

# Zen do R

Caio Lente + Curso-R

Última atualização: 12/07/2021



# Contents

<b>Respire fundo</b>	<b>5</b>
<b>1 Introdução</b>	<b>7</b>
1.1 Sobre o livro . . . . .	7
1.2 Pré-requisitos . . . . .	8
1.3 Principais referências . . . . .	9
<b>2 .RData e .Rhistory</b>	<b>11</b>
2.1 Por que desistir . . . . .	11
2.2 Como desativar . . . . .	12
<b>3 Rproj e diretórios</b>	<b>15</b>
3.1 Caminhos relativos e absolutos . . . . .	15
3.2 Organizando o ambiente . . . . .	16
3.3 Diretório de trabalho . . . . .	18
<b>4 Git e GitHub</b>	<b>21</b>
<b>5 Funções e dependências</b>	<b>27</b>
5.1 Quatro-pontos . . . . .	30
<b>6 Pacotes</b>	<b>33</b>
6.1 Documentação . . . . .	35
6.2 Imports . . . . .	37

<b>7</b>	<b>Pipe</b>	<b>39</b>
7.1	Como funciona . . . . .	40
7.2	Vantagens . . . . .	41
<b>8</b>	<b>Data e data-raw</b>	<b>45</b>
8.1	Documentação . . . . .	47
<b>9</b>	<b>Testes automatizados</b>	<b>49</b>
<b>10</b>	<b>Versões e releases</b>	<b>51</b>
10.1	Versionamento semântico . . . . .	52
10.2	Releases . . . . .	54

# Respire fundo

Este é o *Zen do R*, seja bem-vinda(o)! O objetivo deste livro é ensinar ao leitor que não costuma programar algumas formas simples de melhorar a organização de seus projetos de análise de dados em R.

Ele foi escrito em RMarkdown com o pacote bookdown e está em **construção** e em **revisão aberta**. Fique à vontade para corrigir qualquer tipo de erro que encontrar no nosso material.

O texto deste livro foi elaborado por Caio Lente com o suporte, ajuda e revisão do time da Curso-R. Ele está disponível gratuitamente no Github sob a licença GNU GPLv3.

A Curso-R é o braço de Educação da R6 Consultoria, uma empresa de treinamentos e consultoria em Ciência de Dados e linguagem de programação R. Surgimos em 2015, com cursos de introdução ao R no Programa de Cursos de Verão do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP).

- Conheça os nossos cursos: <https://www.curso-r.com/cursos/>
- Conheça o nosso blog: <https://www.curso-r.com/blog/>
- Conheça o restante do nosso material: <https://www.curso-r.com/material/>
- Saiba mais sobre a Curso-R: <https://www.curso-r.com/sobre/>

Participaram da construção deste livro:

- Athos Damiani
- Beatriz Milz
- Caio Lente
- Daniel Falbel
- Fernando Correa
- Julio Tricenti
- William Amorim

Seja bem-vinda(o) também ao #rstats! Compartilhe seu desenvolvimento. Poste seus avanços nas redes sociais, compartilhe suas dúvidas em fóruns, procure e participe dos grupos e comunidades de programadores.

- Comunidade RLadies: <https://benubah.github.io/r-community-explorer/rladies.html>
- Fórum de dúvidas da Curso-R: <https://discourse.curso-r.com/>
- Grupo de divulgação da Curso-R no Telegram: <https://t.me/r6cursor>
- Grupo R Brasil no Telegram: <https://t.me/rbrasiloficial>

# Chapter 1

## Introdução

O *Zen do R* tem o objetivo de ser um livro sobre programação para não-programadores. Atualmente muitas pessoas de diferentes áreas do conhecimento acabam precisando usar a linguagem R por causa do seu grande potencial para programação estatística, mas ficam perdidas depois que aprendem o básico sobre a linguagem. Nesse sentido, este é um livro para “não-programadores” no que se refere a treinamento formal; isto não é uma introdução ao R, mas sim um guia sobre como usar o R de forma eficiente no dia-a-dia. O *Zen do R* também pretende destoar dos manuais mais secos e técnicos sobre programação, utilizando uma linguagem leve e acessível justamente porque parte-se do princípio de que ele será utilizado em conjunto com algum outro texto (seco e técnico) sobre como de fato programar em R.

A escolha do R é parcialmente arbitrária. Nada impede de você usar Python para a análise de dados, mas ao longo de alguns anos de experiência os autores notaram que o fato de o R ter sido feito com análise de dados em mente acaba sendo uma vantagem muito difícil de ignorar. Fora isso, o *tidyverse*, as infinitas ferramentas do RStudio e o engajamento da comunidade fazem com que nós achemos o R a verdadeira linguagem do *data science*.

Por isso, o *Zen do R* é um guia para acalmar os nervos daqueles que se aventuram pela primeira vez em um grande projeto em R.

### 1.1 Sobre o livro

A grande piada do título é que o caminho para o fluxo ideal de programação é análogo ao caminho descrito pelo Budismo para a libertação do espírito. Parece muito estranho que essas duas coisas tenham alguma relação, mas a vida é uma boa metáfora para muitos processos que encontramos no dia-a-dia!

### 1.1.1 O que você vai aprender

Essencialmente você vai aprender a usar alguns pacotes e como trabalhar com quatro aspectos do processo de análise de dados: ambiente, versões, dados e arquivos. Não é necessário ler o livro em ordem porque cada uma dessas quatro sessões são completamente independentes.

Sinta-se livre para pular todos os tópicos sobre os quais você sentir que já sabe o suficiente. Mas não se iluda, porque nenhum dos tópicos é inútil ou pode ser simplesmente ignorado; quanto maior for um projeto (seja uma tese de mestrado ou uma análise de dados médicos), mais necessários serão os tópicos mais avançados.

### 1.1.2 O que você não vai aprender

Primeiramente, você não vai aprender a programar R. Esse assunto é extremamente extenso e já existem livros o suficiente para ajudar com isso (vide o livro da Curso-R ou R for Data Science), então não vou me preocupar com os detalhes do código ou com qual guia de estilo seguir.

Se você gosta de fazer códigos longos e velozes usando o **base-r**, se você ama as pipelines do **tidyverse**, se você paraleliza todos os seus loops... Nada disso importa aqui; não vou dizer qual é o melhor jeito de programar<sup>1</sup>. Aqui você vai conhecer somente as melhores ferramentas para **organizar o seu fluxo de programação**.

Apesar de terem nomes parecidos, *O Zen do R* e *O Zen do Python* são diferentes justamente por causa disso. O livro do Python pretende dar sugestões de como organizar e escrever o seu código, o que não será feito aqui.

## 1.2 Pré-requisitos

Como talvez já tenha ficado claro, um dos principais pré-requisitos deste livro é saber a programar pelo menos um pouco de R. Você não precisa ser um profissional, mas, para ter a necessidade de melhorar o seu fluxo de análise, você antes precisa estar fazendo alguma análise.

Fora isso, o segundo principal pré-requisito é um ambiente de desenvolvimento. Grande parte das dicas do livro são baseadas em funcionalidades integradas ao RStudio, então se você quiser tirar o maior proveito possível dos ensinamentos talvez valha à pena instalar a IDE. Ainda no tocante ao R, você precisará ter instalado pelo menos os três pacotes a seguir:

---

<sup>1</sup>E também não tenho paciência para entrar em mais nenhum debate “base vs. tidyverse”



```
# Conteúdo principal do livro  
install.packages(c("usethis", "renv", "tidyverse"))  
  
# Se você quiser reproduzir os exemplos  
install.packages("devtools")
```

Você também precisa de um computador funcionando com um sistema operacional razoavelmente moderno. E uma conexão à internet.

## 1.3 Principais referências

O *Zen do R* se baseia em inúmeras referências que normalmente serão citadas juntamente com o próprio conteúdo. Mas algumas mais gerais acabariam sendo citadas o tempo todo e portanto acabarão ficando aqui:

- R for Data Science;
- bookdown: Authoring Books and Technical Documents with R Markdown;
- usethis;
- Packrat: Reproducible package management for R e
- O blog da Curso-R.



## Chapter 2

# .RData e .Rhistory

O fluxo ideal de análise de dados começa na escolha da ferramenta. Por ser uma linguagem especializada em estatística, o R é a primeira escolha de muitos usuários. Normalmente optar por programar em R também implica na escolha de uma IDE (*Integrated Development Environment*) que, em 90%<sup>1</sup> dos casos, será o RStudio.

O R, em combinação com o RStudio, possui um conjunto de funcionalidades cuja intenção é ajudar no processo de desenvolvimento. Entretanto, isso acaba deixando os programadores de R mal acostumados.

Como um pai coruja, o RStudio faz questão de lembrar tudo o que você fez anteriormente. Em sua configuração padrão, a IDE manterá na “memória” todos os últimos comandos executados, todos os dados utilizados e todos os objetos criados. Ao fechar e abrir o RStudio, essas informações serão recarregadas na memória como se o usuário nunca tivesse saído do programa.

Esse recurso é tornado possível pela criação de dois arquivos ocultos: `.RData` e `.Rhistory`. O primeiro abriga absolutamente todos os objetos criados por uma sessão R, enquanto o segundo contém uma lista com os últimos comandos executados. Ao reabrir o RStudio, o conteúdo armazenados nestes arquivos será carregado no ambiente de trabalho atual como se nada tivesse acontecido.

### 2.1 Por que desistir

Apesar de ser uma ótima conveniência, assim como o pai coruja, esse tipo de funcionalidade pode deixar o programador mal acostumado. Se todos os resultados parciais de uma análise estiverem disponíveis a qualquer momento, diminui o

---

<sup>1</sup>Não tenho nenhuma estatística confiável sobre esse número, mas sei que ele não é 100% porque conheço pelo menos uma pessoa que programa R no neovim e eu passei a usar emacs.

incentivo para a escrita de *código reprodutível* e, se todo o histórico de comandos for acessível, acaba a necessidade de experimentos controlados.

Um usuário que dependa ativamente do **.RData** para recuperar seus dados estará aos poucos contando cada vez mais com a sorte. Caso ele acidentalmente sobrescreva o objeto relevante e o código para recriá-lo já tenha sido apagado, não haverá nenhuma forma confiável de recuperar esses dados. Idealmente, todo o código necessário para uma análise de dados deve estar salvo em um arquivo **.R** perfeitamente reprodutível; assim, caso o programador cometa um engano, é possível executar aquele arquivo do início e obter novamente os objetos que estavam sendo utilizados.

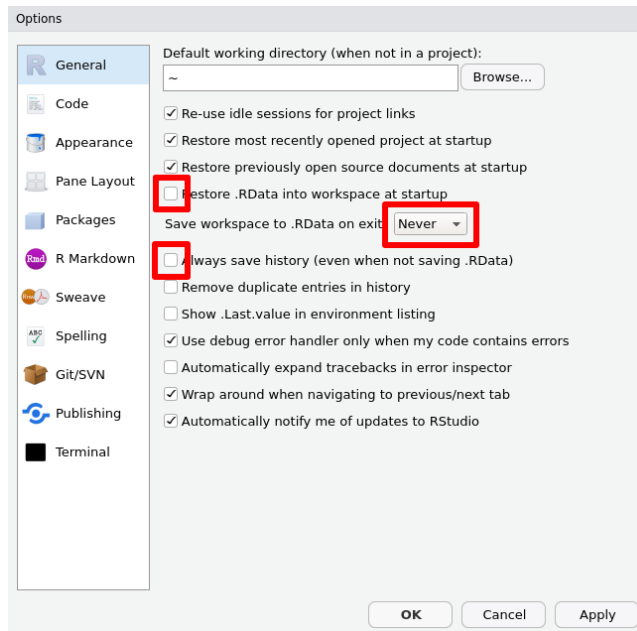
Arquivos reprodutíveis também tem uma outra vantagem: facilidade de compartilhamento. A menos que o programador pretenda sentar com seu colega para explicar como utilizar os objetos do **.RData** e do **.Rhistory**, não pode-se esperar que outra pessoa seja capaz de reproduzir uma análise a partir de arquivos incompletos. Deste modo, abandonar essa funcionalidade permite utilizar ferramentas de compartilhamento e controle de versão da maneira como elas foram idealizadas.

Por fim, é importante notar uma desvantagem sutil, mas muito relevante do uso do **.RData**. O R trata todos os objetos guardados na memória igualmente, sem levar em conta sua utilidade ou tamanho. Isso significa que ele também irá armazenar nos arquivos ocultos todas as bases de dados da sessão (não importando quão grande sejam). Isso faz com que o **.RData** normalmente seja um arquivo de múltiplos gigabytes.

Ao reabrir o RStudio, todos esses dados serão recarregados e provavelmente farão com que o programador espere vários minutos até que ele possa voltar ao seu trabalho. Com o **.RData** é impossível ter controle sobre quais dados devem ser utilizados em cada sessão de programação.

## 2.2 Como desativar

O processo de desabilitar o **.RData** e o **.Rhistory** é bastante simples e afeta todos os projetos do computador, então só é necessário passar por ele uma vez. Basta selecionar **Tools > Global Options...** na aba de ferramentas do RStudio e então ajustar três configurações. No final a página de opções gerais deve ficar similar à da imagem abaixo:



Se acostumar com sessões efêmeras não é uma tarefa fácil e um desconforto inicial é esperado. Pode ser que o programador ache entediante executar o mesmo código toda vez que abrir seu RStudio, mas é importante ter em mente que esse programa só está lá e pode ser executado inúmeras vezes porque o R não estará mais fazendo o trabalho do analista.



## Chapter 3

# Rproj e diretórios

Um programador iniciante corre o risco de não gerenciar seus projetos. Muitas vezes seus arquivos de código ficarão espalhados pelos infinitos diretórios de seu computador, esperando a primeira oportunidade de sumir para sempre. No R isso não é diferente: organizar arquivos é uma parte integral do processo de programação.

Felizmente o RStudio possui uma ferramenta incrível que auxilia na tarefa de consolidar todos os recursos necessários para uma análise. Denominados “projetos”, eles não passam de pastas comuns com um arquivo `.Rproj`.

### 3.1 Caminhos relativos e absolutos

Antes de continuar falando sobre diretórios, é importante falar um pouco sobre como funcionam os caminhos para arquivos no computador. Toda função de importação vai exigir um caminho, uma string que representa o endereço do arquivo no computador e há duas formas de passarmos um caminho de arquivo: a absoluta e a relativa.

Caminhos absolutos são aqueles que têm início na pasta raiz do seu computador, ou seja, que começam com uma barra / ou um disco do Windows (C:/, D:/ e assim por diante). Por exemplo, esse é o caminho absoluto para a pasta onde este livro foi produzido:

```
getwd()
#> [1] "/home/clente/Documents/zen-do-r"
```

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.

Já caminhos relativos são aqueles que têm início no diretório de trabalho da sua sessão, ou em outras palavras, no seu *working directory*. Essa é a pasta em que o R vai procurar arquivos na hora de ler informações ou gravar arquivos na hora de salvar objetos. O caminho relativo para o seu diretório de trabalho é sempre `..`:

```
R.utils::getRelativePath(getwd())  
#> [1] "."
```

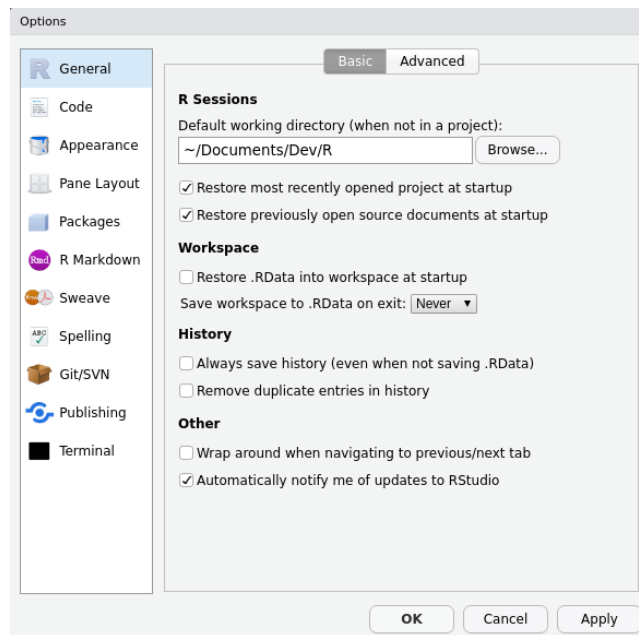
Um arquivo dentro do diretório de trabalho será o seu próprio caminho relativo, enquanto `..` indica “voltar” um diretório em relação ao *working directory*. Por fim, o til `~` é um caractere que indica a *home* do usuário (`/home/clente` no meu caso) e costuma ser classificado como caminho relativo.

```
R.utils::getRelativePath("/home/clente/Documents/zen-do-r/git-github.Rmd")  
#> [1] "git-github.Rmd"  
  
R.utils::getRelativePath("/home/clente/Documents")  
#> [1] ".."  
  
R.utils::getRelativePath("/home/clente/Documents/blog")  
#> [1] "../blog"  
  
R.utils::getRelativePath("~/Documents/blog")  
#> [1] "../blog"
```

## 3.2 Organizando o ambiente

O primeiro passo para organizar um bom ambiente de trabalho para análises de dados é criar um diretório onde todos os seus programas R podem viver. No meu computador eu tenho a pasta `~/Documents/Dev/R/`, mas não importa onde ela está, apenas que seja um lugar o qual você lembre sem dificuldades. Tendo criado esse *workspace*, é importante registrá-lo como o seu ambiente de trabalho no RStudio: basta selecioná-lo em **Tools > Global Options... > General > R Sessions > Browse**.





Desta forma, toda vez que você iniciar um novo projeto no R, ele por padrão usará essa nova pasta como ambiente padrão! Se você já tiver alguns arquivos espalhados pelo seu computador, traga eles para o seu diretório de programas.

O segundo passo no processo de organização dos seus projetos é um pouco mais complexo e demanda mais atenção. Não basta juntar todos os arquivos em um só lugar, é importante colocá-los em subdiretórios para que a sua pasta não vire um equivalente virtual a uma mesa desorganizada. Assim como em uma mesa cada papel e cada utensílio tem uma gaveta, cada arquivo precisa fazer parte de um projeto.

É nesse ponto que os “projetos” do RStudio dialogam com os projetos da vida real. Em uma empresa, cada cliente é um projeto; na academia, cada pesquisa é um projeto; e assim por diante. Cada projeto seu deve ter a sua própria pasta para que seja fácil encontrar todos os códigos e dados pertencentes a um único assunto. Mas esta não deve ser uma pasta comum, ela deve ser um projeto.

O código listado abaixo demonstra como criar um projeto no RStudio. Basta apenas um comando e ele já fará tudo que for necessário para preparar o seu ambiente de desenvolvimento.

```
usethis::create_project("~/Documents/Dev/R/Proj/")
#> Creating '~/Documents/Dev/R/Proj/'
#> Setting active project to '~/Documents/Dev/R/Proj'
#> Creating 'R/'
#> Writing 'Proj.Rproj'
```

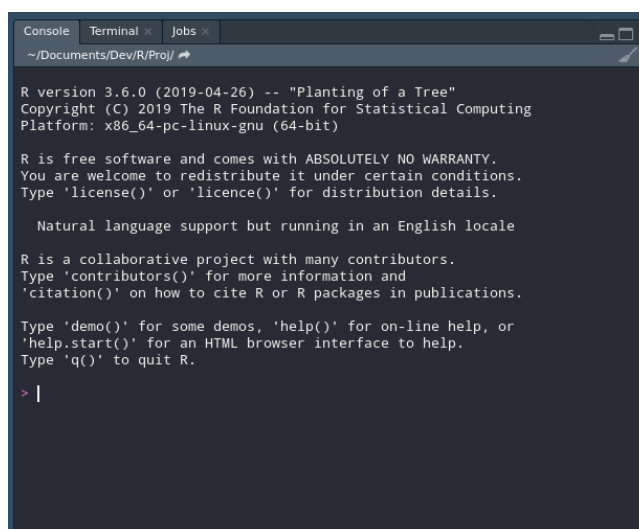
```
#> Adding '.Rproj.user' to '.gitignore'  
#> Opening '~/Documents/Dev/R/Proj/' in new RStudio session  
#> Setting active project to 'Proj'
```

Cada linha da saída do comando representa algo que ele fez para preparar o projeto. A mais importante é a quarta linha, que cria o arquivo `Proj.Rproj`; ele indica para o RStudio que aquele diretório será a raiz de um projeto e que, portanto, várias outras funcionalidades podem ser ativadas. Por exemplo, clicando duas vezes neste arquivo já carrega o RStudio com os arquivos de `Proj`.

Importante também é a pasta `R/` criada. Nela você deve colocar todos os seus arquivos de código referentes àquele projeto com nomes que descrevam bem o que cada um faz. Se você seguiu o conselho anterior e juntou todos os seus códigos no seu diretório de trabalho, crie um projeto novo para cada grupo de programas que você tiver detectado. Talvez um para exercícios de R, um para cada cliente, um para uma nova ideia, etc. Cada um deles deve ter um nome descritivo e conter, em sua pasta `R`, todos os arquivos necessários para aquela análise.

### 3.3 Diretório de trabalho

Mas a funcionalidade mais importante dentre todas as já citadas é o conceito do *working directory* ou diretório de trabalho. No canto esquerdo superior do Console do RStudio existe um caminho denominado diretório de trabalho, que é essencialmente a raiz do seu projeto. Muitos programadores que aprenderam R há muito tempo conhecem uma função chamada `setwd()`; se você nunca ouviu falar disso, não se preocupe e continue assim, mas se você costuma usá-la, siga prestando atenção.

A screenshot of an R console window. The window has three tabs: 'Console', 'Terminal', and 'Jobs'. The 'Console' tab is active, showing the R startup message. The path in the title bar is '~/Documents/Dev/R/Proj/'. The message includes the R version (3.6.0), copyright information, and instructions on how to use R, including commands like 'license()', 'demo()', 'help()', and 'q()'. The prompt '> |' is visible at the bottom.

```
R version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

O R dá a possibilidade de mudar, instantaneamente, o diretório de trabalho. Isso que dizer que os caminhos relativos para arquivos podem mudar em questão de linhas. O exemplo abaixo demonstra superficialmente como isso funciona:

*# Abrindo dois arquivos em diretórios diferentes*

```
setwd("~/Downloads")
a <- read.csv("a.csv")

setwd("~/Documents/Dev/R/Proj")
b <- read.csv("b.csv")

write.csv(a, "c.csv")
```

Isso não parece tão problemático à primeira vista, mas usar `setwd()` cria um vício difícil de abandonar. Essa função estimula que os seus projetos continuem desorganizados com arquivos espalhados pelo computador, mas também confunde o programador na hora de salvar arquivos. Onde será salvo o arquivo `c.csv`? De onde veio o arquivo `a.csv` ou de onde veio o `b.csv`? E se essa linha fosse mudada de lugar para antes do segundo `setwd()`? São perguntas difíceis de responder caso você não esteja atento ao código todo.

A solução que os projetos oferecem para isso é fazer com que o diretório de trabalho seja sempre a pasta do projeto. Neste caso é como se, ao abrir o RStudio, ele executasse o comando `setwd("~/Documents/Dev/R/Proj")` automaticamente. Mas como ler então o arquivo `a.csv`?

```
# Duas formas de ler a.csv

a <- read.csv("~/Downloads/a.csv")

file.copy("~/Downloads/a.csv", "a.csv")
a <- read.csv("a.csv")
```

A primeira forma deixa explícito que aquele arquivo não faz parte do projeto e que portanto deve ser tratado como temporário. A segunda forma, mais indicada, é trazer o arquivo para dentro do projeto! Se ele é importante, é essencial que ele esteja junto com todos os outros dados de Proj. Com o código acima, o comando `write.csv(a, "c.csv")` salvaria `c.csv` dentro do projeto sem sombra de dúvidas.

Os principais benefícios de não usar `setwd()` são dois: saber sempre onde os arquivos utilizados estão/serão salvos e poder compartilhar um projeto com qualquer pessoa. `setwd()` depende que seja explicitado um caminho dentro do seu computador e isso nem sempre é verdade no computador de outra pessoa; fazendo com que todos os arquivos estejam no projeto e com caminhos relativos nos códigos permite que outro usuário replique a sua análise sem ter que modificar nem uma linha do programa.

## Chapter 4

# Git e GitHub

Há poucas coisas mais frustrantes no mundo do que ter que refazer um trabalho. Perder progresso já feito por algum erro ou acidente transforma qualquer pacifista em um vulcão prestes a entrar em erupção. Quando se trata de programação, há várias formas de isso acontecer: um disco rígido que falha, o copo de café derramado no lugar errado, aquela alteração que não pode ser desfeita.

Este problema está longe de ser novo. Em 2005, Linus Torvalds (o criador do Linux) se deparava com essas questões durante o seu desenvolvimento do kernel Linux. Muitas pessoas contribuindo para um mesmo código, fazendo alterações que deveriam ser revistas e possivelmente revertidas, não é uma tarefa facilmente solucionável com métodos convencionais de armazenamento de arquivos. Com isso em mente, Torvalds criou o sistema de controle de versão distribuído conhecido como Git.

Em termos leigos, o Git permite gerenciar versões de arquivos texto (outros tipos também são suportados, mas o foco principal é em arquivos de código). Ele não passa de um programa para linha de comando que observa as mudanças nos arquivos de um diretório e vai guardando essas informações para que seja possível reverter qualquer alteração indesejada. O Git também pode se conectar a um serviço de hospedagem e armazenar todas as versões de um código fora do seu computador; o mais utilizado atualmente se chama GitHub.

Na prática, a utilização do Git e do GitHub tem dois principais benefícios:

- nunca mais precisar controlar versões com `analise.R`, `analise_v2.R`, `analise_v3.R`, `analise_final.R`, `analise_final_final.R`, `analise_final_revisada.R`...
- nunca mais precisar se preocupar em perder seus projetos por causa de falhas no seu computador.

Nada mal para dois serviços gratuitos!

No capítulo anterior, é apresentado o conceito de projeto. Agora o segundo passo é entender como esses projetos podem ser utilizados em conjunto com controle de versão para manter seu trabalho sempre sincronizado na nuvem. Criar uma conta no GitHub e instalar o programa `git` no seu computador são necessários para poder utilizar os recursos descritos a seguir. A partir daqui, assume-se que ambos os requisitos foram cumpridos.

Para permitir que os comandos do R acessem a sua conta do GitHub, é essencial criar um *Personal Access Token* (PAT). Tendo logado no GitHub, clique na sua imagem no canto direito superior e siga para **Settings > Developer settings > Personal access tokens > Generate new token**. Nesta página, basta descrever o seu uso para o token e selecionar o primeiro box de todos; por fim, gere e copie o seu token (uma sequência de mais ou menos 40 letras e números). Se você estiver sem nenhuma paciência, execute o comando abaixo:

```
usethis::create_github_token()
#> Call `gitcreds::gitcreds_set()` to register this token in the local Git credential
#> It is also a great idea to store this token in any password-management software t
#> Opening URL 'https://github.com/settings/tokens/new?scopes=repo,user,gist,workflow'
```

### New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

#### Note

R:GITHUB\_PAT

What's this token for?

#### Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> <b>admin:org</b>	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input type="checkbox"/> <b>admin:public_key</b>	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys
<input type="checkbox"/> <b>admin:repo_hook</b>	Full control of repository hooks
<input type="checkbox"/> write:repo_hook	Write repository hooks
<input type="checkbox"/> read:repo_hook	Read repository hooks
<input type="checkbox"/> <b>admin:org_hook</b>	Full control of organization hooks
<input checked="" type="checkbox"/> <b>gist</b>	Create gists

Independentemente de como um PAT foi gerado, clique em **Generate token** no pé da página para confirmar a criação do token. Agora é necessário seguir as instruções descritas no comando. Execute a função `gitcreds::gitcreds_set()`

e cole o seu token no console quando o mesmo for requisitado. Assim que isso estiver feito, você não precisará mais se preocupar com nenhum tipo de configuração.

Agora, ao criar um novo projeto, é possível associar imediatamente a ele um repositório no GitHub. O comando para criar projetos não muda, mas torna-se possível usar dois outros comando para associar aquela pasta com o sistema de controle de versões.

```

usethis::create_project("~/Documents/demo")
#> Creating '~/Documents/demo/'
#> Setting active project to '~/Documents/demo'
#> Creating 'R/'
#> Writing 'demo.Rproj'
#> Adding '.Rproj.user' to '.gitignore'
#> Opening '~/Documents/demo/' in new RStudio session
#> Setting active project to 'demo'

# No console do novo projeto

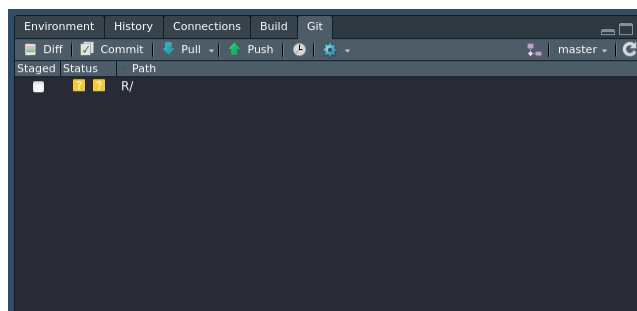
usethis::use_git()
#> Setting active project to '~/Documents/demo'
#> Initialising Git repo
#> Adding '.Rhistory', '.RData' to '.gitignore'
#> There are 2 uncommitted files:
#> * '.gitignore'
#> * 'demo.Rproj'
#> Is it ok to commit them?
#>
#> 1: Negative
#> 2: Not now
#> 3: I agree
#>
#> Selection: 3
#> Adding files
#> Commit with message 'Initial commit'
#> A restart of RStudio is required to activate the Git pane
#> Restart now?
#>
#> 1: Absolutely not
#> 2: No way
#> 3: Yes
#>
#> Selection: 3

usethis::use_github()

```

```
#> Checking that current branch is 'master'
#> Which git protocol to use? (enter 0 to exit)
#>
#> 1: ssh    <-- presumes that you have set up ssh keys
#> 2: https  <-- choose this if you don't have ssh keys (or don't know if you do)
#>
#> Selection: 2
#> Check title and description
#> Name:      demo
#> Description:
#> Are title and description ok?
#>
#> 1: Nope
#> 2: No way
#> 3: Yup
#>
#> Selection: 3
#> Creating GitHub repository
#> Setting remote 'origin' to 'https://github.com/curso-r/demo.git'
#> Pushing 'master' branch to GitHub and setting remote tracking branch
#> Opening URL 'https://github.com/curso-r/demo'
```

Depois de ter executado estes dois novos comandos, será inicializada uma nova aba no RStudio denominada Git. Através dela é possível controlar todas as versões do código e enviá-las ao GitHub para que sejam armazenadas com segurança na nuvem.



Para testar se está tudo funcionando, crie um arquivo na pasta R/ e abra a aba Git. Clique na caixa em branco que lá se encontra, aperte o botão **Commit** (escreva uma mensagem que descreva aquilo que você fez) e então aperte o botão **Push**. Em poucos segundos o repositório deve ser atualizado na sua página correspondente no GitHub.

Explicar todas as funcionalidades do Git e do GitHub estão além do escopo deste material. Os botões mais importantes dessa aba são as *checkboxes*, o **Commit**



e o **Push**. Para saber mais sobre esse assunto, recomendo o livro *Happy Git with R* (especialmente este capítulo) que aborda em detalhes minuciosos todo o processo e uso e manutenção do Git com o RStudio.



## Chapter 5

# Funções e dependências

Até este momento, foi abordada apenas uma forma de organizar os arquivos de uma análise: projetos. Entretanto existe ainda outra maneira, ainda mais interessante, de guardar análises. Se você programou em R, com certeza já se deparou com essa ferramenta, os bons e velhos pacotes ou bibliotecas. É surpreendentemente fácil criar um diretório que pode ser completamente acessado através da função `library()`.

Quando uma tarefa de análise de dados aumenta em complexidade, o número de funções e arquivos necessários para manter tudo em ordem cresce exponencialmente. Um arquivo para ler os dados, outro para limpar os nomes das colunas, mais um para fazer joins... Cada um deles com incontáveis blocos de código que rapidamente se transformam em uma macarronada.

O primeiro passo para sair dessa situação é transformar tudo em funções. Essa tarefa está longe de simples, mas os benefícios são imensos; ao encontrar um erro no resultado, fica bem mais fácil depurar a função culpada do que uma coleção desordenada de código. Funções têm argumentos e saídas, enquanto código solto pode modificar globais e criar resultados tardios que são impossíveis de acompanhar sem conhecer profundamente a tarefa sendo realizada.

```
library(dplyr)
library(tibble)

# Limpar dados
mtcars_clean <- mtcars %>%
  rownames_to_column(var = "model") %>%
  as_tibble() %>%
  filter(cyl < 8)

# Selecionar carros com 4 cyl e tirar média de mpg e wt
```

```
mtcars_clean %>%
  filter(cyl == 4) %>%
  group_by(cyl) %>%
  summarise(
    mpg = mean(mpg),
    wt = mean(wt)
  )
#> # A tibble: 1 x 3
#>   cyl    mpg    wt
#>   <dbl> <dbl> <dbl>
#> 1     4  26.7  2.29

# Selecionar carros com 6 cyl e tirar média de drat e disp
mtcars_clean %>%
  filter(cyl == 6) %>%
  group_by(cyl) %>%
  summarise(
    drat = mean(drat),
    disp = mean(displacement)
  )
#> # A tibble: 1 x 3
#>   cyl  drat  disp
#>   <dbl> <dbl> <dbl>
#> 1     6  3.59 183.
```

**Observação:** caso não esteja claro o que é o pipe (`%>%`), leia o Capítulo 7 para saber mais.

O código acima é somente um exemplo de análise. Como descrito pelos comentários, `mtcars` é limpa e depois são extraídas as médias de diferentes variáveis para duas seleções da tabela (número de cilindros igual a 4 e 6). Abaixo está descrita uma forma de transformar a maioria deste código em funções. É verdade que pela natureza simples do exemplo, fica difícil ver os benefícios do encapsulamento das tarefas de limpeza e resumo, mas perceba, por exemplo, que, se fosse necessário trocar `mean()` por `median()`, antes seria necessário alterar quatro linhas e agora apenas uma. Esse tipo de ganho a longo prazo pode salvar análises inteiras do caos.

```
library(dplyr)
library(tibble)

# Limpa tabela, filtrando cyl < cyl_max
clean <- function(data, cyl_max = 8) {
  data %>%
    rownames_to_column(var = "model") %>%
    as_tibble() %>%
```

```

    filter(cyl < cyl_max)
  }

# Resume tabela onde cyl == cyl_max, tirando média das colunas em ...
summarise_cyl <- function(data, cyl_num, ...) {
  data %>%
    filter(cyl == cyl_num) %>%
    group_by(cyl) %>%
    summarise_at(vars(...), mean)
}

# 4 cyl, média de mpg e wt
mtcars %>%
  clean(cyl_max = 8) %>%
  summarise_cyl(cyl_num = 4, mpg, wt)
#> # A tibble: 1 x 3
#>   cyl   mpg   wt
#>   <dbl> <dbl> <dbl>
#> 1     4  26.7  2.29

# 6 cyl, média de drat e disp
mtcars %>%
  clean(cyl_max = 8) %>%
  summarise_cyl(cyl_num = 6, drat, disp)
#> # A tibble: 1 x 3
#>   cyl drat disp
#>   <dbl> <dbl> <dbl>
#> 1     6  3.59 183.

```

Um código bem encapsulado reduz a necessidade de objetos intermediários (`base_tratada`, `base_filtrada`, etc.) pois para gerar um deles basta a aplicação de uma função. Além disso, programas com funções normalmente são muito mais enxutos e limpos do que *scripts* soltos, pois estes estimulam repetição de código. Às vezes é mais rápido copiar e colar um pedaço de código e adaptá-lo ao novo contexto do que criar uma função que generalize a operação desejada para as duas situações, mas os benefícios das funções são de longo prazo: ao encontrar um *bug*, haverá apenas um lugar para concertar; se surgir a necessidade de modificar uma propriedade, haverá apenas um lugar para editar; se aquele código se tornar obsoleto, haverá apenas um lugar para deletar.

Pense na programação funcional<sup>1</sup> como ir à academia. No início o processo é difícil e exige uma quantidade considerável de esforço, mas depois de um tempo se torna um hábito e traz benefícios consideráveis para a saúde (neste caso, do

<sup>1</sup>Aqui o termo “programação funcional” é usado de forma figurativa. Na computação linguagens denominadas “funcionais” tem um *modus operandi* bastante específico não abordado neste capítulo.

código). As recomendações para quando criar uma nova função ou separar uma função em duas variam muito, mas normalmente é uma boa ideia não deixar uma única função ser encarregada de mais uma tarefa ou ficar longa/complexa demais.

No mundo ideal, na pasta `R/` do seu projeto haverá uma coleção de arquivos, cada um com uma coleção de funções relacionadas e bem documentadas, e apenas alguns arquivos que utilizam essas funções para realizar a análise em si. Como dito anteriormente, isso fica muito mais fácil se você já tiver esse objetivo em mente desde o momento de criação do novo projeto.

## 5.1 Quatro-pontos

No exemplo da seção anterior, é possível notar as chamadas para as bibliotecas `dplyr` e `tibble`. Elas têm inúmeras funções úteis, mas aqui somente algumas poucas foram utilizadas. Além disso, se o código fosse muito maior, ficaria impossível saber de uma biblioteca ainda está sendo utilizada; se não fosse mais necessário utilizar `rownames_to_column()`, qual seria a melhor forma de saber que pode ser removida a chamada `library(tibble)`?

A resposta para essa pergunta pode assustar: no código ideal, a função `library()` nunca seria chamada, todas as funções teriam seus pacotes de origem explicitamente referenciados pelo operador `::`.

Esta subseção está separada porque ela de fato é um pouco radical demais. É excessivamente preciosista pedir para que qualquer análise em R seja feita sem a invocação de nenhuma biblioteca, apenas com chamadas do tipo `biblioteca::funcao()`. Muitas pessoas inclusive nem sabem que é possível invocar uma função diretamente através dessa sintaxe!

Se algum leitor estiver tendendo a seguir o caminho do TOC da programação, existem dois grandes benefícios em chamar todas as funções diretamente:

- o código, no total, executa um pouco mais rápido porque são carregadas menos funções no ambiente global (isso é especialmente importante em aplicações interativas feitas em Shiny).
- as dependências do código estão sempre atualizadas porque elas estão diretamente atreladas às próprias funções sendo utilizadas.

Existe um terceiro e importante benefício, mas este será abordado apenas no próximo capítulo. A título de curiosidade, o código anterior ficaria assim caso fosse escrito sem as chamadas para `library()`:

```

# Referência ao pipe
`%>%` <- magrittr::`%>%`

# Limpa tabela, filtrando cyl < cyl_max
clean <- function(data, cyl_max = 8) {
  data %>%
    tibble::rownames_to_column(var = "model") %>%
    dplyr::as_tibble() %>%
    dplyr::filter(cyl < cyl_max)
}

# Resume tabela onde cyl == cyl_max, tirando média das colunas em ...
summarise_cyl <- function(data, cyl_num, ...) {
  data %>%
    dplyr::filter(cyl == cyl_num) %>%
    dplyr::group_by(cyl) %>%
    dplyr::summarise_at(dplyr::vars(...), mean)
}

# 4 cyl, média de mpg e wt
mtcars %>%
  clean(cyl_max = 8) %>%
  summarise_cyl(cyl_num = 4, mpg, wt)
#> # A tibble: 1 x 3
#>   cyl   mpg   wt
#>   <dbl> <dbl> <dbl>
#> 1     4  26.7  2.29

# 6 cyl, média de drat e disp
mtcars %>%
  clean(cyl_max = 8) %>%
  summarise_cyl(cyl_num = 6, drat, disp)
#> # A tibble: 1 x 3
#>   cyl drat disp
#>   <dbl> <dbl> <dbl>
#> 1     6  3.59 183.

```

Se serve de consolo, o RStudio facilita muito esse tipo de programação por causa da sua capacidade de sugerir continuações para código interativamente. Para escrever `dplyr::`, por exemplo, basta digitar `d`, `p`, `l` e apertar `TAB` uma vez. Com os `::`, as sugestões passarão a ser somente de funções daquele pacote.





## Chapter 6

# Pacotes

Nas palavras do maior guru do R, Hadley Wickham, “pacotes são a unidade fundamental de código R reprodutível”. Toda vez que você utiliza a função `library()`, algum pacote está sendo carregado na sessão. Muitas vezes criar uma biblioteca de funções pode parecer uma tarefa árdua e confusa, restrita a grandes conhecedores da linguagem, mas essa impressão não poderia estar mais distante da realidade: pacotes para o R são bastante simples e intuitivos de fazer.

No início deste livro foi abordado o conceito de projeto. Ele não passa de um arquivo `.Rproj` que indica para o RStudio que aquele diretório é um ambiente de trabalho estruturado. Nesse sentido, pacotes são iguais a projetos porque eles também têm um `.Rproj`; pacotes na verdade *são* projetos.

A diferença entre os dois é que pacotes podem ser documentados e instalados, permitindo toda uma gama de novas possibilidades para o programador. Muitas vezes uma análise de dados pode envolver dezenas de funções e diversas pessoas, fazendo com que o compartilhamento de código seja vital para que a análise não fuja do controle. Pacotes permitem gerenciar dependências, manter documentação, executar testes unitários e muito mais com o objetivo de deixar todos os analistas na mesma página.

Sendo assim, recomenda-se criar um pacote para qualquer análise que envolva pelo menos meia dúzia de funções complexas e mais de uma pessoa, caso contrário, um projeto já é suficiente. Outra motivação para criar um pacote é compartilhar conjuntos úteis de funções com outras pessoas; isso acaba sendo menos comum para a maioria dos usuários, mas é importante ressaltar que o R não seria a linguagem popular que é hoje se não fossem pelas famosas bibliotecas `ggplot2` e `dplyr`.

```
usethis::create_package("~/Documents/demo")
#> Setting active project to '~/Documents/demo'
```

```
#> Creating 'R/'
#> Writing 'DESCRIPTION'
#> Package: demo
#> Title: What the Package Does (One Line, Title Case)
#> Version: 0.0.0.9000
#> Authors@R (parsed):
#>   * First Last <first.last@example.com> [aut, cre] (<https://orcid.org/YOUR-ORCID>)
#> Description: What the package does (one paragraph).
#> License: What license it uses
#> Encoding: UTF-8
#> LazyData: true
#> Writing 'NAMESPACE'
#> Writing 'demo.Rproj'
#> Adding '.Rproj.user' to '.gitignore'
#> Adding '~demo\\.Rproj$', '~\\.Rproj\\.user$' to '.Rbuildignore'
#> Opening '~/Documents/demo/' in new RStudio session
#> Setting active project to 'demo'
```

A função executada acima é exatamente análoga à função de criação de projetos. A principal diferença é que ela cria um arquivo `DESCRIPTION` e assume que o nome do pacote é igual ao nome da pasta onde o mesmo está sendo criado (neste caso, “demo”). Alguns outros arquivos também são criados (como `.Rbuildignore` e `NAMESPACE`), mas eles não vêm ao caso. De resto, o pacote é idêntico a um projeto e pode ser sincronizado com o Git exatamente da mesma maneira.

O primeiro passo para começar a usar um pacote é atribuir a ele uma licença (caso um dia você resolva compartilhá-lo com o mundo) e preencher a descrição. Abaixo encontra-se uma função simples que adiciona uma licença MIT ao pacote.

```
usethis::use_mit_license("Seu Nome")
#> Setting active project to '~/Documents/demo'
#> Setting License field in DESCRIPTION to 'MIT + file LICENSE'
#> Writing 'LICENSE.md'
#> Adding '~LICENSE\\.md$' to '.Rbuildignore'
#> Writing 'LICENSE'
```

O arquivo de descrição, no entanto, é um pouco mais complexo porque ele tem alguns campos que precisam ser preenchidos manualmente. Quando o pacote for criado, eles já estarão populados com instruções para facilitar a vida do programador. Abaixo está um exemplo de como `DESCRIPTION` deve ficar depois de completo:

```
Package: demo
Title: O Que o Pacote Faz (Uma Linha)
```

```

Version: 0.0.0.9000
Authors@R:
  person(given = "Seu",
         family = "Nome",
         role = c("aut", "cre"),
         email = "seunome@dominio.com")
Description: 0 que o pacote faz (um paragrafo curto terminado em ponto final).
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true

```

A partir deste ponto, os metadados do pacote estão essencialmente prontos e não precisam mais ser modificados. Assim como em um projeto, o que resta é adicionar arquivos com funções à pasta `R/`.

## 6.1 Documentação

Para poder programar pacotes com mais facilidade, é necessário instalar o `devtools`. Assim como o `tidyverse`, este é um conjunto de pacotes (que inclui o `usethis` por sinal) que auxiliam no processo de criar e testar um pacote de R.

```
install.packages("devtools")
```

A partir de agora você pode, por exemplo, criar documentações para as funções do seu pacote. Quando outras pessoas o instalarem, elas poderão consultar esses manuais da mesma forma que fazem com qualquer outra função: `?funcao()`.

A documentação mais simples (e obrigatória) envolve dar um título para a função e descrever o que cada parâmetro significa. Para documentar uma função qualquer, basta adicionar comentários em cima dela com `#'` assim como no exemplo abaixo:

```

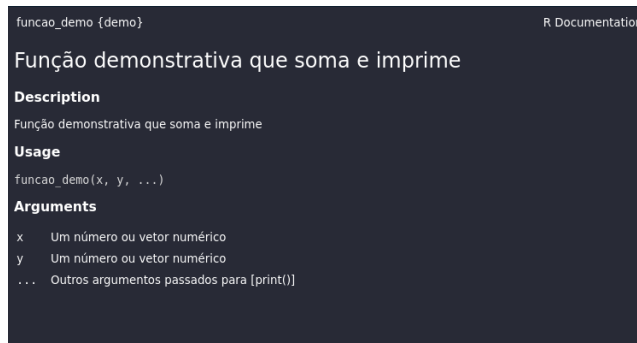
#' Função demonstrativa que soma e imprime
#'
#' @param x Um número ou vetor numérico
#' @param y Um número ou vetor numérico
#' @param ... Outros argumentos passados para [print()]
#'
#' @export
funcao_demo <- function(x, y, ...) {
  z <- x + y
  print(z, ...)
  return(z)
}

```

No RStudio, esse tipo de documentação é tratado diferentemente de outros comentários, então certas palavras-chave ficam coloridas. `@param` por exemplo indica a documentação de um dos parâmetros e `@export` indica que aquela função será exportada pelo pacote, ou seja, ficará disponível ao usuário quando ele executar `library(demo)`.

Para gerar a documentação do pacote, basta chamar uma outra função do `devtools`:

```
devtools::document()  
#> Updating demo documentation  
#> Updating roxygen version in ~/Documents/demo/DESCRIPTION  
#> Writing NAMESPACE  
#> Loading demo  
#> Writing NAMESPACE  
#> Writing funcao_demo.Rd  
  
?funcao_demo()  
#> Rendering development documentation for 'funcao_demo'
```



Conforme o número de funções no pacote for crescendo, basta iterar nesse ciclo descrito até aqui. Além disso, é importante lembrar (como destacado na sessão anterior) que qualquer função utilizada de outro pacote deve ser invocada na forma `pacote::funcao()`; neste momento, o pacote em questão se tornará uma *dependência* do seu pacote e deve ser declarado como tal com `usethis::use_package("pacote")` (mais sobre isso a seguir).

Para garantir que o R não encontrará nenhum problema no seu pacote, basta executar a função de verificação `devtools::check()`. Se nenhum defeito for encontrado, basta compartilhar o pacote com os seus colegas e instalá-lo com `devtools::install_local()`.

```

devtools::check()
#> Updating demo documentation
#> Writing NAMESPACE
#> Loading demo
#> Writing NAMESPACE
#> Building demo
#> Setting env vars:
#> CFLAGS      : -Wall -pedantic -fdiagnostics-color=always
#> CXXFLAGS    : -Wall -pedantic -fdiagnostics-color=always
#> CXX11FLAGS: -Wall -pedantic -fdiagnostics-color=always
#>
#> checking for file '/home/clente/Documents/demo/DESCRIPTION' ...
#>
#> [... omitido por brevidade ...]
#>
#> R CMD check results demo 0.0.0.9000
#> Duration: 8.2s
#>
#> 0 errors | 0 warnings | 0 notes

```

## 6.2 Imports

Como dito anteriormente, um passo importante na criação de um pacote é a declaração de suas dependências. Se você tiver seguido com cuidado a sugestão de já começar a escrever o pacote declarando todas as importações explicitamente (ou seja, com `pacote::funcao()`), isso não será um problema, pois o `devtools::check()` já será capaz de apontar exatamente quais dependências ainda precisam ser registradas.

Suponha que um pacote tem exatamente duas dependências: o `dplyr` e o `kuber` (um pacote desenvolvido pela Curso-R e disponível somente via GitHub). Para passar no `check()` basta executar os dois comandos abaixo:

```

usethis::use_package("dplyr")
usethis::use_dev_package("kuber")

```

O `dplyr`, por estar disponível no CRAN, não é um pacote considerado em fase de desenvolvimento, então basta usar `usethis::use_package()`. Já o `kuber` só está disponível no GitHub e instalável via `remotes::install_github("curso-r/kuber")`, então precisamos neste caso utilizar `usethis::use_dev_package()`. Os comandos acima criam uma nova seção no arquivo `DESCRIPTION`:

Imports:

```
  dplyr,  
  kuber (>= 0.3.1.9000)  
Remotes:  
  curso-r/kuber
```

Note que o `kuber` já é importado com um link remoto para o seu repositório no GitHub e um marcador da versão atualmente instalada na sua máquina. Para saber mais sobre versões, consulte o Capítulo 10.

## Chapter 7

# Pipe

Difícilmente um programador de R passa muito tempo sem ouvir falar do operador pipe (`%>%`). Saber o que ele faz e significa, no entanto, é algo mais complexo. Apesar de não passar de uma função como outra qualquer, os efeitos que ele pode ter no visual e na compreensibilidade de um código são imensos.

Entender o pipe em profundidade pode levar muito tempo, mas o básico já é suficiente para a maioria das pessoas. É importante ter pelo menos uma ideia do que ele faz caso você acabe se deparando com um código que o utiliza e, quem sabe um dia, usá-lo nos seus próprios programas.

No fundo, o conceito de pipe existe pelo menos desde os anos 1970. De acordo com seu criador, Douglas McIlroy, o operador foi concebido em “uma noite febril” e tinha o objetivo de simplificar comandos cujos resultados deveriam ser passados para outros comandos.

```
ls | cat
# Desktop
# Documents
# Downloads
# Music
# Pictures
# Public
# Templates
# Videos
```

Com esse exemplo já é possível ter uma ideia de onde vem o seu nome: *pipe* em inglês significa “cano”, referindo-se ao transporte das saídas dos comandos. Em português o termo é traduzido preferencialmente como “encadeamento”, mas no dia-a-dia é mais comum usar o termo em inglês.

A partir daí o pipe tem aparecido nas mais diversas aplicações, desde HTML até o próprio R. Ele pode aparecer em diferentes formas, mas o seu objetivo é sempre o mesmo: canalizar resultados de um comando para o outro.

## 7.1 Como funciona

Em R, o pipe tem uma aparência bastante particular (`%>%`), mas no fundo ele não passa de uma função infix, ou seja, uma função que aparece entre os seus argumentos (como `a + b` ou `a %in% b`). Na verdade é por isso mesmo que ele tem porcentagens antes e depois: no R uma função infix só pode ser declarada assim (vide o próprio `%in%()`).

Se você estiver no RStudio, como sugerido anteriormente, para usar o pipe basta carregar a biblioteca `magrittr` e utilizar o atalho **Ctrl + Shift + M**; essas inocentes teclas irão fazê-lo aparecer magicamente diante dos seus olhos. Abaixo é possível notar como o pipe não passa de uma função como outra qualquer (ignore os acentos graves):

```
library(magrittr)

`%>%`("oi", print)
#> [1] "oi"
```

Perceba que, no código acima, o primeiro argumento do pipe (`"oi"`) virou a entrada do seu segundo argumento (a função `print()`). Abaixo o pipe está na sua forma mais tradicional entre seus dois argumentos:

```
"oi" %>% print()
#> [1] "oi"
```

Observe agora o comando abaixo. Queremos primeiro somar 3 a uma sequência de números e depois dividi-los por 2:

```
mais_tres <- function(x) { x + 3 }
sobre_dois <- function(x) { x / 2 }

x <- 1:3

sobre_dois(mais_tres(x))
#> [1] 2.0 2.5 3.0
```

Perceba como fica difícil de entender o que acontece primeiro. A linha relevante começa com a divisão por 2, depois vem a soma com 3 e, ao fim, os valores de



entrada. É exatamente o oposto da nossa ordem de leitura da esquerda para a direita.

Nesse tipo de situação é mais legível usar a notação de composição de funções, com as funções sendo exibidas na ordem em que serão aplicadas:  $f \circ g$ . Não é necessário conhecer essa notação, basta imaginar quão mais legível ficaria aquele comando se houvesse algum recurso que passasse o que o resultado do que está à sua esquerda para a função que está à sua direita. Esse é o pipe.

```
x %>% mais_tres() %>% sobre_dois()
#> [1] 2.0 2.5 3.0
```

No comando acima fica evidente que pegamos o objeto `x`, somamos 3 e dividimos por 2.

Perceba que a entrada de um pipe (esquerda) sempre é passada como o *primeiro* argumento da função à sua direita. Isso não impede que as funções utilizadas em uma sequência de pipes tenham outros argumentos.

```
mais_n <- function(x, n) { x + n }

x %>% mais_n(4) %>% sobre_dois()
#> [1] 2.5 3.0 3.5
```

## 7.2 Vantagens

A grande vantagem do pipe não é só enxergar quais funções são aplicadas primeiro, mas sim nos ajudar a programar pipelines (“encanamento” em inglês) de tratamentos de dados.

```
library(dplyr)

starwars %>%
  mutate(bmi = mass/((height/100)^2)) %>%
  select(name, bmi, species) %>%
  group_by(species) %>%
  summarise(bmi = mean(bmi))
#> # A tibble: 38 x 2
#>   species      bmi
#>   <chr>      <dbl>
#> 1 Aleena     24.0
#> 2 Besalisk  26.0
#> 3 Cerean    20.9
#> 4 Chagrian   NA
```

```
#> 5 Clawdite 19.5
#> 6 Droid NA
#> 7 Dug 31.9
#> 8 Ewok 25.8
#> 9 Geonosian 23.9
#> 10 Gungan NA
#> # ... with 28 more rows
```

Acima fica extremamente claro o que está acontecendo em cada passo da pipeline. Partindo da base `starwars`, primeiro transformamos, depois selecionamos, agrupamos e resumimos: em cada linha temos uma operação e elas são executadas em sequência.

Isso não melhora só a legibilidade do código, mas também a sua *debugabilidade*<sup>1</sup>. Se tivermos encontrado um bug na pipeline, basta executar o encadeamento linha a linha até que encontremos o comando problemático. Com o pipe é possível programar de forma mais compacta, legível e correta.

Todos os exemplos acima envolvem passar a entrada do pipe como o primeiro argumento da função à direita, mas esta não é uma obrigatoriedade. Com o operador *placeholder* `.` podemos indicar exatamente onde deve ser colocado o valor que chega no pipe:

```
y_menos_x <- function(x, y) { y - x }

x %>%
  mais_tres() %>%
  purrr::map2(4:6, ., y_menos_x)
# [[1]]
# [1] 0
#
# [[2]]
# [1] 0
#
# [[3]]
# [1] 0
```

Outra funcionalidade interessante e pouco conhecida do pipe são as funções unárias. Se você estiver familiarizado com o pacote `purrr`, esse é um jeito bastante simples de criar funções descartáveis.

```
m3_s2 <- . %>%
  mais_tres() %>%
  sobre_dois()
```

---

<sup>1</sup> Ainda não encontrei um termo melhor em português para esse conceito

```
m3_s2(x)
#> [1] 2.0 2.5 3.0
```

Usando novamente o `.`, definimos uma função que recebe apenas um argumento com uma sequência de aplicações de outras funções. Mas não se preocupe se as funções unárias tiverem parecido algo de outro mundo porque é realmente muito raro encontrá-las nos códigos alheios.

Por fim, caso você queira utilizar o pipe dentro do seu próprio pacote, basta executar uma simples função do `usethis`: `use_pipe()`. Ela adiciona o `magrittr` como uma dependência do seu pacote e faz com que o pipe seja acessível dentro do mesmo, o que realmente facilita o trabalho do desenvolvedor que não quer entender o significado do código a seguir:

```
#' Pipe operator
#'
#' See \code{magrittr::\link[magrittr:pipe]{\%>\%}} for details.
#'
#' @name %>%
#' @rdname pipe
#' @keywords internal
#' @export
#' @importFrom magrittr %>%
#' @usage lhs \%>\% rhs
NULL
```

Se você não estiver trabalhando em um pacote, mas também não quer carregar o pacote com `library()`, há duas formas de trazer o pipe para os seus scripts: puxando ele como uma função ou usando o argumento `include.only` da `library()`.

```
# Puxando como função
`%>%` <- magrittr::`%>%`

# Usando include.only
library(magrittr, include.only = "%>%")
```



## Chapter 8

# Data e data-raw

Na seção anterior, foi discutida a importância de empacotar uma análise. Seja para organizar dependências, reutilizar código, manter testes automatizados, ou qualquer outra razão, pacotes são a melhor forma de guardar e compartilhar código em R. Mas, apesar de toda a conversa sobre programação, pouco foi abordado sobre outro elemento essencial de uma análise de dados: dados.

Felizmente, pacotes em R têm lugares específicos para guardar dados brutos e dados tratados. São as pastas `data` e `data-raw`, cada uma com as suas propriedades e possibilidades. Ambas podem ser criadas com facilidades por funções do pacote `usethis`, então elas se encaixam perfeitamente no fluxo de análise descrito até agora.

Como indicado anteriormente, existem dois tipos de dados: brutos e tratados. Normalmente dados brutos estão em formatos comumente compartilhados em ambientes de trabalho: planilhas Excel, arquivos CSV, etc. Os pacotes `readxl` e `readr` permitem que esses formatos sejam importados para dentro do R, mas normalmente essas funções são mais lentas e menos padronizadas do que `readRDS()`, por exemplo, que lê arquivos no formato nativo do R.

Além disso, raramente os dados recebidos durante uma análise estarão perfeitamente organizados e padronizados. É comum precisar de múltiplos fluxos de tratamento para poder transformar os dados brutos naquilo que de fato pode ser utilizado durante uma análise.

O programador é encorajado a separar essas planilhas brutas daquelas resultantes do processo de limpeza e tratamento. Junto com os dados crus, é importante também guardar os arquivos que fazem o processo de limpeza; caso haja uma mudança nas demandas ou nas bases, o analista precisa ser capaz de alterar os *scripts* de tratamento e gerar novas bases consolidadas.

No exemplo abaixo, supõe-se um diretório com um pacote R e uma base bruta denominada `dados.xlsx`. Primeiramente deve-se executar a função

`usethis::use_data_raw()` para criar a pasta `data-raw` e um arquivo de tratamento para a base em questão.

```
usethis::use_data_raw("dados")
#> Setting active project to '~/Documents/demo'
#> Creating 'data-raw/'
#> Adding '~data-raw$' to '.Rbuildignore'
#> Writing 'data-raw/dados.R'
#> Modify 'data-raw/dados.R'
#> Finish the data preparation script in 'data-raw/dados.R'
#> Use `usethis::use_data()` to add prepared data to package
```

Como indicado pelos três últimos pontos da saída do comando, agora basta colocar o código de tratamento da base `dados` em `data-raw/dados.R` e por fim utilizar `usethis::use_data()` para adicionar os dados preparados ao pacote. Para prosseguir o exemplo, o arquivo `dados.xlsx` foi copiado para o diretório `data-raw` e o código abaixo foi inserido em `data-raw/dados.R`.

```
library(magrittr)

# Limpar a base dados.xlsx
dados <- "data-raw/dados.xlsx" %>%
  readxl::read_xlsx() %>%
  dplyr::filter(cyl > 4) %>%
  dplyr::mutate(
    brand = stringr::str_extract(model, "[A-z]+")
  ) %>%
  dplyr::group_by(brand) %>%
  dplyr::summarise(
    mean_mpg = mean(mpg),
    prop_6_cyl = sum(cyl == 6)/dplyr::n()
  ) %>%
  dplyr::arrange(brand)

# Salvar a base para uso no pacote
usethis::use_data(dados)
#> Creating 'data/'
#> Saving 'dados' to 'data/dados.rda'
```

Neste caso o arquivo Excel foi criado de dentro do próprio R com o comando `writexl::write_xlsx(tibble::rownames_to_column(mtcars, "model"), "data-raw/dados.xlsx")`, mas isso é só um exemplo ilustrativo. O importante é saber o que acontece quando a função `use_data()` é executada para um objeto do ambiente global, ou seja, as duas últimas linhas do bloco de código acima.

Por trás das câmeras, `use_data()` está chamando a função `save()` do R para gerar um arquivo RDA a partir de um objeto do ambiente global. Arquivos RDA são extremamente estáveis, compactos e podem ser carregados rapidamente pelo R, tornando este formato o principal meio de guardar dados de um pacote. Se os dados do pacote forem guardados assim, eles ficarão disponíveis para serem chamados pelo usuário (você mesmo durante a análise)! Para entender como ficam os dados uma vez que eles são incluídos na pasta `data`, basta dar uma olhada no objeto `dplyr::starwars`; neste caso, a base tratada e exportada se chama `starwars`.

Para carregar os dados na sua sessão e poder utilizá-los na análise, basta executar `pkgload::load_all()` ou pressionar a combinação CTRL + SHIFT + L no RStudio. Independentemente do número de tabelas que estiverem salvas na pasta `data`, todas serão carregadas instantaneamente.

A título de curiosidade, existem algumas situações em que as bases brutas são grandes demais para serem sincronizadas com o GitHub. A plataforma tem um (razoável) limite de 1GB por repositório que pode ser insuficiente para armazenar dados brutos e tratados. Para não sincronizar as bases brutas com o Git, basta adicioná-las ao arquivo `.gitignore` do pacote; no caso do exemplo acima, bastaria adicionar a esse arquivo uma linha com o texto `data-raw/dados.xlsx`.

## 8.1 Documentação

Além de funções, também é possível documentar bases de dados com o pacote `roxygen2`. Para isso, crie um arquivo `data.R` na pasta `R/` do pacote e crie um objeto entre aspas com o nome de cada base de dados exportada. Documentar dados é extremamente útil quando o pacote vai ser compartilhado com múltiplas pessoas da mesma organização, pois assim não é necessário compartilhar uma planilha Excel separada descrevendo cada uma das colunas da tabela.

Uma boa documentação de bases de dados não precisa de muita coisa. Abaixo é exemplificado como seria documentada `dados`:

```
#' Dados sobre 15 marcas de carros
#'
#' A tabela, gerada a partir de `mtcars`, apresenta algumas poucas
#' informações sobre carros com mais de 4 cilindros de 15 marcas
#' americanas de carros.
#'
#' @format Uma tabela com 3 colunas e 15 linhas:
#' \describe{
#'   \item{brand}{Marca}
#'   \item{mean_mpg}{Milhas por galão médias para aquela marca}
#'   \item{prop_6_cyl}{Proporção dos carros que apresentam 6 cilindros}
```

```
#' }  
#' @source Henderson and Velleman (1981)  
"dados"
```



## Chapter 9

# Testes automatizados

Um recurso extremamente importante, mas comumente ignorado no mundo do R, para a garantia da longevidade de um pacote são os testes unitários automatizados. Como essa técnica deve ser limitada somente a pacotes complexos e utilizados por mais de uma pessoa, esta seção não passa de uma breve exploração do tópico.

Um teste unitário não passa de uma verificação de corretude referente a uma pequena unidade do código-fonte de um software. Testes unitários garantem que uma alteração pontual em algum lugar do código não vai alterar o comportamento de nenhuma outra parte, já que as outras funções ainda terão que passar nos seus próprios testes. Além disso, antes de fazer uma alteração em código já existente, é comum pensar antes em quais devem ser os novos resultados para os testes daquela função (prática conhecida como Desenvolvimento Orientado a Testes).

Para criar um conjunto de testes é necessário primeiro criar o ambiente para tal dentro do pacote através do pacote `testthat`. Depois disso, basta criar conjuntos individuais de testes para cada função.

```
usethis::use_testthat()
#> Adding 'testthat' to Suggests field in DESCRIPTION
#> Creating 'tests/testthat/'
#> Writing 'tests/testthat.R'
#> Call `use_test()` to initialize a basic test file and open it for editing.

usethis::use_test("funcao_demo")
#> Increasing 'testthat' version to '>= 2.1.0' in DESCRIPTION
#> Writing 'tests/testthat/test-funcao_demo.R'
#> Modify 'tests/testthat/test-funcao_demo.R'
```

Como é possível notar, o pacote `testthat` permite criar um arquivo de testes

para `funcao_demo()` (neste caso `tests/testthat/test-funcao_demo.R`). Esse arquivo já vem com um teste padrão a título de demonstração, mas, depois de reescrito manualmente, um possível conjunto de testes para `funcao_demo()` seria o seguinte:

```
library(demo)

test_that("funcao_demo funciona", {

  expect_equal(funcao_demo(1, 2), 3)
  expect_equal(funcao_demo(-1, -2), -3)
  expect_equal(funcao_demo(1, -2), -1)

  expect_output(funcao_demo(1, 2), "3")

})
```

E o resultado da execução dos testes é o seguinte:

```
devtools::test()
#> Loading demo
#> Testing demo
#> | OK F W S | Context
#> | 4       | funcao_demo
#>
#> Results
#> OK:      4
#> Failed:  0
#> Warnings: 0
#> Skipped: 0
#>
#> Keep up the good work.
```

Para saber mais sobre como desenvolver testes eficazes, consulte a própria documentação do `testthat`.

## Chapter 10

# Versões e releases

A maior parte dos softwares que utilizamos no dia-a-dia possuem versões, até mesmo aqueles que não costumamos associar com esse tipo de prática. Sistemas operacionais, como Windows 10 e iOS 13, têm as versões embutidas em seus próprios nomes, enquanto muitos aplicativos, como WhatsApp 2.20.22, só exibem esse tipo de informação no fundo da página de configurações.

Pacotes do R também possuem versões. Ao executar o comando `update.packages()` (inclusive faça isso agora caso você nunca o tenha feito), o R é forçado a procurar por versões mais recentes dos pacotes instalados na sua máquina. É importante deixar claro que “versão” não é nada mais que o identificador de uma atualização do software; se eu mudar uma linha do código do meu pacote e atualizar esse fonte no GitHub, pode-se dizer que criei uma nova versão do meu pacote.

Mas por que marcar e dar nomes a essas atualizações? É bastante mais fácil simplesmente continuar programando e exigir que seus usuários baixem a versão mais recente do seu código. Essa prática, entretanto, é ruim para o usuário e para o programador.

Você, como programador, quer poder fazer alterações no seu código que não necessariamente estão prontas para o público em geral. Erros acontecem e, muitas vezes, é preciso fazer uma série de modificações antes que o pacote volte a ter certa estabilidade. Por outro lado, o usuário também só quer atualizar seu pacote quando algo suficientemente diferente estiver disponível (sejam correções de erros, sejam novas funcionalidades). Além disso, caso uma nova versão do seu código gere problemas para ele, é necessário ter uma versão estável anterior claramente rotulada para que o usuário possa fazer o *downgrade* do pacote.

E justamente são esses rótulos que precisamos utilizar para ter um bom sistema de versões nos nossos pacotes. Ao contrário de “controle de versão”, para o qual utilizamos o Git, “versionamento semântico” é a prática de dar nomes fáceis de entender para as versões de um projeto. Trazendo isso para termos concretos,

quando este livro foi escrito, o pacote `dplyr` estava na versão 0.8.4, o `ggplot2` estava na 3.2.1 e o `shiny` na 1.4.0.

## 10.1 Versionamento semântico

Em teoria, existe um padrão ouro para a nomenclatura de versões de um software. Alguns dos seus preceitos, como o uso de 3 números inteiros não-negativos separados por pontos, são extremamente valiosos, enquanto outros nem sempre são seguidos ao pé da letra. Aqui tratamos do protocolo padronizado, mas apenas você pode determinar quanto dele o seu esquema de nomenclatura seguirá.

Uma versão é denotada pela forma `X.Y.Z`, onde:

- `X` é a versão maior (denominada *major*);
- `Y` é a versão menor (denominada *minor*) e
- `Z` é a versão do conserto (denominada *patch*).

Um patch não passa de uma versão atualizada na qual apenas bugs foram corrigidos; nenhuma funcionalidade pode ser alterada e qualquer código escrito utilizando a versão sem o patch deve continuar funcionando. Quando alguma nova funcionalidade é introduzida ao programa, mas ele continua sendo retrocompatível (ou seja, compatível com as suas versões anteriores), deve ser incrementada a versão minor. Por fim, se for introduzida alguma mudança que quebra a retrocompatibilidade, deve ser incrementada a versão major. É importante dizer também que, depois de lançada, uma versão **nunca** deverá ser alterada, pois isso confundirá os usuários da mesma.

Alguns pontos do versionamento semântico não são seguidos sempre, mas podem ser úteis. Por exemplo, o patch pode ser omitido se ele for 0, a major 0 normalmente é reservada para software em fase de testes (o `dplyr`, portanto, não segue essa regra) e às vezes sufixos em texto podem ser adicionados (como “-alpha”, “-beta”, etc.).

Para propósitos ilustrativos, imaginemos um pacote chamado `pacotr` que possui apenas uma função com protótipo `f(x, y, z = TRUE)`. Um caminho imaginário para seu desenvolvimento pode ser o seguinte:

- Versão 0.1.0: a primeira versão com código do pacote contém uma função `f(x, y)`;
- Versão 0.1.1: um bug é consertado em `f(x, y)`;
- Versão 0.1.2: mais um bug é consertado em `f(x, y)`;
- Versão 1.0.0: o pacote sai do beta quando a função ganha um novo argumento `z`, agora necessário para o seu funcionamento, e vira `f(x, y, z)`;

- Versão 1.0.1: um bug é consertado em `f(x, y, z)`;
- Versão 1.1.0: o argumento `z` agora tem um valor padrão, transformando a função em `f(x, y, z = FALSE)`;
- Versão 2.0.0: o argumento `z` muda seu valor padrão, quebrando qualquer código escrito utilizando a versão 1.x.x, se tornando `f(x, y, z = TRUE)`

Utilizando o exemplo acima sem perda de generalidade, entre qualquer dois passos subsequentes haveriam versões denominadas de desenvolvimento, como por exemplo 0.1.1.9000, 1.0.0.9000 e assim por diante. Isso não está no padrão ouro do versionamento semântico, mas é comumente utilizado em programas de R.

Uma versão de desenvolvimento é qualquer “subversão” que não deveria ser utilizada pelo usuário final; qualquer commit entre patches, minors ou majors é uma versão de desenvolvimento. Por convenção, a primeira versão de desenvolvimento é marcada como 9000, podendo chegar até 9999. Alguns programadores incrementam esse número a cada commit realizado, mas é mais comum manter o 9000 até que o código esteja pronto para se tornar um verdadeiro patch, minor ou major.

Na prática, é muito simples trabalhar com versões. Assim que um novo pacote é criado através de `usethis::create_package()`, o arquivo `DESCRIPTION` tem a seguinte cara:

```
Package: demo
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
  person(given = "First",
    family = "Last",
    role = c("aut", "cre"),
    email = "first.last@example.com",
    comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: What license it uses
Encoding: UTF-8
LazyData: true
```

Note como o campo `Version` já está populado com a primeira versão possível para um pacote: 0.0.0.9000. Depois de alguns commits, se acharmos que o código está preparado para a primeira minor, utilizamos:

```
usethis::use_version("minor")
#> Setting Version field in DESCRIPTION to '0.1.0'
```

A segunda linha já deixa claro que agora o arquivo `DESCRIPTION` contém uma linha `Version: 0.1.0` (note como a versão de desenvolvimento é removida automaticamente). **Mas atenção**, antes de fazer qualquer outra alteração no pacote, você deve se certificar de que esta alteração receberá um commit só para ela. Atualmente o `usethis` faz isso automaticamente.

O **primeiro** commit após a alteração da versão deve trazer o pacote de volta para o estado de desenvolvimento. Ou seja, deve existir um único commit no qual o pacote é considerado estável para cada sequência de commits de desenvolvimento.

```
usethis::use_dev_version()  
#> Setting Version field in DESCRIPTION to '0.1.0.9000'
```

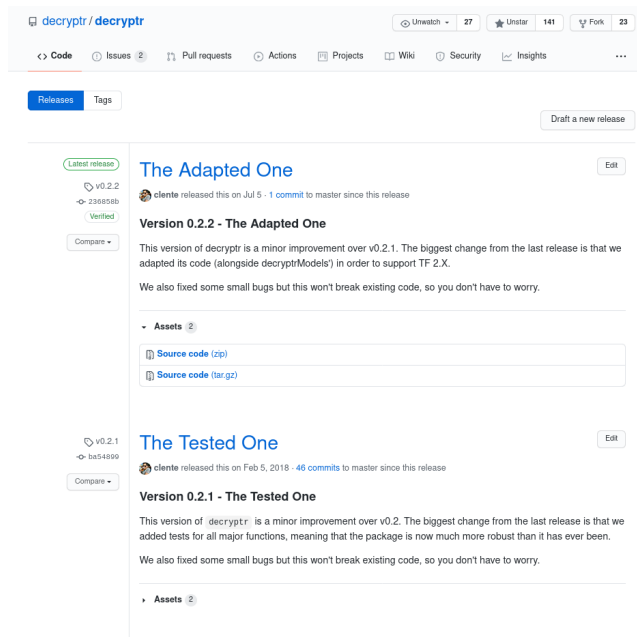
Depois que o pacote voltar para desenvolvimento, a programação pode continuar desimpedida. Por completude, `usethis::use_version()` pode ser utilizada para qualquer versão cheia:

```
usethis::use_version("patch")  
#> Setting Version field in DESCRIPTION to '0.1.1'  
  
usethis::use_dev_version()  
#> Setting Version field in DESCRIPTION to '0.1.1.9000'  
  
usethis::use_version("major")  
#> Setting Version field in DESCRIPTION to '1.0.0'
```

## 10.2 Releases

Uma parte interessante do processo de versionamento é que o GitHub permite marcar commits específicos como estáveis e atribuir um rótulo aos mesmos. No fundo isso não passa de um versionamento semântico integrado ao site.

Como exemplo, podemos ver a aba “Releases” do pacote `decryptr`:



É bom fazer esse tipo de versionamento externo porque os usuários passam a ter um lugar de fácil acesso com o código-fonte de absolutamente todas as versões do seu pacote. Além disso, a função `devtools::install_github()` é capaz de utilizar esses releases como indicador de qual versão de um pacote deve ser baixada.

Para criar releases de um pacote seu, antes você deve dar push no commit da versão atual (depois de `usethis::use_version()` e antes de `usethis::use_dev_version()`), pois assim estará pública no GitHub exatamente a versão do seu pacote que deve ser baixada pelos usuários. Depois basta ir para a aba das releases e clicar em **Draft a new release**. Preencha os campos correspondentes e publique a versão.

Releases Tags

v0.4.0 @ Target: master

Excellent! This tag will be created from the target when you publish this release.

Versão exemplo

Write Preview

Descrição das alterações

Attach files by dragging & dropping, selecting or pasting them.

Attach binaries by dragging & dropping, selecting or pasting them.

☐ This is a pre-release  
We'll point out that this release is identified as non-production ready.

Publish release Save draft

**Tagging suggestions**  
It's common practice to prefix your version names with the letter v. Some good tag names might be v1.0 or v2.3.4.  
If the tag isn't meant for production use, add a pre-release version after the version name. Some good pre-release versions might be v1.2.alpha or v1.0-beta.3.

**Semantic versioning**  
If you're new to releasing software, we highly recommend reading about [semantic versioning](#).

Uma forma eficiente de manter todas as alterações realizadas no pacote de forma organizada é com um arquivo `NEWS`. Ele pode ser criado com `usethis::use_news_md()` e criar um novo título para cada versão. Observe como esse arquivo é utilizado no `ggplot2`.