Programming Basics: Conditions, Loops
Week 2 Monday

Miles Chen, PhD

Department of Statistics

UCLA

Related Reading

- Introduction to Scientific Programming and Simulation Using R
  - ▶ Chapter 3 - Basic Programming
  - ▶ Must be on UCLA Network or connected to VPN to access:
  - ▶ https://www.taylorfrancis.com/books/mono/10.1201/9781420068740/introduction-scientific-programming-simulation-using-owen-jones-robert-maillardet-andrew-robinson
- Advanced R
  - ▶ Chapter 5: Control Flow
  - ▶ https://adv-r.hadley.nz/control-flow.html

Section 1

Operators that produce logical values

Vectorized Logical Operators

R has the following logical operators. Use of a logical operator will coerce non-logical types to logical type. (e.g. 0 becomes FALSE, all other numeric values become TRUE.)

- or: `x | y`
- and: `x & y`
- not: `!x`
- exclusive or: `xor(x,y)`

The above logical operators are vectorized and will perform element-wise comparison as well as recycling. The vectorized operations will return a vector of logical values.

Non-Vectorized Logical operators

Non-Vectorized operators will return only one logical value. The non-vectorized operators are used in if clauses.

- non-vector or: `x || y`
- non-vector and: `x && y`

The 'non-vector' `||` and `&&` accepts only logical vectors of length 1.

If x has length greater than 1 it will return an error.

If y has length greater than 1 and is evaluated, it will return an error. If y has length greater than 1 but is not evaluated, R will not error.

Older R versions (prior to 4.3) evaluated left to right examining only the first element of each vector. If you have an older version of R, you should update.

See `?base::Logic`

Vectorized vs Non-Vectorized

```
x <- c(TRUE,  TRUE, FALSE, FALSE)
y <- c(FALSE, TRUE, FALSE,  TRUE)
x | y
```

```
[1]  TRUE  TRUE FALSE  TRUE
x & y
```

```
[1] FALSE  TRUE FALSE FALSE
x || y  # x length greater than 1 leads to error
```

```
Error in x || y: 'length = 4' in coercion to 'logical(1)'
FALSE && y # FALSE AND blank is always FALSE. y isn't evaluated. no error
```

```
[1] FALSE
TRUE || y # TRUE OR blank is always TRUE. y isn't evaluated. no error
```

```
[1] TRUE
FALSE || y # returns error because y has to be evaluated
```

```
Error in FALSE || y: 'length = 4' in coercion to 'logical(1)'
```

Logical operators with NA

What is `TRUE | NA`?

What is `TRUE & NA`?

What is `FALSE | NA`?

What is `FALSE & NA`?

Solutions:

```
TRUE | NA
```

```
[1] TRUE
```
```
TRUE & NA
```

```
[1] NA
```
```
FALSE | NA
```

```
[1] NA
```
```
FALSE & NA
```

```
[1] FALSE
```

Exclusive Or

xor() indicates elementwise exclusive OR (True if x *or* y is true, but not if both are true). That is, xor(TRUE, TRUE) is FALSE

```r
x <- c(TRUE,  TRUE, FALSE, FALSE,   NA,    NA,  NA)
y <- c(FALSE, TRUE, FALSE,  TRUE, TRUE, FALSE,  NA)
xor(x, y)
```

```
[1]  TRUE FALSE FALSE  TRUE    NA    NA    NA
```

any() and all()

any() and all() are generalizations of OR and AND for more than 2 values.

```r
u <- c( TRUE,  TRUE,  TRUE)
c(any(u), all(u))
```

```
[1] TRUE TRUE
```

```r
v <- c( TRUE,  TRUE, FALSE)
c(any(v), all(v))
```

```
[1]  TRUE FALSE
```

```r
w <- c(FALSE, FALSE, FALSE)
c(any(w), all(w))
```

```
[1] FALSE FALSE
```

Question

If we add a 0-length vector (like `logical(0)` or `NULL`) to a vector of logical values `v`, it should not affect the result of `any()` or `all()` applied to `v`.

That is:

- `any(logical(0), v)` should produce the same result as `any(v)`
- `all(logical(0), v)` should produce the same result as `all(v)`

With these rules in mind:

- What does `any(logical(0))` return?
- What does `all(logical(0))` return?

any(logical(0)) is FALSE

```
x <- logical(0)
any(x)
```

```
[1] FALSE
```

```
u <- c( TRUE,  TRUE,  TRUE)
c(any(u), any(x, u))
```

```
[1] TRUE TRUE
```

```
v <- c( TRUE,  TRUE, FALSE)
c(any(v), any(x, v))
```

```
[1] TRUE TRUE
```

```
w <- c(FALSE, FALSE, FALSE) # if any(logical(0)) were TRUE, this would change
c(any(w), any(x, w))
```

```
[1] FALSE FALSE
```

all(logical(0)) is TRUE

```
x <- logical(0)
all(x)
```

```
[1] TRUE
```

```
u <- c( TRUE,  TRUE,  TRUE) # if all(logical(0)) were FALSE, this would change
c(all(u), all(x, u))
```

```
[1] TRUE TRUE
```

```
v <- c( TRUE,  TRUE, FALSE)
c(all(v), all(x, v))
```

```
[1] FALSE FALSE
```

```
w <- c(FALSE, FALSE, FALSE)
c(all(w), all(x, w))
```

```
[1] FALSE FALSE
```

Comparison operators

Comparison operators produce logical values. They are vectorized and perform recycling.

- x == y
- x != y
- x < y
- x > y
- x <= y
- x >= y
- x %in% y (vectorized for x only)

see help(Comparison)

Comparison operators

For character strings, comparison operations are based on alphabetical order, with the following general arrangement:

```
symbols < "0" < .. < "9" < "a" < "A" < "b" ... < "Z"
```

```
"10" < "2" # with characters, 10 is alphabetized before 2
```

```
[1] TRUE
```

```
10 < 2 # of course, numerically 10 is not less than 2
```

```
[1] FALSE
```

Just for reference, R arranges the ASCII symbols as follows in accordance with ASCII table https://en.wikipedia.org/wiki/ASCII:

"-" "!" "#" "$" "%" "&" "(" ")" "*" "," "." "/" ":" ";" "?" "@" "[" "\\" "]" "^" "_" "(back-tick)" "{" "|" "}"

"~" "+" "<" "=" ">"

is functions

The is.____() family of functions return logical values that can be used in conditional statements.

- is.na()
- is.null()
- is.atomic()
- is.logical()
- is.integer()
- is.double()
- is.numeric()
- is.character()
- is.list()
- is.matrix()
- is.array()
- is.data.frame()
- is.factor()
- is.function()

Section 2

Condtional statements

Conditional statement - `if`

Conditional execution of code blocks is achieved via `if()` statements.

`if(cond)` The condition in an `if()` statement must be a **length-one logical vector that is not NA**.

Conditions of length greater than one result in an error. (Versions of R prior to 4.2.0 accepted longer vectors with a warning, but only the first element is used.) Other types are coerced to logical if possible.

If the condition results in `NA`, you will get an error: `missing value where TRUE/FALSE needed`

If the condition is length 0, you will get an error: `argument is of length zero`

Curly braces { } are used to group the expressions that will run when the condition inside the `if` statement is TRUE. If there is only one expression to execute, the curly braces are optional.

```
x <- c(1, 3)
1 %in% x
```

```
[1] TRUE
```

```
if (1 %in% x) {
    print("Hello!")
}
```

```
[1] "Hello!"
```

if() produces error with logical vector length > 1

```
x <- c(1, 3)
x <= 2
```

```
[1]  TRUE FALSE
```

```
if (x <= 2){
  # we get an error because the length of the logical vector > 1
    print("Hello again!")
}
```

```
Error in if (x <= 2) {: the condition has length > 1
```

Using logical vector length > 1 with any() or all()

```r
x <- c(1, 3)
any(x >= 2)  # if your logical vector has length > 1, you might use any/all
```

```
[1] TRUE
```

```r
if (any(x >= 2)) {
    print("Can you hear me now?")
}
```

```
[1] "Can you hear me now?"
```

if() produces error for NA

```
x <- 1:3
if (x[5] >= 2){
    print("Will this work?")
}
```

Error in if (x[5] >= 2) {: missing value where TRUE/FALSE needed

missing value where TRUE/FALSE needed is a very common error that you will run into.
You'll need to go through your code and see why something inside your if statement is
producing an NA value.

if() produces error for `logical(0)`

```r
x <- 1:3
which(x == 4)
```

```
integer(0)
```

```r
if (which(x == 4) == 4){
    print("Will this work?")
}
```

```
Error in if (which(x == 4) == 4) {: argument is of length zero
```

Nesting Conditionals - `if`, `else if`, and `else`

If you want to use an `else` statements, there must be a preceding `if` statement and you must use curly braces. The conditional inside an `else if` statement will only be evaluated if the starting `if` statement is FALSE. Make sure you put the `else` on the same line as the closing curly brace of the `if` statement, otherwise R will believe the `if` statement is complete.

```r
x <- 3
if (x < 0) {
   print("Negative")
} else if (x > 0) {
   print("Positive")
} else {
   print("Zero")
}
```

```
[1] "Positive"
```

Nested conditionals

```r
x <- 0
if (x < 0) {
   print("Negative")
} else if (x > 0) {
   print("Positive")
} else {
   print("Zero")
}
```

```
[1] "Zero"
```

Nested conditionals

```r
x <- 3
if (x > 0) {
   print("Positive")
} else if (x > 1) {   # will not be evaluated because the first if is TRUE
   print("Bigger than 1")
} else {
   print("Not Positive")
}
```

```
[1] "Positive"
```

if() is not vectorized, ifelse() is vectorized

if() requires a logical vector of length-one. It is not vectorized. Functions that use an if() statement cannot be given a vector of values.

The function ifelse is vectorized. It requires three arguments:

- a condition to test
- the result if the condition is true
- the result if the condition is false

```r
if(x == 5) {
    n <- "yes"
} else {
    n <- "no"
}
```

is reduced to:

```r
n <- ifelse(x == 5, "yes", "no")
```

Section 3

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
```

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

```
# 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

```
# 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

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

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

while loops

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

```
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

```
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

```
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
```

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

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

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

```
seq_along(l)
```

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

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

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

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

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

```
 [1] 1 0
```

```
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

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

```
integer(0)
```

```
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 vector
```

```
integer(0)
```

Section 4

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:

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

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

```
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

```
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
plot(x, plot_values, type = "l")
```
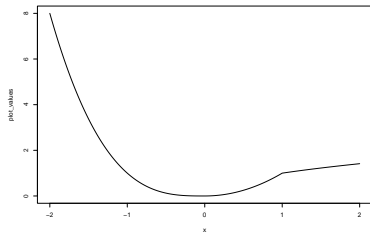
```
Error: object 'plot_values' not found
```

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")
```

Vectorize with `ifelse`

Instead of using `if()`, we can use `ifelse()` to vectorize the function.
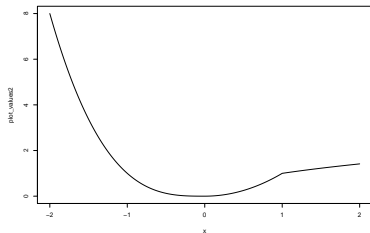
```r
f2 <- function(x) {
  ifelse(x <= 0, # check if x <= 0
         -x ^ 3, # if x <= 0 is true, return x ^ 3
         ifelse(x <= 1,  # if x <= 0 is false, check if x <= 1
                x ^ 2,   # if x <= 1 is true, return x ^ 2
                sqrt(x), # otherwise return sqrt(x)
                )
        )
}

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

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

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 NaNs 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.

```
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
```

```
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

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

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

All three methods produce the same results

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

[1] TRUE

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

[1] TRUE