Data Programming with R

Isabella Gollini

Lecture 2 - Vectors, matrices and arrays

Vectors, matrices and arrays

- Accessing and manipulating
- Arithmetic and operations
- Using `all` and `any` - `NA` and `NULL` values
- Filtering
- `if` statements
- `apply`
- high dimensional arrays
- *Download R packages*

Why are we studying this?

- Vectors, matrices and arrays are the simplest way of storing data in R.
- Almost all the analysis we do later in the course requires us to be able to manipulate data objects efficiently.
- R handles these objects differently than other programming languages.
- Later in the course we will learn other ways to store more complex data (in a `list` or a `dataframe`).

Vectors

What are vectors?

- A vector is just a list of elements. We already created one in week 1:

```
x <- c(1, 2, 4)
```

- We can also create text vectors:

```
y <- c('cat','dog','rabbit')
```

- Vectors must always be of the same mode or type, i.e. you can't mix text with numbers.
- Use the `mode` function to tell you the type of values stored in a vector.
- The `length` function will tell you the number of elements in a vector.

Adding and deleting

- We can add to a vector by using the concatenate command we've already met:

```
x <- c(88, 5, 12, 13)
x <- c(x[1:3], 168, x[4])
```

- To return to the original `x`, we can delete the fourth element:

```
x <- x[-4]
```

Declarations

- In many programming languages you must first declare the variables (e.g. `x`) before you use them.
- You don't have to do this in R if you are using single-length vectors (known as scalars)
- However, if you are working with vectors or matrices you must declare them before allocating values.

```
y <- vector(length = 2)
y
```

```
## [1] FALSE FALSE
```

```
mode(y)
```

```
## [1] "logical"
```

```
y[1] <- 5
y
```

```
## [1] 5 0
```

```
y[2] <- 12
y
```

```
## [1]  5 12
```

or, if we already know that the our vector is numeric we can use:

```
y2 <- vector("numeric", length = 2)
y2
```

```
## [1] 0 0
```

```
y3 <- numeric(2)
y3
```

```
## [1] 0 0
```

Common operations

- Most of R's operations are vectorised, meaning we don't need to access each individual element:

```
x <- c(1, 2, 4)
y <- x + c(5, 0, -1)
```

- Subtraction (`-`), Multiplication (`*`) and division (`/`) work similarly, as do many other functions.

Note: those of you familiar with linear algebra will note that multiplication and division occur element-wise.

More on indexing

- We already know how to access individual elements of vectors and matrices using square brackets.
- There are many more options for doing this, e.g.:

```
y <- c(1.2, 3.9, 0.4, 0.12)
y[c(1, 3)]
```

```
## [1] 1.2 0.4
```

```
v <- 3:4
y[v]
```

```
## [1] 0.40 0.12
```

Generating vector sequences

- A simple way to generate sequences is via a colon:

```
2:7
```

```
## [1] 2 3 4 5 6 7
```

- This is used in `for` loops that we will cover later in the module.
- Another simple way is to use the `seq` function:

```
seq(from = 12, to = 30, by = 3)
```

```
## [1] 12 15 18 21 24 27 30
```

- The function `rep` creates repeated numbers:

```
rep(7, 4)
```

```
## [1] 7 7 7 7
```

Using `all()` and `any()`

- These two functions provide logical answers to statements:

```
x <- 1:10
any(x < 8)
```

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

```
all(x < 8)
```

```
## [1] FALSE
```

- In each case it first evaluates the statement inside the brackets, and then determines whether `any` or `all` of them are true.
- You can see what it is doing by first running:

```
x < 8
```

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

`NA` and `NULL` values

- R has two different placeholders for missing/unknown data, `NA` and `NULL`.
- `NA` is used specifically for missing values:

```
x <- c(88, NA, 12, 168, 13)
x
```

```
## [1]  88  NA  12 168  13
```

```r
mean(x)
```

```
## [1] NA
```

```r
mean(x, na.rm = TRUE)
```

```
## [1] 70.25
```

```r
x <- c(88, NULL, 12, 168, 13)
x
```

```
## [1]  88  12 168  13
```

```r
mean(x)
```

```
## [1] 70.25
```

More on NULL values

- Consider the following simple code:

```r
z <- NULL
for(i in 1:5) z <- c(z, i)
z
```

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

- Compare this with:

```r
z <- NA
for(i in 1:5) z <- c(z, i)
z
```

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

- In the first example `NULL` literally doesn't exist. In the second the `NA` is treated as an unknown missing value and included in `z`.

Examples

Example 1 - Finding the number of consecutive 1s in a vector.

- Suppose we are interested in finding the number of consecutive 1s of a certain length in a vector.
- E.g. the vector `(1, 0, 0, 1, 1, 1, 0, 1, 1)` has a run of length `3` in position `4` and runs of length `2` in position `4`, `5`, and `8`.

```r
findruns <- function(x, k) {
  n <- length(x)
  runs <- NULL
  for(i in 1:(n - k + 1)) {
    if(all(x[i:(i + k - 1)] == 1)) runs <- c(runs, i)
    }
  return(runs)
  }
```

Running the function

- We can run the function via:

```
y <- c(1, 0, 0, 1, 1, 1, 0, 1, 1)
findruns(y, 3)
```

```
## [1] 4
```

```
findruns(y, 2)
```

```
## [1] 4 5 8
```

```
findruns(y, 6)
```

```
## NULL
```

Note that this function uses the other functions `length`, `if`, `all`, and `return`.

- This function gets pretty slow for big data sets as it has a loop and a poor re-allocation step. Later in the course we will talk about how to speed up this type of function.

Example 2 - Predicting discrete-valued time series.

- Suppose we observe a vector of 1s and 0s where `1 = rain` and `0 = no rain`. We have 1 observation each day.
- We'll use a simple prediction rule. If the number of 1s in the previous `k` days is at least `k / 2`, then we'll predict `1` for the next day, and `0` otherwise.
- E.g., if we set `k = 3` and observe `(0, 1, 1)` for the last 3 days then we'll predict the next period to be `1`.
- If we had some training data, we could use different values of `k` and find out which value worked best (i.e. produced the least wrong answers)

Predicting discrete-valued time series

- Here's a function which produces predictions and then works out how wrong we are:

```
pred <- function(x, k) {
  n <- length(x)
  k2 <- k / 2
  pred <- vector(length = n - k)
  for(i in 1:(n - k)) {
    if(sum(x[i:(i + (k - 1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
    }
  return(mean(abs(pred - x[(k + 1):n])))
}
```

```
pred(y, 3)
```

```
## [1] 0.5
```

- Again, this `function` will be slow `for` large data sets (i.e. where we have many observations) because of the `for` loop in the middle and the repeated use of `sum`.

More on `if... else`

- In general the sintax of the `if... else` statement is:

```
if(condition){
  expression1 # when the condition is TRUE
  } else {
    expression2 # only evaluated when the condition is FALSE
    }
```

- Here `condition` is a logical statement, `expression1` is the expression to be evaluated if the statement is `TRUE` and `expression2` is the expression to be evaluated if the statement is `FALSE`.
- See the screencast for a detailed explanation of the function `pred`.

Vectorised operations

Vector in, vector out

Filtering

`ifelse`

Vectorised operations

- The most useful and powerful thing about R are the vectorised operations. This means that a function applied to a vector is applied individually to each element without the need for a loop.
- Some simple examples:

```
u <- 1:5
w <- function(x) return(x + 1)
w(u)
```

```
## [1] 2 3 4 5 6
```

```
sqrt(1:4)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
```

- Many R functions, including addition, subtraction, multiplication and division are all vectorised.
- As above, we can write our own functions which, provided they build on other vectorised functions, will also be vectorised.

Vectorisation and recycling

- We can usually mix vectors and scalars (vectors of length 1) without any error messages:

```
y <- c(12, 5, 13)
y + 3
```

```
## [1] 15  8 16
```

- Here `3` is automatically 'recycled' three times so that it matches `y`.
- Similarly if we have vectors of different lengths R will automatically recycle the shorter one:

```
c(1, 2, 4) + c(6, 0, 9, 20, 22)
```

```
## Warning in c(1, 2, 4) + c(6, 0, 9, 20, 22): longer object length is not a
## multiple of shorter object length
```

```
## [1]  7  2 13 21 24
```

- However, it will produce a warning when doing so!

Filtering

- Very often we have a data set some of which we do not wish to include in our analysis.
- We have already used filtering when we used square brackets to select part of an object:

```
z <- c(5, 2, -3, 8)
w <- z[z^2 > 8]
w
```

```
## [1]  5 -3  8
```

The reason this command works is because the statement in the square brackets produces a logical vector:

```
z^2 > 8
```

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

Filtering with `subset`

- The `subset` function is really a neat shortcut for square brackets, but be aware that by default it removes NA values:

```
x <- c(6, 1:3, NA, 12)
x
```

```
## [1]  6  1  2  3 NA 12
```

```
x[x > 5]
```

```
## [1]  6 NA 12
```

```
subset(x, x > 5)
```

```
## [1]  6 12
```

Filtering with `which`

- The `which` function gives you the element numbers that satisfy a certain condition.

```
z <- c(5, 2, -3, 8)
z^2
```

```
## [1] 25  4  9 64
```

```
which(z^2 > 8)
```

```
## [1] 1 3 4
```

```
z[which(z^2 > 8)]
```

```
## [1]  5 -3  8
```

- The function is returning the element numbers in the logical statement that are true.
- If you are dealing with matrices, then `which` will return both the row and the column providing the extra argument `arr.ind` is set to `TRUE`.

ifelse

- Like many programming languages, R has an if- then-else function called `ifelse`. The format is:

```
ifelse(test, yes, no)
```

- Here `test` is a logical statement, `yes` are the values to be supplied if the statement is true and `no` the values to be supplied if the statement is false.
- Example:

```
x <- 1:10
y <- ifelse(x %% 2 == 0, "even", "odd")
y
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

- Understanding commands for filtering is really important when creating fast R code - see the screencast for more detail.

Matrices

Matrix operations

Filtering

`apply`

Adding and deleting rows and columns

Naming rows and columns

High dimensional arrays

What are matrices?

- A matrix is a rectangular arrangement of numbers. You can create one using the matrix command:

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

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

- This creates a matrix of the numbers 1 to 6 with 3 rows and 2 columns.
- Notice that by default the matrix is filled by column.
- If we want to fill the matrix by row we use:

```
m <- matrix(1:6, nrow = 3, ncol = 2, byrow = TRUE)
```

Matrices

- Entries can be accessed in a similar way to vectors, e.g:

```
M[1,]
```

```
## [1] 1 4
```

```
M[c(1, 3), 2]
```

```
## [1] 4 6
```

- The first command gives the first row of `M`.
- The second command gives the first and third rows and the second column of `M`.
- R treats a matrix as though it's really a vector, so that `M[3:5]` is a perfectly valid command.
- There are commands similar to the function `c` to add columns or rows to a matrix, using the functions `cbind` and `rbind`.

Matrix operations

- In later topics (such as multiple regression) we need to perform linear algebra operations such as matrix multiplication, and addition.
- R handles matrix multiplication via a special operator: `%*%`

```
t(M) %*% M
```

```
##      [,1] [,2]
## [1,]   14   32
## [2,]   32   77
```

- This gives the transpose of the matrix M multiplied by `M`.
- Other operations, e.g. `3 * M` or `M + 2` occur element-wise.

`apply`

- The R function `apply` allows you to perform functions on the rows or columns of a matrix. It is very powerful and very useful.
- The general structure of an `apply` command is:

```
apply(m, dimcode, f, fargs)
```

- Here `m` is a matrix, `dimcode` is `1` for rows and `2` for columns, `f` is a function and `fargs` is a set of optional arguments that are required by `f`.

```
apply(M, 2, mean)
```

```
## [1] 2 5
```

- This applies the function mean to the matrix `M` and performs the operation on each column.

More on `apply`

- You can also give apply your own created function:

```
f <- function(x) x / max(x)
apply(M, 1, f)
```

```
##      [,1] [,2] [,3]
## [1,] 0.25  0.4  0.5
## [2,] 1.00  1.0  1.0
```

- Here the matrix `M` has the function `x / max(x)` applied to each of its rows.

Differences between vectors and matrices

- Many vector commands still work on matrices, e.g.

```
length(M)
```

```
## [1] 6
```

- But as `M` is a matrix it has other attributes, for example:

```
dim(M)
```

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

- The `dim` function tells you the dimension, here `M` has `3` rows and `2` columns
- Similarly `nrow(M)` and `ncol(M)` will tell you directly how many rows and how many columns `M` has.

Common problems with matrices

- We can extract a column (or a row) from a matrix via, e.g.:

```
r2 <- M[2,]
r2
```

```
## [1] 2 5
```

- Unfortunately `r2` is not a matrix, it's a vector. The `str` function (short for structure) gives:

```
str(M)
```

```
##  int [1:3, 1:2] 1 2 3 4 5 6
```

```
str(r2)
```

```
##  int [1:2] 2 5
```

- One solution is to use `r2 <- M[2, ,drop = FALSE]` or `r2 <- as.matrix(M[2,])`

Naming rows and columns

- It is often helpful to give the rows or columns of a matrix some names:

```
colnames(M) <- c('a', 'b')
rownames(M) = c('c', 'd', 'e')
M
```

```
##   a b
## c 1 4
## d 2 5
## e 3 6
```

- The functions `rownames()` and `colnames()` also work to extract the names from a matrix.
- See the screencast for lots more ways of creating, filtering and manipulating matrices.

Arrays

- Most of the time, we will be using vectors and matrices to analyse data, but occasionally we need to work in higher dimensions.

- Suppose we take measurements on body temperature (in degrees C) and heart rate (in beats per minute) on 3 people when resting and when active. The data might look as follows:

```
resting <- matrix(
  c(36.1, 36.0, 36.3, 68, 65, 85),
  nrow = 3, ncol = 2)
resting
```

```
##      [,1] [,2]
## [1,] 36.1   68
## [2,] 36.0   65
## [3,] 36.3   85
```

```
active <-  matrix(
  c(36.3, 36.5, 37.3, 98, 112, 135),
  nrow = 3, ncol = 2)
active
```

```
##      [,1] [,2]
## [1,] 36.3   98
## [2,] 36.5  112
## [3,] 37.3  135
```

- We can put them together via an array:

```
A <-  array(data = c(resting, active), dim = c(3, 2, 2))
```

More on arrays

- The array will now have three dimensions:

```
dim(A)
```

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

- Printing the data (type `A`) will show it layer by layer (and can often take up a lots of screen space).
- We can access and manipulate it in exactly the same way as for matrices:

```
A[3, 1, 2]
```

```
## [1] 37.3
```

- This gives the result for the third person, first measurement (temperature) at the second time point (active).

Example

Image mainipulation

- A common use for matrices is to store images.
- You can think of a black and white image as an `n` by `m` matrix where the numbers in the matrix represent how white that pixel should be.
- The following code uses the package `pixmap`

The first time we use a package we have to install it (see the screencast for more details on how to install this in R)

```
install.packages("pixmap")
```

Then we can use the function `read.pnm` included in the package `pixmap` to create an image with R:

```
library(pixmap)
x <- read.pnm(system.file("pictures/logo.ppm", package = "pixmap"))
y <- as(x, "pixmapGrey")
plot(y)
```

Looking at the structure of y shows the matrix used to create the plot:

```
str(y)
```

```
## Formal class 'pixmapGrey' [package "pixmap"] with 6 slots
##   ..@ grey    : num [1:77, 1:101] 1 1 0.999 0.999 1 ...
##   ..@ channels: chr "grey"
##   ..@ size    : int [1:2] 77 101
##   ..@ cellres : num [1:2] 1 1
##   ..@ bbox    : num [1:4] 0 0 101 77
##   ..@ bbcent  : logi FALSE
```

(Note: this package uses something called the `S4` class which requires `@` rather than `$` to access components).

- The `grey` component here stores the picture as numbers between `1` and `0`. We can access them in exactly the same way as a standard matrix.
- By changing the elements of the matrix, we change the picture, e.g.:

```
y2 <- y
y2@grey <- 1 - y2@grey # Creates a negative
plot(y2)
```

```
y3 <- y
y3@grey <- 0.8 * y3@grey # Makes it darker
plot(y3)
```

```
y4 <- y
y4@grey[y4@grey < 0.4] <- 0 # Makes dark areas black
plot(y4)
```

See if you can find other fun ways to play with the image by using `apply` or other functions we have met today.

Lessons from this week

- We now know how to create and manipulate vectors, matrices and arrays.
- Lots of new functions! (In particular `apply`, `which`, `ifelse`).
- R vectorises operations which makes code run faster.
- We now how to install and load R packages