R for Biologists - An Introduction to R (Lecture 5)

Reading data from files

In order to read an entire data frame directly from an external file into R, the external file will normally have a special form.

The first line of the file should have a name (column header) for each variable in the data frame. Each additional line of the file has as its first item a row label and the values for each variable. If the file has one fewer item in its first line than in its second, this arrangement is presumed to be in force. So the first few lines of a file to be read as a data frame might look as follows.

```
Input file form with names and row labels:

     Price    Floor     Area   Rooms     Age  Cent.heat
01   52.00    111.0      830     5       6.2      no
02   54.75    128.0      710     5       7.5      no
03   57.50    101.0     1000     5       4.2      no
04   57.50    131.0      690     6       8.8      no
05   59.75     93.0      900     5       1.9     yes
...
```

By default numeric items (except row labels) are read as numeric variables and non-numeric variables, such as Cent.heat in the example, as factors. This can be changed if necessary.

The function read.table() can then be used to read the data frame directly

```
HousePrice <- read.table("houses.data")
```

Often you will want to omit including the row labels directly and use the default labels. In this case the file may omit the row label column as in the following.

```
Input file form without row labels:

Price    Floor     Area   Rooms     Age  Cent.heat
52.00    111.0      830     5       6.2      no
54.75    128.0      710     5       7.5      no
57.50    101.0     1000     5       4.2      no
57.50    131.0      690     6       8.8      no
59.75     93.0      900     5       1.9     yes
...
```

The data frame may then be read as

```
HousePrice <- read.table("houses.data", header=TRUE)
```

where the header=TRUE option specifies that the first line is a line of headings, and hence, by implication from the form of the file, that no explicit row labels are given.

There are other functions for reading in data directly to a data.frame: read.csv, read.delim (tab delimeted files). These functions however just call read.table with a predefined set of parameters.

The scan function

Suppose the data vectors are of equal length and are to be read in parallel. Further suppose that there are three vectors, the first of mode character and the remaining two of mode numeric, and the file is input.dat. The first step is to use scan() to read in the three vectors as a list, as follows

```
inp <- scan("input.dat", list("",0,0))
```

The second argument is a dummy list structure that establishes the mode of the three vectors to be read. The result, held in inp, is a list whose components are the three vectors read in. To separate the data items into three separate vectors, use assignments like

```
label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

More conveniently, the dummy list can have named components, in which case the names can be used to access the vectors read in. For example

```
inp <- scan("input.dat", list(id="", x=0, y=0))
```

If you wish to access the variables separately they may either be re-assigned to variables in the working frame:

```
label <- inp$id; x <- inp$x; y <- inp$y
```

or the list may be attached at position 2 of the search path (see Attaching arbitrary lists).

If the second argument is a single value and not a list, a single vector is read in, all components of which must be of the same mode as the dummy value.

```
X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

Accessing builtin datasets

Around 100 datasets are supplied with R (in package datasets), and others are available in packages (including the recommended packages supplied with R). To see the list of datasets currently available use

```
data()
```

As from R version 2.0.0 all the datasets supplied with R are available directly by name. However, many packages still use the earlier convention in which data was also used to load datasets into R, for example

```
data(infert)
```

and this can still be used with the standard packages (as in this example). In most cases this will load an R object of the same name. However, in a few cases it loads several objects, so see the on-line help for the object to see what to expect.

Loading data from other R packages

To access data from a particular package, use the package argument, for example

```
data(package="rpart")
data(Puromycin, package="datasets")
```

If a package has been attached by library, its datasets are automatically included in the search.

User-contributed packages can be a rich source of datasets.

Grouping, loops, conditional execution and functions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

Commands may be grouped together in braces, {expr_1; ...; expr_m}, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used a part of an even larger expression, and so on.

Control Statements

if statements

The language has available a conditional construction of the form

```
if (expr_1) expr_2 else expr_3
```

where expr_1 must evaluate to a single logical value and the result of the entire expression is then evident.

The "short-circuit" operators && and || are often used as part of the condition in an if statement. Whereas & and | apply element-wise to vectors, && and || apply to vectors of length one, and only evaluate their second argument if necessary.

There is a vectorized version of the if/else construct, the ifelse function. This has the form ifelse(condition, a, b) and returns a vector of the length of its longest argument, with elements a[i] if condition[i] is true, otherwise b[i].

for loops, repeat and while

There is also a for loop construction which has the form

```
for (name in expr_1) expr_2
```

where name is the loop variable. expr_1 is a vector expression, (often a sequence like 1:20), and expr_2 is often a grouped expression with its sub-expressions written in terms of the dummy name. expr_2 is repeatedly evaluated as name ranges through the values in the vector result of expr_1.

As an example, suppose ind is a vector of class indicators and we wish to produce separate plots of y versus x within classes. One possibility here is to use coplot(),19 which will produce an array of plots corresponding to each level of the factor. Another way to do this, now putting all plots on the one display, is as follows:

```
xc <- split(x, ind)
yc <- split(y, ind)
for (i in 1:length(yc)) {
  plot(xc[[i]], yc[[i]])
  abline(lsfit(xc[[i]], yc[[i]]))
}
```

(Note the function split() which produces a list of vectors obtained by splitting a larger vector according to the classes specified by a factor. This is a useful function, mostly used in connection with boxplots. See the help facility for further details.)

Warning: for() loops are used in R code much less often than in compiled languages. Code that takes a 'whole object' view is likely to be both clearer and faster in R. Other looping facilities include the

```
repeat expr
```

statement and the

```
while (condition) expr
```

statement.

The `break` statement can be used to terminate any loop, possibly abnormally. This is the only way to terminate repeat loops. The `next` statement can be used to discontinue one particular cycle and skip to the "next".

Writing your own functions

As we have seen informally along the way, the R language allows the user to create objects of mode function. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive.

It should be emphasized that most of the functions supplied as part of the R system, such as mean(), var(), postscript() and so on, are themselves written in R and thus do not differ materially from user written functions.

A function is defined by an assignment of the form

```
name <- function(arg_1, arg_2, ...) expression
```

The expression is an R expression, (usually a grouped expression), that uses the arguments, arg_i, to calculate a value. The value of the expression is the value returned for the function.

A call to the function then usually takes the form name(expr_1, expr_2, ...) and may occur anywhere a function call is legitimate.

Named arguments adn defaults

If arguments to called functions are given in the "name=object" form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.

Thus if there is a function fun1 defined by

```
fun1 <- function(data, data.frame, graph, limit) {
  [function body omitted]
}
```

then the function may be invoked in several ways, for example

```
ans <- fun1(d, df, TRUE, 20)
ans <- fun1(d, df, graph=TRUE, limit=20)
ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

are all equivalent.

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if fun1 were defined as

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }
```

it could be called as

```
ans <- fun1(d, df)
```

which is now equivalent to the three cases above, or as

```
ans <- fun1(d, df, limit=10)
```

which changes one of the defaults.

It is important to note that defaults may be arbitrary expressions, even involving other arguments to the same function; they are not restricted to be constants as in our simple example here.

Session Info

```
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
```