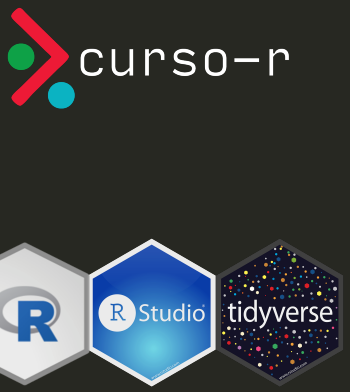
Introdução à programação com R

Material complementar da Aula 4



Tópicos desta aula

- Funções
- Caminhos absolutos e relativos
- Lidando com arquivos com o pacote `fs`

Funções

Funções

Enquanto objetos são *nomes* que guardam *valores*, funções no R são *nomes* que guardam um **código de R**. A ideia é a seguinte: sempre que você rodar uma função, o código que ela guarda será executado e um resultado nos será devolvido.

Funções são tão comuns e intuitivas (provavelmente você já usou funções no Excel), que mesmo sem definir o que elas são, nós já utilizamos funções nas seções anteriores:

- a função `c()` foi utilizada para criar vetores;
- a função `class()` foi utilizada para descobrir a classe de um objeto.
- a função `dim()` foi utilizada para verificarmos a dimensão de um *data frame*.

Argumentos

Diferentemente dos objetos, as funções podem receber **argumentos**. Argumentos são os valores que colocamos dentro dos parênteses e que as funções precisam para funcionar (calcular algum resultado). Por exemplo, a função `c()` precisa saber quais são os valores que formarão o vetor que ela irá criar.

```
c(1, 3, 5)
```

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

Nesse caso, os valores `1`, `3` e `5` são os argumentos da função `c()`. **Os argumentos de uma função são sempre separados por vírgulas**.

Funções no R têm personalidade. Cada uma pode funcionar de um jeito diferente das demais, mesmo quando fazem tarefas parecidas. Por exemplo, vejamos a função `sum()`.

```
sum(1, 3)
```

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

Como você deve ter percebido, essa função retorna a soma de seus argumentos. Também podemos passar um vetor como argumento, e ela retornará a soma dos elementos do vetor.

```
sum(c(1, 3))
```

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

Já a função `mean()`, que calcula a média de um conjunto de valores, exige que você passe valores na forma de um vetor:

```
# Só vai considerar o primeiro número na média
mean(1, 3)
```

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

```
# Considera todos os valores dentro do vetor na média
mean(c(1, 3))
```

```
## [1] 2
```

Os argumentos das funções também têm nomes, que podemos ou não usar na hora de usar uma função. Veja por exemplo a função `seq()`.

```
seq(from = 4, to = 10, by = 2)
```

```
## [1]  4  6  8 10
```

Entre outros argumentos, ela possui os argumentos `from=`, `to=` e `by=`. O que ela faz é criar uma sequência (vetor) de `by` em `by` que começa em `from` e termina em `to`. No exemplo, criamos uma função de 2 em 2 que começa em 4 e termina em 10.

Também poderíamos usar a mesma função sem colocar o nome dos argumentos:

```
seq(4, 10, 2)
```

```
## [1]  4  6  8 10
```

Para utilizar a função sem escrever o nome dos argumentos, você precisa colocar os valores na ordem em que os argumentos aparecem. E se você olhar a documentação da função `seq()`, fazendo `help(seq)`, verá que a ordem dos argumentos é justamente `from=`, `to=` e `by=`.

Escrevendo o nome dos argumentos, não há problema em alterar a ordem dos argumentos:

```
seq(by = 2, to = 10, from = 4)
```

```
## [1]  4  6  8 10
```

Mas se especificar os argumentos, a ordem importa. Veja que o resultado será diferente.

```
seq(2, 10, 4)
```

```
## [1]  2  6 10
```

Vocabulário

A seguir, apresentamos algumas funções nativas do R úteis para trabalhar com *data frames* :

- `head()` - Mostra as primeiras 6 linhas.
- `tail()` - Mostra as últimas 6 linhas.
- `dim()` - Número de linhas e de colunas.
- `names()` - Os nomes das colunas (variáveis).
- `str()` - Estrutura do *data frame*. Mostra, entre outras coisas, as classes de cada coluna.
- `cbind()` - Acopla duas tabelas lado a lado.
- `rbind()` - Empilha duas tabelas.

Criando a sua própria função

Além de usar funções já prontas, você pode criar a sua própria função. A sintaxe é a seguinte:

```
nome_da_funcao <- function(argumento_1, argumento_2) {

  # Código que a função irá executar

}
```

Repare que `function` é um nome reservado no R, isto é, você não pode criar um objeto com esse nome.

Um exemplo: vamos criar uma função que soma dois números.

```
minha_soma <- function(x, y) {
  soma <- x + y

  soma  # resultado retornado
}
```

Essa função tem os seguintes componentes:

- `minha_soma`: nome da função
- `x` e `y`: argumentos da função
- `soma <- x + y`: operação que a função executa
- `soma`: valor retornado pela função

Após rodarmos o código de criar a função, podemos utilizá-la como qualquer outra função do R.

```
minha_soma(2, 2)
```

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

O objeto `soma` só existe *dentro da função*, isto é, além de ele não ser colocado no seu *environment*, ele só existirá na memória (RAM) enquanto o R estiver executando a função. Depois disso, ele será apagado. O mesmo vale para os argumentos `x` e `y`.

O valor retornado pela função representa o resultado que receberemos ao utilizá-la. Por padrão, **a função retornará sempre a última linha de código que existir dentro dela**. No nosso exemplo, a função retorna o valor contido no objeto `soma`, pois é isso que fazemos na última linha de código da função.

Repare que se atribuirmos o resultado a um objeto, ele não será mostrado no console:

```
resultado <- minha_soma(3, 3)

# Para ver o resultado, rodamos o objeto `resultado`
resultado
```

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

Agora, o que acontece se a última linha da função não devolver um objeto? Veja:

```
minha_nova_soma <- function(x, y) {
  soma <- x + y
}
```

A função `minha_nova_soma()` apenas cria o objeto `soma`, sem retorná-lo como na função `minha_soma()`. Se utilizarmos essa nova função, nenhum valor é devolvido no console:

```
minha_nova_soma(1, 1)
```

No entanto, a última linha da função agora é a atribuição `soma <- x + y` e esse será o "resultado retornado". Assim, podemos visualizar o resultado da função fazendo:

```
resultado <- minha_nova_soma(1, 1)

resultado
```

```
## [1] 2
```

É como se, por trás das cortinas, o R estivesse fazendo `resultado <- soma <- x + y`, mas apenas o objeto `resultado` continua existindo, já que os objetos `soma`, `x`e `y` são descartados após a função ser executada.

Caminhos absolutos e relativos

Caminhos

Um passo importante na tarefa de importação de dados para o R é saber onde está o arquivo que queremos importar.

Toda função de importação vai exigir um **caminho**, uma string que representa o endereço do arquivo no computador.

Há duas formas de passarmos o caminho de arquivo: usar o **caminho absoluto** ou usar o **caminho relativo**.

Antes de falarmos sobre a diferença dos dois, precisamos definir o que é o **diretório de trabalho**.

Diretório de trabalho

O diretório de trabalho (*working directory*) é a pasta em que o R vai procurar arquivos na hora de ler informações ou gravar arquivos na hora de salvar objetos.

Se você está usando um projeto, o diretório de trabalho da sua sessão será, por padrão, a pasta raiz do seu projeto (é a pasta que contém o arquivo com extensão `.Rproj`).

Se você não estiver usando um projeto ou não souber qual é o seu diretório de trabalho, você pode descobri-lo usando a seguinte função `getwd()`.

Ela vai devolver uma string com o caminho do seu diretório de trabalho.

A função `setwd()` pode ser utilizada para mudar o diretório de trabalho. Como argumento, ela recebe o caminho para o novo diretório.

Caminhos absolutos

Caminhos absolutos são aqueles que tem início na pasta raiz do seu computador/usuário. Por exemplo:

```
/Users/beatrizmilz/Documents/Curso-R/main-intro-programacao/slides
```

Esse é o caminho absoluto para a pasta onde esses slides foram produzidos.

Na grande maioria dos casos, caminhos absolutos são uma **má prática**, pois deixam o código irreprodutível. Se você trocar de computador ou passar o script para outra pessoa rodar, o código não vai funcionar, pois o caminho absoluto para o arquivo muito provavelmente será diferente.

Caminhos relativos

Caminhos relativos são aqueles que tem início no diretório de trabalho da sua sessão.

O diretório de trabalho da sessão utilizada para produzir esses slides é a pasta `intro-programacao-em-r-mestre`. Veja o caminho absoluto no slide anterior. Então, o caminho relativo para a pasta onde esses slides foram produzidos seria apenas `slides/`.

Trabalhar com projetos no RStudio ajuda bastante o uso de caminhos relativos, pois nos incentiva a colocar todos os arquivos da análise dentro da pasta do projeto.

Assim, se você usar apenas caminhos relativos e compartilhar a pasta do projeto com alguém, todos os caminhos existentes nos códigos continuarão a funcionar em qualquer computador!

Lidando com arquivos

com o pacote `fs`

Lidando com arquivos com o pacote `fs`

- O pacote `fs` tem como foco lidar com arquivos!
- Para utilizá-lo:

```
install.packages("fs") # instale caso seja necessário!
library(fs) # carregue o pacote para usar
```

Algumas funções muito úteis:

- `dir_create()` - Função para criar um diretório (uma pasta no projeto ou outro local no computador). Caso ela já exista, nada acontecerá.
- `file_create()` - Função para criar um arquivo. Caso ele já exista, nada acontecerá.
- `dir_copy()` - Função para copiar um diretório (uma pasta).
- `file_move()` - Função para mover arquivos.
- `file_delete()` - Função para deletar arquivos.

- `dir_tree()` - Função para visualizar a estrutura de uma pasta.

```
fs::dir_tree(path = "../", recurse = 0)
```

```
## ../
## ├── LICENSE
## ├── README.Rmd
## ├── README.md
## ├── _config.yml
## ├── dados
## ├── exemplos_de_aula
## ├── exercicios
## ├── intro-progamacao.Rproj
## ├── logo.png
## ├── material.txt
## ├── programa_anterior.Rmd
## └── slides
```

- Mais funções: documentação do pacote `fs`