BMI 713: Computational Statistics for Biomedical Sciences

Lecture 2 - Lab

September 16, 2010

1 Objects

1.1 Creating objects

1.1.1 Vectors

We can construct a vector of ten zeroes and assign it to the symbol `v` using:

```
> v <- rep(0, 10)   # repeat 0 10 times
> v
 [1] 0 0 0 0 0 0 0 0 0 0
```

Take a moment to consider the *assignment* of the generated vector to the symbol `v`. If we do not assign our vector to some variable, then `R` will "forget" about it once all of the instructions on the same line have finished.

The special `:` operator can be used to create a regular sequence of integers. The more general `seq` function can be used to create other integer sequences.

```
> seq(from=1, to=10, by=1)
 [1]  1  2  3  4  5  6  7  8  9 10
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

It is also possible to create a vector by explicitly supplying each value to the `c` function:

```
> c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

We often work with vectors of numeric data, but non-numeric vectors will certainly appear from time to time. Creating vectors with other types of data is no different:

```
> c("a", "b", "c")     # Character-mode data
[1] "a" "b" "c"
> c(T, T, T, F, F)     # Logical-mode data
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

Note: vectors can only contain *one* type of data.

1.1.2 Matrices

The `matrix` function is used to create matrices.

```
matrix(0, ncol=3, nrow=3)
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

0 is the data to be converted into matrix form, ncol specifies the number of columns, and nrow specifies the number of rows. If neither nrow nor ncol are specified, a matrix with a single column will be created. If only one parameter is specified, the other parameter will be inferred. Notice that we supply only a single 0, but the resulting matrix contains nine 0s. This is because R *recycles* the supplied data to fill the requested number of rows and columns.

To create a matrix from a vector (for example, the values 1 through 9):

```
> matrix(1:9, ncol=3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

In this case, matrix has inferred the value of nrow: the length of the vector 1:9 divided by ncol. If the length of the supplied vector is *not* divisible by the specified number of columns (or rows), R will *recycle* the vector to fill in the unexpected cells. That is, it will fill as far as the vector will allow, then start again from the beginning:

```
> matrix(1:10, ncol=3)  # length(1:10)=10, which is not divisible by 3
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7    1
[4,]    4    8    2
Warning message:
In matrix(1:10, ncol = 3) :
  data length [10] is not a sub-multiple or multiple of the number of rows [4]
```

1.2 Combining objects

As we have already seen, the c function allows multiple data elements to be combined into a vector. In fact, the c function is much more general. It can be used, for example, to combine two vectors:[1]

```
> v <- c(1, 2, 3)
> w <- c(4, 5, 6)
> c(v, w)
[1] 1 2 3 4 5 6
> c(v, v, v)             # Many vectors can be combined at once
[1] 1 2 3 1 2 3 1 2 3
```

The functions **rbind** and **cbind** can be used to attach **r**ows or **c**olumns to a matrix:

```
> rbind(1:3, 4:6, 7:9)   # Create a new matrix via rbind
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> M <- matrix(0, ncol=3, nrow=3)
> M
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
> cbind(M, 1:3)          # Attach a new column to an existing matrix
```

[1] Actually, c(1, 2) is already an example of combining two vectors. R understands the values 1 and 2 as two numeric vectors of length 1.

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

1.3 Indexing (or subsetting) objects

R provides an extremely powerful indexing system via the [] operators. The methods for indexing objects are in many cases intuitive and consistent across different object types. It is essential to understand that most indexing operations can be used both to read subsets of an object *and* to write to them.

1.3.1 Vectors

Vectors are perhaps the simplest objects in R. When data is stored as a vector, each data element is assigned an *index* equal to its position in the original data set. That is, the first element in the original data set is assigned the index 1, the second is assigned 2, and so on. We can access data by index using the [] operators. If v denotes a vector, the syntax is v[*index.vector*]:

```
> v <- c(1, 9, 3, 3, 4, 8, 4)
> v[2]
[1] 9
```

The element at position 2 in v is the number 9.[2]

index.vector need not be of length 1. For example, we might create *index.vector* using the : operator or the c function:

```
> v[c(2, 6, 1)]
[1] 9 8 1
> v[2:4]
[1] 9 3 3
```

The elements at positions 2, 6 and 1 (in that order) have been selected and returned in a vector of length 3.

The previous examples *read* data from vectors, but it is also possible to *write* (or assign) to multiple indexes with one statement. For example, suppose we wanted to set the first 3 values of some vector to 0:

```
> v
[1] 1 9 3 3 4 8 4
> v[1:3] <- 0
> v
[1] 0 0 0 3 4 8 4
```

1.3.2 Matrices

Matrices are two dimensional vectors (actually they're arrays, but the distinction will not be explored at the moment), meaning that each element is indexed by a pair of numbers rather than a single number.[3] The syntax for indexing an array is M[*row.vector, column.vector*].

```
> M <- matrix(1:9, ncol=3)
> M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

[2] In this case, 2 is *index.vector*. While it may seem that the single value 2 is not a vector, R treats it as a numeric vector of length 1.

[3] It is possible to index a matrix by a single value: R first coerces the matrix into a vector and then indexes the vector as explained above.

```
> M[1, 1]               # Row 1, column 1
[1] 1                   # Vector of length 1, mode numeric
> M[c(1, 2), c(1, 2)]   # The intersection of rows 1 and 2 and columns 1 and 2
     [,1] [,2]          # 2x2 matrix, mode numeric
[1,]    1    4
[2,]    2    5
> M[1:2, 1:2]           # A convenient equivalent
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

For convenience, *row.vector* and *column.vector* may be omitted to select all rows or all columns, respectively.

```
> M[1, ]                # column.vector omitted, so all columns are selected
[1] 1 4 7               # This is the first row of the matrix
> M[,]                  # The result of omitting both should be obvious
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Using these constructs, it is easy to select a set of rows in a matrix:

```
> M[c(1,2), ]           # Get rows 1 and 2 (and, implicitly, all columns)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
> M[1:2, ]              # Equivalent
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

In the same manner as vectors, assignments can be made to multiple indexes with a single statement:

```
> M[1:2, 1:2] <- 0      # Set the top-left 2x2 submatrix of M to 0
> M
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0    0    8
[3,]    3    6    9
> M[c(1, 3), c(1, 3)] <- 10  # Set the corners to 10
> M
     [,1] [,2] [,3]
[1,]   10    0   10
[2,]    0    0    8
[3,]   10    6   10
> M2 <- matrix(1:4, ncol=2)
> M2
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> M[2:3, 2:3] <- M2    # Copy M2 into the bottom 2x2 submatrix
> M
     [,1] [,2] [,3]
[1,]   10    0   10
[2,]    0    1    3
[3,]   10    2    4
```

1.3.3 Indexing preserves frequency and order

It might be surprising that indexing respects the *frequency* of the queried positions. That is, it is perfectly valid to ask for a position more than once:

```
> v <- c(1, 9, 3, 3, 4, 8, 4)
> v[c(2, 2, 2)]      # Produce 3 copies of the value at position 2 in v
[1] 9 9 9
> M <- matrix(1:9, ncol=3)
> M[c(1, 1, 1), ]    # Produce 3 copies of row 1, using all available columns
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    1    4    7
[3,]    1    4    7
> M[ , c(2, 2, 2)]   # Produce 3 copies of column 2, using all available rows
     [,1] [,2] [,3]
[1,]    4    4    4
[2,]    5    5    5
[3,]    6    6    6
```

Indexing even preserves order—that is, if asked for positions 2, 6 and 1, they will be returned in that order:

```
> v[c(2, 6, 1)]
[1] 9 8 1
> M[ , c(3, 2, 1)]  # Produce columns 3, 2 and 1 (in that order) with all rows
     [,1] [,2] [,3]
[1,]    7    4    1
[2,]    8    5    2
[3,]    9    6    3
```

Combined with the `order` function, this property can make sorting very simple.

```
> v[order(v)]
[1] 1 3 3 4 4 8 9
> sort.column <- rnorm(3)    # 3 random variables, normally distributed
> M2 <- cbind(M, sort.column)
> M2
           sort.column
[1,] 1 4 7   0.6690487
[2,] 2 5 8  -1.2692793
[3,] 3 6 9  -0.9675783
# Consider the column M2[, "sort.column"] as a vector:
#    0.6690487 -1.2692793 -0.9675783
# Determine the correct increasing ordering (by index) of this vector:
#    2 3 1
# Select rows from M2 in that order (and select all columns for these rows)
> M2[order(M2[, "sort.column"]), ]
           sort.column
[1,] 2 5 8  -1.2692793    # Row 2
[2,] 3 6 9  -0.9675783    # Row 3
[3,] 1 4 7   0.6690487    # Row 1
```

1.4 Boolean subsetting

In the previous sections we used an element's *index* to select it from a vector or matrix. It is sometimes more natural to select elements using boolean (*i.e.*, `TRUE` or `FALSE`) values rather than positions. `R` allows this by supplying a boolean vector `b` to the `[]` operators where `b[i]` is `TRUE` if the element at index `i` should be selected and `FALSE` otherwise. For example:

```
> v <- c(1, 9, 3, 3, 4, 8, 4)
# Positions 1, 3 and 4 will be selected but 2, 5, 6 and 7 will not
> v[c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE)]
[1] 1 3 3
```

In the above example, the boolean vector was exactly as long as `v`. Therefore, each element in `v` was paired with its own `TRUE` or `FALSE` value. If the boolean vector is shorter than `v`, it will be *recycled*:

```
# Select the odd positions
> v[c(TRUE, FALSE)]
[1] 1 3 4 4
```

This boolean syntax is most natural when using a condition—a boolean expression—to select values of interest. For example, it is natural to use the syntax to select only those values of `v` which are not `4`:

```
# Show the boolean vector that will be used to subset v
> v != 4     # "!=" means "not equal to"
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
# Do it
> v[v != 4]
[1] 1 9 3 3 8
```

Unlike positional indexing, boolean subsetting cannot change the order or frequency of elements. Elements are always returned in the same order as they are stored in the vector; and elements always appear exactly 1 or 0 times.

2 State

We have now seen multiple ways to create objects—either by making an entirely new object or by creating a copy of some subset of another object. Simply creating an object does not modify the *state* of the system; that is, once the object has been computed, it is forgotten unless a *state changing* action is taken. The only state changing action presented thus far is the assignment (`<-` or `=`) operator. The assignment operator has the following syntax:

`variable.name <- value.producing.statement`

This command is executed right-to-left: first, *`value.producing.statement`* is evaluated and then its result is *stored* as *`variable.name`*. If no variable named *`variable.name`* existed before this command, one is created; if *`variable.name`* did exist before this command, its old value is overwritten with the new value from *`value.producing.statement`*.[4] Unless *`variable.name`* is overwritten with the same value it previously held, the state of the system has changed.

2.1 State in R

With the ease of both creating new objects and assigning them to variables, the state of an `R` process can become very large very fast. `R` offers a number of tools to manage its state.

[4]Care should be taken when assigning to a variable that already exists. When the variable is overwritten, its data is lost and its original value must be recomputed. In sophisticated analyses, this can cost hours of computing time!

Function	Example	Description
`<-`	`x <- 1`	Add to or modify existing variables.
`rm`	`rm(x)`	Removes an existing variable.
`ls`	`ls()`	Lists all available variables.
`save`	`save(x, file="x.RData")`	Save one or more variables to disk. It is conventional to use the `.RData` extension for files containing `R` variables. The files written to disk can later be used with...
`load`	`load("x.RData")`	Load a state file (as produced by `save`) into the current state.
`save.image`	`save.image(file="state.RData")`	A convenience function: equivalent to `save` on all available variables.

3 Loops

The *loop* is an important example of a programming construct that is inherently state-dependent. Enclosing a series of commands (a *block*) in a loop allows the block to be repeated multiple times. *Looping*—repeating the block—continues so long as a certain expression evaluates to `TRUE`. Consider the following pseudo-program (the following is not a proper `R` program):

1 *command1*
2 *command2*
3 *command3*
4 if *expr* is `TRUE`, go back to `1`. Otherwise, go to `5`.
5 *command4*

The above program begins by executing *command1*, *command2* and *command3* in order. Once the program reaches line 4, it must make a decision. If *expr*—a boolean expression like `x>1`—is `FALSE`, then the program will move on to line 5 and excute *command4*. However, if *expr* evaluates to `TRUE`, the program will jump back to line 1, execute *command1-3* a second time, and then re-evaluate *expr* to decide if it should repeat the process yet again. This is the essence of looping.

Let us consider a more concrete example program (again, this is not a proper `R` program):

```
1 x <- 0
2 x <- x + 1
3 print(x)                                  # Displays the value of x
4 if x < 5 is TRUE go back to 2; otherwise, exit.
```

The program begins by creating a new variable `x` and setting it to `0`. Line 2 then sets `x` to 1 since `x + 1` evaluates to $0 + 1 = 1$. (Would line 2 make sense without line 1?) Line 3 displays `x`'s present value. Line 4 evaluates the expression `x < 5` to determine if it should exit or jump back to line 2. Since `x`'s present value is 1, `x < 5` evaluates to `TRUE` and the program returns to line 2. It is easy to verify by hand that this process continues to increase the value of `x` by 1 and print its value until the program exits at $x = 5$.

The loop in the above program is dependent on the state of `x`. If line 2 did not change the value of `x` at each iteration, the loop would execute lines 2-4 forever. The only loops that do not depend on the state of the system are loops with constant values for *expr*: *e.g.*, `0 < 1` or simply `FALSE`.

3.1 Loops in R

`R` provides three loop commands: `repeat`, `while` and `for`. `repeat` is rarely used, so we will focus on the `while` and `for` loops for now.

3.1.1 The `while` loop

`while` loops take the form:

```
while (expr) {
        command1
        ...
        commandN
}
```

where `expr` is a boolean expression (an expression that evaluates to `TRUE` or `FALSE`). The braces ({}) are not necessary if the block contains only a single command. First, `expr` is evaluated. If it resolves to `FALSE`, then the program skips all of the commands in the block and continues execution after the closing brace (}). If `expr` evaluates to `TRUE`, then `command1-commandN` are executed in sequence and `expr` is re-evaluated. The procedure is repeated while `expr` is `TRUE`.

The following `while` loop builds the value 2^4 in the variable `p`:

```
x <- 1
p <- 1
while (x < 5) {
        p <- p * 2
}
```

It is essential to note that the `while` loop's "value"[5] is *not* 2^4—the loop builds up the power of 2 in the variable `p` as it executes.

3.1.2 The `for` loop

`for` loops take the form:

```
for (variable in vector) {
        command1
        ...
        commandN
}
```

where `variable` is a dummy variable and `vector` is either a vector or a list. The braces ({}) are not necessary if the block contains only a single command. For each value x in `vector`, `R` will first set `variable <- x` and then run `command1` through `commandN`. For example, we can implement the `sum` function using a for loop. Let `v` be the vector we wish to sum; then

```
sum <- 0
for (x in v) {
        sum <- sum + x
}
```

computes the sum of `v` and stores it in a variable named `sum`.

[5] In some recent versions of `R`, `while` loops evaluate to the value of the last statement. Taking the value of a loop is very unnatural and should almost never be done. This behavior has been removed in the most recent `R` version: `while` loops now evaluate to `NULL`.