R para Ciência de Dados 2

Lubridate + Forcats

agosto de 2021

Lubridate

Motivação

É difícil encontrar um tipo de dado mais delicado do que datas (e horas): diferentemente de textos e erros de *encoding*, erros de *locale* podem passar desapercebidos e estragar uma análise inteira.

Operações com tempo são complicadas, pois envolvem precisão e diversos fatores que variam de um lugar para o outro (fuso horário, horário de verão, anos bissextos, formato da data, etc.).

Além das variações normais de como cada país escreve suas datas, cada computador tem seu jeito de interpretá-las e cada programa tem seu jeito de salvá-las.

Entender como é a representação de tempo dentro de linguagens de programação é muito valioso porque isso é um problema relevante independentemente da ferramenta sendo utilizada.

Introdução

- Como representar datas em um universo cheio de fusos e calendários diferentes? Estabelecendo um momento universal e contando os segundos que se passaram desde lá

```
library(lubridate)
now()
```

```
#> [1] "2021-08-19 19:04:15 -03"
```

```
as.numeric(now())
```

```
#> [1] 1629410656
```

- O formato padrão é denominado "Era UNIX" e conta o número de segundos desde o ano novo de 1970 em Londres (01/01/1970 00:00:00 UTC)

O pacote {lubridate}

- O pacote `{lubridate}` vai nos possibilitar trabalhar com datas e data-horas fora do ISO 8601 (`ano-mês-dia hora:minuto:segundo`)
- Para converter uma data-hora do formato brasileiro para o padrão universal, pensamos na ordem das unidades em inglês: *day*, *month*, *year*, *hour*, *minute*, *second*

```
dmy_hms("15/04/2021 02:25:00")
```

```
#> [1] "2021-04-15 02:25:00 UTC"
```

- Também é possível trabalhar só com datas usando a mesma lógica das unidades

```
dmy("15/04/2021")
```

```
#> [1] "2021-04-15"
```

Você acredita em mágica?

- O {lubridate} é tão poderoso que pode parecer mágica

```
dmy("15 de abril de 2021", locale = "pt_BR.UTF-8") # No Win: Portuguese_Brazil.1252
```

```
#> [1] "2021-04-15"
```

```
mdy("April 15th 2021", locale = "en_US.UTF-8")      # Mês-dia-ano
```

```
#> [1] "2021-04-15"
```

- Às vezes o Excel salva datas como o número de dias desde 01/01/1970, mas nem isso pode vencer o {lubridate}

```
as_date(18732)
```

```
#> [1] "2021-04-15"
```

Fusos

- É mais raro precisar lidar com fusos horários porque normalmente trabalhamos com data-horas de um mesmo fuso, mas o `{lubridate}` permite lidar com isso também

```
dmy_hms("15/04/2021 02:25:30", tz = "Europe/London")
```

```
#> [1] "2021-04-15 02:25:30 BST"
```

- Nem o horário de verão consegue atrapalhar um cálculo preciso: com a função `dst()` é possível saber se em um dado dia aquele lugar estava no horário de verão

```
dst(dmy_hms("15/04/2021 02:25:30", tz = "Europe/London"))
```

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

Componentes

- As funções `year()`, `month()`, `day()`... (**no singular**) podem extrair os componentes de uma data

```
month("2021-04-15")
```

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

- Obs.: Note como não foi necessário converter a string para data porque ela já está no formato esperado pelo `{lubridate}`

- As funções `years()`, `months()`, `days()`... (**no plural**) permitem fazer contas com datas e data-horas

```
now() + days(5)
```

```
#> [1] "2021-08-24 19:04:16 -03"
```

Operações

- Com os operadores matemáticos normais também somos capazes de calcular distâncias entre datas e horas

```
dif <- dmy("15/04/2021") - dmy("24/08/2020")
dif
```

```
#> Time difference of 234 days
```

- Podemos transformar um objeto de diferença temporal em qualquer unidade que queiramos usando as funções no plural

```
as.period(dif) / minutes(1)
```

```
#> [1] 336960
```

- Para diferenças entre data-horas pode ser importante usar os fusos

Exemplos intermináveis

```
dmy_hms("15/04/2021 02:25:30") # Data-hora
```

```
#> [1] "2021-04-15 02:25:30 UTC"
```

```
dmy_hm("15/04/2021 02:25")     # Sem segundo
```

```
#> [1] "2021-04-15 02:25:00 UTC"
```

```
dmy_h("15/04/2021 02")         # Sem minuto
```

```
#> [1] "2021-04-15 02:00:00 UTC"
```

```
as_datetime(1618453530)        # Numérico
```

```
#> [1] "2021-04-15 02:25:30 UTC"
```

Exemplos intermináveis (cont.)

```
mdy_hms("4/15/21 2:25:30 PM")                                  # Americano
```

```
#> [1] "2021-04-15 14:25:30 UTC"
```

```
dmy_hms("15/04/2021 02:25:30", tz = "Europe/London")           # Com fuso
```

```
#> [1] "2021-04-15 02:25:30 BST"
```

```
now() - dmy_hms("15/04/2021 02:25:30")                         # Diferença
```

```
#> Time difference of 126.8186 days
```

```
now() - dmy_hms("15/04/2021 02:25:30", tz = "Europe/London") # Com fuso
```

```
#> Time difference of 126.8603 days
```

Exemplos intermináveis (cont.)

```
minute("2021-04-15 02:25:30")                    # Minuto
```

```
#> [1] 25
```

```
year("2021-04-15")                               # Ano
```

```
#> [1] 2021
```

```
wday("2021-04-15")                               # Dia da semana
```

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

```
month("2021-04-15", label = TRUE, abbr = FALSE) # Mês (sem abrev.)
```

```
#> [1] abril
#> 12 Levels: janeiro < fevereiro < março < abril < maio < junho < ... < dezembro
```

Exemplos intermináveis (cont.)

```
today() + months(5)                                  # Dia
```

```
#> [1] "2022-01-19"
```

```
now() + seconds(5)                                   # Segundo
```

```
#> [1] "2021-08-19 19:04:21 -03"
```

```
now() + days(5)                                      # Dia
```

```
#> [1] "2021-08-24 19:04:16 -03"
```

```
as.period(today() - dmy("01/01/2021")) / days(1) # Dia - dia
```

```
#> [1] 230
```

Exemplos intermináveis (cont.)

```
t1 <- dmy_hms("15/04/2021 02:25:00", tz = "America/Sao_Paulo")
t2 <- dmy_hms("15/04/2021 02:25:00")
t1 - t2
```

```
#> Time difference of 3 hours
```

```
t1 <- dmy_hms("15/04/2021 02:25:00", tz = "America/Sao_Paulo")
t2 <- dmy_hms("15/04/2021 02:25:00", tz = "Europe/London")
t1 - t2
```

```
#> Time difference of 4 hours
```

```
head(OlsonNames())
```

```
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

Extra: {clock}

- Como não existe o dia 31/02/2021 (fevereiro tem menos dias), o `{lubridate}` simplesmente considera a operação inválida e não nos avisa!

```
x <- ymd("2021-01-31")
x + months(1)
```

```
#> [1] NA
```

```
# No {clock}, 31/01/2021 + 1 mês é um erro que deve ser corrigido
library(clock)
add_months(x, 1)
```

```
#> Error: Invalid date found at location 1.
#> ℹ Resolve invalid date issues by specifying the `invalid` argument.
```

```
# Agora podemos especificar uma estratégia de correção
add_months(x, 1, invalid = "overflow")
```

```
#> [1] "2021-03-03"
```

Forcats

Motivação

No R, um dos tipos de dado mais importante é o fator: se você está vindo do R antigo, provavelmente já teve que usar o argumento `stringsAsFactors = FALSE`. Mas qual é a razão por trás da existência dos fatores? Não seria melhor tudo ser string como em outras linguagens?

A resposta simples é: não. Fatores são a forma do R lidar com variáveis categóricas (ordenadas ou não) e eles podem facilitar muito a vida, tanto na modelagem quanto na visualização dos dados. Hoje em dia é menos comum ter variáveis categóricas em uma base do que variáveis textuais (por isso o ódio pelo `stringsAsFactors`), mas isso não quer dizer que fatores não sejam uma ferramenta incrível.

Para nos ajudar a trabalhar com fatores, temos o pacote `{forcats}` (***for cat****egorial variable****s***). As suas principais funções servem para alterar a **ordem** e modificar os **níveis** de um fator.

Introdução

- Por padrão, quando criamos um fator manualmente, a função `as_factor()` recebe strings que denotam as categorias. As categorias são guardadas na ordem em que aparecem (o que é diferente do {base}):

```
library(forcats)

x <- as_factor(c("baixo", "médio", "baixo", "alto", NA))
x
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: baixo médio alto
```

- Formalmente, um fator não passa de um **vetor numérico** com níveis (*levels*): os nomes de cada categoria

```
typeof(x)
```

```
#> [1] "integer"
```

Vantagens

- Como já aludido, os fatores são úteis na modelagem estatística: no ANOVA, por exemplo, é útil e adequado interpretar um vetor de textos como um vetor de números inteiros
- Fatores também ocupam significativamente menos espaço em memória do que strings (quando seu uso for apropriado) já que são armazenados como inteiros, mas podem ser trabalhados como strings

```
x[x != "médio"]
```

```
#> [1] baixo baixo alto  <NA>
#> Levels: baixo médio alto
```

- Mais interessante ainda é trabalhar com fatores ordenados, que facilitam muito a criação de gráficos porque permitem ordenar variáveis não-alfabeticamente

```
lubridate::month(Sys.Date(), label = TRUE, abbr = FALSE)
```

```
#> [1] agosto
#> 12 Levels: janeiro < fevereiro < março < abril < maio < junho < ... < dezembro
```

Principais funções

```
# Remover níveis sem representantes
fct_drop(x[x != "médio"])
```

```
#> [1] baixo baixo alto  <NA>
#> Levels: baixo alto
```

```
# Re-rotular os níveis com uma função
fct_relabel(x, stringr::str_to_upper)
```

```
#> [1] BAIXO MÉDIO BAIXO ALTO  <NA>
#> Levels: BAIXO MÉDIO ALTO
```

```
# Concatenar fatores
fct_c(x, as_factor(c("altíssimo", "perigoso")))
```

```
#> [1] baixo     médio     baixo     alto      <NA>      altíssimo perigoso
#> Levels: baixo médio alto altíssimo perigoso
```

Principais funções

```
# Re-nívelar fator (trazer níveis para frente)
( x2 <- fct_relevel(x, "alto", "médio") )
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: alto médio baixo
```

```
# Transformar a ordem dos elementos no ordenamento do fator
fct_inorder(x2, ordered = TRUE)
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: baixo < médio < alto
```

```
# Transformar a ordem dos níveis no ordenamento do fator
as.ordered(x2)
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: alto < médio < baixo
```

Principais funções

```
# Transformar NA em um fator explícito
fct_explicit_na(x, na_level = "(vazio)")
```

```
#> [1] baixo   médio   baixo   alto    (vazio)
#> Levels: baixo médio alto (vazio)
```

```
# Juntar fatores com poucas ocorrências
fct_lump_min(x, min = 2, other_level = "outros")
```

```
#> [1] baixo  outros baixo  outros <NA>
#> Levels: baixo outros
```

```
# Inverter a ordem dos níveis
fct_rev(x)
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: alto médio baixo
```

Principais funções

```
# Usar um vetor para reordenar (útil no mutate())
fct_reorder(x, c(2, 1, 3, 10, 0), .fun = max)
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: médio baixo alto
```

```
# Alterar manualmente os níveis
lvls_revalue(x, c("P", "M", "G"))
```

```
#> [1] P    M    P    G    <NA>
#> Levels: P M G
```

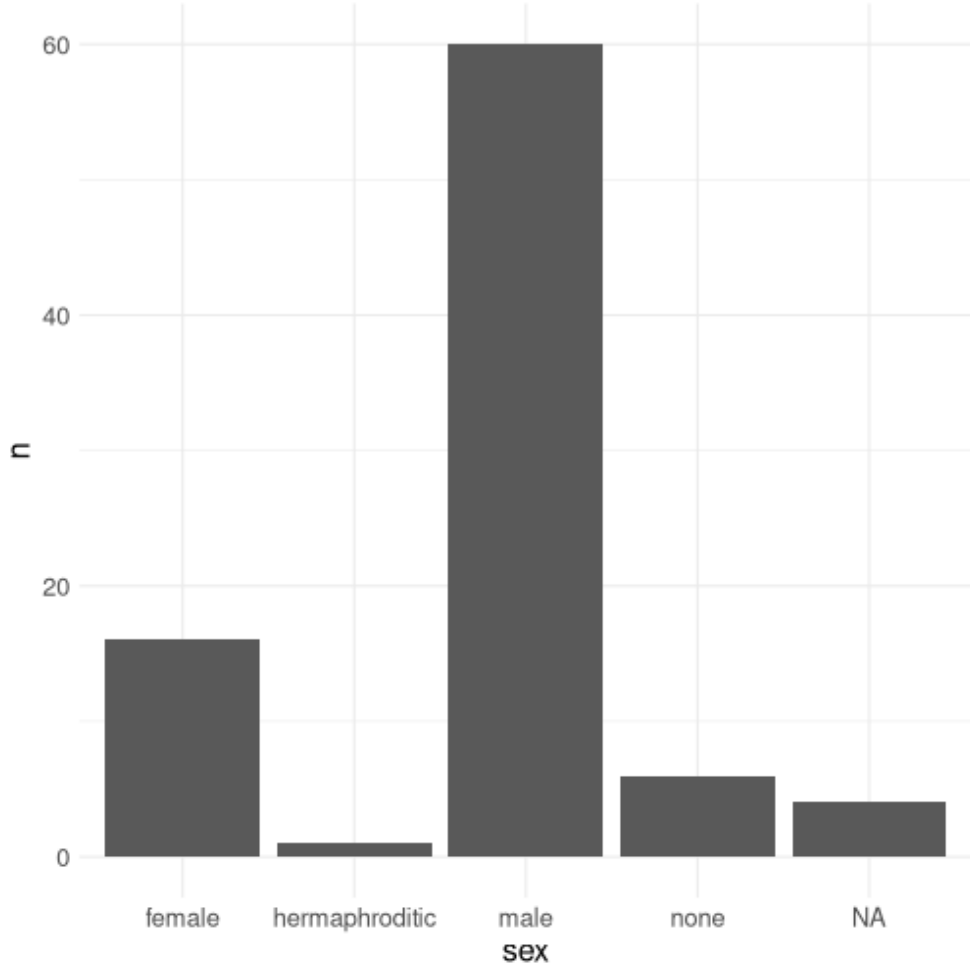
```
# Alterar manualmente a ordem dos níveis
lvls_reorder(x, c(3, 2, 1))
```

```
#> [1] baixo médio baixo alto  <NA>
#> Levels: alto médio baixo
```

Caso de uso

```
starwars %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

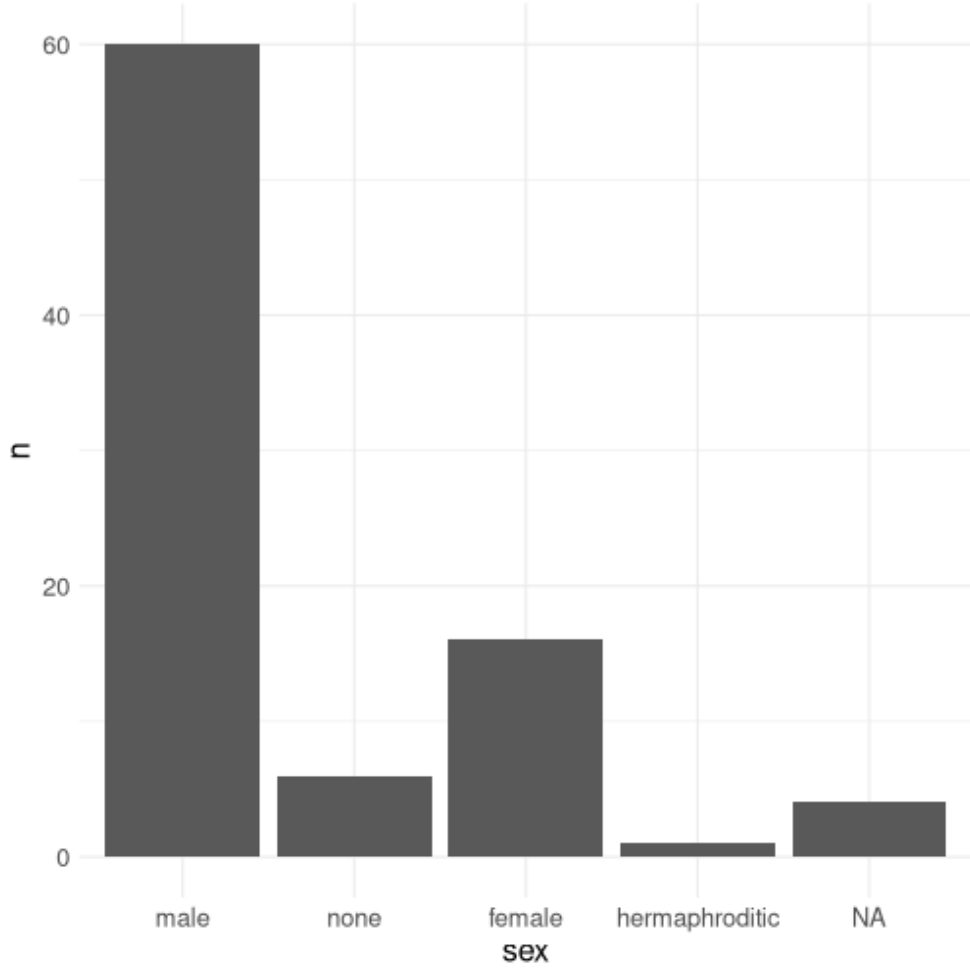
- Um simples gráfico de barras já é ótimo para demonstrar o poder do {`forcats`}
- Note que, ao lado, as barras estão ordenadas pela **ordem alfabética** do sexo



Caso de uso

```
starwars %>%
  mutate(
    sex = as_factor(sex)
  ) %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

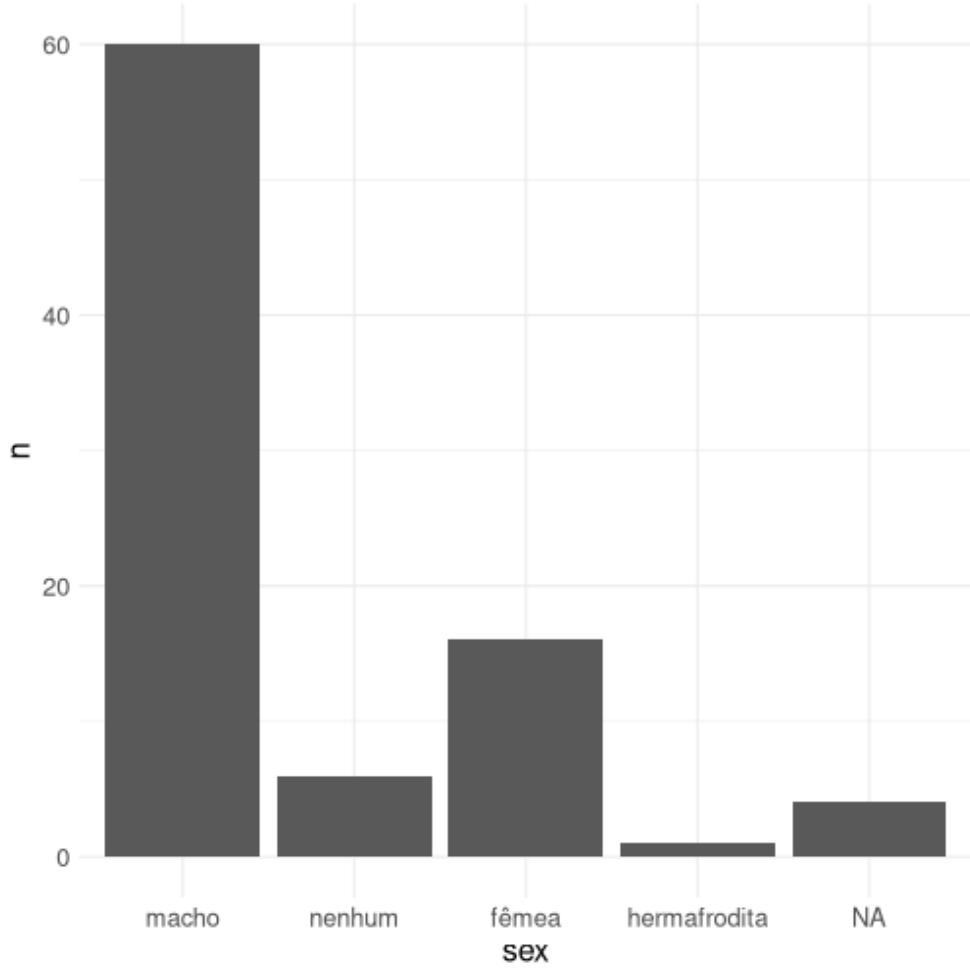
- Transformando a coluna em fator, agora as barras ficam ordenadas pela **precedência na coluna**



Caso de uso

```
starwars %>%
  mutate(
    sex = as_factor(sex),
    sex = fct_relabel(sex, traduzir)
  ) %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

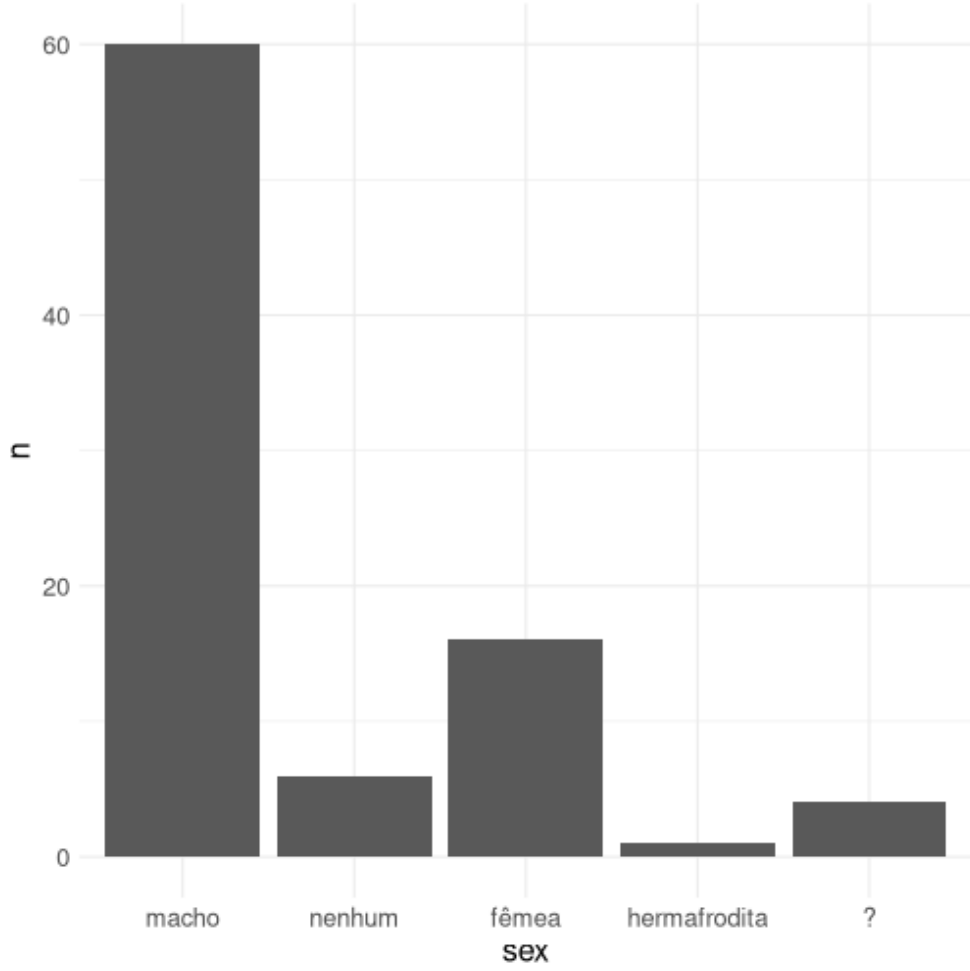
- Para traduzir os níveis, basta uma **pequena função** que retorna o nome em português quando ela recebe o nome em inglês



Caso de uso

```
starwars %>%
  mutate(
    sex = as_factor(sex),
    sex = fct_relabel(sex, traduzir),
    sex = fct_explicit_na(sex, "?")
  ) %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

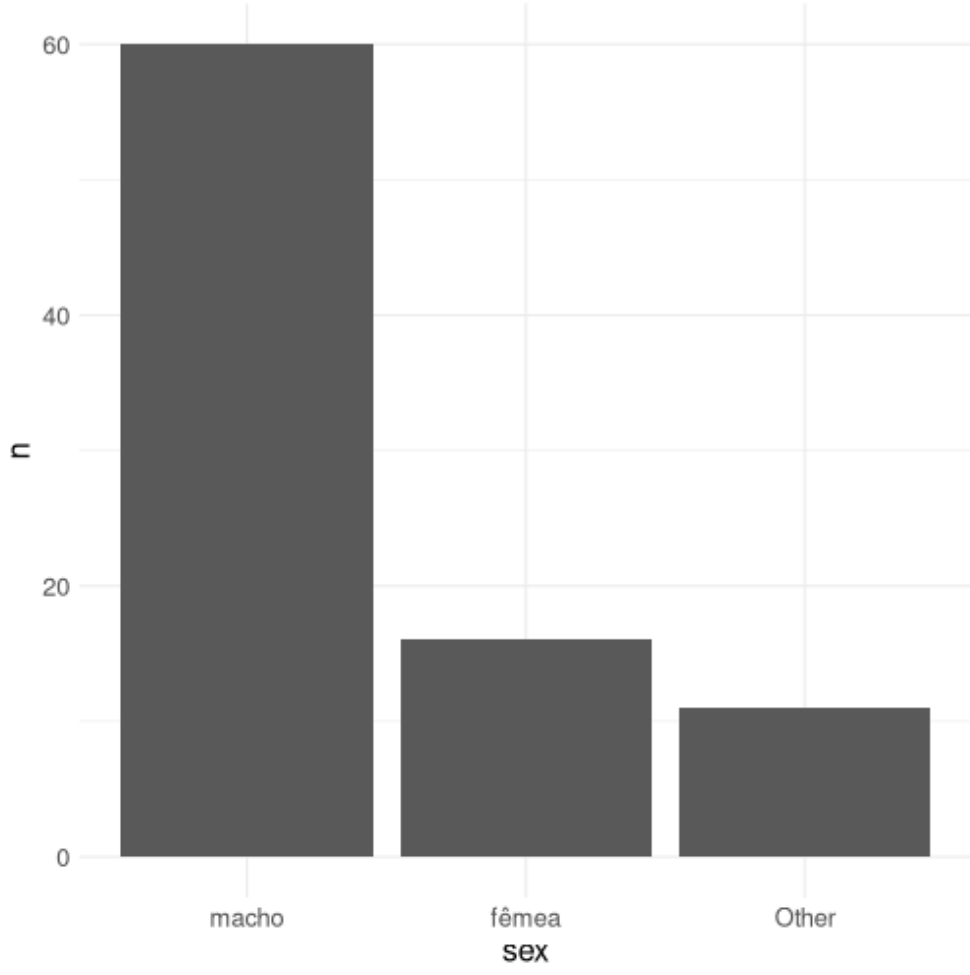
- Fazer com que o `NA` **se torne um fator** também é simples com `fct_explicit_na()`



Caso de uso

```
starwars %>%
  mutate(
    sex = as_factor(sex),
    sex = fct_relabel(sex, traduzir),
    sex = fct_explicit_na(sex, "?"),
    sex = fct_lump_n(sex, 2)
  ) %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

- Se não quisermos todos os níveis, podemos **agrupar os menos frequentes** com a família de funções `fct_lump_***()`



Caso de uso

```
starwars %>%
  mutate(
    sex = as_factor(sex),
    sex = fct_relabel(sex, traduzir),
    sex = fct_explicit_na(sex, "?"),
    sex = fct_lump_n(sex, 2)
  ) %>%
  group_by(sex) %>%
  summarise(n = n()) %>%
  mutate(sex = fct_reorder(sex, n)) %>%
  ggplot(aes(sex, n)) +
  geom_col() +
  theme_custom()
```

- Para **ordenar as barras** de acordo com outra variável, podemos simplesmente usar `fct_reorder()` (trocando o argumento `.fun` quando necessário)