Functions & Functionals in R

PSS SUMMER SCHOOL

Anton Badev, Daniel Nikolic, Justin Skillman

June 22, 2017

Outline

- Loops
- Vectorization
- Functions
- Functionals
  - The `apply` family

Warm-up Exercise

- Take **5 minutes** to complete this task
- The Rdata file `m1_lecture2.Rdata` contains `ones_zeros` and `letts`
- They are both vectors of length 1,000
- Write a for loop to do the following:
  - If `ones_zeros[i]` is **1** change `letts[i]` into an upper case letter using `toupper()` function
  - If `ones_zeros[i]` is **0** leave `letts[i]` as a lower case
  - Record if a change was made
- Report $\frac{\text{number of changes made}}{\text{total length of } \texttt{letts}}$

Looping patterns

- There are three basic ways to loop over an object:
  1. Loop over the **elements** of an object
  2. Loop over **numeric indices**
  3. Loop over the **names** of an object

```
> my_vec <- c(2,4,8)
> names(my_vec) <- c("first","squared","cubed")
> my_vec
  first squared cubed
      2       4     8
```

Looping patterns

Over the elements

```
for(x in my_vec) {
print(x)
}
[1] 2
[1] 4
[1] 8
```

Over numeric indices

```
n_elem <- length(my_vec)
for(x in 1:n_elem) {
print(x)
}
[1] 1
[1] 2
[1] 3
```

Over the names

```
for(x in names(my_vec)) {
print(x)
}
[1] "first"
[1] "squared"
[1] "cubed"
```

Over the elements

- Can seem most natural to beginning programmers
- Becomes inefficient quickly
- *Example*: Take a uniformly random vector of integers and find the difference between each element and the last element in the vector

```
my_vec <- floor(runif(1000,1,100))
res_bin <- c() #empty vector for results
for(i in my_vec) {
  diff <- i - my_vec[length(my_vec)]
  res_bin <- c(res_bin,diff)
}
```

- R copies `res_bin` in memory every time you append to it

Over numeric indices

- Results are usually the same length as inputs because we want to perform some operation on *each* element
- Therefore we know the size of our output!
- *Example*: Take a uniformly random vector of integers and find the difference between each element and the last element in the vector

```
my_vec <- floor(runif(1000,1,100))
n_elem <- length(my_vec)
res_bin <- matrix(NA, n_elem, 1) #bin for results
for(i in 1:n_elem) {
  diff <- my_vec[i] - my_vec[n_elem]
  res_bin[i] <- diff
}
```

- Much faster, as R just assigns that index in `res_bin` a value

What is vectorization?

- Performs what would be a scalar operation on a vector
- Heuristic:
  multiple statements e.g. loops → *single statement*
- Native to R environment
- Translates functions to C instead of R
- C takes advantage of BLAS (Basic Linear Algebra Subprogram)
  - SISD vs SIMD

Simple vectorization example

- Adding two matrices element wise

Scalar operations

```
a <- matrix(c(1,2,3,4),2,2)
b <- matrix(c(4,3,2,1),2,2)
y <- dim(a,1)
x <- dim(a,2)
res <- matrix(NA,x,y)

for (xi in 1:x) {
  for (xj in 1:y) {
    ans <- a[xi,xj] + b[xi,xj]
    res[xi,xj] <- ans
  }
}
```

Vectorized operation

```
a <- matrix(c(1,2,3,4),2,2)
b <- matrix(c(4,3,2,1),2,2)
res <- a + b
```

When to use vectorization?

- Great for replacing
  - Independent iterative processes
  - Making many repeated calls to functions in R (vs C)
- Not suited to replace
  - Dependent iterative processes
  - While loops
  - Functions with non-vector inputs
  - Algorithms where vectorization causes over-complication and readability issues

Advanced functions

- Components of functions
- Anonymous functions
- Closures
- Return values
- Editing a function

Components of a function

- `body()` → the function's code
- `formals()` → the function's arguments (inputs)
- `environment()` → the “sandbox” that maps the names of your function's objects to the locations of the objects themselves
- *Example:*

```
upp_or_low <- function(mtx, tri) {
  if (tri=="upper") {
    return(mtx[upper.tri(mtx)])
  } else if (tri=="lower") {
    return(mtx[lower.tri(mtx)])
  } else {
    stop("argument must be upper or lower!")
  }
}
```

Components of a function: example (cont.)

body

```
> body(upp_or_low)
{
    if (tri == "upper") {
        return(mtx[upper.tri(mtx)])
    }
    else if (tri == "lower") {
        return(mtx[lower.tri(mtx)])
    }
    else {
        stop("wrong arg!")
    }
}
```

formals

```
>formals(upp_or_low)
$mtx


$tri
```

environment

```
> environment(upp_or_low)
<environment: R_GlobalEnv>
```

Anonymous functions

- Functions that are not stored or given a name
- Used when a named function is not necessary (one time use)
- Often used in conjunction with functionals or closures
- *Example:* How can I assign `bin` the function's output and not the function itself using an anonymous function?

Which one will work? What will be their values?

```
>bin_1 <- function(x,y) (x + y)**3(2,1)
>bin_2 <- (function(x,y) (x + y)**3)(2,1)
```

Anonymous functions

- Functions that are not stored or given a name
- Used when a named function is not necessary (one time use)
- Often used in conjunction with functionals or closures
- *Example:* How can I assign `bin` the function's output and not the function itself using an anonymous function?

Which one will work? What will be their values?

```
>bin_1 <- function(x,y) (x + y)**3(2,1)
>bin_2 <- (function(x,y) (x + y)**3)(2,1)
```

```
>bin_1(2,1)
Error in bin_1(2, 1) : attempt to apply non-function
>bin_2(2,1)
[1] 27
```

Closures

- A closure is a function that generates another function
- Anonymous functions are often *enclosed* in them
- Useful in conditional flow of a program
- *Example:*

```
>dffrnc <- function(offset) {
  function(vec) {
    start <- 1 + offset
    end   <- length(vec) - offset
    diff  <- vec[1:end] - vec[start:length(vec)]
    return(diff)
  }
}
>secdff <- dffrnc(2)
>secdff(c(8, 10, 1, 7))
[1] 7 3
```

Return values

- Return values can be specified using `return()` or if written as the last statement in the function
- `return()` is suggested in many style guides for readability

```
f <- function(z) {
  procedure_1
  return(z)
}
```

$\xleftrightarrow{\text{equivalent}}$

```
f <- function(z) {
  procedure_1
  z
}
```

- If you have more than one object to return you can do so in a `list()`
- Useful for returns of differing object type
- *Example:* For `my_optim` return the exposure matrix `exp_mtx` and eigenvector `eig_vec`

```
my_optim <- function(optim_args) {
  optim_procedure
  exp_mtx <- some_result
  eig_vec <- some_other_result
  return(list(exp_mtx, eig_vec))
}
```

Editing a function

- The benefit of open source is that even if you didn't write it, you can modify the code
- Open source also means that *sometimes* there are errors or small bugs in a package
- Use `edit(<function>)` to edit a function in a text editor

Functionals

- ***functionals*** are functions that take another function as an input and return a "vector-like" object as output
- Example: `my_fnctnl <- function(g) g(1:10)`

```
my_fnctnl(mean)
[1] 5.5
my_fnctnl(sum)
[1] 55
```

- Commonly used in R for *vectorization*
- Allows for application of a function over data objects
  - `apply` family of functions

ifelse()

- ifelse() is a vectorized form of an if else statement
- ifelse(boolean vector, function if T, function if F)
- Example from warm-up:

```
ans <- ifelse(ones_zeros==1, toupper(letts), letts)
head(ans)
[1] "k" "S" "p" "E" "w" "w"
```

- Remember that this simply becomes a C if statement
- Avoid viewing functionals as a “black box”

lapply

- ***list apply***: list/vector input → list output
- C loop wrapped in R (much faster)
- Simple example

```
vec_list <- list(c(1:10),c(2:11),c(3:12))
lapply(vec_list, function(x) max(x)**2)
[[1]]
[1] 100
[[2]]
[1] 121
[[3]]
[1] 144
```

- Each iteration uses a vector, `x`, as input

lapply (cont.)

- lapply with functions taking more than 1 input
- lapply **only** iterates over the list elements
- We can specify other arguments using the format lapply(list, function, other_args)

```
vec_list <- list(c(1:10),c(2:11),c(3:12))
lapply(vec_list, function(x,pwr) max(x)**pwr, pwr=3)
[[1]]
[1] 1000
[[2]]
[1] 1331
[[3]]
[1] 1728
```

- Note: unlist coerces a list into a vector

sapply

- Works the same way as lapply but returns a **vector**
- Vectors are often more useful as output

```
vec_list <- list(c(1:10),c(2:11),c(3:12))
sapply(vec_list, function(x,pwr) max(x)**pwr, pwr=3)
[1] 1000 1331 1728
```

apply

- Applies a function over a dimension of a matrix
- `apply(m, dim, func, func_args)`
- Used often for summarizations or data manipulations

```
mtx <- matrix(c(1,2,3,4),2,2)
mtx
     [,1] [,2]
[1,]    1    3
[2,]    2    4

apply(mtx, 2, sum)
[1] 3 7
```

tapply

- Applies a function over multiple groupings of data
- Groupings of data displayed as factors
- `tapply(vec, factors, func)`
- `tapply()` vs. `by()`

```
incomes <- c(40,52,150,61,103,41,55,33,90,55)
states <- c("PA","NY","NY","PA","CA","DC","CA","DC","TX","TX")
tapply(incomes, states, median)
   CA    DC    NY    PA    TX
 79.0  37.0 101.0  50.5  72.5
```

In class exercise

- Complete the following problems in a group, we will ask each group to speak in **15 minutes**
- Babynames problem
  1. Load the package `babynames`
  2. Write a function that determines which name was the most popular in a given year. The only inputs to this function should be the vector of n's for that year and the vector of *name*s, the only output should be a character string that is the most popular name
  3. Write an `lapply` statement that finds the most popular name in all of the years in the dataset.
- Integer problem
  1. Write a function saved as `find_small_div` that takes in n, a, and b and returns the smallest integer in the set of integers between a and b that is divisible by n.
  2. Draw an integer vector of length 100 from a uniform distribution with support from 50 to 8,000. Save this as `samp_vec`.
  3. Take as given that a and b are 10,000 and 100,000 respectively and write an `sapply` statement that applies `find_small_div` over `samp_vec`.

Summary

- Loops
- Vectorization
- Functions
- Functionals
  - The apply family