

6 Estilo de Código e Manipulação das Variáveis

6.1 Questões de Estilo

Como você apresenta o código vai facilitar o seu entendimento do que você fez tanto quanto ajudar uma outra pessoa entendê-lo. Vale a pena seguir umas regras consistentes. Assim, tudo mundo (e R) vai entender o código que você escreveu. Vamos fazer um exemplo para mostrar este ponto.

```
## 1ª Versão
```

```
peso <- 55 ## Pessoa pesa 55 kg.
```

```
## 2ª Versão
```

```
peso_kg <- 55 ## Mais claro
```

```
## Pode Converter à Libra
```

```
peso_lb <- peso_kg * 2.2
```

```
peso_lb
```

```
## [1] 121
```

O primeiro ponto a ressaltar é que este código tem muitos comentários. Fazemos comentários em R utilizando o sinal # (“hashtag”). R não interpreta qualquer caráter depois do primeiro # numa linha de código. Este pode ocorrer numa linha sozinha como ## 1ª Versão ou depois de um operação como na segunda linha (peso <- 55 ## Pessoa pesa 55 kg.).

A primeira versão da atribuição não é muito clara. Se não tivesse o comentário sobre

55 kg , como você saberia que estávamos falando de conversão entre quilos, libras ou toneladas? A segunda versão esclarece diretamente no nome da variável `peso_kg` . E a linha que faz a conversão também tem uma variável com um bom nome que explica claramente o que é este valor.

Frequentemente, pessoas gostam de usar nomes muito breves, mas não muito claros com `x` , `i` ou `n24` . É muito melhor usar nomes que explicam o que contem a variável que de enfatizar brevidade.

Existem muitos outros aspetos de estilo que são padronizado no mundo de programação de R. Uma guia para um estilo claro de programação foi desenvolvido pelo Hadley Wickham de RStudio e pode ser achado aqui: <http://style.tidyverse.org/>. O pacote `styler` que você carregou na instalação criou “Addins” no RStudio que pode acessar do item na barra superior. Esses opções permite que R em si coloca seu código no estilo correto seguindo as normas da guia de Wickham.

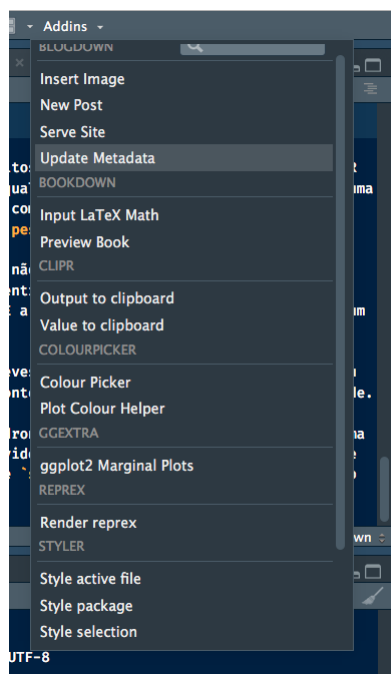
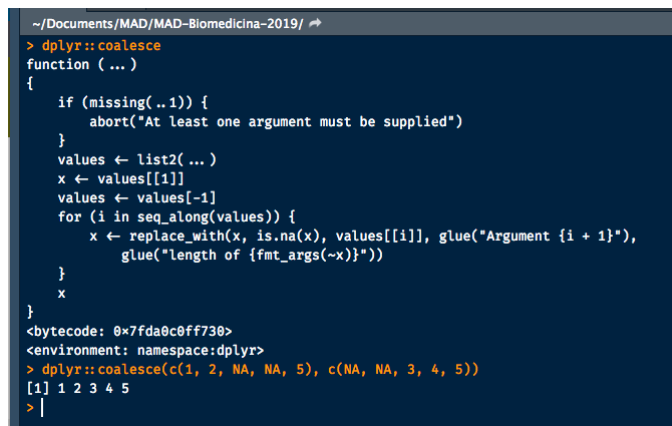


Figure 6.1: Styler Addins

6.1.1 Uma Pergunta sobre o Estilo – Parenteses depois de uma função

Este é um bom momento para comentar porque sempre usamos `()` depois do nome da função. Com os parênteses, R vai executar a função. Sem eles, vai imprimir para a tela o código (ou seja, o conteúdo) da função. Por exemplo, na figura seguinte, mostro

a função `coalesce` do pacote `dplyr`. Na primeira versão, **sem** parênteses, R mostra o código da função. Na segunda, **com** parênteses, R mostra o resultado do cálculo da função.



```
~/Documents/MAD/MAD-Biomedicina-2019/ ➤  
> dplyr::coalesce  
function ( ... )  
{  
  if (missing(..1)) {  
    abort("At least one argument must be supplied")  
  }  
  values <- list2( ... )  
  x <- values[[1]]  
  values <- values[-1]  
  for (i in seq_along(values)) {  
    x <- replace_with(x, is.na(x), values[[i]], glue("Argument {i + 1}"),  
      glue("length of {fmt_args(~x)}"))  
  }  
  x  
}  
<bytecode: 0x7fda0c0ff730>  
<environment: namespace:dplyr>  
> dplyr::coalesce(c(1, 2, NA, NA, 5), c(NA, NA, 3, 4, 5))  
[1] 1 2 3 4 5  
> |
```

Figure 6.2: `coalesce` vs. `coalesce()`

6.2 Conjunto dos Dados Mais Complicado

Este exercício, que estou adaptando do currículo do grupo “Software Carpentry”,⁴ analisa dados médicos sobre tratamentos para a artrite e a inflamação que a doença causa. Ele mostra a avaliação de inflamação de 60 pacientes diariamente durante 40 dias.

Para nossos fins de aprender R, podemos aprender

- Como inserir dados nos objetos R
- Como manipular esses dados
- Gráficos básicos
- Resumos descritivos de dados
- Funções em R
- Loops

Imagine que você trabalha num laboratório farmacêutico que está desenvolvendo um novo tratamento para artrite. Você precisa avaliar os resultados de uma teste deste tratamento para ver se cumpriu os objetivos que a empresa teve para ele. Você recebe o arquivo `inflamação.csv`, um arquivo de Excel gravado no formato `csv` (*comma separated values*). Você precisa colocar este arquivo no seu *working directory* e carregar os dados na memória onde R pode operar neles.

6.2.1 Passo 1 — Criar Um Projeto

Primeiro, criaremos um projeto em RStudio onde vamos colocar a análise de artrite.

Nós trabalharemos neste projeto (e pasta) enquanto trabalhamos com a artrite. No sistema operacional podemos colocar o arquivo `inflamação.csv` na pasta de seu projeto `artrite`.

6.2.2 Passo 2 — Carregar o Tidyverse

Neste exercício e em todo o curso, nós vamos usar muitas funções do Tidyverse. Estas funções existem em vários pacotes. O pacote `tidyverse` é uma coleção dessas outras pacotes e vai carregar eles. Apesar que existem outros pacotes no Tidyverse, aqueles carregados pelo `tidyverse` são os mais importantes e eles permitem que nós podemos ler arquivos, manipular dados, fazer análises e visualizar gráficos. Antes de desenhar gráficos, nós vamos carregar um outro pacote `ggpubr` que simplifica os comandos gráficos utilizando a tecnologia `ggplot`, o componente centrais dos gráficos de Tidyverse.

Para carregar pacotes que já temos instalados no computador, usamos a função `library()` com o nome do pacote como argumento. Entretanto, para ver o efeito de carregar o Tidyverse, vamos olhar na diferença entre o estado da memória antes e depois do Tidyverse. Para isso, usamos a função `sessionInfo()` para mostra o que está carregado. Primeiro, aqui é como sua memória quando você inicia R.

```
sessionInfo() ## Comando para mostrar o estado do sistema R neste momento
```

```
~/Documents/MAD/MAD-Biomedicina-2019/ ➤
> sessionInfo()
R version 3.6.1 (2019-07-05)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS High Sierra 10.13.6

Matrix products: default
BLAS: /System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.framework/Versions/A/libBLAS.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods   base

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.2          compiler_3.6.1      pillar_1.4.2        prettyunits_1.0.2
 [5] tools_3.6.1         zeallot_0.1.0       digest_0.6.20       packrat_0.5.0
 [9] pkgbuild_1.0.3      evaluate_0.14       tibble_2.1.3        gtable_0.3.0
[13] pkgconfig_2.0.2     rlang_0.4.0         cli_1.1.0           rstudioapi_0.10
[17] parallel_3.6.1      charlatan_0.3.0     xfun_0.8            loo_2.1.0
[21] gridExtra_2.3       dplyr_0.8.3         knitr_1.23          vctrs_0.2.0
[25] tidyselect_0.2.5    stats4_3.6.1        rprojroot_1.3-2     grid_3.6.1
[29] glue_1.3.1          inline_0.3.15       here_0.1            R6_2.4.0
[33] processx_3.4.1      fansi_0.4.0         rmarkdown_1.14      rstan_2.19.2
[37] whisker_0.3-2       purrr_0.3.2         callr_3.3.1         ggplot2_3.2.0
[41] magrittr_1.5        matrixStats_0.54.0  backports_1.1.4     scales_1.0.0
[45] ps_1.3.0            StanHeaders_2.18.1-10 htmltools_0.3.6     assertthat_0.2.1
[49] colorspace_1.4-1    utf8_1.1.4          lazyeval_0.2.2      munsell_0.5.0
[53] crayon_1.3.4
```

Figure 6.3: Pacotes Antes Tidyverse

Só têm os sete pacotes básicos que R carregar quando inicia. Agora vamos carregar o Tidyverse.

```
library(tidyverse)
```

```
## — Attaching packages ————— tidyverse 1.
```

```
## ✓ ggplot2 3.2.0          ✓ purrr 0.3.2
## ✓ tibble 2.1.3           ✓ dplyr 0.8.3
## ✓ tidyr 0.8.3.9000       ✓ stringr 1.4.0
## ✓ readr 1.3.1            ✓ forcats 0.4.0
```

```
## — Conflicts ————— tidyverse_conflict
```

```
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag() masks stats::lag()
```

```
sessionInfo()
```

```
~/Documents/MAD/MAD-Biomedicina-2019/
> sessionInfo()
R version 3.6.1 (2019-07-05)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS High Sierra 10.13.6

Matrix products: default
BLAS: /System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.framework/Versions/A/libBLAS.
.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods   base

other attached packages:
[1] forcats_0.4.0  stringr_1.4.0  dplyr_0.8.3   purrr_0.3.2   readr_1.3.1
[6] tidyr_0.8.3.9000 tibble_2.1.3  ggplot2_3.2.0  tidyverse_1.2.1

loaded via a namespace (and not attached):
[1] rstan_2.19.2      tidymodels_0.2.5  xfun_0.8       haven_2.1.1
[5] lattice_0.20-38  colorspace_1.4-1  vctrs_0.2.0    generics_0.0.2
[9] htmltools_0.3.6  stats4_3.6.1      loo_2.1.0      utf8_1.1.4
[13] rlang_0.4.0       pkgbuild_1.0.3    pillar_1.4.2   withr_2.1.2
[17] glue_1.3.1        readxl_1.3.1      modelr_0.1.4   matrixStats_0.54.0
[21] cellranger_1.1.0  munsell_0.5.0     gtable_0.3.0   rvest_0.3.4
[25] charlatan_0.3.0  evaluate_0.14     inline_0.3.15  knitr_1.23
[29] callr_3.3.1       ps_1.3.0          parallel_3.6.1 fansi_0.4.0
[33] broom_0.5.2       Rcpp_1.0.2        scales_1.0.0   backports_1.1.4
[37] jsonlite_1.6      StanHeaders_2.18.1-10 gridExtra_2.3  hms_0.5.0
[41] packrat_0.5.0     digest_0.6.20     stringi_1.4.3  processx_3.4.1
[45] grid_3.6.1        rprojroot_1.3-2   here_0.1       cli_1.1.0
[49] tools_3.6.1       magrittr_1.5       lazyeval_0.2.2 crayon_1.3.4
[53] whisker_0.3-2     pkgconfig_2.0.2   zeallot_0.1.0  xml2_1.2.1
[57] prettyunits_1.0.2 lubridate_1.7.4   httr_1.4.0     assertthat_0.2.1
[61] rmarkdown_1.14    rstudioapi_0.10   R6_2.4.0       nlme_3.1-141
[65] compiler_3.6.1
```

(#fig:tidyverse_load)Tidyverse Básico

Agora, a memória é bem diferente. Temos os nove pacotes carregados como *other attached packages* e 15 pacotes novos carregados em um *namespace* (conceito mais avançado que nosso curso) que são disponíveis para uso pelos pacotes de tidyverse.

6.2.3 Passo 3 — Download os Dados de GitHub

Os dados atualmente existem no GitHub. No capítulo 1, aprendemos como fazer o download dos arquivos do GitHub. Siga o mesmo processo em seu sistema operacional e com seu browser para pôr `inflamacao.csv` na pasta do seu projeto `artrite`.

Para ajudar, aqui é um resumo dos passos:

- aponte a browser para o repo do curso: <https://github.com/jameshunterbr/MAD-Biomedicina-2019>
- clique no nome do arquivo: `inflamacao.csv`
- clique no botão “Raw”
- salvar a página na pasta do projeto. ⁵

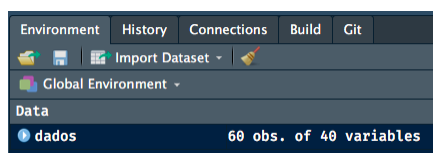
6.2.4 Passo 4 — Carregar os Dados na Memória de R

R precisa ter todos os dados em que vai operar na memória ativa do computador. Não

trabalha nas cópias gravadas dos arquivos para fazer operações. Outras linguagens como SQL e C podem operar em dados gravados, mas R não. Entretanto, este não é um limite em qualquer sentido prático. Nos laptops modernos, o espaço de memória fica suficiente grande para suportar quase qualquer base de dados, mesmo se tivesse múltiplas gigabytes de dados.

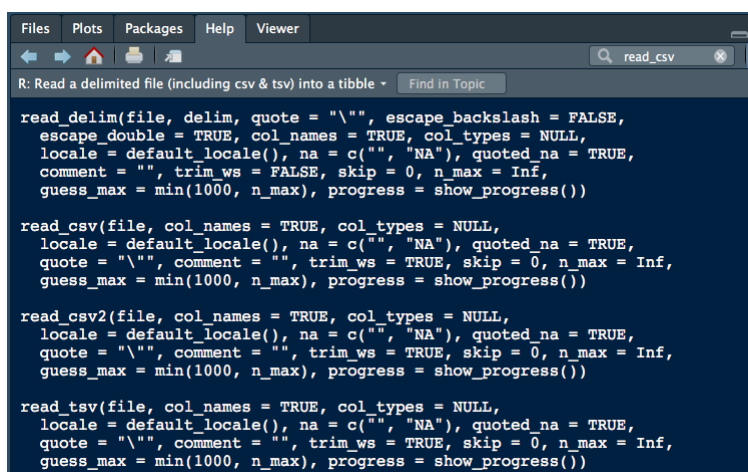
Para carregar, nós vamos usar a função `read_csv()` que vai pegar o arquivo e criar um objeto na memória do R.

```
dados <- read_csv(file = "inflamacao.csv", col_names = FALSE)
```



(#fig:env_dados)Environment com dados

A função `read_csv()` vem do pacote `readr`. Ela tenta automatizar tanto quanto é possível a importação dos dados. Até, ela analisa os dados para decidir qual é o tipo de dados que o conjunto tem: caráter, número inteiro, número decimal, lógico, etc. Também usa a primeira linha como o nome das variáveis (colunas da planilha). Neste caso, as colunas só têm dados, não nomes na primeira linha. Então, temos que avisar `read_csv()` disso. Assim, usamos o argumento `colnames = FALSE` porque o padrão é `colnames = TRUE`, como a tela de *Help* para `read_csv` mostra.



(#fig:help_read_csv)read_csv Help

6.2.5 Passo 5 — Qual É o Tipo do Conjunto dos Dados?

Para saber os tipos de análises que podemos fazer no `dados`, precisamos determinar o que temos. Sabemos que temos 40 variáveis que são os dias em que a avaliação foi feita. Primeiro nível de informação é sobre o tipo de dados que temos. Para isso, usamos a função `class()`.

```
class(dados)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Estes quatro descritores indicam que este é um *tibble*, um tipo avançado de `data.frame`. Este é o tipo básico de um conjunto de dados quadrado, com variáveis nas colunas e observações nas fileiras. As capacidades adicionais têm a maior significância para as operações internas do R e das funções do Tidyverse, mas pode ser muito mais rápido para conjuntos grandes de dados.

Existem duas outras funções que mostram mais detalhe sobre o conjunto. `str()` vai mostrar para todo o conjunto a classe (tipo) da variável e os primeiros valores dela.

`glimpse()` faz a mesma coisa mas com as linhas e colunas trocadas. `glimpse()` oferece uma visão mais completa que `str()`. Por causa do tamanho do output em nosso caso com 40 variáveis, vou limitar ela para cinco variáveis usando uma técnica que aprenderemos abaixo.

```
str(dados[, 1:5])
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    60 obs. of  5 variables:
## $ X1: num  0 0 0 0 0 0 0 0 0 0 ...
## $ X2: num  0 1 1 0 1 0 0 0 0 1 ...
## $ X3: num  1 2 1 2 1 1 2 1 0 1 ...
## $ X4: num  3 1 3 0 3 2 2 2 3 2 ...
## $ X5: num  1 2 3 4 3 2 4 3 1 1 ...
```

```
glimpse(dados[, 1:5])
```



```
## Observations: 60
## Variables: 5
## $ X1 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ X2 <dbl> 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1...
## $ X3 <dbl> 1, 2, 1, 2, 1, 1, 2, 1, 0, 1, 0, 0, 2, 0, 2, 1, 0, 0, 2, 2, 1...
## $ X4 <dbl> 3, 1, 3, 0, 3, 2, 2, 2, 3, 2, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 3...
## $ X5 <dbl> 1, 2, 3, 4, 3, 2, 4, 3, 1, 1, 4, 3, 4, 1, 1, 1, 2, 2, 2, 1, 1...
```

Também, podemos examinar o tamanho do conjunto usando a função `dim()` (dimensões). Esta retorna para você o número e linhas e colunas nos dados.

```
dim(dados)
```

```
## [1] 60 40
```

As variáveis são do mesmo tipo. Por isso, podemos olhar no primeiro cinco para ver suficiente detalhe sobre os dados.

6.2.6 Passo 6 — Subconjuntos (*Subsets*) de dados

Nós queremos focar nas primeiras cinco variáveis porque mais que isso é simplesmente repetitivo e desnecessária por enquanto. Têm duas maneiras que podemos fazer este foco, um que vem de R básico e o outro que introduza a gente para `dplyr` e a manipulação dos dados no Tidyverse.

Subconjuntos em R Básico – Índices

Podemos referir a um dado individual colocando um índice dentro de colchetes (`[]`), com a fileira primeiro e a coluna depois. Por exemplo, para o primeiro dado em `dados`, podemos dizer `dados[1,1]`. Se nós queremos as avaliações para os primeiros cinco dias (colunas) para paciente 1, podemos dizer `dados[1, 1:5]`. Os dois-pontos diz “todos os valores entre valor 1 e valor 5”. Se nós queremos as avaliações dos cinco primeiro pacientes (fileiras) no primeiro dia (coluna), podemos dizer `dados[1:5, 1]`. Se nós queremos ver todos as colunas ou todas as fileiras, simplesmente precisa deixar essa parte do índice em branco. Mas precisa incluir sempre a virgula ou R não vai

saber em qual dimensão você refere. Por exemplo, se queremos ver o valor para todos os dias de paciente 50, podemos dizer `dados[50,]`. O próximo bloco de código mostra todas essas opções.

```
dados[1,1]
```

```
## # A tibble: 1 x 1
##       X1
##   <dbl>
## 1     0
```

```
dados[1, 1:5] # 1º paciente, 1º cinco dias
```

```
## # A tibble: 1 x 5
##       X1     X2     X3     X4     X5
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     0     0     1     3     1
```

```
dados[1:5, 1] # 1º 5 pacientes, 1º dia
```

```
## # A tibble: 5 x 1
##       X1
##   <dbl>
## 1     0
## 2     0
## 3     0
## 4     0
## 5     0
```

```
dados[50, ] # paciente 50, todos os dias
```

```
## # A tibble: 1 x 40
##       X1      X2      X3      X4      X5      X6      X7      X8      X9     X10     X11     X12
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1      0      0      1      2      3      4      5      7      5      4     10      5
## # ... with 28 more variables: X13 <dbl>, X14 <dbl>, X15 <dbl>, X16 <dbl>,
## #   X17 <dbl>, X18 <dbl>, X19 <dbl>, X20 <dbl>, X21 <dbl>, X22 <dbl>,
## #   X23 <dbl>, X24 <dbl>, X25 <dbl>, X26 <dbl>, X27 <dbl>, X28 <dbl>,
## #   X29 <dbl>, X30 <dbl>, X31 <dbl>, X32 <dbl>, X33 <dbl>, X34 <dbl>,
## #   X35 <dbl>, X36 <dbl>, X37 <dbl>, X38 <dbl>, X39 <dbl>, X40 <dbl>
```

Subconjuntos no Tidyverse – dplyr

Este código parece razoável, mas não muito claro. Não segue as recomendações sobre nomes de variáveis por exemplo. É típico das linguagens de computação. Com o Tidyverse, podemos fazer este aparecer mais como um idioma normal. Nós vamos usar umas novas funções para fazer os subconjuntos que queremos.

Primeiro dessas funções permite que combinamos operações das funções diferentes na mesma expressão. Este símbolo está chamada o “pipe” e está escrito assim: `%>%` . Este conceito é muito comum em linguagens de computação, especialmente do sistema UNIX, apesar que está expressa por vários símbolos. Em R, o pipe quer dizer “aceite o resultado da operação no lado esquerdo e aplique a função no lado direito ao este resultado”. Com o uso do pipe, nós não precisamos repetir o nome do objeto do lado esquerdo da operação quando você especifica a função no lado direito.

O termo *pipe* não vem do sentido de tubo, mas da palavra “cachimbo” vindo de arte surrealista, especificamente do quadro do pintor belga René Magritte, “Ceci n’est pas une pipe”. Por quê? Porque veio originalmente do pacote `magritte` , mas agora está incorporado no `dplyr` , o pacote principal do `tidyverse` para a manipulação dos dados.



Para nosso exemplo, o código vai parecer complicado em comparação ao código acima, mas ele ilustra funções importantes que ajudaria você em projetos mais complicados. As funções que vamos usar são os seguintes:

- `slice()` – seleciona as linhas segundo a posição delas
- `select()` – seleciona as colunas por nome

```
# equivalente a dados[1,1]
```

```
dados %>%  
  slice(1) %>%  
  select(X1)
```

```
## # A tibble: 1 x 1  
##       X1  
##   <dbl>  
## 1     0
```

```
# equivalente a dados[1, 1:5]
```

```
dados %>%  
  slice(1) %>%  
  select(X1:X5)
```

```
## # A tibble: 1 x 5  
##       X1     X2     X3     X4     X5  
##   <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1     0     0     1     3     1
```

```
# equivalente a dados[1:5, 1]
```

```
dados %>%  
  slice(1:5) %>%  
  select(X1)
```

```
## # A tibble: 5 x 1
```

```
##       X1
```

```
##   <dbl>
```

```
## 1     0
```

```
## 2     0
```

```
## 3     0
```

```
## 4     0
```

```
## 5     0
```

```
# equivalente a dados[50, ]
```

```
dados %>%
```

```
  slice(50)
```

```
## # A tibble: 1 x 40
```

```
##       X1     X2     X3     X4     X5     X6     X7     X8     X9    X10    X11    X12
```

```
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
## 1     0     0     1     2     3     4     5     7     5     4    10     5
```

```
## # ... with 28 more variables: X13 <dbl>, X14 <dbl>, X15 <dbl>, X16 <dbl>,
```

```
## #   X17 <dbl>, X18 <dbl>, X19 <dbl>, X20 <dbl>, X21 <dbl>, X22 <dbl>,
```

```
## #   X23 <dbl>, X24 <dbl>, X25 <dbl>, X26 <dbl>, X27 <dbl>, X28 <dbl>,
```

```
## #   X29 <dbl>, X30 <dbl>, X31 <dbl>, X32 <dbl>, X33 <dbl>, X34 <dbl>,
```

```
## #   X35 <dbl>, X36 <dbl>, X37 <dbl>, X38 <dbl>, X39 <dbl>, X40 <dbl>
```

Uma vantagem da gramática do Tidyverse com o pipe é que facilita a explicação da operação do comando. Por exemplo, a última expressão pode ser lida como: “começa com o conjunto `dados` ; depois utiliza as primeiras 5 fileiras e depois seleciona a primeira variável”. Quando você tem expressões complicadas, que usam muitos verbos (palavra que o Tidyverse usa para referir às ações como `slice` ou `select`), esta habilidade de ler a expressão em quase-português faz a programação mais compreensível.

Você pode ver que as variáveis têm nomes estranhos, `x1` , etc. Para nossos fins, este não é horrível porque refere corretamente ao número de dia, 1 até 40. Entretanto, podemos fazer melhor. No próximo bloco, nós vamos dar nomes mais claros às variáveis introduzindo o conceito de um “*loop*” ou uma série repetitiva das ações que

podemos executar repetidamente.

6.2.7 Novos Nomes para as Variáveis — Loops

Nomes mais claros para as variáveis ajudaria nosso entendimento do conjunto `dados` e faria a apresentação dos dados mais profissional. Existem várias técnicas que podemos usar para fazer, mas é um bom momento para ver um loop, um dos conceitos mais básicos da programação. Nós vamos criar um loop de tipo “*for*”, que vai repetir comandos um certo número de vezes. O loop tem o formato:

```
for (expressão que define o número de repetições) { código para executar dentro do loop }
```

Queremos que o loop considerar todas as variáveis, o número de que podemos determinar automaticamente com a função `seq_along()`. Esta função define uma sequência que começa com 1 e continua até o número das variáveis. Dentro do loop nós vamos criar um vetor com 40 cadeias de caracteres composto da palavra “dia” mais um número para o dia. Para fazer isso, usamos a função `paste0()` que concatena os argumentos da função e produz uma cadeia de caracteres. O número resultará do contador das iterações do loop, `i`. `paste0` difere da função `paste()` só pelo fato que `paste()` insere um espaço entre os elementos da função mas `paste0()` não, como implica o nome da função. Quando o loop completa as 40 iterações, ele para e passa controle para a próxima expressão no programa. Nesta linha, a função `col_names()` vai transferir o vetor dos nomes para `dados` como novos nomes das variáveis. [^6]

[^6] O Tidyverse tem uma função equivalente, `glue()`, que vem do pacote homônimo. O funcionamento é em grande parte o mesmo. Pessoalmente, quando trabalho só com concatenação de textos, uso `paste()` e quando estou combinando o resultado dos cálculos com texto, uso `glue()`. Mas, aqui, para consistência, ficarei com `paste()` e `paste0()`.

Antes de começar a operação do loop, devemos instruir R onde colocar os nomes que o loop está compondo. Vou criar uma variável `nome` que será um vetor vazio. Durante a operação do loop, ele vai acrescentar os novos valores usando a função `c()`.

```

nome <- vector(mode = "character")
for (i in seq_along(dados)) {
  nome <- c(nome, paste0("dia", i))
}
colnames(dados) <- nome
str(dados[, 1:5])

```

```

## Classes 'tbl_df', 'tbl' and 'data.frame':    60 obs. of  5 variables:
## $ dia1: num  0 0 0 0 0 0 0 0 0 0 ...
## $ dia2: num  0 1 1 0 1 0 0 0 0 1 ...
## $ dia3: num  1 2 1 2 1 1 2 1 0 1 ...
## $ dia4: num  3 1 3 0 3 2 2 2 3 2 ...
## $ dia5: num  1 2 3 4 3 2 4 3 1 1 ...

```

Deixa-me explicar todos as linhas e os elementos deste código:

- `nome <-` a linha que especifica o vetor para conter os resultados do loop
- `for` a definição do loop; a linha termina com a chave que inicia o bloco de código que o loop vai executar
- `(i in seq_along(dados))` instrução para substituir a variável `i` no bloco de código com os números definidos pela expressão `seq_along`
- `seq_along(dados)` - a sequência a ser definido que representa todos as variáveis conforme o cálculo abaixo

```
seq_along(dados)
```

```

## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

```

- `nome <- c(nome, paste0("dia", i))` Concatenar a palavra “dia” com o valor de `i` (e.g. “dia1”) e acrescentar este valor ao vetor `nome`
- `}` indicação do fim do bloco de código; quando encontra isso, o loop volta ao

início e processa o próximo valor de `i`

- `colnames(dados) <- nome` Quando termina o loop, dar novos nomes às colunas (variáveis) de `dados`
- `str(dados[, 1:5])` Mostra a nova estrutura das primeira cinco colunas do tibble `dados` para confirmar que fizemos a operação de dar novos nomes com êxito.

6.3 Análise de Dados de Inflamação

Vamos fazer um resumo dos dados e olhar em alguns gráficos para poder entender os dados melhor e preparar para tirar conclusões sobre o novo tratamento. Só para esta apresentação dos dados e das funções, vamos limitar os resumos aos dias 20 até dia 25. Mas os gráficos incluirá todos os dias (variáveis). Uma função básica para resumir dados é `summary()` que relata os valores seguintes:

- valor mínimo da variável (`min`)
- 1º quartil (25º percentil)
- mediana (50º percentil) (`median`)
- 3º quartil (75º percentil)
- valor máximo (`max`)

Também, existem muito outros resumos dos dados em R básico e nos pacotes que já fez o download. Durante o curso, vai encontrar vários desses pacotes e funções. O que `summary()` não relata que é muito importante é uma medida de variabilidade.

Normalmente, usamos o desvio padrão para medir isso. Precisamos calcular isso separadamente de `summary()` usando a função `sd()` (*standard deviation*). Vamos calcular um resumo dos dados depois de fazer o subset que queremos.

Mas, para fazer este cálculo de `sd()`, podemos usar um loop, mas tem outra maneira de conseguir a mesma coisa. O objetivo é calcular uma função para um número de variáveis em uma única operação. A família das funções `map()` calcula uma função (neste caso `sd()`) para todas as variáveis de nosso subset das variáveis. A função é muito simples e flexível. `map()` normalmente produz um tipo de dados chamado uma lista. Nós não queremos isso; queremos um vetor dos desvios padrão. Para conseguir isso, nós usamos o variante `map_dbl()` que produz um vetor dos números reais.

Estamos colocando isso numa expressão *tidy* em que o pipe entrega o subset (`sub_dados`) para a função `map_dbl()`, a expressão só precisa o nome de função que

queremos calcular (`sd`) sem os parênteses.

```
sub_dados <- dados %>%  
  select(dia20:dia25)  
glimpse(sub_dados)
```

```
## Observations: 60  
## Variables: 6  
## $ dia20 <dbl> 18, 6, 10, 7, 17, 12, 6, 19, 11, 8, 18, 11, 15, 16, 9, 17,...  
## $ dia21 <dbl> 6, 18, 19, 17, 9, 12, 9, 20, 6, 18, 8, 9, 13, 15, 12, 9, 1...  
## $ dia22 <dbl> 13, 4, 14, 4, 14, 5, 17, 8, 16, 15, 15, 16, 12, 9, 13, 8, ...  
## $ dia23 <dbl> 11, 12, 12, 4, 9, 18, 15, 5, 12, 16, 15, 18, 8, 11, 10, 12...  
## $ dia24 <dbl> 11, 5, 17, 7, 7, 9, 8, 13, 6, 14, 16, 6, 7, 4, 4, 11, 17, ...  
## $ dia25 <dbl> 7, 12, 7, 6, 13, 5, 9, 15, 8, 12, 11, 12, 4, 6, 12, 11, 5,...
```

```
summary(sub_dados)
```

```
##      dia20      dia21      dia22      dia23  
## Min.   : 5.00   Min.    : 5.00   Min.    : 4.00   Min.    : 4.00  
## 1st Qu.: 8.75   1st Qu.: 9.00   1st Qu.: 8.00   1st Qu.: 7.75  
## Median :13.00   Median :14.00   Median :13.00   Median :11.00  
## Mean   :12.35   Mean    :13.25   Mean    :11.97   Mean    :11.03  
## 3rd Qu.:16.00   3rd Qu.:16.25   3rd Qu.:15.25   3rd Qu.:15.00  
## Max.   :19.00   Max.    :20.00   Max.    :19.00   Max.    :18.00  
##      dia24      dia25  
## Min.    : 4.00   Min.    : 4.0  
## 1st Qu.: 6.75   1st Qu.: 7.0  
## Median :10.00   Median :10.5  
## Mean    :10.17   Mean    :10.0  
## 3rd Qu.:13.25   3rd Qu.:13.0  
## Max.    :17.00   Max.    :16.0
```

```
sub_dados %>% map_dbl(sd)
```

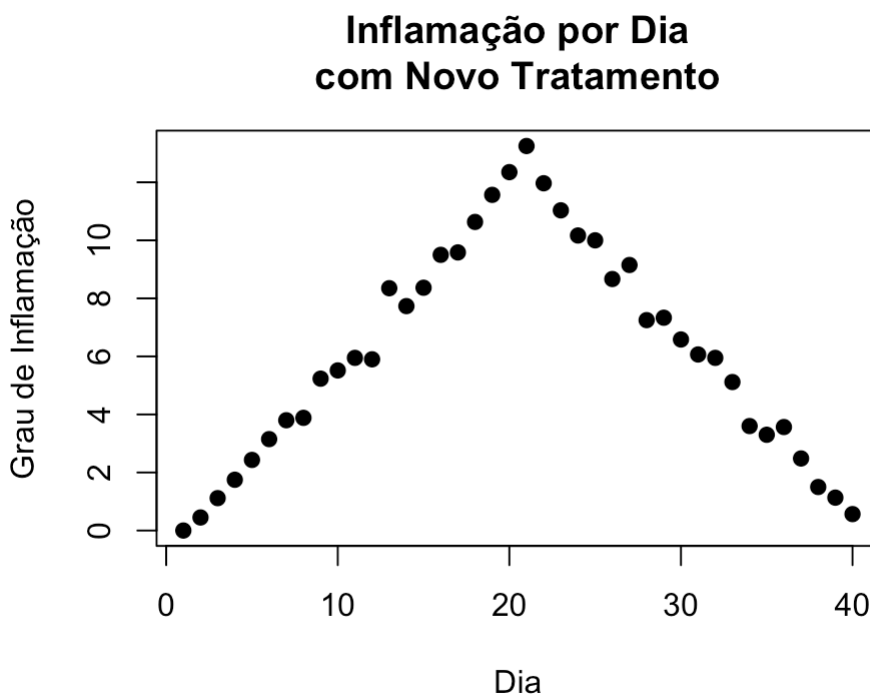
```
##      dia20      dia21      dia22      dia23      dia24      dia25
## 4.539189 4.276958 4.587997 4.234510 4.134053 3.741657
```

6.4 Gráficos do Conjunto dos Dados

Com um conjunto de dados intensivo, olhando nas tabelas e resumos não ajuda muito no entendimento do que dizem os dados. Gráficos comunicam melhor. Primeiro, queremos ver a tendência da inflamação no decorrer do estudo. Cresceu, diminuiu, quando, por quanto? Para descobrir isso, podemos calcular a média para cada dia e mostrar isso num gráfico de dispersão (*scatter plot*). Para este gráfico, nós vamos ficar com o plotagem simples de base R. Apesar disso, o gráfico seria o resultado de um expressão *tidy* que calcula as médias dos dias e entregam eles para a função `plot()`. Os argumentos incluídos aqui para a plotagem tem a ver com a apresentação, como títulos (`main` , `xlab` , `ylab`) e estilo de caráter (`pch`).

Anote também que usamos a função `map_dbl()` de novo para calcular as médias e entregar esses valores para comando gráfico.

```
dados %>%
  map_dbl(mean) %>%
  plot(type = "p", pch = 19,
       main = "Inflamação por Dia\ncom Novo Tratamento",
       xlab = "Dia",
       ylab = "Grau de Inflamação")
```



(#fig:graf_scatter)Gráfico de Dispersão

6.5 Whew! Final da Introdução

Em pouco tempo, você já fez bastante R. Vale a pena tentar os blocos de código em casa e talvez ler a apostila mais de uma vez. Ao final do curso, este documento seria muito maior, com minhas notas para todas as aulas gravadas aqui também. Vai ser um pequeno livro sobre R.

Você olhou em cálculos simples e um conjunto de dados mais avançado. Você viu várias funções. Quando estamos juntos, nós vamos aplicar estas técnicas e muitas outras para problemas relacionados à biomedicina. Se você tem alguns assuntos que quer que eu cubra nas aulas, por favor entre em contato. Estou sempre aberto às sugestões.

4. Agradeço Software Carpentry pelo uso desses dados.

(<http://swcarpentry.github.io/>)↩

5. Se o browser quer que você use o sufixo “txt” para seu arquivo, precisa insistir no uso de “csv”. Um arquivo “csv” é um tipo de arquivo de texto simples, mas outros programas como Excel só querem ver o sufixo “csv”.↩

