UPPSALA
UNIVERSITET

Uppsala Child & Baby Lab

Conditionals, Control Flow, and Functions

R for Psychology Research

Marcus Lindskog, Docent
2019-09-08

Overview

1. Control flow and conditionals
2. Functions

Control flow and conditionals

Control flow

1. In programming you often your code to make a decision...
2. ...or you want your code to run, until a condition is met.
3. This can be achieved with control flow.

How can you control the flow of your code

1. `if...else`: to make a decision
2. `for – loop`: to iterate over a vector
3. `while – loop`: to run your code until a condition is met.

if...else

`if...else`

- You often run into the problem of wanting to run a specific piece of code if a condition is met, but not otherwise.
- Such decisions are best handled in R with the `if...else` statement.
- Basic syntax:

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

- If the `test_expression` is `TRUE` the `statement` will be executed. If it is `FALSE` it will not.

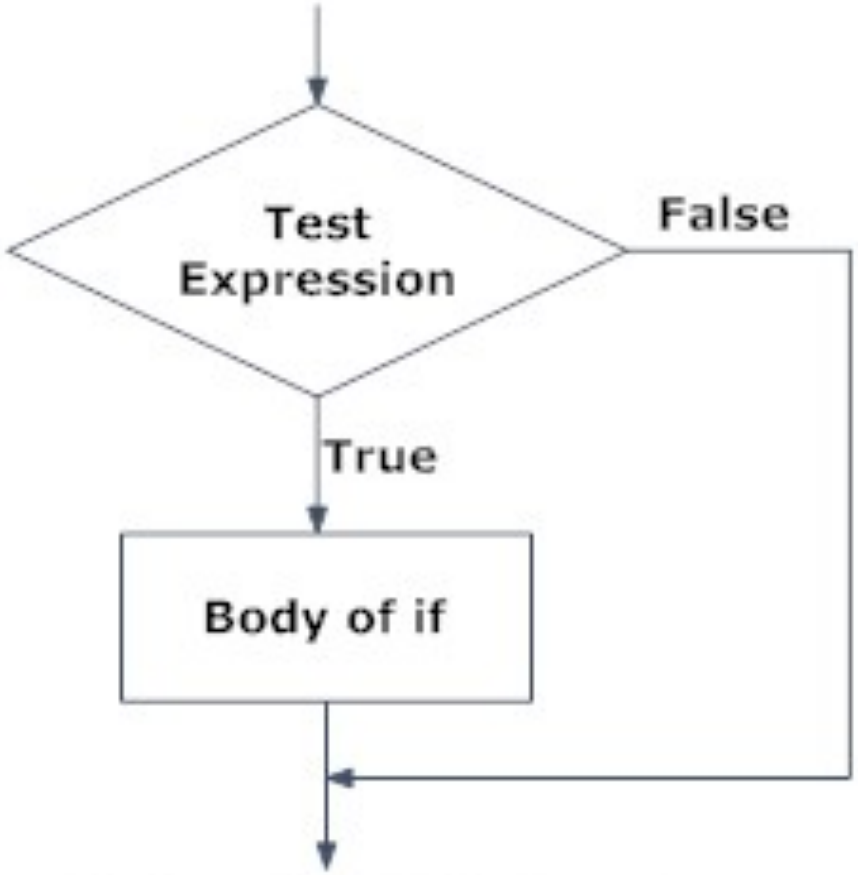

Fig: Operation of if statement

Let's try

```
x <- 5


if(x > 0){
  print("Positive number")
}
```

- See what happens with your code when `x` is changed.

If...else

- Sometimes you know what you want the code to do if the `if` - statement is not met.
- In such cases you can also add an `else` statement.

```
if(test_expression){
  true_statement
} else {
  false_statement
}
```

- Here, the `true_statement` will execute if `test_expression` is `TRUE` and the `false_statement` will execute if it is `FALSE`

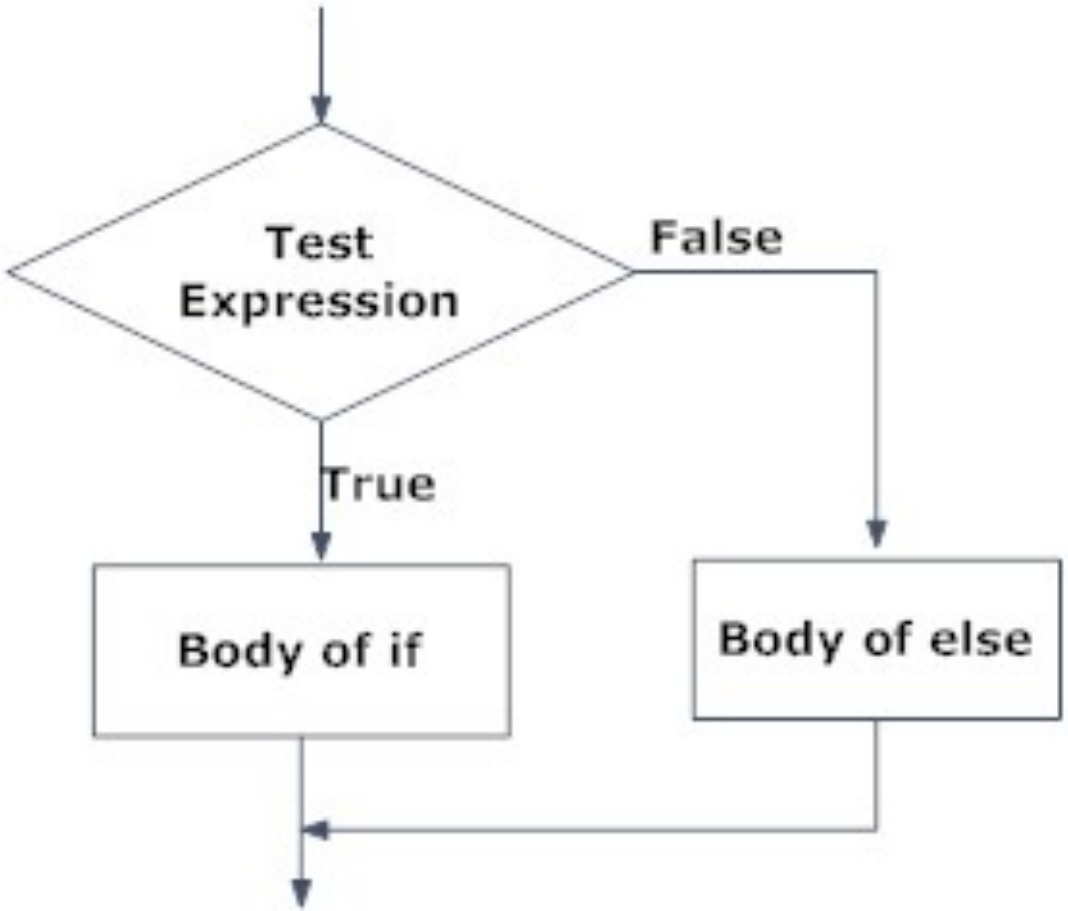

Fig: Operation of if...else statement

Let's try

```
x <- -5

if(x > 0){
  print("Non-negative number")
} else {
  print("Negative number")
}
```

`if...else` ladder

- Sometimes you have more than two conditions to check for.
- This can be achieved with a `if...else` ladder.

```
if(test_exp1){
  statement1
} else if (test_exp2) {
  statement2
} else if(test_exp3){
  statement3
} else
  statement4
}
```

Let's try

```
x <- 0

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

The function `ifelse()`

- Using the full `if...else` we saw above is a powerful way to make decisions.
- But sometimes you just want something that can re-code a variable.
- For that we have the `ifelse()` function

```
ifelse(test, yes, no)

x <- 8

y <- ifelse(x < 9, 1, 0)
```

`for` - loop

`for` - loop

- The `for` - loop is used to iterate over a vector.
- For each item in the vector, the body of the for loop is executed.
- This is an efficient way to carry out an operation a predefined number of times. That is, when you know how many times you want something to happen.

```
for (val in sequence) {
  statement
}
```

- Here `val` is each value in `sequence`. During each iteration, `statement` is executed.

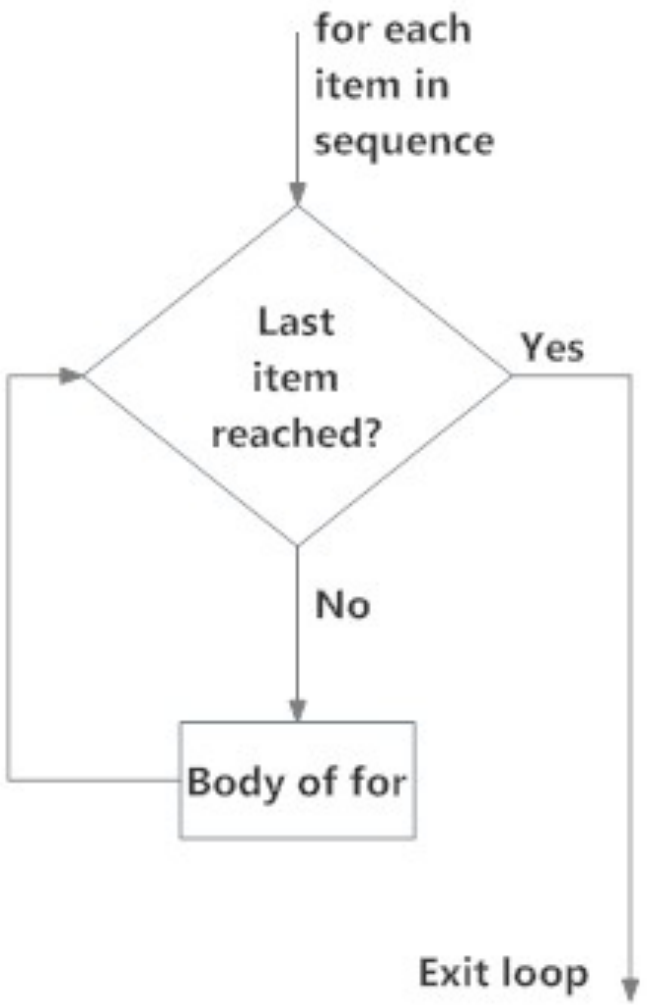

Fig: operation of for loop

Let's try

```
vec <- 1:10

for(val in  vec){
  k = val^2
  print(k)
}
```

Let's try

```
vec <- 1:20

for(val in  vec){
  if(val %% 2 == 0){
    print(paste(val, "is even even"))
  } else{
    print(paste(val, "is even odd"))
  }
}
```

`while`-loop

`while` - loop

- What if you don't know for how long you want to iterate?
- Or if you don't have a vector to iterate over?
- Then you should use a `while` - loop

```
while(test_expression){
  statement
}
```

- Here, `statement` gets executed over and over again, until `test_expression` becomes `FALSE`.
- Put differently, the loop continues as long as `test_expression` is `TRUE`
- Be careful, if your `test_expression` never becomes `FALSE`, the loop never stops.

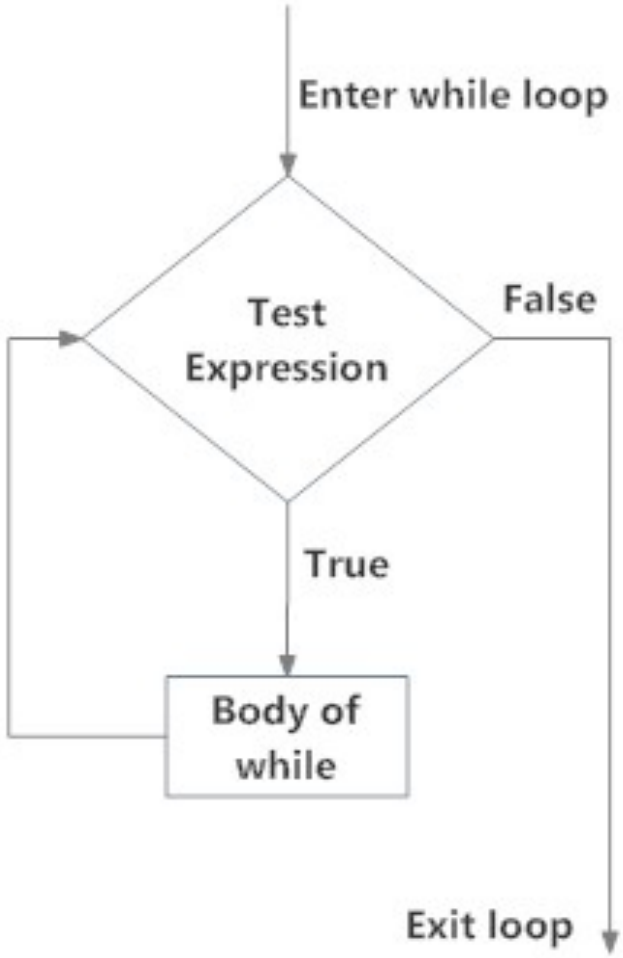

Fig: operation of while loop

Let's try

```
i <- 1
while (i < 6) {
  print(i)
  i = i+1
}
```

Let's try

```
condition <- TRUE
n_iterations <- 0

while(condition){
  rand_number <- runif(1)
  n_iterations <- n_iterations + 1
  if(rand_number < .2){
    condition <- FALSE
    print(paste('The loop ended after', n_iterations,
    'with the number', rand_number))
  }
}
```

Let's try

```
r_criterion <- .3
x <- rnorm(100)
y <- rnorm(100)
r <- cor.test(x, y)
count <- 0

while( abs(r$estimate) < r_criterion ){
  x <- rnorm(100)
  y <- rnorm(100)
  r <- cor.test(x, y)
  count = count + 1
}

print(paste('It took', count,
            'iterations to find a correlation of',
            round(r$estimate, 3) , 'with a p-value of',
            round(r$p.value, 4)))
```

break

- `break` can be used to break a loop.

```
x <- 1:5

for (val in x) {
  if (val == 3){
    break
  }
  print(val)
}
```

next

- `next` can be used to skip an iteration of a loop.

```
x <- 1:5

for (val in x) {
  if (val == 3){
    next
  }
  print(val)
}
```

Functions

Functions

- Functions are an efficient way to automate common tasks.
- You have already bumped into several of the *built in* functions in R e.g.,
  - `c()`
  - `str()`
  - `ifelse()`
  - `mean()`
- R also allows you to write your own functions
- Consider writing a function when you've copied and pasted a block of code more than twice.

Example

```
library(psych)

my_data <- bfi

se_A1 <- sd(my_data$A1, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A1)))

se_A2 <- sd(my_data$A2, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A2)))

se_A3 <- sd(my_data$A3, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A1)))

se_A4 <- sd(my_data$A4, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A4)))
```

Example

```r
library(psych)

my_data <- bfi

se_A1 <- sd(my_data$A1, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A1)))

se_A2 <- sd(my_data$A2, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A2)))

se_A3 <- sd(my_data$A3, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A1)))

se_A4 <- sd(my_data$A4, na.rm = TRUE)/
  sqrt(sum(!is.na(my_data$A4)))
```

function-me!

```
x <- my_data$A1

sd_x <- sd(x, na.rm = TRUE)
sqrt_length_x <- sqrt(sum(!is.na(x)))
sd_x / sqrt_length_x
```

- How many inputs do we have to this code?

Function syntax

```
func_name <- function (argument) {
  statement
}
```

function - me!

```
standard_error <- function(x){
    sd_x <- sd(x, na.rm = TRUE)
    sqrt_length_x <- sqrt(sum(!is.na(x)))
    sd_x / sqrt_length_x
}
```

- This function lives in your global environment as an object. Note that we have assigned the function statement to the variable `standard_error`
- If you **clean your environment**, you will need to assign it again.
- If you **change anything in your function**, you will need to assign it again.

Three key steps

1. Pick a name for your function. Make it a good one.
2. List the *arguments* inside `function`.
3. Place the code you've developed inside the body block `{ }`

- Don't write your function until you can make your code work on simple input.

Check your function on a few outputs.

```
se_A1 <- standard_error(my_data$A1)

se_A2 <- standard_error(my_data$A2)

se_A3 <- standard_error(my_data$A3)

se_A4 <- standard_error(my_data$A4)
```

Function arguments

- Arguments are the objects you pass to a function.
- They come in two broad classes:
  - **data** to compute on.
  - **details** of the computation.
- Generally, **data** argument should come first.
- **Detail** arguments should come at the end and have default values.

```
# Compute confidence interval around
# mean using normal approximation

mean_ci <- function(x, conf = 0.95){
  se <- sd(x)/sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se*qnorm(c(alpha/2, 1-alpha/2))

}
```

What is returned?

- The value returned by a function is usually the last statement it evaluates.
- If the last line of your function is what you want returned. You don't need an explicit `return` statement.

- On occasion, you might want to return values early, or be super obvious what is returned from your code.

- This can be done with the `return()` function

```
standard_error <- function(x){
    sd_x <- sd(x, na.rm = TRUE)
    sqrt_length_x <- sqrt(sum(!is.na(x)))
    return(sd_x / sqrt_length_x)
}
```

Checking values

- If you have functions that where arguments need to have certain properties, it is good practice to check those properties.
- You also want to let the user know, if an argument does not have a certain property and through an error.
- This is done with the `stop()` function.

Let's try

```
two_vector_function <- function(x, y){
  if(length(x) != length(y)){
    stop("'x' and 'y' must be of the same length",
         call. = FALSE)
    }
  (x + y)^2
}

x <- c(2, 3, 4)
y <- c(4, 5)
two_vector_function(x,y)

y <- c(4, 5, 9)
two_vector_function(x,y)
```

A few thoughts on naming

- Your function is both for the computer and for you. Name it properly.
- Use a short name that clearly states what the function does.
- Use a verb for your function name and a noun for arguments.

- Choose a convention and stick with it.
  - `snake_case`
  - `camleCase`

- Make sure you comment your code properly (#)

Environment

- Let's begin with an example

```
a <- 10

assign_value_to_a <- function(){
  a <- 20
}
assign_value_to_a()
print(a)
```

- What do you think will be printed at the end?

Environment

- Let's do one more

```
add_two_values <- function(x){
  x+y
}
print(add_two_values(10))
```

- What do you think will be printed at the end?

```
y <- 10
print(add_two_values(10))
```

- What do you think will be printed at the end?

Environment

- An **Environment** can be thought of as a collection of objects (functions, variables etc.).
- An environment is created when we start the R interpreter.
- Any variable we define, is now in this environment.
- The top level environment available is `R_GlobalEnv`.
- The `ls()` function shows what is in the current environment.

Scope

- **Global variables** exists throughout the execution of a program.
  - They can be changed and accessed from any part of the program
- **Local variables** exist only within a certain part of a program like a function.
  - They are released when the function call ends.
- R uses **lexical scoping** to find a value associated with a name.
- If it can't find a value inside the function, it will look in the environment it was defined.
- The value of a global variable is not changed when calling a function, if you don't explicitly ask for it.

Functions with side effects

- All functions in R return a value.
- Some functions also produce **side effects**:
    - Change variables in the current environment.
    - Plot graphics.
    - Load files.
    - Save files.
- Take care not to write functions that produce side effects, if that's not what you explicitly want.

That's all folks!