

Programação Estatística

Introdução ao R - Estruturas de controlo

Rachid Muleia, PhD in Statistics

Universidade Eduardo Mondlane
Faculdade de Ciências
Departamento de Matemática e Informática

2023-04-06

Estruturas de controle

- Programar é mais do que colocar os comandos que aprendeu num ficheiro R
- Um elemento chave em programação é que pode-se usar estruturas de controlo para controlar a execução de um programa
- Estruturas de controlo são usadas para tomar uma decisão após avaliar uma variável

Estruturas de controlo

As estruturas de controle podem ser divididas em três categorias

- Estruturas de controle condicional
- Estruturas de controle iterativas (de repetição)
- Estruturas de controle de “salto”

A linguagem R tem 08 estruturas de controle : declaração `if`, declaração `if-else`, `for loop`, `for loops aninhados`, `while loop`, `repeat` e `break`, declaração `next` e a declaração `return`

Estruturas de controlo

- `if, else` : usado para testar uma condição e executar uma instrução com base na condição
- `for`: usado para executar um ciclo (loop) para um número fixo de iterações
- `while`: usado para executar um ciclo (loop) em quanto a condição for verdadeira
- `repeat`: usado para executar um loop indefinidamente até encontrar um `break`
- `break`: interrompe uma execução em um ciclo
- `next`: usado para saltar uma iteração em um ciclo
- `return`: usado para sair de uma função

Declaração if e else

if e else são usadas para executar uma instrução com base em uma condição, onde a instrução é executada quando a condição é satisfeita e uma instrução alternativa quando a condição não for satisfeita

```
if(<condição>) {  
## faça alguma coisa  
} else {  
## caso contrario faça outra coisa  
}
```

Declaração if e else

O exemplo a seguir mostra o funcionamento de uma estrutura de controlo if.

```
x <- 5
y <- 5
if(x<5) {
  y <- 0
}
y
```

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

Condição é avaliada como falsa → a instrução dentro de parênteses não é executada

Declaração if e else

A linguagem R permite usar uma série de if de forma aninhada

```
x <- 1
if(x > 0) {
  cat("x é maior que zero")
} else if(x < 0) {
  cat("x é meno que zeo")
} else {
  cat("x dever ser zero!")
  cat("\n")
}
```

```
## x é maior que zero
```

`&&(E)` e `|| (OU)` numa condição

Operadores comumente usados em condições `&&` (E) e `||` (OU)

```
if(x>0 && x<1) {  
  y<-x^2  
} else {  
  y<-x^4  
}  
y
```

&&(E) e || (OU) numa condição

```
> x <- c(1>2, 2<3, 3==4)
> x
## [1] FALSE TRUE FALSE
> y <- c(1<2, 2<3, 3!=4)
> y
## [1] TRUE TRUE TRUE
> x&& y
## [1] FALSE
```

&&(E) e || (OU)

```
> x
[1] FALSE TRUE FALSE
> y
[1] TRUE TRUE TRUE
> x&& y
[1] FALSE
> x&y
```

$\&\&(E) e \mid \mid (OU)$

```
> x
[1] FALSE TRUE FALSE
> y
[1] TRUE TRUE TRUE
> x&& y
[1] FALSE
> x&y
```

```
> x
[1] FALSE TRUE FALSE
> y
[1] TRUE TRUE TRUE
> x&& y
[1] FALSE
> x&y
[1] FALSE TRUE FALSE
```

$\&\&(E)$ e $||$ (OU)

Compare as seguintes expressões

```
> 1<2 || 2>3 && 1>2
```

```
[1] TRUE
```

```
> (1<2 || 2>3) && 1>2
```

```
[1] FALSE
```

```
> 1<2 || (2>3 && 1>2)
```

```
[1] TRUE
```

Porque temos resultados diferentes?

`&&(E)` e `|| (OU)`

No R, os operadores pertencem a diferentes grupos de precedência. `&&` tem precedência maior que `||`, portanto `&&` é avaliado primeiro.

- Mais detalhes sobre precedência de operadores pode ser encontrada na documentação do R (`help(Syntax)`)
- Numa expressão operadores de igual precedência são avaliados da esquerda para a direita
- Se você não tiver certeza sobre qual operador é avaliado primeiro, melhor especificar explicitamente a prioridade usando `()`

Declaração if e else

Não é uma boa prática usar as instruções if e else de forma isolada. Estas são usadas em funções ou ciclos.

```
IsNegative = function(value) {  
  is_pos = FALSE  
  if(value < 0) {  
    is_pos = TRUE  
  }  
  return(is_pos)  
}
```

```
IsNegative(1)  
## [1] FALSE  
IsNegative(-5.6)  
## [1] TRUE
```

Ciclos - for loop

```
for(var in seq) {  
  expr  
}
```

- For loop geralmente é usado para executar um conjunto de instruções um número fixo de vezes
- O loop for é sempre usado em combinação com um objecto iterável, como uma lista, vector, etc

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

Ciclos - for loop

Os ciclos abaixo tem o mesmo comportamento

```
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])
```

Ciclos - for loop

Suponha que deseja calcular $\sum_{i=1}^{10} i^2$

```
x = 0
for(i in 1:10) {
  x = x+i^2
}
x
## [1] 385
```

Ciclos -for loop

```
data("iris")
data_iris <- cbind(iris, rep(iris[,1:dim(iris)[2]],8))
head(data_iris,n=3)
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length
## 1 5.1 3.5 1.4 0.2 setosa 5.1
## 2 4.9 3.0 1.4 0.2 setosa 4.9
## 3 4.7 3.2 1.3 0.2 setosa 4.7
## Sepal.Width Petal.Length Petal.Width Species Sepal.Length Sepal.Width
## 1 3.5 1.4 0.2 setosa 5.1 3.5
## 2 3.0 1.4 0.2 setosa 4.9 3.0
## 3 3.2 1.3 0.2 setosa 4.7 3.2
## Petal.Length Petal.Width Species Sepal.Length Sepal.Width Petal.Length
## 1 1.4 0.2 setosa 5.1 3.5 1.4
## 2 1.4 0.2 setosa 4.9 3.0 1.4
## 3 1.3 0.2 setosa 4.7 3.2 1.3
## Petal.Width Species Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 0.2 setosa 5.1 3.5 1.4 0.2 setosa
## 2 0.2 setosa 4.9 3.0 1.4 0.2 setosa
## 3 0.2 setosa 4.7 3.2 1.3 0.2 setosa
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length
## 1 5.1 3.5 1.4 0.2 setosa 5.1
## 2 4.9 3.0 1.4 0.2 setosa 4.9
## 3 4.7 3.2 1.3 0.2 setosa 4.7
## Sepal.Width Petal.Length Petal.Width Species Sepal.Length Sepal.Width
## 1 3.5 1.4 0.2 setosa 5.1 3.5
## 2 3.0 1.4 0.2 setosa 4.9 3.0
## 3 3.2 1.3 0.2 setosa 4.7 3.2
## Petal.Length Petal.Width Species Sepal.Length Sepal.Width Petal.Length
## 1 1.4 0.2 setosa 5.1 3.5 1.4
## 2 1.4 0.2 setosa 4.9 3.0 1.4
## 3 1.3 0.2 setosa 4.7 3.2 1.3
## Petal.Width Species
## 1 0.2 setosa
```

Ciclos -for loop

Calcule a média e o desvio padrão para todas variáveis numéricas na data frame

```
data_iris <- data_iris[, -which(names(data_iris)=='Species')]
media_vec<-c()
desvio_vec<-c()
for(i in 1:dim(data_iris)[2]){
  media_vec[i] <- mean(data_iris[,i],na.rm=TRUE)
  desvio_vec[i] <- sd(data_iris[,i],na.rm=TRUE)
}
```

media_vec

```
## [1] 5.843333 3.057333 3.758000 1.199333 5.843333 3.057333 3.758000 1.199333
## [9] 5.843333 3.057333 3.758000 1.199333 5.843333 3.057333 3.758000 1.199333
## [17] 5.843333 3.057333 3.758000 1.199333 5.843333 3.057333 3.758000 1.199333
## [25] 5.843333 3.057333 3.758000 1.199333 5.843333 3.057333 3.758000 1.199333
## [33] 5.843333 3.057333 3.758000 1.199333
```

desvio_vec

```
## [1] 0.8280661 0.4358663 1.7652982 0.7622377 0.8280661 0.4358663 1.7652982
## [8] 0.7622377 0.8280661 0.4358663 1.7652982 0.7622377 0.8280661 0.4358663
## [15] 1.7652982 0.7622377 0.8280661 0.4358663 1.7652982 0.7622377 0.8280661
## [22] 0.4358663 1.7652982 0.7622377 0.8280661 0.4358663 1.7652982 0.7622377
## [29] 0.8280661 0.4358663 1.7652982 0.7622377 0.8280661 0.4358663 1.7652982
## [36] 0.7622377
```

Funções apply

- As funções apply são um conjunto de funções que permitem executar uma instrução repetidas vezes
- Estas funções equiparam-se aos for-loop. Contudo, são rápidas em relação aos for loops tradicionais
- Não precisam de linhas extensas de código
- É sempre recomendado a usar as funções internas do R, visto que são mais optimizadas

lapply

- A função `lapply` aplica uma função sobre uma lista ou vector e devolve uma lista do mesmo comprimento

```
lapply(X, FUN, ...)
```

```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

```
replicate(n, expr, simplify = "array")
```

```
simplify2array(x, higher = TRUE, except = c(0L, 1L))
```

lapply

Visitemos a data frame `data_iris` para calcular a média

```
media_vec1 <- lapply(data_iris, FUN=mean, na.rm=TRUE)
```

```
media_vec1
```

```
## $Sepal.Length
```

```
## [1] 5.843333
```

```
##
```

```
## $Sepal.Width
```

```
## [1] 3.057333
```

```
##
```

```
## $Petal.Length
```

```
## [1] 3.758
```

```
##
```

```
## $Petal.Width
```

```
## [1] 1.199333
```

```
##
```

```
## $Sepal.Length.1
```

```
## [1] 5.843333
```

```
##
```

```
## $Sepal.Width.1
```

```
## [1] 3.057333
```

```
##
```

```
## $Petal.Length.1
```

```
## [1] 3.758
```

```
##
```

```
## $Petal.Width.1
```

```
## [1] 1.199333
```

```
##
```

```
## $Sepal.Length.2
```

```
## [1] 5.843333
```

```
##
```

```
## $Sepal.Width.2
```

lapply

Talvez seja ideal usar a função `sapply`. Tende a produzir resultados mais “elegantes”

```
media_vec1 <- sapply(data_iris, FUN=mean, na.rm=TRUE, USE.NAMES = FALSE)
media_vec1
## Sepal.Length Sepal.Width Petal.Length Petal.Width Sepal.Length.1
## 5.843333 3.057333 3.758000 1.199333 5.843333
## Sepal.Width.1 Petal.Length.1 Petal.Width.1 Sepal.Length.2 Sepal.Width.2
## 3.057333 3.758000 1.199333 5.843333 3.057333
## Petal.Length.2 Petal.Width.2 Sepal.Length.3 Sepal.Width.3 Petal.Length.3
## 3.758000 1.199333 5.843333 3.057333 3.758000
## Petal.Width.3 Sepal.Length.4 Sepal.Width.4 Petal.Length.4 Petal.Width.4
## 1.199333 5.843333 3.057333 3.758000 1.199333
## Sepal.Length.5 Sepal.Width.5 Petal.Length.5 Petal.Width.5 Sepal.Length.6
## 5.843333 3.057333 3.758000 1.199333 5.843333
## Sepal.Width.6 Petal.Length.6 Petal.Width.6 Sepal.Length.7 Sepal.Width.7
## 3.057333 3.758000 1.199333 5.843333 3.057333
## Petal.Length.7 Petal.Width.7 Sepal.Length.8 Sepal.Width.8 Petal.Length.8
## 3.758000 1.199333 5.843333 3.057333 3.758000
## Petal.Width.8
## 1.199333
```

lapply vs for loop

Geralmente as funções apply são mais rápidas que os loops

```
media_vec<-c()
data_iris<-cbind(data_iris, rep(data_iris[,1:dim(data_iris)[2]],100))
dim(data_iris)
## [1] 150 3636
system.time(

for(i in 1:dim(data_iris)[2]){
  media_vec[i] <- mean(data_iris[,i],na.rm=TRUE)
}

)
##   user  system elapsed
##  0.05   0.00   0.07

system.time(media_vec1 <- sapply(data_iris, FUN=mean, na.rm=TRUE, USE.NAMES = FALSE))
##   user  system elapsed
##  0.02   0.00   0.03
```

lapply e funções anónimas

As funções apply podem receber **funções anónimas**

Funções anónimas

Uma função anónima (também conhecida como expressão lambda) é uma definição de função que não está vinculada a um identificador. Ou seja, é uma função que é criada e utilizada, mas nunca atribuída a uma variável.

lapply e funções anónimas

Calcule a média de todas as observações com valores maiores que 5

```
nossa.funcao<-function(x) { mean(x[x > 5]) }  
  
media_maior<-sapply(data_iris, nossa.funcao)  
  
head(media_maior,10)  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Sepal.Length.1  
## 6.129661 NaN 5.688095 NaN 6.129661  
## Sepal.Width.1 Petal.Length.1 Petal.Width.1 Sepal.Length.2 Sepal.Width.2  
## NaN 5.688095 NaN 6.129661 NaN
```

Talvez a gente queira apenas usar esta função instantaneamente

```
nossa.funcao<-function(x) { mean(x[x > 5]) }  
  
media_maior1<-sapply(data_iris, function(x) { mean(x[x > 5]) })  
  
head(media_maior1,10)  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Sepal.Length.1  
## 6.129661 NaN 5.688095 NaN 6.129661  
## Sepal.Width.1 Petal.Length.1 Petal.Width.1 Sepal.Length.2 Sepal.Width.2  
## NaN 5.688095 NaN 6.129661 NaN
```

Loops aninhados (nested loops)

O R permite ter um cliço dentro do outro

```
x<-matrix(1:60,6,10)
for(i in seq_len(nrow(x))){
  for(j in seq_len(ncol(x))) {
    print(x[i,j])
  }
}
```

Pode-se também incluir uma instrução if-else dentro de um ciclo

While loop (ciclo while)

```
while(cond) {  
  expr  
}
```

O `while` loop avalia uma condição repetidamente. Se a condição for verdadeira, então a expressão no corpo do loop é executada. Caso contrário, o loop será encerrado. Por exemplo:

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

while loop: exemplo

simular um passeio aleatório

```
while(z>=3 && z<=10) {  
  print(z)  
  coin<-rbinom(1,1,0.5)  
  if(coin == 1) {  
    z<-z+1  
  } else {  
    z<-z-1  
  }  
}
```

repeat loop

```
repeat
{
    instrução

    if( condicaco )
    {
        break
    }
}
```

O ciclo reapt executa uma instrução até encontrar um `break`.

repeat loop: exemplo

```
# demonstração do ciclo repeat

val = 1

# using repeat loop
repeat
{
    # instrução(código)
    print(val)
    val = val + 1

    # condição de paragem
    if(val > 5)
    {
        # aplicacao do código break
        # para terminar o ciclo
        break
    }
}
```