R for Biologists - An Introduction to R (Lecture 2)

Getting Help

One of the most important things to know when learning a new language is how to get help!

As I said last time, one of the great things about R is that it requires developers to provide documentation on every function and package created and that documentation is always in a consistant form.

R has a few layers to its documentation system, the first layer being system documentation, the simplest way to get to this documentation is via the command

```
help.start()
```

or in ***RStudio***, the help tab and home

Help documentation on particular functions is best produced using

```
help(solve)
## or, equivically
?solve
```

to search for help using keywords use

```
help.search(solve)
## or, equivically
??solve
```

for help on help, try

```
?help.start
?help
?help.search
```

Finally, most functions come with examples at the end of the help documentation, to run these examples (without copy/paste), try

```
example(solve)
## also,  example(plot) is always fun to see how flexible this one function is
example(plot, ask=FALSE)
```

R Commands

R is case-sensitive to ***solve*** and ***Solve*** are two different commands. ***a*** and ***A*** are two different variables.

R functions and variables must start with either a '.' or a letter. if a function or variable starts with a '.', the second character must not be digit. functions and variables that begin wiht a '.' are hidden and do not show up in `ls()`. The length of names is effectively unlimited.

Commands can be seperated by a newline or by a semi-color (';'). Commands can be grouped in to compound expressions by braces (begin compound expression '{', end compound expression '}'), more on this later.

Comments begin with the hash-tag ('#') and continue to the newline character, they can be put almost anywhere (not inside strings, or inside arguement list of a function definition).

If a command in not complete, R will give a different prompt,

```
+
```

on second and subsequent lines and continue to read input until the command is syntactically complete.

Elementary commands consist of either expressions

```
5+5
```

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

or assignments.

```
a <- 5+5
a
```

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

If an expression is given as a command, it is evaluated, printed (unless specifically made invisible), and the value is lost. An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed.

Executing commands from and diverting output to files

R commands can be stored in an external file, say commands.R, that may be executed at any time in an R session with the command

```
source("commands.R")
```

Assuming commands.R is in your current working directory, the full path is needed if it is outside your directory. `source` can also be used to 'source' commands from the internet, as in

```
source("http://webpages.uidaho.edu/msettles/Rcode/Venn.R")
```

The function sink,

```
sink("record.lis")
```

will divert all subsequent output from the console to the external file, record.lis in the current working directory. The command

```
sink()
```

restores it to the console once again.

Data permanency and removing objects

The entities that R creates and manipulates are known as objects. objects can be **variables, arrays of numbers, character strings, functions, or more general structures built from such components**.

During an R session, objects are created and stored by name. The R commands

```
objects()
```

```
## [1] "a"
```

```
ls()
```

```
## [1] "a"
```

can be used to display the names of the non-hidden objects which are currently stored within the R environment. The collection of objects currently stored is called the workspace. To remove objects

```
rm(x, y, z, ink, junk, temp, foo, bar)
rm(list=ls()) ## removes all objects
```

All objects created during an R session can be stored permanently in a file for use in future R sessions. At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a file called .RData in the current directory, and the command lines used in the session are saved to a file called .Rhistory.

When R is started at later time from the same directory it reloads the workspace from these files (.RData and .Rhistory).

To save either the current workspace, data, or the history at any time during an R session you can use the commands

```
savehistory("myhistory.Rhistory")
loadhistory("myhistory.Rhistory")
save.image("myData.RData")
```

all of these default to ".Rhistory", if the filename is left out

To save only certain objects to a file use

```
save("A", file="myData.RData")
```

Simple Manipulations

Vectors and assignment

R operates on *named* data structures. The simplest such structure is the vector, which is a single entity consisting of an ordered collection of a common data type. To set up a vector named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
x
```

```
## [1] 10.4  5.6  3.1  6.4 21.7
```

This is an assignment statement using the function c() which in this context can take an arbitrary number of vector arguments and whose value is a vector got by concatenating its arguments end to end.

A number occurring by itself in an expression is taken as a vector of length one.

Notice that the assignment operator (<-), which consists of the two characters < ("less than") and - ("minus") occurring strictly side-by-side and it 'points' to the object receiving the value of the expression. In most contexts the '=' operator can be used as an alternative. Assignment can also be made using the function assign(). An equivalent way of making the same assignment as above is with:

```
assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

The usual operator, <-, can be thought of as a syntactic short-cut to this.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

```
c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

BUT NEVER DO THIS

If an expression is used as a complete command, the value is printed and lost. So now if we were to use the command

```
1/x
```

```
## [1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

the reciprocals of the five values would be printed at the terminal, and the value of x, of course, unchanged.

```
y <- c(x, 0, x)
```

would create a vector y with 11 entries consisting of two copies of x with a zero in the middle place.

Vector Arithmatic (assuming working with a numeric vector)

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are recycled as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
v <- 2*x + y + 1
```

```
## Warning in 2 * x + y: longer object length is not a multiple of shorter
## object length
```

```
v
```

```
##  [1] 32.2 17.8 10.3 20.2 66.1 21.8 22.6 12.8 16.9 50.8 43.5
```

generates a new vector v of length 11 constructed by adding together, element by element, 2*x repeated 2.2 times, y repeated just once, and 1 repeated 11 times.

The elementary arithmetic operators are the usual **“+, -, *, /“** and **“^”** for raising to a power. In addition all of the common arithmetic functions are available. **“log, exp, sin, cos, tan, sqrt”**, and so on, all have their usual meaning. **max** and **min* select the largest and smallest elements of a vector respectively. *range* is a function whose value is a vector of length two, namely ‘c(min(x), max(x))’. *length(x)* is the number of elements in x, *sum(x)* gives the total of the elements in x, and *prod(x)* their product. Two statistical functions are *mean(x)* which calculates the sample mean, which is the same as ‘sum(x)/length(x)’, and *var(x)* which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

```
## [1] 53.853
```

or sample variance.

sort(x) returns a vector of the same size as x with the elements arranged in increasing order; however there are other more flexible sorting facilities available (see order() or sort.list() which produce a permutation to do the sorting). Note that max and min select the largest and smallest values in their arguments, even if they are given several vectors. For most purposes the user will not be concerned if the “numbers” in a numeric vector are integers, reals or even complex. Internally calculations are done as double precision real numbers, or double precision complex numbers if the input data are complex.

```
max(x)
```

```
## [1] 21.7
```

```
min(x)
```

```
## [1] 3.1
```

```
range(x)
```

```
## [1]  3.1 21.7
```

```
length(x)
```

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

```
sum(x)
```

```
## [1] 47.2
```

```
prod(x)
```

```
## [1] 25073.95
```

```
mean(x)
```

```
## [1] 9.44
```

```
var(x)
```

```
## [1] 53.853
```

```
sort(x)
```

```
## [1]  3.1  5.6  6.4 10.4 21.7
```

```
order(x)
```

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

```
sort.list(x)
```

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

Generating Regular Sequences

R has a number of facilities for generating commonly used sequences of numbers. For example 1:30 is the vector c(1, 2, . . . , 29, 30). The colon operator has high priority within an expression, so, for example

```
2*1:15
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30
```

DO Put n <- 10 and compare the sequences 1:n-1 and 1:(n-1).

The construction 30:1 may be used to generate a sequence backwards.

The function seq() is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is seq(2,10) is the same vector as 2:10.

The fifth parameter "along", which if used must be the only parameter, and creates a sequence 1, 2, . . . , length(vector), or the empty sequence if the vector is empty (as it can be).

A related function is rep() which can be used for replicating an object in various complicated ways. The simplest form is

```
s5 <- rep(x, times=5)
```

which will put five copies of x end-to-end in s5. or,

```
s6 <- rep(x, each=5)
```

which repeats each element of x five times before moving on to the next.

Other types of Vectors

Vectors can be logical **TRUE** or ***FALSE**, the logical operators are <, <=, >, >=, == for exact equality and != for inequality, will create logical vectors

```
x > 3
```

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

In addition if c1 and c2 are logical expressions, then c1 & c2 is their intersection ("and"), c1 | c2 is their union ("or"), and !c1 is the negation of c1.

Character quantities and character vectors are used frequently in R, for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character, e.g., "x-values", "New iteration results".

Character strings are entered using either matching double (") or single (') quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using as the escape character, so \ is entered and printed as \, and inside double quotes" is entered as ". Other useful escape sequences are \n, newline, \t, tab and \b, backspace—see ?Quotes for a full list.

Character vectors may be concatenated into a vector by the c() function; examples of their use will emerge frequently. The paste() function takes an arbitrary number of arguments and concatenates them one by one into character strings. Any numbers given among the arguments are coerced into character strings in the evident way, that is, in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named parameter, sep=string, which changes it to string, possibly empty.

For example

```
labs <- paste(c("X","Y"), 1:10, sep="")
labs
```

```
##  [1] "X1"  "Y2"  "X3"  "Y4"  "X5"  "Y6"  "X7"  "Y8"  "X9"  "Y10"
```

makes labs into the character vector. Note particularly that recycling of short lists takes place here too; thus c("X", "Y") is repeated 5 times to match the sequence 1:10. 8

Session Info

```r
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.11.3 (El Capitan)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] magrittr_1.5    formatR_1.2.1   tools_3.2.3     htmltools_0.3
##  [5] yaml_2.1.13     stringi_1.0-1   rmarkdown_0.9.5 knitr_1.12.3
##  [9] stringr_1.0.0   digest_0.6.9    evaluate_0.8
```