R para Ciência de Dados 2

Metaprogramação

Iniciar gravação!

rlang

Motivação

Todos os pacotes que abordamos neste curso são do tidyverse, então faz sentido querermos realizar nossas análises usando seus princípios. Funções cujo primeiro argumento é um data frame e operações pipeáveis são apenas o começo.

Uma das características mais marcantes do tidyverse é a possibilidade de trabalhar com colunas como se elas fossem objetos comuns, criados fora da função. Basta pensar na `mutate()`: como ela sabe que `mpg` é uma coluna da tabela e não um objeto externo com o mesmo nome?

Existe uma ferramenta especial que nos permite fazer esse tipo de mágica: a metaprogramação. Usamos essa funcionalidade desde a primeira vez que escrevemos o comando `library()`, mas o seu funcionamento ainda é um mistério para a maioria.

O pacote rlang é quem permite que meros mortais escrevam funções que usam metaprogramação (como as do tidyverse), mas antes precisamos entender o básico de como esse novo mundo funciona...

Metaprogramação

Metaprogramação é a ideia de que o código pode ser inspecionado e modificado programaticamente. No R, *non-standard evaluation* é a propriedade dos argumentos que nos permite usar metaprogramação.

```
tidyverse
```

```
#> Error in eval(expr, envir, enclos): object 'tidyverse' not found
```

Se tentarmos executar `tidyverse` direto, recebemos um erro porque este objeto não existe. Em contrapartida, dentro da `library()`, o R consegue ler o código que nós escrevemos e agir em cima dessa informação antes de gerar um erro.

```
library(tidyverse) # Já carrega o rlang
```

Ou seja, o R deixa de apenas executar os nossos comandos e passa a ter a capacidade de interpretar e trabalhar com eles.

A metáfora da bomba

Para entender melhor o que aconteceu no slide passado, vamos pensar em uma bomba: `stop("BOOM")`. Se o R (em qualquer momento) executar essa expressão, ele retornará um erro.

```
stop("BOOM") # Equivalente a executar `tidyverse` diretamente
```

```
#> Error in eval(expr, envir, enclos): BOOM
```

Abaixo temos duas funções para as quais vamos passar a nossa bomba. A primeira, chamada `ignorar()`, não faz nada com o argumento `bomba` e simplesmente retorna `NULL`. Já a segunda, chamada `executar()`, retorna a `bomba` diretamente.

```
ignorar <- function(bomba) NULL
executar <- function(bomba) bomba
```

O que vai acontecer quando passarmos a bomba para cada uma das funções?

A metáfora da bomba: resposta

A avaliação padrão do R é **preguiçosa**, ou seja, os argumentos de uma função só são avaliados se alguma coisa encosta neles. Sendo assim, a bomba só "explode" na função que efetivamente faz alguma coisa com ela.

```
ignorar(stop("BOOM"))
```

```
#> NULL
```

```
executar(stop("BOOM"))
```

```
#> Error in executar(stop("BOOM")): BOOM
```

Para replicar o comportamento da `library()` e das funções do tidyverse, vamos precisar de um jeito de (seletivamente) ignorar a avaliação de certos argumentos mesmo que eles sejam usados dentro da função...

O pacote rlang

Se quisermos usar metaprogramação em nossas funções como o tidyverse, vamos empregar o pacote rlang. Ele implementa uma versão da metaprogramação chamada *tidy evaluation* ou **tidy eval**.

```
expr(stop("BOOM")) # Bomba desarmada
```

```
#> stop("BOOM")
```

Essa implementação é mais simples do que a metaprogramação padrão do R. A tidy eval resolve vários problemas complicados por trás dos panos para que fiquemos só com a parte legal.

Vamos aprender como o pacote funciona através de um exemplo. Usando a função `summarise()`, vamos criar uma função `summarise_mean()` que tira a média de uma coluna fornecida pelo(a) usuário(a). Para critérios ilustrativos, vamos usar a tabela `mtcars` que está disponível no R.

Desarmando argumentos

Por que o código abaixo não funciona? O problema é que a nossa função `summarise_mean()` não está "desarmando" o `col`: o dplyr sabe o que fazer com `cyl`, mas o R está tentando executá-la antes de que ela chegue na `summarise()`. Precisamos de uma maneira de fazer o R ignorar `col` até que ele chegue no seu destino.

```
summarise_mean <- function(df, col) {
  summarise(df, media = mean(col))
}
mtcars |>
  summarise_mean(cyl)
```

```
#> Error in `summarise()`:
#> ℹ In argument: `media = mean(col)`.
#> Caused by error:
#> ! object 'cyl' not found
```

Embracing

O operador de **embracing** (também chamado de curly-curly ou chave-chave) faz o trabalho de "proteger" um argumento da avaliação padrão do R. Para usá-lo basta envolver o argumento em `{{ }}`.

```
summarise_mean <- function(df, col) {
  summarise(df, media = mean({{ col }}))
}
mtcars |>
  summarise_mean(cyl)
```

```
#>    media
#> 1 6.1875
```

É importante notar que o chave-chave é parte do tidy eval e, portanto, só funciona dentro do tidyverse. Usar metaprogramação fora do tidyverse é um pesadelo e foge do tema desta aula.

Injetando nomes

Por que o código abaixo não funciona? Aqui nosso objetivo é manter o nome da coluna depois da `summarise()`, mas obtemos um erro mesmo usando o chave-chave como indicado no slide anterior. O problema é que o operador de igual espera um objeto do lado esquerdo e não uma expressão envolta por chaves.

```
summarise_mean <- function(df, col) {
  summarise(df, {{ col }} = mean({{ col }}))
}
mtcars |>
  summarise_mean(cyl)
```

```
#> Error: <text>:2:27: unexpected '='
#> 1: summarise_mean <- function(df, col) {
#> 2:   summarise(df, {{ col }} =
#>                              ^
```

Walrus

O operador de **walrus** (também chamado de morsa) faz o trabalho de permitir chave-chaves do lado esquerdo de uma expressão com tidy eval. Para usá-lo basta substituir o igual por `:=` quando precisarmos injetar alguma coisa que vai virar o nome de uma coluna.

```
summarise_mean <- function(df, col) {
  summarise(df, {{ col }} := mean({{ col }}))
}
mtcars |>
  summarise_mean(cyl)
```

```
#>      cyl
#> 1 6.1875
```

Obs.: O time que desenvolve o rlang quer acabar com o operador morsa porque ele é um pouco confuso. Para saber mais, veja o progresso aqui.

Indireção

Por que o código abaixo não funciona? Às vezes queremos poder passar strings com o nome de colunas para certas funções, mas o tidyverse não lida bem com elas. Essa é uma tarefa comum no shiny, onde as seleções sempre retornam como strings.

Apesar de a string injetada funcionar perfeitamente no lado esquerdo da expressão (cortesia do operador morsa), o lado direito fica errado porque estamos na verdade tirando a média da string `"cyl"`.

```
summarise_mean <- function(df, col) {
  summarise(df, {{ col }} := mean({{ col }}))
}
mtcars |>
  summarise_mean("cyl")
```

```
#>   cyl
#> 1  NA
```

.data

O pronome `.data` é uma funcionalidade do tidy eval que nos permite acessar, dentro das funções do tidyverse, a tabela que foi passada de argumento. Sendo assim, podemos usar `[[]]` normalmente para pegar uma coluna da tabela usando seu nome.

```
summarise_mean <- function(df, col) {
  summarise(df, {{ col }} := mean(.data[[col]]))
}
mtcars |>
  summarise_mean("cyl")
```

```
#>      cyl
#> 1 6.1875
```

Na prática, o `.data[[col]]` é traduzido internamente para `df[["cyl"]]`. Como este comando não tem nada para desarmar, tudo volta a funcionar.

Desarmando expressões

Por que o código abaixo não funciona? Às vezes queremos permitir que o(a) usuário(a) especifique expressões completas que precisamos desarmar, mas o chave-chave não funciona para isso. Também não temos uma maneira de passar um número arbitrário de expressões.

```
summarise_expr <- function(df, expr1, expr2) {
  summarise(df, {{ expr1 }}, {{ expr2 }})
}
mtcars |>
  summarise_expr(
    cyl = mean(cyl),
    gear = mean(gear)
  )
```

```
#> Error in summarise_expr(mtcars, cyl = mean(cyl), gear = mean(gear)): unused
```

Dots

Os **dots** (também chamado de ellipsis ou elipse), que vimos na aula de funções, resolve nosso problema aqui perfeitamente. Funções que aceitam dots como argumento não dependem de nenhum tipo de desarme.

```
summarise_expr <- function(df, ...) {
  summarise(df, ...)
}
mtcars |>
  summarise_expr(
    cyl = mean(cyl),
    gear = mean(gear)
  )
```

```
#>       cyl   gear
#> 1 6.1875 3.6875
```

Rodada bônus!

O tema de metaprogramação é longo e complexo. Aqui vimos somente os operadores principais, que nos permitem criar funções com os verbos do tidyverse.

Situação	Sintaxe
Normal	`x = y`
Objeto do lado direito	`x = {{ arg }}`
Objeto do lado esquerdo	`{{ arg }} := y`
String do lado direito	`x = .data[[arg]]`
String do lado esquerdo	`{{ arg }} := y`

Existe mais uma pilha de conceitos que precisaríamos aprender para criar uma função com tidy eval do zero. Hoje vimos desarme, injeção e indireção, mas isso é só a ponta do iceberg! A própria documentação do rlang não faz a menor questão de ser consistente com os termos e ao longo do tempo eles foram mudando muito.

Fim