Pacotes

curso-r

Sumário

Sobre a Curso-R

A empresa

curso-r

R6 consultoria e treinamentos

Ministrantes

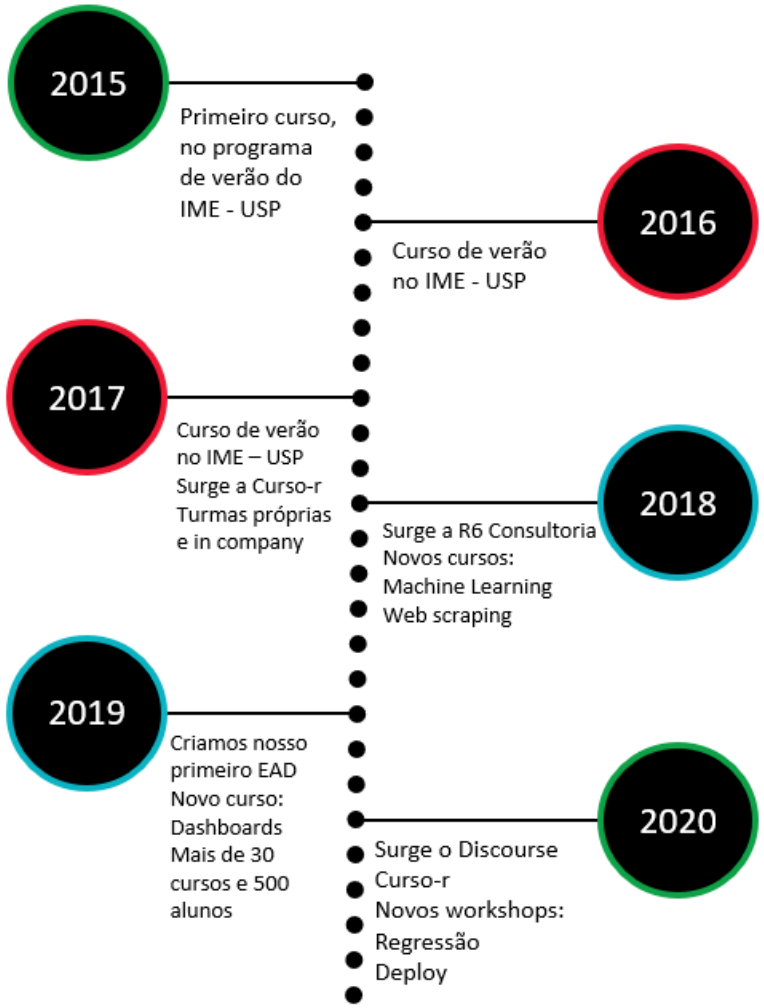
Caio Lente

Mestrando em Ciência da Computação no IME-USP e cientista de dados na Terranova Consultoria. Programador desde os 15 anos, começou a se apaixonar pelo R em 2016 e agora não fala em outra coisa. Metido a designer, maníaco da organização e metade texano.

Beatriz Milz

Doutoranda em Ciência Ambiental (PROCAM/IEE/USP) na Universidade de São Paulo. Co-organizadora da R-Ladies São Paulo. Instrutora da Carpentries, um projeto que tem como missão ensinar habilidades de ciência de dados para pessoas pesquisadoras. Instrutora de tidyverse certificada pela RStudio.

Linha do tempo



Nossos cursos

Programação em R

Introdução à programação com R

R para Ciência de dados I

R para Ciência de dados II

Pacotes

Introdução ao R com C++

Modelagem

Regressão Linear

Machine Learning

XGBoost

Deep Learning

Extração de dados

Faxina de dados

Web scraping

Comunicação e automação

Relatórios e visualização de dados

Dashboards com R

Deploy

curso-r

Sobre o curso

Nesse curso vamos falar de... 📦

- **Fundamentos de desenvolvimento de pacotes em R**
- **Documentação**
- **Testes unitários e consistência de código**
- **Disponibilizando seu pacote**

Informações gerais

- As aulas vão das 9h às 13, com uma pausa de 10 min em torno das 11:00
- As aulas serão gravadas e disponibilizadas no Google Classroom
- Podem mandar dúvidas no chat do Zoom ou abrir o microfone para perguntar
- Teremos bastante exercícios para resolver durante o workshop, então se prepare!

Informações de vocês

- Nós gostaríamos de saber sobre vocês (escreva no chat ✍😀):

    1. Qual é o seu nome?
    2. Com o que você trabalha?
    3. Qual é o seu pacote favorito?
    4. Como imagina usar pacotes no futuro?

Tirando dúvidas

- **Não existe dúvida idiota**.
- Fora do horário do workshop:
  - perguntas gerais sobre o workshop deverão ser feitas no Classroom.
  - perguntas sobre R, principalmente as que envolverem código, deverão ser enviadas no nosso discourse.
- Veja aqui dicas de como fazer uma boa pergunta.

Por que usar o discourse?

- Muito melhor para escrever textos que possuem códigos. Com ele, podemos usar o pacote `{reprex}`!
- Saber pesquisar sobre erros e fazer a pergunta certa é essencial para aprender e resolver problemas de programação.
- No discourse, teremos mais pessoas acompanhando e respondendo as dúvidas.
- Em um ambiente aberto, as suas dúvidas vão contribuir com a comunidade.

https://discourse.curso-r.com/

Preparando o ambiente de desenvolvimento !

Check-list antes de começar a aula

- R instalado + RStudio instalado
- Ferramentas de desenvolvimento: Windows: RTools instalado; Linux: r-base-dev; macOS: Xcode command line tools.
- Verificar se as ferramentas de desenvolvimento estão disponíveis: `devtools::has_devel()`

```
> devtools::has_devel()
# Your system is ready to build packages!
```

- Pacotes necessários instalados
- Git instalado + Conta no GitHub criada

.RData e .Rhistory

Os arquivos .RData e .Rhistory

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.

Leia mais neste capítulo do livro Zen do R.

Por que desistir do .RData e .Rhistory

- Se todos os resultados parciais de uma análise estiverem disponíveis a qualquer momento, **diminui o incentivo para a escrita de código reprodutível**.
- Se todo o histórico de comandos for acessível, **acaba a necessidade de experimentos controlados**.
- Ao dependermos ativamente do `.Rdata`, **se acidentalmente sobrescrevemos um objeto** relevante e o código para recriá-lo não estiver mais acessível, **não haverá nenhuma forma confiável de recuperá-lo**.
- A menos que pretendamos sentar com colegas para explicar como utilizar os objetos do `.RData` e do `.Rhistory`, **não pode-se esperar que outra pessoa seja capaz de reproduzir uma análise**.
- O R trata todos os objetos guardados na memória igualmente. Isso significa que ele também irá armazenar nos arquivos ocultos todas as bases de dados da sessão. Assim, **o `.RData` pode ser um arquivo de múltiplos gigabytes**.

Fundamentos de desenvolvimento de pacotes em R

Pacotes

Pacotes

Um pacote do R é uma forma específica de organizar seus código, seguindo o protocolo descrito pela R Foundation.

> Pacotes são a unidade fundamental de código R reprodutível.
> Hadley Wickham

- Pacotes incluem:
  - Funções em R
  - Documentação sobre como usá-las
  - Testes
  - Dados de exemplo

Motivação

"Por que vamos aprender a fazer um pacote se scripts funcionam bem o suficiente?" e "Por que divulgar meus pacotes em algum serviço open source?"

Porque:

- Compartilhar código é sempre uma boa ideia para que a comunidade se beneficie dos avanços individuais das pessoas que fazem parte
- Para sermos pessoas desenvolvedoras melhores, precisamos receber ajuda e sugestões de autras desenvolvedoras mais experientes
- É muito mais fácil usar controle de versão e integração contínua se você estiver programando um pacote
- Reprodutibilidade, Reprodutibilidade, Reprodutibilidade

Vantagens

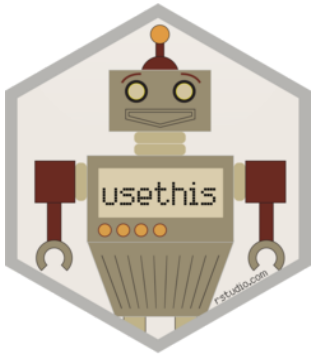
- Padroniza a organização dos códigos
- Integração com pacotes que aceleram desenvolvimento
- Motiva e facilita a documentação do código
- Facilita o compartilhamento e a reutilização de códigos em outros projetos e com outras pessoas

> Anything that can be automated, should be automated
>
> Tradução: Qualquer coisa que possa ser automatizada, deve ser automatizada.
>
> Fonte: Wickham & Bryan - Livro R Packages

Simplificando tudo: usethis

O pacote `{usethis}` ajuda com todo o fluxo de desenvolvimento em R.

Ele ajuda a criar arquivos, projetos, usar o Git, criar repositórios no GitHub e muito mais.



Apresentaremos várias funções do `{usethis}` ao longo deste tópico.

Nomes

Suponha que você criou uma função incrível que você quer compartilhar com o mundo. Ela tem várias funções auxiliares e um comportamento complexo o suficiente para depender de uma documentação... Você precisa criar um pacote.

Mas qual deve ser o seu nome?

> “There are only two hard things in Computer Science: cache invalidation and naming things.” --- Phil Karlton

Os melhores nomes são simples e descritivos. Pense em algo que possa ser procurado facilmente no Google e que, preferencialmente, seja em inglês (a menos que você não ache que pessoas de outro país usarão o seu pacote).

Evite usar letras maiúsculas ou mesmo números pois isso pode confundir os usuários. Você definitavente ganha pontos extras se você conseguir inserir alguma brincadeira com a letra R no nome (`stringr`, `decryptr`, `plyr`, `purrr`, etc.).

Dica para pesquisar nomes

```
available::available("stockfish")
# Urban Dictionary can contain potentially offensive results,
#   should they be included? [Y]es / [N]o:
# 1: Yes
# ── stockfish ──────────────────────────────
# Name valid: ✔
# Available on CRAN: ✖
# Available on Bioconductor: ✔
# Available on GitHub:  ✔
# Abbreviations: http://www.abbreviations.com/stockfish
# Wikipedia: https://en.wikipedia.org/wiki/stockfish
# Wiktionary: https://en.wiktionary.org/wiki/stockfish
# Sentiment:???
```

Criando um pacote

Para criar um pacote, usamos a função `usethis::create_package()`.

Tenha em mente que:

- Você deve passar um caminho como `~/Documents/MeuProjeto` e uma nova pasta chamada "Meu projeto" na pasta será criada dentro da pasta `Documents`. Essa pasta será tanto um `Rproj` quanto um pacote, ambos chamados `MeuProjeto`.
- Nomes de pacotes só podem conter letras, números e pontos, devem começar com uma letra e não podem acabar com um ponto.
- **Dica geral:** não adicione acentos, caracteres especiais e espaços no nome dos caminhos, arquivos, funções, pacotes, etc.

```r
usethis::create_package("~/Documents/Meuprojeto")
# ✓ Creating '/Users/beatrizmilz/Documents/Meuprojeto/'
# ✓ Setting active project to '/Users/beatrizmilz/Documents/Meuprojeto'
# ✓ Creating 'R/'
# ✓ Writing 'DESCRIPTION'
# Package: Meuprojeto
# 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] (YOUR-ORCID-ID)
# Description: What the package does (one paragraph).
# License: use_mit_license(), use_gpl3_license() or friends to
#     pick a license
# Encoding: UTF-8
# LazyData: true
# Roxygen: list(markdown = TRUE)
# RoxygenNote: 7.1.1
# ✓ Writing 'NAMESPACE'
# ✓ Writing 'Meuprojeto.Rproj'
# ✓ Adding '^Meuprojeto\\.Rproj$' to '.Rbuildignore'
# ✓ Adding '.Rproj.user' to '.gitignore'
# ✓ Adding '^\\.Rproj\\.user$' to '.Rbuildignore'
# ✓ Opening '/Users/beatrizmilz/Documents/Meuprojeto/' in new RStudio session
# ✓ Setting active project to '<no active project>'
```

Diretório de trabalho

O arquivo `NomeDoPacote.Rproj` indica para o RStudio que aquele diretório será a raiz de um projeto e que, sempre que o projeto estiver aberto, será utilizado por padrão como o diretório de trabalho.

Fixar o diretório de trabalho como a pasta raiz do projeto, ao lado da regra de manter todos os arquivos dentro da pasta do projeto, garante que sua análise poderá ser executada por qualquer pessoa e em qualquer computador sem a preocupação de ajustar caminhos até os arquivos utilizados ou criados pelo seu código.

Estrutura básica do pacote

Essa é a estrutura criada quando usamos a função `usethis::create_package()`:

- `MeuProjeto.Rproj`: este arquivo faz com que este diretório seja um projeto no RStudio (RStudio Project).
- `DESCRIPTION`: define o nome, descrição, versão, licença, dependências e outras caracaterísticas do seu pacote.
- `R/`: aqui ficam as funções desenvolvidas em R.
- `.Rbuildignore`: Lista arquivos que não devem ser incluídos ao compilar o pacote R a partir do código-fonte.
- `NAMESPACE`: Não devemos editar este arquivo manualmente. O NAMESPACE declara: as funções que o pacote exporta para uso externo e as funções externas que seu pacote importa de outros pacotes.

Licença de uso

- `LICENSE`: especifica os termos de uso e distribuição do seu pacote.
- O pacote `{usethis}` possui algumas funcões para nos ajudar com isso. Por exemplo:

```
usethis::use_cc0_license()
usethis::use_mit_license()
usethis::use_mit_license()
```

No geral, as funções tem essa estrutura:

```
use_nome_da_licenca_license()
```

Modifique o código

Recarregue com devtools::load_all()

Explore no console

Parte prática 

- Crie um pacote, usando a função `usethis::create_package()`.
- Observe a estrutura do diretório
- Carregue o pacote com `devtools::load_all()`. Cheque se está tudo OK com o seu pacote com `devtools::check()` .
- Adicione uma licença no pacote, com `usethis::use_***_license()`.
- Altere o DESCRIPTION: nome da pessoa autora, descrição do pacote.

Funções

A pasta R

Dentro de um pacote, a pasta `R/` só pode ter funções.

Uma função é responsável por executar uma tarefa pequena, mas muito bem. Quando trabalhamos com funções, nossas operações ficam mais confiáveis.

A ideia da pasta `R/` é guardar em um local comum tudo aquilo que nós utilizamos como ferramenta interna para nossas análises, bem como aquilo que queremos que outras pessoas possam usar no futuro.

Podemos usar a função `usethis::use_r("nome_da_funcao")` para que um arquivo seja criado para começarmos a escrever uma função.

Assim que escrevermos/modificarmos alguma função, podemos carregá-las para testá-las manualmente com a função `devtools::load_all()`

Vantagens de usar funções

- 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.
- Programas com funções normalmente são muito mais enxutos e limpos do que *scripts* soltos, pois estes estimulam repetição de código.
- 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.

Leia mais neste capítulo do livro R for Data Science.

Criando a sua própria função

Quando estamos desenvolvendo pacotes, iremos criar funções para executar as tarefas necessárias.

A sintaxe é a seguinte:

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

  # Código que a função irá executar

}
```

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

Dependências

Sem os inúmeros pacotes criados pela comunidade, o R provavelmente já estaria no porão da Ciência de Dados.

Por isso, é a primeira coisa que escrevemos nos nossos *scripts* quase sempre é `library(algumPacoteLegal)`.

Quatro pontos

Quando lidamos com pacotes, a função `library()` não é utilizada, e todas as funções devem ter seus pacotes de origem explicitamente referenciados pelo operador `::`.

Existem vantagens de se fazer isso. E, se serve como 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.

Vantagens dos quatro pontos

- 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.
- O uso do `::` é fundamental na criação e organização de pacotes.

Documentar as dependências

Além de usar o `::` ao usar funções de outros pacotes, é importante também que essas dependências sejam documentadas na seção Imports do arquivo DESCRIPTION.

Existem 3 funções principais para isso:

- `usethis::use_package("nome_do_pacote")`: Essa função permite adicionar pacotes que foram instalados via CRAN na lista de Imports.
- `usethis::use_dev_package("nome_do_pacote")`: Essa função permite adicionar pacotes que não foram instalados via CRAN (ex. instalados via GitHub) na lista de Imports.
- `usethis::use_pipe()`: Se você usa o pipe nas funções do seu pacote, será necessário usar essa função.

Discussão em sala

- Como podemos transformar o código a seguir em uma função que faz parte de um pacote?
- Exemplo: Importar a base mananciais, e calcular a média do volume de água armazenado (em %) para o sistema Cantareira, em abril de 2021.

```
library(tidyverse)
library(lubridate)
url_mananciais <- "https://github.com/beatrizmilz/mananciais/raw/master/inst/extdata/mananc
mananciais <- read_csv2(url_mananciais)
mananciais %>%
  mutate(ano = year(data), mes = month(data)) %>%
  group_by(sistema, ano, mes) %>%
  summarise(media_volume_porcentagem = mean(volume_porcentagem)) %>%
  filter(ano == 2021, mes == 4, sistema == "Cantareira")
```

Código original | Passo 1 | Passo 2 | Passo 3

```
library(tidyverse)
library(lubridate)
url_mananciais <- "https://github.com/beatrizmilz/mananciais/raw/master/inst/extdata/mananc
mananciais <- read_csv2(url_mananciais)
mananciais %>%
  mutate(ano = year(data), mes = month(data)) %>%
  group_by(sistema, ano, mes) %>%
  summarise(media_volume_porcentagem = mean(volume_porcentagem)) %>%
  filter(ano == 2021, mes == 4, sistema == "Cantareira")
```

Pensar em quais podem ser os argumentos da nossa função. E se eu quiser calcular a média em outro mês? Outro ano? Outro sistema?

```
calcular_media_volume <- function(sistema, ano, mes) {
  url_mananciais <-
    "https://github.com/beatrizmilz/mananciais/raw/master/inst/extdata/mananciais.csv"
  mananciais <- readr::read_csv2(url_mananciais)
  mananciais %>%
    dplyr::mutate(ano = lubridate::year(data), mes = lubridate::month(data)) %>%
    dplyr::group_by(sistema, ano, mes) %>%
    dplyr::summarise(media_volume_porcentagem = base::mean(volume_porcentagem)) %>%
    dplyr::filter(ano == {{ano}}, mes == {{mes}}, sistema == {{sistema}})
}
```

Exemplo de uso:

```
> calcular_media_volume(sistema = "Guarapiranga", ano = 2021, mes = 3)
```

Recomendações

Algumas recomendação sobre como organizar seu código:

- Evite usar . no nome das suas funções (hoje em dia usar _ é muito mais comum)
- Use nomes descritivos para as funções, pois isso facilita a manutenção e o uso do pacote
- Tente se limitar a 80 caracteres por linha porque isso permite que seu código caiba confortavelmente em qualquer tela
- Não use `library()` ou `require()`, pois isso vai causar problemas (use a notação `pacote::função()`)
- Nunca use `source()`, todo o código já será carregado automaticamente
- Não adicionar "metapackages" no Imports (como o tidyverse).

Parte prática 

- Crie um arquivo onde iremos escrever a função para o pacote. Dica : use a função `usethis::use_r("nome_funcao")` !
- Copie o código disponível neste arquivo e cole no arquivo criado. (precisamos criar o código)
- Adapte o código e o transforme em uma função! Não esqueça das dependências.
- Verifique que o arquivo onde sua função foi escrita está no diretório `R/`, e que as dependências estão descritas no arquivo DESCRIPTION.
- Carregue o pacote com `devtools::load_all()`, e confira se está tudo ok com `devtools::check()` !

Bases de dados

Dados

Se você quiser inserir dados ao seu pacote, você pode utilizar a função `usethis::use_data(meus_dados)`.

Ela criará uma pasta `data/` na raiz do seu pacote, caso ela não exista ainda, e salvará nela o objeto `meus_dados` em formato `.rda`.

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.

Manipulando dados crus

Se a base que você quiser colocar no pacote for o resultado de um processo de manipulação de uma base crua, você pode salvar o código desse processo na pasta `data-raw`.

Para isso, utilize a função `usethis::use_data_raw("meus_dados")`. Ela criará uma pasta `data-raw/` na raiz do seu pacote, caso ela não exista ainda, e um arquivo `meus_dados.R` onde você colocará o código de manipulação da base crua.

Qual a diferença entre R/ e data-raw/?

`data-raw`

- A pasta `data-raw/` é sua caixa de areia.
- Apesar de existirem formas razoáveis de organizar seus pacotes aqui, nessa parte você será livre.

`R/`

- Já a pasta `R/` conterá funções bem organizadas e documentadas.
- Por exemplo, uma função que ajusta um modelo estatístico, outra que arruma um texto de um jeito patronizado, ou uma que contém seu tema customizado do `ggplot`.
- Dentro dessa pasta você não deve carregar outros pacotes com `library()`, mas sim usar o operador `::`.

Documentação

Documentação de funções

Se quisermos adicionar uma documentação para o nosso pacote (as instruções que aparecem quando vamos usar uma função ou o documento mostrado quando rodamos `?função()`) precisamos usar um comentário especial: `#'`

```
#' Título da função
#'
#' Descrição da função
#'
#' @param a primeiro parâmetro
#' @param b segundo parâmetro
#'
#' @return descrição do resultado
#'
#' @export
fun <- function(a, b) {
  a + b
}
```

Documentação de funções (cont.)

- O parâmetro `@export` indica que a função ficará disponível quando rodarmos `library(MeuProjeto)`. Não se esqueça de exportar todas (e somente) as funções públicas!
- O RStudio disponibiliza um atalho para criar a estrutura da documentação de uma função. No menu superior, clique em `Code` -> `Insert Roxygen Skeleton`.
- Para deixar a documentação das suas funções acessível (no help do R), use a função `devtools::document()`. Atalhos: Ctrl + Shift + D (Windows e Linux) ou Cmd + Shift + D (macOS).

Documentação de bases de dados

falar sobre documentação de base de dados

Acentos, encoding e variáveis globais

Acentos, encoding e variáveis globais

- Prefira manter os arquivos em inglês ou em ASCII (sem acentos) para que seu pacote possa ser submetido no CRAN; se precisar escapar strings, dê uma olhada no add-on `abjutils::escape_unicode()`

```
- utils::globalVariables(c("variavel1", "variavel2"))
```

Comunicação

Vignettes

É muito comum a construção de *vignettes* para documentar o pacote. Elas são documentos em HTML melhor formatados do que a tradicional documentação do R.

Você pode usar a função `usethis::use_vignette()` para criar *vignettes*.

Outros

Se você precisar construir sites, relatórios, dashboards estáticos (flexdashboard) dentro do seu pacote, você pode criar uma pasta chamada `docs/` na raiz do seu projeto para guardar esses arquivos.

Veremos também como criar um site do pacote criado com o `{pkgdown}`.

Boas práticas no desenvolvimento

- Não rode as funções diretamente. Utilize sempre a função `devtools::load_all()`. Ela carrega todas as funções da pasta `R/` e as bases salvas na pasta `data/`. Isso diminuirá a chance de elas estarem sendo afetadas por valores externos que estão no seu *Environment*.
- Limpe o seu *Environment* sempre que possível. Um atalho útil: `CTRL+SHIFT+F10`.
- Para deixar a documentação das suas funções acessível (no help do R), use a função `devtools::document()`.
- Se você precisar instalar o seu pacote (equivalente ao que fazemos com pacotes do CRAN quando rodamos `install.packages()`), use a função `devtools::install()`. Ela deve ser utilizada quando o seu pacote estiver pronto (ou pelo menos alguma versão dele).

Testes unitários e consistência de código

Leia mais neste capítulo do livro Zen do R.

Testes

Se quisermos verificar que todo o pacote continua funcionando mesmo depois fazer alguma alteração, precisamos de testes automatizados.

Para isso, basta rodar `usethis::use_testthat()` (apenas uma vez por pacote) e depois `usethis::use_test("nome_do_teste")` (para cada novo arquivo de testes que quiser criar).

Com o pacote testthat podemos criar quantos arquivos de testes quisermos, cada um com um número ilimitado de testes e sub-testes. Quando tivermos todos os testes prontos, basta rodar `devtools::test()`. Atalho: Ctrl + Shift + T (Windows e Linux) ou Cmd + Shift + T (macOS).

Exemplo de teste

Considerando a função:

```
tira_media <- function(x, rm_na = TRUE) {
  purrr::reduce(x, sum, na.rm = rm_na)/conta_itens(x, rm_na)
}
```

Podemos escrever os seguintes testes:

```
test_that("taking the mean works", {
            expect_equal(tira_media(c(1, 2, 3, 4, NA, 6)), 3.2)
            expect_equal(tira_media(c(1, 2, 3, 4, 6)), 3.2)
          })
test_that("rm_na works as expected", {
            expect_output(tira_media(c(1, 2, NA), rm_na = FALSE), NA)
            expect_equal(tira_media(c(NA, NA, NA)), NaN)
          })
```

Função check

Podemos usar a função `devtools::check()`. Ela carrega todo o código, gera toda a documentação e executa todos os testes, verificando em todo passo se tudo está funcionando como o esperado.

Disponibilizando seu pacote

Git e GitHub

Git

- Git é um **sistema de versionamento**, criado por Linus Torvalds, autor do Linux.
- É capaz de guardar o histórico de alterações de todos os arquivos dentro de uma pasta, que chamamos de repositório.
- Funciona como o "*Track changes*" do word, mas muito melhor.
- Torna-se importante à medida que seu trabalho é **colaborativo**.
- **Git é um software que você instala no computador.**
- Arquivo `.gitignore`: Lista arquivos que deverão ser ignorados ao versionar o pacote com Git.

GitHub

- GitHub é um site onde você coloca e compartilha repositórios Git.
- Utilizado por milhões de pessoas em projetos de código aberto ou fechado.
- Útil para colaborar com outros programadores em projetos de ciência de dados.
- Existem alternativas, como GitLab e BitBucket.
- **GitHub é um site que você acessa na internet.**

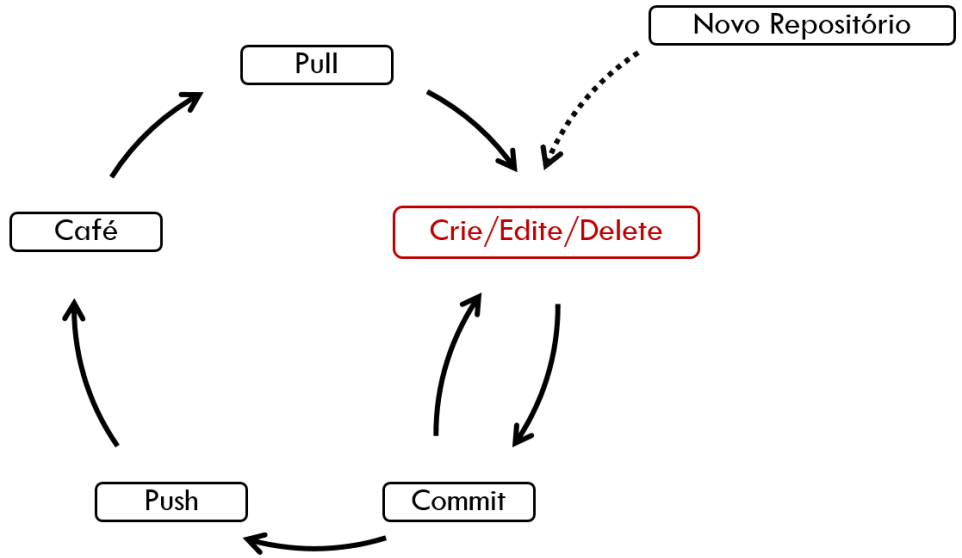
Pacotes e GitHub

Pacotes do R e repositórios do GitHub são melhores amigos.

O grande cupido dessa amizade é o `{usethis}`.

Fluxo de trabalho

O diagrama abaixo exemplifica o fluxo de trabalho de um projeto com versionamento.



Configure o Git e o GitHub no seu RStudio

Antes de começarmos a versionar o código do nosso pacote, vamos configurar o Git e o GitHub no RStudio.

Esse processo precisa ser feito apenas uma vez!

Configure seu usuário do Git

```
usethis::use_git_config(
  user.name = "SEU NOME NO GITHUB",
  user.email = "seu_email_no@github.com"
)
```

- Em `user.name`, pode ser seu nome mesmo, não precisa ser o nickname.
- O `user.email` precisa ser o que está vinculado à sua conta do GitHub.

Configure o Personal Access Token

- Ao conectar com o GitHub, você será instruída(o) a criar um *Personal Access Token* (PAT).
- O PAT serve para autenticar ao GitHub, podendo ser utilizado como senha de acesso ou internamente para automatizar tarefas (como criar um repositório).
- Para criar um novo PAT, use a função `usethis::create_github_token()`. Uma janela do navegador será aberta, e você deve autenticar no GitHub (se necessário), criar o novo token, e copiá-lo.
- Use a função `usethis::edit_r_environ()` para abrir o arquivo `.Renviron` para salvar seu token. Use essa estrutura, substituindo os 0 pelo código copiado na etapa anterior:

```
GITHUB_PAT="0000000000000000000000000000000000000000"
```

- Lembre-se de reiniciar sua sessão do R!

Usar o Git para versionar seu pacote

Passo 1: crie e configure seu pacote

```
usethis::create_package("meuPacote")
```

Lembrando que nomes de pacotes

- Só podem ter letras, números e ponto
- Devem começar com uma letra
- Não podem acabar com ponto

Boa prática: Não é aconselhável usar pontos no nome de pacotes. (fonte)

Passo 2: Versione com o Git

```
usethis::use_git()
```

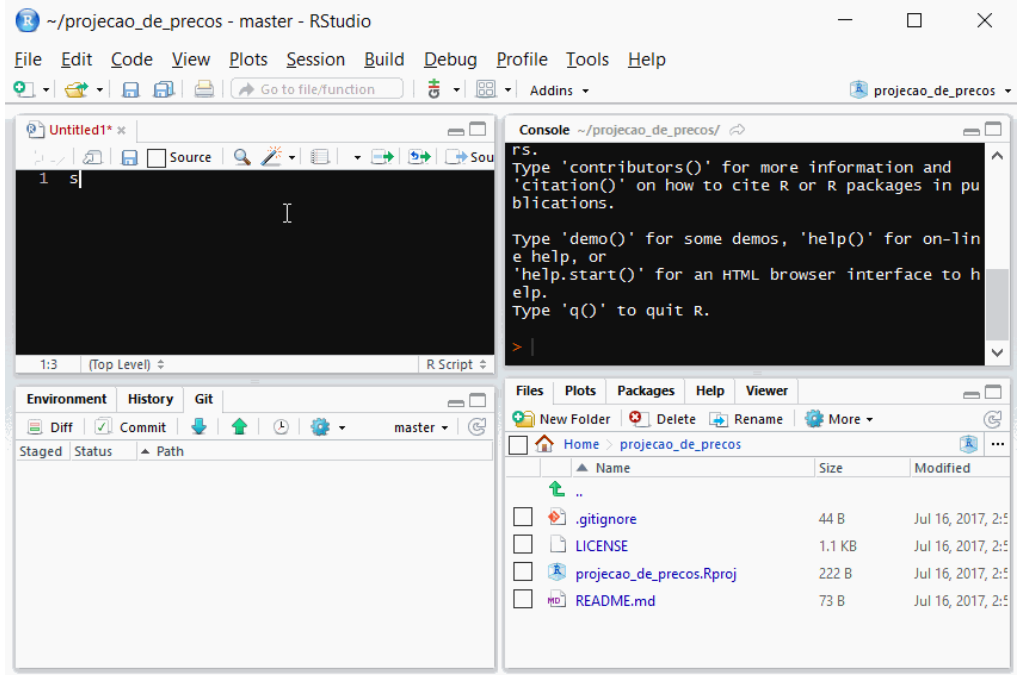
- Rodando o comando acima na pasta do projeto (a nova aba do RStudio que apareceu) você adiciona controle de versão.
- Você receberá algumas instruções para seguir, mas está tudo certo.

Passo 3: Adicione o GitHub

```
usethis::use_github()
```

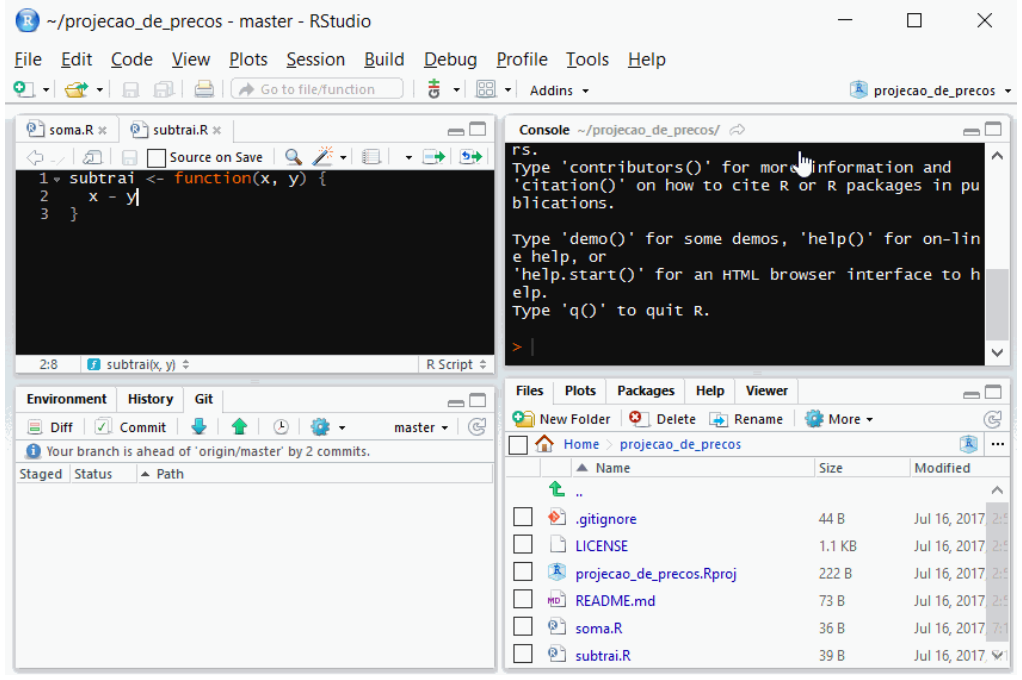
- O comando acima sincroniza a pasta com o GitHub.
- Mais uma vez, você receberá algumas instruções, mas lembre-se apenas de selecionar o método de autenticação `https`.

Passo 4: Stage & Commit



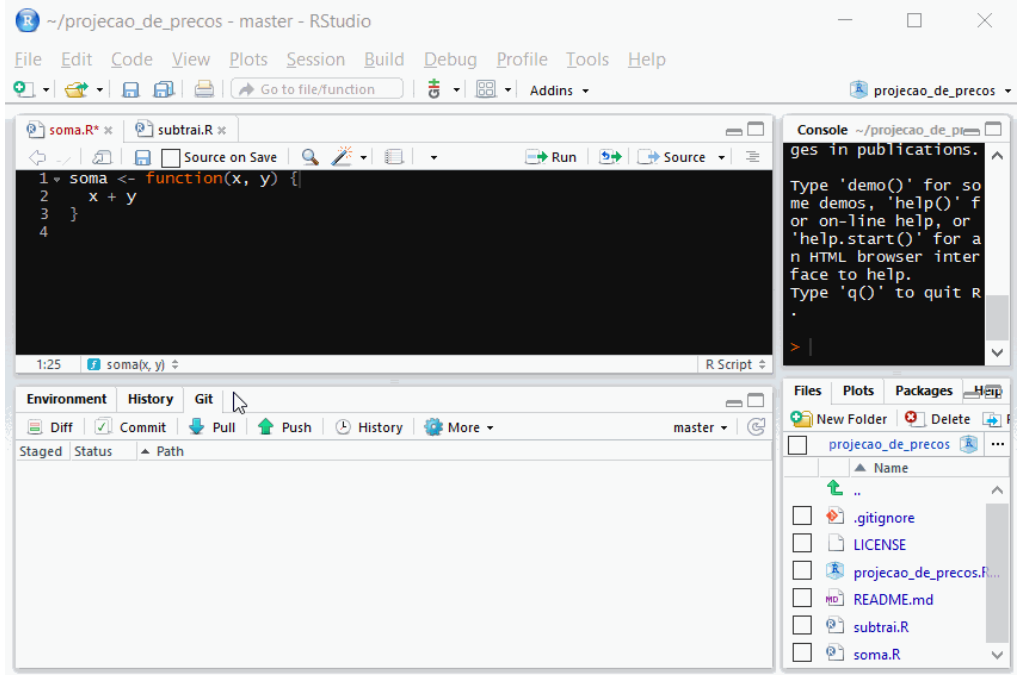
- Nesta etapa, você estará descrevendo as modificações que fez nos arquivos selecionados.
- **Observação**: o ato de clicar no item é o passo de Stage.

Passo 5: Push



- *Push* (ou *dar push*) significa atualizar o seu repositório remoto (GitHub) com os arquivos que você *commitou* no passo anterior.

Passo Extra: Pull



- *Pull* é a ação inversa do *Push*: você trará a versão mais recente dos arquivos do seu repositório remoto (GitHub) para a sua máquina (caso você tenha subido uma versão de um outro computador ou uma outra pessoa tenha subido uma atualização).

Resumo

1. Repositório: Criar projeto/pacote
2. Adicionar Git
3. Adicionar GitHub
4. Commit: Edite e "Commite" as mudanças no código
5. Push: Suba os commits para o GitHub Extra. Pull: Baixe o estado atual do projeto

Onde podemos falar do usethis::use_readme_rmd() + usethis::build_readme() ?

Cuidados

- Se uma base de dados tem mais do que 50Mb de tamanho, ela não deveria estar no seu repositório.
- Nem sempre o comando Pull dá certo. Às vezes, você e a colega de trabalho fizeram mudanças no mesmo arquivo e, quando vão juntar, ocorre um conflito.

Criando um site para o pacote com {pkgdown}

O pacote pkgdown permite criar um site para o pacote. Para criar, use a função a seguir uma vez, para configurar o pacote para usar o pkgdown:

```
usethis::use_pkgdown()
```

Para atualizar o site, use a função:

```
pkgdown::build_site()
```

Criando um site para o pacote com {pkgdown} (cont.)

Conteúdo do site será gerado a partir dos documentos já existentes no pacote:

- O conteúdo do site estará no diretório `docs/`.
- O arquivo `README.md` será usado para criar a página principal do site;
- A documentação das funções serão usadas para criar a seção 'references'
- As vignettes serão usadas para criar a seção 'articles'

Integração contínua com GitHub Actions

Escrever!

Regras para colocar um pacote no CRAN

Um pacote é uma coleção de código, dados, documentação e testes que qualquer pessoa pode instalar em sua máquina.

Se quisermos criar apenas um conjunto de funções que provavelmente não serão utilizadas por muitas pessoas, podemos subir esse pacote para o GitHub e mantê-lo lá somente para garantir controle de versão.

Mas se quisermos que o máximo número possível de pessoas tenha acesso ao nosso pacote, pode ser que precisemos subi-lo para o CRAN (Comprehensive R Archive Network). Neste caso precisaremos criar teste e documentação (em inglês) para nosso pacote.

Use a função `devtools::submit_cran()` para auxiliar no processo de submissão do pacote para o CRAN.

Escrever!

Resumo

Resumo

- Criar um pacote usando a função `usethis::create_package("~/caminho/ate/o/pacote")`
- No arquivo DESCRIPTION, adicionar o nome das pessoas autoras, além do título e descrição do pacote.
- Adicionar uma licença com `usethis::use_**_licence()`
- Versionar o projeto usando `usethis::use_git()`
- Criar funções: a função `usethis::use_r("nome_da_funcao")` cria o arquivo para isso

Resumo

- A funcão `devtools::load_all()` simula o processo de instalação e carregamento do pacote. As funções criadas ficam disponíveis para uso. Atalho: Ctrl + Shift + L (Windows e Linux) ou Cmd + Shift + L (macOS).

- A função `devtools::check()` verifica se o pacote está funcionando. Pode apresentar erros, avisos e notas. Leia a mensagem no console :)

Referências e materiais para estudo

- Zen do R, livro em desenvolvimento pela Curso-R.
- R Packages, livro aprofundado sobre desenvolvimento de pacotes.
- R for Data Science - capítulo sobre Funcões
- Materiais da R-Ladies SP sobre a Hacktoberfest 2020.
- Folha de dicas do Git em Português