R para Ciência de Dados 2

Controle de fluxo e funções

Iniciar gravação!

Controle de fluxo

Motivação

Já aprendemos muito sobre o R até agora e sabemos fazer coisas realmente incríveis com os pacotes que vimos. Mas ainda não falamos (formalmente) sobre um dos aspectos mais fundamentais da programação: controle de fluxo.

Controle de fluxo é o termo usado para descrever as instruções que nos permitem controlar a ordem de execução do código. Normalmente elas são divididas em **condicionais** (`if`, `else`) e **loops** (`while`, `for`), pois assim fica mais fácil de entender a utilidade de cada instrução.

Apesar da sua importância, esse conceito nem sempre é necessário para fazer uma análise de dados, por isso esperamos tanto para aprender mais sobre ele. Na prática, já usamos muito controle de fluxo neste curso, só nunca tão explicitamente quanto hoje.

Mas por que teríamos interesse em mudar a ordem de execução do código? Até agora, "de cima para baixo" serviu para tudo que precisávamos... Qual é a utilidade do controle de fluxo?

Ordem de execução

O que fazer se quisermos executar um certo código somente se uma condição for atendida? No exemplo abaixo, só podemos fazer a multiplicação de x for numérico.

```
x <- 10
x * 2
```

```
#> [1] 20
```

Se x puder assumir outro tipo de valor, a multiplicação não fará sentido e irá retornar um erro. Neste caso, o ideal seria "pular" a multiplicação.

```
x <- "dez"
x * 2
```

```
#> Error in x * 2: non-numeric argument to binary operator
```

if

A instrução que nos permite executar comandos **condicionalmente** se chama `if`.

> ***Se*** *x for numérico, multiplicá-lo por 2.*

```
x <- 10
if (is.numeric(x)) {
  x * 2
}
```

```
#> [1] 20
```

```
x <- "dez"
if (is.numeric(x)) {
  x * 2 # Não executa
}
```

Anatomia do if

O `if` é sempre acompanhado de parênteses com uma **condição** (mais sobre isso no próximo slide) e um par de chaves com um **corpo**, ou seja, todos os comandos a serem executados caso a condição seja verdadeira.

```
if (CONDICAO) {
  COMANDO_1
  COMANDO_2
  COMANDO_3
  # Resto do corpo...
}
```

Não há nenhuma restrição sobre o que pode ir dentro do corpo de um `if`; qualquer código R válido pode ser colocado lá dentro, inclusive outros `if`s.

Antes de prosseguir, precisamos entender exatamente o que conta como condição e como podemos criar condições mais complexas do que `is.numeric()`.

Condições

As condições do `if` estão intimamente ligadas aos valores lógicos do R: `TRUE` e `FALSE`. Um `if` precisa que a expressão entre parênteses retorne um único valor lógico, senão ele não sabe se deve executar o corpo ou não.

```
x <- 10
is.numeric(x)
```

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

Existem inúmeras outras funções do R que retornam lógicos (chamadas **predicados**): `is.na()`, `is.character()`, `is.Date()`, `is.factor()`, `is.list()`, etc.

```
is.character(x)
```

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

Condições: comparações

Já falamos sobre comparações no *R para Ciência de Dados I*, então basta só dizer que todas retornam valores lógicos que podem ser usados no `if`.

```
1 == 1 # Igual
```

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

```
1 != 1 # Diferente
```

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

```
2 > 2 # Maior
```

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

```
2 >= 2 # Maior ou igual
```

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

```
3 < 3 # Menor
```

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

```
3 <= 3 # Menor ou igual
```

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

Condições: álgebra booleana

Podemos combinar comparações e predicados usando álgebra booleana, um campo da matemática que lida com valores lógicos.

```
TRUE & FALSE # x E y
```

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

```
TRUE | FALSE # x OU y
```

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

```
!TRUE # NÃO x
```

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

```
xor(TRUE, TRUE) # x XOU y
```

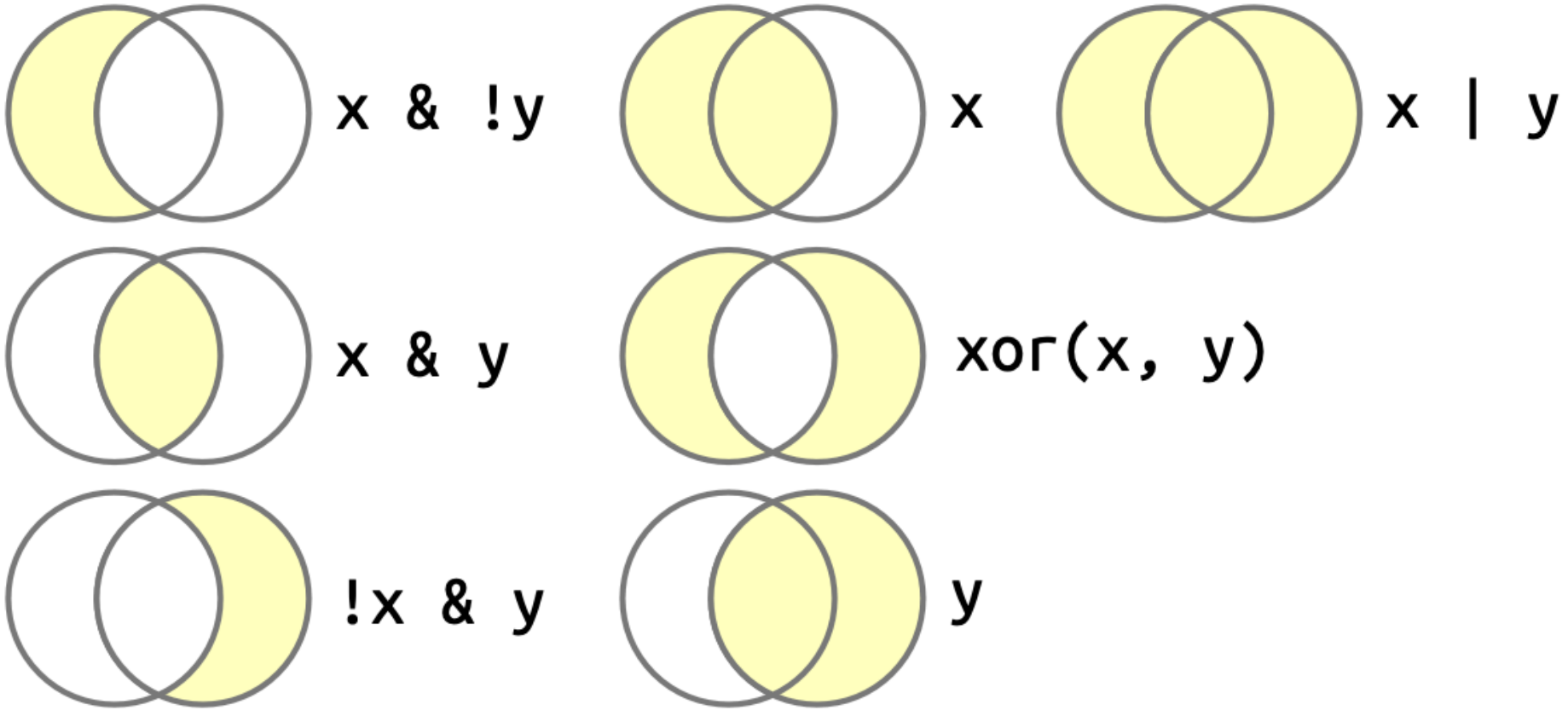
```
#> [1] FALSE
```

```
!TRUE & FALSE # NÃO x E y
```

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

```
TRUE & !FALSE # x E NÃO y
```

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


x & !y
x
x | y
x & y
xor(x, y)
!x & y
y

Condições: conjuntos

Existem operações lógicas para conjuntos. Algumas (como `%in%`) retornam apenas um valor lógico, mas outras (como `all()` e `any()`) sumarizam vetores lógicos.

```
x <- c(1, 5, 10, 15)
10 %in% x # Pertence ao conjunto
```

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

```
all(x > 7) # Verdadeiro para todos
```

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

```
any(x > 7) # Verdadeiro para algum
```

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

Voltando para o if

Podemos usar todas as condições que vimos nos últimos slides dentro do `if`.

```
x <- 10
if (is.numeric(x) & (x < 15)) {
  x * 2
}
```

```
#> [1] 20
```

```
x <- c(1, 2, 3, 4)
if (all(x > 3) | !any(x < 2)) {
  x / 5
}
```

Só não aprendemos ainda como executar algo caso a condição retorne `FALSE`...

else

O maior parceiro do `if` é o `else`. Ele não tem uma condição, apenas um corpo que é executado caso a condição do `if` seja falsa.

> ***Se** x for maior que 0, imprimir "positivo". **Senão**, imprimir "negativo".*

```
x <- -10
if (x > 0) {
  print("positivo")
} else {
  print("negativo")
}
```

```
#> [1] "negativo"
```

Agora só nos falta uma instrução que permita adicionar condições intermediárias.

else if

O `else if` é uma instrução que encadeia condições extras depois de um `if`. Ela pode aparecer uma ou mais vezes e pode ou não ser fechada com um `else`.

> ***Se*** *x for menor que 3, imprimir "reprovar".* ***Senão, se*** *x for menor que 5, imprimir "recuperar".* ***Senão****, imprimir "aprovar".*

```
x <- 8
if (x < 3) {
  print("reprovar")
} else if (x < 5) {
  print("recuperar")
} else {
  print("aprovar")
}
```

```
#> [1] "aprovar"
```

Repetições

O que fazer se quisermos executar um certo código várias vezes? No exemplo abaixo, tentamos duplicar `x` até que ele fique maior ou igual a 50:

```
x <- 10
if (x < 50) {
  x <- x * 2
  if (x < 50) {
    x <- x * 2
    if (x < 50) {
      x <- x * 2
    }
  }
}
x
```

```
#> [1] 80
```

while

A instrução que nos permite executar comando repetidamente se chama `while`. O nome técnico deste processo é **iteração**, mas é mais comum falar **loop**.

> ***Enquanto*** *x for menor que 50, multiplicá-lo por 2.*

```
x <- 10
while (x < 50) {
  x <- x * 2
}
x
```

```
#> [1] 80
```

```
x <- 60
while (x < 50) {
  x <- x * 2 # Não executa
}
x
```

```
#> [1] 60
```

O que acontece se `x` for negativo quando chegar a hora do `while` ser executado? O que acontece se esquecermos de atualizar o valor de `x` dentro do corpo do `while`?

Anatomia do while

O `while` é muito parecido com tudo que vimos até agora: ele tem uma **condição** entre parênteses e um **corpo** entre chaves. O corpo será executado repetidamente até que a condição se torne `FALSE`.

```
while (CONDICAO) {
  COMANDO_1
  COMANDO_2
  COMANDO_3
  # Resto do corpo...
}
```

É importantíssimo ter certeza de que existe uma **condição de parada**, ou seja, uma situação em que o loop termina de executar. Se não fizermos isso, ele vai rodar permanentemente e teremos que interromper o nosso programa.

Seria bom ter um jeito mais simples de garantir que o `while` vai parar...

for

Um outro tipo de loop é o `for`. Ele é um pouco diferente do que vimos até agora, mas segue o mesmo espírito do `while`.

> ***Para** `i` indo de 1 a 3, imprimi-lo.*

```
for (i in 1:3) {
  print(i)
}
```

```
#> [1] 1
#> [1] 2
#> [1] 3
```

Diferentemente do `while`, o `for` é usado quando já sabemos quantas vezes queremos executar o código do corpo. Nessas ocasiões, é mais fácil definir isso já no começo, assim não precisamos nos preocupar com condições de parada.

Anatomia do for

O `for`, diferente dos outros, tem uma **declaração** dentro dos seus parênteses: criamos um objeto, chamado **iterador**, que vai assumir cada um dos valores de um **vetor**. A partir daí, o **corpo** será executado uma vez com cada valor do iterador.

```
for (ITERADOR in VETOR) {
  COMANDO_1
  COMANDO_2
  COMANDO_3
  # Resto do corpo...
}
```

O iterador, apesar de criado dentro da declaração do `for`, continua existindo *fora* do loop! Ele mantém o seu valor da última iteração.

Obs.: Se quisermos, também podemos usar uma lista no lugar do vetor.

Iterando direto nos dados

Não necessariamente o iterador precisa ser uma contagem das iterações a serem feitas. Se quisermos, podemos passar nossos dados (uma coluna de uma tabela, por exemplo) para o iterador e trabalhar direto com eles.

```
for (nota in c(1, 4, 6)) {
  if (nota < 5) {
    print("recuperar")
  } else {
    print("aprovar")
  }
}
```

```
#> [1] "recuperar"
#> [1] "recuperar"
#> [1] "aprovar"
```

break() e next()

Existem situações em que queremos sair mais cedo de um loop e em que queremos pular uma iteração específica. Fazemos isso, respectivamente, com `break()` e `next()`, e ambos funcionam tanto com `while` quanto com `for`.

```
for (x in 1:5) {
  if (x == 4) {
    break()
  }
  print(x)
}
```

```
#> [1] 1
#> [1] 2
#> [1] 3
```

```
for (x in 1:5) {
  if (x %% 2 == 0) { # x é par
    next()
  }
  print(x)
}
```

```
#> [1] 1
#> [1] 3
#> [1] 5
```

Rodada bônus!

O dplyr tem funções que nos permitem colocar controle de fluxo dentro de um `mutate()`. A `if_else()` emula um `if` e um `else`, enquanto a `case_when()` emula um `if` e vários `else if`s. Ambas iteram nas entradas como um `for`.

```
x <- c(-1, 1, 2, 4, 6)
if_else(x > 0, "positivo", "negativo")
```

```
#> [1] "negativo" "positivo" "positivo" "positivo" "positivo"
```

```
case_when(
  x < 3 ~ "reprovar",
  x < 5 ~ "recuperar",
  TRUE  ~ "aprovar" # Equivale a `else if (TRUE)`, ou seja, `else`
)
```

```
#> [1] "reprovar"  "reprovar"  "reprovar"  "recuperar" "aprovar"
```

Funções

Motivação

Apesar de já termos aprendido sobre funções no *R para Ciência de Dados I*, ainda há alguns tópicos relevantes que precisamos cobrir. Funções são a base da programação em qualquer linguagem, então este é um assunto que vale a pena relembrar e aprofundar.

Pode não parecer, mas funções também são uma forma de controle de fluxo! Uma função é a execução não-linear de um trecho de código com algumas regras sobre o escopo das variáveis criadas lá dentro.

Também precisamos preparar o terreno para a próxima aula, quando vamos aprender sobre listas e o pacote purrr. Grande parte do tempo vamos estar falando sobre **programação funcional**, então a aula de hoje é essencial.

Mas o que está faltando? Entendemos como usar uma função na prática, mas quanto de fato sabemos tudo que é necessário para construir boas funções?

function

A instrução `function` define um trecho de código como sendo uma função. O termo técnico para isso é **sub-rotina**.

> *f é uma **função** que recebe x e y, e os multiplica.*

```
f <- function(x, y) {
  x * y
}
f(2, 3)
```

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

```
f(3, 4)
```

```
#> [1] 12
```

Anatomia do function

O `function` tem uma sintaxe semelhante ao que já vimos até agora. A maior diferença é que ela tem um **nome**, que será usado para invocá-la posteriormente. Os **argumentos** também merecem atenção especial, então logo vamos falar sobre eles.

```
NOME <- function(ARGUMENTOS) {
  COMANDO_1
  COMANDO_2
  # Resto do corpo...
  RETORNO
}
```

O mais importante de entender é que os comandos são executados em um **escopo** isolado, ou seja, o que é criado dentro do corpo não afeta o resto do programa. Existem exceções, mas é melhor evitarmos um um comportamento diferente.

No geral, o último comando é o valor de **retorno**, mas isso também tem exceções.

return()

No R, o uso do `return()` é incomum porque uma função automaticamente retorna o resultado da sua última linha, mas às vezes precisamos de um **retorno antecipado**.

```
quadrado <- function(x) {
  if (!is.numeric(x)) {
    return("Erro")
  }
  x ^ 2
}
quadrado("10")
```

```
#> [1] "Erro"
```

```
quadrado(10)
```

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

Argumentos

Os argumentos de uma função são objetos que o(a) usuário(a) pode passar para dentro do escopo na hora de chamar a função. Eles aparecem como uma sequência de nomes separados por vírgula e podem ou não ter **valores padrões**.

```
soma4 <- function(arg1, arg2, arg3, arg4 = 4) {
  arg1 + arg2 + arg3 + arg4
}
tres <- 3
soma4(1, 4 / 2, tres)
```

```
#> [1] 10
```

Um argumento pode receber um valor, um resultado ou um objeto. Um argumento com valor padrão pode ser omitido na hora da chamada e, somente neste caso, ele receberá o seu valor padrão.

Argumentos: ordem

Não é necessário especificar os nomes dos argumentos se usarmos todos em ordem (como no slide passado). Se quisermos especificar um argumento sem especificar um anterior, aí precisamos usar seu nome.

```
soma3 <- function(arg1, arg2 = 2, arg3) {
  arg1 + arg2 + arg3
}
soma3(1, arg3 = 3) # Pulando um argumento
```

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

```
soma3(arg3 = 3, arg2 = 2, arg1 = 1) # Fora de ordem (incomum)
```

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

Argumentos: dots

A primeira grande novidade sobre funções são os **dots** (formalmente chamados de "ellipsis"). O usuário pode passar um número qualquer de argumentos e todos serão capturados juntos em uma lista (mais sobre elas na próxima aula).

```
lista_dots <- function(...) {
  list(...)
}
lista_dots(123, "abc")
```

```
#> [[1]]
#> [1] 123
#>
#> [[2]]
#> [1] "abc"
```

Argumentos: usando dots

Para usar os dots, podemos transformá-los em lista como no slide passado ou podemos redirecioná-los para outra função que recebe dots. É importante notar que todos os argumentos que vêm *depois* dos dots precisam ser nomeados pelo usuário.

```
somaN <- function(arg1, ..., arg3) {
  sum(arg1, arg3, ...)
}
somaN(1, 2, 3, 4, arg3 = 5)
```

```
#> [1] 15
```

Na prática, é muito raro precisarmos dos dots, mas em situações específicas eles são essenciais. Notem como quase todas as funções do dplyr têm os dots como segundo argumento para que possamos criar/filtrar/selecionar quantas colunas quisermos.

Vamos ver os dots de novo na última aula quando falarmos de NSE.

Funções vetorizadas

Funções **vetorizadas** são aquelas que conseguem trabalhar com uma entrada de qualquer tamanho. No geral queremos garantir que as nossas funções se comportam assim para que elas funcionem dentro do `mutate()`.

```
pequeno <- function(xs) {
  if (xs < 5) {
    print("pequeno")
  }
}
pequeno(3:5)
```

```
#> Error in if (xs < 5) {: the
condition has length > 1
```

Precisamos vetorizar a aplicação do `if`.

```
pequeno <- function(xs) {
  for (x in xs) {
    if (x < 5) {
      print("pequeno")
    }
  }
}
pequeno(3:5)
```

```
#> [1] "pequeno"
#> [1] "pequeno"
```

Rodada bônus!

Fábricas de funções são funções que retornam funções 😳 Isso é uma prática bastante rara, mas existem funções importantes que se comportam assim (como a `possibly()` do pacote purrr).

```
cria_arredondador <- function(n_casas) {
  function(valor) {
    round(valor, digits = n_casas)
  }
}

arredonda2 <- cria_arredondador(2)

arredonda2(pi)
```

```
#> [1] 3.14
```

Fim