R for Biologists - An Introduction to R (Lecture 4)

Arrays and Matrices

R allows simple facilities for creating and handling arrays, and in particular the special case of matrices. An array in R can have one, two or more dimensions. It is simply a vector which is stored with additional attributes giving the dimensions (attribute "dim") and optionally names for those dimensions (attribute "dimnames").

A dimension vector is a vector of non-negative integers. If its length is k then the array is k-dimensional, e.g. a matrix is a 2-dimensional array. The dimensions are indexed from one up to the values given in the dimension vector.

A vector can be used by R as an array only if it has a dimension vector as its dim attribute.

Suppose, for example, z is a vector of 1500 elements. The assignment

```
z <- c(rep(1:5,each=3),rep(6:10,each=3),rep(11:15,each=3))
dim(z) <- c(3,5,3)
```

Other functions such as matrix() and array() are available for simpler and more natural looking assignments.

The values in the data vector give the values in the array in "column major order," with the first subscript moving fastest and the last subscript slowest.

Arrays can be one-dimensional: such arrays are usually treated in the same way as vectors (including when printing), but their are exceptions (though none that I can think of at the moment).

Array Indexing

Individual elements of an array may be referenced by giving the name of the array followed by the subscripts in square brackets, separated by commas.

More generally, subsections of an array may be specified by giving a sequence of index vectors in place of subscripts; however if any index position is given an empty index vector, then the full range of that subscript is taken.

So, continuing the previous example, z[2„] is a 5 * 3 array with dimension vector c(5,3)

```
z[2,,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

z[„] stands for the entire array, which is the same as omitting the subscripts entirely and using z alone.

For any array, say Z, the dimension vector may be referenced explicitly as dim(Z) (on either side of an assignment).

Also, if an array name is given with just one subscript or index vector, then the corresponding values of the data vector only are used; in this case the dimension vector is ignored. This is not the case, however, if the single index is not a vector but itself an array.

Index Matrices

As well as an index vector in any subscript position, a matrix may be used with a single index matrix in order either to assign a vector of quantities to an irregular collection of elements in the array, or to extract an irregular collection as a vector.

A matrix example makes the process clear. In the case of a doubly indexed array, an index matrix may be given consisting of two columns and as many rows as desired. The entries in the index matrix are the row and column indices for the doubly indexed array. Suppose for example we have a 4 by 5 array X and we wish to do the following:

Extract elements X[1,3], X[2,2] and X[3,1] as a vector structure, and Replace these entries in the array X by zeroes. In this case we need a 3 by 2 subscript array, as in the following example.

```
x <- array(1:20, dim=c(4,5))   # Generate a 4 by 5 array.
x
```

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

```
i <- array(c(1:3,3:1), dim=c(3,2))
i                              # i is a 3 by 2 index array.
```

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

```
x[i]                           # Extract those elements
```

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

```
x[i] <- 0                      # Replace those elements by zeros.
x
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    0   13   17
## [2,]    2    0   10   14   18
## [3,]    0    7   11   15   19
## [4,]    4    8   12   16   20
```

Negative indices are not allowed in index matrices. NA and zero values are allowed: rows in the index matrix containing a zero are ignored, and rows containing an NA produce an NA in the result.

Index matrices must be numerical: any other form of matrix (e.g. a logical or character matrix) supplied as a matrix is treated as an indexing vector.

Matrix facilities

A matrix is just an array with two subscripts. R contains many operators and functions that are available only for matrices. For example t(X) is the matrix transpose function. The functions nrow(A) and ncol(A) give the number of rows and columns in the matrix A respectively.

Other matrix facilities * Outer Product - using `%o%` * Matrix Product - `%*%` * Cross Product - crossproc() * Diagonal =- diag() * Solve a set of linear equations - solve() * Eigenvalues and Eigenvectors - eigen() * Least squars fitting - lsfit() * QR decomposition - qr()

Forming and paritioning matrices, with cbind() and rbind()

As we've already seen, you can form matrices by folding vectors in to multiple dimensions. You can also form matrices with combinding row, or column, vectors. Roughly cbind() forms matrices by binding together matrices (or vectors) horizontally, or column-wise, and rbind() vertically, or row-wise.

In the assignment

```
X <- cbind(c(1,2,3), c(4,5,6), c(7,8,9))
Y <- rbind(c(1,2,3), c(4,5,6), c(7,8,9))
```

the arguments to cbind() must be either vectors of any length, or matrices with the same column size, that is the same number of rows.

If some of the arguments to cbind() are vectors they may be shorter than the column size of any matrices present, in which case they are cyclically extended to match the matrix column size (or the length of the longest vector if no matrices are given).

The function rbind() does the corresponding operation for rows. In this case any vector argument, possibly cyclically extended, are of course taken as row vectors.

Suppose X and Y have the same number of rows. To combine these by columns into a matrix X, together with an initial column of 1s we can use

```
Z <- cbind(1, X, Y)
```

It should be noted that whereas cbind() and rbind() are concatenation functions that respect dim attributes, the basic c() function does not, but rather clears numeric objects of all dim and dimnames attributes. This is occasionally useful in its own right.

The official way to coerce an array back to a simple vector object is to use as.vector()

```
vec <- as.vector(X)
```

However a similar result can be achieved by using c() with just one argument, simply for this side-effect:

```
vec <- c(X)
```

There are slight differences between the two, but ultimately the choice between them is largely a matter of style (with the former being preferable). The result of rbind() or cbind() always has matrix status. Hence cbind(x) and rbind(x) are possibly the simplest ways explicitly to allow the vector x to be treated as a column or row matrix respectively.

Frequency tables from Factors

Recall that a factor defines a partition into groups. Similarly a pair of factors defines a two way cross classification, and so on. The function table() allows frequency tables to be calculated from equal length factors. If there are k factor arguments, the result is a k-way array of frequencies.

Suppose, for example, that statef is a factor giving the state code for each entry in a data vector. The assignment

```r
state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
                  "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
                  "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
                  "sa",  "act", "nsw", "vic", "vic", "act")
statef <- factor(state)
statefr <- table(statef)
```

gives in statefr a table of frequencies of each state in the sample. The frequencies are ordered and labelled by the levels attribute of the factor. This simple case is equivalent to, but more convenient than,

Further suppose that incomef is a factor giving a suitably defined "income class" for each entry in the data vector, for example with the cut() function:

```r
incomef <-  rep(1:5,times=10)[sample(1:50,30)]
```

Then to calculate a two-way table of frequencies:

```r
table(incomef,statef)
```

```
##        statef
## incomef act nsw nt qld sa tas vic wa
##       1   0   1  1   0  0   2   0  1
##       2   1   1  0   3  1   0   0  1
##       3   0   1  1   1  1   0   2  0
##       4   0   1  0   1  1   0   2  2
##       5   1   2  0   0  1   0   1  0
```

Extension to higher-way frequency tables is immediate.

Lists and data frames

Lists

An R list is an object consisting of an ordered collection of objects known as its components.

There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. Here is a simple example of how to make a list:

```r
Lst <- list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))
```

Components are always numbered and may always be referred to as such. Thus if Lst is the name of a list with four components, these may be individually referred to as Lst[[1]], Lst[[2]], Lst[[3]] and Lst[[4]]. If, further, Lst[[4]] is a vector subscripted array then Lst[[4]][1] is its first entry.

If Lst is a list, then the function length(Lst) gives the number of (top level) components it has.

Components of lists may also be named, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form

```
Lst$child.ages
```

```
## [1] 4 7 9
```

for the same thing.

This is a very useful convention as it makes it easier to get the right component if you forget the number.

So in the simple example given above:

Lst$name is the same as Lst[[1]] and is the string "Fred",

Lst$wife is the same as Lst[[2]] and is the string "Mary",

Lst$child.ages[1] is the same as Lst[[4]][1] and is the number 4.

Additionally, one can also use the names of the list components in double square brackets, i.e., Lst[["name"]] is the same as Lst$name. This is especially useful, when the name of the component to be extracted is stored in another variable as in

```
Lst[["name"]]
```

```
## [1] "Fred"
```

```
x <- "name"
Lst[[x]]
```

```
## [1] "Fred"
```

It is very important to distinguish Lst[[1]] from Lst[1]. '[[. . .]]' is the operator used to select a single element, whereas '[. . .]' is a general subscripting operator. Thus the former is the first object in the list Lst, and if it is a named list the name is not included. The latter is a sublist of the list Lst consisting of the first entry only. If it is a named list, the names are transferred to the sublist.

The vector of names is in fact simply an attribute of the list like any other and may be handled as such. Other structures besides lists may, of course, similarly be given a names attribute also.

Constructing lists

New lists may be formed from existing objects by the function list(). An assignment of the form

```
Lrt <- list(name_1=c(1,3), letters, name_m=list(1,3,45))
```

sets up a list Lst of m components using object_1, . . . , object_m for the components and giving them names as specified by the argument names, (which can be freely chosen). If these names are omitted, the components are numbered only. The components used to form the list are copied when forming the new list and the originals are not affected.

Lists, like any subscripted object, can be extended by specifying additional components. For example

```
Lst[5] <- list(matrix=X)
```

Concatenating lists

When the concatenation function c() is given list arguments, the result is an object of mode list also, whose components are those of the argument lists joined together in sequence.

```
list.ABC <- c(Lrt, Lst)
```

Recall that with vector objects as arguments the concatenation function similarly joined together all arguments into a single vector structure. In this case all other attributes, such as dim attributes, are discarded.

Data Frames

A data frame is a list with class "data.frame". There are restrictions on lists that may be made into data frames, namely

- The components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames.
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Numeric vectors, logicals and factors are included as is, and character vectors are coerced to be factors, whose levels are the unique values appearing in the vector.
- Vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.

A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

Making data frames

Objects satisfying the restrictions placed on the columns (components) of a data frame may be used to form one using the function data.frame:

```
accountants <- data.frame(home=statef, loot=incomef*10e3, shot=incomef)
```

A list whose components conform to the restrictions of a data frame may be coerced into a data frame using the function as.data.frame() The simplest way to construct a data frame from scratch is to use the read.table() function to read an entire data frame from an external file.

Attaching and detaching

The $ notation, such as accountants$home, for list components is not always very convenient. A useful facility would be somehow to make the components of a list or data frame temporarily visible as variables under their component name, without the need to quote the list name explicitly each time.

The attach() function takes a 'database' such as a list or data frame as its argument. Thus suppose lentils is a data frame with three variables lentils$u, lentils$v, lentils$w. The attach

```r
lentils <- data.frame(u=1:50,v=seq(2,100,by=2), w=51:100)
attach(lentils)
```

places the data frame in the search path at position 2, and provided there are no variables u, v or w in position 1, u, v and w are available as variables from the data frame in their own right. At this point an assignment such as

```r
u <- v+w
```

does not replace the component u of the data frame, but rather masks it with another variable u in the working directory at position 1 on the search path. To make a permanent change to the data frame itself, the simplest way is to resort once again to the $ notation:

```r
lentils$u <- v+w
```

However the new value of component u is not visible until the data frame is detached and attached again.

To detach a data frame, use the function

```r
detach()
```

More precisely, this statement detaches from the search path the entity currently at position 2. Thus in the present context the variables u, v and w would be no longer visible, except under the list notation as lentils$u and so on. Entities at positions greater than 2 on the search path can be detached by giving their number to detach, but it is much safer to always use a name, for example by detach(lentils) or detach("lentils")

Note: In R lists and data frames can only be attached at position 2 or above, and what is attached is a copy of the original object. You can alter the attached values via assign, but the original list or data frame is unchanged.

Working with data frames

A useful convention that allows you to work with many different problems comfortably together in the same working directory is

- gather together all variables for any well defined and separate problem in a data frame under a suitably informative name;
- when working with a problem attach the appropriate data frame at position 2, and use the working directory at level 1 for operational quantities and temporary variables;
- before leaving a problem, add any variables you wish to keep for future reference to the data frame using the $ form of assignment, and then detach();
- finally remove all unwanted variables from the working directory and keep it as clean of left-over temporary variables as possible.

In this way it is quite simple to work with many problems in the same directory, all of which have variables named x, y and z, for example.

Attaching any old list

object as well. In particular any object of mode "list" may be attached in the same way:

```
 > attach(any.old.list)
```

Anything that has been attached can be detached by detach, by position number or, preferably, by name.

Managing the search path

The function 'search' shows the current search path and so is a very useful way to keep track of which data frames and lists (and packages) have been attached and detached. Initially it gives

```r
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

where .GlobalEnv is the workspace

After lentils is attached we have

```r
attach(lentils)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     u
```

```r
search()
```

```
##  [1] ".GlobalEnv"        "lentils"           "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

```r
ls(2)
```

```
## [1] "u" "v" "w"
```

and as we see ls (or objects) can be used to examine the contents of any position on the search path.

Finally, we detach the data frame and confirm it has been removed from the search path.

```r
detach("lentils")
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

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    tools_3.2.3     htmltools_0.3   yaml_2.1.13
##  [5] stringi_1.0-1   rmarkdown_0.9.5 knitr_1.12.3    stringr_1.0.0
##  [9] digest_0.6.9    evaluate_0.8
```