# CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

http://www.cs.utoronto.ca/∼radford/csc120/

Week 8

## Operations on Vectors

We've seen before that R can do many operations on entire vectors (or matrices), not just on single numbers. For example, we can add 1 to all elements of a vector:

```
> u <- c(3,5,1,9)
> v <- u + 1
> v
[1]  4  6  2 10
```

Instead of the statement `v <- u + 1` we could have written a loop:

```
> v <- u
> for (i in 1:length(v)) v[i] <- v[i] + 1
> v
[1]  4  6  2 10
```

But `v <- u + 1` is easier to write, easier to read, and also faster in R.

This isn't magic, though — there still **is** a loop hidden within the implementation of R, and in some other languages writing a loop yourself would be just as fast.

R has many other facilities for doing operations on vectors, matrices, or lists without having to write a loop, which often are also faster.

## Replacing Loops with “apply” Functions

Functions in the “apply” family take as arguments both a data structure and a function to apply to parts of the data structure — an example of “functional programming”, using functions to construct more complex operations.

The `lapply` and `sapply` functions can operate on a list, returning a list or vector of results of applying a given function to each element of the list:

```
> L <- list ("abc", 123, TRUE)
> lapply(L,is.numeric)           # Returns a list
[[1]]
[1] FALSE

[[2]]
[1] TRUE

[[3]]
[1] FALSE

> sapply(L,is.numeric)           # Tries to convert list to vector
[1] FALSE  TRUE FALSE
```

## More Arithmetic on Vectors and Matrices

As we saw before, all the basic arithmetic operations can be done on vectors, or on a vector and a scalar (or on matrices, or on a matrix and a scalar):

```
  > u <- c(3,5,1,9)
  > v <- c(10,100,1000,10000)
  > u * v
[1]    30   500  1000 90000
```

All the usual mathematical functions also can be applied to vectors or matrices:

```
  > v <- c(10,100,1000,10000)
  > log10(v)
  [1] 1 2 3 4
  > M <- matrix(1:9,nrow=3,ncol=3)
  > sqrt(M)
           [,1]     [,2]     [,3]
  [1,] 1.000000 2.000000 2.645751
  [2,] 1.414214 2.236068 2.828427
  [3,] 1.732051 2.449490 3.000000
```

## Vector Comparisons and Other Functions Giving Logical Values

The comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) also work with vectors (and matrices), producing a vector (or matrix) of `TRUE`, `FALSE`, or `NA` values:

```
> u <- c(3,5,1,9,NA,3)
> u < 4
[1]  TRUE FALSE  TRUE FALSE    NA  TRUE
```

Many functions that produce logical values also operate on vectors (and matrices):

```
> is.na(u)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

But some functions produce a single logical value that pertains to the entire vector, not many values, for each individual element:

```
> v <- c("fred","mary","bert")
> is.numeric(v)
[1] FALSE
> is.character(v)
[1] TRUE
```

## Logical Operators

There are operators that operate on logical values, returning logical values. For scalar operands, they are defined as follows:

| | |
|---|---|
| `!` | Logical "not": `TRUE` if its operand is `FALSE`, `FALSE` if its operand is `TRUE`. |
| `&` | Logical "and": `TRUE` only if both operands are `TRUE`. |
| `\|` | Logical "or": `TRUE` if either operand is `TRUE`. |

These can be applied to logical vectors, with the operations done on each element in turn:

```
> a <- c (TRUE,  TRUE,  FALSE, FALSE)
> b <- c (TRUE,  FALSE, TRUE,  FALSE)
> a & b
[1]  TRUE FALSE FALSE FALSE
> a | b
[1]  TRUE  TRUE  TRUE FALSE
> !a
[1] FALSE FALSE  TRUE  TRUE
```

# The Short-Circuit Logical Operators

The logical operators are useful in `if` and `while` conditions. Also, because we want only a single logical value for such a condition, we can use "short-circuit" versions of the "and" and "or" operators, written as `&&` and `||`, which don't evaluate their second operand if it's not needed to determine the result.

An example:

```
    if (percent < 0 || percent > 100)
        stop("percent must be between 0 and 100")
```

The `stop` happens if either `percent` is less than 0 or `percent` is greater than 100. We could use `|` rather than `||`, but if we use `||`, R doesn't bother to check whether `percent` is greater than 100 if it already knows `percent` is less than 0.

Using short-circuit operators can be faster, and also can avoid some spurious errors, as below:

```
    if (length(x) == 1 && x > 0) cat("x is OK!\n")
```

If we use `&` rather than `&&` above, we get an error if `length(x)` is 0, and a warning if `length(x)` is greater than 1. These are avoided with `&&`.

# Operations on Numeric Vectors that Produce One Number

R has several functions that take a numeric vector or matrix as their argument, and return a single number as their value, including:

| | |
|---|---|
| `sum` | finds the sum of all elements. |
| `prod` | finds the product of all elements. |
| `mean` | finds the mean (average) of all elements. |

For example:

```
> u <- c(3,5,1,9)
> sum(u)
[1] 18
```

This does pretty much the same thing as the following loop:

```
> s <- 0
> for (x in u) s <- s + x
> s
[1] 18
```

However, `sum(u)` is faster, and in some cases more accurate.

## Operations on Logical Vectors that Produce One Logical Value

R also has two functions that take a logical vector as their argument, and return a single logical value:

| | |
|---|---|
| `any` | Return `TRUE` if any elements are `TRUE` |
| `all` | Return `TRUE` if all elements are `TRUE` |

Looked at another way, `any` finds the “or” of all elements in its argument, and `all` finds the “and” of all elements.

Here’s an example of the use of these functions:

```
    check_age <- function (df) {
        if (any(is.na(df$age)))
            stop("Age is missing for some people")
        if (!all (df$age >= 0 & df$age < 150))
            stop("Age is invalid for some people")
}
```

Can you think of a way to replace the second `if` condition with one that uses `any` rather than `all`?

# Indexing Vectors and Matrices with Numeric Vectors

As mentioned before, you can get a subset of vector elements by indexing it with a vector of indexes.

```
> v <- c(66,33,99,10,12)
> v[c(3,1,2,1)]
[1] 99 66 33 66
> v[2:4]
[1] 33 99 10
```

Note: `v[[2:4]]` does not do the same thing — `[[` gets a *single* element from a vector or list. Since R treats a single number the same as a vector of length one, `v[2]` and `v[[2]]` do (almost) the same thing when `v` is a numeric vector.

You can also index with a vector of *negative* numbers. This gets you all elements *except* those whose indexes are in the index vector (negated):

```
> v[-3]
[1] 66 33 10 12
> v[c(-1,-length(v))]
[1] 33 99 10
```

Vector indexes can also be used as matrix row/column indexes (eg, `M[1:10,-2]`).

## Re-Ordering a Vector, Matrix, or Data Frame

We can change the order of elements in a matrix, or of rows in a matrix or data frame, using an index that is a *permutation* of the possible indexes.

One use is to change the order to be increasing in some variable. The `order` function produces the permutation needed to do this. For example:

```
> heights_and_weights
  name height weight
1 Fred     62    144
2 Mary     60    131
3  Joe     71    182
> by_weight <- order (heights_and_weights$weight)
> by_weight
[1] 2 1 3
> new <- heights_and_weights [by_weight, ]
> new
  name height weight
2 Mary     60    131
1 Fred     62    144
3  Joe     71    182
```

# Indexing Vectors and Matrices with Logical Vectors

You can also get a subset of vector elements using a logical vector as an index. Normally, the logical vector is the same length as the vector it indexes. The elements selected are those for which the corresponding index element is `TRUE`.

```
> v <- c(66,33,99,10,12)
> v[c(TRUE,FALSE,FALSE,TRUE,FALSE)]
[1] 66 10
> v[v<40]
[1] 33 10 12
```

Logical vectors can also be used to index rows or columns of a matrix:

```
> M <- matrix(1:6,nrow=2,ncol=3)
> M
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> M[,c(TRUE,FALSE,TRUE)]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
```

## Selecting a Subset of Rows in a Data Frame

Another use of logical indexes is in selecting a subset of rows in a data frame for which the variables have certain values.

For example, here we select only people with weight greater than 140:

```
> heights_and_weights
  name height weight
1 Fred     62    144
2 Mary     60    131
3  Joe     71    182
> heights_and_weights [heights_and_weights$weight > 140, ]
  name height weight
1 Fred     62    144
3  Joe     71    182
```

And here we get only people with weight greater than 140 and height less than 70:

```
> heights_and_weights [heights_and_weights$weight > 140
+                        & heights_and_weights$height < 70, ]
  name height weight
1 Fred     62    144
```

## Using Vector Indexes to Replace Elements in a Vector

Numeric and logical vectors can be used as indexes when we replace elements in a vector rather than get them out.

For example:

```
> v <- c(66,33,99,10,12)
> v[c(2,4,1)] <- c(100,200,300)
> v
[1] 300 100  99 200  12
> v[c(TRUE,FALSE,FALSE,FALSE,TRUE)] <- c(800,900)
> v
[1] 800 100  99 200 900
```

# Using Vector Indexes to Replace Elements in a Matrix

Vector indexes can be used to replace matrix elements too:

```
> M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> A
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> M[2:3,c(TRUE,FALSE,TRUE)] <- A
> M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]   10    5   30
[3,]   20    6   40
```

## Some Design Flaws in R

R is a very useful language, but like all programming languages, it's not perfect. Indeed, some of R's features are poorly designed, making it too easy to write code that doesn't always work.

I'll talk about two of these here:

- You can't get an empty vector when making a sequence with an expression like `i:j`.
- R will sometimes convert matrices to plain vectors when you don't want it to.

# The Problem of Reversing Sequences

The `:` operator will produce either an increasing sequence or a decreasing sequence, depending on whether the first operand is less or greater than the second:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

This may seem convenient, but it's a bad idea. When you use `:` in a program, you need to be sure which sort of sequence you're going to get!

## An Illustration of Why Reversing Sequences are Bad

Here's a function that is supposed to return a modified square matrix in which all the elements above the diagonal have been set to one:

```
ones_above_diagonal <- function (M) {
    n <- nrow(M)
    for (i in 1:n)
        for (j in (i+1):n)
            M[i,j] <- 1
    M
}
```

Here's what happens when we try to use it:

```
> ones_above_diagonal(matrix(0,nrow=4,ncol=4))
Error in M[i, j] <- 1 : subscript out of bounds
```

(The exact error message depends on the version of R used.)

Why the error? We need to get a zero-length sequence from `(i+1):n` when `i` equals `n`. But instead we get a sequence of length two, containing `n+1` and `n`.

How could we fix it?

## The Problem of Dropped Dimensions

When you index a matrix with a single row or column index, R converts the result to a vector, rather than keep it as a matrix.

Sometimes this is what you want:

```
> M <- matrix(1:6,nrow=2,ncol=3)
> M[1,2]
[1] 3
> M[1,2:ncol(M)]
[1] 3 5
```

But sometimes not:

```
> A <- M[,2:ncol(M)]
> A[1,1]
[1] 3
> B <- M[2:nrow(M),]
> B[1,1]
Error in B[1, 1] : incorrect number of dimensions
```

# Stopping R From Dropping Dimensions

You can tell R to not drop dimensions from a matrix with the `drop=FALSE` option:

```
> M <- matrix(1:6,nrow=2,ncol=3)
> M[,2:ncol(M)]
     [,1] [,2]
[1,]    3    5
[2,]    4    6
> M[2:nrow(M),]
[1] 2 4 6
> M[,2:ncol(M),drop=FALSE]
     [,1] [,2]
[1,]    3    5
[2,]    4    6
> M[2:nrow(M),,drop=FALSE]
     [,1] [,2] [,3]
[1,]    2    4    6
```

But adding `drop=FALSE` all the time makes everything longer and messier. So it's tempting not to. But then you may get unexpected bugs once in a while...