MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2018

Prof. Dr. Luis Carlos de Castro Santos

lsantos@ime.usp.br/lccs13@yahoo.com

Introduction to the R Language

Control Structures

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

Learn R Programming

The Definitive Guide

R is a programming language and environment commonly used in statistical computing, data analytics and scientific research.

It is one of the most popular languages used by statisticians, data analysts, researchers and marketers to retrieve, clean, analyze, visualize and present data.

Due to its expressive syntax and easy-to-use interface, it has grown in popularity in recent years.

TABLE OF CONTENTS

https://www.datamentor.io/r-programming/

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

R if statement

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

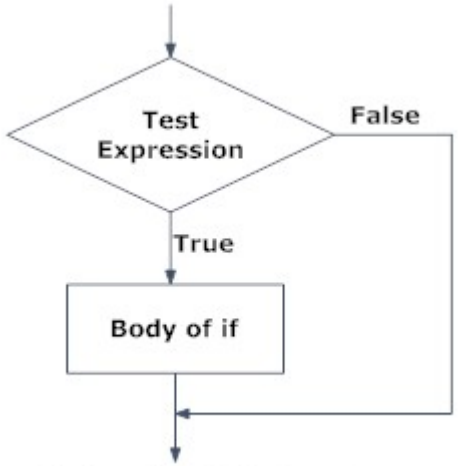


Fig: Operation of if statement

```
x <- 5
if(x > 0){
print("Positive number")
}
```

Output

```
[1] "Positive number"
```

if...else statement

```
if (test_expression) {
statement1
} else {
statement2
}
```

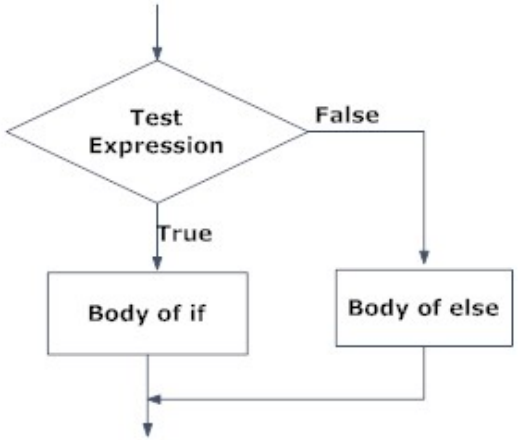


Fig: Operation of if...else statement

```
x <- -5
if(x > 0){
print("Non-negative number")
} else {
print("Negative number")
}
```

Output

```
[1] "Negative number"
```

Control Structures: if

```
if(<condition>) {
        ## do something
} else {
        ## do something else
}
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

if

This is a valid if/else structure.

```
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

So is this one.

```
y <- if(x > 3) {
        10
} else {
        0
}
```

if

Of course, the else clause is not necessary.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

if...else Ladder

```
if ( test_expression1) {
statement1
} else if ( test_expression2) {
statement2
} else if ( test_expression3) {
statement3
} else {
statement4
}
```

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

Output

```
[1] "Zero"
```

for

`for` loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

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

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

Syntax of for loop

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

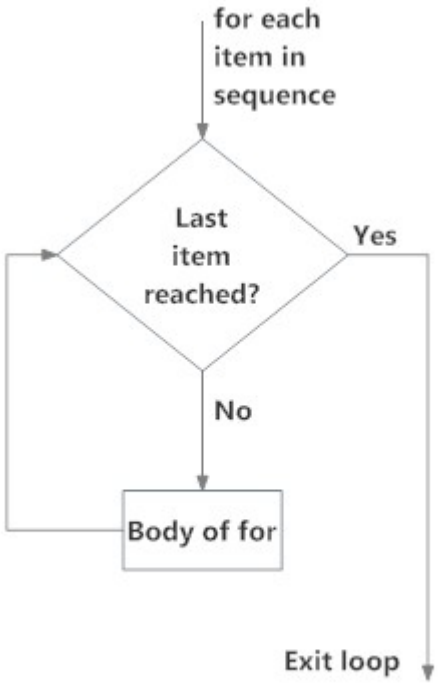


Fig: operation of for loop

```
x <- c(2,5,3,9,8,11,6)
count <- 0
for (val in x) {
if(val %% 2 == 0)   count = count+1
}
print(count)
```

Output

```
[1] 3
```

for

These three loops have the same behavior.

```
x <- c("a", "b", "c", "d")

for(i in 1:4){
        print(x[i])
}

for(i in seq_along(x)) {
        print(x[i])
}

for(letter in x) {
        print(letter)
}

for(i in 1:4) print(x[i])
```

sequencia numérica explicita

sequencia dos elementos de um vetor

sequencia das classes dos elementos de um vetor

Nested for loops

`for` loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

Qual o resultado da aplicação desse comando ?

```
> x <- matrix(1:6, 2, 3)
> for(i in seq_len(nrow(x))){
+   for(j in seq_len(ncol(x))){
+     print(x[i, j])
+   }
+ }
[1] 1
[1] 3
[1] 5
[1] 2
[1] 4
[1] 6
```

Example: Find the factorial of a number

```
# take input from the user
num = as.integer(readline(prompt="Enter a number: "))
factorial = 1
# check is the number is negative, positive or zero
if(num < 0) {
print("Sorry, factorial does not exist for negative numbers")
} else if(num == 0) {
print("The factorial of 0 is 1")
} else {
for(i in 1:num) {
factorial = factorial * i
}
print(paste("The factorial of", num ,"is",factorial))
}
```

Output

```
Enter a number: 8
[1] "The factorial of 8 is 40320"
```

while

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
        print(count)
        count <- count + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

R while Loop

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

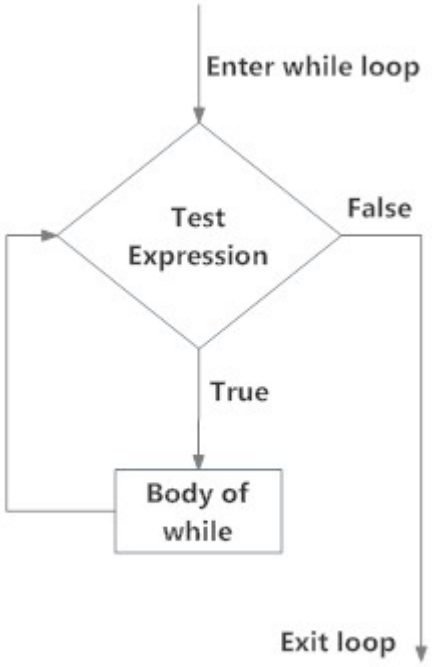


Fig: operation of while loop

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

Output

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

while

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)

        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
```

Conditions are always evaluated from left to right.

```
> source('C:/Users/User/Dropbox/USP/2019/MAP2112/Notas de Aula/random_walk.R')
[1] 5
[1] 6
[1] 5
[1] 6
[1] 5
[1] 6
[1] 7
[1] 6
[1] 7
[1] 6
[1] 7
[1] 6
[1] 5
[1] 6
[1] 7
[1] 6
[1] 7
[1] 8
[1] 7
[1] 6
[1] 7
[1] 6
[1] 5
[1] 4
[1] 5
[1] 4
[1] 5
[1] 6
[1] 7
[1] 6
[1] 5
[1] 4
[1] 3
```

```
> source('C:/Users/User/Dropbox/USP/2019/MAP2112/Notas de Aula/random_walk.R')
[1] 5
[1] 4
[1] 3
[1] 4
[1] 5
[1] 4
[1] 3
```

Duas execuções distintas geram caminhos diferentes devido a aleatoriadade

repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a `repeat` loop is to call `break`.

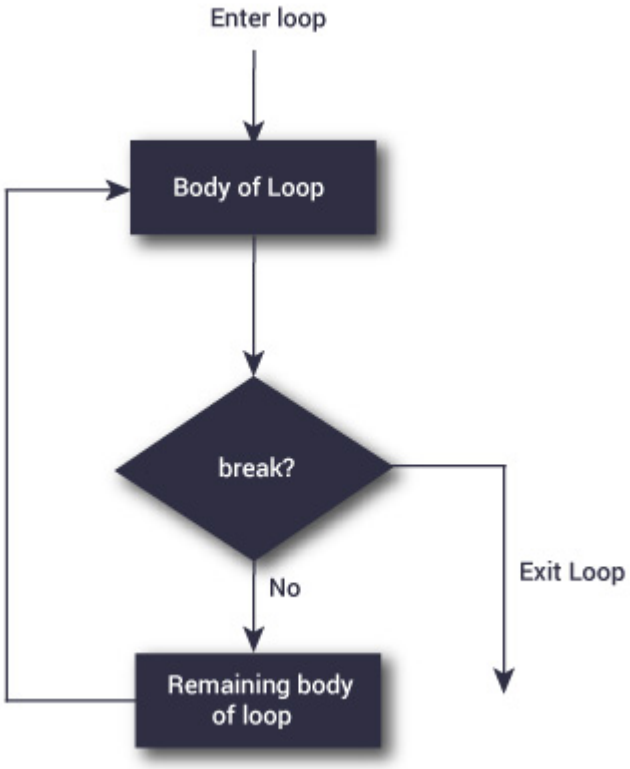
```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()

        if(abs(x1 - x0) < tol) {
                break
        } else {
                x0 <- x1
        }
}
```

repeat

The loop in the previous slide is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

R repeat loop


```
repeat {
statement
}
```




```
x <- 1
repeat {
print(x)
x = x+1
if (x == 6){
break
}
}
```

Output

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

break statement

A `break` statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.

In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

```
x <- 1:5
for (val in x) {
if (val == 3){
break
}
print(val)
}
```

Output

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

next, return

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`return` signals that a function should exit and return a given value

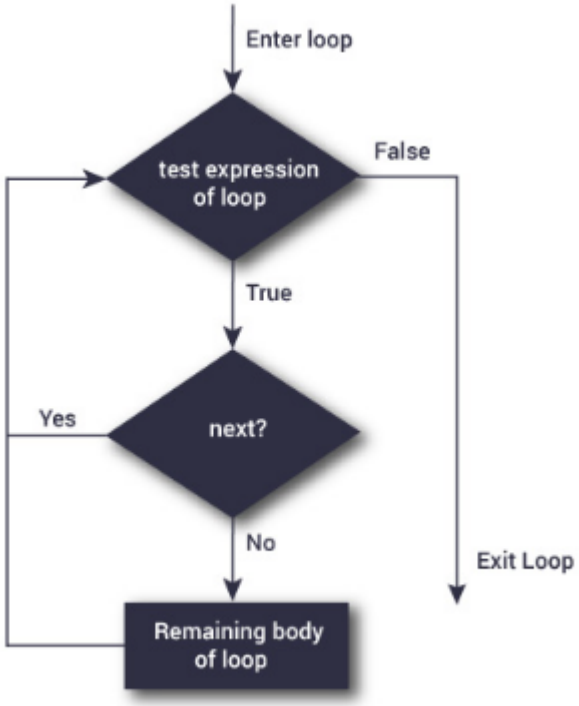
next statement

A next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering `next`, the R parser skips further evaluation and starts next iteration of the loop.

The syntax of next statement is:

```
if (test_condition) {
next
}
```

Note: the next statement can also be used inside the `else` branch of `if...else` statement.




```
x <- 1:5
for (val in x) {
if (val == 3){
next
}
print(val)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
```

Example: Check Prime Number

```
# Program to check if the input number is prime or not
# take input from the user
num = as.integer(readline(prompt="Enter a number: "))
flag = 0
# prime numbers are greater than 1
if(num > 1) {
# check for factors
flag = 1
for(i in 2:(num-1)) {
if ((num %% i) == 0) {
flag = 0
break
}
}
}
if(num == 2)      flag = 1
if(flag == 1) {
print(paste(num,"is a prime number"))
} else {
print(paste(num,"is not a prime number"))
}
```

Output 1

```
Enter a number: 25
[1] "25 is not a prime number"
```

Output 2

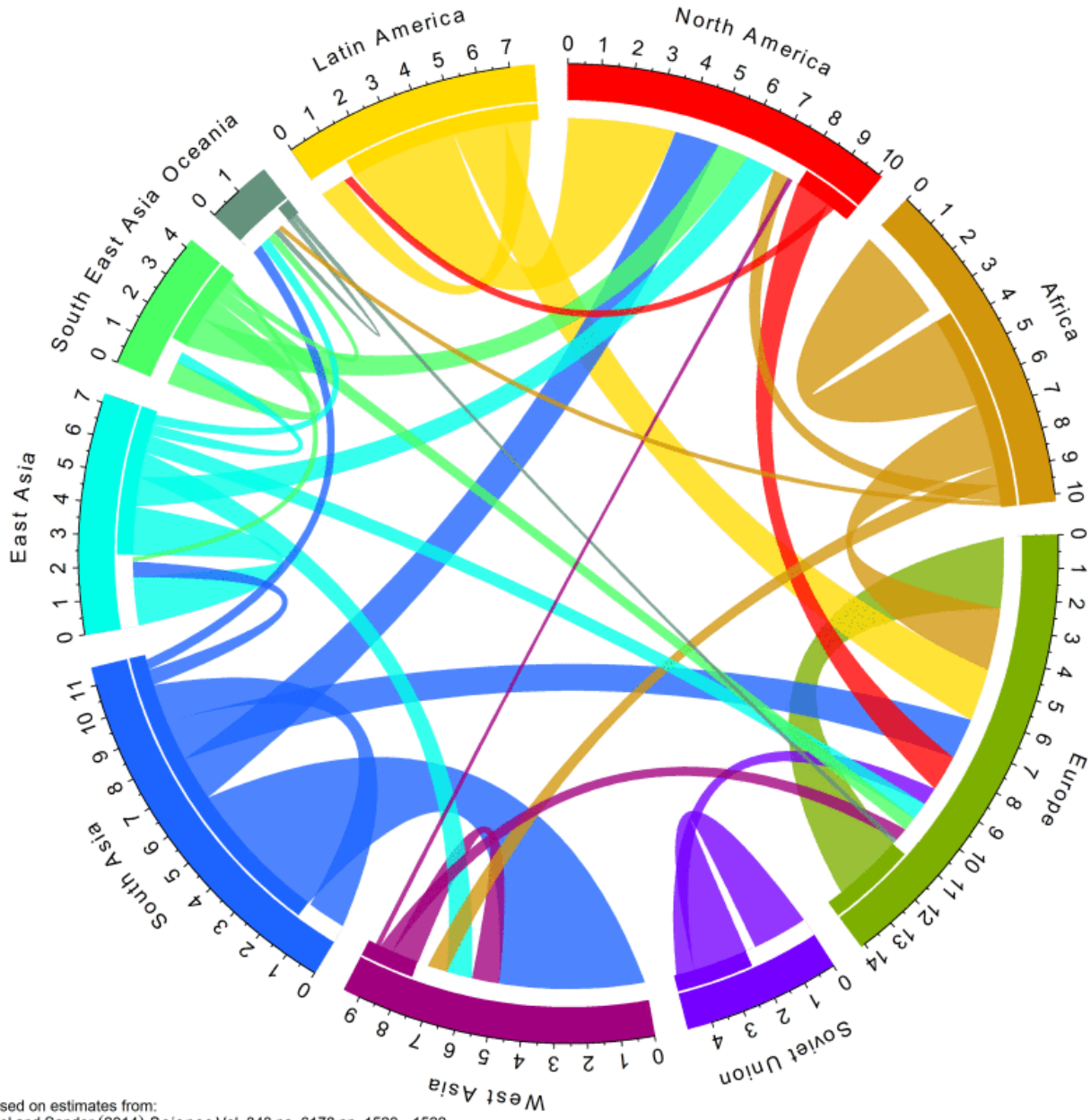
```
Enter a number: 19
[1] "19 is a prime number"
```

Modifique esse programa e produza outro que encontre todos os primos entre 1 e 1000

Control Structures

Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentiond here are primarily useful for writing programs; for command-line interactive work, the *apply functions are more useful.



Based on estimates from:
Abel and Sander (2014) *Science* Vol. 343 no. 6178 pp. 1520 – 1522