



# Fundamentals of Package Development

**Andy Teucher**  
**posit::conf(2023)**  
**Hyatt Regency Hotel, Chicago**

# Housekeeping

- WiFi:
  - Network: Posit Conf 2023
  - Password: conf2023
- There are **gender-neutral bathrooms** located are among the Grand Suite Bathrooms.
- There are two **meditation/prayer rooms**: Grand Suite 2A and Grand Suite 2B. Open Sunday - Tuesday 7:30 a.m. - 7:00 p.m., Wednesday 8:00 a.m. - 6:00 p.m.
- The **lactation room** is located in Grand Suite 1. Open Sunday - Tuesday 7:30 a.m. - 7:00 p.m., Wednesday 8:00 a.m. - 6:00 p.m.
- Participants who do not wish to be photographed have **red lanyards**; please note everyone's lanyard colors before taking a photo and respect their choices.

# Code of Conduct & COVID Policies

<https://posit.co/code-of-conduct/>

Everyone who comes to learn and enjoy the experience should feel welcome at posit::conf. Posit is committed to providing a professional, friendly and safe environment for all participants at its events, regardless of gender, sexual orientation, disability, race, ethnicity, religion, national origin or other protected class.

This code of conduct outlines the expectations for all participants, including attendees, sponsors, speakers, vendors, media, exhibitors, and volunteers. Posit will actively enforce this code of conduct throughout posit::conf.

## Reporting:

- any posit::conf staff member
- [conf@posit.com](mailto:conf@posit.com)
- 844-448-1212

# Welcome!

This is a one-day course for people looking to learn how to build R packages in an efficient way, make them easy to maintain, and easy for users to use.

- Introductions
  - Instructor
  - TAs:
    - Michele Peruzzi
    - Thomas Pedersen
  - Yourselves

# Welcome!

This is a one-day course for people looking to learn how to build R packages in an efficient way, make them easy to maintain, and easy for users to use.

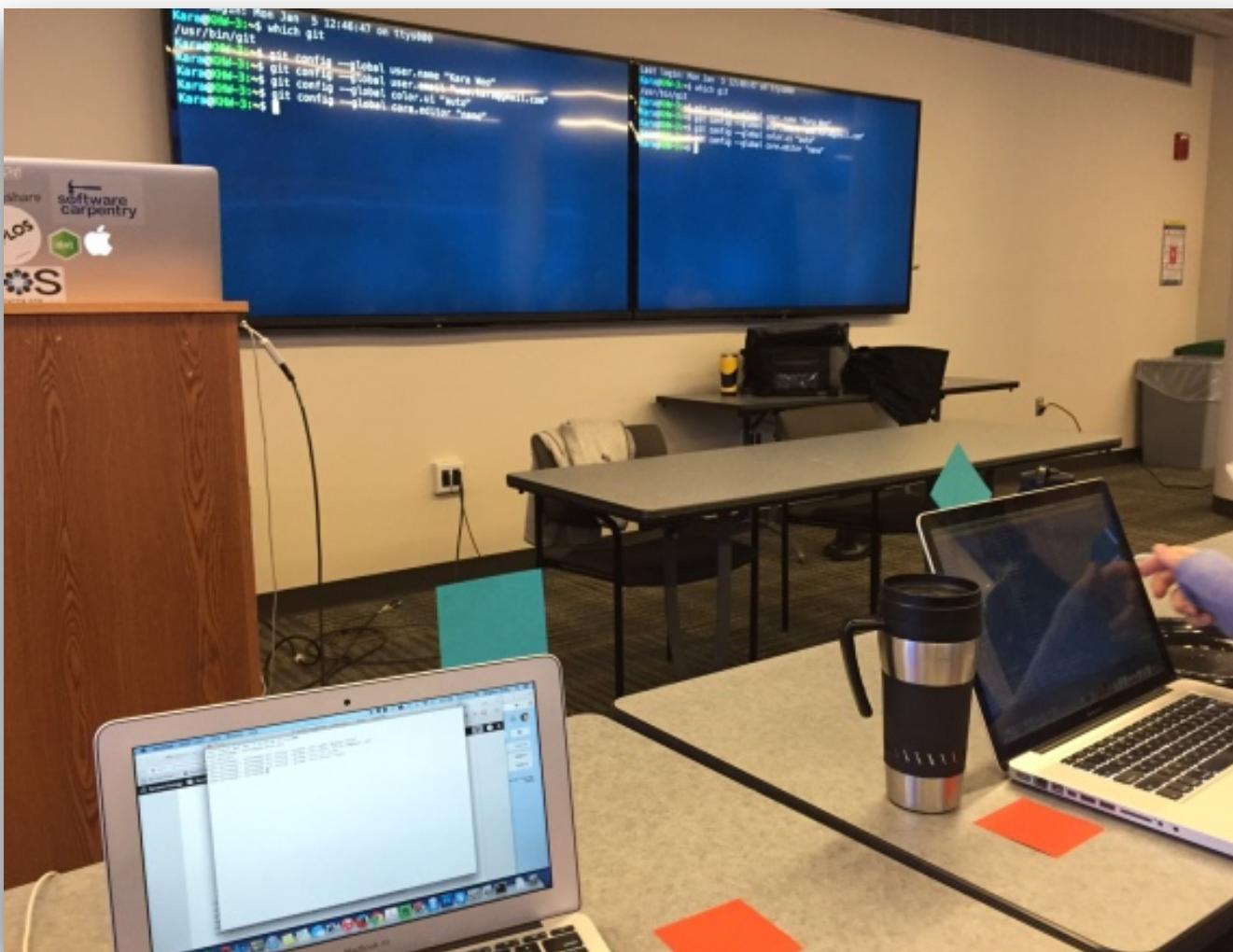
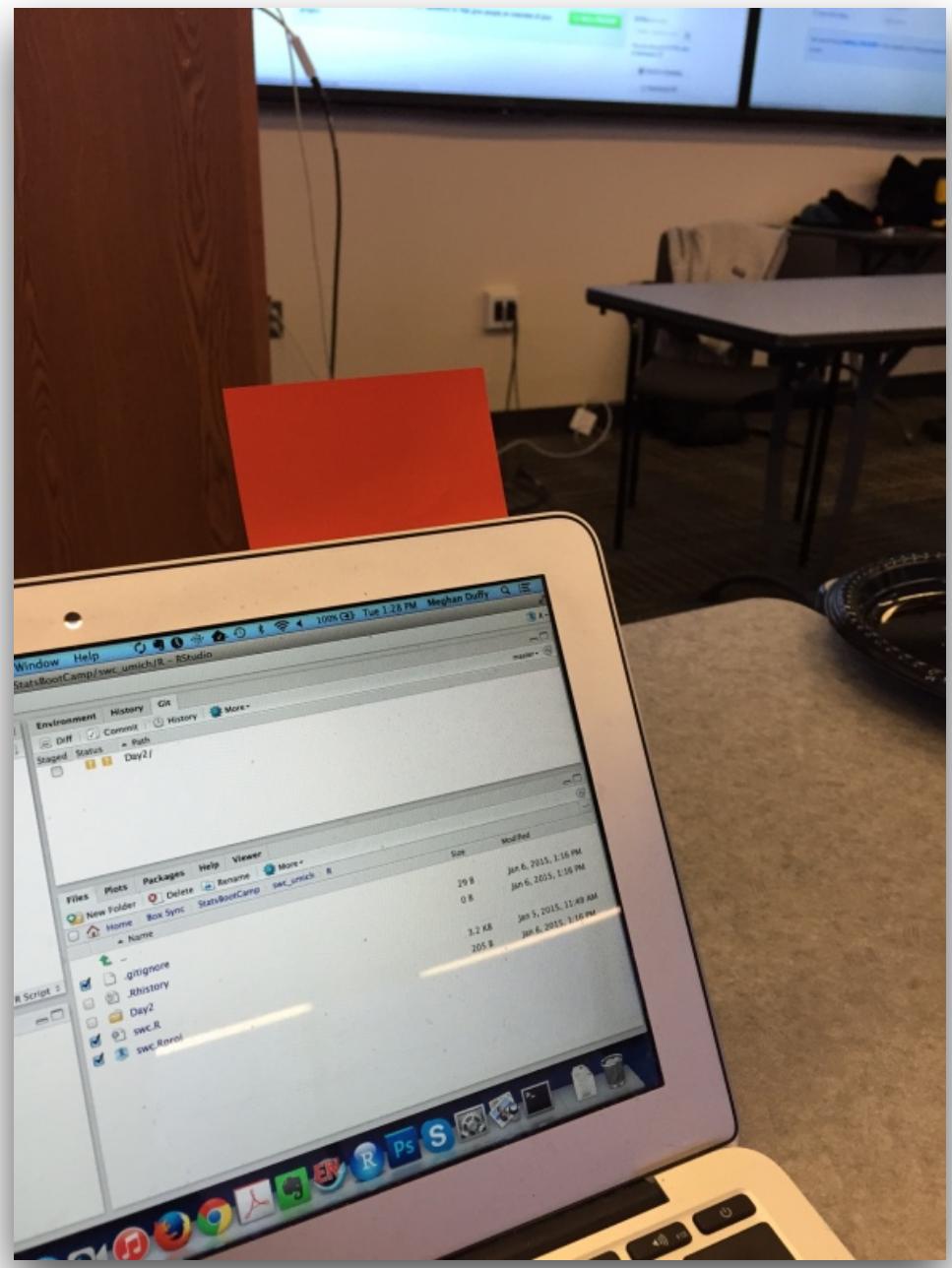
- Introductions
  - Instructor
  - TAs:
    - Nic Crane
    - Simon Couch
  - Yourselves

# Discord

1. Go to your event portal: <http://pos.it/conf-event-portal>
2. Click on the image that says, “Click here to join Discord, the virtual networking platform!”.
3. Accept the invitation, and review and agree to the code of conduct.
4. Click "Browse Channels" in the top-left corner and find **#fundamentals-of-package-development**
5. Click the checkbox to join the channel
6. Click on the channel in the left sidebar.
7. Introduce yourself!



# Sticky Notes



# Schedule and Learning Objectives

---

- What is a package and why should you make one? START: 9:00
- Package Structure and State - where do they come from, where do they live?

---

- Package Creation and Metadata BREAK: 10:30 - 11:00
- Documentation

---

- Testing LUNCH: 12:30 - 1:30
- Package Dependencies

---

- Continuous Integration BREAK: 3:00 - 3:30
- Package Website & Vignettes
- Package Distribution (CRAN)

---

END: 5:00

# R Packages (2e)

**Hadley Wickham**

**Jenny Bryan**

<https://r-pkgs.org>

O'REILLY®

Second  
Edition

# R Packages

Organize, Test, Document, and Share Your Code



Hadley Wickham  
& Jennifer Bryan

# Packages in a nutshell



# Why make a package?



- Easier to reuse functions you write
- A consistent framework which encourages you to better organize, document, and test your code
- This framework means you can use many standardized tools
- Easiest way to distribute code (and data)
  - To your team
  - To the world

# Script vs Package

<https://r-pkgs.org/package-within.html>

## Script

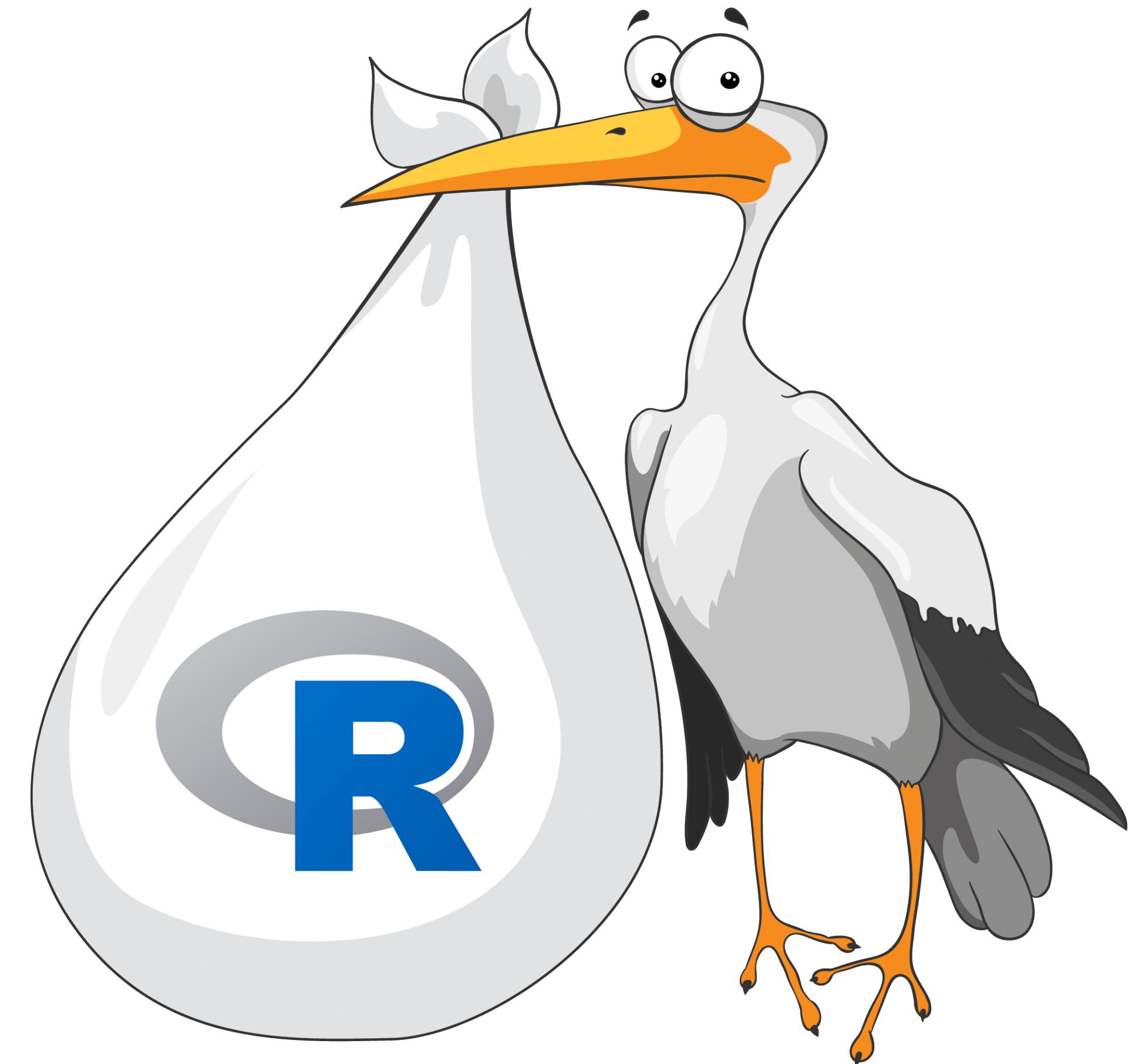
- Performs data analysis
- Collection of one or more `.R` files
- `library()` calls
- Documentation in `#` comments
- Run with `source()` or `select+run`

## Package

- Reusable functions to use in analyses
- Defined by particular file organization
- Required packages in `DESCRIPTION`
- Documentation in `Roxygen` comments and “man” files
- Functions available when package attached

# Where do packages come from?

- Discuss with your neighbour and put up a green sticky when you have:
  - Your favourite package
  - **2 places** from which you install packages
  - **2 functions** you can use to install packages
- Write them in the Discord



# R Libraries - where do packages live?

- A **library** is a directory containing installed **packages**
- You have at least one library on your computer
- Common (and recommended) to have two libraries:
  1. A **system** library with **base** (14) and **recommended** (15) packages; installed with R.
  2. A **user** library with user-installed packages
- We use **library(pkg)** function to **attach** a package
- 7 base packages are always attached (**base**, **methods**, **utils**, **stats**, **grDevices**, **datasets**, **graphics**)

# Your turn

Type `.libPaths()` to see your libraries

- How many libraries do you have?
- What are they? (Put them in the Discord)

# Package Structure and State

## Five forms

### Source

- Directory of files with specific structure
  - What you interact with as you build a package
- 

### Bundle

- Package compressed into a single file (tar.gz) via `devtools::build()` -> `R CMD build`
  - Vignettes are built and files listed in `.Rbuildignore` are left behind
- 

### Binary

- Platform-specific compressed file (.tgz, .zip)
  - Made with `devtools::build(binary = TRUE)` -> `R CMD INSTALL --build`
- 

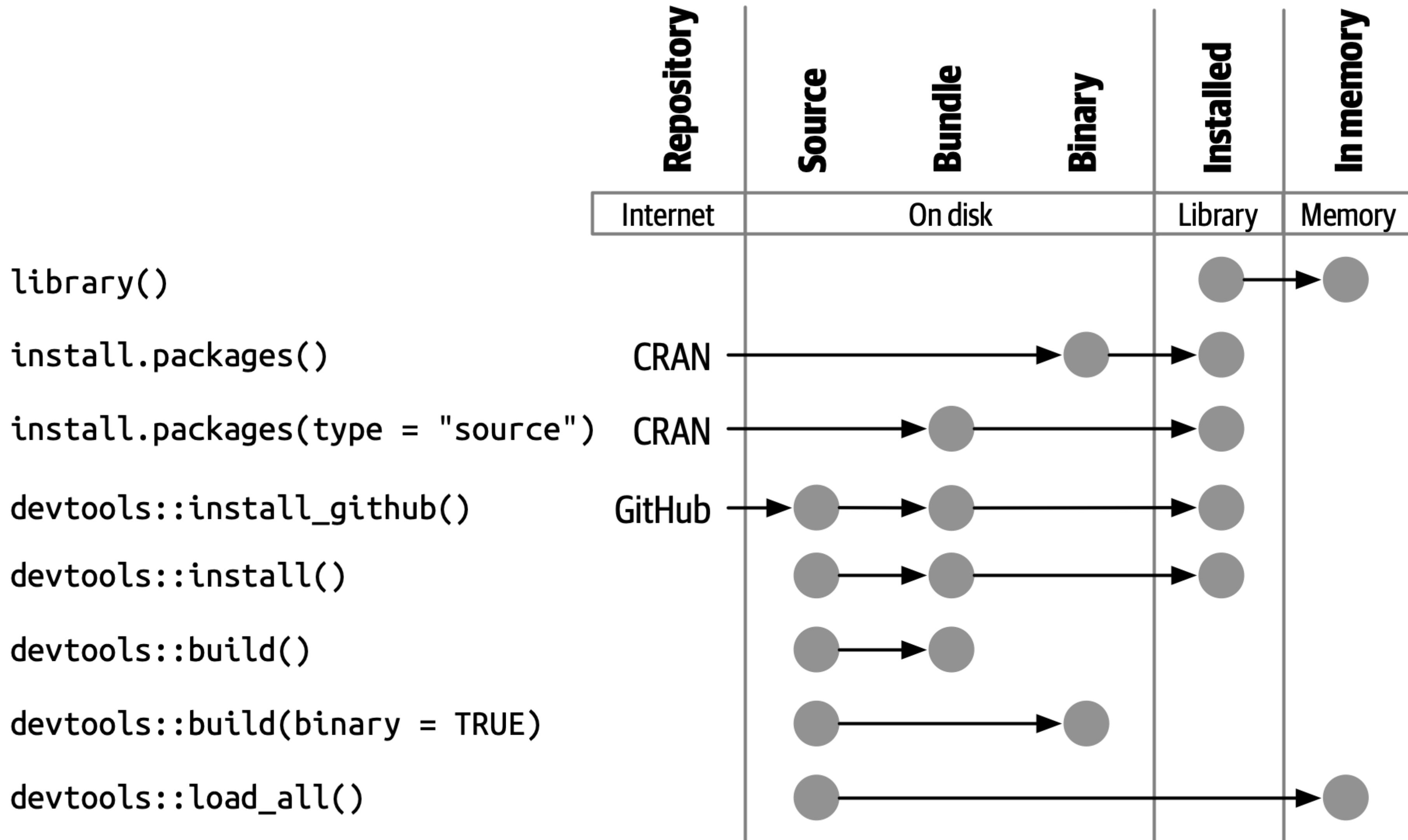
### Installed

- Binary package decompressed into a user's library
  - `install.packages()`
- 

### In Memory

- Loaded and ready for use in an R session
- `library()`

# Package Structure and State



# Let's make a package together

## We will:

- Create a simple package
- Use git to track our changes
- Push the code to a repository on GitHub
- Create tests for our functions
- Create documentation for our functions
- Create a package website (if we have time)

## We won't:

- Talk (much) about function writing and design
- Talk about how to include data in your package

# libminer

**Sneak peak of our end goal on GitHub**

- <https://github.com/ateucher/libminer.master>
- A package to explore our local R package libraries

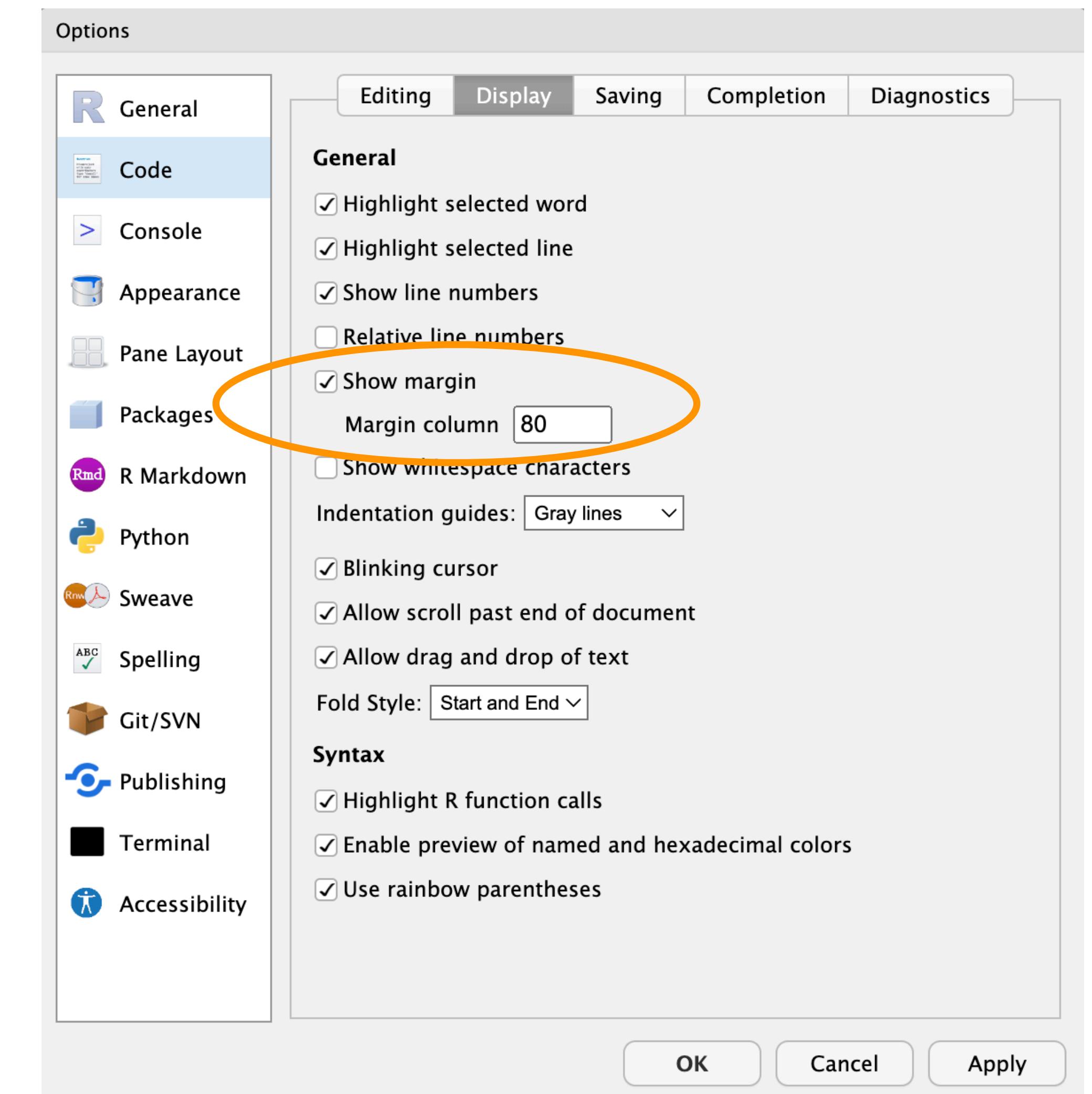
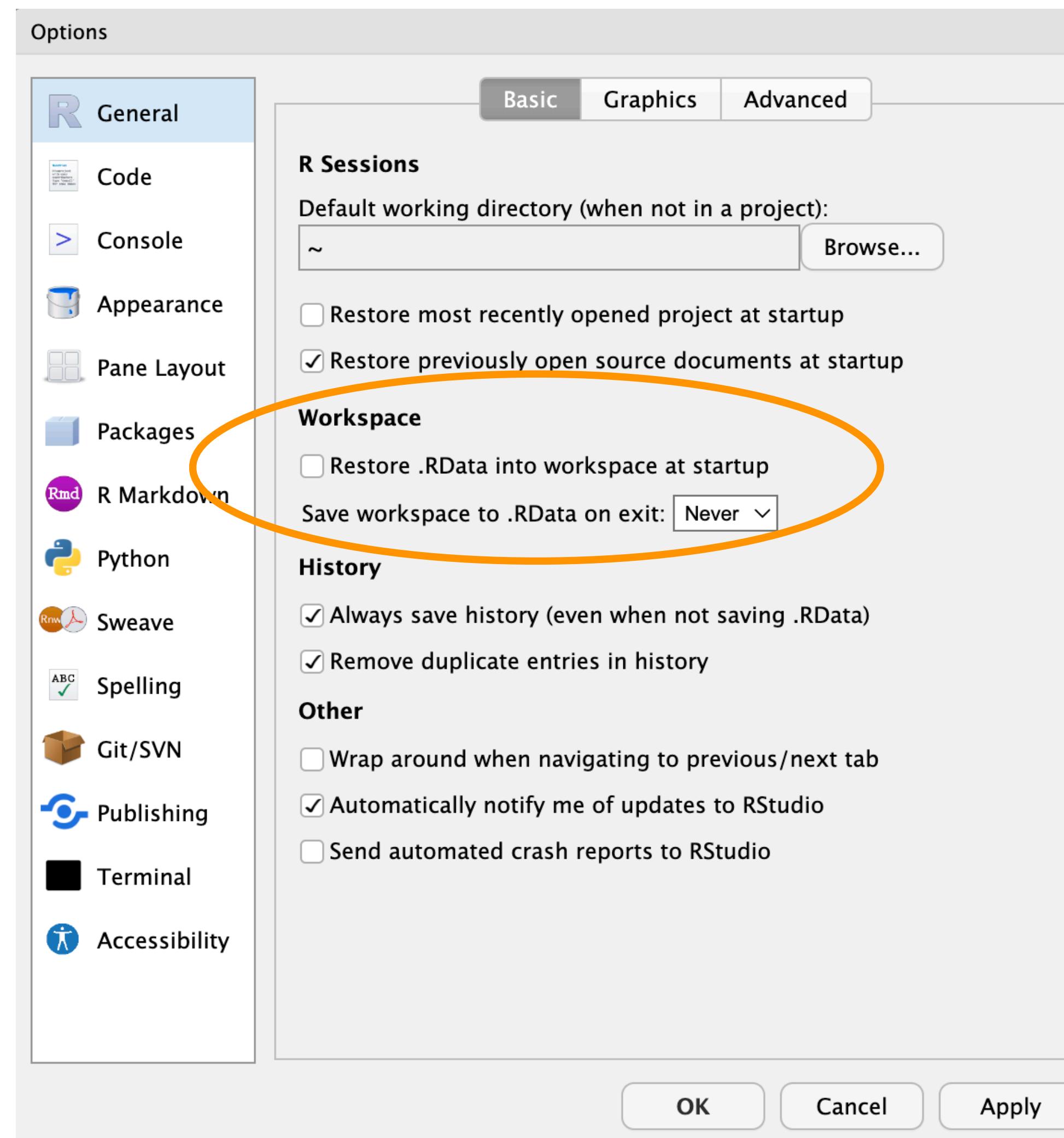


# Get Ready



# Configure RStudio

## Tools > Global Options

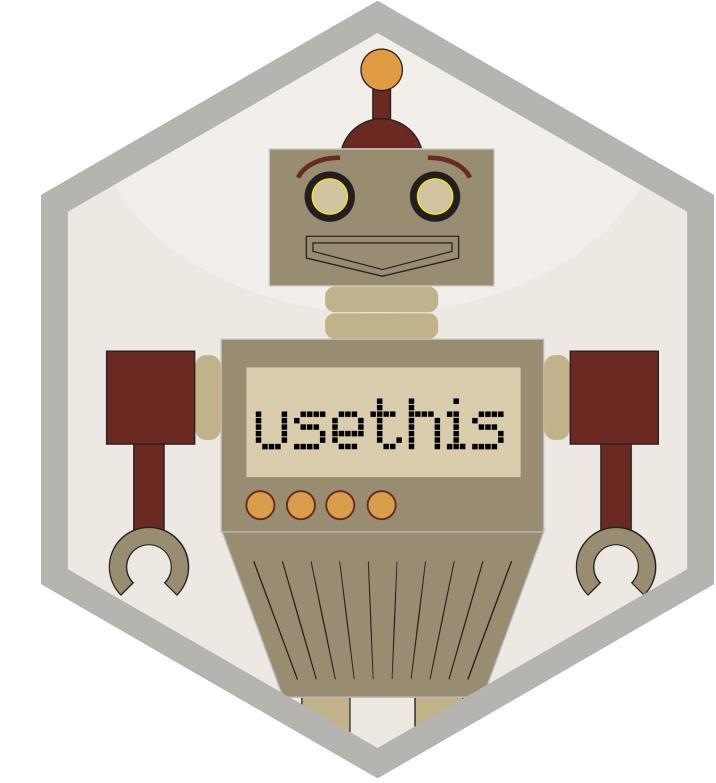


# Tools



- R >= 4.3.0
- R Studio® (<https://posit.co/download/rstudio-desktop/>)
- Packages:

```
install.packages(  
  c("devtools", "roxygen2", "testthat", "knitr", "pkgdown")  
)
```



# Create a package



# Load devtools



```
library(devtools)
#> Loading required package: usethis
```

```
packageVersion("devtools")
#> [1] '2.4.5'
```

- Update if necessary!
- Provides a suite of functions to aid package development
- Loads **usethis**, the source of most functions we will be using

# create\_package()



```
create_package("~/Desktop/mypackage")
```

```
└── .Rbuildignore  
└── .Rproj.user  
└── .gitignore  
└── DESCRIPTION  
└── NAMESPACE  
└── R  
└── mypackage.Rproj
```

- Creates directory
- Sets up basic package skeleton
- Opens a new RStudio project
- Activates "build" pane in RStudio

# create\_package()



```
create_package("~/Desktop/mypackage")
#> ✓ Creating '/Users/jane/Desktop/mypackage/'
#> ✓ Setting active project to '/Users/jane/Desktop/mypackage'
#> ✓ Creating 'R/'
#> ✓ Writing 'DESCRIPTION'
#> Package: mypackage
#> 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
#> Roxygen: list(markdown = TRUE)
#> RoxygenNote: 7.2.3
#> ✓ Writing 'NAMESPACE'
#> ✓ Writing 'mypackage.Rproj'
#> ✓ Adding '^mypackage\\\\.Rproj$' to '.Rbuildignore'
#> ✓ Adding '.Rproj.user' to '.gitignore'
#> ✓ Adding '^\\\\.Rproj\\\\.user$' to '.Rbuildignore'
#> ✓ Setting active project to '<no active project>'
```

➡ Your Turn

# use\_git()

- `use_git_config(  
 user.name = "Jane Doe",  
 user.email = "jane@example.org")`
- `use_git()`
- Turns package directory into a git repository
- Commits your files (with a prompt)
- Restarts RStudio (with a prompt)
  - Activates "git" pane in RStudio

```
use_git()
```

```
#> ✓ Setting active project to  
#>   '/Users/Jane/rrr/mypackage'  
#> ✓ Adding '.Rhistory', '.Rdata',  
#>   '.httr-oauth', '.DS_Store',  
#>   '.quarto' to '.gitignore'  
#> There are 5 uncommitted files:  
#> * '.gitignore'  
#> * '.Rbuildignore'  
#> * 'DESCRIPTION'  
#> * 'metrify.Rproj'  
#> * 'NAMESPACE'  
#> Is it ok to commit?  
#> 1: Abort  
#> 2: No  
#> 3: Yes
```

→ Your Turn

# devtools::use\_devtools()

**Automatically load devtools when R starts**

- Opens .Rprofile file
- Copies code to your clipboard
- Paste into .Rprofile
- Restart R

```
if (interactive()) {  
  # Load package dev packages:  
  suppressMessages(require("devtools"))  
}
```

⌨️ Ctrl+Shift+F10 (Windows & Linux)  
⌨️ Cmd+Shift+Ø (macOS)

➡️ Your Turn

# use\_r()

## Write your first function

- R code goes in **R/**
- Name the file after the function it defines

```
use_r("my-fun")
```

```
#> ✓ Setting active project to '/Users/jane/rrr/mypackage'  
#> • Edit 'R/my-fun.R'
```

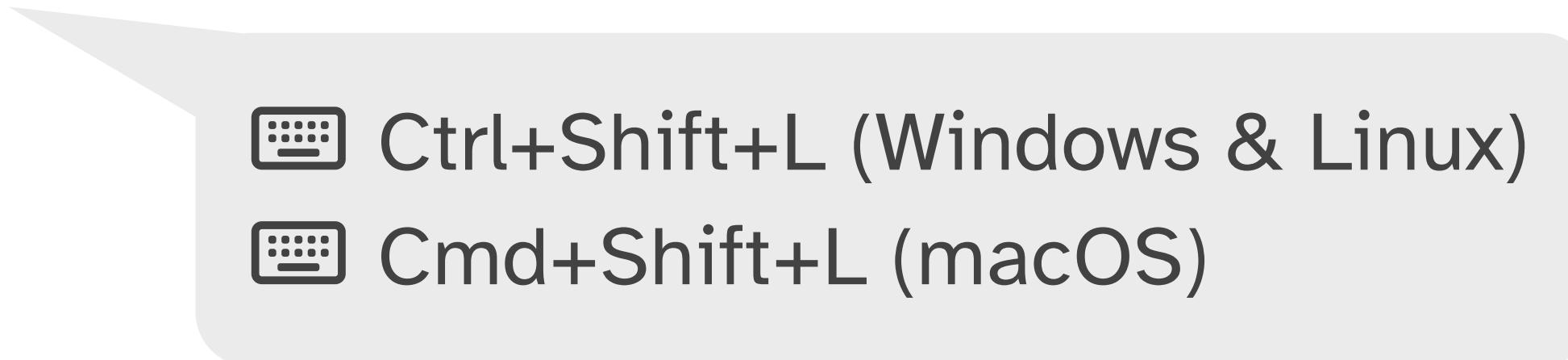
- Put the definition of your function (and only the defin

→ Your Turn

# Test your function in the new package

But how?

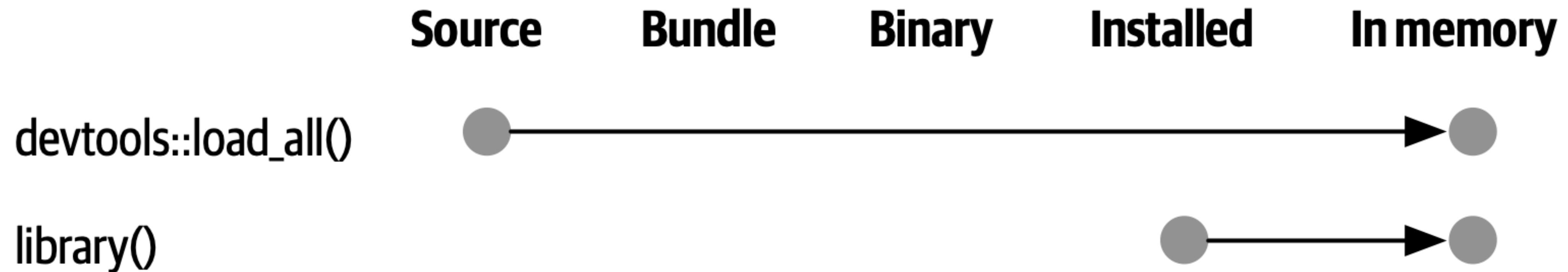
- ~~source("R/my-fun.R")~~
- ~~Send function to console using RStudio (Ctrl/CMD+Return)~~
- ~~devtools::load\_all()~~



- Ctrl+Shift+L (Windows & Linux)
- Cmd+Shift+L (macOS)

# load\_all()

**== install.packages() + library()**

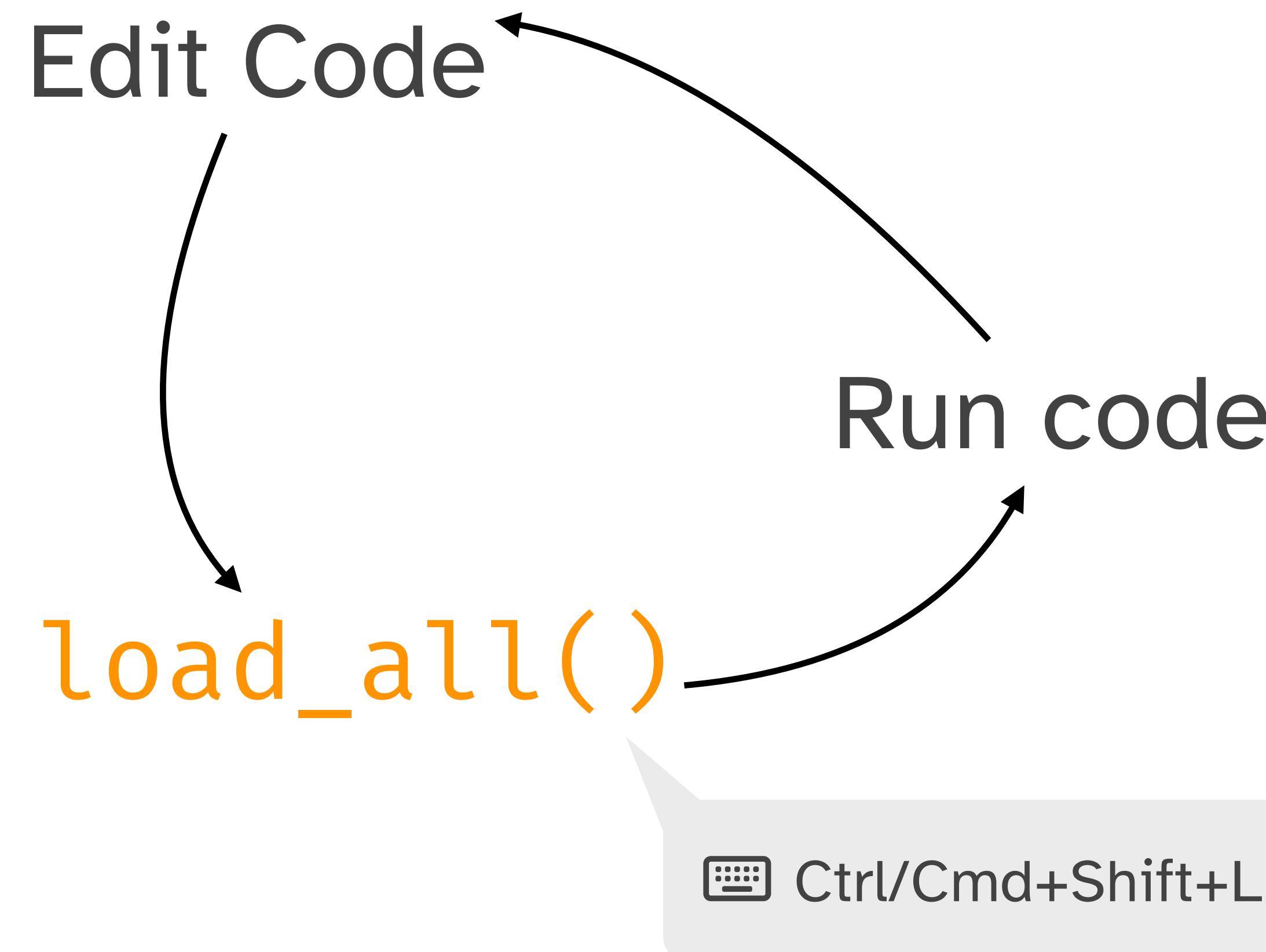


- Simulates building, installing, and attaching your package
- Makes all of the functions from your package immediately available to use
- Allows fast iteration of editing and test-driving your functions
- Good reflection of how users will interact with your package

→ Your Turn

\*With `load_all()`, unexported functions are also made available.

# Workflow



Try it out, and commit your  
changes



# R CMD check

## 3 types of messages

- **ERRORs:** Severe problems - always fix.
- **WARNINGs:** Problems that you should fix, and must fix if you're planning to submit to CRAN.
- **NOTEs:** Mild problems or, in a few cases, just an observation.
  - When submitting to CRAN, try to eliminate all NOTEs.

# check()

## Run R CMD check from within R

```
check()
```

```
#> — R CMD check results ——————  
#> Duration: 3.1s  
#>  
#> ✓ checking DESCRIPTION meta-information ... WARNING  
#> Invalid license file pointers: LICENSE  
#>  
#> 0 errors ✓ | 1 warning ✘ | 0 notes ✓
```

- **check()** early and often
- Reduce future pain by catching problems early\*

→ Your Turn

# Licenses

## `use_*_license()`

- Permissive:
  - **MIT**: simple and permissive.
  - **Apache 2.0**: MIT + provides patent protection.
- Copyleft:
  - Requires sharing of improvements.
  - **GPL (v2 or v3)**
  - **AGPL, LGPL** (v2.1 or v3)
- Creative commons licenses:
  - Appropriate for data packages.
  - **CC0**: dedicated to public domain.
  - **CC-BY**: Free to share and adapt, must give appropriate credit.

# use\_mit\_license()

- ✓ Adding 'MIT + file LICENSE' to License
- ✓ Writing 'LICENSE'
- ✓ Writing 'LICENSE.md'
- ✓ Adding '^LICENSE\\\\.md\$' to '.Rbuildignore'

➡ Your Turn

# The DESCRIPTION file

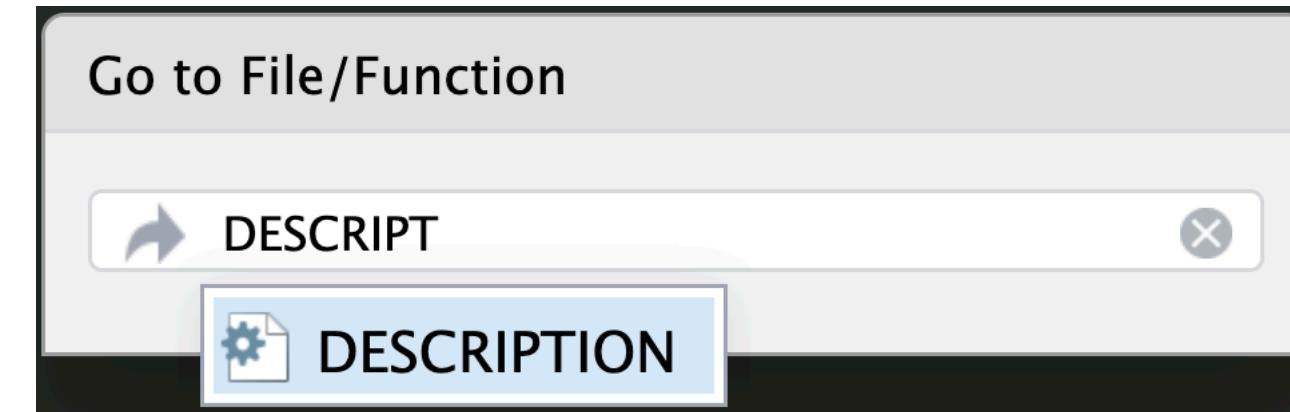
## Package metadata

- Make yourself the author

- Name & Email
- Role
- ORCID (optional)

- Write descriptive

- Title:
- Description:



⌨️ Ctrl+.

start typing DESCRIPTION

```
Package: mypackage
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R: person(
  "First", "Last", ,
  "first.last@example.com",
  role = c("aut", "cre"),
  comment = c(ORCID = "YOUR-ORCID-ID"))
)
Description: What the packa
License: `use_mit_license()
  friends to pick a licenc
Encoding: UTF-8
Roxygen: list(markdown = T
RoxygenNote: 7.2.3
```

➡ Your Turn

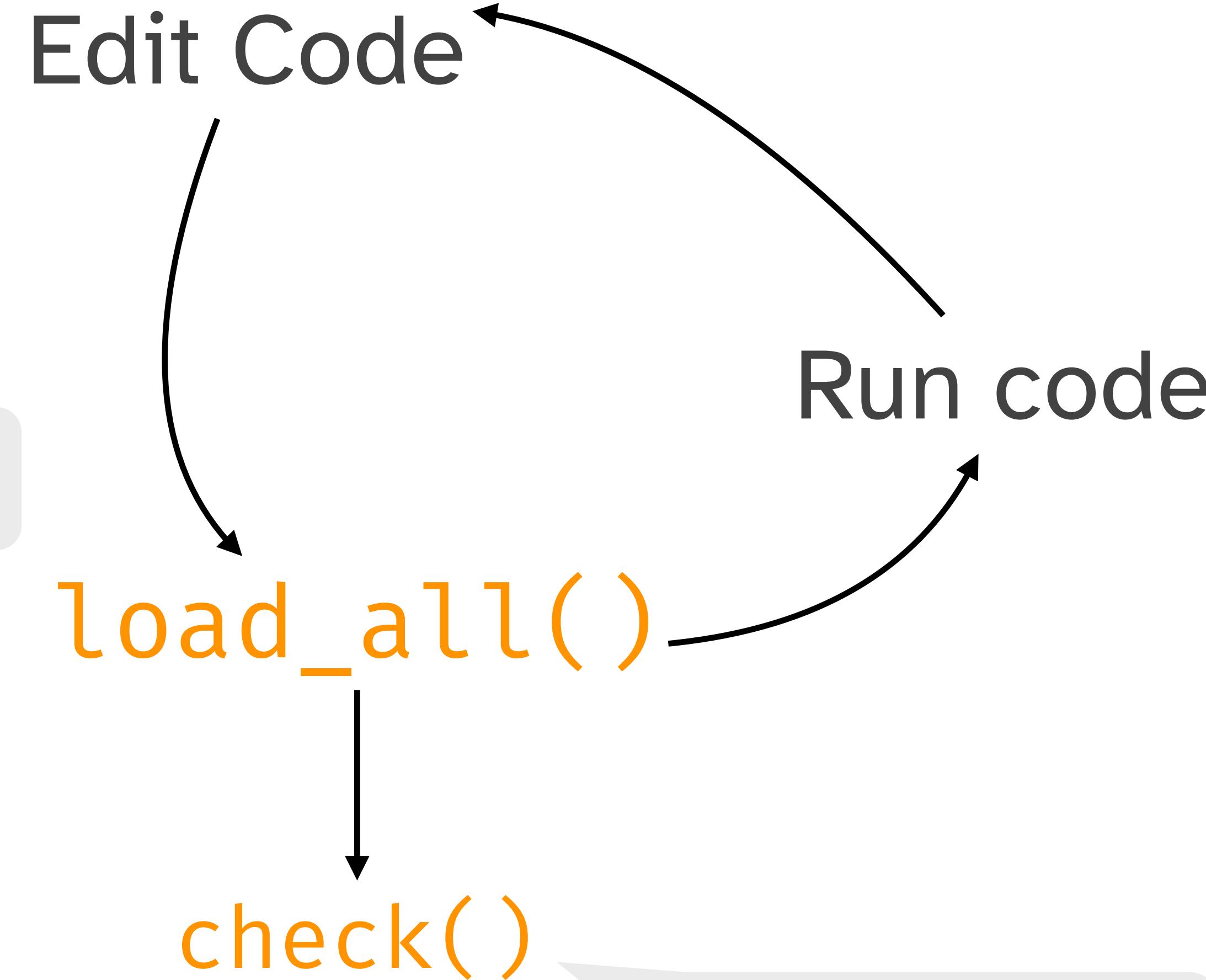
# The DESCRIPTION file

## Package metadata

- Take a look at the DESCRIPTION for ggplot2.
  - CRAN Package page
  - DESCRIPTION on GitHub
    - Note other Author roles:
      - ‘cph’ (copyright holder, often your employer)
      - ‘fnd’ (funder)

# Workflow

**Code + check**



Ctrl/Cmd+Shift+L

Ctrl/Cmd+Shift+E

# check() again

```
check()
```

```
#> == Documenting ==
...
#> == Building ==
...
#> == Checking ==
...
#> — R CMD check results —
#> Duration: 3.1s
#>
#> 0 errors ✅ | 0 warnings ✅ | 0 notes ✅
```





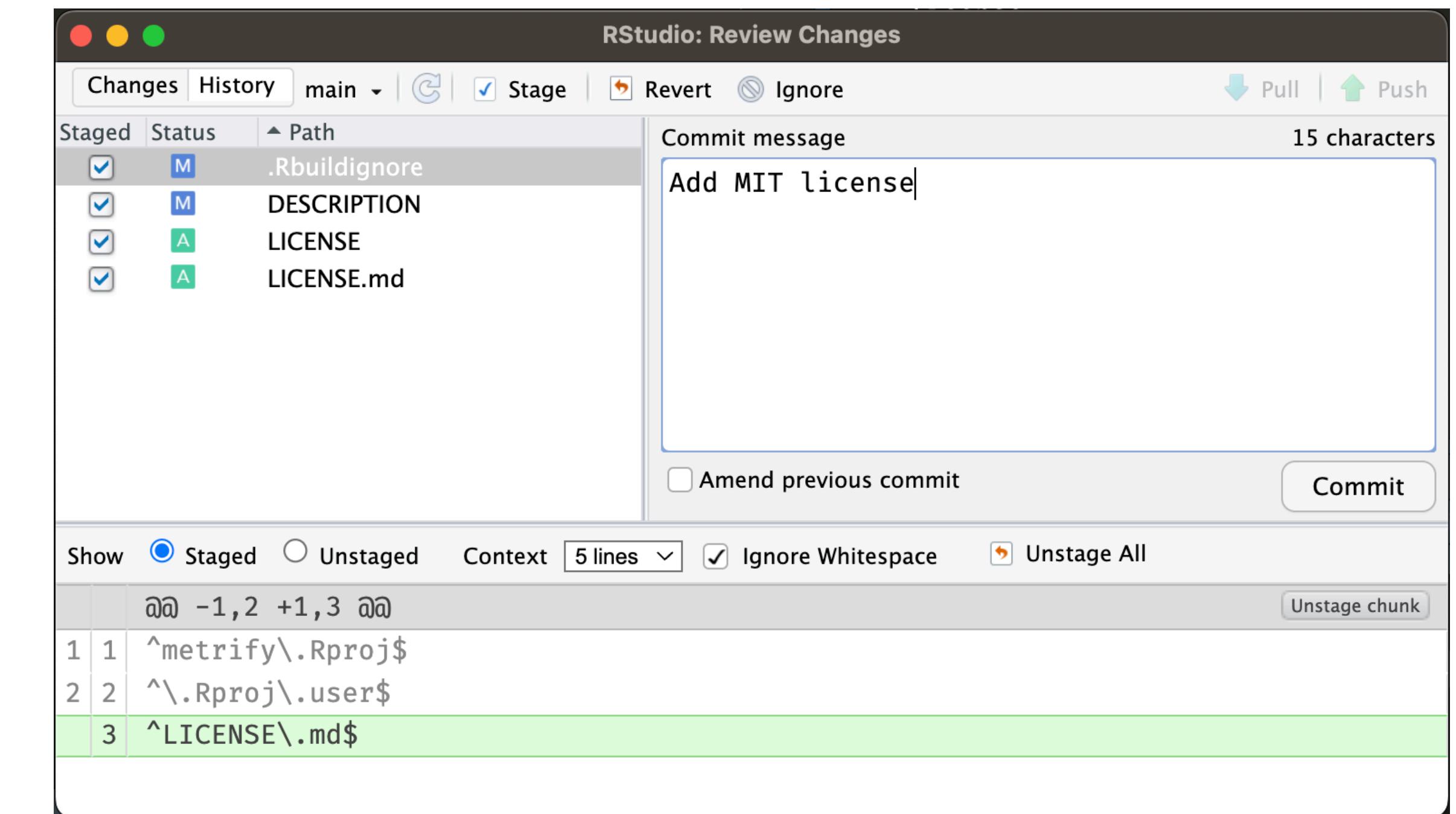
**TED  
LASSO**

# Commit changes to git



```
$ git add DESCRIPTION \
LICENSE \
LICENSE.md \
.Rbuildignore
```

```
$ git commit -m "Add MIT license"
```



# use\_github()

**Put your package code on GitHub**

- Prerequisites:
  - GitHub account
  - `create_github_token()` - follow instructions
  - `gitcreds::gitcreds_set()` - paste PAT
  - `git_sitrep()` - verify

`use_github()` - push content to new repo

 Your Turn

# Avoid some pain of package setup: `edit_r_profile()`

**And set default `DESCRIPTION` values**

```
# Set usethis options:  
options(  
  usethis.description = list(  
    "Authors@R" = utils::person(  
      "Jane", "Doe",  
      email = "jane@example.com",  
      role = c("aut", "cre"),  
      comment = c(ORCID = "0000-1111-2222-3333"))  
  ))
```

\*<https://usethis.r-lib.org/articles/usethis-setup.html>

# While you're in there...

## Set some other helpful defaults

```
options(  
  warnPartialMatchArgs = TRUE,  
  warnPartialMatchDollar = TRUE,  
  warnPartialMatchAttr = TRUE  
)
```

# Documentation



# Documentation

man/\* .Rd

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/git.R
\name{use_git}
\alias{use_git}
\title{Initialise a git repository}
\usage{
use_git(message = "Initial commit")
}
\arguments{
\item{message}{Character string specifying message to use for first commit.}
}
\description{
}
\code{use_git()}
\code{.gitignore}
}
\examples{
}
\dontrun{
use_git()
}
\seealso{
}
Other git helpers
\code{\link{use_git_config()}}
\code{\link{use_git_hook()}}
\code{\link{use_git_ignore()}}
}
\concept{git helpers}
```

The image shows a scene from the TV show Schitt's Creek. David Rose, played by Eugene Levy, is looking directly at the camera with a confused expression. He has dark hair and is wearing a black t-shirt. In the background, there's a window showing a room with some furniture and a red bookshelf. On the left side of the image, there is a vertical column of R code related to the 'use\_git' function. On the right side, there is a caption that says 'files to it.'

Function documentation

```
> ?use_git
use_git          package:usethis
Initialise a git repository
Description:
  'use_git()' initialises a Git repository and adds important files
  to '.gitignore'. If user consents, it also makes an initial
  commit.
Usage:
  use_git(message = "Initial commit")
Arguments:
  message: Message to use for first commit.
See Also:
  Other git helpers: 'use_git_config()', 'use_git_hook()',
  'use_git_ignore()'
Examples:
  ## Not run:
  use_git()
  ## End(Not run)
```

The image shows a comparison between two types of documentation. On the left, under the heading 'man/\* .Rd', there is a screenshot of a computer screen displaying R code for the 'use\_git' function. The code includes a warning about not editing it by hand and provides details about its usage, arguments, and description. On the right, under the heading 'Function documentation', there is a screenshot of the same 'use\_git' function as it appears in the R documentation. This version is formatted as a help page with sections for 'Description', 'Usage', 'Arguments', 'See Also', and 'Examples'. An orange arrow points from the left side towards the right side, indicating a transformation or relationship between the two types of documentation.

# roxygen2



- RStudio: *Code > Insert Roxygen Skeleton*
- Special comments (#') above function definition in `R/*.R`
  - Title
  - Description
  - Parameters (`@param`)
  - Return value (`@return`)
  - Export tag (`@export`)
  - Example usage (`@examples`)
  - ...
- Markdown-like syntax
- Keep documentation with code!

Cmd/Ctrl+Alt+Shift+R

```
#' Title  
#' A longer description of what the function  
#' is used for  
#'  
#' @param x  
#' @param y  
#'  
#' @return  
#' @export  
#'  
#' @examples  
add <- function(x, y) {  
  x+y  
}
```

# document()

R/use-git.R

```
#' Initialise a git repository
#'
#' `use_git()` initialises a Git
#' repository and adds important
#' files to `.gitignore`. If user
#' consents, it also makes an
#' initial commit.
#'
#' @param message Message to use
#'   for first commit.
#' @export
#' @examples
#' \dontrun{
#'   use_git()
#' }
use_git <- function(message = "Initial
commit") {
  . . .
}
```

Cmd/Ctrl  
+Shift+D

document()  
→

man/\*.Rd

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/git.R
\name{use_git}
\alias{use_git}
\title{Initialise a git repository}
\usage{
use_git(message = "Initial commit")
}
\arguments{
\item{message}{Message to use for first commit.}
}
\description{
\code{use_git()} initialises a Git repository and adds
important files to \code{.gitignore}. If user consents,
it also makes an initial commit.
}
\examples{
\dontrun{
use_git()
}
}
```

# Create roxygen comments

- Go to function definition
- Cursor in function definition
- Insert roxygen skeleton
- Complete the roxygen fields
- `document()`
- `?myfunction`
- 🎉

⌨️ Ctrl+.

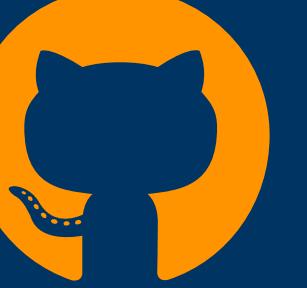
(Start typing function name...)

⌨️ Cmd/Ctrl+Alt+Shift+R

⌨️ Cmd/Ctrl+Shift+D



Your Turn

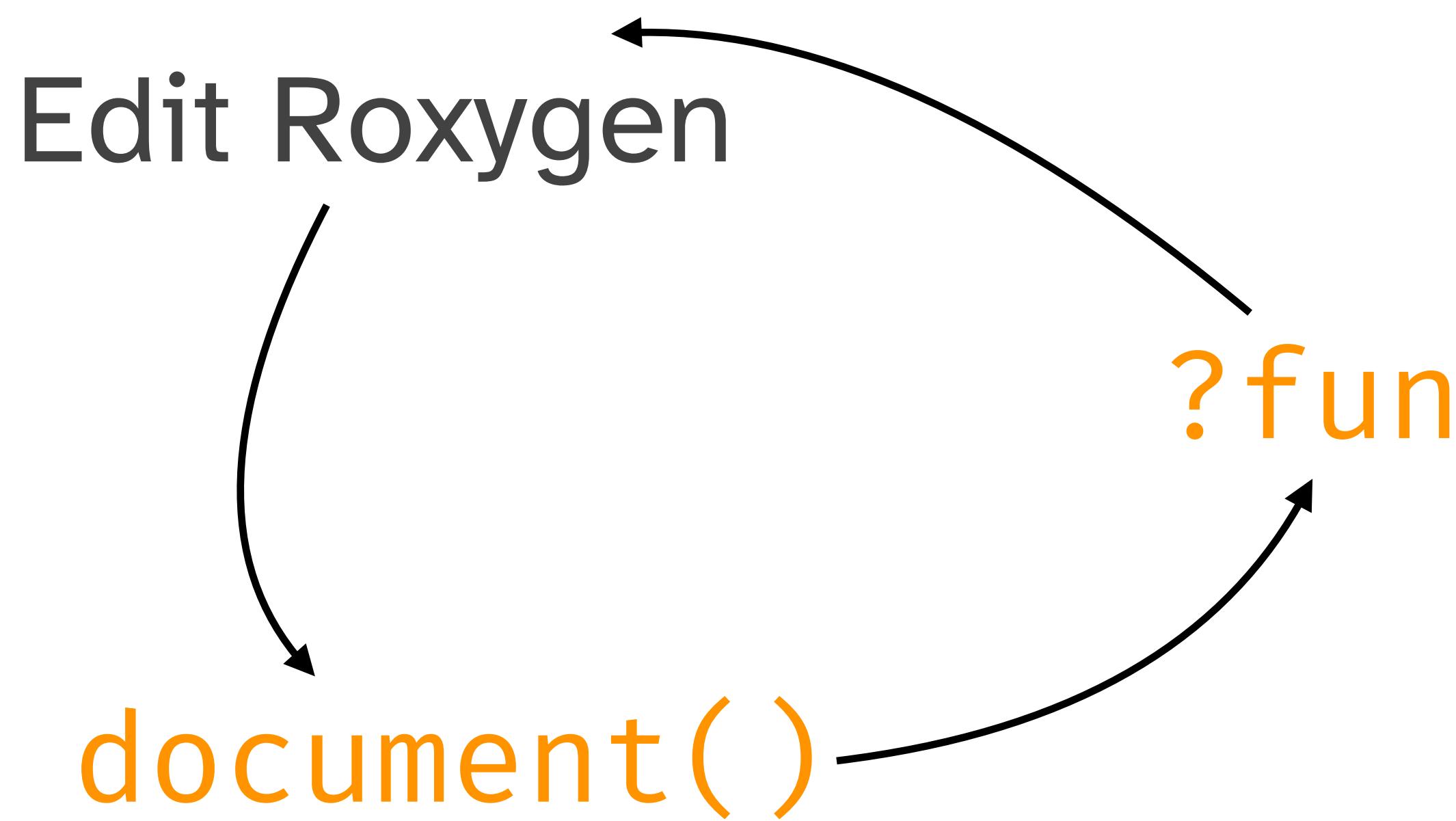
check()   
Commit your changes   
Push to Github 

# NAMESPACE

## An introduction

- Lists R objects that are:
  - **Exported** from your package to be used by package users
    - `export()`, `S3method()`, ...
  - **Imported** from another package to be used internally by your package
    - `import()`, `importFrom()`, ...
- `document()` updates the NAMESPACE file with directives from Roxygen comments in your R code.

# Documentation workflow



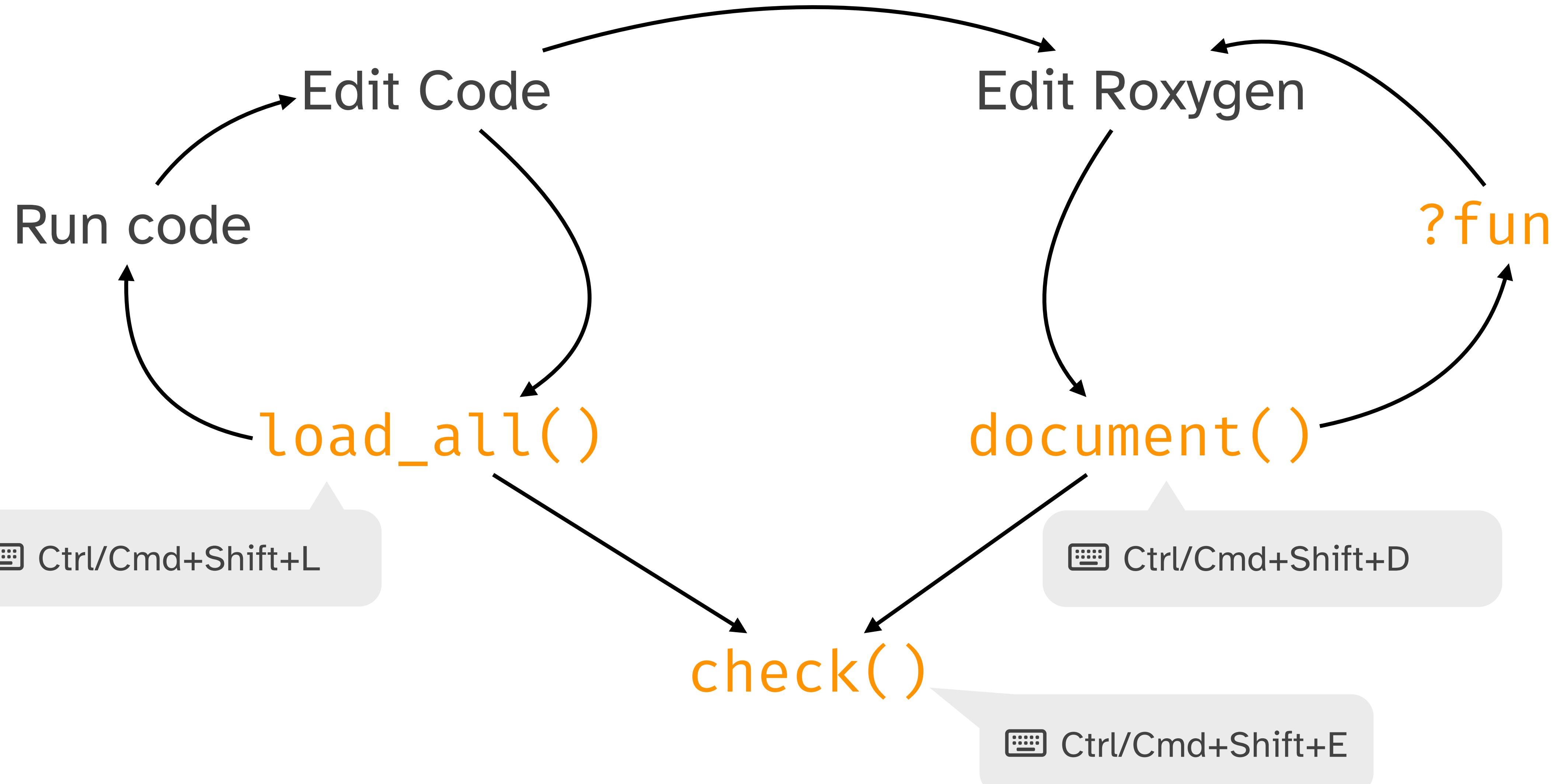
Ctrl+Shift+D (Windows & Linux)



Cmd+Shift+D (macOS)

# Workflow

**Code + documentation + check**



# Package-level documentation

## use\_package\_doc()

```
use_package_doc()  
  
#> ✓ Writing 'R/mypackage-package.R'  
#> • Modify 'R/mypackage-package.R'  
  
document()
```

- Package-level help available via `?mypackage`
- Creates relevant `.Rd` file from `DESCRIPTION`
- A good place for roxygen dependency directives

 Your Turn

# check() again

```
check()
```

```
#> == Documenting ==
...
#> == Building ==
...
#> == Checking ==
...
#> — R CMD check results —
#> Duration: 3.1s
#>
#> 0 errors ✅ | 0 warnings ✅ | 0 notes ✅
```

# install()

## Install package to your library

- R CMD INSTALL

⌨️ Ctrl+Shift+B (Windows & Linux)  
⌨️ Cmd+Shift+B (macOS)

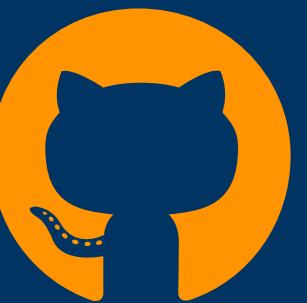
- Restart R

⌨️ Ctrl+Shift+F10 (Windows & Linux)  
⌨️ Cmd+Shift+Ø (macOS)

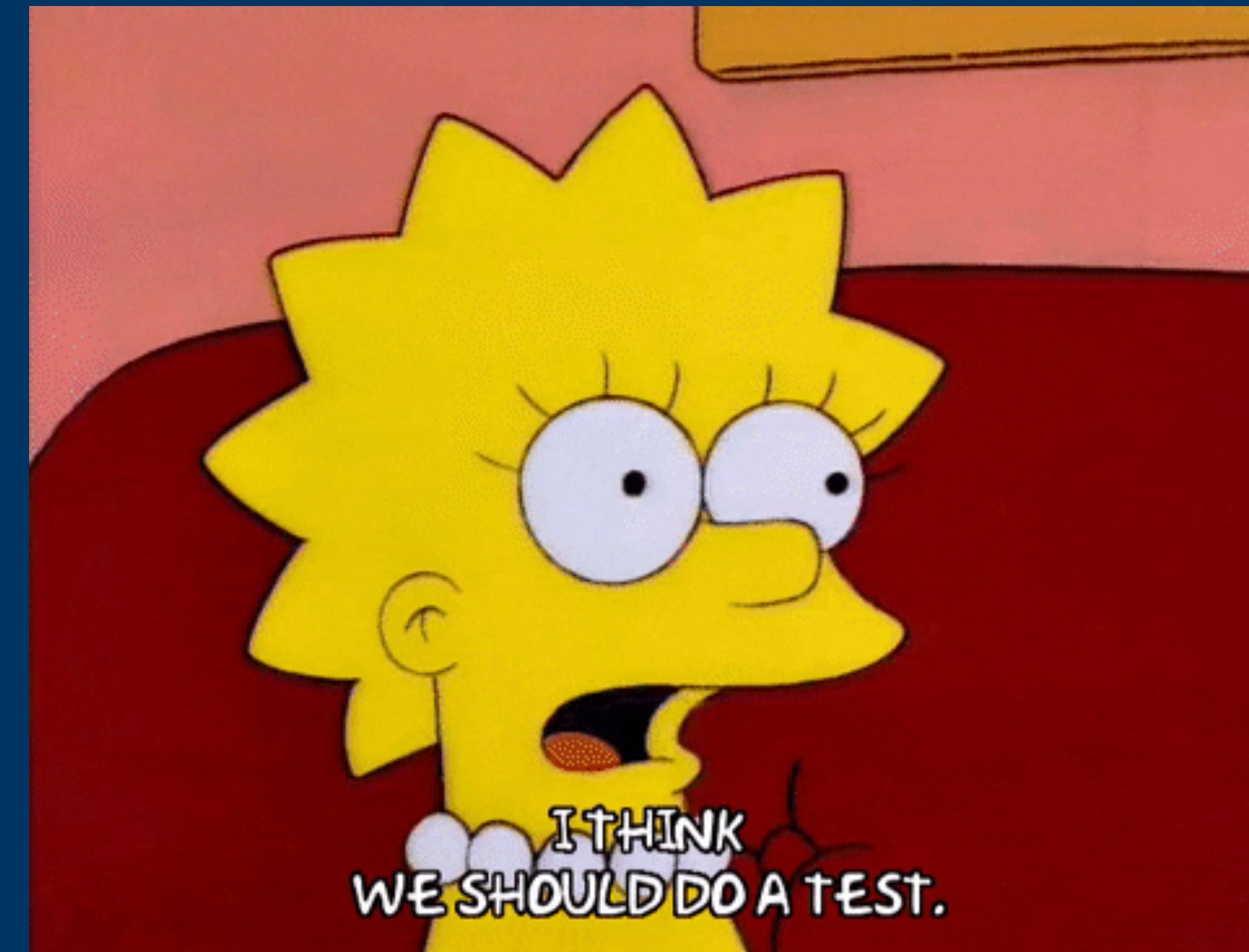
- Load package with `library()` like any other package

➡ Your Turn

Commit your changes  
Push to Github

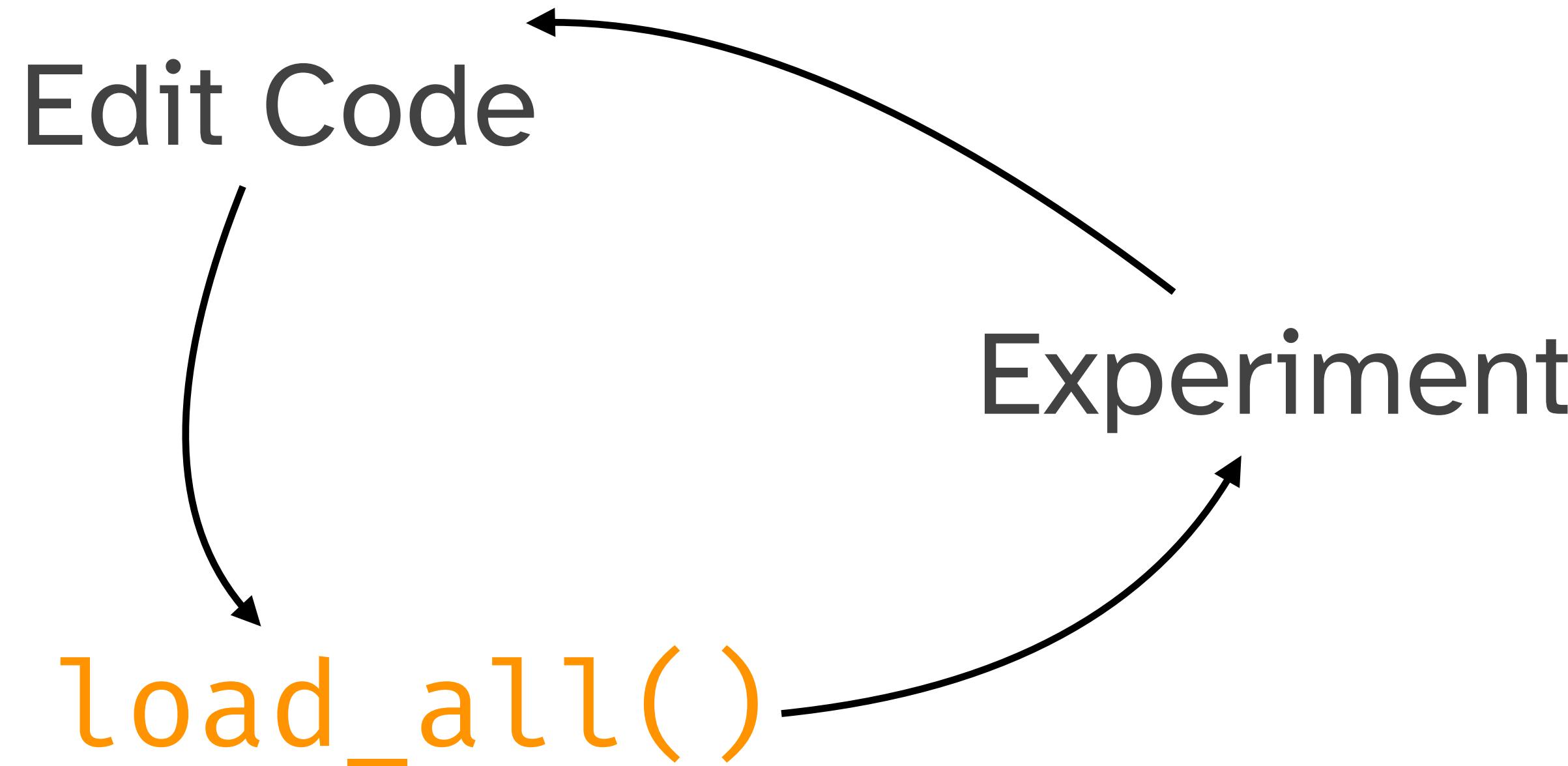


# Testing



# Testing

## Current workflow



Ctrl+Shift+L (Windows & Linux)



Cmd+Shift+L (macOS)

# Automated Testing

## Benefits

- Fewer bugs
- Better code structure
- Call to action when fixing bugs
- Robust (future-proof) code



# use\_testthat()

**Set up formal testing of your package\***

```
use_testthat()
```

```
#> ✓ Adding 'testthat' to Suggests field in DESCRIPTION
#> ✓ Adding '3' to Config/testthat.edition
#> ✓ Creating 'tests/testthat/'
#> ✓ Writing 'tests/testthat.R'
#> • Call `use_test()` to initialize a basic test file and
open it for editing.
```

\*Sorry, you still have to write the tests

# use\_test()

```
use_test('my-fun.R')*
```

```
#> ✓ Writing 'tests/testthat/test-my-fun.R'  
#> • Edit 'tests/testthat/test-my-fun.R'
```

\*Omit file name when '**R/my-fun.R**' is active file

# File structure

```
libminer
├── DESCRIPTION
├── LICENSE
├── LICENSE.md
├── NAMESPACE
├── R
│   └── lib_summary.R
│   └── libminer-package.R
└── libminer.Rproj
man
└── lib_summary.Rd
└── libminer-package.Rd
tests
└── testthat
    └── test-lib_summary.R
    └── testthat.R
```

# Test structure

```
testthat("description of what you're testing", {  
  expect_equal([function output], [expected output])  
})
```

- **File:** one or more related tests
- **Test:** `test_that("...")`
  - Tests a unit of functionality (hence unit tests)
  - Contains one or more expectations
- **Expectation:** `expect_``that`(...)
  - Tests a specific computation and compares it to an expected value

# test()

- Runs all tests in your test suite

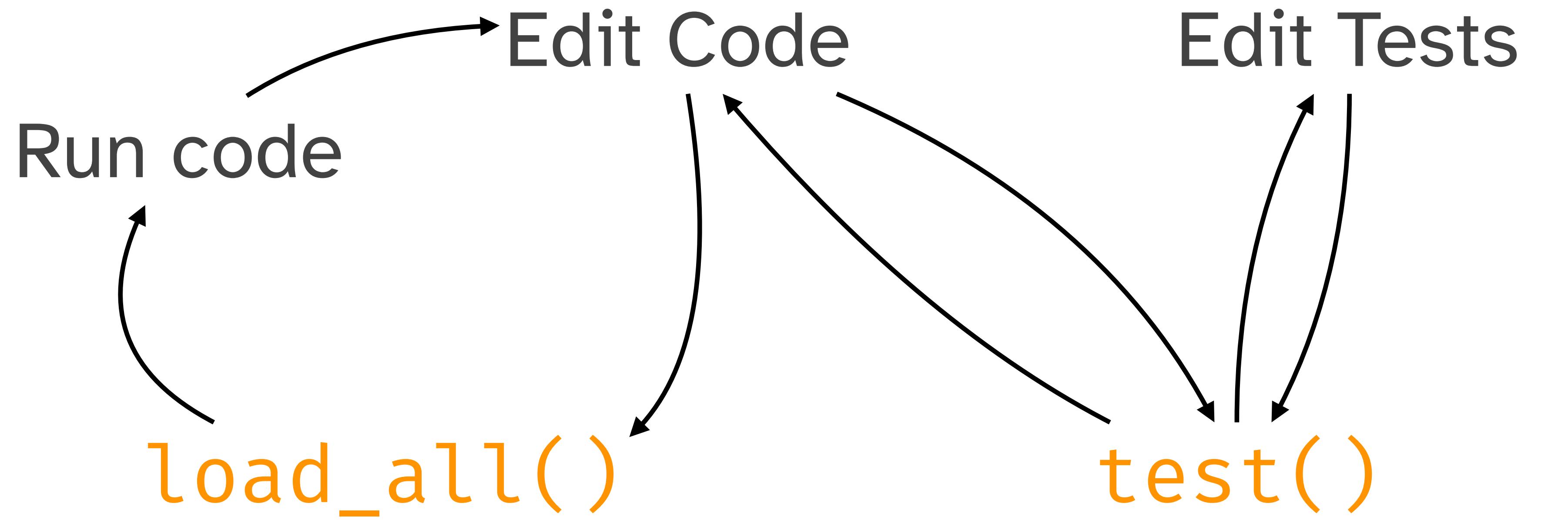
```
test()  
#> i Testing  
#> ✓ | F W S  OK | Context  
#>  
#> :: | 0 |  
#> ✓ | 1 |  
#>  
#> == Results ==  
#> [ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
```

⌨️ Ctrl+Shift+T (Windows & Linux)  
⌨️ Cmd+Shift+T (macOS)

➡ Your Turn

# Updated workflow

## Code + testing

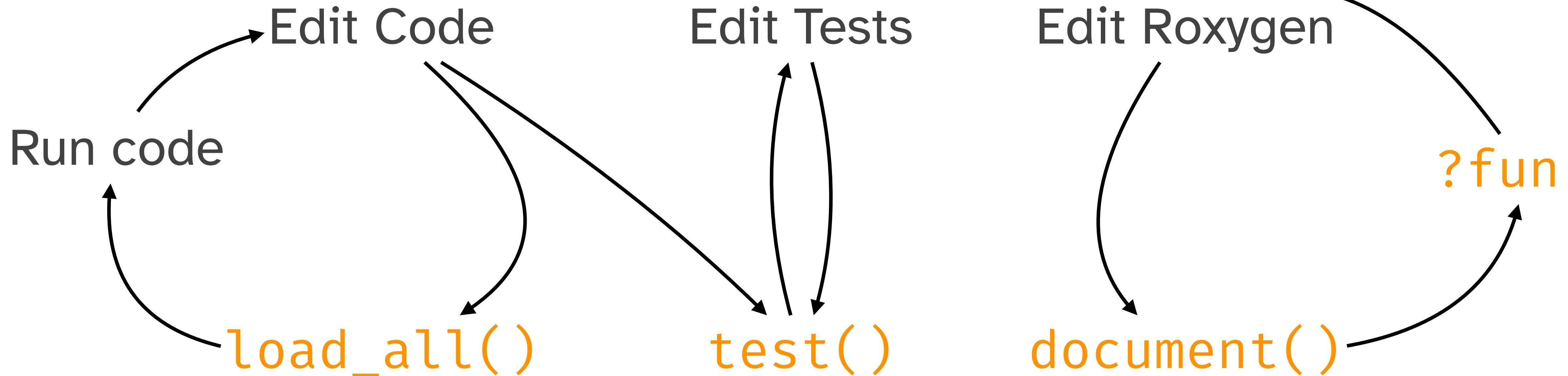


Ctrl/Cmd+Shift+L

Ctrl/Cmd+Shift+T

# Workflow

**Code + testing + documentation**



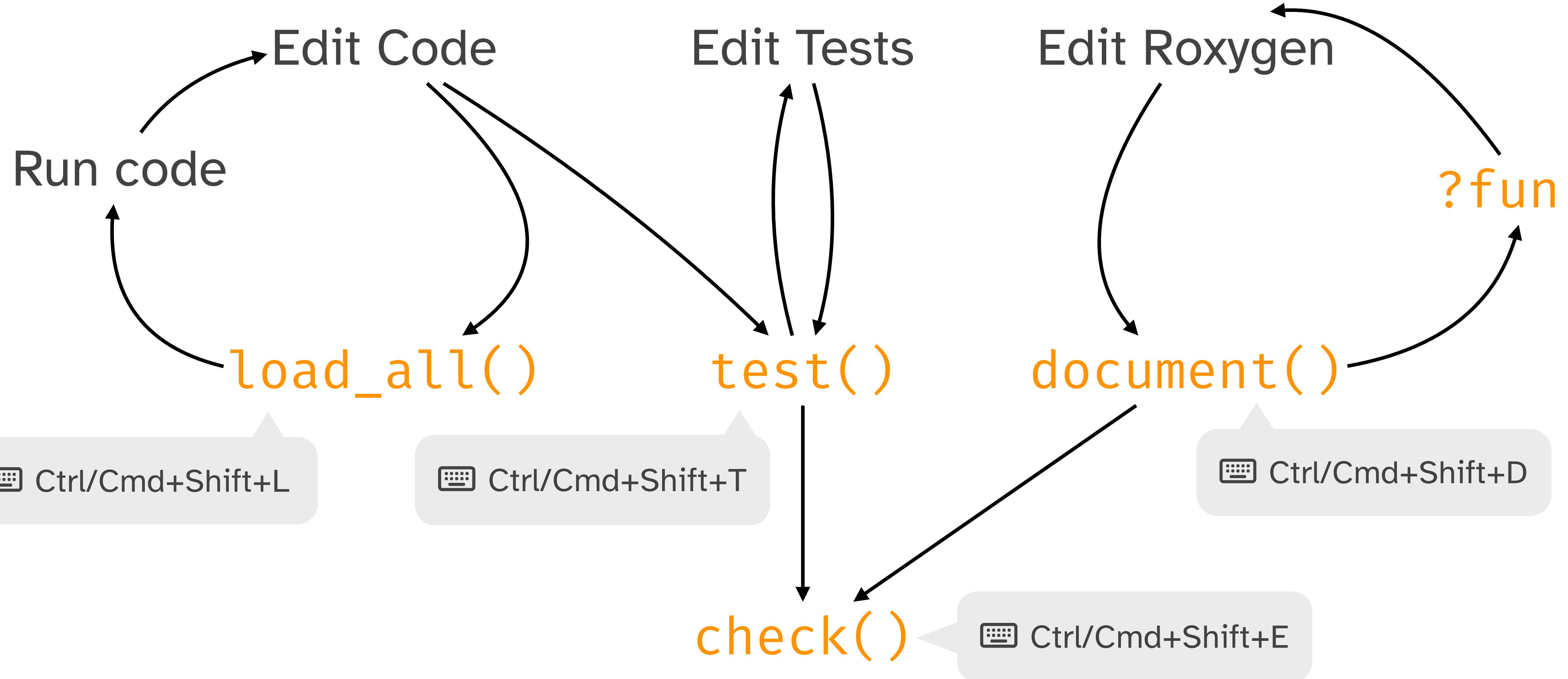
⌨️ `Ctrl/Cmd+Shift+L`

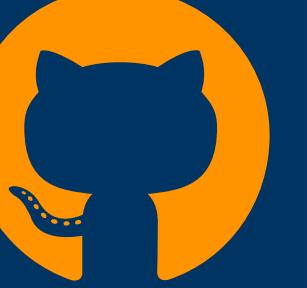
⌨️ `Ctrl/Cmd+Shift+T`

⌨️ `Ctrl/Cmd+Shift+D`

# Workflow

Code + testing + documentation + check



check()   
Commit your changes   
Push to Github 

# Dependencies



# Use functions from another package inside your package

```
library("fs")  
# use fun...
```

...kage...

# use\_package()

## Add a dependency

- Use functions from another package inside your package
- Dependencies must be declared
  - Even from included packages (`stats::sd()`, `tools::file_ext()` etc.)
- Never call `library(pkg)` in code below R/!

```
use_package("fs")
#> ✓ Adding 'fs' to Imports field in DESCRIPTION
#> • Refer to functions with `fs::fun()`
```

# Listing dependencies in DESCRIPTION

## Three options

- Depends:
  - Ensures the package is installed with your package
  - *Attaches the package when yours is attached*
  - Rarely needed or recommended
- Imports:
  - Ensures the package is installed with your package
  - Most common location for dependencies
- Suggests:
  - Does not ensure installation automatically
  - Packages required for development (running tests, building vignettes, etc).
  - Rarely used functionality (especially if the dependency is difficult to install)

**devtools DESCRIPTION file:**

[https://github.com/r-lib/devtools/blob/main/  
DESCRIPTION](https://github.com/r-lib/devtools/blob/main/DESCRIPTION)

# Imports: DESCRIPTION vs NAMESPACE

## DESCRIPTION

- Lists packages that your package requires
- Ensures required packages are **installed** during package installation
- **Does not** import that package into your package's namespace
- Add via `use_package()` (or manually)

## NAMESPACE

- **Imports** R objects from another package into your package's namespace
- Available to be used internally by your package
- Don't edit manually - use roxygen tags:

```
#' @importFrom pkg fun
#' @importpkg
```

# 3 ways to use functions from another package

## 1 - Call function with namespace qualifier

1. Add package to DESCRIPTION file in Imports
2. Call function like package::fun()

Most common and recommended pattern

DESCRIPTION

Imports:  
purrr

R/my-fun.R

```
#' @export
myfun <- function(x) {
  purrr::map(x, mean)
}
```

NAMESPACE

export(myfun)

document()



# 3 ways to use functions from another package

## 2 - Import just the functions you want to use via `@importFrom` tag:

1. Add package to `DESCRIPTION` file in `Imports`
2. Use `@importFrom` roxygen tags
3. Call function like `fun()`

DESCRIPTION

Imports:  
purrr

R/my-fun.R

```
#' @importFrom purrr map
#' @export
myfun <- function(x) {
  map(x, mean)
}
```

NAMESPACE

```
importFrom(purrr, map)
export(myfun)
```

document()



# 3 ways to use functions from another package

## 3 - Import the entire package via `@import roxygen` tag:

1. Add package to `DESCRIPTION` file in `Imports`
2. Use `@import roxygen` tag
3. Call functions like `fun()`

DESCRIPTION

Imports:  
purrr

R/my-fun.R

```
#' @import purrr
#' @export
myfun <- function(x) {
  y <- map(x, mean)
  reduce(y, `+`)
}
```

NAMESPACE

import(purrr)
export(myfun)

document()

# 3 ways to use functions from another package

1. `package::fun()`

2. Import just the functions you want to use via `@importFrom` roxygen tag:

```
#' @importFrom pkg fun1 fun2
```

Adds to `NAMESPACE`:

```
importFrom(pkg, fun1)
importFrom(pkg, fun2)
```

\*Shortcut: `usethis::use_import_from("pkg", "function")`

3. Import the entire package with `@import`:

```
#' @import pkg
```

Adds to `NAMESPACE`:

```
import(pkg)
```



# Use your new dependency

**Write a function using a function from the dependent package**

- `use_package("fs")`
- Write/edit function using dependency: `pkg::fn()`
- Edit roxygen comments
- `document()`
  - Writes `man/*.Rd` files & regenerates `NAMESPACE`
- Update tests

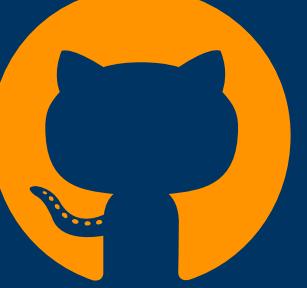
 Your Turn

# Let's add one more

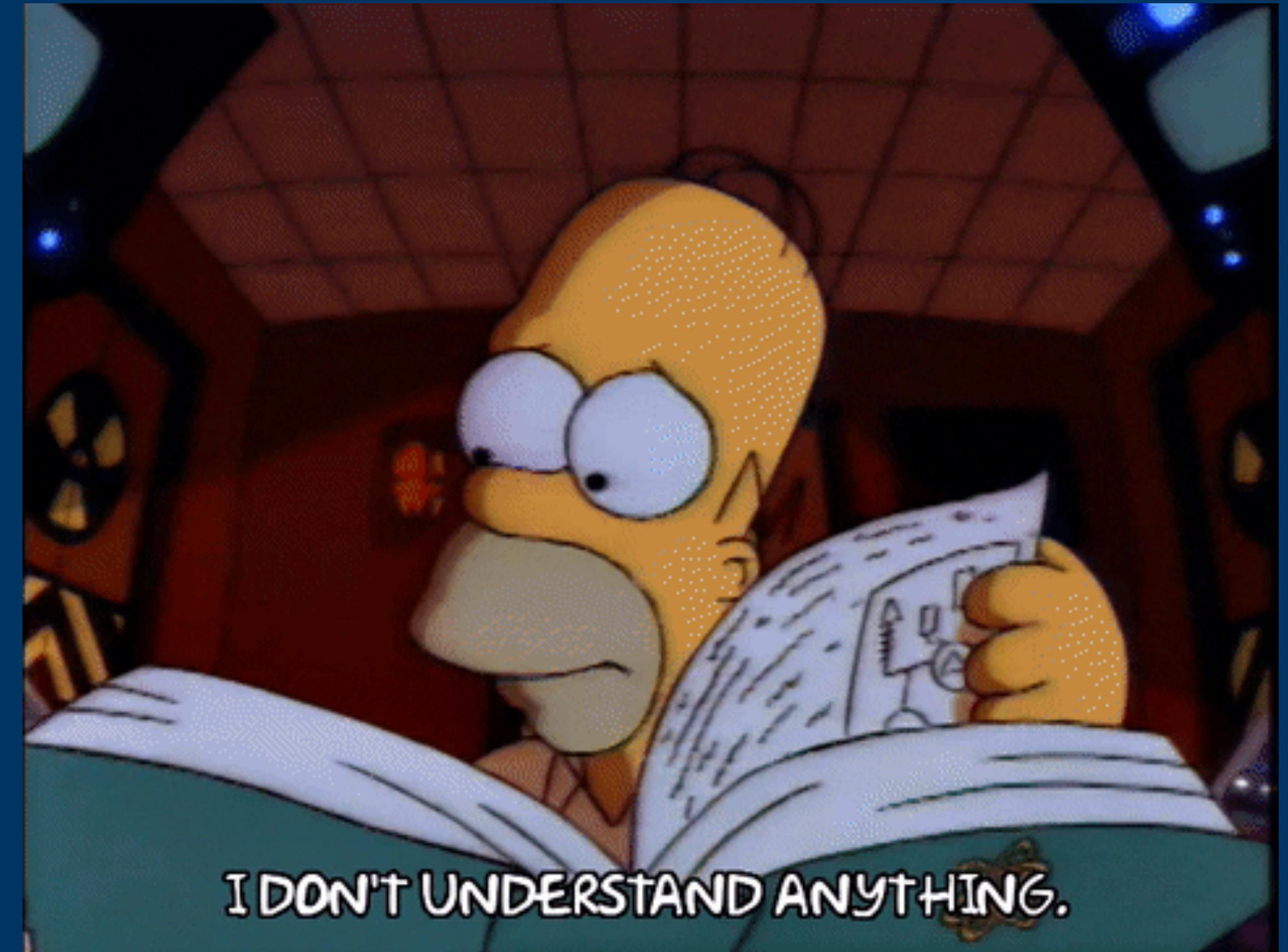
```
use_import_from("pkg", "function")
```

- See:
  - DESCRIPTION
  - R/mypackage-package.R (remember use\_package\_doc()?)
  - NAMESPACE
- Write/edit function using dependency: fn()
- \*test()

→ Your Turn

check()   
Commit your changes   
Push to Github 

# README



I DON'T UNDERSTAND ANYTHING.

# use\_readme\_rmd()

**Generates README.md, your package's home page on GitHub**

- The purpose of the package
- Installation instructions
- Example usage
- Contributing guide

```
use_readme_rmd()
```

```
#> ✓ Writing 'README.Rmd'  
#> ✓ Adding '^README\\\\.Rmd$' to '.Rbuildignore'  
#> • Update 'README.Rmd' to include installation instructions.  
#> ✓ Writing '.git/hooks/pre-commit'
```

# build\_readme()

**README.Rmd -> README.md**

- Installs package to a temporary directory before rendering
- README.md renders on the front page of your GitHub repo

 Your Turn

# Final check() and install()

You did it!

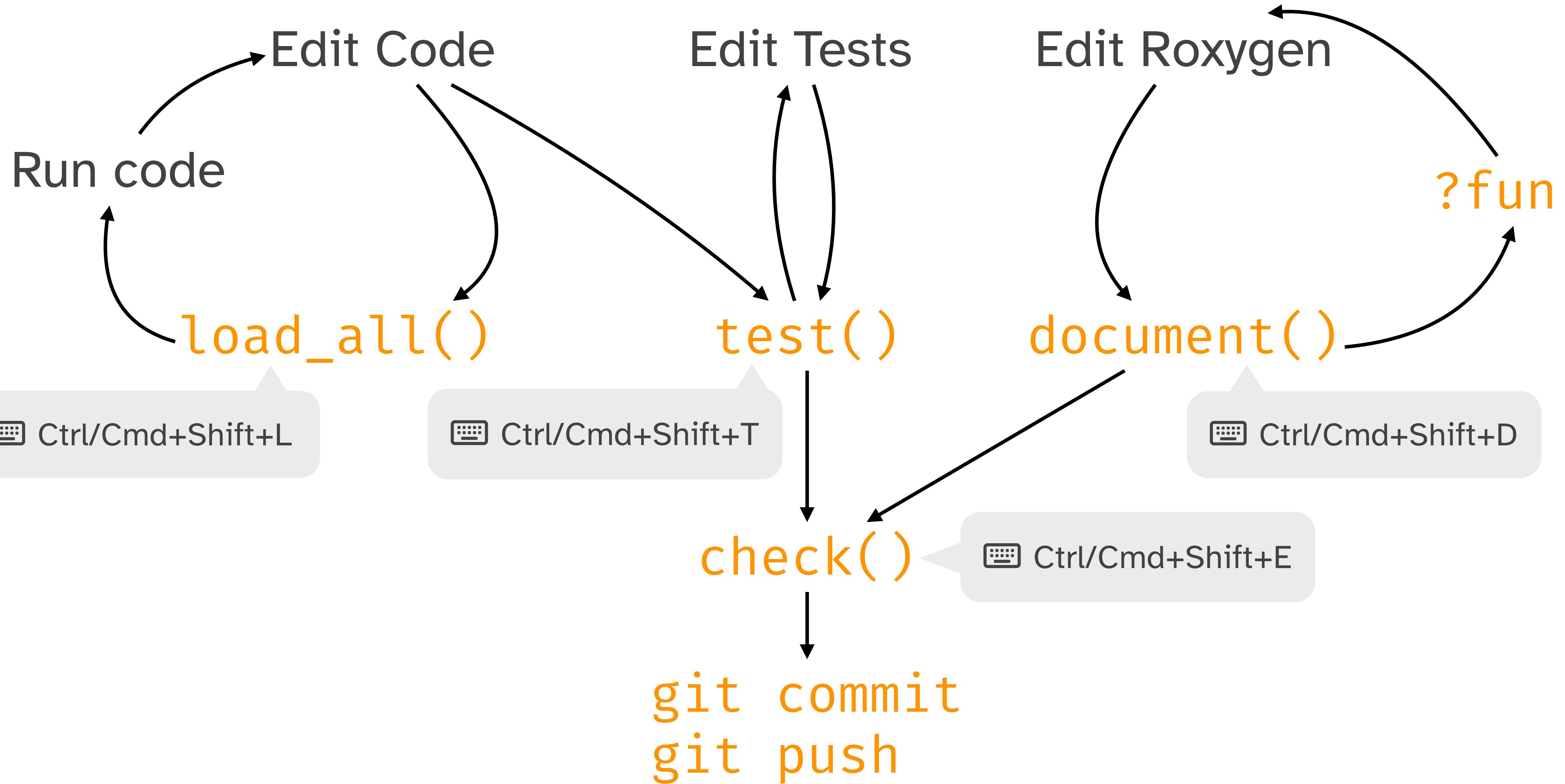
```
check()
```

```
#> — R CMD check results —————  
#> Duration: 3.1s  
#>  
#> 0 errors ✓ | 0 warnings ✓ | 0 notes
```

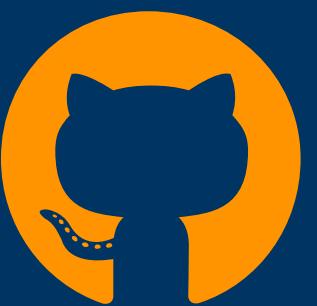
```
install()
```

```
#> — R CMD build —————  
#> checking for file '/Users/jane/rrr/mypackage/DESCRIPTION' ... ✓  
#> preparing 'mypackage':  
#> checking DESCRIPTION meta-information ... ✓  
#> checking for LF line-endings in source and make files and shell  
scripts  
#> checking for empty or unneeded directories  
#> building 'mypackage_0.0.0.9000.tar.gz'  
#> Running /usr/local/bin/R CMD INSTALL \  
#>   /tmp/RtmpK6WnOX/mypackage_0.0.0.9000.tar.gz --install-tests  
#> * installing to library '/Users/jane/Library/R/arm64/4.3/library'  
#> * installing *source* package 'mypackage' ...  
#> ** using staged installation  
#> ** help  
#> *** installing help indices  
#> ** building package indices  
#> ** testing if installed package can be loaded from temporary  
location  
#> ** testing if installed package can be loaded from final location  
#> ** testing if installed package keeps a record of temporary  
installation path  
#> * DONE (mypackage)
```

# Review: Workflow



Commit your changes  
Push to Github



# Review: functions

## Run once

- `create_package()`
- `use_git()`
- `use_github()`
- `use_mit_license()`
- `use_testthat()`
- `use_readme_rmd()`

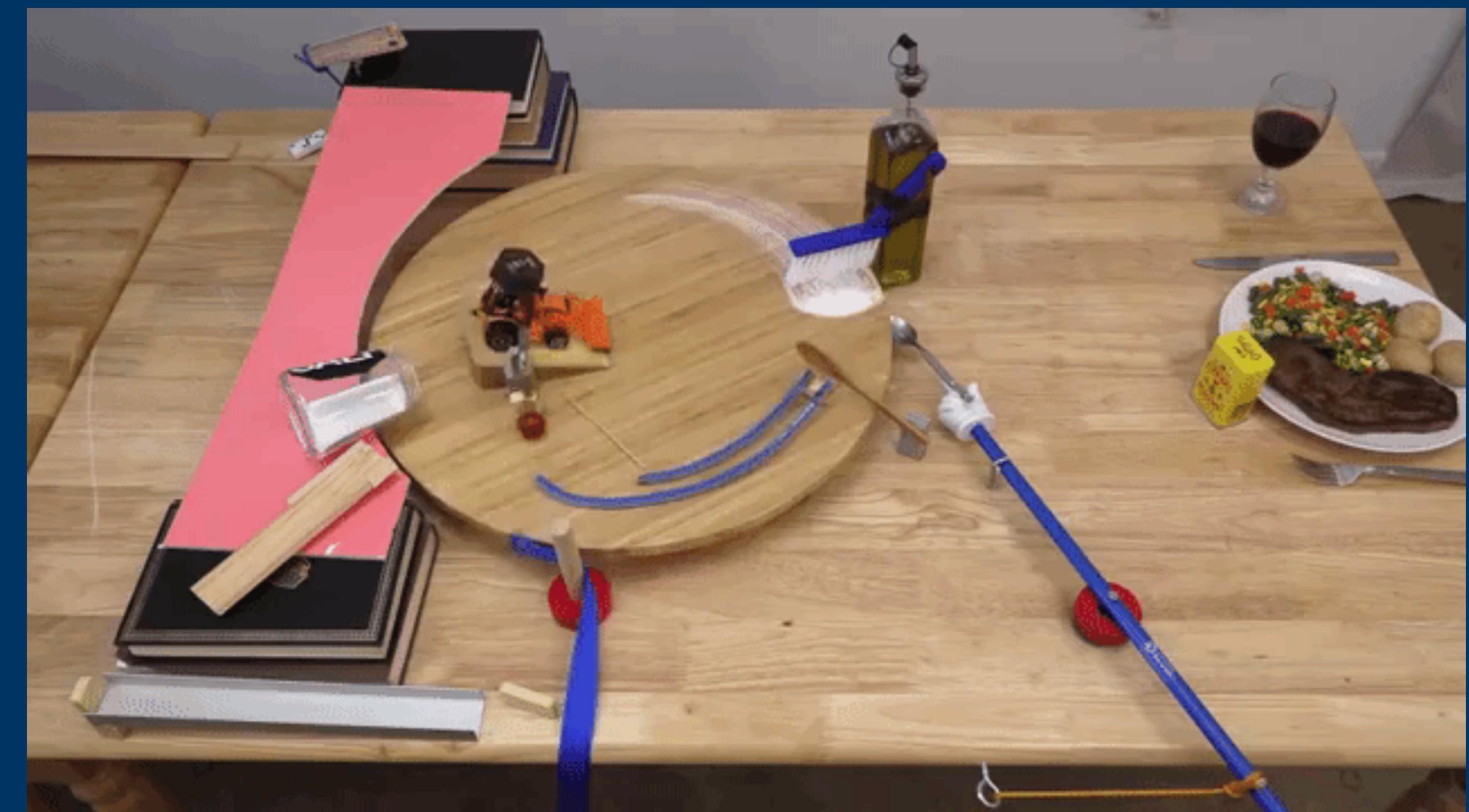
## Run periodically

- `use_r()`
- `use_test()`
- `use_package()`
- `rename_files()`

## Run frequently

- `load_all()`  
⌨️ Ctrl/Cmd+Shift+L
- `document()`  
⌨️ Ctrl/Cmd+Shift+D
- `test()`  
⌨️ Ctrl/Cmd+Shift+T
- `check()`  
⌨️ Ctrl/Cmd+Shift+E

# Continuous Integration



# use\_github\_action()

[github.com/r-lib/actions](https://github.com/r-lib/actions)

- "check-standard": Runs R CMD check when you push
- "test-coverage": Compute test coverage and report at [codecov.io](https://codecov.io)
  - (Requires an account and API key)
- "pr-comments": Enables automatic documentation and styling of code via special PR comments
- And more...\*

→ Your Turn

\*<https://github.com/r-lib/actions>

# Website



# use\_pkgdown\_github\_pages()

[pkgdown.r-lib.org/](https://pkgdown.r-lib.org/)



- Automatically creates a website using pkgdown:
  - Function documentation
  - Dataset documentation
  - Vignettes (and articles)
  - README and NEWS
- Sets up a GitHub Action to deploy on GitHub Pages



# Vignettes

## Long-form documentation

- “Article” format
- Demonstrate a common use case or problem your package is designed to solve.
- `use_vignette("short-name", "Longer Title")`

# Review: functions

## Run once

- `create_package()`
- `use_git()`
- `use_github()`
- `use_mit_license()`
- `use_testthat()`
- `use_readme_rmd()`
- `use_pkdown_github_pages()`

## Run periodically

- `use_r()`
- `use_test()`
- `use_package()`
- `rename_files()`
- `use_github_action()`

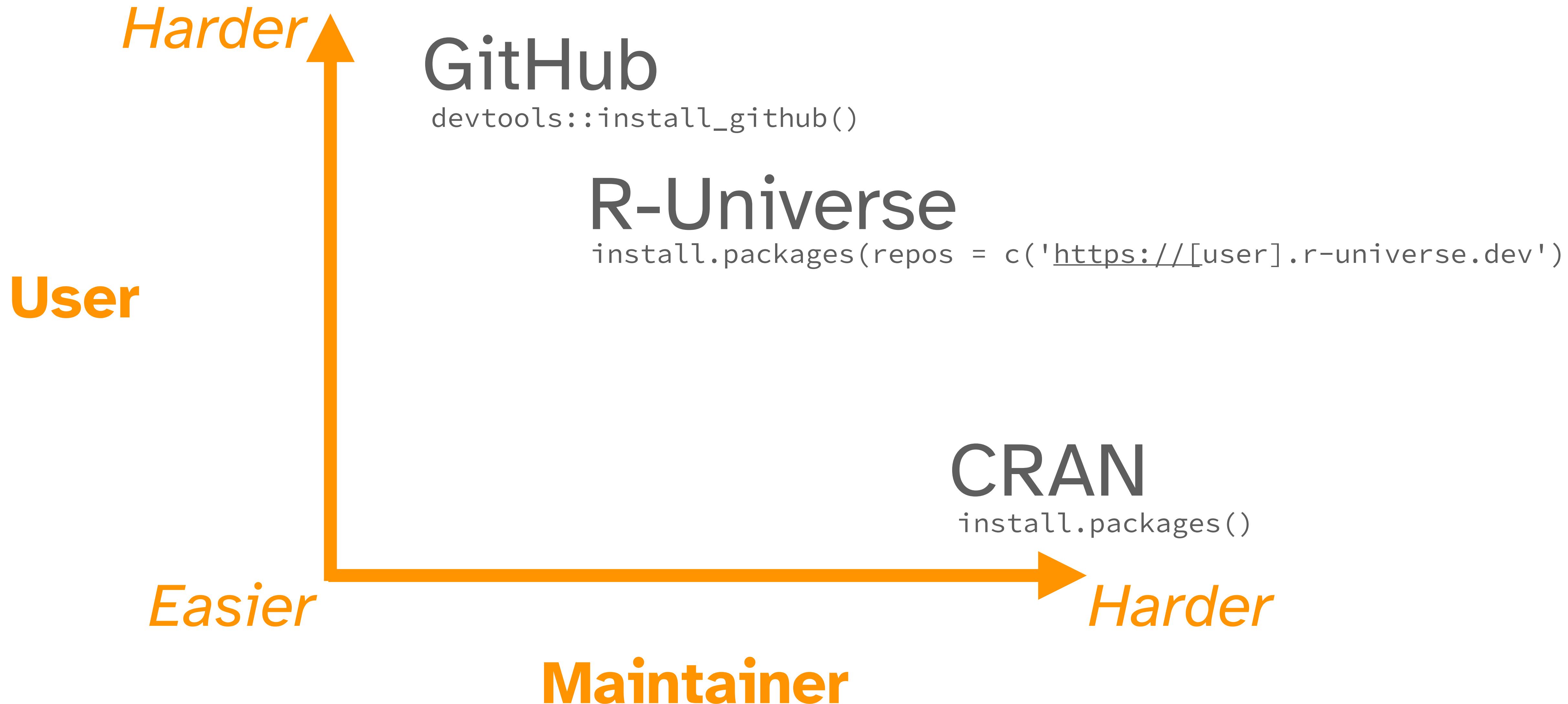
## Run frequently

- `load_all()`  
⌨️ Ctrl/Cmd+Shift+L
- `document()`  
⌨️ Ctrl/Cmd+Shift+D
- `test()`  
⌨️ Ctrl/Cmd+Shift+T
- `check()`  
⌨️ Ctrl/Cmd+Shift+E

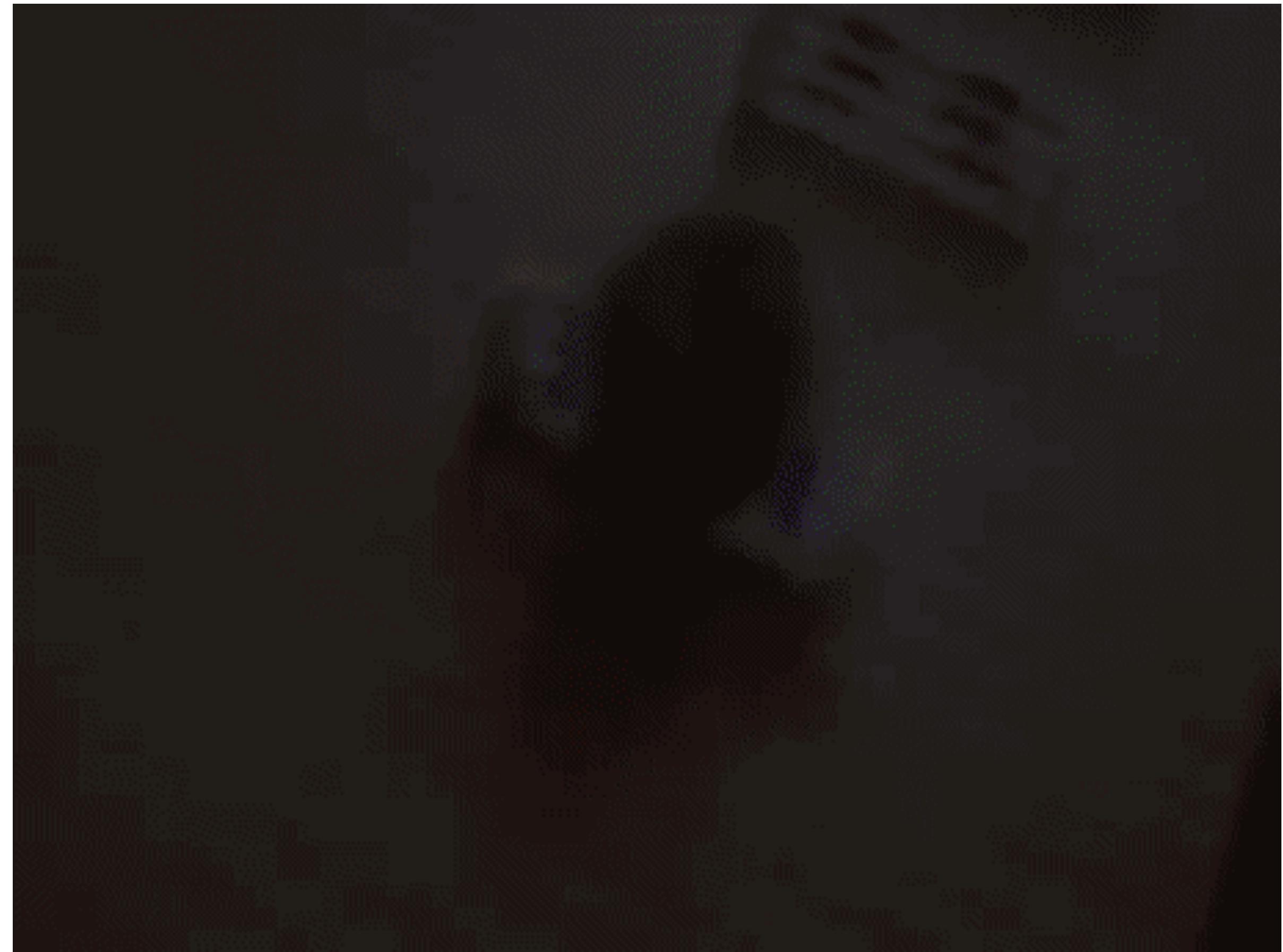
# Sharing your Package



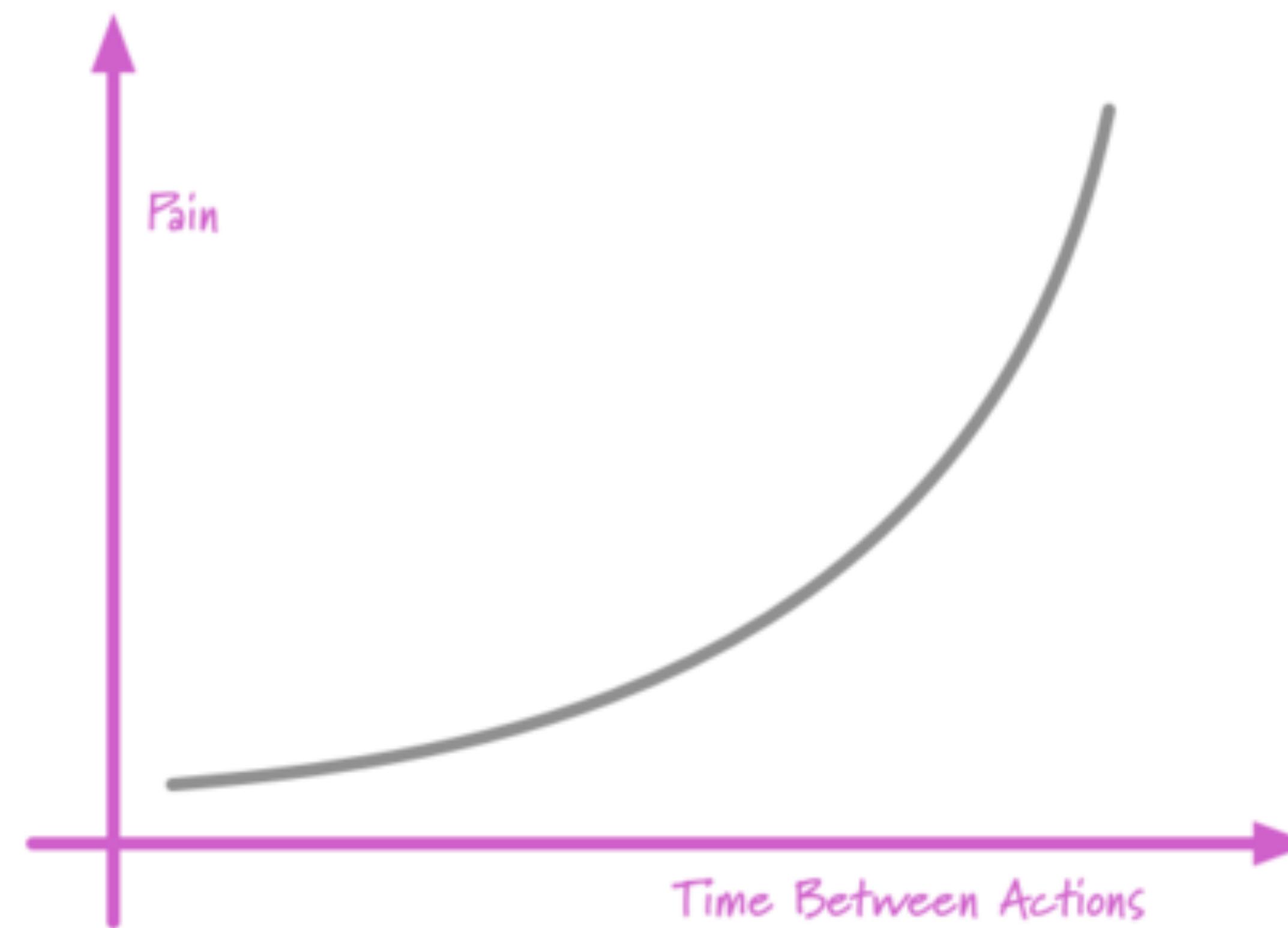
# Package Distribution



# Releasing your package to CRAN



# “If it hurts, do it more often”



# Releasing to CRAN

TLDR:

- `use_release_issue()`
  - Opens a GitHub issue with a checklist
- `release()`
  - Runs through an additional list of checks
  - Builds package bundle and submits to CRAN

## Release libminer 0.1.0 #1

[Open](#)

[25 tasks](#)

ateucher opened this issue 4 minutes ago · 0 comments



ateucher commented 4 minutes ago

Owner ...

First release:

- `usethis::use_news_md()`
- `usethis::use_cran_comments()`
- Update (aspirational) install instructions in README
- Proofread `Title:` and `Description:`
- Check that all exported functions have `@return` and `@examples`
- Check that `Authors@R:` includes a copyright holder (role 'cph')
- Check [licensing of included files](#)
- Review <https://github.com/DavisVaughan/extrachecks>

Prepare for release:

- `git pull`
- `urlchecker::url_check()`
- `devtools::build_readme()`
- `devtools::check(remote = TRUE, manual = TRUE)`
- `devtools::check_win-devel()`
- `git push`
- Draft blog post

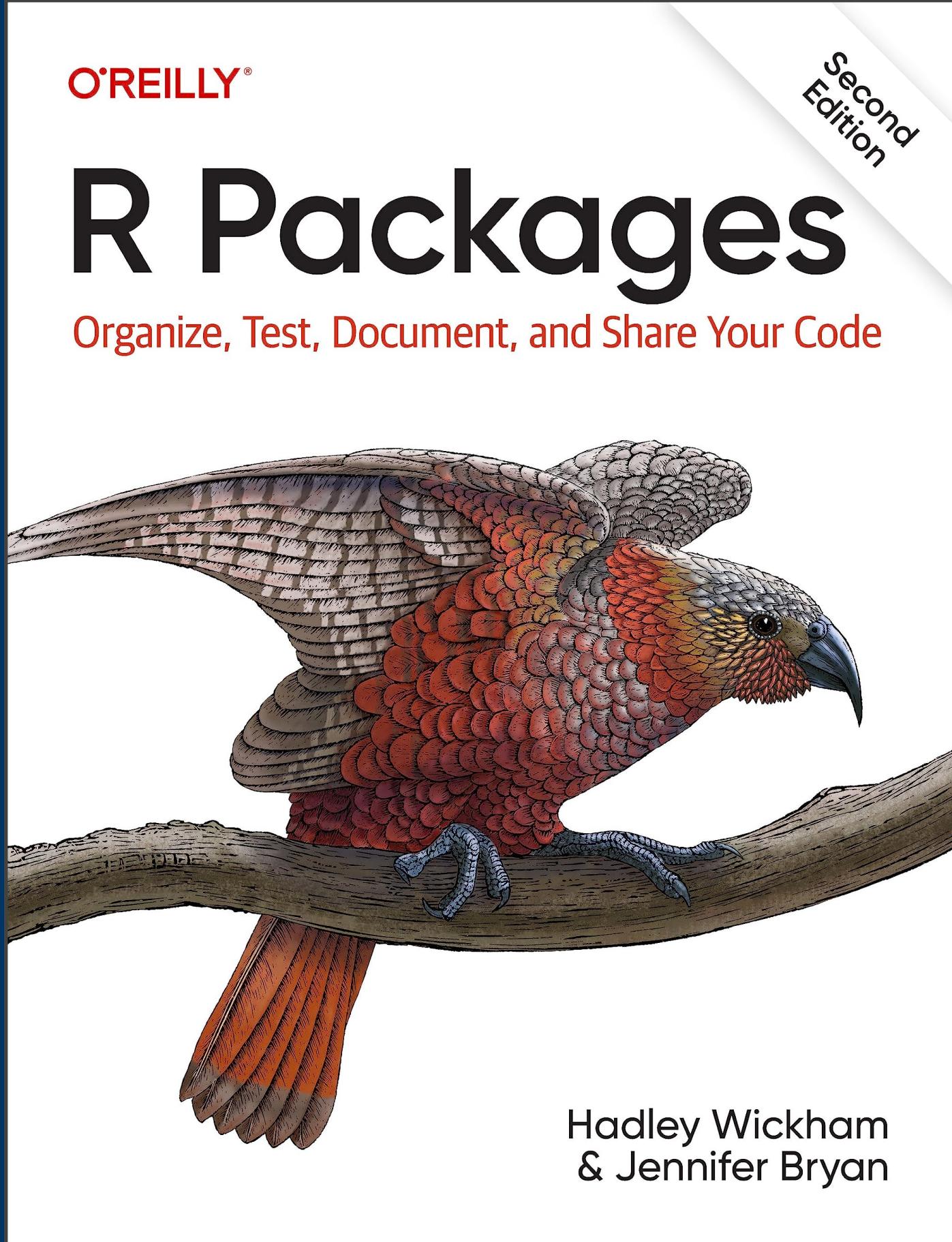
Submit to CRAN:

- `usethis::use_version('minor')`
- `devtools::submit_cran()`
- Approve email

Wait for CRAN...

- Accepted 🎉
- Add preemptive link to blog post in pkgdown news menu
- `usethis::use_github_release()`
- `usethis::use_dev_version(push = TRUE)`
- `usethis::use_news_md()`
- Finish blog post
- Tweet

# Resources



r-pkgs.org

Happy Git and GitHub for the userR

Table of contents

- Let's Git started
- 1 Why Git? Why GitHub?
- 2 Contributors
- 3 Workshops
- Installation
- Half the battle
- 4 Register a GitHub account
- 5 Install or upgrade R and RStudio
- 6 Install Git
- 7 Introduce yourself to Git
- 8 Install a Git client
- Connect Git, GitHub, RStudio
- Can you hear me now?
- 9 Personal access token for HTTPS
- 10 Set up keys for SSH
- 11 Connect to GitHub
- 12 Connect RStudio to Git and GitHub
- 13 Detect Git from RStudio
- 14 RStudio, Git, GitHub Hell

Still from Heaven King video

Happy Git provides opinionated instructions on how to:

- Install Git and get it working smoothly with GitHub, in the shell and in the RStudio IDE.
- Develop a few key workflows that cover your most common tasks.
- Integrate Git and GitHub into your daily work with R and R Markdown.

happygitwithr.com

Tidy design principles

Welcome

- 1 Unifying principles
- Implementation
- 2 Names attribute
- 3 Name all but the most important arguments
- 4 Function names
- Scannable definitions
- 5 Make inputs explicit
- 6 Put the most important arguments first
- 7 Required args shouldn't have defaults
- 8 Put ... after required arguments
- 9 Keep defaults short and sweet
- 10 Enumerate possible options
- Function arguments
- 11 Case study: rep()
- 12 Avoid magical defaults
- 13 Explain important defaults

1 Unifying principles →

design.tidyverse.org  
tidydesign.substack.com

# Survey

<http://pos.it/conf-workshop-survey>

- Your feedback is crucial!
- Data from the survey informs curriculum and format decisions for future conf workshops, and we really appreciate you taking the time to provide it.