Video 11: Loops

Stats 102A

Miles Chen, PhD

Loops

for Loops

The simplest and most common type of loop in R is the `for` loop. Given a vector (including lists), it iterates through the elements and evaluates the code block for each element.

```r
for(x in 1:10) {
  cat(x ^ 2, " ", sep = "")
}
```

```
1 4 9 16 25 36 49 64 81 100
```

for loops can iterate through any vector

```r
l <- list(1:3, LETTERS[1:7], c(TRUE, FALSE)) # l is a list of three
l
```

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

[[2]]
[1] "A" "B" "C" "D" "E" "F" "G"

[[3]]
[1]  TRUE FALSE
```

```r
for(y in l) {
  print(length(y))
}
```

```
[1] 3
[1] 7
[1] 2
```

for loops

```r
library(datasets)
state.name[1:5]
```

```
[1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

```r
for(state in state.name[1:5]) {
  cat(state, "has", nchar(state),"letters in its name.\n")
}
```

```
Alabama has 7 letters in its name.
Alaska has 6 letters in its name.
Arizona has 7 letters in its name.
Arkansas has 8 letters in its name.
California has 10 letters in its name.
```

Loops - Storing results

It is almost always better to create an object to store your results first, rather than growing the object as you go. You can only do this if you know how many values to expect.

```r
# Good
res <- rep(NA, 10^7) # create an object to store results.
system.time(
  for(x in seq_along(res)){
    res[x] <- x ^ 2
  }
)
```

```
##    user  system elapsed 
##    0.64    0.00    0.64
```

For loops

```r
# slower
res <- 0
system.time(
  for (x in 1:10^7){
    res[x] <- x ^ 2  # each iteration requires that res be resized
  }
)
```

```
##    user  system elapsed 
##    2.56    0.26    2.83 
```

For loops

```r
# Slowest
res <- c()
system.time(
  for (x in 1:10^5) {     # we are only doing 1/100 of the work, but
    res <- c(res, x ^ 2) # each iteration copies res into a redefir
  }
)
```

```
##    user  system elapsed
##   13.55    0.09   13.64
```

`while` loops

Repeat until the given condition is not met (condition is `FALSE`)

```r
i <- 1
res <- rep(NA, 10)
while (i <= 10) {
  res[i] <- i ^ 2
  i <- i + 1
}
res
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Question: what is the value of `i` when the code finishes?

Answer

```r
i <- 1
res <- rep(NA, 10)
while (i <= 10) {
  res[i] <- i ^ 2
  i <- i + 1
}
i
```

```
[1] 11
```

repeat loops

Repeat until break is executed

```r
i <- 1
res <- rep(NA, 10)
repeat {
  res[i] <- i ^ 2
  i <- i + 1
  if (i > 10){
    break
  }
}
res
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Special keywords - `break` and `next`

These are special actions that only work inside of a loop

- `break` - ends the current (inner-most) loop
- `next` - ends the current iteration and starts the next iteration

```r
for(i in 1:10) {
    if (i %% 2 == 0){
        break
    }
  cat(i,"")
}
```

Question: What will this output?

Answer:

```r
for(i in 1:10) {
    if (i %% 2 == 0){
        break
    }
  cat(i,"")
}
```

```
1
```

Keyword: next

```r
for(i in 1:10) {
    if (i %% 2 == 0){
        next
    }
  cat(i,"")
}
```

Question: What will this output?

Answer

```r
for(i in 1:10) {
    if (i %% 2 == 0){
        next
    }
  cat(i,"")
}
```

```
1 3 5 7 9
```

`seq_len()`, `seq_along()`

Often we want to use a loop across the indexes of an object and not the elements themselves. There are several useful functions to help you do this: `:`, `seq`, `seq_along`, `seq_len`, etc.

```r
l = list(1:3, LETTERS[1:7], c(TRUE,FALSE))
res = rep(NA, length(l))
for(x in seq_along(l)) {
  res[x] = length(l[[x]])
}
res
```

```
[1] 3 7 2
```

`seq_len()`, `seq_along()` produce similar results

```r
1:length(l)
```

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

```r
seq_along(l)
```

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

```r
seq_len(length(l))
```

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

Using `1:length(l)` can cause problems for length-0 vectors

```r
l <- list() # l is an empty 0-length list
1:length(l) # creates a length-2 vector that counts down
```

```
[1] 1 0
```

```r
res <- rep(NA, length(l))
for(x in 1:length(l)) {
  res[x] <- length(l[[x]]) # l[[1]] doesn't exist
}
```

```
Error in l[[x]]: subscript out of bounds
```

`seq_len()`, `seq_along()` avoids those issues

```r
l <- list()
seq_along(l)
```

```
integer(0)
```

```r
res <- rep(NA, length(l))
for(x in seq_along(l)) { # nothing gets executed
  res[x] <- length(l[[x]])
}

seq_len(length(l)) # similarly seq_len produces a 0-length integer
```

```
integer(0)
```

Vectorizing Code

Vectorizing Code

The performance of R code can be frequently improved if we think of ways to vectorize it.

It is not always possible to vectorize code and it is not always worth the effort. In some cases, it is better to write inefficient code that takes 20 seconds to run than getting stuck for a few hours trying to write more efficient code.

If you are aware of some of R's vectorized operations, you can write your code to be vectorized from the start.

> "We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%" - Donald Knuth (Computer science legend, and creator of TeX typesetting)

`if` vs `ifelse` for vectorization

Example from *Scientific Programming and Simulation Using R*. Pg. 43. Section 3.9. Exercise 1.

Consider the function $y = f(x)$ defined by:

$$f(x) = \begin{cases} -x^3, & \text{for } x \le 0 \\ x^2, & \text{for } 0 < x \le 1 \\ \sqrt{x}, & \text{for } 1 < x \end{cases}$$

Here is a function that uses `if()`. It works but is not vectorized.

```r
f <- function(x) {
  if (x <= 0) {
    value <- -x ^ 3
  } else if (x <= 1) { # note: I do not need to check if x > 0
    value <- x ^ 2
  } else {
    value <- sqrt(x)
  }
  return(value)
}
```

Using a non-vector function

If you try to use `f()` as-is, it will not have the desired effect. It evaluates the first value of `x` which is negative. It then evaluates `-x^3` and returns that. Note that the power operation `^3` is vectorized, so it returns all of `-x^3`

```r
x <- seq(-2, 2, by = .01)
plot_values <- f(x) # doesn't work because f is not vectorized
```

```
Error in if (x <= 0) {: the condition has length > 1
```

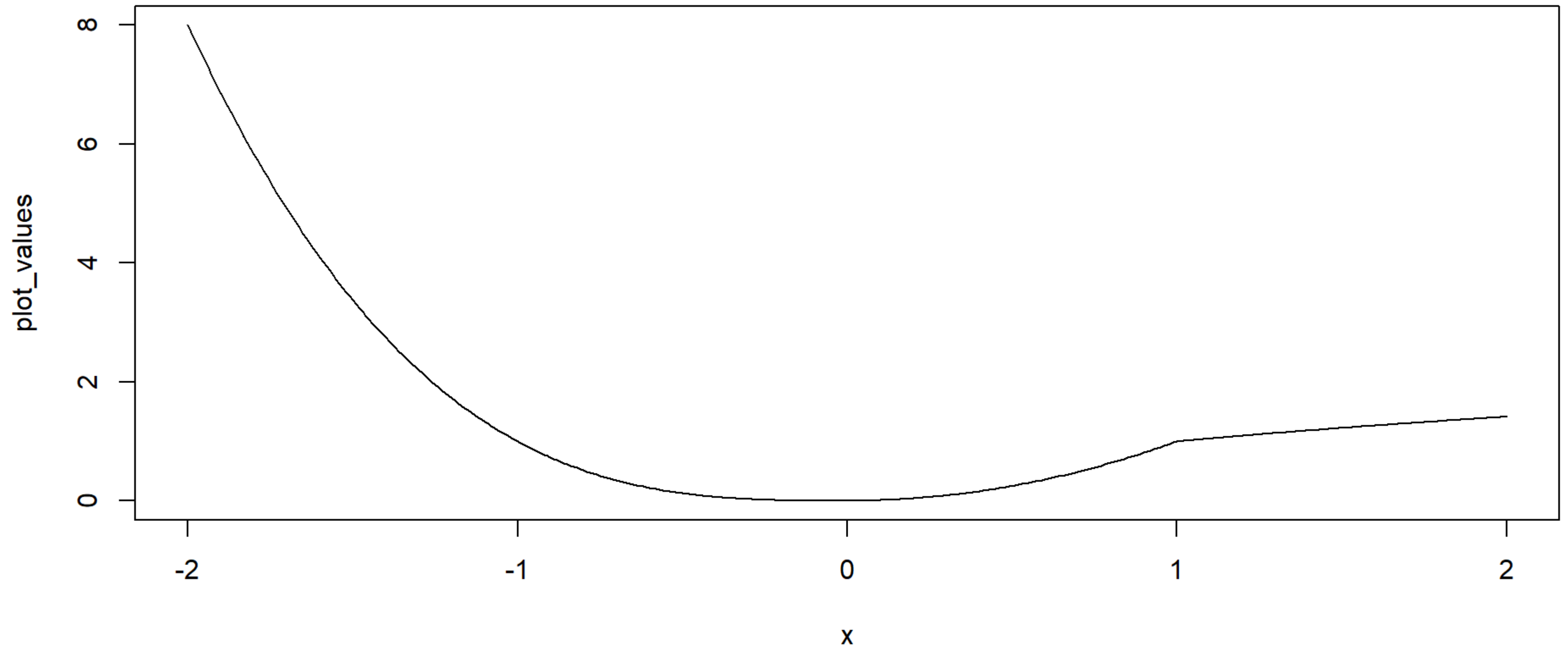
```r
plot(x, plot_values, type = "l")
```

```
Error in eval(expr, envir, enclos): object 'plot_values' not found
```

Using a non-vector function

To get the desired result, we have to combine the non-vectorized function with a loop. The loop will subset `x` to `x[i]`. It takes each value `x[i]` and calculates and stores `f(x[i])`. This will produce the desired results.

```r
x <- seq(-2, 2, by = .01)
plot_values <- rep(0, length(x))
# f is not vectorized and requires a loop to evaluate each value in x separately
for(i in seq_along(x)) {
  plot_values[i] <- f(x[i])
}
plot(x, plot_values, type = "l")
```



plot_values
0
2
4
6
8
-2
-1
0
1
2
x

Vectorize with `ifelse`

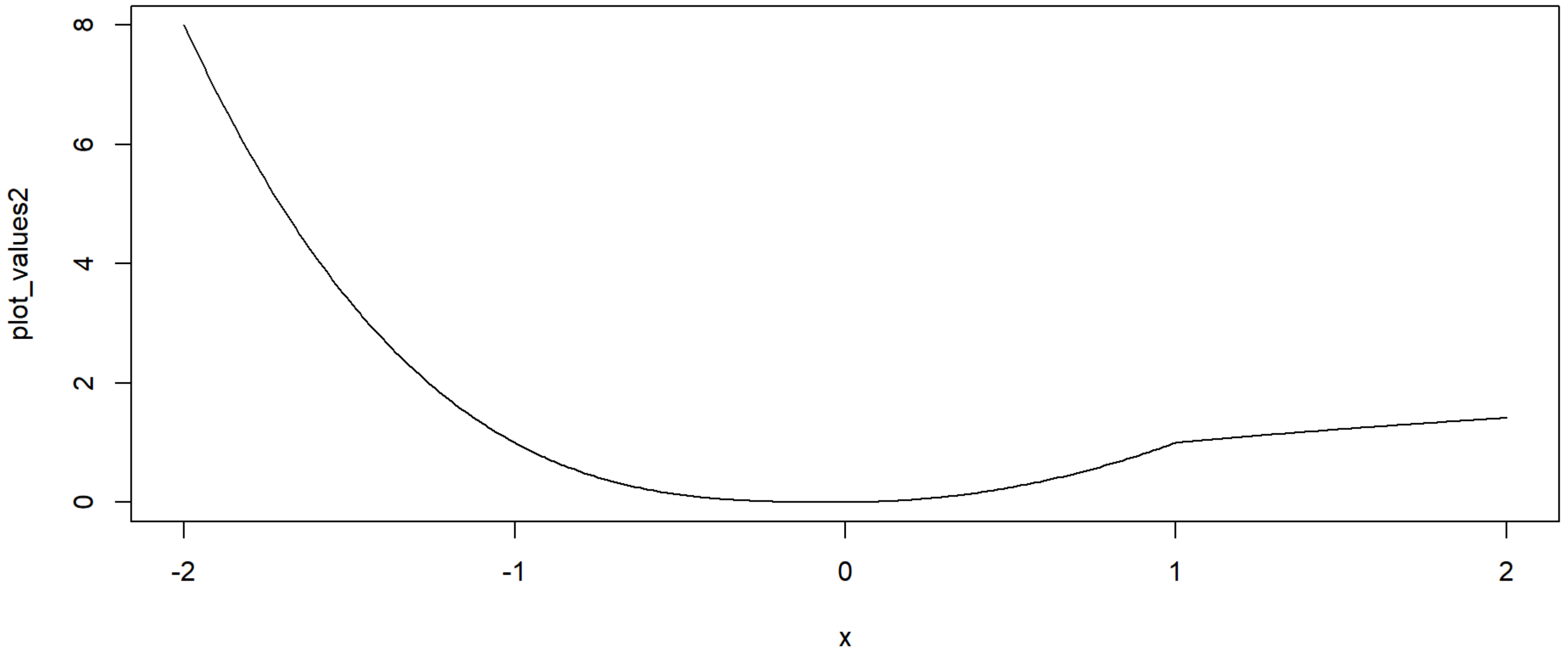
Instead of using `if()`, we can use `ifelse()` to vectorize the function.

```r
f2 <- function(x) {
  ifelse(x <= 0, # check if x <= 0
         -x ^ 3, # if x <= 0 is true, return x ^ 3
         ifelse(x <= 1,  # if x <= 0 is false, check if x <= 1
                x ^ 2,   # if x <= 1 is true, return x ^ 2
                sqrt(x), # otherwise return sqrt(x)
                )
        )
}

# with comments removed, it's compact:
f2 <- function(x) {
  ifelse(x <= 0, -x ^ 3, ifelse(x <= 1, x ^ 2, sqrt(x)))
}
```

Vectorized function f2() does not require a loop

```r
x <- seq(-2, 2, by = .01)
plot_values2 <-  f2(x)
plot(x, plot_values2, type = "l")
```


plot_values2
0
2
4
6
8
-2
-1
0
1
2
x

Note on the warning

In the previous slide, a warning is produced.

The warning is produced because `sqrt(x)` is calculated for all of `x` which contains negative values.

The `ifelse` statement evaluates the entire TRUE-expression and the entire FALSE-expression. It then decides whether to return the values from the TRUE expression or the FALSE expression based on the condition.

The result is that the vector `plot_values2` consists only of real numbers, but because `sqrt(x)` was evaluated for all of `x`, we see the warning about `NaN`s produced when `sqrt()` was evaluated on negative values of `x`.

`rowSums()` and `colSums()`

R offers two very fast operations for matrices: `rowSums()` and `colSums()`

If you can think of a way to use these operations, you can save time from your code.

I will perform the same operation in three different ways.

We create a large matrix: 1 million rows x 100 columns. We want to find the mean of each row.

Method 1: Subsetting by row, find the mean

```r
x <- matrix(1:10^8, ncol = 100)
results1 <- rep(NA, nrow(x))
system.time(
  for(i in seq_along(results1)){
    results1[i] <- mean(x[i, ])
  }
)
```

```
##    user  system elapsed 
##    3.66    0.00    3.67 
```

Method 2: apply()

```r
system.time(
  results2 <- apply(x, MARGIN = 1, FUN = mean)
)
```

```
##    user  system elapsed 
##    6.44    0.05    6.48 
```

Method 3: Use `rowSums()` and divide by the number of columns

```r
# fastest
system.time(
  results3 <- rowSums(x)/ncol(x)
)
```

```
##    user  system elapsed 
##    0.26    0.00    0.27 
```

All three methods produce the same results

```r
all.equal(results1, results2)
```

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

```r
all.equal(results1, results3)
```

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