Condicionais, loops e funções

Leonardo Sangali Barone

March 20, 2017

Botando o computador para trabalhar

Se há um tutorial sobre lógica de programação, é este. Os tópicos deste tutorial são os mais importantes para que possamos escrever algoritmos e botar o computador para fazer o que sabe de melhor: repetir instruções e tarefas. Veremos, em primeiro lugar, operadores relacionais e lógicos, cláusulas condicionais, loops e funções.

Neste tutorial faremos uma introdução mais breve e intuitiva. Voltaremos aos 3 últimos assuntos – condicionais, loops e funções – no próximo tutorial com um pouco mais de complexidade.

Operadores relacionais

Uma das especialidades do computador é verificar se proposições simples são verdadeiras ou falsas. **Operadores relacionais** servem para verificar se objetos são iguais, diferentes, maiores ou menores. São seis operadores relacionais e a tabela abaixo apresenta os seus símbolos.

Operador	Símbolo
Igual	==
Diferente	!=
Maior	>
Maior ou igual	>=
Menor	<
Menor ou igual	<=

Não discutiremos todas as regras de comparação de objetos, mas passaremos por um conjunto grande de exemplos a partir do qual elas podem ser inferidas.

Vamos começar com alguns exemplos simples:

```
42 == 41
```

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

```
42 != 41
```

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

```
(2 + 2) == (3 + 1)
```

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

```
(2 + 2) != (3 + 1)
```

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

```
5 > 3
```

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

```
5 < 3
```

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

```
42 > 42
```

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

```
42 < 41
```

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

```
42 >= 42
```

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

```
42 <= 41
```

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

Antes de avanças, tenha certeza que entendeu os exemplos acima.

Operadores relacionais também vale para textos:

```
"texto" == "texto"
```

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

```
"texto" == "texTo"
```

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

```
"texto" != "texto"
```

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

Note no segundo exemplo que o R é "case sensitive", ou seja, diferencia maiúsculas de minúsculas ao comparar textos.

Textos também podem ser ordenados:

```
"a" > "b"
```

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

```
"a" < "b"
```

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

```
"A" < "b"
```

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

```
"A" > "a"
```

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

Inclusive palavras inteiras e sentenças:

```
"cachorro" < "cachorro quente"
```

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

```
"churrasco de gato" >  "cachorro quente"
```

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

E valores lógicos? Veja se entende o que acontece nos exemplos abaixo:

```
TRUE == 1
```

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

```
FALSE == 0
```

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

```
TRUE > FALSE
```

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

Podemos comparar valores armazenados em variáveis da mesma maneira que fizemos nos exemplos até aqui:

```
x <- 5
y <- 10
x > y
```

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

Operadores relacionais e vetores

É possível comparar um vetor com um valor. Neste caso, cada elemento do vetor é comparado individualmente ao valor e o resultado é um vetor lógico de tamanho igual ao vetor comparado.

```
votos16 <- c(1030, 551, 992, 345, 203, 2037)
votos16 >= 1000
```

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

Vamos usar o vetor "votos16", que contém votos de candidatos fictícios em 2016, com os votos dos mesmo candidatos em 2012:

```
votos12 <- c(890, 354, 950, 400, 50, 3416)
votos16 > votos12
```

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

Veja que, na comparação entre dois vetores, os elementos são comparados par a par de acordo com a sua posição no vetor. O vetor lógico resultante tem o mesmo tamanho dos vetores comparados.

Operadores Lógicos (Booleanos)

É perfeitamente possível combinar proposições com os **operadores lógicos** "e", "ou" e "não":

Operador	Símbolo
E	&
Ou	|
Não	!

Por exemplo, se queremos verificar todos os candidatos que obtivaram acima de 500 (exclusive) **E** abaixo de 1500 (inclusive) votos, fazemos:

```
votos16 > 500 & votos16 <= 1500
```

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

Veja a tabela de possibilidades de combinação de duas proposições com a conjunção "e":

Proposição 1	Proposição 2	Combinação
TRUE	TRUE	TRUE

Proposição 1	Proposição 2	Combinação
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Se o valor atende às duas condições, então o resultado é TRUE. Se ao menos uma proposição é falsa, sob a conjunção é, então a combinação das proposições também é.

Com o operador "Ou", a combinação de proposições é verdadeira se pelo menos uma delas for verdadeira.

```
votos16 < 500 | votos12 > 1500
```

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

Veja a tabela de possibilidades de combinação de duas proposições com a conjunção "ou":

Proposição 1	Proposição 2	Combinação
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Finalmente, o operador lógico "não" tem a única função de reverter um proposição:

```
!TRUE
```

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

```
!(5 > 3)
```

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

```
!(votos16 > 500 & votos16 <= 1500)
```

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

Lembre-se que, quando trabalhamos com vetores lógicos, podemos tratá-los como se fossem zeros e uns, tal qual no exemplo:

```
sum(votos16 > votos12)
```

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

Cláusulas condicionais

Um dos usos mais importantes dos operadores relacionais e lógicos é na construção de **cláusulas condicionais**, "if", "else" e "else if". Elas são fundamentais para a construção de funções e algoritmos. Veja um uso simples do condicional *if*, para o cálculo do valor absoluto de uma variável:

```
# exemplo com x negativo
x <- -23

if (x < 0){
  x <- -x
}

print(x)
```

```
## [1] 23
```

```
# exemplo com x positivo
x <- 23

if (x < 0){
  x <- -x
}

print(x)
```

```
## [1] 23
```

A condição que o *if* deve atender vem entre parênteses. A instrução a ser atendida caso a cláusula seja verdadeira, vem dentro das chaves. Aliás, é boa prática (na maioria dos casos) abrir as chaves em uma linha, escrever as instruções em outra, e fechar as chaves na linha seguinte ao fim das instruções, como no exemplo. Também é boa prática "identar", ou seja, desalinhar as instruções do restante do código. Falaremos sobre "estilo" em algum momento do curso, Por enquanto, apenas observe e não se assuste. Diferentemente de outras linguagens, R não requer identação para funcionar.

Vamos supor que um candidato teve determinada quantia de votos. Ele precisava de 700 para ser eleito. Vamos criar uma nova variável, "status", que receberá valor "eleito" se "votos" for maior que 700. Supondo que o candidato recebeu 800 votos, fazemos:

```
votos <- 800

if (votos > 700){
  status <- "eleito"
}

print(status)
```

```
## [1] "eleito"
```

Mas e se quisermos dar o valor "nao eleito" a "status" caso ele não tenha recebido mais de 700 votos? Usamos *else* para indicar o que fazer em todos os casos em que a condição em *if* não foi atendida.

```
if (votos > 700){
  status <- "eleito"
} else {
  status <- "nao eleito"
}

print(status)
```

```
## [1] "eleito"
```

Por fim, vamos imaginar um regra mais complexa. Se o total de "votos" do candidato for maior que 1200, então ele está eleito. Se a votação tiver ficado entre 700 (exclusive) e 1200 (inclusive), ele recebe "status" igual a "suplente". Com 700 votos ou menos, o "status" é não eleito.

Veja como traduzir a regra acima em código usando *if*, *else if* e *else*.

```
if (votos > 1200){
  status <- "eleito"
} else if (votos > 700  & votos <= 1200){
  status <- "suplente"
} else {
```

```
  status <- "nao eleito"
}

print(status)
```

```
## [1] "suplente"
```

Outro exemplos simples do uso de condicionais, agora com variável de texto e cláusulas

```
partido <- "PMDB"

if (partido == "PMDB" | partido == "PSD" | partido == "DEM") {
  print("governo")
} else {
  print("oposição")
}
```

```
## [1] "governo"
```

É possível complicar bastante o uso dos condicionais "aninhando" uma cláusula dentro da outra e criando labirintos de condições. Fazer isso, porém, é mais uma questão de lógica do que de uso da linguagem. Não há variações relevantes do uso em relação aos exemplos simples apresentados acima. Se você consegue fazer os condicionais em papel e caneta, então consegue fazê-los em R se os exemplos anteriores tiverem ficado claro.

Exercício

- Anote quantos cafés você no fim de semana
- Reprense com condicionais a regra: se você tomou 3 ou mais, imprima "Hummm, café!". Se você tomou menos de 3, imprima "zzzzzz".

Repetindo tarefas - while loop

Uma das vantagens dos computadores em relação aos seres humanos é a capacidade de repetir tarefas a um custo baixo. Vamos ver um exemplo simples: contar até 42. Usaremos como recurso um *while* loop, ou seja, daremos um estado inicial, uma condição e uma instrução para o computador e pediremos para ele repetir **enquanto** a instrução enquanto a condição não for atendida.

Nosso caso, a instrução será: imprima o número "atual" (você já entenderá isso), armazenado na variável "contador", e some mais um. A condição será: enquanto a variável "contador" for menor ou igual a 42. Se vamos começar a contar a partir do 1, nosso estado inicial será igualar o "contador" a um Veja:

```
contador <- 1

while (contador <= 42) {
  print(contador)
  contador <- contador + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

```
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
```

```
print(contador)
```

```
## [1] 43
```

Traduzindo para o português: "enquanto o contador for menor ou igual a 42, imprima e some um". A estrutura de um *while* loop é sempre: "enquanto" (condição), "faça".

Veja que, precisamos planejar muito bem o *while* loop. Se, por exemplo, esquecermos de pensar como a condição inicial será alterada a cada **iteração** (sem o "n" mesmo, pois é diferente de "interação"), corremos o risco de criar um "loop infinito". O critério de parada (condição entre parênteses)

Vamos complicar. Seguiremos contando até 42, mas todas as vezes em que o número for par (ou seja, múltiplo de 2), deixaremos de imprimir o número. Como fazer isso? Com *if*.

Dica: para saber se um número é divisível por outro, basta usar o resto da divisão (consulte o tutorial anterior %%) e checar se é igual a zero.

```
contador <- 1

while (contador <= 42) {
  if ((contador %% 2) != 0){
```

```
    print(contador)
  }
  contador <- contador + 1
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
## [1] 11
## [1] 13
## [1] 15
## [1] 17
## [1] 19
## [1] 21
## [1] 23
## [1] 25
## [1] 27
## [1] 29
## [1] 31
## [1] 33
## [1] 35
## [1] 37
## [1] 39
## [1] 41
```

```
print(contador)
```

```
## [1] 43
```

Veja que temos agora um código "aninhado", pois colocamos um condicional dentro de um loop. Novamente, combinar estruturas de código são mais um problema de lógica do que de linguagem e, se você consegue fazer no papel, consegue traduzir para o R.

Faremos exercícios com loops e condicionais no futuro.

Repetindo tarefas - for loop

E se em vez de repetir uma tarefa até atingir uma condição já soubermos quantas vezes queremos repetí-la? Neste caso, podemos usar o *for* loop. O loop não será mais do tipo "enquanto x faça y" mas "para todo i em a até b". Veja como o exemplo de contar até 42 ficaria com *for* loop:

```
for (i in 1:42){
  print(i)
}
```

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

```
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
```

Neste caso, lemos "para cada i em 1 até 42, faça". O que o *for* loop faz é variar o i a cada iteração de acordo com a sequência estabelecida. Outro exemplo, agora na ordem reversa:

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

```
## [1] 10
## [1] 9
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
## [1] -1
## [1] -2
```

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

Agora com a condição de não imprimir os pares:

```
for (i in 1:42){
  if((i %% 2) != 0){
    print(i)
  }
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
## [1] 11
## [1] 13
## [1] 15
## [1] 17
## [1] 19
## [1] 21
## [1] 23
## [1] 25
## [1] 27
## [1] 29
## [1] 31
## [1] 33
## [1] 35
## [1] 37
## [1] 39
## [1] 41
```

for loops não precisam ser apenas com números. Na verdade, você pode colocar após o "in" qualquer vetor. Por exemplo, um vetor das regiões brasileiras (ou UFs, se você tiver paciência de escrever todas):

```
vetor_regioes <- c("norte", "nordeste", "sudeste","sul", "centro-oeste")

for (regiao in vetor_regioes){
  print(regiao)
}
```

```
## [1] "norte"
## [1] "nordeste"
## [1] "sudeste"
## [1] "sul"
## [1] "centro-oeste"
```

Se você já trabalhou com dados eleitorais brasileiros, você certamente teve de abrir diversos arquivos semelhantes, cada um contendo informações de um estado brasileiro. Ou ainda, se você já obteve informações na internet, talvez tenha precisado "passar" por diversas páginas semelhantes. Loops resolvem problemas

desse tipo: eles repetem procedimentos variando apenas um índice. Aprender a usar loops economiza um tempo enorme, pois conseguimos automatizar tarefas ou, pelo menos, escrever um código mais curto para aplicar o mesmo comando inúmeras vezes.

Vamos para por aqui com *loops* e voltaremos a eles para fazermos exercícios.

Escrevendo funções

Ao longo dos três tutorais que fizemos, usamos diversas funções e já estamos acostumados com elas. Vamos agora aprender a construir funções simples. Vamos do exemplo à aplicação e nosso exemplo será um conversos de farenheit para celsius:

```
conversor <- function(farenheit){
  celsius <- ((farenheit - 32) / 9) * 5
  return(celsius)
}

conversor(212)
```

```
## [1] 100
```

```
conversor(32)
```

```
## [1] 0
```

Temos vários elementos no "construtor" de funções. Em primeiro lugar, criamos a função como criamos um objeto. Quer dizer, escolhemos um nome para ela e atribuímos a função criada a esse nome.

Para criar uma função, usamos *function*. Basicamente, o "construtor" tem duas partes: or argumentos da função, que inserimos no parênteses após *function*; e o corpo da função, que utiliza os argumentos para realizar uma tarefa e retorna um resultado, indicado pela função *return*.

Exercício

Crie uma função chamada "quadrado" que recebe um número "x" e retorna o quadrado de x.

Reposta

```
quadrado <- function(x){
  resultado <- x * x
  return(resultado)
}

quadrado(4)
```

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

```
quadrado(17)
```

```
## [1] 289
```

Exercício

- Crie uma função que recebe um valor em reais e retorna o valor em dólares (use a 3.2 como cotação do dólar)
- Crie uma função que recebe um valor em reais e uma cotação do dólar e retorna o valor em dólares.

Reposta do segundo item

```
conversor_moeda <- function(valor, cotacao){
  dolares <- valor / cotacao
  return(dolares)
}

conversor_moeda(42, 3.2)
```

```
## [1] 13.125
```

Paramos por aqui

Hoje percorremos um caminho longo no aprendizado da linguagem R. Vamos parar por aqui para respirar e assentar o que aprendemos.