R-Ladies NL Book-Club

Advanced R: Control Flows (Chapter 5)

Margaux

2020-05-25

Welcome R-Ladies Netherlands Book-Club!

- R-Ladies is a global organization to promote gender diversity in the R community via meetups, mentorship in a safe and inclusive environment.

- **R-Ladies Netherlands Book-Club** is a collaborative effort between RLadies-NL chapters in Nijmengen, Rotterdam, Den Bosch, Amsterdam, Utrecht.
- We meet every **2 weeks** to go through one of the chapters of Hadley Wickam *Advanced R*, and run through exercises to put the concepts into practice.

Today's Session!

- Starts with a 30-45 min presentation
- Breakout session - we **split** into breakout rooms to practice exercises.

- Please use the **HackMD** (shared in email and in chat) to present yourself, ask overarching questions, and to find your break out room.
- Use the chat to participate in the discussion during the presentation and your breakout session.
- The Bookclub github repository has also been made available.
- Any questions?

Resources

- Solutions to the exercises from *Advanced R* can be found in the (Advanced R Solutions Book)[https://advanced-r-solutions.rbind.io/index.html]
- The R4DS book club repo has a Q&A section section.https://github.com/r4ds/bookclub-Advanced_R
- We are always looking for new speakers! If you are interested, please sign up to present a chapter at https://rladiesnl.github.io/book_club/

Control Flows

Outline

The outline for today is:

1. What do we mean by control flows?
2. Choices
    - if()
    - ifelse()
    - switch()
3. Loops
    - for loops
    - break
    - next
    - `repeat()` and `while()`
4. Breakout Sessions

Let's get to it!

1. What do we mean by control flows

- Controls flows are a **fundamental concept** in computer programming
- Allow us to express the **order** and the **way** a command of execution components are put together to perform a specific task.
-
    - Control flow commands allow your R code to choose between different options, in other words, **make decisions**.
- Control flows are used to:
    - Execute a action using **certain conditions** --> ifelse()
    - Execute an action **repetitively** --> for loop
    - **manipulate a sequential flow** --> breaking code

1. What do we mean by control flows

- There are 3 main groups of control flows in programming:
    1. Sequencing (do this, THEN this, THEN this ...)
    2. Selection/choices (if, unless)
    3. Iteration (for, while, repeat...)

Examples of control flows:

1. Convert a list of daily recorded air temperatures from Fahrenheit to Celcius.

1. Produce a function but skips all input values that aren't numeric.
2. Prevent an iterative function from performing if the input value is NA.

- In the chat box, volunteer and share an example, of a recently use of control flows commands.

1. What do we mean by control flows

- In Hadley Wickam's book chapter, we look at **choices** and **loops**

2. Choices

- Choices are expressed using **If statement**

`if (condition){true_action}`

- if the condition is true, than the action is evaluated

`if (condition){true_action} else {false_action}`

- Using `else`, an optional other action can be evaluated if the condition is `FALSE`.

- in R, we use `{}` to compound the action statements.

School Grade Example

Example given using a function to translate to letter grades:

```
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
    "B"
  } else if (x > 50) {
    "C"
  } else {
    "F"
  }
}
```

If student gets a above 90, received A. If student gets above 80, recieves B. If student gets above 50, receives C. If the grade does not meet this above conditions, student receives F.

Note the order of conditional statements here

`Else` in R-Optional

Choice statements don't always need an `else`. `if` invisibly returns `NULL` if the condition is `FALSE`.

```
x = 3

if(x == 3){

  print('yes!')

  }
```

```
## [1] "yes!"
```

```
x = 3

if(x == 1){

  print('yes!')

  }
```

`Else` in R-Optional

Longer example

```
## greetings is a function that concatenates words depending on condition:

greetings <- function(name, birthday = FALSE) {

  paste0("Hi ", name,
         if (birthday) " and HAPPY BIRTHDAY")

}
```

```
greetings("Maria", FALSE)
```

```
## [1] "Hi Maria"
```

```
greetings("Jaime", TRUE)
```

```
## [1] "Hi Jaime and HAPPY BIRTHDAY"
```

2.1 Invalid Inputs

The `condition` inputed in the `if()` function must be evaluated to a `TRUE` or `FALSE`. Here are some examples of inputs that are invalid:

```
if ("x") 1
```

```
## Error in if ("x") 1: argument is not interpretable as logical
```

```
if (logical()) 1
```

```
## Error in if (logical()) 1: argument is of length zero
```

```
if (NA) 1
```

```
## Error in if (NA) 1: missing value where TRUE/FALSE needed
```

2.1 Invalid Inputs

Another invalid input are logical vectors of **length greater than 1.**

```
vector <-c("a","b","c")

if(vector == "a") print("yes!!")
```

```
## Warning in if (vector == "a") print("yes!!"): the condition has length > 1 and
## only the first element will be used

## [1] "yes!!"
```

- notice that `'yes!!'` result still appears, because it is the first element in the vector

```
vector <-c("a","b","c")

if(vector == "b") print("yes!!")
```

```
## Warning in if (vector == "b") print("yes!!"): the condition has length > 1 and
## only the first element will be used
```

- doesn't work this time! But all we get a is a **warning**, not an **error**.

...This brings us to our next section...

2.2 Vectorised if

- There are three other functions presented that can be used as alternative to `if()`:
  - `ifelse()`
  - `dplyr::case_when()`
  - `switch()`

2.2 `ifelse()`

- `ifelse()` function can handle vectors longer than 1
- This function **tests** the condition:

  `ifelse(condition, action if TRUE, action if FALSE)`

- Taking my vector used above. With ifelse(), our output will be a list:

```
vector <-c("a","b","c")

ifelse(vector == "b","yes!!","no!")
```

```
## [1] "no!"   "yes!!" "no!"
```

2.2 ifelse()

- Again, taking from the examples from the chapter:

```
x <- 1:10

# print xxx when the remainder of x divided by 5 is 0.

ifelse(x %% 5 == 0, "XXX", x)
```

```
##  [1] "1"   "2"   "3"   "4"   "XXX" "6"   "7"   "8"   "9"   "XXX"
```

```
# print 'even' when  when the remainder of x divided by 2 is 0, print 'odd'.

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

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

2.2 case_when()

- Another example presented is the `dplyr::case_when()` which allows for **multiple conditions.**

In this example, we have 3 different conditions to apply:

```
dplyr::case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "???",
  TRUE ~ as.character(x)
)
```

```
##  [1] "1"    "2"    "3"    "4"    "fizz" "6"    "buzz" "8"    "9"    "fizz"
```

2.3 `switch()` statement

- `switch()` is closely related to the `if()` statement.
- Typically, we use `if()` in the following way:

```
x_option <- function(x){

  if (x == "a") {
    "option 1"
  } else if (x == "b") {
    "option 2"
  } else {
    stop("Invalid `x` value")
  }
}
```

- `switch()` is more succinct:

```
x_option <- function(x) {
  switch(x,
    a = "option 1",
    b = "option 2",
    stop("Invalid `x` value")
  )
}
```

2.3 `switch()` statement

- Indeed, each condition is listed and no need for `else`.
- **Note! The last component of the `switch()` should throw an error.**

- With `switch()`, if multiple inputs have the same output, `switch()` can be written the following way:

```
legs <- function(x) {
  switch(x,
    cow = ,
    dog = 4,         ## cat and dog are given the condition "= 4"
    human = ,
    chicken = 2,      ## human and chicken are given the condition "= 2"
    plant = 0,
    stop("Unknown input")
  )
}
```

Note that it is recommended that the inputs for `switch()` function is recommended to be of type character.

Breather

3.0 Loops

- For loops are used to perform an action **iteratively** over indices in a vector.
- In R, the format is the following:

  `for (item in vector) action_to_perform`

- So, for example, action is to **print** every item in a vector.

```
for (i in 1:3) {

  print(i)

  }
```

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

3.0 Loops

```
vector <- 1:4

for (j in vector){

 print(

   j/(j+1)

   )
}
```

```
## [1] 0.5
## [1] 0.6666667
## [1] 0.75
## [1] 0.8
```

3.0 Loops

- And with an `if else` statement:

```
vector <- 1:8

for (k in vector){

if(k < 4){
  print(
    paste0(k," is less than 4!")
  )

} else{
  print(
    paste0(k," is greater than 4!"))

      }
  }
```

```
## [1] "1 is less than 4!"
## [1] "2 is less than 4!"
## [1] "3 is less than 4!"
## [1] "4 is greater than 4!"
## [1] "5 is greater than 4!"
## [1] "6 is greater than 4!"
## [1] "7 is greater than 4!"
## [1] "8 is greater than 4!"
```

Overwriting variables with for loops

- A for loop will overwrite a previously defined variable.

```
i <- 100

for (i in 1:3) {
  print(i)
  }
```

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

```
print(i)
```

```
## [1] 3
```

Terminating For Loops **early**

2 ways to terminate a for loop early:

- `next` to exit the current interation
- `break` to exit the entire `for` loop

```
for (i in 1:10) {

  if (i < 3)
    next

    print(i)

  if (i >= 5)
    break

}
```

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

- Keep in mind the order of the actions here

3.1 Common pitfalls

1. Preallocate the output container for faster process. Here are two examples:

```
means <- c(1, 50, 20)
out <- vector("list", length(means))

for (i in 1:length(means)) {
  out[[i]] <- rnorm(10,means[[i]])

  }
out
```

```
## [[1]]
##  [1] 0.4965342 1.0589140 1.8629051 1.0445799 2.2064184 1.4839206 1.0722014
##  [8] 1.2060870 1.7629527 3.0216966
##
## [[2]]
##  [1] 51.42726 50.40303 50.33508 49.75692 49.72180 50.10537 49.35783 51.73834
##  [9] 50.29270 52.42563
##
## [[3]]
##  [1] 19.14145 20.95941 19.25223 20.30258 21.41086 20.37636 20.87550 20.02916
##  [9] 17.81960 19.73872
```

3.1 Common pitfalls

1. Preallocate the output container for faster process. Here are two examples:

```
vector <- c("a", "b", "c")
output_list <-list()

for (i in 1:length(vector)){

  output_list[i] <- paste(vector[i],vector[i+1])
}
output_list
```

```
## [[1]]
## [1] "a b"
##
## [[2]]
## [1] "b c"
##
## [[3]]
## [1] "c NA"
```

3.1 Common pitfalls

1. Using 1:length(x) gives error when x has a lenghth of 0.

```
means <- c()
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[i] <- rnorm(10, means[i])
  }
```

```
## Error in rnorm(10, means[i]): invalid arguments
```

```
1:length(means)
```

```
## [1] 1 0
```

3.1 Common pitfalls

1. Using 1:length(x) gives error when x has a lenghth of 0.

- Alternatively, in the for loop, use `seq_along()` instead of `1:length()`:

```
means <- c()

out <- vector("list", length(means))

for (i in seq_along(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}

out
```

```
## list()
```

3.1 Common Loopholes

1. Problems arise when iterating over S3 Vectors (Categorical Data, Dates, Time, etc)

```
dates <- as.Date(c("2020-01-01", "2010-01-01"))

for (i in dates) {
  print(i)
}

## For loop strips the attributes of s3 vectors
```

```
## [1] 18262
## [1] 14610
```

```
dates <- as.Date(c("2020-01-01", "2010-02-01"))

for (i in seq_along(dates)) {

  print(dates[i])

  }
```

```
## [1] "2020-01-01"
## [1] "2010-02-01"
```

3.2 Related tools

For loops are helpful when you know exactly what you want to iterate over.

However, if you do not know what you want to iterate over, there are two other loops we can use.

- `while(){}` performs action when condition is `TRUE`

```
i <- 5

while (i < 6) {

  print(i)

  i = i+1

  }
```

```
## [1] 5
```

3.2 Related tools

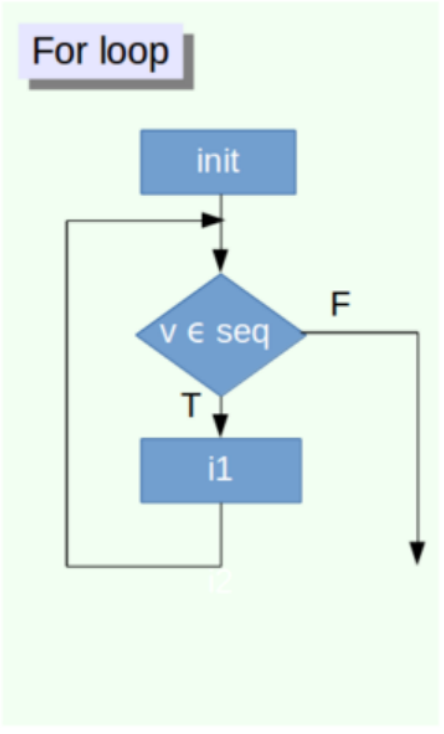
For loops are helpful when you know exactly what you want to iterate over.

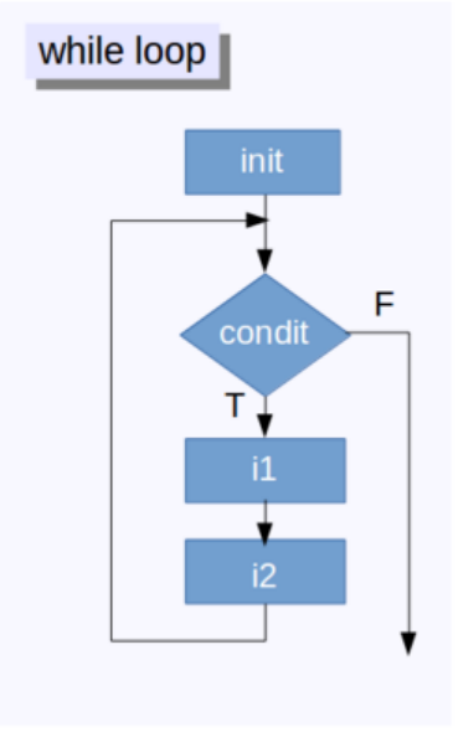
However, if you do not know what you want to iterate over, there are two other loops we can use.

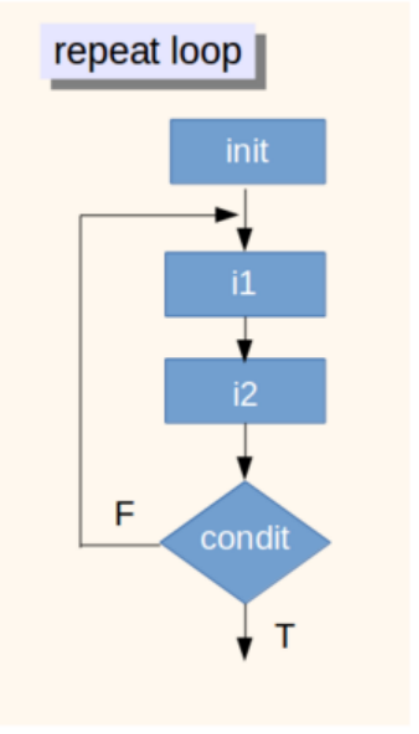
- `repeat(){}` performs action forever - repeat is an infinite loop! A `break` is therefore necessary here.

```
i <- 0

repeat{

  print(i)

  if(i > 4)

    break

  i <- i+1

}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```


For loop
init
v ∈ seq
F
T
i1
i2


while loop
init
condit
F
T
i1
i2


repeat loop
init
i1
i2
F
condit
T

3.3 Nested for loops

Some instances, you will need to put a loop inside a loop!

This is the case if you want to iterate through rows and columns.

```
matrix <- matrix(c(1,2,3, 3,4,5, 1,2,3),
                 nrow=3, ncol=3, byrow = T)
matrix
```

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

```
for(i in 1:dim(matrix)[1]) {
  for(j in 1:dim(matrix)[2]) {

    matrix[i,j] = matrix[i,j] * 2
  }
}

matrix
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    6    8   10
## [3,]    2    4    6
```

Thank you

Questions? Break for 10 min, and meet in your breakout group

Exercises - break out sessions

1. Choices

Q1: What type of vector does each if the following calls to ifelse() return?

```
ifelse(TRUE, 1, "no")
```

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

```
ifelse(FALSE, 1, "no")
```

```
## [1] "no"
```

```
ifelse(NA, 1, "no")
```

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

- Recall that the arguments of ifelse() are `test` , `yes` and `no`.
- The function returns the entry for **yes when test is TRUE**, **no when test is FALSE**, or **NA when test is NA**.

Q2: Why do the following code chunks work?

```
a <- 1:10

if (length(a)) "not empty" else "empty"
```

```
## [1] "not empty"
```

```
a <- numeric()

if (length(a)) "not empty" else "empty"
```

```
## [1] "empty"
```

- Typically, if() expects a logical condition that it can test
    - `if(a>4)`, `if(5 %in% a)`.
- But `if()` also accepts a numeric vector where 0 is treated as false and all other numbers are treated as TRUE.

That is why the condition is:

- TRUE - i.e. `not empty` - when length>0.
- FALSE - i.e. `empty` when length=0.

2. Loops

Q3: Given that x is `length(x) = 0`, why does this code succeed with errors or warnings?

```
x <- numeric()
out <- vector("list", length(x))

for (i in 1:length(x)) {
  out[i] <- x[i] ^ 2
}

out
```

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

Let's break down the code behavior:

- Because the vector is of length 0, the loop goes from `i = 1` to `i = 0`. This works still, because `:` counts down, as well as up.

- During first iteration, `x[1]` will generate `NA` because it is out of the bounds of x. And, `NA^2` leads to `NA`.
- `x[0]` returns `numeric(0)` which does not change when squared. Here, we assign a 0-length vector to a 0-length subset `out[0]` which works but changes nothing.

2. Loops

Q4: What does the following code tell you about when the vector being iterated over is evaluated? Specifically, we are interested in `xs`

```
xs <- c(1, 2, 3)

for (x in xs) {

    xs <- c(xs, x * 2)
    print(xs)

    }
```

```
## [1] 1 2 3 2
## [1] 1 2 3 2 4
## [1] 1 2 3 2 4 6
```

```
xs
```

```
## [1] 1 2 3 2 4 6
```

- `x` takes the values of `xs` which gets redefined in this loop.
- Based on the output, x is evaluated once at the beginning on the initial xs, not after each iteration.

2. Loops

Q5: What does the following code tell you about how/when the index is updated?

```
for (i in 1:3) {

   i <- i * 2
   print(i)

}
```

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

The index is updated in the beginning of each iteration.