R e RStudio para Iniciantes

Material de Apoio para Cursos Quantitativos do Instituto de Economia da Universidade Federal do Rio de Janeiro (IE/UFRJ)

GPEQ/UFRJ

2024-04-12

Índice

1 Objects **3**
1.0.1 Tipo & Forma 3

2 Primeiros passos **5**
2.1 Operadores Aritméticos 5
2.2 Operadores Lógicos 6
2.3 Possíveis complicações 7

3 Funções e pacotes **8**
3.1 O que é uma função? 8
3.2 O que é uma função **para o R**? 9
3.2.1 Vantagens 10
3.2.2 Criando 10
3.2.3 Utilizando 11
3.3 Pacotes 14
3.3.1 Instalando 15
3.3.2 Carregando 15
3.4 Operador Pipe `%>%` 16
3.4.1 Pipe nativo 17

1 Objects

[...]

1.0.1 Tipo & Forma

Vamos nos aprofundar um pouco mais. Ao lidar formalmente com dados, **devemos ter mente que eles são compostos por uma ou mais variáveis e seus valores**. *Uma variável é uma dimensão ou propriedade que descreve uma unidade de observação* (por exemplo, uma pessoa) e normalmente pode assumir valores diferentes. Por outro lado, os *valores são as instâncias concretas que uma variável atribui a cada unidade de observação e são ainda caracterizados por seu intervalo* (por exemplo, valores categóricos versus valores contínuos) *e seu tipo* (por exemplo, valores lógicos, numéricos ou de caracteres). Estaremos interessados no *tipo* dos dados. A Tabela 1.1 apresenta os que podem aparecer com maior frequência.

Tabela 1.1: Tipos mais comuns de dados

Tipo	Serve para representar...	Exemplo
Númerico	números do tipo *integer* (inteiro) ou *double* (reais)	1, 3.2, 0.89
Texto *(string)*	caracteres (letras, palavras ou setenças)	"Ana jogou bola"
Lógico	valores verdade do tipo lógico (valores booleanos)	TRUE, FALSE, NA
Tempo	datas e horas	14/04/1999

Voltando ao primeiro exemplo, uma pessoa pode ser descrita pelas variáveis *nome*, *número de horas dormidas* e *se dormiu ou não mais de oito horas*. Os valores correspondentes a essas variáveis seriam do tipo texto (por exemplo, "Pedro"), numéricos (número de horas) e lógicos (`TRUE` ou `FALSE`, definido em função do tempo descansado[1]). **Note a diferença entre *dado* e *valor*.** O número 10 é um valor, sem significado. Por outro lado, *"10 horas dormidas"* é um dado, caracterizado pelo valor 10 e pela variável *"horas dormidas"*.

Outro aspecto importante sobre os dados está em sua forma, ou seja, como os dados podem ser organizados. A Tabela 1.2 apresenta as formas mais comuns de organização.

[1] Se o número de horas que a pessoa descansou for maior do que 8, então a variável deverá apresentar valor igual a `TRUE` – ou seja, é verdade que a pessoa dormiu mais de 8 horas. Caso contrário, `FALSE`.

Tabela 1.2: Formas pelas quais os dados podem ser organizados

Formato	Os dados se apresentam como...	Exemplo
Escalar	elementos individuais	"AB", 4, TRUE
Retangular	dados organizados em i linhas e j colunas	Vetores e Tabelas de Dados
Não-retangular	junção de uma ou mais estruturas de dados	Listas

[...]

2 Primeiros passos

> Partes deste capítulo são baseadas na seção 3.2 'R como calculadora' do livro *Ciência de Dados em R*, feito pelo Curso-R. De qualquer modo, eventuais erros são inteiramente de nossa responsabilidade.

Como vimos nos capítulos anteriores, o papel do **Console** no R é interpretar os nossos comandos à luz da linguagem. Ele avalia o código que o passamos e devolve a saída correspondente — se tudo der certo — ou uma mensagem de erro — se o seu código tiver algum problema. Essa operação é chamada de **avaliar**, **executar** ou **rodar** o código. Para que seu código seja executado diretamente no Console, escreva-o e, na sequência, aperte `Enter`. A outra forma de executar uma expressão é escrever o código em um *script* no **Editor**, deixar o cursor em cima da linha e usar o atalho `Ctrl + Enter`. Assim, o comando é enviado para o Console, onde é diretamente executado.

Nesse capítulo, você *rodará* suas primeiras linhas de código com intuito de realizar operações aritméticas como *adição*, *subtração*, *multiplicação* e *divisão*, além de comparações lógicas simples. O objetivo aqui não é te ensinar matemática básica, mas te preparar para a execução de linhas de código mais avançadas. É a forma mais fácil de um iniciante ganhar familiaridade e experiência prática com o R.

2.1 Operadores Aritméticos

De agora em diante, cada região sombreada de cinza representa código, ao passo que seu resultado estará exposto logo na sequência. Vamos começar com um exemplo simples:

```
1 + 1
```

```
[1] 2
```

Nesse caso, o nosso comando foi o código `1 + 1` e a saída foi o valor `2`. Como você pode reproduzir esse comando no RStudio? Inicialmente, copie o que está escrito acima ao clicar no símbolo de prancheta no canto superior direito da região sombreada. Na sequência, cole no Editor de Código e aperte `Ctrl + Enter` (ou então no Console, pressionando apenas `Enter`). Observe abaixo!

Tente agora jogar no Console a expressão: `2 * 2 - (4 + 4) / 2`. Deu zero? Pronto! Você já é capaz de pedir ao R para fazer *qualquer uma das quatro operações aritméticas básicas*. Repare que as operações e suas precedências são mantidas como na matemática, ou seja, divisão e multiplicação são calculadas antes da adição e subtração, além de os parênteses ditarem a ordem na qual serão realizadas. A seguir, apresentamos a Tabela 2.1 resumindo como fazer as principais operações no R.

Tabela 2.1: Operadores matemáticos do R

Operação	Operador	Exemplo	Resultado
Adição	+	1 + 1	2.00
Subtração	-	4 - 2	2.00
Multiplicação	*	2 * 3	6.00
Divisão	/	5 / 3	1.67
Potenciação	^	4 ^ 2	16.00
Resto da Divisão	%%	5 %% 3	2.00
Parte Inteira da Divisão	%/%	5 %/% 3	1.00

2.2 Operadores Lógicos

O R permite também testar comparações lógicas. Os valores lógicos básicos em R são `TRUE` (ou apenas `T`) e `FALSE` (ou apenas `F`). Por exemplo, podemos pedir ao R que nos diga se é verdadeiro que 5 é menor do que 3. Como a resposta é obviamente negativa, ele retornará `FALSE`, nos dizendo que a proposição que fizemos é falsa.

```
5 < 3
```

```
[1] FALSE
```

Abaixo, introduzimos a Tabela 2.2 com outros operadores lógicos da linguagem.

Tabela 2.2: Operadores lógicos do R

Operação	Operador	Exemplo	Resultado
Maior que	>	2 > 1	TRUE
Maior ou igual que	>=	2 >= 2	TRUE
Menor que	<	2 < 3	TRUE

Tabela 2.2: Operadores lógicos do R

Operação	Operador	Exemplo	Resultado
Menor ou igual que	<=	5 =< 3	FALSE
Igual à	==	4 == 4	TRUE
Diferente de	!=	5 != 3	TRUE
x **e** y	&	x <- c(1, 4, NA, 8) x[!is.na(x) & x > 5]	8
x **ou** y	|	x <- c(1, 4, NA, 8) x[!is.na(x) | x > 5]	1, 4, 8

2.3 Possíveis complicações

Se você digitar um comando incompleto, como `5 +`, e apertar `Enter`, o R mostrará um `+`, o que não tem nada a ver com a adição da matemática. Isso significa que o R está esperando você enviar **mais** algum código para completar o seu comando. Termine o seu comando ou aperte `Esc` para recomeçar.

```
5 -
+
+ 5
```

```
[1] 0
```

Se você digitar um comando que o R não reconhece, ele retornará uma mensagem de erro. **Não entre em pânico.** Ele só está te avisando que não conseguiu interpretar o comando.

```
5 % 2
```

```
Error: <text>:1:3: unexpected input
1: 5 % 2
      ^
```

Você pode digitar outro comando normalmente em seguida.

```
5 ^ 2
```

```
[1] 25
```

3 Funções e pacotes

A segunda parte da citação dizia:

> ***Everything that happens is a function call.***

Se algum objeto foi criado, armazenado ou transformado, tivemos a participação de uma *função*. Dessa forma, podemos dizer que o R é uma linguagem de programação *funcional*, ou seja, ocorre através da execução de funções. Na prática, quase todos os comandos que iremos realizar tem como base uma função. Tenha em mente que uma função também é um objeto, assim como tudo que *existe* no R. Nesse capítulo, iremos aprender o que são funções no contexto da linguagem R, além de suas utilidades, como escrevê-las e, por fim, como utilizá-las.

3.1 O que é uma função?

Antes, vamos pensar em funções no contexto matemático. Segundo Stewart (2015),

> Uma função é uma regra que atribui, para cada elemento x em um conjunto A, exatamente um elemento, chamado $f(x)$, em um conjunto B.

Em que o conjunto A chamamos de *domínio*, compreendendo todos os valores que a função pode *aceitar*, ao passo que o conjunto B é conhecido como *imagem*, compreendendo todos valores que a função consegue *retornar*. Podemos representar uma função de quatro formas diferentes:

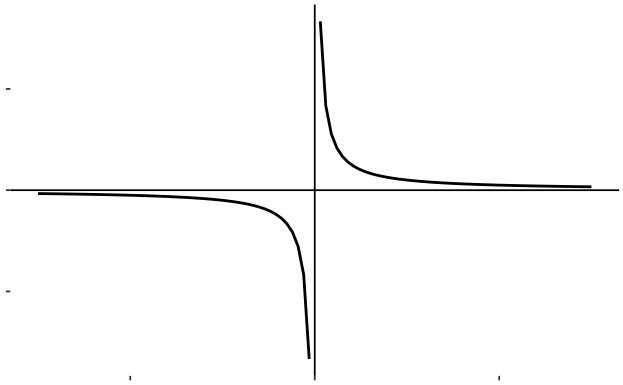
- Verbalmente (através de palavras);
- Numericamente (através uma tabela de dados);
- Visualmente (através de um gráfico);
- Algebricamente (através de uma fórmula)

Vamos deixar os conceitos mais claros. Imagine que te propronho a seguinte regra: dividirei o número 2 por todos os números *possíveis*. Observe que, nesse exemplo, os números possíveis são todos *exceto* zero – afinal de contas, qualquer número dividido por zero resulta em uma indefinição matemática. Podemos dizer, portanto, que o domínio da nossa função é dado por *todos os números reais exceto zero*. Ao mesmo tempo, quais valores podem surgir como

resultado dessa nossa regrinha? Novamente, *todos os reais exceto zero* – afinal, nenhum número que utilizemos será tão grande em valor absoluto tal que a divisão resulte em zero, ao passo que qualquer valor extremo (tanto negativo, quanto positivo) pode ser alcançado *plugando* na função valores menores do que um em módulo[1]! Portanto, nesse caso, a imagem da nossa função é idêntica ao domínio.

Perceba que definimos a função acima de modo *verbal*, ou seja, com palavras. Poderíamos também descrevê-la numericamente através de uma tabela ou, então, de forma visual com um gráfico (ambas abaixo). De todo modo, a forma mais comum de se descrever uma função é através de sua *fórmula* que, no nosso exemplo, seria $f(x) = 2/x$.

x	$f(x)$
-3	-0.67
-2	-1.00
-1	-2.00
1	2.00
2	1.00
3	0.67



Não precisamos nos aprofundar nos conceitos. **O que você precisa guardar dessa seção é o fato de podermos pensar na função como uma *caixa preta* que, ao receber elementos de A, os transforma em um determinado valor *(output)* presente em B.** Como vimos, a forma pela qual essa 'caixa preta' irá transformar x em $f(x)$ é a nossa regra propriamente dita.

3.2 O que é uma função para o R?

E por quê essa historinha toda sobre funções matemáticas nos interessa? Simplesmente pois podemos traçar um paralelo com o conceito de função para a linguagem R! **No R, uma função é um objeto que recebe argumentos *(inputs)* e executa uma ação sobre ou a partir deles, de acordo com o bloco de código nela embutido, te devolvendo um determinado resultado *(output)*. A lógica é a mesma de uma função matemática!** O nome não é por acaso.

[1] Pense em números como $0,01$, por exemplo. Inserindo na nossa função, teríamos então $f(x) = 2/0,01 = 200$. Se inseríssemos $0,001$, obteríamos $f(x) = 2/0,001 = 2000$. O mesmo vale para valores negativos, apenas alterando o sinal do resultado. Perceba que, sendo em módulo menor do que um, quanto mais próximo de zero for o número, mais extremo será o resultado da função.

De certa forma, continuamos a ter um domínio, pois cada função atende um número específico de estruturas de objeto e/ou tipos de variável – algumas funções podem aceitar apenas vetores como argumento, por exemplo, enquanto outras podem ser específicas para variavéis númericas. Ao mesmo tempo, continuamos a ter uma imagem, associada aos resultados possíveis. E qual é o equivalente à regra? É o *bloco de código embutido na função!*

3.2.1 Vantagens

Nas seções anteriores, compreendemos um pouco melhor como funciona o mecanismo de uma função. Mas ainda pode haver dúvida do tipo: *"Beleza, mas em qual contexto prático que ela será útil?"*

O grande benefício de uma função se constitui no fato de seu bloco de código interior, condicionado ao valor dos argumentos, realizar sempre a mesma tarefa quando a rodamos! Isso significa que as funções permitem automatizar tarefas comuns de uma forma mais legível, evitando a prática de 'copiar e colar' repetidamente as *mesmas* linhas de código, que serão substituídas pelo nome da função e seus argumentos. Na prática, além da melhor compreensão do código, eliminamos a chance de cometer erros bobos ao copiar e colar (por exemplo, acabar atualizando o nome de uma variável em um lugar, mas não em outro) e tornamos mais fácil reutilizar o trabalho que foi escrito em outros projetos, aumentando a produtividade.

3.2.2 Criando

A *sintaxe* para criar uma função é a seguinte:

```
nome_da_funcao <- function(arg1 = default1, ..., argn = defaultn) {
  >bloco de código<
}
```

Perceba que o uso o operador `<-` nos mostra que, ao criar uma função, estamos criando um *objeto* – que, nesse caso, não é designado especificamente a armazenar dados. Entre parentêses, definimos o nome dos argumentos e, caso necessário, seus respectivos valores de *default*. Na sequência, entre chaves, escrevemos o bloco de código que rodará sobre os *inputs*. Como exemplo, vamos criar a função `soma2`.

```
soma2 <- function(somando1, somando2) {
  (somando1 + somando2) ^ 2
}
```

O que ela faz? Soma dois números e eleva esse resultado intermediário ao quadrado. A função criada poderá ser vista no quadrante superior direito, no painel **Environment**.

3.2.3 Utilizando

Por sua vez, a *sintaxe* para usar uma função é:

```
nome_da_funcao(arg1, ..., argn)
```

Em primeiro lugar, é necessário escrever o nome da função no Editor de Código. Ao lado, entre parênteses, escreveremos seus argumentos – no exemplo acima, `arg1`, `arg2` até `argn`; uma função pode ser construída de modo a ter qualquer número `n` de argumentos e eles serão sempre separados por vírgula. Esses argumentos são os nossos *inputs*. Por fim, rodamos a linha em que a escrevemos, fazendo com que seja executada pelo R e seu resultado apareça no Console.

Exemplo 1

Vamos tomar como exemplo a função `sum()`. O que ela faz? Segundo sua *documentação*:

> `sum` retorna a soma de todos os valores presentes em seus argumentos

Na prática, como o nome já nos indica, ela tem como serventia somar todos os números que lhe forem passados. Se quisessemos utilizá-la para obter o resultado da soma dos números 4, 7 e 9, como faríamos? Se você pensou em `sum(4, 7, 9)`, acertou!

```
sum(4, 7, 9)
```

```
[1] 20
```

Perceba que utilizamos três argumentos, um para cada número que somamos: 4, 7 e 9 estão associados a `arg1`, `arg2` e `arg3`, respectivamente. No entanto, dado que a função `sum()` aceita objetos como *vetores*, *matrizes* e *dataframes*, poderíamos ter utilizado apenas um único argumento!

```
sum(c(4, 7, 9))
```

```
[1] 20
```

```
sum(matrix(c(4, 7, 9)))
```

```
[1] 20
```

```
sum(data.frame(c(4, 7, 9)))
```

```
[1] 20
```

No *chunk* acima, `c(4, 7, 9)`, `matrix(c(4, 7, 9))` e `data.frame(c(4, 7, 9))` estão associados apenas ao `arg1`! Em muitos casos, entender os tipos de objetos aceitos pela função será importante para a *eficiência* do código. Imagine que estivessemos com interesse de somar todos os valores de uma certa coluna em determinado dataframe. Como poderíamos realizar essa tarefa? Dado que cada coluna de um dataframe é simplesmente um vetor, poderíamos inserí-la diretamente na função, como um único argumento!

```
df1 = data.frame(x = c(4, 7, 9))
sum(df1$x)
```

```
[1] 20
```

Lembre-se de ficar atento com relação à estrutura e/ou tipo de variável que determinada função pode aceitar. Será que se trocassemos o número 4 por "4", a função ainda rodaria? A respota é **não**, afinal de contas "4" é interpretado como texto, e não como número (e você não consegue somar textos)!

```
sum(c("4", 7, 9))
Error in sum(df1$x) : 'type' inválido (character) do argumento
```

Em muitas situações, teremos argumentos *nomeados*. Isto ocorre pois nem todo argumento será processado da mesma forma pelo código embutido na função. Também é muito comum que certos argumentos tenham um valor pré-determinado como *default*, isto é, caso você não especifique algum valor para aquele argumento, o valor de *default* será utilizado.

i Exemplo 2

Nesse caso, vamos tomar como exemplo a função `paste()`. Qual seu papel?

> Concatenar vetores após converter em caractere

Portanto, a função `paste` irá transformar os vetores que introduzirmos em vetores com dados do tipo *character* e, na sequência, irá juntá-los. Na prática, serve para juntar palavras e/ou caracteres que estão inicialmente separados em uma única variável do tipo texto. Por exemplo, podemos estar dispostos a juntar as palavras "Estou", "apredendendo" e "a usar o R" em um único vetor.

```
paste("Estou", "aprendendo", "a usar o R")
```

```
[1] "Estou aprendendo a usar o R"
```

Note que o caractere *default* (padrão) utilizado para separar as palavras é o espaço em branco! E se quisessemos separá-las por vírgula? Nesse caso, teríamos que especificar o argumento `sep` com esse delimitador!

```
paste("Estou", "aprendendo", "a usar o R", sep = ",")
```

```
[1] "Estou,aprendendo,a usar o R"
```

Por fim note que, mesmo sem você saber, já utilizamos funções nos últimos capítulos!

- `c()`: função utilizada para criar um vetor;
- `matrix()`: para criar uma matriz;
- `data.frame()`: para criar um *data frame*;
- `list()`: para criar uma lista.

Todas essas funções utilizavam como argumento principal os dados que tínhamos interesse e, através de seu código embutido, criavam e armazenavam o objeto na memória do R!

Operadores são funções! (Opcional)

Sim, é isso mesmo que você leu! Lembre-se das nossas máximas: tudo que *existe* no R é um *objeto* e tudo que *acontece* é uma execução de *função*. É algo simples e direto ver que operadores como `+` , `:` e `<-` existem na linguagem – não à toa estão sendo mencionados. Mas perceba que eles também criam, armazenam ou transformam objetos! Por exemplo, o operador `+` transforma `1+1` em `2`! Pense nos operadores como um tipo especial de função de dois argumentos, o primeiro posicionado à esquerda e o segundo à direita.
"Mas uma função não deveria ser escrita como `nome_da_funcao(arg1, ..., argn)`*?"*
Você pode escrevê-los dessa forma também! Só não será tão útil.

```
`+`(1, 3)     # O mesmo que 1 + 3
```

```
[1] 4
```

```
`:`(1, 10)    # 1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
`<-`(x, 10)   # x <- 10
```

Funções no Excel (Opcional)

Se você já usou o Microsoft Excel em algum momento da sua vida, com certeza já teve contato com o alguma função! Por exemplo, lá temos a função `=SOMA()`, que realiza a mesma tarefa da função `sum()` no R! Inclusive, também é possível criar funções no Excel através de sua linguagem de programação própria, o *Visual Basic for Applications* (VBA).

3.3 Pacotes

Chamamos de *pacote* um conjunto de dados e/ou funções, acompanhadas de suas respectivas *documentações*, que foram criadas e disponibilizadas por alguma pessoa. Existem pacotes padrão (ou básicos) que são considerados parte do código-fonte do R e estão automaticamente disponíveis como parte da instalação do R (ou seja, foram criados pelos desenvolvedores da linguagem). No entanto, definitivamente não são a maioria: o grosso dos pacotes disponíveis é de autoria dos membros da comunidade. Normalmente, as funções que integram um pacote estão relacionadas à determinado tema, ainda que isso não seja obrigatório.

Por exemplo, suponha que criemos quatro funções que antes não existiam na linguagem: `soma()`, `subtração()`, `multiplicacao()` e `divisao()`, representando as quatro operações aritméticas básicas. Como estão relacionadas à um mesmo tema, poderiamos agrupá-las em um pacote chamado `aritmetica` e, na sequência, disponibilizá-lo em algum *repositório* para que outros usuários da linguagem pudessem baixá-lo. **As pessoas que instalassem e carregassem nosso pacote teriam acesso às quatro funções acima, sem necessidade de ter que criá-las do zero!** Ao mesmo tempo, o autor poderia adicionar um dataframe ao pacote, com intuito de possibilitar que você teste as funções que ele criou[2].

Perceba que, quando alguém cria uma função ou disponibiliza dados através de um pacote, fica complicado compreender inicialmente todas as características que esses objetos possuem. Por exemplo: ainda que os nomes sejam sugestivos, você saberia dizer com precisão o comportamento das quatro funções do pacote que criamos? Quais os tipos de objeto e variável que elas aceitam? Quais e quantos argumentos cada uma aceita? Provavelmente não. Para responder à todas essas perguntas, os autores disponibilizam uma *documentação* para cada objeto do pacote! Para acessá-la, basta rodar o nome da função/conjunto de dados acrescido de `?` no início. O texto irá aparecer no quadrante inferior direito do RStudio, no painel **Help**[3].

[2] Outra possibilidade seria adicionar um dataframe simplesmente pelo interesse em utilizar as informações que podem estar contidas nele. Por exemplo, existem pacotes que contém apenas dataframes com informações de tabelas de livros-texto específicos; com isso, os usuários ganham o poder de replicar os resultados encontrados pelo autor, facilitando o aprendizado. É importante ressaltar, contudo, que pacotes normalmente são compostos apenas por funções.

[3] As documentações sempre aparecerão escritas em inglês.

3.3.1 Instalando

A maneira mais comum de se baixar e instalar um pacote é através do CRAN! O mesmo local em que baixamos o R também atua como um repositório centralizado de pacotes. Mas fique tranquilo: você não precisará acessar o site novamente! Para baixar e instalar um pacote que está no CRAN, utilizaremos a função `install.packages()`, pertencente ao pacote `utils` (que é um pacote básico). Basta rodar `install.packages("nome_do_pacote")`. No exemplo abaixo, a instalação do pacote `readr`.

```
install.packages("readr")
```

Realizada a instalação com sucesso, já passa a ser possível utilizar suas funções/dados. *Nesse momento*, você deverá escrever o nome do pacote acrescido de `::` e, na sequência, o nome da função/dado.

```
readr::read_csv(...)
```

3.3.2 Carregando

"Mas será sempre necessário escrever o nome do pacote antes da função?" **Não.** Perceba que os objetos oriundos de pacotes básicos podem ser executados diretamente. Isso ocorre pois eles são automaticamente *carregados* na sua sessão atual.

Resumindo: mesmo que você tenha instalado um pacote externo com sucesso, para usar seus objetos diretamente (sem precisar escrever seu nome antes) é necessário *carregá-lo* na sessão atual. Para carregar um pacote, rode a função `library()` acrescida do nome do pacote, *sem* aspas.

```
library(readr)
```

Agora, é possível executar o objeto sem precisar escrever o nome do pacote antes.

```
read_csv(...)
```

A prática de carregar pacotes é a mais utilizada. Em outras palavras: sempre que instalarmos algum pacote, na sequência iremos carregá-lo na sessão atual para que possamos utilizar seus objetos de forma direta.

3.4 Operador Pipe %>%

Nem sempre conseguiremos atingir o resultado que queremos utilizando apenas uma função. Por esse motivo, em muitas situações utilizaremos o *resultado* de uma função como *argumento* de *outra* função.

Por exemplo, suponha que você queira somar dois números e, na sequência, comparar este resultado intermediário com 5 e 9, de modo a retornar o valor máximo entre os três números. Observe que a função `sum()` não é suficiente para realizar tal tarefa de modo completo: você até conseguirá somar dois números quaisquer, mas não será capaz de posteriormente comparar o resultado com o restante para saber qual é o valor máximo entre eles. Nesse caso, poderíamos então utilizar o resultado da função `sum()` como *argumento* da função `max()`!

```
max(sum(4, 3), 5, 9)
```

```
[1] 9
```

O R sempre executará as funções *interiores* primeiro. Ou seja, primeiro executa `sum(4,3)`, retornando 7, e na sequência executa `max(7, 5, 9)`, cujo resultado será 9, dado que este é o maior dentre os três números utilizados como argumento.

O problema com esse tipo de *sintaxe* é que, conforme aproveitamos os resultados anteriores de outra função como argumento para as seguintes, mais confuso o código fica. Imagine se o número 4 também fosse resultado de alguma outra função: o código estaria bem mais difícil de entender!

```
max(sum(outra_funcao(...), 3), 5, 9)
```

Com a finalidade de simplificar situações desse tipo, criou-se o operador Pipe, representado por `%>%`. Este operador permite com que o resultado da função anterior se torne, *implicitamente,* o primeiro argumento da função seguinte! Poderíamos então reescrever nosso exemplo para:

```
sum(4,3) %>% max(5, 9)
```

```
[1] 9
```

O que o código acima nos diz é que `sum(4,3)` será interpretado como o primeiro argumento da função `max()`; automaticamente, 5 e 9 se tornam o segundo e o terceiro argumentos, respectivamente.

Para utilizar o operador Pipe, antes é necessário instalar e carregar o pacote `magrittr`:

```r
install.packages("magrittr")
library(magrittr)
```

Na sequência, aperte `Ctrl + Shift + M` no Editor ou Console.

3.4.1 Pipe nativo

O uso do operador Pipe se tornou tão poopular que os desenvolvedores do R resolveram incorporar uma versão própria que já vem pré-instalada, conhecida como Pipe nativo *(native Pipe)*. Ele exerce o mesmo papel principal (organização de código, como demonstrado na seção anterior) mas deixa a desejar em outras partes. A única vantagem é não precisar instalar e carregar um pacote. Portanto, recomendamos ainda o uso do Pipe 'original'.

De todo modo, caso você queira utilizá-lo, basta substituir `%>%` por `|>`. Para continuar usando o atalho `Ctrl + Shift + M`, vá em *Tools* > Global Options > *Code* e marque *Use native pipe operator*.