Contents

An introduction to the use of R

R is often used by bioinformaticians and biologists to perform analyses that are provided by packages, and this is often the first experience of R that research students have. The documentation and tutorials for those packages, quite rea-

sonably, are concerned primarily with how the user can use the functions and data structures provided by the package. This is using `R` as a fancy *shell*[1], and learning how to do this doesn't teach you how to use `R`. In this tutorial we shall be concerned only with *basic* `R` commands and how these can be combined to achieve more complex tasks. This is essentially programming.

Download this file and others and create a sane directory structure

In Canvas, Modules, "Session 02", you should find this file, "R_introduction.md" , a pdf file ("R_introduction.pdf") and two data files ("dr_exons.tsv", "dr_genes.tsv"). Download these to a suitable directory on your computer. That might be something like "~/Documents/BI229F/R/session_2", where "~"is shorthand for your home directory. On Windows that will be something like "C:\Users\<your_id>", on Macs more like "/home/".

Having done that, you can now proceed to open this file in `RStudio`; you can probably do that by finding the file in the file explorer (Windows) or `Finder` (Mac) and right-clicking on it and selecting "Open with". If you are using Linux, then I assume you know how to do this.

You may also wish to open the accompanying pdf as it might be somewhat easier to read than this.

Having opened `RStudio`, enter the console (click on it!, or better yet use the keyboard shortcut, "Ctl-1", or "Ctl-2") and type the following commands:

```
## to check our current working directory
getwd()

## to see what files are present in your current working directory
list.files()
```

You current working directory should be the same one that you downloaded the files to, and you should see the four files you downloaded. You may also see this in the file browser in the right hand bottom corner. But better to ask `R` itself.

If the directory is not what you expect or you are unable to see the files you downloaded then let me know from the beginning. Do not wait until the end of the day or the end of the course!

You may now start to play with R.

[1] A shell is a program that is used to start other programs; it is your interface to your computer and can be graphical (like a Mac or Windows desktop), but it can also be text based (you type commands rather than clicking on icons). Both have their own strengths, but text based ones (command line) are usually more efficient for complex tasks (though the meaning of *complex* can of course be a point of argument).

Data types and structures in R

We use R to analyse data. This means that we need some way to represent data in R. We have several types of data:

1. Numbers, which can be either integers (whole numbers) and continuous numbers (floats).
2. Text, which is expressed as sequences of characters (represented by numbers). We usually refer to pieces of text as *strings*.
3. Logical values. These can take one of two values: TRUE or FALSE.
4. Special values. R also has NULL and NA values. They are both used when there is no appropriate or reasonable value to use. They have somewhat different behaviour but are both used for missing values somewhat interchangably.

Numbers

Entering numbers

In R you can express numbers in a number of ways. Try the following by executing the following commands in R. You can either type the commands directly into the console panel of RStudio or into the editor followed by Ctrl-enter or selecting the text and doing something (exactly what seems to vary) to cause RStudio to send the commands to the R console.

```
## Note that anything following a # character is a comment
## so these two lines are comments
##
## YOU DO NOT NEED TO TYPE any comments
## BUT YOU SHOULD READ THEM!

## the usual decimal way:

x <- 10
y <- 3.14159

## this assigned the respective values to the variables
## x and y. We can inspect their values by simply typing
## their names and pressing enter;
x

y

## We can also combine them using arithmetic operators:
x + y

x * y
```

```
x / y

x^2 * y

## and we can assign the resulting values to new variables:
z <- x^2 * y

## We can use exponential form If you want to write a really big number
## like Avogadros constant:
##
## xxeyy = xx times 10 to the power y

ac <- 6.022045e23
ac
## which is how R reports big numbers anyway

## We can also use an expression that means the same thing:
ac <- 6.022045 * 10^23
```

Hexadecimal numbers

In `R` we can also enter numbers in hexadecimal. You probably don't have any reason to do so for now, and you will probably find hexadecimal numbers a bit weird to begin with. However, you may come across them in the future and recognising them and how they work can be useful.

Hexadecimal numbers work like decimal; but instead of having 10 digits (0-9) we have 16: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The letters A-F indicate the values 10-15.

Remember that a decimal number indicated by two digits

$$ab$$

takes the value:

$$a \times 10 + b$$

For a two digit hexadecimal number ab the value indicated is simply:

$$a \times 16 + b$$

Hexadecimal numbers are usually denoted by putting the characters `0x` in front of the actual number. We say that they are *prefixed* by `0x`.

```
## We can also use hexadecimal notation. Try
0xA
0xB


## what do you think happens here?
0xG

## and what about here?
0xF
0xFF
0xFFFF

## and see if you can work out what is going on. The square root of
## a related number might make sense:
0xF + 1
sqrt(0xF + 1)
sqrt(0xFF + 1)
sqrt(0xFFFF + 1)

## the (base 2) log of these numbers may make even more sense:
## eg.

log2(0xF+1)
log2(0xFF+1)

## But do not despair if the above does not make any sense what-so-ever
## Just continue to the next session.
```

Entering numbers exercise

Please do try these exercises. You should be able to do at least 1 and 2; if you have trouble with these then ask someone for help and make sure that you can do them. Exercises 3-5 are more difficult, and it is fine if you are unable to work out the answers.

1. Write an expression that:
   a. Assigns numeric values (you choose which) to the variables `a` and `b`
   b. Calculates their product and assigns the value to a variable `p`. This should require about 3 lines of code. Change the values of `a` and `b` and see what happens.
2. Write an expression that assigns the value of Plancks constant (6.26×10^{-34}) to the variable `pc`.
3. What are the hexadecimal representations of 2^4 and 2^8?
4. In hexadecimal, what is the result of `2 * 0x8`?
5. Assign the value $2^{32} - 1$ to a variable using hexadecimal notation.

Types of numbers

Numbers can be integral (integers) or continuous (referred to as *floating point*, or *double* in R). Although you can mostly ignore this when using R, it is useful to have some idea of how they behave differently.

```
x <- 10
## adding an L at the end of the number tells R you want this number to be
## an integer
y <- 10L

## we can ask what class a variable is:
class(x)
class(y)

## you should notice a difference:

## We can also ask directly if a variable is a particular class:
is.numeric(x)
is.numeric(y) ## ??

is.integer(x)
is.integer(y)

## floating point numbers are represented as double precision floating
## point values (64 bit). Because of this you can also ask if a number is
## a double..
is.double(x)
is.double(y)

## what happens if we combine the two?
is.integer(x + y)
is.double(x + y)

## combine an integer with a float get a float

## but combine an integer with an integer and get
## an integer
is.integer(y + 2L)

## but this is a bit inconsistent:
## note that;
## multiply: *
## divide:   /
is.integer(y * 2L)
is.integer(y / 2L)
```

```r
is.integer(2L^2L)

## That is; any division of integers will give a floating point.
## Unless we use the integer division operator:
is.integer( 2L %/% 2L )
## TRUE
is.integer( 2L %/% 2 )
## FALSE
is.integer( 3L %/% 2L )
## TRUE ...
```

As I hope you have inferred R has a strong tendency to convert integer numbers to floating point numbers. Mostly this doesn't matter, though you should be aware that the floating point numbers in R take up twice as much memory as the integers. This can make a difference if you are dealing with a large number of values (which we often do).

Number ranges

Each number handled by R is encoded using a fixed number of *bits* (32 for integers and 64 for floats). This means that there is a finite set of numbers that can be expressed. Trying to express a number outside of this set will result in an error:

```r
## for integers:
x <- 256L
## then multiply x by itself and assign to x
x <- x * x
x

## always good to check one's assumptions
is.integer(x)
## multiply again by 256
x <- x * 256L
x  ## about 16.7 million

## and
x <- x * 256L
x

## something weird happened.
is.na(x)

## we can also use the equality operator, consider:
8 == 8
## this is TRUE, eight is equal to eight
```

```r
8 == 9
## this is FALSE, eight is not equal to nine

## but
x == x
## x SHOULD be equal to itself. But this is a special property
## of NA


## But if we do:
x <- 256L
x <- x * x * x
x
is.integer(x)

x * 256L
## as before, this is a problem
## it is referred to as an integer overflow, because we are
## computing a value that is too large to fit in the integer type
## we are using

x * 256
is.integer(x * 256)
## multiplying by a double made x a double...
```

For now you can ignore the reasons for the above behaviour. We will come back to it at some future point. However, you should remember that integers in `R` can only hold values between about `-2e9` to `+2e9`[2].

This isn't *usually* much of a problem in your own work as numeric values almost always end up getting converted to *64 bit floating point* values. However, there are various packages that may fail when dealing with data sets that have more than `2^31` values and it is useful to recognise the cause of such failures.

Floating point values can hold a much larger range of numbers, but you need to understand that the difference between adjacent numbers increases with the size of the number. This will be demonstrated further down.

Number ranges exercise

These questions are all somewhat advanced but you should nevertheless try to think about how you can approach the problems and discuss with those around you. Let me know if you come up with a solution, or if you get stuck.

1. The largest integer that `R` can encode is about 2×10^9. Why do you think this might be the case? Remember that all data is expressed as a series of

[2]The actual range is from -2^31 to +2^31 - 1; or at least it would be if `R` didn't use the smallest value for some special usecase.

0s and 1s (each one called a bit) and that integers in R are encoded using 32 bits. Consider how many numbers can be encoded by a single bit (2: 0 and 1), two bits (4: 00, 01, 10, and 11) and so on.
2. How could you use R to estimate the largest floating point value that can be expressed by R. Consider using the power notation (eg. 10^10 and so on).
3. How can you test if the difference between two adjacent floating point numbers is larger than 1?

Text (strings)

Entering strings

Pieces of text are usually called *strings* in computer science. To define a string in R you need to use quotation marks to indicate the beginning and the end of the string. Both single (') and double (") quotes can be used as long as they are paired correctly.

```
x <- "hello world"
y <- 'goodbye'

## DON'T just enter the above, but look at the contents of
## x and y afterwards. There IS something that you should note.

## use single quotes if you want to use double quotes
## inside the text:
z <- 'he said, "That is not a good idea"'
## but note how R prints it:
z
print(z)

## you can ouput it as you've written it by using the cat
## function
cat(z)

## but in this case it is better to do:
cat(z, '\n')
## or indeed
cat(z, "\n")

## \n denotes an end of line character and tells R to move the cursor
## back to the beginning of the line.

## if you need to use both single and double quotes inside a string
## you will need to escape the internal quote marks>

z <- 'he said, "That\'s not a good idea"'
```

Escaping characters and control characters

In the above section I stated that you needed to *escape* the internal quote mark; that is because the quote character in "That's"has a special meaning; here it tells R where the end of the string is. To escape the special meaning of a character we usually use the backslash character (\). To enter an actual backslash into the text we simply use two backslash characters (\\), where the first one escapes the special meaning of \.

We also use backslashes for different types of control characters. There are two that you should now about:

1. "\n": The newline character; this causes the cursor to go to the beginning of the next line.
2. "\t": The tab character (usually interepreted as variable width space used to line up columns). The tab character is often used to delimit columns in tabular data and you may need to specify this to functions that read such data in.

Note the difference in the behaviour between `cat(z)` and `cat(z, '\n')` in the above code.

If you lucky, the following may do something interesting on your computer:

```
cat('\a')
```

Try it and see what happens. On my computer it doesn't do anything, but that may be because I disabled this functionality at some point in the past.

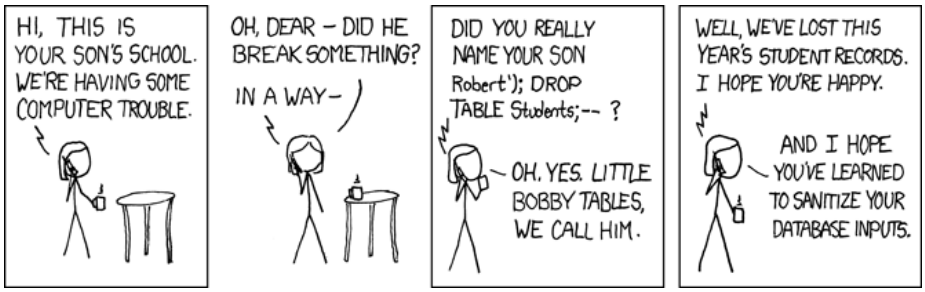

Figure 1: The exploits of a mom (https://xkcd.com/327/), demonstrating the importance of escaping control characters. Note the surreptitious single quote!

Character encoding

Text encoding is actually somewhat complicated given that there are a large number of different character sets in use across the world. We don't usually have to care too much about this as R generally does the correct thing. R does have a function that returns the byte values used to encode the text. This can

be used simply to investigate the encoding or to extract values from specially encoded strings (eg. quality scores in fastq sequence data).

```
x <- "Hello"
x

## charToRaw extract the exact bytes used to represent
## the string.
y <- charToRaw(x)
y

## nchar(x) gives the number of characters in
## a string
nchar(x)
length(y)

## You may need to run this from the markdown file
## or copy and paste.

x <- " クローズアップ科学道"
x

nchar(x)

y <- charToRaw(x)
y
length(y)

## note that these are hexadecimal values:
as.integer(y)

## R also has
z <- utf8ToInt(x)
z
length(z)

## and you can check if this makes more sense
nchar(x) == length(z)
```

String operations

With numbers we can perform all sorts of mathematical operations; however we can't reasonably add two strings together and expect a sane result.[3] But we

[3] You will not be suprised that there are several programming languages where you can do `"string1" + "string2"`. But obviously in these cases the meaning of the `+` sign isn't numerical addition but something else.

can compare strings to each other and we can search for patterns within strings. You can try the following:

```
s1 <- "abcde"
s2 <- "bcde"

## You should be able to guess the result of:
s1 == s2

## and maybe also
s1 > s2

## we can also split a string into several strings:

s3 <- "two words"
s4 <- strsplit( s3, split=" " )
## look at s4
s4

## the result of the strsplit operation is a list; this is a special type of
## data structure that we will cover at a later stage.

## You can also search for patterns in strings:

grep( "wo ", s3 )
grep( "wo ", s1 )
## grep returns the index of the strings matching the pattern. This won't make
## sense until we introduce vectors later.

grepl( "wo ", s3 )
grepl( "wo ", s1 )

## l stands for logical; logical vectors contain TRUE or FALSE values and
## are very useful.
```

String exercise

1. Create an `R` string object (a `character vector` in `R`) containing the following phrase (including the quotation marks):
   "Let's get out of here", she said
2. Copy the following code (from the `.md` file) and then run it in `R`.

   ```
   a <- "a word"
   b <- "a word"
   a == b
   ```

   To work out why that happened use `charToRaw` and `utf8ToInt` on a and b.

3. Assign a variable containing a new-line character ("\n"). Print it to the terminal so that it prints the newline rather than the newline character.

Data structures

So far we have looked at assigning individual values to single variables; clearly this isn't a very efficient way of dealing with large sets and sequences of numbers or other data types. In fact `R` doesn't really have data types that hold single values. Recall that if you type `10` in the console and hit enter, R prints out: `[1] 10`. That `[1]` indicates that the result of the previous operation is a *vector* of length 1 (the `[1]` doesn't give the length, but the index of the first value on each line). A *vector* is an ordered set of values of the same date type whose values can be accessed and set by their position in the vector. All[4] data structures in `R` are represented by some form of a vector, and pretty much all functions return some form of a vector. `R` also has `list`, `matrix`, `array` and `dataframe` types of data structures; these are all special kinds of vectors with special attributes.

Vectors

Creating and subsetting vectors There are a number of functions that can be used to specifically to create vectors of different types:

```
## the concatenate function. Makes a vector of its arguments:
c(1, 3, 2)

## you can use with any types of objects
c("one", "two", "three")

## but note what happens here:
c(1, "two", 3)

## The quotation marks indicate that vector contains strings:
class( c(1, "two", 3) )

## class( c(1, "two", 3) )
## is what's known as a 'nested' function call.
## In a nested function call, the 'argument' to the call
## is the value returned by a call to another function.
## the above is equivalent to:

## assign the return value of c() to a temporary variable:
tmp <- c(1, "two", 3)
## ask what the class of the variable is:
```

[4] It's not a good idea to use the words 'all' or 'none'; you could argue that functions are data structures that are not vectors for example. But you could also argue that functions are different from data.

```r
class( tmp )
## remove the variable from our workspace:
rm( tmp )

## nested function calls are convenient, but can be difficult
## to read if too complex.

## You can try and coerce such a vector to numeric values:
as.numeric( c(1, "two", 3) )
## but not entirely succesfully.

## however consider the following:
class( c(1, 2e10, 0xA5) )
class( c("1", "2e10", "0xA5") )
as.numeric( c("1", "2e10", "0xA5") )

## the as.numeric onversion follows the same rules that we have for entering
## numbers in the console.


## To access an element we use its position in the vector and the
## subsetting operator []

x <- c(1, 3, 10, 4, 2)
## Make sure that you understand what happens
## when you do the following!
x[1]
x[3]

## we can check the length of the vector using length()
length(x)
## and this shouldn't work
x[ length(x) + 1]

## but this might
x[ length(x) - 1]

## we can subset several values using a range notation:
x[1:2]
x[3:4]

## The :
## character is an operator that creates a vector of values:
1:10
is.integer(1:10)
## reverse also works
```

```r
10:1

## So when we do:
x[1:3]

## we are using a vector to subset (extract specific parts of) x
## we can do this explicitly as well:
y <- c(1,2,3)
x[ y ]

## We can extract the same value more than once:
y <- c(1,3,1,3,1,3,1,3)
x[y]
## This is really rather useful.

## R also has a function for repeating an element or vector
## its called rep:

rep( 2, 10 )
rep( c(2,3), 10 )

## so we can also do:
y <- rep( c(2,3), 10 )
x[y]

## or ..
x[ rep(c(2,3), 10) ]
```

Vector operations Many (most) operators and functions do not expect a single value as an argument but will happily do things to every element of a vector. For example we can do arithmetic with whole vectors using single expressions.

```r
## define a vector v1:
v1 <- 1:10
## look at the vector
v1

## we can add a single value to every element of v1:
v1 + 10

## we can also add a vector to a vector:
v2 <- 1:3

## then consider what happens now:
v1 + v2
```

```
## there was a Warning message, but the operation was carried out
## as there was no error.
## Try to understand why the warning was printed to your screen.

## To increase the value of v1 you can obviously do:

v1 <- v1 + v2

v1
```

Named vectors The elements of vectors can also be named; this allows you to use text labels to extract and set specific elements of a vector.

```
## you can use the c function to create a named vector:

x <- c('one'=1, 'two'=2, 'five'=4)
## elements can be accessed using the label or their position (index)
x[1]
x['five']

## as before we can use a vector of labels:
x[ c('one', 'five') ]

## but this doesn't work for what should be obvious reasons:
x[ c('one', 2) ]

## you get the 'NA' value, because R looks for an entry with the label
## '2', since that doesn't exist it returns an NA

## You can also use the names() function to set the names of a vector
## after it's been created:

x <- 1:10
names(x)  ## no names
## And one of R's strange behaviours:
names(x) <- c('a', 'b', 'c', 'd', 'e', 'f')

x ## some have names, but some are missing
names(x)  ## these have NA values

## To set the name of a specific element we can do:
names(x)[7] <- 'seven'
x

## but this is not a good idea
```

```r
names(x)[7:8] <- 'seven'
x

x['seven']
## gives you only the first one..
```

Subsetting a vector using logical vectors Logical values (`TRUE` or `FALSE`) can also be used to extract values from a vector. Most of the time you should use a vector of logical values that has the same length as the vector you are subsetting from.

```r
x <- 1:3
## specify a logical vector using c
y1 <- c(TRUE, FALSE, FALSE)
y2 <- c(FALSE, TRUE, TRUE)

x[y1]
x[y2]

## your logical vector would normally be the result of some logical
## operation. Eg:

x <- 1:10
y <- x > 5

## look at the value of y
y
## make sure that you understand how y got the values it has
x[y]

## You could also do something like:
x[ c(TRUE, FALSE) ]
x[ c(FALSE, TRUE) ]
x[ c(TRUE, TRUE, FALSE) ]
## you should be able to work out what happens there.

## The power of the logical vectors is that you can
## combine them using boolean operations

## consider
z <- x < 8
z

## combine using AND
y & z
x[ y & z ]
```

```
## combine using OR
y | z
x[ y | z ]

## we can invert the value using !
y
!y

## and combine
!y & z
!y | z

x[ !y & z ]
x[ !y | z ]
```

Setting the values of vector elements In the preceding sections you have used indices (positions), names and logical vectors to access the values of specific vector elements. You can use all of these operations combined with the assignment operator (`<-`) to set the values of the accessed elements.

```
x <- 1:10

x[1] <- 100
x
x[5:6] <- 2
x

x[5:6] <- c(-10, 3)
x

## but
x[5] <- c(-10, 3)
## read the error message!!

## with logical vectors
## make sure that no values are bigger than 6
y <- x > 6
y
x[y] <- 6
x

## and remove all negative values:
x[ x < 0 ] <- 0
```

Vector exercises

1. Create a vector containing the numbers 3 to 13. Then:
   a. Modify the vector so that it contains the numbers 0 to 10.
   b. Create another vector that contains the 4th, 5th, 6th and 1st element of this vector (in that order).
   c. Create a third vector that contains all numbers in the first vector that are larger than 3 but smaller than 8.
2. Create a vector containing the numbers 1 to a 100. Then:
   a. Use a vector of logical values (`TRUE` or `FALSE`) to extract all odd or even numbers.
   b. Then use the same approach to extract all values that are even multiples of 5 or some other small number.
   c. (Advanced) Use the modulus operator to achieve the same results as in (a) and (b) above. The modulus operator `%%` returns the remainder of the division of its two operands.

Matrices

The matrix is one of the most commonly used data structures in `R`. This is related to the origin of `R` as a language created by mathematicians in order to solve systems of equations. Matrices are essentially tables of values of all the same datatype.

To make a matrix with `matrix()`

```
## To create a matrix we can use the matrix function:

x  <- matrix(1:12, nrow=4)
## The first argument here is the 'data' that will fill the matrix
## we specify a vector containing the numbers 1 to 12. The second argument
## nrow : this is simply how many columns the matrix should have
x

## note how the numbers are filled in by column with the first column
## having the first four numbers.

## we can optionally fill by row:
y  <- matrix(1:12, nrow=4, byrow=TRUE)
## and do look at it before going any further!

## to get the dimensions of a matrix use the dim() function
dim( x )

## you can also get the number of rows and columns using the
## nrow() and ncol() functions.
nrow(x)
```

```
ncol(x)

## note that dim() returns a vector (like most things) and that you
## can directly subset that vector without assigning it to a variable:
dim(x)
dim(x)[1]
dim(x)[2]
## This is true for any function that returns a vector and can be
## quite useful.

## If you are looking at tutorials and examples you may often see
## the use of the t() function without any explanation. If you
## don't know what something does, simply try it and see if you
## can work it out..

x
t(x)

y
t(y)

## of course, note:
dim(x)
dim( t(x) )

## you can also try:
t(x) == matrix(1:12, ncol=4, byrow=TRUE)
## remember that == is the equality operator...
## so this comparison returns a matrix of logical values.;

## Note that you don't have to specify any data:
matrix(nrow=4, ncol=3)

## and the data does specified does not need to have
## the same length as the matrix:

## these matrices have a useful property
## (can you see what it is?)
matrix(1:4, nrow=4, ncol=3)
matrix(1:3, nrow=4, ncol=3, byrow=TRUE)
```

In the above examples I have called the `matrix()` function with *named* arguments. Functions can be called with or without names, but if names are not specified then the arguments have to be in the correct order (otherwise there would be no way for `R` to know which one is supposed to refer to which argument. You may also have noticed that I have called `matrix()` with differing numbers of

arguments. Function arguments can have default values, and when they do, you do not need to specify those arguments to the function. In those cases it's better to use named arguments as it is not always obvious what the order should be. It can also be argued that it's better to *always* use named arguments as it makes your code easier to read and reduces the number of mistakes.

Accessing elements of matrices Elements of matrices are accessed in pretty much the same way as vectors; however, you (usually) specify both the row and column position of the values that you wish to access.

```
x  <- matrix(1:20, nrow=5)
x

## to get the value in the second row third column:
x[2, 3]

## to get the full second row:
x[2, ]

## Note the difference of:
x[2, , drop=FALSE]
x[2, , drop=TRUE]

## compare the output of:
class(x[2, , drop=FALSE])
class(x[2, , drop=TRUE])
## That is something that has and continues to cause
## me a lot of trouble.

## we can specify a range of rows or columns as well
x[4:5, 2:3]

## you can also specify a set of points using
## the following expression
x[ cbind(1:4, 1:4) ]

## look at the original vector to work out what that did:
x

## What is this cbind thing? As always if you don't know
## just try it:
cbind(1:4, 1:4)

## remember, 1:4 will make a vector
## and it looks like cbind is making a matrix
## from the vectors 1:4 and 1:4 by combining them
```

```
## as columns. That's probably what cbind does.
## You can also try..

cbind(1:4, 2:5)
cbind(1:4, 1:6) ## hmm, what happened there?

## there's also a corresponding rbind function
rbind(1:4, 7:10)

## If you are confused. Try
?cbind

## I didn't say it before, but the c() function can also combine
## vectors, but it always flattens the vectors into a single one
c(1:4, 10:5, c(110, 32))

## you can also access elements of a matrix using a single
## position

## consider the following:
x
length(x)
nrow(x)
x[1]
x[ nrow(x) + 1 ]
x[ 1:nrow(x) ]
x[ length(x) ]
## you should be able to work out how this works.
## if not, then DO ask someone and discuss it.
```

Matrices can also be used in matrix multiplication and other matrix operations supported by `R`. We will not consider these here, but we will see that at a later point in the course.

Column and rownames Like vectors, matrices can have names; but now we have one set of names for the columns and one set for the rows. These can be set using the `rownames` and `colnames` functions in the same way as we used `names` for vectors.

```
x  <- matrix(1:12, nrow=4)
colnames(x)  <- c('a', 'b', 'c')
rownames(x)  <- c('A', 'B', 'C', 'D')
x

x['A',]
x['B', c('b', 'c')]
```

```
## can use subsetting to change the order
x[c('B', 'A'), c('b', 'c')]
```

Matrix exercises

1. Create the following matrix using the `matrix` function:

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

2. And then this one also using a call to the `matrix` function:

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

3. What happens if you call `t()` on these two functions (try to work it out before simply calling the function to see).

4. Create a `transposed` version of the first matrix using `cbind` or `rbind`. Use `t` and then the equality operator (`==`) to compare it to the first matrix and thus to check if you have made the correct matrix.

Lists in R

Lists are special kind of vectors that can hold element of different types. Consider the difference between:

```
x  <- c(1, 'two', 3)
x
class(x)
class(x[1])
```

and

```
x  <- list(1, 'two', 3)
x
class(x)
x[1]
class(x[1])
x[[1]]
class(x[[1]])
```

When you print[5] the list to the terminal you see that each entry actually has two sets of square brackets, with the first one having double square brackets (`[[ ]]`). This is because each element of a list is an independent object that can be

[5]This is really the wrong word, but I can't think of anything better at the moment.

of a different type. Here, each element is in fact a vector of length 1. And when we print any vector to the console it will indicate index positions. This may be a bit clearer if we modify the code a little bit:

```
x  <- list(1, c('two', 'three'), 1:100)
x
```

When we subset a list using single square brackets (`[ ]`) we obtain another list, not the object itself. This sort of makes sense if we consider the action equivalent to selecting a range of elements (`[a:b]`), since that could contain different data types and thus would need to be a list as well. To access the actual data at a single position we need to use the double square brackets (`[[ ]]`) as shown above and below.

```
x  <- list(1, c('two', 'three'), 1:100)
x[[1]]
class(x[[1]])

x[[3]]
class(x[[3]])

## subset the vector by just adding another set of brackets:
x[[3]][34:40]
```

Named lists Like vectors, lists can be named. Again to access a specific element requires the use of brackets and for the name to be quoted (unless it is the name of a variable that contains a string).

In addition you can acess named list members using the `$` operator; this can be easier to type and read, but you cannot use a variable containing a string for this.

```
## make a names list
x  <- list('a'=1, 'b'=c('two', 'three'), 'g'=1:100)

## the same behaviour as with numbers
x['a']
x[['a']]

## but we can also:
x$a
x$b
x$g

## we can subset using a defined character vector
y  <- c('a', 'b')
x[y] ## two elements of a list
x[[ y[2] ]] ## and y can be subset at the same time.
```

```
## but you cannot do:
x$y[1]
x$y
```

Lists of lists Lists can themselves contain lists; this means that a list can represent a tree like data structure of undefined depth (number of levels of branching). However, accessing such a tree is slow, and not really feasible for very large trees.

List exercises

1. Create a named vector and a named list containing at least 3 elements. Demonstrate (to yourself!) two ways in which you can access the second element of the vector and 3 ways of doing the same for the list.

Dataframes

Dataframes are like matrices except that they can contain different data types. Dataframes are actually implemented as lists of lists of the same length. Each list forms the column of a table. Dataframes are useful data structures for holding normal data tables since these often contain both character and numeric data. Like matrices, dataframes can also have named columns and named rows. You will usually come across dataframes as data read in from tabular data using functions like `read.table()`, `read.csv()`, `read.delim()`, but they can also be created with the `data.frame()` function from lists or vectors of the same length.

```
## make a dataframe with data.frame() function
## Combine a character vector and a numeric vector

df1 <- data.frame('nums'=1:4, 'letters'=c('a', 'b', 'c', 'd'))
df1

## then try to redefine df1:
df1 <- data.frame('nums'=1:5, 'letters'=c('A', 'B', 'C', 'D'))

## note the error message, and
df1
## that df1 has not changed. In general, when you get an error
## nothing should change in your workspace. That's generally
## good, but sometimes horrible...

## You can access elements like a matrix:
df1[2,1]
class(df1[2,1])

df1[2, ]
```

```r
class(df1[2,])
## Note the difference in the class of a row, vs an element.

## you can also access columns using the $ notation
## remember that the dataframe is a list of lists
## so whatever works for lists also works for dataframes.
df1$nums
## And that can be subset:
df1$nums[3]

class(df1$nums)
## the class of a row is a dataframe, whereas the class of
## a columns is whatever it contains. This should make
## sense to you

df1$letters
## ?? You may see something about levels here..
## what is that about:
class(df1$letters)

## maybe a factor (depends on R version)
## On older versions of R (before v 4)
## this will be of a "factor" class
## In this case you should
## probably have created the dataframe with:

df2 <- data.frame('nums'=1:4, 'letters'=c('a', 'b', 'c', 'd'), stringsAsFactors=FALSE)

## then check..
class(df2$letters)

## If you are using R version 4 (or later), then you can try to reverse
## the stringsAsFactors argument
df2 <- data.frame('nums'=1:4, 'letters'=c('a', 'b', 'c', 'd'), stringsAsFactors=TRUE)
class(df2$letters)

## Finally; you can usually convert a dataframe into a matrix:
as.matrix(df1)
## note what happened to the numbers and the way in which R
## changed the way it displays both text and numbers
```

Most of the above should be fairly easy to understand. But what about this `stringsAsFactors` argument to the `data.frame()` function? Earlier versions of `R` would by default convert character vectors in dataframes into something called a `factor`. That `factor` is the same kind of `factor` that you might have seen in statistics. The expectation was that you were performing some sort of statistical

tests and that you cared not about the actual text labels, but only about the identities of them, so R would convert strings to numbers and keep a mapping between numbers and the texts represented by them. You can see this if you do:

```
x <- c("b", "c", "c", "b", "a")
x

x <- as.factor(x)
x

## and finally you can see the numbers behind
as.numeric(x)
## note that the value of the first entry is not 1,
## but 2. This comes from how the characters are sorted
```

Converting strings to factors saves a bit of memory if there are many repeats of the same strings (i.e. the levels), and it can be convenient for statistical tests as these often require a set of data to be subdivided by the levels of its factors. However, it creates problems when we do *care* about the actual values of the strings[6]. It has been recognised for many years that defaulting to converting strings to factors was a bad decision; however, that has remained the default until recently in order to be compatible with S. However, this has now changed with version 4 of R; of course this now means that any old code that relied on this behaviour will not work in newer versions of R.

Conditionals, loops and program flows

A program can in theory be written down as a sequential series of instructions (commands, function calls, etc.). For example, to read in n lines from a specified file and do something with each line. That could be written as:

```
## open the file and create an object that we can use to
## read data from the file.
fh <- open(file)

## read one line from the file into variable x
x <- readLine(fh, 1)
## call a function that does something with x
do.something( x )
```

[6]The problems caused by this conversion happen only when doing certain specific operations; things mostly work fine, until at some point you have strange results. This is the worst kind of problem, because it has the potential to break your analysis without you noticing. When does it occur you ask? Unfortunately, I do not remember exactly which operations are broken by having strings as factors, but I've come across them often enough to know that they exist, and that they bite you when you least expect.

```
## read in another line (note the 1 indicates read
## 'one' line, not read the 'first' line
x <- readLine(fh, 1)
do.something( x )

## and then we can do this over and over again exactly n times
```

You should appreciate that writing your program like this has two problems:

1. It's very inefficient; you write essentially the same thing over and over again.
2. It's very inflexible; every time you read a line you have to explicitly write one command to do so. That means that you would need to rewrite the program if you wanted to read a different number of lines from the file. And you have no way of processing all the lines of a file since you can't know beforehand how many lines the files has.

To overcome these problems we make use of conditional statements of the type:

```
If something is true:
  Do something
else:
  Do something else
```

We can then rewrite the previous set of instructions:

```
if possible to read one more line:
  do:
   read one line
   process line
   go back to 1 and try to read another line
else
  do:
   exit program
```

In order words, "*while not at end of file, read one line and process it*". We call this kind of statement a `loop`, because we keep going back to the same point in the code for as long as some `condition` is `TRUE`. In `R` we generally would not read in data like this as there are functions that automate the reading in of data, and to do this directly in `R` would be really very slow. In fact these functions will implement some form of this logic, though it may be expressed in a different way.

You should note that the implied `while` loop above depends on a `conditional` statement. We can demonstrate such a statement in a much simpler form:

```
x <- 1:10
x
## do something depending on the sum of all the values
## of x
```

```r
if(sum(x) > 50){
    print("The sum of x is more than 50")
    x <- x - 10
}else{
    print("The sum of x is not more than 50")
    x <- x + 10
}
x
## since the statement above will have modified x in some way we may want to test
## it again:

if(sum(x) > 50){
    print("The sum of x is more than 50")
    x <- x - 10
}else{
    print("The sum of x is not more than 50")
    x <- x + 10
}
x
```

There are a couple of points to note here:

1. In `R` (and a lot of other languages), the conditional statement is enclosed in brackets `(sum(x) > 50)` immediately after `if`.
2. Immediately following the conditional statement is a section of code enclosed in curly brackets (`{}`), this is referred to as a *block* of code and contains the instructions that will be executed if the condition is `TRUE`. Such a block may be followed by an `else` statement and an additional block of code that will be executed in case the conditional was not `TRUE`.
3. Some languages (including `R`) allow you to omit the curly brackets if the section of code to be executed is a single line. Doing so can make your code look clearer (at least if you indent it properly!).

You should hopefully understand the above piece of code. You should also realise that the fact that we have written exactly the same statement twice indicates that we have done something bad, and that we should feel a deep sense of shame. This is for two reasons:

1. Clearly if you have to write something many times it will take you more time. Well, of course, you can copy and paste the code and then maybe it will not take that much time, so maybe OK.
2. More importantly, you may need to do exactly the same thing twice, but you probably don't know exactly what you want to do or how from the very beginning. That means that you probably will want to change the code as your understanding of the problem develops. Now you have to make exactly the same changes in both places, or delete, copy paste and all sorts of things; and you will almost certainly make some mistakes that

you may never notice. Now your code does something that you didn't intend it to do.

The second of these points is by far the more important one; in fact it is sometimes quicker to repeat some code two or three times rather than *restructuring* the code to avoid this situation. So, even *I* do this sometimes; but when I do, I do actually feel some sense of shame and guilt. And this is good.

Condition exercises

1. Use the `sample()` function like this `sample(1:20, 1)` to generate a random number from 1-20. Assign the generated number to a variable x. If the number is larger than 10 multiply it by 2, if it is smaller than 10 multiply it by 5. For this exercise, it makes sense to use the editor instead of the console to write the code.
2. Assign the number 1e100 (`1 * 10^100`) to a variable x. Check what logical value the condition `x + 1 == x` returns. Also test what `x + 1e100 == x` returns. Try to think about the reason for this behavior.

Loops

`R` has several types of loop like statements, but the most commonly used is the `for` loop. This stands for, *for each*, and indicates that you want to do something for each element of a set of objects[7]. In `R` of course, a set of objects are usually held in a vector of some sort, and the `R for` statement takes a vector as its argument followed by the block of code to be executed.

```
x <- c(3, 2, 1, 4, 23, 12, 42)

## to do something to each element of x use a for loop
for(y in x){
    print(y)
}
```

We specify the loop by the `for(y in x)` statement; this tells `R` to set the value of y to each element of x sequentially and to execute the block of code in enclosed in the curly brackets. It is equivalent to:

```
x <- c(3, 2, 1, 4, 23, 12, 42)

y <- x[1]
print(y)

y <- x[2]
print(y)
```

[7]There are several forms of the `for` loop, and not all of them fit the above description, but, the `R for` loop does and the description may make it easier to understand and remember.

```
y <- x[3]
print(y)
...
```

The problem with writing out each statement explicitly[8], rather than using the loop are both that it takes more time and, more importantly, you usually do not know how long the vector will be when writing the code (this will normally depend on data that has been read into the session).

Another common way of doing something with each element of a vector is to use the indices of the vector rather than the elements themselves (`for(i in 1:length(x))`); this is more prone to mistakes, but it is more flexible. For example we can use it to calculate a set of windowed mean values across a vector:

```
x <- c(3, 2, 1, 4, 15, 23, 12, 24, 42, 24)

## we will calculate the mean of windows of three
## values. For this reason we want to start the index
## at position 3

for(i in 3:length(x)){
    print( mean(x[ (i-2):i ]) )
}

## if we want to assign the means to a vector we can do:
y <- vector()
for(i in 3:length(x)){
    y[i-2] <- mean(x[ (i-2):i ])
}

## that's allowed, but not considered good form. We will come to something
## better eventually.
```

Here we start to get fairly convoluted expressions like, `x[ (i-2):i ]`. These can be confusing, and if you have a hard time imagining what that means, try to substitute actual numbers and see what you get. In this case remember that `x:y` means a range of numbers from `x` to `y`, that `i` can be any value between 3 and the length of `x`. So if we take the lowest value of `i` possible, the expression will expand to `x[ (3-2):3 ]`, which simplifies to, `x[ 1:3 ]`, i.e. the first three elements of x. On which we then simply call the `mean()` function and print or assign to `y`. Note that if you specify an index longer than the length of a vector in an assignment operation then the vector will be silently expanded. It may be better to specify the length of the vector beforehand, and generally the above would not be considered good `R`.

[8] In certain situations, compilers will actually create assembly code that looks similar to this; of course this can only happen if the compiler knows how long the vector is (i.e. it can't change). It's an optimisation that trades a little bit of memory for execution speed.

The above bit of code implements a square smoothening[9]; something that is actually used in some circumstances. It's not very efficient, but fine for smaller data sets. It is probably the first potentially useful code in this document.

As previously mentioned, `R` also has a `while` loop. The `while` loop is very simple to implement, but can be a little dangerous. Consider the following code.

```
### This is OK:
x <- 1
while( x < 2000 ){
    print(x)
    x <- x * 2
}

### This is not so good. You can try, as long as you know
### how to interrupt R
x <- 1
while(x < 2000){
    print(x)
}
### we forgot to increase the value of x. The loop will run for
### all enternity, or until we interrupt it, or kill R, or run
### out of electricity
```

Loop exercises

1. Assign a vector of numeric values 5, 10, 15 and 20 to a variable x. Write a for-loop that multiplies each index of the vector with 2. Have a look at the vector afterwards.
2. Assign 10 to a variable x. Use the `while()` statement to subtract x by 1 as long as x is greater than 5. Print the values of x during each iteration of the loop.

Functions

Functions contain a set of instructions which act on the arguments provided to the function. In `R`, functions can also see (read) variables that are present in the parent environment, but they cannot[10] change them.

Functions are treated as normal objects and can be passed as arguments to other functions (as we saw for `lapply` and `sapply`). They are created by the `function` function. The `function` function is used a little bit differently to other functions, as it not only has arguments specified in the brackets, but also has block of code enclosed in curly brackets immediately following the argument list.

[9] Technically it's a kernel convolution and there is a built-in `convolve()` function.

[10] Actually it is possible for functions to change the values of variables in the parent (calling) environment. But it is heavily frowned upon and so I will not include it here.

In this sense it is similar to a conditional statement, except that it returns an object; the function.

```
## make a simple function that takes two arguments and prints these
## to the console. We will use the paste() function that combines
## different types of data and generates suitable text:

## we will use a, and b as the argument names
## here; but the specific names do not matter. Feel free
## to change them.
print.two <- function(a, b){
    print( paste("The arguments were", a, b) )
}

## you can now call this function with whatever arguments you
## like.

print.two(10, 100)
print.two("one", "three")

## You can think of the values of the arguments given to the
## function as being copied to the function and assigned to
## function arguments.

print.two(c(1, 2, 3), 'three')
##?? this happens because of the way the paste function
## handles vectors. Do not care too much about it.

## We can also modify the function so that it will return
## the string that it prints:

print.two <- function(a, b){
    x <- paste("The arguments were", a, b)
    print( x )
    ## we could also write
    ## return(x), but the result of the
    ## last expression is returned by defaul
    ## so we can simply:
    x
}

x <- "an unrelated thing"
y <- print.two("one", 235)

## inspect the contents of x and y
x
```

```
y

## note that the value of x specified here did not
## change even though we modified a variable named
## x in the print.two function.
```

There are a lot of things to learn about how to use functions, but we will cover that later. For now there are two important things to understand:

1. The values of the arguments are copied to the function; within the function the values of the arguments can be specified by the variable names defined in the call to the `function()` function.
2. Assignments of values to variables within the function body do not affect the values of variables outside of the function. Similarly, variables created within the function do not exist outside of it.

Function exercises

1. Create a function that takes two arguments and returns the value of the first argument raised to the second argument. You could call it `pow` or `power` or something similar. Call your function with different arguments to make sure it works. What happens when you call it with arguments that don't make sense?

The `apply` family of functions

Using loops to do stuff to elements of vectors or parts of tables is considered sort of *old style*[11] in `R`. Instead you should be using `apply` statements.

There are at least five types of apply statements:

1. `apply` : apply functions to rows or columns of matrices
2. `lapply` : apply to elements of vectors or lists and return a `list`
3. `sapply` : like `lapply`, but if possible return a simple vector or matrix
4. `tapply` : apply a function to elements of a vector grouped by a factor
5. `mapply` : apply a function to elements from several vectors or lists

We do not really have time to go into all of these here; for brevity I will introduce only the `lapply` and `sapply` statements. We will see examples of some of the others later on.

```
x <- c(3, 2, 1, 4, 15)

## Calculate the sqrt for each element
```

[11] Developments in computer languages are driven by two sometimes conflicting motivations; to make the language more convenient to write in, and to make it safer (less easy to make mistakes). Using indices, especially denoted by `i` and `j` which look similar can make it easy to make mistakes, hence, the use of `apply` like constructs.

```r
## using lapply and sapply

lapply( x, sqrt )

## compare the output of sapply
sapply( x, sqrt )

## The sapply and lapply causes sqrt() to be called on
## each element of x and then to return the results as
## either a list (lapply) or a vector if possible (sapply).
## sqrt() takes a single argument, and returns a single value
## so this is straightforward.


## We can use functions that require more than one argument by
## giving the additional arguments afterwards. For example
## the seq() function. This takes three arguments:

seq(from=1, to=10, by=2)
## create a sequence from 1 to 10, with a spacing of 2
## we can specify the additional arguments in an apply like
## statement by naming them:

lapply( x, seq, to=1, by=-1 )
## this gives vectors of different sizes which cannot be simplified
## to a matrix, so we get the same if we use sapply

sapply( x, seq, to=1, by=-1 )

## we can try the same but with the rep() function. This simply repeats
## a value a certain number of times:
rep(10, times=3)

lapply(x, rep, times=4)

## This gives something that can be simplified to a matrix:
sapply(x, rep, times=4)
## note that we get one column for each element of x.

## we can do a square window mean, like we did using the for loop. But we
## now need to specify a new function; this function will not have a name
## but must take one argument, which will come from a vector of indices
## that we specify as before.

x <- c(3, 2, 1, 4, 15, 23, 12, 24, 42, 24)
```

```
sapply( 3:length(x), function(i){ mean(x[ (i-2):i ]) })

## if you have trouble to understand the above,
## consider the output of first
3:length(x)

## and then compare the output of the sapply statement
## to the output of:

mean( x[1:3] )
mean( x[2:4] )
mean( x[3:5] )

## and if that still confuses you can even do:
mean( c(3,2,1) )
mean( c(2,1,4) )
## and so on until it somehow makes sense.
```

The `lapply` and `sapply` functions take two mandatory arguments:

1. The vector or list to which the function (second argument) should be applied to.
2. The function that will be called for each element of the list or vector.

You can also optionally give additional named arguments to the specified function, and the second argument can use an anonymous function that is specified by a call to the `function` function. We call it anonymous, because we do not give it a name and it will not exist outside of the `lapply` or `sapply` call. We will cover functions briefly below.

Reading in data

Until now I have introduced you to some of the basic `R` concepts; but these are essentially useless in the absence of some data. Data should never[12] be entered into to program itself. Instead we usually read the data from files. `R` has several functions to read data from text files; I will introduce one of them today (`read.table()`), but you should know that there are several others.

I will have made available two text files containing information about gene structures in zebrafish (*Danio rerio*), `dr_genes.tsv` and `dr_exons.tsv`. You should (if you have not already done so), download them to your computer and save them to a directory that is easy to specify in relationship to the working directory of your R session. You can find out what that is by doing, `getwd()` in the console.

[12] Shouldn't really say never; we may sometimes put supporting data into the source code (eg. the genetic code), but these are essentially constants and so could be considered as not actual data.

The absolutely simplest way to do this is to save them to your current working directory.

```
## To check if the files are present in your current working
## directory, we can use the list.files() function
list.files()

## hopefully you will see the files, dr_genes.tsv and dr_exons.tsv
## listed. If not you can have look at the parent directory

list.files("../")
## or grandparent
list.files("../../")
## the ../ indicates the parental or containing directory

## if all that fails you could try:
list.files("/", recursive=TRUE, full.names=TRUE, pattern="dr_genes.tsv")
## but not that this might be a bit slow.


## the files are tab separated text files. That means we need to tell
## R that. We also probably want to tell R that text has not been quoted
## because it is possible that some genes may contain quote characters and
## these will otherwise cause problems.

dr.genes <- read.table("dr_genes.tsv", header=TRUE, sep="\t", quote="", stringsAsFactors=FALSE)
dr.exons <- read.table("dr_exons.tsv", header=TRUE, sep="\t", quote="", stringsAsFactors=FALSE)
## If you get errors, or this doesn't work, then you have not specified the
## location of the files properly. You need to first work out where the files
## are and then how to specify them.
## This can be difficult on Windows, because Windows lies to you.
```

Once we have read in some data, we should always have a look at the data to make sure that it is what we expect. For this there are many functions, `dim`, `head`, `tail`, `summary` that you can use:

```
## look at the top of drgenes
head(dr.genes)
```

That should show you something like:

```
> head(dr.genes)
  gene_id               id            gene       biotype seq_region_id chr
1   70477 ENSDARG00000063344         fam162a protein_coding       33576  24
2   70476 ENSDARG00000097685 si:ch211-235i11.3 protein_coding       33569  17
3   70475 ENSDARG00000036008            caly protein_coding       33552   1
4   70471 ENSDARG00000069301         tmem177 protein_coding       33574   9
5   70465 ENSDARG00000104901            ostc protein_coding       33572   7
```

```
6   70457 ENSDARG00000031836          vps37c protein_coding       33552   1
     start      end seq_region_strand
1 20927135 20934666                 1
2 50460321 50472985                -1
3 49266886 49295059                 1
4 29034221 29039506                -1
5 59169081 59176338                 1
6 44949909 44963259                 1
                                                                      strand
1    family with sequence similarity 162 member A [Source:NCBI gene;Acc:336363]
2                   si:ch211-235i11.3 [Source:ZFIN;Acc:ZDB-GENE-131125-9]
3 calcyon neuron-specific vesicular protein [Source:ZFIN;Acc:ZDB-GENE-
090313-405]
4                  transmembrane protein 177 [Source:NCBI gene;Acc:556875]
5      oligosaccharyltransferase complex subunit [Source:NCBI gene;Acc:393239]
6                   VPS37C subunit of ESCRT-I [Source:ZFIN;Acc:ZDB-GENE-
061110-126]
```

You can see that the columns have names indicating what kind of data they hold. You should be able to work out what is what there.

We should consider how big the table is (how many rows and columns):

```
dim(dr.genes)
## [1] 37241    10

dim(dr.exons)
## [1] 546158      9

## Not strangely we have many more exons than genes.
## We can ask how many exons per genes by:

nrow(dr.exons) / nrow(dr.genes)
## 14.665
## That might be inflated by the presence of alternatively
## spliced transcripts. To check this we can look at the
## dr.exons table

head(dr.exons)
## and or
colnames(dr.exons)

## you will see that dr.exons contains a column transcript_id
## and one called transcript. We can use the unique() function
## to determine how many transcripts are considered:

length(unique( dr.exons$transcript_id ))
```

```
## 65905

## But we only have 37,241 genes. So, it might be more reasonable
## to ask how many exons per unique transcript:
nrow(dr.exons) / length(unique( dr.exons$transcript_id ))
## 8.28

## We might also note that exons specified in the dr.exons table
## have ids (exon_id). Since most alternative splice forms share
## exons it seems reasonable that the total number of exons is not
## equal to the number of rows, but to the number of unique exon
## ids.

length(unique( dr.exons$exon_id ))
## 383243

## we can then use that number and divide by the numbe of genes:
length(unique( dr.exons$exon_id )) / nrow(dr.genes)
## 10.29089

## but we probably shouldn't be assuming that all the genes are unique
## we should check that in the same way
length(unique( dr.exons$exon_id )) / length(unique( dr.genes$gene_id ))
## 10.29089 same, good..

length(unique( dr.exons$exon_id )) / length(unique( dr.genes$gene ))
## 11.67498
## but that is slightly higher? What is going on here?
```

We will cover other topics later; including how to summarise and visualise the distributions of this kind of data and how to look at sequences and their properties.