Miscellaneous Tips and Tricks in R

EC 425/525, Lab 7

Edward Rubin
16 May 2019

Prologue

Schedule

Last time

Simulation in R

Today

Helpful tips and tricks in R

Tips and tricks

Tips and tricks

The apply family

In general, `for` loops are not the "preferred" route in R.

Tips and tricks

The apply family

In general, `for` loops are not the "preferred" route in `R`.

1. Many functions are vectorized—you can apply a function over a vector.

Tips and tricks

The apply family

In general, `for` loops are not the "preferred" route in `R`.

1. Many functions are vectorized—you can apply a function over a vector. *E.g.*, the square root of the numbers from 1 to 10: `sqrt(1:10)`.

Tips and tricks

The apply family

In general, `for` loops are not the "preferred" route in `R`.

1. Many functions are vectorized—you can apply a function over a vector. *E.g.*, the square root of the numbers from 1 to 10: `sqrt(1:10)`.

2. That said, sometimes you just gotta loop.

Tips and tricks

The apply family

In general, `for` loops are not the "preferred" route in `R`.

1. Many functions are vectorized—you can apply a function over a vector.
   E.g., the square root of the numbers from 1 to 10: `sqrt(1:10)`.

2. That said, sometimes you just gotta loop.
   For these situations, `base R` offers a family of `apply` functions.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Example `toupper()` capitalizes characters

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Example `toupper()` capitalizes characters, *e.g.*, `toupper("a")` yields `"A"`.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Example `toupper()` capitalizes characters, *e.g.*, `toupper("a")` yields `"A"`.

```
lapply(X = c("a", "pig"), FUN = toupper)
```

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Example `toupper()` capitalizes characters, *e.g.*, `toupper("a")` yields `"A"`.

`lapply(X = c("a", "pig"), FUN = toupper)` returns `list("A", "PIG")`.

Tips and tricks

The apply family

The `apply` family *applies* a function over a vector, list, data frame, *etc.*

For example, `lapply()` takes two arguments: `X` and `FUN`.

- `X` A vector/list of values.
- `FUN` The function you want to evaluate on each value of `X`.

`lapply()` returns a list of the results.

Example `toupper()` capitalizes characters, *e.g.*, `toupper("a")` yields `"A"`.

`lapply(X = c("a", "pig"), FUN = toupper)` returns `list("A", "PIG")`.

Note This is a silly example, as you can directly use `toupper()` on vectors.

Tips and tricks

Plain apply

The related `apply()` function *applies* a given function (`FUN`) along the margins (`MARGIN`) of a given array/matrix (`X`).

Tips and tricks

Plain apply

The related `apply()` function *applies* a given function (`FUN`) along the margins (`MARGIN`) of a given array/matrix (`X`).

Your options for `MARGIN` are `1` for rows and `2` for columns.

Tips and tricks

Plain apply

The related `apply()` function *applies* a given function (`FUN`) along the margins (`MARGIN`) of a given array/matrix (`X`).

Your options for `MARGIN` are `1` for rows and `2` for columns.

Example Let's find the maximum value in each row of a matrix.

```
# Create a matrix
ex_matrix ← matrix(data = 1:16, nrow = 4, byrow = T)
# Find the maximum value in each row.
apply(X = ex_matrix, MARGIN = 1, FUN = max)
```

```
#> [1]  4  8 12 16
```

Tips and tricks

Multiple apply

Like `lapply()`, `mapply()` repeatedly evaluates a function (`FUN`) for each value in a vector of inputs.

Tips and tricks

Multiple apply

Like `lapply()`, `mapply()` repeatedly evaluates a function (`FUN`) for each value in a vector of inputs.

However, `mapply()` allows you to evaluate across **multiple** vectors.

Tips and tricks

Multiple apply

Like `lapply()`, `mapply()` repeatedly evaluates a function (`FUN`) for each value in a vector of inputs.

However, `mapply()` allows you to evaluate across **multiple** vectors.

In addition `mapply()` allows you to dictate whether/how the results are simplified (*e.g.*, `SIMPLIFY = T` for vector or matrix) or kept as a `list`.

Tips and tricks

Multiple apply

Like `lapply()`, `mapply()` repeatedly evaluates a function (`FUN`) for each value in a vector of inputs.

However, `mapply()` allows you to evaluate across **multiple** vectors.

In addition `mapply()` allows you to dictate whether/how the results are simplified (*e.g.*, `SIMPLIFY = T` for vector or matrix) or kept as a `list`.

Example Random normal draws with different means and variances.

```
mapply(FUN = rnorm, n = 1, mean = c(0, 10, 20), sd = 1:3)
```

```
#> [1]  0.9761572  8.0805632 20.0179475
```

Tips and tricks

Custom apply

All of our examples used already-defined functions for `FUN`, *e.g.*,

Tips and tricks

Custom apply

All of our examples used already-defined functions for `FUN`, *e.g.*,

```
lapply(X = c("a", "pig"), FUN = toupper)
```

Tips and tricks

Custom apply

All of our examples used already-defined functions for `FUN`, *e.g.*,

```
lapply(X = c("a", "pig"), FUN = toupper)
```

Alternatively, you define your own function at `FUN`, *e.g.*,

Tips and tricks

Custom apply

All of our examples used already-defined functions for `FUN`, *e.g.*,

```
lapply(X = c("a", "pig"), FUN = toupper)
```

Alternatively, you define your own function at `FUN`, *e.g.*,

```
lapply(X = 1:2, FUN = function(i) {i > 1})
```

```
#> [[1]]
#> [1] FALSE
#>
#> [[2]]
#> [1] TRUE
```

Tips and tricks

Other packages

Other packages offer similar (and parallelized) functions.

`base`

`lapply()`

`apply()`

`mapply()`

Tips and tricks

Other packages

Other packages offer similar (and parallelized) functions.

`base`	`purrr` / `furrr`
`lapply()`	`map()`
`apply()`	?
`mapply()`	`map2()`

Tips and tricks

Other packages

Other packages offer similar (and parallelized) functions.

`base`	`purrr` / `furrr`	`future.apply`
`lapply()`	`map()`	`future_lapply()`
`apply()`	?	`future_apply()`
`mapply()`	`map2()`	`future_mapply()`

Tips and tricks

Other packages

Other packages offer similar (and parallelized) functions.

base	purrr / furrr	future.apply	parallel
`lapply()`	`map()`	`future_lapply()`	`mclapply()`
`apply()`	?	`future_apply()`	`mcapply()`
`mapply()`	`map2()`	`future_mapply()`	`mcmapply()`

Tips and tricks

`for()` loops

However, if you're really committed to running for loops, the syntax is

```
# Create an empty vector
our_vector ← c()
# Run the for loop for some numbers
for (i in c(1, 1, 2, 3, 5, 8) {
  # Print 'i'
  print(i)
  # Append 'i' to the end of our_vector
  our_vector ← c(our_vector, i)
})
```

Tips and tricks

Lists and unlisting

Lists (*e.g.*, as outputted by `lapply()`) can be helpful—but they can also be fairly annoying.

Tips and tricks

Lists and unlisting

Lists (*e.g.*, as outputted by `lapply()`) can be helpful—but they can also be fairly annoying. Enter `unlist()`.

Tips and tricks

Lists and unlisting

Lists (*e.g.*, as outputted by `lapply()`) can be helpful—but they can also be fairly annoying. Enter `unlist()`.

List output

```
lapply(
  X = 1:2,
  FUN = as.character
)
```

```
#> [[1]]
#> [1] "1"
#>
#> [[2]]
#> [1] "2"
```

Tips and tricks

Lists and unlisting

Lists (*e.g.*, as outputted by `lapply()`) can be helpful—but they can also be fairly annoying. Enter `unlist()`.

List output

```
lapply(
  X = 1:2,
  FUN = as.character
)
```

```
#> [[1]]
#> [1] "1"
#>
#> [[2]]
#> [1] "2"
```

`unlist()`-ing to vector

```
lapply(
  X = 1:2,
  FUN = as.character
) %>% unlist()
```

```
#> [1] "1" "2"
```

Tips and tricks

From lists to data frames

Sometimes you don't want to entirely `unlist()` a list.

Tips and tricks

From lists to data frames

Sometimes you don't want to entirely `unlist()` a list.

For example, you might have a list of data frames that you want to bind into a new data frame.

Tips and tricks

From lists to data frames

Sometimes you don't want to entirely `unlist()` a list.

For example, you might have a list of data frames that you want to bind into a new data frame.

In this case, you can use `bind_rows()` or `bind_cols()` from `dplyr`.

Tips and tricks

From lists to data frames

Sometimes you don't want to entirely `unlist()` a list.

For example, you might have a list of data frames that you want to bind into a new data frame.

In this case, you can use `bind_rows()` or `bind_cols()` from `dplyr`.

Alternatively, you might be able to make use of `map_dfr()` or `map_dfc()`.

Tips and tricks

Indexing lists

Also Don't forget that you can index lists using double-brackets.

```
# Capitalize the alphabet
our_list ← lapply(X = letters, FUN = toupper)
# The third letter
our_list[[3]]
```

```
#> [1] "C"
```

Tips and tricks

Logical vectors and `which()`

Finally, the simply function `which()` can be surprisingly helpful.

Tips and tricks

Logical vectors and `which()`

Finally, the simply function `which()` can be surprisingly helpful.

`which()` tells you *which* of the entries in a logical vector are `TRUE`

Tips and tricks

Logical vectors and `which()`

Finally, the simply function `which()` can be surprisingly helpful.

`which()` tells you *which* of the entries in a logical vector are `TRUE`, *i.e.*, *which* element—or elements—satisfies your logical condition(s).

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters > "m"
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE
```

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters > "m"
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE
```

```
which(letters > "m")
```

```
#>  [1] 14 15 16 17 18 19 20 21 22 23 24 25 26
```

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters > "m"
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE
```

```
which(letters > "m")
```

```
#>  [1] 14 15 16 17 18 19 20 21 22 23 24 25 26
```

```
letters[which(letters > "m")]
```

```
#>  [1] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Alternatively, we could have just used the logical vector.

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters > "m"
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE
```

```
letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters > "m"
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE
```

```
letters[letters > "m"]
```

```
#>  [1] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Tips and tricks

Logical vectors, continued

This logic-based selection works on many classes of objects, but it may change the class/structure of the object.

```
# Create a matrix
mat ← matrix(1:9, ncol = 3)
# Print it out
mat
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
```

Tips and tricks

Logical vectors, continued

This logic-based selection works on many classes of objects, but it may change the class/structure of the object.

```
# Create a matrix
mat ← matrix(1:9, ncol = 3)
# Print it out
mat
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
```

```
# Is the entry even?
mat %% 2 == 0
```

```
#>       [,1]  [,2]  [,3]
#> [1,] FALSE  TRUE FALSE
#> [2,]  TRUE FALSE  TRUE
#> [3,] FALSE  TRUE FALSE
```

Tips and tricks

Logical vectors, continued

This logic-based selection works on many classes of objects, but it may change the class/structure of the object.

```
# Create a matrix
mat ← matrix(1:9, ncol = 3)
# Print it out
mat
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
```

```
# Is the entry even?
mat %% 2 == 0
```

```
#>       [,1]  [,2]  [,3]
#> [1,] FALSE  TRUE FALSE
#> [2,]  TRUE FALSE  TRUE
#> [3,] FALSE  TRUE FALSE
```

```
# Print the even entries
mat[mat %% 2 == 0]
```

```
#> [1] 2 4 6 8
```

Table of contents

Tips and tricks