not a sub-multiple or multiple of the number of columns [3]
```

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

You can also create a matrix from a collection of vectors by using `rbind` or `cbind` to bind rows or columns together. Thus each of the following produce the same matrix `m` as above

```
rbind(c(1, 3, 5), c(2, 4, 6))
```

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

```
cbind(c(1, 2), c(3, 4), c(5, 6))
```

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

For certain frequently-needed special kinds of matrices there are some purpose-built construction functions. The function `diag` , for example, constructs diagonal matrices. Thus

```
diag(1, nrow = 4)
```

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

sets up the 4×4 unit matrix.

7.2 Subsetting matrices

An individual element of a matrix is selected by specifying its row and column.

```
(x <- matrix(seq(1,25), nrow=5))
```

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

```
x[2, 5]                # selects element in row 2, column 5 of matrix x
```

```
## [1] 22
```

Other examples of subsetting are given below

```
x[1:3, ]               # selects the first three rows
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
```

```
x[, 2]                 # selects the second column
```

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

```
x[c(3, 5), c(1, 4)]       # selects elements in rows 3 and 5 and cols 1 and 4
```

```
##      [,1] [,2]
## [1,]    3   18
## [2,]    5   20
```

```
x[c(1, 4), -2]         # rows 1 and 4 of all the columns except column 2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   11   16   21
## [2,]    4   14   19   24
```

```
z <- c(0, 1, -5, -6, 7)
x[z > 0, ]  # all rows of x where elements of z are positive
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    7   12   17   22
## [2,]    5   10   15   20   25
```

7.3 Preserving dimensions when subsetting matrices

When you subset a matrix R by a single row or column (eq `x[3,]` or `x[, 2]`) R will return a vector with no dimension. If you want to keep the dimension attribute then there is an optional argument called `drop` that allows you to do this. For example

```
x <- matrix(seq(1,25), nrow=5)
dim(x[1,])
```

```
## NULL
```

has no dimension (it is a vector), whereas

```
dim(x[1, , drop = FALSE])
```

```
## [1] 1 5
```

has dimension 1×5 (i.e. is a row vector)

7.4 Arrays

Arrays are multi-way generalizations of matrices. Whereas matrices have two dimensions, arrays can have one, two, three or more. They are created by the `array` function, which has an argument `dim` which specifies the number of dimensions and the range of each. For example

```
f <- array(1:24, dim = c(3, 4, 2))
```

creates a three-way array with three rows, four columns and two layers. subsetting works in the same way as for matrices so you could find the element in the third row, second column and first layer using

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

```
## [1] 6
```

7.5 Matrix multuplication

To multiply two matrices together use the `%*%` operator. For example the following code multiplies a 2 × 3 matrix by a 3 × 2 matrix to give a 2 × 2 matrix

```
(matrix_1 <- matrix(1:6, nrow = 2))
```

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

```
(matrix_2 <- matrix(1:6, nrow = 3))
```

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

```
(product_matrix <- matrix_1 %*% matrix_2)
```

```
##      [,1] [,2]
## [1,]   22   49
## [2,]   28   64
```

```
dim(product_matrix)
```

```
## [1] 2 2
```

What happens if we mistakenly use the `*` operator rather than `%*%`? Consider multiplying two 2 × 3 matrices together. These are not conformable matrices as the error message shows (you need to copy and paste the code to see the error message)

```
matrix_1 <- matrix(1:6, nrow = 2)
matrix_2 <- matrix(1:6, nrow = 2)
matrix_1 %*% matrix_2
```

So we correctly get an error message in this case. But what about the following:

```
(matrix_1 <- matrix(1:6, nrow = 2))
```

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

```
(matrix_2 <- matrix(1:6, nrow = 2))
```

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

```
matrix_1 * matrix_2
```

```
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

So why do we **not** get an error message using `*` instead of `%*%`? The reason is that `*` will multiply element by element if the two matrices are of identical dimension. It will return a matrix of the same dimension where each element is the product of the elements of the two individual matrices (e.g. the element in row 1, column 3 is 25, which is 5 times 5).

So don't forget to use `%*%` when multiplying matrices, i.e. you may get code that runs but doesn't do the right thing. It is always a good idea to check the dimensions (using `dim`) of your matrices at each stage.

8 Lists

A list is a flexible object that can store collections of objects of different types and lengths. Lists are created with the `list` function, which can assign names to components. As a simple example, a patient record might be created by

```
(p_lst <- list(patient = "John", spouse = "Yiyang", children = 2, child_ages = c(5,10
),
               diabetic = TRUE))
```

```
## $patient
## [1] "John"
##
## $spouse
## [1] "Yiyang"
##
## $children
## [1] 2
##
## $child_ages
## [1]  5 10
##
## $diabetic
## [1] TRUE
```

The object `p_lst` is a list with

- two character components: patient and spouse
- two numerical components: children (of length 1) and child_ages (of length 2)
- one logical component: diabetic

There are two ways to access the components of a list:

- by name, using the special symbol `$`
- by subsetting, using *double* square brackets `[[ ]]`.

So these two commands produce the same output

```
p_lst$patient
```

```
## [1] "John"
```

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

```
## [1] "John"
```

> Try this:
>
> What would you expect each line of the following to return?

```
p_lst[[2]]
p_lst$child_ages
```

As for any object, you can use the `str` function to find out the structure of an object

```
str(p_lst)
```

```
## List of 5
##  $ patient   : chr "John"
##  $ spouse    : chr "Yiyang"
##  $ children  : num 2
##  $ child_ages: num [1:2] 5 10
##  $ diabetic  : logi TRUE
```

Alternatively the `names` function returns the names of the components of the objects

```
names(p_lst)
```

```
## [1] "patient"    "spouse"     "children"   "child_ages" "diabetic"
```

Individual elements of a list **component** can be addressed by standard subsetting

```
p_lst$child_ages[2]
```

```
## [1] 10
```

This is the second element of the vector of child ages.

8.1 `[ ]` versus `[[ ]]` (a common mistake with lists)

The most common error when subsetting to extract list elements is to use `[]` instead of `[[]]`. So what is the difference?

```
str(p_lst[4])
```

```
## List of 1
##  $ child_ages: num [1:2] 5 10
```

```
str(p_lst[[4]])
```

```
##  num [1:2] 5 10
```

Notice that

- `p_lst[4]` returns a length 1 **list** containing the length 2 numeric vector `child_ages`. It hasn't extracted `child ages` from the list, it has created a new list with `child ages` as the only element.
- `p_lst[[4]]` correctly extracts the length 2 numeric vector (not a list).

9 Exercises C

1. Let

$$A = \begin{pmatrix} 1 & 2 \\ -3 & 1 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 3 & 5 & 0 \\ 4 & -2 & 3 \end{pmatrix}$$

   Calculate $A \times B$ on paper and check it in R.

2. Predict what the following return and check them in R

```
A <- matrix(c(1, -3, 2, 1), nrow = 2)
A[1,]
```

```
A[1,] + 5
```

```
A[1, , drop = FALSE] + 5
dim(A[1, , drop = FALSE] + 5)
```

3. Predict what the following returns and check it in R

```
A <- matrix(c(1, -3, 2, 1, 4, 2), byrow = TRUE, nrow = 2)
A %*% A
A * A
```

3. Consider the three-dimensional array

```
f <- array(1:24, dim = c(3, 4, 2))
```

Use the help function to determine how R fills in elements of an `array`. Use this information to predict what the following would return

```
f[1, 1, 2]
```

```
f[14, 1, 2]
```

```
f[1:2, 2:4, 1]
```

4. Consider a list defined by

```
some_list <- list(1:4, matrix(1:4, nrow = 2), letters)
```

Predict what the following return and check the output in R

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

```
some_list[[3]][5]
```

```
some_list[[2]][2,2]
```

```
dim(some_list[[2]][2,])
```

```
length(some_list[[2]][2,])
```

```
dim(some_list[[2]][2, , drop = F])
```

```
some_list[1]
```

```
some_list[1:2]
```

10 References