

Aula 06 – Funções, Estruturas de Controle e Repetição

Adhemar Ranciaro Neto

Introdução

Bem-vindo ao curso de R focado em funções, estruturas de controle e repetição! Este curso é projetado para iniciantes e intermediários que desejam aprofundar seus conhecimentos em programação em R. Vamos cobrir os conceitos fundamentais, com exemplos práticos e exercícios.

Pré-requisitos: Conhecimento básico de R, como vetores, data frames e operações simples.

Objetivos: - Entender como criar e usar funções em R. - Dominar estruturas de controle como `if`, `else` e `ifelse`. - Aprender loops como `for`, `while` e `repeat`. - Explorar alternativas vetorizadas para loops eficientes.

Vamos começar!

Seção 1: Funções em R

Funções são blocos de código reutilizáveis que executam tarefas específicas. Em R, você pode definir suas próprias funções usando a palavra-chave `function`. Elas permitem modularizar o código, tornando-o mais organizado, legível e fácil de depurar. Funções podem receber entradas (argumentos), processá-las e retornar saídas.

Definindo uma Função

Uma função básica tem a forma:

```
minha_funcao <- function(argumento1, argumento2 = valor_padrao) {  
  # Código aqui  
  return(resultado)  
}
```

Exemplo simples: Uma função para calcular a soma de dois números.

```
soma <- function(a, b) {  
  resultado <- a + b  
  return(resultado)  
}  
  
soma(5, 3)
```

```
## [1] 8
```

Outro exemplo: Uma função para calcular o quadrado de um número.

```
quadrado <- function(x) {  
  return(x^2)  
}
```

```
quadrado(4)
```

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

```
quadrado(-2)
```

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

Argumentos e Valores Padrão

Você pode definir valores padrão para argumentos, tornando a função mais flexível.

```
saudacao <- function(nome = "Mundo", saudacao = "Olá") {  
  paste(saudacao, ",", nome, "!")  
}
```

```
saudacao() # Usa valores padrão
```

```
## [1] "Olá , Mundo !"
```

```
saudacao("Aluno", "Bem-vindo")
```

```
## [1] "Bem-vindo , Aluno !"
```

```
saudacao(saudacao = "Oi", nome = "Equipe")
```

```
## [1] "Oi , Equipe !"
```

Situação prática: Uma função para calcular IMC (Índice de Massa Corporal), com unidades padrão.

```
calcular_imc <- function(peso, altura, unidade_peso = "kg", unidade_altura = "m") {  
  if (unidade_peso == "lb") peso <- peso / 2.20462  
  if (unidade_altura == "cm") altura <- altura / 100  
  imc <- peso / (altura^2)  
  return(round(imc, 2))  
}
```

```
calcular_imc(70, 1.75) # kg e m
```

```
## [1] 22.86
```

```
calcular_imc(154, 175, "lb", "cm") # lb e cm
```

```
## [1] 22.81
```

Argumentos Variáveis com ...

O operador ... permite passar um número variável de argumentos.

Exemplo: Uma função que soma qualquer número de valores.

```
soma_variavel <- function(...) {  
  numeros <- list(...)  
  sum(unlist(numeros))  
}  
  
soma_variavel(1, 2, 3)
```

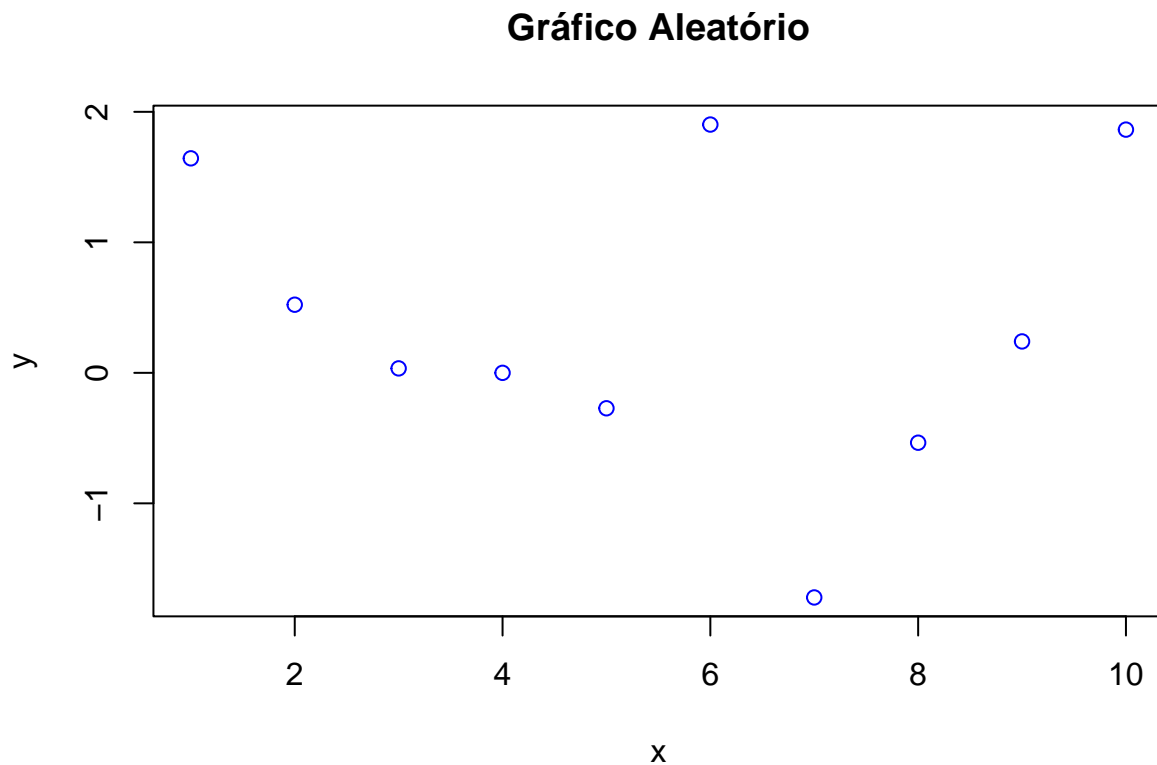
```
## [1] 6
```

```
soma_variavel(10, 20)
```

```
## [1] 30
```

Situação: Passando argumentos para funções internas, como em plotagens.

```
plot_personalizado <- function(x, y, ...) {  
  plot(x, y, ...)  
}  
  
plot_personalizado(1:10, rnorm(10), main = "Gráfico Aleatório", col = "blue")
```



Funções que Retornam Múltiplos Valores

Funções podem retornar listas ou vetores para múltiplas saídas.

Exemplo: Função que retorna média e desvio padrão.

```
estatisticas_basicas <- function(vetor) {  
  list(media = mean(vetor), desvio_padrao = sd(vetor))  
}  
  
dados <- c(1, 2, 3, 4, 5)  
estatisticas_basicas(dados)
```

```
## $media  
## [1] 3  
##  
## $desvio_padrao  
## [1] 1.581139
```

Funções Anônimas

Funções sem nome, úteis para operações rápidas, especialmente com `apply` e família.

```
lapply(1:3, function(x) x^2)
```

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

Situação: Aplicando a uma lista de vetores.

```
listas <- list(a = 1:5, b = 6:10)  
lapply(listas, function(v) sum(v))
```

```
## $a  
## [1] 15  
##  
## $b  
## [1] 40
```

Funções Recursivas

Funções que chamam a si mesmas, úteis para problemas como fatorial.

Exemplo: Fatorial recursivo.

```
fatorial <- function(n) {
  if (n <= 1) return(1)
  else return(n * fatorial(n - 1))
}

fatorial(5)
```

```
## [1] 120
```

Situação: Sequência de Fibonacci.

```
fibonacci <- function(n) {
  if (n <= 1) return(n)
  else return(fibonacci(n-1) + fibonacci(n-2))
}

fibonacci(6)
```

```
## [1] 8
```

Uso de Funções com Datasets Embutidos

R possui datasets embutidos em pacotes como `datasets`. Vamos usar `mtcars` para exemplos.

Exemplo: Função para resumir consumo de combustível por cilindros.

```
data(mtcars) # Carrega o dataset

resumir_mpg_por_cilindros <- function(df = mtcars) {
  aggregate(mpg ~ cyl, data = df, FUN = mean)
}

resumir_mpg_por_cilindros()
```

```
##   cyl   mpg
## 1   4 26.66364
## 2   6 19.74286
## 3   8 15.10000
```

Outro exemplo com `iris`: Função para calcular médias por espécie.

```
data(iris)

medias_por_especie <- function(df = iris) {
  aggregate(. ~ Species, data = df, FUN = mean)
}

medias_por_especie()
```

```
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1   setosa      5.006      3.428      1.462      0.246
## 2 versicolor      5.936      2.770      4.260      1.326
## 3 virginica      6.588      2.974      5.552      2.026
```

Situação: Função para filtrar e sumarizar.

```
filtrar_e_sumarizar <- function(df, coluna_filtro, valor_filtro) {  
  subset_df <- df[df[[coluna_filtro]] == valor_filtro, ]  
  summary(subset_df)  
}  
  
filtrar_e_sumarizar(iris, "Species", "setosa")
```

```
##   Sepal.Length   Sepal.Width   Petal.Length   Petal.Width  
##   Min.    :4.300   Min.    :2.300   Min.    :1.000   Min.    :0.100  
##   1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200  
##   Median :5.000   Median :3.400   Median :1.500   Median :0.200  
##   Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246  
##   3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300  
##   Max.    :5.800   Max.    :4.400   Max.    :1.900   Max.    :0.600  
##           Species  
##   setosa      :50  
##   versicolor : 0  
##   virginica   : 0  
##  
##  
##
```

Funções com Tidyverse

O tidyverse é uma coleção de pacotes para manipulação de dados em R, promovendo um estilo de programação mais legível e eficiente. Vamos carregar o tidyverse e mostrar exemplos de funções que o utilizam.

Primeiro, instale e carregue o tidyverse se necessário (assumindo que está instalado):

```
# install.packages("tidyverse") # Descomente se necessário  
library(tidyverse)
```

Exemplo: Função para resumir dados usando dplyr.

```
resumir_mpg_por_cilindros_tidy <- function(df = mtcars) {  
  df %>%  
    group_by(cyl) %>%  
    summarise(media_mpg = mean(mpg))  
}  
  
resumir_mpg_por_cilindros_tidy()
```

```
## # A tibble: 3 x 2  
##       cyl media_mpg  
##   <dbl>     <dbl>  
## 1     4       26.7  
## 2     6       19.7  
## 3     8       15.1
```

Situação: Função para filtrar, selecionar e arranjar dados.

```

processor_iris <- function(df = iris, especie = "setosa") {
  df %>%
    filter(Species == especie) %>%
    select(Petal.Length, Petal.Width) %>%
    arrange(desc(Petal.Length))
}

processor_iris()

```

```

##      Petal.Length Petal.Width
## 1           1.9         0.2
## 2           1.9         0.4
## 3           1.7         0.4
## 4           1.7         0.3
## 5           1.7         0.2
## 6           1.7         0.5
## 7           1.6         0.2
## 8           1.6         0.2
## 9           1.6         0.4
## 10          1.6         0.2
## 11          1.6         0.2
## 12          1.6         0.6
## 13          1.6         0.2
## 14          1.5         0.2
## 15          1.5         0.2
## 16          1.5         0.1
## 17          1.5         0.2
## 18          1.5         0.4
## 19          1.5         0.3
## 20          1.5         0.4
## 21          1.5         0.2
## 22          1.5         0.4
## 23          1.5         0.1
## 24          1.5         0.2
## 25          1.5         0.2
## 26          1.5         0.2
## 27          1.4         0.2
## 28          1.4         0.2
## 29          1.4         0.2
## 30          1.4         0.3
## 31          1.4         0.2
## 32          1.4         0.1
## 33          1.4         0.3
## 34          1.4         0.2
## 35          1.4         0.2
## 36          1.4         0.1
## 37          1.4         0.3
## 38          1.4         0.2
## 39          1.4         0.2
## 40          1.3         0.2
## 41          1.3         0.4
## 42          1.3         0.2
## 43          1.3         0.2

```

```
## 44      1.3      0.3
## 45      1.3      0.3
## 46      1.3      0.2
## 47      1.2      0.2
## 48      1.2      0.2
## 49      1.1      0.1
## 50      1.0      0.2
```

Exemplo com mutate: Função para adicionar uma nova coluna.

```
adicionar_coluna_imc <- function(df, peso_col, altura_col) {
  df %>%
    mutate(IMC = !!sym(peso_col) / (!!sym(altura_col))^2)
}

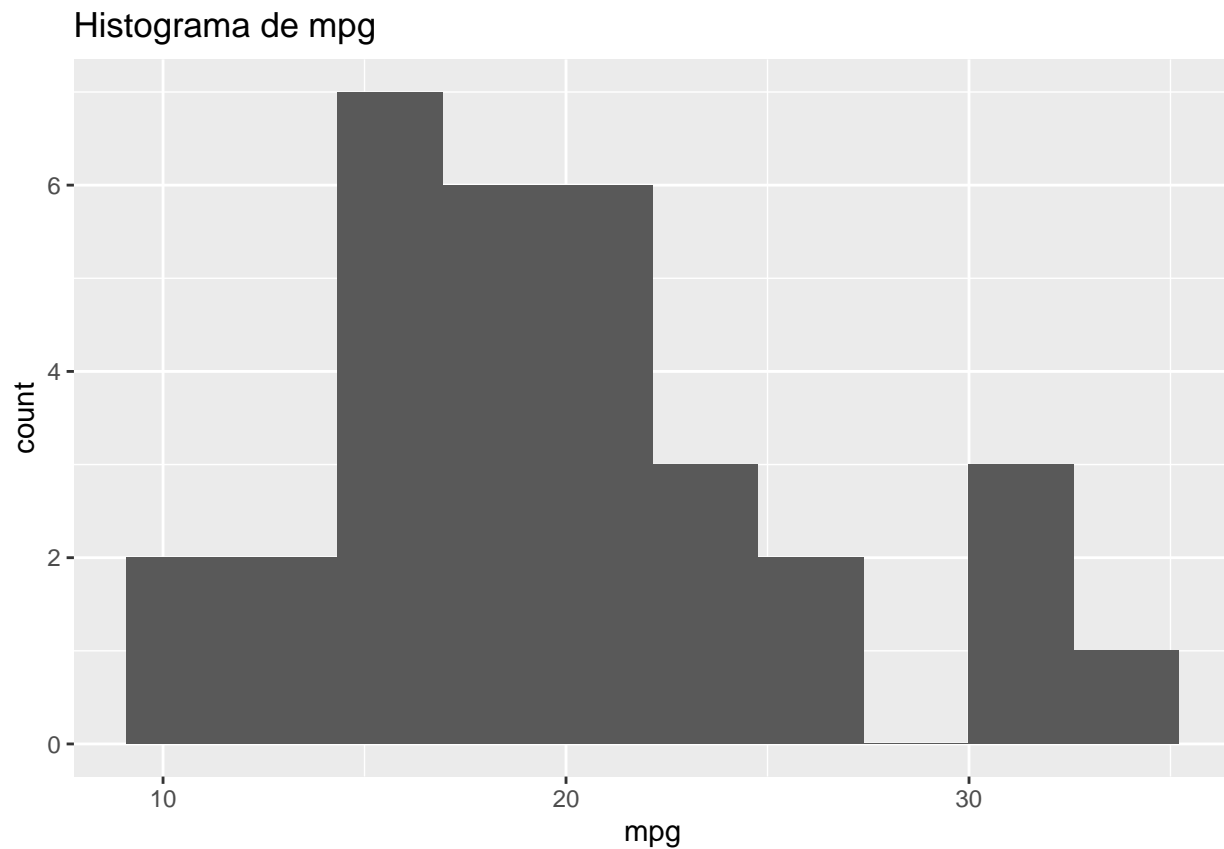
# Exemplo hipotético com dados
dados_pessoas <- tibble(peso = c(70, 80), altura = c(1.75, 1.80))
adicionar_coluna_imc(dados_pessoas, "peso", "altura")
```

```
## # A tibble: 2 x 3
##   peso altura  IMC
##   <dbl> <dbl> <dbl>
## 1    70  1.75  22.9
## 2    80  1.8   24.7
```

Função com ggplot2 para plotar (parte do tidyverse).

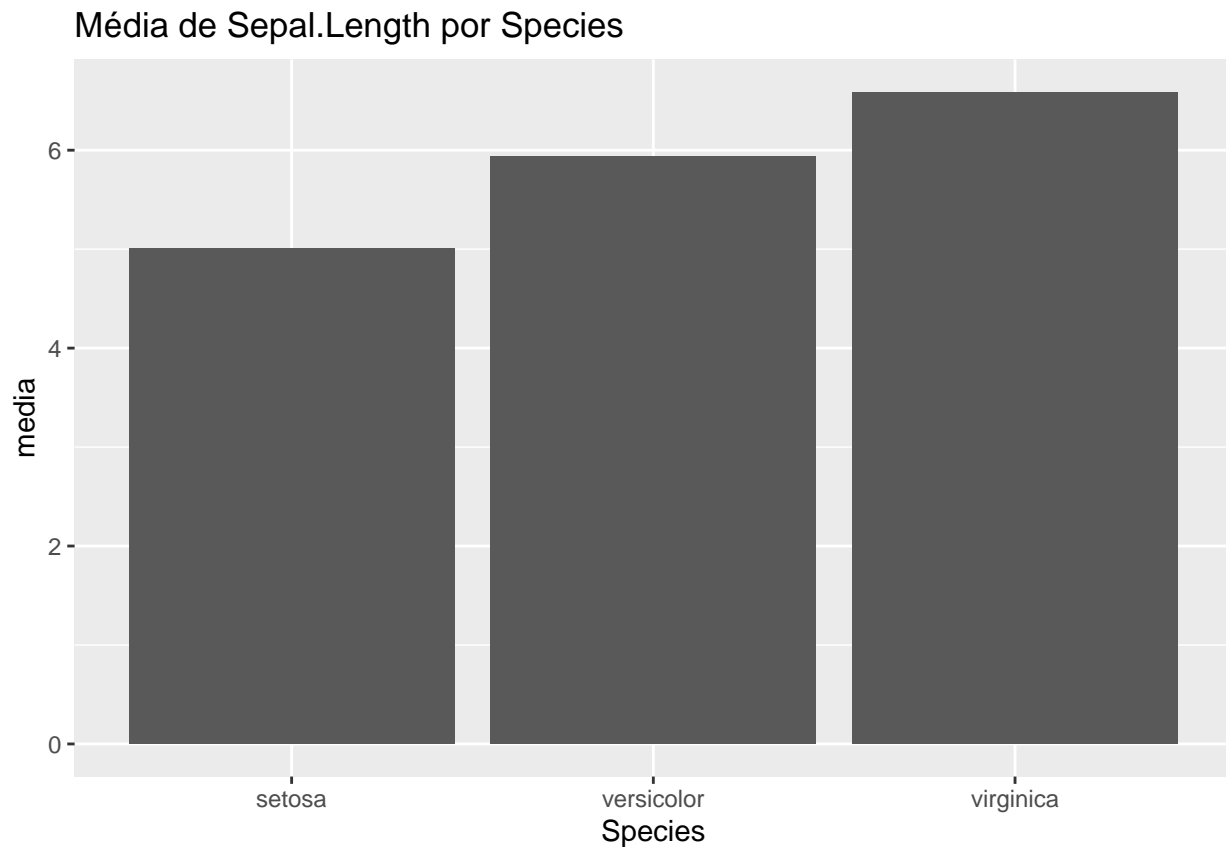
```
plotar_histograma <- function(df, coluna) {
  ggplot(df, aes(x = !!sym(coluna))) +
    geom_histogram(bins = 10) +
    labs(title = paste("Histograma de", coluna))
}

plotar_histograma(mtcars, "mpg")
```



Função que combina dplyr e ggplot2.

```
analisar_e_plotar <- function(df, grupo_col, valor_col) {  
  resumo <- df %>%  
    group_by(!!sym(grupo_col)) %>%  
    summarise(media = mean(!!sym(valor_col)))  
  
  ggplot(resumo, aes(x = !!sym(grupo_col), y = media)) +  
    geom_bar(stat = "identity") +  
    labs(title = paste("Média de", valor_col, "por", grupo_col))  
}  
  
analisar_e_plotar(iris, "Species", "Sepal.Length")
```



Depuração de Funções

Para depurar, use `print()` ou `browser()`.

Exemplo com `print`:

```
funcao_depurada <- function(x) {  
  print(paste("Valor de x:", x))  
  y <- x * 2  
  print(paste("Valor de y:", y))  
  return(y)  
}
```

```
funcao_depurada(3)
```

```
## [1] "Valor de x: 3"  
## [1] "Valor de y: 6"
```

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

Exercício 1

Crie uma função que calcule a média de um vetor numérico. Teste com o vetor `c(1, 2, 3, 4, 5)`.

```
# Sua solução aqui
media_vetor <- function(vetor) {
  # Complete
}
```

Exercício 1.1

Crie uma função recursiva para calcular a soma de 1 a n. Teste com n=10.

```
# Sua solução aqui
soma_recursiva <- function(n) {
  # Complete
}
```

Exercício 1.2

Usando o dataset `airquality`, crie uma função que retorne a média da temperatura (Temp) por mês (Month).

```
data(airquality)
# Sua solução aqui
media_temp_por_mes <- function(df = airquality) {
  # Complete
}
```

Exercício 1.3 (com Tidyverse)

Crie uma função usando `dplyr` para calcular a média e o desvio padrão de `Sepal.Length` por `Species` no dataset `iris`. Retorne um tibble.

```
# Sua solução aqui
estatisticas_iris_tidy <- function(df = iris) {
  # Complete usando group_by e summarise
}
```

Seção 2: Estruturas de Controle

Estruturas de controle são fundamentais na programação, pois permitem que o seu código tome decisões baseadas em condições lógicas. Elas controlam o fluxo de execução do programa, decidindo quais blocos de código serão executados dependendo de se certas condições são verdadeiras ou falsas. Em R, as principais estruturas de controle incluem `if`, `else`, `ifelse`, `switch` e, no contexto do tidyverse, funções como `case_when`. Vamos explorar cada uma delas de forma didática, passo a passo, com explicações claras, exemplos simples e situações mais avançadas. Ao final, discutiremos boas práticas, armadilhas comuns e exercícios para reforçar o aprendizado.

Conceitos Básicos: Condições Lógicas

Antes de mergulharmos nas estruturas, é importante entender as condições lógicas em R. Uma condição é uma expressão que avalia para `TRUE` ou `FALSE`. Operadores comuns incluem: `==` (igual a) - `!=` (diferente

de) - > (maior que), < (menor que), >=, <= - & (E lógico), | (OU lógico), ! (NÃO lógico) - Funções como `is.na()`, `is.null()`, etc.

Exemplo simples de condição:

```
x <- 10
x > 5 # Retorna TRUE
```

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

```
x == 10 & x < 20 # Retorna TRUE
```

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

Essas condições são o coração das estruturas de controle.

If e Else

A estrutura `if` é usada para executar um bloco de código se uma condição for verdadeira. O `else` é opcional e executa um bloco alternativo se a condição for falsa.

Sintaxe básica:

```
if (condicao) {
  # Código se TRUE
} else {
  # Código se FALSE
}
```

Exemplo básico: Verificar se um número é positivo.

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

```
## [1] "x é positivo"
```

Agora, vamos torná-lo mais didático. Imagine que você está analisando dados de temperatura. Se a temperatura for acima de 30 graus, avise que está quente; caso contrário, está normal.

```
temperatura <- 35
if (temperatura > 30) {
  print("Está quente! Use protetor solar.")
} else {
  print("Temperatura normal.")
}
```

```
## [1] "Está quente! Use protetor solar."
```

If-Else Aninhados (Nested If) Você pode aninhar `if` dentro de outros `if` para condições mais complexas. Isso é útil para múltiplos cenários.

Exemplo: Classificar um número como positivo, negativo ou zero.

```
numero <- -5
if (numero > 0) {
  print("Positivo")
} else {
  if (numero < 0) {
    print("Negativo")
  } else {
    print("Zero")
  }
}
```

```
## [1] "Negativo"
```

Else If Para evitar aninhamentos profundos, use `else if` para condições sequenciais.

Exemplo: Classificar notas de alunos.

```
nota <- 85
if (nota >= 90) {
  print("Aprovado com distinção")
} else if (nota >= 70) {
  print("Aprovado")
} else if (nota >= 50) {
  print("Recuperação")
} else {
  print("Reprovado")
}
```

```
## [1] "Aprovado"
```

Situação prática com dataset: Usando `mtcars`, verifique se a média de `mpg` é alta ou baixa.

```
media_mpg <- mean(mtcars$mpg)
if (media_mpg > 25) {
  print("Eficiência alta")
} else if (media_mpg > 20) {
  print("Eficiência média")
} else {
  print("Eficiência baixa")
}
```

```
## [1] "Eficiência média"
```

Ifelse

O `ifelse` é uma versão vetorizada do `if-else`, ideal para aplicar condições a vetores inteiros de forma eficiente, sem loops.

Sintaxe:

```
ifelse(condicao, valor_se_TRUE, valor_se_FALSE)
```

Exemplo básico: Classificar números pares e ímpares.

```
numeros <- c(1, 2, 3, 4, 5)
ifelse(numeros %% 2 == 0, "Par", "Ímpar")
```

```
## [1] "Ímpar" "Par" "Ímpar" "Par" "Ímpar"
```

Exemplo mais avançado: Em um data frame, criar uma nova coluna baseada em condição.

```
mtcars$eficiencia <- ifelse(mtcars$mpg > 20, "Alta", "Baixa")
head(mtcars[, c("mpg", "eficiencia")])
```

```
##           mpg eficiencia
## Mazda RX4      21.0      Alta
## Mazda RX4 Wag  21.0      Alta
## Datsun 710     22.8      Alta
## Hornet 4 Drive  21.4      Alta
## Hornet Sportabout 18.7    Baixa
## Valiant        18.1    Baixa
```

Ifelse Aninhado Você pode aninhar `ifelse` para múltiplas condições, similar ao `else if`.

Exemplo: Classificar IMC.

```
imc <- c(18, 22, 27, 32)
ifelse(imc < 18.5, "Abaixo do peso",
      ifelse(imc < 25, "Normal",
            ifelse(imc < 30, "Sobrepeso", "Obesidade"))))
```

```
## [1] "Abaixo do peso" "Normal" "Sobrepeso" "Obesidade"
```

Armadilha comum: `ifelse` retorna um vetor do mesmo comprimento da condição, mas preserva o tipo dos valores. Se os valores TRUE/FALSE forem de tipos diferentes, pode haver coerção.

Switch

O `switch` é útil para selecionar entre múltiplas opções baseadas em um valor exato (como uma string ou número inteiro). É mais eficiente que múltiplos `if-else` para casos enumerados.

Sintaxe:

```
switch(expressao,
      valor1 = codigo1,
      valor2 = codigo2,
      ...,
      codigo_default)
```

Exemplo básico: Dias da semana.

```

dia <- "segunda"
switch(dia,
  "segunda" = "Início da semana",
  "terca" = "Dia de trabalho",
  "quarta" = "Meio da semana",
  "quinta" = "Quase fim",
  "sexta" = "Fim de semana chegando",
  "sabado" = "Descanso",
  "domingo" = "Domingo",
  "Dia desconhecido")

```

```
## [1] "Início da semana"
```

Exemplo numérico: `switch` também funciona com números (convertidos para string internamente).

```

opcao <- 2
switch(opcao,
  "1" = "Opção 1 selecionada",
  "2" = "Opção 2 selecionada",
  "Opção inválida")

```

```
## [1] "Opção 2 selecionada"
```

Situação prática: Em análise de dados, mapear códigos para descrições.

```

codigo_cor <- "azul"
switch(codigo_cor,
  "vermelho" = "Cor quente",
  "azul" = "Cor fria",
  "verde" = "Cor neutra",
  "Cor desconhecida")

```

```
## [1] "Cor fria"
```

Armadilha comum: `switch` com números requer as chaves como strings (“1”, “2”, etc.).

Estruturas de Controle no Tidyverse: `case_when`

No tidyverse (especificamente no pacote `dplyr`), `case_when` é uma alternativa poderosa ao `ifelse` aninhado, mais legível para condições complexas. É vetorizado e integra-se bem com pipes (`%>%`).

Exemplo básico:

```

numeros <- c(1, 2, 3, 4, 5)
case_when(
  numeros %% 2 == 0 ~ "Par",
  TRUE ~ "Ímpar" # Condição default
)

```

```
## [1] "Ímpar" "Par" "Ímpar" "Par" "Ímpar"
```

Exemplo em data frame: Adicionar coluna de classificação no `iris`.

```
iris %>%
  mutate(classificacao = case_when(
    Sepal.Length > 6 ~ "Longa",
    Sepal.Length > 5 ~ "Média",
    TRUE ~ "Curta"
  )) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species classificacao
## 1         5.1         3.5         1.4         0.2   setosa         Média
## 2         4.9         3.0         1.4         0.2   setosa         Curta
## 3         4.7         3.2         1.3         0.2   setosa         Curta
## 4         4.6         3.1         1.5         0.2   setosa         Curta
## 5         5.0         3.6         1.4         0.2   setosa         Curta
## 6         5.4         3.9         1.7         0.4   setosa         Média
```

Situação avançada: Múltiplas condições.

```
df <- tibble(idade = c(15, 25, 45, 70))
df %>%
  mutate(faixa_etaria = case_when(
    idade < 18 ~ "Menor de idade",
    idade < 30 ~ "Jovem adulto",
    idade < 60 ~ "Adulto",
    TRUE ~ "Idoso"
  ))
```

```
## # A tibble: 4 x 2
##   idade faixa_etaria
##   <dbl> <chr>
## 1    15 Menor de idade
## 2    25 Jovem adulto
## 3    45 Adulto
## 4    70 Idoso
```

Vantagens do `case_when`: Mais legível que `ifelse` aninhado, suporta condições complexas e integra com manipulação de dados.

Combinando Estruturas de Controle com Funções

Estruturas de controle são frequentemente usadas dentro de funções para lógica condicional.

Exemplo: Função que classifica um valor.

```
classificar_numero <- function(x) {
  if (x > 0) {
    return("Positivo")
  } else if (x < 0) {
    return("Negativo")
  } else {
    return("Zero")
  }
}
```

```
}

classificar_numero(0)
```

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

Boas Práticas e Armadilhas Comuns

- **Legibilidade:** Use chaves {} sempre, mesmo para blocos de uma linha, para evitar erros.
- **Evite aninhamentos profundos:** Prefira `else if` ou `switch` para simplificar.
- **Vetorização:** Use `ifelse` ou `case_when` em vez de loops com `if` para eficiência em vetores.
- **Teste condições:** Sempre verifique tipos de dados (ex: numérico vs. string).
- **Erros comuns:** Esquecer o `else` quando necessário, ou condições que não cobrem todos os casos.

Exercício 2

Escreva um código que verifique se um número é positivo, negativo ou zero usando `if-else`.

```
# Sua solução aqui
numero <- -3
if (...) {
  # Complete
}
```

Exercício 2.1

Usando `ifelse`, crie um vetor que classifique temperaturas como “Frio” (< 10), “Agradável” (10-25), “Quente” (>25). Teste com `c(5, 15, 30)`.

```
# Sua solução aqui
temps <- c(5, 15, 30)
classificacao <- ifelse(...)
```

Exercício 2.2

Crie uma função usando `switch` que retorne o nome do mês dado um número (1-12). Inclua um default para valores inválidos.

```
# Sua solução aqui
nome_mes <- function(mes) {
  switch(mes, ...)
}
```

Exercício 2.3 (com Tidyverse)

No dataset `mtcars`, use `mutate` e `case_when` para adicionar uma coluna `potencia` classificando `hp` como “Baixa” (<100), “Média” (100-200), “Alta” (>200).

```
# Sua solução aqui
mtcars %>%
  mutate(potencia = case_when(...))
```

Seção 3: Estruturas de Repetição

Estruturas de repetição, também conhecidas como loops, são essenciais na programação para executar um bloco de código várias vezes, automatizando tarefas repetitivas. Em R, os loops principais são **for**, **while** e **repeat**. Embora R seja otimizado para operações vetorizadas (que são mais eficientes), loops são úteis para cenários onde a vetorização não é direta, como simulações, iterações condicionais ou construção incremental de objetos. Vamos explorar cada um de forma didática, passo a passo, com explicações claras, exemplos básicos e avançados, incluindo usos com data frames, tibbles e outras estruturas de dados. Discutiremos também controles como **break** e **next**, alternativas vetorizadas, boas práticas, armadilhas comuns e exercícios.

Conceitos Básicos: Por Que Usar Loops?

Loops permitem repetir ações sem duplicar código. Por exemplo, em vez de somar números manualmente, um loop pode iterar sobre uma sequência. No entanto, em R, prefira operações vetorizadas (ex: `sum(1:10)`) para performance, especialmente com grandes dados. Use loops quando precisar de controle fino, como em simulações Monte Carlo ou processamento de listas complexas.

For Loop

O **for** itera sobre uma sequência (vetor, lista, etc.), executando o código para cada elemento.

Sintaxe básica:

```
for (variavel in sequencia) {
  # Código a repetir
}
```

Exemplo básico: Imprimir quadrados de 1 a 5.

```
for (i in 1:5) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

Explicação passo a passo: 1. *i* assume o valor 1, executa `print(1^2)` → 1 2. *i* assume 2, executa `print(2^2)` → 4 3. E assim por diante até 5.

Exemplo com vetor: Iterar sobre nomes.

```
nomes <- c("Ana", "João", "Maria")
for (nome in nomes) {
  print(paste("Olá,", nome))
}
```

```
## [1] "Olá, Ana"
## [1] "Olá, João"
## [1] "Olá, Maria"
```

For com Data Frames Data frames são como tabelas. Você pode iterar sobre linhas ou colunas.

Exemplo: Iterar sobre colunas de `mtcars` e imprimir médias.

```
for (col in names(mtcars)) {
  if (is.numeric(mtcars[[col]])) {
    print(paste("Média de", col, ":", mean(mtcars[[col]])))
  }
}
```

```
## [1] "Média de mpg : 20.090625"
## [1] "Média de cyl : 6.1875"
## [1] "Média de disp : 230.721875"
## [1] "Média de hp : 146.6875"
## [1] "Média de drat : 3.5965625"
## [1] "Média de wt : 3.21725"
## [1] "Média de qsec : 17.84875"
## [1] "Média de vs : 0.4375"
## [1] "Média de am : 0.40625"
## [1] "Média de gear : 3.6875"
## [1] "Média de carb : 2.8125"
```

Exemplo: Iterar sobre linhas usando índices.

```
for (i in 1:nrow(mtcars)) {
  if (mtcars$mpg[i] > 20) {
    print(paste("Carro", rownames(mtcars)[i], "tem alta eficiência"))
  }
}
```

```
## [1] "Carro Mazda RX4 tem alta eficiência"
## [1] "Carro Mazda RX4 Wag tem alta eficiência"
## [1] "Carro Datsun 710 tem alta eficiência"
## [1] "Carro Hornet 4 Drive tem alta eficiência"
## [1] "Carro Merc 240D tem alta eficiência"
## [1] "Carro Merc 230 tem alta eficiência"
## [1] "Carro Fiat 128 tem alta eficiência"
## [1] "Carro Honda Civic tem alta eficiência"
## [1] "Carro Toyota Corolla tem alta eficiência"
## [1] "Carro Toyota Corona tem alta eficiência"
## [1] "Carro Fiat X1-9 tem alta eficiência"
## [1] "Carro Porsche 914-2 tem alta eficiência"
## [1] "Carro Lotus Europa tem alta eficiência"
## [1] "Carro Volvo 142E tem alta eficiência"
```

For com Tibbles (Tidyverse) Tibbles são data frames aprimorados. O loop é similar.

Exemplo: Converter `iris` para tibble e iterar.

```
iris_tibble <- as_tibble(iris)
for (i in 1:nrow(iris_tibble)) {
  print(paste("Espécie:", iris_tibble$Species[i], "- Comprimento da sépala:", iris_tibble$Sepal.Length[
}

```

```
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 4.9"
## [1] "Espécie: setosa - Comprimento da sépala: 4.7"
## [1] "Espécie: setosa - Comprimento da sépala: 4.6"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 5.4"
## [1] "Espécie: setosa - Comprimento da sépala: 4.6"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 4.4"
## [1] "Espécie: setosa - Comprimento da sépala: 4.9"
## [1] "Espécie: setosa - Comprimento da sépala: 5.4"
## [1] "Espécie: setosa - Comprimento da sépala: 4.8"
## [1] "Espécie: setosa - Comprimento da sépala: 4.8"
## [1] "Espécie: setosa - Comprimento da sépala: 4.3"
## [1] "Espécie: setosa - Comprimento da sépala: 5.8"
## [1] "Espécie: setosa - Comprimento da sépala: 5.7"
## [1] "Espécie: setosa - Comprimento da sépala: 5.4"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 5.7"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 5.4"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 4.6"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 4.8"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 5.2"
## [1] "Espécie: setosa - Comprimento da sépala: 5.2"
## [1] "Espécie: setosa - Comprimento da sépala: 4.7"
## [1] "Espécie: setosa - Comprimento da sépala: 4.8"
## [1] "Espécie: setosa - Comprimento da sépala: 5.4"
## [1] "Espécie: setosa - Comprimento da sépala: 5.2"
## [1] "Espécie: setosa - Comprimento da sépala: 5.5"
## [1] "Espécie: setosa - Comprimento da sépala: 4.9"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 5.5"
## [1] "Espécie: setosa - Comprimento da sépala: 4.9"
## [1] "Espécie: setosa - Comprimento da sépala: 4.4"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 4.5"
## [1] "Espécie: setosa - Comprimento da sépala: 4.4"
## [1] "Espécie: setosa - Comprimento da sépala: 5"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 4.8"
## [1] "Espécie: setosa - Comprimento da sépala: 5.1"
## [1] "Espécie: setosa - Comprimento da sépala: 4.6"
```

[illegible]

```

## [1] "Espécie: virginica - Comprimento da sépala: 7.1"
## [1] "Espécie: virginica - Comprimento da sépala: 6.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.5"
## [1] "Espécie: virginica - Comprimento da sépala: 7.6"
## [1] "Espécie: virginica - Comprimento da sépala: 4.9"
## [1] "Espécie: virginica - Comprimento da sépala: 7.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.7"
## [1] "Espécie: virginica - Comprimento da sépala: 7.2"
## [1] "Espécie: virginica - Comprimento da sépala: 6.5"
## [1] "Espécie: virginica - Comprimento da sépala: 6.4"
## [1] "Espécie: virginica - Comprimento da sépala: 6.8"
## [1] "Espécie: virginica - Comprimento da sépala: 5.7"
## [1] "Espécie: virginica - Comprimento da sépala: 5.8"
## [1] "Espécie: virginica - Comprimento da sépala: 6.4"
## [1] "Espécie: virginica - Comprimento da sépala: 6.5"
## [1] "Espécie: virginica - Comprimento da sépala: 7.7"
## [1] "Espécie: virginica - Comprimento da sépala: 7.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6"
## [1] "Espécie: virginica - Comprimento da sépala: 6.9"
## [1] "Espécie: virginica - Comprimento da sépala: 5.6"
## [1] "Espécie: virginica - Comprimento da sépala: 7.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.7"
## [1] "Espécie: virginica - Comprimento da sépala: 7.2"
## [1] "Espécie: virginica - Comprimento da sépala: 6.2"
## [1] "Espécie: virginica - Comprimento da sépala: 6.1"
## [1] "Espécie: virginica - Comprimento da sépala: 6.4"
## [1] "Espécie: virginica - Comprimento da sépala: 7.2"
## [1] "Espécie: virginica - Comprimento da sépala: 7.4"
## [1] "Espécie: virginica - Comprimento da sépala: 7.9"
## [1] "Espécie: virginica - Comprimento da sépala: 6.4"
## [1] "Espécie: virginica - Comprimento da sépala: 6.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.1"
## [1] "Espécie: virginica - Comprimento da sépala: 7.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.4"
## [1] "Espécie: virginica - Comprimento da sépala: 6"
## [1] "Espécie: virginica - Comprimento da sépala: 6.9"
## [1] "Espécie: virginica - Comprimento da sépala: 6.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6.9"
## [1] "Espécie: virginica - Comprimento da sépala: 5.8"
## [1] "Espécie: virginica - Comprimento da sépala: 6.8"
## [1] "Espécie: virginica - Comprimento da sépala: 6.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6.7"
## [1] "Espécie: virginica - Comprimento da sépala: 6.3"
## [1] "Espécie: virginica - Comprimento da sépala: 6.5"
## [1] "Espécie: virginica - Comprimento da sépala: 6.2"
## [1] "Espécie: virginica - Comprimento da sépala: 5.9"

```

For com Listas Listas podem conter elementos heterogêneos.

Exemplo: Iterar sobre uma lista.

```

minha_lista <- list(a = 1:3, b = c("x", "y"), c = mtcars[1:2,])
for (elem in minha_lista) {
  print(summary(elem))
}

```

```

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      1.0    1.5    2.0    2.0    2.5    3.0
##      Length      Class    Mode
##      2 character character
##      mpg      cyl      disp      hp      drat
## Min.   :21  Min.   :6  Min.   :160  Min.   :110  Min.   :3.9
## 1st Qu.:21  1st Qu.:6  1st Qu.:160  1st Qu.:110  1st Qu.:3.9
## Median :21  Median :6  Median :160  Median :110  Median :3.9
## Mean   :21  Mean   :6  Mean   :160  Mean   :110  Mean   :3.9
## 3rd Qu.:21  3rd Qu.:6  3rd Qu.:160  3rd Qu.:110  3rd Qu.:3.9
## Max.   :21  Max.   :6  Max.   :160  Max.   :110  Max.   :3.9
##      wt      qsec      vs      am      gear
## Min.   :2.620  Min.   :16.46  Min.   :0  Min.   :1  Min.   :4
## 1st Qu.:2.684  1st Qu.:16.60  1st Qu.:0  1st Qu.:1  1st Qu.:4
## Median :2.748  Median :16.74  Median :0  Median :1  Median :4
## Mean   :2.748  Mean   :16.74  Mean   :0  Mean   :1  Mean   :4
## 3rd Qu.:2.811  3rd Qu.:16.88  3rd Qu.:0  3rd Qu.:1  3rd Qu.:4
## Max.   :2.875  Max.   :17.02  Max.   :0  Max.   :1  Max.   :4
##      carb      eficiencia
## Min.   :4  Length:2
## 1st Qu.:4  Class :character
## Median :4  Mode  :character
## Mean   :4
## 3rd Qu.:4
## Max.   :4

```

Loops Aninhados (Nested For) Loops dentro de loops para iterações multidimensionais.

Exemplo: Matriz de multiplicação.

```

for (i in 1:3) {
  for (j in 1:3) {
    print(paste(i, "*", j, "=", i*j))
  }
}

```

```

## [1] "1 * 1 = 1"
## [1] "1 * 2 = 2"
## [1] "1 * 3 = 3"
## [1] "2 * 1 = 2"
## [1] "2 * 2 = 4"
## [1] "2 * 3 = 6"
## [1] "3 * 1 = 3"
## [1] "3 * 2 = 6"
## [1] "3 * 3 = 9"

```

Situação prática: Simular dados em um data frame.

```
df_simulado <- data.frame()
for (grupo in 1:3) {
  for (rep in 1:5) {
    valor <- rnorm(1, mean = grupo * 10)
    df_simulado <- rbind(df_simulado, data.frame(Grupo = grupo, Rep = rep, Valor = valor))
  }
}
head(df_simulado)
```

```
##   Grupo Rep      Valor
## 1     1   1 10.574779
## 2     1   2 10.488892
## 3     1   3 11.678288
## 4     1   4 11.711327
## 5     1   5  9.933634
## 6     2   1 20.049788
```

Aqui usamos `rbind` para adicionar linhas incrementalmente.

Usando `rbind` e `cbind` em Loops

- `rbind`: Adiciona linhas a um data frame.
- `cbind`: Adiciona colunas.

Exemplo com `rbind`: Construir data frame de simulações.

```
df <- data.frame()
for (i in 1:5) {
  nova_linha <- data.frame(ID = i, Valor = rnorm(1))
  df <- rbind(df, nova_linha)
}
df
```

```
##   ID      Valor
## 1  1  0.4403330
## 2  2  0.6820820
## 3  3 -1.6632841
## 4  4  0.3114421
## 5  5 -0.5003281
```

Exemplo com `cbind`: Adicionar colunas calculadas.

```
df_base <- data.frame(A = 1:3)
for (i in 1:2) {
  nova_col <- df_base$A * i
  df_base <- cbind(df_base, nova_col)
  colnames(df_base)[ncol(df_base)] <- paste("Col", i)
}
df_base
```

```
##   A Col 1 Col 2
## 1 1     1     2
## 2 2     2     4
## 3 3     3     6
```

Armadilha: Loops com `rbind/cbind` podem ser ineficientes para grandes dados, pois recriam o objeto a cada iteração. Pré-aloque espaço quando possível (ex: inicialize com NA).

While Loop

O `while` repete enquanto uma condição for verdadeira. Útil quando o número de iterações é desconhecido.

Sintaxe:

```
while (condicao) {
  # Código
  # Atualize a condição!
}
```

Exemplo básico: Contar até 5.

```
contador <- 1
while (contador <= 5) {
  print(contador)
  contador <- contador + 1
}
```

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

Explicação: O loop verifica `contador <= 5`; se TRUE, executa e incrementa. Quando FALSE, para.

Exemplo avançado: Simular até atingir um valor.

```
soma <- 0
i <- 1
while (soma < 20) {
  soma <- soma + i
  i <- i + 1
  print(paste("Soma atual:", soma))
}
```

```
## [1] "Soma atual: 1"
## [1] "Soma atual: 3"
## [1] "Soma atual: 6"
## [1] "Soma atual: 10"
## [1] "Soma atual: 15"
## [1] "Soma atual: 21"
```

While com Data Frames Exemplo: Adicionar linhas até um critério.

```
df_while <- data.frame(Valor = numeric())
soma_valores <- 0
while (soma_valores < 10) {
  novo_valor <- runif(1, 0, 3)
  df_while <- rbind(df_while, data.frame(Valor = novo_valor))
  soma_valores <- sum(df_while$Valor)
}
df_while
```

```
##      Valor
## 1 2.4464520
## 2 0.9922829
## 3 0.4895978
## 4 1.8185087
## 5 1.9203373
## 6 1.2199604
## 7 1.9643386
```

Repeat Loop

O `repeat` cria um loop infinito até um `break`. Útil para condições complexas de saída.

Sintaxe:

```
repeat {
  # Código
  if (condicao_de_saida) break
}
```

Exemplo básico:

```
contador <- 1
repeat {
  print(contador)
  contador <- contador + 1
  if (contador > 5) break
}
```

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

Exemplo avançado: Loop até convergência (ex: aproximação de raiz).

```
x <- 2 # Aproximação inicial para sqrt(2)
repeat {
  x_novo <- (x + 2/x) / 2
  if (abs(x_novo - x) < 0.001) break
}
```

```
x <- x_novo
print(x)
}
```

```
## [1] 1.5
## [1] 1.416667
## [1] 1.414216
```

Repeat com Estruturas de Dados Exemplo: Construir lista até tamanho desejado.

```
minha_lista_repeat <- list()
i <- 1
repeat {
  minha_lista_repeat[[i]] <- rnorm(3)
  i <- i + 1
  if (i > 4) break
}
minha_lista_repeat
```

```
## [[1]]
## [1] -1.87629438  0.38852502 -0.01596367
##
## [[2]]
## [1]  0.7106183 -1.3846310  1.2985303
##
## [[3]]
## [1] -0.6757822  0.6360171  0.6122234
##
## [[4]]
## [1] -0.60144340 -0.08162777  0.64750488
```

Break e Next

- **break**: Sai do loop imediatamente.
- **next**: Pula para a próxima iteração, ignorando o resto do código atual.

Exemplo com next: Pular números pares.

```
for (i in 1:10) {
  if (i %% 2 == 0) next
  print(i) # Imprime ímpares
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
```

Exemplo com break: Parar ao encontrar valor.

```

numeros <- c(1, 3, 5, 7, 2, 4)
for (num in numeros) {
  if (num %% 2 == 0) {
    print("Encontrado par! Parando.")
    break
  }
  print(paste("Ímpar:", num))
}

```

```

## [1] "Ímpar: 1"
## [1] "Ímpar: 3"
## [1] "Ímpar: 5"
## [1] "Ímpar: 7"
## [1] "Encontrado par! Parando."

```

Alternativas Vetorizadas: Família Apply e Purrr (Tidyverse)

Loops podem ser lentos em R; prefira alternativas.

Família Apply (Base R)

- `lapply`: Aplica função a lista/vetor, retorna lista.
- `sapply`: Similar, retorna vetor/matriz.
- `apply`: Para matrizes/data frames.

Exemplo com `sapply`:

```

vetor <- 1:5
sapply(vetor, function(x) x^2)

```

```
## [1] 1 4 9 16 25
```

Exemplo com `apply` em data frame:

```

apply(mtcars[, 1:4], 2, mean) # Média por coluna

```

```

##      mpg      cyl      disp      hp
## 20.09062  6.18750 230.72188 146.68750

```

Purrr (Tidyverse) Mais consistente e legível.

Exemplo com `map`:

```

listas <- list(a = 1:3, b = 4:6)
map(listas, sum)

```

```

## $a
## [1] 6
##
## $b
## [1] 15

```

Exemplo com `map_dbl`:

```
map_dbl(mtcars[, 1:4], mean)
```

```
##      mpg      cyl    disp     hp  
## 20.09062  6.18750 230.72188 146.68750
```

Vantagem: Evita loops para eficiência.

Combinando Loops com Funções

Loops dentro de funções para tarefas repetitivas.

Exemplo: Função que simula dados.

```
simular_dados <- function(n) {  
  df <- data.frame()  
  for (i in 1:n) {  
    df <- rbind(df, data.frame(Valor = rnorm(1)))  
  }  
  return(df)  
}  
  
simular_dados(3)
```

```
##      Valor  
## 1 -0.1910515  
## 2  0.3610293  
## 3  0.8453001
```

Boas Práticas e Armadilhas Comuns

- **Eficiência:** Evite loops para operações vetorizáveis. Use `apply` ou `purrr`.
- **Pré-alocação:** Para `rbind/cbind`, inicialize objetos grandes para evitar realocações.
- **Depuração:** Use `print()` dentro de loops para rastrear.
- **Erros comuns:** Esquecer de atualizar contadores em `while` (loop infinito); não usar `next/break` quando necessário.
- **Tidyverse:** Prefira `map` e pipes para código limpo.

Exercício 3

Use um `for` loop para somar os números de 1 a 10.

```
# Sua solução aqui  
soma <- 0  
for (...) {  
  # Complete  
}  
print(soma)
```

Exercício 3.1

Use um `while` loop para encontrar o primeiro número cujo quadrado exceda 100, começando de 1.

```
# Sua solução aqui
i <- 1
while (...) {
  # Complete
}
```

Exercício 3.2

Crie um data frame vazio e use um `for` loop com `rbind` para adicionar 5 linhas com valores aleatórios (use `rnorm`).

```
# Sua solução aqui
df <- data.frame()
for (...) {
  # Complete
}
```

Exercício 3.3 (com Tidyverse)

Use `map` do `purrr` para aplicar uma função (ex: quadrado) a uma lista de vetores.

```
# Sua solução aqui
minha_lista <- list(1:3, 4:6)
map(minha_lista, ...)
```

Exercício 3.4

Use um `for` loop para iterar sobre as linhas de `iris` e imprimir apenas as da espécie “setosa”.

```
# Sua solução aqui
for (i in 1:nrow(iris)) {
  # Complete
}
```

Conclusão

Parabéns! Você concluiu o curso básico sobre funções, estruturas de controle e repetição em R. Pratique os exercícios e experimente criar suas próprias funções e loops.

Para mais recursos: - Documentação oficial do R: cran.r-project.org - Livro “R for Data Science” de Hadley Wickham.

Se precisar de mais ajuda, pergunte!