



Fundamentals of Package Development

Andy Teucher and Sam Albers
Department of Fisheries and Oceans
November 4th to 6th, 2024

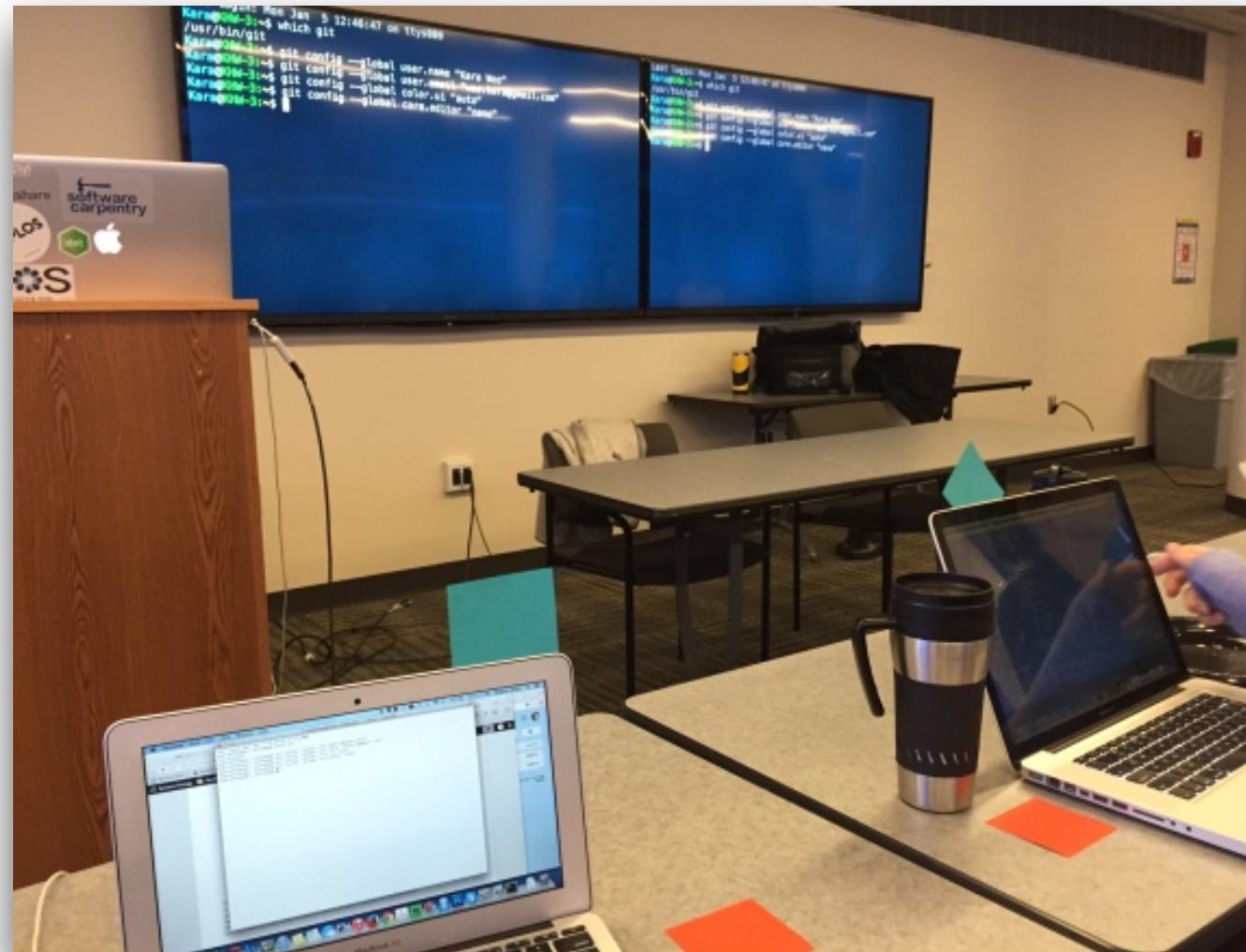
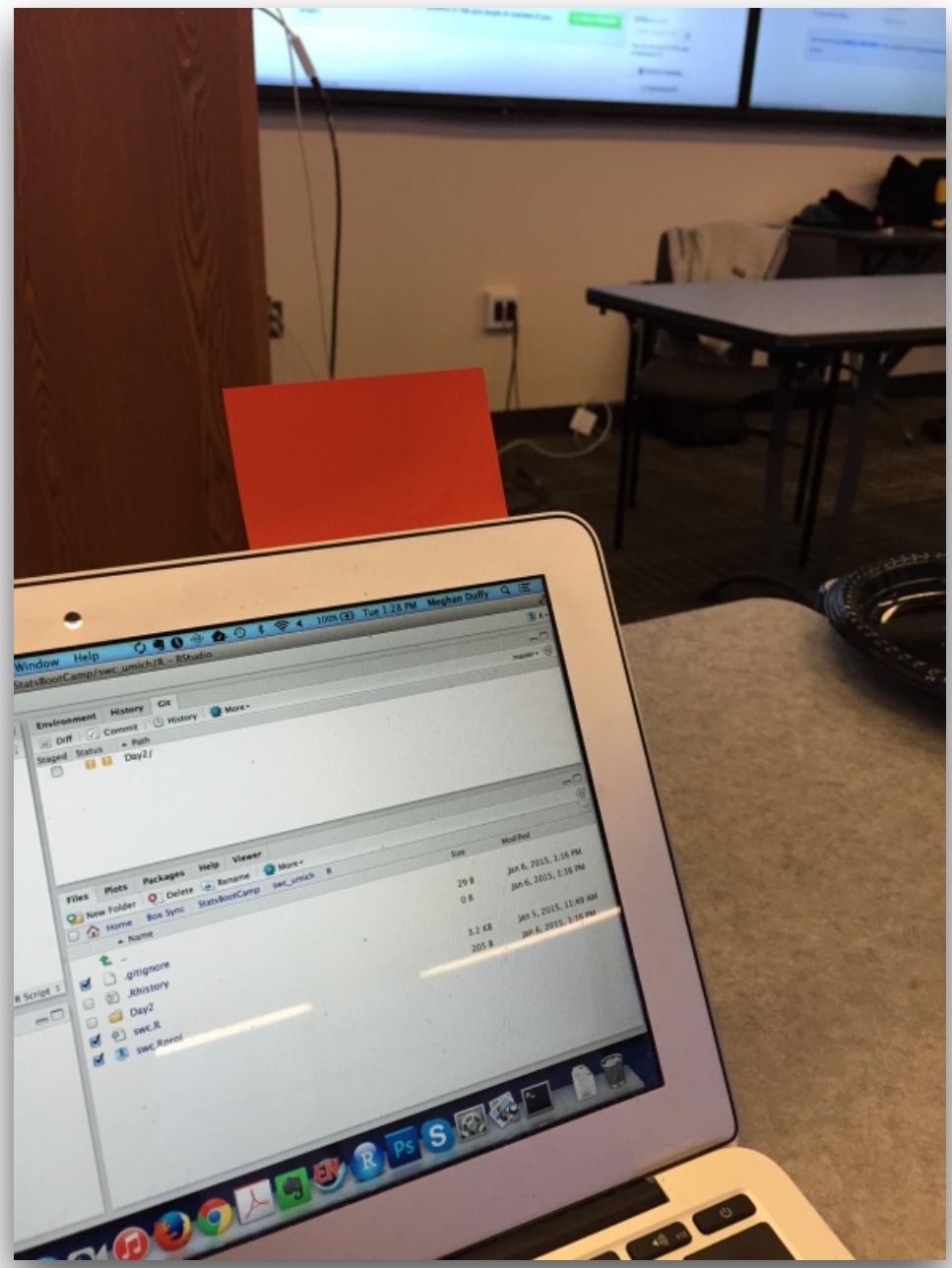
Welcome!

- Instructors:
 - Andy Teucher
 - andyteucher.ca
 - GitHub: [ateucher](https://github.com/ateucher)
 - Mastodon: [@andyteucher@fosstodon.org](https://fosstodon.org/@andyteucher)
 - Sam Albers
 - samalbers.science
 - GitHub: [boshek](https://github.com/boshek)

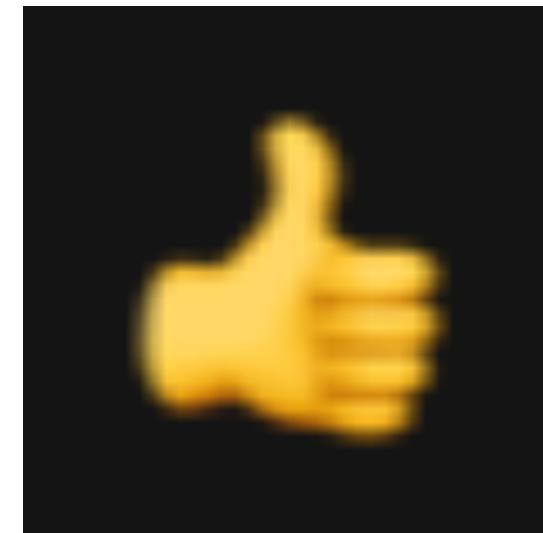
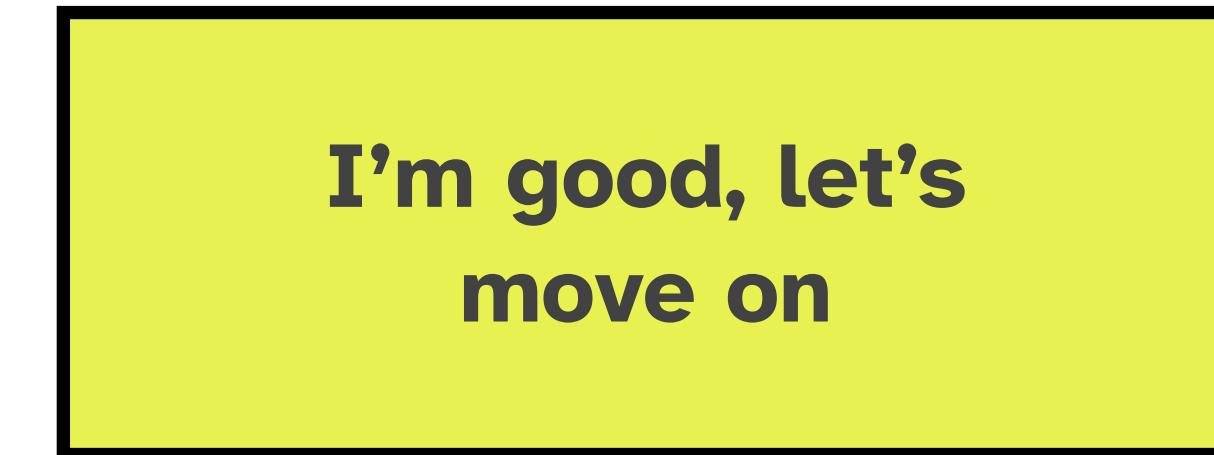
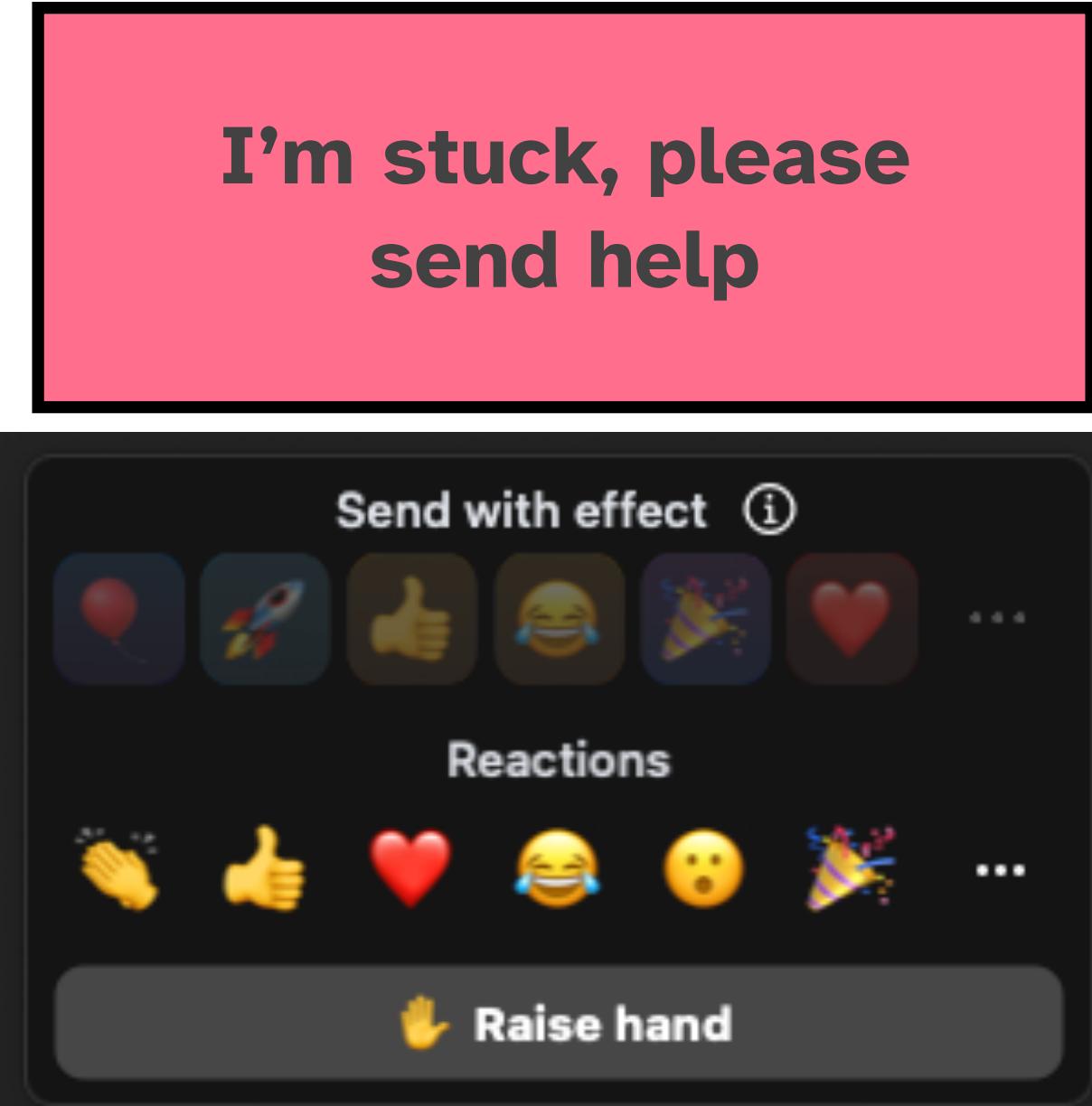
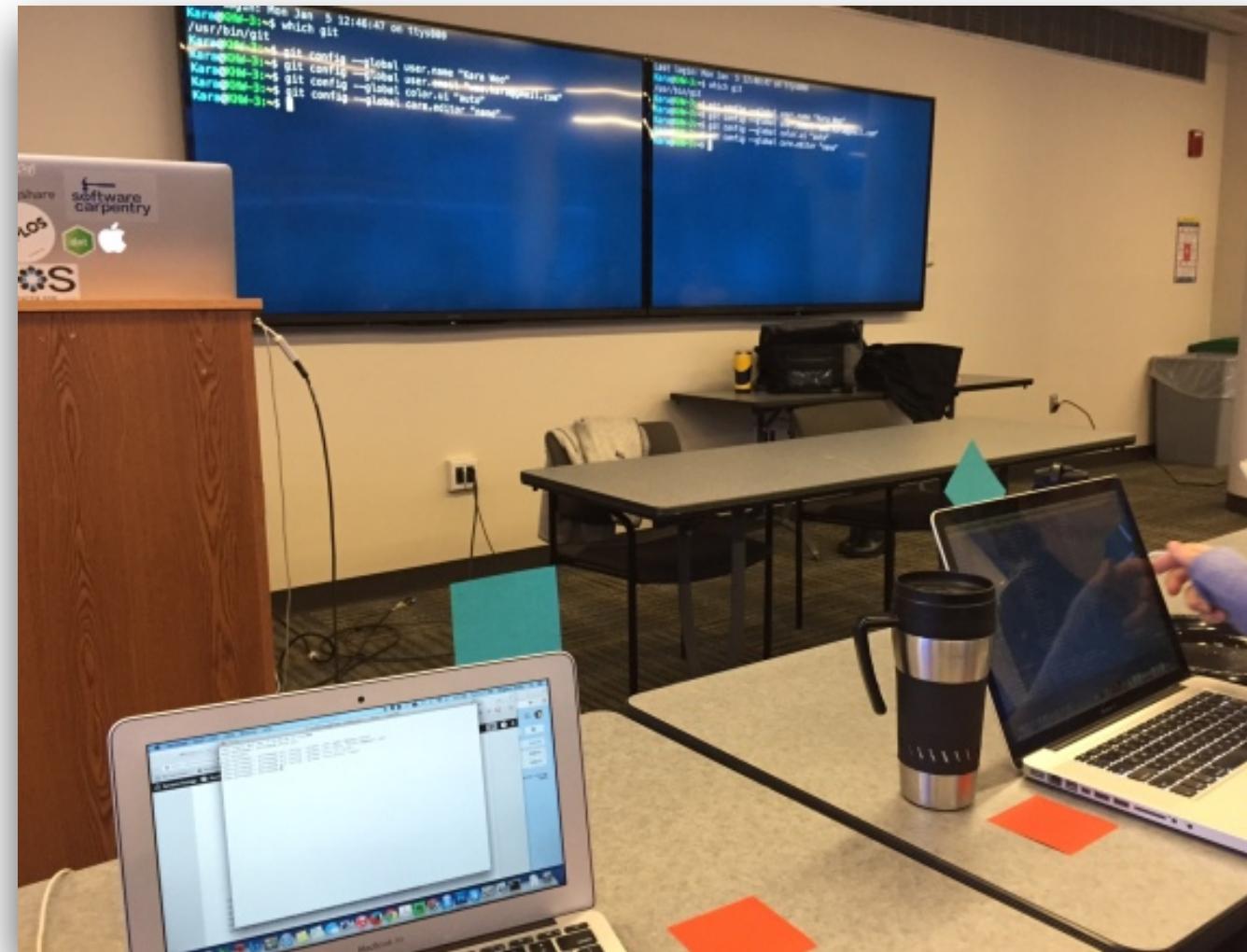
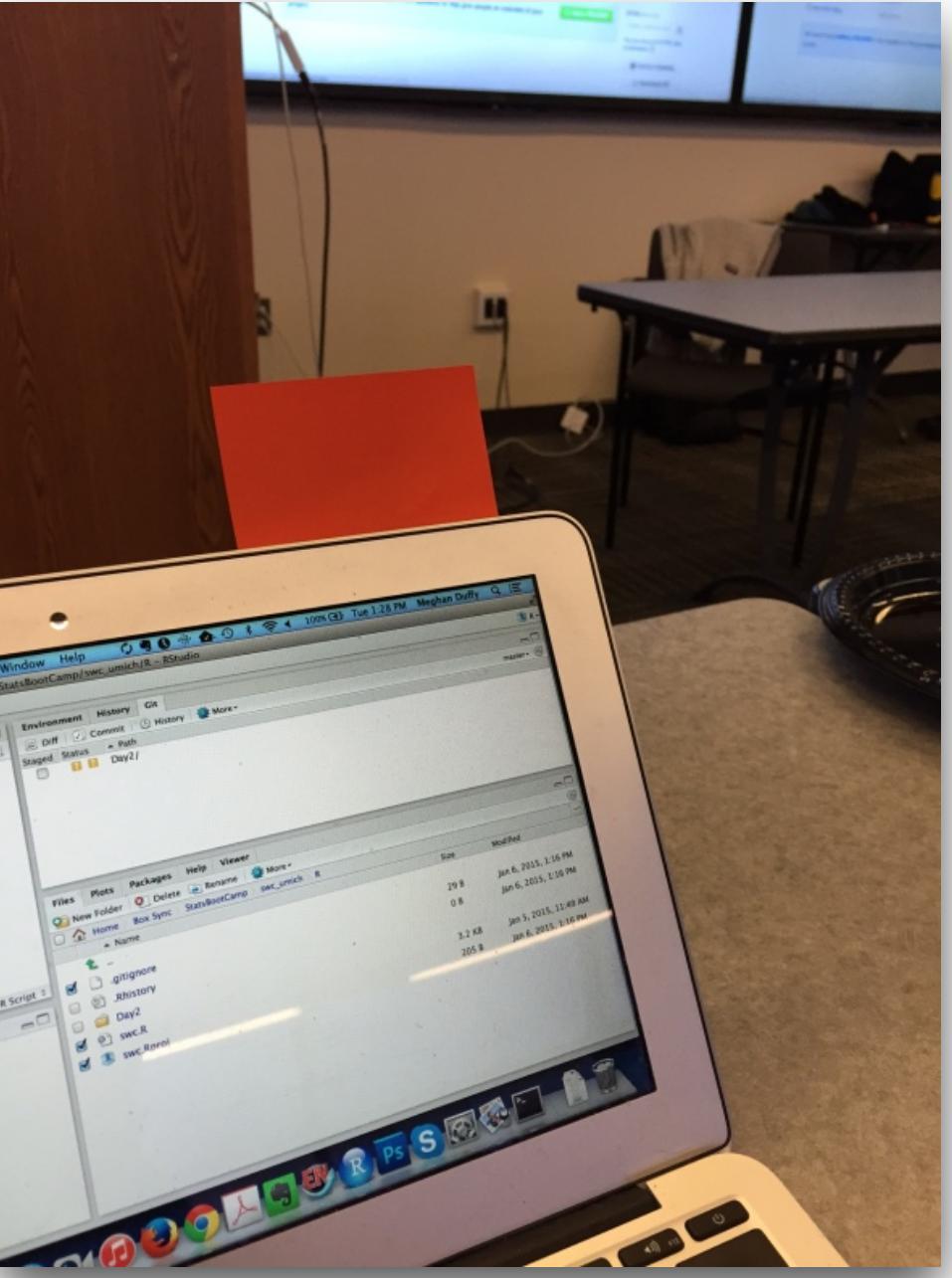
Welcome!

- This is a three part 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.
- Code of Conduct:
 - [https://github.com/ateucher/pkg-dev-psc-2024-04-29/blob/main/
CODE_OF_CONDUCT.md](https://github.com/ateucher/pkg-dev-psc-2024-04-29/blob/main/CODE_OF_CONDUCT.md)
 - ❤️ Treat everyone with respect
 - 🌟 Everyone should feel welcome

Sticky Notes



Sticky Notes



Discussions / Q & A

- Zoom Chat
- GitHub Discussions:
<https://github.com/ateucher/pkg-dev-dfo-2024-11/discussions>
- Open the welcome discussion:
 - Introduce yourself:
 - Office/department, city/town
 - Why you're here
 - Share something you're good at

Resources

- Prework:

<https://andyteucher.ca/pkg-dev-dfo-2024-11/setup.html>

- Workshop website:

<https://andyteucher.ca/pkg-dev-dfo-2024-11/>

- Cheatsheet:

<https://rstudio.github.io/cheatsheets/html/package-development.html>

Day 1: Schedule and Learning Objectives

08:00 - 09:20	What is a Package? Package Structure and State	80 min
09:20 - 09:35	Break	15 min
09:35 - 10:55	Package Creation and Metadata; Documentation	80 min
10:55 - 11:10	Break	15 min
11:10 - 12:30	Testing Your Package	80 min

Day 2: Schedule and Learning Objectives

08:00 - 09:20	Dependencies and Continuous Integration	80 min
09:20 - 09:35	Break	15 min
09:35 - 10:55	Design Principles	80 min
10:55 - 11:10	Break	15 min
11:10 - 12:30	Website Creation and General Package Discussion	80 min

Day 3: Schedule and Learning Objectives

08:00 - 09:20	Using the Tidyverse in Your Package	80 min
09:20 - 09:35	Break	15 min
09:35 - 10:55	Communicating with your Users	80 min
10:55 - 11:10	Break	15 min
11:10 - 12:30	Effective Use of GitHub + Distribution	80 min

R Packages (2e)

Hadley Wickham

Jenny Bryan

<https://r-pkgs.org>

O'REILLY®

Second
Edition

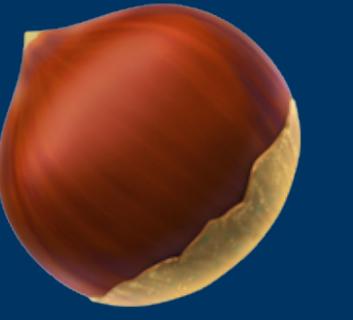
R Packages

Organize, Test, Document, and Share Your Code



Hadley Wickham
& Jennifer Bryan

Packages in a nutshell



Have you ever worked on a package before?

What do you feel the most confused or curious about?

 **Your Turn**

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

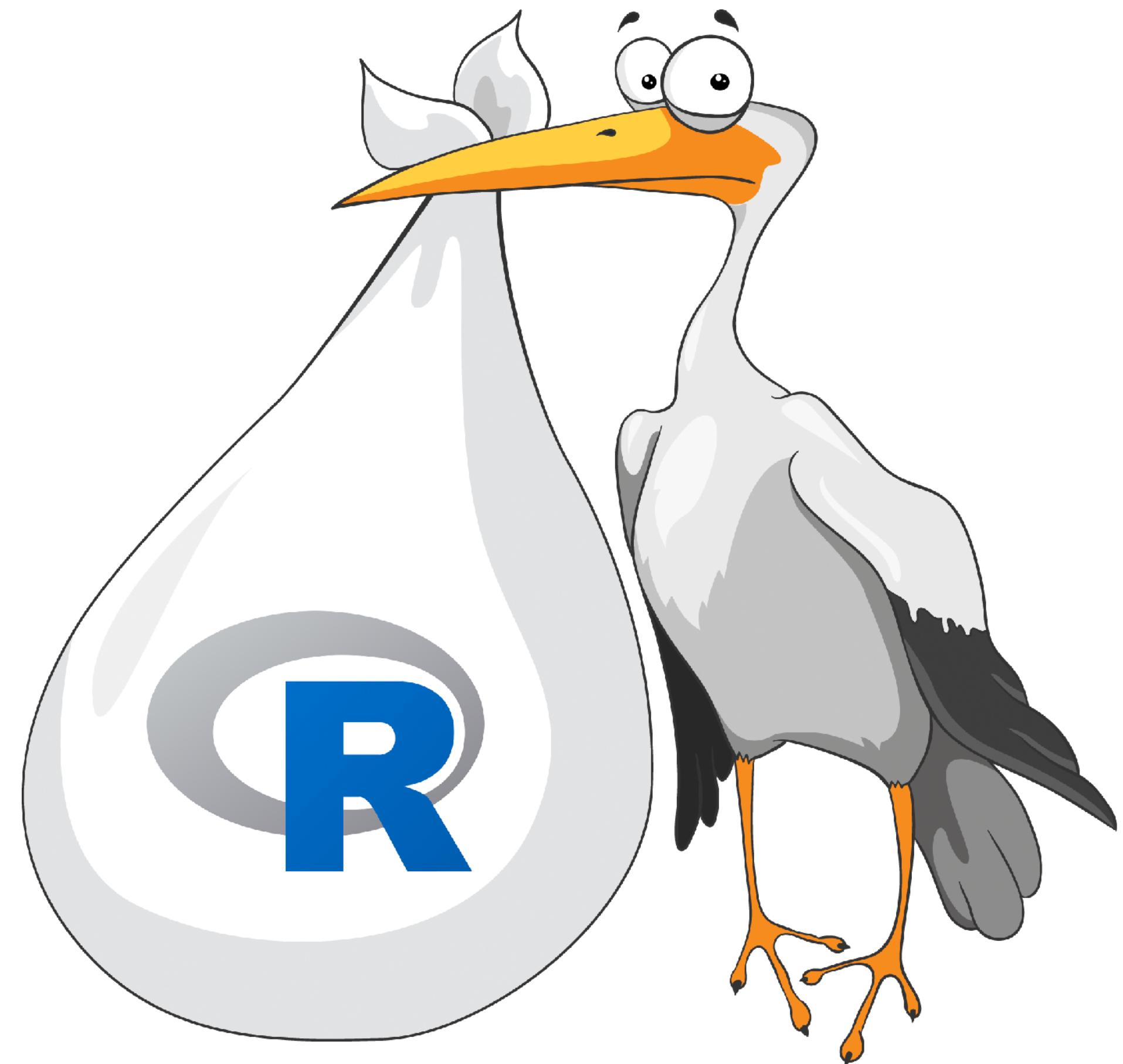
- Performs data analysis
- Collection of one or more `.R` files
- `library()` calls
- Documentation in `#` comments
- Run with `source()` or select+run

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 Chat



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 Chat)



Your Turn

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`
-

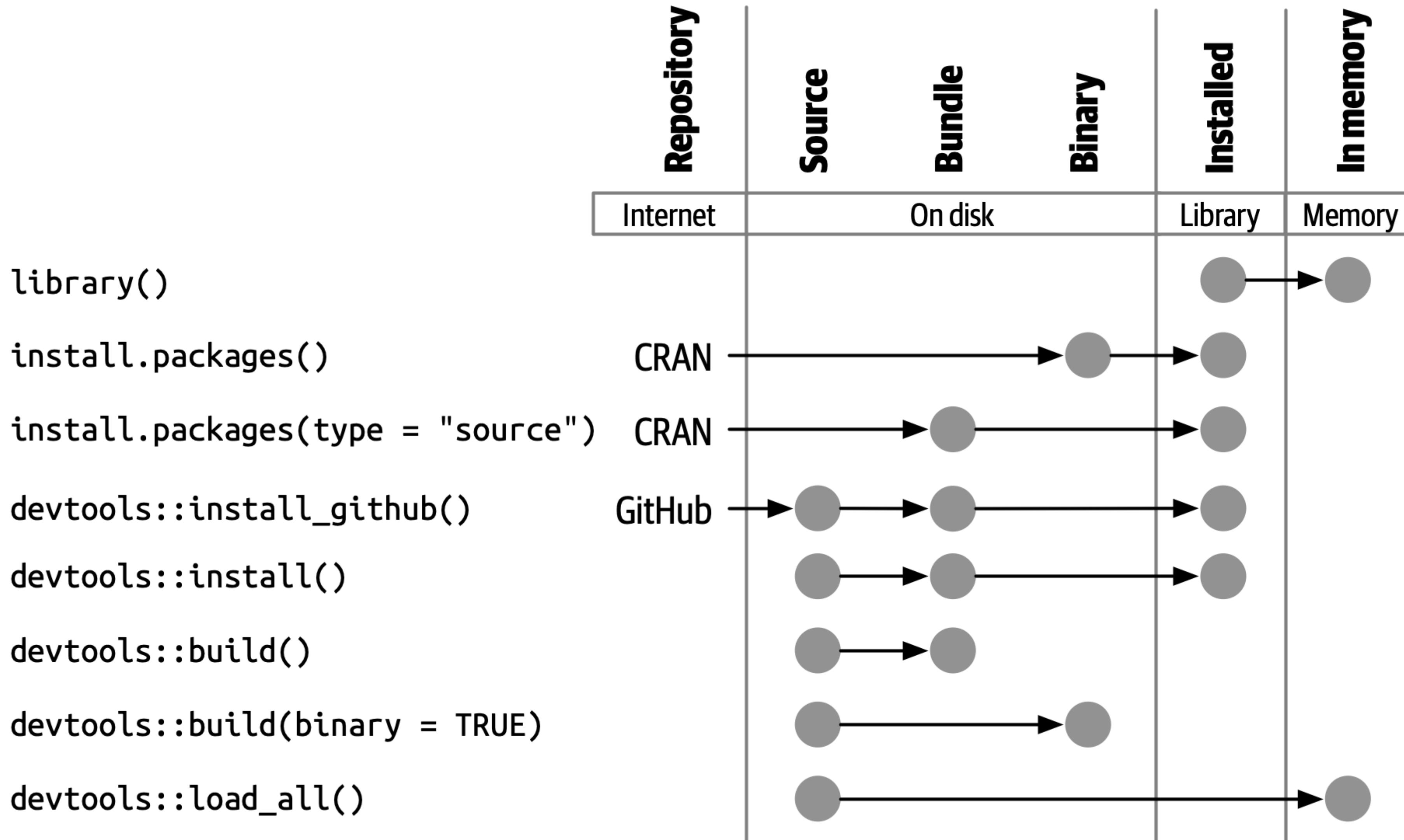
Installed

- 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
- Focus on workflows

We (probably) won't:

- Talk (much) about function writing
- Talk about how to include data in your package (even though it's possible and often helpful)

libminer

Sneak peak of our end goal on GitHub

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

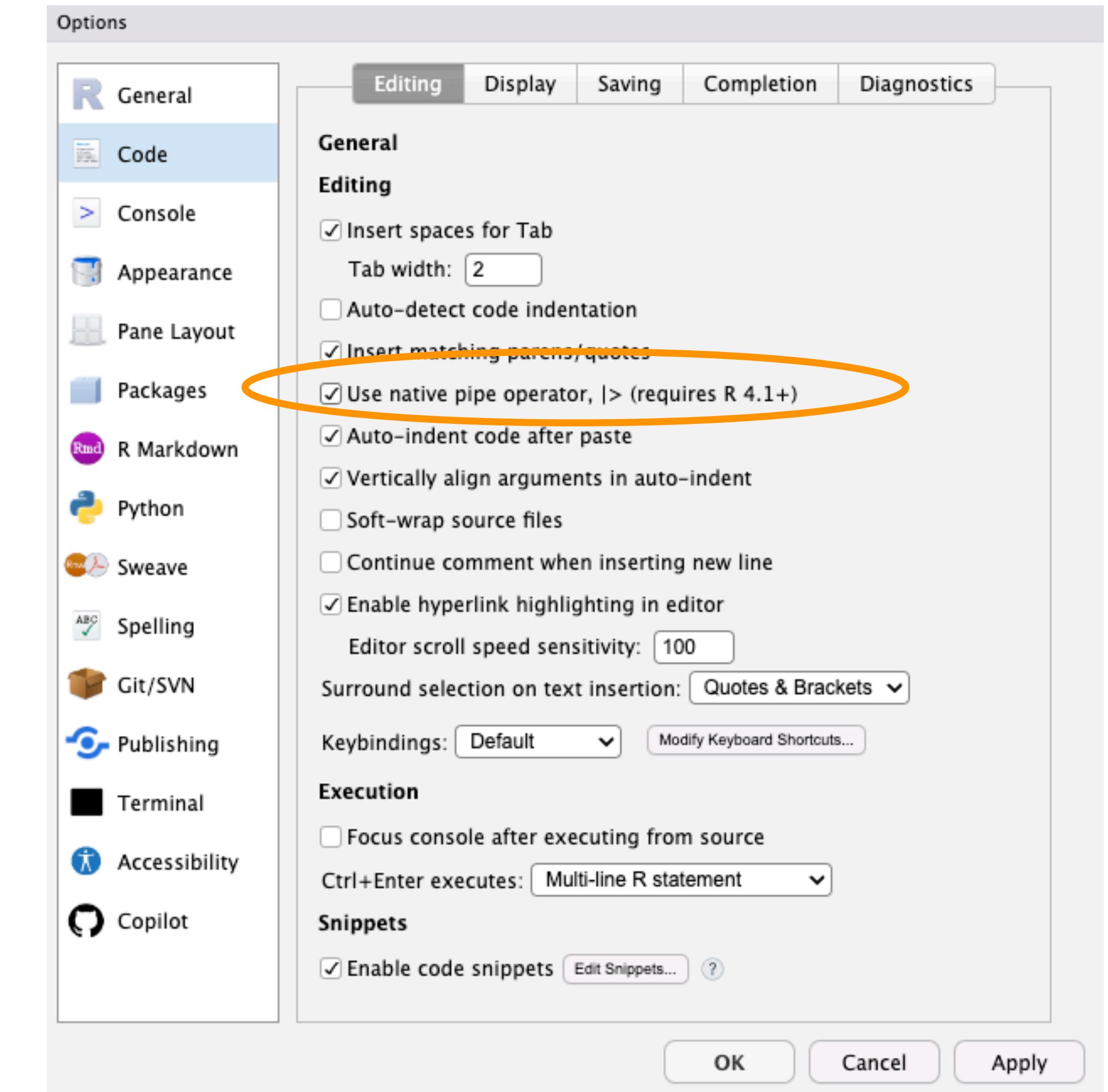
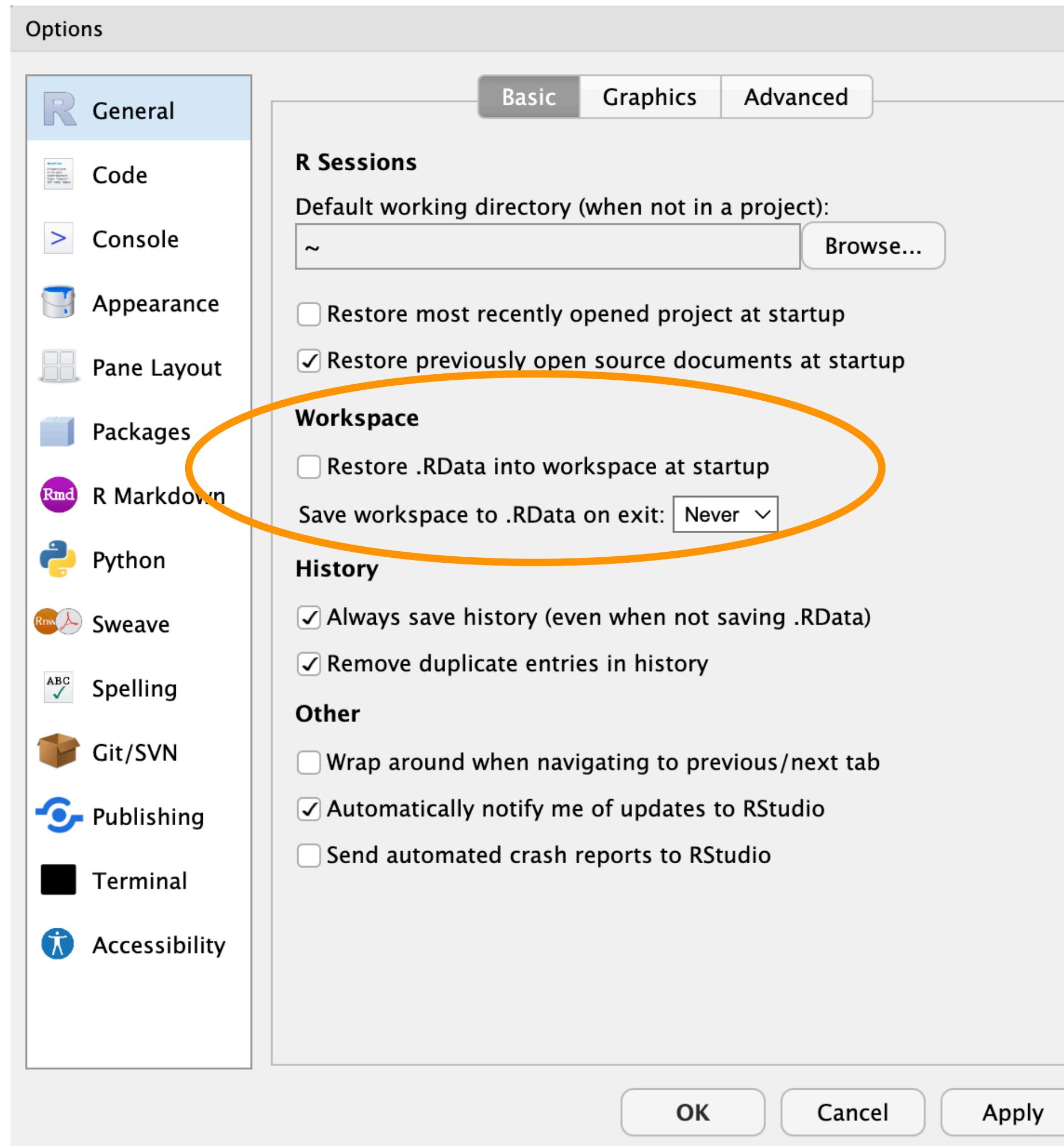


Get Ready



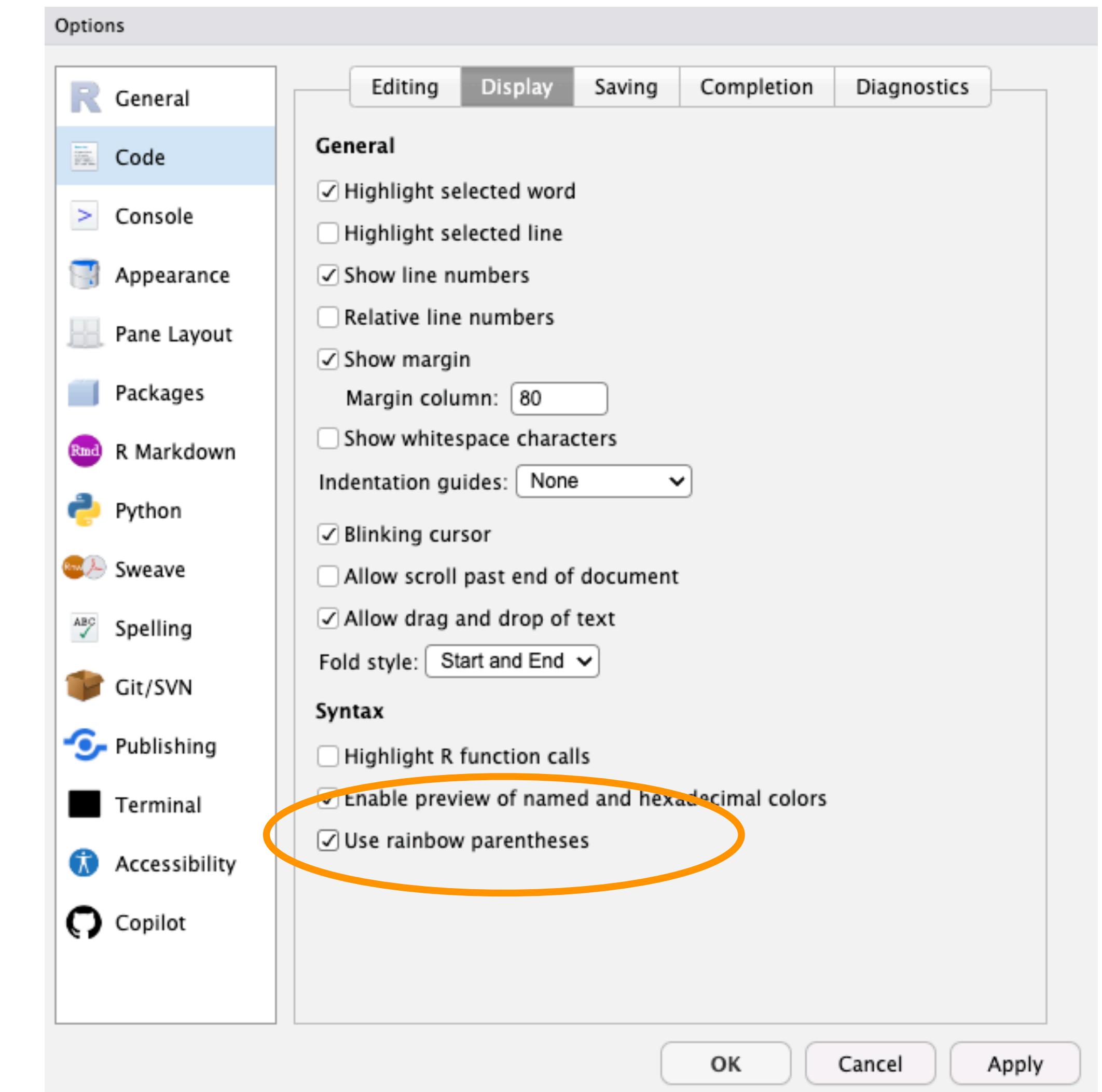
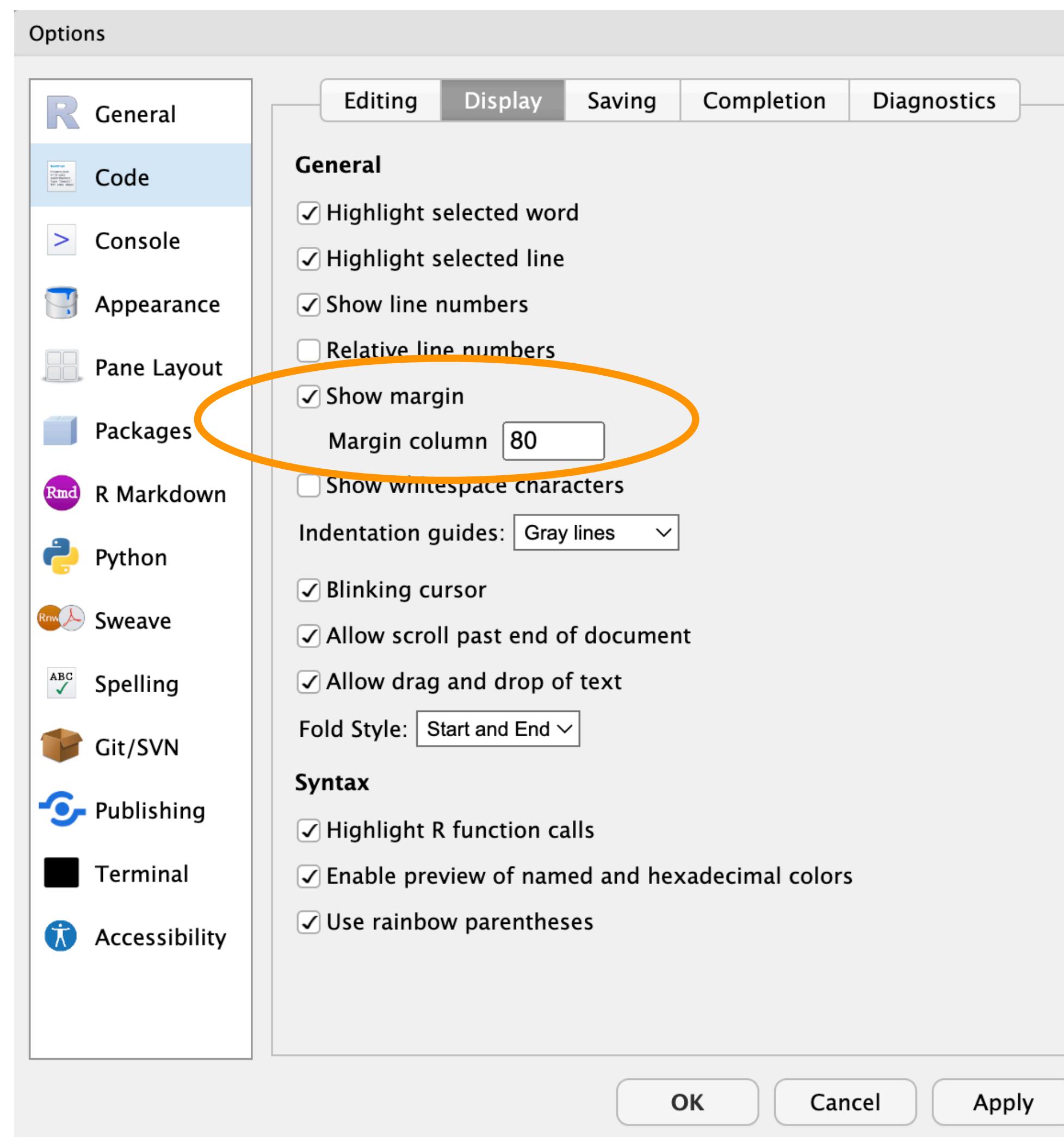
Configure RStudio

Tools > Global Options



Configure RStudio

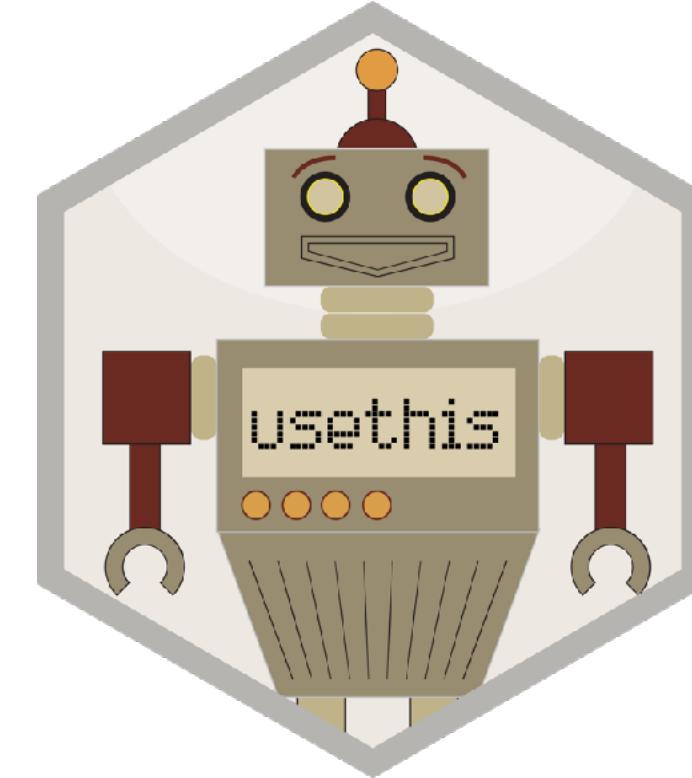
Tools > Global Options



Tools



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



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



Check your setup

```
library(devtools)  
# package development "situation report"  
dev_sitrep()  
# git/github "situation report"  
git_sitrep()
```

➡ Your Turn

Break Time!

I NEED A
BREAK!



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
 - Final part of path will be the package name
- 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? [y/n]: 
#> 1: Abort
#> 2: No
#> 3: Yes
```

→ Your Turn

usethis::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 &

→ 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'
```

➡ Your Turn

Try your function in the new package

But how?

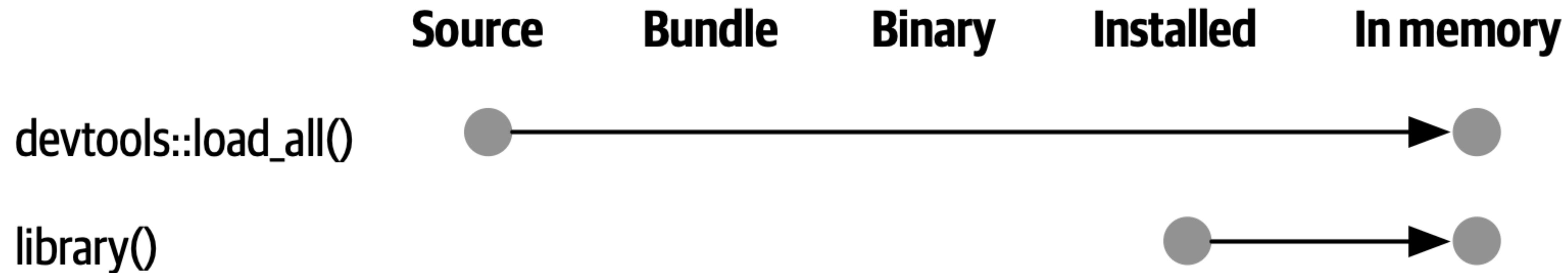
- ~~source("R/my-fun.R")~~
- ~~Send function to console using RStudio (Ctrl/CMD+Return)~~
- ~~devtools::load_all()~~



Ctrl+Shift+L (Windows & Linux)

`load_all()`

~= `install.packages() + library()`

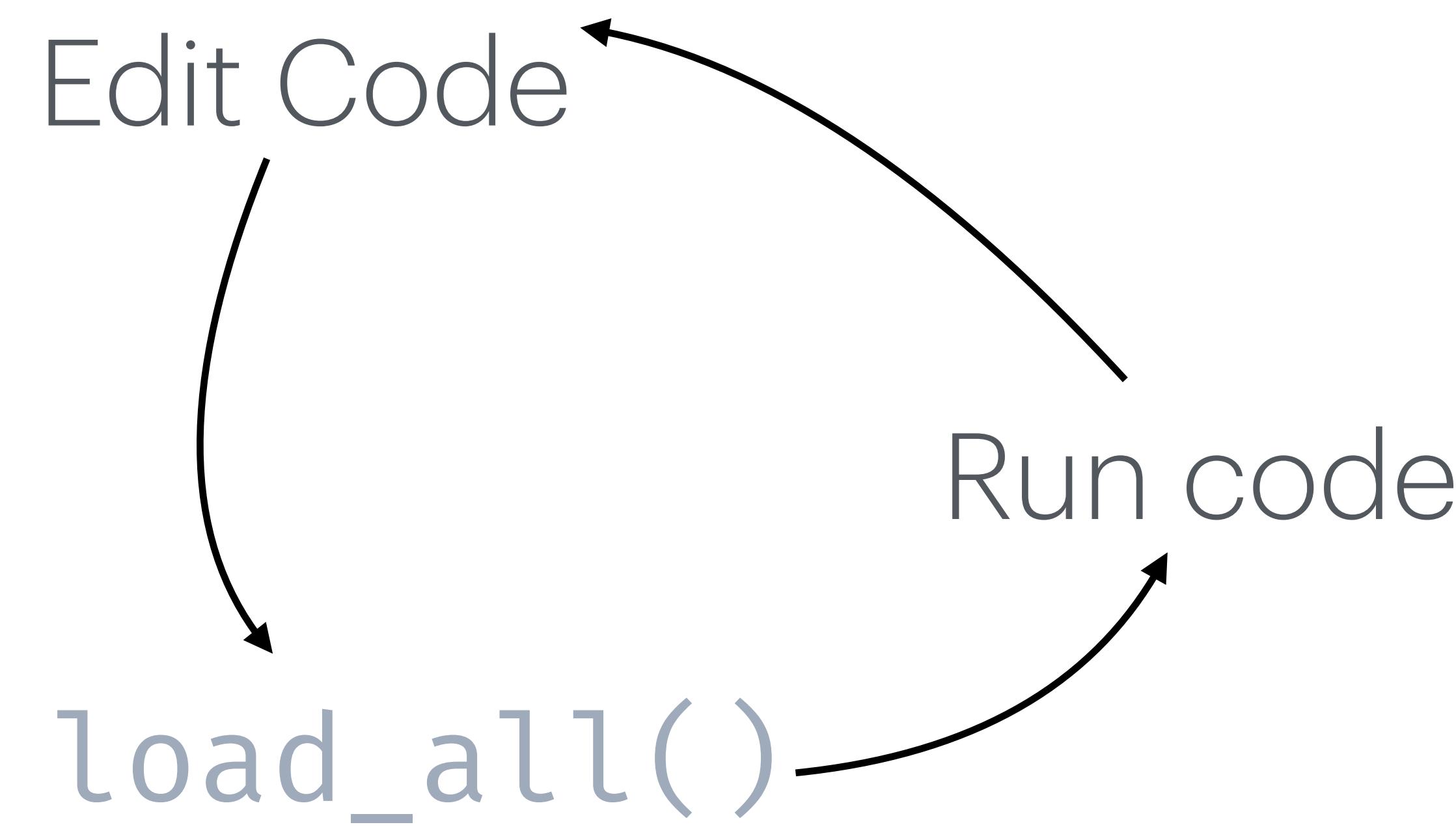


- Simulates building, installing, and attaching your package
- Makes all of the functions from your package immediately available to use
- Allows fast iteration of editing and test-driving your functions
- Good reflection of how users will interact with your package*

Try it out, and commit your
changes



Workflow



⌨️ Ctrl/Cmd+Shift+L

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 ✅
```



Your Turn

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.

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.
 - **CCO:** dedicated to public domain.
 - **CC-BY:** Free to share and adapt, must give appropriate credit.

use_mit_license()

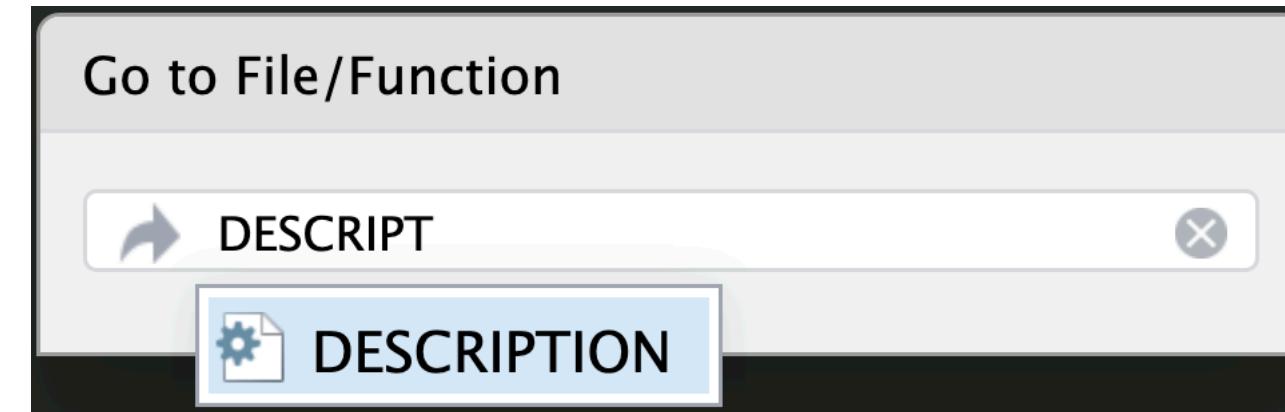
- ✓ Adding 'MIT + file LICENSE' to License
- ✓ Writing 'LICENSE'
- ✓ Writing 'LICENSE.md'
- ✓ Adding '^LICENSE\\\\.md\$' to '.Rbuildignore'

➡ 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: 7.3.
```

➡ Your Turn

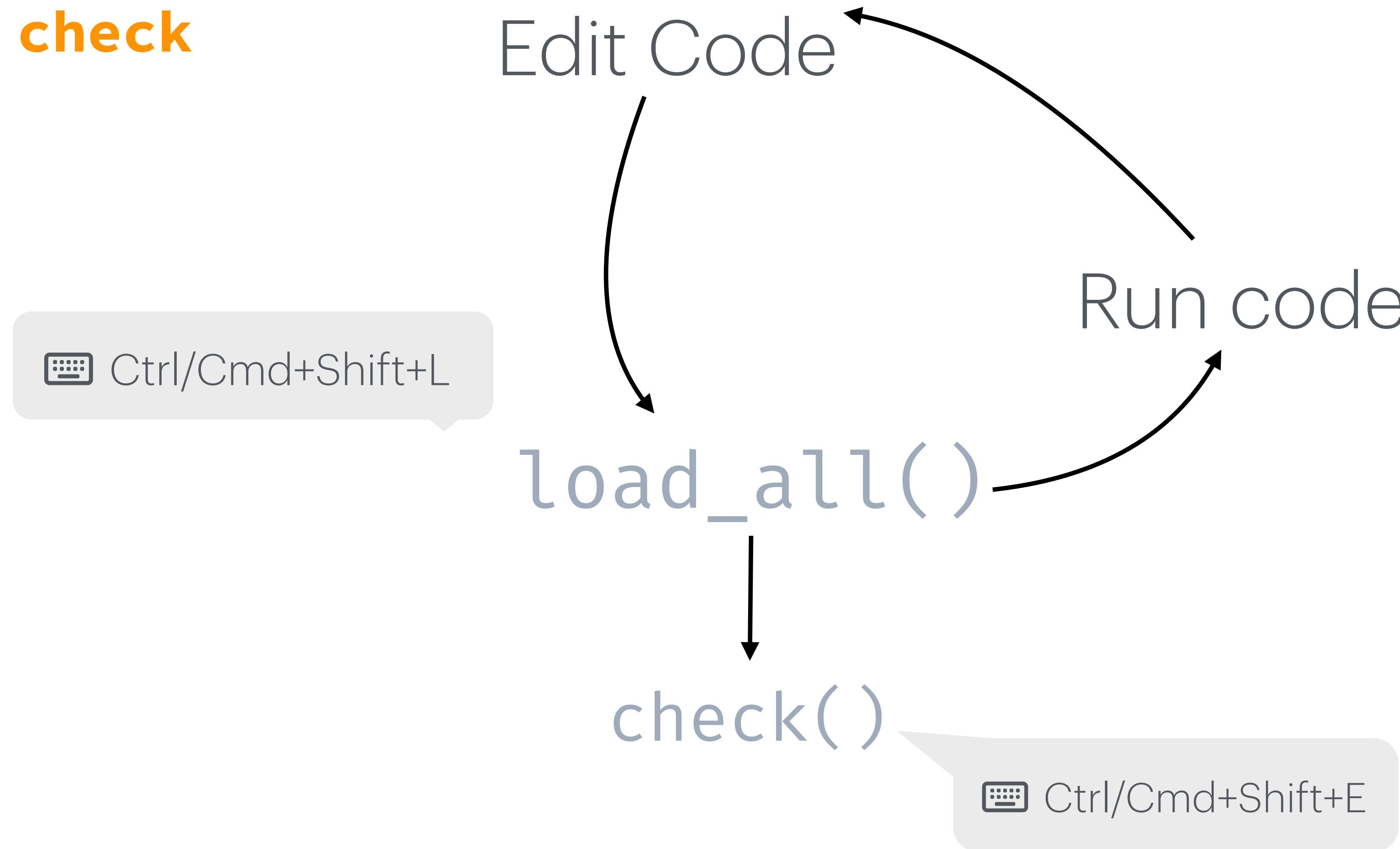
The DESCRIPTION file

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



check() again

```
check()
```

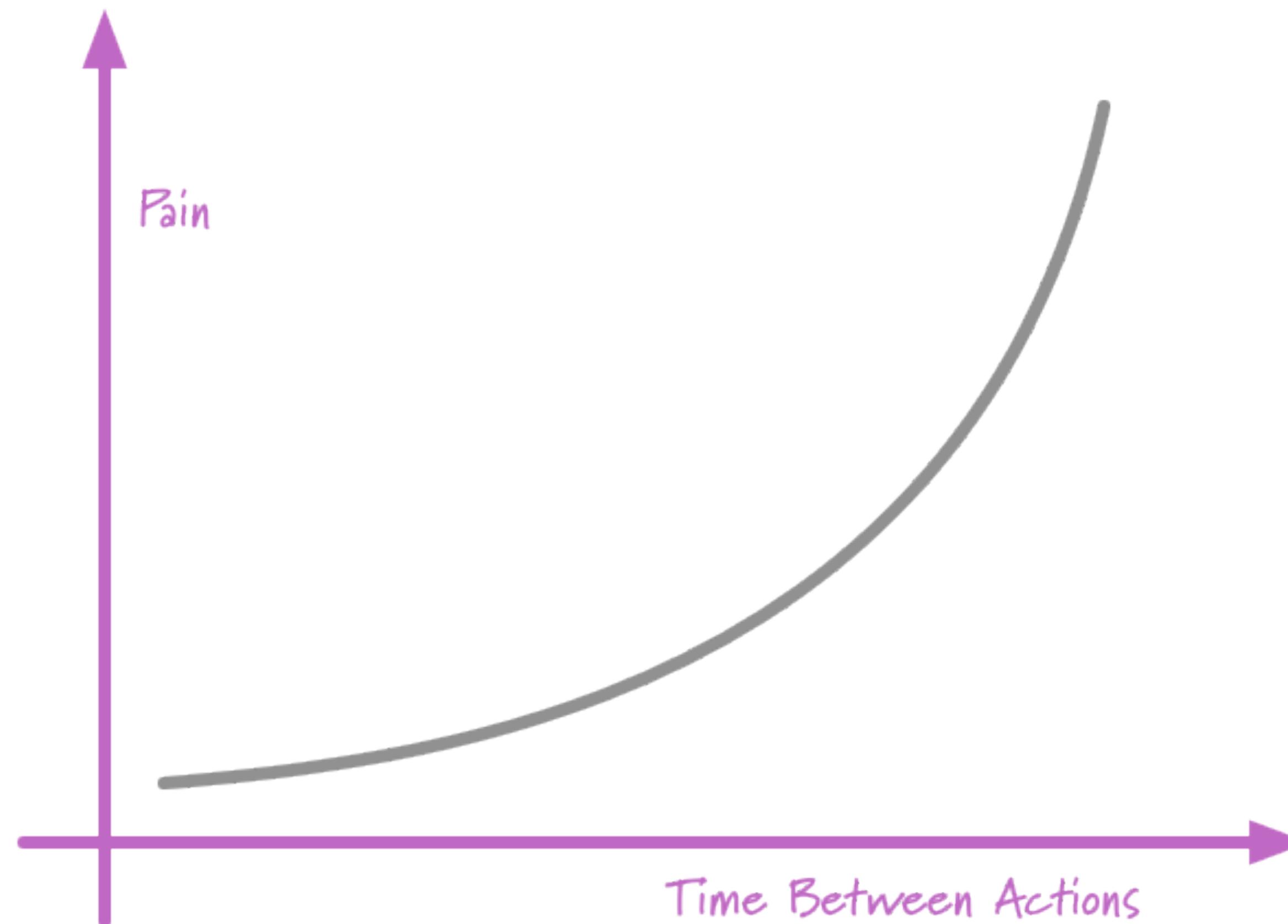
```
#> == Documenting ==
...
#> == Building ==
...
#> == Checking ==
...
#> — R CMD check results —
#> Duration: 3.1s
#>
#> 0 errors ✅ | 0 warnings ✅ | 0 notes ✅
```





**TED
LASSO**

"If it hurts, do it more often."



"If it hurts, do it more often."

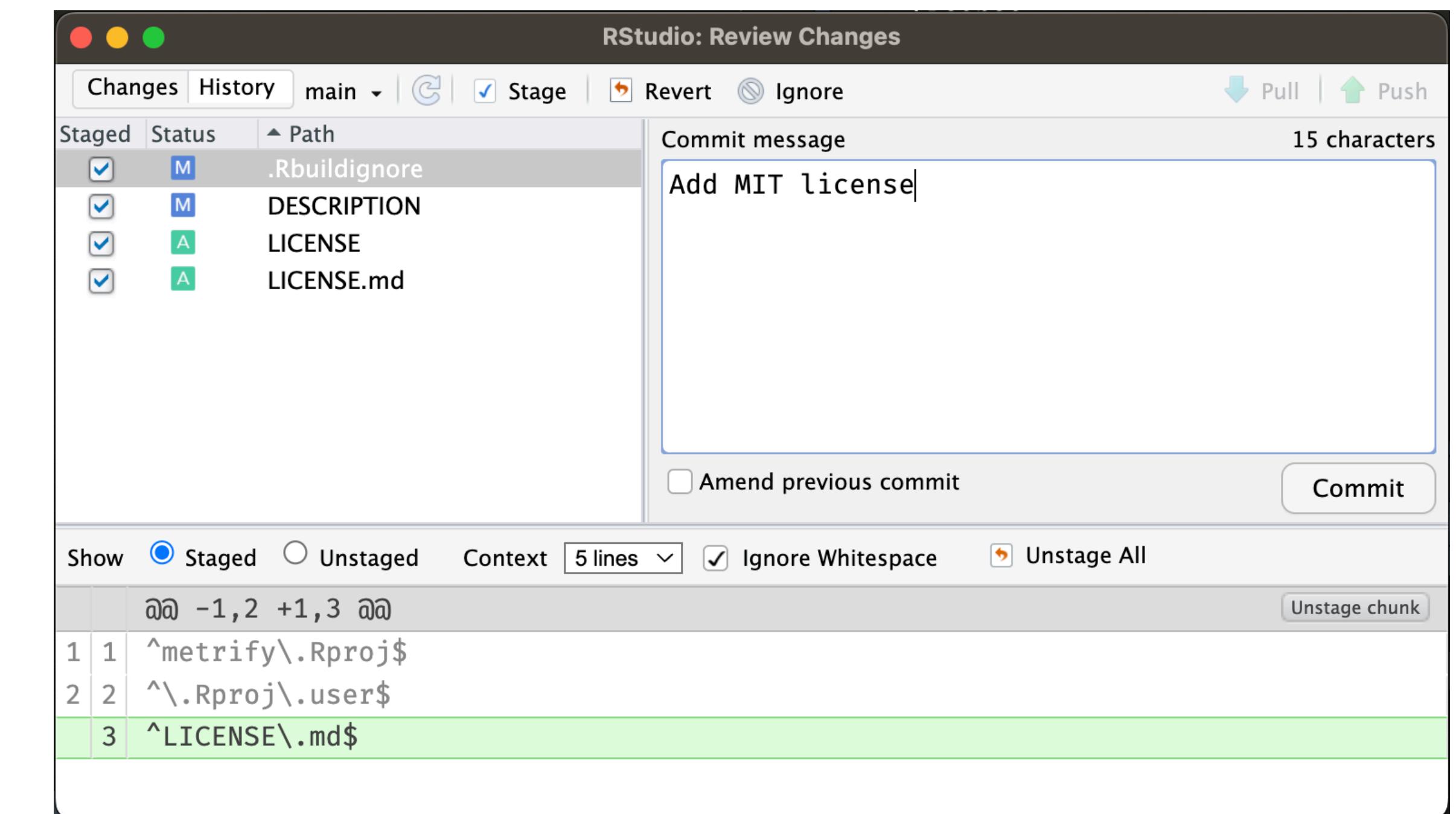
- If `check()` goes from passing to failing after you've tinkered with one function or touched 10 lines of code, it's clear what to troubleshoot.
- If you've added 15 functions that all call each other and touched 100s of lines of code, bless your heart.
- Better to learn of a regrettable choice before you've built upon it for days or weeks.

Commit changes to git



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

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



GitHub

Put your package code on GitHub

- Prerequisites:
 - GitHub account
 - Followed instructions to get GitHub PAT
 - <https://andyteucher.ca/pkg-dev-dfo-2024-11/setup.html#github>
 - Verify
 - `git_sitrep()`
 - Some things really are magic:
 - `use_github()`



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

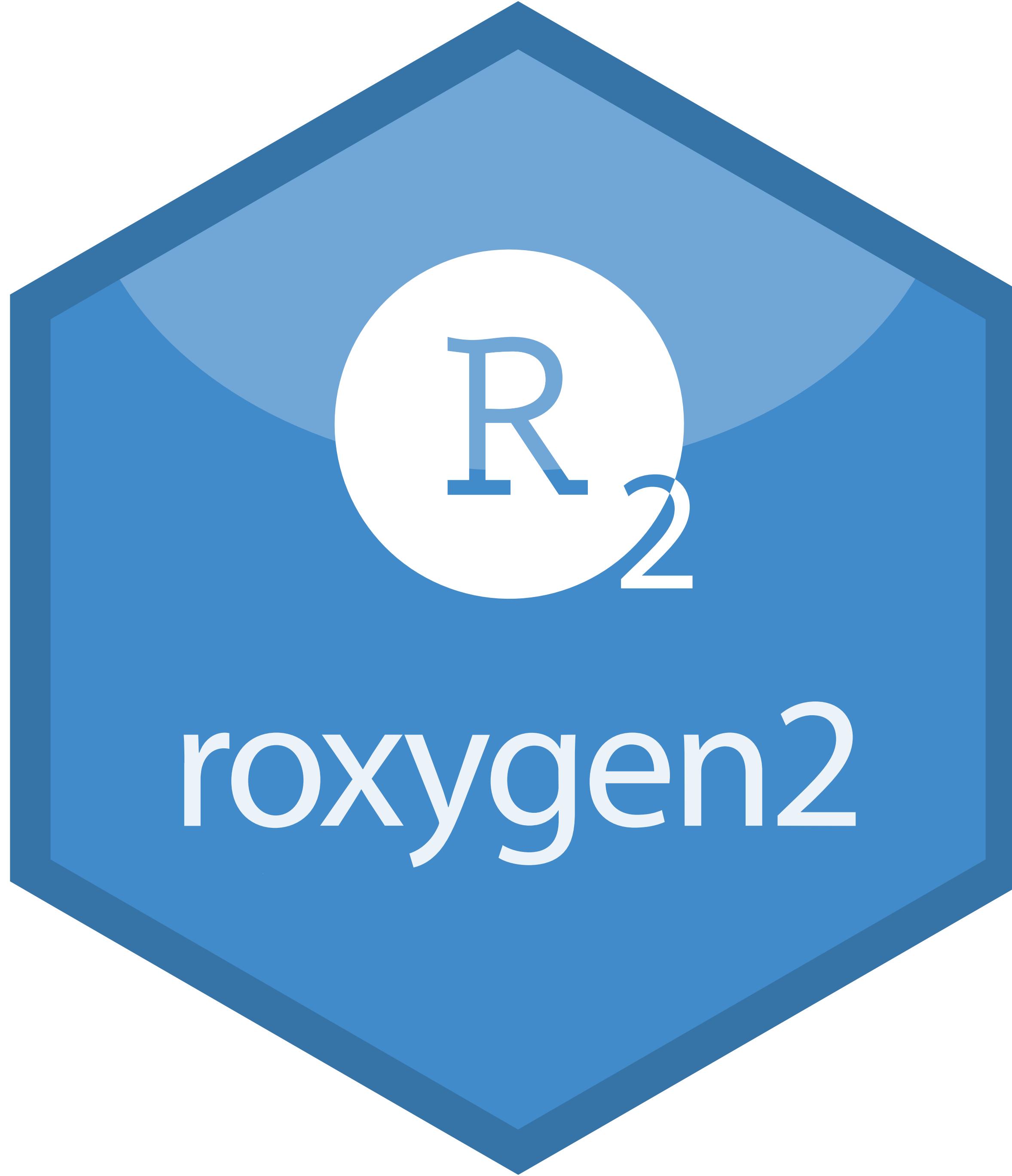
```
options(  
  warnPartialMatchArgs = TRUE,  
  warnPartialMatchDollar = TRUE,  
  warnPartialMatchAttr = TRUE  
)
```

Documentation



Documentation





roxygen2



📁 .github
📁 R
📁 inst
📁 man
📁 pkgdown/favicon
📁 revdep
📁 tests
📁 vignettes
📄 .Rbuildignore
📄 .covrignore
📄 .gitignore
📄 DESCRIPTION
📄 LICENSE
📄 LICENSE.md
📄 NAMESPACE
📄 NEWS.md
📄 README.Rmd
📄 README.md
📄 _pkgdown.yml

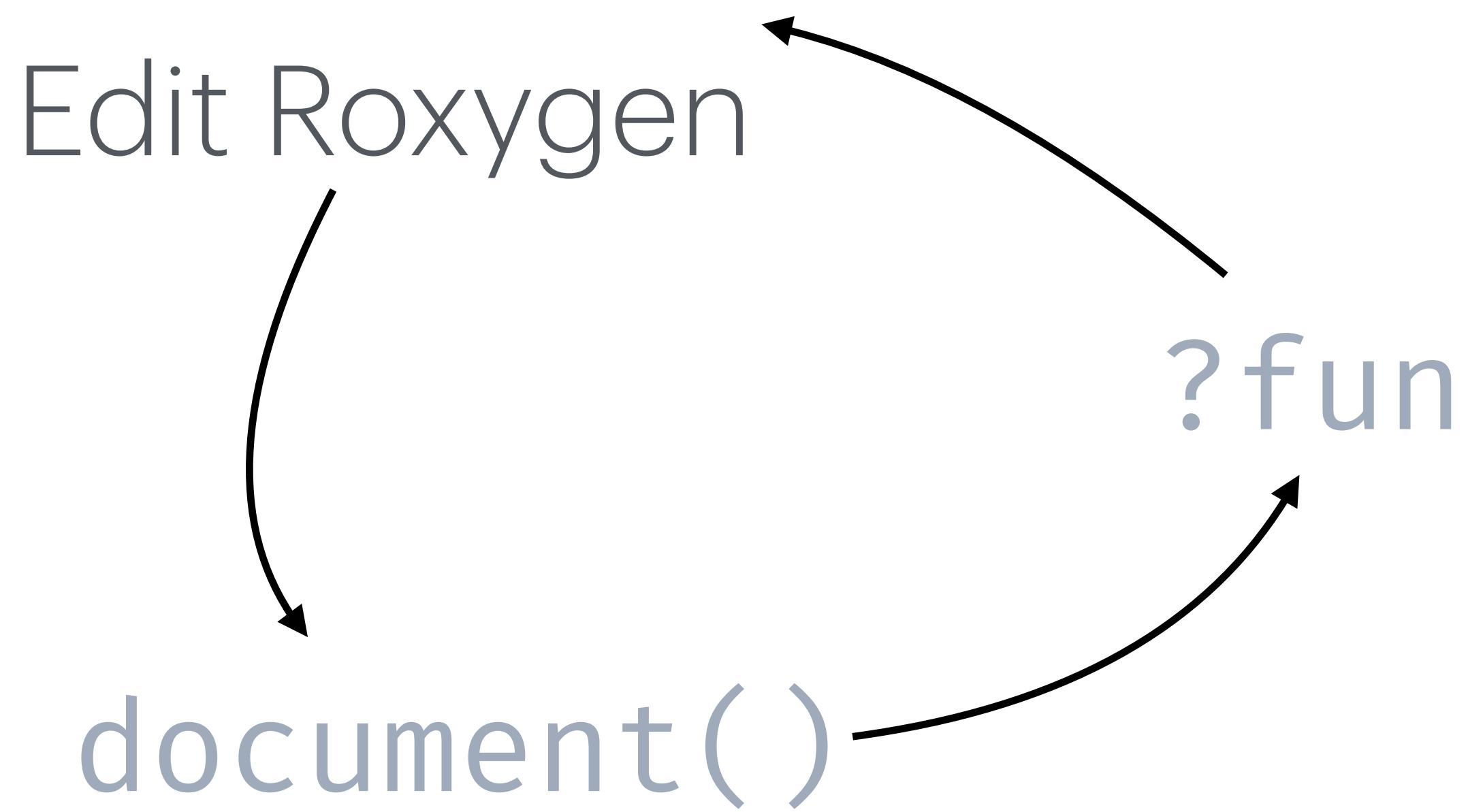
Help topics live here, in `.Rd` files

But you (mostly) don't write to `man/` with your bare hands.

📁 .github
📁 R
📁 inst
📁 man
📁 pkgdown/favicon
📁 revdep
📁 tests
📁 vignettes
📄 .Rbuildignore
📄 .covrignore
📄 .gitignore
📄 DESCRIPTION
📄 LICENSE
📄 LICENSE.md
📄 NAMESPACE
📄 NEWS.md
📄 README.Rmd
📄 README.md
📄 _pkgdown.yml

Help topics are generated from the roxygen comments you write alongside your functions (or other objects), inside R/.

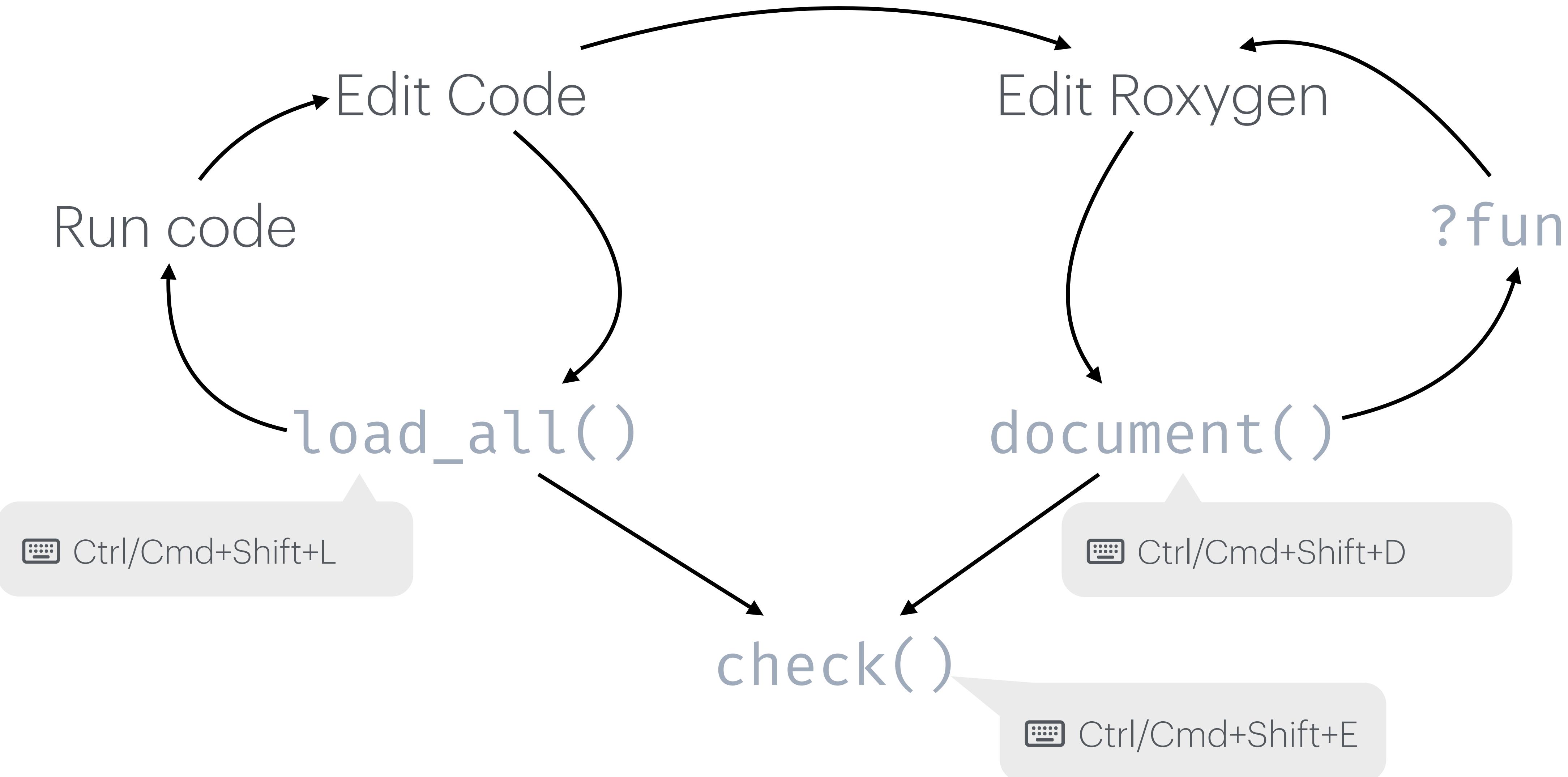
Documentation workflow



Ctrl+Shift+D (Windows & Linux)

Workflow

Code + documentation + check



📁 .github
📁 R
📁 inst
📁 man
📁 pkgdown/favicon
📁 revdep
📁 tests
📁 vignettes
📄 .Rbuildignore
📄 .covrignore
📄 .gitignore
📄 DESCRIPTION
📄 LICENSE
📄 LICENSE.md
📄 NAMESPACE
📄 NEWS.md
📄 README.Rmd
📄 README.md
📄 _pkgdown.yml

README.md is important:

- on CRAN
- on GitHub
- in the pkgdown site (it's the default **index.html**)

Ideally, **README.md** is generated from **README.Rmd**.

build_readme() is the best way to convert **README.Rmd** to **README.md**, because it actually installs the current dev package prior to rendering.

📁 .github
📁 R
📁 inst
📁 man
📁 pkgdown/favicon
📁 revdep
📁 tests
📁 vignettes
📄 .Rbuildignore
📄 .covrignore
📄 .gitignore
📄 DESCRIPTION
📄 LICENSE
📄 LICENSE.md
📄 NAMESPACE
📄 NEWS.md
📄 README.Rmd
📄 README.md
📄 _pkgdown.yml

Vignettes are a great way to show truly authentic usage, combining multiple functions in your package to accomplish a realistic task.

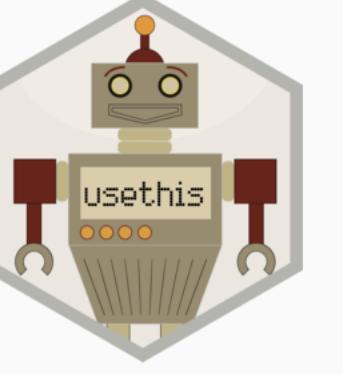
If authentic usage involves moves that are impossible on CRAN, demonstrate that in an **article** instead. An article appears only in the pkgdown site.

	.github
	R
	inst
	man
	pkgdown/favicon
	revdep
	tests
	vignettes
	.Rbuildignore
	.covignore
	.gitignore
	DESCRIPTION
	LICENSE
	LICENSE.md
	NAMESPACE
	NEWS.md
	README.Rmd
	README.md
	_pkgdown.yml

A user's experience with the `pkgdown` website is top of my mind when writing docs.

usethis 3.0.0 Setup Reference Articles ▾ News ▾

Search for 

 usethis

usethis is a workflow package: it automates repetitive tasks that arise during project setup and development, both for R packages and non-package projects.

Installation

Install the released version of usethis from CRAN:

```
install.packages("usethis")
```

LINKS

[View on CRAN](#)
[Browse source code](#)
[Report a bug](#)

LICENSE

[Full license](#)
[MIT + file LICENSE](#)

COMMUNITY

[Contributing guide](#)
[Code of conduct](#)
[Getting help](#)

CITATION

Items Needed for a help topic

- Insert a roxygen skeleton in RStudio with *Code > Insert Roxygen Skeleton*. Or do the equivalent "by hand".
- Complete the prepared fields
 - (Usually implicit) title and description
 - **@param** for function arguments
 - **@returns** for the return value
 - **@examples** for usage
 - **@export** to put in NAMESPACE
- Related workflow happiness: Cmd/Ctrl + Enter helps you develop your examples. Plays nicely with `load_all()` for staying synced with package source.

Typical roxygen comment

```
#' Remove duplicated strings  
#'  
#' `str_unique()` removes duplicated  
#' how duplication is measured.  
#'  
  
#' @param string Input vector. Either a character vector, or something  
#' coercible to one.  
#' @param ... Other options used to control matching behavior between duplicate  
#'   strings. Passed on to [stringi::stri_opts_collator()].  
#' @returns A character vector, usually shorter than `string`.  
#' @seealso [unique()], [stringi::stri_unique()] which this function wraps.  
#' @examples  
#' str_unique(c("a", "b", "c", "b", "a"))  
#'  
  
#' # Use ... to pass additional arguments to stri_unique()  
#' str_unique(c("motley", "mötley", "pinguino", "pingüino"))  
#' str_unique(c("motley", "mötley", "pinguino", "pingüino"), strength = 1)  
#' @export  
str_unique <- function(string, ...) {  
  ...  
}
```

Title

Description

Parameters

What it returns

Examples

Hard and/or fiddly but worth it

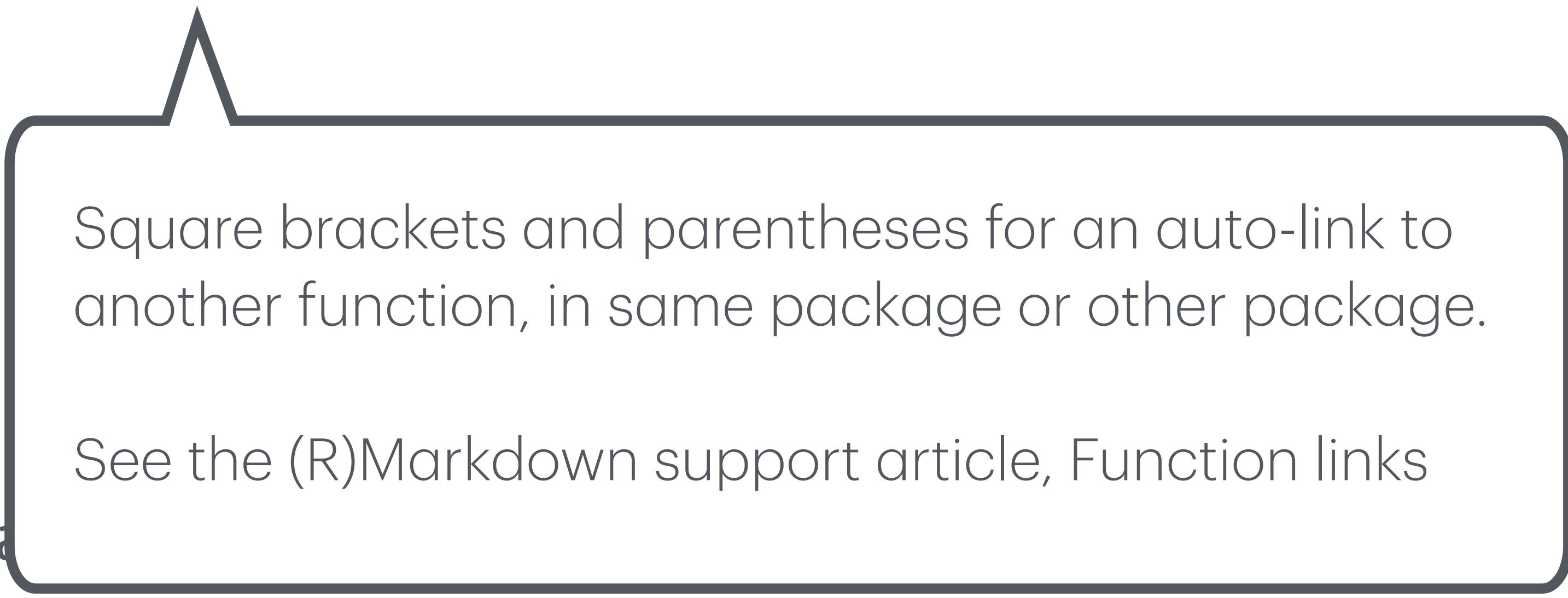
- Links!
 - Other help topics in same package
 - Topics in other packages
 - Vignettes in same or other package
 - Other URLs
- Examples
 - Show authentic usage that hopefully does not ...
 - Create huge headaches in CI and on CRAN

Backticks to format as code. Typical way to refer to a function in its own topic. Use trailing parentheses.

```
#' It's obvious that `thisfunction()` is better than  
#' [otherpkg::otherfunction()] or even our own [olderfunction()].
```

```
#' Read more in our own `vignette("stuff")` or elsewhere in  
#' `vignette("things", package = "otherpkg")`.
```

```
#' It's obvious that `thisfunction()` is better than  
#' [otherpkg::otherfunction()] or even our own [olderfunction()].
```



Square brackets and parentheses for an auto-link to another function, in same package or other package.

See the (R)Markdown support article, Function links

```
#' Read  
#' `vignette("things", package = "otherpkg")`.  
elsewhere in
```

```
#' It's obvious that `thisfunction()` is better than  
#' [otherpkg::otherfunction()] or even our own [olderfunction()].
```

This vignette syntax is auto-linked in pkgdown (and elsewhere) and gives working code, otherwise.

```
#' Read more in our own `vignette("stuff")` or elsewhere in  
#' `vignette("things", package = "otherpkg")`.
```

URL links

```
#' Use C++ via the cpp11 package  
#'  
#' Adds infrastructure needed to use the [cpp11](https://cpp11.r-lib.org)  
#' package, a header-only R package that helps R package developers handle R  
#' objects with C++ code: ...
```

[link text](URL)

```
#' See <https://happygitwithr.com/common-remote-setups.html> for more about  
#' GitHub remote configurations and, e.g., what we mean by the source repo. This  
#' function works for the configurations ` "ours" ` , ` "fork" ` , and ` "theirs" ` .
```

<URL>

URL links

- "The goal of urlchecker is to run the URL checks from R 4.1 in older versions of R and automatically update URLs as needed."
- Extracts the URL-checking logic inside R CMD check and exposes as `urlchecker :: url_check()`.

```
library(urlchecker)
```

```
# `url_check()` will check all URLs in a package, as is done by CRAN when  
# submitting a package.  
url_check("path/to/pkg")
```

```
# `url_update()` will check all URLs in a package, then update any 301  
# redirects automatically to their new location.  
url_update("path/to/pkg")
```

What's tricky about examples?

- Tension between
 - Showing readable and realistic code
 - However, there's
 - no user, i.e. for interaction or credentials
 - side effects are forbidden
 - can't throw an error
- It's tempting to use `\dontrun{ }` but you can get pushback from CRAN

@examplesIf is a mighty weapon

Put something that evaluates to TRUE/FALSE here.

```
#' @examplesIf rlang::is_interactive()  
#' # load/refresh existing credentials, if available  
#' # otherwise, go to browser for authentication and authorization  
#' drive_auth()  
#'  
#' # see user associated with current token  
#' drive_user()
```

@examples If is a mighty weapon

Examples

```
if (FALSE) { # rlang::is_interactive()
  # load/refresh existing credentials, if available
  # otherwise, go to browser for authentication and authorization
  drive_auth()

  # see user associated with current token
  drive_user()
```

Examples should not change the world

- If at all possible, just don't do anything like this:
 - write a file
 - set an option
 - change working directory
- If you must change the world, put it back the way you found it, e.g.
 - write to the temp directory AND delete the file
 - restore the option or working directory to original state
- Sadly, `withr` functions don't work here. Neither does `on.exit()`.
- Sadly, nothing like `@examplesIf` is available to hide the machinery.

Use `try()` to show an error

```
#' @examples  
#' ...  
#' # Row sizes must be compatible when column-binding  
#' try(bind_cols(tibble(x = 1:3), tibble(y = 1:2)))
```

Put code that errors inside `try()`.

```
# Row sizes must be compatible when column-binding  
try(bind_cols(tibble(x = 1:3), tibble(y = 1:2)))  
#> Error in bind_cols(tibble(x = 1:3), tibble(y = 1:2)) :  
#>   Can't recycle `..1` (size 3) to match `..2` (size 2).
```

Create roxygen comments

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

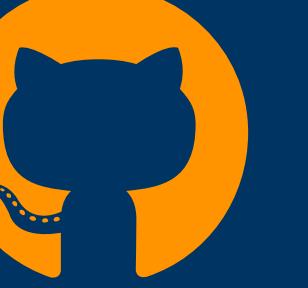
⌨️ Ctrl+.

(Start typing function name...)

⌨️ Cmd/Ctrl+Alt+Shift+R

⌨️ Cmd/Ctrl+Shift+D

➡️ Your Turn

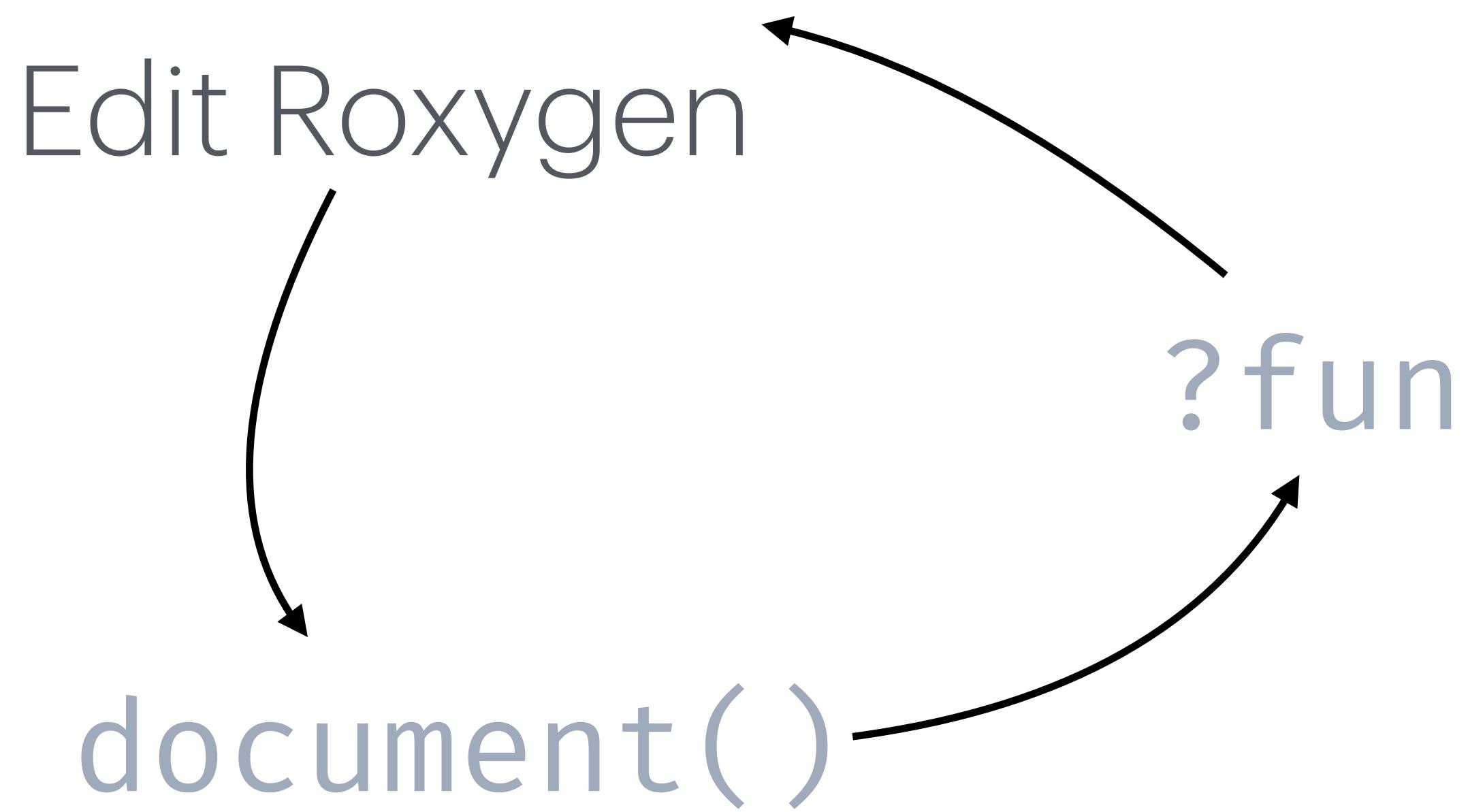
check() 
Commit your changes 
Push to GitHub 

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.

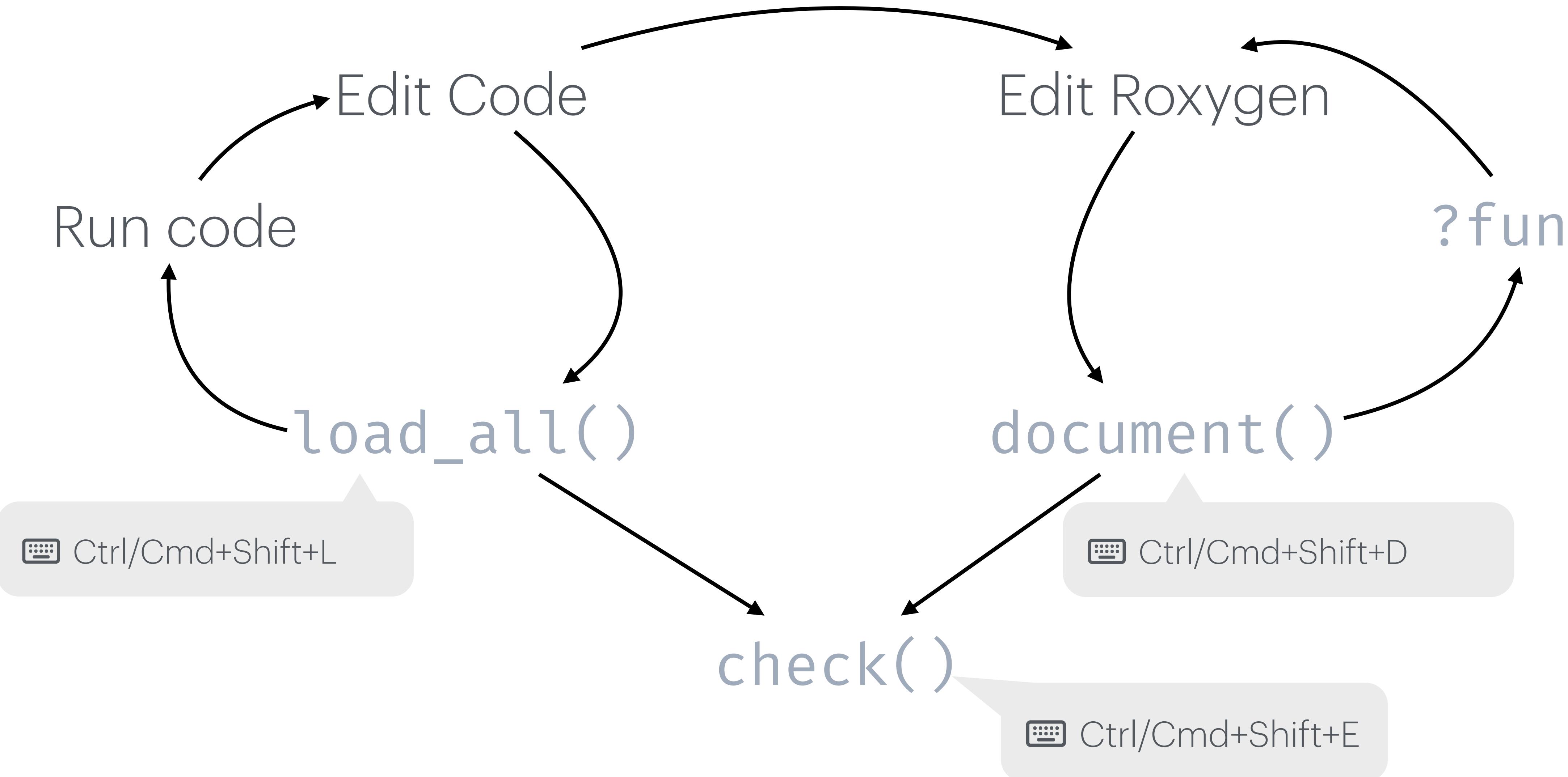
Documentation workflow



Ctrl+Shift+D (Windows & Linux)

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 ✅
```

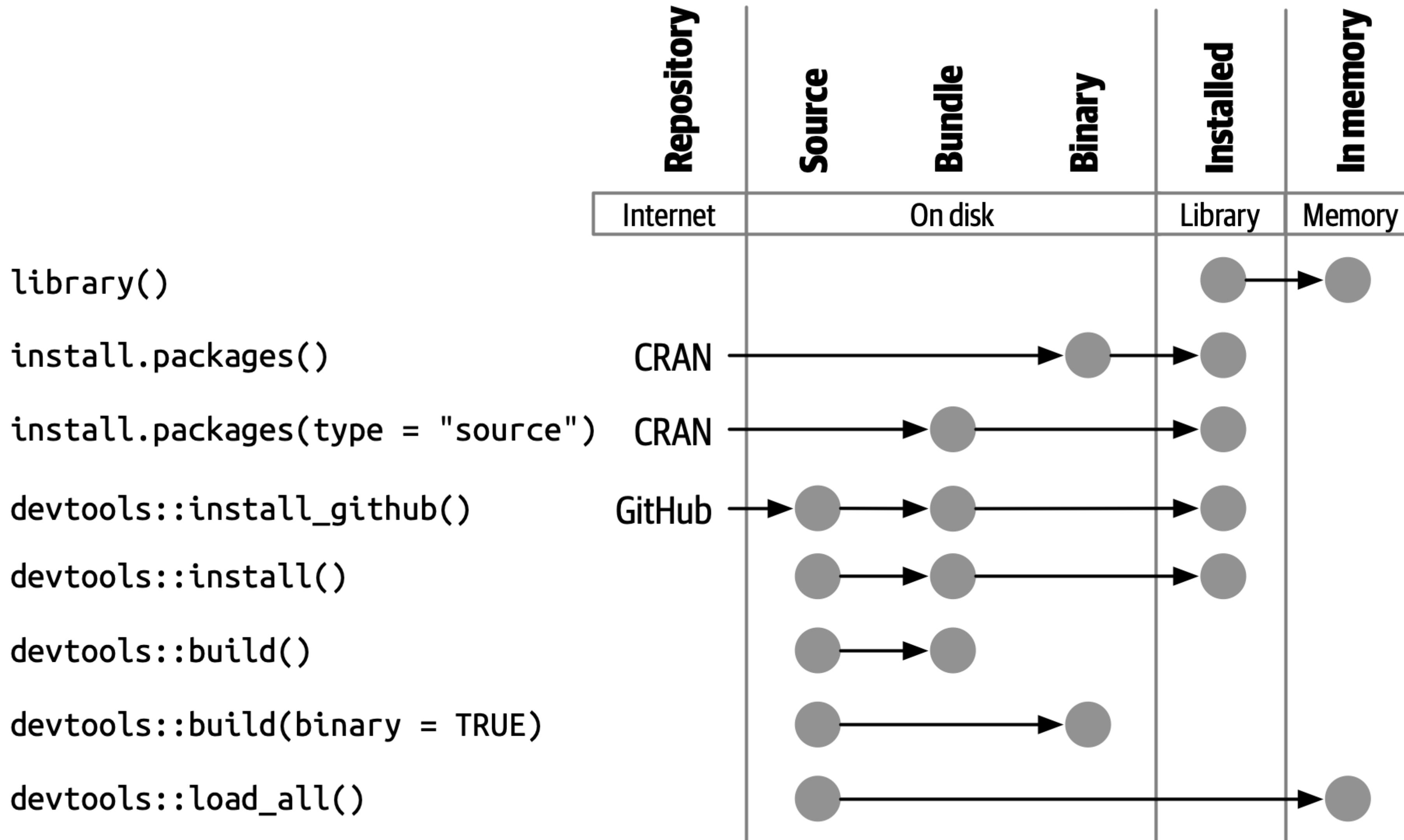
install()

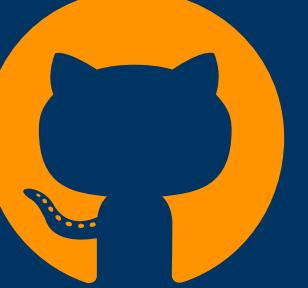
Install package to your library

- R CMD INSTALL  Ctrl+Shift+B (Windows & Linux)
- Restart R  Ctrl+Shift+F10 (Windows & Linux)
- Attach package with **library()** like any other package

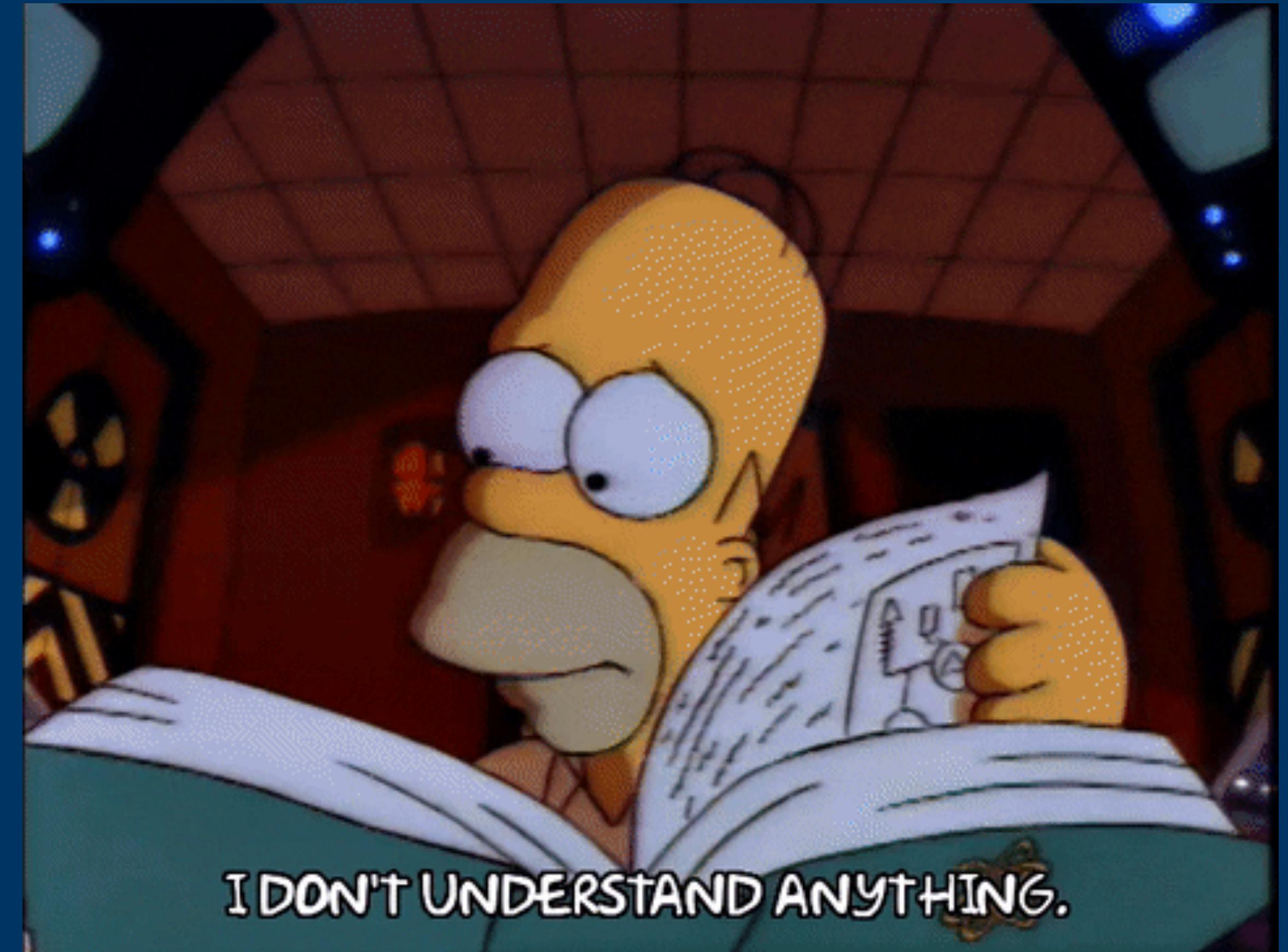
→ Your Turn

Package Structure and State



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!

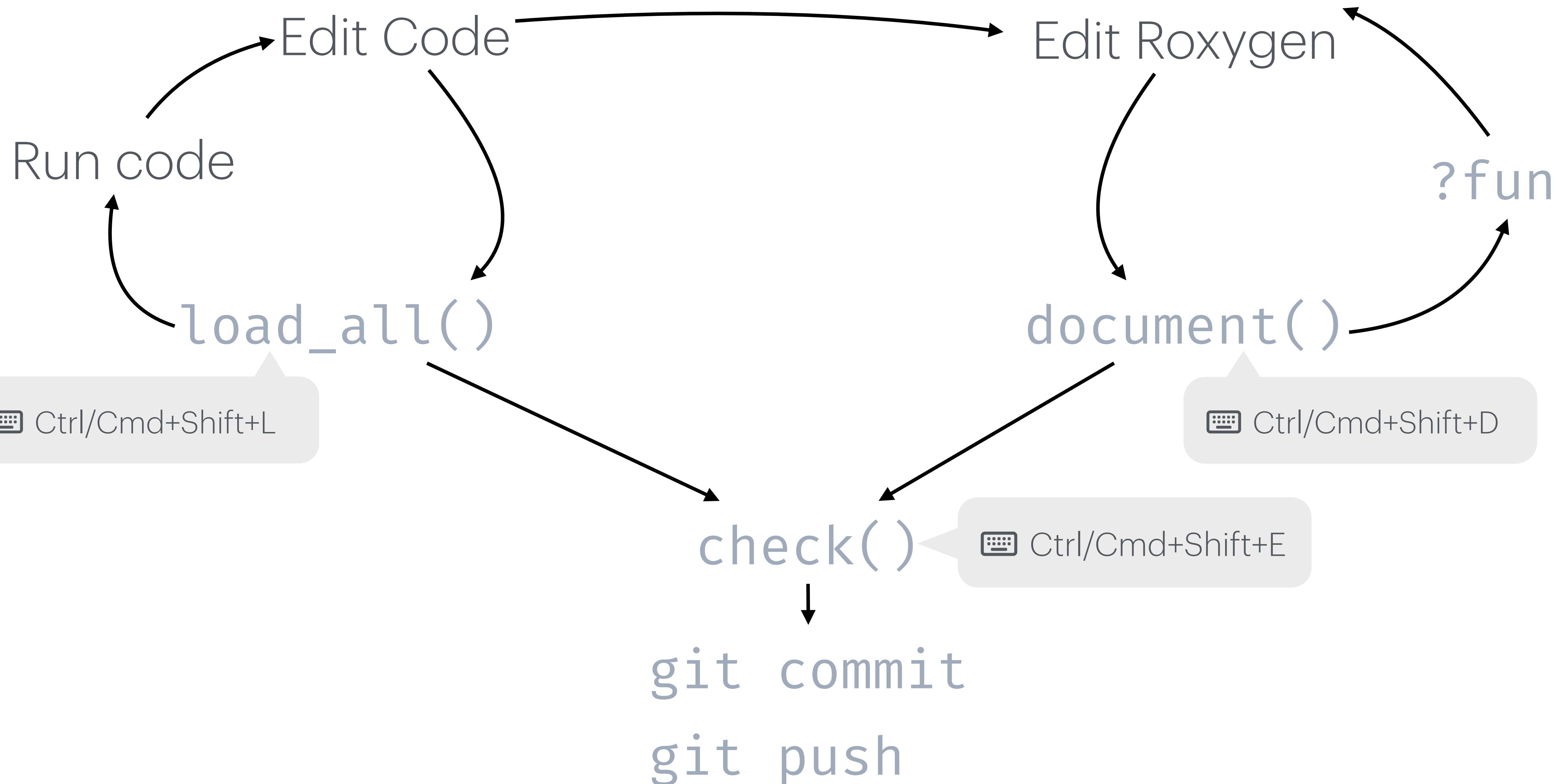
```
check()
```

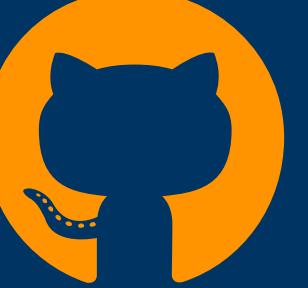
```
#> — R CMD check results —————  
#> Duration: 3.1s  
#>  
#> 0 errors ✓ | 0 warnings ✓ | 0 notes
```

```
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 

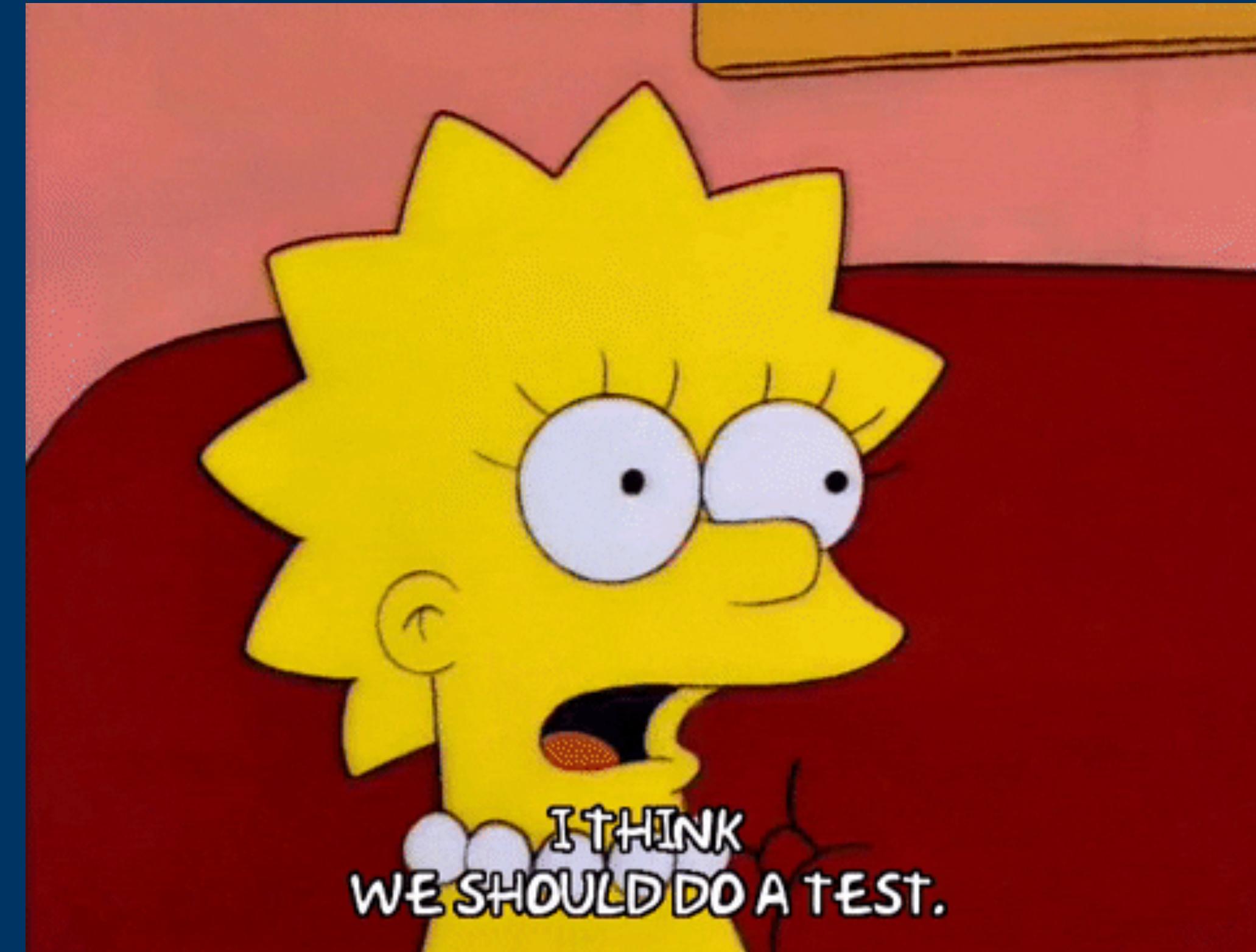
Break Time!



Day one complete!

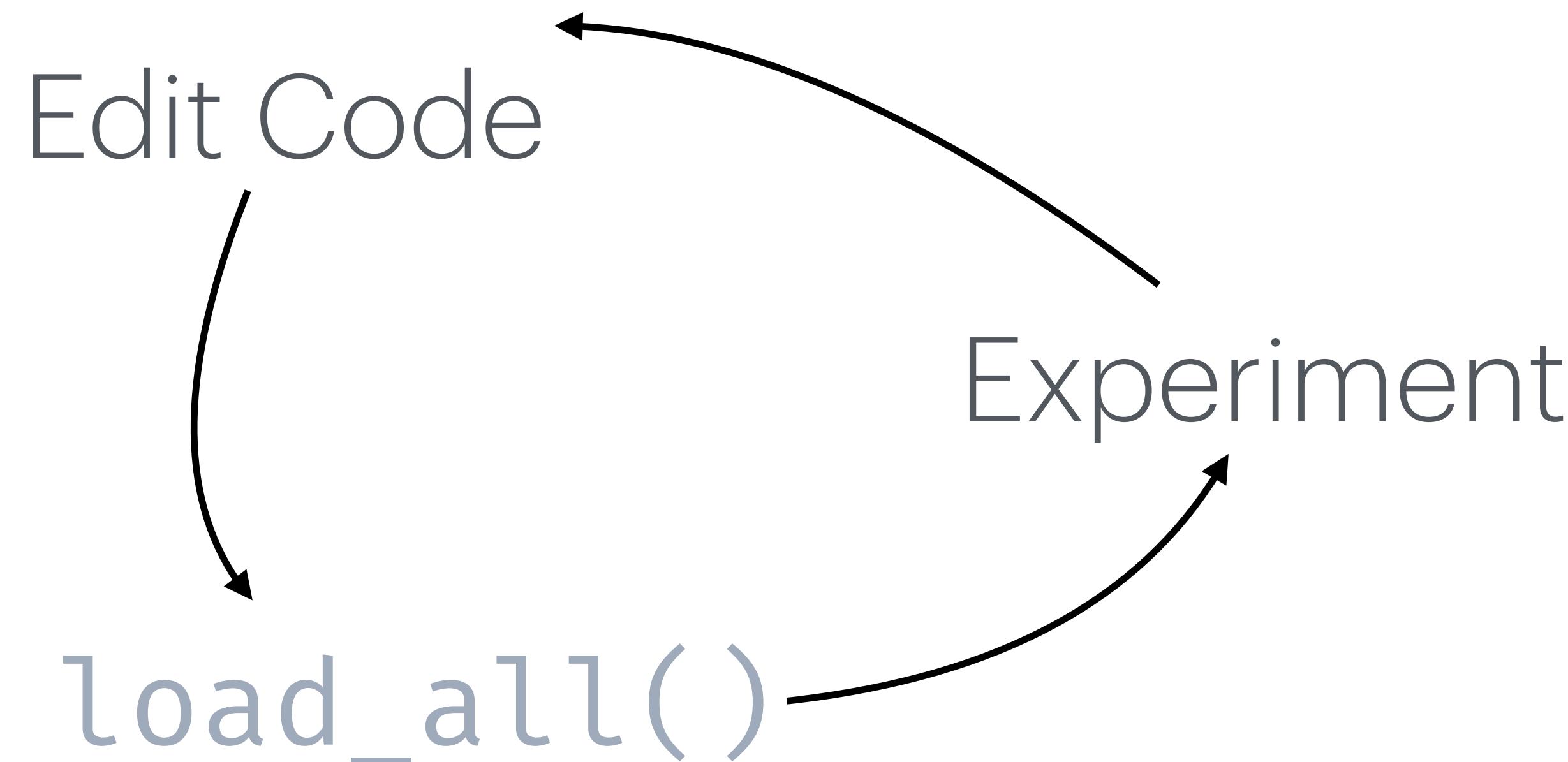
Welcome back to day two

Testing



Testing

Current workflow



Ctrl+Shift+L (Windows & Linux)

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
```

use_test() + use_r() vibe with file pairs

R/a.R	tests/testthat/test-a.R
R/b.R	tests/testthat/test-b.R
R/c.R	tests/testthat/test-c.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_*(...)`
 - Tests a specific computation and compares it to an expected value

Expectations

- **Objects**
 - `expect_equal()`, `expect_type()`, `expect_s3_class()`, ...
- **Vectors**
 - `expect_length()`, `expect_lt()`, `expect_gte()`, `expect_true()`,
`expect_contains()`, ...
- **Side-effects**
 - `expect_error()`, `expect_silent()`, `expect_no_warning()`,
`expect_output()`, ...
- **Snapshots**
 - `expect_snapshot()`, `expect_snapshot_value()`, ...

test()

- Runs all tests in your test suite

```
test()
```

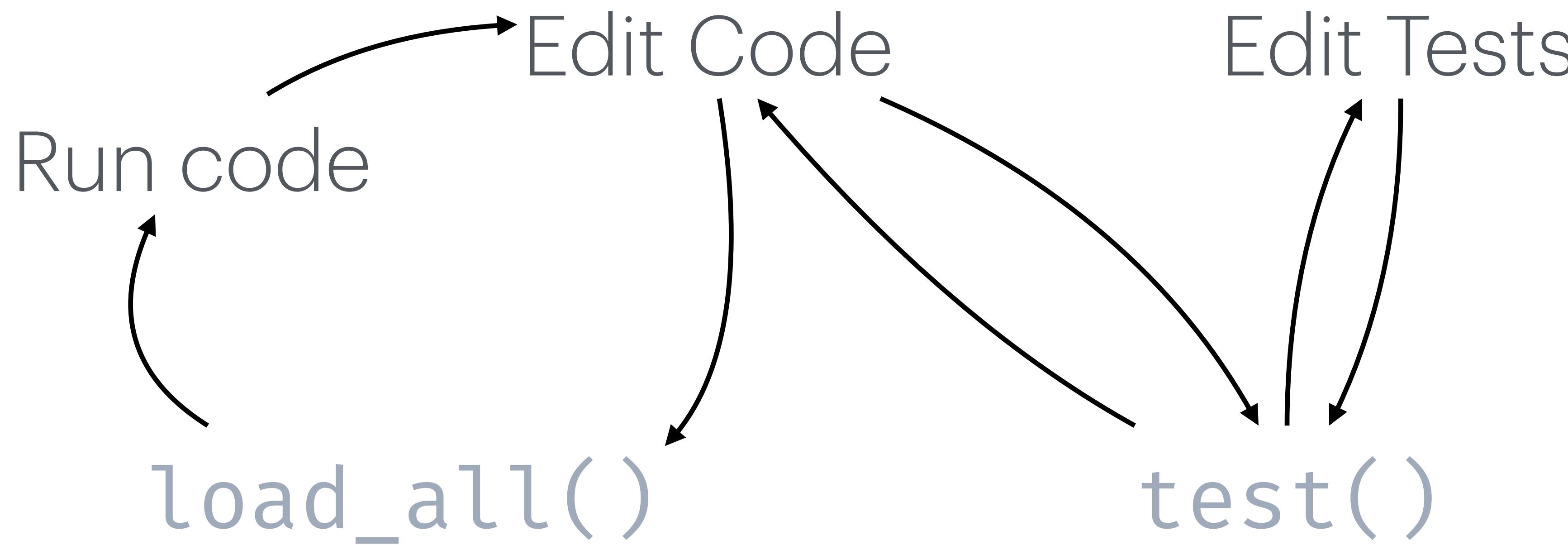
```
#> i Testing
#> ✓ | F W S  OK | Context
#>
#> :: |      0 |
#> ✓ |      1 |
#>
#> == Results =====
#> [ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
```

⌨️ Ctrl+Shift+T (Windows & Linux)

➡ Your Turn

Updated workflow

Code + testing

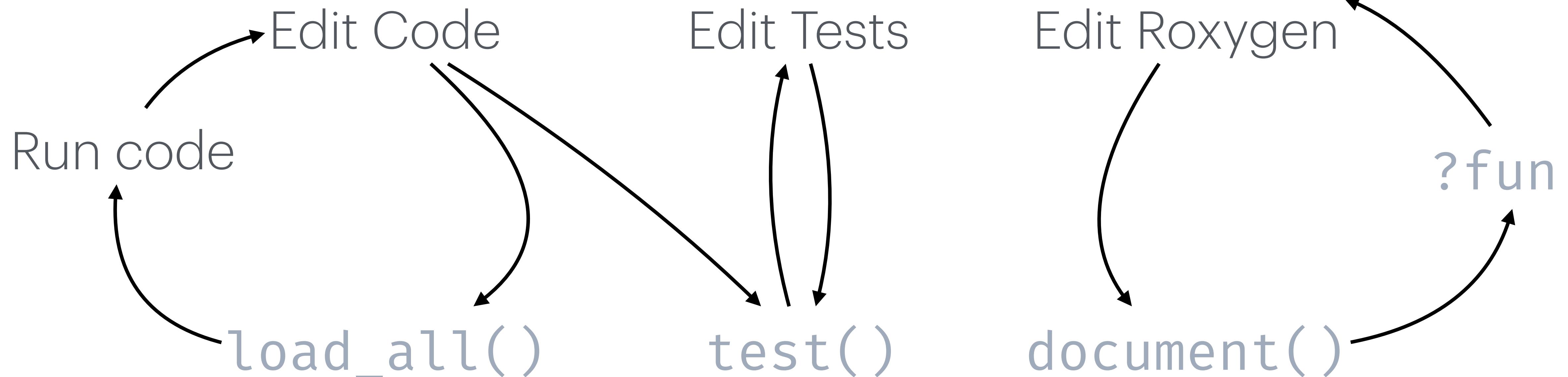


⌨️ Ctrl/Cmd+Shift+L

⌨️ Ctrl/Cmd+Shift+T

Workflow

Code + testing + documentation



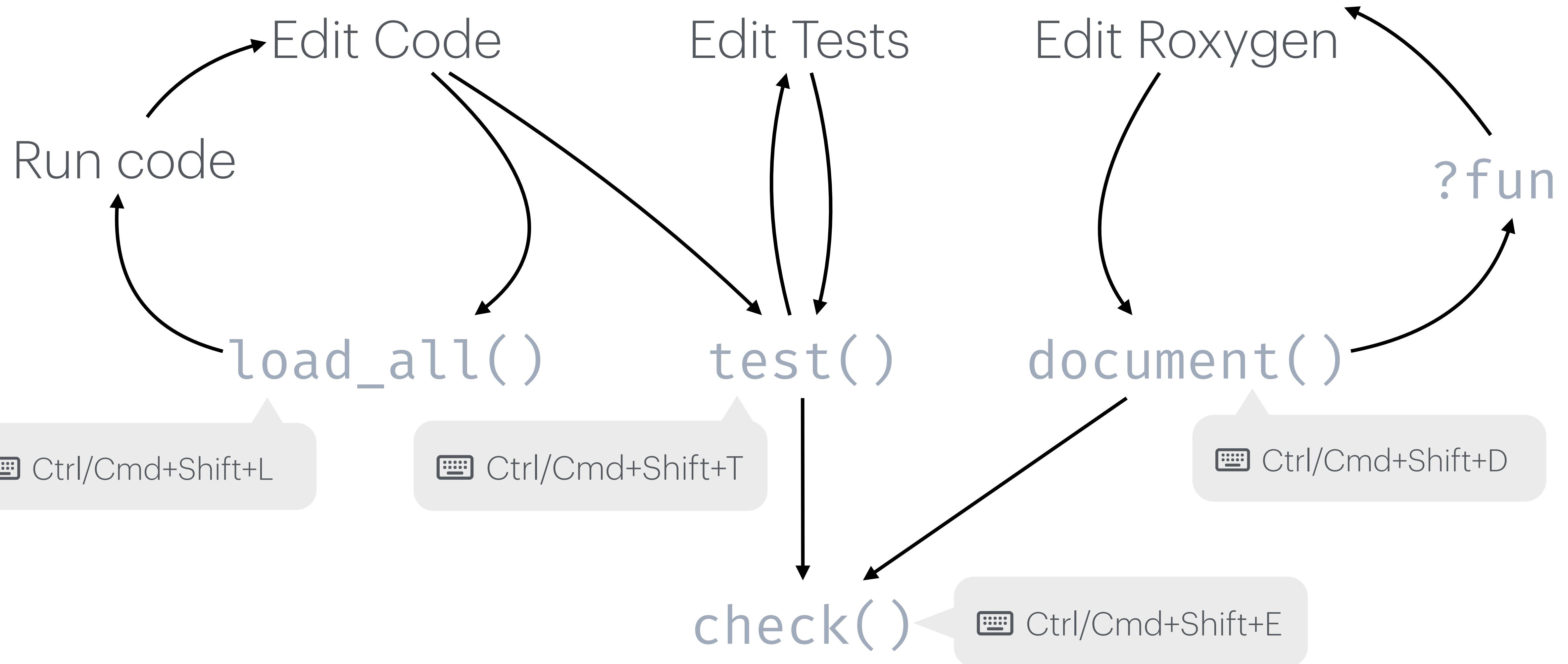
⌨️ Ctrl/Cmd+Shift+L

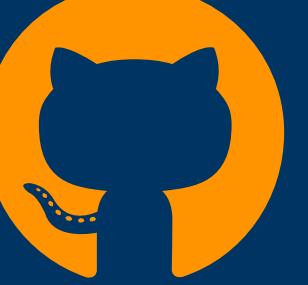
⌨️ Ctrl/Cmd+Shift+T

⌨️ Ctrl/Cmd+Shift+D

Workflow

Code + testing + documentation + check



check() 
Commit your changes 
Push to GitHub 

Review: functions

Run once

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

Run periodically

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

Run frequently

- `load_all()`

 Ctrl/Cmd+Shift+L

- `document()`

 Ctrl/Cmd+Shift+D

- `test()`

 Ctrl/Cmd+Shift+T

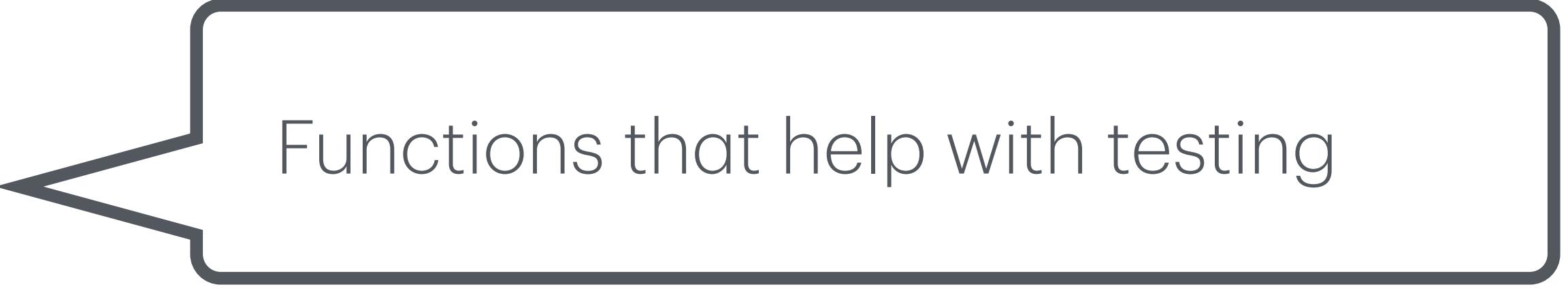
- `check()`

 Ctrl/Cmd+Shift+E

testthat workflow patterns

load_all()

- testthat's workflow is designed around `load_all()`
- Makes entire package namespace available
- Attaches testthat
- Sources `tests/testthat/helper.R`



Functions that help with testing

Workflow: micro-iteration, interactive experimentation

- tweak the `foofy()` function and re-load it with `load_all()`
 - also attaches `testthat` and sources any test helpers
- interactively explore and refine expectations
`expect_equal(foofy(...), EXPECTED_FOOFY_OUTPUT)`
- and tests
`test_that("foofy does good things", { ... })`

Workflow: mezzo-iteration, whole test file

```
test_file("tests/testthat/test-foofy.R")
```

- In RStudio, with test or R file focused

```
test_active_file()
```

```
test_coverage_active_file()
```

*Consider binding these to Cmd + T, Cmd + R

Keyboard Shortcuts

Show: All Customized

[Customizing Keyboard Shortcuts](#)

Name	Shortcut	Scope
Compare test results for Shiny application		Workbench
Record a test for Shiny		Workbench
Run shinytest Test		Workbench
Run tests for Shiny application		Workbench
Run testthat Tests		Workbench
Test Package	Shift+Cmd+T	Package Development
Calculate package test coverage		Addin
Calculate package test coverage		Addin
Format test_that test file		Addin
Initialize test_that()		Addin
Navigate To Test File		Addin
Report test coverage for a file		Addin
Report test coverage for a file	Cmd+R	Addin
Report test coverage for a package		Addin
Report test coverage for a package	Shift+Cmd+R	Addin
Run a test file		Addin
Run a test file	Cmd+T	Addin
View Latest Run		Addin

Workflow: macro-iteration, all files

test()

⌨️ Ctrl/Cmd+Shift+T

test_coverage()

⌨️ Ctrl/Cmd+Shift+R

check()

⌨️ Ctrl/Cmd+Shift+E

➡️ Your Turn

Test suite design principles

- A test should be self-sufficient and self-contained.
- The interactive workflow is important.
- Obvious >>> DRY
- Don't let a nonstandard workflow "leak".

Test smell: top-level code that's outside `test_``that()`

```
dat <- data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))
```

```
skip_if(today_is_a_monday())
```

```
test_that("foofy() does this", {  
  expect_equal(foofy(dat), ...)  
})
```

```
dat2 <- data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))
```

```
skip_on_os("windows")
```

```
test_that("foofy2() does that", {  
  expect_snapshot(foofy2(dat, dat2))  
})
```

Deodorizing the previous example

```
test_that("foofy() does this", {  
  skip_if(today_is_a_monday())  
  
  dat ← data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))  
  
  expect_equal(foofy(dat), ...)  
})
```

Move file-scope logic to a narrower scope (as done here) or a broader scope (test setup and/or test helpers).

```
test_that("foofy() does that", {  
  skip_if(today_is_a_monday())  
  skip_on_os("windows")  
  
  dat ← data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))  
  dat2 ← data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))  
  
  expect_s3_class(foofy(dat, dat2), "data.frame")  
})
```

It's OK to repeat yourself!

Leave the world the way you found it

```
circumference <- function(r) {2 * pi * r}  
  
test_that("calculations work", {  
  withr::local_options(digits = 3)  
  expect_equal(circumference(r = 8), 50.3))  
})
```

withr's `local_*` functions are super useful for making changes that are scoped to a single `test_that()`.

Things to avoid in your tests

```
library(somedependency)
```

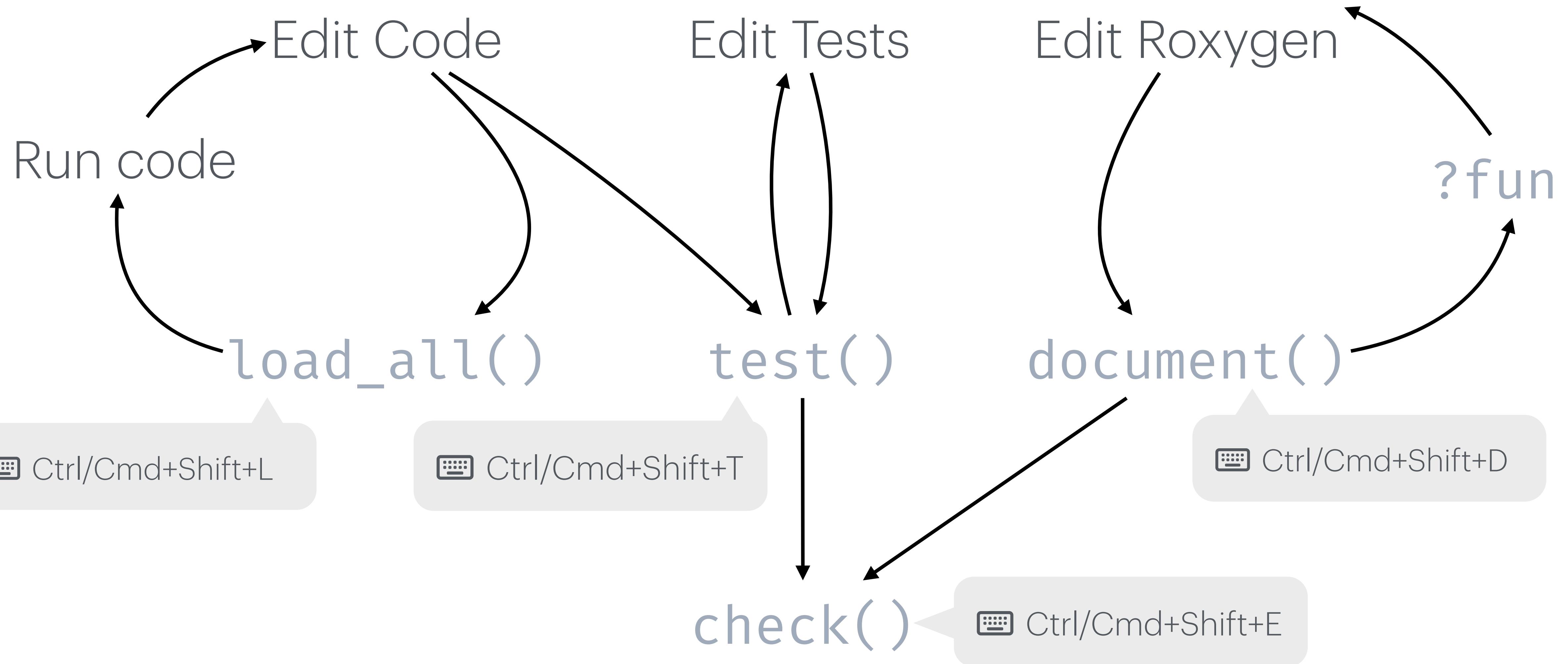
Please no! Access functions from your dependencies, in your tests, exactly as you do below R/.

```
source("stuff-thats-handy-for-your-tests.R")
```

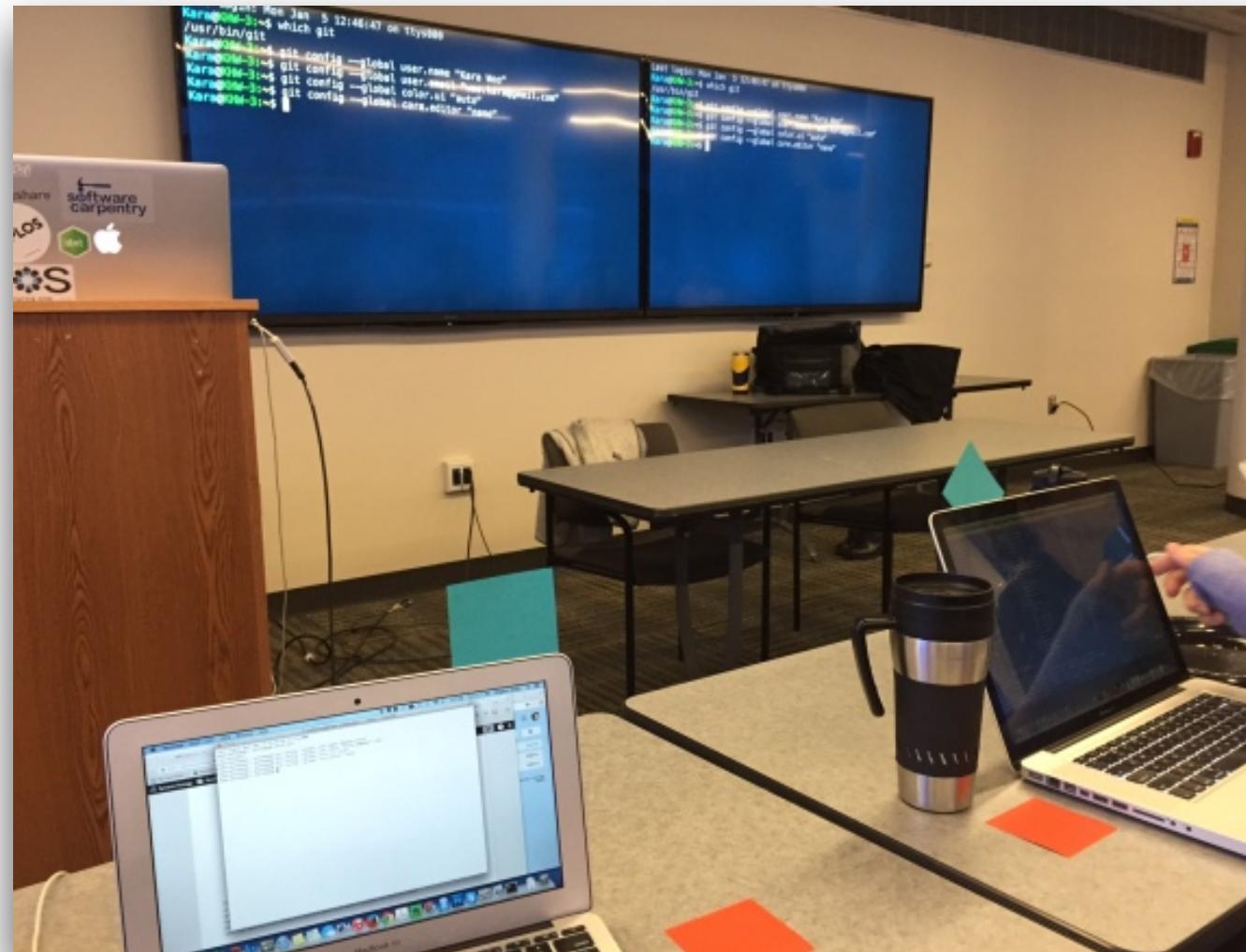
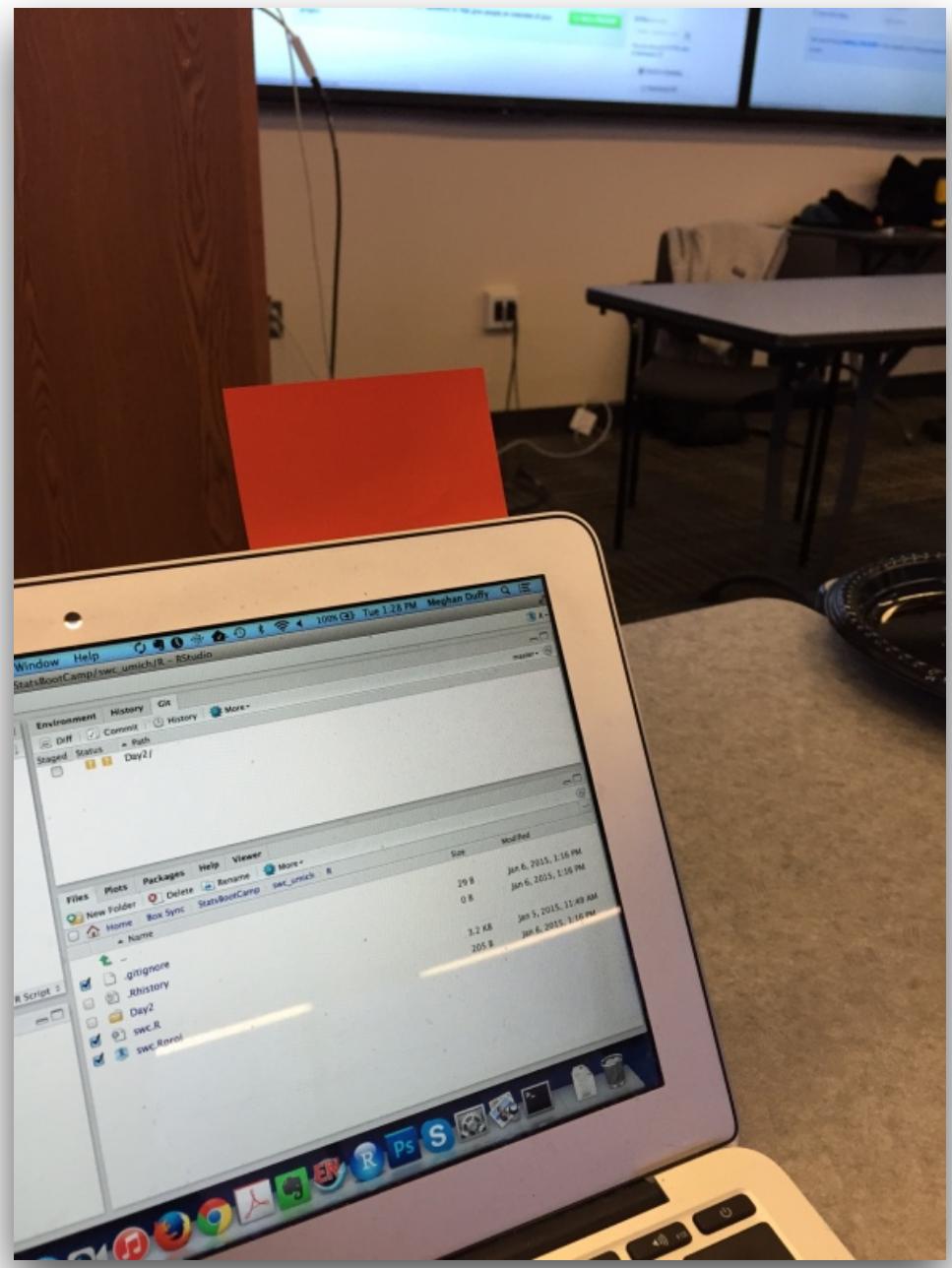
Please no! Unexported helper functions and test helper files are a better mechanism for this.

Workflow

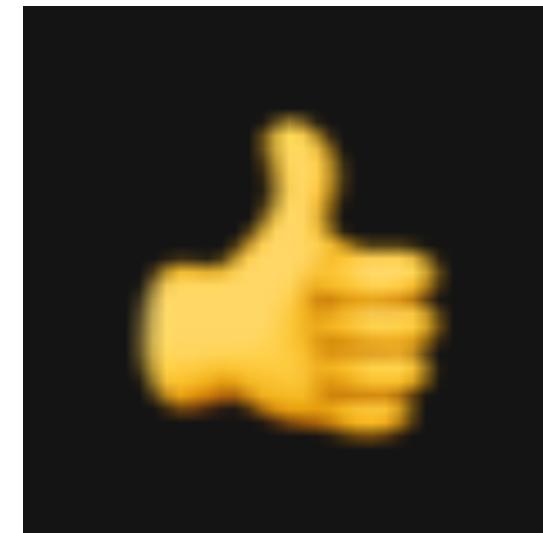
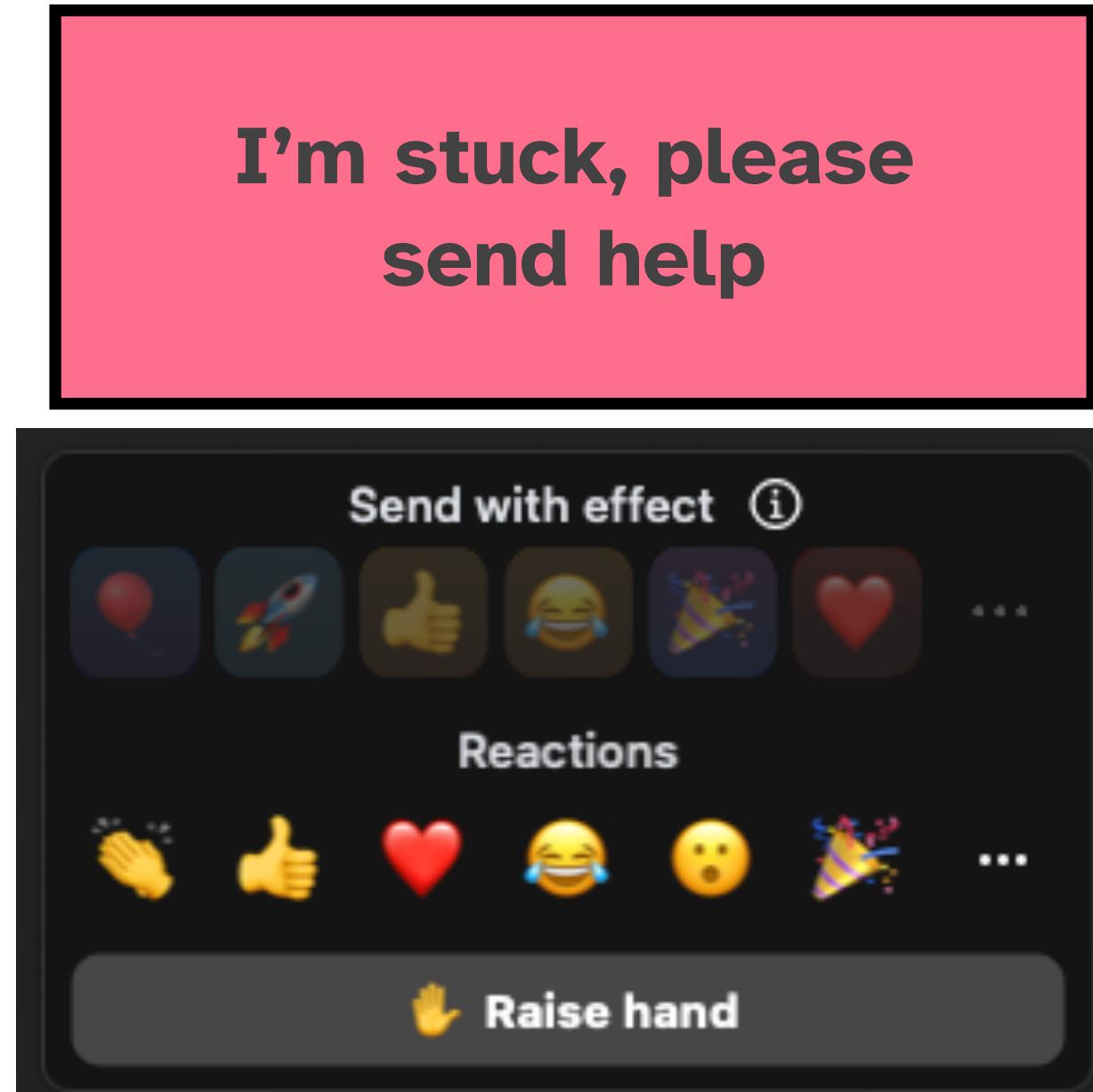
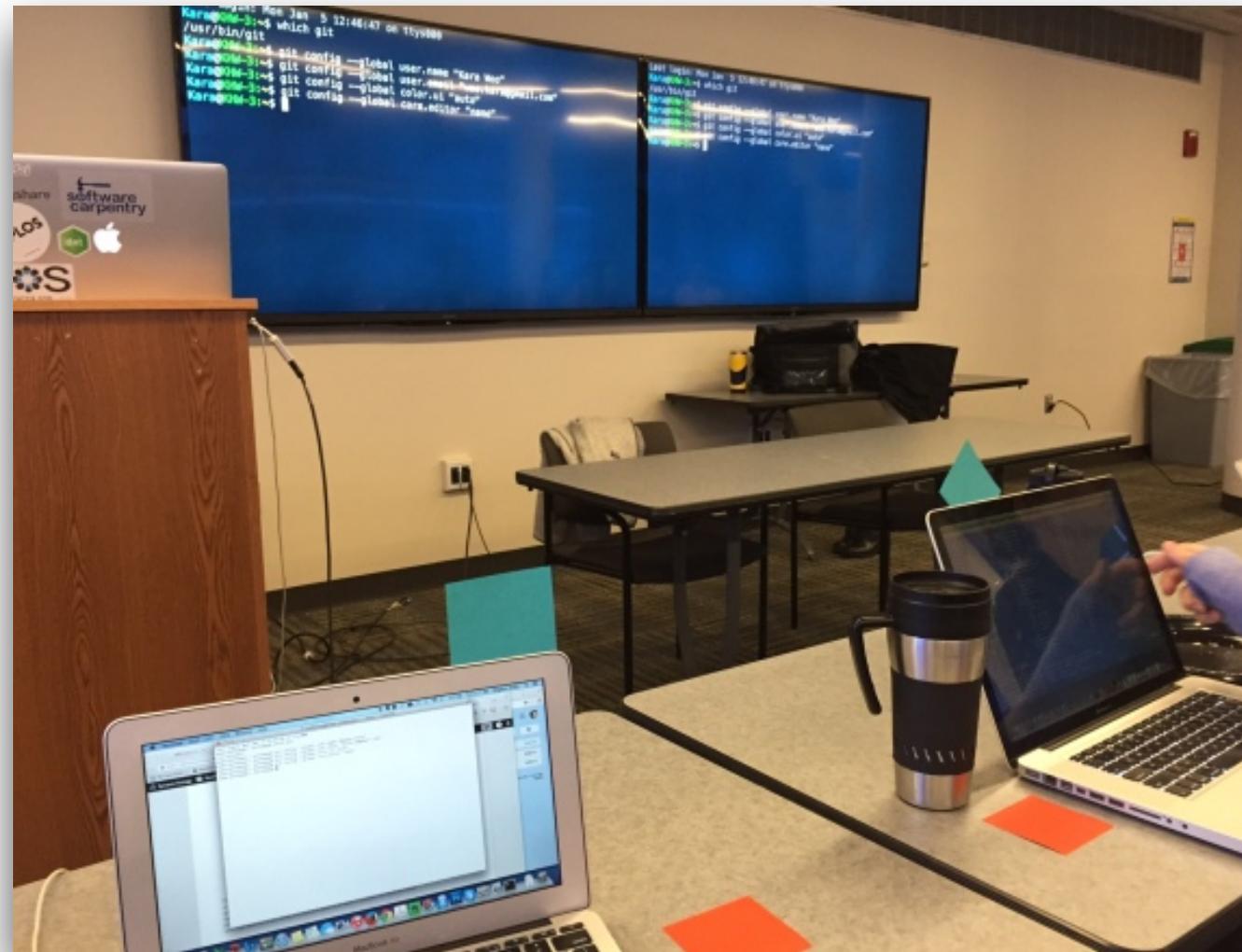
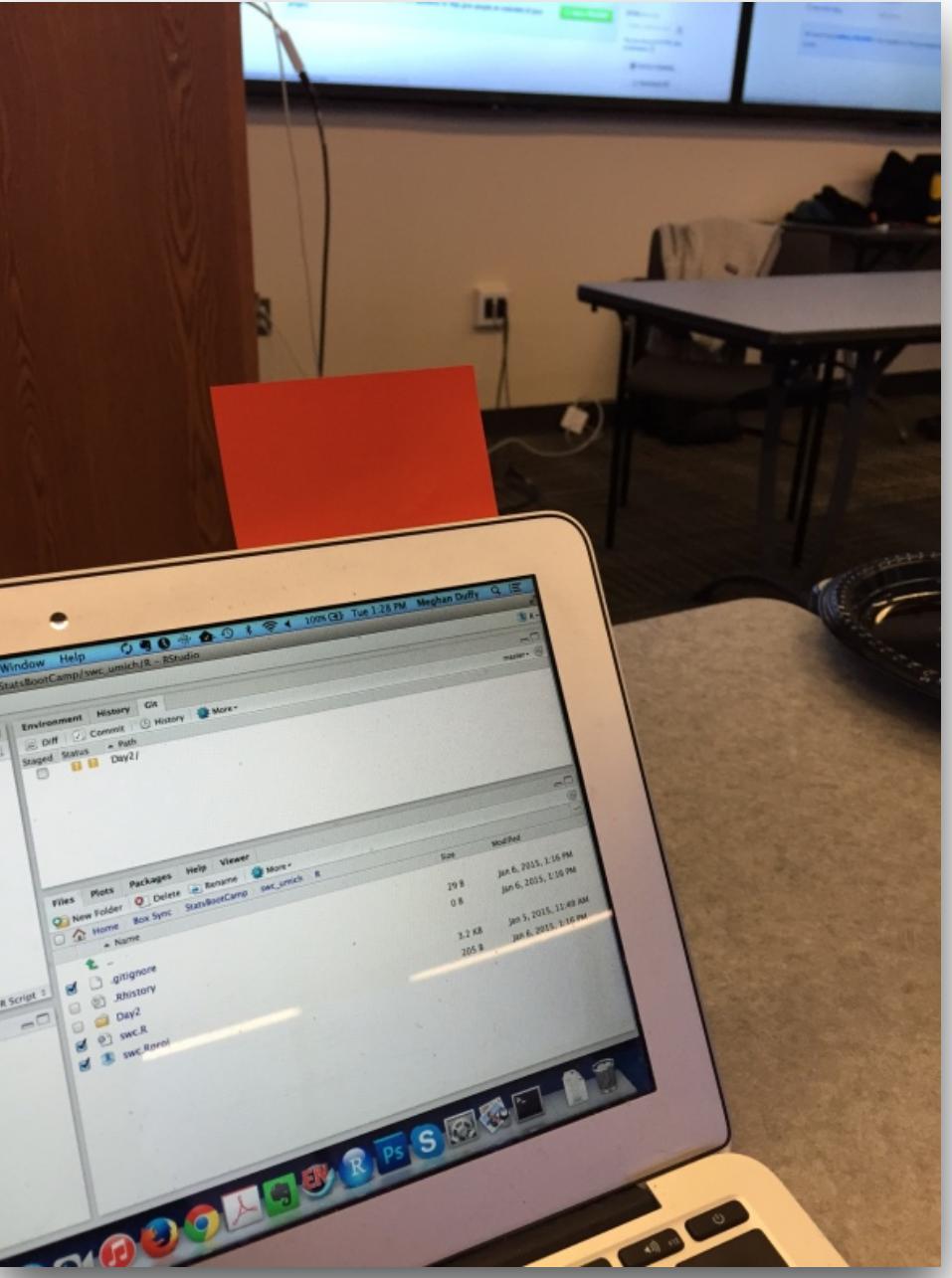
Code + testing + documentation + check



Sticky Notes



Sticky Notes



Day 2: Schedule and Learning Objectives

08:00 - 09:20	Dependencies and Continuous Integration	80 min
09:20 - 09:35	Break	15 min
09:35 - 10:55	Design Principles	80 min
10:55 - 11:10	Break	15 min
11:10 - 12:30	Website Creation and General Package Discussion	80 min

Review: functions

Run once

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

Run periodically

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

Run frequently

- `load_all()`

⌨️ Ctrl/Cmd+Shift+L

- `document()`

⌨️ Ctrl/Cmd+Shift+D

- `test()`

⌨️ Ctrl/Cmd+Shift+T

- `check()`

⌨️ Ctrl/Cmd+Shift+E

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
- **import** == Available to be used internally by your package
- Don't edit manually - use roxygen tags:

```
#' @importFrom pkg fun  
#' @import pkg
```

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

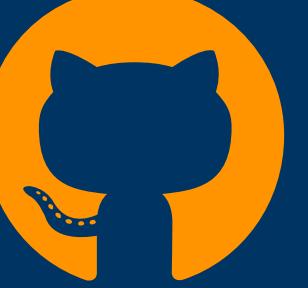
 Your Turn

Let's add one more

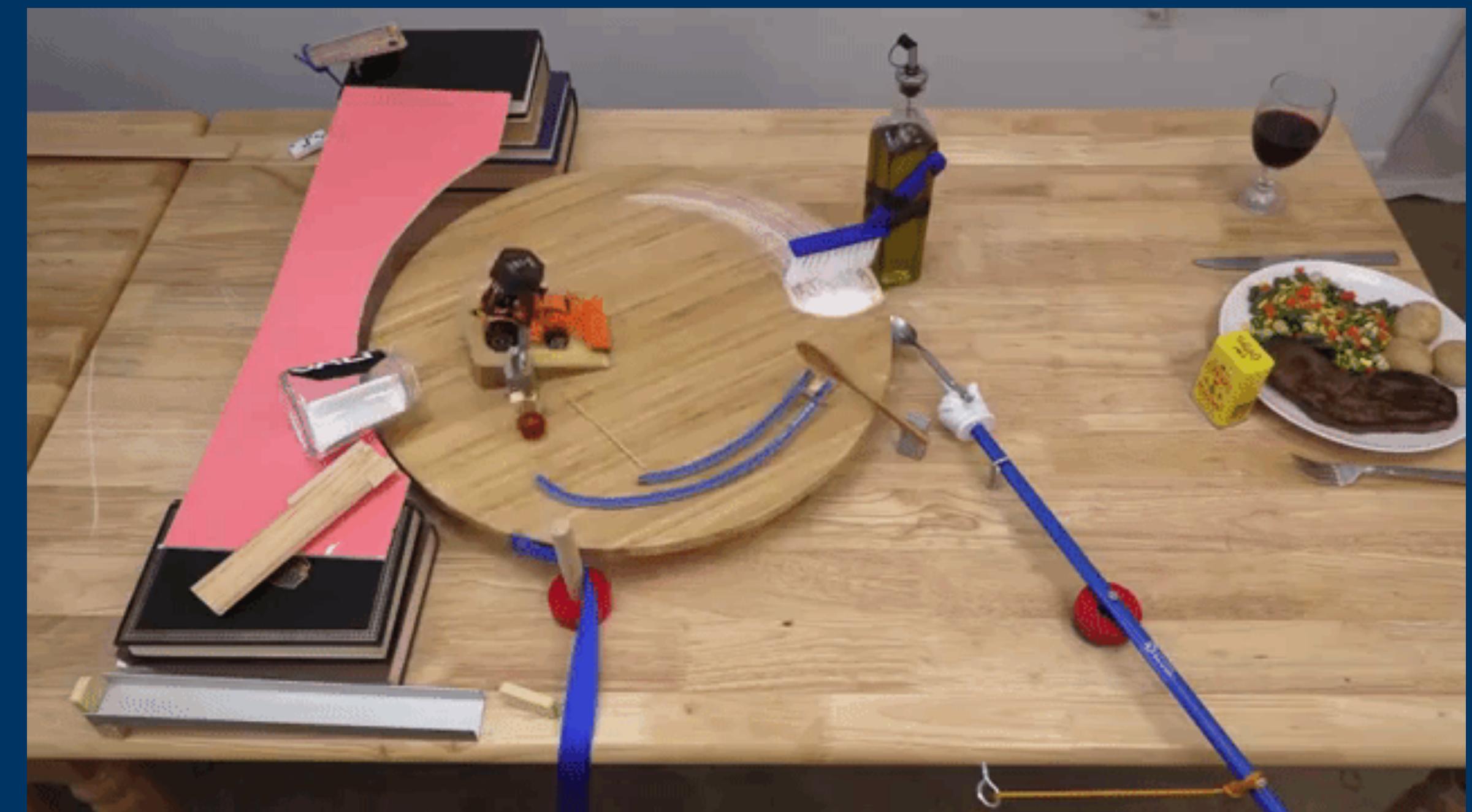
```
use_import_from("pkg", "function")
```

- 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 

Continuous Integration



use_github_action()

github.com/r-lib/actions

- "check-standard": Runs R CMD check when you push
- "test-coverage": Compute test coverage and report at 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

Break Time!

Design Principles

Review: functions

Run once

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

Run periodically

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

Run frequently

- `load_all()`
- `document()`
- `test()`
- `check()`

⌨️ Ctrl/Cmd+Shift+L

⌨️ Ctrl/Cmd+Shift+D

⌨️ Ctrl/Cmd+Shift+T

⌨️ Ctrl/Cmd+Shift+E

Learning Objectives

- At the end of this section you will be able to:
 - Understand core principles about **function design**
 - Apply the **DRY principle** to avoid redundancy and improve maintainability in your R code
 - Design functions with **default arguments, input validation, and helper functions** for greater clarity and flexibility
 - Enhance **code readability and consistency** by following structured style guidelines and effective naming conventions

Why Design?

Focus on core principles to make R code cleaner and easier to manage

Tidyverse guiding design principles

- Programming is a task performed by **humans**
- Reduce your cognitive load with the **consistent** design (as much as possible)
- Make your functions and systems **composable**
- Think about others who are not like us to create an **inclusive** space

```
cor(1:10, 2:11, "pairwise.complete.obs", "spearman")
```

```
cor(1:10, 2:11, use = "pairwise.complete.obs", method = "spearman")
```

Naming things

Some general guidelines

- Use verbs to ascribe an action
- Use consistent style
- Use singular (because English is weird)
- Consider short prefixes to unify package functions
- Don't be afraid to be verbose

```
## bad  
gse(date = "1977-05-25")  
  
## good  
get_salmon_escaping(date = "1977-05-25")
```

Pure functions versus Side effects

- A pure function:
 - produces the same output for the same input, with no impact on other parts of the program
 - only change is what is returned
 - Easier to test and maintain
- Function with side effects
 - A function that interacts with the outside environment

Pure functions

```
add <- function(x, y) {  
  x + y  
}  
  
add(2, 3)  
#> [1] 5
```

Functions with Side effects

```
library(readr)  
read_csv(readr_example("mtcars.csv"))  
#> # A tibble: 32 × 11  
#>   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  
#>   <dbl>  
#> 1 21       6   160   110   3.9   2.62  16.5     0     1     4     4  
#> 2 21       6   160   110   3.9   2.88  17.0     0     1     4     4  
#> 3 22.8     4   108   93    3.85  2.32  18.6     1     1     4     1  
#> 4 21.4     6   258   110   3.08  3.22  19.4     1     0     3     1  
#> 5 18.7     8   360   175   3.15  3.44  17.0     0     0     3     2  
#> 6 18.1     6   225   105   2.76  3.46  20.2     1     0     3     1  
#> 7 14.3     8   360   245   3.21  3.57  15.8     0     0     3     4
```

The DRY Principle: Don't Repeat Yourself

What is DRY?

- Minimize redundancy by centralizing logic
- Benefits
 - Consistency
 - Easier debugging
 - Maintenance

The DRY Principle: Don't Repeat Yourself

Identifying Redundant Code

- Signs of redundant code
 - Repeated data transformations
 - Hardcoded values
- Think MODULAR
- Look for patterns

The DRY Principle: Don't Repeat Yourself

Refactor to use functions

```
# Without DRY
```

```
se_age ← sd(df$age) / sqrt(length(df$age))
```

```
se_income ← sd(df$income) / sqrt(length(df$income))
```

```
# With DRY
```

```
se ← function(x) sd(x) / sqrt(length(x))
```

```
# Now we can reuse the function for any column
```

```
se_age ← se(df$age)
```

```
se_income ← se(df$income)
```

Helper Functions

- Breaking (**decompose**) up Large Functions
 - Limitations of lengthy, complex functions
 - How small, focused functions increase readability and testing ease
- Can make help to DRY but also has value to itself
- When and How to Use Helper Functions
 - Define helper functions to handle subtasks within a larger function

Helper Functions

Example

```
sum_of_squares ← function(a, b) {  
  (a * a) + (b * b)  
}
```

```
sum_of_squares(3, 5)  
#> [1] 34
```

Helper Functions

Example

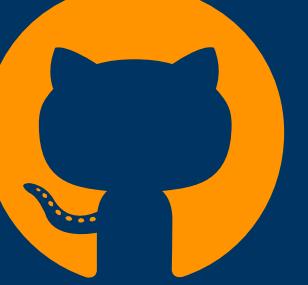
```
square ← function(x) {  
  x * x  
}  
  
sum_of_squares ← function(a, b) {  
  square(a) + square(b)  
}  
sum_of_squares(3, 5)  
#> [1] 34
```

Helper Functions

libminer

```
lib_summary ← function(sizes = FALSE) {  
  pkgs ← utils::installed.packages()  
  pkg_tbl ← table(pkgs[, "LibPath"])  
  pkg_df ← as.data.frame(pkg_tbl, stringsAsFactors = FALSE)  
  names(pkg_df) ← c("Library", "n_packages")  
  
  if (sizes) {  
    pkg_df$lib_size ← map_dbl(  
      pkg_df$Library,  
      ~ sum(fs::file_size(fs::dir_ls(.x, recurse = TRUE))),  
    )  
  }  
  pkg_df  
}
```

→ Your Turn

check() 
Commit your changes 
Push to GitHub 

Arguments

- What Are Function Arguments?
 - Values passed to a function that specify how it should operate
 - Allow customization and flexibility in function behaviour
- Types of Arguments
 - Required arguments: Must be provided when the function is called
 - Default arguments: Have preset values; can be overridden
 - Variable arguments: Use ... to accept a flexible number of inputs

Default Argument Values

- Why Use Default Arguments?
 - Adds flexibility by setting typical values that simplify function calls.
 - Allow functions to handle common cases without needing all arguments specified every time.
 - Useful in cases where arguments have default values that cover frequent use-cases

Arguments

Refactor to use functions

```
greet ← function(name, greeting = "Hello") {  
  paste(greeting, name)  
}
```

```
# Usage  
greet(name = "Andy")  
#> [1] "Hello Andy"  
greet(name = "Andy", greeting = "Hi")  
#> [1] "Hi Andy"
```

Default Argument Values

Setting Defaults

- Use defaults for parameters that often remain the same, making it easier to call the function without redundant inputs.
- Avoid defaults that might hide potential issues (e.g., setting a default that could lead to silent errors if overlooked).
- Document defaults clearly, so it's clear what will happen if arguments are left unspecified.

```
#' @param name a person's name.  
#' @param greeting the greeting you wish to choose. defaults to  
"Hello"
```

Input Checking and Validation

Why Input Checking Matters

- Prevents Unexpected Errors: Ensures that the function receives valid inputs, reducing the likelihood of errors.
- Improves Code Robustness: Increases confidence in code behaviour
- Helps make failures intentional by catching errors early in the process, making debugging easier by validating assumptions

Tools for Input Checking

- Packages: assertthat, check, validate
- Built-in functions: stop, is.numeric, is.character

Input Checking and Validation

An example

```
calculate_area <- function(length, width) {  
  area <- length * width  
  area)  
}  
  
calculate_area(4, 2)  
#> [1] 8
```

Input Checking and Validation

An example - bad inputs!

```
calculate_area(-5, 2)
#> [1] -10
calculate_area('5', 2)
#> Error in length * width: non-numeric argument to binary operator
```

Input Checking and Validation

An example - bad inputs!

```
calculate_area <- function(length, width) {  
  if (length ≤ 0 || width ≤ 0) {  
    stop("Length and width must be positive numbers.", call. = FALSE)  
  }  
  area <- length * width  
  area  
}  
  
calculate_area(-5, 2)  
#> Error: Length and width must be positive numbers.
```

Input Checking and Validation

An example - bad inputs!

```
calculate_area('5', 2)  
#> Error in length * width: non-numeric argument to binary operator
```

→ Your Turn

Quickly back to helper functions

```
validate_inputs <- function(length, width) {  
  if (length <= 0 || width <= 0) {  
    stop("Length and width must be positive numbers.", call. = FALSE)  
  }  
  if (!is.numeric(length) || !is.numeric(width)) {  
    stop("Length and width must be a number.", call. = FALSE)  
  }  
}
```

Quickly back to helper functions

```
calculate_area <- function(length, width) {  
  validate_inputs(length, width)  
  area <- length * width  
  area  
}  
  
calculate_area(-5, 2)  
#> Error: Length and width must be positive numbers.  
  
calculate_area('5', 2)  
#> Error: Length and width must be a number.
```

→ Your Turn

libminer::Input Checking and Validation

```
lib_summary <- function(sizes = FALSE) {  
  pkg_df <- lib()  
  pkg_df <- table(pkg_df$LibPath)  
  names(pkg_df) <- c("Library", "n_packages")  
  
  if (sizes) {  
    pkg_df <- calculate_sizes(pkg_df)  
  }  
  pkg_df  
}
```

Code Style of Readability

Why should you care about code readability?

Facilitates Teamwork

- Clear and consistent code makes it easier for team members to understand each other's work, enhancing collaboration.

Enhances Maintainability

- Readable code simplifies debugging and updates, making it easier to maintain and extend functionality.

Reduces cognitive load

- Well-structured code reduces the time spent reading and understanding code, allowing developers to focus on problem-solving.

Break Time!

Website Creation and Long Form Documentation

Why make a website?

- Professional Documentation
 - Offers a polished interface that enhances the credibility of your package.
- Improved Accessibility
 - Makes documentation easily accessible online, allowing potential users to quickly find information, tutorials, and examples.
- Enhanced Discoverability
 - Increases visibility of your package, helping users easily learn about its features and functionalities.
- It's easy!

use_pkgdown_github_pages()

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

- `use_vignette()` is a good way to initiate a vignette
- `use_article()` initiates a article that only exists in the pkgdown website
- Today these functions create an `.Rmd` document
- pkgdown just gained experimental support for Quarto (`.qmd`) and usethis will adapt soon
- A vignette or article focuses on how to solve a target problem, versus documenting a specific function

Vignettes

Long-form documentation

- Vignettes can only use packages you formally depend on, i.e. they appear in **Imports** or **Suggests**
- Help topics live as **.Rd** files in a source package, but there's no equivalent for rendered vignettes
 - Creates some inherent awkwardness in vignette workflow
 - Similar to examples, **R CMD check** treats vignettes like big, weird tests.
 - Creates tension between showing realistic usage and ensuring the code can be evaluated, 100% of the time and quickly.
 - Challenges: wrapping an external service, long-running code, etc.

Vignettes workflows

Long-form documentation

- Interactive work: `load_all()` + usual gestures for developing code chunks
- See a draft: install the in-dev package + usual gesture for rendering .Rmd
- See a draft: install the in-dev package and request that vignettes be built, via
`install(build_vignettes = TRUE)`
- See a draft: `devtools::build_rmd("vignettes/my-vignette.Rmd")`, which temporarily installs in-dev package before rendering
- See a draft: use pkgdown and preview your site locally or remotely

Eval chunk option

- `eval` chunk option controls whether code is evaluated
- Similar to `@examplesIf`, you can call a function to determine if it's safe to evaluate a chunk

- `eval = requireNamespace("somedependency")`
- `eval = !identical(Sys.getenv("SOME_THING_YOU_NEED"), "")`
- `eval = file.exists("credentials-you-need")`

- You can use `eval` local to individual chunks or set it globally in a setup chunk.

```
```{r setup, include = FALSE}
can_decrypt ← gargle:::secret_can_decrypt("googlesheets4")
knitr::opts_chunk$set(
 collapse = TRUE,
 comment = "#>",
 error = TRUE,
 eval = can_decrypt
)
```
``
```

```
```{r eval = !can_decrypt, echo = FALSE, comment = NA}
message("No token available. Code chunks will not be evaluated.")
```
``
```

```
```{r index-auth, include = FALSE}
googlesheets4:::gs4_auth_docs()
```
``
```

Review: functions

Run once

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

Run periodically

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

Run frequently

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

DFO Package Discussion

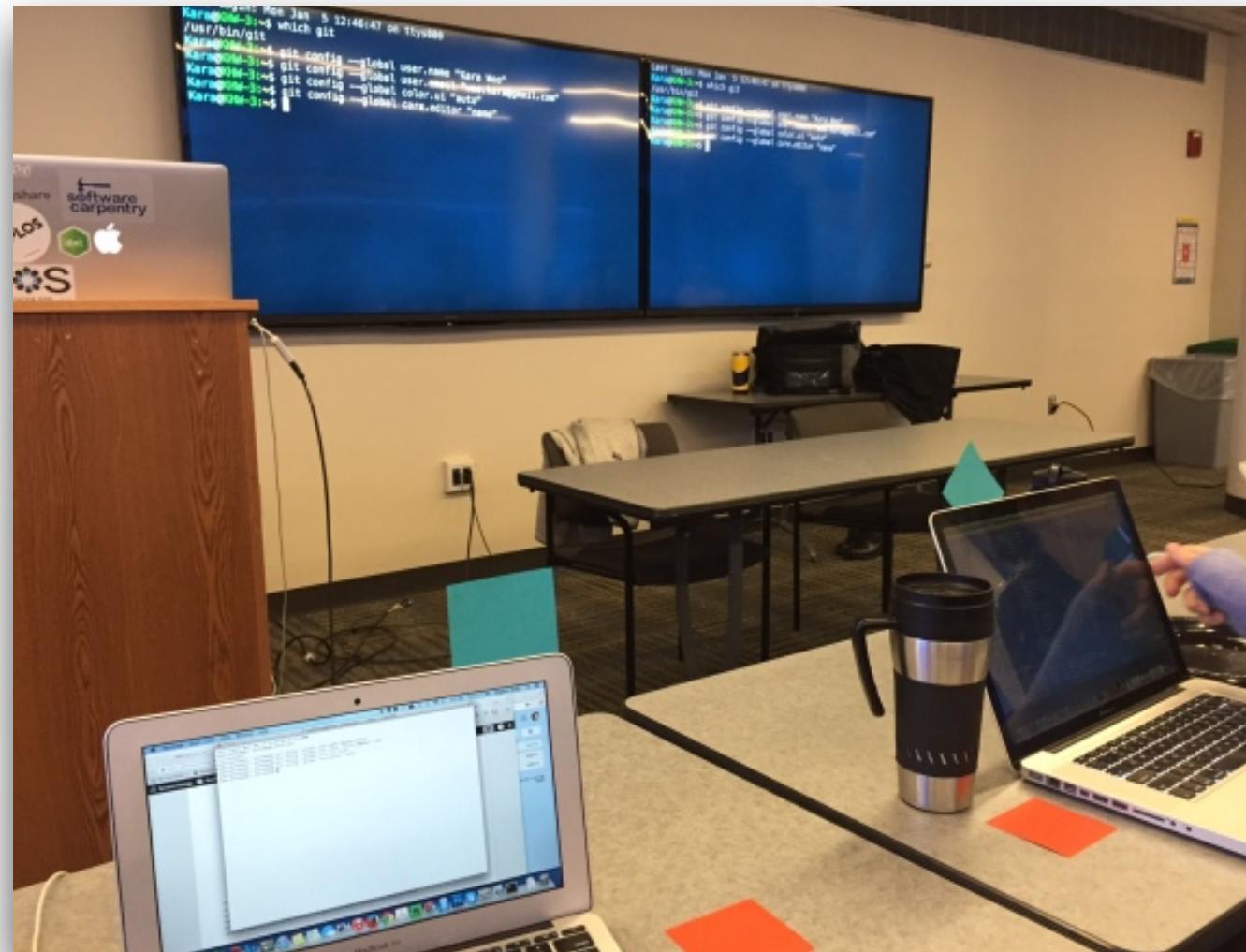
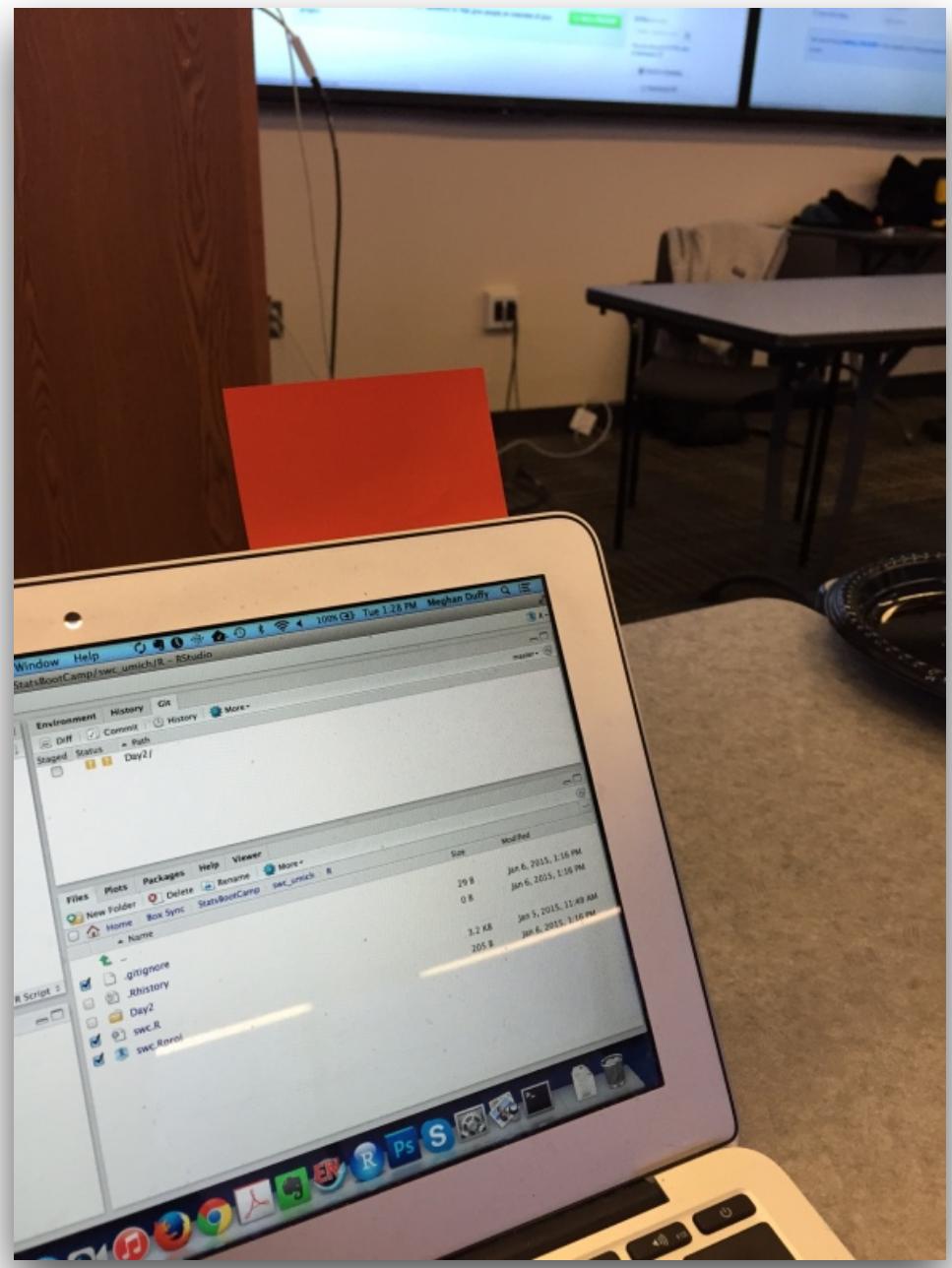
Day two complete!

Welcome back to day three

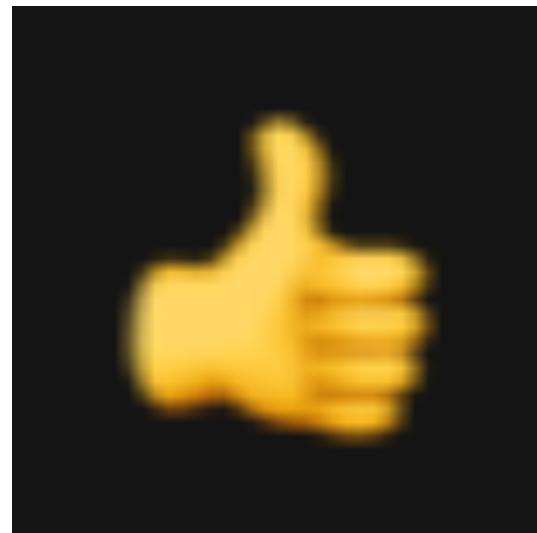
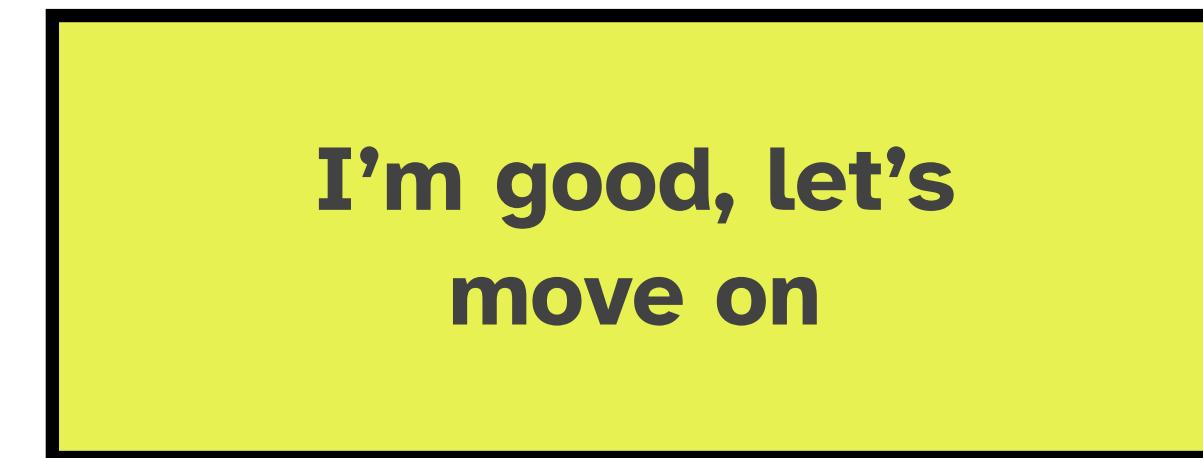
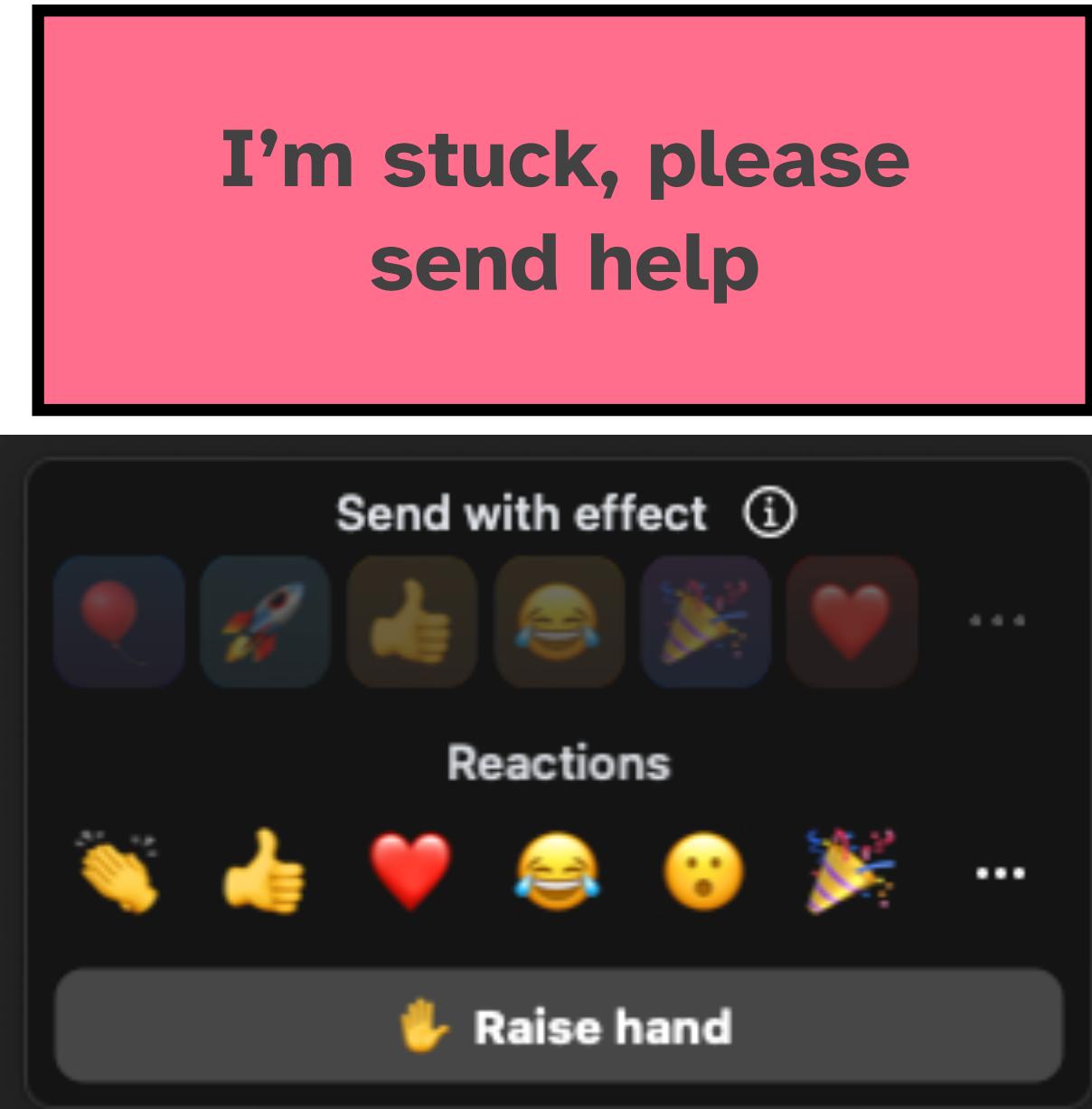
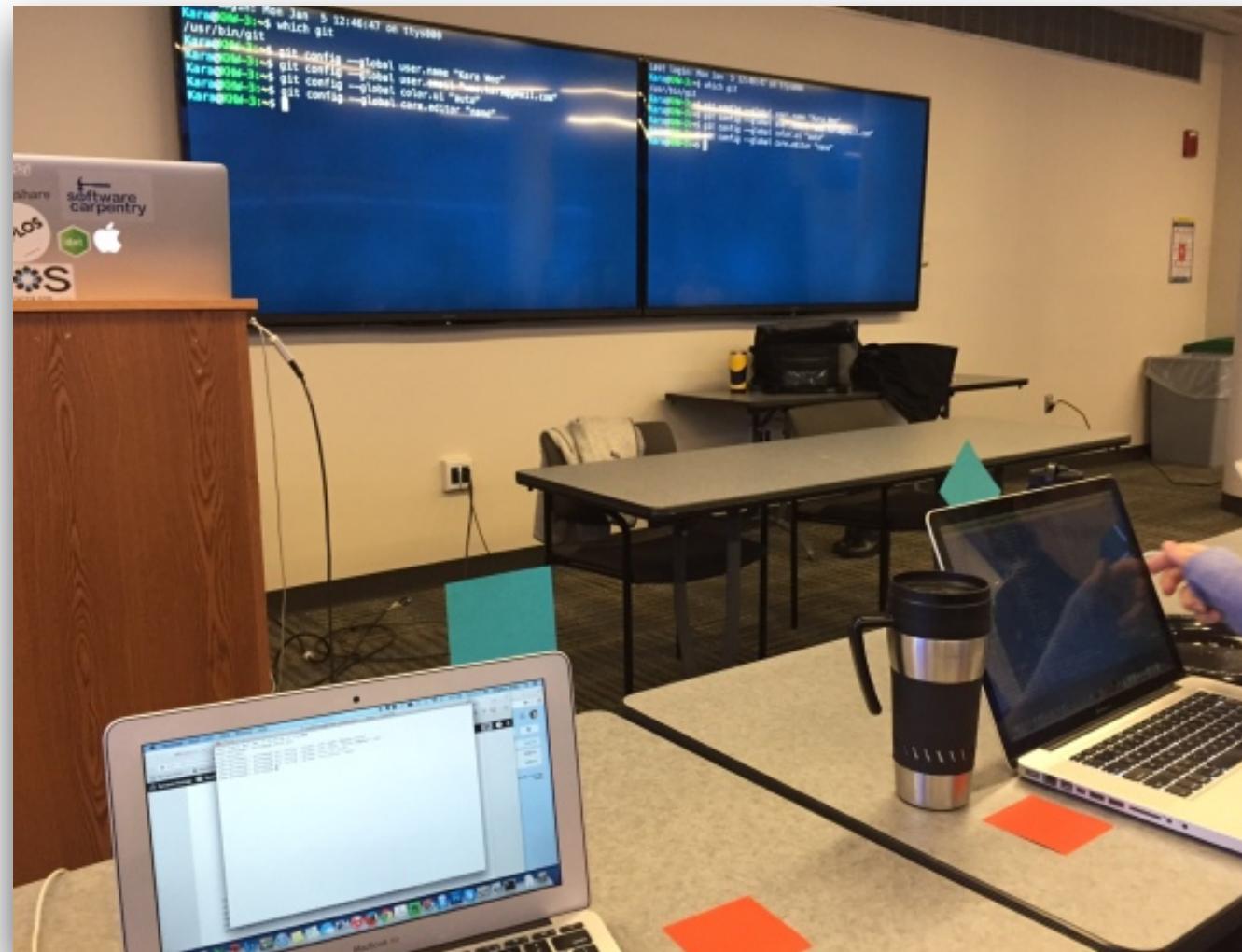
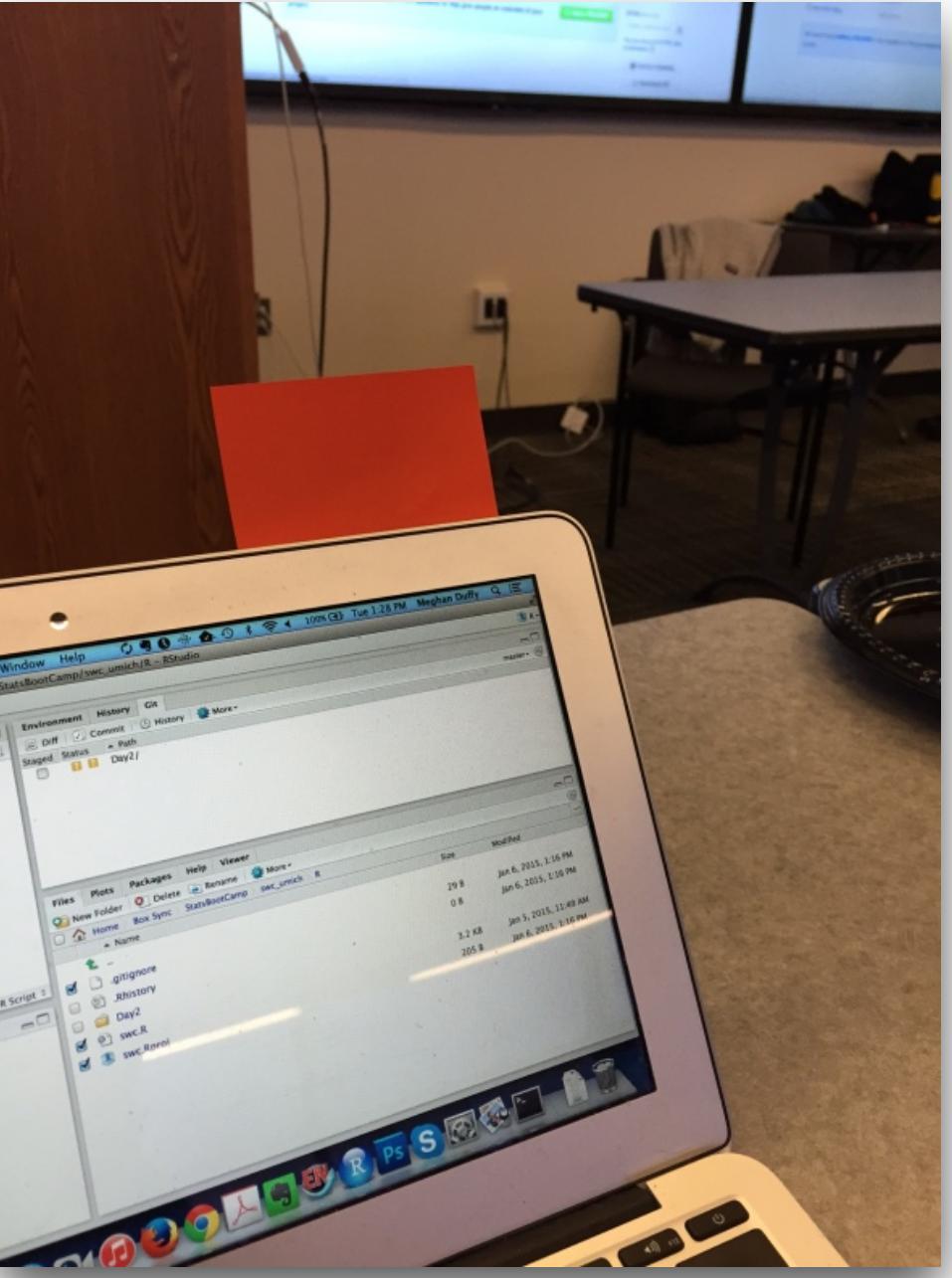
Day 3: Schedule and Learning Objectives

| | | |
|---------------|--|--------|
| 08:00 - 09:20 | Using the Tidyverse in Your Package | 80 min |
| 09:20 - 09:35 | Break | 15 min |
| 09:35 - 10:55 | Communicating with your Users | 80 min |
| 10:55 - 11:10 | Break | 15 min |
| 11:10 - 12:30 | Effective Use of GitHub + Distribution | 80 min |

Sticky Notes



Sticky Notes



Using the tidyverse in your package

Learning Objectives

- At the end of this section you will be able to:
 - build functions that call tidyverse functions, using bare-name arguments:
 - designed for interactive use, i.e. like `dplyr::filter()`
 - understand terms: **data masking, tidy select, dynamic dots**

Tidy Evaluation

Motivation

- Tidy eval (and non-standard evaluation generally in R) exists so that we can refer to data columns using bare names.
 - `dplyr::filter(mtcars, cyl = 4)`

vs

- `mtcars[mtcars$cyl = 4,]`
- It makes things easier if you are working interactively (it allows *indirection*).
 - It makes things more interesting if you are writing functions.

Welcome to *more interesting*.

Families of Tidy-eval Functions

- **data-masking:** compute on values within a dataset:

```
mtcars |>  
  dplyr::mutate(wt_kg = wt * 1000 / 2.2)
```

- **tidy-select:** specify columns within a dataset

```
mtcars |>  
  dplyr::select(starts_with("w"))
```

Families of Tidy-eval Functions

- **data-masking:** evaluate variables in context of data frame:
`arrange()`, `count()`, `filter()`, `group_by()`, `mutate()`,
and `summarize()`
- **tidy-select:** specify columns within a data frame
`across()`, `relocate()`, `rename()`, `select()`, and `pull()`

Data Masking

```
starwars[starwars$homeworld == "Naboo" & starwars$species == "Human", ]
```

```
starwars |>  
  filter(homeworld == "Naboo", species == "Human")
```

```
mtcars |>  
  group_by(cyl) |>  
  summarise(n = n(), mpg = mean(mpg))
```

Data Masking

Two types of variables

- `mtcars`, `starwars` - datasets, variables available in the environment
 - **environment-variables**
- `cyl`, `mpg`, `homeworld`, and `species` - columns in datasets, variables available in data
 - **data-variables**

Data Masking

Identify env-variables vs data-variables

```
starwars[starwars$homeworld == "Naboo" & starwars$species == "Human", ]
```

```
starwars |>  
  filter(homeworld == "Naboo", species == "Human")
```

```
mtcars |>  
  group_by(cyl) |>  
  summarise(n = n(), mpg = mean(mpg))
```

Programming with data variables

```
var_summary <- function(data, var) {  
  data |>  
    summarise(  
      min = min(var),  
      max = max(var)  
    )  
}
```

```
mtcars |>  
  group_by(cyl) |>  
  var_summary(mpg)  
#> Error in `summarise()`:  
#>   i In argument: `min = min(var)`.  
#>   i In group 1: `cyl = 4`.  
#> Caused by error:  
#> ! object 'mpg' not found
```

Programming with data variables

{} embrace {} data variables as function arguments

```
var_summary <- function(data, var) {  
  data |>  
    summarise(  
      min = min({{ var }}),  
      max = max({{ var }})  
    )  
}
```

```
mtcars |>  
  group_by(cyl) |>  
  var_summary(mpg)  
#> # A tibble: 3 × 3  
#>   cyl   min   max  
#>   <dbl> <dbl> <dbl>  
#> 1     4   21.4  33.9  
#> 2     6   17.8  21.4  
#> 3     8   10.4  19.2
```

Programming with data variables

Use `.data` pronoun for variables you know about

```
big_cars_summary <- function(var) {  
  mtcars |>  
    filter(.data$cyl >= 6) |>  
    group_by(.data$cyl) |>  
    summarise(  
      n = n(),  
      mean = mean({{ var }}),  
    )  
}
```

```
big_cars_summary(disp)
```

Write a function that summarizes `starwars` by any grouping variable

```
library(dplyr)
head(starwars)
#> # A tibble: 6 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>
#> 1 Luke Skywalker 172    77  blond      fair        blue        19   male   masculin...
#> 2 C-3PO          167    75  <NA>       gold        yellow     112   none   masculin...
#> 3 R2-D2          96     32  <NA>       white, bl... red         33   none   masculin...
#> 4 Darth Vader  202    136  none       white        yellow     41.9  male   masculin...
#> 5 Leia Organa  150     49  brown      light       brown       19   female feminin...
#> 6 Owen Lars    178    120  brown, gr... light       blue        52   male   masculin...
#> # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

* Do this in a file that is not inside `R/`

* Add path to `.Rbuildignore`

→ Your Turn

Write a function that summarizes `starwars` by any grouping variable

```
height_sum <- function(data, group_var) {  
  data |>  
    dplyr::group_by({{ group_var }}) |>  
    dplyr::summarise(  
      n = dplyr::n(),  
      mean_height = mean(.data$height)  
    )  
}
```

```
height_sum(starwars, hair_color)
```

Modify that function to take > 1 grouping variables

- Hint: '...' can get passed through unaltered.



Your Turn

Modify that function to take > 1 grouping variables

```
height_sum <- function(data, ...) {  
  data |>  
    dplyr::group_by(...) |>  
    dplyr::summarise(  
      n = dplyr::n(),  
      mean_height = mean(.data$height)  
    )  
}  
  
height_sum(starwars, hair_color, eye_color)
```

Data Masking - Summary

- dplyr: `arrange()`, `filter()`, `group_by()`, `mutate()`, and `summarise()`;
- ggplot2: `aes()`
- Data-variable is passed as a **function argument** x:
 - `arrange(df, {{ x }})`
- You (as developer) know the **data-variable** is x:
 - `arrange(df, .data$x)`
- Name of data-variable is a **character string** in an **env-variable** x:
 - `arrange(df, .data[[x]])`
- Remember ... can be passed through - no special treatment required.

Name injection with dynamic dots

Dynamically create new variable names

- Generate names programmatically by using `:=` instead of `=`
- Two ways:
 - From an env variable using `{glue}` syntax
 - From a data variable using `{{embrace}}` syntax
- To use in your package:

```
usethis::use_import_from("rlang", ":=")
```

Name injection with dynamic dots

Dynamically create new variable names

```
var_summary <- function(data, var, new_col_name) {  
  data |>  
    summarise(  
      "{new_col_name}" := min({{ var }})  
    )  
}
```

Name injection with dynamic dots

Dynamically create new variable names

```
var_summary <- function(data, var) {  
  data |>  
    summarise(  
      "{var}_min" := min({var})  
    )  
}
```

Write a function that summarizes `starwars`, and dynamically creates new column names

```
library(dplyr)
head(starwars)
#> # A tibble: 6 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>
#> 1 Luke Skywalker 172    77  blond      fair        blue        19   male   masculin...
#> 2 C-3PO          167    75  <NA>       gold        yellow     112   none   masculin...
#> 3 R2-D2          96     32  <NA>       white, bl... red         33   none   masculin...
#> 4 Darth Vader  202    136  none       white        yellow     41.9  male   masculin...
#> 5 Leia Organa  150     49  brown      light       brown       19   female feminin...
#> 6 Owen Lars    178    120  brown, gr... light       blue        52   male   masculin...
#> # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

* Do this in a file that is not inside `R/`

* Add path to `.Rbuildignore`

→ Your Turn

Write a function that summarizes `starwars`, and dynamically creates new column names

```
dynamic_sum <- function(data, group_var, sum_var) {  
  data |>  
    dplyr::group_by({{ group_var }}) |>  
    dplyr::summarise(  
      n = dplyr::n(),  
      "mean_{{sum_var}}" := mean({{ sum_var }})  
    )  
}  
  
dynamic_sum(starwars, hair_color, mass)
```

Tidy Selection

Select columns in a data frame

- Using bare names, or tidyselect helpers:
 - `select(df, 1)`: selects the first column
 - `select(df, last_col())` selects the last column.
 - `select(df, c(a, b, c))` selects columns **a**, **b**, and **c**.
 - `select(df, starts_with("a"))` selects all columns whose name starts with “**a**”
 - `select(df, ends_with("z"))` selects all columns whose name ends with “**z**”.
 - `select(df, where(is.numeric))` selects all numeric columns.
- Using character vectors:
 - `select(df, all_of(c("a", "b", "c")))` strictly selects all specified
 - `select(df, any_of(c("a", "b", "c")))` selects specified columns if they exist

dplyr::across()

- Used inside data-masking verbs, e.g., `summarize()`
- Specify:
 - Which columns to consider, using *tidy-select*
 - What functions to apply, if any

```
mtcars |>
  dplyr::group_by(cyl) |>
  dplyr::summarise(
    dplyr::across(c("mpg", "disp"), mean)
  )
#> # A tibble: 3 × 3
#>   cyl   mpg   disp
#>   <dbl> <dbl> <dbl>
#> 1     4   26.7 105.
#> 2     6   19.7 183.
#> 3     8   15.1 353.
```

all_of(); any_of()

- Used to disambiguate external vector (env-var) of column names
- all_of() is strict

```
cols <- c("mpg", "disp", "xyz")
mtcars |>
  dplyr::group_by(cyl) |>
  dplyr::summarise(
    dplyr::across(dplyr::all_of(cols), mean)
  )
#> Error in `dplyr::summarise()`:
#> i In argument: `dplyr::across(dplyr::all_of(cols), mean)`.
#> ! Can't subset elements that don't exist.
#> ✘ Element `xyz` doesn't exist.
```

all_of(); any_of()

- Used to disambiguate external vector (env-var) of column names
- any_of() is permissive

```
cols <- c("mpg", "disp", "xyz")
mtcars |>
  dplyr::group_by(cyl) |>
  dplyr::summarise(
    dplyr::across(dplyr::any_of(cols), mean)
  )
#> # A tibble: 3 × 3
#>   cyl  mpg  disp
#>   <dbl> <dbl> <dbl>
#> 1     4  26.7 105.
#> 2     6  19.7 183.
#> 3     8  15.1 353.
```

Tidyversify libminer

- Use:
 - tidyverse functions
 - Data Masking with `{{ }}`
 - Data Masking with `.data`
 - ...
 - `across()` and `any_of()`
- Update tests
- Update documentation



Our turn

Break Time!

Communicating with your user

Signaling Conditions

Three conditions

- **Errors**
 - most severe
 - no way for a function to continue and execution must stop.
- **Warnings**
 - in between errors and message
 - something has gone wrong or is unexpected but the function has executed
- **Messages**
 - mildest
 - Inform users that some action has been performed on their behalf.

Signaling Conditions

Three conditions

- **Errors**
 - `stop("You provided an invalid input")`
- **Warnings**
 - `warning("Unknown value converted to NA")`
- **Messages**
 - `message("Reticulating splines... this may take a while")`

Conditions are better than `cat()`

- They inherently indicate the severity of the message
- They can be handled in special ways with things like `try()`, `tryCatch()`, etc.
- Warnings and messages can silenced by users:

```
chatty <- function() {  
  message("Hello there!")  
  warning("You have been warned!")  
}  
  
chatty()  
#> Hello there!  
#> Warning in chatty(): You have been warned!  
  
suppressMessages(chatty())  
#> Warning in chatty(): You have been warned!  
  
suppressWarnings(chatty())  
#> Hello there!
```

beautiful command line interfaces

With the {cli} package

- Build a CLI using semantic elements: headings, bullet lists, etc.
- All cli text can contain interpreted string literals, via the glue package.
- Support for pluralization and list concatenation.
- Inline text formatting.
- Text wrapping.
- Theming via a CSS-like language.
- Progress bars from R and C code.
- Emit conditions -- messages, warnings, and errors -- with rich text formatting.

usethis's UI is built with cli

Character vector of lines.

Names control the type of bullet.

```
> ui_bullets(c(  
+   "v" = "A great success!",  
+   "_" = "Something you need to do.",  
+   "x" = "Bad news.",  
+   "i" = "The more you know.",  
+   " " = "I'm just here for the indentation.",  
+   "No indentation at all. Not used much in usethis."  
+ ))  
✓ A great success!  
□ Something you need to do.  
✖ Bad news.  
ℹ The more you know.  
    I'm just here for the indentation.  
No indentation at all. Not used much in usethis.
```

cli's greatest hits

- Single line alerts:
 - `cli_alert_success()`, `cli_alert_danger()`, `cli_alert_info()`, `cli_alert_warning()`
- Multi-line bullet lists:
 - `cli_bullets()`
- Conditions with all the cli (+ rlang) goodness
 - `cli_inform()`, alternative for `message()` or `rlang::inform()`
 - `cli_warn()`, alternative for `warning()` or `rlang::warn()`
 - `cli_abort()`, alternative for `stop()` or `rlang::abort()`

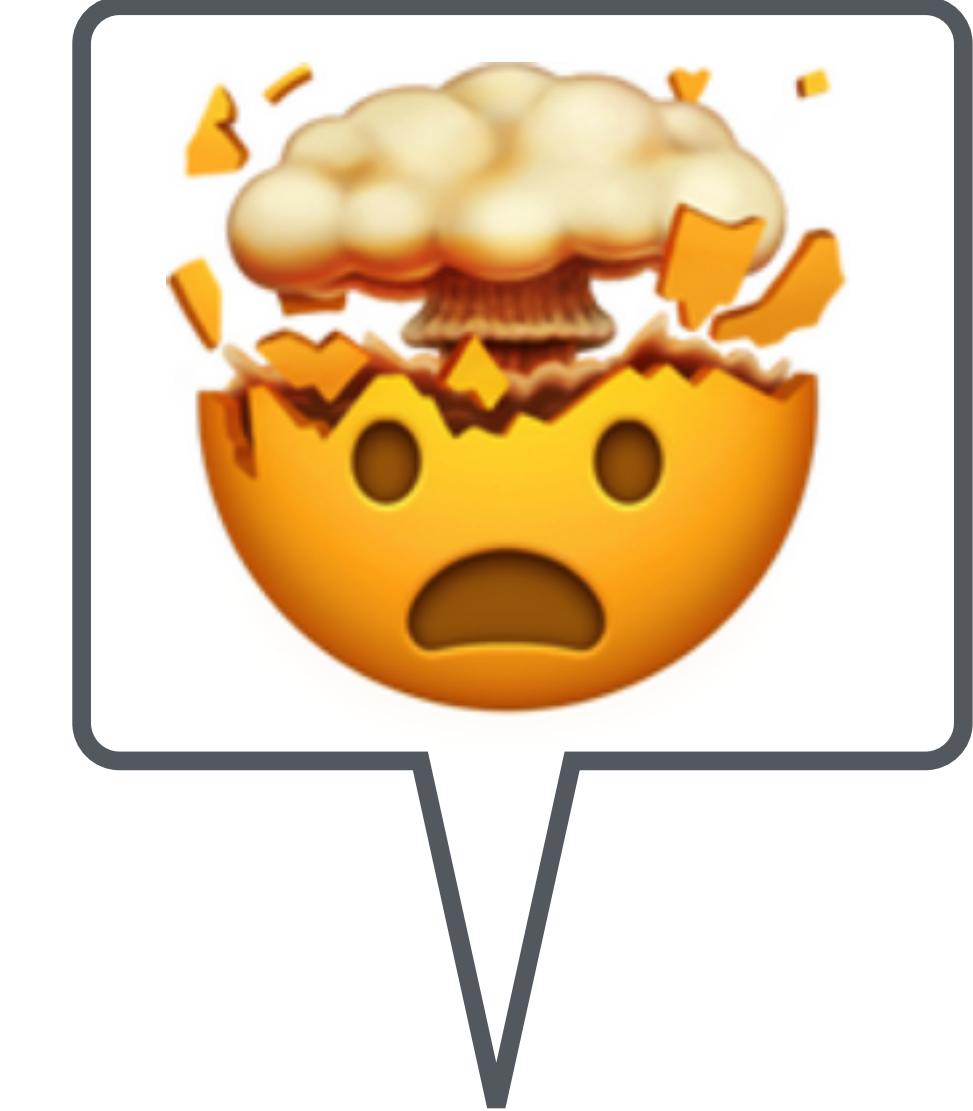
* All cli outputs are `message` conditions and can be handled as such

Interpolation

Code inside curly braces is evaluated, a la the glue package.

```
> pkgs <- "pkg1"
> cli_text("Will install the {pkgs} package{?s}")
Will install the pkg1 package
> pkgs <- c("pkg1", "pkg2", "pkg3")
> cli_text("Will install the {pkgs} package{?s}")
Will install the pkg1, pkg2, and pkg3 packages
```

Pluralization



```
> nfiles <- 0; cli_text("Found {nfiles} file{?s}")
Found 0 files
> nfiles <- 1; cli_text("Found {nfiles} file{?s}")
Found 1 file
> nfiles <- 2; cli_text("Found {nfiles} file{?s}")
Found 2 files
```

Pluralization and concatenation

```
> pkgs <- "pkg1"
> cli_text("Will install the {pkgs} package{s}")
Will install the pkg1 package

> pkgs <- c("pkg1", "pkg2", "pkg3")
> cli_text("Will install the {pkgs} package{s}")
Will install the pkg1, pkg2, and pkg3 packages
```

Inline text formatting

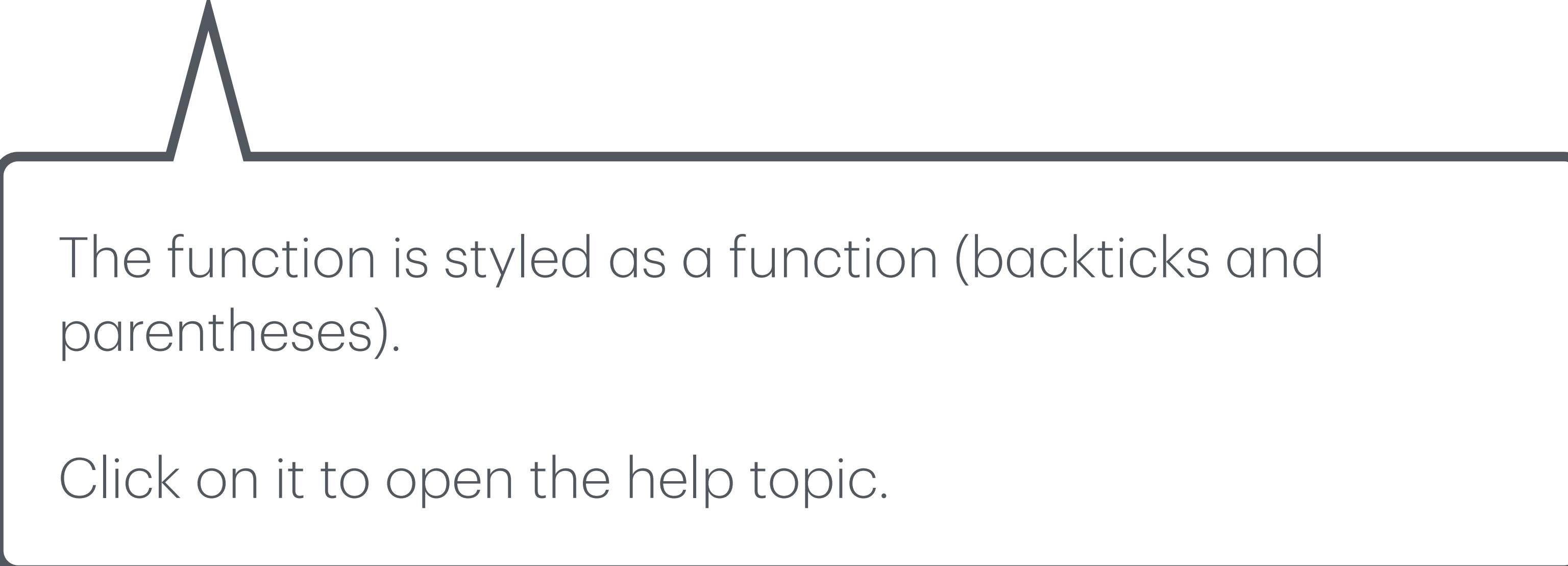
"{.thm text}"

```
cli_bullets(c(  
  "*" = "{.emph Emphasized} text",  
  "*" = "{.strong Strong} importance",  
  "*" = "A piece of code: {.code sum(a) / length(a)}",  
  "*" = "A package name: {.pkg cli}",  
  "*" = "A function name: {.fn cli_text}",  
  "*" = "A keyboard key: press {.kbd ENTER}",  
  "*" = "A file name: {.file /usr/bin/env}",  
  "*" = "An email address: {.email bugs.bunny@acme.com}",  
  "*" = "A URL: {.url https://acme.com}",  
  "*" = "An environment variable: {.envvar R_LIBS}",  
  "*" = "Some {.field field}"  
)
```

- *Emphasized* text
- **Strong** importance
- A piece of code: `sum(a) / length(a)`
- A package name: `cli`
- A function name: ``cli_text()``
- A keyboard key: press `[ENTER]`
- A file name: `/usr/bin/env`
- An email address: `bugs.bunny@acme.com`
- A URL: `<https://acme.com>`
- An environment variable: ``R_LIBS``
- Some `field`

Help topic links

```
> cli_text("Try out the {.fun cli::cli_abort} function")  
Try out the `cli::cli_abort()` function
```

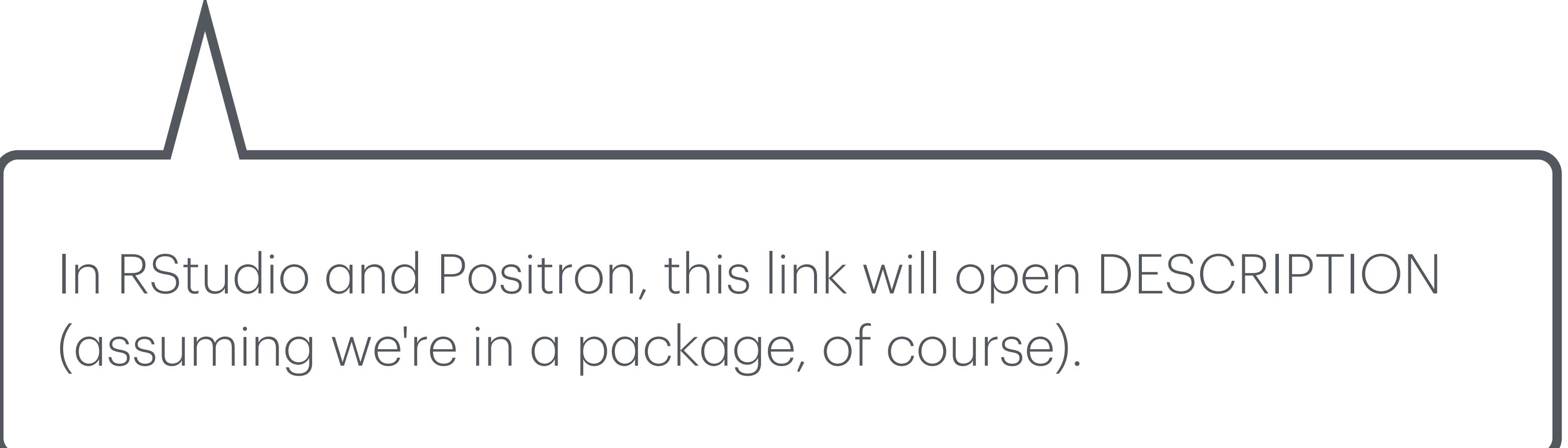


The function is styled as a function (backticks and parentheses).

Click on it to open the help topic.

File hyperlinks

```
> cli_text("A file name: {.file DESCRIPTION}")  
A file name: DESCRIPTION.
```

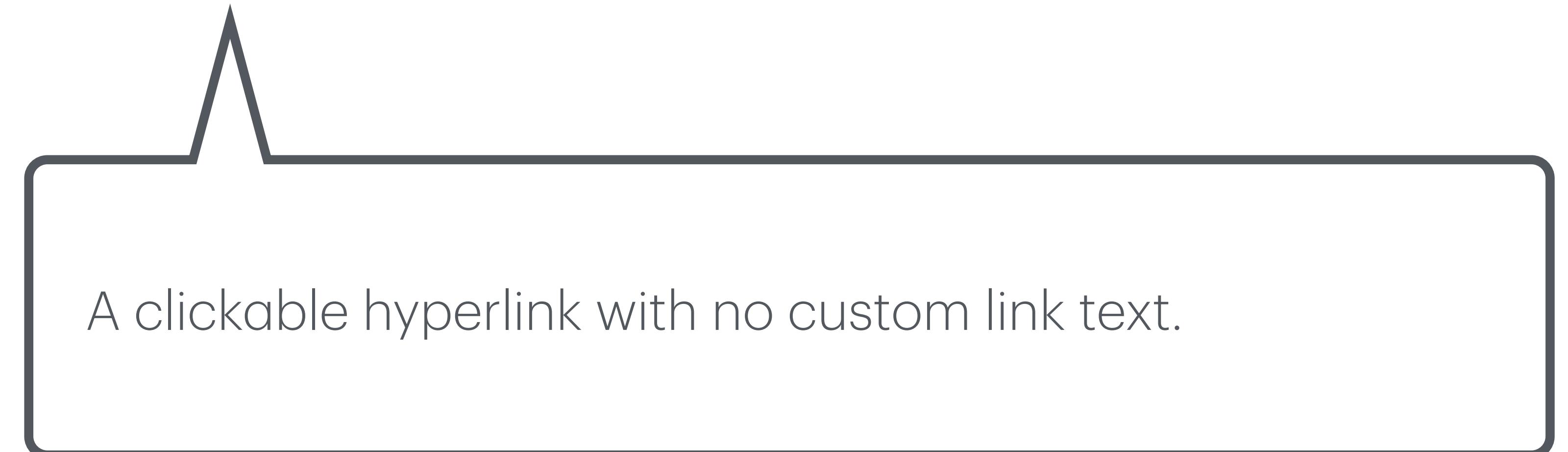


In RStudio and Positron, this link will open DESCRIPTION
(assuming we're in a package, of course).

URLs

```
> cli_text(  
  "See the cli homepage at {.url https://cli.r-lib.org} for details"  
)
```

See the cli homepage at <https://cli.r-lib.org> for details.



Hyperlinks

```
> cli_text(  
  "See the {.href [cli homepage](https://cli.r-lib.org)} for details"  
)  
See the cli.homepage for details.
```

Markdown-style hyperlinks

A clickable hyperlink with custom link text.

Formatting plus interpolation

Use double curly braces

```
> cli_url <- "https://cli.r-lib.org"  
> cli_text(  
  "See the cli homepage at {.url {cli_url}} for details"  
)
```

See the cli homepage at <https://cli.r-lib.org> for details.

Your turn! Some prompts

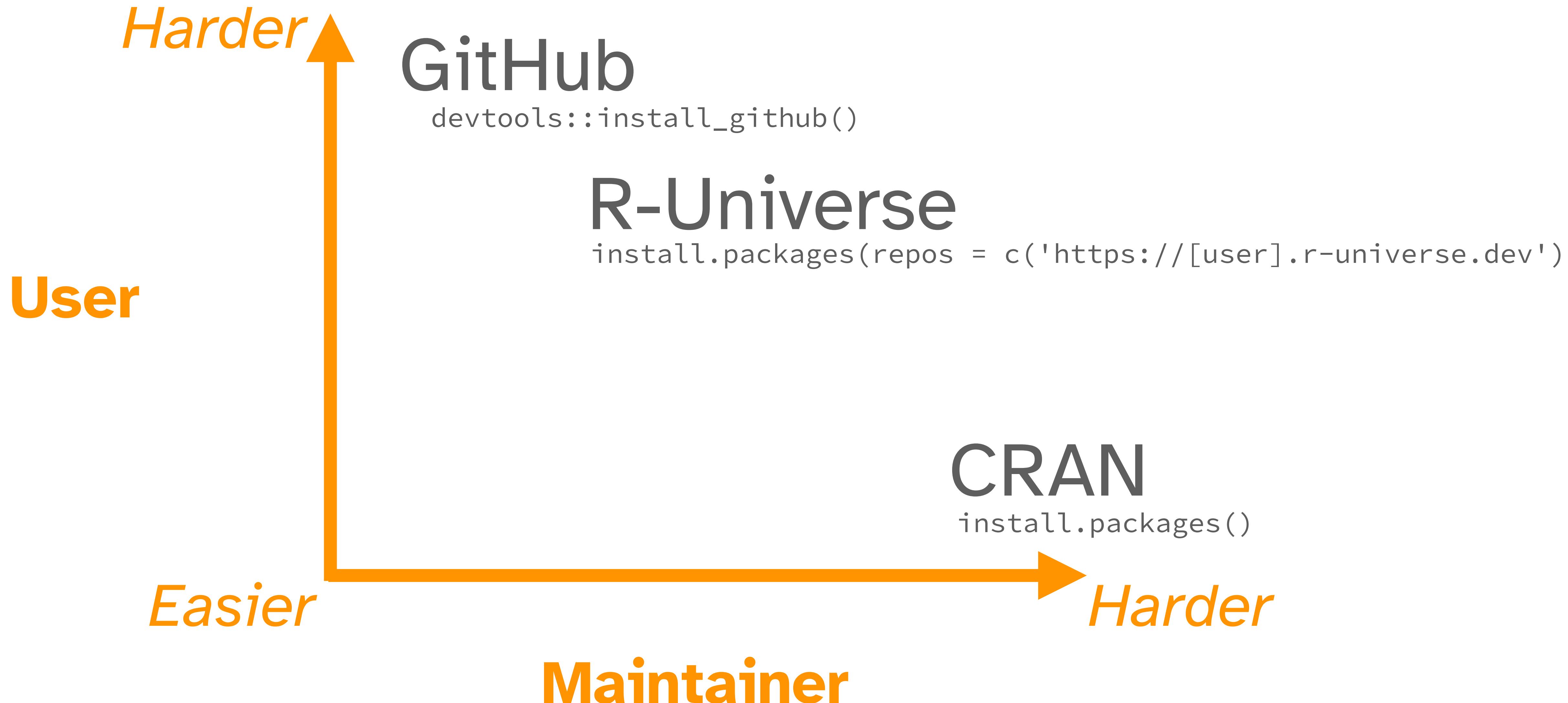
- Choose a package: libminer or one of your own
- Study the existing user interface (which functions are used?).
- Incrementally convert the UI-providing functions (e.g. `message()` or `stop()`) to its cli equivalent.
- Add some more bling
- Look at the cli documentation, especially:
 - cli's articles: <https://cli.r-lib.org/articles/index.html>
 - Help topics in the "Introduction" section here: <https://cli.r-lib.org/reference/index.html>

Break Time!

Sharing your Package + Effective use of Github



Package Distribution



Releasing your package to CRAN



GitHub Tour

The screenshot shows the GitHub repository page for `tidyverse / dplyr`. The page includes a header with navigation links for Code, Issues (76), Pull requests (12), Actions, Security, and Insights. Below the header are buttons for Edit Pins, Watch (244), Fork (2.1k), and Star (4.8k). The main content area displays a list of commits from the `main` branch, showing changes made by `DavisVaughan` and others. The commits are dated from 2 months ago to 6 years ago. To the right of the commit list is an **About** section with details about the repository, including its purpose as "A grammar of data manipulation", its website at dplyr.tidyverse.org/, and its tags: `r`, `grammar`, and `data-manipulation`. The **About** section also lists the repository's statistics: 4.8k stars, 244 watching, 2.1k forks, and a report button.

dplyr Public

`main` 38 Branches 58 Tags

Go to file Add file Code

About

dplyr: A grammar of data manipulation

dplyr.tidyverse.org/

`r` `grammar` `data-manipulation`

Readme

Unknown, MIT licenses found

Code of conduct

Activity

Custom properties

4.8k stars

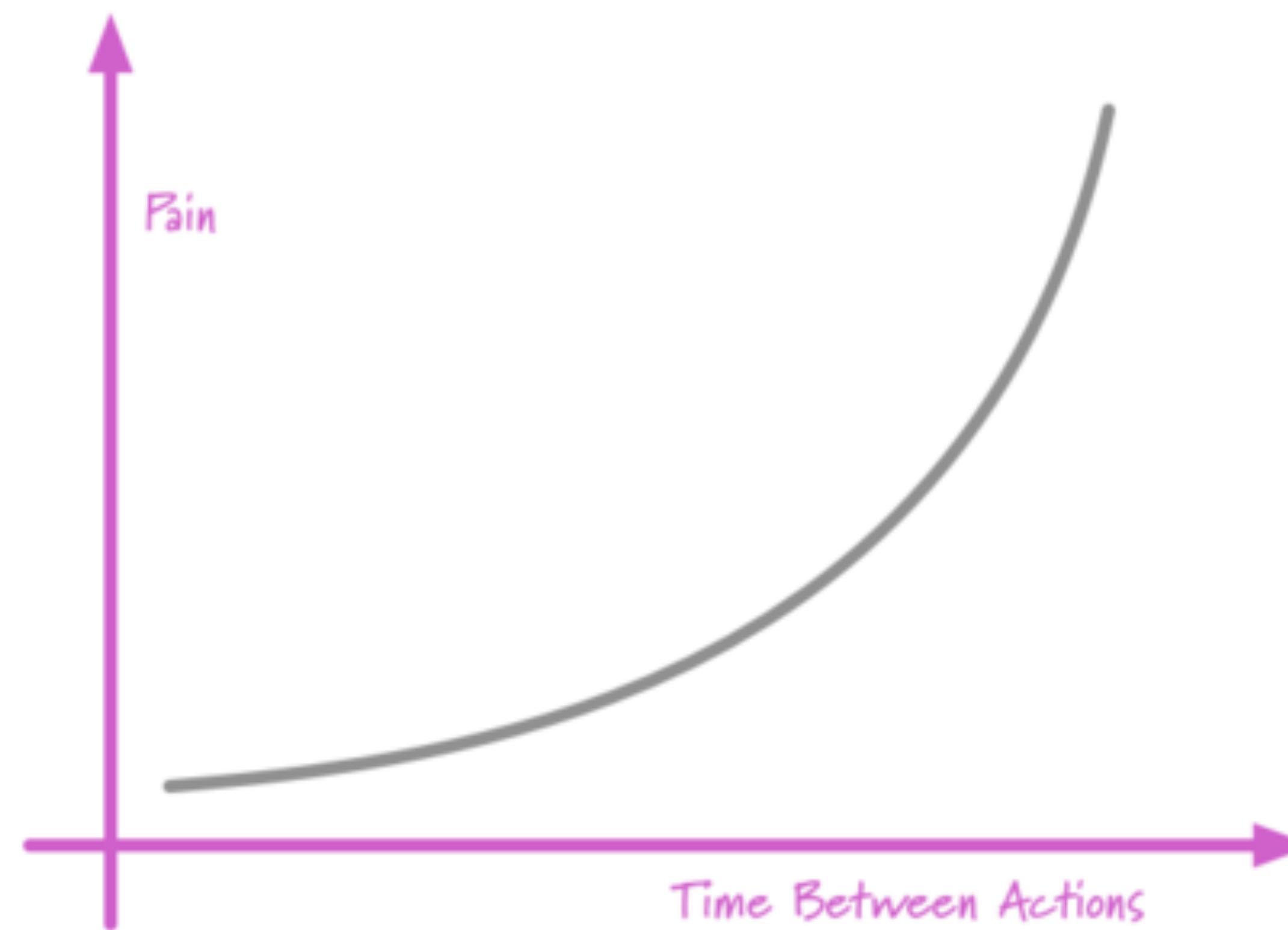
244 watching

2.1k forks

Report repository

| Commit | Author | Message | Date |
|-----------------------|---------------------------|--|--------------|
| <code>.github</code> | <code>DavisVaughan</code> | Remove not needed new_expanded_quosures() (#7090) ✓ | 2 months ago |
| <code>.vscode</code> | | Use latest GHA workflows (#7065) | 3 months ago |
| <code>R</code> | | Add compilation-database infrastructure | 3 months ago |
| <code>archive</code> | | Remove not needed new_expanded_quosures() (#7090) | 2 months ago |
| <code>data-raw</code> | | run-in and run-all | 6 years ago |
| <code>data</code> | | Update storms data through 2022 (#6937) | last year |
| <code>inst</code> | | Update storms data through 2022 (#6937) | last year |
| <code>man</code> | | master -> main (#6065) | 3 years ago |
| | | Add documentation clarifying appropriate use of weights i... | 3 months ago |

“If it hurts, do it more often”



“If it hurts, do it more often”

- If `check()` goes from passing to failing after you've tinkered with one function or touched 10 lines of code, it's clear what to troubleshoot.
- If you've added 15 functions that all call each other and touched 100s of lines of code, bless your heart.
- Better to learn of a regrettable choice before you've built upon it for days or weeks.

Releasing to CRAN

TLDR:

- `release()`
 - Runs through an additional list of checks
 - Builds package bundle and submits to CRAN

`use_news_md()`

Adds a `NEWS.md` file to your package

- Tracks version numbers
- Tracks user-facing changes between versions

use_cran_comments()

Comments that go along with your submission

```
## R CMD check results
```

```
0 errors | 0 warnings | 1 note
```

* This is a new release.

- Most times don't need more than this
- Note if you are fixing failures seen in CRAN checks
- Note reverse dependency checks
- If your submission was rejected, note that it is a resubmission and say how you have addressed the issues

*There is always one NOTE with initial submission

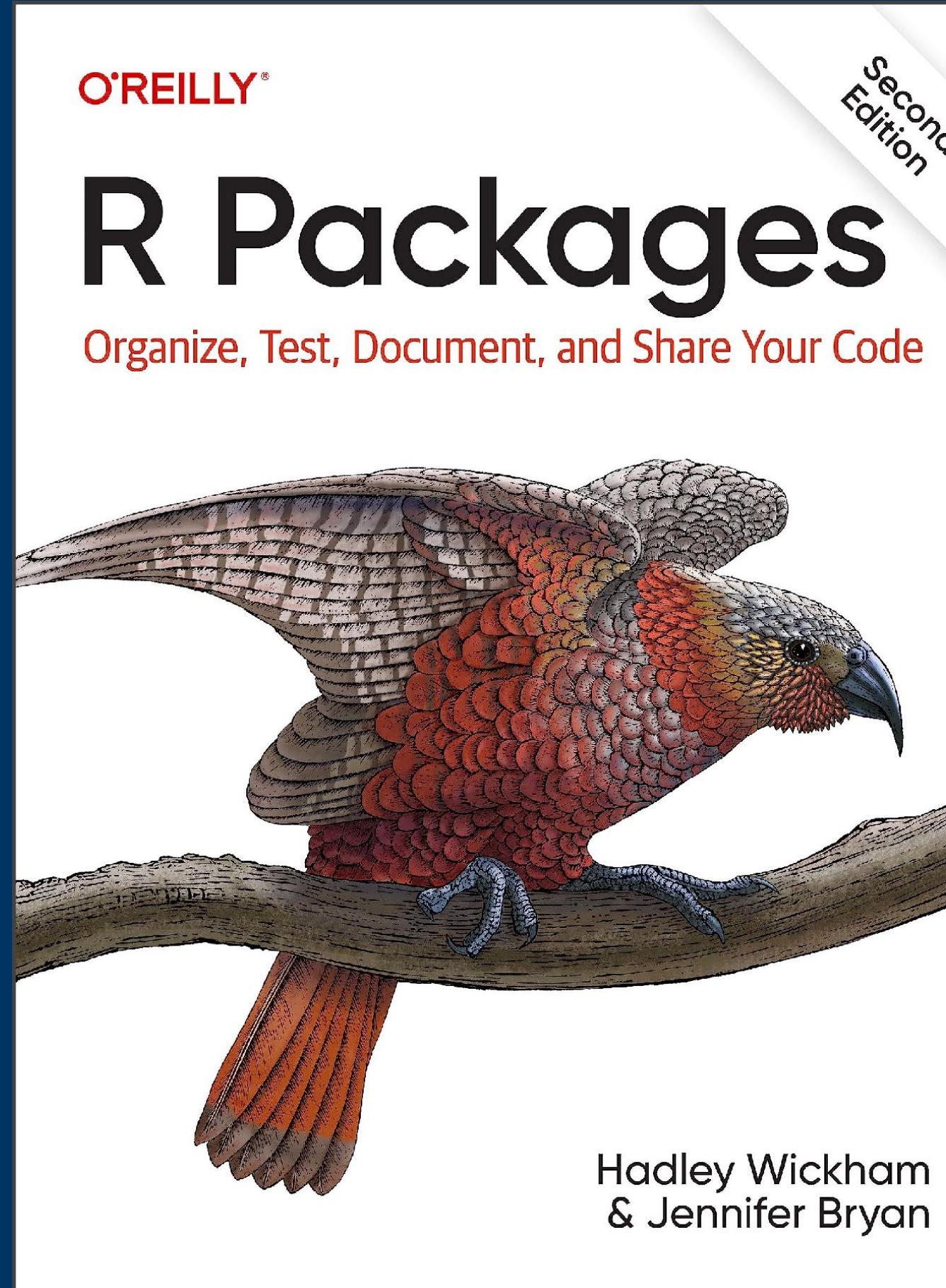
GitHub Code Search

GitHub code search pro tips

- `user:cran` is super powerful for searching across the source code of every package on CRAN
- I am biased, but I often use `user:tidyverse` or `user:r-lib` to focus on solutions by folks who generally approach R and package development the way I do.
- GitHub search is a tool where consulting the docs regularly really pays off:
 - Understanding GitHub Code Search syntax
- `repo:wch/r-source` narrows the search to a GitHub mirror of the R source
- Other qualifiers I use a lot:
 - `path`
 - `language`
- A qualifier that is newly relevant for R: `symbol`

Thank You!

Resources



r-pkgs.org

<https://community.rstudio.com/c/package-development>

Happy Git and GitHub for the user

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

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

Package Development : : CHEAT SHEET

Package Structure

Workflow

DESCRIPTION

R/

NAMESPACE

Tidy design principles

Welcome

The goal of this book is to help you write better R code. It has four main components:

- Identifying design **challenges** that often lead to suboptimal outcomes.
- Introducing useful **patterns** that help solve common problems.
- Defining key **principles** that help you balance conflicting patterns.
- Discussing **case studies** that help you see how all the pieces fit together with real code.

While I've called these principles "tidy" and they're used extensively by the tidyverse team to promote consistency across our packages, they're not exclusive to the tidyverse. Think tidy in the sense of tidy data (broadly useful regardless of what tool you're using) not tidyverse (a collection of functions designed with a singular point of view in order to facilitate learning and use).

This book will be under heavy development for quite some time; currently we are loosely aiming for completion in 2025. You'll find many chapters contain disjointed text that mostly serve as placeholders for the authors, and I do not recommend attempting to systematically read the book at this time. If you'd like to follow along with my journey writing this book, and learn which chapters are ready to read, please sign up for my [tidy design substack mailing list](#).

1 Unifying principles →

design.tidyverse.org
tidydesign.substack.com

posit.co/resources/cheatsheets/

Attribution

- <https://github.com/posit-conf-2024/pkg-dev>
- <https://cli.r-lib.org/articles/index.html>
- <https://adv-r.hadley.nz/conditions.html>
- <https://rstudio-conf-2022.github.io/build-tidy-tools/>
- <https://dplyr.tidyverse.org/articles/programming.html>
- <https://design.tidyverse.org/>

Course Materials

<https://andyteucher.ca/pkg-dev-dfo-2024-11>

Released under an open license: [Create Commons Attribution 4.0 International](#) - you are free to use, reuse, and remix (with attribution).

Survey

Your feedback is crucial! Please complete the post-workshop survey! 🙏

Data from the survey informs curriculum and format decisions for future workshops, and we really appreciate you taking the time to provide it.