Introduction to programming in R: vectors and data frames

Erik Spence

SciNet HPC Consortium

22 February 2023

Today's slides

Today's slides can be found here. Go to the "Introduction to programming in R" page, under "Lectures", "Data frames".

https://scinet.courses/1277

About today's class

Today's class will explore the wild wild world of:

- Vectors.
- Slicing.
- Data frames.

R vectors

Vectors are baked right into R:

- Homogeneous (same type).
- Compact.
- Not nested.

```
> a <- c(1,2,3)
> b <- c("Hello", "World", "From", "A", "Vector")
> str(b)
chr [1:5] "Hello" "World" "From" "A" "Vector"
> d <- 1:17
> str(d)
int [1:17] 1 2 3 4 5 6 7 8 9 10 ...
>
```

The "c" command combines values into a vector or list.

R vectors, continued

There are many ways to create vectors in R:

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
>
> seq(2, 20, 4)
[1] 2 6 10 14 18
>
> paste("A", 1:5, sep = "")
[1] "A1" "A2" "A3" "A4" "A5"
>
> rep(letters[1:5], 3)
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d" "e"
>
> is.vector(1:10)
[1] TRUE
>
```

Using sample to create vectors

The "sample" function samples from a vector:

- By default, sample removes the previously sampled elements from the set.
- As such, you can't sample more than the set size.
- To keep sampled elements in the set, use the "replace = TRUE" argument.

```
>
> sample(1:10, 4)
[1] 10 5 4 7
>
> sample(1:10, 4)
[1] 9 5 3 8
>
> sample(1:10, 4, replace = TRUE)
[1] 7 10 3 7
>
> sample(c(T, F), 4, replace = TRUE)
[1] TRUE FALSE FALSE FALSE
>
```

Slicing using integers

Use single square brackets to access the elements of a vector.

Slicing means pulling out just the elements of the vector that you want (filtering the values).

- You can use vectors of integers to slice.
- This will return the values at the given indices.

Negative indices indicate entries you do not want returned.

All of these slicing techniques also work on data frames.

```
> a <- seq(2, 20, 4)
> a
[1] 2 6 10 14 18
>
> a[4]
[1] 14
> a[2:4]
[1] 6 10 14
>
> a[c(1, 2, 4)]
[1] 2 6 14
>
> a[-c(1, 2, 3)]
[1] 14 18
>
> a[seq(1, 5, 2)]
[1] 2 10 18
```

Slicing using booleans

You can also use vectors of booleans to slice your vectors.

- The statement "a < 8" returns a vector of booleans, indicating where the elements of 'a' meet the criterion in question.
- When slicing using vectors of booleans, every element where the boolean vector is TRUE will be returned.
- The "greater than or equal to" operator is given by the ">=" symbol.

```
>
> a
[1] 2 6 10 14 18
>
> indices <- a < 8
>
> indices
[1] TRUE TRUE FALSE FALSE FALSE
>
> a[indices]
[1] 2 6
>
> a[a >= 10]
[1] 10 14 18
>
```

Slicing using booleans, continued

All manner of boolean operations can be combined to slice vectors.

- The "==" is the equivalence test ("is this equal to this?"). "!=" does the opposite.
- The & symbol is the "AND" operator.
- The | symbol is the "OR" operator.
- The "which" command will give the indices of the TRUE entries.
- Generally speaking, you don't need to use the indices to slice your data, boolean vectors are simpler and easier.

```
> b <- seq(1, 60, 13)
> b
[1] 1 14 27 40 53
>
> b == 38
[1] FALSE FALSE FALSE FALSE FALSE
>
> (b > 5) & (b < 50)
[1] FALSE TRUE TRUE TRUE FALSE
>
> b[(b < 10) | (b > 30)]
[1] 1 40 53
>
> which((b < 10) | (b > 30))
[1] 1 4 5
>
```

Appending to vectors

You can add elements to the end of existing vectors:

- Use sparingly! It's better to fill the whole length you need first, using seq() or rep(), rather than set elements as needed.
- Increasing length of vector/list one at a time is:
  - slow
  - at risk of causing memory problems
- Recall that the # symbol starts comments.

```
>
> a <- c(1, 2, 3)
>
> # probably bad, certainly slow
> a <- c(a, 4)
> a <- c(a, 5)
> a
[1] 1 2 3 4 5
>
> # probably bad,
> # certainly funny-looking
> a[length(a) + 1] <- 6
> a
[1] 1 2 3 4 5 6
>
```

Appending to vectors, continued

It's much better to allocate your vector once, and then set the elements as you go.

If you extend your vector one element at a time, the contents of the vector must be copied each time the vector is extended. This is slow.

```
>
> # one way
> a <- vector(length = 5)
>
> # another way
> a <- rep(0,5)
>
> a[4] <- 4
> a[5] <- 5
> a
[1] 0 0 0 4 5
>
```

Not Available (NA)

Let's try extending the vector by another 3 items, and only set the last one.

We can use the "is.na" function to pick out NAs.

Recall that the "!" symbol is the "NOT" operator.

Math operations on NAs will usually return NA; but most generally have built-in optional ways of dealing with them.

Use the "help(sum)" to learn the optional arguments of the "sum" function.

```
> a[length(a) + 3] <- 9
>
> a
[1] 0 0 0 4 5 NA NA 9
>
> is.na(a)
[1] FALSE FALSE FALSE FALSE FALSE TRUE
TRUE FALSE
>
> a[!is.na(a)]
[1] 0 0 0 4 5 9
>
> sum(a)
[1] NA
>
> sum(a, na.rm = TRUE)
[1] 18
```

R vectors behave intuitively

Most operations happen automatically on all elements of a vector.

All operations either happen to all the elements or the elements are operated on one-by-one, depending on the situation.

Only in very exceptional cases is it necessary to loop over vectors or data frames.

```
> a <- 1:5 + 1
> a
[1] 2 3 4 5 6
>
> b <- rep(2.,5)
>
> a * b
[1] 4 6 8 10 12
>
> sin(a)
[1] 0.9092974 0.1411200 -0.7568025
[4] -0.9589243 -0.2794155
>
> a / 2
[1] 1.0 1.5 2.0 2.5 3.0
>
```

Use vectors instead of loops!

Whenver possible, operate on whole vectors ('vectorization') rather than looping and operating one element at a time.

- Your computer has built-in abilities which speed up vectorized calculations.
- The difference, especially on large amounts of data, can be enormous.

```
> a <- 3:7;       b <- 6:10
> e <- rep(0,5)
>
> seq_along(a)
[1] 1 2 3 4 5
>
> # do this!
> d <- a * b
>
> # don't do this!
> for (i in seq_along(a)) {
+     e[i] <- a[i] * b[i] }
>
> d
[1] 18 28 40 54 70
> e
[1] 18 28 40 54 70
```

Vector recycling

Oddly, R will perform vector "recycling" if you attempt to operate on two vectors of incompatible lengths:

- R will take the shorter of the two vectors, and repeat elements until they are both the same length.
- If the length of the shorter is not a multiple of the longer, you'll get a warning.
- For the sake of clarity it would be best not to do this. Use the "rep" function to repeat a vector if need be.

```
>
> 1:3 + 1:6
[1] 2 4 6 5 7 9
>
> c(1:3, 1:3) + 1:6
[1] 2 4 6 5 7 9
>
> 1:3 + 1:5
[1] 2 4 6 5 7
Warning message:
In 1:3 + 1:5 :
longer object length is not a multiple of
shorter object length
>
> 1:3 / 1:6
[1] 1.00 1.00 1.00 0.25 0.40 0.50
>
```

Data frames

Data frames are a building block for data analysis in R.

A data frame is a named list of vectors (similar to a spreadsheet). Each vector (a column of the frame) has the same length, but different columns may have different types.

```
>
> class(trees)
[1] "data.frame"
>
> str(trees)
'data.frame':  31 obs.  of 3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
>
```

Data frames, continued

You build your own data frames by specifying the names of the columns when you declare the data frame.

- The "names" and "colnames" functions give the names of the data frame columns.
- The "rownames" function gives the name of the rows, though ther aren't any for the trees data set.
- The "nrow" function gives the number of rows.

```
>
> my.df <- data.frame(pants = 1:3,
+           name = c("Larry", "Susie", "Bob"))
>
> names(my.df)
[1] "pants" "name"
>
> colnames(my.df)
[1] "pants" "name"
>
> rownames(data)
[1] "1" "2" "3"
>
> nrow(data)
[1] 3
>
```

Data frames, continued more

Data frames are the bread-and-butter of R. They are quite intuitive.

- You can add a column to your data frame by just referencing it.
- Use the "rbind" ("row bind") function to add a row to an existing data frame.
- The rbind function expects 2 data frames, with the same column names.
- To combine two data frames with dissimilar column names, use the "merge" function.

```
> my.df$Socks <- "argyle"
> names(my.df)
[1] "pants" "name" "Socks"
> my.df
  pants  name  Socks
1     1 Larry argyle
2     2 Susie argyle
3     3   Bob argyle
>
> my.df2 <- rbind(my.df, data.frame(pants = 6,
+       name = "Jane", Socks = "black"))
> my.df2
  pants  name  Socks
1     1 Larry argyle
2     2 Susie argyle
3     3   Bob argyle
4     6  Jane  black
```

Data frames, accessing columns

You can access the columns of a data frame one of several ways:

- Using the $ to indicate the column.
- Using the string name of the column.
- Using the index of the column.
- You can use a vector of string column names or indices to request multiple columns.
- Generally speaking, it's easier to remember the names of columns, rather than column indices, so I usually just use column names.

```
> str(trees)
'data.frame':  31 obs.  of 3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 ...
 $ Height: num  70 65 63 72 81 83 66 ...
 $ Volume: num  10.3 10.3 10.2 16.4 ...
>
> trees$Girth
[1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 ...
> trees[, "Girth"]
[1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 ...
> trees[, 1]
[1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 ...
> trees[, c("Height", "Volume")]
   Height Volume
1      70   10.3
2      65   10.3
⋮
```

Data frames, accessing columns within functions

If you use the $ to access a column within a function, you must "hard code" the name of the column into the function. This is not ideal.

Instead, it is better to use the column's name, as a string, to access the column within the function.

This gives the flexibility of changing the name of the column that is accessed by the function.

```
# MyScript1.R

# Not the best way.
some.analysis.1 <- function(input.data) {
  # This function can only work on "somecolumn".
  a <- input.data$somecolumn
  b <- some.other.analysis(a)
}

# A better way.
some.analysis.2 <- function(input.data,
                            working.col = "somecolumn") {
  # This function can work on any column.
  a <- input.data[, working.col]
  b <- some.other.analysis(a)
}
```

Slicing data frames

Accessing parts of the data frame makes a lot more sense when you remember it's just a named list of vectors.

Recall that data frames are 2 dimensional:

- the first entry in the square brackets is the row,
- the second entry is the column.

We can use slicing to rip out the rows of the data that we're interested in.

```
>
> trees[1:3, "Girth"]
[1] 8.3 8.6 8.8
>
> trees[2,]
   Girth Height Volume
2    8.6     65   10.3
>
> trees$Girth[trees$Height > 80]
[1] 10.7 10.8 12.9 13.3 17.3 17.5 20.6
>
> new.data <- trees[trees$Volume > 40 &
+                   trees$Girth <= 14, ]
>
```

Updating data frames

Performance tip: While you can update individual items in a data frame via direct access:

```
> data <- trees
> data[1, "Girth"] <- 8.7
```

It turns out this is extremely slow and memory intensive. If you have do a number of such updates, try to minimize the number of updates to the data frame.

It's better to extract a column, update it, and then update the whole column at once:

```
> Girth <- data$Girth
> Girth <- 2. * Girth + 1
> data$Girth <- Girth
```

Getting external data

Getting data from online is as simple as putting in the URL, or file name:

```
> data <- read.csv("https://support.scinet.utoronto.ca/~ejspence/Dental-2011-2012.csv")
> str(data)
 $ Quarter:  Factor w/ 3 levels "Q1","Q2","Q3":  1 1 1 1 1 1 1 1 ...
                  ⋮
 $ Total   : num 9317 14948 23136 18546 40536 ...
> colnames(data)
 [1] "Quarter"                    "Year"                    "Data"
 [4] "CCG_Code"                   "AT_CODE"                 "Region_Code"
 [7] "Patient_type"               "Band_1"                  "Band_2"
[10] "Band_3"                     "Urgent_Occasional"       "Free___Arrest_of_Bleeding"
[13] "Free___Bridge_Repairs"      "Free___Denture_Repair"   "Free___Prescription_Issue"
[16] "Free___Removal_of_sutures"  "Total"
```

The original URL for this data is http://www.hscic.gov.uk/catalogue/PUB07163/nhs-dent-stat-udas-eng-2011-2012-anx5.csv.

Excel files

You can also read Excel files using R, though not out of box.

There are many packages out there that will do this, but you'll need to download them separately.

- readxl
- gdata
- XLConnect
- xlsx

If your data is in Excel you should input your data into Excel but do your analysis in R.

```
>
> install.packages("readxl")
>
> library(readxl)
>
> data = read_excel('marks.xls', sheet = 1)
>
> is.data.frame(data)
[1] TRUE
>
> names(data)
[1] "First Name" "Last Name" "username" "Mark"
[5] "Date submitted" "Days late" "Comments"
>
```