Programming

Javier Garcia-Bernardo (based on material by Gerko Vink)

Introduction to R and RStudio

Recap

Last week

- How to use `R`, `RStudio`, `R`-scripts and `R`-notebooks
- Data types (elements)
  - character, numeric, integer, logical, factor
- Data structures: composed of data types
  - vector, matrix, list, **data.frame**
- Subsetting data structures
- Reading files in different formats

This week

How to organize and automate your code:

First half

- Control-flow:
  - Choice: if-else statements
  - Loops: For loops
- Functions
- Environments

Second half

- Principles of tidy data and short comparison of base R and the tidyverse
- Inferential statistics: A primer of linear regression
- Best practices in R

Control-flow

New controls and functions

New control flow constructs

- Choice:
  - We often want to run some code ***only if*** some ***condition*** is true.
  - `if(cond) {cons.expr} else {alt.expr}`
- Loops:
  - We often want to repeat the execution of a piece of code many times.
  - `for(var in seq) {expr}`

Loops in R happen often under the hood. New functions:

- `apply()`: apply a function to margins of a matrix
- `sapply()`: apply a function to elements of a list, `vector` or `matrix` return
- `lapply()`: apply a function to elements of a list, `list` return

Control-flow (I): Choice

If statement

Operation of a if statement:

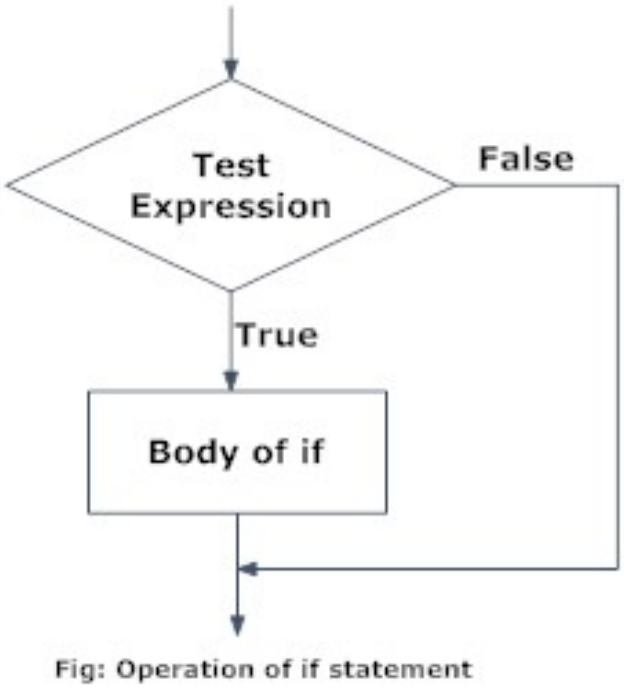


Fig: Operation of if statement

Figure 1: Source: datamentor.io

Code of an if statment:

```
value <- 3
if (value > 3) { #condition
  print("Value greater than 3") #conditional code
}
```

If-else statements

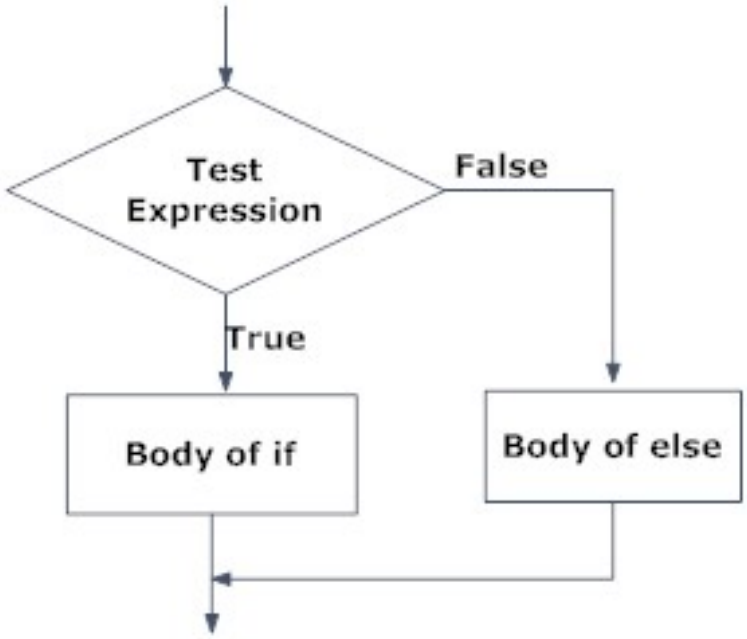


Fig: Operation of if...else statement

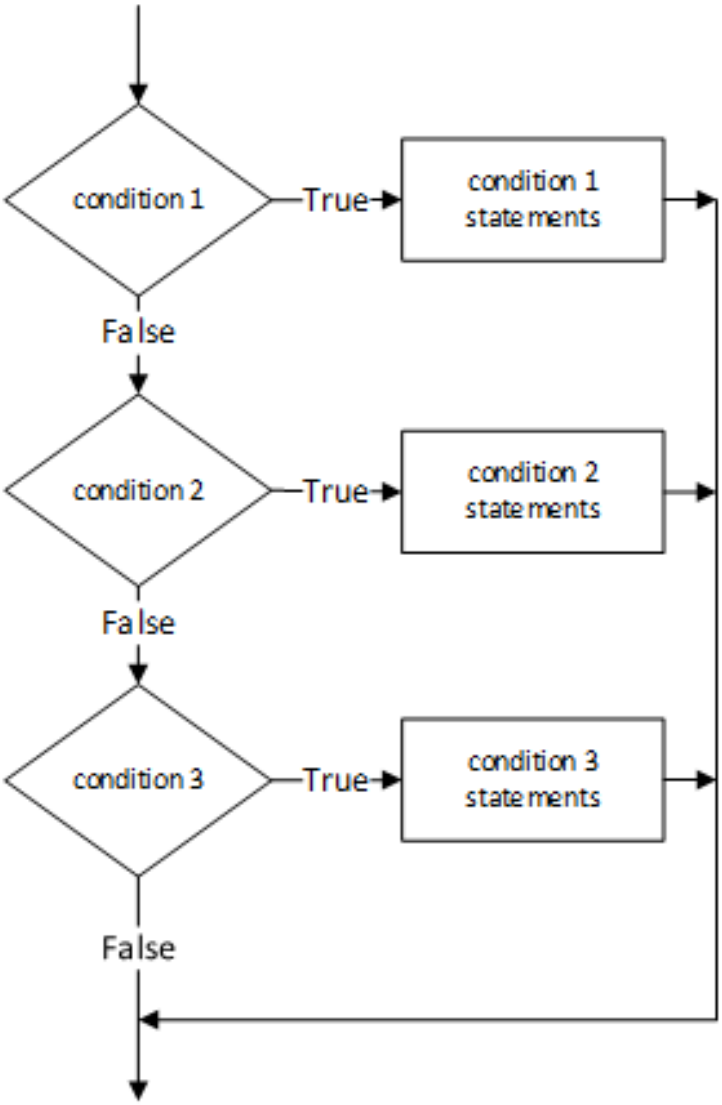
Operation of a if-else statement:

Code of an if-else statment:

```
value <- 3
if (value > 3) { #condition true?
  print("Value greater than 3") #code in if block
} else {
    print("Value lower or equal to one") #code in else block
}
```

```
## [1] "Value lower or equal to one"
```

If-else statements



Operation of a if-else statement:

Code of an if-else statment:

```
value <- 3
if (value > 3) { #condition 1
  print("Value greater than 3") #condition 1 statements
} else if (value > 1) { #condition 2
 print("Value greater than 1")  #condition 2 statements
} else if (value > 0) { #condition 3
 print("Value greater than 0")  #condition 3 statements
}
```

```
## [1] "Value greater than 1"
```

Subsetting consists of if-else statements

Remember our example from last time

The computer keeps the value of the element i of example_vector ***if*** the element i of the condition (`example_vector>3`) is true.

```
example_vector = c(1,2,3,4,5,6,7,8,9)
example_vector>3
```

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

```
example_vector[example_vector>3]
```

```
## [1] 4 5 6 7 8 9
```

Control-flow (II): Loops

For loops

For loops are used when we want to perform some repetitive calculations.

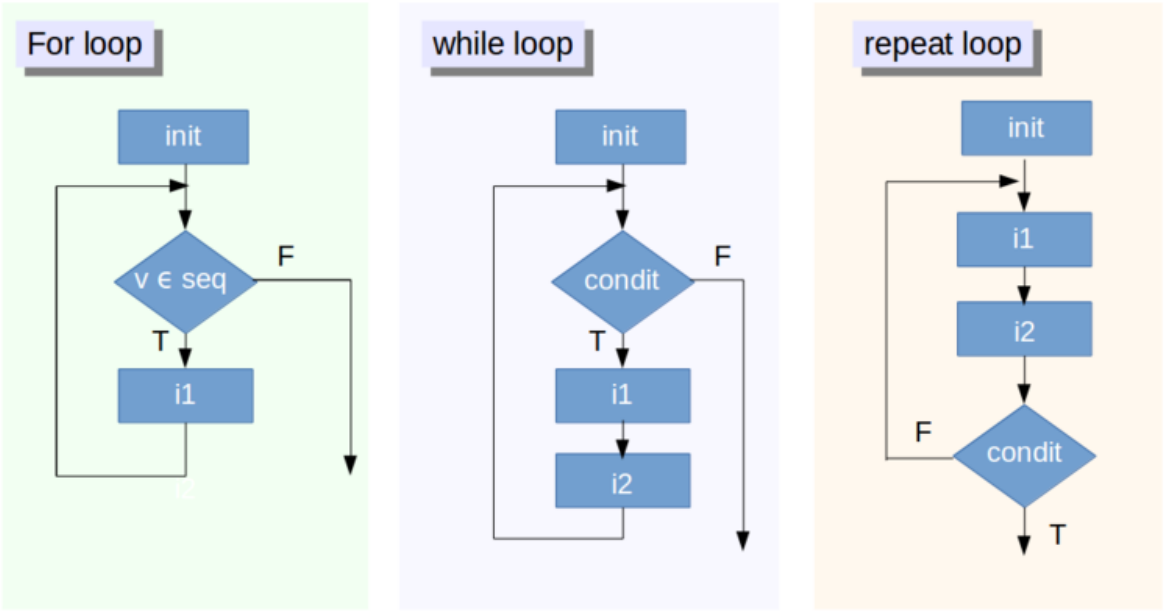


Figure 2: Source: datacamp.com

For-loops

```
# Let's print the numbers 1 to 6 one by one.
print(1)
## [1] 1
print(2)
## [1] 2
print(3)
## [1] 3
print(4)
## [1] 4
print(5)
## [1] 5
```

```
print(6)
## [1] 6
```

For-loops

For-loops allow us to automate this!

For each element of `1:6`, print the element:

```
for (i in 1:6){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

For-loops

You can use any variable name, `i` is a convention for counting/index.

```
for (some_var_name in 1:6){
  print(some_var_name)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

Subsetting consists of for-loops and if-else statements

For each element i, keep the value of the element i of example_vector ***if*** the element i of the condition (`example_vector>3`) is true.

```
example_vector = c(1,2,3,4,5,6,7,8,9)
example_vector>3
```

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

```
example_vector[example_vector>3]
```

```
## [1] 4 5 6 7 8 9
```

For-loops

Often you don't want to iterate over a range, but over an object

```
for (element in c("Amsterdam","Rotterdam","Eindhoven")){
  if (element == "Amsterdam"){
    print(paste(element,"Terrible football team.", sep=": "))
  } else {
    print(paste(element,"Not the prettiest city, but at least their football team is okay.", sep=": "))
```

```
  }
}
```

```
## [1] "Amsterdam: Terrible football team."
## [1] "Rotterdam: Not the prettiest city, but at least their football team is okay."
## [1] "Eindhoven: Not the prettiest city, but at least their football team is okay."
```

For-loops

Something a bit more useful

```
df <- data.frame("V1" = rnorm(5),
                 "V2" = rnorm(5, mean = 5, sd = 2),
                 "V3" = rnorm(5, mean = 6, sd = 1))

head(df)
```

```
##            V1       V2       V3
## 1 -1.59177940 4.316317 7.368464
## 2  0.76324704 6.760209 6.956894
## 3 -0.22899353 4.715755 5.895623
## 4 -0.04584141 2.125445 5.913738
## 5  0.90252283 4.809158 5.467448
```

```
for (col in names(df)) {
  print(col)
}
```

```
## [1] "V1"
## [1] "V2"
## [1] "V3"
```

For-loops

Doing an operation on each column

```
df <- data.frame("V1" = rnorm(5),
                 "V2" = rnorm(5, mean = 5, sd = 2),
                 "V3" = rnorm(5, mean = 6, sd = 1))

for (col in names(df)) {
  print(paste("Mean of ", col, ": ", mean(df[, col]), sep=""))
}
```

```
## [1] "Mean of V1: -0.264995603020639"
## [1] "Mean of V2: 3.64834854552737"
## [1] "Mean of V3: 6.15423122717128"
```

For-loops

Doing an operation on each row

```
for (row in 1:nrow(df)) {
  row_values = df[row, ]

  print(paste("Row ", row, ": ", sum(row_values>5), " values over 2", sep=""))
}
```

```
## [1] "Row 1: 1 values over 2"
## [1] "Row 2: 0 values over 2"
## [1] "Row 3: 1 values over 2"
## [1] "Row 4: 1 values over 2"
## [1] "Row 5: 2 values over 2"
```

While loops

Do something forever until a condition is (not) met

```
i  = 0
while (i < 10) {
  i =  i + 1
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

More info on loops: https://www.datamentor.io/r-programming/break-next/

The `apply()` family

`apply()`

The `apply` family is a group of very useful functions that allow you to easily execute a function of your choice over a list of objects, such as a `list`, a `data.frame`, or `matrix`.

We will look at three examples:

- `apply`
- `sapply`
- `lapply`

`apply()`

`apply` is used for matrices/dataframes. It applies a function to each *row* or *column*. It returns a vector or a matrix.

```
head(df, 1)
##          V1       V2       V3
## 1 0.4826716 4.034959 5.709234
```

Apply it by row (MARGIN = 1):

```
apply(df, MARGIN = 1, mean)
## [1] 3.408955 3.142947 4.236908 1.738974 3.368189
```

Apply it by column (MARGIN = 2):

```
apply(df, MARGIN = 2, mean) #Identical to colMeans(df)
##         V1         V2         V3
## -0.2649956  3.6483485  6.1542312
```

sapply()

`sapply()` is used on `list`-objects. It returns a vector or a matrix.

```
my.list <- list(A = c(4, 2, 1:3), B = "Hello.", C = TRUE)
sapply(my.list, class)
```

```
##           A           B           C
##   "numeric" "character"   "logical"
```

```
my.list <- list(A = c(4, 2, 1:3), B = c("hello","Hello.","Aa","aa"), C = c(FALSE,TRUE))
sapply(my.list, range)
```

```
##      A   B        C
## [1,] "1" "aa"     "0"
## [2,] "4" "Hello." "1"
```

Why is each element a character string?

sapply()

Any `data.frame` is also a `list`, where each column is one `list`-element.

This means we can use `sapply` on data frames as well, which is often useful.

```
sapply(df, mean)
```

```
##         V1         V2         V3
## -0.2649956  3.6483485  6.1542312
```

lapply()

`lapply()` is *exactly* the same as `sapply()`, but it returns a list instead of a vector.

```
lapply(df, class)
```

```
## $V1
## [1] "numeric"
##
## $V2
## [1] "numeric"
##
## $V3
## [1] "numeric"
```

Writing your own functions

What are functions?

Functions are reusable pieces of code that

1. take some standard input (e.g. a vector of numbers)
2. do some computation (e.g. calculate the mean)
3. return some standard output (e.g. one number with the mean)

We have been using a lot of functions: code of the form `something()` is usually a function.

```
mean(1:6)
```

```
## [1] 3.5
```

Our own function

We can make our own functions as follows:

```
squared <- function (x){
  x.square <- x * x
  return(x.square)
}

squared(4)
```

```
## [1] 16
```

`x`, the input, is called the (formal) *argument* of the function. `x.square` is called the *return value*.

Our own function

If there is no `return()`, the last line is automatically returned, so we can also just write:

I do not recommend this.

```
squared <- function(x){
  x * x
}

squared(-2)
```

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

Our own function

We can also combine this with `apply()`

```
df
```

```
##           V1       V2       V3
## 1  0.4826716 4.034959 5.709234
## 2  0.6331722 4.127596 4.668073
## 3  0.4748632 3.950741 8.285120
## 4 -2.0082158 1.110719 6.114420
## 5 -0.9074691 5.017728 5.994309
```

```
apply(df, 2, squared)
```

```
##             V1        V2       V3
## [1,] 0.2329718 16.280896 32.59536
## [2,] 0.4009070 17.037048 21.79090
## [3,] 0.2254950 15.608353 68.64322
## [4,] 4.0329308  1.233696 37.38613
## [5,] 0.8235002 25.177597 35.93174
```

Default options in functions

- Default options for some arguments are provided in many functions.

- They allow us to provide an additional option, but if no choice is provided, we can choose for the user of the function.

```
is_contained <- function(str_1, str_2, print_input = TRUE){
  if (print_input){
    cat("Testing if", str_1, "contained in", str_2, "\n")
  }
  return(str_1 %in% str_2)
}
```

```
is_contained("R", "rstudio")
```

```
is_contained("R", "rstudio")
## Testing if R contained in rstudio
## [1] FALSE
is_contained("R", "rstudio", print_input = TRUE)
## Testing if R contained in rstudio
## [1] FALSE
is_contained("R", "rstudio", print_input = FALSE)
## [1] FALSE
```

Troubleshooting

- Your first self-written for-loop, or function, will probably not work.
- Don't panic! Just go line-by-line, keeping track of what is currently inside each variable.
- Stackoverflow is your friend.

Scoping rules in R

Scoping rules in R: Global environemnt (workspace)

When you write the name of a variable, R needs to find the value.

In the interactive computation (outside of functions, e.g., your console), this happens in the following order: - First, search the global environment (i.e., your workspace) - If it cannot be found, search each of the loaded packages

```
search()
```

```
##  [1] ".GlobalEnv"        "package:forcats"   "package:stringr"
##  [4] "package:dplyr"     "package:purrr"     "package:readr"
##  [7] "package:tidyr"     "package:tibble"    "package:ggplot2"
## [10] "package:tidyverse" "package:stats"     "package:graphics"
## [13] "package:grDevices" "package:utils"     "package:datasets"
## [16] "package:methods"   "Autoloads"         "package:base"
```

The order of packages is important.

Scoping rules in R: Functions

Inside a function, this happens in the following order: - First, search within the function. - If it cannot be found, search in the global environment (i.e., your workspace) - If it cannot be found, search each of the loaded packages

```
y <- 3
```

```
test_t <- function() {
  print(y)
}

test_t()
```

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

```
y <- 3

test_t <- function() {
  y <- 3
  print(y)
}

test_t()
```

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

Scoping rules in R: Functions

What happens inside a function, stays within a function (unless you specify differently)

```
y <- 3

test_t <- function() {
  y <- 2
  print(y)
}

test_t()
```

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

```
y
```

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

Scoping rules in R: Packages

Packages are neatly contained/isolated, so they are not affected by your code.

They do so through namespaces. - Allow the package developer to hide functions and data. - Objects in the global environment that match objects in the function's namespace are ignored when running functions from packages (prevent clashes) - Functions are executed within the namespace of the package and have access to the global environment - They provide a way to refer to an object, with the double colon `::`

```
dplyr::n_distinct(c(1,2,3,4,2))
```

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

Practical