Control Statements and Programming with functions

Yebelay Berehan

Biostatistician

yebelay.ma@gmail.com

2022-05-23

Control flow constructs

- Often when we're coding we want to control the flow of our actions.
- Control flow is simply the order in which we code and have our statements evaluated.

1. **Conditional statements**: things to happen only if a condition or a set of conditions are met

    - single condition, single response: `if()`,
    - multiple conditions, multiple responses:
        - `if{} else{}` vs. `ifelse()`
        - extending `if{} else{}`
        - `case_when()`

2. **Repeating operations**: an action to be computed for a particular number of times.

    - perform operation *n* times: `for()`
    - perform operation unlimited times while condition holds: `while() / repeat()`
    - break out of loop or advance to next iteration: `break() / next()`

if()

We use `if()` to execute code when a single condition holds.

1. evaluates if the test expression is `TRUE`
2. if `TRUE`, execute statement
3. if `FALSE`, do nothing

Say, for example, that we want R to print a message depending on the value of `x`

```
if (test_expression) {
  statement
}
```

```
set.seed(10)
x <- runif(1)
if(x > .5) {
  paste("x equals",
        round(x, 2),
        "which is greater than 0.5")
}
```

When `TRUE`

```
## [1] "x equals 0.51 which is greater than 0.
```

What do you expect to happen?

- What do you expect to happen with this example?

```
x <- c(8, 3, -2, 5)
```

```
if(x < 0) { print("x contains a negative number") }
```

⚠⚠ Common mistake ⚠⚠

- `if()` is not vectorized
- `if()` looks for a single logical value
- this code block evaluated the first element in `x` but not the rest

```
if(x < 0) {
  print("x contains a negative number")
}
## Warning message:
## In if (x < 0) { :
##   the condition has length > 1 and on
```

if else

We can extend an `if()` with `else` to return two responses based on a single condition.

1. evaluates if the test expression is `TRUE`
2. if `TRUE`, execute statement 1
3. if `FALSE`, exectute statement 2

```
# syntax of if statement
if (test_expression) {
  statement 1
} else {
  statement 2
}
```

- Now our earlier example will return one result when the test expression returns `TRUE` and another when it returns `FALSE`

```
set.seed(10); x <- runif(1)
if(x > .7) {
  paste("x equals", round(x, 2),"which is greater than 0.7")
} else {
  paste("x equals", round(x, 2),"which is less than 0.7")
}
```

A right way to do things and a wrong way

- which statement is correct and which is not? Why?

```
if(x > .5) {
  TRUE
} else {
  FALSE
}
```

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

```
if(x > .5) {
  TRUE
}
else {
  FALSE
}
## Error: unexpected '}' in "}"
```

⚠ Common mistake: `else` must start on the same line as the closing `}` of the prior `if` statement.

Considering multiple conditions

We can continue to expand an `if...else` statement:

```
if(x < 3) {
  print("x has low tolerance")
} else if(x == 3 & x < 8) {
  print("x has moderate tolerance")
} else {
  print("x has high tolerance")
}
```

```
## [1] "x has low tolerance"
```

Note how we extend by following `else` with `if()`. But we should always end with an `else`.

Loop

- Often, we need to execute repetitive code statements a particular number of times.
- Or, we may even need to execute code for an undetermined number of times until a certain condition no longer holds.
- `for` loops: execute code for a prescribed number of times (Rule of 👍, 3x).
- `while` & `repeat` loops: repeat code while condition holds
- `next` & `break`: additional clauses to control flow

The for() Loop

- `for()` loops are the most general kind of loop, found in pretty much every programming language.
- For each of these values—in order—do this

Conceptually:

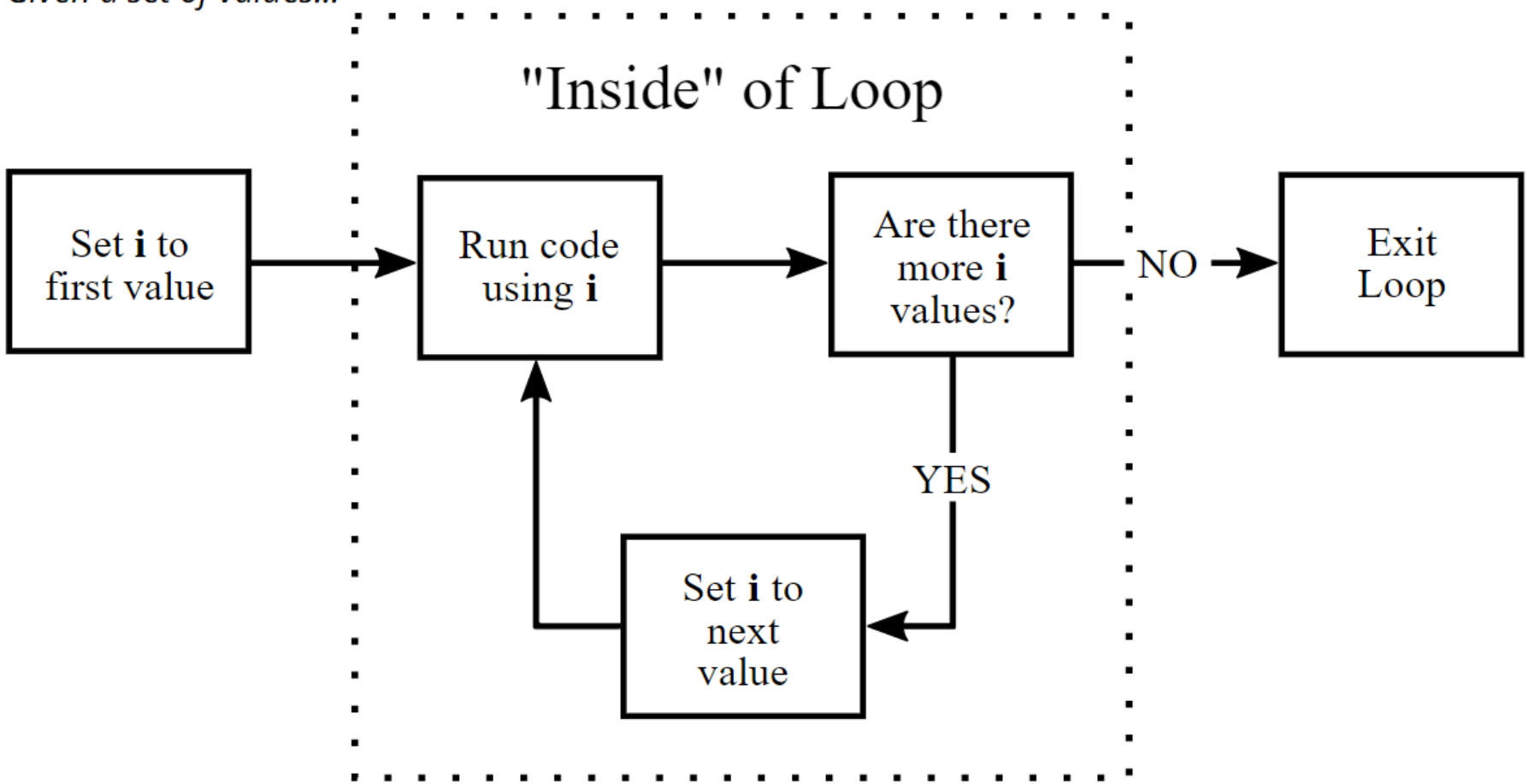
- Given a set of values...

1. You set an index variable (often `i`) equal to the first value
2. Do some set of things (usually depending on current value)
3. Is there a next value?
    - `YES:` Update to next value, go back to 2.
    - `NO:` Exit loop

- We are looping through values and repeating some actions.

for() Loop: Diagram

- *Given a set of values...*



- `for()` Loop: Example

```
for(i in 1:10) {
    # inside for, output won't show up without print()
    print(i^2)
}
```

- Note this runs 10 separate print commands, which is why each line starts with [1].
- These Do the Same Thing

```
for(i in 1:3) {
print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
```

```
i <- 1
print(i^2)
i <- 2
print(i^2)
i <- 3
print(i^2)
```

`for` loop: example

- For each element in sequence (1-100) perform some defined task
- What will this `for` loop do?
- `i` takes on the value of the numbers in the sequence

```
for(i in 1:100) {
  <do stuff here with i>
}
```

```
for (i in 2010:2017) {
  print(paste("The year is", i))
}
```

```
## [1] "The year is 2010"
## [1] "The year is 2011"
## [1] "The year is 2012"
## [1] "The year is 2013"
## [1] "The year is 2014"
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
```

`for` loop: sequence input

- Often, the length of the sequence is determined by some pre-defined variable
- For example, say we have a variable `x` and we want to iterate over each element and do something
- We can do this two ways

```r
x <- 2010:2017
# option 1
for(i in 1:length(x)) {
  <do stuff here with x[i]>
}
# option 2
for(i in seq_along(x)) {
  <do stuff here with x[i]>
}
```

Talk to your neighbor. Do you prefer one over the other? 🤔

`for` loop: sequence input

- Often, the length of the sequence is determined by some pre-defined variable
- For example, say we have a variable `x` and we want to iterate over each element and do something
- We can do this two ways
- But `seq_along(x)` is safer for edge cases where the sequence has length zero

```
x <- c()
# option 1: executes although
for(i in 1:length(x)) {
  print(i)
}
```

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

```
# option 2: does not execute
for(i in seq_along(x)) {
  print(i)
}
```

Beware iterating over `1:length(x)`. This will fail in unhelpful ways if `x` has length 0

`for` loop: generating output

- Often we are generating output from the `for` loop that we want to use later
- We can do this by either growing or filling the output
- Pre-allocating the output and filling in the results is more efficient

```
# Let's create 100 different vectors
means <- seq(0, 100, by = 1)
out <- list()
system.time({
  for(i in seq_along(means)) {
    gen_data<-rnorm(10000,means[[i]])
    out <- c(out, gen_data)
  }
})
```

```
##    user  system elapsed
##    1.53    0.14    1.72
```

```
out<- vector("list", length(means))
system.time({
  for(i in seq_along(means)) {
    out[[i]]<- rnorm(10000, means[[i]])
  }
})
```

Controlling sequences

- There are two ways to control the progression of a loop:
  - `next`: terminates the current iteration and advances to the next.
  - `break`: exits the entire `for` loop.

```
x <- c(1, 2, NA, 3)
for (i in x) {
  if (is.na(i)) next#if NA don't execute
  print(i)
}
```

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

```
for (i in x) {
  if (is.na(i)) break # if NA stop!
  print(i)
}
```

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

Warning

Note that `break` and `next` only control the progression of the direct loop environment they are in.

Break out of inner loop only

```
x <- 1:3
y <- c("a", "b", NA)
for(i in x) {
  print(i)
  for(j in y) {
    if(is.na(j)) break
    print(j)
  }
}
```

```
## [1] 1
## [1] "a"
## [1] "b"
## [1] 2
```

Stop entire process.

```
x <- 1:3
y <- c("a", "b", NA)
for(i in x) {
  print(i)
  for(j in y) {
    if(is.na(j)) stop
    print(j)
  }
}
```

```
## [1] 1
## [1] "a"
## [1] "b"
## [1] NA
```

Repeating code for undefined number of iterations

- Sometimes we need to execute code for an undetermined number of times until a certain condition no longer holds.
- Two very similar options:

while loop

```
while(condition) {
  <do stuff>
}
```

- Test condition first
- Then execute code

repeat loop

```
repeat {
  <do stuff>

  if(condition) break
}
```

- Execute code first
- Then test condition

Repeating code for undefined number of iterations

- The probability of flipping 10 coins and getting all heads or tails is $(\frac{1}{2})^{10} = 0.0009765625$ (1 in 1024 tries). How quickly can we achieve this accomplishment?

Using `while`

```
coin <- c("heads", "tails")
n_tries <- 0
flip <- NULL
while(length(unique(flip)) != 1) {
   flip<- sample(coin, 10,replace= TRUE)
    # add to the number of tries
  n_tries <- n_tries + 1
}
n_tries
```

```
## [1] 49
```

Using `repeat`

```
coin <- c("heads", "tails")
n_tries <- 0
repeat {
  flip<- sample(coin, 10, replace=TRUE)
    # add to the number of tries
  n_tries <- n_tries + 1
#if current flip contains heads/tails
  if(length(unique(flip)) == 1) {
    print(n_tries)
    break
  }
}
```

Stop the while loop: break

- There are some very rare situations in which severe speeding is necessary: what if a hurricane is approaching and you have to get away as quickly as possible?
- This seems like a great opportunity to include the `break` statement in the `while` loop you've been working on.
- Remember that the `break` statement is a control statement.
- When R encounters it, the `while` loop is abandoned completely.

example

Finish the `while` loop so that it:

- *Prints out the triple of `i`, so `3 * i`, at each run.*
- *Loop is abandoned with a `break` if the triple of `i` is divisible by 8, but still prints out this triple before breaking.*

```
# Initialize i as 1
i <- 1
while (i <= 10) {
  print(i  3)
  if (i  3 %% 8 == 0) {
    break
  }
  i <- i + 1
}
```

.pull-right[

```
## [1] 3
## [1] 6
## [1] 9
## [1] 12
## [1] 15
## [1] 18
## [1] 21
## [1] 24
## [1] 27
## [1] 30
```

Writing Functions

Why Write Your Own Functions?

- A function centralizes a common task to a single, abstract method
- This helps to:
  - increase reusability
  - reduces verbosity
  - reduces repetitiveness
  - reduces the chance of making an error
  - reduces instances requiring updates when changes are needed
  - reduces amount of code to test

Consider the following. Where's the error?

```
df <- tibble::tibble(a = rnorm(10),  b = rnorm(10),
  c = rnorm(10),  d = rnorm(10))

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Key ingredients of a function

1. Name
2. arguments
3. body
4. environment

```
my_fun <- function(arg1, arg2) {
  << body >>
 }
```

Defining a function

- We define a function with `<-` just like we define any other R object
- Use informative names; strive to use verbs when possible
- We can define a function with no arguments; however, this is rarely useful
- Consequently, most functions have all key ingredients

```
present_value <- function() {

}
```

```
compute_pv <- function() {

}
```

```
compute_pv <- function() {
  1000 / (1 + 0.05)^10
}
```

```
compute_pv <- function(fv, r, n) {
  fv / (1 + r)^n
}
compute_pv(fv = 1000, r = .05, n = 10)
```

```
## [1] 613.9133
```

Calling arguments in different ways

Many ways to call arguments:

- Using argument names
- Positional matching
- Must use names if you change order
- ...otherwise error or incorrect computation will occur
- missing arguments results in error

```
compute_pv(fv = 1000, r = .05, n = 10)
```

```
## [1] 613.9133
```

```
compute_pv(1000, .05, 10)
```

```
## [1] 613.9133
```

```
compute_pv(r = .05, fv = 1000, n = 10)
```

```
## [1] 613.9133
```

```
compute_pv(.05, 1000, 10)
```

```
## [1] 4.950274e-32
```