Advanced Loops

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/teaching/stat133

Advanced Looping

Outline

- Vectorizing a function
- Loops over elements of data structures

Motivation

```r
# fahrenheit to celsius
to_celsius <- function(x) {
  (x - 32) * (5/9)
}
```

The function `to_celsius()` happens to be a vectorized function:

```r
to_celsius(c(32, 40, 50, 60, 70))
```

```
## [1]  0.000000  4.444444 10.000000 15.555556 21.111111
```

Motivation

- In general, R functions defined on scalar values are expected to be vectorized
- You should have noticed that many functions in R are vectorized

Motivation

What happens in this situation?

```r
# trying to_celsius() on a list
to_celsius(list(32, 40, 50, 60, 70))
```

Motivation

```
# trying to_celsius() on a list
to_celsius(list(32, 40, 50, 60, 70))

## Error in x - 32:  non-numeric argument to binary
operator
```

`to_celsius()` does not work with a list

Motivation

One solution is to use a `for` loop:

```
temps_farhenheit <- list(32, 40, 50, 60, 70)

temps_celsius <- numeric(5)
for (i in 1:5) {
  temps_celsius[i] <- to_celsius(temps_farhenheit[[i]])
}

temps_celsius

## [1]  0.000000  4.444444 10.000000 15.555556 21.111111
```

Vectorizing Functions - Vectors

- R provides a set of functions to “vectorize” functions over the elements of data structures:
  - `lapply()`, `sapply()`, `apply()`, etc
- These functions allow us to avoid writing loops
- These are functions that have grown organically
- They have common names but unfortunately not all of them use the same arguments naming conventions

`lapply()`

Loops over vectors or lists

- The simplest apply function is `lapply()`
- `lapply()` stands for **list apply**
- It takes a list or vector and a function as inputs
- It applies the function to each element of the list
- The output is another list

lapply()

```r
players <- list(
  warriors = c('kurry', 'iguodala', 'thompson', 'green'),
  cavaliers = c('james', 'shumpert', 'thompson'),
  rockets = c('harden', 'howard')
)

lapply(players, length)

## $warriors
## [1] 4
##
## $cavaliers
## [1] 3
##
## $rockets
## [1] 2
```

lapply()

```
# convert to upper case
lapply(players, toupper)

## $warriors
## [1] "KURRY"    "IGUODALA" "THOMPSON" "GREEN"
##
## $cavaliers
## [1] "JAMES"    "SHUMPERT" "THOMPSON"
##
## $rockets
## [1] "HARDEN" "HOWARD"
```

lapply()

You can pass arguments to the applied functions

```
# collapsing with paste()
lapply(players, paste, collapse = '-')

## $warriors
## [1] "kurry-iguodala-thompson-green"
##
## $cavaliers
## [1] "james-shumpert-thompson"
##
## $rockets
## [1] "harden-howard"
```

lapply()

You can pass your own functions

```
num_chars <- function(x) {
  nchar(x)
}

lapply(players, num_chars)

## $warriors
## [1] 5 8 8 5
##
## $cavaliers
## [1] 5 8 8
##
## $rockets
## [1] 6 6
```

Anonymous functions

You can define a function with no name (i.e. anonymous function):

```r
# anonymous function
lapply(players, function(x) paste('mr', x))

## $warriors
## [1] "mr kurry"    "mr iguodala" "mr thompson" "mr green"
##
## $cavaliers
## [1] "mr james"    "mr shumpert" "mr thompson"
##
## $rockets
## [1] "mr harden" "mr howard"
```

Anonymous functions

```r
# anonymous function
lapply(players, function(x) grep('a', x, value = TRUE))

## $warriors
## [1] "iguodala"
##
## $cavaliers
## [1] "james"
##
## $rockets
## [1] "harden" "howard"
```

lapply()

Remember that a `data.frame` is internally stored as a list:

```r
df <- data.frame(
  name = c('Luke', 'Leia', 'R2-D2', 'C-3PO'),
  gender = c('male', 'female', 'male', 'male'),
  height = c(1.72, 1.50, 0.96, 1.67),
  weight = c(77, 49, 32, 75)
)
```

lapply()

Remember that a `data.frame` is internally stored as a list:

```
lapply(df, class)

## $name
## [1] "factor"
##
## $gender
## [1] "factor"
##
## $height
## [1] "numeric"
##
## $weight
## [1] "numeric"
```

sapply()

Loops over vectors or lists

- `sapply()` is a modified version of `lapply()`
- `sapply()` stands for **simplified apply**
- It takes a list or vector and a function as inputs
- It applies the function to each element of the list
- `sapply()` attempts to simplify the output (possibly as a vector or list)

sapply()

```
players <- list(
  warriors = c('kurry', 'iguodala', 'thompson', 'green'),
  cavaliers = c('james', 'shumpert', 'thompson'),
  rockets = c('harden', 'howard')
)

sapply(players, length)

##  warriors cavaliers   rockets
##         4         3         2
```

sapply()

```
sapply(players, nchar)

## $warriors
## [1] 5 8 8 5
##
## $cavaliers
## [1] 5 8 8
##
## $rockets
## [1] 6 6
```

when the output cannot be simplified, `sapply()` returns the same output as `lapply()`

`apply()`

Loops on matrices (or arrays)

Consider a matrix:

```
(m <- matrix(1:20, 4, 5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

How can we get the median of each row?

Loops on matrices (or arrays)

We could write a for loop

```
medians <- numeric(nrow(m))

for (r in 1:nrow(m)) {
  medians[r] <- median(m[r, ])
}

medians

## [1]  9 10 11 12
```

Or we could use the apply() function

Loops over matrices of arrays

- `apply()` is perhaps the most popular apply function
- It takes a matrix or array, an index and a function as inputs
- Additionaly, it can take more arguments
- The `MARGIN` index gives the subscript which the function will be applied over
  - `MARGIN = 1` indicates rows
  - `MARGIN = 2` indicates columns
  - `MARGIN = c(1, 2)` indicates both rows and columns

apply()

```
(m <- matrix(1:20, 4, 5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20

# median of rows
apply(m, 1, median)

## [1]  9 10 11 12
```

apply()

```
(m <- matrix(1:20, 4, 5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20

# median of columns
apply(m, 2, median)

## [1]  2.5  6.5 10.5 14.5 18.5
```

apply()

apply() can be used on data frames

```
# mean height and weight (on columns)
apply(df[ ,c('height', 'weight')], 2, mean)

##  height  weight
##  1.4625 58.2500
```

apply()

apply() can be used on data frames

```
# product of height and weight (on rows)
apply(df[ ,c('height', 'weight')], 1, prod)

## [1] 132.44  73.50  30.72 125.25
```

tapply()

Loops over vectors split by a factor

- `tapply()`
- the name does not mean anything
- very useful to aggregate data

tapply()

Say you need to obtain average height and weight by gender

```
df
```

```
##     name gender height weight
## 1  Luke   male   1.72     77
## 2  Leia female   1.50     49
## 3 R2-D2   male   0.96     32
## 4 C-3PO   male   1.67     75
```

tapply()

```
# mean height by gender
tapply(df$height, df$gender, mean)

## female   male
##   1.50   1.45

# mean weight by gender
tapply(df$weight, df$gender, mean)

##   female     male
## 49.00000 61.33333
```

`mapply()`

Multiple-Input Apply

- `lapply()` only accepts a single vector or list to loop over
- `lapply()` does not give you access to the names of the elements
- `mapply()` solves this issues

Multiple-Input Apply

- `mapply()` stands for **multiple argument list apply**
- it lets you pass in as many vectors as you like
- the first argument is the function to be applied
- the following arguments are vectors

mapply()

```
# pasting player name and team
mapply(paste, players, names(players))

## $warriors
## [1] "kurry warriors"    "iguodala warriors" "thompson warrior
## [4] "green warriors"
##
## $cavaliers
## [1] "james cavaliers"    "shumpert cavaliers" "thompson cavali
##
## $rockets
## [1] "harden rockets" "howard rockets"
```

mapply()

How would you generate this list:

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

mapply()

```
lst <- vector('list', 4)
for (k in 1:4) {
  lst[[k]] <- rep(k, 5-k)
}
lst

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

mapply()

```
# multiple input argument
mapply(rep, 1:4, 4:1)

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

`apply()` Related Functions

Related Functions

Some convenient functions (faster than using `apply()`)

- `colMeans()`
- `rowMeans()`
- `colSums()`
- `rowSums()`

colMeans()

```
# column means of height and weight
colMeans(df[ ,c('height', 'weight')])

##  height  weight
##  1.4625 58.2500

# equivalent to:
apply(df[ ,c('height', 'weight')], 2, mean)

##  height  weight
##  1.4625 58.2500
```

rowMeans()

```r
# row means of height and weight
rowMeans(df[ ,c('height', 'weight')])

## [1] 39.360 25.250 16.480 38.335

# equivalent to:
apply(df[ ,c('height', 'weight')], 1, mean)

## [1] 39.360 25.250 16.480 38.335
```

rowSums()

```
# row sums of height and weight
rowSums(df[ ,c('height', 'weight')])

## [1] 78.72 50.50 32.96 76.67

# equivalent to:
apply(df[ ,c('height', 'weight')], 1, sum)

## [1] 78.72 50.50 32.96 76.67
```

colSums()

```
# column sums of height and weight
colSums(df[ ,c('height', 'weight')])

## height weight
##   5.85 233.00

# equivalent to:
apply(df[ ,c('height', 'weight')], 2, sum)

## height weight
##   5.85 233.00
```

`aggregate()`

Apply a function to data subsets

- `aggregate()` can be thought as a generalization of `tapply()`
- It splits the data into subsets, and applies a function
- The subsets must be provided as a list
- The output is returned in a “convenient” form

aggregate()

```r
df <- data.frame(
  name = c('Luke', 'Leia', 'R2-D2', 'C-3PO'),
  gender = c('male', 'female', 'male', 'male'),
  species = c('human', 'human', 'robot', 'robot'),
  height = c(1.72, 1.50, 0.96, 1.67),
  weight = c(77, 49, 32, 75)
)
```

aggregate()

```
# mean height and weight by gender
aggregate(df[ ,c('height', 'weight')],
          list(df$gender), mean)

##   Group.1 height   weight
## 1  female   1.50 49.00000
## 2    male   1.45 61.33333
```

aggregate()

```
# mean height and weight by species
aggregate(df[ ,c('height', 'weight')],
          list(df$species), mean)

##   Group.1 height weight
## 1   human  1.610   63.0
## 2   robot  1.315   53.5
```

aggregate()

```
# mean height and weight by gender and species
aggregate(df[ ,c('height', 'weight')],
          list(df$gender, df$species), mean)

##   Group.1 Group.2 height weight
## 1  female   human  1.500   49.0
## 2    male   human  1.720   77.0
## 3    male   robot  1.315   53.5
```

`sweep()`

Sweep out array summaries

- Sometimes we need to sweep out a summary statistic
- e.g. removing the mean on each column
- `sweep()` is specially designed for this

sweep() mean

```r
# mean height and weight
hw_mean <- colMeans(df[ ,c('height', 'weight')])

# centering height and weight
sweep(df[ ,c('height', 'weight')], 2, hw_mean)

##    height weight
## 1  0.2575  18.75
## 2  0.0375  -9.25
## 3 -0.5025 -26.25
## 4  0.2075  16.75
```

sweep() median

```
# mean height and weight
hw_median <- apply(df[ ,c('height', 'weight')], 2, median)

# centering height and weight
sweep(df[ ,c('height', 'weight')], 2, hw_median)

##   height weight
## 1  0.135     15
## 2 -0.085    -13
## 3 -0.625    -30
## 4  0.085     13
```

R Package `"plyr"`

R package "plyr"

- "plyr" provides alternative functions to the apply-family functions in base R
- functions in "plyr" are better designed, usually faster, and with better names of arguments
- Read the paper **The Split-Apply-Combine Strategy for Data Analysis**
- http://www.jstatsoft.org/v40/i01