Módulo 4 - Estruturas de controle e funções Loop no R

Marcus Suassuna Santos

26/10/2020

Estruturas de controle no R

Estruturas de controle, são ferramentas para inserir lógica, julgamento e inteligência aos códigos. Estruturas de controle comuns no R são:

- `if` / `else if` / `else`: testa condição e executa algo de acordo
- `ifelse`: função que retorna valores condicionados a um vetor teste
- `for`: executa um *loop* por um número definido de vezes
- `while`: executa um *loop* enquanto uma determinada condição é verdadeira
- Funções de *loops*. Família `apply`
- Outras: `repeat`, `break`, `next`.

Estruturas de controle no R

Antes de tudo, vamos ler uma base de dados pra utilizar nos testes que iremos fazer.

```
Cotas <- read_csv2("dados/cotas_T_14990000.txt", skip = 12) %>%
  filter(complete.cases(Maxima, Media, Minima)) %>%
  filter(NivelConsistencia == 1 & MediaDiaria == 1) %>%
  select(c("Data", "Maxima")) %>%
  mutate(Data = as.Date(Data, "%d/%m/%Y"),
         Normalizada = as.numeric(scale(Maxima))) %>%
  arrange(Data)

head(Cotas)
```

```
## # A tibble: 6 x 3
##   Data       Maxima Normalizada
##   <date>      <dbl>       <dbl>
## 1 1902-10-01   1970      -1.36
## 2 1902-11-01   1774      -1.95
## 3 1902-12-01   1909      -1.54
## 4 1903-01-01   2094      -0.986
## 5 1903-02-01   2172      -0.751
## 6 1903-03-01   2332      -0.269
```

`if` / `else` / `else if`

A estrutura `if` permite testar uma determinada condição. Em seu bloco, é executada uma ação, caso a condição seja verdadeira. A estrutura `else` define o que fazer caso a condição definida em `if` seja falsa.

```
x <- 6
print(x)
```

```
## [1] 6
```

```
if(x > 5) {
  print("Maior que 5")
}
```

```
## [1] "Maior que 5"
```

`if` / `else` / `else if`

A estrutura `if` permite testar uma determinada condição. Em seu bloco, é executada uma ação, caso a condição seja verdadeira. A estrutura `else` define o que fazer caso a condição definida em `if` seja falsa.

```
x <- 4
print(x)
```

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

```
if(x > 5) {
  print("Maior que 5")
}
```

`if` / `else` / `else if`

A estrutura `if` permite testar uma determinada condição. Em seu bloco, é executada uma ação, caso a condição seja verdadeira. A estrutura `else` define o que fazer caso a condição definida em `if` seja falsa.

```
x <- 4
print(x)
```

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

```
if(x > 5) {
  print("Maior que 5")
} else {
  print("Menor que ou igual a 5")
}
```

```
## [1] "Menor que ou igual a 5"
```

`if` / `else` / `else if`

A estrutura `if` permite testar uma determinada condição. Em seu bloco, é executada uma ação, caso a condição seja verdadeira. A estrutura `else` define o que fazer caso a condição definida em `if` seja falsa.

```
x <- 5
if(x > 5) {
  print("Maior que 5")
} else if(x < 5){
  print("Menor que 5")
} else {
  print("Igual a 5")
}
```

```
## [1] "Igual a 5"
```

`if` / `else` / `else if`

A estrutura `if` permite testar uma determinada condição. Em seu bloco, é executada uma ação, caso a condição seja verdadeira. A estrutura `else` define o que fazer caso a condição definida em `if` seja falsa.

```
x <- 9
if(x > 5) {
  print("Maior que 5")
} else if(x < 5){
  print("Menor que 5")
} else if(x > 7){
  print("Igual a 5")
} else {
  print("Igual a 5")
}
```

```
## [1] "Maior que 5"
```

ifelse

Retomando a tabela “Cotas”.

```
head(Cotas)
```

```
## # A tibble: 6 x 3
##   Data       Maxima Normalizada
##   <date>      <dbl>       <dbl>
## 1 1902-10-01   1970      -1.36
## 2 1902-11-01   1774      -1.95
## 3 1902-12-01   1909      -1.54
## 4 1903-01-01   2094      -0.986
## 5 1903-02-01   2172      -0.751
## 6 1903-03-01   2332      -0.269
```

Vamos supor que a cota de inundação do rio Negro em Manaus seja a cota de 2700 cm. Podemos usar uma função para criar uma variável categórica, que indique todos os meses que o rio inundou.

ifelse

Podemos usar a função `ifelse` pra isso.

```
Cotas$Categoria <- ifelse(Cotas$Maxima > 2700, "Inundação", "Normal")
tail(Cotas)
```

```
## # A tibble: 6 x 4
##   Data       Maxima Normalizada Categoria
##   <date>      <dbl>       <dbl> <chr>
## 1 2014-01-01   2334      -0.263 Normal
## 2 2014-02-01   2467       0.138 Normal
## 3 2014-03-01   2665       0.735 Normal
## 4 2014-04-01   2798       1.14  Inundação
## 5 2014-05-01   2934       1.55  Inundação
## 6 2014-06-01   2948       1.59  Inundação
```

```
Cotas$Categoria <- NULL
```

ifelse

Podemos usar a função `ifelse` pra isso, combinada com o pacote `dplyr`

```
Cotas %>%
  mutate(Categoria = ifelse(Maxima > 2700, "Inundação", "Normal")) %>%
  tail()
```

```
## # A tibble: 6 x 4
##   Data       Maxima Normalizada Categoria
##   <date>      <dbl>       <dbl> <chr>
## 1 2014-01-01   2334      -0.263 Normal
## 2 2014-02-01   2467       0.138 Normal
## 3 2014-03-01   2665       0.735 Normal
## 4 2014-04-01   2798       1.14  Inundação
## 5 2014-05-01   2934       1.55  Inundação
## 6 2014-06-01   2948       1.59  Inundação
```

for

Em minha experiência, pouquíssimas vezes, um *loop* do tipo `for` não foi suficiente para executar operações com estruturas de controle.

Um exemplo simples:

```
for (i in 1:5) {
  print(i)
}
```

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

for

Usando `i` como um índice de posição de um vetor:

```
a <- c("a", "b", "c", "d", "e")
for (i in 1:5) {
  print(a[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

for

Atentar que aqui foi utilizado o índice `i` mas poderia ser qualquer outro índice:

```
a <- c("a", "b", "c", "d", "e")
for (t in 1:5) {
  print(a[t])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

for

Usando a função `seq_along`:

```
a <- c("a", "b", "c", "d", "e")
seq_along(a)
```

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

```
for (i in seq_along(a)) {
  print(a[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

for

Não é necessário utilizar índice, embora muitas vezes ele permita um melhor controle do *loop*:

```
a <- c("a", "b", "c", "d", "e")
for (letra in a) {
  print(letra)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

for

As chaves são dispensáveis pra execução do código, o que permite uma forma compacta de redigir o *loop*:

```
a <- c("a", "b", "c", "d", "e")
for (i in seq_along(a)) print(a[i])
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

for

Exemplo usando os dados de `Cotas` de Manaus:

```
VatorMaximas <- array(NA, dim = nrow(tail(Cotas)))
for (i in 1:nrow(tail(Cotas))) {
  VatorMaximas[i] <- tail(Cotas)[i,]$Maxima
}
print(VatorMaximas)
```

```
## [1] 2334 2467 2665 2798 2934 2948
```

for

Nested loops ou *loops* concatenados, aninhados:

```
x <- matrix(NA, 3, 3)
for(i in seq_len(nrow(x))){
  for(j in seq_len(ncol(x))){
    x[i,j] <- i * j
  }
}
print(x)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

`while`

`while` *loops* começam testando uma condição, se for verdadeira ele executa o corpo do *loop*. Após a execução, ele testa novamente e executa novamente o código, até que a condição deixe de ser verdadeira.

```
contagem <- 0
while(contagem < 5){
  print(contagem)
  contagem <- contagem + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

`while`

O *loop* do tipo `while` se não executado apropriadamente, pode levar a *loops* infinitos. Usar com cuidado, é a recomendação de 100% dos instritures de cursos de programação que fiz até hoje.

while

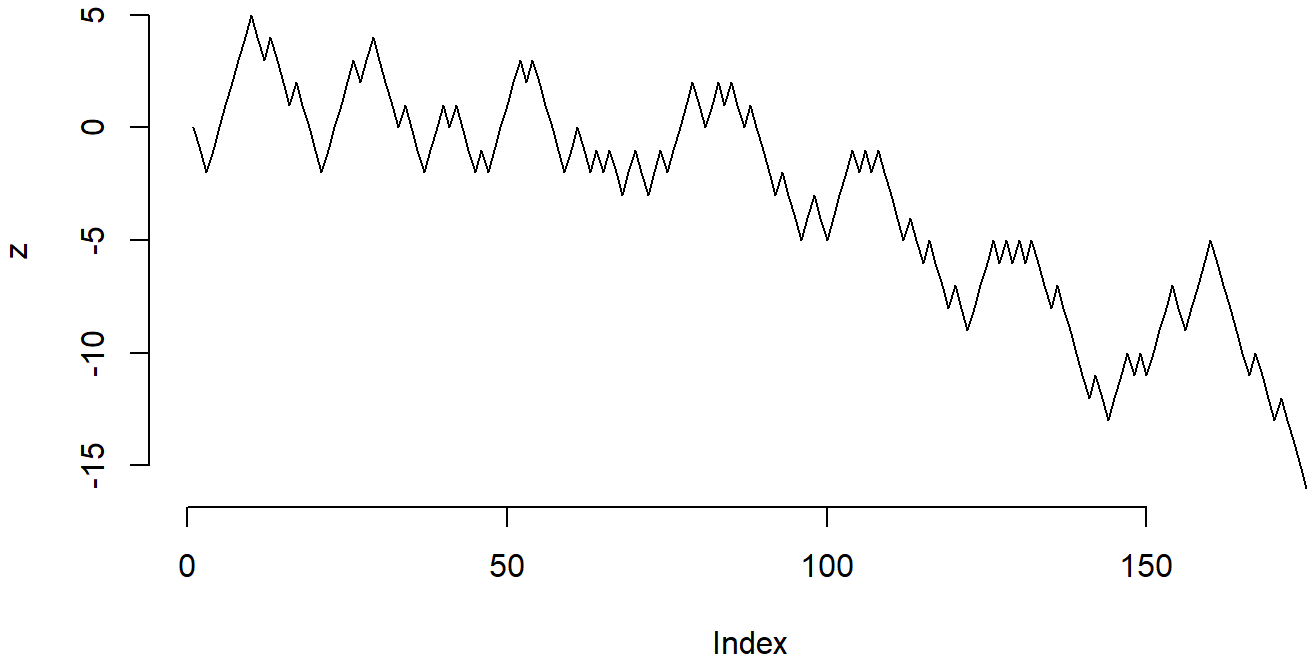
Exemplo passeio aleatório:

```
z <- 0
contagem <- 0

while(tail(z,1) >= -15 && tail(z,1) <= 15 && contagem < 200) {
  moeda <- rbinom(1, 1, 0.5)

  ## passeio aleatório
  if(moeda == 1) {
    z <- c(z, tail(z,1) + 1)
  } else {
    z <- c(z, tail(z,1) - 1)
  }

  contagem <- contagem + 1
}
```

while

Exemplo passeio aleatório:

```
plot(z, type = "l", bty = "n")
```



Família de funções `apply`

Funções úteis pra rodar *loops* em linhas de comando. Facilitam a programação em *loops* com uma estrutura muito compacta.

- `lapply()`: Percorre uma lista e avalia a função em cada elemento. Sempre retorna uma lista.
- `sapply()`: Igual a `lapply`, mas o resultado é simplificado ao final
- `apply()`: Aplica uma função nas linhas ou colunas de um `array`
- `tapply()`: Aplica uma função em partes de um vetor
- `split`: quebra uma tabela em função de um de seus elementos

lapply

```
x <- list(a = 1:5,
          b = rnorm(10),
          c = c("Inundou", "Normal"),
          d = c(TRUE, FALSE, FALSE, TRUE, FALSE))
print(x)
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
##  [1]  0.8877277  2.2827280 -0.4873206 -0.7011213 -0.1292795  0.2041234
##  [7]  1.3349569  1.1192125 -3.7125430 -1.7242690
##
## $c
## [1] "Inundou" "Normal"
##
## $d
## [1]  TRUE FALSE FALSE  TRUE FALSE
```

lapply

```
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.09257851
##
## $c
## [1] NA
##
## $d
## [1] 0.4
```

sapply

```
sapply(x, mean)
```

```
##           a           b           c           d
##  3.00000000 -0.09257851          NA  0.40000000
```

OBSERVAÇÃO!

As funções podem receber argumentos adicionais, que pertencem não às funções `lapply` ou `sapply`, mas sim à função que elas chamam, no exemplo acima, a função `mean`. Por exemplo:

```
sapply(x, mean, na.rm = TRUE)
```

```
##           a           b           c           d
##  3.00000000 -0.09257851          NA  0.40000000
```

split

Vamos retornar aos dados de Cotas para mostrar aplicações das funções `split` e `tapply`

```
Cotas %>%
  head()
```

```
## # A tibble: 6 x 3
##   Data       Maxima Normalizada
##   <date>      <dbl>       <dbl>
## 1 1902-10-01   1970      -1.36
## 2 1902-11-01   1774      -1.95
## 3 1902-12-01   1909      -1.54
## 4 1903-01-01   2094      -0.986
## 5 1903-02-01   2172      -0.751
## 6 1903-03-01   2332      -0.269
```

split

Vamos retornar aos dados de Cotas para mostrar aplicações das funções `split` e `tapply`

```
library(lubridate)
Cotas <- Cotas %>%
  mutate(Ano = year(Data))
head(Cotas)
```

```
## # A tibble: 6 x 4
##   Data       Maxima Normalizada   Ano
##   <date>      <dbl>       <dbl> <dbl>
## 1 1902-10-01   1970      -1.36   1902
## 2 1902-11-01   1774      -1.95   1902
## 3 1902-12-01   1909      -1.54   1902
## 4 1903-01-01   2094      -0.986  1903
## 5 1903-02-01   2172      -0.751  1903
## 6 1903-03-01   2332      -0.269  1903
```

split

Vamos retornar aos dados de Cotas para mostrar aplicações das funções `split` e `tapply`

```
s <- split(Cotas, Cotas$Ano)
print(s)
```

```
## $`1902`
## # A tibble: 3 x 4
##   Data       Maxima Normalizada   Ano
##   <date>      <dbl>       <dbl> <dbl>
## 1 1902-10-01   1970       -1.36  1902
## 2 1902-11-01   1774       -1.95  1902
## 3 1902-12-01   1909       -1.54  1902
##
## $`1903`
## # A tibble: 12 x 4
##    Data       Maxima Normalizada   Ano
##    <date>      <dbl>       <dbl> <dbl>
##  1 1903-01-01   2094      -0.986  1903
##  2 1903-02-01   2172      -0.751  1903
##  3 1903-03-01   2332      -0.269  1903
##  4 1903-04-01   2523       0.307  1903
```

split

Vamos retornar aos dados de Cotas para mostrar aplicações das funções `split` e `tapply`

```
length(s)
```

```
## [1] 113
```

```
names(s)
```

```
##   [1] "1902" "1903" "1904" "1905" "1906" "1907" "1908" "1909" "1910" "1911"
##  [11] "1912" "1913" "1914" "1915" "1916" "1917" "1918" "1919" "1920" "1921"
##  [21] "1922" "1923" "1924" "1925" "1926" "1927" "1928" "1929" "1930" "1931"
##  [31] "1932" "1933" "1934" "1935" "1936" "1937" "1938" "1939" "1940" "1941"
##  [41] "1942" "1943" "1944" "1945" "1946" "1947" "1948" "1949" "1950" "1951"
##  [51] "1952" "1953" "1954" "1955" "1956" "1957" "1958" "1959" "1960" "1961"
##  [61] "1962" "1963" "1964" "1965" "1966" "1967" "1968" "1969" "1970" "1971"
##  [71] "1972" "1973" "1974" "1975" "1976" "1977" "1978" "1979" "1980" "1981"
##  [81] "1982" "1983" "1984" "1985" "1986" "1987" "1988" "1989" "1990" "1991"
##  [91] "1992" "1993" "1994" "1995" "1996" "1997" "1998" "1999" "2000" "2001"
## [101] "2002" "2003" "2004" "2005" "2006" "2007" "2008" "2009" "2010" "2011"
## [111] "2012" "2013" "2014"
```

split

Combinando com `lapply`:

```
sapply(s, function(x) {
  colMeans(x[, c("Maxima", "Normalizada")])
})
```

```
##                    1902        1903         1904         1905        1906
## Maxima      1884.333333 2330.583333 2496.7500000 2308.0833333 2184.333333
## Normalizada   -1.618363   -0.273102    0.2278225   -0.3409303   -0.713986
##                    1907         1908         1909          1910          1911
## Maxima      2315.5833333 2539.6666667 2.422250e+03 2410.83333333 2402.50000000
## Normalizada   -0.3183209    0.3571987 3.235459e-03   -0.03118112   -0.05630271
##                    1912         1913         1914         1915         1916
## Maxima      2210.7500000 2558.7500000 2.437833e+03 2.421917e+03 2271.7500000
## Normalizada   -0.6343505    0.4147271 5.021283e-02 2.230595e-03   -0.4504605
##                    1917         1918         1919         1920         1921
## Maxima      2274.3333333 2457.4166667 2313.5000000 2535.3333333 2494.3333333
## Normalizada   -0.4426728    0.1092486   -0.3246013    0.3441354    0.2205372
##                    1922         1923        1924         1925        1926
## Maxima      2607.4166667 2512.7500000 2301.25000 2467.9166667 1981.833333
## Normalizada    0.5614372    0.2760559   -0.36153    0.1409018   -1.324441
##                    1927         1928         1929          1930         1931
```

tapply

A função `tapply` pode ser pensada como uma combinação de `split` e `sapply`. Vejamos:

```
head(Cotas)
```

```
## # A tibble: 6 x 4
##   Data       Maxima Normalizada   Ano
##   <date>      <dbl>       <dbl> <dbl>
## 1 1902-10-01   1970      -1.36   1902
## 2 1902-11-01   1774      -1.95   1902
## 3 1902-12-01   1909      -1.54   1902
## 4 1903-01-01   2094      -0.986  1903
## 5 1903-02-01   2172      -0.751  1903
## 6 1903-03-01   2332      -0.269  1903
```

tapply

Pode-se calcular diretamente a média para subgrupos da tabela original:

```
tapply(Cotas$Maxima, Cotas$Ano, mean, na.rm = TRUE)
```

```
##     1902     1903     1904     1905     1906     1907     1908     1909
## 1884.333 2330.583 2496.750 2308.083 2184.333 2315.583 2539.667 2422.250
##     1910     1911     1912     1913     1914     1915     1916     1917
## 2410.833 2402.500 2210.750 2558.750 2437.833 2421.917 2271.750 2274.333
##     1918     1919     1920     1921     1922     1923     1924     1925
## 2457.417 2313.500 2535.333 2494.333 2607.417 2512.750 2301.250 2467.917
##     1926     1927     1928     1929     1930     1931     1932     1933
## 1981.833 2441.417 2429.833 2368.167 2415.333 2290.250 2446.250 2382.167
##     1934     1935     1936     1937     1938     1939     1940     1941
## 2491.917 2407.500 2249.167 2288.750 2380.083 2517.750 2385.250 2386.000
##     1942     1943     1944     1945     1946     1947     1948     1949
## 2376.667 2433.583 2486.750 2277.583 2443.250 2376.500 2389.333 2498.833
##     1950     1951     1952     1953     1954     1955     1956     1957
## 2397.667 2450.500 2406.083 2600.167 2499.750 2412.167 2471.333 2357.250
##     1958     1959     1960     1961     1962     1963     1964     1965
## 2361.500 2418.083 2419.750 2347.667 2456.083 2243.000 2257.417 2284.500
##     1966     1967     1968     1969     1970     1971     1972     1973
## 2274.083 2441.500 2400.167 2349.250 2462.667 2589.583 2559.000 2559.333
```

apply

A função `apply` aplica uma função nas linhas ou colunas de um `array`

```
apply(Cotas, 2, max, na.rm = TRUE)
```

```
##           Data         Maxima    Normalizada            Ano
##   "2014-06-01"         "2997" " 1.735871564"         "2014"
```

apply

A função `apply` aplica uma função nas linhas ou colunas de um `array`. O argumento `1` indica que a função vai ser aplicada ao longo das linhas, e `2` ao longo das colunas.

```
s <- matrix(rnorm(20), nrow = 5, ncol = 4)
s
```

```
##             [,1]       [,2]        [,3]         [,4]
## [1,]  0.06733069 -0.6849311  0.62334145 -0.594457477
## [2,] -1.90847322 -0.5635834 -0.91724147 -0.574666977
## [3,]  0.13766655 -0.5396565  0.54504315  0.003547333
## [4,]  0.45607935  0.0182417 -0.07192825 -1.036494559
## [5,]  1.11301410 -0.7061358 -0.51442433 -0.612578137
```

apply

A função `apply` aplica uma função nas linhas ou colunas de um `array`. O argumento `1` indica que a função vai ser aplicada ao longo das linhas, e `2` ao longo das colunas.

```
apply(s, 1, mean)
```

```
## [1] -0.14717910 -0.99099127  0.03665014 -0.15852544 -0.18003104
```

```
apply(s, 2, mean)
```

```
## [1] -0.02687651 -0.49521301 -0.06704189 -0.56292996
```

apply

A função `apply` aplica uma função nas linhas ou colunas de um `array`. O argumento `1` indica que a função vai ser aplicada ao longo das linhas, e `2` ao longo das colunas.

```
rowMeans(s)
```

```
## [1] -0.14717910 -0.99099127  0.03665014 -0.15852544 -0.18003104
```

```
colMeans(s)
```

```
## [1] -0.02687651 -0.49521301 -0.06704189 -0.56292996
```

Usando pacote `dplyr`

O pacote `dplyr` aqui também é útil pra gerar sumários de dados. A combinação da funções `group_by` e `summarize` permite gerar resultados semelhantes que aqueles usando as funções da família `appply`.

```
Cotas <- read_csv2("dados/cotas_T_14990000.txt", skip = 12) %>%
  filter(NivelConsistencia == 1 & MediaDiaria == 1) %>%
  mutate(Data = as.Date(Data, "%d/%m/%Y"), Ano = year(Data)) %>%
  group_by(Ano) %>%
  summarize(MaximaMedia = mean(Maxima, na.rm = TRUE)) %>%
  ungroup()
```

Usando pacote `dplyr`

O pacote `dplyr` aqui também é útil pra gerar sumários de dados. A combinação da funções `group_by` e `summarize` permite gerar resultados semelhantes que aqueles usando as funções da família `appply`.

```
tail(Cotas)
```

```
## # A tibble: 6 x 2
##     Ano MaximaMedia
##   <dbl>       <dbl>
## 1  2009       2592.
## 2  2010       2327.
## 3  2011       2409.
## 4  2012       2534.
## 5  2013       2574.
## 6  2014       2691
```

Material adicional

- Manual das funções (p. ex. ?tapply)
- Consultar o material do curso
- Stack Overflow
- R for Data Science