Introdução Programação em R com GitHub, ChatGPT e Claude

Vinícius Silva Junqueira

2025-10-08

Sumário

1 Dia 2 — Lógica, Funções e Introdução ao Tidyverse **1**
1.1 1. Operadores e Condicionais (30 min) 1
1.2 2. Loops, Vetorização e Funções (25 min) 3
1.3 3. Introdução ao Tidyverse (45 min) 5
1.4 4. Datas com `lubridate` (10 min) 6
1.5 5. Exercícios Práticos (20–25 min) 6
1.6 6. Boas Práticas e Debugging (20 min) 7
1.7 7. Commit do Dia 8
1.8 8. Checklist de encerramento 8
1.9 9. Referências rápidas 8

1 Dia 2 — Lógica, Funções e Introdução ao Tidyverse

Objetivos do dia

- Dominar operadores lógicos/relacionais e condicionais (`if`, `ifelse`, `case_when`).
- Entender loops vs. vetorização e criar **funções próprias**.
- Aplicar um **pipeline básico** com `dplyr` e introduzir **datas** com `lubridate`.
- Registrar o aprendizado com um commit no seu fork no GitHub.

Tempo previsto 19h00–22h00 (intervalo 20h30–20h50)

1.1 1. Operadores e Condicionais (30 min)

1.1.1 O que são operadores?

Operadores são símbolos especiais que realizam operações entre valores. Eles são fundamentais para tomar decisões no código e controlar o fluxo de execução.

1.1.2 1.1 Operadores lógicos e relacionais

Operadores relacionais comparam dois valores e retornam `TRUE` ou `FALSE`:

- `==` : igual a
- `!=` : diferente de

- `>` : maior que
- `<` : menor que
- `>=` : maior ou igual
- `<=` : menor ou igual

Operadores lógicos combinam condições:

- `&` : E (AND) - ambas condições devem ser verdadeiras
- `|` : OU (OR) - pelo menos uma condição deve ser verdadeira
- `!` : NÃO (NOT) - inverte o valor lógico
- `xor()` : OU EXCLUSIVO - apenas uma condição pode ser verdadeira

Operador especial: - `%in%` : verifica se um valor está presente em um vetor

```r
# Lógicos: & | ! xor()
TRUE & FALSE    # FALSE (ambos precisam ser TRUE)
TRUE | FALSE    # TRUE (pelo menos um é TRUE)
!TRUE           # FALSE (inverte)
xor(TRUE, FALSE) # TRUE (apenas um é TRUE)

# Relacionais: == != > < >= <=
3 == 3          # TRUE (igual)
5 != 2          # TRUE (diferente)
5 > 2; 1 < 0    # TRUE; FALSE
2 >= 2; 3 <= 10 # TRUE; TRUE

# %in% (teste de pertinência)
2 %in% c(1, 2, 3)                          # TRUE
"Adelie" %in% c("Chinstrap", "Gentoo")    # FALSE
```

1.1.3 1.2 Condicionais: tomando decisões no código

Condicionais permitem que seu código tome decisões baseadas em condições. São como perguntas "se... então... senão...".

Três formas principais:

1. **`if/else`** - Estrutura clássica para **um único valor**
   - Avalia uma condição e executa diferentes blocos de código
   - Útil para controle de fluxo em funções
2. **`ifelse()`** - Versão **vetorizada** para múltiplos valores
   - Aplica a condição a cada elemento de um vetor
   - Retorna um vetor de resultados
   - Ideal para criar novas colunas em data.frames
3. **`case_when()`** - Para **múltiplas condições** complexas
   - Avalia várias regras em sequência
   - Para na primeira regra verdadeira
   - Mais legível que `ifelse()` aninhados

```r
# if/else (escalar - um valor por vez)
x <- 18
if (x >= 18) {
  status <- "maior_de_idade"
} else {
  status <- "menor_de_idade"
}
status

# ifelse() (vetorizado - múltiplos valores)
notas <- c(5.9, 7.5, 9.2, 6.0)
resultado <- ifelse(notas >= 7, "Aprovado", "Recuperação")
resultado

# case_when() (múltiplas regras em ordem)
library(dplyr)
faixa <- case_when(
  notas >= 9             ~ "Excelente",
  notas >= 7 & notas < 9 ~ "Bom",
  notas >= 5 & notas < 7 ~ "Regular",
  TRUE                   ~ "Insuficiente"  # TRUE = "caso contrário"
)
faixa
```

> **Dica didática**: use `ifelse()` quando quiser **vetorizar**; `case_when()` quando houver **várias regras**.

1.2 2. Loops, Vetorização e Funções (25 min)

1.2.1 2.1 Loops vs. operações vetorizadas

O que são loops?

Um **loop** (laço) é uma estrutura que repete um bloco de código várias vezes. O loop `for` é o mais comum e executa o código uma vez para cada elemento de uma sequência.

Por que evitar loops em R?

R é uma linguagem **vetorizada**, o que significa que muitas operações funcionam automaticamente em vetores inteiros, sem precisar de loops explícitos. Operações vetorizadas são: - **Mais rápidas** (otimizadas internamente em C/Fortran) - **Mais legíveis** (menos linhas de código) - **Mais idiomáticas** (o "jeito R" de fazer)

Quando usar loops: - Quando não existe alternativa vetorizada - Para operações que dependem de iterações anteriores - Em simulações e processos iterativos

```r
valores <- 1:5

# Loop for (didático, mas não idiomático)
```

```r
soma <- 0
for (v in valores) {
  soma <- soma + v
}
soma

# Vetorizado (preferido em R!)
sum(valores)  # Muito mais simples e rápido
```

1.2.2 2.2 Funções: empacotando lógica reutilizável

O que são funções?

Funções são blocos de código que realizam uma tarefa específica e podem ser reutilizados. São fundamentais para: - **Organizar** código em partes lógicas - **Reutilizar** lógica sem repetir código - **Documentar** intenções através de nomes descritivos - **Facilitar** manutenção e debugging

Estrutura de uma função:

```r
nome_funcao <- function(argumento1, argumento2 = valor_padrao) {
  # corpo da função
  resultado <- alguma_operacao
  return(resultado)  # return é opcional (retorna última expressão)
}
```

Boas práticas: - Use nomes descritivos que indiquem o que a função faz - Valide entradas com `stop()`, `stopifnot()` ou `if` - Documente com comentários o que a função faz e quais são os argumentos - Retorne sempre o mesmo tipo de objeto

Exemplo prático: calculadora de IMC

```r
# Fórmula: IMC = peso(kg) / altura(m)^2
imc <- function(peso, altura) {
  # Validação: altura não pode ser zero ou negativa
  if (any(altura <= 0)) stop("Altura deve ser > 0")

  # Cálculo vetorizado (funciona com um ou vários valores)
  peso / (altura ^ 2)
}

# Testando com múltiplos valores
imc(c(70, 80), c(1.70, 1.80))

# Função para classificar IMC usando case_when()
classificar_imc <- function(imc) {
  dplyr::case_when(
    imc < 18.5              ~ "Abaixo do peso",
    imc >= 18.5 & imc < 25  ~ "Normal",
    imc >= 25   & imc < 30  ~ "Sobrepeso",
    imc >= 30               ~ "Obesidade"
```

```
  )
}

# Combinando as duas funções
val <- imc(80, 1.75)
classificar_imc(val)
```

> **Princípio DRY** (Don't Repeat Yourself): se você copiou e colou código mais de 2 vezes, provavelmente deveria criar uma função!

1.3 3. Introdução ao Tidyverse (45 min)

Vamos aplicar `dplyr` no dataset **palmerpenguins** e criar um pequeno pipeline.

```
library(dplyr)
library(palmerpenguins)

# Remover linhas com NAs nas colunas essenciais
peng <- penguins |>
  filter(!is.na(species),
         !is.na(bill_length_mm),
         !is.na(bill_depth_mm),
         !is.na(flipper_length_mm),
         !is.na(body_mass_g))

# Selecionar só o que precisamos
peng_sel <- peng |>
  select(species, island, bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g)

# Criar nova variável (razão do bico) e reordenar
peng_feat <- peng_sel |>
  mutate(raz_bico = bill_length_mm / bill_depth_mm) |>
  arrange(species, desc(raz_bico))

# Resumo por espécie
resumo <- peng_feat |>
  group_by(species) |>
  summarize(
    n = n(),
    media_flipper = mean(flipper_length_mm),
    sd_flipper    = sd(flipper_length_mm),
    media_massa   = mean(body_mass_g)
  )
resumo
```

1.3.1 3.1 Pipe: %>% vs. |>

```
# Ambos funcionam; escolha um padrão para a turma.
# Exemplo com |> (pipe nativo do R >= 4.1):
penguins |>
  tidyr::drop_na(bill_length_mm) |>
  dplyr::summarize(media = mean(bill_length_mm))
```

1.4 4. Datas com `lubridate` (10 min)

Datas aparecem em **quase todos** os projetos. Vamos ilustrar rapidamente.

```
library(lubridate)

# Criação e parsing
ymd("2025-11-18")
dmy("18/11/2025")
mdy("11-18-2025")

# Componentes
hoje <- today()
ano(hoje); mes(hoje); wday(hoje, label = TRUE, abbr = FALSE)

# Operações simples
hoje + days(14)
interval(ymd("2025-11-01"), ymd("2025-11-18"))
```

Integrando no pipeline: quando houver colunas de data, transforme-as e derive **mês/ano** para agregações.

1.5 5. Exercícios Práticos (20–25 min)

Dataset: `palmerpenguins::penguins`

1.5.1 Exercício 1 — Condicionais

1. Crie um vetor de 8 notas qualquer.
2. Classifique com `ifelse()` como **Aprovado/Recuperação** (corte em 7).
3. Depois, crie uma classificação mais rica usando `case_when()` com 4 faixas.

```
# Seu código aqui
```

1.5.2 Exercício 2 — Funções

1. Escreva uma função `zscore(x)` que centraliza e escala (média 0, desvio 1).
2. Aplique em `bill_length_mm` **removendo NAs** antes.
3. Faça um segundo argumento opcional `na_rm = TRUE` dentro da função.

```
# Seu código aqui
```

1.5.3 Exercício 3 — Pipeline `dplyr`

1. Crie `peng3` filtrando linhas completas nas 4 medidas principais.
2. Calcule, por espécie, média e desvio da nadadeira (`flipper_length_mm`).
3. Ordene do maior para o menor e mostre as 5 primeiras linhas.

```
# Seu código aqui
```

1.5.4 Exercício 4 — Datas com `lubridate`

1. Crie um vetor com 5 datas em formato "dd/mm/aaaa".
2. Converta com `dmy()` e extraia `month()` (com rótulo).
3. Some 30 dias à primeira data e compute o intervalo até a última.

```
# Seu código aqui
```

1.6 6. Boas Práticas e Debugging (20 min)

- Use **nomes descritivos** em `snake_case`.
- Comente o **porquê** (não só o que) no código.
- Valide entradas em funções (`stop()` para erros previsíveis).
- Leia mensagens de erro **de baixo para cima** (stack trace).
- Mantenha scripts curtos e reutilizáveis.

1.6.1 Ferramentas úteis

```
# message(), warning(), stop() para sinalizar eventos
# browser() para inspecionar dentro de uma função (quando eval=TRUE)
# traceback() após um erro
```

> **IA como apoio (responsável):** use ChatGPT/Claude para **explicar erros** e sugerir melhorias, mas sempre **entenda e teste** o código.

1.7 7. Commit do Dia

1. Salve como `scripts/02_logica_funcoes.R` ou `materiais/dia2_logica_funcoes.Rmd` (este arquivo).

2. No **Terminal do RStudio**:

```
git add scripts/02_logica_funcoes.R
git commit -m "Dia 2: lógica, funções e tidyverse (com lubridate)"
git push origin main
```

> Lembre-se: você está trabalhando **no SEU fork**. O repositório original permanece protegido.

1.8 8. Checklist de encerramento

- ☐ Dominou operadores lógicos e relacionais
- ☐ Entendeu diferenças entre `if/else`, `ifelse()` e `case_when()`
- ☐ Compreendeu por que vetorização é preferível a loops
- ☐ Criou suas primeiras funções com validação
- ☐ Aplicou pipeline básico com `dplyr`
- ☐ Explorou manipulação de datas com `lubridate`
- ☐ Realizou commit e push no seu fork

1.9 9. Referências rápidas

- **`dplyr` cheatsheet**: https://posit.co/resources/cheatsheets/
- **R for Data Science (2e)**: https://r4ds.hadley.nz/
- **Happy Git with R**: https://happygitwithr.com/
- **palmerpenguins**: https://allisonhorst.github.io/palmerpenguins/
- **lubridate**: https://lubridate.tidyverse.org/

Nos vemos no Dia 3 para transformação de dados e visualização com ggplot2!