STATS 266 Handout - Control Flow

Qi Wang

2025-03-01

Contents

1 Introduction

Welcome to **STATS 266: Introduction to R**. This handout provides an introduction about control flow in R. By the end of this document, you should be able to:

- Write conditional statements with if...else statements and ifelse().
- Write and understand `for()` loops.
- Write and understand `while()` loops.

For this part, valuable materials to refer to include https://www.datacamp.com/tutorial/tutorial-on-loops-in-r and https://swcarpentry.github.io/r-novice-gapminder/07-control-flow.html.

2 Choices/Conditions

Sometimes, we want to run different codes under different conditions. For example,

```
x <- 1
if(x > 0){
  print("the variable x is positive")
}
```

```
## [1] "the variable x is positive"
```

However, for the previous case, if x has a negative value, the condition in `if()` will not be satisfied, therefore the code under the `if()` chunk will not run. But that is definitely not what we want, we want it to also report "x is negaive" when x is actually negative. Therefore, we can introduce `elseif()` and `else`.

```
x <- 0
if(x > 0){
  print("the variable x is positive")
}else if(x < 0){
  print("the variable x is negative")
}else{
  print("x is zero")
}
```

```
## [1] "x is zero"
```

If we want to transform the variable to binary outcome, we can use the function `ifelse()`:

```r
x <- 1:5
ifelse(x>3, "Bigger than 3", "Not Bigger than 3")
```

```
## [1] "Not Bigger than 3" "Not Bigger than 3" "Not Bigger than 3"
## [4] "Bigger than 3"     "Bigger than 3"
```

```r
ifelse(x %% 2 == 0, "even", "odd")
```

```
## [1] "odd"  "even" "odd"  "even" "odd"
```

Another possible way to do choice is the function `switch()`:

```r
# Define choice
color_choice <- "blue"

# Use switch() to return a message based on choice
message <- switch(color_choice,
  "red" = "You chose red!",
  "blue" = "You chose blue!",
  "green" = "You chose green!",
  "Invalid choice"
)

print(message)
```

```
## [1] "You chose blue!"
```

This is more readable than multiple if-else statements for many choices. We can also use the function `case_when()` from dplyr:

```r
# Load dplyr
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
# Create a sample dataset
data <- data.frame(
  score = c(85, 45, 67, 90, 72, 50, 30)
)

# Categorize scores into Grades
data <- data %>%
  mutate(grade = case_when(
    score >= 80 ~ "A",
    score >= 70 ~ "B",
    score >= 60 ~ "C",
    score >= 50 ~ "D",
    TRUE ~ "F"  # Default case
  ))

# View result
print(data)
```

```
##   score grade
## 1    85     A
## 2    45     F
## 3    67     C
## 4    90     A
## 5    72     B
## 6    50     D
## 7    30     F
```

3 "For" Loop

Loops are core to computer programming. The first important loop is for loop whose basicform is `for(iter in vector)` action.

```r
result = 0
for (i in 1:10) {
  result <- result + i
}
result
```

```
## [1] 55
```

How to understand this? Each time of the loop, the current result is updated to be the previous result plus the current i, and the i is looped from 1 to 10. Therefore, the previous loop is actually adding all integers from 1 to 10. Let's introduce another loop to get all diagonal element of a matrix:

```r
m1 <- matrix(1:16,4,4)
result <- rep(NA,4)
for (i in 1:4) {
  result[i] <- m1[i,i]
}
print(m1)
```

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

```r
print(result)
```

```
## [1]  1  6 11 16
```

4 "While" Loop

In the for loop, we use the set to limit the times of the loop, when all elements in the set has gone through once, the loop ends. However, in the while loop, it's different. First, the argument in the `while()` is a condition, in other words, it's a logical value (TRUE/FALSE). Here is the example:

```r
result = 0
i = 1
while (i <= 10) {
  result <- result + i
  i <- i + 1
}
result
```

```
## [1] 55
```

Imagine if we don't have `i <- i+1`... The loop will never ends! So it's important to update the condition. However, a special case is by using `break()`. This `break` function will jump outside of the current loop:

```r
result = 0
i = 1
while (1) {
  result <- result + i
  if(i == 10){
    break()
  }
```

```
  i = i+1
}
result
```

```
## [1] 55
```

Sometimes, we need double loop:

```
m1 <- matrix(1:16,4,4)
result <- 0
for (i in 1:4) {
  for(j in 1:4){
    result <- result + m1[i,j]
  }
}
result
```

```
## [1] 136
```

5 Repeat Loop

The basic structure of a repeat loop is:

```
# repeat {
#   if (condition) {
#     break  # Exit the loop
#   }
# }
```

Here is an example of printing numbers from 1 to 5:

```
# Print numbers from 1 to 5 using repeat loop
counter <- 1

repeat {
  print(counter)  # Print the counter value
  counter <- counter + 1  # Increment the counter

  if (counter > 5) {
    break  # Exit the loop when counter exceeds 5
  }
}
```

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

```
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

When do we use repeat loop? One possible case is the keyword input. A common use case for a repeat loop is continuously asking for user input until a valid condition is met.

```
# repeat {
#   password <- readline(prompt = "Enter password: ")
#
#   if (password == "stat226") {
#     print("Access Granted!")
#     break  # Exit loop if password is correct
#   } else {
#     print("Incorrect password. Try again.")
#   }
# }
```

Repeat loops are not as widely use as for and while loop. If you are interested in them, refer to: https://www.datacamp.com/tutorial/tutorial-on-loops-in-r.

6 Vectorization

Although loops are powerful and necessary parts for any language, vectorization is the specific an even more powerful way in R to avoid loops and improve the computational efficiency. We have seen part of the vectorization in R:

```
2*(1:5)
```

```
## [1]  2  4  6  8 10
```

```
1:100 + 101:200
```

```
##   [1] 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136
##  [19] 138 140 142 144 146 148 150 152 154 156 158 160 162 164 166 168 170 172
##  [37] 174 176 178 180 182 184 186 188 190 192 194 196 198 200 202 204 206 208
##  [55] 210 212 214 216 218 220 222 224 226 228 230 232 234 236 238 240 242 244
##  [73] 246 248 250 252 254 256 258 260 262 264 266 268 270 272 274 276 278 280
##  [91] 282 284 286 288 290 292 294 296 298 300
```

If we don't take the advantage of vectorization...

```r
n <- 10000000
x <- 1:n
y <- (n+1):2*n
z <- rep(NA, n)

t_loop <-
system.time(
for (i in 1:n) {
  z[i] <- x[i] + y[i]
}
)
t_vec <-
system.time(
z <- x + y
)
print(t_loop)
```

```
##    user  system elapsed
##    0.38    0.03    0.41
```

```r
print(t_vec)
```

```
##    user  system elapsed
##    0.01    0.02    0.03
```

Note: The function system.time returns a named vector with:

- user: (CPU time spent in R),
- system: (CPU time spent in system calls),
- elapsed: (wall-clock time).

One of the most important examples of vectorization is the function `apply()`.

```r
m1 <- matrix(1:16,4,4)
m1
```

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

```r
apply(m1, 1, sum)
```

```
## [1] 28 32 36 40
```

```r
apply(m1, 2, sum)
```

```
## [1] 10 26 42 58
```

```r
rowSums(m1)
```

```
## [1] 28 32 36 40
```

```r
colSums(m1)
```

```
## [1] 10 26 42 58
```

Let's compare the time if we use the `apply` function and use loop:

```r
n <- 2000
m1 <- matrix(1:n^2,n,n)
res <- rep(NA, n)
t_loop <-
system.time(
for (i in 1:n) {
  tmp_res <- 0
  for (j in 1:n) {
    tmp_res <- tmp_res + m1[i,j]
  }
  res[i] <- tmp_res
}
)
t_vec <-
system.time(
apply(m1, 1, sum)
)
print(t_loop)
```

```
##    user  system elapsed
##    0.10    0.00    0.09
```

```r
print(t_vec)
```

```
##    user  system elapsed
##    0.02    0.02    0.03
```

```r
print(sum(res==apply(m1, 1, sum)))
```

```
## [1] 2000
```

In the previous example, we did the same task via both a loop and an `apply` function. We can see the vectorization operations can significantly improve the computational efficiency of the program.

7 Ackowledgement

This teaching material is adapted from the previous material of this course made by Marcela Alfaro-Córdoba and Sheng Jiang.